Homework

November 25, 2025

1 Unit 7 - Homework 10

MATH620

Joshua Dunne

1.1 Question 1

1.1.1 Find the singular value decomposition of the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

I'm going to borrow substantially from something I wrote for the first question here, we want the SVD, but, we want to show the steps.

[325]:
```python
import numpy as np

A = np.array([
    [1,1],
    [1,0],
    [0,1]
])

a_transpose_a = A.T @ A

# Hermitian, so, we can use eigh shortcut
a_transpose_a_eigen_values, a_transpose_a_eigen_vectors = np.linalg.
 ↪eigh(a_transpose_a)
sort_indices = np.argsort(a_transpose_a_eigen_values)[::-1]
# We want that these are ordered, and the corr. eigenvectors are likewise sorted
sorted_eigenvalues = a_transpose_a_eigen_values[sort_indices]
sorted_eigenvectors = a_transpose_a_eigen_vectors[:, sort_indices]
# From those we can compose Sigma
singular_values = np.sqrt(sorted_eigenvalues)
sigma = np.diag(singular_values)
# And compose our left singular
left_singular = (A @ sorted_eigenvectors) / singular_values
```

```
# And our right singular
right_singular = sorted_eigenvectors.T
# Lastly we check
print(f"left * sigma * right: \n{np.around(left_singular @ sigma @␣
 ↪right_singular)}")
print(f"left_singular: \n{np.around(left_singular)}")
print(f"sigma: \n{np.around(sigma)}")
print(f"right_singular: \n{np.around(right_singular)}")
```

```
left * sigma * right:
[[ 1.  1.]
 [ 1. -0.]
 [-0.  1.]]
left_singular:
[[ 1.  0.]
 [ 0. -1.]
 [ 0.  1.]]
sigma:
[[2. 0.]
 [0. 1.]]
right_singular:
[[ 1.  1.]
 [-1.  1.]]
```

That should surmise. The SVD is given by above. To summate, we find the eigenvectors of $\mathbf{A}^T\mathbf{A}$ to get $\mathbf{V}$, the eigenvectors of $\mathbf{A}\mathbf{A}^T$ to get $\mathbf{U}$, and the square roots of the eigenvalues of either to get . The only hangup here is getting the sorted order of the singular values, but that is easy enough to do. Lastly, we can just check against the built-in SVD function in numpy to verify our answer.

[326]:
```
print(f"left_singular: \n{np.around(np.linalg.svd(A)[0])}")
print(f"sigma: \n{np.around(np.diag(np.linalg.svd(A)[1]))}")
print(f"right_singular: \n{np.around(np.linalg.svd(A)[2])}")
```

```
left_singular:
[[-1.  0. -1.]
 [-0. -1.  1.]
 [-0.  1.  1.]]
sigma:
[[2. 0.]
 [0. 1.]]
right_singular:
[[-1. -1.]
 [-1.  1.]]
```

There's some funk going on there, but, I'm happy with the right_singular and sigma matrices. The left_singular matrix seems to have some sign differences, but, that's not a problem since eigenvectors are only defined up to a sign.

1.2 Question 2

1.2.1 Introduction

We're given a few things that are worth listing first here. Given points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

we can find a linear function $y = c_0 + c_1 x$ from

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

via least squares. This is done by solving the normal equations

$$\mathbf{A}^T \mathbf{A} \mathbf{c} = \mathbf{A}^T \mathbf{y}$$

where $\mathbf{A}$ is the matrix on the left hand side above, $\mathbf{c}$ is the vector of coefficients, and $\mathbf{y}$ is the vector of y values. If we substitue in the terms we can see the relationship more clearly:

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$

Using calculus we can do this another way.

$$||r(c)||^2 = ||y - Ac||^2 = [y_1 - (c_0 + c_1 x_1)]^2 + \cdots + [y_m - (c_0 + c_1 x_m)]^2 = f(c_0, c_1)$$

1.3 Question 2

1.3.1 Part a

We want to show that if we set the partial derivatives equal to 0 we get the same normal equations. We have two variables here we're actually interested in, c_0 and c_1. So we compute the partial derivatives with respect to each variable. I think we can just get by with throwing this in sympy, so, let's see.

[327]:
```python
from sympy import Matrix, symbols, diff, Sum, IndexedBase, Idx

# We have four things we need
c_0, c_1, x, y = symbols('c0 c1 x y')
# From a function we can define as such
error_term_squared = (y - (c_0 + c_1*x))**2
# If we take the derivative of each
d_c_0 = diff(error_term_squared, c_0)
d_c_1 = diff(error_term_squared, c_1)
print(f"c_0 derivative: {d_c_0}")
print(f"c_1 derivative: {d_c_1}")
```

```
c_0 derivative: 2*c0 + 2*c1*x - 2*y
c_1 derivative: -2*x*(-c0 - c1*x + y)
```

We can restate the results a little more cleanly here. This was given in terms of a sum of all the possible points given, so, the derivative is better written for c_0 as

$$\frac{\partial f}{\partial c_0} = -2\sum_{i=1}^{m}[y_i - (c_0 + c_1 x_i)]$$

and for c_1 as

$$\frac{\partial f}{\partial c_1} = -2\sum_{i=1}^{m}[y_i - (c_0 + c_1 x_i)]x_i$$

We can afford a quick aside and do this symbolically. We want to set the partials equal to zero, so, let's use sympy and see where we get.

[328]:
```python
import sympy as sp

critical_points_d_c_0 = sp.solve([d_c_0], [c_0])
critical_points_d_c_1 = sp.solve([d_c_1], [c_1])
print(f"c_0 partials against 0: {sp.latex(critical_points_d_c_0)}")
print(f"c_1 partials against 0: {sp.latex(critical_points_d_c_1)}")
```

```
c_0 partials against 0: \left\{ c_{0} : - c_{1} x + y\right\}
c_1 partials against 0: \left\{ c_{1} : \frac{- c_{0} + y}{x}\right\}
```

$$\left\{ c_{0} : - c_{1} x + y\right\}$$

$$\left\{ c_{1} : \frac{- c_{0} + y}{x}\right\}$$

We can write than in terms of c_0 and c_1 as

$$\sum_{i=1}^{m} y_i = \sum_{i=1}^{m}(c_0 + c_1 x_i)$$

and

$$\sum_{i=1}^{m} x_i y_i = \sum_{i=1}^{m}(c_0 + c_1 x_i)x_i$$

Rearranging these gives us

$$mc_0 + c_1\sum_{i=1}^{m} x_i = \sum_{i=1}^{m} y_i$$

and

$$c_0\sum_{i=1}^{m} x_i + c_1\sum_{i=1}^{m} x_i^2 = \sum_{i=1}^{m} x_i y_i$$

This is exactly the same as the normal equations we had before, so, we've shown what we wanted to show.

1.4 Question 2

1.4.1 Part b

We want to use a quadratic to fit $\{(-1,1),(0,-1),(1,0),(2,2)\}$, so, our system is going to look like

$$\begin{bmatrix}1 & x_1 & x_1^2\\1 & x_2 & x_2^2\\1 & x_3 & x_3^2\\1 & x_4 & x_4^2\end{bmatrix}\begin{bmatrix}c_0\\c_1\\c_2\end{bmatrix}=\begin{bmatrix}y_1\\y_2\\y_3\\y_4\end{bmatrix}$$

Substituting in our points gives us

$$\begin{bmatrix}1 & -1 & 1\\1 & 0 & 0\\1 & 1 & 1\\1 & 2 & 4\end{bmatrix}\begin{bmatrix}c_0\\c_1\\c_2\end{bmatrix}=\begin{bmatrix}1\\-1\\0\\2\end{bmatrix}$$

Where we are trying to find coefficients c_0, c_1, c_2 to fit the quadratic $y = c_0 + c_1x + c_2x^2$. We can use sympy for the lulz.

[329]:
```python
import sympy as sp

from sympy import symbols, Matrix, latex

x1, x2, x3, x4 = symbols('x_1 x_2 x_3 x_4')
y1, y2, y3, y4 = symbols('y_1 y_2 y_3 y_4')

A = Matrix([
    [1, x1, x1**2],
    [1, x2, x2**2],
    [1, x3, x3**2],
    [1, x4, x4**2]
])



y_vec = Matrix([y1, y2, y3, y4])

ATA = A.T * A
ATy = A.T * y_vec
substitutions = {
    x1: -1, x2: 0, x3: 1, x4: 2,
    y1: 1, y2: -1, y3: 0, y4: 2
}
print("--- A Transpose A (Left side of Normal Eq) ---")
print(f"ATA: {latex(ATA.subs(substitutions))}")
print(f"ATy: {latex(ATy.subs(substitutions))}")
```

```
--- A Transpose A (Left side of Normal Eq) ---
ATA: \left[\begin{matrix}4 & 2 & 6\\2 & 6 & 8\\6 & 8 & 18\end{matrix}\right]
ATy: \left[\begin{matrix}2\\3\\9\end{matrix}\right]
```

So, we've taken $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{y}$. From this we get

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix}2\\3\\9\end{bmatrix}$$

and

$$\mathbf{A}^T\mathbf{y} = \begin{bmatrix}4 & 2 & 6\\2 & 6 & 8\\6 & 8 & 18\end{bmatrix}$$

```
[330]: # We can then solve for c_0, c_1, c_2
       ata_substituted = ATA.subs(substitutions)
       aty_substitued = ATy.subs(substitutions)
       print(latex(ata_substituted.solve(aty_substitued)))
```

```
\left[\begin{matrix}- \frac{7}{10}\\- \frac{3}{5}\\1\end{matrix}\right]
```

Which gives us

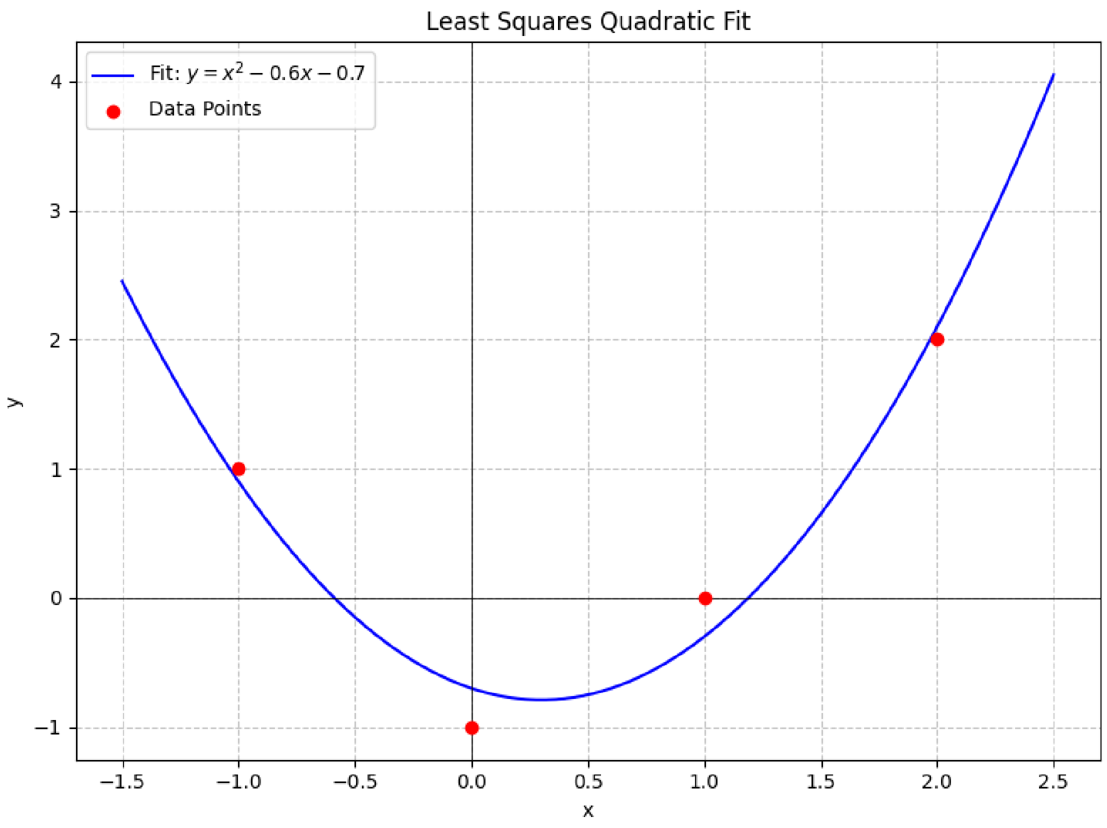
$$\begin{bmatrix}c_0\\c_1\\c_2\end{bmatrix} = \begin{bmatrix}-\frac{7}{10}\\-\frac{3}{5}\\1\end{bmatrix}$$

And putting that back in to our original equation gives us the quadratic fit of

$$y = -\frac{7}{10} - \frac{3}{5}x + x^2$$

We can take a moment and plot this

```
[331]: import numpy as np
       import matplotlib.pyplot as plt

       # Points: (-1, 1), (0, -1), (1, 0), (2, 2)
       x_points = np.array([-1, 0, 1, 2])
       y_points = np.array([1, -1, 0, 2])

       # Coefficients: c0 = -0.7, c1 = -0.6, c2 = 1
       def quadratic_model(x):
           return -0.7 - 0.6 * x + 1.0 * x**2

       x_line = np.linspace(-1.5, 2.5, 100)
       y_line = quadratic_model(x_line)

       plt.figure(figsize=(8, 6))
       plt.plot(x_line, y_line, label=r'Fit: $y = x^2 - 0.6x - 0.7$', color='blue')
       plt.scatter(x_points, y_points, color='red', zorder=5, label='Data Points')

       plt.title('Least Squares Quadratic Fit')
       plt.xlabel('x')
       plt.ylabel('y')
       plt.axhline(0, color='black', linewidth=0.5) # x-axis
       plt.axvline(0, color='black', linewidth=0.5) # y-axis
       plt.grid(True, linestyle='--', alpha=0.7)
       plt.legend()

       plt.tight_layout()
       plt.show()
```



Pure gravey. I'm liking the fit.

1.5 Question 3

1.5.1 Part A

Exponential growth by

$$p = 2.56e^{kt} \rightarrow ln(p) = ln(2.56) + ln(e^{kt}) \rightarrow lp(p) = ln(2.56) + kt$$

Year	Population	k	t	ln(p)
1950	2.56 billion	$k = 0$	0	$ln(2.56) = 0.939$

Year	Population	k	t	ln(p)
1960	3.04 billion	$k = 0.172$	$3.04 = 2.56e^{1k}$	$ln(\frac{3.04}{2.56}) = k, k = 0.17185$
1970	3.71 billion	$2k = 0.317$	$3.71 = 2.56e^{2k}$	$ln(\frac{3.71}{2.56}) = 2k, k = 0.185512$
1980	4.46 billion	$3k = 0.555$	$4.46 = 2.56e^{3k}$	$ln(\frac{4.46}{2.56}) = 3k, k = 0.185047$
1990	5.28 billion	$4k = 0.724$	$5.28 = 2.56e^{4k}$	$ln(\frac{5.28}{2.56}) = 4k, k = 0.18098$
2000	6.08 billion	$5k = 0.865$	$6.08 = 2.56e^{5k}$	$ln(\frac{6.08}{2.56}) = 5k, k = 0.173$

If we write this as a matrix equation we'd be looking for

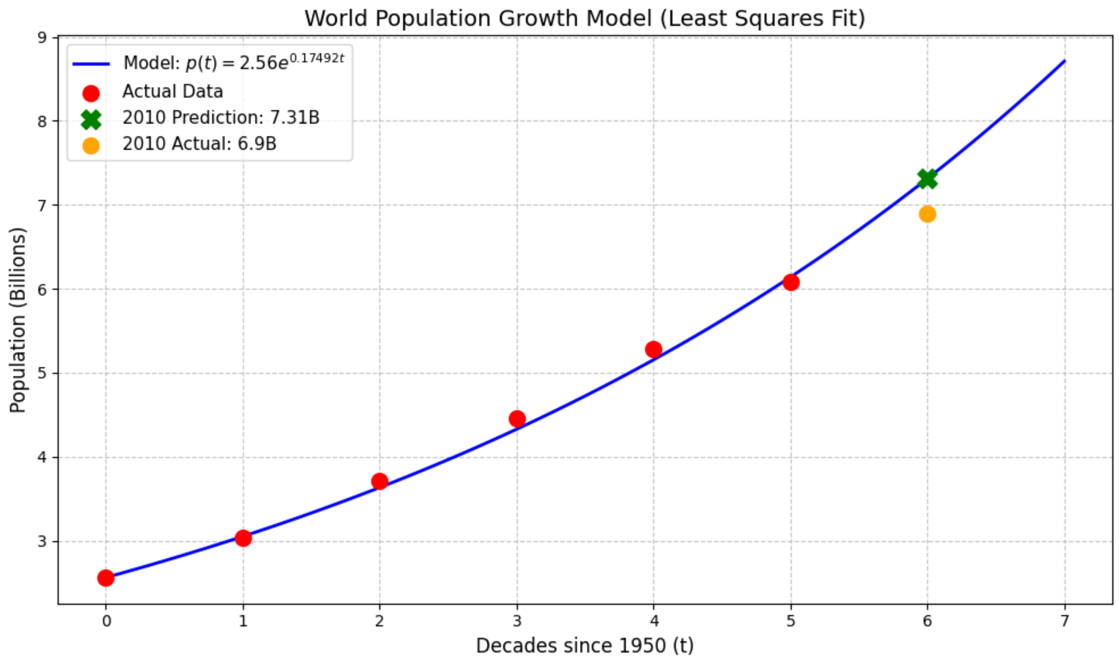
$$\begin{bmatrix} 1k \\ 2k \\ 3k \\ 4k \\ 5k \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} k = \begin{bmatrix} \ln\left(\frac{3.04}{2.56}\right) \\ \ln\left(\frac{3.71}{2.56}\right) \\ \ln\left(\frac{4.46}{2.56}\right) \\ \ln\left(\frac{5.28}{2.56}\right) \\ \ln\left(\frac{6.08}{2.56}\right) \end{bmatrix}$$

[332]:
```python
from sympy import Matrix, log, symbols, exp, N

A = Matrix([1, 2, 3, 4, 5])

b = Matrix([
    log(3.04 / 2.56),
    log(3.71 / 2.56),
    log(4.46 / 2.56),
    log(5.28 / 2.56),
    log(6.08 / 2.56)
])

ATA = A.T * A
ATb = A.T * b
print(f"ATA: {ATA}")
print(f"ATb: {ATb.evalf(4)}")

k_matrix = ATA.solve(ATb)
k_value = k_matrix[0]

t = symbols('t')
model = 2.56 * exp(k_value * t)
```

```
p_2010 = model.subs(t, 6)
print(f"model: {latex(model)}")
print(f"\nprediction for 2010 (t=6): {p_2010.evalf(3)} billion")
```

```
ATA: Matrix([[55]])
ATb: Matrix([[9.800]])
model: 2.56 e^{0.178181573820795 t}

prediction for 2010 (t=6): 7.46 billion
```

[333]:
```
import numpy as np
import matplotlib.pyplot as plt

t_data = np.array([0, 1, 2, 3, 4, 5])
p_data = np.array([2.56, 3.04, 3.71, 4.46, 5.28, 6.08])

k_optimal = 0.17492
p0 = 2.56

def model_func(t):
    return p0 * np.exp(k_optimal * t)

t_smooth = np.linspace(0, 7, 100)
p_smooth = model_func(t_smooth)

t_2010 = 6
p_2010_pred = model_func(t_2010)
p_2010_actual = 6.9

plt.figure(figsize=(10, 6))

plt.plot(t_smooth, p_smooth, label=f'Model: $p(t) = 2.56e^{{{k_optimal}t}}$',␣
 ↪color='blue', linewidth=2)

plt.scatter(t_data, p_data, color='red', s=100, zorder=5, label='Actual Data')

plt.scatter(t_2010, p_2010_pred, color='green', marker='X', s=150, zorder=5,␣
 ↪label=f'2010 Prediction: {p_2010_pred:.2f}B')
plt.scatter(t_2010, p_2010_actual, color='orange', marker='o', s=100, zorder=5,␣
 ↪label=f'2010 Actual: {p_2010_actual}B')

plt.title('World Population Growth Model (Least Squares Fit)', fontsize=14)
plt.xlabel('Decades since 1950 (t)', fontsize=12)
plt.ylabel('Population (Billions)', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=11)
plt.tight_layout()
plt.show()
```



1.6 Question 3

1.6.1 Part B

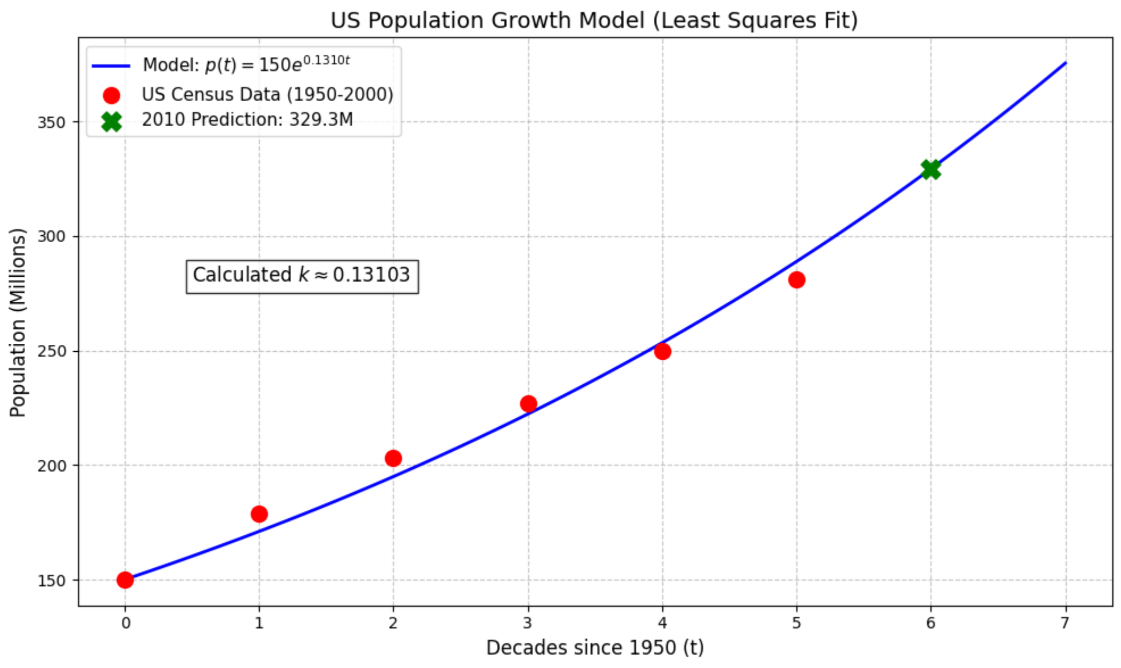
We're given another table for the growth of the US population from 1950 to 2000. Just doing the same here. Ignore the mess, for whatever reason jupyter doesn't want to render this table properly. Below are just included the values of the initial conditions we're using. | Year | Population | k (approx) | Exponential Model | Log Calculation | | :— | :— | :— | :— | :— | | 1950 | 150 million | $k = 0$ | 0 | $\ln(1) = 0$ | | 1960 | 179 million | $k = 0.172$ | $179 = 150e^{1k}$ | $\ln(\frac{179}{150}) = k,\ k \approx 0.177$ | | 1970 | 203 million | $2k = 0.303$ | $203 = 150e^{2k}$ | $\ln(\frac{203}{150}) = 2k,\ k \approx 0.151$ | | 1980 | 227 million | $3k = 0.414$ | $227 = 150e^{3k}$ | $\ln(\frac{227}{150}) = 3k,\ k \approx 0.138$ | | 1990 | 250 million | $4k = 0.511$ | $250 = 150e^{4k}$ | $\ln(\frac{250}{150}) = 4k,\ k \approx 0.128$ | | 2000 | 281 million | $5k = 0.628$ | $281 = 150e^{5k}$ | $\ln(\frac{281}{150}) = 5k,\ k \approx 0.126$ |

[334]:
```
from sympy import Matrix, log, symbols, exp, N

A = Matrix([1, 2, 3, 4, 5])

b = Matrix([
    log(179/150),
    log(203/150),
    log(227/150),
    log(250/150),
    log(281/150)
])

ATA = A.T * A
ATb = A.T * b
print(f"ATA: {ATA}")
print(f"ATb: {ATb.evalf(4)}")

k_matrix = ATA.solve(ATb)
k_value = k_matrix[0]

t = symbols('t')
model = 150 * exp(k_value * t)

p_2010 = model.subs(t, 6)
print(f"model: {latex(model)}")
print(f"\nprediction for 2010 (t=6): {p_2010.evalf(3)} million")
```

```
ATA: Matrix([[55]])
ATb: Matrix([[7.207]])
model: 150 e^{0.131031553146226 t}

prediction for 2010 (t=6): 329 million
```

[335]:
```
import numpy as np
import matplotlib.pyplot as plt

t_data = np.array([0, 1, 2, 3, 4, 5])
p_data = np.array([150, 179, 203, 227, 250, 281])
p0 = 150

A = np.array([1, 2, 3, 4, 5])
b = np.log(p_data[1:] / p0)

ATA = np.dot(A, A)
ATb = np.dot(A, b)

k_optimal = ATb / ATA

def model_func(t):
    return p0 * np.exp(k_optimal * t)


t_smooth = np.linspace(0, 7, 100)
p_smooth = model_func(t_smooth)

t_2010 = 6
```

```
p_2010_pred = model_func(t_2010)


plt.figure(figsize=(10, 6))

plt.plot(t_smooth, p_smooth, label=f'Model: $p(t) = 150e^{{{k_optimal:.
 ↪4f}t}}$', color='blue', linewidth=2)
plt.scatter(t_data, p_data, color='red', s=100, zorder=5, label='US Census Data␣
 ↪(1950-2000)')
plt.scatter(t_2010, p_2010_pred, color='green', marker='X', s=150, zorder=5,␣
 ↪label=f'2010 Prediction: {p_2010_pred:.1f}M')

plt.title('US Population Growth Model (Least Squares Fit)', fontsize=14)
plt.xlabel('Decades since 1950 (t)', fontsize=12)
plt.ylabel('Population (Millions)', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=11)
plt.text(0.5, 280, f'Calculated $k \\approx {k_optimal:.5f}$', fontsize=12,␣
 ↪bbox=dict(facecolor='white', alpha=0.8))

plt.tight_layout()
plt.show()
```



1.7 Question 4

1.7.1 Statement

We're being given a matrix

$$\mathbf{A} = \begin{bmatrix} 3 & -1 \\ 2 & 4 \end{bmatrix}$$

and we want the singular values from this.

```
[336]: import numpy as np

A = np.array([
    [3, -1],
    [2, 4]
])

ATA = A.T @ A
eigenvalues, eigenvectors = np.linalg.eigh(ATA)
ordering = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[ordering]
eigenvectors = eigenvectors[:, ordering]
print(latex(np.sqrt(eigenvalues)))
```

```
\mathtt{\text{[4.51499333 3.10077977]}}
```

$$_n = [4.51499333\ 3.10077977]$$

1.8 Question 4

1.8.1 Part A

This would mean that we could calculate the Frobenius norm as

$$||\mathbf{A}||_F = \sqrt{(4.51499333)^2 + (3.10077977)^2} = 5.464985704219043$$

Similarly, we can calculate the Frobenius norm directly from the matrix as

$$||\mathbf{A}||_F = \sqrt{3^2 + (-1)^2 + 2^2 + 4^2} = \sqrt{9 + 1 + 4 + 16} = \sqrt{30} = 5.477225575051661$$

1.9 Question 4

1.9.1 Part B

We know want the l-2 norm, $||\mathbf{A}||_2 = \max(\sigma)$.

$$||\mathbf{A}||_2 = \max(\ ) = \sigma_1 = 4.51499333$$