# CS111 F21 Homework 6

Michael Glushchenko, 9403890
Partners with: Shiv Kapoor

November 5, 2021

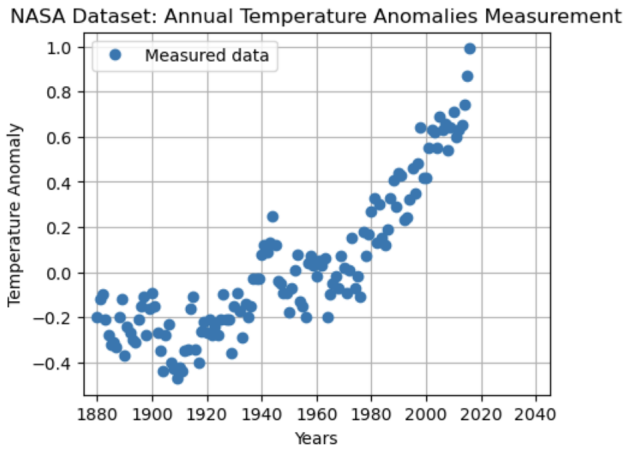
## Least Square's Fitting & Floating Point

# 1 Problem 1

## 1.1

We used the following code to produce a plot of the temperature abnormalities:

```
filename = "/Users/Mike/desktop/UCSB/CS/CS111/cs111-2021-fall/Homework/h06/annual_temps.json"
years, temperatures = cs111.get_gistemp(filename)

plt.figure( dpi=100, figsize=(5,4) ) # plot size
plt.plot( years, temperatures, "o", label="Measured data" )
plt.xlabel( "Years" )
plt.ylabel( "Temperature Anomaly" )
plt.title( "NASA Dataset: Annual Temperature Anomalies Measurement" )
plt.grid() # grid lines behind the plot
plt.xlim([1875, 2045]) # shows until year 2040
plt.legend() # displays legend
plt.show() # shows the plot
```



## 1.2

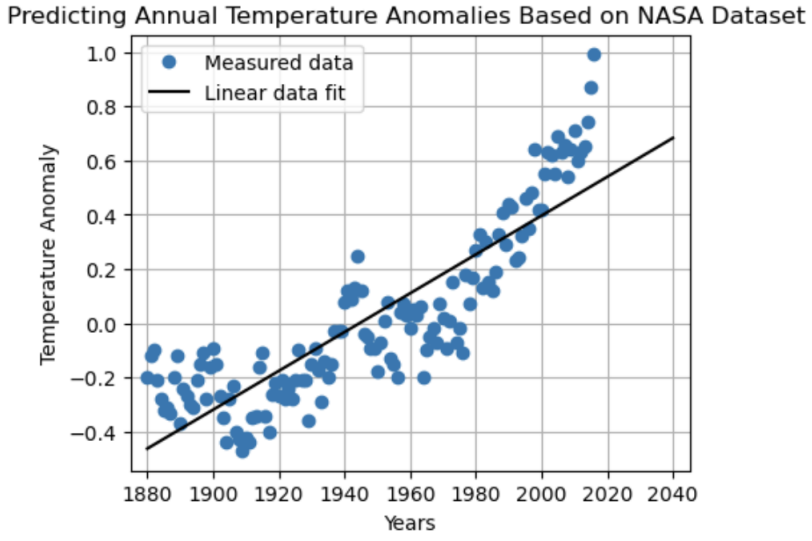
Here's the code used to set up the data for this question, as well as to get the estimate for year 2040, and the calculated coefficients for the regression line:

```
# x=[x0, x1], we are using the linear equation
# want the line temperature=x0+x1*years for years 1880-2040
A = np.ones((len(years),2))
A[:,-1] = years
x, res, rank, s = npla.lstsq(A, temperatures, rcond=None)
years1 = np.linspace(1880, 2040, len( range(1880, 2041) ))
AA = np.ones((len(years1),2)) # create the matrix for this range of years
AA[:,-1] = years1
y1 = AA @ x # gives us the temperature estimates
print( "x based off the linear model:\n", x, sep="" )
print( "year 2040 estimate:\n", y1[-1], sep="" )
```

```
x based off the linear model:
[-1.3912e+01  7.1544e-03]
year 2040 estimate:
0.6825848003434967
```

Then, using the code below, we produce the following plot:

```
# same plotting procedure as before, but we also add the y = 2x+1 line
plt.figure( dpi=100, figsize=(5,4) )
plt.plot( years, temperatures, "o", label="Measured data" )
plt.plot(years1, y1, 'black', label='Linear data fit')
plt.xlabel( "Years" )
plt.ylabel( "Temperature Anomaly" )
plt.title( "Predicting Annual Temperature Anomalies Based on NASA Dataset " )
plt.grid()
plt.xlim([1875, 2045])
plt.legend()
plt.show()
```



## 1.3

We use the same idea as before, but we modify the data arrays to only contain the years required. Here's the estimates we get for the $x$ coefficients, as well as the 2040 estimates for the 1970-2016 and the 2010-2016 time ranges:

```
# for the second graph we modify the years and temp arrays
start_year = 1970
years = np.copy(save_years[start_year-1880:])
temperatures = np.copy(save_temps[start_year-1880:])
# now we can proceed to run the exact same code to obtain the second
# line of estimates
A = np.ones((len(years),2))
A[:,-1] = years
x1, res, rank, s = npla.lstsq(A, temperatures, rcond=None)

years2 = np.linspace(start_year, 2040, len( range(start_year, 2041) ))
AA = np.ones((len(years2),2)) # create the matrix for this range of years
AA[:,-1] = years2
y1 = AA @ x1 # gives us the temperature estimates

print( "\n1970-2016 data produces the following x:\n", x1, sep="" )
print( "\nyear 2040 estimate:\n", y1[-1], sep="" )
```

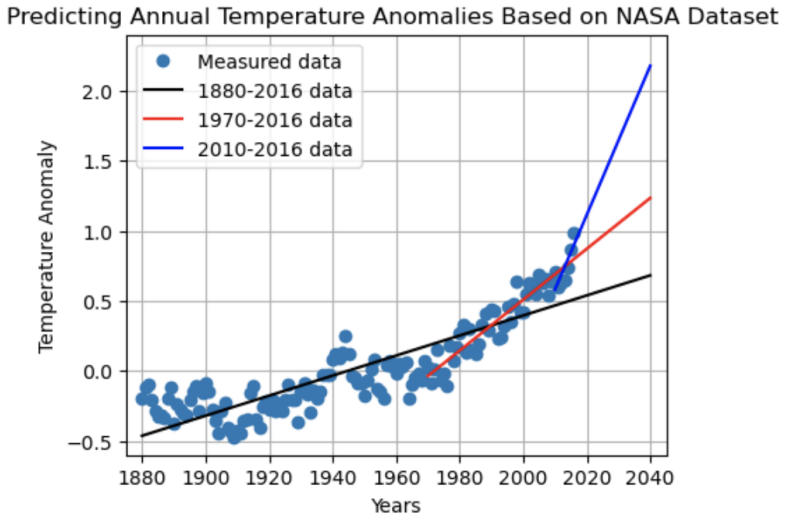
```
1970-2016 data produces the following x:
[-3.5788e+01  1.8149e-02]

year 2040 estimate:
1.2351167900092506
```

```
# for the third graph we modify the years and temp arrays again
start_year = 2010
years = np.copy(save_years[start_year-1880:])
temperatures = np.copy(save_temps[start_year-1880:])
# now we can proceed to run the exact same code to obtain the second
# line of estimates
A = np.ones((len(years),2))
A[:,-1] = years
x2, res, rank, s = npla.lstsq(A, temperatures, rcond=None)

years3 = np.linspace(start_year, 2040, len( range(start_year, 2041) ))
AA = np.ones((len(years3),2)) # create the matrix for this range of years
AA[:,-1] = years3
y2 = AA @ x2 # gives us the temperature estimates

print( "\n2010-2016 data produces the following x:\n", x2, sep="" )
print( "\nyear 2040 estimate:\n", y2[-1], sep="" )
```

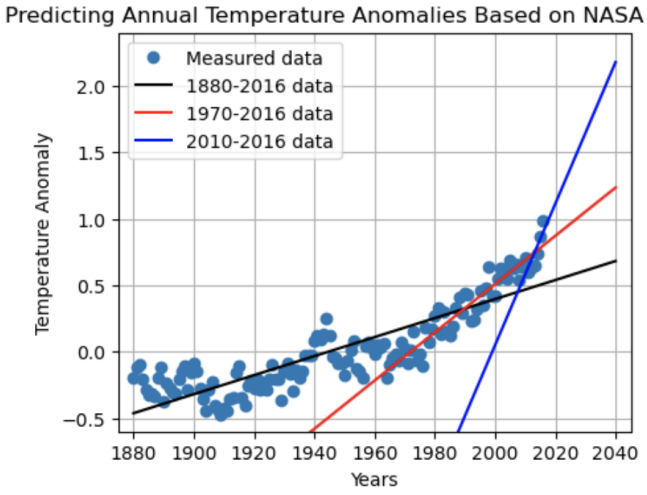
```
2010-2016 data produces the following x:
[-1.0638e+02  5.3214e-02]

year 2040 estimate:
2.178214285713949
```

Now, using the $y$, $y_1$, and $y_2$ result vectors, we plot our 3 fitted lines:

```
# same plotting procedure as before, but we plot all 3 lines
plt.figure( dpi=100, figsize=(5,4) )
plt.plot( save_years, save_temps, "o", label="Measured data" )
plt.plot(years1, y, 'black', label='1880-2016 data')
plt.plot(years2, y1, 'red', label='1970-2016 data')
plt.plot(years3, y2, 'blue', label='2010-2016 data')
plt.xlabel( "Years" )
plt.ylabel( "Temperature Anomaly" )
plt.title( "Predicting Annual Temperature Anomalies Based on NASA Dataset " )
plt.grid()
plt.xlim([1875, 2045])
plt.ylim([-0.6, 2.4])
plt.legend()
plt.show()
```




If you wanted us to extend the least-squares-fit lines all the way, here's that plot:



## 1.4

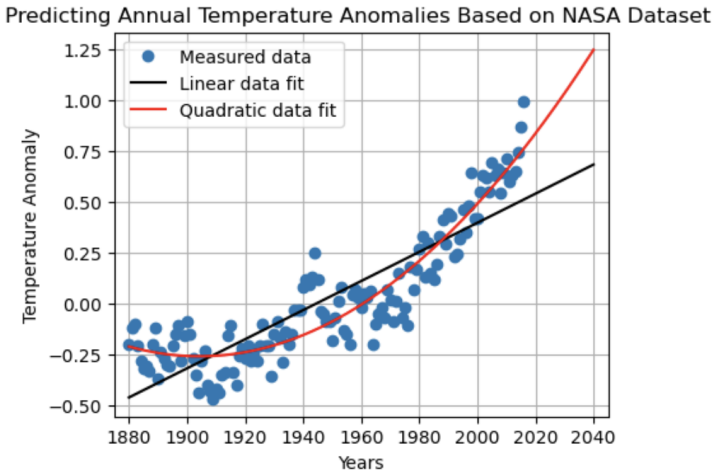
Since we now want a quadratic fit, rather than a linear fit, we add a $years^2$ variable, and basically do the exact same thing again:

```
# x = [x0, x1, x2], we are using the quadratic equation
# want the line temperature=x0+x1*years+x2*years^2 for years 1880-2040

A = np.ones((len(save_years),3))
A[:,-2] = save_years
A[:,-1] = save_years**2
x_quad, res, rank, s = npla.lstsq(A, save_temps, rcond=None)

# second matrix goes until year 2040
AA = np.ones((len(years1),3))
AA[:,-2] = years1
AA[:,-1] = years1**2
y_quad = AA @ x_quad

print( "\nx based off the quadratic model:\n", x_quad, sep="" )
print( "\nyear 2040 estimate:\n", y_quad[-1], sep="" )
```

```
x based off the quadratic model:
[ 2.9612e+02 -3.1128e-01  8.1734e-05]

year 2040 estimate:
1.2465514884599997
```

Running the plot code again, we obtain the following:

```
# same plot procedure, add the quadratic line, omit 1.3 lines
plt.figure( dpi=100, figsize=(5,4) )
plt.plot( save_years, save_temps, "o", label="Measured data" )
plt.plot( years1, y, 'black', label='Linear data fit' )
plt.plot( years1, y_quad, 'red', label='Quadratic data fit' )
plt.xlabel( "Years" )
plt.ylabel( "Temperature Anomaly" )
plt.title( "Predicting Annual Temperature Anomalies Based on NASA Dataset " )
plt.grid()
plt.xlim([1875, 2045])
plt.legend()
plt.show()
```



## 2 Problem 2

Here's the code I used for this part:

```
A = np.array( [[1,0], [0,1000]] )
cs111.print_float64(npla.cond(A))
print("condition number of A is", npla.cond(A))
print("Is k(A)=1000?", npla.cond(A)==1000.0)
print("\n\n")
B = np.array( [[1,0], [1,0]] )
cs111.print_float64(npla.cond(B))
print("condition number of B is", npla.cond(B))
print("is the condition number of B infinite?", npla.cond(B)==np.inf)
```

```
input     : 1000.0
as float64: 1.0000000000000000e+03
as hex    : 408f400000000000
sign      : 0 means +
exponent  : 408 means 1032 - 1023 = 9
mantissa  : 1.1111010000000000000000000000000000000000000000000000

condition number of A is 1000.0
Is k(A)=1000? True



input     : inf
as float64: inf
as hex    : 7ff0000000000000
sign      : 0 means +
exponent  : 7ff means inf or nan

condition number of B is inf
is the condition number of B infinite? True
```

---

The condition number of $A$ is $1e3$ in ordinary base 10, written as $0x408f400000000000$ in hexadecimal. The condition number of $B$ is $\infty$, as shown above, written as $0x7ff0000000000000$ in IEEE 64-bit hexadecimal.

# 3 Problem 3

## 3.1 Machine $\epsilon$

The exact value of machine $\epsilon$ is $2^{-52}$, because IEEE 64-bit floating point standard allocates 52 bits for the frac part of the floating point.

```
machine_epsilon = 2**(-52) # this is the arithmetic expression
print(1 + machine_epsilon / 2 > 1) # quick check

cs111.print_float64(machine_epsilon) # get the 16-bit hex representation
print("How close is machine epsilon to 10^-16?\n", np.absolute(machine_epsilon - 10e-16)/machine_epsilon)
```

```
False
input     : 2.220446049250313e-16
as float64: 2.220446049250313e-16
as hex    : 3cb0000000000000
sign      : 0 means +
exponent  : 3cb means 971 - 1023 = -52
mantissa  : 1.0000000000000000000000000000000000000000000000000000

How close is machine epsilon to 10^-16?
 3.5035996273704963
```

The code above is used to produce the answers for the rest of the questions to **3.1**.

## 3.2 $\frac{1}{\epsilon}$

```
inv_machine_epsilon = 1 / machine_epsilon
cs111.print_float64(inv_machine_epsilon)
```

```
input     : 4503599627370496.0
as float64: 4.5035996273704960e+15
as hex    : 4330000000000000
sign      : 0 means +
exponent  : 433 means 1075 - 1023 = 52
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

Following from **3.1**, we can see that the approximate value of $\frac{1}{\epsilon}$ is approximately $4.5036e15$, exactly $2^{52}$, with its hex representation being $0x4330000000000000$.

## 3.3 Largest Positive Floating Point$< \infty$

The function below uses uses the number of bits allocated for the exponent to calculate the initial exponent to which we raise our num_of_bits variable to get a number that's smaller than the infinity representation of the system by a factor of whatever base the system is in (I know, num of bits is a poor variable choice, since bits implies binary, but I made it so that the equation worked for any base system). Then, we multiply it by a number "very close" to num_of_bits, which we get by $num\_of\_bits -$ machine epsilon. Here's the expression the code uses for the 64−bit IEEE system:

$$2^{2^{10}-1} * (2 - \epsilon_{64})) \approx 1.7976931348623157e + 308.$$

Above, $\epsilon_{\text{IEEE64}}$ represents the machine epsilon for the 64-bit IEEE standard. Here's the function and output, showing the necessary results:

```
def return_largest_num(num_of_bits, exponent_bits, frac_bits):
    machine_epsilon = num_of_bits**(-1.0 * frac_bits)
    x = (num_of_bits ** (num_of_bits**(exponent_bits - 1) - 1)) * (num_of_bits - machine_epsilon)
    print("Machine epsilon for this system:", machine_epsilon, "\n")
    cs111.print_float64(x)
    return x

x = return_largest_num(2.0, 11.0, 52.0) #IEEE 64-bit standard
print(x * 2.0 == np.inf)
```

```
Machine epsilon for this system: 2.220446049250313e-16

input     : 1.7976931348623157e+308
as float64: 1.7976931348623157e+308
as hex    : 7fefffffffffffff
sign      : 0 means +
exponent  : 7fe means 2046 - 1023 = 1023
mantissa  : 1.1111111111111111111111111111111111111111111111111111

True
```

## 3.4 $\epsilon^{10}$

```
# consider the following:
x = machine_epsilon**10
cs111.print_float64(x) # yes, this should be an exact representation in IEEE
```

```
input     : 2.9134143481250081e-157
as float64: 2.9134143481250808e-157
as hex    : 1f70000000000000
sign      : 0 means +
exponent  : 1f7 means 503 - 1023 = -520
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

The approximate base-10 value of $\epsilon^{10}$ is $2.9134e-157$. Such a number can, indeed, be represented as an exact floating-point, since $(2^{-52})^{10} = 2^{-520}$. That's exactly what we get when running cs111.print_float64 on this number. It's hex representation is $0x1f70000000000000$.

## 3.5 How many numbers?

Since we have 52 bits allocated for the frac, and there is 2 different values that we can choose for each bit, the number of values in each interval given is $2^{52} - 1$ (since we want to exclude the actual bounds from the count). We can also observe that the number of floating points in the interval $[2^0, 2^1)$ is the same as the interval $[2^{12}, 2^{13})$, which is the same as the number of floating points in the interval $[2^{-6}, 2^{-5})$, and that this number can be obtained by convertin each bound to a 16-bit hex, and simply substracting their hex values. This is because the lower and upper bounds of each interval differ by a factor of 2, which is when the frac runs out of precision for a given exp value, and the exp value is incremented, and the frac thrown back to 0. The below code confirms what I just said, maybe explains it a little better than I do:

```
def f(lower, upper):
    num_of_points = int(cs111.double_to_hex(np.float64(upper)),16) - int(cs111.double_to_hex(np.float64(lower)),16)
    print(num_of_points)
    print(math.log(num_of_points,2))
print("num of floating points between 1 and 2: ")
f(1,2)
print("num of floating points between 4096 and 8192: ")
f(4096,8192)
print("num of floating points between 1/64 and 1/32: ")
f(1/64,1/32)
```

```
num of floating points between 1 and 2:
4503599627370496
52.0
num of floating points between 4096 and 8192:
4503599627370496
52.0
num of floating points between 1/64 and 1/32:
4503599627370496
52.0
```

The code, however, includes the lower bound in its count, and the question asks us to exclude the bound, so the answer for all of them is $2^{52} - 1$.

# 4 Problem 4

## 4.1 bfloat & 16-bit IEEE Standard

Following similar logic to **3.5**, the 16-bit IEEE standard has 10 bits allocated for the frac portion. Hence, the number of floating points between 1 and 2, exclusively, is $2^{10} - 1$. bfloat allocates 7 bits for the frac, and thus the same range can represent a total of $2^7 - 1$ floating points.

## 4.2 Machine $\epsilon$

Once again, same logic here as **3.1**, it's 2 raised to the power of ($-1$ times number of bits allocated for frac):

$$\epsilon_{\text{IEEE16}} = 2^{-10},$$

and

$$\epsilon_{\text{bfloat}} = 2^{-7}.$$

We can use float64 to see what these numbers are in base 10. The code below shows that:

```
# float 16 machine epsilon:
cs111.print_float64(2.0**(-10.0))

# bfloat machine epsilon:
cs111.print_float64(2.0**(-7.0))
```

```
input     : 0.0009765625
as float64: 9.7656250000000000e-04
as hex    : 3f50000000000000
sign      : 0 means +
exponent  : 3f5 means 1013 - 1023 = -10
mantissa  : 1.0000000000000000000000000000000000000000000000000000

input     : 0.0078125
as float64: 7.8125000000000000e-03
as hex    : 3f80000000000000
sign      : 0 means +
exponent  : 3f8 means 1016 - 1023 = -7
mantissa  : 1.0000000000000000000000000000000000000000000000000000
```

## 4.3 Largest Non-Infinite Positive Numbers

Using the logic from **3.3**: for 16-bit IEEE, that would mean that

$$2^{2^4-1} * (2 - \epsilon_{\text{IEEE16}}) = 65504.0$$

is the largest number we can represent. The code below exemplifies it, as well as shows the base-10 representation of the number. As for bfloat, the expression would be

$$2^{2^4-1} * (2 - \epsilon_{\text{bfloat}}) \approx 3.3895313892515355e + 38.$$

We use the same function we used in **3.3**, now inputting different exponent and frac bits for these two new systems. Here's what we get:

```
def return_largest_num(num_of_bits, exponent_bits, frac_bits):
    machine_epsilon = num_of_bits**(-1.0 * frac_bits)
    x = (num_of_bits ** (num_of_bits**(exponent_bits - 1) - 1)) * (num_of_bits - machine_epsilon)
    print("Machine epsilon for this system:", machine_epsilon, "\n")
    cs111.print_float64(x)
    return x

return_largest_num(2.0, 5.0, 10.0) #IEEE 16-bit standard
return_largest_num(2.0, 8.0, 7.0) #bfloat 16-bit standard
```

```
Machine epsilon for this system: 0.0009765625

input     : 65504.0
as float64: 6.5504000000000000e+04
as hex    : 40effc0000000000
sign      : 0 means +
exponent  : 40e means 1038 - 1023 = 15
mantissa  : 1.1111111111000000000000000000000000000000000000000000

Machine epsilon for this system: 0.0078125

input     : 3.3895313892515355e+38
as float64: 3.3895313892515355e+38
as hex    : 47efe00000000000
sign      : 0 means +
exponent  : 47e means 1150 - 1023 = 127
mantissa  : 1.1111111000000000000000000000000000000000000000000000
```

In the output above, the former chunk relates the the 16-bit IEEE system, while the latter chunk relates to Google's bfloat. The decimal representation is the input field, the exact expression was given above the code.

## 4.4 Advantages & Disadvantages

### 4.4.1 Advantage of TPU

The clear adventage of Google's TPU is that, since we don't need as high precision, and there's less floating point representations between each integer, the linear algebra computation is accelerated significantly, meaning machine learning/neural network algorithms run faster here. We can also see that even though there's less floating points between each number, since the exponent number of bits is larger in bfloat, it can represent a much larger number before hitting the $\infty$ representation.

### 4.4.2 Advantage of TEEE 16-bit Standard

TPU's advantage is also an obvious disadvantage, as the system is mostly applied where precision does not matter – since it cannot. With Google's TPU, we lost 3 bits of precision (3 bits go from frac to the exponent), and thus an incredible amount of precision is lost; bfloat calculations should only be expected to be accurate within $1e-3$, since machine epsilon for this system is $7.8125e-3$. This means that the number of applications the TPU system has is narrowed fairly significantly; because of this, IEEE 16-bit standard has a greater number of applications, I presume.