H2C Per-Antenna Calibration to Get Tsys

April 4, 2019

by Aaron Parsons

In this memo, we calibrate H2C data (with 8 functional antennas) to H1C observations between 100 and 200 MHz. The goal is to place the two data sets on a similar gain scale to ascertain whether the new Vivaldi-type feeds and associated signal chains have noticeably different system temperatures. This procedure follows the outline of HERA Memo 62, but leverages the larger number of functioning antennas to obtain a per-antenna (rather than per-baseline) calibration solution.

```
In [79]: from __future__ import print_function
         %matplotlib inline
         import pylab as plt, numpy as np
         import uvtools, pyuvdata, glob, aipy, hera_qm
         import hera_cal.abscal
         from scipy.interpolate import RectBivariateSpline
         from scipy.signal import medfilt
         from hera_cal import utils
         from hera_cal.noise import predict_noise_variance_from_autos
         import linsolve
```

1 Acquire H2C Wide-Band Data

```
In [80]: files = glob.glob('zen.2458560.2[0-9]*.HH.uvh5')
         files += glob.glob('zen.2458560.3[0-9]*.HH.uvh5')
         files += glob.glob('zen.2458560.4[0-2]*.HH.uvh5')
         ants = [0,1,11,13,23,25,26,39]

In [81]: FQ_MIN, FQ_MAX = 0.1e9, 0.2e9
         uvf = hera_cal.io.HERAData(files)
         meta = uvf.get_metadata_dict()
         antpos = dict(zip(uvf.antenna_numbers, uvf.antenna_positions))
         lsts = np.concatenate([uvf.lsts[f] for f in files])
         freqs = meta['freqs'].flatten()
         freqs = freqs[np.logical_and(FQ_MIN <= freqs, freqs <= FQ_MAX)]

In [82]: print('Reading', files[0], 'to', files[-1])
         print('LST Range:', lsts[0], lsts[-1])
         data = []
         for f in files:
```

```
            uvf = hera_cal.io.HERAData(f)
            data.append(uvf.read(frequencies=active_freqs)[0])
        data = data[0].concatenate(data[1:])
```

```
Reading zen.2458560.20069.HH.uvh5 to zen.2458560.42826.HH.uvh5
LST Range: 1.5471735448169264 2.9935033804294857
```

In [83]:
```
POL = 'xx'
ks = [k for k,d in data.items() if k[-1] == POL and k[0] != k[1] \
      and k[0] in ants and k[1] in ants]
ks_plot = ks[:3]
```

2 Acquire H1C IDR2.1 Data

In [85]:
```
IDR2_DIR = '../hera_idr2.1/'
files_h1c = glob.glob(IDR2_DIR + 'zen.grp1.of1.xx.LST.1.[5-9]*.uvOCRSL')
files_h1c += glob.glob(IDR2_DIR + 'zen.grp1.of1.xx.LST.2.[0-9]*.uvOCRSL')
```

In [86]:
```
uvf = pyuvdata.UVData()
uvf.read_miriad(files_h1c[0], run_check=False)
antpos_h1c = dict(zip(uvf.antenna_numbers, uvf.antenna_positions))
freqs_h1c = uvf.freq_array.flatten()
```

In [87]:
```
# Match redundant baselines between H1C and H2C
for ant in antpos:
    np.testing.assert_almost_equal(antpos[ant]-antpos[0], antpos_h1c[ant] - antpos_h1c[
reds = hera_cal.redcal.get_pos_reds(antpos_h1c)
ks_h1c = []
for k in ks:
    k_h1c = k
    f = uvf.get_flags(k_h1c)
    if np.all(f):
        # Find an equivalent redundant bl
        bl = k[:2]
        red = [red for red in reds if bl in red or bl[::-1] in red][0]
        if bl[::-1] in red:
            red = [blr[::-1] for blr in red]
        for blr in red:
            k_h1c = blr + k[-1:]
            f = uvf.get_flags(k_h1c)
            if not np.all(f):
                break
    ks_h1c.append(k_h1c)
```

In [88]:
```
print('Reading', files_h1c[0], 'to', files_h1c[-1])
data_h1c = {}
lsts_h1c = {}
```

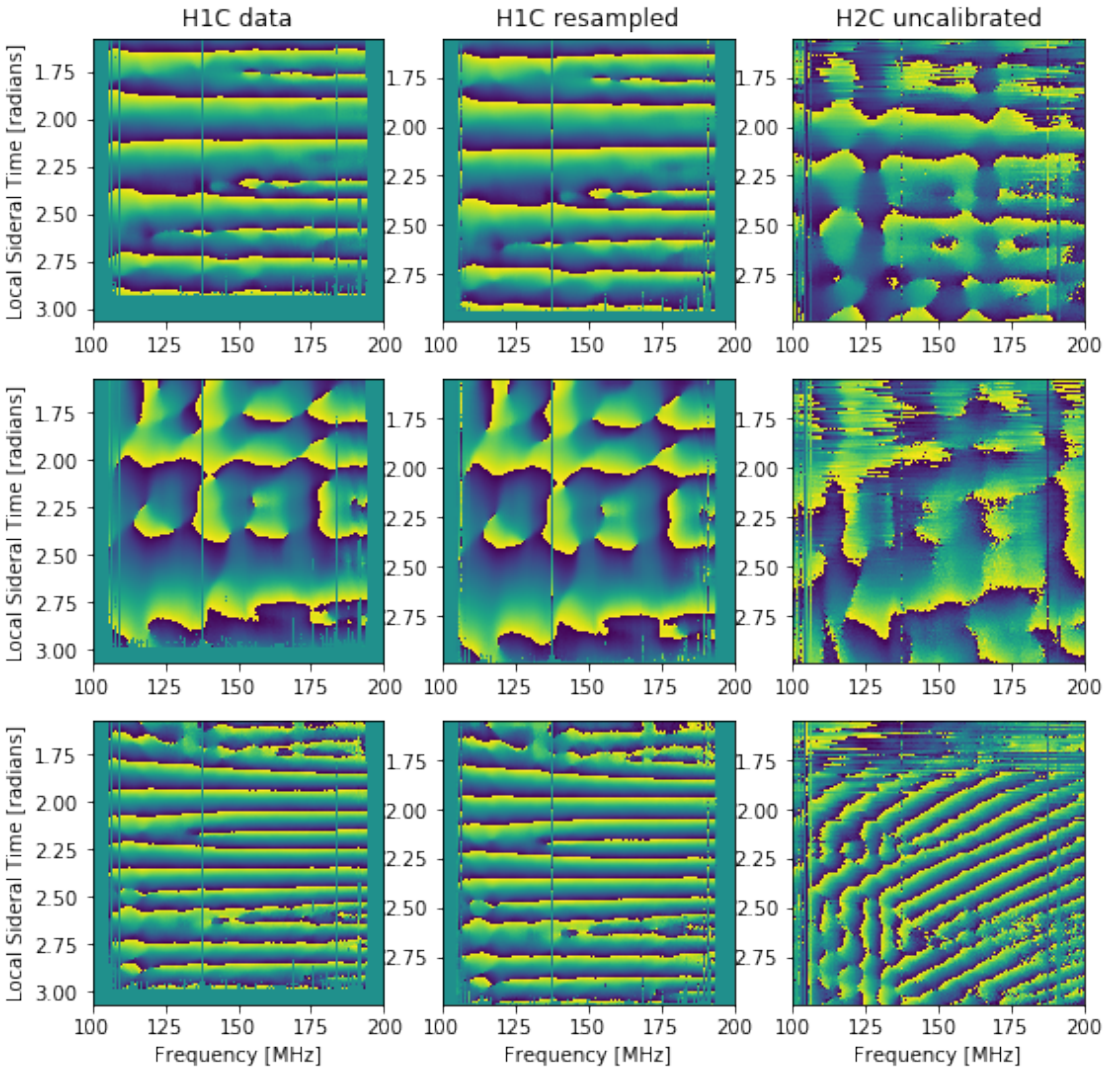
```
        for f in files_h1c:
            uvf = pyuvdata.UVData()
            uvf.read_miriad(f)
            for k,kh1c in zip(ks,ks_h1c):
                data_h1c[k] = data_h1c.get(k,[]) + [uvf.get_data(kh1c)]
                _lst = uvf.lst_array[np.append(*uvf._key2inds(kh1c)[:2])]
                lsts_h1c[k] = lsts_h1c.get(k,[]) + [_lst]
        data_h1c = {k: np.concatenate(dk, axis=0) for k,dk in data_h1c.items()}
        lsts_h1c = {k: np.concatenate(tk, axis=0) for k,tk in lsts_h1c.items()}
```

```
Reading ../hera_idr2.1/zen.grp1.of1.xx.LST.1.57015.uvOCRSL to ../hera_idr2.1/zen.grp1.of1.xx.LST
```

2.1 Resample H1C IDR2.1 Data onto H2C Observations

```
In [89]: data_h1c_interp = {}
         for k in ks:
             t,f,d = lsts_h1c[k], freqs_h1c, data_h1c[k]
             intr = RectBivariateSpline(t, f, d.real)
             inti = RectBivariateSpline(t, f, d.imag)
             data_h1c_interp[k] = intr(lsts,active_freqs) + 1j*inti(lsts,active_freqs)
```

```
In [120]: plt.figure(figsize=(9,3*len(ks_plot)))
          for cnt,k in enumerate(ks_plot):
              plt.subplot(len(ks_plot), 3, 3*cnt+1)
              uvtools.plot.waterfall(data_h1c[k], mode='phs',
                  extent=(freqs_h1c[0]/1e6,freqs_h1c[-1]/1e6,
                          lsts_h1c[k][-1],lsts_h1c[k][0]))
              plt.ylabel('Local Sideral Time [radians]')
              plt.subplot(len(ks_plot), 3, 3*cnt+2)
              uvtools.plot.waterfall(data_h1c_interp[k], mode='phs',
                  extent=(freqs[0]/1e6,freqs[-1]/1e6,
                          lsts[-1],lsts[0]))
              plt.subplot(len(ks_plot), 3, 3*cnt+3)
              uvtools.plot.waterfall(data[k], mode='phs',
                  extent=(freqs[0]/1e6,freqs[-1]/1e6,
                          lsts[-1],lsts[0]))
          plt.subplot(len(ks_plot),3,3*len(ks_plot)-2); plt.xlabel('Frequency [MHz]')
          plt.subplot(len(ks_plot),3,3*len(ks_plot)-1); plt.xlabel('Frequency [MHz]')
          plt.subplot(len(ks_plot),3,3*len(ks_plot)-0); plt.xlabel('Frequency [MHz]')
          plt.subplot(len(ks_plot),3,1); plt.title('H1C data')
          plt.subplot(len(ks_plot),3,2); plt.title('H1C resampled')
          plt.subplot(len(ks_plot),3,3); plt.title('H2C uncalibrated')
          plt.show()
```



3 Calibration of H2C Data to H1C Model

In this section, we do a per-antenna (for 9 antennas) calibration using a model with a single phase parameter (electronic delay, τ_i) and a single amplitude parameter (overall gain, g_i):

$$V_{\text{cal}} = V_{\text{uncal}} g_i g_j^* e^{2\pi i \nu(\tau_i - \tau_j)} \tag{1}$$

```
In [93]: def ant2key(a):
             return 'tau_%d_%s' % a

         def key2ant(k):
             _,ant,pol = k.split('_')
             return (int(ant),pol)
```

```
        def bl2eq(bl,s='-'):
            a1,a2 = utils.split_bl(bl)
            return s.join([ant2key(a1), ant2key(a2)])

        df = np.median(active_freqs[1] - active_freqs[0])
        dly_ants = {a:np.array([[0.]]) for k in ks for a in utils.split_bl(k)}
        print_lines = {k:str(k)+': ' for k in dly_ants.keys()}
        for i in range(3):
            dly_data = {}
            for cnt, k in enumerate(ks):
                a1,a2 = utils.split_bl(k)
                wgt = np.where(np.abs(data_h1c_interp[k]) < 2, 0, 1)
                wgt = np.mean(wgt, axis=0); wgt.shape = (1,-1)
                ratio = data[k] * data_h1c_interp[k].conj() * wgt
                ratio *= np.exp(-2j*np.pi * (dly_ants[a1]-dly_ants[a2]) * freqs)
                ratio /= np.abs(ratio)
                ratio = np.median(ratio, axis=0); ratio.shape = (1,-1)
                ratio *= wgt
                tau,_ = utils.fft_dly(ratio, df, wgts=wgt, medfilt=True, kernel=(1,11))
                dly_data[bl2eq(k)] = tau
            ls_dly = linsolve.LinearSolver(dly_data)
            sol = ls_dly.solve()
            dly_ants = {k: v + sol[ant2key(k)] for k,v in dly_ants.items()}
            for k in print_lines:
                print_lines[k] += '%+4.3f ns,' % (dly_ants[k] * 1e9)
        print('\n'.join(print_lines.values()))
```

```
invalid value encountered in divide
Invalid value encountered in median for 94 results
Invalid value encountered in median for 93 results
Invalid value encountered in median for 95 results
Invalid value encountered in median for 92 results
Invalid value encountered in median for 96 results
Invalid value encountered in median for 99 results


(11, 'Jxx'): -29.242 ns,-29.307 ns,-29.357 ns,
(26, 'Jxx'): +54.452 ns,+54.480 ns,+54.541 ns,
(23, 'Jxx'): -44.471 ns,-44.439 ns,-44.473 ns,
(1, 'Jxx'): -20.804 ns,-20.733 ns,-20.686 ns,
(0, 'Jxx'): -19.051 ns,-18.959 ns,-18.921 ns,
(13, 'Jxx'): +49.013 ns,+48.981 ns,+48.979 ns,
(25, 'Jxx'): +60.755 ns,+60.617 ns,+60.524 ns,
(39, 'Jxx'): -50.652 ns,-50.641 ns,-50.607 ns,
```

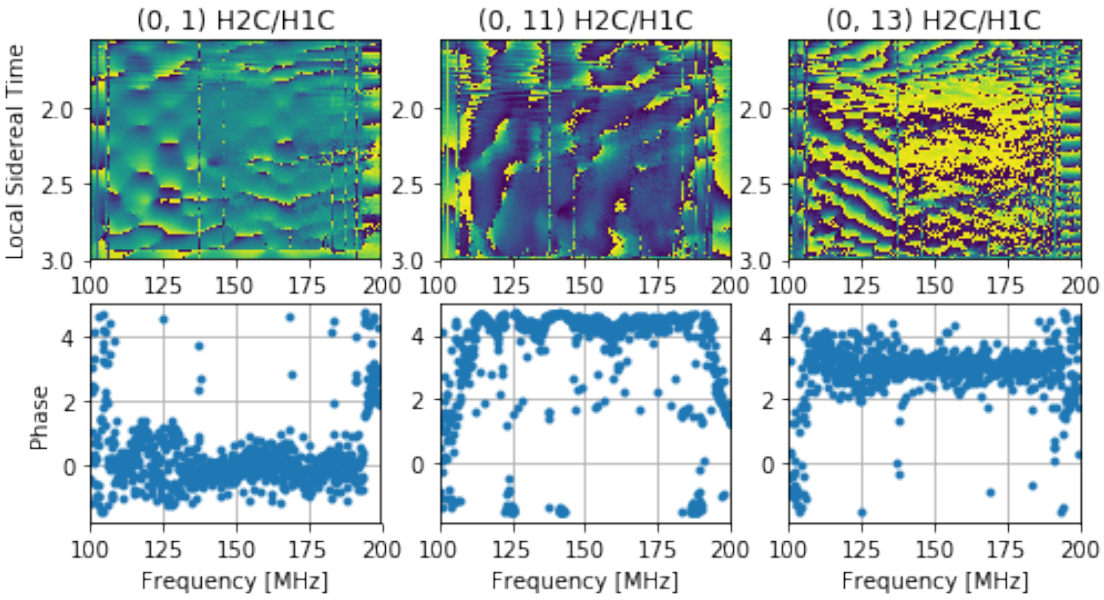
```
In [118]: data_cal = {}
          for k in data.keys():
```

```
        a1,a2 = utils.split_bl(k)
        tau1,tau2 = dly_ants.get(a1,0), dly_ants.get(a2,0)
        data_cal[k] = data[k] * np.exp(-2j*np.pi*freqs*(tau1 - tau2))
```

3.1 Verify Phase calibration

Below, we show that the calibration parameters used above do a reasonable job of flattening the phase difference between H2C and H1C data. The waterfall plots below show that there remain phase differences between H1C and H2C data. These are likely the result of beam differences between the Vivaldi-type and sleeved-dipole-type antenna feeds. Nonetheless, averaged over time (bottom plot), the overall phase difference is close to zero.

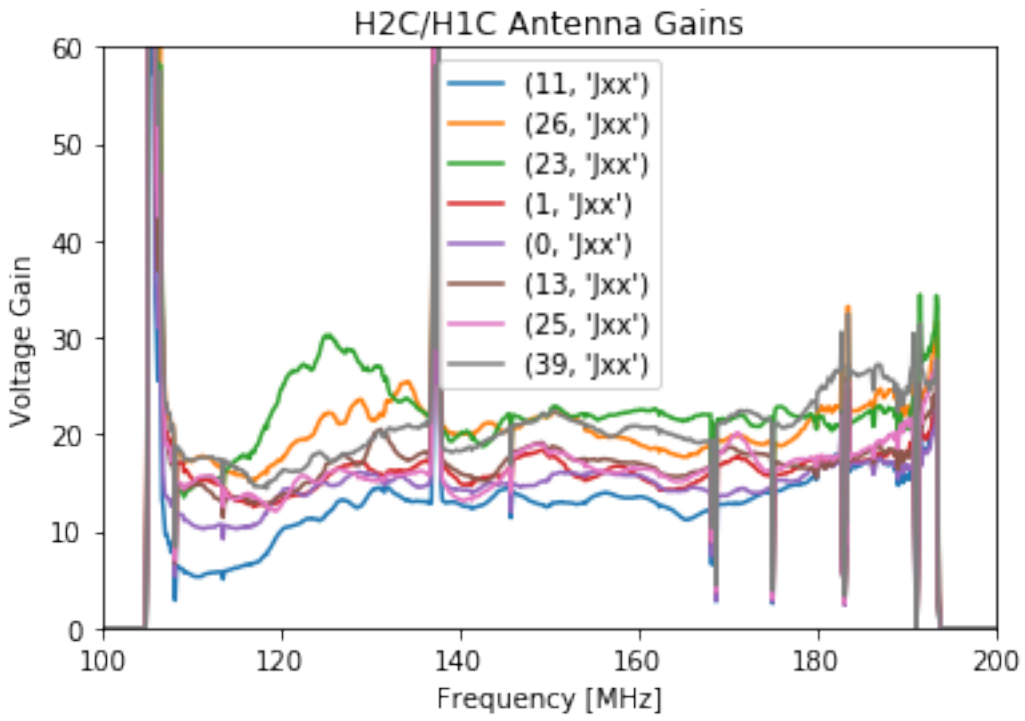
```
In [119]: plt.figure(figsize=(8,4))
          for cnt, k in enumerate(ks_plot):
              ratio = data_cal[k] * data_h1c_interp[k].conj() / np.abs(data_h1c_interp[k])**2
              plt.subplot(2, len(ks_plot), cnt + 1)
              uvtools.plot.waterfall(ratio, mode='phs',
                  extent=(freqs[0]/1e6,freqs[-1]/1e6,
                          lsts[-1],lsts[0]))
              plt.title(str(k[:-1]) + ' H2C/H1C')
              plt.subplot(2, len(ks_plot), cnt + 1 + len(ks_plot))
              medphs = np.angle(np.median(ratio, axis=0))
              plt.plot(freqs/1e6, np.where(medphs < -np.pi/2, medphs+2*np.pi, medphs),'.')
              plt.xlim(freqs_h1c[0]/1e6, freqs_h1c[-1]/1e6)
              plt.xlabel('Frequency [MHz]')
              plt.grid()
          plt.subplot(2, len(ks_plot), 1); plt.ylabel('Local Sidereal Time')
          plt.subplot(2, len(ks_plot), len(ks_plot)+1); plt.ylabel('Phase')
          plt.show()
```



3.2 Verify Amplitude Calibration

Here we do the same for the amplitude parameter, showing that after calibration, the gain ratio of H2C data to H1C is near unity. There will be a lot of spectral structure in the gains we use (we aren't smoothing them, and that's not ideal for delay analysis, but it should be sufficient for looking at noise.

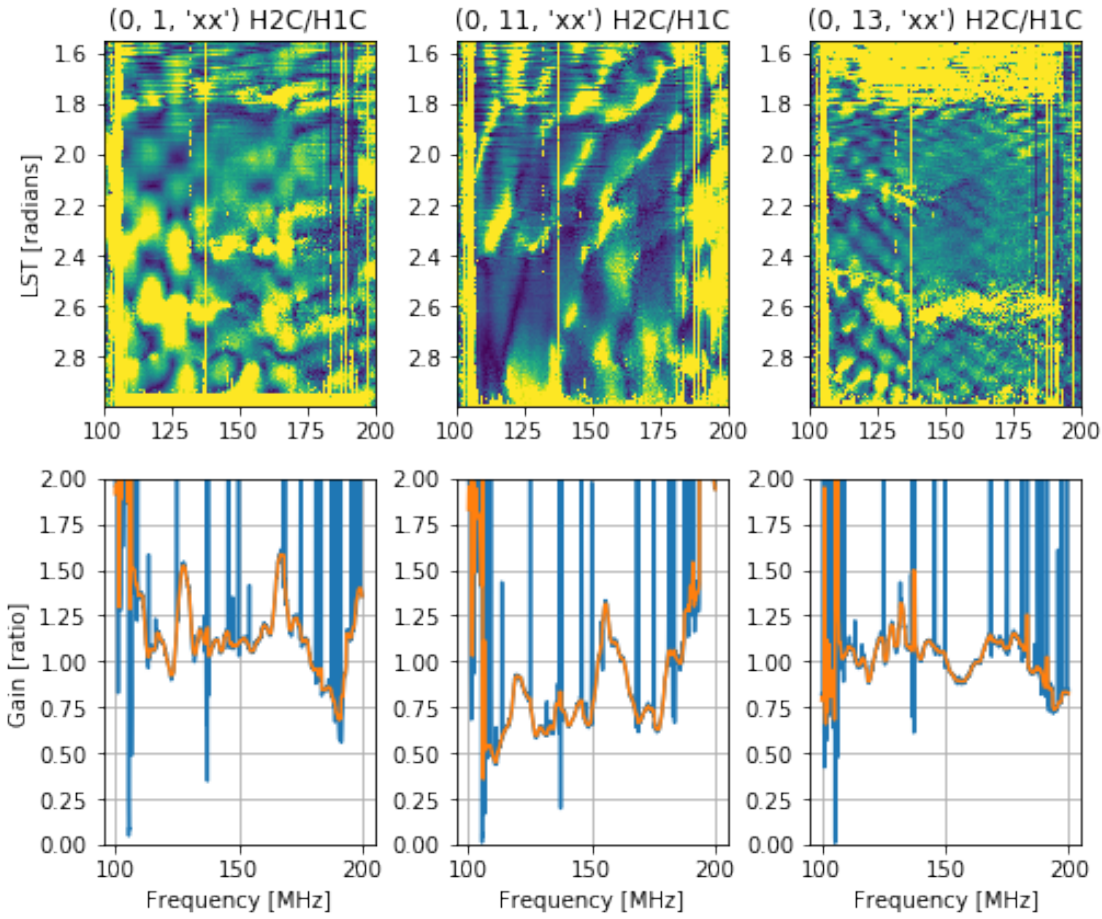
```
In [122]: amp_ants = {a: None for k in ks for a in utils.split_bl(k)}
          amp_data = {}
          for cnt, k in enumerate(ks):
              ratio = data_cal[k] * data_h1c_interp[k].conj() / np.abs(data_h1c_interp[k])**2
              gain = np.median(np.abs(ratio), axis=0)
              gain = medfilt(gain, kernel_size=(11,))
              amp_data[bl2eq(k,'+')] = np.log(gain)
          ls = linsolve.LinearSolver(amp_data)
          sol = ls.solve()
          amp_ants = {k: np.exp(sol[ant2key(k)]) for k in amp_ants.keys()}

In [123]: plt.figure()
          for k,gain in amp_ants.items():
              plt.plot(freqs/1e6, gain * wgt[0], label=k)
          plt.title('H2C/H1C Antenna Gains')
          plt.xlabel('Frequency [MHz]')
          plt.ylabel('Voltage Gain')
          plt.xlim(100,200)
          plt.ylim(0,60)
          plt.legend(loc='best')
          plt.show()
```




```
In [124]: for k in data.keys():
              a1,a2 = utils.split_bl(k)
              g1,g2 = amp_ants.get(a1,1), amp_ants.get(a2,1)
              data_cal[k] /= g1 * g2
```

```
In [125]: plt.figure(figsize=(8,6))
          for cnt, k in enumerate(ks_plot):
              ratio = data_cal[k] * data_h1c_interp[k].conj() / np.abs(data_h1c_interp[k])**2
              plt.subplot(2, len(ks_plot), cnt + 1)
              uvtools.plot.waterfall(ratio, mode='lin', mx=2, drng=2,
                  extent=(freqs[0]/1e6,freqs[-1]/1e6,
                          lsts[-1],lsts[0]))
              plt.title(str(k) + ' H2C/H1C')
              plt.subplot(2, len(ks_plot), cnt + 1 + len(ks_plot))
              gain = np.median(np.abs(ratio), axis=0)
              plt.plot(freqs/1e6, gain)
              gain = medfilt(gain, kernel_size=(11,))
              plt.plot(freqs/1e6, gain)
              plt.ylim(0,2)
              plt.xlabel('Frequency [MHz]')
              plt.grid()
          plt.subplots_adjust(top=0.95, bottom=0.1, hspace=.2, wspace=.3)
          plt.subplot(2, len(ks_plot), 1); plt.ylabel('LST [radians]')
```

```
plt.subplot(2, len(ks_plot), len(ks_plot)+1); plt.ylabel('Gain [ratio]')
plt.show()
```



3.3 Estimating Visibility Noise

Now that we finally have a gain scale set (at the ~25% level), we can look at the ratio of visibility scatter between H1C and H2C data to get an estimate of the relative system temperatures between the two data sets.

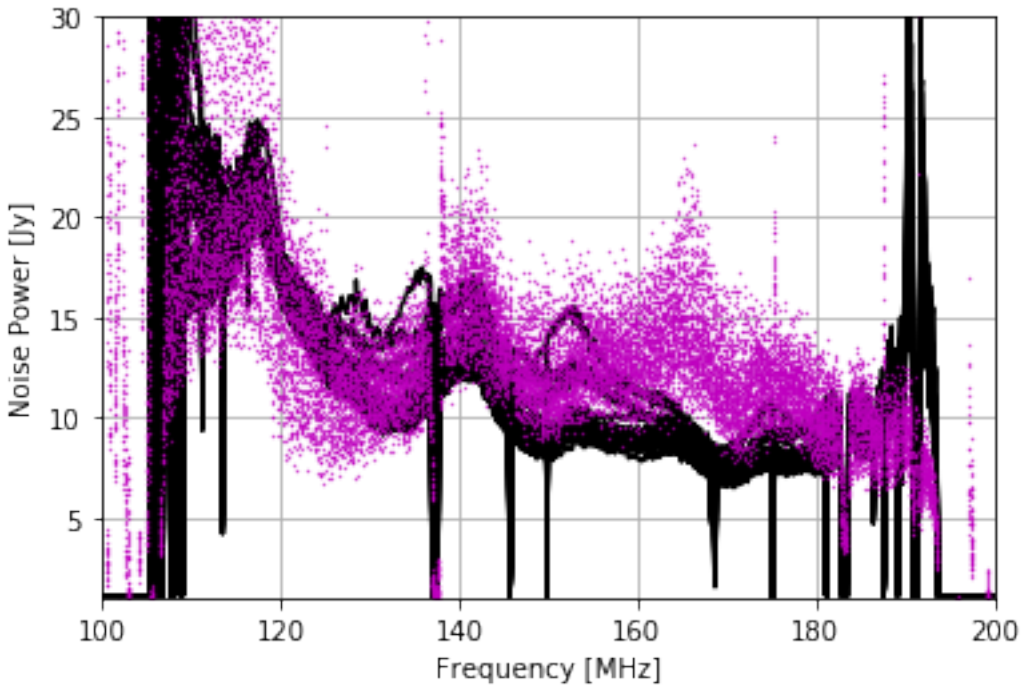
To estimate noise in H1C data, we take the STD data files output by the LST binner, which measure the scatter of points around the mean for observations over multiple days falling the the same LST bin.

For H2C data, we don't have this data product, so we instead compute derivatives of visibilities over the time and frequency axes (to remove the sky) and then renormalize the root-median-square of the result by the square root of the number of measurements (4) involved.

Alternately, we have another estimate of the noise in the H2C data, which is the auto-correlations, which we can place on the appropriate scale using our known observing parameters.

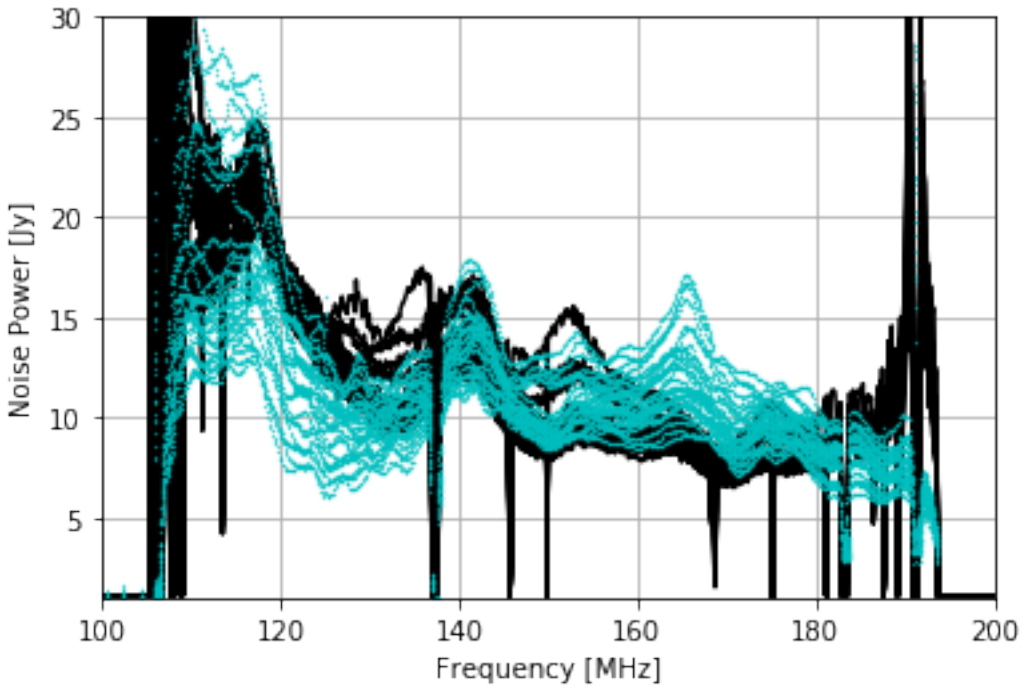
Finally, in order to compare apples to apples, we need to take into account the different in integration times between H1C (10.7 s) and H2C (8.6 s), and restrict ourselves to looking at the same LST range so that the sky temperatures are about the same.

```
In [126]: files = glob.glob('../hera_idr2.1/zen.grp1.of1.xx.STD.2.6*.uvOCRSL')
          lsts_std = {}
          std_h1c = {}
          for f in files:
              uvf = pyuvdata.UVData()
              uvf.read_miriad(f)
              for k,kh1c in zip(ks,ks_h1c):
                  _lst = uvf.lst_array[np.append(*uvf._key2inds(kh1c)[:2])]
                  lsts_std[k] = lsts_std.get(k,[]) + [_lst]
                  std_h1c[k] = std_h1c.get(k,[]) + [uvf.get_data(kh1c)]
          std_h1c = {k: np.concatenate(dk, axis=0) for k,dk in std_h1c.items()}
          lsts_std = {k: np.concatenate(tk, axis=0) for k,tk in lsts_std.items()}

In [136]: DT_H1C = 10.7 # s
          DT_H2C =  8.6 # s

          def estimate_noise(d, w):
              dd = d[1:,1:] + d[:-1,:-1] - d[1:,:-1] - d[:-1,1:]
              ww = w[1:,1:] * w[:-1,:-1] * w[1:,:-1] * w[:-1,1:]
              dd *= ww
              ww = ww * 4
              return np.sqrt(np.median(np.abs(dd)**2, axis=0) / 4)

          plt.figure()
          dfqs = 0.5 * (freqs[1:] + freqs[:-1])

          for k in ks:
              plt.plot(freqs_h1c/1e6, np.mean(np.abs(std_h1c[k]) \
                       * np.sqrt(DT_H1C/DT_H2C), axis=0), 'k-', label=k[:-1])

          for k in ks:
              _lsts = np.where(np.logical_and(lsts >= lsts_std[k][0],
                                              lsts <= lsts_std[k][-1]))
              noise = estimate_noise(data_cal[k][_lsts],
                                     np.ones_like(data_cal[k][_lsts]))
              # divide by np.log(2) for
              # the ratio of root-median-square to root-mean-square
              noise /= np.log(2)
              plt.plot(dfqs / 1e6, noise, 'mo', ms=.3, label=k)

          plt.ylabel('Noise Power [Jy]')
          plt.xlabel('Frequency [MHz]')
          plt.xlim(100,200)
          plt.ylim(1,30)
          plt.grid()
          plt.show()
```



In the plot above, black indicates the noise per H1C visibility near LST=2.6, and magenta indicates the same for H2C visibilities, as computed by differencing visibilities at adjacent times and frequencies.

In [137]:
```
plt.figure()

for k in ks:
    plt.plot(freqs_h1c/1e6, np.mean(np.abs(std_h1c[k]) \
              * np.sqrt(DT_H1C/DT_H2C), axis=0), 'k-', label=k[:-1])

for k in ks:
    df = np.mean(np.diff(freqs)) # Hz
    var = predict_noise_variance_from_autos(k, data_cal, dt=DT_H2C, df=df)
    noise = np.sqrt(var)
    plt.plot(freqs / 1e6, np.median(noise, axis=0), 'co', ms=.3)

plt.ylabel('Noise Power [Jy]')
plt.xlabel('Frequency [MHz]')
plt.xlim(100,200)
plt.ylim(1,30)
plt.grid()
plt.show()
```



This is the same plot, but this time, the H2C noise is estimated from auto-correlations.

4 Conclusions

Based on the final two plots, it looks like H2C observations have a lower system temperature than H1C observations from 100 to 150 MHz and above 180 MHz. In the 150-180 MHz range, H2C observations exhibit significant antenna-to-antenna variation, but broadly they appear to have a system temperature approximately 10-20% higher than in H1C, and a few outliers have a $T_{\rm sys}$ of ~50% more.

On this basis, it appears that H2C feeds are performing adequately in the 100-200 MHz band. We may wish to revisit the match between the feed and the FEM at ~165 MHz where there appears to be a higher receiver temperature.

In [ ]: