Draft version May 16, 2018
Typeset using LaTeX **twocolumn** style in AASTeX61

CHARACTERIZING A CCD

Jagdeep Singh

(Dated: May 14, 2018)

1. INTRODUCTION

In this lab, we measured and characterized many properties of the KAF-3200E that is part of the ST-10XME camera. We compared our results with the specifications provided by the manufacturer. We analyzed the CCD's linearity, gain, read noise, and dark current by capturing a series of bias and flat field images.

Linearity is the range of the amount of light incident to the CCD that yields a linear relation when converting incoming photons to an output signal. The upper limit determined by full well capacity or ADU saturation limit. Full well capacity is the total amount of electrons a pixel can hold and ADU saturation is determined by the number of bits in the ADU converter. 1x1 binding gives the highest resolution since each pixel reads as a single entity.

Gain is the conversion rate of electron to ADU signal. This is the linear proportion between electrons and output ADU. A higher gain amplifies a weak signal but also amplifies read noise.

Dark current is the inherent noise in the CCD produced by electrons being in an excited state. This applies to every CCD and is proportional to temperature. However, the dark current can be isolated and removed from the data by taking bias images.

2. SETTING-UP AND GETTING DATA

In order to collect the best possible bias and flat images at the University of Washington computer lab, we had to collect data using a few methods. First, we placed the CCD facing the inside of a cardboard box with very little ambient light illuminating the surface around the CCD. Bias frames were obtained with the CCD shutter shut and with the lens cap on. After a few exposures, the lens cap was taken off and we began taking flats. We attempted to get flats by removing the lens cap and placing the CCD under the cardboard box but the flats were not uniform due to light leaking into the box. We were able to get the flats by placing a napkin over the CCD and then placing the box over it. This method proved successful for getting a uniform light source and keeping the CCD from over saturating. We then took several images across a range of 0-120 seconds to obtain a linearity graph for the CCD.

3. DATA

3.1. *Linearity*

For this lab, we set the CCD to 1x1 binning for all the images collected. The full well capacity is 59,000 ADU and the ADU saturation was set by its 16 bit converter and equal to 65,536 ADU. For the 1x1 binning, the full well capacity will be reached before the ADU saturation limit. This limit can be changed by setting the CCD to 2x2 binning. With 2x2 binning, every 4 pixels is treated as one pixel, which increases the full well capacity. However, the ADU saturation is the same, thus ADU saturation is reached first.

By graphing the exposure time vs the mean count of the incident photons to the CCD, we obtained the linearity. Using the graph, the linearity can be found by following the linear trend with increasing exposure time. The data for the 0-120 sec exposure times ranged from 1,176 to 55,499. From the graph, its clear that the mean counts follow linearly with the exposure times.

Since the CCD has a full well capacity of 150,000 electrons and an ADU saturation of 104,857 electrons, ADU saturation will be reached before full well no matter the binning since it is the lower limit.

$$ADUSaturation = 2^b its * gain$$

It is important to note that full well and ADU saturation are measures of the CCD and not the observed light. So no matter what is observed, the same linearity applies.

3.2. *Gain and Read Noise*

We then calculated the gain and read noise using our collected images to compare with the manufacturer's value of 1.3 e-/ADU and 8.8 RMS. This was done using two biases and two flats and comparing their mean and standard deviations. A 0.563 and 0.28 second exposures (Sky 67 and Sky 68) used for the flat field images. The gain and read noise calculated were 0.41577 e-/ADU and 3.53388 RMS. These values vary widely from the manufactures values. One reason for this is that the flats were taken at different exposure times, so our comparison has inherent error. Attempted to correct this by using flats with similar mean counts since they are closest for same length exposure flats. Also, we had over exposure in some regions of the images from uneven light exposure to the CCD.

Also, we calculated gain and read noise using a sub-region of the images, using a 100x100 sub-region that looked to have uniform exposures. This method calculated gain and read noise as 1.3747 e-/ADU and 11.94 RMS. We then repeated this method with a 200x200 sub-region and calculated nearly identical gain and read noise values of 1.284 e-/ADU and 10.846 RMS.

3.3. *Dark Current*

The relationship between bulk dark current and temperature follows the formula:

$$D = 2.5 * 10^{15} AIT^{1.5} e^{-Eg/2kT}$$

The dark current value was calculated by graphing the dark current vs inverse temperature and using a least squares fit curve. Using an initial guess of Eg = 1.1 eV, we obtained a measurement of 1.219 eV. We then used this estimate to find the dark current at zero degrees Celsius as 0.424 e-/pixel/sec. Which is widely off the manufacturer's value of 1 e-/pixel/sec. This can be explained by looking at the curve fit and seeing that it varies from the data at the edges. We could calculate a better dark current estimate at zero degrees Celsius using a smaller data set around zero Celsius. This will allow for a better fit for the data and thus a better estimate.

As temperature increases, the Dark Current will approach infinity since as at higher temperatures, more and more electrons are in an excited state and enter the conduction band. However, infinite temperature is not possible so the actual limit on temperature will be determined by other factors of the CCD, factors such as melting point of silicon. And as temperature decreases, Dark Current will approach zero.

4. CONCLUSION

We calculated and compared linearity, gain, read noise and dark current with the values given by the manufacturer. For linearity, we did not reach the provided full well capacity so more data is needed. We used two methods for calculating gain and read noise and found sub-regions were best for accurate measurements that agree with the manufacturer's provided values. Lastly dark current was found to be approximately 0.424 e-/pixel/sec at 0 Celsius, varying from the manufacturer's given value of 1 e-/pixel/sec.

Detector Gain

You could calculate the mean of each flat image manually using IRAF. Alternatively, you could calculate the means for all of the images automatically using Python.

In [26]:
```
from astropy.io import fits
import numpy as np
import matplotlib.pyplot as plt
# glob serves some of the same functions as ls in the terminal
import glob
```

FITS Headers

The headers of the FITS files contain the exposure times of the flat images. Now we use `fits.open` instead of `fits.getdata`. HDU stands for Header/Data Unit.

In [30]:
```
hdu = fits.open('data/480_2018_.Flat.180S0X1.Sky.51.fits')
header = hdu[0].header
print(header['exptime'])
```

```
180.0
```

Calculating Mean Counts

We can find all of the flat images, assuming they all have 'Flat' in the name.

You will need to change the path to the directory containing your data.

In [31]:
```
# This is equivalent to $ ls Flat*.fits
flat_list = glob.glob('data/480_2018_.Flat.*.fits')
flat_list
```

Out[31]:
```
['data/480_2018_.Flat.162S0X1.Sky.52.fits',
 'data/480_2018_.Flat.0S0X1.Sky.68.fits',
 'data/480_2018_.Flat.9S0X1.Sky.62.fits',
 'data/480_2018_.Flat.180S0X1.Sky.51.fits',
 'data/480_2018_.Flat.120S0X1.Sky.56.fits',
 'data/480_2018_.Flat.1S0X1.Sky.66.fits',
 'data/480_2018_.Flat.36S0X1.Sky.59.fits',
 'data/480_2018_.Flat.18S0X1.Sky.61.fits',
 'data/480_2018_.Flat.54S0X1.Sky.58.fits',
 'data/480_2018_.Flat.90S0X1.Sky.69.fits',
 'data/480_2018_.Flat.4S0X1.Sky.63.fits',
 'data/480_2018_.Flat.0S0X1.Sky.67.fits',
 'data/480_2018_.Flat.144S0X1.Sky.53.fits',
 'data/480_2018_.Flat.126S0X1.Sky.54.fits',
 'data/480_2018_.Flat.72S0X1.Sky.57.fits']
```

Now we can loop through each flat image, and keep track of the exposure time and mean counts

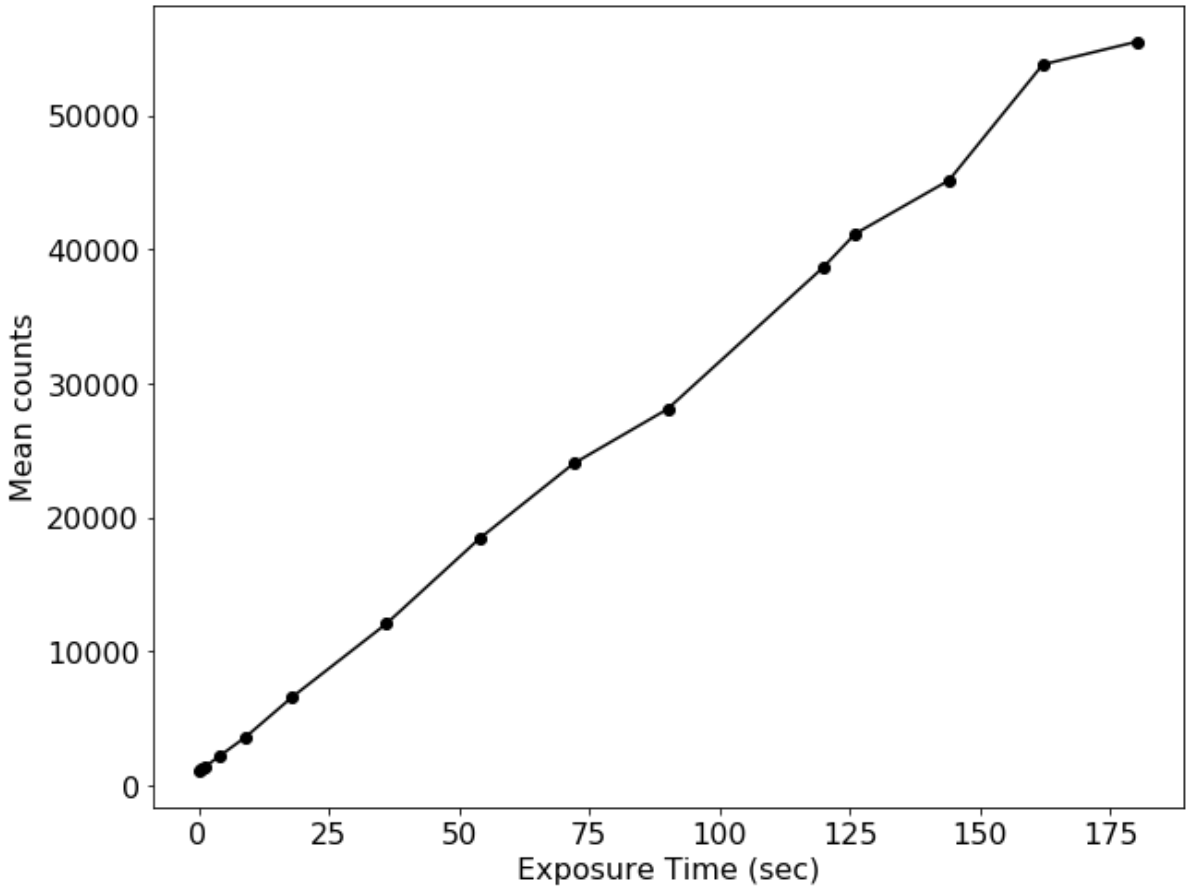
In [32]:
```
# These are empty lists (arrays) to store the exposure times and mean co
unts
exp_times = []
means = []

for filename in flat_list:
    # Open the FITS file
    hdu = fits.open(filename)
    print(filename)

    exptime = hdu[0].header['exptime']
    print('Exposure time {} sec'.format(exptime))

    # This will append the exposure time for each image to the array
    exp_times.append(exptime)

    # Same for mean counts
    mean_counts = np.mean(hdu[0].data)
    print('Mean counts: {:.2f}\n'.format(mean_counts))
    means.append(mean_counts)

# Convert to Numpy arrays so they can be sorted
exp_times = np.array(exp_times)
means = np.array(means)

# Sort by exposure time so the plot looks correct
time_sort = np.argsort(exp_times)
exp_times = exp_times[time_sort]
means = means[time_sort]
```

```
data/480_2018_.Flat.162S0X1.Sky.52.fits
Exposure time 162.0 sec
Mean counts: 53797.94

data/480_2018_.Flat.0S0X1.Sky.68.fits
Exposure time 0.281 sec
Mean counts: 1099.78

data/480_2018_.Flat.9S0X1.Sky.62.fits
Exposure time 9.0 sec
Mean counts: 3591.21

data/480_2018_.Flat.180S0X1.Sky.51.fits
Exposure time 180.0 sec
Mean counts: 55499.41

data/480_2018_.Flat.120S0X1.Sky.56.fits
Exposure time 120.0 sec
Mean counts: 38706.95

data/480_2018_.Flat.1S0X1.Sky.66.fits
Exposure time 1.125 sec
Mean counts: 1345.15

data/480_2018_.Flat.36S0X1.Sky.59.fits
Exposure time 36.0 sec
Mean counts: 12045.93

data/480_2018_.Flat.18S0X1.Sky.61.fits
Exposure time 18.0 sec
Mean counts: 6597.05

data/480_2018_.Flat.54S0X1.Sky.58.fits
Exposure time 54.0 sec
Mean counts: 18446.11

data/480_2018_.Flat.90S0X1.Sky.69.fits
Exposure time 90.0 sec
Mean counts: 28085.63

data/480_2018_.Flat.4S0X1.Sky.63.fits
Exposure time 4.0 sec
Mean counts: 2180.11

data/480_2018_.Flat.0S0X1.Sky.67.fits
Exposure time 0.563 sec
Mean counts: 1176.45

data/480_2018_.Flat.144S0X1.Sky.53.fits
Exposure time 144.0 sec
Mean counts: 45158.02

data/480_2018_.Flat.126S0X1.Sky.54.fits
Exposure time 126.0 sec
Mean counts: 41164.23

data/480_2018_.Flat.72S0X1.Sky.57.fits
```

```
Exposure time 72.0 sec
Mean counts: 24023.60
```

Plot mean counts versus exposure time

In [33]:
```
plt.figure(figsize=(10, 8))
plt.rcParams['font.size'] = 16

plt.plot(exp_times, means, '-ko')
plt.xlabel('Exposure Time (sec)')
plt.ylabel('Mean counts')
plt.show()
```



Detector Gain

This is a Jupyter notebook, which allows us to write and run Python code in a realtime, interactive fashion. There are two main kinds of cells

- Markdown cells like this one. We can make notes and write equations in LaTeX (e.g, $e = mc^2$)
- Code cells like the one below. Type Shift-Enter to run a code cell.

Importing packages

We first need to import the packages we'll use to load, analyze, and visualize the data.

In [52]:
```
# Astropy is
from astropy.io import fits

# Numpy is a powerful package for numerical analysis.
import numpy as np

# Matplotlib is the most popular Python plotting package.
import matplotlib.pyplot as plt
```

Working with FITS files in Python

At their most basic, FITS files are just arrays of integers corresponding to the counts in each pixel. Usually FITS files also contain metadata in a header, such as the exposure time, or transformations from pixel coordinates to sky coordinates (RA and Dec.). The astropy fits package allows us to load FITS files as Numpy arrays, which we can then perform calculations on.

Note the call to `astype(np.int32)` at the end of `fits.getdata`. This is very important for getting the correct result for computer science reasons you can ignore if you want to. If you're interested in why this matters, see the end of the notebook.

Replace 'data/Flat.15S0X1.V.14.fits' with one of your images.

In [53]:
```
image = fits.getdata('data/480_2018_.Flat.0S0X1.Sky.68.fits').astype(np.
int32)
print(image.shape)
print('')
print(image)
```

```
(1472, 2184)

[[1091 1085 1113 ... 1057 1062 1055]
 [1125 1101 1111 ... 1081 1074 1078]
 [1084 1123 1114 ... 1078 1069 1071]
 ...
 [1095 1109 1125 ... 1084 1093 1068]
 [1106 1127 1120 ... 1083 1068 1050]
 [1100 1105 1103 ... 1081 1066 1067]]
```

We can see that the image is just a 1472 x 2184 array of integers.

Load Images

Using the `fits.getdata` function, Load the two flat images and two bias images that you will use to calculate the gain. Remembers to put `.astype(np.int32)` at the end

In [55]:
```
flat1 = fits.getdata('data/480_2018_.Flat.0S0X1.Sky.68.fits').astype(np.
int32)
flat2 = fits.getdata('data/480_2018_.Flat.0S0X1.Sky.67.fits').astype(np.
int32)
bias1 = fits.getdata('data/480_2018_.Bias(0.0S0X1).70.fits').astype(np.i
nt32)
bias2 = fits.getdata('data/480_2018_.Bias(0.0S0X1).71.fits').astype(np.i
nt32)
```

You can write Equation 3 in code much like it appears in the instructions. You can make variables that are the same as the variables in the equation.

- For the mean: `mean_f1 = np.mean(flat1)`
- For the squared standard deviation of the image difference (σ^2): `sigma_f1f2 = np.std(flat_1 - flat_2)`

In the cell below, write out the equation. Use parentheses to ensure the correct order of operations.

In [56]:
```
mean_f1 = np.mean(flat1)
print(mean_f1)

mean_f2 = np.mean(flat2)
print(mean_f2)

mean_b1 = np.mean(bias1)
print(mean_b1)

mean_b2 = np.mean(bias2)
print(mean_b2)

sigma_f1f2 = np.std(flat1 - flat2)
print(sigma_f1f2)

sigma_b1b2 = np.std(bias1 - bias2)
print(sigma_f1f2)

gain = (((mean_f1 + mean_f2) - (mean_b1 + mean_b2))
        / (sigma_f1f2 ** 2 - (sigma_b1b2 **2)))
print('gain is: ', gain)
```

```
1099.779774969143
1176.4469010665512
1026.8766439346432
1025.2086873158544
26.145429810051535
26.145429810051535
gain is:  0.4157716002625594
```

Read noise

Calculate the read noise. The square root function is `np.sqrt()`

In [58]:
```
read_noise = (gain * sigma_b1b2) / np.sqrt(2)
read_noise_adu = sigma_b1b2/np.sqrt(2)

print('read noise is: ', read_noise)
print('read noise in ADU is: ', read_noise_adu)
```

```
read noise is:  3.533889911885895
read noise in ADU is:  8.499594271600674
```

Working with Subregions

Is your gain value close to the manufacturer's specification (1.3 e-/ADU)? What happens when you calculate the gain for a subregion of the detector? In the example below, we select a 100 x 100 pixel sub region from `row0` to `row1` and `colm0` to `colm1`.

In [59]:
```
row0 = 100
row1 = 200
colm0 = 200
colm1 = 300

# This is called taking a 'slice' of the array, i.e. a subregion
sub_image = image[row0:row1, colm0:colm1]

print(sub_image.shape)
print('')
print(sub_image)
```

```
(100, 100)

[[1128 1134 1115 ... 1124 1103 1116]
 [1121 1099 1133 ... 1101 1104 1112]
 [1128 1108 1103 ... 1119 1138 1107]
 ...
 [1155 1134 1137 ... 1139 1160 1124]
 [1119 1145 1114 ... 1135 1132 1135]
 [1131 1095 1103 ... 1109 1100 1132]]
```

Now try calculating the gain in different subregions by varying `row0`, `row1`, `colm0`, and `colm1` Try different sizes, e.g. 100 x 100, 200 x 200, etc

In [60]:
```
row0 = 100
row1 = 200
colm0 = 200
colm1 = 300

sub_flat1 = flat1[row0:row1, colm0:colm1]
sub_flat2 = flat2[row0:row1, colm0:colm1]
sub_bias1 = bias1[row0:row1, colm0:colm1]
sub_bias2 = bias2[row0:row1, colm0:colm1]

sub_mean_f1 = np.mean(sub_flat1)
print(sub_mean_f1)

sub_mean_f2 = np.mean(sub_flat2)
print(sub_mean_f2)

sub_mean_b1 = np.mean(sub_bias1)
print(sub_mean_b1)

sub_mean_b2 = np.mean(sub_bias2)
print(sub_mean_b2)

sub_sigma_f1f2 = np.std(sub_flat1 - sub_flat2)
print(sub_sigma_f1f2)

sub_sigma_b1b2 = np.std(sub_bias1 - sub_bias2)
print(sub_sigma_b1b2)

sub_gain = ((sub_mean_f1 + sub_mean_f2 - sub_mean_b1 - sub_mean_b2)
            / (sub_sigma_f1f2**2 - sub_sigma_b1b2**2))
print(sub_gain)

sub_read_noise = sub_gain * sub_sigma_b1b2 / np.sqrt(2)
print(sub_read_noise)
```

```
1119.9435
1216.8025
1027.3396
1026.6571
18.88252946509021
12.282821082715486
1.3746900086107472
11.939548501874592
```

In [61]:
```
row0 = 800
row1 = 1000
colm0 = 1600
colm1 = 1800

sub_flat1 = flat1[row0:row1, colm0:colm1]
sub_flat2 = flat2[row0:row1, colm0:colm1]
sub_bias1 = bias1[row0:row1, colm0:colm1]
sub_bias2 = bias2[row0:row1, colm0:colm1]

sub_mean_f1 = np.mean(sub_flat1)
print(sub_mean_f1)

sub_mean_f2 = np.mean(sub_flat2)
print(sub_mean_f2)

sub_mean_b1 = np.mean(sub_bias1)
print(sub_mean_b1)

sub_mean_b2 = np.mean(sub_bias2)
print(sub_mean_b2)

sub_sigma_f1f2 = np.std(sub_flat1 - sub_flat2)
print(sub_sigma_f1f2)

sub_sigma_b1b2 = np.std(sub_bias1 - sub_bias2)
print(sub_sigma_b1b2)

sub_gain = ((sub_mean_f1 + sub_mean_f2 - sub_mean_b1 - sub_mean_b2)
            / (sub_sigma_f1f2**2 - sub_sigma_b1b2**2))
print(sub_gain)

sub_read_noise = sub_gain * sub_sigma_b1b2 / np.sqrt(2)
print(sub_read_noise)
```

```
1080.489875
1137.89135
1025.484725
1023.8022
16.56597545647026
11.946716478780896
1.2838622665552797
10.845560320184884
```

Adding Bells and Whistles

Below is a little more advanced Python programming. You can try running it on your own data and tweaking the code to see how it works.

You probably found varying and typing everything out by hand to be a bit tedious. Alternatively, we can write a function to calculate the gain. The text in triple quotations is called the doc(umentation)string. It tells the user what the function does, what the arguments are, and what the function returns.

In [62]:
```
def calculate_gain(flat_1, flat_2, bias_1, bias_2):
    """
    Calculate detector gain given two flat frames and two bias frames.

    Parameters
    ----------
    flat_1, flat_2 : numpy.array_like
        The flat frames
    bias_1, bias_2 : numpy.array_like
        The bias frames

    Returns
    -------
    gain : float
        The detector gain
    """
    # This is Equation 3 from the assignment
    numerator = (np.mean(flat_1) + np.mean(flat_2)) - (np.mean(bias_1) +
 np.mean(bias_2))
    denominator = np.std((flat_1 - flat_2)) ** 2 - np.std((bias_1 - bias
_2)) ** 2
    gain = numerator / denominator

    return gain
```

Replace these files with your own data to try out the code.

In [63]:
```
flat_1 = flat1
flat_2 = flat2
bias_1 = bias1
bias_2 = bias2
```

Calculating the gain in different subregions

We can divide the detector up into a grid of subregions. The `bins` are the row and column boundaries of the subregions. Effectively we are making a coarse "map" of the gain calculated on different parts of the detector.

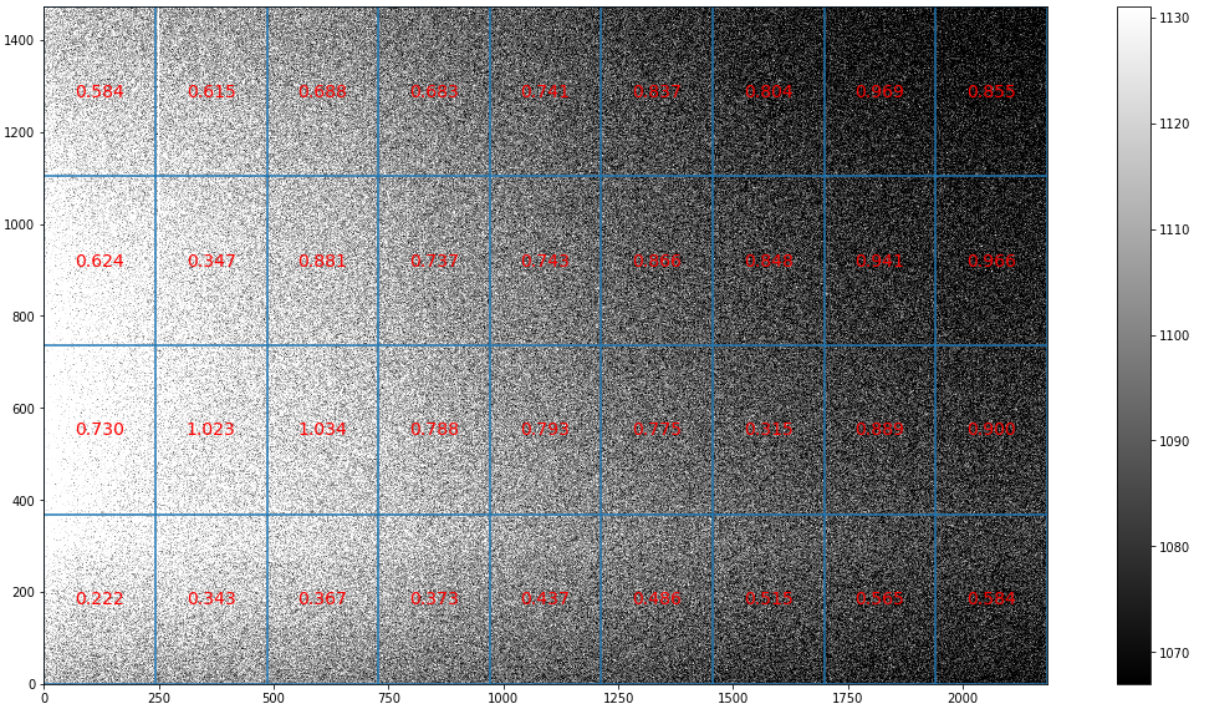
In [44]:
```
row_bins = np.linspace(0, flat_1.shape[0], 5).astype(int)
print(row_bins)
col_bins = np.linspace(0, flat_1.shape[1], 10).astype(int)
print(col_bins)
```

```
[   0  368  736 1104 1472]
[   0  242  485  728  970 1213 1456 1698 1941 2184]
```

In [45]:
```
# This is an array to store the gain in each subregion.
gain_map = np.zeros((len(row_bins) - 1, len(col_bins - 1)))

# This nested for loop goes through each subregion.
for ii in range(len(row_bins) - 1):
    for jj in range(len(col_bins) - 1):
        row_slice = slice(row_bins[ii], row_bins[ii + 1])
        col_slice = slice(col_bins[jj], col_bins[jj + 1])

        # The `local_gain` is the gain in the subregion
        local_gain = calculate_gain(flat_1[row_slice, col_slice], flat_2
[row_slice, col_slice],
                                    bias_1[row_slice, col_slice], bias_2
[row_slice, col_slice])

        # Store the local gain in the `gain_map`
        gain_map[ii, jj] = local_gain
```

Plot the gain values

In [46]:
```
plt.figure(figsize=(20, 10))

# Use the imshow function to plot one of the flat images for reference.
plt.imshow(flat_1, vmin=np.percentile(flat_1, 5), vmax=np.percentile(fla
t_1, 90),
           origin='lower', cmap='binary_r', interpolation='nearest')


# Plot the boundaries of the subregions
for row in row_bins:
    plt.axhline(row)
for col in col_bins:
    plt.axvline(col)

# Print the local gain value in each subregion
for ii in range(len(row_bins) - 1):
    for jj in range(len(col_bins) - 1):
        row_loc = (row_bins[ii] + row_bins[ii + 1]) / 2
        col_loc = (col_bins[jj] + col_bins[jj + 1]) / 2

        plt.text(col_loc, row_loc, '{:.3f}'.format(gain_map[ii, jj]),
                 ha='center', va='center', fontsize=14, color='r')

plt.xlim(0, col_bins[-1])
plt.ylim(0, row_bins[-1])

# Add a colorbar
plt.colorbar()
plt.show()
```



Unsigned versus signed integers

As their names suggest, unsigned integers can only represent non-negative numbers. That's 0 up to $2^n - 1$, where n is the number of bits the variable takes up in memory. Signed integers can have values from $-(2^{n-1})$ up to $2^{n-1} - 1$.

Negative values have no real physical meaning on an astronomical image, so the negative values are a waste of memory. Therefore images are usually stored as unsigned integers.

The problem is when we need to subtract two sets of unsigned integers, like we did in the gain calculation. This can give weird results if we're not careful.

In [47]:
```
a = np.array([1], dtype=np.uint16)
b = np.array([3], dtype=np.uint16)

print('a =', a)
print('b =', b)
print('a - b =', a - b)
```

```
a = [1]
b = [3]
a - b = [65534]
```

We can fix this by converting the unsigned 16-bit integers to signed 32-bit integers.

In [48]:
```
a = a.astype(np.int32)
b = b.astype(np.int32)

print('a =', a)
print('b =', b)
print('a - b =', a - b)
```

```
a = [1]
b = [3]
a - b = [-2]
```

Silicon Band Gap Energy

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
# We will use the scipy curve_fit function to fit a model to data.
from scipy.optimize import curve_fit
```

Enter data from table as numpy arrays

In [3]:
```
t_cel = np.array([-15., -10., -8., -6., -4.2, -2.3, 0., 2.,
                  3.6, 5.8, 8.2, 10., 12.8, 16.2, 20.])
adu = np.array([13, 15, 16, 17, 19, 22, 24, 28, 32, 37,
                  43, 50, 62, 89, 139])
```

Convert Celsius to Kelvin

In [4]:
```
t_kel = t_cel + 273
```

Convert ADU (counts) to electrons

In [6]:
```
electrons = adu * 2.3
electrons
```

Out[6]:
```
array([ 29.9,  34.5,  36.8,  39.1,  43.7,  50.6,  55.2,  64.4,  73.6,
        85.1,  98.9, 115. , 142.6, 204.7, 319.7])
```

Convert electrons to electrons per second

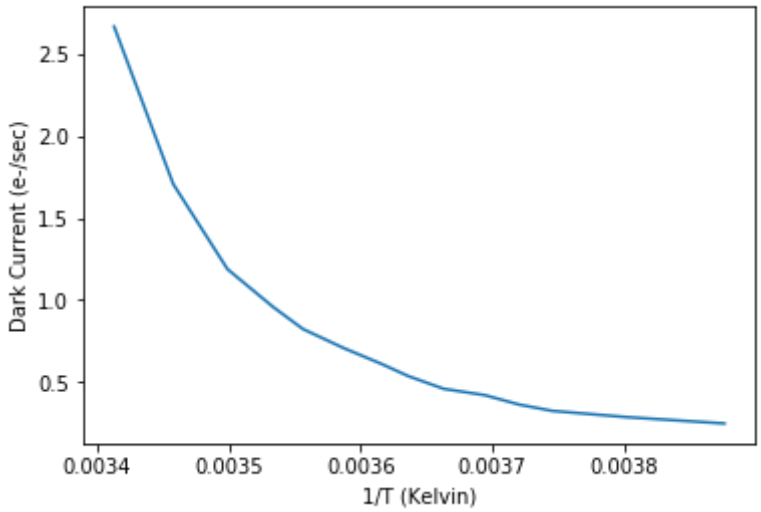
In [7]:
```
electrons_per_sec = electrons / 120
electrons_per_sec
```

Out[7]:
```
array([0.24916667, 0.2875    , 0.30666667, 0.32583333, 0.36416667,
       0.42166667, 0.46      , 0.53666667, 0.61333333, 0.70916667,
       0.82416667, 0.95833333, 1.18833333, 1.70583333, 2.66416667])
```

Plot dark current (e-/sec) vs. inverse temperature

Make a plot in the cell below

In [9]:
```
plt.plot(1/t_kel, electrons_per_sec, '-');
plt.xlabel("1/T (Kelvin)");
plt.ylabel("Dark Current (e-/sec)");
```



Fit for the band gap energy

We will try to fit a model for dark current of the form

$D = \alpha e^{-e_g/2k_bT}$

In [10]:
```
# The Boltzmann constant
k_b = 8.6175e-5

def dark_current(t_k, alpha, e_g):
    """
    Analytic expression for dark current as a function of temperature.

    Parameters
    ----------
    t_k : numpy.ndarray
        Temperature in Kelvin
    alpha : float
        Constant coefficient in front of exponential funtion.
    e_g : float
        Band gap energy in eV.

    Returns
    -------
    dark_current : numpy.ndarray
        Dark current in electrons/pixel/second.
    """
    dark_current = alpha * np.exp(-e_g / (2 * k_b * t_k))
    return dark_current
```

Inital guesses for parameters

We need initial guesses for the values of α and e_g .

In [11]:
```
t_0 = t_kel[-1]
d_0 = electrons_per_sec[-1]

alpha_0 =  d_0 / np.exp(-1.1 / (2 * k_b * t_0))
e_g_0 = 1.1
```

Least squares fit

In [12]:
```
p_opt, p_cov = curve_fit(dark_current, t_kel, electrons_per_sec, p0=[alp
ha_0, e_g_0])

# Errors in the fit
sig_alpha, sig_e_g = np.sqrt(np.diag(p_cov))
```

Best fit values

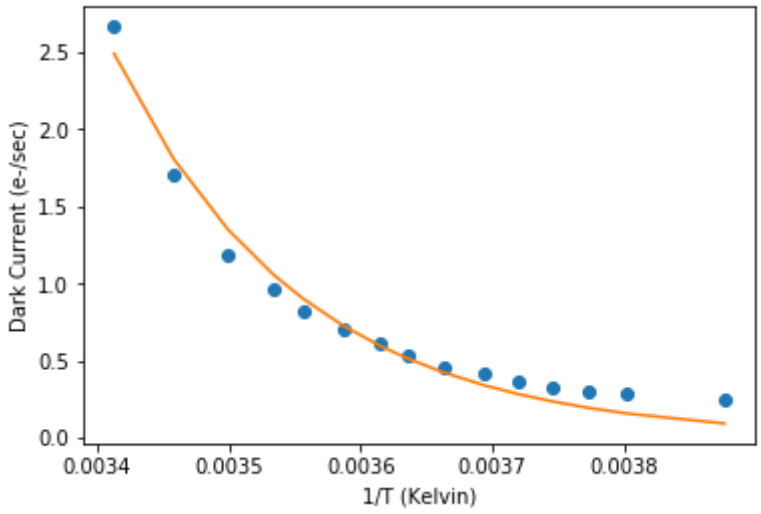
In [13]:
```
alpha_fit, e_g_fit = p_opt
print(alpha_fit)
print(e_g_fit)
```

```
76088119939.13002
1.2192320337409752
```

Plot the data and best fit model

Make a plot in the cell below. To plot the model, use the `dark_current` function with `alpha_fit` and `e_g_fit`.

In [103]:
```
plt.plot(1/t_kel, electrons_per_sec, 'o')
plt.plot(1/t_kel, dark_current(t_kel, alpha_fit, e_g_fit), '-')
plt.xlabel("1/T (Kelvin)")
plt.ylabel("Dark Current (e-/sec)")
plt.show()
```



01. How does your derived value of Eg compare with the energy gap of silicon (1.1 eV)?

Value of E_g is close but not exactly the E_g of silicon (1.1 eV).

02. What will D converge to at high temperatures? What will determine the upper temperature limit? What will D converge to at low temperatures?

$$D = \alpha e^{-e_g/2k_bT}$$

At high T values, the value of the exponent goes to 1, so the dark current is essentially equal to α. The limit is determined by the pixel area, dark current at 300K.

At low T values, the value of the exponent goes to 0, so D converges to 0.

04. Dark Current for ST-8XME is 1 e-/pixel/sec at 0 C. How does our value compare? Hypothesize as to the reasons for any discrepancies. How would you go about testing your hypotheses?

In [104]:
```
print(dark_current(273, alpha_fit, e_g_fit))
```

```
0.42420948894946114
```

Our value of the dark current at 0 C (273 K) is 0.424, which is much smaller. The reason for for this could be the curve fit we did in order to get alpha and e_g. We could test this by trying a different curve fit.