Cooker_Power5.1 - Solar Cooker Power Data Processing

Code in Python 2.7 (c) Paul Arveson and Solar Household Energy, Inc., 2018.

This program calculates power and efficiency for one test with up to four thermocouple channels. Power calculations are in compliance with ISO Standards 19867-1, 19869-1 and ASAE Standard S.580.1.

Data must be read in from three Excel files as defined below.
For 2016 the file name formats are Meta_yyyymmdd.xslx, Weather_yyyymmdd.xslx and Therm_yyyymmdd.xslx. For 2017 and later the underscores are not used.
Each input file must have one worksheet with the same file name (minus .xslx). This program generates up to four output Excel files containing (temp. diff, power) pairs, along with the metadata and power value at 50 deg. C temperature difference.

Import libraries

In [1]:
```
# Import libraries needed
# Allow about 20 seconds for these to load
import xlrd
import openpyxl
import pandas as pd
from pandas import Series, DataFrame
from pandas import DataFrame
import pandas as pd
import numpy as np
#import matplotlib.pyplot as plt
#import matplotlib
#%matplotlib
#matplotlib.style.use('ggplot')
%pylab inline
from scipy import linspace, polyval, polyfit, sqrt, stats, randn
from pylab import plot, title, show , legend
import string
import xlsxwriter
print "System is ready."
```

```
Populating the interactive namespace from numpy and matplotlib
System is ready.
```

Store physical constants

In [162]:
```
# Solar cooking physical constants

# Materials data
# Heat capacity of materials
cvw = 4186         # water, in J/kg deg. C
cvc = 2300         # canola oil
cvsteel = 466
cvalum = 897
cvcopper = 385
cvglass = 840
cvplastic = 1670    # average
cvsil = 1200         # average

# Density of materials (not used unless heating something other than water)
# From http://www.engineeringtoolbox.com/density-solids-d_1265.html
densw = 1          # water in g/ml
densc = 0.875      # canola oil at 65 deg. C
densalum = 2.7
densteel = 7.82
dencopper = 8.94
denglass = 2.6    # average
denpc = 1.2       # polycarbonate
denpe = 0.95      # polyethylene, polypropylene
densil = 1.7      # silicone rubber, average
```

Type in test date here:

Three files will be read in: Meta*, Weather* and Therm_.
All files for an experiment have the same date, e.g. 20160827.

In [313]:
```
# Please type experiment date below:
filedate = "20180714"
path = "C:\Users\paul\Test\\"
print "All files will be read from this path: ", path
print "filedate = ", filedate
```

```
All files will be read from this path:  C:\Users\paul\Test\
filedate =  20180714
```

In [314]:
```
# Read Metadata Excel file
#file = "Meta_" + filedate   # For 2016 data only
file = "Meta" + filedate
pathfile = "C:\Users\paul\Test\\" + file + ".xlsx"
print "Metadata file path: ", pathfile
xls_file = pd.ExcelFile(pathfile)
table = xls_file.parse(file)
meta = pd.DataFrame(table)
meta
```

```
Metadata file path:  C:\Users\paul\Test\Meta20180714.xlsx
```

Out[314]:

	Variable	Channel	Value
0	date	0	20180714
1	delay	0	0
2	location	0	Rockville, MD USA
3	technician	0	Paul Arveson
4	latitude	0	39.0476
5	longitude	0	-77.1412
6	elevation	0	122
7	pdesc	0	Apogee SP-212 powered pyranometer
8	poffset	0	0
9	psens	0	500
10	adesc	0	Adafruit anemometer m/s/V
11	aoffset	0	0.4
12	asens	0	21.17
13	label	0	T1
14	label	1	T2
15	label	2	T3
16	label	3	T4
17	disregard	0	0
18	disregard	1	0
19	disregard	2	1
20	disregard	3	1
21	cooker	0	Haines Model 2 #1
22	cooker	1	Haines Model 2 #1
23	cooker	2	not used
24	cooker	3	not used

	Variable	Channel	Value
25	intercept	0	60
26	intercept	1	60
27	intercept	2	0
28	intercept	3	0
29	area	0	0.5
30	area	1	0.5
31	area	2	0
32	area	3	0
33	mass	0	3.5
34	mass	1	3.5
35	mass	2	0
36	mass	3	0
37	cv	0	4186
38	cv	1	4186
39	cv	2	4186
40	cv	3	4186
41	notes	0	Clear sky.

Define all experiment parameters from metadata file

In [315]:
```
# Read metadata for one experiment
v = meta['Value']
testdate = str(v[0])
delay = str(v[1])  # No. of minutes at which to start processing
if testdate != filedate:
    print "filedate = ", filedate
    print "testdate = ", testdate
    print "Test date in Meta is not the same as requested file date!"
location = v[2]
technician = v[3]    # Test technician name
latitude = v[4]
longitude = v[5]
elevation = v[6]     # in meters re. sea level - used for finding local BP of wat
# BP of water based on elevation, from quora.com
bpf = 212 - .006036*elevation
bpc = (bpf - 32) * 5/9
print "Boiling point of water at local elevation = ", round(bpc,2), "deg. C"

# Instrument data
pdesc = v[7]                # Pyranometer description
poffset = float(v[8])       # Pyranometer offset V
psens = float(v[9])         # pyranometer sensitivity W/m2/V
adesc = v[10]                # Anemometer description
aoffset = float(v[11])      # Anemometer offset V
asens = float(v[12])        # Anemometer sensitivity m/s/V

# ALL OF THE PARAMETERS BELOW NEED TO BE DEFINED FOR EACH OF 4 CHANNELS:

# Define cooker test default parameters for each thermocouple channel:
label = [0]*4
disregard = [0]*4
cooker = [""]*4
intercept = [0.]*4
area = [0.]*4
mass = [0]*4
cv = [0.]*4

label[0] = v[13]   # Label in data columns   # THESE ARE NOT USED - HARD WIRED IN
label[1] = v[14]
label[2] = v[15]
label[3] = v[16]
disregard[0] = v[17]       # If channel is not used, disregard = 1
disregard[1] = v[18]       # There are always 4 channels of data, even if they ar
disregard[2] = v[19]
disregard[3] = v[20]
cooker[0] = v[21]          # Cooker 1 name
cooker[1] = v[22]          # Cooker 2 name
cooker[2] = v[23]          # Cooker 3 name
cooker[3] = v[24]          # Cooker 4 name
intercept[0] = float(v[25])       # Altitude angle in degrees for maximum area of
intercept[1] = float(v[26])
intercept[2] = float(v[27])
intercept[3] = float(v[28])
area[0] = v[29]            # Cooker area in sq. m. at intercept angle for maximum
area[1] = v[30]
area[2] = v[31]
```

```
area[3] = v[32]
mass[0] = v[33]            # load mass, kg
mass[1] = v[34]
mass[2] = v[35]
mass[3] = v[36]
cv[0] = float(v[37])             # Heat capacity of load, J/g deg. C
cv[1] = float(v[38])
cv[2] = float(v[39])
cv[3] = float(v[40])

# Notes to be displayed in reports:
notes = v[41]
nday = 0
```

```
Boiling point of water at local elevation =  99.59 deg. C
```

Read Weather data from Excel file

In [316]:
```
# Read the Weather Excel file
#file = "Weather_" + filedate   # For 2016 data only
file = "Weather" + filedate
pathfile = "C:\Users\paul\Test\\" + file + ".xlsx"
print "Weather data file path: ", pathfile
xls_file = pd.ExcelFile(pathfile)
table = xls_file.parse(file)
weather= pd.DataFrame(table)
title = weather.columns[0]  # Use for plot title
wrows = len(weather)
weather.head(5)
```

```
Weather data file path:  C:\Users\paul\Test\Weather20180714.xlsx
```

Out[316]:

	Plot Title: Weather	Unnamed: 1	500	21.17	5600	5600.1	Unnamed: 6
0	#	Date Time, GMT-04:00	Volt, V (SEN S/N: 20096285, LBL: Apogee1)	Volt, V (SEN S/N: 20096285, LBL: Anemometer)	Volt, V (SEN S/N: 20096285, LBL: Brooks1)	Volt, V (SEN S/N: 20096285, LBL: Brooks2)	NaN
1	1	2018-07-14 10:58:00	0.00019	0.00069	0.13748	0.15904	NaN
2	2	2018-07-14 10:58:00	0.00019	0.00069	0.13748	0.15904	NaN
3	3	2018-07-14 10:59:00	1.56878	0.42069	0.13752	0.16648	NaN
4	4	2018-07-14 11:00:00	1.57893	0.43641	0.13798	0.16735	NaN

Remove unused columns and label the columns

In [317]:
```
# Remove unneeded columns and label the weather columns
row1 = 1  # Index of the first row of valid data
# If there is a delay, add it to row1 here
wf = weather.iloc[row1:wrows-row1, 1:4]
# Rename the columns
label1 = "Time"
label2 = "Pyranometer"
label3 = "Anemometer"
wf.columns=[label1,label2,label3]   # Rename the 3 columns
# Remove the header columns from the data
tw = len(wf)   # time in minutes at end of test - for weather data
wf.head(5)
```

Out[317]:

	Time	Pyranometer	Anemometer
1	2018-07-14 10:58:00	0.00019	0.00069
2	2018-07-14 10:58:00	0.00019	0.00069
3	2018-07-14 10:59:00	1.56878	0.42069
4	2018-07-14 11:00:00	1.57893	0.43641
5	2018-07-14 11:01:00	1.57778	0.39967

Function to check for mismatches in arrays before plotting

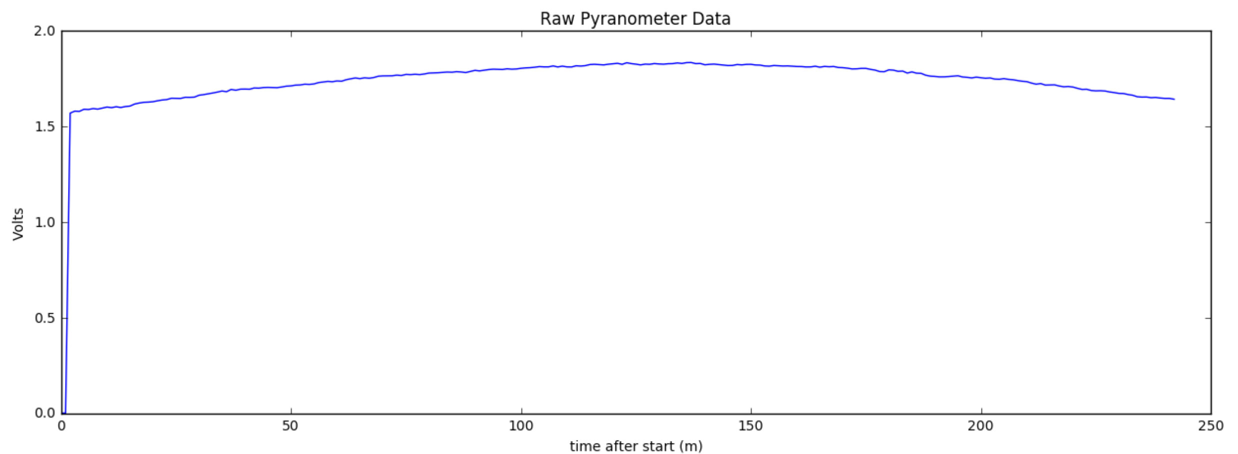
In [318]:
```
# Insert this function before plotting any graph:
def mismatch(x,y):
    lx = len(x)
    ly = len(y)
    if lx != ly:
        print "Error - mismatched array sizes:"
        print "X length = ",lx, "   Y length = ",ly
```

Plot the raw Weather data to check

(Note: we cannot calibrate the data yet because it is not in numerical form; it is a dataframe object.)

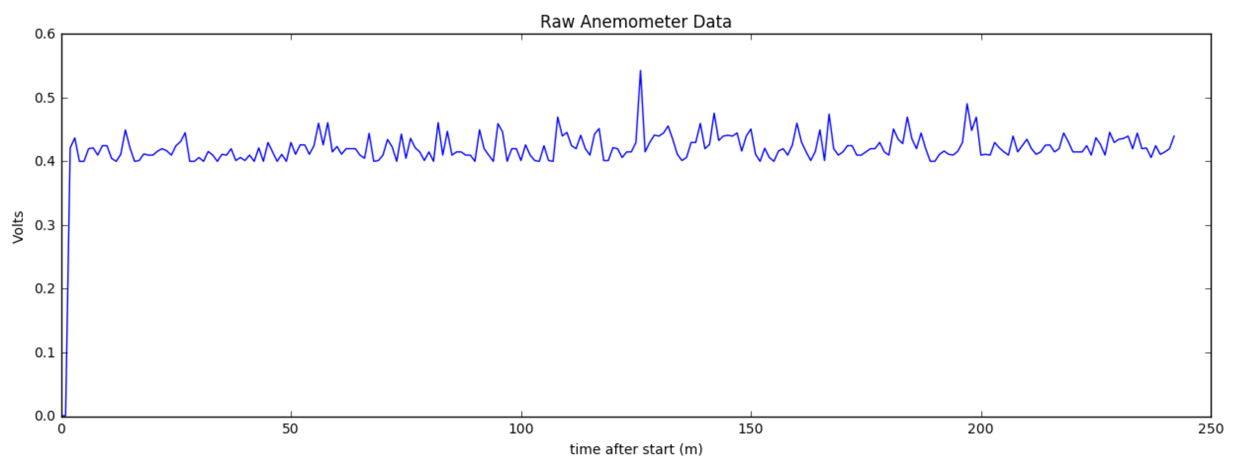
In [319]:

```
# Plot pyranometer raw data
x = np.linspace(0,tw-1,tw)
y = wf.loc[:,'Pyranometer']
y = y.astype(float)
fig, ax = plt.subplots()
figsize(15, 5);
mismatch(x,y)
line1, = ax.plot(x,y, label  ='Pyranometer')
plt.title('Raw Pyranometer Data')
plt.xlabel('time after start (m)')
plt.ylabel('Volts')
plt.show()
```



In [320]:

```
# Plot anemometer raw data
x = np.linspace(0,tw-1,tw)
y = wf.loc[:,'Anemometer']
y = y.astype(float)
fig, ax = plt.subplots()
figsize(15, 5);
line1, = ax.plot(x,y, label  ='Anemometer')
plt.title('Raw Anemometer Data')
plt.xlabel('time after start (m)')
plt.ylabel('Volts')
plt.show()
```



Read in the Therm data

In [321]:
```
# Read the Therm Excel file
#file = "Therm_" + filedate   # For 2016 data only
file = "Therm" + filedate
pathfile = "C:\Users\paul\Test\\" + file + ".xlsx"
print "Therm data file path: ", pathfile
xls_file = pd.ExcelFile(pathfile)
table = xls_file.parse(file)
therm = pd.DataFrame(table)
title = therm.columns[0]  # Use for plot title
trows = len(therm)    # number of rows of Therm data
# In future, do some regular expressions to extract the label from the data, but j
therm.describe()
therm.info()
therm.head(5)
```

```
Therm data file path:  C:\Users\paul\Test\Therm20180714.xlsx
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 245 entries, 0 to 244
Data columns (total 7 columns):
Plot Title: Therm    245 non-null object
Unnamed: 1           245 non-null object
Unnamed: 2           245 non-null object
Unnamed: 3           245 non-null object
Unnamed: 4           245 non-null object
Unnamed: 5           245 non-null object
Unnamed: 6           245 non-null object
dtypes: object(7)
memory usage: 13.5+ KB
```

Out[321]:

	Plot Title: Therm	Unnamed: 1	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6
0	#	Date Time, GMT-04:00	T-Type, °C (SEN S/N: 20104844, LBL: T1)	T-Type, °C (SEN S/N: 20104844, LBL: T2)	T-Type, °C (SEN S/N: 20104844, LBL: T3)	T-Type, °C (SEN S/N: 20104844, LBL: T4)	Temp, °C (SEN S/N: 20104844, LBL: T5)
1	1	2018-07-14 10:58:08	23.48	23.69	30.65	30.22	27.998
2	2	2018-07-14 10:58:11	23.48	23.69	30.65	30.22	27.998
3	3	2018-07-14 10:59:08	23.94	23.89	30.44	30.19	28.171
4	4	2018-07-14 11:00:08	23.94	25.3	30.72	30.43	28.295

Remove unnecessary columns and label the columns of Therm data

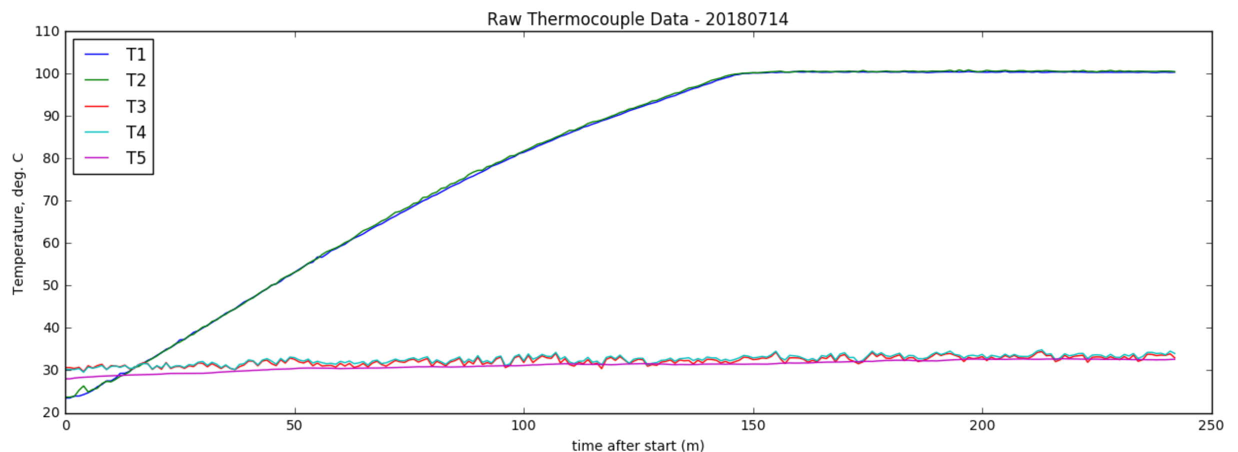
In [322]:
```
# Remove unneeded columns from the data and rename the columns
# This is important because these labels are used by pandas to select the columns
# The first row of valid data is = row1
# Rename the columns
tf = therm.iloc[row1:trows-row1, 1:7]
label4 = 'Time'   # Column labels
#label5 = '#'     #  An extra column of row numbers that was manually added to 20.
label5 = 'T1'
label6 = 'T2'
label7 = 'T3'
label8 = 'T4'
label9 = 'T5'
tf.columns=[label4,label5,label6,label7,label8,label9]   # Rename the columns
tt = len(tf)
tf.head(5)
```

Out[322]:

	Time	T1	T2	T3	T4	T5
1	2018-07-14 10:58:08	23.48	23.69	30.65	30.22	27.998
2	2018-07-14 10:58:11	23.48	23.69	30.65	30.22	27.998
3	2018-07-14 10:59:08	23.94	23.89	30.44	30.19	28.171
4	2018-07-14 11:00:08	23.94	25.3	30.72	30.43	28.295
5	2018-07-14 11:01:08	24.29	26.33	29.72	29.56	28.369

Plot the raw Therm data to check

In [323]:
```
# Plot the raw Therm data
x = np.linspace(0,tt-1,tt)
y = tf.loc[:,'T1']
y = y.astype(float)
fig, ax = plt.subplots()
figsize(15, 5);
mismatch(x,y)
line1, = ax.plot(x,y, label='T1')
y = tf.loc[:,'T2']
y = y.astype(float)
line2, = ax.plot(x,y, label='T2')
y = tf.loc[:,'T3']
y = y.astype(float)
line2, = ax.plot(x,y, label='T3')
y = tf.loc[:,'T4']
y = y.astype(float)
line2, = ax.plot(x,y, label='T4')
y = tf.loc[:,'T5']
y = y.astype(float)
line2, = ax.plot(x,y, label='T5')
plottitle = "Raw Thermocouple Data - " + str(testdate)
plt.title(plottitle)
plt.xlabel('time after start (m)')
plt.ylabel('Temperature, deg. C')
plt.legend(loc='best')
plt.show()
```



Align the Weather and Therm dataframes

In [324]:
```
# Compute the difference between two time values
import datetime as dt
from datetime import datetime
s1 = pd.Series(wf['Time'])  # Select the Time column of the dataframe and convert
s2 = pd.Series(tf['Time'])
s1 = pd.to_datetime(s1)     #  Convert the Series object values to datetime value
s2 = pd.to_datetime(s2)
# The index starts at row = row1, not 1.
startdate = s1[row1]
startdate = s1.dt.date[row1]
starthour = s1.dt.hour[row1]
nday = s1.dt.dayofyear[row1]
print "day no.",nday
print "start date = ",startdate  # These will be used later to calculate sun alti
hours1 = s1.dt.hour
hours2 = s2.dt.hour
m1 = s1.dt.minute.iloc[row1]       #  Select minutes from the datetime values
m2 = s2.dt.minute.iloc[row1]
h1 = hours1.iloc[row1]
h2 = hours2.iloc[row1]
t1 = 60*h1 + m1
t2 = 60*h2 + m2
diff = t2 - t1
lwf = len(wf)
ltf = len(tf)
diff = ltf - lwf
# Shift the data to align times and truncate the later excess rows, and
# truncate the later columns by the number of good rows in the earlier column min
# The series that starts later controls the start time.
wfs = wf
tfs = tf
starthour = s1.dt.hour[row1]
startmin = s1.dt.minute[row1]
if diff > 0:
    print "Therm data starts later than Weather data by ", abs(diff), " minute(s)
    wfs = wf.shift(-1*diff)
    starthour = s2.dt.hour[row1]
    startmin = s2.dt.minute[row1]
elif diff < 0:
    print "Weather data starts later than Therm data by ", abs(diff), " minute(s)
    tfs = tf.shift(-1*diff)
    starthour = s1.dt.hour[row1]
    startmin = s1.dt.minute[row1]
nrows = wrows-abs(diff) -1
wfs = wf.head(nrows)
tfs = tf.head(nrows)
print nrows, "rows of data"
```

```
day no. 195
start date =  2018-07-14
244 rows of data
```

Compute test times and limit to 10am - 2pm local solar time

In [325]:

```
# Compute test times
starthour1 = s1.dt.hour[row1]
startmin = s1.dt.minute[row1]
endhour1 = s1.dt.hour[nrows-row1]
endmin = s1.dt.minute[nrows-row1]
# Subtract 1 hour during daylight saving time (3/12 - 11/5 in US)
if nday >= 71 and nday < 309:
    starthour = starthour1 - 1
    endhour = endhour1 - 1
    print "Time index values have been reduced by 1 hour due to daylight time in t
else:
    starthour = starthour1
    endhour = endhour1
starttime = 60*starthour + startmin - 720 # minutes before noon local solar time
if endhour >= 12:
    endtime = 60*(endhour - 12) + endmin
elif endhour < 12:
    endtime = endhour + endmin
# Trim start and end times to comply with the standard (10am - 2pm solar time)
if starttime <-120:
    starttime = -120
if endtime > 120:
    endtime = 120
duration = endtime - starttime
print "Starhour1 =",starthour1, " Endhour1 = ", endhour1
print "Test started at ",starthour,":", startmin, " local solar time."
print "Test ended at ", endhour, ":", endmin, "local solar time."
print "Calculations start time = ", starttime, " minutes re. noon"
print "Calculations end time = ", endtime, " minutes re. noon"
print "Test duration = ", duration, " minutes."   # Duration is a maximum of 240 m
```

```
Time index values have been reduced by 1 hour due to daylight time in the US.
Starhour1 = 10  Endhour1 =  14
Test started at  9 : 58  local solar time.
Test ended at  13 : 59 local solar time.
Calculations start time =  -120  minutes re. noon
Calculations end time =  119  minutes re. noon
Test duration =  239  minutes.
```

Remove the time columns and add an index column

In [326]:

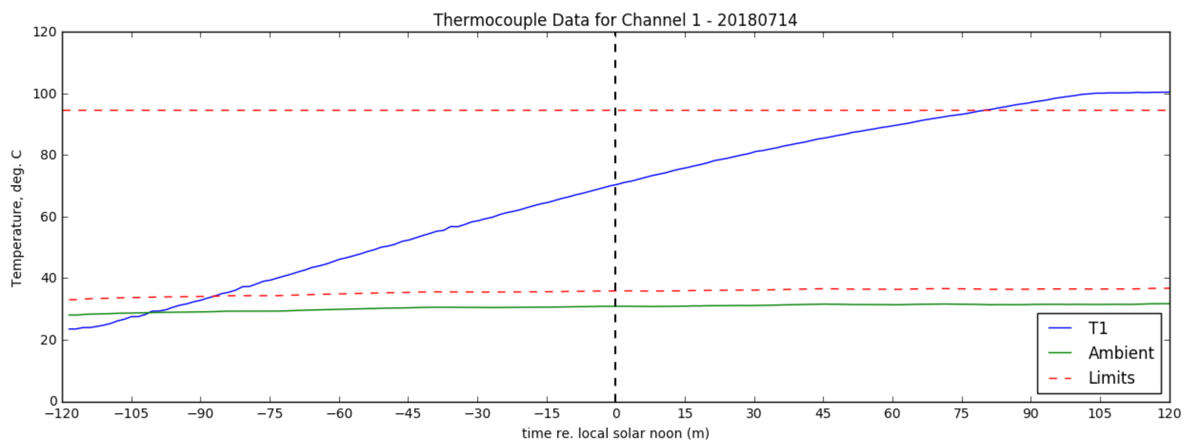
```
# Remove the time columns and add one index column.
wfst = wfs.iloc[:,[1,2]]
tfst = tfs.iloc[:,[1,2,3,4,5]]   # Note: column 1 is the extra set of row numbers
s = range(0,nrows)
mins = pd.DataFrame(s)  #  Index of data in minutes
```

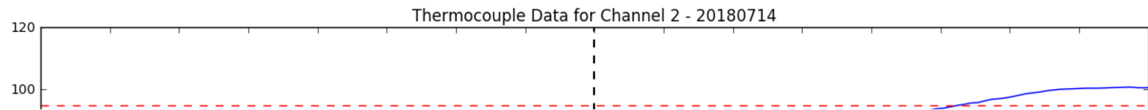
Concatenate Weather and Therm data into one frame

In [327]:
```
# Concatenate weather and therm data into one frame
rawdata = pd.concat([wfst, tfst], axis=1)    # Weather and Therm dataframes
rawdata = pd.concat([mins,rawdata], axis=1)  # Add a minutes column
nflag = [0]*nrows
flag = pd.DataFrame(nflag)    # Flag will be used to identify errors later on
rawdata = pd.concat([flag, rawdata],axis=1)
rawdata = rawdata.head(duration)
# Rename the columns
rawdata.columns=['Flag','Min','Pyranometer','Anemometer','T1','T2','T3','T4','T5'
```

Plot thermocouple data for each valid channel

In [328]:
```
x = np.linspace(starttime,duration-1,duration)
# Loop on channels T1 - T4
for i in range(0,4):
    if disregard[i] != 1:    # Skip channels that have no data
        stri = str(i+1)
        if i==0:
            y = rawdata.loc[:,'T1'].values
            tlabel = 'T1'
        if i==1:
            y = rawdata.loc[:,'T2'].values
            tlabel = 'T2'
        if i==2:
            y = rawdata.loc[:,'T3'].values
            tlabel = 'T3'
        if i==3:
            y = rawdata.loc[:,'T4'].values
            tlabel = 'T4'
        fig, ax = plt.subplots()
        figsize(15, 5);
        mismatch(x,y)
        line1, = ax.plot(x,y, label=tlabel)
        y = rawdata.loc[:,'T5'].values
        line2, = ax.plot(x,y, label='Ambient')
        # Plot lower bound of valid data (ambient + 5)
        y = y + 5
        line3, = ax.plot(x,y,label="Limits",linestyle="--")
        plottitle = "Thermocouple Data for Channel " +stri +" - "+ str(testdate)
        plt.title(plottitle)
        plt.xlabel('time re. local solar noon (m)')
        plt.ylabel('Temperature, deg. C')
        plt.legend(loc='best')
        plt.xlim(starttime+1,starttime+duration)
        xt = duration//15 +1
        #plt.xticks(np.linspace(starttime,120,xt,endpoint=True))
        plt.xticks(np.linspace(-120,120,17,endpoint=True))
        # Plot upper bound of valid data (about 95 degrees)
        tlimit = bpc - 5
        plt.plot([-120,120],[tlimit,tlimit],color="red",linestyle="--")
        plt.plot([0,0],[0,120], color ='black', linewidth=1.5, linestyle="--")
        plt.show()
```



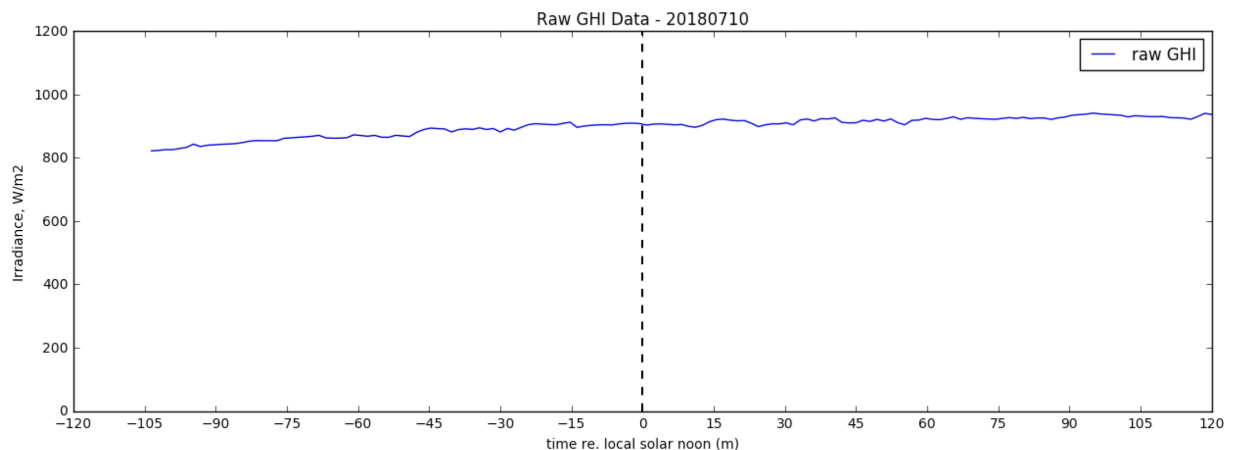


Calibrate pyranometer data

In [261]:
```
# Select the Pyranometer series from the dataframe and make into an array
ar = rawdata.loc[:,'Pyranometer'].values
ar = ar.astype(float64)  # Floating point array of data for each minute
rawghi = ar*psens       # Raw GHI in W/m2
```

Plot raw GHI data

In [263]:
```
x = np.linspace(starttime,duration-1,duration)
fig, ax = plt.subplots()
figsize(15, 5);
line2, = ax.plot(x,rawghi, label='raw GHI')
plottitle = "Raw GHI Data - " + str(testdate)
plt.title(plottitle)
plt.xlabel('time re. local solar noon (m)')
plt.ylabel('Irradiance, W/m2')
plt.legend(loc='best')
plt.xlim(starttime+1,starttime+duration)
xt = duration//15 +1
#plt.xticks(np.linspace(starttime,120,xt,endpoint=True))
plt.xticks(np.linspace(-120,120,17,endpoint=True))
plt.plot([0,0],[0,1200], color ='black', linewidth=1.5, linestyle="--")
plt.show()
```

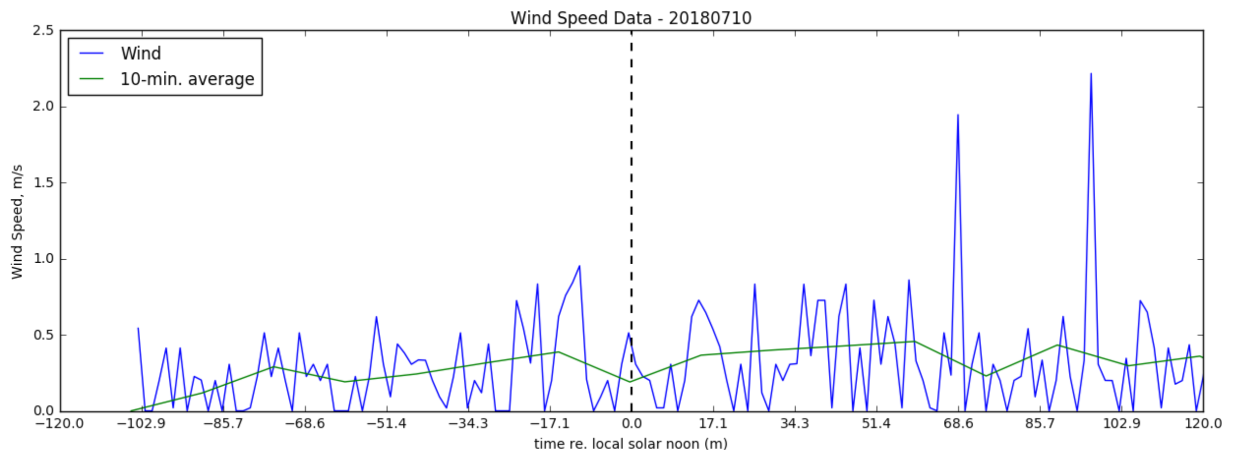


Calibrate wind speed data

In [264]:
```
# Calibrate wind speed data
ar = rawdata.loc[:,'Anemometer'].values
ar = ar.astype(float64)      # Floating point array of data for each minute
wind = (ar-aoffset)*asens    # Wind speed in m/s
for i in range(0,duration):
    if wind[i] < 0:
        wind[i] = 0.
```

Plot calibrated wind speed data and 10-minute averages

In [266]:
```
# Plot calibrated wind speed data
x = np.linspace(starttime,duration-1,duration)
blocks = duration//10    # No. of 10-minute intervals in data
blocks1 = blocks + 1
iduration = 10*blocks
z = [0]*blocks1
xz = [0]*blocks1
j = 0
for n in range(0,iduration,10):  # Loop over 10-minute blocks
    j = j + 1
    for m in range(0,10):
        p = n + m
        z[j] = mean(wind[p:p+10])
xz = np.linspace(starttime, duration, blocks1)
fig, ax = plt.subplots()
figsize(15, 5);
line1, = ax.plot(x,wind, label ='Wind')
line2, = ax.plot(xz,z,label='10-min. average')
plottitle = "Wind Speed Data - " + str(testdate)
plt.title(plottitle)
plt.xlabel('time re. local solar noon (m)')
plt.ylabel('Wind Speed, m/s')
plt.xlim(starttime+1,starttime+duration)
plt.ylim(0,2.5)
xt = duration//15 +1
#plt.xticks(np.linspace(starttime,120,xt,endpoint=True))
plt.xticks(np.linspace(-120,120,xt,endpoint=True))
plt.legend(loc='best')
plt.plot([0,0],[0,1200], color ='black', linewidth=1.5, linestyle="--")
plt.show()
```



Calculate sun declination angle on day of test

In [267]:
```
# calculate declination angle using Wikipedia model
degrad = np.pi/180.
declination = -23.44*np.cos(degrad*(360./365.24)*(nday + 10))
print "Solar declination on day ", nday, " is ",declination, "degrees."

# Calculation from handbook:
declination2 = 23.45*np.sin(degrad*(360*(284+nday)/365))
print "Check from handbook formula: ", declination2
```

```
Solar declination on day  191  is  22.2780152272 degrees.
Check from handbook formula:  22.3022753071
```

Function to calculate altitude of the sun

In [268]:
```
# Function to calculate sun altitude in degrees each minute
# This routine requires numpy and math
def alt(latitude, longitude, declination, hour_angle):
    degrad = np.pi/180.
    latitude_rad = degrad*latitude
    declination_rad = degrad*declination
    # Code adapted from function in Pysolar by Brandon Stafford under GPL
    # Appears to be accurate within less than 1 degree compared to USNO values
    first_term = np.cos(latitude_rad) * np.cos(declination_rad) * np.cos(degrad*(
    second_term = np.sin(latitude_rad) * np.sin(declination_rad)
    altitude = np.degrees(math.asin(first_term + second_term))
    return altitude
```

Function to model DHI (Diffuse Horizontal Irradiance)

In [269]:
```
# Function to model DHI in W/m2
def estdhi(altitude):
    degrad = np.pi/180.
    dhimodel = 100    # Ad hoc constant that sets noon DHI irradiance (range abou
    # On a very clear day in Tucson the noon DHI is about 60; on a hazy day it co
    # Therefore this gives a very crude estimate of DHI.
    estdhi = dhimodel*sqrt(sin(degrad*altitude))  # A 'flattened' sine function (
    return estdhi
```

Estimate DNI (Direct Normal Irradiance)

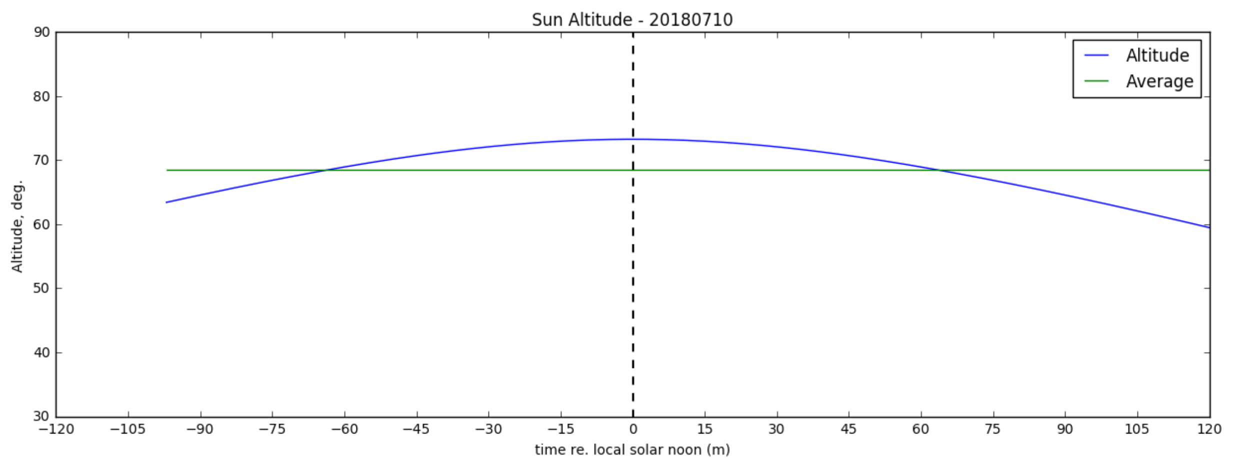
In [270]:

```
# Adjust for cosine response of pyranometer pointed vertically by finding sun alt
# Also subtract estimated DHI.
# Note: hour angle at test site may introduce a shift in time of solar noon.
# It may be a few minutes away from 1200.
# This can cause a slight asymmetry in the estimated DNI values relative to 1200.
# Time index 1 = start time of test; times in minutes
remhour = abs(longitude) % 15    # Longitude mod 15 degrees = remainder
mrem = int(60*remhour/15) # minutes before or after a 15-degree hour angle longit
if longitude > 0:
    mrem = -mrem  # Change for West vs. East longitude
# This is local solar time, irrespective of DST time shifts
latfract = (latitude - int(latitude))*60
longfract = (longitude - int(longitude))*60
print "latitude of test site = ", int(latitude), " deg. ", latfract, " minutes N"
print "longitude of test site = ", int(longitude), " deg. ", longfract, " minutes
altitude = [0]*duration
dhi = [0]*duration
dni = [0]*duration
mf = [0]*duration    # Solar hour angle in minutes re. solar noon at test site
zeny = [0]*duration
altsum = 0
for m in range(0,duration):
    mf[m] = starttime + float(m) + mrem  # Shift the hour angle for longitude of
    hour_angle = mf[m]/4  # moves 1 degree in 4 minutes starting from starttime
    altitude[m] = alt(latitude, longitude, declination, hour_angle)
    altsum = altsum + altitude[m]
    zen = np.sin(degrad*altitude[m])   # Cosine of the zenith angle of the sun
    zeny[m] = 50./zen  # for checking
    dhi[m] = estdhi(altitude[m])
    dni[m] = (rawghi[m]-dhi[m])/zen  # Compute DNI estimate
avgsun = altsum/(duration)  # Average sun angle in degrees during entire test per
dni = clip(dni,0,1500)  # Limit values to 0 - 1500 to limit outliers before plott
print "Remhour = ",remhour, " degrees  mrem = ", mrem, " minutes"
print "Average sun altitude during test was ", avgsun, " degrees re. horizontal."
print "Max. sun altitude was ", max(altitude)," degrees."
```

```
latitude of test site =  39  deg.  2.85378  minutes N
longitude of test site =  -77  deg.  -8.4693  minutes E
Remhour =  2.141155  degrees  mrem =  8  minutes
Average sun altitude during test was  68.4683720599  degrees re. horizontal.
Max. sun altitude was  73.2304522272  degrees.
```

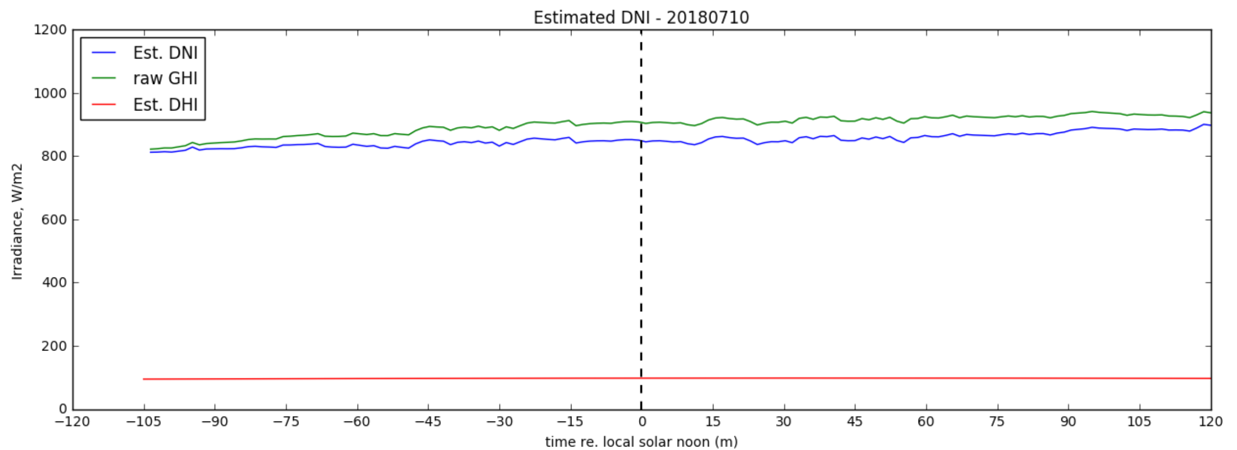
Plot altitudes of the sun

In [272]:
```
# Plot altitudes
avgsuny = [avgsun]*duration
fig, ax = plt.subplots()
figsize(15, 5);
x = mf    # local solar time
line1, = ax.plot(x,altitude, label ='Altitude')
line2, = ax.plot(x,avgsuny, label='Average')
#line3, = ax.plot(x,zeny, label="50/Cos_zenith")
plottitle = "Sun Altitude - " + str(testdate)
plt.title(plottitle)
plt.xlabel('time re. local solar noon (m)')
plt.ylabel('Altitude, deg.')
plt.legend(loc='best')
plt.xlim(starttime+1,starttime+duration)
plt.ylim(30,90)
xt = duration//15 +1
plt.xticks(np.linspace(-120,120,17,endpoint=True))
plt.plot([0,0],[0,1000], color ='black', linewidth=1.5, linestyle="--")
plt.show()
```



Plot Estimated DNI

In [274]:
```
# Plot estimated irradiance data
x = np.linspace(starttime,duration-1,duration)
fig, ax = plt.subplots()
figsize(15, 5);
line1, = ax.plot(x,dni, label ='Est. DNI')
line2, = ax.plot(x,rawghi, label='raw GHI')
line3, = ax.plot(x,dhi, label='Est. DHI')
plottitle = "Estimated DNI - " + str(testdate)
plt.title(plottitle)
plt.xlabel('time re. local solar noon (m)')
plt.ylabel('Irradiance, W/m2')
plt.legend(loc='best')
plt.xlim(starttime+1,starttime+duration)
plt.ylim(0,1200)
xt = duration//15 +1
#plt.xticks(np.linspace(starttime,120,xt,endpoint=True))
plt.xticks(np.linspace(-120,120,17,endpoint=True))
plt.plot([0,0],[0,1200], color ='black', linewidth=1.5, linestyle="--")
plt.show()
```



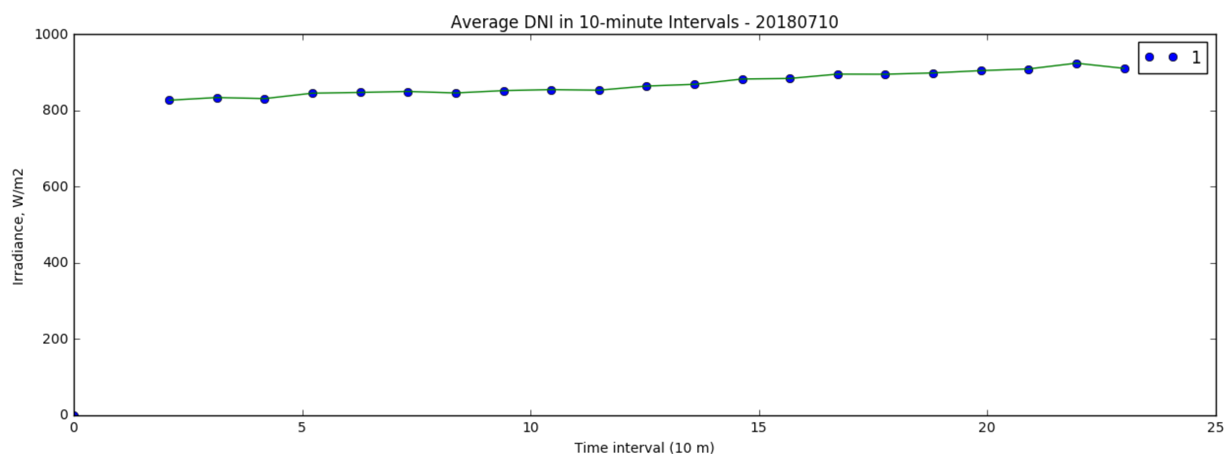
Average DNI data in 10-minute blocks and screen data

In [275]:
```
# Average DNI and wind data in 10-minute intervals and apply test criteria
blocks = duration//10    # No. of 10-minute intervals in data
blocks1 = blocks + 1
print "Total duration of test = ", duration, " minutes."
print "There are ", blocks, " full 10-minute intervals in the Weather data."
iduration = 10*blocks
tag = [0]*blocks1   # Error flags
flag = [0]*blocks1
dniavg = [0]*blocks1
# Average data in blocks of 10 minutes and check for violations of test criteria
print "Flags: "
print "Block      Reason for rejection"
j = 0
for n in range(0,iduration,10):  # Loop over 10-minute blocks
    j = j + 1
    kount = 0
    flag[j] = 0
    dnisum = 0.
    flagsum = 0
    if ptp(dni[n:n+10]) > 100:    # Test for DNI peak-to-peak variation in block
        flag[j] = 1
        print n,"        p-p DNI variation > 100"
    #maxd = max(dni[n:n+10])      # Alternative way of testing for peak-to-peak v
    #mind = min(dni[n:n+10])
    #if (maxd - mind) > 3.:
        #flag[j] = 1
    for m in range(0,10):        # Loop over minutes within a block (CHECK - 1 to
        p = n + m
        if dni[p] < 450:         # Test for DNI too low
            flag[j] = 1
            print n,"        DNI < 450"
        if dni[p] > 1100:        # Test for DNI too high
            flag[j] = 1
            print n,"        DNI > 1100"
        if mean(wind[p:p+10]) > 2.5:   # Test for high mean wind speed in 10-minu
                flag[j] = 1
                print n,"        windy"
        #if wind[p] > 2.5:        # Test for high wind speed in any minute of dat
            #flag[j] = 1    # This test is not used; mean within interval is used
                            # based on interpretation of standard
        else:
            kount = kount + 1
            dnisum = dnisum + float(dni[p])
        # end m loop (one block)
    #print j, flag[j]
    if flag[j] == 0:  # This block passed all tests: store averages in block
        dniavg[j] = float(dnisum/kount)  # Avg. DNI in each valid interval
# end n loop (blocks)
```

```
Total duration of test =  224  minutes.
There are  22  full 10-minute intervals in the Weather data.
Flags: 
Block      Reason for rejection
```

Plot averaged DNI data in 10-minute intervals

In [277]:
```
# Plot the averaged irradiance data
fig, ax = plt.subplots()
figsize(15, 5);
x = np.linspace(0,blocks1,blocks1)
y1 = dniavg
line1, = ax.plot(x,y1,'o',label='1')
line1, = ax.plot(x,y1,'-')
plottitle = "Average DNI in 10-minute Intervals - " + str(testdate)
plt.title(plottitle)
plt.xlabel('Time interval (10 m)')
plt.ylabel('Irradiance, W/m2')
plt.legend(loc='best')
plt.show()
print len(x)-1, "ten-minute samples."
```



22 ten-minute samples.

Define ambient temperature data

In [278]:
```
# Define ambient temperatures from channel 5 internal thermistor data
t5 = rawdata.loc[:,'T5'].values
t5 = t5.astype(float64)      # Floating point array of data for each minute
```

Average thermocouple channels in 10 minute intervals, compute temperature differences and screen data

In [279]:
```
# Average temperature difference data in 10-minute intervals and apply test criter
blocks = duration//10    # No. of 10-minute intervals in data
blocks1 = blocks + 1
print "Total duration of test = ", duration, " minutes."
print "There are ", blocks, " full 10-minute intervals in the data."
iduration = 10*blocks
tdiff = [0]*blocks1
comb = [[0]*blocks1 for k in range(4)]  # Temp. differences in 4 rows of blocks1 d
tag = [[0]*blocks1 for k in range(4)]   # Error flags in 4 rows of blocks1 data
# Loop on channels T1 - T4
for i in range(0,4):
    if disregard[i] != 1:    # Skip channels that have no data
        stri = str(i+1)
        if i==0:
            tdata = rawdata.loc[:,'T1'].values
        if i==1:
            tdata = rawdata.loc[:,'T2'].values
        if i==2:
            tdata = rawdata.loc[:,'T3'].values
        if i==3:
            tdata = rawdata.loc[:,'T4'].values
        tdiff = [0]*blocks1
        bad = [0]*blocks1
        # Average data in blocks of 10 minutes and check for violations of test cr
        j = 0
        for n in range(0,iduration,10):  # Loop over 10-minute blocks
            j = j + 1
            kount = 0
            bad[j] = 0
            ambsum = 0.
            avgsum = 0.
            for m in range(0,10):        # Loop over minutes within a block (CHECK
                p = n + m
                if t5[p] < 20.:          # Test for low ambient temperature
                    bad[j]= 1
                    #print n,"ambient < 20 C in block ",n
                if t5[p] > 35.:          # Test for high ambient temperature
                    bad[j] = 1
                    #print n,"ambient > 35 C"
                td1 = float(tdata[p])
                td2 = float(t5[p])
                if (td1 - td2) < 5.0:
                    bad[j] = 1
                    #print n,"temp. < 5 deg. C above ambient in block ",n
                if td1 > bpc-5.0:  # Test for pot temperature within 5 degrees of
                    if cv[i] == cvw:     # if load is water only
                        bad[j] = 1
                        #print n,"temp. > 95 deg. C"
                else:
                    kount = kount + 1
                    avgsum = avgsum + td1
                    ambsum = ambsum + td2
            # end m loop (one block)
            if bad[j] == 0:  # This block passed all tests: store averages in blo
                tdiff[j] = float(avgsum/kount) - float(ambsum/kount)  # temperatu
                tdiff[j] = round(tdiff[j],3)
```

```
            # end n loop (blocks)
        tag[i] = bad
    comb[i] = tdiff
    # end if (disregard channel)
# end i loop (channels)
print "Flags: "
for i in range(0,4):
    if disregard[i] != 1:    # Skip channels that have no data
        print "Channel ", str(i), ": "
        print tag[i]
        print comb[i]
```

```
Total duration of test =  224  minutes.
There are  22  full 10-minute intervals in the data.
Flags:
Channel  0 :
[0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 14.076, 20.917, 27.374, 33.764, 39.457, 44.883, 50.174, 55.103, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Channel  1 :
[0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 14.576, 21.379, 27.875, 33.983, 39.849, 45.145, 50.47, 55.352, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Find the range of contiguous samples

The standard requires data from a series of contiguous samples, with no breaks. Such breaks are indicated by flags = 1 in the screening critieria for DNI and temperatures. The code selects only the first set of contiguous samples in the data. There may be other valid 10-minute samples later on, but they cannot be used to calculate power.

In [280]:

```
# Find index of first valid block in both DNI and temperature data
for j in range(0,blocks1-1):
    lowerflag = flag[j]
    upperflag = flag[j+1]
    if lowerflag !=0 or upperflag != 0:
        firstflag = upperflag
    else:
        # First two contiguous samples of valid DNI data for this channel
        firstflag = j
        break
firsttag = [0]*4
first = [0]*4
for chan in range(0,4):
    if disregard[chan] != 1:    # Skip channels that have no data
        j = 0
        for j in range(0,blocks1-1):
            lowertag = tag[chan][j]
            uppertag = tag[chan][j+1]
            if lowertag !=0 or uppertag != 0:
                firsttag[chan] = uppertag
            else:
                # First two contiguous samples of valid temperature data for this
                firsttag[chan] = j
                break
        first[chan] = max(firstflag,firsttag[chan])
print "First valid index for each channel: ", first
```

```
First valid index for each channel:  [4, 4, 0, 0]
```

Find the last index of valid data in each channel and trim data array

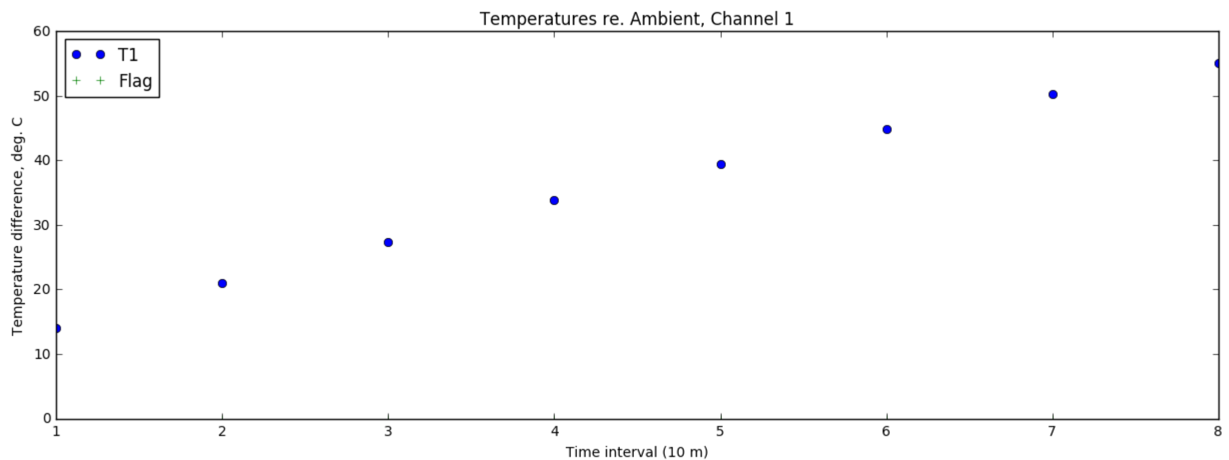
In [281]:

```
# Find last valid block in data:
goodones = [0]*4
las = [0]*4
last = [0]*4
ast = [0]*4
comb2 = [[0]*blocks1 for k in range(4)]  # Trimmed to valid range of blocks
tag2 = [[0]*blocks1 for k in range(4)]   # Trimmed
for chan in range(0,4):
    goodones[chan]=0
    if disregard[chan] != 1:    # Skip channels that have no data
        for j in range(first[chan],blocks1-1):
            lowerflag = flag[j]
            upperflag = flag[j+1]
            if lowerflag !=1 and upperflag != 0:
                # First two contiguous samples of valid data for this channel
                las[chan] = j
                break
            else:
                las[chan] = blocks1
        # print "las = ", las[chan]
        # Find last valid block in temperature data:
        # Average data in blocks of 10 minutes and check for violations of test c
        for j in range(first[chan],blocks1-1):
            lowertag = tag[chan][j]
            uppertag = tag[chan][j+1]
            if lowertag == 0 and uppertag == 1:
                # Flag jumps from 0 to 1
                break
        ast[chan] = j
    else:
        ast[chan] = blocks1
    last[chan] = min(las[chan],ast[chan])
    # Trim arrays to include only valid samples
    goodones[chan] = last[chan]-first[chan] +1
    if last[chan]==0:
        goodones[chan] = 0
    comb2[chan] = comb[chan][first[chan]:last[chan]+1]
    tag2[chan] = tag[chan][first[chan]:last[chan]+1]
print "Last valid index for each channel: ", last
print "Number of good samples in each channel: ",goodones
```

```
Last valid index for each channel:  [11, 11, 0, 0]
Number of good samples in each channel:  [8, 8, 0, 0]
```

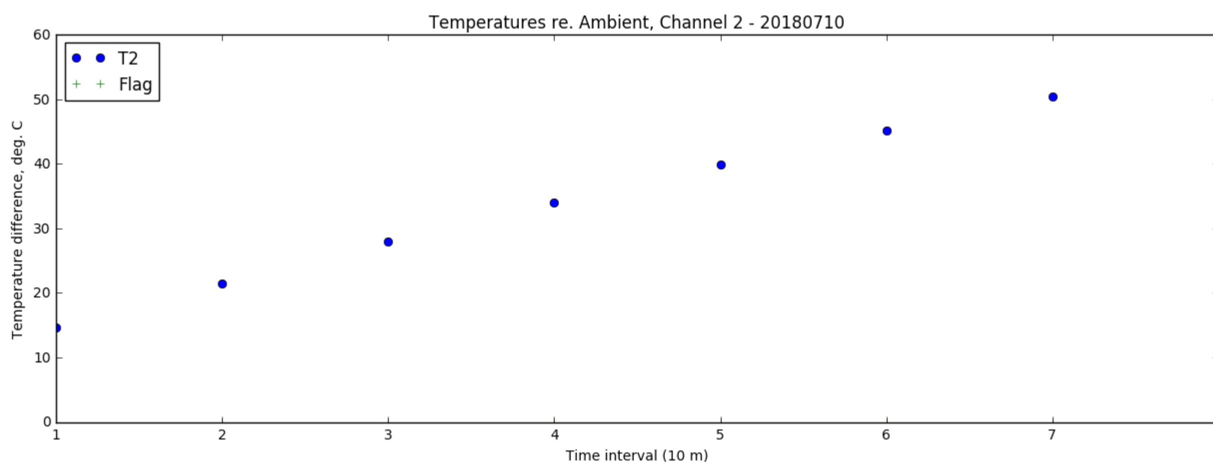
Plot the averaged temperature difference data

In [282]:
```
# Plot the averaged temperature data for channel 1
if disregard[0] != 1:    # Skip channels that have no data
    fig, ax = plt.subplots()
    figsize(15, 5);
    x = np.linspace(1,goodones[0],goodones[0])
    y = comb2[0]
    z = tag2[0]
    lx = len(x)
    ly = len(y)
    if lx != ly:
        print "Error - mismatched array sizes:"
        print "X length = ",lx, "   Y length = ",ly
    line1, = ax.plot(x,y,'o',label='T1')
    lz = len(z)
    if lx != lz:
        print "Error - mismatched array sizes:"
        print "X length = ",lx, "   Z length = ",lz
    line2, = ax.plot(x,z,'+',label='Flag')
    plt.title('Temperatures re. Ambient, Channel 1')
    plt.xlabel('Time interval (10 m)')
    plt.ylabel('Temperature difference, deg. C')
    plt.legend(loc='best')
    plt.show()
```



In [283]:

```
# Plot the averaged temperature differences for channel 2
if disregard[1] != 1:    # Skip channels that have no data
    fig, ax = plt.subplots()
    figsize(15, 5);
    x = np.linspace(1,goodones[1],goodones[1])
    y = comb2[1]
    z = tag2[1]
    lx = len(x)
    ly = len(y)
    if lx != ly:
        print "Error - mismatched array sizes:"
        print "X length = ",lx, "   Y length = ",ly
    line1, = ax.plot(x,y,'o',label='T2')
    line2, = ax.plot(x,z,'+',label='Flag')
    plottitle = "Temperatures re. Ambient, Channel 2 - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Time interval (10 m)')
    plt.ylabel('Temperature difference, deg. C')
    plt.legend(loc='best')
    plt.show()
```



In [197]:

```
# Plot the averaged temperature differences for channel 3
if disregard[2] != 1:    # Skip channels that have no data
    fig, ax = plt.subplots()
    figsize(15, 5);
    x = np.linspace(1,goodones[2],goodones[2])
    y = comb2[2]
    z = tag2[2]
    line1, = ax.plot(x,y,'o',label='T3')
    line2, = ax.plot(x,z,'+',label='Flag')
    plottitle = "Temperatures re. Ambient, Channel 3 - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Time interval (10 m)')
    plt.ylabel('Temperature difference, deg. C')
    plt.legend(loc='best')
    plt.show()
```

In [284]:
```
# Plot the averaged temperature differences for channel 4
if disregard[3] != 1:    # Skip channels that have no data
    fig, ax = plt.subplots()
    figsize(15, 5);
    x = np.linspace(1,goodones[3],goodones[3])
    y = comb2[3]
    z = tag2[3]
    line1, = ax.plot(x,y,'o',label='T4')
    line2, = ax.plot(x,z,'+',label='Flag')
    plottitle = "Temperatures re. Ambient, Channel 4 - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Time interval (10 m)')
    plt.ylabel('Temperature difference, deg. C')
    plt.legend(loc='best')
    plt.show()
```

Adjust the load mass to compensate for deviation from maximum intercept angle

In [285]:
```
# Increase the effective cooker load due to deviation in the
# average sun angle from the intercept angle of maximum reflector area (for each
corr = [0]*4
cmass = [0]*4
for chan in range(0,4):
    if disregard[chan] != 1:    # Skip channels that have no data
        stri = str(chan + 1)
        corr[chan] = 1/cos(degrad*(abs(intercept[chan] - avgsun)))
        cmass[chan] = corr[chan] * mass[chan]
        rcorr = round(corr[chan],3)
        print "Load for channel ",stri, " has been increased by the factor ", rco
```

```
Load for channel  1  has been increased by the factor  1.011  to compensate for
deviation from maximum intercept angle.
Load for channel  2  has been increased by the factor  1.011  to compensate for
deviation from maximum intercept angle.
```

Compute power for each time interval and channel

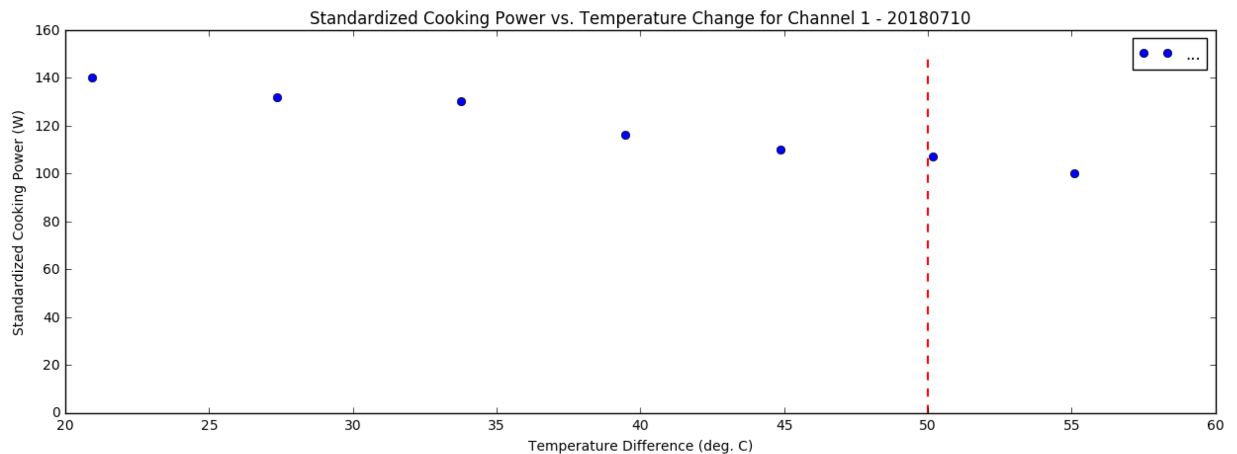
In [286]:
```
# Compute power for each time interval for ALL channels
inpwr = [0]*4
stdpwr = [[0]*blocks1 for k in range(4)]
k = -1
for chan in range(0,4):
    k = k+1
    if disregard[chan] != 1:    # Skip channels that have no data
        #print "Channel: ",chan," valid temp. samples: ", goodones[chan]
        #print "index, comb2, dniavg, pwrn:"
        gain = [0]*blocks1
        pwr = [0]*blocks1
        pwrn = [0]*goodones[chan]
        thiscooker = cooker[chan]        # Cooker name
        thisarea = area[chan]            # Maximum reflector area at intercept an
        #thisarea = area[chan]/corr[chan] # Reflector area reduced by deviation f
        thismass = cmass[chan] # Load mass, kg (corrected for intercept angle)
        thiscv = cv[chan]                # Heat capacity of load, J/g deg. C
        thisnotes = notes[chan]          # Notes to be displayed in report
        inpwr[chan] = 700.*thisarea      # solar input power in W given 700 W/m2
        for j in range(1,goodones[chan],1): # note the 1
            gain[j] = comb2[chan][j] - comb2[chan][j-1]
            pwr[j] = gain[j]*thismass*thiscv / 600.  # Cooking power for interval
            # Standardize output power to 700 W/m2 irradiance:
            denom = dniavg[first[chan]+j]
            pwrn[j] = round(700.*pwr[j]/denom, 3)
            if pwrn[j] < 0:
                pwrn[j] = 0
            #print j, comb2[chan][j],round(denom),pwrn[j]
            stdpwr[k] = pwrn[1:goodones[chan]]
        #print stdpwr[k]
for k in range(0,4):
    print "k = ",k, " stdpwr at end:",stdpwr[k]
print inpwr
```

```
k =  0  stdpwr at end: [139.982, 131.822, 130.065, 116.408, 110.154, 107.075, 9
9.935]
k =  1  stdpwr at end: [139.205, 132.618, 124.325, 119.946, 107.515, 107.763, 9
8.982]
k =  2  stdpwr at end: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
k =  3  stdpwr at end: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
[350.0, 350.0, 0, 0]
```

Plot Standardized Power vs. Temperature Differences for Channel 1

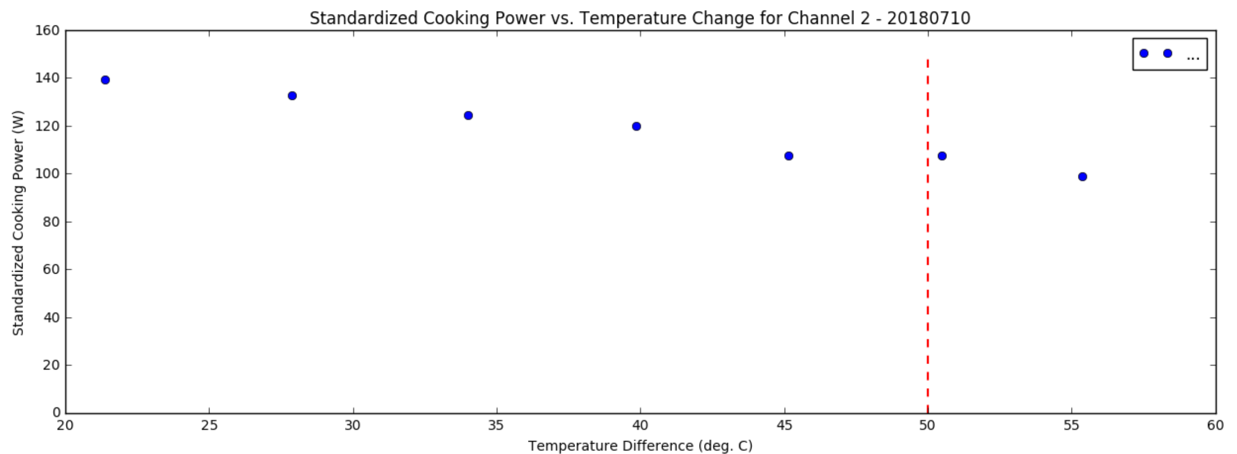
In [289]:

```
# Plot power vs. temperature difference for channel 1
chan = 0
if disregard[chan] != 1:    # Skip channels that have no data
    x = comb2[chan][1:goodones[chan]]
    y = stdpwr[chan]
    mismatch(x,y)
    fig, ax = plt.subplots()
    figsize(15, 5);
    line1, = ax.plot(x,y, 'o',label='...')
    plottitle = "Standardized Cooking Power vs. Temperature Change for Channel "
    plottitle = plottitle + str(chan+1) + " - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    plt.legend(loc='best')
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    plt.show()
else:
    stri = str(chan + 1)
    print "Channel ", stri, " was not measured."
```



Plot Standardized Power vs. Temperature Differences for Channel 2

In [290]:
```
# Plot power vs. temperature difference for channel 2
chan = 1
if disregard[chan] != 1:    # Skip channels that have no data
    x = comb2[chan][1:goodones[chan]]
    y = stdpwr[chan]
    mismatch(x,y)
    fig, ax = plt.subplots()
    figsize(15, 5);
    line1, = ax.plot(x,y, 'o',label='...')
    plottitle = "Standardized Cooking Power vs. Temperature Change for Channel "
    plottitle = plottitle + str(chan+1) + " - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    plt.legend(loc='best')
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    plt.show()
else:
    stri = str(chan + 1)
    print "Channel ", stri, " was not measured."
```



Plot Standardized Power vs. Temperature Differences for Channel 3

In [292]:
```
# Plot power vs. temperature difference for channel 3
chan = 2
if disregard[chan] != 1:    # Skip channels that have no data
    x = comb2[chan][1:goodones[chan]]
    y = stdpwr[chan]
    mismatch(x,y)
    fig, ax = plt.subplots()
    figsize(15, 5);
    line1, = ax.plot(x,y, 'o',label='...')
    plottitle = "Standardized Cooking Power vs. Temperature Change for Channel "
    plottitle = plottitle + str(chan+1) + " - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    plt.legend(loc='best')
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    plt.show()
else:
    stri = str(chan + 1)
    print "Channel ", stri, " was not measured."
```

```
Channel  3  was not measured.
```

Plot Standardized Power vs. Temperature Differences for Channel 4

In [293]:
```
# Plot power vs. temperature difference for channel 4
chan = 3
if disregard[chan] != 1:    # Skip channels that have no data
    x = comb2[chan][1:goodones[chan]]
    y = stdpwr[chan]
    mismatch(x,y)
    fig, ax = plt.subplots()
    figsize(15, 5);
    line1, = ax.plot(x,y, 'o',label='...')
    plottitle = "Standardized Cooking Power vs. Temperature Change for Channel "
    plottitle = plottitle + str(chan+1) + " - " + str(testdate)
    plt.title(plottitle)
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    plt.legend(loc='best')
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    plt.show()
else:
    stri = str(chan + 1)
    print "Channel ", stri, " was not measured."
```

```
Channel  4  was not measured.
```

Linear Regression for all channels measured

In [294]:
```
# Linear regression
power1 = [0]*4
power2 = [0]*4
samples = [0]*4
r2 = [0.]*4
rr = [0.]*4
efficiency2 = [0]*4
print "input power:",inpwr
for chan in range(0,4):
    stri = str(chan + 1)
    if disregard[chan] !=1:
        x = comb2[chan][1:goodones[chan]]  # Remove the first temp. diff. value
        # because there will always be one less diff. than power values since
        # power is based on the difference between 2 temp. diff. values.
        n = last[chan] - first[chan]  # Number of valid samples
        samples[chan] = n
        rr[chan] = [0.]*n
        y = stdpwr[chan]
        mismatch(x,y)
        # Screen final data for various problems:
        print ""
        print "Channel ",stri, " number of samples = ",n
        mtdiff = max(comb2[chan])
        print "Maximum temperature difference = ", mtdiff
        # Exclude data if sufficient heat gain has not occurred within the set of
        # This parameter is not explicit in the standard but may be much less tha
        if mtdiff < 10:
            print "Results excluded; max. temperature gain is less than ",mtdiff,
        else:
            if n < 2:
                print "Cannot calculate regression with only 1 valid sample."
            else:
                if n < 3:
                    print "Warning -- cannot calculate standard error or r^2 with
                else:
                    # Linear regression using stats.linregress
                    # These values will be stored in the Excel file
                    (slope,inter,r,tt,stderr)=stats.linregress(x,y)
                    for k in range(0,n):
                        rr[chan][k] = slope*x[k] + inter
                        #print rr[chan][k]
                    r2[chan] = r**2
                    print('Linear regression using stats.linregress:')
                    print('regression: a=%.2f b=%.2f, r=%.3f, std error= %.3f' %

                    # Solve for y at x=50:
                    power2[chan] = slope*50. + inter
                    efficiency2[chan] = 100.*power2[chan]/inpwr[chan]
                    print "Power at 50 deg. C = ", round(power2[chan],1), "Watts,
    else:
        print ""
        print "Channel ", stri, "was not measured."
```

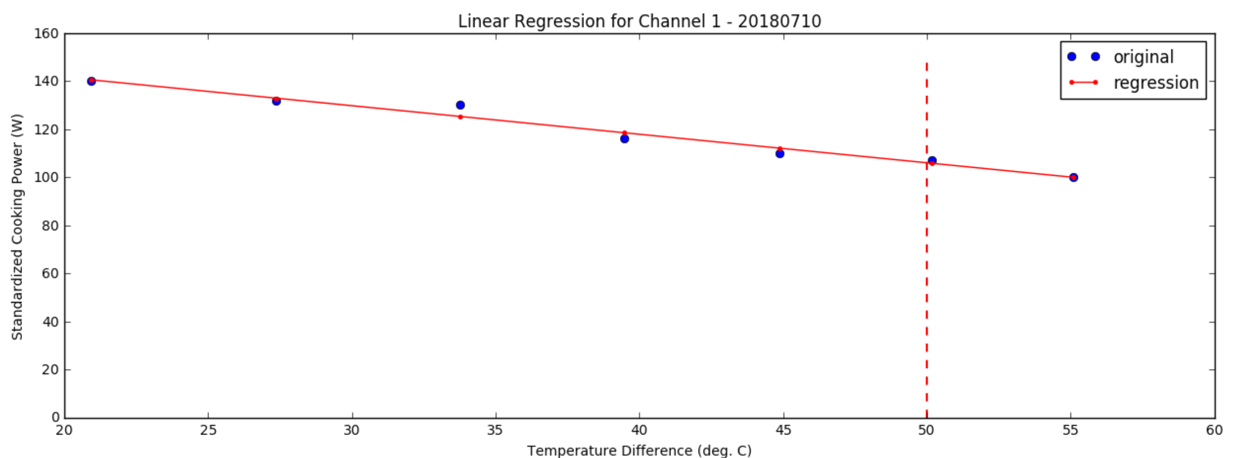
```
input power: [350.0, 350.0, 0, 0]
```

```
Channel  1  number of samples =  7
Maximum temperature difference =  55.103
Linear regression using stats.linregress:
regression: a=-1.19 b=165.47, r=-0.987, std error= 0.087 , r-squared = 0.974
Power at 50 deg. C =  106.1 Watts, Energy efficiency =  30.3 %

Channel  2  number of samples =  7
Maximum temperature difference =  55.352
Linear regression using stats.linregress:
regression: a=-1.18 b=164.90, r=-0.989, std error= 0.079 , r-squared = 0.978
Power at 50 deg. C =  105.8 Watts, Energy efficiency =  30.2 %

Channel  3 was not measured.

Channel  4 was not measured.
```

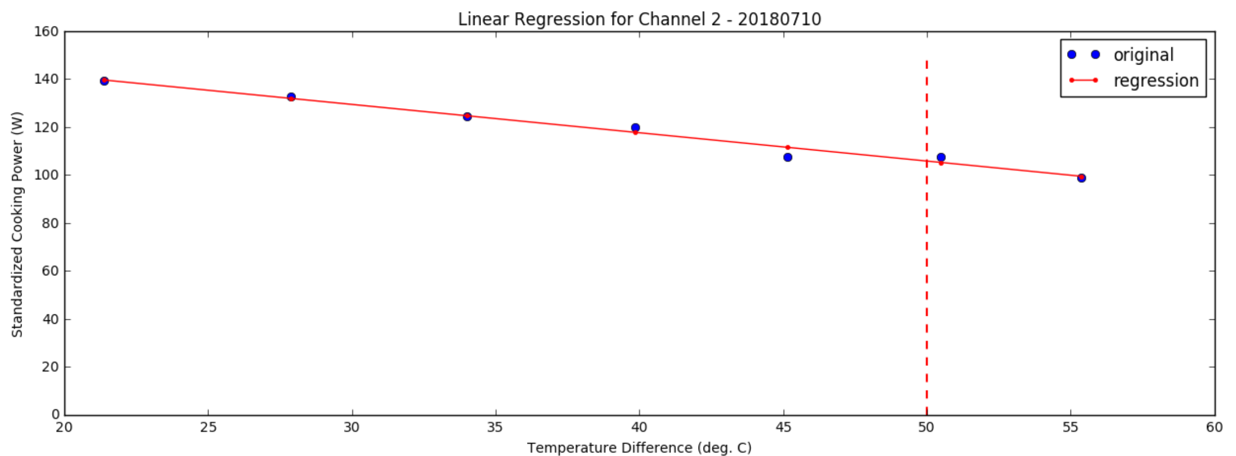
Plot final regression lines

In [307]:

```
chan = 0
stri = str(chan + 1)
if disregard[chan] !=1:
    #matplotlib plotting
    plottitle = 'Linear Regression for Channel ' + stri + " - " + str(testdate)
    plt.title(plottitle)
    x = comb2[chan][1:goodones[chan]]  # See note in previous block
    mismatch(x,stdpwr[chan])
    plot(x,stdpwr[chan],'o')
    mismatch(x,rr[chan])
    plot(x,rr[chan],'r.-')
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    legend(['original','regression'])
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    show()
```



In [308]:
```
chan = 1
stri = str(chan + 1)
if disregard[chan] !=1:
    #matplotlib plotting
    plottitle = 'Linear Regression for Channel ' + stri + " - " + str(testdate)
    plt.title(plottitle)
    x = comb2[chan][1:goodones[chan]]  # See note in previous block
    mismatch(x,stdpwr[chan])
    plot(x,stdpwr[chan],'o')
    mismatch(x,rr[chan])
    plot(x,rr[chan],'r.-')
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    legend(['original','regression'])
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    show()
```



In [309]:
```
chan = 2
stri = str(chan + 1)
if disregard[chan] !=1:
    #matplotlib plotting
    plottitle = 'Linear Regression for Channel ' + stri + " - " + str(testdate)
    plt.title(plottitle)
    x = comb2[chan][1:goodones[chan]]  # See note in previous block
    mismatch(x,stdpwr[chan])
    plot(x,stdpwr[chan],'o')
    mismatch(x,rr[chan])
    plot(x,rr[chan],'r.-')
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    legend(['original','regression'])
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    show()
```

In [310]:
```
chan = 3
stri = str(chan + 1)
if disregard[chan] !=1:
    #matplotlib plotting
    plottitle = 'Linear Regression for Channel ' + stri + " - " + str(testdate)
    plt.title(plottitle)
    x = comb2[chan][1:goodones[chan]]  # See note in previous block
    mismatch(x,stdpwr[chan])
    plot(x,stdpwr[chan],'o')
    mismatch(x,rr[chan])
    plot(x,rr[chan],'r.-')
    plt.xlabel('Temperature Difference (deg. C)')
    plt.ylabel('Standardized Cooking Power (W)')
    legend(['original','regression'])
    plt.plot([50,50],[0,150], color ='red', linewidth=1.5, linestyle="--")
    show()
```

```
It is necessary to store the regression data at this point so that data
measured on another day may be combined with these data.  The standard requires
at least three sets of data to be combined, and then a final regression is to
be performed on these values to arrive at a final power number.
```

Store regression data in Excel files (optional)

In [311]:
```
print "Store data?  If so, continue."
```

```
Store data?  If so, continue.
```

In [312]:
```
# Write the valid data to Excel files called "Power" + date + .xslx
# Files will be stored where the initial files were read in.
# If a file with the same name exists, it will not overwrite.
for chan in range(0,4):
    stri = str(chan + 1)
    if disregard[chan] !=1:
        n = len(stdpwr[chan])
        # Screen final data for various problems:
        print ""
        print 'Channel ',stri, ' number of samples = ',n
        # Exclude data if sufficient heat gain has not occurred within the set of
        # This parameter is not explicit in the standard but may be much less thai
        mtdiff = max(comb2[chan])
        if mtdiff < 10:
            print "Results not stored; max. temperature gain is less than ",mtdif
        else:
            if n < 2:
                print "Cannot calculate regression with only 1 valid sample."
            else:
                if n < 3:
                    print "Cannot calculate standard error or r^2 with only 2 sam
                else:
                    # Write the valid data to Excel files
                    powerfile = str(path + "Power_" + filedate + "_" + stri + ".x
                    workbook = xlsxwriter.Workbook(powerfile)
                    worksheet = workbook.add_worksheet()
                    worksheet.set_column(0,0,20)
                    worksheet.set_column(1,1,40)
                    worksheet.write('A1','date')
                    worksheet.write('B1', filedate)
                    worksheet.write('A2','location')
                    worksheet.write('B2', location)
                    worksheet.write('A3', 'technician')
                    worksheet.write('B3', v[2])
                    worksheet.write('A4', 'latitude deg.')
                    worksheet.write('B4', latitude)
                    worksheet.write('A5', 'longitude deg.')
                    worksheet.write('B5', longitude)
                    worksheet.write('A6', 'elevation, m')
                    worksheet.write('B6', elevation)
                    worksheet.write('A7', 'pdesc')
                    worksheet.write('B7', v[6])
                    worksheet.write('A8', 'poffset')
                    worksheet.write('B8', poffset)
                    worksheet.write('A9', 'psens')
                    worksheet.write('B9', psens)
                    worksheet.write('A10', 'adesc')
                    worksheet.write('B10', v[9])
                    worksheet.write('A11', 'aoffset')
                    worksheet.write('B11', aoffset)
                    worksheet.write('A12', 'asens')
                    worksheet.write('B12', asens)
                    worksheet.write('A13', 'channel')
                    worksheet.write('B13', chan)
                    worksheet.write('A14', 'label')
                    worksheet.write('B14', label[chan])
```

```
                    worksheet.write('A15', 'test item')
                    worksheet.write('B15', cooker[chan])
                    worksheet.write('A16', 'intercept alt. deg.')
                    worksheet.write('B16', intercept[chan])
                    worksheet.write('A17', 'area')
                    worksheet.write('B17', area[chan])
                    worksheet.write('A18', 'mass')
                    worksheet.write('B18', mass[chan])
                    worksheet.write('A19', 'cv')
                    worksheet.write('B19', cv[chan])
                    worksheet.write('A20', 'notes')
                    worksheet.write('B20', v[40])
                    worksheet.write('A21', 'Power @ 50 deg. diff.')
                    worksheet.write('B21', round(power2[chan],1))
                    worksheet.write('A22', 'r-squared')
                    worksheet.write('B22', round(r2[chan],2))
                    worksheet.write('A23', 'Efficiency')
                    worksheet.write('B23', round(efficiency2[chan],1))
                    worksheet.write('A24', 'samples')
                    worksheet.write('B24', samples[chan])
                    #  Write pairs of temp. diff., power samples
                    for n in range(0,samples[chan]):
                        x = comb2[chan][n]
                        y = stdpwr[chan][n]
                        worksheet.write(n+24,0,x)
                        worksheet.write(n+24,1,y)
                    workbook.close()
                    print "Valid data data stored in file " + powerfile
    else:
        print "Channel ", stri, "was disregarded."
```

```
Channel  1  number of samples =  7
Valid data data stored in file C:\Users\paul\Test\Power_20180710_1.xlsx

Channel  2  number of samples =  7
Valid data data stored in file C:\Users\paul\Test\Power_20180710_2.xlsx
Channel  3 was disregarded.
Channel  4 was disregarded.
```

Processing done!

In [ ]: