High speed decoding a signal from an undocumented rf source

A project work in the course - Advanced Scientific Python Programming 2021

Urban Wiklund

Content

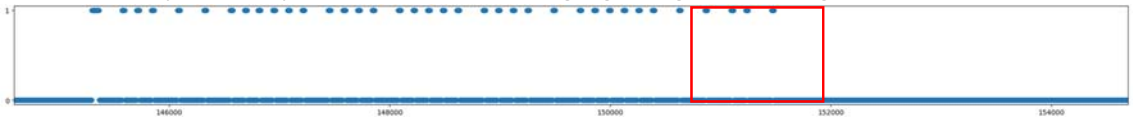
Background and setting:

The digital signal is sent using rf at unknown intervals and unknown rates and our receiver starts recording the signal when waked up by a first “high”. After that 170000 samples are recorded, which is sufficient to contain the whole signal. The signal is not every time of equal length but it always ends with the data containing part. Knowing this, we have to start at the end of the recorded signal and search towards the front of the signal to find the first (or rather last) high. Moreover, the clock frequency of the signal is neither constant nor known (actually it varies between different sensors we have tried) which means we have to be tolerant for variation in clock frequency when decoding the signal into a binary code. So we step towards the beginning of the signal until the signal goes low, this time is marked, and then we step again until it returns high and the time at that position is also marked. The difference, the “low time”, is either short which means a digital “0” or long which means a digital “1”. An extra-long “low time” is used to separate repetitions. Anything else is coded as “w” as in warning because that would indicate a read error. We continue forward until the signal character is changed completely which means we have tried decode beyond the digital part of the signal and we start getting lots of “w”.

The signal looks like this:

At 0 there is a wake up high level, the actual signal arrives much later; the cloud of highs to the right. The zoomed in signal looks like:

Where it is only the final part that is the interesting digital signal. Zooming in on that looks like:



This is what the program should decode into binary code. Zooming in on the last “101” looks like:

A single high level lasts for a number of samles. Zooming in on the last high looks like

Admittedly, it’s not the prettiest example of communication, but it works and it presents plenty of problems to handle while programming.

The output from the code after correctly decoding the signal (in the middle figure above) would be

Out:
ww_10011100000100010000100011000000111011_10011100000100010000100011000000111011
10011100000100010000100011000000111011 translates into 14.0 grader C 29 %RH - sent from sensor 319624

Aim of the project:

We would like the time to analyse the signal to be as short as possible. That means we are ready to capture the next burst of signals as quickly as possible.

Status before the project (before the course):

The analysis routine was written In Python and it was sort of known to not be very optimised for speed. The code and timings at this stage are referred to as “Original”

Work flow during the project

The following steps were performed during the project

1. Only a small part of the full code is to be rewritten so the first work was to isolate the part of the code and arrange that into a few functions. Together these are then called the “original” code. The code was then adjusted to read a signal from a file instead of reading from the air. After this stage I had some code that could be called “original code”, shown page 4. All profiling of the code was made on the function level, excluding of course the file reading part of the code.

2. Six different incremental attempts were tried in search for faster code. That means I first changed the code with strategy A, performed profiling, then changed the code also with strategy B, performed profiling, and so on.

   A. Avoiding having a signals with floats and more use of integers instead

   B. Storing the signal as Nympy arrays, during the Conversion function to start with

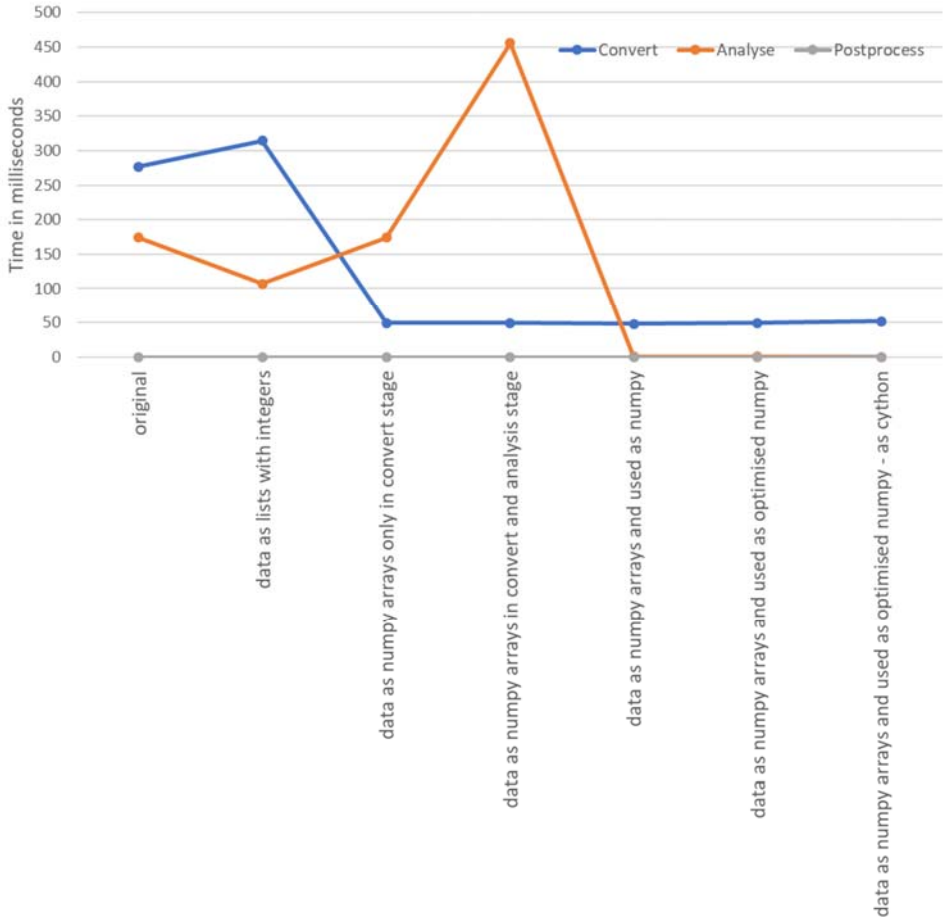
   C. Storing the signal as Nympy arrays, during the Conversion and Analysis functions

   D. Storing the signal as Nympy arrays, and make use of Numpy operations

   E. Storing the signal as Nympy arrays, and attempting to optimize Numpy operations

   F. Converting it all into Cython

Result



Summary

After both ups and downs I did indeed manage to speed up the code;

- the Conversion part was somewhat improved
- the Analysis part was dramatically improved
- the Post treatment part I didn't care to attempt to improve, but it was improved in Cython.

From the original total time of about 451 ms to a final 50 ms, so 9 times faster!

Well, that's when line profiling is used. Using the times as printed by the code itself shows only some 44% reduction of time. From a total of about 85 ms to about 49 ms. Still an improvement, but a moderate one.

Profiling was new to me and I used that a lot during the project. Mostly profiling and line-profiling in the Spyder environment where I also used %timeit many times to test code snippet and for profiling of the Cython code.

Taking advantage of Numpy was also new to me. It dramatically reduced the time taken for analysis (some 50 times faster). But gave only some improvement in the Conversion part. The Conversion part was already after strategy C very short slimmer than the original. So I could not find any way to improve that further.

Also the possibilities with Cython was new to me. I must admit I was hoping for a dramatic change with strategy F. But apparently the Numpy conversion step in strategy C was already very optimized and Cython couldn't improve on that. The Analysis function was actually somewhat improved, but as the Analysis function had become very fast compared to the Conversion function already after the optimised Numpy strategy D, Cythonizing had only a marginal effect on the total time.

Along with these improvements I also got an overall better structure of the code (and a somewhat better documented code).

Work ahead

The conversion function is now the only real time-consuming part of the code. Data collection and storage of the signal were placed outside the scope of the project, although perhaps it should have been included. So now, an efficient data collection and storage without intermediate storage as a Python list would be the next thing to target. But that is another story.

Attached as separate files are

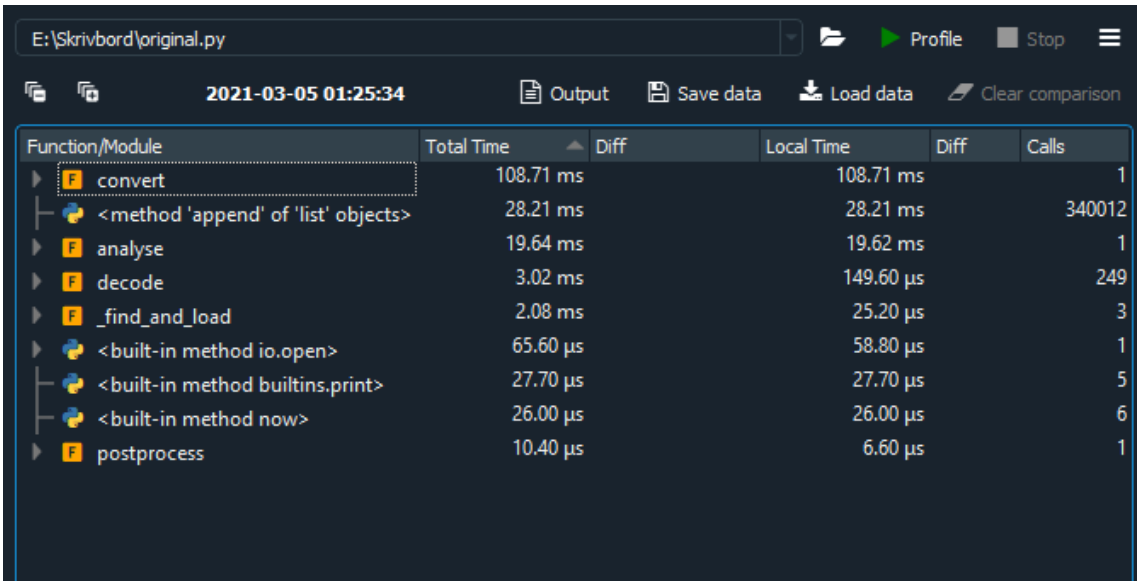
- Original code
- Code after Strategy E (best Python version)
- The file with the signal

```python
from datetime import datetime

""" converts radiosignal to binary strings"""


def convert(signal):
    signalTime=float(signal[0][0])
    for i in range(len(signal[0])):
        signal[0][i] = float(signal[0][i]) - signalTime
        signal[1][i] = int(signal[1][i])
    return(signal)


def analyse(signal):
    n = len(signal[0]) - 1

    while signal[1][n] == 0 and n>0: n = n - 1
    up = signal[0][n]
    m = n
    while signal[1][m] == 1 and m>0: m = m - 1
    high = n - m

    binary=''
    n = n - int(high/2)

    warnings = 0
    first = 0
    tooManyErrors = False

    while len(binary)<250 and n>20 and warnings<3:
        if warnings > len(binary): tooManyErrors = True
        up = signal[0][n]
        while signal[1][n] == 1 or signal[1][n-1] == 1: n = n - 1
        down = signal[0][n]
        while signal[1][n] == 0 or signal[1][n-1] == 0: n = n - 1
        up = signal[0][n]
        lowTime = down - up

        if lowTime >= 0.0016 and lowTime <= 0.0025:
            binary = '0' + binary
            continue
        elif lowTime >= 0.0037 and lowTime < 0.0045:
            binary = '1' + binary
            continue
        elif lowTime >= 0.0083 and lowTime < 0.0089:
            binary = '_' + binary
            continue
        else:
            warnings = warnings + 1
            binary = 'w' + binary
            if warnings == 1: first = int(10000*lowTime)

    longBinary = binary
    binaries=binary.split('_')
    whichToUse = len(binaries)
    binary = binaries[whichToUse-1]
    if 'w' in binary:
        binary = ''
    while len(binary) != 36 and whichToUse>=0:
        binary = binaries[whichToUse-1]
        if len(binary)>36:
            binary = binary[:36]
        if 'w' in binary:
            binary = ''
        whichToUse = whichToUse - 1
    return(binary, longBinary, tooManyErrors)


def postprocess(binary, longBinary, tooManyErrors):
    if len(binary) == 36 and tooManyErrors == False:
        word1 = binary[19:28]
        temperature= int(word1,2)/10.0
        word2 = binary[28:36]
        moisture = int(word2,2)
        print(longBinary)
        print(binary,'translates into'+' '*(4-len(str(temperature))), temperature,'grader C' , moisture ,'
    else:
        print('Communication error')


""" main """

signal=[[],[]]
with open("./radiosignaler/24 3 signal2021-03-02 11-16-50.636838.txt","r") as f:
    for data in f:
        signal[0].append(data[:-3])
        signal[1].append(data[-2:-1])

starttime = datetime.now()
convertedSignal = convert(signal)
print('Time for conversion: ', datetime.now()-starttime)
starttime = datetime.now()
binary, longBinary, tooManyErrors = analyse(convertedSignal)
print('Time for analysis: ', datetime.now()-starttime)
starttime = datetime.now()
postprocess(binary, longBinary, tooManyErrors)
print('Time for postprocessing: ', datetime.now()-starttime)
```

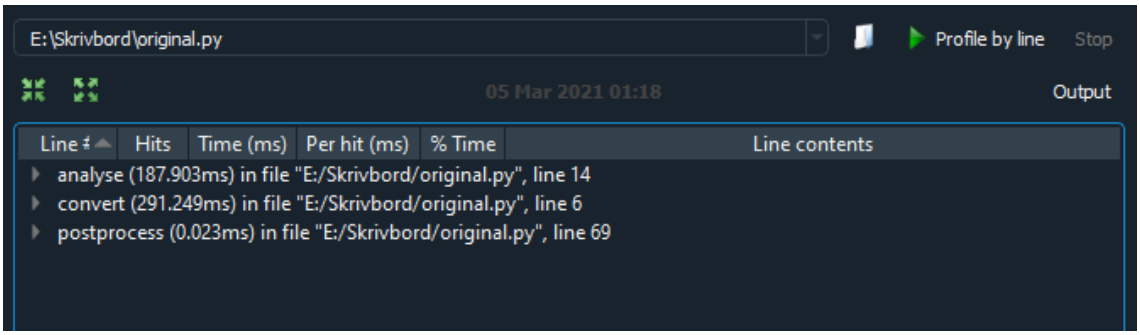
Comments on the Original code

Profiling and line-profiling showed that converting the signal was the most time-consuming part, but also the Analyse function took considerable time.

Profiling of the Original code:



Line profiling of the Original code

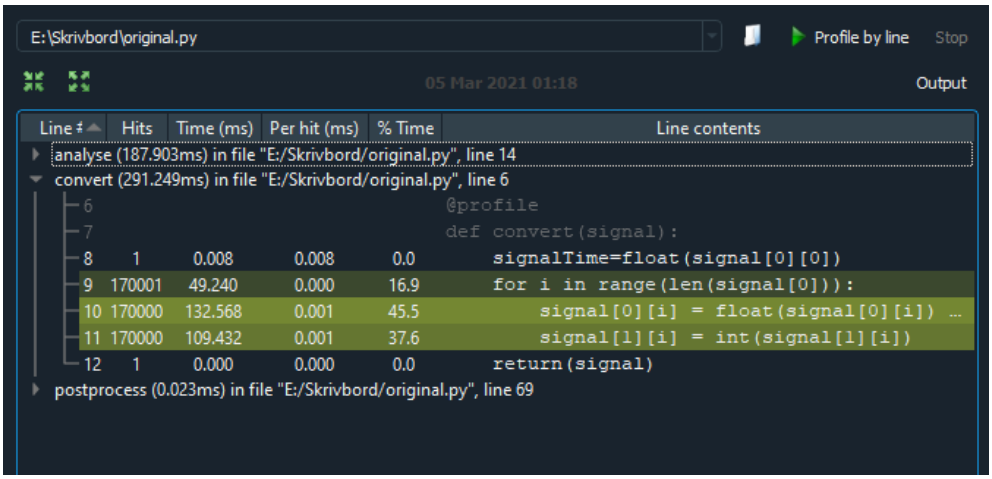


Best vales (of 10) when line profiling the Original code:

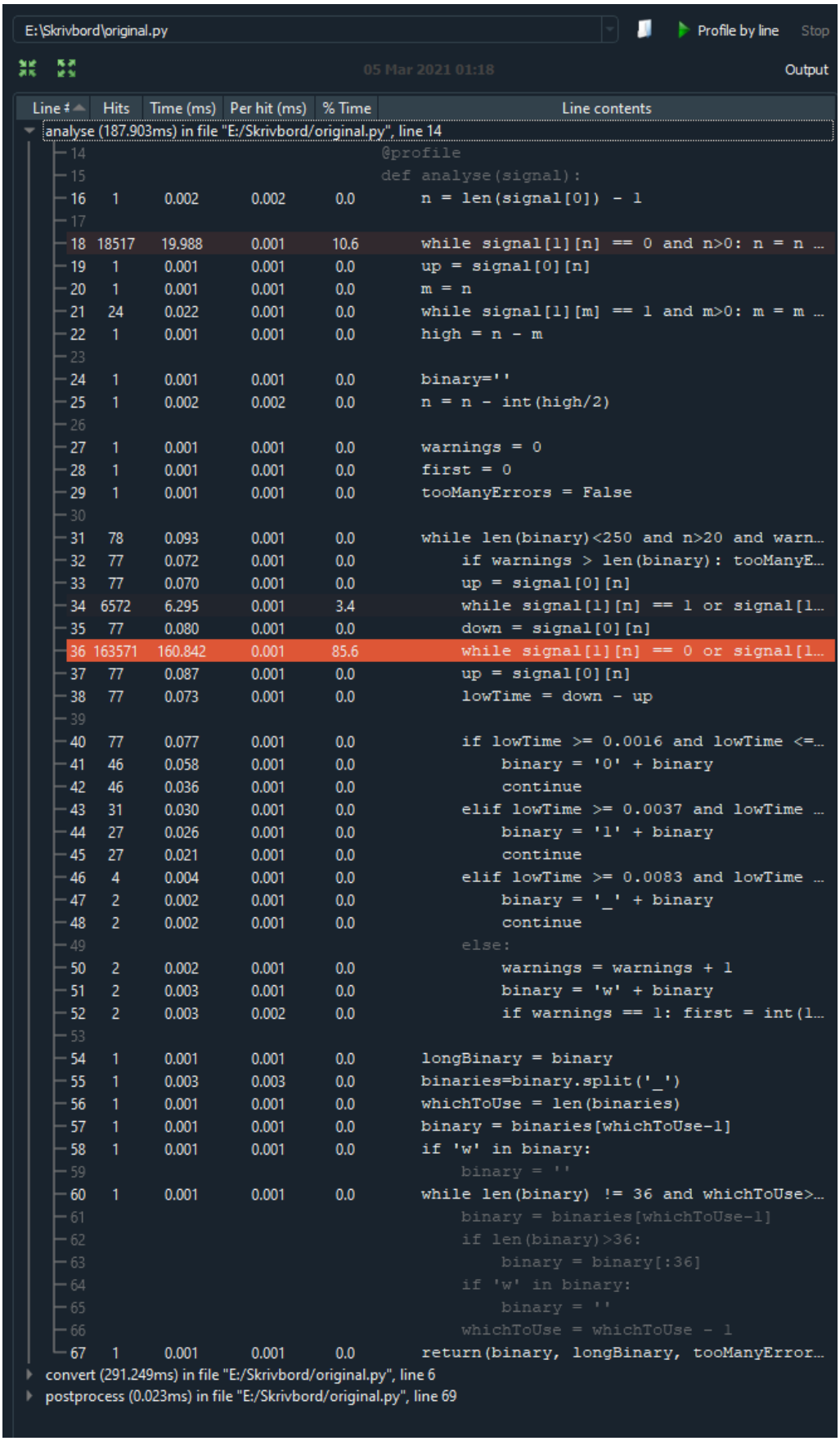
Convert function:	277 ms
Analyse function:	174 ms
Post process function:	0.02 ms

The Convert and Analyse functions are pretty similar considering time consumption. And apparently, the Post process function doesn't need/deserve further attention.

Line profiling of the **Convert function of the Original code** showed that the lines responsible for zeroing the time in signal[0] and making sure the data signal in signal[1] was an integer took more than 73% of the time. 170000 loops take some time!



Line profiling of the **Analyse function of the Original code** showed that stepping along the Python list of signals took more than 99% of the time.



Line #	Hits	Time (ms)	Per hit (ms)	% Time	Line contents
analyse (187.903ms) in file "E:/Skrivbord/original.py", line 14					
14					@profile
15					def analyse(signal):
16	1	0.002	0.002	0.0	n = len(signal[0]) - 1
17					
18	18517	19.988	0.001	10.6	while signal[1][n] == 0 and n>0: n = n …
19	1	0.001	0.001	0.0	up = signal[0][n]
20	1	0.001	0.001	0.0	m = n
21	24	0.022	0.001	0.0	while signal[1][m] == 1 and m>0: m = m …
22	1	0.001	0.001	0.0	high = n - m
23					
24	1	0.001	0.001	0.0	binary=''
25	1	0.002	0.002	0.0	n = n - int(high/2)
26					
27	1	0.001	0.001	0.0	warnings = 0
28	1	0.001	0.001	0.0	first = 0
29	1	0.001	0.001	0.0	tooManyErrors = False
30					
31	78	0.093	0.001	0.0	while len(binary)<250 and n>20 and warn…
32	77	0.072	0.001	0.0	if warnings > len(binary): tooManyE…
33	77	0.070	0.001	0.0	up = signal[0][n]
34	6572	6.295	0.001	3.4	while signal[1][n] == 1 or signal[1…
35	77	0.080	0.001	0.0	down = signal[0][n]
36	163571	160.842	0.001	85.6	while signal[1][n] == 0 or signal[1…
37	77	0.087	0.001	0.0	up = signal[0][n]
38	77	0.073	0.001	0.0	lowTime = down - up
39					
40	77	0.077	0.001	0.0	if lowTime >= 0.0016 and lowTime <=…
41	46	0.058	0.001	0.0	binary = '0' + binary
42	46	0.036	0.001	0.0	continue
43	31	0.030	0.001	0.0	elif lowTime >= 0.0037 and lowTime …
44	27	0.026	0.001	0.0	binary = '1' + binary
45	27	0.021	0.001	0.0	continue
46	4	0.004	0.001	0.0	elif lowTime >= 0.0083 and lowTime …
47	2	0.002	0.001	0.0	binary = '_' + binary
48	2	0.002	0.001	0.0	continue
49					else:
50	2	0.002	0.001	0.0	warnings = warnings + 1
51	2	0.003	0.001	0.0	binary = 'w' + binary
52	2	0.003	0.002	0.0	if warnings == 1: first = int(1…
53					
54	1	0.001	0.001	0.0	longBinary = binary
55	1	0.003	0.003	0.0	binaries=binary.split('_')
56	1	0.001	0.001	0.0	whichToUse = len(binaries)
57	1	0.001	0.001	0.0	binary = binaries[whichToUse-1]
58	1	0.001	0.001	0.0	if 'w' in binary:
59					binary = ''
60	1	0.001	0.001	0.0	while len(binary) != 36 and whichToUse>…
61					binary = binaries[whichToUse-1]
62					if len(binary)>36:
63					binary = binary[:36]
64					if 'w' in binary:
65					binary = ''
66					whichToUse = whichToUse - 1
67	1	0.001	0.001	0.0	return(binary, longBinary, tooManyError…
convert (291.249ms) in file "E:/Skrivbord/original.py", line 6					
postprocess (0.023ms) in file "E:/Skrivbord/original.py", line 69					

Code changes after strategy A

The changes were small. Only three rows (8, 10 and 11) to use int versions of signal values and times in seconds* 10^6

```
 7 def convert(signal):
 8     signalTime=int(float(signal[0][0])*10**6)
 9     for i in range(len(signal[0])):
10         signal[0][i] = int(float(signal[0][i])*10**6) - signalTime
11         signal[1][i] = int(signal[1][i])
12     return(signal)
```

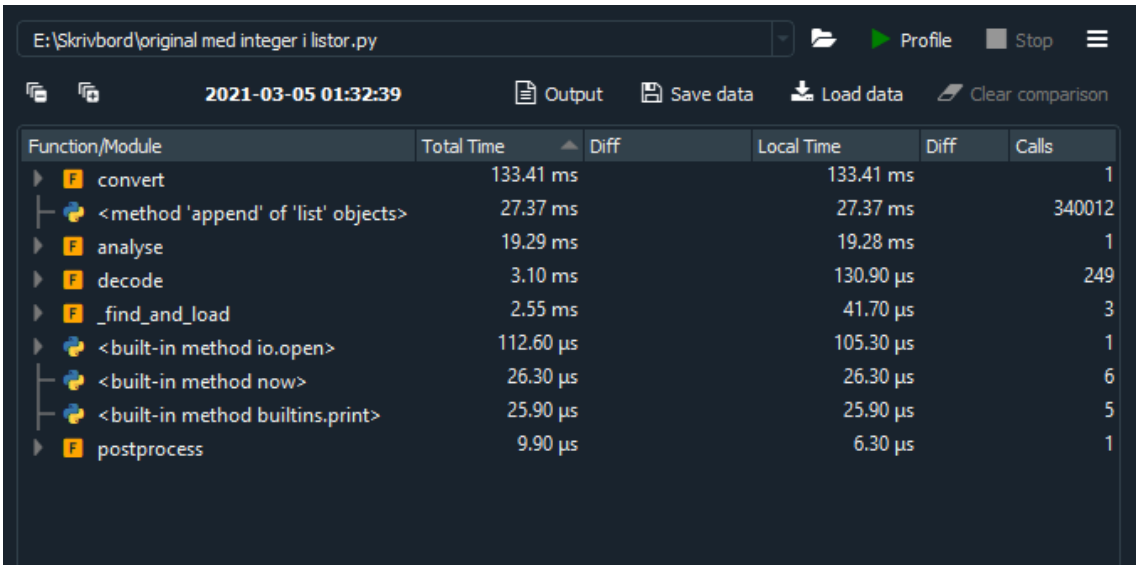
And some needed changes to have the int-version of times.

```
40         if lowTime >= 1600 and lowTime <= 2500:
41             binary = '0' + binary
42             continue
43         elif lowTime >= 3700 and lowTime < 4500:
44             binary = '1' + binary
45             continue
46         elif lowTime >= 8300 and lowTime < 8900:
47             binary = '_' + binary
```

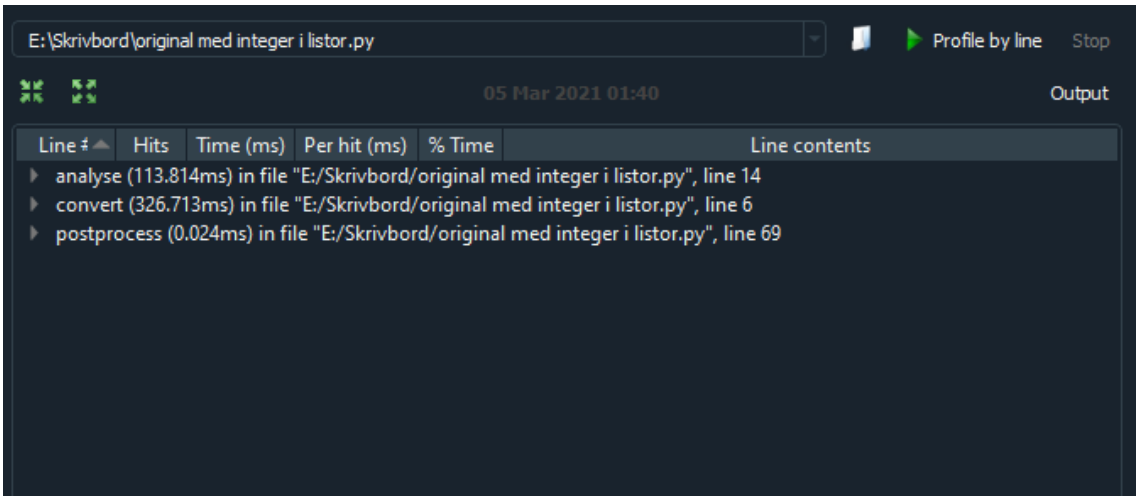
Comments on the timings after strategy A

Profiling and line-profiling showed that the time for conversion is now clearly the slowest part.

Profiling of the code:



Line profiling of the code



Best vales (of 10) when line profiling:

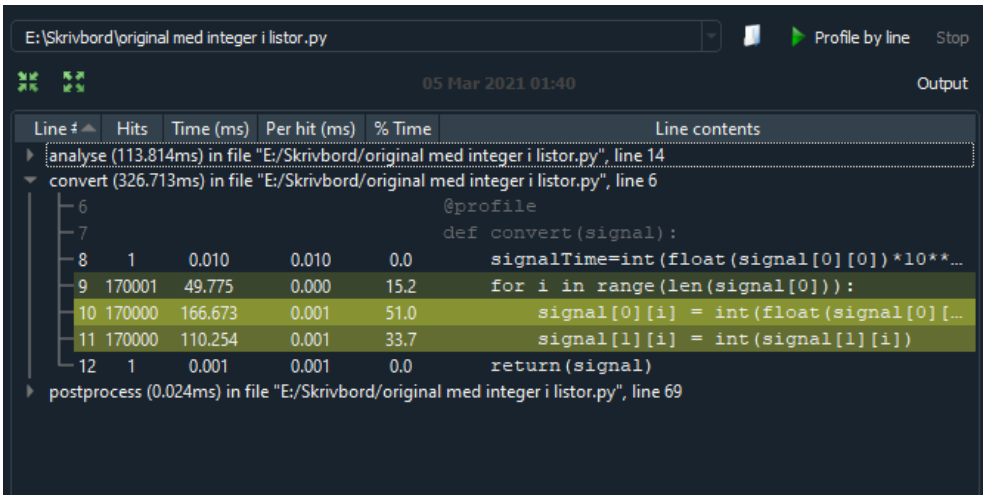
Convert function: 314 ms

Analyse function: 107 ms

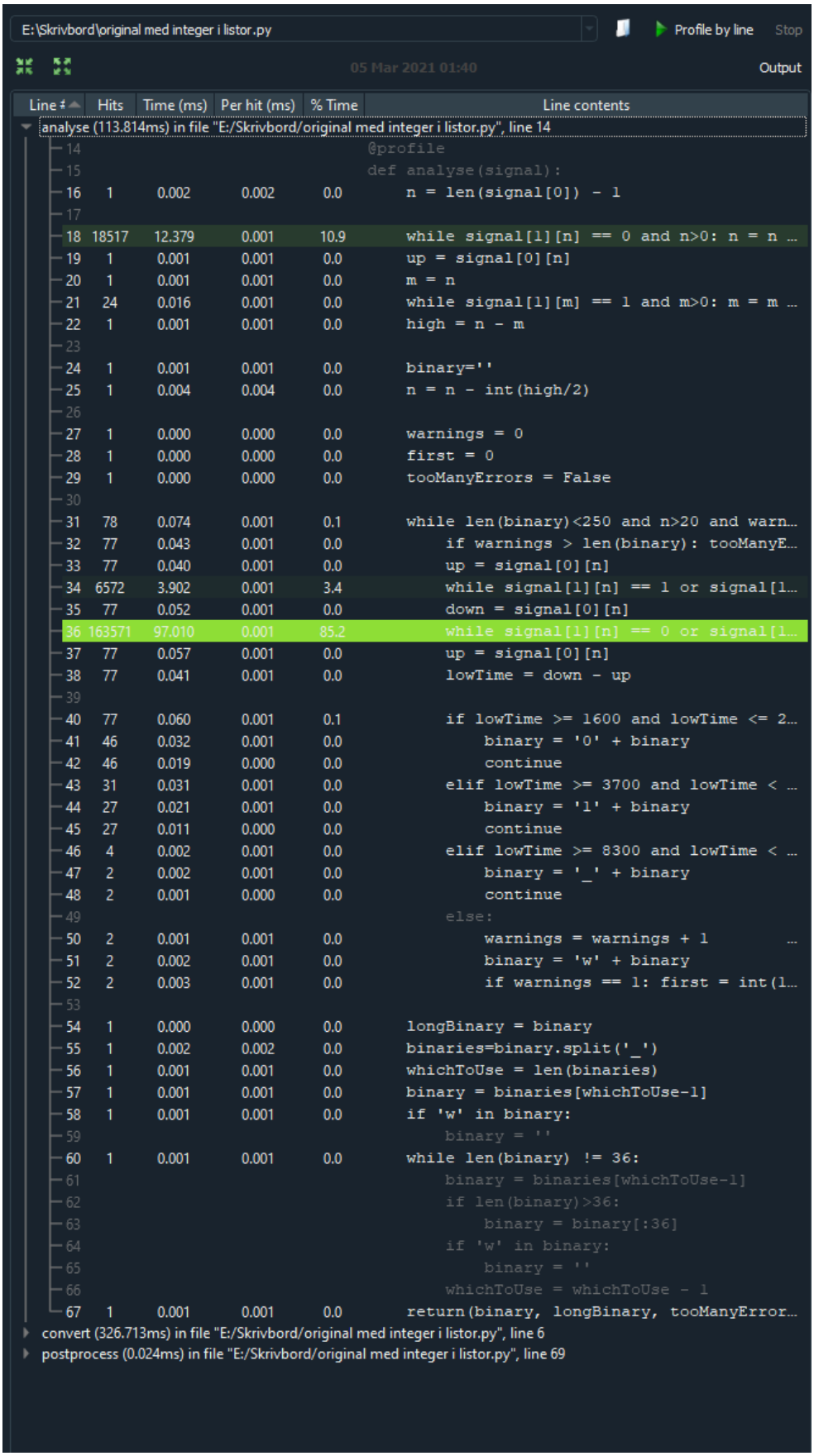
Post process function: 0.02 ms

The Convert function was significantly slowed down by having to convert to int. The Analyse function was faster using int, but in total the benefit was not convincing..

Line profiling of the **Convert function of the code after Strategy A** showed making the time list integers required an extra 37 ms on top of the time to just subtract the start time and leaving it as float.



Line profiling of the **Analyse function of the code after Strategy A** showed that although overall faster when using integers, the very same two lines take most time.



Code changes after strategy B

Since Strategy A wasn't very successful the program was modified to use Numpy arrays instead of lists.

I added an import line

```
3 import numpy as np
```

The Convert function became

```
 8 def convert(signal):
 9     timeArray = np.array(signal[0],dtype=float)
10     signalArray = np.array(signal[1], dtype='b')
11     timeArray += timeArray[0]
```

Since I wanted to see the effect of this alone I had to temporarily add some lines to do the original convert function, but outside the function, but before calling the Analyse function.

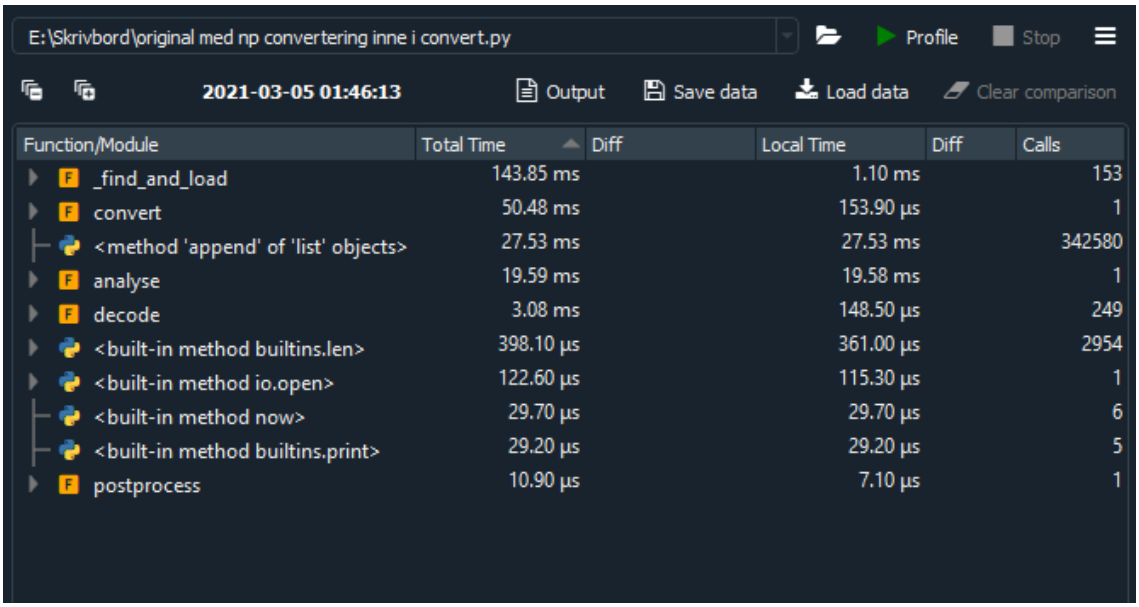
I added the lines 92-95

```
 89 starttime = datetime.now()
 90 convertedSignal = convert(signal)
 91 print('Time for conversion: ', datetime.now()-starttime)
 92 signalTime=float(signal[0][0])
 93 for i in range(len(signal[0])):
 94     signal[0][i] = float(signal[0][i]) - signalTime
 95     signal[1][i] = int(signal[1][i])
 96 starttime = datetime.now()
 97 binary, longBinary, tooManyErrors = analyse(signal)
 98 print('Time for analysis: ', datetime.now()-starttime)
 99 starttime = datetime.now()
100 postprocess(binary, longBinary, tooManyErrors)
101 print('Time for postprocessing: ', datetime.now()-starttime)
```

Comments on the timings after strategy B

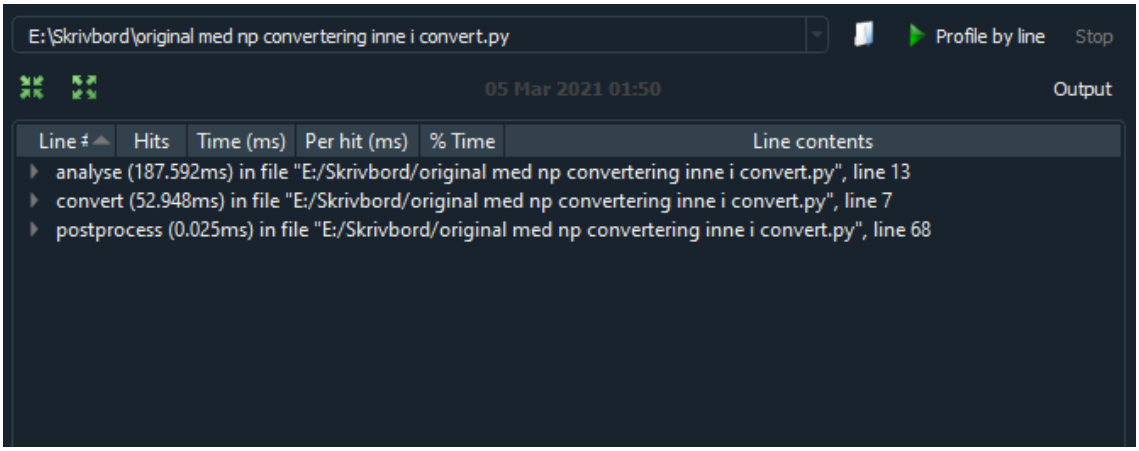
Profiling and line-profiling showed that the Convert function was now much faster, six times faster. As the strategy A was abandoned, the timings for the Analyse function was back to the original

Profiling of the code:



Function/Module	Total Time	Diff	Local Time	Diff	Calls
_find_and_load	143.85 ms		1.10 ms		153
convert	50.48 ms		153.90 µs		1
<method 'append' of 'list' objects>	27.53 ms		27.53 ms		342580
analyse	19.59 ms		19.58 ms		1
decode	3.08 ms		148.50 µs		249
<built-in method builtins.len>	398.10 µs		361.00 µs		2954
<built-in method io.open>	122.60 µs		115.30 µs		1
<built-in method now>	29.70 µs		29.70 µs		6
<built-in method builtins.print>	29.20 µs		29.20 µs		5
postprocess	10.90 µs		7.10 µs		1

Line profiling of the code



Best vales (of 10) when line profiling:

Convert function: 49 ms

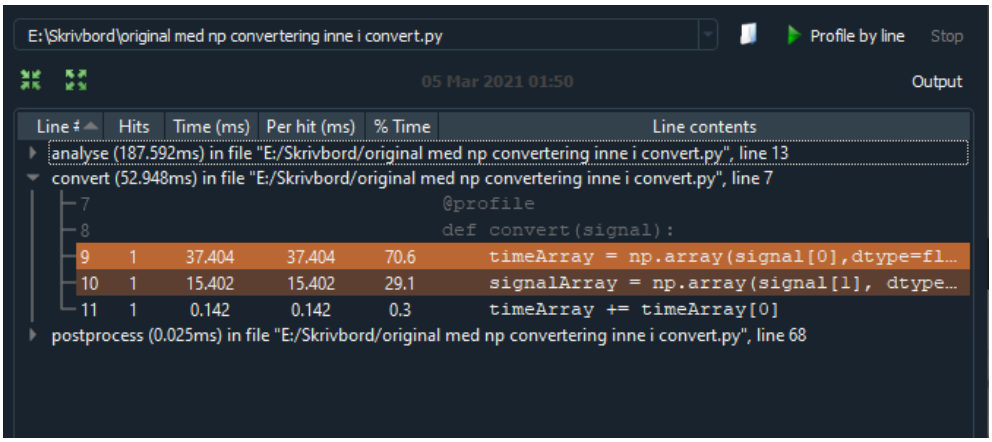
Analyse function: 174 ms (back to the original)

Post process function: 0.01 ms

The Convert function was significantly faster, but the rest of the code was the original.

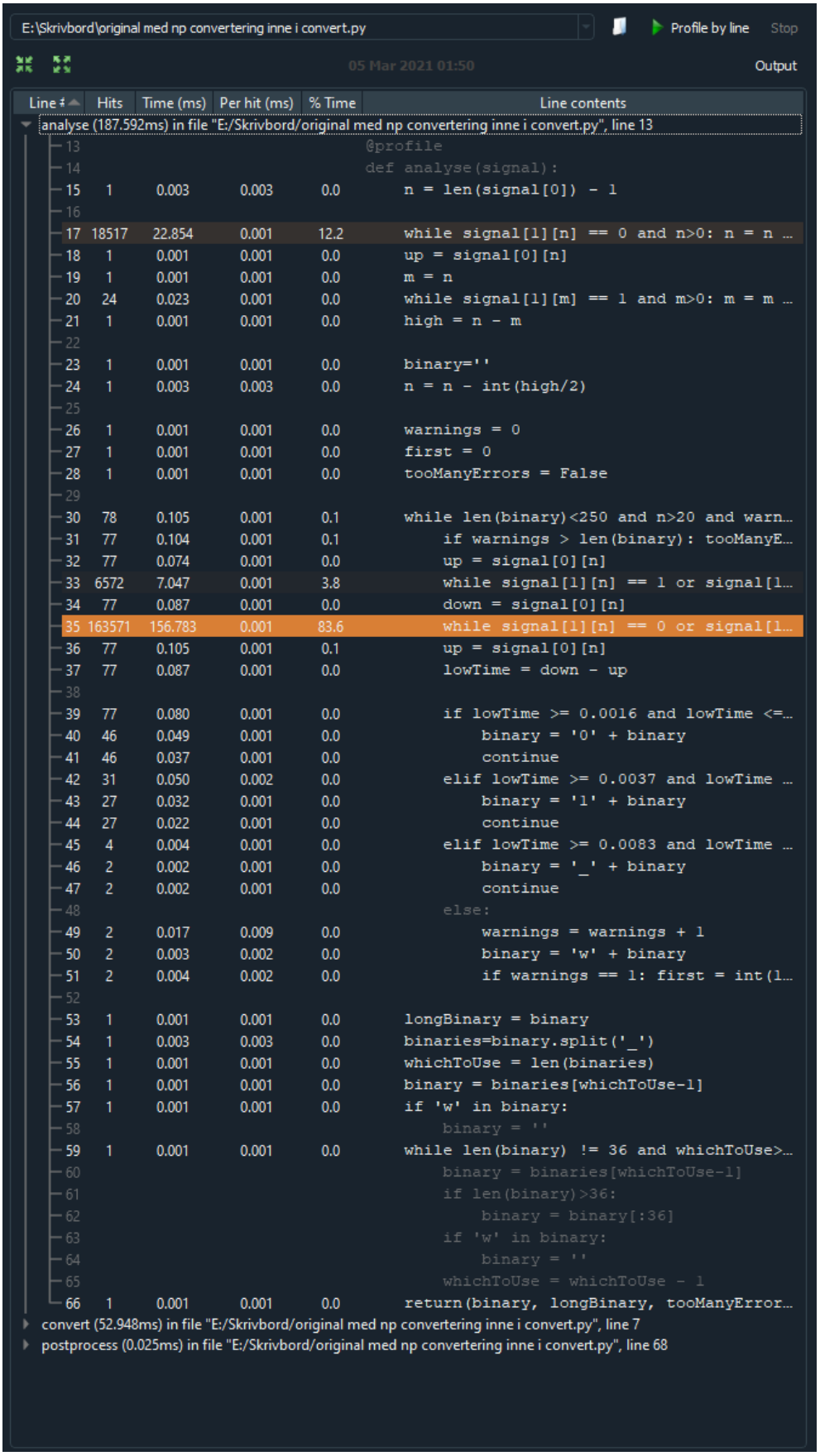
Promising though!

Line profiling of the **Convert function of the code after Strategy B** showed that getting the data into Numpy arrays was quicker than making lists of them. I'm not aware of any way to make this even quicker. The code is very slimmed as I see it.



(Now, when producing the report, I realize it should be `-=` instead of `+=`, but since the data in my test signal had the first time already being 0, I didn't notice anything wrong. When I take the code back to the application I will have to change that)

Line profiling of the **Analyse function of the code after Strategy B**.Since this function and the input to the function was identical as the original code the timings should be similar, and indeed they are.



analyse (187.592ms) in file "E:/Skrivbord/original med np convertering inne i convert.py", line 13

Line #	Hits	Time (ms)	Per hit (ms)	% Time	Line contents
13					@profile
14					def analyse(signal):
15	1	0.003	0.003	0.0	n = len(signal[0]) - 1
16					
17	18517	22.854	0.001	12.2	while signal[1][n] == 0 and n>0: n = n …
18	1	0.001	0.001	0.0	up = signal[0][n]
19	1	0.001	0.001	0.0	m = n
20	24	0.023	0.001	0.0	while signal[1][m] == 1 and m>0: m = m …
21	1	0.001	0.001	0.0	high = n - m
22					
23	1	0.001	0.001	0.0	binary=''
24	1	0.003	0.003	0.0	n = n - int(high/2)
25					
26	1	0.001	0.001	0.0	warnings = 0
27	1	0.001	0.001	0.0	first = 0
28	1	0.001	0.001	0.0	tooManyErrors = False
29					
30	78	0.105	0.001	0.1	while len(binary)<250 and n>20 and warn…
31	77	0.104	0.001	0.1	if warnings > len(binary): tooManyE…
32	77	0.074	0.001	0.0	up = signal[0][n]
33	6572	7.047	0.001	3.8	while signal[1][n] == 1 or signal[1…
34	77	0.087	0.001	0.0	down = signal[0][n]
35	163571	156.783	0.001	83.6	while signal[1][n] == 0 or signal[1…
36	77	0.105	0.001	0.1	up = signal[0][n]
37	77	0.087	0.001	0.0	lowTime = down - up
38					
39	77	0.080	0.001	0.0	if lowTime >= 0.0016 and lowTime <=…
40	46	0.049	0.001	0.0	binary = '0' + binary
41	46	0.037	0.001	0.0	continue
42	31	0.050	0.002	0.0	elif lowTime >= 0.0037 and lowTime …
43	27	0.032	0.001	0.0	binary = '1' + binary
44	27	0.022	0.001	0.0	continue
45	4	0.004	0.001	0.0	elif lowTime >= 0.0083 and lowTime …
46	2	0.002	0.001	0.0	binary = '_' + binary
47	2	0.002	0.001	0.0	continue
48					else:
49	2	0.017	0.009	0.0	warnings = warnings + 1
50	2	0.003	0.002	0.0	binary = 'w' + binary
51	2	0.004	0.002	0.0	if warnings == 1: first = int(1…
52					
53	1	0.001	0.001	0.0	longBinary = binary
54	1	0.003	0.003	0.0	binaries=binary.split('_')
55	1	0.001	0.001	0.0	whichToUse = len(binaries)
56	1	0.001	0.001	0.0	binary = binaries[whichToUse-1]
57	1	0.001	0.001	0.0	if 'w' in binary:
58					binary = ''
59	1	0.001	0.001	0.0	while len(binary) != 36 and whichToUse>…
60					binary = binaries[whichToUse-1]
61					if len(binary)>36:
62					binary = binary[:36]
63					if 'w' in binary:
64					binary = ''
65					whichToUse = whichToUse - 1
66	1	0.001	0.001	0.0	return(binary, longBinary, tooManyError…

convert (52.948ms) in file "E:/Skrivbord/original med np convertering inne i convert.py", line 7

postprocess (0.025ms) in file "E:/Skrivbord/original med np convertering inne i convert.py", line 68

Code changes after strategy C

Beginning to introduce Numpy arrays was promising. I now tried to use them as input in the Analyse function (purposely not changing the code too much, i.e I was not taking advantage of them just yet).

I let the Convert function return the two Numpy arrays

```
12      return(signalArray, timeArray)
```

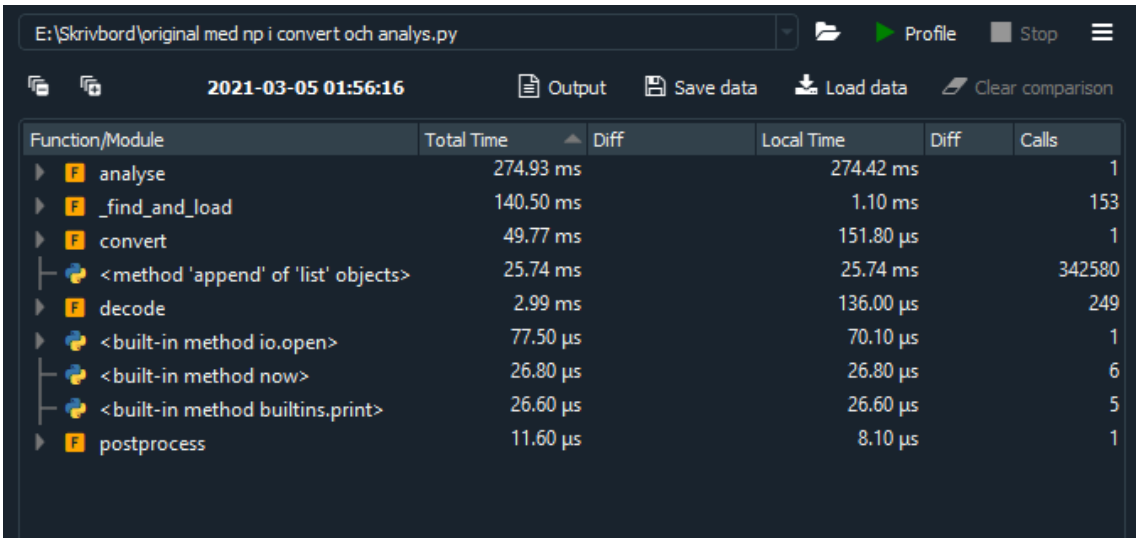
And adjusted the lines 15-17, 20-21, 23-24, 34-38 to use Numpy arrays.

```
15  def analyse(signalArray, timeArray):
16      n = timeArray.size - 1
17      stepArray = signalArray - np.append(np.delete(signalArray,0,None),0)
18      n -= 1
19
20      while signalArray[n] == 0 and n>0: n -= 1
21      up = timeArray[n]
22      m = n
23      while signalArray[m] == 1 and m>0: m -= 1
24      high = n - m
25      binary=''
26      n = n - int(high/2)
27
28      warnings = 0
29      first = 0
30      tooManyErrors = False
31
32      while len(binary)<250 and n>20 and warnings<3:
33          if warnings > len(binary): tooManyErrors = True
34          up = signalArray[n]
35          while signalArray[n] == 1 or signalArray[n-1] == 1: n -= 1
36          down = timeArray[n]
37          while signalArray[n] == 0 or signalArray[n-1] == 0: n -= 1
38          up = timeArray[n]
39          lowTime = down - up
```

Comments on the timings after strategy C

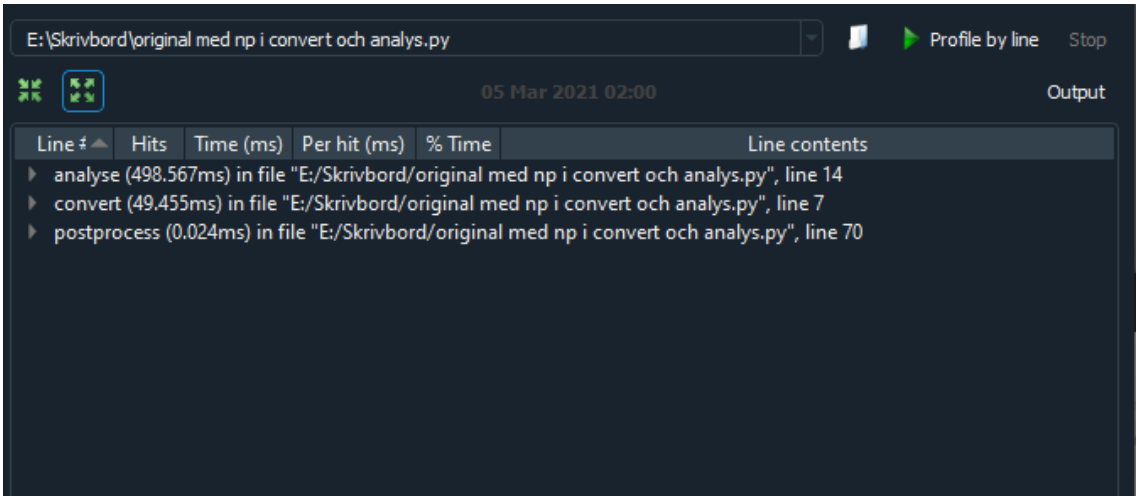
Profiling and line-profiling showed that feeding Numpy arrays into the Analyse function without properly handling them as Numpy array is a very bad idea. Of course I suspected so, but it was interesting to see just how bad idea that was.

Profiling of the code:



Function/Module	Total Time	Diff	Local Time	Diff	Calls
analyse	274.93 ms		274.42 ms		1
_find_and_load	140.50 ms		1.10 ms		153
convert	49.77 ms		151.80 µs		1
<method 'append' of 'list' objects>	25.74 ms		25.74 ms		342580
decode	2.99 ms		136.00 µs		249
<built-in method io.open>	77.50 µs		70.10 µs		1
<built-in method now>	26.80 µs		26.80 µs		6
<built-in method builtins.print>	26.60 µs		26.60 µs		5
postprocess	11.60 µs		8.10 µs		1

Line profiling of the code



Best vales (of 10) when line profiling:

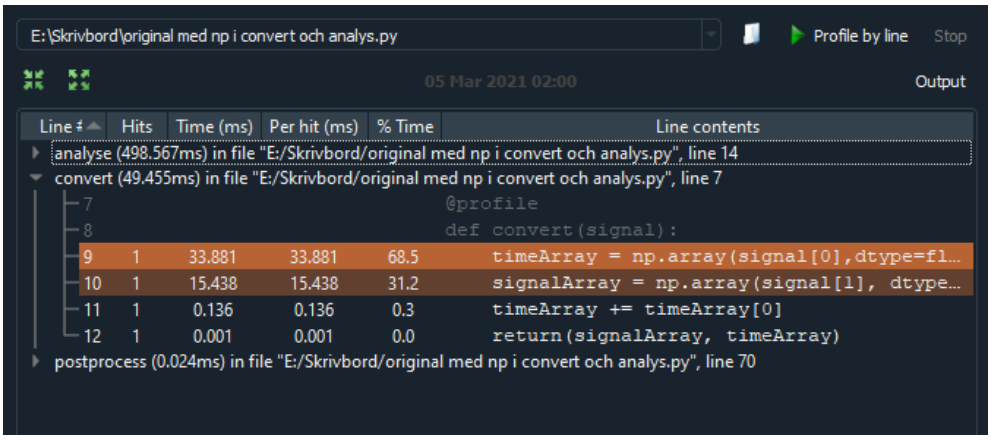
Convert function: 49 ms (same code and same timings as Strategy B)

Analyse function: 456 ms

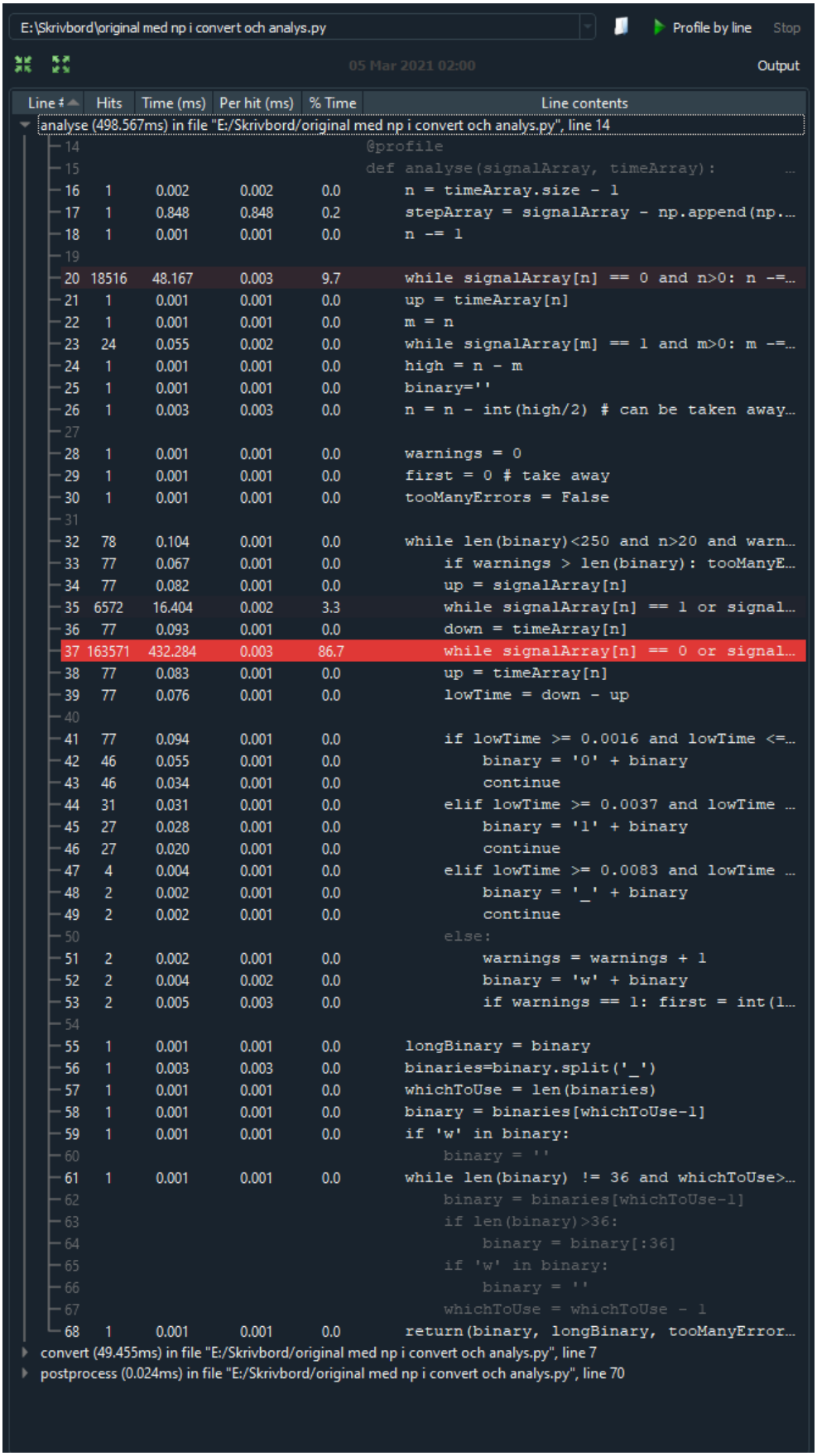
Post process function: 0.02 ms

The Analyse function took more than twice the original time. I definitely have to make use of the Numpy arrays, otherwise it's of no use.

Line profiling of the **Convert function of the code after Strategy C** of course gave the same result as in strategy B – it's the same code.



Line profiling of the **Analyse function of the code after Strategy C** showed that stepping along the Numpy arrays is no better (actually worse) than stepping along a list. The lines that take most time are the same as in the original code.



Line #	Hits	Time (ms)	Per hit (ms)	% Time	Line contents
					analyse (498.567ms) in file "E:/Skrivbord/original med np i convert och analys.py", line 14
14					@profile
15					def analyse(signalArray, timeArray): …
16	1	0.002	0.002	0.0	n = timeArray.size - 1
17	1	0.848	0.848	0.2	stepArray = signalArray - np.append(np.…
18	1	0.001	0.001	0.0	n -= 1
19					
20	18516	48.167	0.003	9.7	while signalArray[n] == 0 and n>0: n -=…
21	1	0.001	0.001	0.0	up = timeArray[n]
22	1	0.001	0.001	0.0	m = n
23	24	0.055	0.002	0.0	while signalArray[m] == 1 and m>0: m -=…
24	1	0.001	0.001	0.0	high = n - m
25	1	0.001	0.001	0.0	binary=''
26	1	0.003	0.003	0.0	n = n - int(high/2) # can be taken away…
27					
28	1	0.001	0.001	0.0	warnings = 0
29	1	0.001	0.001	0.0	first = 0 # take away
30	1	0.001	0.001	0.0	tooManyErrors = False
31					
32	78	0.104	0.001	0.0	while len(binary)<250 and n>20 and warn…
33	77	0.067	0.001	0.0	if warnings > len(binary): tooManyE…
34	77	0.082	0.001	0.0	up = signalArray[n]
35	6572	16.404	0.002	3.3	while signalArray[n] == 1 or signal…
36	77	0.093	0.001	0.0	down = timeArray[n]
37	163571	432.284	0.003	86.7	while signalArray[n] == 0 or signal…
38	77	0.083	0.001	0.0	up = timeArray[n]
39	77	0.076	0.001	0.0	lowTime = down - up
40					
41	77	0.094	0.001	0.0	if lowTime >= 0.0016 and lowTime <=…
42	46	0.055	0.001	0.0	binary = '0' + binary
43	46	0.034	0.001	0.0	continue
44	31	0.031	0.001	0.0	elif lowTime >= 0.0037 and lowTime …
45	27	0.028	0.001	0.0	binary = '1' + binary
46	27	0.020	0.001	0.0	continue
47	4	0.004	0.001	0.0	elif lowTime >= 0.0083 and lowTime …
48	2	0.002	0.001	0.0	binary = '_' + binary
49	2	0.002	0.001	0.0	continue
50					else:
51	2	0.002	0.001	0.0	warnings = warnings + 1
52	2	0.004	0.002	0.0	binary = 'w' + binary
53	2	0.005	0.003	0.0	if warnings == 1: first = int(l…
54					
55	1	0.001	0.001	0.0	longBinary = binary
56	1	0.003	0.003	0.0	binaries=binary.split('_')
57	1	0.001	0.001	0.0	whichToUse = len(binaries)
58	1	0.001	0.001	0.0	binary = binaries[whichToUse-1]
59	1	0.001	0.001	0.0	if 'w' in binary:
60					binary = ''
61	1	0.001	0.001	0.0	while len(binary) != 36 and whichToUse>…
62					binary = binaries[whichToUse-1]
63					if len(binary)>36:
64					binary = binary[:36]
65					if 'w' in binary:
66					binary = ''
67					whichToUse = whichToUse - 1
68	1	0.001	0.001	0.0	return(binary, longBinary, tooManyError…
					convert (49.455ms) in file "E:/Skrivbord/original med np i convert och analys.py", line 7
					postprocess (0.024ms) in file "E:/Skrivbord/original med np i convert och analys.py", line 70

Code changes after strategy D

To take advantage of Numpy line 17 was changed. That single line replaced all stepping along lists or Numpy arrays that has so far been very time consuming. The outcome, stepTimes is now a short Numpy array only containing the times when the signal is changed from 1 to 0 or from 0 to 1.

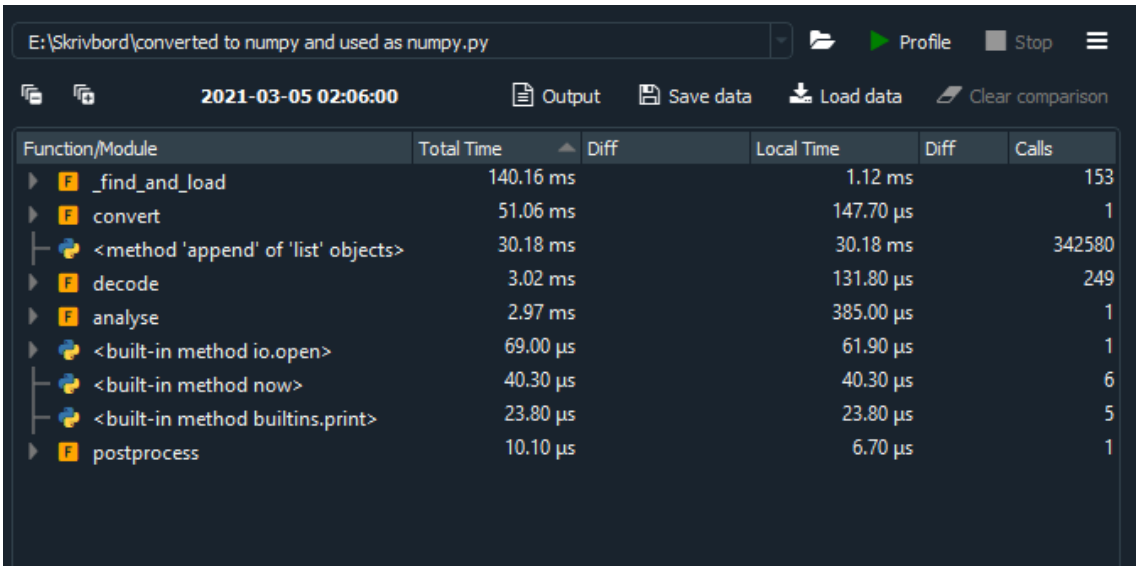
The while loop was changed into a for loop in line 24 and lowTime was easily calculated from the difference in stepTimes.

```
15 def analyse(signalArray, timeArray):
16
17     stepTimes = timeArray[np.logical_xor(signalArray, np.roll(signalArray, 1))]
18
19     warnings = 0
20     binary = ''
21     tooManyErrors = False
22     n = stepTimes.size
23
24     for i in range(n-1, 0, -2):
25         lowTime = stepTimes[i-1]-stepTimes[i-2]
26         if lowTime >= 0.0016 and lowTime <= 0.0025:
27             binary = '0' + binary
28             continue
29         elif lowTime >= 0.0037 and lowTime < 0.0045:
30             binary = '1' + binary
31             continue
32         elif lowTime >= 0.0083 and lowTime < 0.0089:
33             binary = '_' + binary
34             continue
35         else:
36             warnings = warnings + 1
```

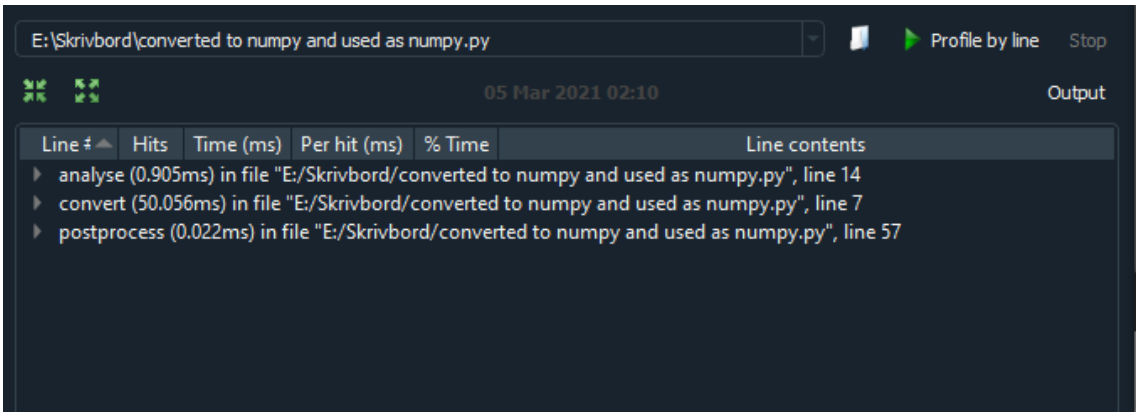
Comments on the timings after strategy D

Profiling and line-profiling showed that using Numpy in a proper way was extremely efficient.

Profiling of the code:



Line profiling of the code



Best vales (of 10) when line profiling:

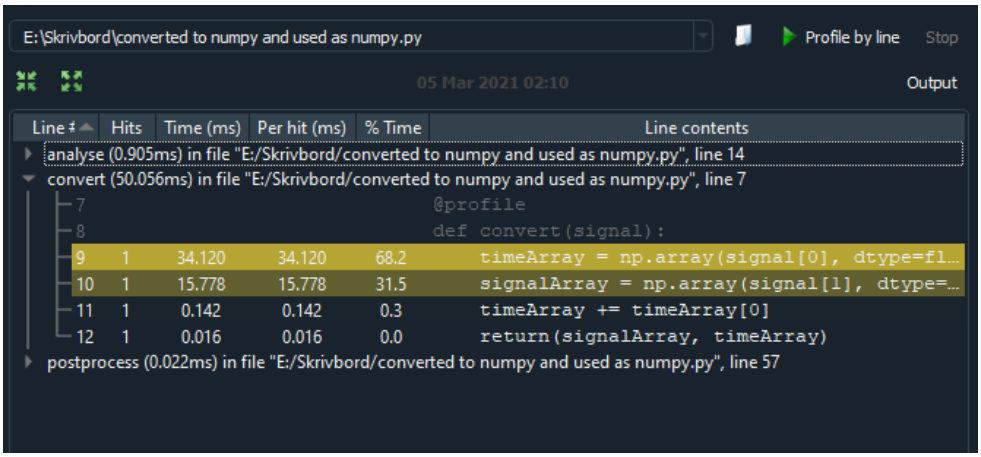
Convert function: 48 ms (same code and same timings as Strategy B and C)

Analyse function: 0.86 ms

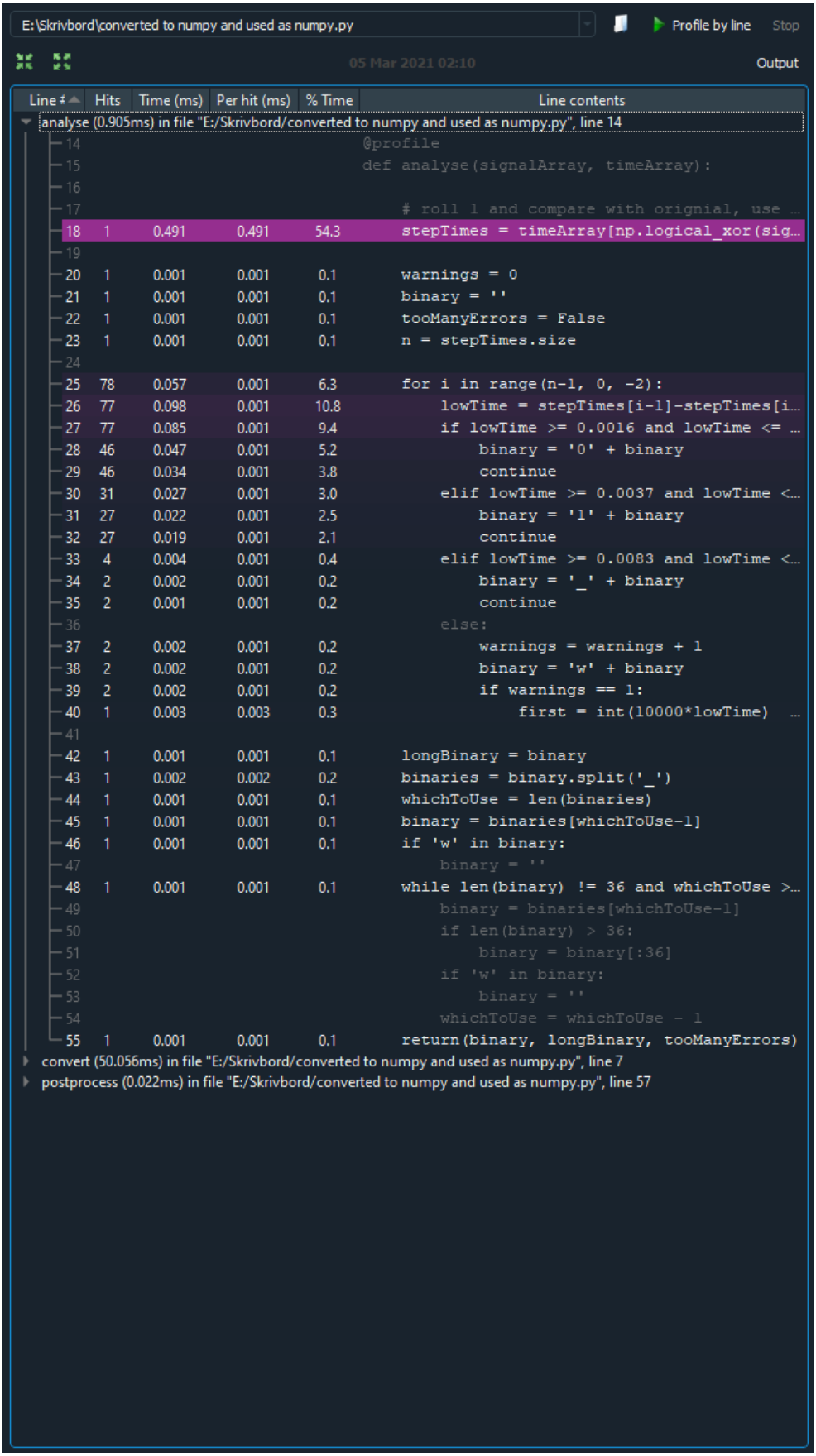
Post process function: 0.02 ms

The time required for the Analyse function was reduced by 99.8 %, some 500 times faster now!

Line profiling of the **Convert function of the code after Strategy D** of course gave the same result as in strategy B and C – it's the same code.



Line profiling of the **Analyse function of the code after Strategy D** showed that most time is taken by the new line including rolling of the Numpy array with the signal. No wonder, but it makes a good job. Almost an equal time is taken by the for loop and stepping through stepTimes.



Line #	Hits	Time (ms)	Per hit (ms)	% Time	Line contents
analyse (0.905ms) in file "E:/Skrivbord/converted to numpy and used as numpy.py", line 14					
14					@profile
15					def analyse(signalArray, timeArray):
16					
17					# roll 1 and compare with orignial, use …
18	1	0.491	0.491	54.3	stepTimes = timeArray[np.logical_xor(sig…
19					
20	1	0.001	0.001	0.1	warnings = 0
21	1	0.001	0.001	0.1	binary = ''
22	1	0.001	0.001	0.1	tooManyErrors = False
23	1	0.001	0.001	0.1	n = stepTimes.size
24					
25	78	0.057	0.001	6.3	for i in range(n-1, 0, -2):
26	77	0.098	0.001	10.8	lowTime = stepTimes[i-1]-stepTimes[i…
27	77	0.085	0.001	9.4	if lowTime >= 0.0016 and lowTime <= …
28	46	0.047	0.001	5.2	binary = '0' + binary
29	46	0.034	0.001	3.8	continue
30	31	0.027	0.001	3.0	elif lowTime >= 0.0037 and lowTime <…
31	27	0.022	0.001	2.5	binary = '1' + binary
32	27	0.019	0.001	2.1	continue
33	4	0.004	0.001	0.4	elif lowTime >= 0.0083 and lowTime <…
34	2	0.002	0.001	0.2	binary = '_' + binary
35	2	0.001	0.001	0.2	continue
36					else:
37	2	0.002	0.001	0.2	warnings = warnings + 1
38	2	0.002	0.001	0.2	binary = 'w' + binary
39	2	0.002	0.001	0.2	if warnings == 1:
40	1	0.003	0.003	0.3	first = int(10000*lowTime) …
41					
42	1	0.001	0.001	0.1	longBinary = binary
43	1	0.002	0.002	0.2	binaries = binary.split('_')
44	1	0.001	0.001	0.1	whichToUse = len(binaries)
45	1	0.001	0.001	0.1	binary = binaries[whichToUse-1]
46	1	0.001	0.001	0.1	if 'w' in binary:
47					binary = ''
48	1	0.001	0.001	0.1	while len(binary) != 36 and whichToUse >…
49					binary = binaries[whichToUse-1]
50					if len(binary) > 36:
51					binary = binary[:36]
52					if 'w' in binary:
53					binary = ''
54					whichToUse = whichToUse - 1
55	1	0.001	0.001	0.1	return(binary, longBinary, tooManyErrors)
convert (50.056ms) in file "E:/Skrivbord/converted to numpy and used as numpy.py", line 7					
postprocess (0.022ms) in file "E:/Skrivbord/converted to numpy and used as numpy.py", line 57					

Code changes after strategy E

I decided to try to replace also the calculation of times

I changed line 19 to calculate that using rolling of stepTimes

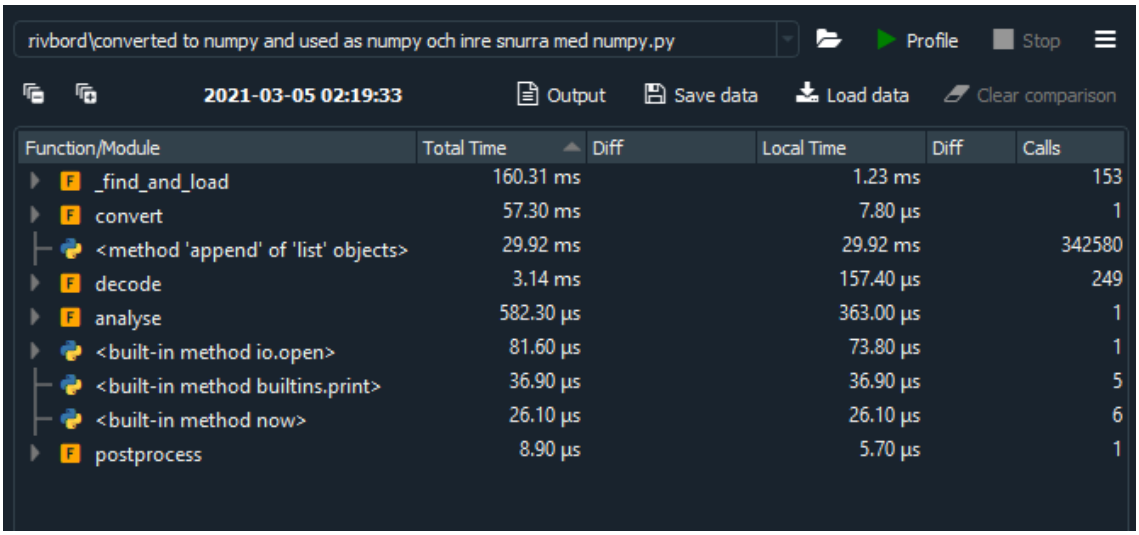
Lines 23 and 24 were changed to loop over the values in plateauTimes instead.

```
15 def analyse(signalArray, timeArray):
16
17     stepTimes = timeArray[np.logical_xor(signalArray, np.roll(signalArray,1))]
18
19     plateaTimes = stepTimes - np.roll(stepTimes,1)
20     warnings = 0
21     binary = ''
22     tooManyErrors = False
23     n = plateauTimes.size-1
24     for i in range(n-1, 0, -2):
25         if plateauTimes[i] >= 0.0016 and plateauTimes[i] <= 0.0025:
26             binary = '0' + binary
27             continue
28         elif plateauTimes[i] >= 0.0037 and plateauTimes[i] < 0.0045:
29             binary = '1' + binary
30             continue
31         elif plateauTimes[i] >= 0.0083 and plateauTimes[i] < 0.0089:
32             binary = '_' + binary
33             continue
34         else:
35             warnings = warnings + 1
```

Comments on the timings after strategy E

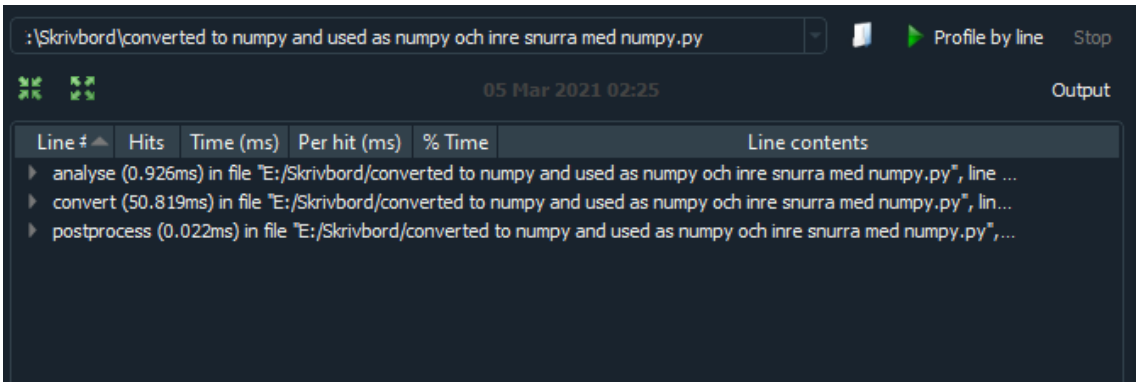
Considering the small number of calculations that was involved I could not hope for much of an improvement, but slightly faster.

Profiling of the code:



Function/Module	Total Time	Diff	Local Time	Diff	Calls
_find_and_load	160.31 ms		1.23 ms		153
convert	57.30 ms		7.80 µs		1
<method 'append' of 'list' objects>	29.92 ms		29.92 ms		342580
decode	3.14 ms		157.40 µs		249
analyse	582.30 µs		363.00 µs		1
<built-in method io.open>	81.60 µs		73.80 µs		1
<built-in method builtins.print>	36.90 µs		36.90 µs		5
<built-in method now>	26.10 µs		26.10 µs		6
postprocess	8.90 µs		5.70 µs		1

Line profiling of the code



Best vales (of 10) when line profiling:

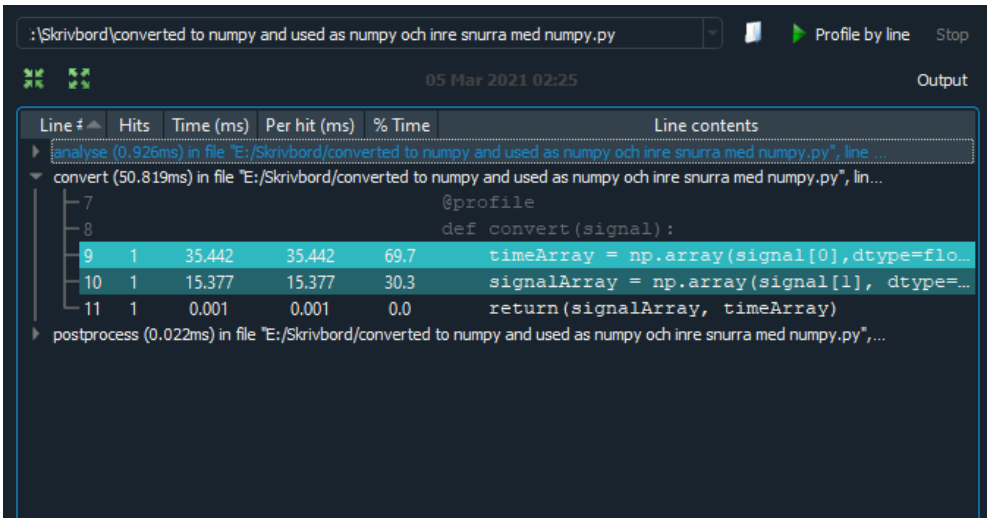
Convert function: 49 ms (same code and same timings as Strategy B, C and D)

Analyse function: 0.78 ms

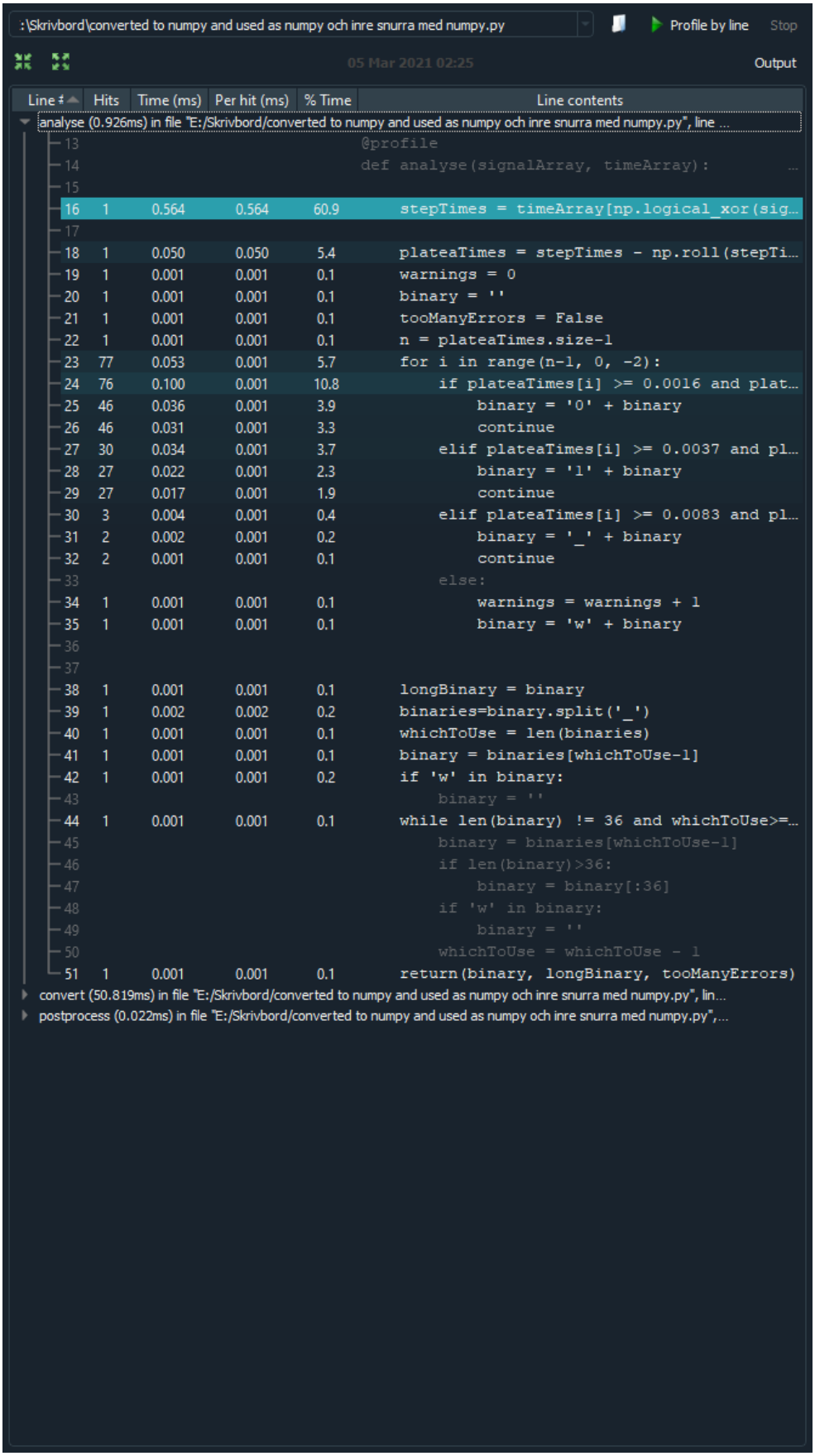
Post process function: 0.02 ms

The time required for the Analyse function was by some 10%. Improvement, but it's only about 0.1 ms in absolute values (at best).

Line profiling of the **Convert function of the code after Strategy** of course gave the same result as in strategy B, C and D– it's the same code.



Line profiling of the **Analyse function of the code after Strategy E** confirmed that the lines calculating and using the plateauTimes were more efficient than using the stepping along stepTimes as in strategy D.



Line #	Hits	Time (ms)	Per hit (ms)	% Time	Line contents
analyse (0.926ms) in file "E:/Skrivbord/converted to numpy and used as numpy och inre snurra med numpy.py", line ...					
13					@profile
14					def analyse(signalArray, timeArray): ...
15					
16	1	0.564	0.564	60.9	stepTimes = timeArray[np.logical_xor(sig...
17					
18	1	0.050	0.050	5.4	plateaTimes = stepTimes - np.roll(stepTi...
19	1	0.001	0.001	0.1	warnings = 0
20	1	0.001	0.001	0.1	binary = ''
21	1	0.001	0.001	0.1	tooManyErrors = False
22	1	0.001	0.001	0.1	n = plateaTimes.size-1
23	77	0.053	0.001	5.7	for i in range(n-1, 0, -2):
24	76	0.100	0.001	10.8	if plateaTimes[i] >= 0.0016 and plat...
25	46	0.036	0.001	3.9	binary = '0' + binary
26	46	0.031	0.001	3.3	continue
27	30	0.034	0.001	3.7	elif plateaTimes[i] >= 0.0037 and pl...
28	27	0.022	0.001	2.3	binary = '1' + binary
29	27	0.017	0.001	1.9	continue
30	3	0.004	0.001	0.4	elif plateaTimes[i] >= 0.0083 and pl...
31	2	0.002	0.001	0.2	binary = '_' + binary
32	2	0.001	0.001	0.1	continue
33					else:
34	1	0.001	0.001	0.1	warnings = warnings + 1
35	1	0.001	0.001	0.1	binary = 'w' + binary
36					
37					
38	1	0.001	0.001	0.1	longBinary = binary
39	1	0.002	0.002	0.2	binaries=binary.split('_')
40	1	0.001	0.001	0.1	whichToUse = len(binaries)
41	1	0.001	0.001	0.1	binary = binaries[whichToUse-1]
42	1	0.001	0.001	0.2	if 'w' in binary:
43					binary = ''
44	1	0.001	0.001	0.1	while len(binary) != 36 and whichToUse>=...
45					binary = binaries[whichToUse-1]
46					if len(binary)>36:
47					binary = binary[:36]
48					if 'w' in binary:
49					binary = ''
50					whichToUse = whichToUse - 1
51	1	0.001	0.001	0.1	return(binary, longBinary, tooManyErrors)
convert (50.819ms) in file "E:/Skrivbord/converted to numpy and used as numpy och inre snurra med numpy.py", lin...					
postprocess (0.022ms) in file "E:/Skrivbord/converted to numpy and used as numpy och inre snurra med numpy.py",...					

Comments on timings when comparing Strategy E and Cython versions of Strategy E

%timeit on the functions after Strategy E

%timeit signalArray, timeArray = radiop.convert(signal)
51.5 ms ± 432 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
51.6 ms ± 400 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
53.3 ms ± 1.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit binary, longBinary, tooManyErrors = radiop.analyse(signalArray, timeArray)
316 µs ± 2.89 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
328 µs ± 3.73 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
317 µs ± 2.55 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

%timeit radiop.postprocess(binary, longBinary, tooManyErrors)
748 µs ± 15.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
749 µs ± 14.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
726 µs ± 28.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

%timeit on the Cython versions of the fuctions after Strategy E

%timeit radio.convert(signal)
51.3 ms ± 144 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
51.7 ms ± 396 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
51.4 ms ± 189 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

%timeit binary, longBinary, tooManyErrors = radio.analyse(signalArray, timeArray)
305 µs ± 2.68 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
305 µs ± 1.04 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
307 µs ± 3.23 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

%timeit radio.postprocess(binary, longBinary, tooManyErrors)
180 µs ± 1.79 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
176 µs ± 3.32 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
178 µs ± 2.57 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Best vales (of 10) when calling the function and using %timeit In Spyder:

Convert function:	49 ms using strategy E became 51,5 ms In Cython	+4% or +2,5 ms
Analyse function:	0.32 ms using strategy E became 0.30 ms In Cython	-6% or -0.02 ms
Post process function:	0.073 ms using strategy E became 0,18 ms In Cython	-75% or -0.055 ms

Finally some improvement in the Post process function!
But the fact I don't get much improvement in the other two functions is an indication that Numpy is very efficient, on par with Cython. The critical parts of the code I had apparently already made fast using Numpy in Python, and the rest of the code in the Convert and Analyse functions could not be improved. But the Post process function could!