APPENDIX B - Python Scripts

Python script for Waves

```
#!/usr/bin/env python

import argparse
import os
import pandas as pd
import sys
import scipy.constants as constants
import math
import xarray as xr
import numpy as np
from datetime import datetime, timedelta

#metocean libraries
# http://github.com/metocean/wavespectra/
from wavespectra.specarray import SpecArray
from wavespectra.specdataset import SpecDataset

#plot
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='take a NOAA buoy data
and water depth and tells you what happens if oil spills near it')
    parser.add_argument('--file', '-f', required=True, help="buoy to
read")
    parser.add_argument('--depth', '-d', required=True, help="buoy depth")
    parser.add_argument('--output_dir', '-o', required=True)
    parser.add_argument('--buoy_name', '-b', required=True)
    args = parser.parse_args()

    #parse year off file
    buoy_num =
os.path.splitext(os.path.basename(args.file))[0].split('h')[0]
    year = os.path.splitext(os.path.basename(args.file))[0].split('h')[1]

    csv = pd.read_csv(args.file, header=[0,1])
    csv.columns = csv.columns.droplevel(-1)
    # create proper time format with timezone
```

```
    csv['datetime'] = pd.to_datetime(pd.DataFrame({
        'year': csv['#YY']
        ,'month': csv['MM']
        , 'day': csv['DD']
        , 'hour': csv['hh']
        , 'minute': csv['mm']
    })).dt.tz_localize('UTC')

    # compute the interval each sample covers
    interval_hour = 1/48 # default 30 minutes in fractional days
    if len(csv) > 1:
        # if we have more than one sample, we can figure out the interval
        intervals = (csv['datetime'] - csv['datetime'].shift())
        #interval = (csv['datetime'] - csv['datetime'].shift())[1]
        intervals = intervals[intervals > timedelta(minutes=0)]
        interval = intervals.dt.floor('T').value_counts().index[0]
        interval_hour = interval.total_seconds() / (60*60*24)

    # remove rows with measurement errors for WSPD and WVHT and DPD
    orig_csv_len = len(csv)
    csv = csv.query('WVHT != 99 and WSPD != 99 and DPD != 99')
    csv_len = len(csv)

    #add fields to csv needed for metocean
    csv['freq'] = 1/csv['DPD']
    csv['efth'] = 0
    csv.reset_index()
    csv = csv.set_index('freq')

    #convert to x array to get metocean helper functions
    x = csv.to_xarray()
    csv = csv.reset_index()

    #compute wave celerity, the propagation speed of a wave
    wave_celerity = x.spec.celerity(depth=float(args.depth), freq=x.freq)
    csv['wave_celerity'] = wave_celerity

    # write full data table
    output_prefix = os.path.splitext(os.path.basename(args.file))[0]
    output_csv = os.path.join(args.output_dir, 'csv',
"{}.csv".format(output_prefix))
    csv.to_csv(output_csv)

    # ******** make plot
    # plt.style.use('dark_background')
```

```
    # compute wave steepness for nuka
    csv['WVSTEEP'] = csv['WVHT'] / (constants.g * csv['DPD']**2)

    # compute nuka classifications
    def nuka_impaired(r):
        if r['WVSTEEP'] <= 0.0025 and r['WVHT'] >= 1.2:
            return 1
        elif r['WVSTEEP'] > 0.0025 and r['WVHT'] >= 0.9:
            return 1
        return 0
    csv['nuka_impaired'] = csv.apply(nuka_impaired, axis=1)

    def nuka_impossible(r):
        # impaired conditions become impossible in high windspeed
        if r['WVSTEEP'] <= 0.0025 and r['WVHT'] >= 1.2 and r['WSPD'] >=
10:
            return 1
        elif r['WVSTEEP'] > 0.0025 and r['WVHT'] >= 0.9 and r['WSPD'] >=
10:
            return 1

        #always impossible conditions
        if r['WVSTEEP'] <= 0.0025 and r['WVHT'] >= 2.4:
            return 1
        elif r['WVSTEEP'] >= 0.0025 and r['WVHT'] >= 1.8:
            return 1
    csv['nuka_impossible'] = csv.apply(nuka_impossible, axis=1)

    nuka_impaired_df = csv.query('nuka_impaired == 1')
    nuka_impossible_df = csv.query('nuka_impossible == 1')
    plots = [
        ("75% reduction in boom performance due to wave height >= 2 m
[Fingas]", csv.query('WVHT >= 4') )
        ,( "75% reduction in boom performance due to wind speed >= 4 m/s
[Fingas]", csv.query('WSPD >= 4') )
        ,( "Wind Speed > 9.26 m/s Failure [Tedeschi]", csv.query('WSPD >=
9.26') )
        #,( "NUKA Response Impaired", csv.query('nuka_impaired == 1') )
        #,( "NUKA Response Impossible", csv.query('nuka_impossible == 1')
)
    ]

    num_plots = len(plots)+1+1 #1 for data present, 2 for nuka
    fig, ax = plt.subplots(num_plots,1, sharex=True)
```

```
    style = {
        "figure.subplot.top": 0.0 #0.985
        ,"figure.subplot.left": 0.0
        ,"figure.subplot.right": 0.999
        ,"figure.subplot.bottom": 0.0#0.15
        ,'axes.titlesize': 10
    }
    for (key,value) in style.iteritems():
        matplotlib.rcParams[key] = value

    #colors to assign
    color_iter = plt.rcParams['axes.prop_cycle']
    colors = [p['color'] for p in color_iter]

    #render charts
    # passing a negative width will align the bars to right edge, which is
what
    # the NOAA samples represent
    td = datetime(int(year)+1, 1, 1) - datetime(int( year ), 1, 1)
    total_time_intervals = td.total_seconds() / interval.total_seconds()
    pct_present = round(len(csv['datetime']) / float(total_time_intervals)
* 100, 2)
    measurement_days =  len(csv['datetime'].dt.floor('d').value_counts())

    dp = "Data Present (Samples: {}, {}% of the year, {} days with at
least one)".format(
        len(csv['datetime']), pct_present, measurement_days)
    ax[0].set_title(dp)
    ax[0].bar(csv['datetime'].values, 1, width=-interval_hour,
align='edge', color=colors[0])

    # want to show for how many days the threshold is exceeded for more
than 2 hours total
    hour_2_threshold = timedelta(hours=2).total_seconds() /
interval.total_seconds()
    def days_above_2_hour_threshold(threshold_df):
        group_by_days =
threshold_df['datetime'].dt.floor('d').value_counts()
        days_over_2_hours = len(group_by_days[group_by_days >
hour_2_threshold])
        return days_over_2_hours

    for i,(name,data) in enumerate(plots):
        pct = round((len(data) / float(len(csv))) * 100,2)
```

```
        full_name = "{} ({}% {} days)".format(name, pct,
days_above_2_hour_threshold(data))
        ax[i+1].set_title(full_name)
        ax[i+1].bar(data['datetime'].values, 1, width=interval_hour,
align='edge', color=colors[i+1])

    impaired_pct = round((len(nuka_impaired_df) / float(len(csv))) *
100,2)
    impaired_days = days_above_2_hour_threshold(nuka_impaired_df)
    impossible_pct = round((len(nuka_impossible_df) / float(len(csv))) *
100,2)
    impossible_days = days_above_2_hour_threshold(nuka_impossible_df)
    ax[i+2].set_title("Nuka Response Impaired (yellow) ({}% {} days) /
Impossible (red) ({}% {} days)".format(
        impaired_pct, impaired_days, impossible_pct, impossible_days))
    ax[i+2].bar(nuka_impaired_df['datetime'].values, 1,
width=-interval_hour, align='edge', color='#FFBA2A')
    ax[i+2].bar(nuka_impossible_df['datetime'].values, 1,
width=-interval_hour, align='edge', color='#FF320D')

    plt.axis('tight')
    fig.suptitle("Buoy {}: {} {}".format(buoy_num, args.buoy_name, year),
fontsize=16)

    # format each axis
    for a in ax:
        a.xaxis_date('UTC')

a.xaxis.set_major_locator(mdates.MonthLocator(bymonth=range(1,13)))
        a.xaxis.set_major_formatter(mdates.DateFormatter('%Y %b'))
        a.xaxis.set_minor_locator(mdates.DayLocator(bymonthday=[1,15]))
        a.xaxis.set_minor_formatter(mdates.DateFormatter('\n%d'))
        a.yaxis.set_visible(0)
        # xlim needs to move one interval back to get the intervals
properly aligned
        a.set_xlim([ csv['datetime'].values[0]-interval,
csv['datetime'].values[-1] ])
        a.set_ylim([ 0, 1 ])
        #for label in a.get_xticklabels():
            #label.set_rotation(35)
            #label.set_horizontalalignment("right")

    trim_note = ""
    if orig_csv_len != csv_len:
```

```
        trim_note = "Removed {} ({}%) samples with non-existent wave
height, wind speed, or dominant period measurements (99
placeholder).\n".format(
                orig_csv_len - csv_len, round((orig_csv_len -
csv_len)/float(orig_csv_len)*100, 2))
    text="""{trim_note}Buoy Depth: {depth}m. Average wave height:
{wave_height_avg} m. Average wind speed: {wind_speed_avg} m/s.
NOAA Buoy data from:
https://www.ndbc.noaa.gov/station_page.php?station={buoy_num}
Sample interval is {measure_mins} mins.
Days for each failure is when failure condition is reached more than 2
hours total on each day.
Dominant wave period used to calculate steepness for Nuka response."""
    text = text.format(wave_height_avg = round(csv['WVHT'].mean(), 2)
            , depth= args.depth
            , wind_speed_avg =  round(csv['WSPD'].mean(), 2)
            , measure_mins=interval.total_seconds() / 60
            , trim_note = trim_note
            , buoy_num=buoy_num)
    plt.figtext(0.2, -0.120, text, fontsize='small')

    plt.tight_layout(rect=[0, 0, 0.90, 0.96])

    # output
    fig.set_figwidth(20) #longest was 30
    output_plot = os.path.join(args.output_dir, 'pdf',
"{}.pdf".format(output_prefix))
    plt.savefig(output_plot, dpi=300, bbox_inches='tight')

    # print summary
    print "--------------"
    print "output written to: {}".format(output_csv)
    print "data file:"
    print os.path.basename(args.file)
    print "buoy depth: {} m".format(args.depth)
    print ""
    print "\nresults:"
    print "samples: {}".format(len(csv))
    print "average wave celerity: {} m/s ".format(
csv['wave_celerity'].mean() )
    print "average wind speed: {} m/s ".format( csv['WSPD'].mean() )
    print "average wave height: {} m/s ".format( csv['WVHT'].mean() )
```

Python script for Currents

```
#!/usr/bin/env python
import argparse
import os
import pandas as pd
import sys
import numpy as np
import math
from datetime import datetime, timedelta
from scipy.interpolate import CubicSpline, interp1d

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='use current data and
chart when failures occur')
    parser.add_argument('--file', '-f', required=True, help="buoy to
read")
    parser.add_argument('--output_file', '-o', required=True)
    parser.add_argument('--name', '-n', required=True)
    args = parser.parse_args()

    # idea here is to compute when and *for how long* the current is over
a
    # given failure threshold, so we're basically doing solve
    # or maybe even peice-wise integration?

    # expects a column of 'time' and 'knots'
    real_speeds = pd.read_csv(args.file, header=[0])
    real_speeds['time'] = pd.to_datetime(real_speeds['time'])

    # build high resolution model dataframe
    print "building high res model"
    times = (real_speeds['time'] -
real_speeds['time'][0]).dt.total_seconds()
    cs = CubicSpline(times, real_speeds['knots'])
    xs = np.arange(times.values[0], times.values[-1], 30) #TODO adjust
interp resolution, 30s seems fine
    model = pd.DataFrame({'time': xs, 'knots': cs(xs)})
    model['time'] = pd.to_datetime(model['time'], unit='s',
```

```
origin=real_speeds['time'][0])

    #resample into 30 minute buckets, taking the max of knots
    print "resampling"
    resample_interval = '30T'
    bar_width = 1/48.0
    bucketed = model.set_index('time').resample(resample_interval,
label='right').max()
    max_knots =
pd.DataFrame({'knots':bucketed['knots'].abs()}).reset_index()

    #criterias
    print "querying failure conditions"
    failures = [
        ('Highest catenary "first failure" in Schulze (>=1.36 knots)',
'knots >= 1.36')
        , ('Kepner boom maximum (>=1.5 knots)', 'knots >= 1.5')
        , ('Highest diversionary "first failure" in Schulze (=>1.7
knots)', 'knots >= 1.7')
        , ('Gross Failure in Swift (>=2 knots)', 'knots >= 2')
        , ('RoBoom 2000 maximum (>=3 knots)', 'knots >= 3')
        , ('Current Buster maximum (>=4 knots)', 'knots >= 4')
    ]

    #plot
    num_plots = len(failures)+1
    fig, ax = plt.subplots(num_plots,1, sharex=True)
    #plt.figure(figsize=(6.5, 4))

    #colors to assign
    color_iter = plt.rcParams['axes.prop_cycle']
    colors = [p['color'] for p in color_iter]

    #ax[0].set_title("Wave Speed")
    #ax[0].plot(real_speeds['time'].values,real_speeds['knots'].values,
'o', label='data')
    #ax[0].plot(model['time'].values,model['knots'].values, label='model')
    dp = "Data Present"
    ax[0].set_title("Data Present")
    ax[0].bar(max_knots['time'].values, 1, width=bar_width, align='edge',
color=colors[0])

    i = 1
    for fname,fquery in failures:
        fdf = max_knots.query(fquery)
```

```
        pct = round(len(fdf) / float(len(max_knots)) * 100,2)
        full_name = "{}: ({} intervals {}%)".format(fname, len(fdf), pct)
        print full_name
        ax[i].set_title(full_name)
        ax[i].bar( fdf['time'].values, 1, width=bar_width, align='edge',
color=colors[i])
        i += 1

    print "plotting"
    plt.axis('tight')
    fig.suptitle(args.name, fontsize=16)

    # unclear which axis is the one that formatting comes from for
suboplots, so we just do em all
    for a in ax:
        a.xaxis_date('UTC')

a.xaxis.set_major_locator(mdates.MonthLocator(bymonth=range(1,13)))
        a.xaxis.set_major_formatter(mdates.DateFormatter('%Y %b'))
        a.xaxis.set_minor_locator(mdates.DayLocator(bymonthday=[1,15]))
        a.xaxis.set_minor_formatter(mdates.DateFormatter('\n%d'))
        a.yaxis.set_visible(0)
        interval = (max_knots['time'] - max_knots['time'].shift())[1]
        a.set_xlim([ max_knots['time'].values[0]-interval,
max_knots['time'].values[-1]+interval ])
        a.set_ylim([ 0, 1 ])

    text="""Using current speed predictions, cubic splines are used to
interpolate speed between maximum flood/slack/ebb.
Sample interval is 30 mins. Knots compared is maximum during interval.
Average absolute current speed: {avg_current_speed} knots"""
    text =
text.format(avg_current_speed=round(max_knots['knots'].mean(),2))
    plt.figtext(0.2, -0.105, text, fontsize='small')

    plt.tight_layout(rect=[0, 0, 0.90, 0.96])
    fig.set_figwidth(20) #longest was 30
    plt.savefig(args.output_file, dpi=300, bbox_inches='tight')
```