1_TWA_TWS_STW_from_InfluxDBv2_calc_perf

July 10, 2021

1 Analyze DashT data off-line

The purpose of this tutorial is to give an insight how the data stored by DashT, an OpenCPN chartplotter plug-in can be viewed and analyzed, outside the *DashT*, numerically, without using graphical analysis tools such as InfluxDB or Grafana.

The simplified example is using Python 3 with Pandas DataFrame.

> **JavaScript/TypeScript** is used by *DashT* in its Line Chart instrument in quite similar manner, please consult the source code available elsewhere in this repository. However, it can be more daunting for understanding...

The example is introducing a small error in end results by not taking into account the exact timestamp of each retrieved sample but using indexing, for clarity's sake. The error and how to avoid it is explained in detail in the more complex paper `2_TWA_TWS_STW_from_InfluxDBv2_collect_data_for_polar.ipynb`. The error is insignificant in the short period of time, like in a regatta leg.

1.1 Retrieve TWA, TWS and STW from InfluxDB v2.0, estimate performance against polar

In this example we analyze the data which has been collected in a past event into the InfluxDB v2.0 time series database using DashT database streamer.

We retrieve True Wind data for speed and angle (TWS and TWA) and speed through water data (STW) from the database during a period of interest.

We compare the STW value for the optimal polar performance value, provided by a local CSV file. For this purpose we develop a function which return extrapolated polar speed value for given TWS and TWA value.

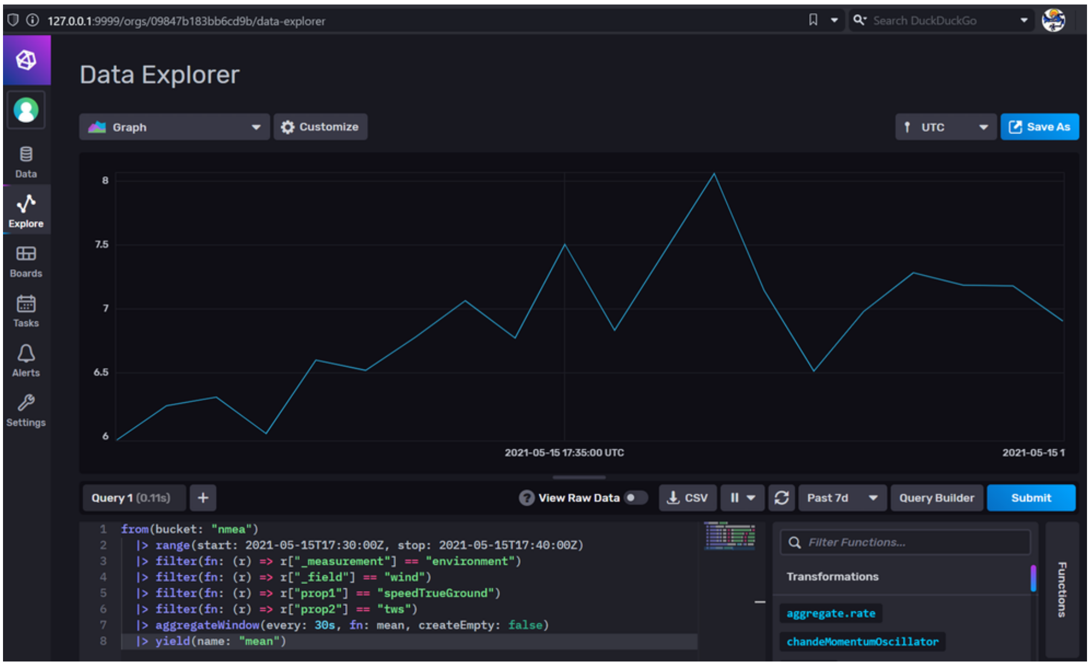
Finally, we plot the polar performance percentage during the period of interest.

This example is executed on the data still on my computer, collected during a training session. DashT InfluxDB output streamer was registering all activated instruments into InfluxDB v2.0.

I have previously visualized the data on InfluxDB v2.0 web interface explorer and determined that the efficient sail time with a search for performance was from 2021-05-15T11:54:21Z to 2021-05-15T19:30:00Z :

> It is good idea to record, while sailing the interesting time period. If you did not remember to it, you can use the InfluxDB's data explorer to find a suitable timeframe region of interest. In this example the Flux query to visualize the TWS data built (you do not have to type it) with InfluxDB 2.0 data explorer user interface is:

```
from(bucket: "nmea")
  |> range(start: 2021-05-15T17:30:00Z, stop: 2021-05-15T17:40:00Z)
  |> filter(fn: (r) => r["_measurement"] == "environment")
  |> filter(fn: (r) => r["_field"] == "wind")
  |> filter(fn: (r) => r["prop1"] == "speedTrueGround")
  |> filter(fn: (r) => r["prop2"] == "tws")
  |> aggregateWindow(every: 30s, fn: mean, createEmpty: false)
  |> yield(name: "mean")
```



(zoom)

Here we use simple https://github.com/influxdata/influxdb-client-python#queries method :

1.2 Python example

This example is executed with Jupyter 2.2.8 and thus with Python 3, making connection to the InfluxDB v2.0 server where the above data is located.

When opened on an external data retrieval system, this .ipynb file would not execute but shows only the code: * There is a PDF-printout of a run of code, with the plots: `1_TWA_TWS_STW_from_InfluxDBv2_calc_perf.pdf` * A self-contained HTML-page with table of contents inside `1_TWA_TWS_STW_from_InfluxDBv2_calc_perf.zip` * The Python code does not require Jupyter - it is used for development and documentation purposes

[1]:
```python
import os
import sys

sys.path.insert(0, os.path.abspath('../'))
```

1.2.1 Python tools

Needed to install:

```
pip install influxdb_client
pip install matplotlib
pip install numpy
```

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

1.2.2 InfluxDB Client API

[3]:
```
from influxdb_client import InfluxDBClient, Point, Dialect
from influxdb_client.client.write_api import SYNCHRONOUS
```

1.2.3 From DashT InfluxDB Out streamer settings

[4]:
```
client = InfluxDBClient(url='http://127.0.0.1:9999', token='xzB3..ow==',␣
↪org='myboat')
```

1.2.4 Data range and aggregation

[5]:
```
# start_time = '2021-05-15T17:30:00Z'
# stop_time = '2021-05-15T17:40:00Z'
start_time = '2021-05-15T11:54:21Z'
stop_time = '2021-05-15T19:30:00Z'
aggregate_window = '10s'
aggregate_function = 'mean'
```

[6]:
```
query_api = client.query_api()
```

True Wind Speed Data

[7]:
```
tws_flux = '''
    from(bucket: "nmea")
        |> range
''' +   '(start: ' + start_time + ', stop: ' + stop_time + ')' + '''
        |> filter(fn: (r) => r["_measurement"] == "environment")
        |> filter(fn: (r) => r["_field"] == "wind")
        |> filter(fn: (r) => r["prop1"] == "speedTrueGround")
        |> filter(fn: (r) => r["prop2"] == "tws")
        |> keep(columns: ["_time", "_value"])
        |> aggregateWindow(every:
''' +     aggregate_window + ', fn: ' + aggregate_function + ','  + '''
         createEmpty: false)
'''
```

Record format query

[8]:
```
# tws_flux
```

[9]:
```
# records = query_api.query_stream( tws_flux )
```

[10]:
```
# for record in records:
#    print(f'TWS in {record["_time"]} is {record["_value"]}')
```

Panda DataFrame

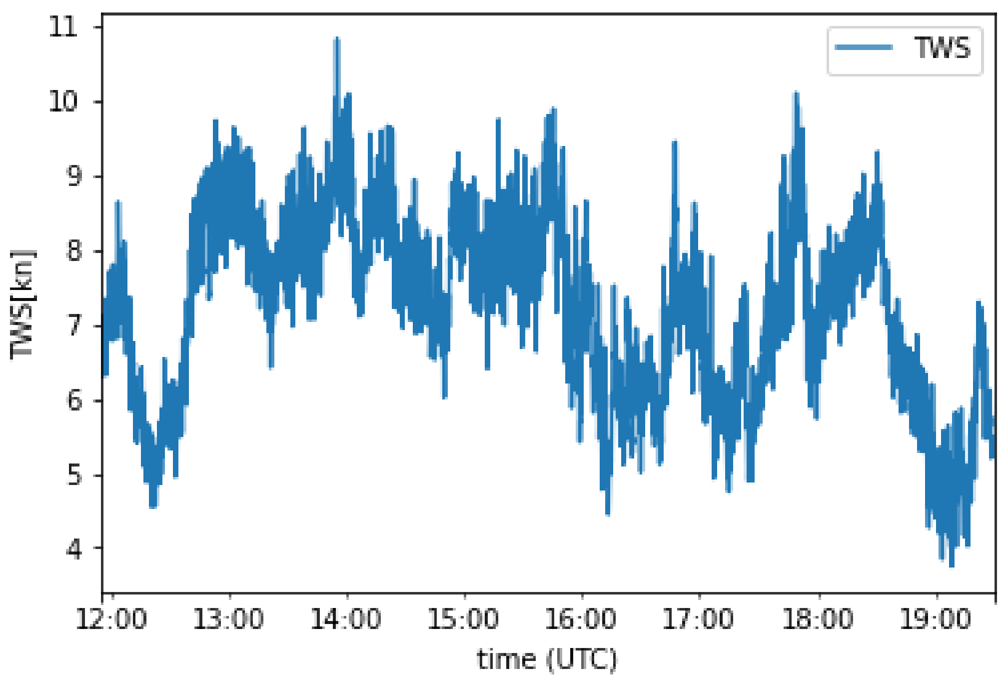
[11]:
```python
tws_df = query_api.query_data_frame( tws_flux )
```

[12]:
```python
tws_df.head()
```

[12]:
```
    result  table                     _start                     _stop  \
0  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
1  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
2  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
3  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
4  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00

     _value                     _time
0  6.451623 2021-05-15 11:54:30+00:00
1  6.573691 2021-05-15 11:54:40+00:00
2  6.382371 2021-05-15 11:54:50+00:00
3  6.291570 2021-05-15 11:55:00+00:00
4  7.098524 2021-05-15 11:55:10+00:00
```

[13]:
```python
tws_df.plot(x='_time',y='_value', label='TWS')
plt.ylabel('TWS[kn]')
plt.xlabel('time (UTC)')
plt.legend()
plt.show()
```



1.2.5 True Wind Angle Data

[14]:
```
twa_flux = '''
    from(bucket: "nmea")
        |> range
''' +   '(start: ' + start_time + ', stop: ' + stop_time + ')' + '''
        |> filter(fn: (r) => r["_measurement"] == "environment")
        |> filter(fn: (r) => r["_field"] == "wind")
        |> filter(fn: (r) => r["prop1"] == "angleTrue")
        |> keep(columns: ["_time", "_value"])
        |> aggregateWindow(every:
''' +     aggregate_window + ', fn: ' + aggregate_function + ','  + '''
          createEmpty: false)
'''
```

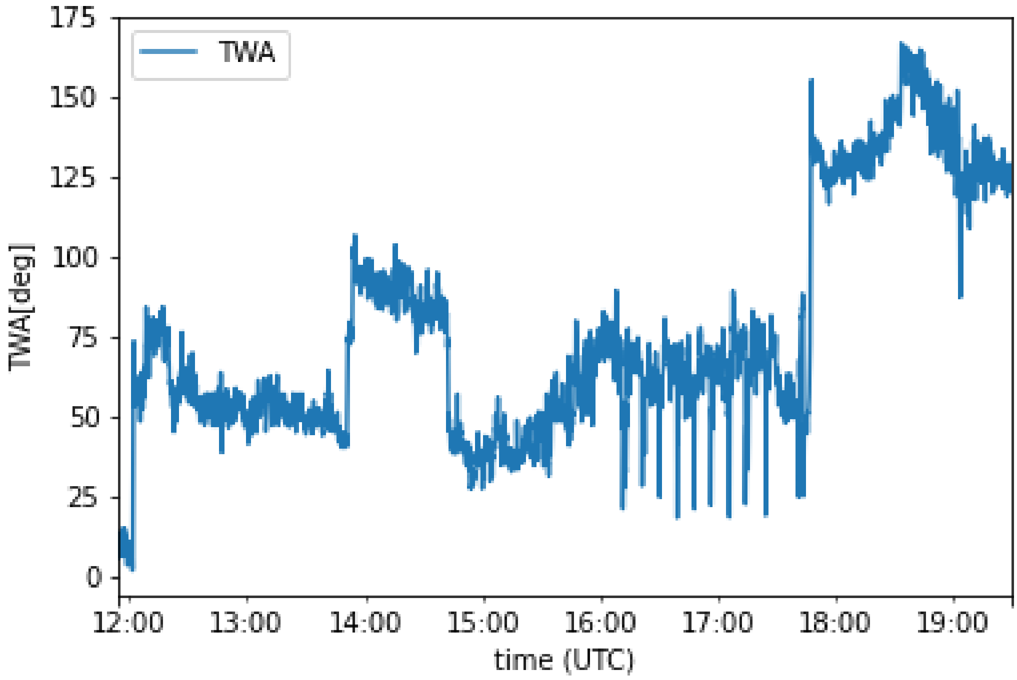
[15]:
```
twa_df = query_api.query_data_frame( twa_flux )
```

[16]:
```
twa_df.head()
```

[16]:
```
    result table                     _start                     _stop  \
0  _result     0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
1  _result     0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
2  _result     0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
3  _result     0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
4  _result     0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00

     _value                     _time
0  5.576830 2021-05-15 11:54:30+00:00
1  5.622653 2021-05-15 11:54:40+00:00
2  6.231672 2021-05-15 11:54:50+00:00
3  5.942707 2021-05-15 11:55:00+00:00
4  9.956448 2021-05-15 11:55:10+00:00
```

[17]:
```
twa_df.plot(x='_time',y='_value', label='TWA')
plt.ylabel('TWA[deg]')
plt.xlabel('time (UTC)')
plt.legend()
plt.show()
```



1.2.6 Speed Through Water Data

[18]:
```
stw_flux = '''
    from(bucket: "nmea")
        |> range
''' +   '(start: ' + start_time + ', stop: ' + stop_time + ')' + '''
        |> filter(fn: (r) => r["_measurement"] == "navigation")
        |> filter(fn: (r) => r["_field"] == "speedThroughWater")
        |> keep(columns: ["_time", "_value"])
        |> aggregateWindow(every:
''' +     aggregate_window + ', fn: ' + aggregate_function + ','  + '''
         createEmpty: false)
'''
```

[19]:
```
stw_df = query_api.query_data_frame( stw_flux )
```

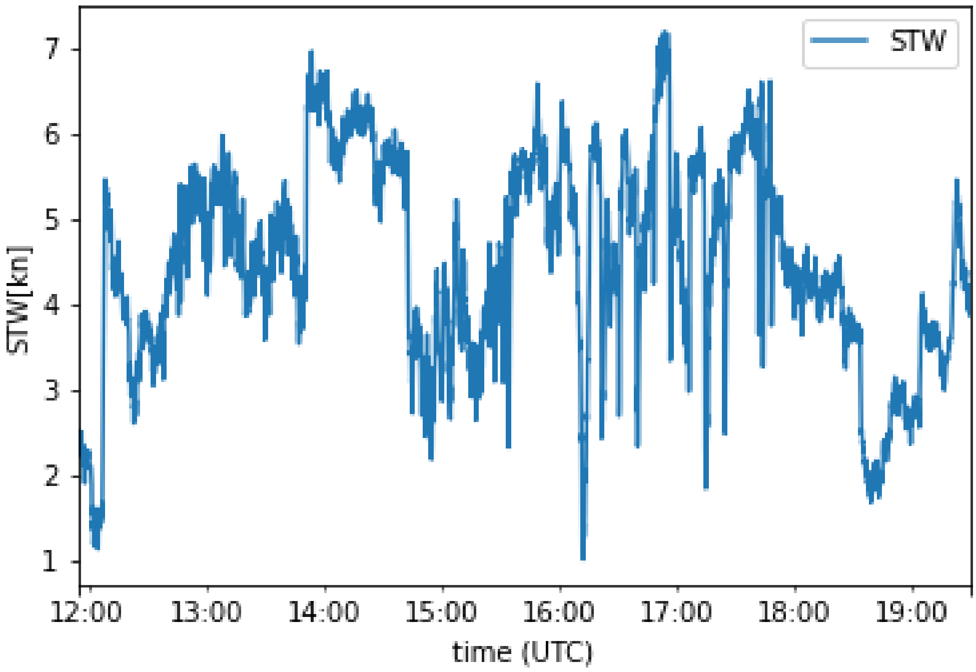
[20]:
```
stw_df.head()
```

[20]:
```
    result  table                    _start                     _stop  \
0  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
1  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
2  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
3  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00
4  _result      0 2021-05-15 11:54:21+00:00 2021-05-15 19:30:00+00:00

     _value                     _time
0  2.410000 2021-05-15 11:54:30+00:00
```

```
1  2.370000 2021-05-15 11:54:40+00:00
2  2.512857 2021-05-15 11:54:50+00:00
3  2.254286 2021-05-15 11:55:00+00:00
4  2.211667 2021-05-15 11:55:10+00:00
```

[21]:
```
stw_df.plot(x='_time',y='_value', label='STW')
plt.ylabel('STW[kn]')
plt.xlabel('time (UTC)')
plt.legend()
plt.show()
```



Close database streaming connection

[22]:
```
client.close()
```

1.2.7 Reference Polar Data

Your boat's polar in CSV file (not .pol)

[23]:
```
pol_df = pd.read_csv("ref_polar.csv", sep=',')
```

Inspecting Polar DataFrame

> **NOTE**: it is a good idea to look how the Panda DataFrame is indexed now: we need to be able to find a label for the column of TWA values. In our example it is `TWA/TWS` while it can be as well something else, like `0` (for 0 kn Wind Speed…) - there is no real standard for this. The example is coming out from Adrena navigation software. You can skip this study once you know your polar file resulting index key.

[24]:
```
pol_df.head()
```

[24]:
```
   TWA/TWS     2     4    6    8   10   12   14   16   18  ...   24   26   28␣
 ↪ \
0        0  0.00  0.00  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0  0.0
1       25  0.70  0.80  0.4  0.4  0.5  0.6  0.7  0.8  0.7  ...  1.0  1.1  1.2
2       30  2.10  2.90  1.4  1.4  1.7  1.8  1.9  2.0  1.9  ...  2.2  2.4  2.5
3       35  2.45  3.00  2.8  2.9  3.5  3.7  3.9  4.0  3.9  ...  4.4  4.9  5.1
4       40  2.50  3.05  3.6  3.7  4.0  4.3  4.4  4.5  4.4  ...  5.8  6.1  6.6

    30   32   34   36   38   40   60
0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
1  1.4  1.5  1.7  1.9  0.0  0.0  0.0
2  2.9  3.1  3.6  4.1  0.0  0.0  0.0
3  5.8  6.0  6.5  7.0  0.0  0.0  0.0
4  7.1  7.3  7.8  8.3  0.0  0.0  0.0

[5 rows x 22 columns]
```

for column_name, item in pol_df.iteritems(): print(type(column_name)) print(column_name) print('~~~~~~')

```
print(type(item))
print(item)
print('------')
```

[25]:
```
pol_df = pd.read_csv("ref_polar.csv", sep=',', header=0, index_col='TWA/TWS')
```

[26]:
```
pol_df.head()
```

[26]:
```
            2     4    6    8   10   12   14   16   18   20  ...   24   26  \
TWA/TWS                                                      ...
0        0.00  0.00  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  ...  0.0  0.0
25       0.70  0.80  0.4  0.4  0.5  0.6  0.7  0.8  0.7  0.8  ...  1.0  1.1
30       2.10  2.90  1.4  1.4  1.7  1.8  1.9  2.0  1.9  2.0  ...  2.2  2.4
35       2.45  3.00  2.8  2.9  3.5  3.7  3.9  4.0  3.9  4.1  ...  4.4  4.9
40       2.50  3.05  3.6  3.7  4.0  4.3  4.4  4.5  4.4  4.9  ...  5.8  6.1

          28   30   32   34   36   38   40   60
TWA/TWS
0        0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
25       1.2  1.4  1.5  1.7  1.9  0.0  0.0  0.0
30       2.5  2.9  3.1  3.6  4.1  0.0  0.0  0.0
35       5.1  5.8  6.0  6.5  7.0  0.0  0.0  0.0
40       6.6  7.1  7.3  7.8  8.3  0.0  0.0  0.0

[5 rows x 21 columns]
```

The below block is only needed for better understanding if you so wish

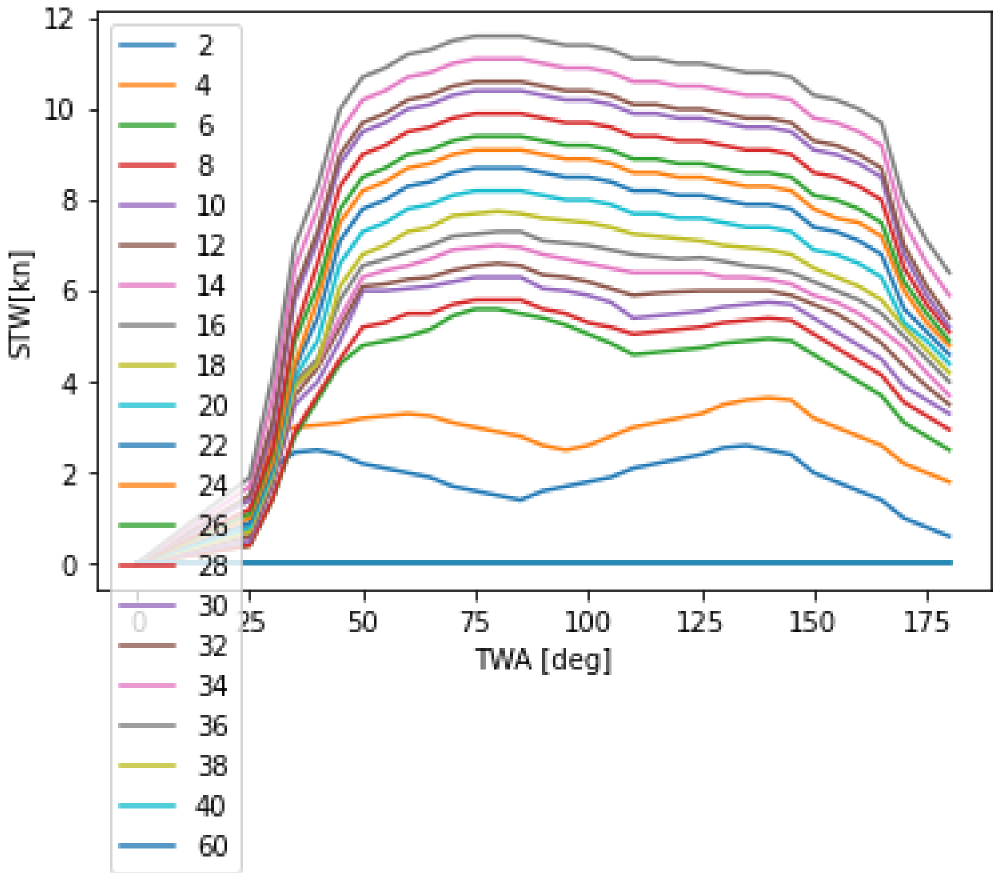
[27]:
```
# for column_name, item in pol_df.iteritems():
#     print(type(column_name))
#     print(column_name)
#     print('~~~~~~~')

#     print(type(item))
#     print(item)
#     print('------')
```

We have now clear column (wind) and row (angle) indices obtained from the polar file and not those sequentially autogenerated by Panda.

Plotting out TWA/STW polar curves

[28]:
```
pol_df.plot(label='Polar')
plt.ylabel('STW[kn]')
plt.xlabel('TWA [deg]')
plt.legend()
plt.show()
```



Looking up STW values from the polar DataFrame We will hardly get values which will fall spot on a cell of the polar file. We need to find group of cells which allows us to extrapolate by averaging the polar value against the floating value measurement value. Below we select test values to develop that:

1.2.8 Make a subroutine returning polar speed

Note that for performance reasons we expect the polar data having been read from the CSV-file into a Panda DataFrame as explained above - there is no need to open the file for every single TWA and TWS tuple. The routing expects and is using the provided example polar file where the TWA column has header `TWA/TWS` (mandatory for his routine, used in selecting that column) and all other columns the respective TWS values.

> **NOTE**The understanding of how this routine works is not required to understand the rest of the tutorial, provided that you use the example CSV-file.

[237]:
```
def getpolspeed( pol_df, twa, tws ) :
    # print('twa: ', twa, 'tws: ', tws)
    twa_coldf = pol_df['TWA/TWS']
    lowertws_coldf = None
    lowertws = None
    uppertws_coldf = None
    uppertws = None

    for column_name, item in pol_df.iteritems():
        try:
            pol_tws = float(column_name)
            # print( pol_tws )
            if tws >= pol_tws :
                lowertws_coldf = item
                lowertws = pol_tws
            else :
                if pol_tws >= tws :
                    uppertws_coldf = item
                    uppertws = pol_tws
                    break
        except ValueError:
                # float() failure on the first column TWA/TWS
                    continue

    # print('lowertws: ', lowertws, ' type df: ', type(lowertws_coldf))
    # print('uppertws: ', uppertws, ' type df: ', type(uppertws_coldf))

    null_stw      = None
    lowertwa      = None
    uppertwa      = None
    lwrtwa_lwrtws = null_stw
    uprtwa_lwrtws = null_stw
    lwrtwa_uprtws = null_stw
    uprtwa_uprtws = null_stw

    for column_name, item in lowertws_coldf.iteritems():
        pol_twa = float(twa_coldf[column_name])
        # print( pol_twa )
```

```
        if twa >= pol_twa :
            lwrtwa_lwrtws = item
            lowertwa = pol_twa;
            # print('lowertwa: ', lowertwa, ' lwrtwa_lwrtws: ', lwrtwa_lwrtws)
        else :
            if pol_twa >= twa :
                if type(uprtwa_lwrtws) is type(null_stw) :
                    uprtwa_lwrtws = item
                    uppertwa = pol_twa;
                    # print('uppertwa: ', uppertwa, ' uprtwa_lwrtws: ', \
                    #        uprtwa_lwrtws)
                    break

    for column_name, item in uppertws_coldf.iteritems():
        pol_twa = float(twa_coldf[column_name])
        # print(pol_twa)
        if twa >= pol_twa :
            lwrtwa_uprtws = item
            # print('lwrtwa_uprtws: ', lwrtwa_uprtws)
        if pol_twa >= twa :
            if type(uprtwa_uprtws) is type(null_stw) :
                uprtwa_uprtws = item
                # print('uprtwa_uprtws: ', uprtwa_uprtws)
                break

    # Here we have the "square" of values in which the extrapolated value is␣
↪located:

    # print('lowertws: ', lowertws)
    # print('lowertwa: ', lowertwa)
    # print('uppertws: ', uppertws)
    # print('uppertwa: ', uppertwa)
    # print('lwrtwa_lwrtws: ', lwrtwa_lwrtws)
    # print('uprtwa_lwrtws: ', uprtwa_lwrtws)
    # print('lwrtwa_uprtws: ', lwrtwa_uprtws)
    # print('uprtwa_uprtws: ', uprtwa_uprtws)

    # Consider the result being on a 3D plane with four corners, all inequal␣
↪height.
    # There will be a point on that plane, the height of which is the␣
↪estimated
    # polar speed for that point:

    # lwrtwa_lwrtws (x)--+-------(x) lwrtwa_uprtws
    #                  |   |        |
    #                  |   |        |
    #                  +--(?)-------+
    #                  |   |        |
    # uprtwa_lwrtws (x)--+-------(x) uprtwa_uprtws
```

```
    # Taking upper left corner as origin, we will calculate the point's
    # projected elevations.

    # **NOTE**: We can have planes which are sliding down (negative␣
↪elevations),
    # especially when the wind speed is getting higher and the polar takes␣
↪average
    # of different sail sets.

    lwrtwa_lwrtws_orig = 0.0
    lwrtwa_uprtws_elev = lwrtwa_uprtws - lwrtwa_lwrtws
    # print('lwrtwa_uprtws_elev: ', lwrtwa_uprtws_elev)
    uprtwa_lwrtws_elev = uprtwa_lwrtws - lwrtwa_lwrtws
    # print('uprtwa_lwrtws_elev: ', uprtwa_lwrtws_elev)
    uprtwa_uprtws_elev = uprtwa_uprtws - lwrtwa_lwrtws
    # print('uprtwa_uprtws_elev: ', uprtwa_uprtws_elev)

    #                     0.0 (x)--+-------(x) 0.2 lwrtwa_uprtws_elev
    #                          |   |        |
    #                          |   |        |
    #                          +--(?)-------+
    #                          |   |        |
    # uprtwa_lwrtws_elev 0.5 (x)--+-------(x) 0.8 uprtwa_uprtws_elev

    tws_ratio = (test_tws - lowertws)/(uppertws - lowertws)
    # print('tws_ratio: ', tws_ratio)

    twa_ratio = (test_twa - lowertwa)/(uppertwa - lowertwa)
    # print('twa_ratio: ', twa_ratio)

    y_lwrtws = twa_ratio * uprtwa_lwrtws_elev
    # print('y_lwrtws: ', y_lwrtws)

    y_uprtws = lwrtwa_uprtws_elev + (twa_ratio * (uprtwa_uprtws_elev -␣
↪lwrtwa_uprtws_elev))
    # c('y_uprtws: ', y_uprtws)

    x_lwrtws = tws_ratio * lwrtwa_uprtws_elev
    # print('x_lwrtws: ', x_lwrtws)

    x_uprtws = uprtwa_lwrtws_elev + (tws_ratio * (uprtwa_uprtws_elev -␣
↪uprtwa_lwrtws_elev))
    # print('x_uprtws: ', x_uprtws)

    #                              0.09 x_lwrtws
    #                     0.0 (x)--+-------(x) 0.2 lwrtwa_uprtws_elev
    #                          |   |        |
    #                          |   |        |
    #           y_lwrtws 0.22 +--(?)-------+ 0.46 y_uprtws
    #                          |   |        |
```

```
    # uprtwa_lwrtws_elev 0.5 (x)--+-------(x) 0.8 uprtwa_uprtws_elev
    #                              0.64 x_uprtws

    ##### Result

    x_res = x_lwrtws + (x_uprtws - x_lwrtws) * twa_ratio
    # print('x_res: ', x_res)
    # print('result: ', (lwrtwa_lwrtws + x_res))

    ##### Verification

    y_res = y_lwrtws + (y_uprtws - y_lwrtws) * tws_ratio
    # print('y_res: ', y_res)
    if round(y_res,2) != round(x_res,2) :
        printf('WARNING: polar explaration function verification fails: ',␣
↪x_res, y_res)

    return round((lwrtwa_lwrtws + x_res),2)
```

Test subroutine

[30]:
```
test_twa = 37.2
test_tws = 10.9
```

[31]:
```
test_pol_speed = getpolspeed( pol_df, test_twa, test_tws )
print('test_pol_speed: ', test_pol_speed)
if round(test_pol_speed,2) != 3.83 :
    print('getpolspeed test FAIL: was expecting 3.83 with the test polar␣
↪file')
```

```
test_pol_speed:  3.83
```

[32]:
```
test_twa = 60.24671838095237
test_tws = 6.030811476190476
```

[33]:
```
test_pol_speed = getpolspeed( pol_df, test_twa, test_tws )
print('test_pol_speed: ', test_pol_speed)
if round(test_pol_speed,2) != 5.01 :
    print('getpolspeed test FAIL: was expecting 5.01 with the test polar␣
↪file')
```

```
test_pol_speed:  5.01
```

1.2.9 Compare STW dataset to polar values resulting from TWA/TWS values

Below block can be used to dump-list the data for debugging purposes but its main purpose is to compose a new Panda DataFrame, which will contain the data to be plotted from the retreived and from the processed data, in this case we compare STW to the polar speed.

[34]:
```
idx = 0
stwpol_df = pd.DataFrame(columns=['_time', 'stw', 'pol', 'per'])
for row in stw_df.iterrows() :
```

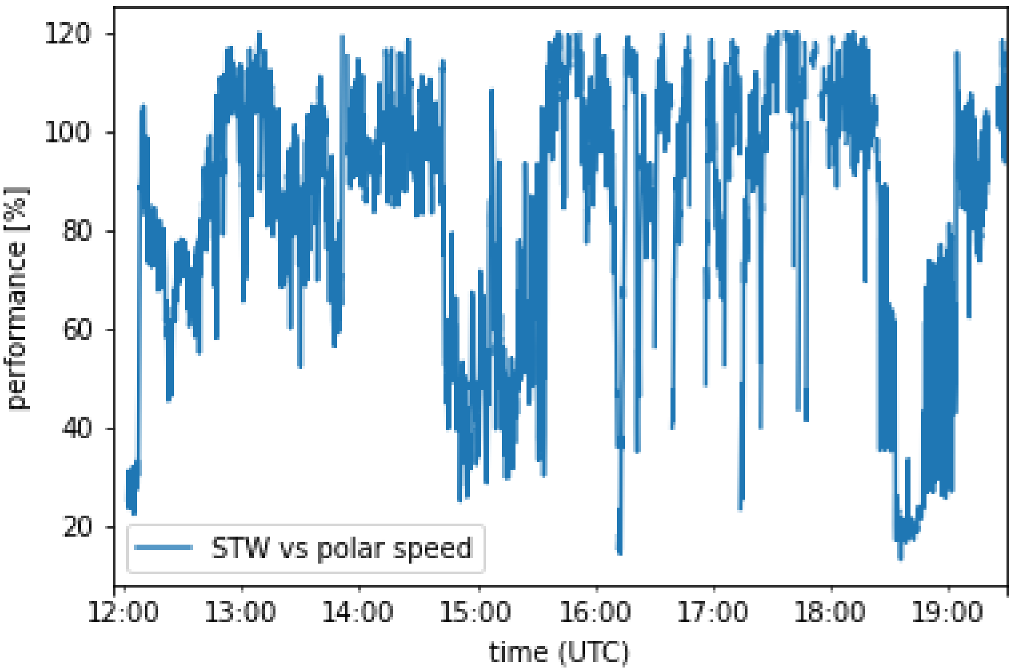
```
    # print(twa_df['_time'][idx],' ', twa_df['_value'][idx],' ',␣
↪tws_df['_value'][idx],' ', )
    polspeed = getpolspeed( pol_df, twa_df['_value'][idx],␣
↪tws_df['_value'][idx] );
    stwspeed = stw_df['_value'][idx]
    performance = stwspeed / polspeed * 100
    # Something is wrong with the polar or a typhoon hit us if performance is␣
↪over 120%?
    if performance > 120 :
        performance = None
    else :
        performacne = int(performance)
    stwpol_df =stwpol_df.append({'_time': twa_df['_time'][idx], 'stw':␣
↪round(stwspeed,2), \
                                'pol': polspeed, 'per': performance},\
                                ignore_index=True)
    idx = idx + 1
```

[35]:
```
stwpol_df.head()
```

[35]:

	_time	stw	pol	per
0	2021-05-15 11:54:30+00:00	2.41	0.96	NaN
1	2021-05-15 11:54:40+00:00	2.37	0.96	NaN
2	2021-05-15 11:54:50+00:00	2.51	0.96	NaN
3	2021-05-15 11:55:00+00:00	2.25	0.96	NaN
4	2021-05-15 11:55:10+00:00	2.21	0.96	NaN

[36]:
```
stwpol_df.plot(x='_time',y='per', label='STW vs polar speed')
plt.ylabel('performance [%]')
plt.xlabel('time (UTC)')
plt.legend()
plt.show()
```


120
100
80
60
40
20
performance [%]
STW vs polar speed
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
time (UTC)