Sample solutions

Bookmark this page

Part 0: Sample dataset (LSD)

In 1968, Wagner Agahajanian, and Bing conducted a study to determine whether you could improve a student's math test scores using lysergic diethylamide, also known as "LSD."

Here is the original data sources. The code cell below downloads the file from an alternative location, for compatibility with the Azure Notebook platforms you are using.

- Raw data: http://www.stat.ufl.edu/~winner/data/lsd.dat (http://www.stat.ufl.edu/~winner/data/lsd.dat)
- Data file description: http://www.stat.ufl.edu/~winner/data/lsd.txt (http://www.stat.ufl.edu/~winner/data/lsd.txt)

In [ ]:
```
from pandas import read_fwf
from IPython.display import display

import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/"
    DATA_PATH = ""

def download(file, local_dir="", url_base=URL_BASE, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(loca
                                                                                      body_
                                                                                      checl
    print("'{}' is ready!".format(file))

datasets = {'lsd.dat': '4c119057baf86cff8da03d825d7ce141'}
for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)
print("\n(All data appears to be ready.)")
```

Let's take a look at the data, first as a table and then using a scatter plot.

In [ ]:
```
df = read_fwf('{}lsd.dat'.format(DATA_PATH),
              colspecs=[(0, 4), (7, 13)],
              names=['lsd_concentration', 'exam_score'])
display(df)
```

In [ ]:
```
from matplotlib.pyplot import scatter, xlabel, title, plot
%matplotlib inline

scatter(df['lsd_concentration'], df['exam_score'])
xlabel ('LSD Tissue Concentration')
title ('Shocking news: Math scores degrade with increasing LSD!');
```

Fitting a model

Exercise 0 (2 points). Complete the function below so that it computes α and β for the univariate model, $y \sim \alpha \cdot x + \beta$, given observations sto arrays `y[:]` for the responses and `x[:]` for the predictor.

Use the linear regression formulas derived in class.

In [ ]:
```
def linreg_fit(x, y):
    """Returns (alpha, beta) s.t. y ~ alpha*x + beta."""
    from numpy import ones
    m = len(x) ; assert len(y) == m

    ### BEGIN SOLUTION
    u = ones(m)
    alpha = x.dot(y) - u.dot(x)*u.dot(y)/m
    alpha /= x.dot(x) - (u.dot(x)**2)/m
    beta = u.dot(y - alpha*x)/m
    ### END SOLUTION

    return (alpha, beta)

# Compute the coefficients for the LSD data:
x, y = df['lsd_concentration'], df['exam_score']
alpha, beta = linreg_fit(x, y)

print("alpha:", alpha)
print("beta:", beta)
```

In [ ]:
```
# Test cell: `linreg_fit_test`

x, y = df['lsd_concentration'], df['exam_score']
alpha, beta = linreg_fit(x, y)

r = alpha*x + beta - y
ssqr = r.dot(r)
ssqr_ex = 253.88132881

from numpy import isclose
assert isclose(ssqr, ssqr_ex, rtol=.01), "Sum-of-squared residuals is {} instead of {}.".format
_ex)

print("\n(Passed!)")
```

In [ ]:
```
from numpy import linspace, floor, ceil

# Two points make a line:
x_fit = linspace(floor(x.min()), ceil(x.max()), 2)
y_fit = alpha*x_fit + beta

scatter(x, y, marker='o')
plot(x_fit, y_fit, 'r--')
xlabel('LSD Tissue Concentration')
title('Best-fit linear model');
```

Fin! If you've gotten this far without errors, your notebook is ready to submit.

Part 1: Gradients example

This notebook is designed to illustrate the concept of the gradient.

Let $x \equiv \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ be a two-dimensional vector, and let

$$f(x) \equiv x^T x = x_0^2 + x_1^2.$$

Exercise 0 (1 point). Implement the Python function, `f(x0, x1)`, so that it computes $f(x)$.

In [1]:
```
def f(x0, x1):
    ### BEGIN SOLUTION
    return x0*x0 + x1*x1
    ### END SOLUTION
```

In [2]:
```
# Test cell: `f_test`

from numpy import sin, cos, vectorize, isclose
from numpy.random import randn
```

```
f_vec = vectorize(f)
theta = randn(1000)
assert all(isclose(f_vec(sin(theta), cos(theta)), 1.0))

print("\n(Passed!)")
```

```
(Passed!)
```

The gradient

Let's create a mesh of $[x_0, x_1]$ coordinate values:

In [3]:
```
from numpy import linspace, meshgrid
x0 = linspace(-2, 2, 11)
x1 = linspace(-2, 2, 11)
X0, X1 = meshgrid(x0, x1)
```

In [4]:
```
print("X0:\n", X0)
print("X1:\n", X1)
```

```
X0:
 [[-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]
 [-2.  -1.6 -1.2 -0.8 -0.4  0.   0.4  0.8  1.2  1.6  2. ]]
X1:
 [[-2.  -2.  -2.  -2.  -2.  -2.  -2.  -2.  -2.  -2.  -2. ]
 [-1.6 -1.6 -1.6 -1.6 -1.6 -1.6 -1.6 -1.6 -1.6 -1.6 -1.6]
 [-1.2 -1.2 -1.2 -1.2 -1.2 -1.2 -1.2 -1.2 -1.2 -1.2 -1.2]
 [-0.8 -0.8 -0.8 -0.8 -0.8 -0.8 -0.8 -0.8 -0.8 -0.8 -0.8]
 [-0.4 -0.4 -0.4 -0.4 -0.4 -0.4 -0.4 -0.4 -0.4 -0.4 -0.4]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.4  0.4  0.4  0.4  0.4  0.4  0.4  0.4  0.4  0.4  0.4]
 [ 0.8  0.8  0.8  0.8  0.8  0.8  0.8  0.8  0.8  0.8  0.8]
 [ 1.2  1.2  1.2  1.2  1.2  1.2  1.2  1.2  1.2  1.2  1.2]
 [ 1.6  1.6  1.6  1.6  1.6  1.6  1.6  1.6  1.6  1.6  1.6]
 [ 2.   2.   2.   2.   2.   2.   2.   2.   2.   2.   2. ]]
```

Apply `f()` to each of the points in the mesh:

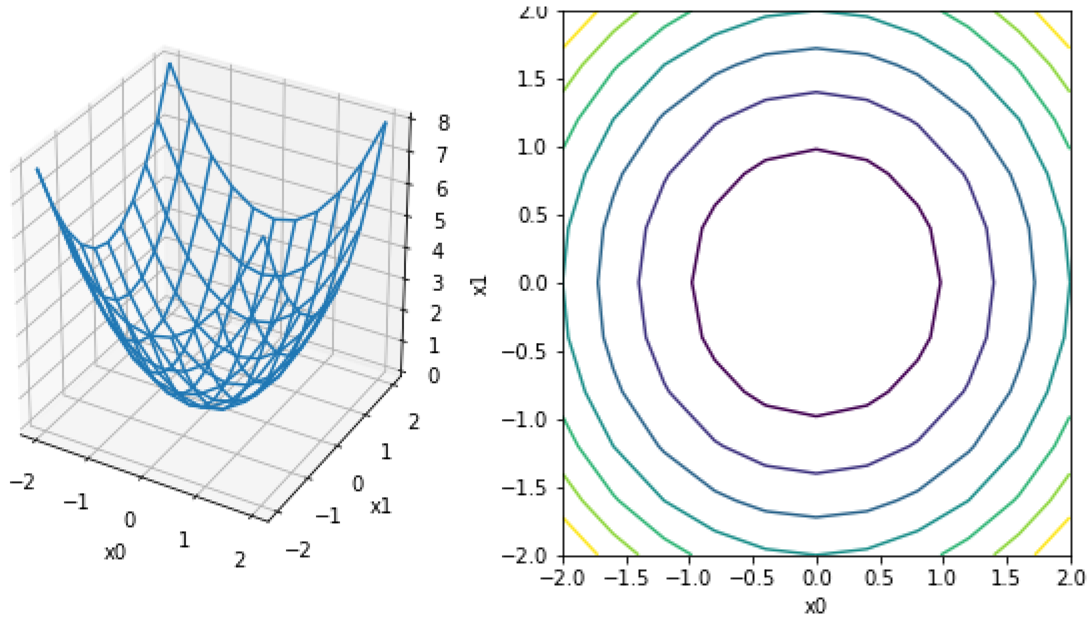
In [5]:
```
Z = f_vec(X0, X1)
print("Z:\n", Z)
```

```
Z:
 [[8.   6.56 5.44 4.64 4.16 4.   4.16 4.64 5.44 6.56 8.  ]
 [6.56 5.12 4.   3.2  2.72 2.56 2.72 3.2  4.   5.12 6.56]
 [5.44 4.   2.88 2.08 1.6  1.44 1.6  2.08 2.88 4.   5.44]
 [4.64 3.2  2.08 1.28 0.8  0.64 0.8  1.28 2.08 3.2  4.64]
 [4.16 2.72 1.6  0.8  0.32 0.16 0.32 0.8  1.6  2.72 4.16]
 [4.   2.56 1.44 0.64 0.16 0.   0.16 0.64 1.44 2.56 4.  ]
 [4.16 2.72 1.6  0.8  0.32 0.16 0.32 0.8  1.6  2.72 4.16]
 [4.64 3.2  2.08 1.28 0.8  0.64 0.8  1.28 2.08 3.2  4.64]
 [5.44 4.   2.88 2.08 1.6  1.44 1.6  2.08 2.88 4.   5.44]
 [6.56 5.12 4.   3.2  2.72 2.56 2.72 3.2  4.   5.12 6.56]
 [8.   6.56 5.44 4.64 4.16 4.   4.16 4.64 5.44 6.56 8.  ]]
```

Plot `Z[:, :]` as a three-dimensional surface:

In [6]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.pyplot import figure, xlabel, ylabel
%matplotlib inline

fig = figure(figsize=(10, 5))

ax3d = fig.add_subplot(121, projection='3d')
ax3d.plot_wireframe(X0, X1, Z)
xlabel('x0')
ylabel('x1')

ax2d = fig.add_subplot(122)
cp = ax2d.contour(X0, X1, Z)
xlabel('x0')
ylabel('x1')
```

Out[6]:
```
Text(0,0.5,'x1')
```



The gradient of $f(x)$ with respect to x is

$$\nabla_x f(x) \equiv \left[\begin{matrix} \dfrac{\partial f}{\partial x_0} \\ \dfrac{\partial f}{\partial x_1} \end{matrix}\right] = \left[\begin{matrix} \dfrac{\partial}{\partial x_0}\left(x_0^2 + x_1^2\right) \\ \dfrac{\partial}{\partial x_1}\left(x_0^2 + x_1^2\right) \end{matrix}\right] = \left[\begin{matrix} 2x_0 \\ 2x_1 \end{matrix}\right].$$

Exercise 1 (1 point). Implement a function, `grad_f(x0, x1)`, that implements the gradient $\nabla_x f(x)$ shown above. It should return a pair of val
gradient for this $f(x)$ has two components.

In [7]:
```
def grad_f(x0, x1):
    ### BEGIN SOLUTION
    return (2*x0, 2*x1)
    ### END SOLUTION
```

In [8]:
```
# Test cell: `grad_f_test`

grad_f_vec = vectorize(grad_f)
z = randn(5)
gx, gy = grad_f_vec(z, -z)
assert all(isclose(gx*0.5, z)) and all(isclose(gy*(-0.5), z)), "Your function might have a bug..

print("\n(Passed!)")
```

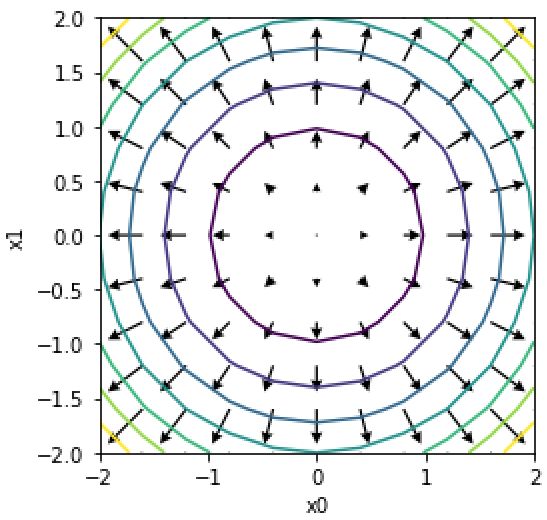
```
(Passed!)
```

Visualizing the gradient

Let's generate and plot $\nabla_x f(x)$:

In [9]:
```
dX0, dX1 = grad_f(X0, X1)

from matplotlib.pyplot import contour, quiver, axis
cp = contour (X0, X1, Z)
quiver(X0, X1, dX0, dX1, scale=40, headwidth=5)
xlabel('x0')
ylabel('x1')
axis('square')
```

Out[9]:
```
(-2.0, 2.0, -2.0, 2.0)
```



Fin! If you've gotten this far without errors, your notebook is ready to submit.

Part 2: Algorithms for the linear least squares problem

Recall the linear regression problem: given a data matrix, X, and responses y, we wish to determine the model parameters θ^* that minimizes $\|X$
problem is also known as the *linear least squares* problem.

Numpy has a function, [np.linalg.lstsq() (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html), that will compute linea
solutions for you. However, the purpose of this notebook is to give you a sense of how `lstsq()` works. So, instead of using it as a black box,
how one might implement `lstsq()` using two different numerical algorithms.

> You may rightly ask, why bother with such details? Here are three reasons it's worth looking more closely.
>
> 1. It's helpful to have some deeper intuition for how one formalizes a mathematical problem and derives a computational solution, in case you ever encounter a problem that does not exactly fit what a canned library can do for you.
> 2. If you have ever used a statistical analysis package, it's likely you have encountered "strange" numerical errors or warnings. Knowing how problems are derived can help you understand what might have gone wrong. We will see an example below.
> 3. Because data analysis is quickly evolving, it's likely that new problems and new models will not exactly fit the template of existing models. Therefore, it's possible you will need to derive a new model or know how to talk to someone who can derive one for you.

Implementation note. In this notebook, we ask that you use the following convention: any column vector should be *explicit*. That means its sh
have two dimensions where the column dimension equals one (1).

Exercise 0 (ungraded). Inspect the following code cell and make sure you understand the difference between two conventions for storing a vec
as a one-dimensional array versus as a two-dimensional array (matrix) where the number of columns equals one (1). When you are asked to pro
we will generally ask you to follow the second convention (`z_colvec`).

In [1]:
```
import numpy as np

# By default, Numpy vectors constructed from a list are 1-D
# and have neither a row nor a column orientation.
z_array = np.array([1.0, 2.0, 3.0])

# By contrast, we want you to ensure your vectors are
# column vectors.
z_colvec = np.reshape(z_array, (len(z_array), 1))

print("`z_array`:\n\n", z_array, "\n\n==> shape:", z_array.shape)
print("\n")
print("`z_colvec`:\n\n", z_colvec, "\n\n==> shape:", z_colvec.shape)
```

```
`z_array`:

 [1. 2. 3.]

==> shape: (3,)


`z_colvec`:

 [[1.]
 [2.]
 [3.]]

==> shape: (3, 1)
```

Before beginning, run this code cell to load some of the key modules you'll need.

In [2]:
```
# Data and computation
import numpy as np
import scipy as sp
import scipy.linalg
import pandas as pd

# Viz
from IPython.display import display, Math
from matplotlib.pyplot import figure, subplot, xlim, ylim
from matplotlib.pyplot import scatter, axis, xlabel, ylabel, title, plot
%matplotlib inline

# Some functions we'll use later to display results
def show_cond_fancy(x, name, opt=''):
    """Display a condition number in 'fancy' format (using LaTeX)."""
    def sci_to_latex(x, fmt='{:.2e}'):
        s_raw = fmt.format(x)
        s, e = s_raw.split('e')
        return s + r'\times 10^{{{}}}'.format(int(e))
    from IPython.display import Math
    x_s = sci_to_latex(x)
```

```
    display(Math(r'\kappa({}){} \approx {}'.format(name, opt, x_s)))

def show_2vecs_tibble(x, y, xname='x', yname='y', error=False):
    """Display two column vectors side-by-side in a tibble."""
    assert type(x) is np.ndarray and x.ndim >= 2 and x.shape[1] == 1
    assert type(y) is np.ndarray and y.ndim >= 2 and y.shape[1] == 1
    assert x.shape == y.shape
    x_df = pd.DataFrame(x, columns=[xname])
    y_df = pd.DataFrame(y, columns=[yname])
    df = pd.concat([x_df, y_df], axis=1)
    if error:
        df['error'] = x - y
    display(df)

# Display (X, y) problem as a tibble
def make_data_tibble(X, y=None):
    df = pd.DataFrame(X, columns=['x_{}'.format(i) for i in range(X.shape[1])])
    if y is not None:
        y_df = pd.DataFrame(y, columns=['y'])
        df = pd.concat([y_df, df], axis=1)
    return df

# From: https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
def nparray_to_bmatrix(a):
    """Returns a LaTeX bmatrix"""
    assert len(a.shape) <= 2, 'bmatrix can at most display two dimensions'
    lines = str(a).replace('[', '').replace(']', '').splitlines()
    rv = [r'\begin{bmatrix}']
    rv += ['  ' + ' & '.join(l.split()) + r'\\' for l in lines]
    rv +=  [r'\end{bmatrix}']
    return '\n'.join(rv)

# Stash this function for later:
SAVE_LSTSQ = np.linalg.lstsq # You may ignore this line, which some test cells will use
```

Notation and review

Here is a quick summary of how we can formulate and approach the linear regression problem. For a more detailed derivation, see these accor notes (https://courses.edx.org/asset-v1:GTx+CSE6040x+1T2018+type@asset+block@nb12-notes-linreg.html).

Your data consists of m observations and $n + 1$ variables. One of these variables is the *response* variable, y, which you want to predict from the variables, $\{x_0, \ldots, x_{n-1}\}$. You wish to fit a *linear model* of the following form to these data,

$$y_i \approx x_{i,0}\theta_0 + x_{i,1}\theta_1 + \cdots + x_{i,n-1}\theta_{n-1} + \theta_n,$$

where $\{\theta_j \,|\, 0 \leq j \leq n\}$ is the set of unknown coefficients. Your modeling task is to choose values for these coefficients that "best fit" the data.

If we further define a set of dummy variables, $x_{i,n} \equiv 1.0$, associated with the θ_n parameter, then the model can be written more compactly in ma as

$$y \approx X\theta,$$

where we will refer to X as the (input) data matrix.

Visually, you can also arrange the observations into a tibble like this one:

y	x₀	x₁	⋯	x_{n-1}	x_n
y_0	$x_{0,1}$	$x_{0,2}$	⋯	$x_{0,n-1}$	1.0
y_1	$x_{1,1}$	$x_{1,2}$	⋯	$x_{1,n-1}$	1.0
y_2	$x_{2,1}$	$x_{2,2}$	⋯	$x_{2,n-1}$	1.0
⋮	⋮	⋮	⋮	⋮	1.0
y_{m-1}	$x_{m-1,1}$	$x_{m-1,2}$	⋯	$x_{m-1,n-1}$	1.0

This tibble includes an extra column (variable), x_n, whose entries are all equal to 1.0.

Synthetic problem generator. For the exercises in this notebook, we will generate synthetic data. The function, `gen_problem(m, n)`, will re y, `theta`, which are an `m` x `(n+1)` data matrix `X`, a response vector `y`, and the "true" model parameters `theta`. We will then run two different nu algorithms that estimate `theta` from `X` and `y`, and see how their answers compare against the true value.

> **Note 1.** The problem generator constructs the data matrix X such that each entry (i, j) is i^j. This structure makes it an instance of a Vandermonde matrix (https://en.wikipedia.org/wiki/Vandermonde_matrix), which arises when fitting a polynomial to data. The "true" parameter vector θ is set to all ones, and y computed simply by summing the rows.
>
> **Note 2.** Although our usual convention is to make the *last* column all ones, the Vandermonde matrix has its *first* column set to all ones. ordering is not important in this problem, but it does mean one would interpret θ_0 as the intercept rather than θ_n, which will be our usua convention.

In [3]:
```
def gen_problem(m, n):
    from numpy import arange, tile, cumprod, insert, ones
    # 1 + x + x^2 + ... + x^n, x = 0:m
    X = np.empty((m, n+1))
    x_col = arange(m).reshape((m, 1)) # 0, 1, 2, ..., m-1
    X[:, 0] = 1.0
    X[:, 1:] = tile(x_col, reps=(1, n))
    X[:, 1:] = cumprod(X[:, 1:], axis=1)
    theta = ones((n+1, 1))
    y = np.sum(X, axis=1).reshape((m, 1))
    return X, y, theta

print("Sample generated problem:")
m, n = 10, 2
X, y, theta_true = gen_problem(m, n)

display(Math(r'X = {}, \quad y = {} \quad \implies \quad \theta^* = {}'.format(nparray_to_bmatr:
                                                                                nparray_to_bmatr:
                                                                                nparray_to_bmatr:
ue))))
```

```
Sample generated problem:
```

$$X = \begin{bmatrix} 1. & 0. & 0. \\ 1. & 1. & 1. \\ 1. & 2. & 4. \\ 1. & 3. & 9. \\ 1. & 4. & 16. \\ 1. & 5. & 25. \\ 1. & 6. & 36. \\ 1. & 7. & 49. \\ 1. & 8. & 64. \\ 1. & 9. & 81. \end{bmatrix}, \quad y = \begin{bmatrix} 1. \\ 3. \\ 7. \\ 13. \\ 21. \\ 31. \\ 43. \\ 57. \\ 73. \\ 91. \end{bmatrix} \quad \implies \quad \theta^* = \begin{bmatrix} 1. \\ 1. \\ 1. \end{bmatrix}$$

We are interested primarily in *overdetermined systems*, meaning X has more rows than columns, i.e., $m > n + 1$, as shown above. That's becaus have more observations (data points, or rows) than predictors (variables or columns). For such problems, there is generally no unique solution.

Therefore, to identify some solution, we need to ask for the "best" fit and say what we mean by "best." For linear regression, the usual definitio *minimizing* the sum-of-squared residual error:

$$\theta^* = \arg\min_\theta \|X\theta - y\|_2^2.$$

Solving this minimization problem is equivalent to solving a special system known as the *normal equations*,

$$X^T X\theta^* = X^T y.$$

So, our computational task is to solve this problem.

Algorithm 1: Direct solution of the normal equations

The preceding calculation immediately suggests the following algorithm to estimate θ^*. Given X and y:

1. Form $C \equiv X^T X$. This object is sometimes called the Gram matrix (https://en.wikipedia.org/wiki/Gramian_matrix) or Gramian of X.
2. Form $b \equiv X^T y$.
3. Solve $C\theta^* = b$ for θ^*.

But, is this a "good" algorithm? There are at least three dimensions along which we might answer this question.

1. Is it accurate enough?
2. Is it fast enough?
3. Is it memory-efficient enough?

Let's examine these questions by experiment.

Exercise 1 (3 points). Implement a function, `solve_neq(X, y)` that implements Algorithm 1. It should return a Numpy vector containing the r parameter estimates.

Recall the steps of the algorithm as previously outlined:

1. Form the Gramian of X, $C \equiv X^T X$.
2. Form $b \equiv X^T y$.

3. Solve $C\theta^* = b$ for θ^*.

Your algorithm should carry out these steps. For the third step, use Scipy's routine, `scipy.linalg.solve()` (https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html#solving-linear-system). It has an option that allows you to indicate that C is symr positive definite, which will be true of C for our synthetic problem.

> The code cell will run your function to compute a set of parameter estimates. It will store these in a variable named `theta_neq`, which refer to later.

In [4]:
```
def solve_neq(X, y):
    ### BEGIN SOLUTION
    C = X.T.dot(X)
    b = X.T.dot(y)
    theta_est = sp.linalg.solve(C, b, sym_pos=True)
    return theta_est
    ### END SOLUTION

theta_neq = solve_neq(X, y)

print("Your implementation's solution versus the true solution:")
show_2vecs_tibble(theta_neq, theta_true, xname='theta_neq', yname='theta_true', error=True)
```

```
Your implementation's solution versus the true solution:
```

	theta_neq	theta_true	error
0	1.0	1.0	-2.065015e-14
1	1.0	1.0	4.218847e-15
2	1.0	1.0	0.000000e+00

In [5]:
```
# Test cell: `solve_neq_test`

try:
    del np.linalg.lstsq
    solve_neq(X, y)
except NameError as n:
    if re.findall('lstsq', n.args[0]):
        print("*** Double-check that you did not try to use `lstsq()`. ***")
    raise n
except AttributeError as a:
    if re.findall('lstsq', a.args[0]):
        print("*** Double-check that you did not try to use `lstsq()`. ***")
    raise a
finally:
    np.linalg.lstsq = SAVE_LSTSQ

assert type(theta_neq) is np.ndarray, "`theta_neq` should be a Numpy array, but isn't."
assert theta_neq.shape == (n+1, 1), "`theta_neq.shape` is {} instead of {}.".format(theta_neq.sl
1))

assert (np.abs(theta_neq - theta_true) <= 1e-12).all(), \
       "Your `theta_neq` does not match the true solution, `theta_true`."

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 2 (1 point). Write a function to calculate the residual norm, $\|r\|_2 = \|X\theta^* - y\|_2$.

> Although we are minimizing $\|r\|_2^2$, for this exercise your function should return $\|r\|_2$.

In [6]:
```
def calc_residual_norm(X, y, theta):
    ### BEGIN SOLUTION
    from numpy.linalg import norm
    return norm(X.dot(theta) - y, ord=2)
    ### END SOLUTION

r_norm_neq = calc_residual_norm(X, y, theta_neq)
print("\nThe squared residual norm:", r_norm_neq)
```

```
The squared residual norm: 3.766978705903275e-14
```

In [7]:
```
# Test cell: `calc_residual_norm_test`

r_norm_neq = calc_residual_norm(X, y, theta_neq)
assert 1e-16 <= np.abs(r_norm_neq) <= 1e-12
print ("\n(Passed.)")
```

```
(Passed.)
```

Sources of error

We said before that one question we should ask about our algorithm is whether it is "accurate enough." But what does that mean?

Exercise 3 (ungraded). For any modeling problem, there will be several sources of error. Describe at least three such sources.

Answer. Here are some possibilities.

1. There will be errors in the inputs. That is, the data itself may only represent measurements of a certain accuracy.
2. There will be errors in the model. That is, the model is only an approximation of the underlying phenomena.
3. There will be errors in the algorithm. That is, you may implement an algorithm that can only approximately estimate the parameters of the r
4. There will be roundoff errors. Recall that floating-point arithmetic necessarily represents all values finitely, which means you may lose accu time you do an arithmetic operation.

Perturbations. One way to understand error in a numerical computation is to consider how sensitive the computed solution is to perturbations

That is, suppose we change X by an amount ΔX. We can then ask by how much the computed model parameters θ^* change. If they change by our method for computing them may be overly sensitive to perturbations. Instead, we might prefer one method over another one that is more s changes.

Let's see how Algorithm 1 fares under small perturbations. But first, we'll need a method to generate a random perturbation of a certain maxim

Exercise 4 (2 points). Implement a function that returns an $m \times n$ matrix whose entries are uniformly randomly distributed in the interval, $[0, \epsilon]$ fo of ϵ.

> Hint: Check out Numpy's module for generating (pseudo)random numbers: `numpy.random` (https://docs.scipy.org/doc/numpy/reference/routines.random.html)

In [8]:
```
def random_mat (m, n, eps):
    ### BEGIN SOLUTION
    return np.random.random((m, n)) * eps
    ### END SOLUTION

print(random_mat(3, 2, 1e-3))
```

```
[[0.00059339 0.00057468]
 [0.00092299 0.00063703]
 [0.0003482  0.00091574]]
```

In [9]:
```
# Test cell: `rand_eps_test`

Z = random_mat (5, 3, 1e-2)
assert Z.shape == (5, 3)
assert ((Z >= 0) & (Z <= 1e-2)).all()
print("\n(Passed.)")
```

```
(Passed.)
```

Exercise 5 (2 points). Use your `random_mat()` function to write another function, `perturb_system(X, y, eps)`, that creates two "perturb system defined by `X` and `y`.

1. Let ΔX be the first perturbation. It should have the same dimensions as X, and its entries should lie in the interval $[-\epsilon, \epsilon]$. The value of ϵ is g `eps`.
2. The second is Δy, a small perturbation to the response variable, y. Its entries should also lie in the same interval, $[-\epsilon, \epsilon]$,

Your function should return a perturbed system, $X + \Delta X$ and $y + \Delta y$, as a pair.

In [10]:
```
def perturb_system(X, y, eps):
    ### BEGIN SOLUTION
    Delta_X = random_mat(X.shape[0], X.shape[1], 2*eps) - eps
    Delta_y = random_mat(y.shape[0], y.shape[1], 2*eps) - eps
    return X + Delta_X, y + Delta_y
    ### END SOLUTION

EPSILON = 0.1
X_perturbed, y_perturbed = perturb_system(X, y, EPSILON)

Delta_X = X_perturbed - X
Delta_y = y_perturbed - y
display(Math(r'\Delta X = {}, \quad \Delta y = {}'.format(nparray_to_bmatrix(Delta_X[:5, :]),
                                                         nparray_to_bmatrix(Delta_y[:5]))))
```

$$\Delta X = \begin{bmatrix} -0.0106095 & 0.01422098 & -0.06028698 \\ 0.06029219 & -0.02538373 & 0.08494938 \\ 0.06021075 & -0.09076259 & 0.07316003 \\ 0.03928084 & -0.05499615 & -0.07508389 \\ -0.00887685 & 0.07215012 & -0.03684651 \end{bmatrix}, \quad \Delta y = \begin{bmatrix} -0.04424354 \\ 0.01997222 \\ -0.02917007 \\ 0.04818775 \\ 0.06929618 \end{bmatrix}$$

In [11]:

```
# Test cell: `delta_X_test`

Delta_X = X_perturbed - X
Delta_y = y_perturbed - y

assert Delta_X.shape == X.shape, "`Delta_X` has shape {} instead of {}.".format(Delta_X.shape, X
assert (np.abs(Delta_X) <= EPSILON).all(), "The perturbation lies outside the interval, [-{}, {}
EPSILON, EPSILON)

assert Delta_y.shape == y.shape, "`Delta_y` has shape {} instead of {}.".format(Delta_y.shape, y
assert (np.abs(Delta_y) <= EPSILON).all(), "The perturbation lies outside the interval, [-{}, {}
EPSILON, EPSILON)
print ("\n(Passed.)")
```

```
(Passed.)
```

Sensitivity of Algorithm 1

Let's now run the following code, which uses your code from above to perform a "sensitivity experiment." In particular, the function `run_perturbation_trials()` will repeatedly perturb the system and measure the resulting change to the estimated θ^*.

All of the estimated θ^* are stored in an array, `Thetas_neq`. Each *column* `k` of `Thetas_neq`, or `Thetas_neq[:, k]`, is one of the calculated under a random perturbation of the system.

The size of the random perturbation is set, by default, to `eps=0.01`. Recall that our synthetic problem consists of numerical values that are all equal to one, so this perturbation may be regarded as fairly small.

In [12]:

```
def run_perturbation_trials(solver, X, y, eps=0.01, trials=100):
    Thetas = np.zeros((X.shape[1], trials)) # Store all computed thetas
    for t in range(trials):
        X_p, y_p = perturb_system(X, y, eps)
        Thetas[:, t:t+1] = solver(X_p, y_p)
    return Thetas

Thetas_neq = run_perturbation_trials(solve_neq, X, y)

print("Unperturbed solution:")
print(theta_neq)

print("First few perturbed solutions (columns):")
print(Thetas_neq[:, :5])
```

```
Unperturbed solution:
[[1.]
 [1.]
 [1.]]
First few perturbed solutions (columns):
[[0.99028632 0.99231283 1.02243806 1.00139275 1.01358837]
 [1.00529885 1.00407045 0.99118553 0.99889269 0.9952955 ]
 [0.99954928 0.99960384 1.00074977 1.00007595 1.0003454 ]]
```

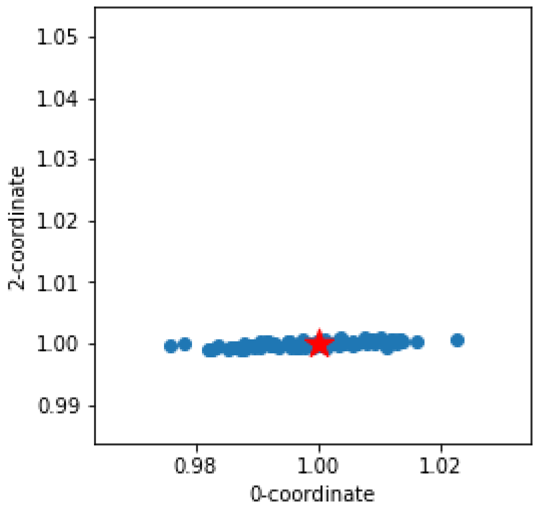
Here is a quick plot of the that shows two coordinates of the true parameters (red star), compared to all perturbed estimates (blue points). We more confidence in the algorithm's computed solutions if it did not appear to be too sensitive to changes in the input.

Since θ may have more than two coordinates, the code below shows the first two coordinates.

In [13]:

```
# Makes a 2-D scatter plot of given theta values.
# If the thetas have more than two dimensions, only
# the first and last are displayed by default.
# (Override by setting ax and ay.)

def scatter_thetas(Thetas, theta_true=None, ax=0, ay=-1, xylim=None, title=None):
    import matplotlib.pyplot as plt
    assert type(Thetas) is np.ndarray and Thetas.shape[0] >= 2
    scatter(Thetas[ax, :], Thetas[ay, :])
    xlabel('{}-coordinate'.format(ax if ax >= 0 else Thetas.shape[0]+ax))
    ylabel('{}-coordinate'.format(ay if ay >= 0 else Thetas.shape[0]+ay))
    if xylim is not None:
        axis(xylim)
    else:
        axis('equal')
    if theta_true is not None:
        assert type(theta_true) is np.ndarray and theta_true.shape[0] >= 2 and theta_true.shape
        scatter(theta_true[ax], theta_true[ay], marker='*', color='red', s=15**2)
    if title is not None:
        plt.title(title)
```

```
        plt.title(title)

def calc_lims(x, buffer=0.1):
    xmin = x.min()
    xmax = x.max()
    dx = (xmax - xmin) * buffer
    return xmin-dx, xmax+dx

scatter_thetas(Thetas_neq, theta_true=theta_true, ax=0, ay=2)
axis('square');
```



You should observe that the change in the estimates are of the same order as the perturbation. So for this example system, the algorithm seem enough.

Stress-testing Algorithm 1

This experiment suggests all is fine. But what should we *expect* to happen?

We've prepared another notebook (https://courses.edx.org/asset-v1:GTx+CSE6040x+1T2018+type@asset+block@nb12-notes-cond.html) that through an analysis of solving linear systems. It turns out you can estimate how hard it is to solve a linear system using a measure called the cc *number*. We can denote the condition number of solving a system by $\kappa(X)$ where X is the matrix. The larger this number is, the more sensitive th

In Numpy, there is a condition number estimator that will tell us approximately what the condition number is for a given matrix. Let's compare κ $\kappa(C) = \kappa(X^T X)$:

In [14]:
```
cond_X = np.linalg.cond(X)
cond_XTX = np.linalg.cond(X.T.dot(X))

assert 1. <= cond_X <= 3e3
assert 1. <= cond_XTX <= 6e6

show_cond_fancy(cond_X, 'X')
show_cond_fancy(cond_XTX, 'X^T X')
show_cond_fancy(cond_X**2, 'X', opt='^2')
```

$$\kappa(X) \approx 1.07 \times 10^2$$

$$\kappa(X^T X) \approx 1.15 \times 10^4$$

$$\kappa(X)^2 \approx 1.15 \times 10^4$$

Ill-conditioning. As it happens, $\kappa(C)$ is roughly the **square** of $\kappa(X)$. So, by forming C explicitly and then trying to solve a system based on it, we problem *more* difficult. Indeed, if the problem is ill-conditioned enough, this algorithm based on directly constructing the normal equations will different results even under small changes, and we call the algorithm *unstable*.

In this particular example, the condition numbers are not very "big." You would be more concerned if the condition numbers were close to $1/\epsilon$, machine epsilon. In double-precision, recall that $\epsilon_d \approx 10^{-15}$, so the values shown above is nothing to be worried about.

But what if we had a "hard" problem, that is, one whose condition number is large? The synthetic data generator allows us to create such a prc making the problem bigger. Let's try that next.

In [15]:
```
# Generate a "hard" problem
m_hard, n_hard = 100, 6
X_hard, y_hard, theta_hard_true = gen_problem(m_hard, n_hard)

df_hard = make_data_tibble(X_hard, y_hard)
print("First few rows of data:")
df_hard.head()
print("True parameter estimates:\n{}".format(theta_hard_true))

cond_X_hard = np.linalg.cond(X_hard)
```

```
cond_XTX_hard = np.linalg.cond(X_hard.T.dot(X_hard))

name_X_hard = 'X_h'
show_cond_fancy(cond_X_hard, name_X_hard)
show_cond_fancy(cond_XTX_hard, '{}^T {}'.format(name_X_hard, name_X_hard))
```

```
First few rows of data:
True parameter estimates:
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]
 [1.]]
```

$$\kappa(X_h) \approx 1.72 \times 10^{12}$$

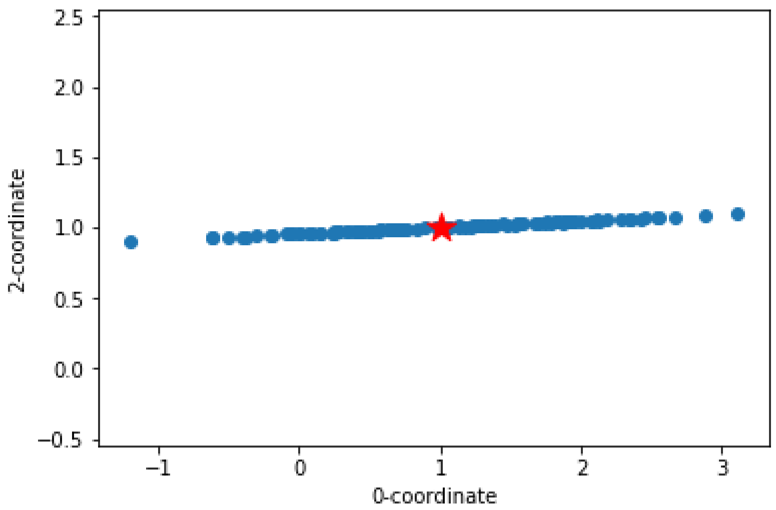
$$\kappa(X_h^T X_h) \approx 2.82 \times 10^{23}$$

These condition numbers are much larger. So, let's run the same sensitivity experiment as before, and see how the estimate varies for the hard does it compare to the well-conditioned case?

In [16]:
```
Thetas_hard_neq = run_perturbation_trials(solve_neq, X_hard, y_hard)
scatter_thetas(Thetas_hard_neq, theta_true=theta_hard_true, ax=0, ay=2)

print("Residual norm for one of the trials:")
theta_hard_neq_example = np.random.randint(Thetas_hard_neq.shape[1])
calc_residual_norm(X_hard, y_hard, theta_hard_neq_example)
```

```
Residual norm for one of the trials:
```

Out[16]: `72708670496575.33`



Observe that the computed estimates can be relatively far from the true value, even getting the sign completely wrong in the case of the θ_0.

Algorithm 2: QR decomposition

A different method for solving an overdetermined systems is to use a tool from linear algebra known as the *QR decomposition* (https://en.wikipedia.org/wiki/QR_decomposition).

Here is how we can use QR. If X has linearly independent columns, then we would first factor the $m \times n$ matrix X into the product $X = QR$, where orthogonal matrix and R is an invertible $n \times n$ upper-triangular matrix. (These dimensions assume $m \geq n$.) That Q is orthogonal means that $Q^T Q$ = matrix; R being upper-triangular means all of its entries below the main diagonal are zero.

Next, observe that the normal equations can be transformed if we substitute $X = QR$:

$$\begin{aligned} X^T X \theta^* &= X^T y \\ R^T Q^T Q R \theta^* &= R^T Q^T y \\ R \theta^* &= Q^T y. \end{aligned}$$

Lastly, because R is triangular, solving a system is "easy" using *(backward) substitution*. Consider the following 3×3 example (taken from here (http://www.purplemath.com/modules/systlin6.htm)):

$$\begin{bmatrix} 5 & 4 & -1 \\ & 10 & -3 \\ & & 1 \end{bmatrix} \cdot \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 11 \\ 3 \end{bmatrix}.$$

Because it is upper-triangular, you can see right away that $1 \cdot \theta_2 = 3 \implies \theta_2 = 3$. Then, going to the equation above it, $10\theta_1 - 3\theta_2 = 10\theta_1 - 3(3) = 11 \implies \theta_1 = 2$. Lastly, $5\theta_0 + 4\theta_1 - \theta_2 = 5\theta_0 + 4(2) - 3 = 0 \implies \theta_0 = -1$.

So, to summarize, a different algorithm to solve $X\theta^* \approx y$ using QR would look like the following:

1. Compute $X = QR$.
2. Form the modified right-hand side, $z = Q^T y$.
3. Use back-substitution to solve $R\theta^* = z$.

Conditioning. What about the sensitivity of this algorithm? Given R, we only need to solve linear systems involving R. Therefore, it's $\kappa(R)$ that w the stability of the algorithm. So if $\kappa(R)$ is comparable to $\kappa(X)$, then the algorithm should be as stable as one can expect any algorithm to be.

Exercise 6 (1 point). Use numpy.linalg.qr() (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.qr.html) to compute the QR de X (precomputed above as the variable, `X`). Store the Q and R factors in two variables named `Q` and `R`.

In [17]:
```
print(X[:5], "\n ...\n")

### BEGIN SOLUTION
Q, R = np.linalg.qr(X)
### END SOLUTION

# Print the dimensions of your result
print("Q:", Q.shape, "\n")
print("R:", R.shape, "==")
print(R)
```

```
[[ 1.  0.  0.]
 [ 1.  1.  1.]
 [ 1.  2.  4.]
 [ 1.  3.  9.]
 [ 1.  4. 16.]]
 ...

Q: (10, 3)

R: (3, 3) ==
[[ -3.16227766 -14.23024947 -90.12491331]
 [  0.           9.08295106  81.74655956]
 [  0.           0.          22.97825059]]
```

In [18]:
```
# Test cell: `qr_test`

assert type(Q) is np.ndarray, "`Q` is not a Numpy array but should be."
assert type(R) is np.ndarray, "`R` is not a Numpy array but should be."
assert Q.shape == (m, n+1), "`Q` has the wrong shape: it's {} rather than {}.".format(Q.shape, 
assert R.shape == (n+1, n+1), "`R` has the wrong shape: it's {} rather than {}.".format(R.shape,
for i in range(R.shape[0]):
    for j in range(i):
        assert np.isclose(R[i][j], 0.0), "R[{}][{}] == {} instead of 0!".format(i, j, R[i][j])

QTQ = Q.T.dot(Q)
assert np.isclose(QTQ, np.eye(Q.shape[1])).all(), "Q^T Q is not nearly the identity matrix, as 
e."

assert np.isclose(X, Q.dot(R)).all(), "QR is not sufficiently close in values to X!"

print("\n(Passed!)")
```

```
(Passed!)
```

Condition number of R. Let's check the condition number of R empirically, to verify that it is comparable to $\kappa(X)$.

In [19]:
```
cond_R = np.linalg.cond(R)

show_cond_fancy(cond_X, 'X')
show_cond_fancy(cond_XTX, 'X^T X')
show_cond_fancy(cond_R, 'R')
```

$$\kappa(X) \approx 1.07 \times 10^2$$

$$\kappa(X^T X) \approx 1.15 \times 10^4$$

$$\kappa(R) \approx 1.07 \times 10^2$$

Exercise 7 (3 points). Implement a function, `solve_qr(X, y)`, which uses the QR-based algorithm to estimate θ^*.

> To solve the triangular system, use Scipy's specialized function, available as sp.linalg.solve_triangular() (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve_triangular.html).

In [20]:
```
import scipy.linalg

def solve_qr(X, y):
    ### BEGIN SOLUTION
    Q, R = np.linalg.qr(X)
    b = Q.T.dot(y)
    theta = sp.linalg.solve_triangular(R, b) # Solves R u = b
    return theta
    ### END SOLUTION

theta_qr = solve_qr(X, y)

print("Comparing your QR solution to the true solution:")
show_2vecs_tibble(theta_qr, theta_true, xname='theta_qr', yname='theta_true', error=True)

print("Residual norm:")
calc_residual_norm(X, y, theta_qr)
```

```
Comparing your QR solution to the true solution:
```

	theta_qr	theta_true	error
0	1.0	1.0	-1.154632e-14
1	1.0	1.0	3.552714e-15
2	1.0	1.0	-2.220446e-16

```
Residual norm:
```

Out[20]: `1.5434895314732317e-14`

In [21]:
```
# Test cell: `solve_qr_test`
import re

try:
    del np.linalg.lstsq
    solve_qr(X, y)
except NameError as n:
    if re.findall('lstsq', n.args[0]):
        print("*** Double-check that you did not try to use `lstsq()`. ***")
    raise n
except AttributeError as a:
    if re.findall('lstsq', a.args[0]):
        print("*** Double-check that you did not try to use `lstsq()`. ***")
    raise a
finally:
    np.linalg.lstsq = SAVE_LSTSQ

assert np.isclose(theta_qr, theta_true).all(), "Your QR-based solution should be closer to the t
on."

print("\n(Passed!)")
```

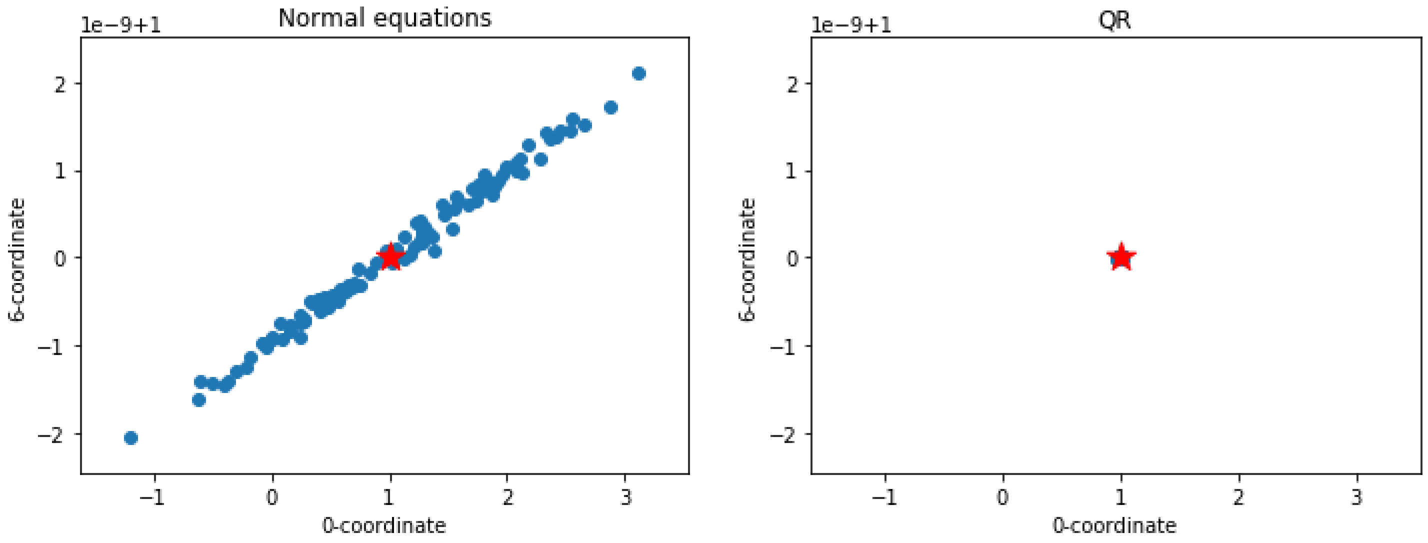
```
(Passed!)
```

Is QR more stable? Let's run the same perturbation experiments on the "hard" regression problem and see the result.

In [22]:
```
Thetas_hard_qr = run_perturbation_trials(solve_qr, X_hard, y_hard)

# Plot side-by-side against normal equations method
def compare_scatter_thetas(T0, title0, T1, title1, ax=0, ay=1, **kwargs):
    xmin, xmax = calc_lims(np.array([Thetas_hard_neq[ax, :], Thetas_hard_qr[ax, :]]))
    ymin, ymax = calc_lims(np.array([Thetas_hard_neq[ay, :], Thetas_hard_qr[ay, :]]))
    xylim = [xmin, xmax, ymin, ymax]
    figure(figsize=(12, 4))
    subplot(1, 2, 1)
    scatter_thetas(T0, title=title0, ax=ax, ay=ay, xylim=xylim, **kwargs)
    subplot(1, 2 ,2)
    scatter_thetas(T1, title=title1, ax=ax, ay=ay, xylim=xylim, **kwargs)

compare_scatter_thetas(Thetas_hard_neq, 'Normal equations',
                       Thetas_hard_qr, 'QR',
                       ax=0, ay=-1, theta_true=theta_hard_true)

print("Sample estimate for one of the trials:")
theta_hard_neq_example = Thetas_hard_neq[:, np.random.randint(Thetas_hard_neq.shape[1])]
theta_hard_qr_example = Thetas_hard_qr[:, np.random.randint(Thetas_hard_qr.shape[1])]
msg = "- {}-based method: theta^T =\n\t{}"
print(msg.format("Gramian", theta_hard_neq_example.T))
```

```
print(msg.format("QR", theta_hard_qr_example.T))
```

```
Sample estimate for one of the trials:
- Gramian-based method: theta^T =
        [0.64091071 1.16266565 0.98363847 1.00065254 0.99998782 1.00000011
 1.        ]
- QR-based method: theta^T =
        [0.98958625 0.99839116 1.00028587 0.99998727 1.00000024 1.
 1.        ]
```



You should observe that the QR-based method does, indeed, produce estimates much closer to the true value despite the problem's high conc

Performance tradeoff. Although QR produces more reliable results, there can be a performance tradeoff, as the following quick test should sh

In [23]:
```
print("=== Performance of the normal equations-based algorithm ===")
%timeit solve_neq(X_hard, y_hard)

print("\n=== Performance of the QR-based algorithm ===")
%timeit solve_qr(X_hard, y_hard)
```

```
=== Performance of the normal equations-based algorithm ===
87.7 µs ± 1.46 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

=== Performance of the QR-based algorithm ===
146 µs ± 2.17 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Part 3: The cost of solving the normal equations

This notebook helps you explore the execution time cost of solving the normal equations,

$$X^T X\theta^* = X^T y.$$

This notebook only has one exercise, but it is not graded. So, you should complete it for your own edification.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Scalability with the problem size

To start, here is some code to help generate synthetic problems of a certain size, namely, $m \times (n + 1)$, where m is the number of observations number of predictors. The $+1$ comes from our usual dummy coefficient for a non-zero intercept.

We will also implement a linear least squares solver, `estimate_coeffs()`, that simply calls Numpy's `lstsq()` routine.

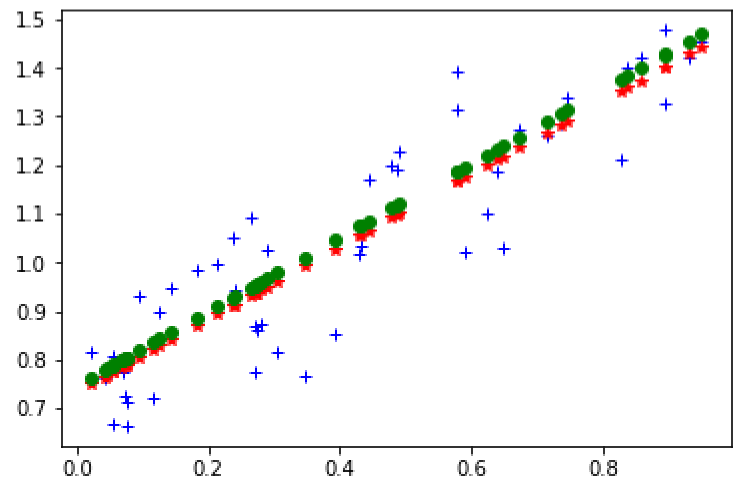
In [2]:
```
def generate_model (n):
    """Returns a set of (random) n+1 linear model coefficients."""
    return np.random.rand (n+1, 1)

def generate_data (m, theta, sigma=1.0/(2**0.5)):
    """
    Generates 'm' noisy observations for a linear model whose
    predictor (non-intercept) coefficients are given in 'theta'.
    Decrease 'sigma' to decrease the amount of noise.
    """
    assert (type (theta) is np.ndarray) and (theta.ndim == 2) and (theta.shape[1] == 1)
    n = len (theta)
    X = np.random.rand (m, n)
    X[:, 0] = 1.0
```

```
    y = X.dot (theta) + sigma*np.random.randn (m, 1)
    return (X, y)

def estimate_coeffs(X, y):
    """
    Solves X*theta = y by a linear least squares method.
    """
    result = np.linalg.lstsq (X, y, rcond = None)
    theta = result[0]
    return theta
```

In [3]:
```
# Demo the above routines for a 2-D dataset.

m = 50
theta_true = generate_model (1)
(X, y) = generate_data (m, theta_true, sigma=0.1)

print ("Dimensions of X:", X.shape)
print ("Dimensions of theta_true:", theta_true.shape)
print ("Dimensions of y:", y.shape)

print ("Condition number of X: ", np.linalg.cond (X))
print ("True model coefficients:", theta_true.T)

theta = estimate_coeffs (X, y)

print ("Estimated model coefficients:", theta.T)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.plot (X[:, 1], y, 'b+') # Noisy observations
ax1.plot (X[:, 1], X.dot (theta), 'r*') # Fit
ax1.plot (X[:, 1], X.dot (theta_true), 'go') # True solution
```

```
Dimensions of X: (50, 2)
Dimensions of theta_true: (2, 1)
Dimensions of y: (50, 1)
Condition number of X:  4.1056142618693565
True model coefficients: [[0.74734942 0.76050905]]
Estimated model coefficients: [[0.73296025 0.74693774]]
```

Out[3]: [<matplotlib.lines.Line2D at 0x7f04ece9b8d0>]



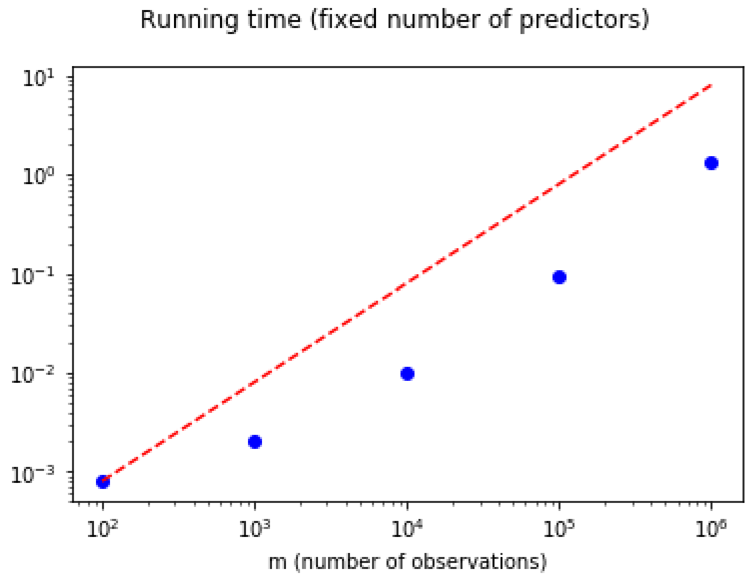
Benchmark varying m. Let's benchmark the time to compute x when the dimension n of each point is fixed but the number m of points varies the running time scale with m?

In [4]:
```
# Benchmark, as 'm' varies:

n = 32 # dimension
M = [100, 1000, 10000, 100000, 1000000]
times = [0.] * len (M)
for (i, m) in enumerate (M):
    theta_true = generate_model (n)
    (X, y) = generate_data (m, theta_true, sigma=0.1)
    t = %timeit -o estimate_coeffs (X, y)
    times[i] = t.best
```

```
804 µs ± 7.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.13 ms ± 931 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
10.8 ms ± 899 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
107 ms ± 13.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.39 s ± 47.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [5]:
```
t_linear = [times[0]/M[0]*m for m in M]

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.loglog (M, times, 'bo')
ax1.loglog (M, t_linear, 'r--')
ax1.set_xlabel ('m (number of observations)')
fig.suptitle ('Running time (fixed number of predictors)')
```

```
Out[5]: Text(0.5,0.98,'Running time (fixed number of predictors)')
```



Exercise 0 (ungraded). Now fix the number m of observations but vary the dimension n. How does time scale with n? Complete the benchmar to find out. In particular, given the array `N[:]`, compute an array, `times[:]`, such that `times[i]` is the running time for a problem of size `m`$\times$(

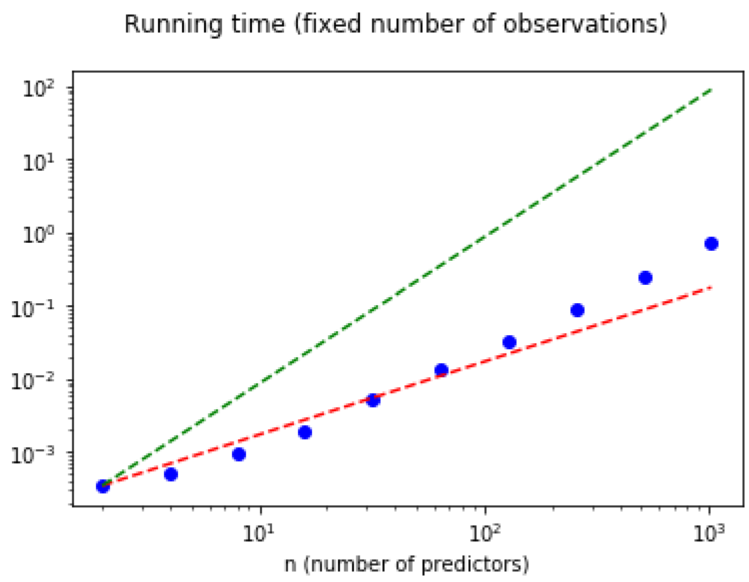
> Hint: You can adapt the preceding benchmark. Also, note that the code cell following the one immediately below will plot your results a $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$.

```
In [6]: N = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
m = 5000
times = [0.] * len (N)

# Implement a benchmark to compute the time,
# `times[i]`, to execute a problem of size `N[i]`.
for (i, n) in enumerate (N):
    ### BEGIN SOLUTION
    theta_true = generate_model (n)
    (X, y) = generate_data (m, theta_true, sigma=0.1)
    t = %timeit -o estimate_coeffs (X, y)
    times[i] = t.best
    ### END SOLUTION
```

```
347 µs ± 238 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
520 µs ± 372 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
998 µs ± 75.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.96 ms ± 171 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
5.82 ms ± 454 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
16.7 ms ± 1.78 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
40.1 ms ± 7.43 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
97.9 ms ± 12.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
249 ms ± 366 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
876 ms ± 178 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [7]: t_linear = [times[0]/N[0]*n for n in N]
t_quadratic = [times[0]/N[0]/N[0]*n*n for n in N]

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.loglog (N, times, 'bo')
ax1.loglog (N, t_linear, 'r--')
ax1.loglog (N, t_quadratic, 'g--')
ax1.set_xlabel ('n (number of predictors)')
fig.suptitle ('Running time (fixed number of observations)')
```

```
Out[7]: Text(0.5,0.98,'Running time (fixed number of observations)')
```



Thus, the empirical scaling appears to be pretty reasonable, being roughly linear in m. And while being quadratic in n sounds bad, one expects that $n \ll \sqrt{m}$ in practical regression problems.

Fin! If you've gotten this far without errors, your notebook is ready to submit.

Part 4: "Online" linear regression

When you are trying to fit a model to data and you get to see all of the data at once, we refer to the problem as an *offline* or *batch* problem, anc to use certain algorithms to compute the fit that can take advantage of the fact that you have a lot of available data.

But what if you only get to see one or a few data points at a time? In that case, you might want to get an initial model from whatever data you'v gradually improve the model as you see new data points. In this case, we refer to the problem as being an *online* problem.

The goal of this notebook is to introduce you to online algorithms. You'll start by reviewing the offline linear regression problem, and then look a variant. The neat thing about the online method is that you can derive it using all the tools you already have at your disposal, namely, multivaria

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Review: Offline or batch linear regression

Let's start with a quick review of the linear regression problem: given a response vector, y, and a data matrix X---whose rows are observations a are variables---the problem is to find the best linear model, $y \approx X\theta^*$, where θ^* is the vector of best-fit model parameters that we wish to compu Computing it using a conventional batch linear least squares method has an asymptotic running time of $\mathcal{O}(mn^2)$.

To start, here is some code to help generate synthetic problems of a certain size, namely, $m \times (n+1)$, where m is the number of observations an number of predictors. The $+1$ comes from our usual dummy coefficient for a non-zero intercept.

In [2]:
```
def generate_model (n):
    """Returns a set of (random) n+1 linear model coefficients."""
    return np.random.rand (n+1, 1)

def generate_data (m, theta, sigma=1.0/(2**0.5)):
    """
    Generates 'm' noisy observations for a linear model whose
    predictor (non-intercept) coefficients are given in 'theta'.
    Decrease 'sigma' to decrease the amount of noise.
    """
    assert (type (theta) is np.ndarray) and (theta.ndim == 2) and (theta.shape[1] == 1)
    n = len (theta)
    X = np.random.rand (m, n)
    X[:, 0] = 1.0
    y = X.dot (theta) + sigma*np.random.randn (m, 1)
    return (X, y)

def estimate_coeffs (X, y):
    """
    Solves X*theta = y by a linear least squares method.
    """
    result = np.linalg.lstsq (X, y, rcond=None)
    theta = result[0]
    return theta
```

In [3]:
```
def rel_diff(x, y, ord=2):
    """
    Computes ||x-y|| / ||y||. Uses 2-norm by default;
    override by setting 'ord'.
    """
    return np.linalg.norm (x - y, ord=ord) / np.linalg.norm (y, ord=ord)
```

An online algorithm

The empirical scaling of linear least squares appears to be pretty good, being roughly linear in m or at worst quadratic in n. But there is still a dc time and storage: each time there is a change in the data, you appear to need to form the data matrix all over again and recompute the solutior possibly touching the entire data set again!

This begs the question, is there a way to incrementally update the model coefficients whenever a new data point, or perhaps a small batch of n points, arrives? Such a procedure would be considered *incremental* or *online*, rather than batched or offline.

Setup: Key assumptions and main goal. In the discussion that follows, assume that you only get to see the observations *one-at-a-time*. Let (
the current observation. (Relative to our previous notation, this tuple is just element k of y and row k of X.

> We will use $\hat{x}_k^T$ to denote a row k of X since we previously used x_j to denote column j of X. That is,
>
> $$X = \begin{pmatrix} x_0 & \cdots & x_n \end{pmatrix} = \begin{pmatrix} \hat{x}_0^T \\ \vdots \\ \hat{x}_{m-1}^T \end{pmatrix},$$
>
> where the first form is our previous "columns-view" representation and the second form is our "rows-view."

Additionally, assume that, at the time the k-th observation arrives, you start with a current estimate of the parameters, $\tilde{\theta}(k)$, which is a vector. If
reason you need to refer to element i of that vector, use $\tilde{\theta}_i(k)$. You will then compute a new estimate, $\tilde{\theta}(k+1)$ using $\tilde{\theta}(k)$ and $(y_k, \hat{x}_k^T)$. For the disc
further assume that you throw out $\tilde{\theta}(k)$ once you have $\tilde{\theta}(k+1)$.

As for your goal, recall that in the batch setting you start with *all* the observations, (y, X). From this starting point, you may estimate the linear re
model's parameters, θ, by solving $X\theta = y$. In the online setting, you compute estimates one at a time. After seeing all m observations in X, your
compute an $\tilde{\theta}_{m-1} \approx \theta$.

An intuitive (but flawed) idea. Indeed, there is a technique from the signal processing literature that we can apply to the linear regression prob
the *least mean square (LMS) algorithm*. Before describing it, let's start with an initial idea.

Suppose that you have a current estimate of the parameters, $\theta(k)$, when you get a new sample, $(y_k, \hat{x}_k^T)$. The error in your prediction will be,

$$y_k - \hat{x}_k^T \tilde{\theta}(k).$$

Ideally, this error would be zero. So, let's ask if there exists a *correction*, Δ_k, such that

$$\begin{array}{rrcl} & y_k - \hat{x}_k^T \left(\tilde{\theta}(k) + \Delta_k\right) & = & 0 \\ \iff & y_k - \hat{x}_k^T \tilde{\theta}(k) & = & \hat{x}_k^T \Delta_k \end{array}$$

Then, you could compute a new estimate of the parameter by $\tilde{\theta}(k+1) = \tilde{\theta}(k) + \Delta_k$.

This idea has a major flaw, which we will discuss below. But before we do, please try the following exercise.

Mental exercise (no points). Verify that the following choice of Δ_k would make the preceding equation true.

$$\Delta_k = \frac{\hat{x}_k}{\|\hat{x}_k\|_2^2} \left(y_k - \hat{x}_k^T \tilde{\theta}(k)\right).$$

Refining (or rather, "hacking") the basic idea: The least mean square (LMS) procedure. The basic idea sketched above has at least one m
choice of Δ_k might allow you to correctly predict y_k from x_k and the new estimate $\tilde{\theta}(k+1) = \tilde{\theta}(k) + \Delta_k$, but there is no guarantee that this new est
preserves the quality of predictions made at all previous iterations!

There are a number of ways to deal with this problem, which includes carrying out an update with respect to some (or all) previous data. Howev
also a simpler "hack" that, though it might require some parameter tuning, can be made to work in practice.

That hack is as follows. Rather than using Δ_k as computed above, let's compute a different update that has a "fudge" factor, ϕ:

$$\begin{array}{rrcl} & \tilde{\theta}(k+1) & = & \tilde{\theta}(k) + \Delta_k \\ \text{where} & \Delta_k & = & \phi \cdot \hat{x}_k \left(y_k - \hat{x}_k^T \tilde{\theta}(k)\right). \end{array}$$

A big question is how to choose ϕ. There is some analysis out there that can help. We will just state the results of this analysis without proof.

Let $\lambda_{\max}(X^T X)$ be the largest eigenvalue of $X^T X$. The result is that as the number of samples $s \rightarrow \infty$, any choice of ϕ that satisfies the following co
eventually converge to the best least-squares estimator of $\tilde{\theta}$, that is, the estimate of $\tilde{\theta}$ you would have gotten by solving the linear least squares
all of the data.

$$0 < \phi < \frac{2}{\lambda_{\max}(X^T X)}.$$

This condition is not very satisfying, because you cannot really know $\lambda_{\max}(X^T X)$ until you've seen all the data, whereas we would like to apply th
online as the data arrive. Nevertheless, in practice you can imagine hybrid schemes that, given a batch of data points, use the QR fitting procec
starting estimate for $\tilde{\theta}$ as well as to estimate a value of ϕ to use for all future updates.

Summary of the LMS algorithm. To summarize, the algorithm is as follows:

- Choose any initial guess, $\tilde{\theta}(0)$, such as $\tilde{\theta}(0) \leftarrow 0$.
- For each observation $(y_k, \hat{x}_k^T)$, do the update:
  - $\tilde{\theta}(k+1) \leftarrow \tilde{\theta}_k + \Delta_k$,

  where $\Delta_k = \phi \cdot \hat{x}_k \left(y_k - \hat{x}_k^T \tilde{\theta}(k) \right)$.

Trying out the LMS idea

Now *you* should implement the LMS algorithm and see how it behaves.

To start, let's generate an initial 1-D problem (2 regression coefficients, a slope, and an intercept), and solve it using the batch procedure.

Recall that we need a value for ϕ, for which we have an upper-bound of $\lambda_{\max}(X^T X)$. Let's cheat by computing it explicitly, even though in practic need to do something different.

In [4]:
```
m = 100000
n = 1
theta_true = generate_model(n)
(X, y) = generate_data(m, theta_true, sigma=0.1)

print("Condition number of the data matrix:", np.linalg.cond(X))

theta = estimate_coeffs(X, y)
e_rel = rel_diff(theta, theta_true)

print("Relative error:", e_rel)
```

```
Condition number of the data matrix: 4.400944337141808
Relative error: 0.0016564742343831158
```

In [5]:
```
LAMBDA_MAX = max(np.linalg.eigvals(X.T.dot(X)))
print(LAMBDA_MAX)
```

```
126970.43769457837
```

Exercise 1 (5 points). Implement the online LMS algorithm in the code cell below where indicated. It should produce a final parameter estimate as a column vector.

In addition, the skeleton code below uses `rel_diff()` to record the relative difference between the estimate and the true vector, storing the k difference in `rel_diffs[k]`. Doing so will allow you to see the convergence behavior of the method.

Lastly, to help you out, we've defined a constant in terms of $\lambda_{\max}(X^T X)$ that you can use for ϕ.

> In practice, you would only maintain the current estimate, or maybe just a few recent estimates, rather than all of them. Since we want inspect these vectors later, go ahead and store them all.

In [6]:
```
PHI = 1.99 / LAMBDA_MAX # Fudge factor
rel_diffs = np.zeros((m+1, 1))

theta_k = np.zeros((n+1))
for k in range(m):
    rel_diffs[k] = rel_diff(theta_k, theta_true)

    # Implement the online LMS algorithm.
    # Take (y[k], X[k, :]) to be the k-th observation.
    ### BEGIN SOLUTION
    x_k = X[k, :]
    r_k = y[k] - x_k.T.dot(theta_k)
    delta_k = PHI * r_k * x_k
    theta_k = theta_k + delta_k
    ### END SOLUTION

theta_lms = theta_k
rel_diffs[m] = rel_diff(theta_lms, theta_true)
```

Let's compare the true coefficients against the estimates, both from the batch algorithm and the online algorithm. The values of the variables b change if the notebooks are re-run from start.

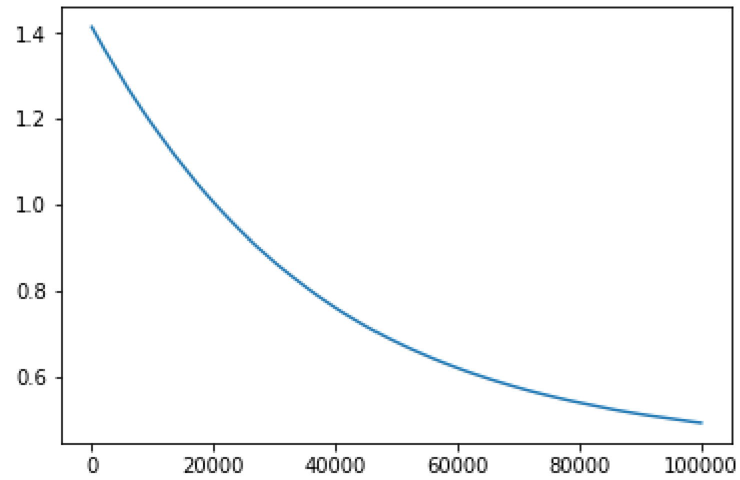
In [7]:
```
print (theta_true.T)
print (theta.T)
print (theta_lms.T)

print("\n('Passed' -- this cell appears to run without error, but we aren't checking the soluti
```

```
[[0.87103594 0.46153527]]
[[0.87172037 0.46005275]]
[0.74984566 0.40192802]

('Passed' -- this cell appears to run without error, but we aren't checking the solution.)
```

Let's also compute the relative differences between each estimate `Theta[:, k]` and the true coefficients `theta_true`, measured in the two-
the estimate is converging to the truth.

In [8]:
```
plt.plot(range(len(rel_diffs)), rel_diffs)
```

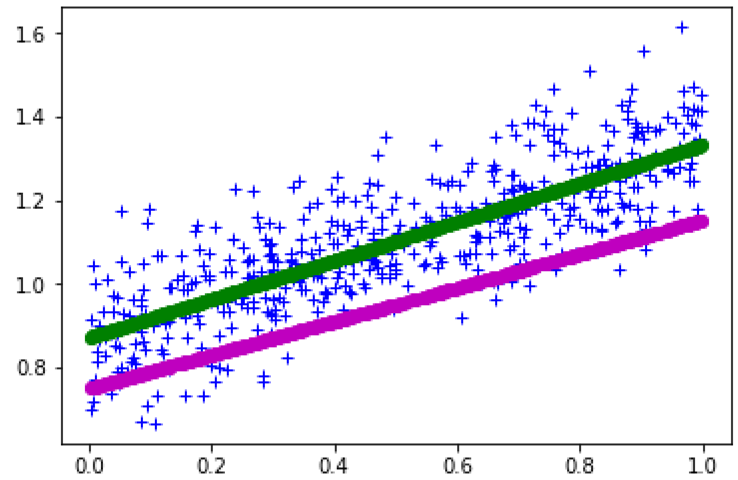
Out[8]: `[<matplotlib.lines.Line2D at 0x7ff8bd23eb00>]`



You should see it converging, but not especially quickly.

Finally, if the dimension is `n=1`, let's go ahead and do a sanity-check regression fit plot. The plot can change if the notebooks are re-run from st

In [9]:
```
STEP = int(X.shape[0] / 500)
if n == 1:
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(X[::STEP, 1], y[::STEP], 'b+') # blue - data
    ax1.plot(X[::STEP, 1], X.dot(theta_true)[::STEP], 'r*') # red - true
    ax1.plot(X[::STEP, 1], X.dot(theta)[::STEP], 'go') # green - batch
    ax1.plot(X[::STEP, 1], X.dot(theta_lms)[::STEP], 'mo') # magenta - pure LMS
else:
    print("Plot is multidimensional; I live in Flatland, so I don't do that.")
```

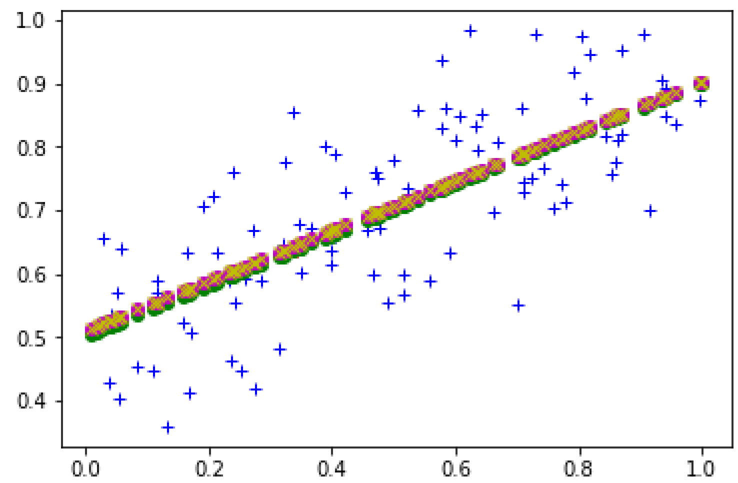


Exercise 2 (*ungraded*, optional). We said previously that, in practice, you would probably do some sort of *hybrid* scheme that mixes full batch u
(possibly only initially) and incremental updates. Implement such a scheme and describe what you observe. You might observe a different plot e
cell is re-run.

In [10]:
```
# Setup problem and compute the batch solution
m = 100000
n = 1
theta_true = generate_model(n)
(X, y) = generate_data(m, theta_true, sigma=0.1)
theta_batch = estimate_coeffs(X, y)

# Your turn, below: Implement a hybrid batch-LMS solution
# assuming you observe the first few data points all at
# once, and then see the remaining points one at a time.

### BEGIN SOLUTION
def lms(X, y, theta_0=None, PHI=1e-6):
    """Implements the LMS algorithm for the system X theta = y."""
    print(X.shape)
    (m, n) = X.shape

    if theta_0 is None:
        theta_k = np.zeros((n))
    else:
```

```
        theta_k = theta_0
    theta_k = np.reshape(theta_k, (n, 1))

    for k in range(m):
        x_k = X[k:k+1, :].T
        r_k = y[k] - x_k.T.dot(theta_k)
        delta_k = PHI * r_k * x_k
        theta_k = theta_k + delta_k
    return theta_k

# Pure LMS solution
START = 100
lambdas = np.linalg.eigvals(X[:START, :].T.dot(X[:START, :]))
lambda_max = max(lambdas)
lambda_min = min(lambdas)
phi = 2.0 / (lambda_max + lambda_min) # See: https://en.wikipedia.org/wiki/Least_mean_squares_f
theta_lms = lms(X, y, PHI=phi)

# Batch on first START observations + LMS on the rest:
theta_batch_START = estimate_coeffs(X[:START, :], y[:START])
print(theta_batch_START)
theta_hybrid = lms(X[START:, :], y[START:], theta_0=theta_batch_START, PHI=phi)
print(theta_hybrid)

def calc_ssqr(theta, X, y):
    r = X.dot(theta) - y
    return r.T.dot(r)

print("=== Sum of squared errors ===")
print("True:", calc_ssqr(theta_true, X, y))
print("Batch:", calc_ssqr(theta_batch, X, y))
print("Batch-{}:".format(START), calc_ssqr(theta_batch_START, X, y))
print("LMS(phi={}):".format(phi), calc_ssqr(theta_lms, X, y))
print("Batch (k <= %d) + LMS:" % START, calc_ssqr(theta_hybrid, X, y))

print("=== Relative error in the coefficients (theta) ===")
print("Batch:", rel_diff(theta_batch, theta_true))
print("Batch-{}:".format(START), rel_diff(theta_batch_START, theta_true))
print("LMS(phi={}):".format(phi), rel_diff(theta_lms, theta_true))
print("Batch (k <= %d) + LMS:" % START, rel_diff(theta_hybrid, theta_true))

STEP = int(X.shape[0] / 100)
if n == 1:
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(X[::STEP, 1], y[::STEP], 'b+') # blue - data
    ax1.plot(X[::STEP, 1], X.dot(theta_true)[::STEP], 'r*') # red - true
    ax1.plot(X[::STEP, 1], X.dot(theta_batch)[::STEP], 'go') # green - batch
    ax1.plot(X[::STEP, 1], X.dot(theta_lms)[::STEP], 'mo') # magenta - pure LMS
    ax1.plot(X[::STEP, 1], X.dot(theta_hybrid)[::STEP], 'yx') # yellow - hybrid
else:
    print ("Plot is multidimensional; I live in Flatland, so I don't do that.")

### END SOLUTION
```

```
(100000, 2)
[[0.54400102]
 [0.33553129]]
(99900, 2)
[[0.51075904]
 [0.39327854]]
=== Sum of squared errors ===
True: [[994.76733987]]
Batch: [[994.76638474]]
Batch-100: [[1040.65200804]]
LMS(phi=0.014605068287428212): [[999.28578087]]
Batch (k <= 100) + LMS: [[999.28578087]]
=== Relative error in the coefficients (theta) ===
Batch: 0.00025460403740161595
Batch-100: 0.12047813595445639
LMS(phi=0.014605068287428212): 0.018194600645439905
Batch (k <= 100) + LMS: 0.018194600645439905
```



< Previous

Next Up: Topic 13: Classification (Logistic Regression) >
30 min

edX

About
Affiliates
edX for Business
Open edX
Careers
News

Legal

Terms of Service & Honor Code
Privacy Policy
Accessibility Policy
Trademark Policy
Sitemap

Connect

Blog
Contact Us
Help Center
Media Kit
Donate

GET IT ON Google play
Download on the App Store