Blog

July 20th 2019

Export and Analysis: Supporting Material

In this supporting document methods and their implementation in the Python script will be explained and discussed. The blog entry this article refers to can be found here in English and here in German.

Reading data from the text file

We want to use Python's abilities to read the raw data from the data.csv file and save it internally for easier access. We do this by creating a list object `D` in which we save the data. `D` itself contains only lists: two lists for time information (one of which is redundant and will be deleted) plus one list for each sensor. Inside these lists the raw data is stored. We can write the corresponding method `readFile` as follows:

```
1  def readFile(file):
2      D=[]
3      with open(file) as rawdata:
4          for (i,line) in enumerate(rawdata):
5              for (j,element) in enumerate(
6                  line.strip('\n').replace(',','.').split(';')):
7                  if i==0:
8                      D.append([])
9                  else:
10                     D[j].append(element)
11     del D[1] #remove human-readable time
12     M=np.array(D,dtype='float64')
13     #conversion: unix timestamp -> hours from first data point
14     M[0]=(M[0]-M[0,0])/np.float64(60**2)
15     return M
```

In line `3` we use Python's `with` statement to avoid explicitly opening and closing the file. In line `6` we remove the line break `'\n'` from each line and replace the decimal delimiter `','` by a point. We will need this for later converting the list object `D` into a NumPy array.

The `if-else` clause in lines `7` to `10` creates the lists inside `D` for the case where the line number `i` is equal to zero. That is the case in which the first line of the text file is processed in which only the names of the columns are saved. We do not want them to be part of `D`, but we can use them to initialize the lists inside `D`. Only when `i` is bigger than zero, real measured data points are read from the file. These are saved in the `j`th list inside `D` according to their column position `j`.

After removing the column inside `D` that contains the redundant time information (leaving only the unix timestamp in column zero), we can convert the list object `D` into a NumPy array `M` with shape (number of sensors + 1 (one for time), number of recorded data points per sensor). In our case that means `M` has shape (9, 4206).

Before we go on, we want to simplify the time information. In line `14` we therefore subtract the timestamp of the first data point from all timestamps. These are saved in the first element of `M`. We can access all these 4206 values by indexing the array as `M[0]`. We can access the first element inside `M[0]` by writing `M[0,0]`[1]. After that, we divide the result by the number of seconds per hour (that is 60^2 s/h).

[1] We could also write `M[0][0]` as we would do for Python's lists. Using NumPy's array objects this would internally create an array `m=M[0]` and then return the 0th element of `m`. This is slow and inefficient. We can directly access the desired element out of `M` by writing `M[0,0]`. This avoids the creation of array `m`.

Calculating mean values

In step 3 of the blog article we calculate mean values for one day out of data from three days. We can do this by executing function `calc_M_mean` with `M` as the input argument.

```
1  def calc_M_mean(M):
2      def merge(A,B,A_to_B=np.float64(1)):
3          S=np.subtract.outer(A[0],B[0])
4          i,j=np.asarray(np.abs(S)<1E-5).nonzero()
5          C=np.empty((A.shape[0],i.shape[0]),dtype='float64')
6          C[0]=A[0,i]
7          C[1:]=(A[1:,i]*A_to_B+B[1:,j])/(A_to_B+np.float64(1))
8          return C
9      for day in range(0,int(np.ceil(M[0,-1]/np.float64(24)))):
10         M_day=M[:,np.all(
11             [M[0]>=np.float64(day*24),M[0]<np.float64(24+day*24)],axis=0)]
12         M_day[0]-=np.float64(day*24)
13         if day==0:
14             M_mean=M_day
15         else:
16             M_mean=merge(M_mean,M_day,A_to_B=np.float64(day))
17     return M_mean
```

Most of this function except lines 2 to 8 should be self-explanatory. In lines 2 to 8 we defined a function `merge` that takes two arrays `A` and `B` and a weight factor `A_to_B` as input arguments where the default value of `A_to_B` is set to 1 (that is `A` and `B` are weighted equally to compute the mean). The weighted mean of `A` and `B` is called `C` and will be returned as the output argument of the function `merge`.

Of course, we want to merge the data of both arrays `A` and `B` according to the datas' timestamps. Here one problem arises: We cannot be sure that the recorded data of each day is complete. In fact, if we split `M` along axis 1 (the resulting arrays will have the shape (9, number of data points for this day)) according to the timestamps into three arrays (one for each 24h period) and ask for the shape of these subarrays of `M`, we get the results (9, 1400), (9, 1404) and (9, 1402). For a complete dataset we expected (9, number of minutes of one day = 60*24 = 1440). We can explain this by taking a look at the data.csv file:

```
179  1560128220;Mon, 10 Jun 2019 00:57:00 GMT;25,9;25,6;27,5;31,6;33,4;35,1;52,56;20,2
180  1560128280;Mon, 10 Jun 2019 00:58:00 GMT;25,9;25,6;27,4;31,5;33,4;35,1;52,55;20,3
181  1560130740;Mon, 10 Jun 2019 01:39:00 GMT;25,6;25;26,7;30,6;32,9;34,7;52,54;20
182  1560130800;Mon, 10 Jun 2019 01:40:00 GMT;25,5;25;26,7;30,6;32,8;34,7;52,55;20
```

```
1578  1560214560;Tue, 11 Jun 2019 00:56:00 GMT;27,7;28,2;30;34,3;34,6;35,1;53,98;22,5
1579  1560214620;Tue, 11 Jun 2019 00:57:00 GMT;27,7;28,1;30;34,3;34,6;35,1;53,98;22,5
1580  1560216840;Tue, 11 Jun 2019 01:34:00 GMT;27,3;27,4;29,9;33,9;34,7;34,9;53,98;22,3
1581  1560216900;Tue, 11 Jun 2019 01:35:00 GMT;27,3;27,4;29,9;34;34,7;34,9;53,98;22,3
```

```
2985  1560301140;Wed, 12 Jun 2019 00:59:00 GMT;28,4;28,7;31,3;34,2;34,7;34,9;55,78;23
2986  1560301200;Wed, 12 Jun 2019 01:00:00 GMT;28,4;28,6;31,4;34,2;34,8;34,9;55,79;23
2987  1560303540;Wed, 12 Jun 2019 01:39:00 GMT;27,6;27,8;30,5;34,1;34,5;35,1;55,75;22,4
2988  1560303600;Wed, 12 Jun 2019 01:40:00 GMT;27,5;27,8;30,5;34,1;34,5;35,1;55,75;22,3
```

Each day between 00:50 and 01:50 (UTC+0) around 40 minutes of data is missing! This is due to BeeBIT's server. It is processing a backup each night. Lately (after July 1st 2019) we scheduled the backups less often. Nevertheless in this dataset from June 2019 we do not have complete data. In consequence, `A` and `B` have different shapes along axis 1. We would have found it very easy to write a merge function `merge_easy` as follows:

```
1  def merge_easy(A,B,A_to_B=np.float64(1)):
2      C=np.empty(A.shape,dtype='float64')
3      C[0]=A[0]
4      C[1:]=(A[1:]*A_to_B+B[1:])/(A_to_B+np.float64(1))
5      return C
```

There we would have copied the timestamps from `A[0]` and calculated the mean values `C[1:]` for the sensor data. Of course, we now have to do something less simple to calculate the merged

dataset `C` according to the right timestamps. If we would ignore the timestamps we would merge data points from two different timestamps and this would make no sense. On top of this, we would not even be able to merge both datasets since both do not have the same length along axis 1.

The question now is: What would be a reasonable strategy to merge both datasets? We have to do the following:

a) find the pairs of indices of `A[0]` and `B[0]` with matching timestamps, ignore all non-matching timestamps

b) initialize `C` with the right dimensions

c) copy all matching timestamps into `C[0]`

d) calculate the weighted mean of all sensor data points with matching timestamps and store the result in `C[1:]` at the right position with the right timestamp

A working implementation is presented here:

```
1 def merge(A,B,A_to_B=np.float64(1)):
2     S=np.subtract.outer(A[0],B[0])
3     i,j=np.asarray(np.abs(S)<1E-5).nonzero()
4     C=np.empty((A.shape[0],i.shape[0]),dtype='float64')
5     C[0]=A[0,i]
6     C[1:]=(A[1:,i]*A_to_B+B[1:,j])/(A_to_B+np.float64(1))
7     return C
```

In line 2 we calculate the pairwise difference of all possible element pairings `A[0,i]` and `B[0,j]` for `i` between 0 and n, `j` between 0 and m. The result is a matrix `S` of shape (n, m). We now check in line 3 if some of these timestamps match by asking if the absolut value of an element of `S` is less than 10^{-5}. The result of this operation will be an array of the shape of `S` containing only the values `True` (element pairing `A[0,i]` and `B[0,j]` matches) or `False` (no match). We now ask for the indices of the elements of `S` whose values are `True` by applying the `nonzero()` function. The result of this operation is stored in two new one-dimensional arrays `i` and `j` that contain the indices of matching elements of `S` in the right order. For each pair `(i[k],j[k])` with k going from zero to the length of `i` (or `k`), the timestamps `A[0,i[k]]` and `B[0,j[k]]` will match. Of course, arrays `i` and `j` have the same shape.

We now initialize `C` with shape (9, length of array `i`) and copy the timestamps from `A[0]` to `C[0]`. We use array `i` to index only those timestamps in `A[0]` that have a match in `B[0]`[2]. We do the same for calculating the weighted mean of the sensor data points and use arrays `i` and `j` for indexing only data points with matching timestamps.

Calculating the discrete time derivative

In step 4 of the blog article we calculate the change rate of the eHive's weight. That is we calculate the time derivative of a discrete dataset. The derivative df/dx of a function $f(x)$ depending on the continuous variable x is defined as:

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{1}$$

If only discrete values for x with equal distances Δx are allowed, equation (1) becomes:

$$\frac{df(x)}{dx} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{2}$$

If we interpret the sensor datasets `M[1:]` as a function of the time data `M[0]` we can compute the time derivative d`M[1:]`$/dt$. In our case the size of Δx corresponds to the time difference of two

[2] Alternatively, we could use array `j` to index the timestamps in `B[0]` that have a match in `A[0]`. The result will be the same.

neighbouring data points. For this computation we will ignore missing data[3] and simply write the time derivative `wdot` of the weight dataset `w=M[7]` as

```
wdot=(M[7,1:]-M[7,:-1])*60.0
```
(3)

with $60 = \frac{1}{\Delta x}$ and Δx in our case being the representation of one minute in the units of hours, that is 1/60. Please note that `M[7,1:]` and `M[7,:-1]` are designed to have the same shape. Otherwise NumPy would raise an error.

Applying the Gaussian filter

In step 4 of the blog article we saw that the calculated weight change rate is very noisy. We generated a smoothed dataset by applying the Gaussian filter. It can be implemented in just a few lines:

```
1 def gaussianFilter(x,y,Np=5E+2,sigma=0.25):
2     p=np.linspace(x[0],x[-1],int(Np),endpoint=True)
3     G=np.subtract.outer(p,x)
4     G=np.exp(-(G**2)/(2*(sigma**2)))
5     g=np.sum(G,axis=1)[:,np.newaxis]
6     q=np.dot(G/g,y)
7     return (p,q)
```

The function has four input arguments: `x` and `y` correspond to a discrete variable (time in our case) and the values of a function depending on the discrete variable. Arrays `x` and `y` are one-dimensional and have the same shape. Two parameters `Np` and `sigma` are used to control the function with `Np` being the number of interpolation points the smoothed values of `y` are calculated on and `sigma` being the width of the Gauss curve.

A Gauss curve $g(x)$ centred around x_0 can be written down as:

$$g(x) \propto \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right) \tag{4}$$

We now want to compute the value of g for each interpolation point taking the role of x_0 and each element of the input array `x` taking the role of x. First, we construct the array `p` for the interpolation points. NumPy offers the function `linspace` which generates an array of shape (`Np`,) with equally distant elements between the first element `x[0]` and the last element `x[-1]` of input array `x`.

Now, we compute the pairwise difference between elements of `p` and `x`. The result is an array `G` of shape (length of `p`, length of `x`). With a given `sigma`, we now compute the value of g being defined by equation (4) for each element of `G` and store it in a new array `G` overwriting the old one. In the above code this is done in line `4`.

Because g is only proportional to $\exp(\cdots)$ we now want to normalize the values of `G` so that the sum along axis 1 is equal to 1. We can do this by dividing the subarrays `G[i]` (with `i` going from zero to `Np`) by the sum over this subarray. We can do this in one step, if we divide `G` by a one-dimensional array `g` containing all the sums of the subarrays along axis 1. Because of NumPy's broadcasting rules, we have to reshape `g` from shape (`Np`,)=(1,`Np`) to shape (`Np`,1). This is done using NumPy's indexing rules and the `newaxis` object in line `5`[4]. The division `G/g` itself is done in line `6`.

In this last step we compute the smoothed dataset `q` of shape (`Np`,) using the dot product between the normalized two-dimensional array `G/g` and the input dataset `y`. We have now successfully computed the filtered dataset over a given number of interpolation points and can return the tuple `(p,q)` of arrays `p` and `q`.

[3] This will effect only one value since the missing data is between around 00:50 and 01:50 and appears as one block. Only the difference between the last data point before and after the discontinuity will be of no value to us.

[4] NumPy's broadcasting rules are a very useful tool. As an example, one could write the outer subtraction in line `3` as `G=p[:,np.newaxis]-x`. Nevertheless, explicitly calling the `subtract.outer` function results in a more readable code that is easier to understand.