

# **Time Series Analysis**

Yair Mau

# Table of contents

<b>about</b>	<b>13</b>
disclaimer . . . . .	13
what, who, when and where? . . . . .	13
syllabus . . . . .	14
course description . . . . .	14
course aims . . . . .	14
learning outcomes . . . . .	14
books and other sources . . . . .	14
grading . . . . .	14
<b>2024/2025 Schedule</b>	<b>15</b>
Week 1, 29 Oct 2024 . . . . .	15
Week 2, 5 Nov 2024 . . . . .	15
Week 3, 12 Nov 2024 . . . . .	15
Week 4, 19 Nov 2024 . . . . .	15
Week 5, 26 Nov 2024 . . . . .	16
Week 6, 3 Dec 2024 . . . . .	16
Week 7, 10 Dec 2024 . . . . .	16
Week 8, 17 Dec 2024 . . . . .	16
24 Dec 2024 . . . . .	16
Week 9, 31 Dec 2024 . . . . .	16
Week 10, 7 Jan 2025 . . . . .	16
Week 11, 14 Jan 2025 . . . . .	16
Week 12, 21 Jan 2025 . . . . .	17
Week 13, 28 Jan 2025 . . . . .	17
<b>who cares?</b>	<b>18</b>
why “Time Series Analysis?” . . . . .	18
why “Environmental Sciences” . . . . .	18
what is it good for? . . . . .	18
do I need it? . . . . .	19
what will I <b>actually</b> gain from it? . . . . .	19

<b>I</b>	<b>start here</b>	<b>20</b>
<b>1</b>	<b>the boring stuff you absolutely need to do</b>	<b>21</b>
1.1	Anaconda . . . . .	21
1.2	VSCode . . . . .	21
1.3	jupyter notebooks . . . . .	21
1.4	folder structure . . . . .	22
<b>2</b>	<b>numpy, pandas, matplotlib</b>	<b>23</b>
2.1	pandas . . . . .	23
2.2	pyplot . . . . .	24
<b>3</b>	<b>learn by example</b>	<b>28</b>
3.1	open a new Jupyter Notebook . . . . .	28
3.2	import packages . . . . .	29
3.3	load data . . . . .	29
3.4	dealing with dates . . . . .	30
3.5	first plot . . . . .	31
3.6	first plot, v2.0 . . . . .	32
3.7	modifications . . . . .	34
3.8	playing with the code . . . . .	36
<b>4</b>	<b>AI policy</b>	<b>37</b>
<b>II</b>	<b>resampling</b>	<b>39</b>
<b>5</b>	<b>motivation</b>	<b>40</b>
5.1	Jerusalem, 2019 . . . . .	40
<b>6</b>	<b>resampling</b>	<b>42</b>
<b>7</b>	<b>upsampling</b>	<b>51</b>
7.1	Potential Evapotranspiration using Penman's equation . . . . .	51
7.2	Forward/Backward fill . . . . .	54
7.3	Interpolation . . . . .	56
7.4	Dealing with missing rows . . . . .	57
<b>8</b>	<b>interpolation</b>	<b>61</b>
<b>9</b>	<b>practice</b>	<b>62</b>

<b>10 FAQ</b>	<b>66</b>
10.1 How to resample by year, but have it end in September? . . . . .	66
10.2 When upsampling, how to fill missing values with zero? . . . . .	70
<b>III smoothing</b>	<b>71</b>
<b>11 motivation</b>	<b>72</b>
11.1 Tumbling vs Sliding . . . . .	76
<b>12 sliding window</b>	<b>77</b>
12.1 convolution . . . . .	79
12.2 kernels . . . . .	79
12.3 math . . . . .	80
12.4 numerics . . . . .	81
12.4.1 7-day average of COVID-19 infections . . . . .	82
12.4.2 gaussian . . . . .	85
12.4.3 triangular . . . . .	86
12.5 which window shape and width to choose? . . . . .	87
<b>13 not only averages</b>	<b>89</b>
13.1 Confidence Interval . . . . .	92
<b>14 fit</b>	<b>97</b>
14.1 linear fit . . . . .	101
14.2 polynomial fit . . . . .	102
14.3 any function you want . . . . .	103
<b>15 Savitzky–Golay</b>	<b>109</b>
<b>IV outliers and gaps</b>	<b>114</b>
<b>16 motivation</b>	<b>115</b>
<b>17 outlier identification</b>	<b>117</b>
17.1 visual inspection . . . . .	117
17.2 Z-score . . . . .	118
17.2.1 ATTENTION! . . . . .	119
17.3 IQR . . . . .	120

17.4 non-stationary time series . . . . .	122
17.5 Sources . . . . .	123
<b>18 robust analysis</b>	<b>124</b>
18.1 MAD . . . . .	124
<b>19 sliding algorithms</b>	<b>126</b>
19.1 Sliding Z-score . . . . .	126
19.2 Sliding IQR . . . . .	126
19.3 Sliding MAD . . . . .	127
19.4 Challenges . . . . .	128
<b>20 substituting outliers</b>	<b>129</b>
20.1 Do nothing . . . . .	129
20.2 NaN . . . . .	129
20.3 impute values . . . . .	131
<b>21 challenge</b>	<b>133</b>
21.1 importing bad .csv files . . . . .	133
21.1.1 import . . . . .	133
21.2 dataset 1 . . . . .	134
21.3 dataset 2 . . . . .	137
21.4 dataset 3 . . . . .	143
<b>22 challenge part 2</b>	<b>147</b>
22.1 outliers and missing values . . . . .	147
22.2 cleaning df1 from outliers . . . . .	149
22.2.1 method 1: rolling standard deviation envelope . . . . .	149
22.2.2 plotting the results: . . . . .	151
22.3 plateaus . . . . .	154
22.3.1 TO DO: . . . . .	156
<b>V stationarity</b>	<b>157</b>
<b>23 motivation</b>	<b>158</b>
23.1 questions . . . . .	158
23.2 goals . . . . .	159
<b>24 random variables</b>	<b>160</b>
discrete random variable . . . . .	160

continuous random variable . . . . .	160
24.1 white noise . . . . .	161
24.2 random walk . . . . .	163
<b>25 autoregressive processes</b>	<b>166</b>
25.1 AR(1) . . . . .	167
25.2 AR(2) . . . . .	168
25.3 AR(p) . . . . .	169
<b>26 autocorrelation</b>	<b>170</b>
26.1 mean and standard deviation . . . . .	170
26.2 expected value . . . . .	171
26.3 covariance . . . . .	172
26.4 correlation . . . . .	172
26.5 autocorrelation . . . . .	173
<b>27 stationarity</b>	<b>174</b>
27.1 weak stationarity . . . . .	174
27.2 strict/strong stationarity . . . . .	174
27.3 stationarity of AR(p) . . . . .	174
27.4 stationarity of AR(1) . . . . .	176
27.5 stationarity of AR(2) . . . . .	177
27.6 stationarity of AR(4) . . . . .	179
<b>28 ACF and PACF graphs</b>	<b>180</b>
28.1 ACF . . . . .	181
28.1.1 problem? . . . . .	184
28.2 PACF . . . . .	185
28.3 discussion . . . . .	186
28.3.1 sine wave . . . . .	186
28.3.2 white noise . . . . .	188
28.3.3 random walk . . . . .	189
<b>29 from AR to ARIMA</b>	<b>191</b>
29.1 AR(p) . . . . .	191
29.2 MA(q) . . . . .	191
29.3 ARMA(p,q) . . . . .	192
29.4 ACF and PACF . . . . .	192
29.5 Non-stationary data and ADF test . . . . .	197
29.6 ARIMA(p,d,q) . . . . .	201
<b>30 practice</b>	<b>207</b>

<b>31 White noise</b>	<b>208</b>
<b>32 Random walk</b>	<b>209</b>
32.1 Differencing . . . . .	209
<b>33 AR(1)</b>	<b>211</b>
33.1 AR(p) . . . . .	212
33.1.1 using specific $\phi$ values . . . . .	213
33.1.2 Weak stationarity . . . . .	214
<b>34 ACF</b>	<b>216</b>
34.1 Now let's work with actual data . . . . .	219
<b>35 filling missing values</b>	<b>223</b>
35.1 Forward fill . . . . .	226
35.2 Backwrds fill . . . . .	226
35.3 Interpolation . . . . .	227
35.3.1 linear . . . . .	227
35.3.2 cubic splines . . . . .	228
35.4 random forest . . . . .	232
<b>36 SARIMAX</b>	<b>235</b>
<b>37 SARIMAX cross fade</b>	<b>237</b>
<b>VI seasonality</b>	<b>241</b>
<b>38 motivation</b>	<b>242</b>
38.1 questions . . . . .	243
<b>39 seasonal decomposition from scratch</b>	<b>244</b>
39.1 sea surface temperature . . . . .	244
39.2 trend . . . . .	245
39.3 detrend . . . . .	246
39.4 seasonal component . . . . .	247
39.4.1 group by . . . . .	248
39.5 residual . . . . .	256
39.6 seasonal decomposition . . . . .	257
<b>40 theory</b>	<b>261</b>
40.1 additive model . . . . .	264
40.2 multiplicative model . . . . .	264

40.3 STL . . . . .	267
<b>41 widgets</b>	<b>269</b>
41.1 range slider . . . . .	271
41.2 temperature vs DOY, with colorbar . . . . .	273
41.3 seasonal decomposition . . . . .	277
<b>VII time lags</b>	<b>279</b>
<b>42 motivation</b>	<b>280</b>
<b>43 cross-correlation</b>	<b>281</b>
<b>44 dynamic time warping</b>	<b>287</b>
44.1 causality . . . . .	297
<b>45 time lags practice</b>	<b>302</b>
<b>46 Dynamic Time Warping (DTW)</b>	<b>314</b>
46.1 Sound and DTW . . . . .	315
46.1.1 “I like to eat Hummus” . . . . .	316
<b>VIII frequency</b>	<b>323</b>
<b>47 motivation</b>	<b>324</b>
<b>48 Fourier transform</b>	<b>325</b>
48.1 basic wave concepts . . . . .	325
48.2 Fourier’s theorem . . . . .	330
48.3 Fourier series . . . . .	330
48.4 Fourier transform . . . . .	332
48.5 fun calculation of a Fourier series . . . . .	332
<b>49 DFT and FFT</b>	<b>337</b>
49.1 DFT . . . . .	337
49.2 FFT . . . . .	341
<b>50 sound &amp; music</b>	<b>344</b>
50.1 power spectrum . . . . .	348
50.2 harmonics . . . . .	349
50.3 timbre . . . . .	350

50.4 linear vs. logarithmic scale . . . . .	353
50.5 chords . . . . .	355
<b>51 frequencies</b>	<b>357</b>
51.1 spectrum . . . . .	357
51.2 resolution . . . . .	358
51.3 Nyquist–Shannon sampling theorem . . . . .	367
<b>52 filtering</b>	<b>369</b>
52.1 decibels . . . . .	375
<b>53 convolution theorem</b>	<b>379</b>
53.1 theorem . . . . .	379
53.2 kernels and their spectral signatures . . . . .	380
53.3 theorem in action . . . . .	384
53.4 now let's play some more . . . . .	388
<b>54 practice 1</b>	<b>392</b>
54.0.1 Apply FFT . . . . .	393
<b>55 practice 2</b>	<b>398</b>
<b>56 practice 3</b>	<b>402</b>
56.1 apply FFT . . . . .	403
56.2 Apply FFT on resampled timeseries . . . . .	406
<b>57 filtering 1</b>	<b>409</b>
57.0.1 Apply FFT . . . . .	410
57.1 filtering . . . . .	412
57.2 low-pass filter . . . . .	413
57.3 high-pass filter . . . . .	415
57.4 band-pass filter . . . . .	417
57.5 band-stop filter . . . . .	418
<b>58 filtering 2</b>	<b>421</b>
58.1 lowpass . . . . .	424
58.2 highpass . . . . .	425
58.3 bandpass . . . . .	426
58.4 bandstop . . . . .	427
<b>59 filtering 3</b>	<b>429</b>
59.1 apply FFT . . . . .	430

59.2 apply bandpass filter . . . . .	431
<b>IX rates of change</b>	<b>434</b>
<b>60 motivation</b>	<b>435</b>
<b>61 finite differences</b>	<b>437</b>
61.1 dead sea level . . . . .	443
<b>62 Fourier-based derivatives</b>	<b>448</b>
62.1 dead sea level . . . . .	449
<b>63 LOESS-based derivatives</b>	<b>456</b>
<b>64 does order matter?</b>	<b>459</b>
<b>X forecasting</b>	<b>461</b>
<b>65 motivation</b>	<b>462</b>
<b>66 ARIMA</b>	<b>463</b>
<b>XI assignments</b>	<b>464</b>
<b>67 Evaluation</b>	<b>465</b>
67.1 Penalty for Late Submission . . . . .	465
67.2 Getting Help from Other Sources . . . . .	465
<b>68 assignment 1</b>	<b>467</b>
68.1 task . . . . .	467
68.2 guidelines . . . . .	468
68.3 evaluation . . . . .	468
<b>69 assignment 2</b>	<b>469</b>
69.1 Smoothing . . . . .	469
69.1.1 1. Comparative Smoothing Methods Analysis . . . . .	469
69.1.2 2. Rolling Average Window Size Impact .	469
69.1.3 3. Savitzky-Golay Polynomial Order Variation . . . . .	470

69.1.4 4. Kernel Shape Influence in Rolling Mean	470
69.1.5 5. Moving Average with Confidence In- terval . . . . .	470
<b>70 assignment 3</b>	<b>471</b>
70.1 background . . . . .	471
70.2 analysis . . . . .	473
<b>71 final project</b>	<b>477</b>
71.1 dataset selection . . . . .	477
71.1.1 data approval . . . . .	477
71.2 Technical Requirements . . . . .	478
71.2.1 Pool 1: implement at least 3 . . . . .	478
71.2.2 Pool 2: implement at least 3 . . . . .	478
71.2.3 Pool 3: implement at least 5 . . . . .	479
71.2.4 Pool 4: implement all . . . . .	479
71.3 Project Objectives . . . . .	480
71.4 Report Structure . . . . .	480
71.4.1 What to submit . . . . .	480
71.4.2 When to submit . . . . .	481
71.4.3 How to submit . . . . .	481
<b>XII technical stuff</b>	<b>482</b>
<b>72 technical stuff</b>	<b>483</b>
72.1 operating systems . . . . .	483
72.2 software . . . . .	483
72.3 python packages . . . . .	483
<b>73 datasets</b>	<b>484</b>
73.1 Sunspots . . . . .	484
73.2 Covid-19 Open Data . . . . .	484
<b>74 date formatting</b>	<b>486</b>
<b>75 sources</b>	<b>494</b>
75.1 books . . . . .	494
75.2 videos . . . . .	494
75.3 references . . . . .	495

<b>XIII</b>	<b>behind-the-scenes</b>	<b>496</b>
<b>sliding window video</b> <b>497</b>		
75.4	Rectangular kernel . . . . .	501
75.5	Triangular kernel . . . . .	507
75.6	Gaussian kernel . . . . .	511
75.7	Comparison . . . . .	515
<b>savgol video</b> <b>518</b>		
75.8	Savgol filter . . . . .	522
<b>API to download data from IMS</b> <b>530</b>		
<b>remove consecutive values</b> <b>532</b>		
<b>outliers graphs</b> <b>536</b>		
75.9	define functions . . . . .	536
75.10	load and process data . . . . .	538
75.11	stationary signal . . . . .	538
75.12	visual inspection . . . . .	539
75.13	mean +- 3 std . . . . .	540
75.14	IQR . . . . .	542
75.15	non stationary signal . . . . .	548
75.16	running +- 3 std . . . . .	549
75.17	running: Q +- IQR . . . . .	553
75.18	Hampel, running MAD . . . . .	557
75.19	stationary MAD . . . . .	559
75.20	save data as csv for later usage . . . . .	561
75.21	generate datasets . . . . .	564
<b>make your own website</b> <b>581</b>		
random tips . . . . .	581	
extensions . . . . .	581	
quarto.yml . . . . .	581	
configure your notebook with a suitable header .	582	
obvious statement . . . . .	583	

# **about**

Welcome to **Time Series Analysis for Environmental Sciences** (71106) at the Hebrew University of Jerusalem. This is Yair Mau, your host for today. I am a senior lecturer at the Institute of Environmental Sciences, at the Faculty of Agriculture, Food and Environment, in Rehovot, Israel.

This website contains (almost) all the material you'll need for the course. If you find any mistakes, or have any comments, please email me.

## **disclaimer**

The material here is not comprehensive and **does not** constitute a stand alone course in Time Series Analysis. This is only the support material for the actual presential course I give.

## **what, who, when and where?**

Course number 71106, 3 academic points

Yair Mau (lecturer), Erez Feuer (TA)

Tuesdays, from 11:15 to 14:00

Computer [classroom #18](#)

Office hours: Tuesdays, from 09:45 to 10:45 (you should send an email to let me know you are coming)

## **syllabus**

### **course description**

Data analysis of time series, with practical examples from environmental sciences.

### **course aims**

This course aims at giving the students a broad overview of the main steps involved in the analysis of time series: data management, data wrangling, visualization, analysis, and forecast. The course will provide a hands-on approach, where students will actively engage with real-life datasets from the field of environmental science.

### **learning outcomes**

On successful completion of this module, students should be able to:

- Explore a time-series dataset, while formulating interesting questions.
- Choose the appropriate tools to attack the problem and answer the questions.
- Communicate their findings and the methods they used to achieve them, using graphs, statistics, text, and a well-documented code.

### **books and other sources**

[Click here.](#)

### **grading**

There will be assignments during the semester (totaling 50% of the final grade), and one final project (50%).

# 2024/2025 Schedule

## Week 1, 29 Oct 2024

### Introduction

Course overview, setting of expectations, introduction to Jupyter Notebooks, loading data and plotting it. We will review the [learn by example](#) page you were asked to run before the semester started.

Resampling. Downsampling and upsampling. Date formats.

[Assignment 1](#), due 12 Nov 2024

## Week 2, 5 Nov 2024

Smoothing. Sliding windows to compute averages and other operations, fitting a model to data, the Savitsky-Golay filter.

## Week 3, 12 Nov 2024

Outliers. Identification using non robust and robust methods, leveraging sliding windows to identify outliers. Should we replace outliers? For what?

[Assignment 2](#), due 26 Nov 2024

## Week 4, 19 Nov 2024

Stationarity: random processes, statistics refresher, AR processes

## **Week 5, 26 Nov 2024**

Stationarity: ACF and PACF graphs

[Assignment 3](#), due 10 Dec 2024

## **Week 6, 3 Dec 2024**

Stationarity: Forecasting, ARIMA, SARIMA, SARIMAX

## **Week 7, 10 Dec 2024**

[Assignment 4](#), due 31 Dec 2024

## **Week 8, 17 Dec 2024**

Seasonality

## **24 Dec 2024**

Time lags No classes.

## **Week 9, 31 Dec 2024**

Frequency

[Assignment 5](#), due 14 Jan 2025

## **Week 10, 7 Jan 2025**

Frequency

## **Week 11, 14 Jan 2025**

Frequency

[Assignment 6](#), due 28 Jan 2025

## **Week 12, 21 Jan 2025**

Rate of change

## **Week 13, 28 Jan 2025**

[Final project](#), due 4 Mar 2025

## **who cares?**

### **why “Time Series Analysis?”**

Time has two aspects. There is the arrow, the running river, without which there is no change, no progress, or direction, or creation. And there is the circle or the cycle, without which there is chaos, meaningless succession of instants, a world without clocks or seasons or promises.

URSULA K. LE GUIN

You are here because you are interested in how things change, evolve. In this course I want to discuss with you how to make sense of data whose temporal nature is in its very essence. We will talk about randomness, cycles, frequencies, correlations, and more.

### **why “Environmental Sciences”**

This same time series analysis (TSA) course could be called instead “TSA for finance”, “TSA for Biology”, or any other application. The emphasis in this course is **not** Environmental Sciences, but the concepts and tools of TSA. Because my research is in Environmental Science, and many of the graduate students at HUJI-Rehovot research this, I chose to use examples “close to home”. The same toolset should be useful for students of other disciplines.

## **what is it good for?**

In many fields of science we are flooded by data, and it’s hard to see the forest for the trees. I hope that the topics we’ll

discuss in this course can help you find meaningful patterns in your data, formulate interesting hypotheses, and design better experiments.

## **do I need it?**

Maybe. If you are a grad student and you have temporal data to analyze, then probably yes. However, I have very fond memories of courses that I took as a grad student that were completely unrelated to my research. Sometimes “because it’s fun” is a perfectly good answer.

## **what will I actually gain from it?**

By the end of this course you will have gained:

- a **hands-on** experience of fundamental time-series analysis tools
- an **intuition** regarding the basic concepts
- **technical** abilities
- a **springboard** for learning more about the subject by yourself

## **Part I**

**start here**

# **1 the boring stuff you absolutely need to do**

I assume everyone registered has taken a basic Python course.  
On your computer, do the following:

## **1.1 Anaconda**

Install [Anaconda's Python distribution](#). The Anaconda installation brings with it all the main python packages we will need to use. In order to install extra packages, refer to these two tutorials: [tutorial 1](#), [tutorial 2](#).

## **1.2 VSCode**

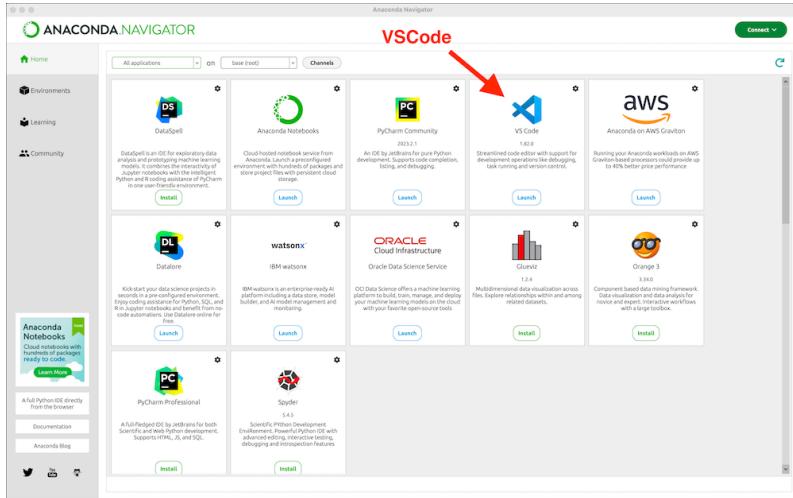
Install [VSCode](#). Visual Studio Code is a very nice IDE (Integrated Development Environment) made by Microsoft, available to all operating systems. Contrary to the title of this page, it is not absolutely necessary to use it, but I like VSCode, and as my student, so do you .

## **1.3 jupyter notebooks**

We will code exclusively in Jupyter Notebooks. [Get acquainted with them](#). Make sure you can [point VSCode](#) to the Anaconda environment of your choice (“base” by default). Don’t worry, this is easier than it sounds.

One failproof way of making sure VSCode uses the Anaconda installation is the following:

- Open Anaconda Navigator
- If you are using HUJI's computers, in “Environments”, choose “asgard”. If you are using your own computer, ignore this step.
- open VSCode from inside Anaconda Navigator (see image below).



Sometimes you will need to manually install the Jupyter extension on VSCode. In this case follow [this tutorial](#).

## 1.4 folder structure

You **NEED** to be comfortable with your computer's folder (or directory) structure. Where are files located? How to navigate through different folders? How is my stuff organized? If you don't feel **absolutely** comfortable with this, then read this, [Windows](#), [MacOS](#). If you use Linux then you surely know this stuff. **Make yourself a “time-series” folder** wherever you want, and have it backed up regularly (use Google Drive, Dropbox, do it manually, etc). “My dog deleted my files” is not an excuse.

## 2 numpy, pandas, matplotlib

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

The three lines above are the most common way you will start every project in this course.

- **numpy** = numerical python. This library has a ton useful mathematical functions, and most importantly, it has an object called **numpy array**, which is one of the most useful data structures we have for time series analysis.
- **pandas** is built upon numpy, and allows us to easily manipulate data stored in **dataframes**, a fancy name for a table.
- **pyplot** is a submodule of **matplotlib**, and allows us to beautifully plot data.

The best resource I know to get acquainted with all three packages is [Python Data Science Handbook, by Jake VanderPlas](#). This is a free online book, with excellent step by step examples.

### 2.1 pandas

We will primarily use the Pandas package to deal with data. Pandas has become the standard Python tool to manipulate time series, and you should get acquainted with its basic usage. This course will provide you the opportunity to learn by example, but I'm sure we will only scratch the surface, and you'll be left with lots of questions.

I provide below a (non-comprehensive) list of useful tutorials, they are a good reference for the beginner and for the experienced user.

- [Python Data Science Handbook](#), by Jake VanderPlas
- [Data Wrangling with pandas Cheat Sheet](#)
- [Working with Dates and Times in Python](#)
- [Cheat Sheet: The pandas DataFrame Object](#)
- [YouTube tutorials](#) by Corey Schafer

## 2.2 pyplot

Matplotlib, and its submodule pyplot, are probably the most common Python plotting tool. Pyplot is both great and horrible:

- Great: you'll have absolutely full control of everything you want to plot. The sky is the limit.
- Horrible: you'll cry as you do it, because there is so much to know, and it is not the most friendly plotting package.

Pyplot is *object oriented*, so you will usually manipulate the **axes** object like this.

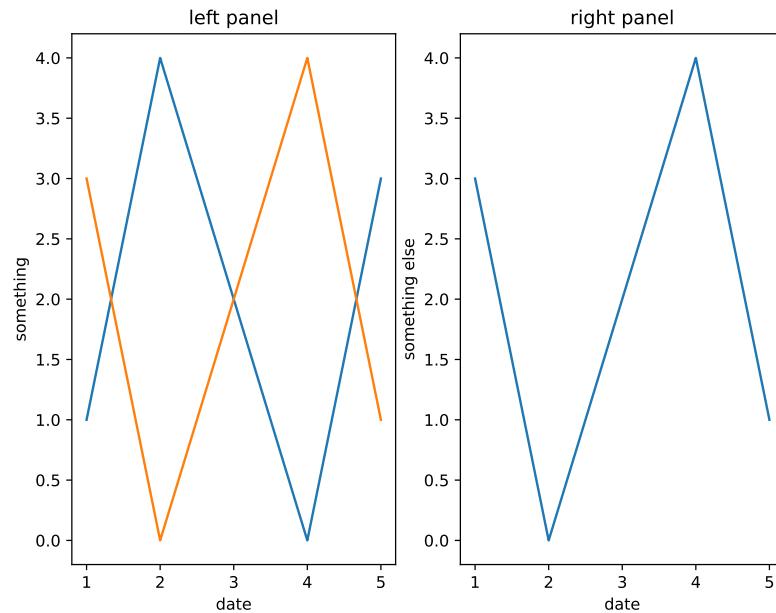
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 2, 0, 3]

# Figure with two plots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (8, 6))
# plot on the left
ax1.plot(x, y, color="tab:blue")
ax1.plot(x, y[::-1], color="tab:orange")
ax1.set(xlabel="date",
        ylabel="something",
        title="left panel")
# plot on the right
ax2.plot(x, y[::-1])
ax2.set(xlabel="date",
```

```
        ylabel="something else",
        title="right panel")
```

```
[Text(0.5, 0, 'date'),
Text(0, 0.5, 'something else'),
Text(0.5, 1.0, 'right panel')]
```



For the very beginners, you need to know that `figure` refers to the whole white canvas, and `axes` means the rectangle inside which something will be plotted:

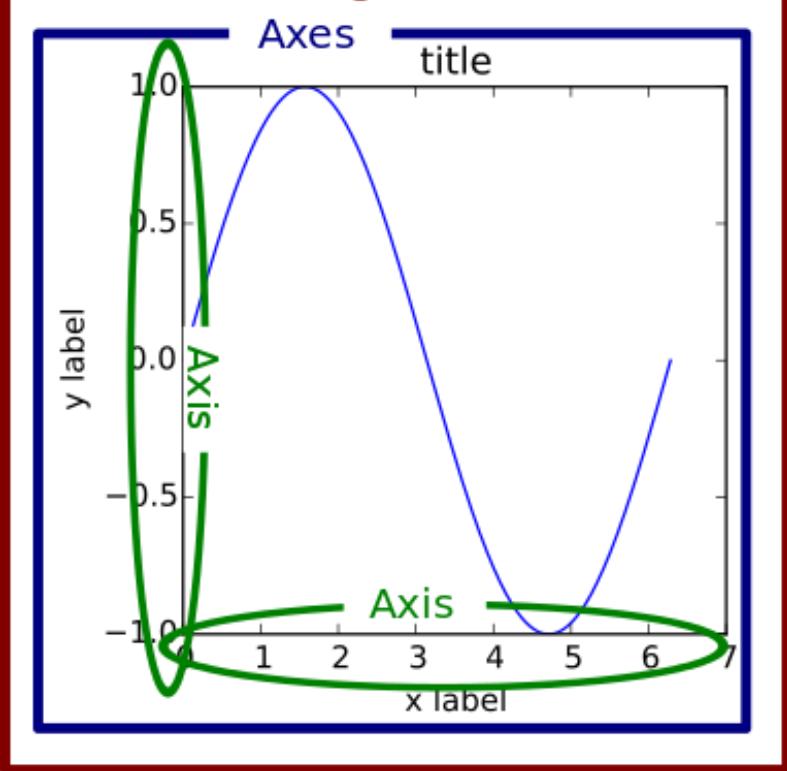


@earthlabcu

The image above is good because it has 2 panels, and it's easy to understand what's going on. Sadly, they mixed the two terms, axis and axes.

- **axes** is where the whole plot will be drawn. In the figure above it is the same as each panel.
- **axis** is each of the vertical and horizontal lines, where you have ticks and numbers.

Figure



If you are new to all this, I recommend that you go to:

- [Earth Lab's Introduction to Plotting in Python Using Matplotlib](#)
- [Jake VanderPlas's Python Data Science Handbook](#)

# 3 learn by example

Now that everything is installed, try to run the code below *before* the first lecture. Don't worry if you don't understand everything.

- If you manage to run everything without errors, this means that your computer is good to go!
- You might encounter a few problems. That's ok. Make a note and we will solve everything in the first lecture.

Let's make a first plot of real data. We will use NOAA's Global Monitoring Laboratory data on [Trends in Atmospheric Carbon Dioxide](#).

## 3.1 open a new Jupyter Notebook

1. On your computer, open the program **Anaconda Navigator** (it may take a while to load).
2. Find the white box called **VS Code** and click **Launch**.
3. Now go to **File > Open Folder**, and open the folder you created for this course. VS Code may ask you if you trust the authors, and the answer is "yes" (it's your computer).
4. **File > New File**, and call it `example.ipynb`
5. You can start copying and pasting code from this website to your Jupyter Notebook. To run a cell, press Shift+Enter.
6. You may be asked to choose to Select Kernel. This is VS Code wanting to know which python installation to use. Click on "Python Environments", and then choose the option with the word `anaconda` in it.
7. That's all! Congratulations!

## 3.2 import packages

First, import packages to be used. They should all be already included in the Anaconda distribution you installed.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
```

## 3.3 load data

Load CO<sub>2</sub> data into a Pandas dataframe. You can load it directly from the URL (option 1), or first download the CSV to your computer and then load it (option 2). The link to download the data directly from NOAA is [this](#). If for some reason this doesn't work, download [here](#).

```
# option 1: load data directly from URL
# url = "https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_weekly_mlo.csv"
# df = pd.read_csv(url,
#                   header=34,
#                   na_values=[-999.99]
#                   )

# option 2: download first (use the URL above and save it to your computer), then load csv
filename = "co2_weekly_mlo.csv"
df = pd.read_csv(filename,
                  comment='#', # will ignore rows starting with #
                  na_values=[-999.99] # substitute -999.99 for NaN (Not a Number), data not available
                  )
# check how the dataframe (table) looks like
df
```

	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 1800
0	1974	5	19	1974.3795	333.37	5	NaN	NaN	50.39
1	1974	5	26	1974.3986	332.95	6	NaN	NaN	50.05

	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 1800
2	1974	6	2	1974.4178	332.35	5	NaN	NaN	49.59
3	1974	6	9	1974.4370	332.20	7	NaN	NaN	49.64
4	1974	6	16	1974.4562	332.37	7	NaN	NaN	50.06
...	...	...	...	...	...	...	...	...	...
2566	2023	7	23	2023.5575	421.28	4	418.03	397.30	141.60
2567	2023	7	30	2023.5767	420.83	6	418.10	396.80	141.69
2568	2023	8	6	2023.5959	420.02	6	417.36	395.65	141.41
2569	2023	8	13	2023.6151	418.98	4	417.25	395.24	140.89
2570	2023	8	20	2023.6342	419.31	2	416.64	395.22	141.71

### 3.4 dealing with dates

Create a new column called `date`, that combines the information from three separate columns: `year`, `month`, `day`.

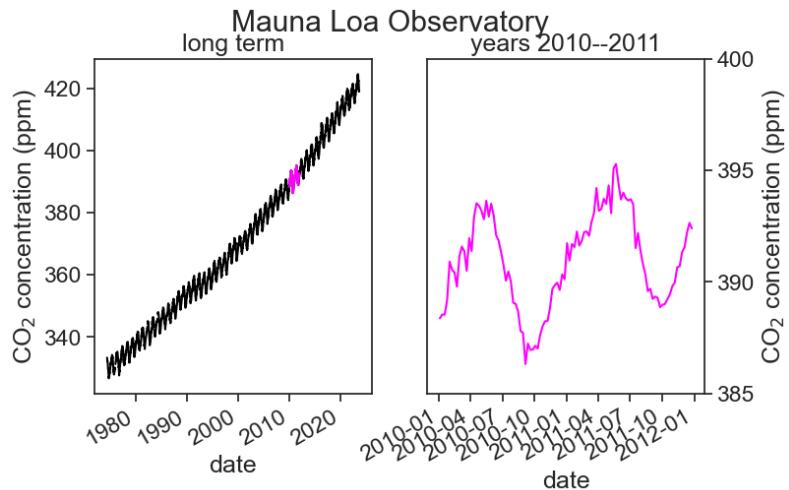
```
# function to_datetime translates the full date into a pandas datetime object,
# that is, pandas knows this is a date, it's not just a string
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
# make 'date' column the dataframe index
df = df.set_index('date')
# now see if everything is ok
df
```

	year	month	day	decimal	average	ndays	1 year ago	10 years ago	increase since 1800
date									
1974-05-19	1974	5	19	1974.3795	333.37	5	NaN	NaN	50.39
1974-05-26	1974	5	26	1974.3986	332.95	6	NaN	NaN	50.05
1974-06-02	1974	6	2	1974.4178	332.35	5	NaN	NaN	49.59
1974-06-09	1974	6	9	1974.4370	332.20	7	NaN	NaN	49.64
1974-06-16	1974	6	16	1974.4562	332.37	7	NaN	NaN	50.06
...	...	...	...	...	...	...	...	...	...
2023-07-23	2023	7	23	2023.5575	421.28	4	418.03	397.30	141.60
2023-07-30	2023	7	30	2023.5767	420.83	6	418.10	396.80	141.69
2023-08-06	2023	8	6	2023.5959	420.02	6	417.36	395.65	141.41
2023-08-13	2023	8	13	2023.6151	418.98	4	417.25	395.24	140.89
2023-08-20	2023	8	20	2023.6342	419.31	2	416.64	395.22	141.71

### 3.5 first plot

We are now ready for our first plot! Let's see the weekly CO<sub>2</sub> average.

```
# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,5) # width, height, in inches
)
# left panel
ax1.plot(df['average'], color="black")
ax1.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        title="long term");
# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax2.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        ylim=[385, 400], # choose y limits
        yticks=np.arange(385, 401, 5), # choose ticks
        title="years 2010--2011");
# put ticks and label on the right for ax2
ax2.yaxis.tick_right()
ax2.yaxis.set_label_position("right")
# title above both panels
fig.suptitle("Mauna Loa Observatory")
# makes slanted dates
plt.gcf().autofmt_xdate()
```



### 3.6 first plot, v2.0

The dates in the x-label are not great. Let's try to make them prettier.

We need to import a few more packages first.

```
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
```

Now let's replot.

```
# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,5) # width, height, in inches
)
# left panel
ax1.plot(df['average'], color="black")
ax1.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
```

```

        title="long term");
# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="magenta")
ax2.set(xlabel="date",
         ylabel=r"CO$_2$ concentration (ppm)",
         ylim=[385, 400], # choose y limits
         yticks=np.arange(385, 401, 5), # choose ticks
         title="years 2010--2011");
# put ticks and label on the right for ax2
ax2.yaxis.tick_right()
ax2.yaxis.set_label_position("right")
# title above both panels
fig.suptitle("Mauna Loa Observatory", y=1.00)

locator = mdates.AutoDateLocator(minticks=3, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax1.xaxis.set_major_locator(locator)
ax1.xaxis.set_major_formatter(formatter)

locator = mdates.AutoDateLocator(minticks=4, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax2.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_formatter(formatter)

ax1.annotate(
    "2010/11",
    xy=('2011-12-25', 389), xycoords='data',
    xytext=(-10, -80), textcoords='offset points',
    arrowprops=dict(arrowstyle="->",
                   color="black",
                   connectionstyle="arc3,rad=0.2"))
fig.savefig("CO2-graph.png", dpi=300)

```

```

/var/folders/hc/jhnmlst937d27zzq9fhfks780000gn/T/ipykernel_10652/850389963.py:42: UserWarning:
  fig.savefig("CO2-graph.png", dpi=300)
/opt/anaconda3/lib/python3.9/site-packages/IPython/core/pylabtools.py:151: UserWarning: AutoData
  fig.canvas.print_figure(bytes_io, **kw)

```



The dates on the horizontal axis are determined thus:

1. `locator = mdates.AutoDateLocator(minticks=3, maxticks=5)`  
This determines the location of the ticks (between 3 and 5 ticks, whatever “works best”)
2. `ax1.xaxis.set_major_locator(locator)`  
This actually puts the ticks in the positions determined above
3. `formatter = mdates.ConciseDateFormatter(locator)`  
This says that the labels will be placed at the locations determined in 1.
4. `ax1.xaxis.set_major_formatter(formatter)`  
Finally, labels are written down

The arrow is placed in the graph using `annotate`. It has a tricky syntax and a million options. Read [Jake VanderPlas’s excellent examples](#) to learn more.

### 3.7 modifications

Let’s change a lot of plotting options to see how things could be different.

```

sns.set(style="darkgrid")
sns.set_context("notebook")

# %matplotlib widget
# uncomment the above line if you want dynamic control of the figure when using VSCode
fig, (ax1, ax2) = plt.subplots(1, 2, # 1 row, 2 columns
                               figsize=(8,4) # width, height, in inches
)
# left panel
ax1.plot(df['average'], color="tab:blue")
ax1.plot(df.loc['2010-01-01':'2011-12-31','average'], color="tab:orange")
ax1.set(xlabel="date",
        ylabel=r"CO$_2$ concentration (ppm)",
        title="long term");
# right panel
ax2.plot(df.loc['2010-01-01':'2011-12-31','average'], color="tab:orange")
ax2.set(xlabel="date",
        ylim=[385, 400], # choose y limits
        yticks=np.arange(385, 401, 5), # choose ticks
        title="years 2010--2011");
# title above both panels
fig.suptitle("Mauna Loa Observatory", y=1.00)

locator = mdates.AutoDateLocator(minticks=3, maxticks=5)
formatter = mdates.ConciseDateFormatter(locator)
ax1.xaxis.set_major_locator(locator)
ax1.xaxis.set_major_formatter(formatter)

locator = mdates.AutoDateLocator(minticks=5, maxticks=8)
formatter = mdates.ConciseDateFormatter(locator)
ax2.xaxis.set_major_locator(locator)
ax2.xaxis.set_major_formatter(formatter)

ax1.annotate(
    "2010/11",
    xy=('2010-12-25', 395), xycoords='data',
    xytext=(-100, 40), textcoords='offset points',
    bbox=dict(boxstyle="round4,pad=.5", fc="white"),
    arrowprops=dict(arrowstyle="->",
                   color="black",

```

```
connectionstyle="angle,angleA=0,angleB=-90,rad=40"))
```

```
Text(-100, 40, '2010/11')
```



The main changes were:

1. Using the Seaborn package, we changed the fontsize and the overall plot style. [Read more](#).  

```
sns.set(style="darkgrid")
sns.set_context("notebook")
```
2. We changed the colors of the lineplots. To know what colors exist, [click here](#).
3. The arrow annotation has a different style. [Read more](#).

## 3.8 playing with the code

I encourage you to play with the code you just ran. An easy way of learning what each line does is to comment something out and see what changes in the output you see. If you feel brave, try to modify the code a little bit.

## 4 AI policy



The guidelines below are an adaptation of [Ethan Mollick's extremely useful ideas on AI](#) as an assistant tool for teaching.

**I EXPECT YOU** to use LLMs (large language models) such as ChatGPT, Bing AI, Google Bard, or whatever else springs up since the time of this writing. You should familiarize yourself with the AI's capabilities and limitations.

**Use LLMs to help you learn**, chat with them about what you want to accomplish and learn from them how to do it. **Ask** your LLM what each part of the code means, copy and pasting blindly is unacceptable. You are here to learn.

Consider the following important points:

- Ultimately, you, the student, are responsible for the assignment.
- Acknowledge the use of AI in your assignment. Be transparent about your use of the tool and the extent of assistance it provided.



A.I  
taking my job



Learning  
to use A.I  
and make  
my job easier

imgflip.com

# **Part II**

## **resampling**

# 5 motivation

## 5.1 Jerusalem, 2019

Data from the [Israel Meteorological Service](#), IMS.

See the temperature at a weather station in Jerusalem, for the whole 2019 year. This is an interactive graph: to zoom in, play with the bottom panel.

```
alt.VConcatChart(...)
```

### 5.1.0.0.1 \* discussion

The temperature fluctuates on various time scales, from daily to yearly. Let's think together a few questions we'd like to ask about the data above.

Now let's see precipitation data:

```
alt.VConcatChart(...)
```

### 5.1.0.0.2 \* discussion

What would be interesting to know about precipitation?

We have not talked about what kind of data we have in our hands here. The csv file provided by the IMS looks like this:

	Station	Date & Time (Winter)	Diffused radiation (W/m <sup>2</sup> )	Global radiation (W/m <sup>2</sup> )
0	Jerusalem Givat Ram	01/01/2019 00:00	0.0	0.0
1	Jerusalem Givat Ram	01/01/2019 00:10	0.0	0.0
2	Jerusalem Givat Ram	01/01/2019 00:20	0.0	0.0
3	Jerusalem Givat Ram	01/01/2019 00:30	0.0	0.0

	Station	Date & Time (Winter)	Diffused radiation (W/m^2)	Global radiation (W/m^2)
4	Jerusalem Givat Ram	01/01/2019 00:40	0.0	0.0
...	...	...	...	...
52549	Jerusalem Givat Ram	31/12/2019 22:20	0.0	0.0
52550	Jerusalem Givat Ram	31/12/2019 22:30	0.0	0.0
52551	Jerusalem Givat Ram	31/12/2019 22:40	0.0	0.0
52552	Jerusalem Givat Ram	31/12/2019 22:50	0.0	0.0
52553	Jerusalem Givat Ram	31/12/2019 23:00	0.0	0.0

We see that we have data points spaced out evenly every 10 minutes.

Useful functions compatible with `pandas.resample()` can be found [here](#). The full list of resampling frequencies can be found [here](#).

# 6 resampling

We can only really understand how to calculate monthly means if we do it ourselves.

First, let's import a bunch of packages we need to use.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
```

Now we load the csv file for Jerusalem (2019), provided by the [IMS](#).

## 6.0.0.1 \* discussion

We will go to the IMS website together and see what are the options available and how to download. If you just need the csv right away, download it [here](#).

- We substitute every occurrence of - for NaN (not a number, that is, the data is missing).
- We call the columns Temperature ( $^{\circ}\text{C}$ ) and Rainfall ( $\text{mm}$ ) with more convenient names, since we will be using them a lot.

- We interpret the column Date & Time (Winter) as a date, saying to python that day comes first.
- We make date the index of the dataframe.

```
filename = "../archive/data/jerusalem2019.csv"
df = pd.read_csv(filename, na_values=['-'])
df.rename(columns={'Temperature (°C)': 'temperature',
                   'Rainfall (mm)': 'rain'}, inplace=True)
df['date'] = pd.to_datetime(df['Date & Time (Winter)'], dayfirst=True)
df = df.set_index('date')
df
```

	Station	Date & Time (Winter)	Diffused radiation (W/m^2)	Global rad
date				
2019-01-01 00:00:00	Jerusalem Givat Ram	01/01/2019 00:00	0.0	0.0
2019-01-01 00:10:00	Jerusalem Givat Ram	01/01/2019 00:10	0.0	0.0
2019-01-01 00:20:00	Jerusalem Givat Ram	01/01/2019 00:20	0.0	0.0
2019-01-01 00:30:00	Jerusalem Givat Ram	01/01/2019 00:30	0.0	0.0
2019-01-01 00:40:00	Jerusalem Givat Ram	01/01/2019 00:40	0.0	0.0
...	...	...	...	...
2019-12-31 22:20:00	Jerusalem Givat Ram	31/12/2019 22:20	0.0	0.0
2019-12-31 22:30:00	Jerusalem Givat Ram	31/12/2019 22:30	0.0	0.0
2019-12-31 22:40:00	Jerusalem Givat Ram	31/12/2019 22:40	0.0	0.0
2019-12-31 22:50:00	Jerusalem Givat Ram	31/12/2019 22:50	0.0	0.0
2019-12-31 23:00:00	Jerusalem Givat Ram	31/12/2019 23:00	0.0	0.0

With `resample` it's easy to compute monthly averages. Resample by itself only divides the data into buckets (in this case monthly buckets), and waits for a further instruction. Here, the next instruction is `mean`.

```
df_month = df['temperature'].resample('M').mean()
df_month
```

date	
2019-01-31	9.119937
2019-02-28	9.629812
2019-03-31	10.731571

```

2019-04-30    14.514329
2019-05-31    22.916894
2019-06-30    23.587361
2019-07-31    24.019403
2019-08-31    24.050822
2019-09-30    22.313287
2019-10-31    20.641868
2019-11-30    17.257153
2019-12-31    11.224131
Freq: ME, Name: temperature, dtype: float64

```

Instead of ME for month, which other options do I have? The full list can be [found here](#), but the most commonly used are:

ME	month end frequency
MS	month start frequency
YE	year end frequency
YS	year start frequency
D	calendar day frequency
h	hourly frequency
min	minutely frequency
s	secondly frequency

The results we got for the monthly means were given as a pandas series, not dataframe. Let's correct this:

```

df_month = (df['temperature'].resample('ME')          # resample by month
            .mean()                                # take the mean
            .to_frame('mean temp')) # make output a dafaframe
)
df_month

```

mean temp	
date	
2019-01-31	9.119937
2019-02-28	9.629812
2019-03-31	10.731571
2019-04-30	14.514329

mean temp	
date	
2019-05-31	22.916894
2019-06-30	23.587361
2019-07-31	24.019403
2019-08-31	24.050822
2019-09-30	22.313287
2019-10-31	20.641868
2019-11-30	17.257153
2019-12-31	11.224131

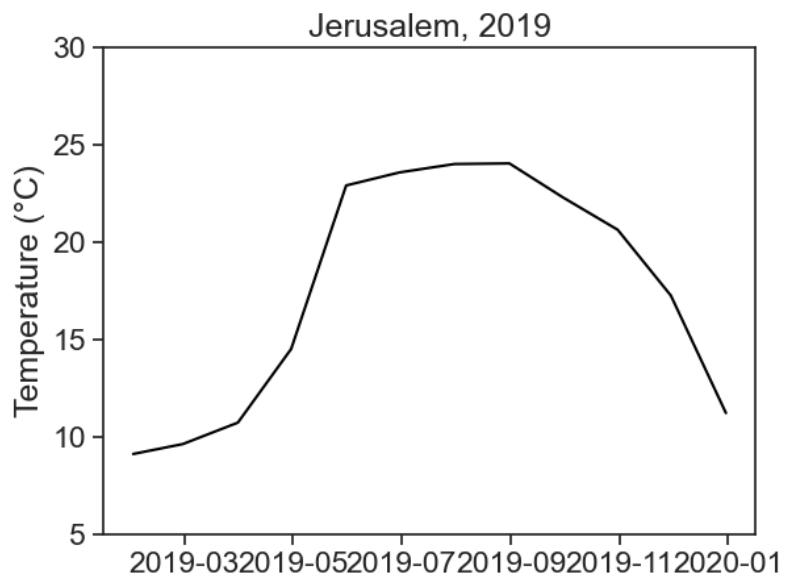
#### 6.0.0.2 \* hot tip

Sometimes, a line of code can get too long and messy. In the code above, we broke line for every step, which makes the process so much cleaner. We **highly** advise you to do the same. **Attention:** This trick works as long as all the elements are inside the same parenthesis.

Now it's time to plot!

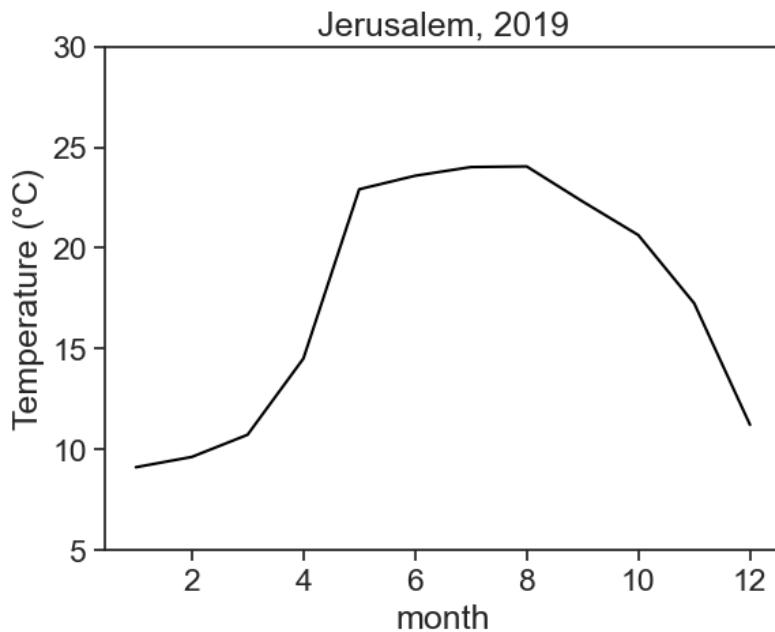
```
fig, ax = plt.subplots()
ax.plot(df_month['mean temp'], color='black')
ax.set(ylabel='Temperature (°C)',
       yticks=np.arange(5,35,5),
       title="Jerusalem, 2019")
```

```
[Text(0, 0.5, 'Temperature (°C)'),
 [matplotlib.axis.YTick at 0x7faf784c6d60>,
 <matplotlib.axis.YTick at 0x7faf7843a220>,
 <matplotlib.axis.YTick at 0x7faf784c62b0>,
 <matplotlib.axis.YTick at 0x7faf784f3400>,
 <matplotlib.axis.YTick at 0x7faf784f3760>,
 <matplotlib.axis.YTick at 0x7faf784fa5b0>],
 Text(0.5, 1.0, 'Jerusalem, 2019')]
```



The dates in the horizontal axis are not great. An easy fix is to use the month numbers instead of dates.

```
fig, ax = plt.subplots()
ax.plot(df_month.index.month, df_month['mean temp'], color='black')
ax.set(xlabel="month",
       ylabel='Temperature (°C)',
       yticks=np.arange(5,35,5),
       title="Jerusalem, 2019",);
```



#### 6.0.0.0.3 \* discussion

When you have datetime as the dataframe index, you don't need to give the function `plot` two arguments, date and values. You can just tell `plot` to use the column you want, the function will take the dates by itself.

What does this line mean?

```
df_month['mean temp'].index.month
```

Print on the screen the following, and see yourself what each thing is:

- `df_month`
- `df_month.index`
- `df_month.index.month`
- `df_month.index.day`

We're done! Congratulations :)

Now we need to calculate the average minimum/maximum daily temperatures. We start by creating an empty dataframe.

```
df_day = pd.DataFrame()
```

Now resample data by day (D), and take the min/max of each day.

```
df_day['min temp'] = df['temperature'].resample('D').min()
df_day['max temp'] = df['temperature'].resample('D').max()
df_day
```

	min temp	max temp
date		
2019-01-01	7.5	14.1
2019-01-02	6.6	11.5
2019-01-03	6.3	10.7
2019-01-04	6.6	14.6
2019-01-05	7.0	11.4
...	...	...
2019-12-27	4.4	7.4
2019-12-28	6.6	10.3
2019-12-29	8.1	12.5
2019-12-30	6.9	13.0
2019-12-31	5.2	13.3

The next step is to calculate the average minimum/maximum for each month. This is similar to what we did above.

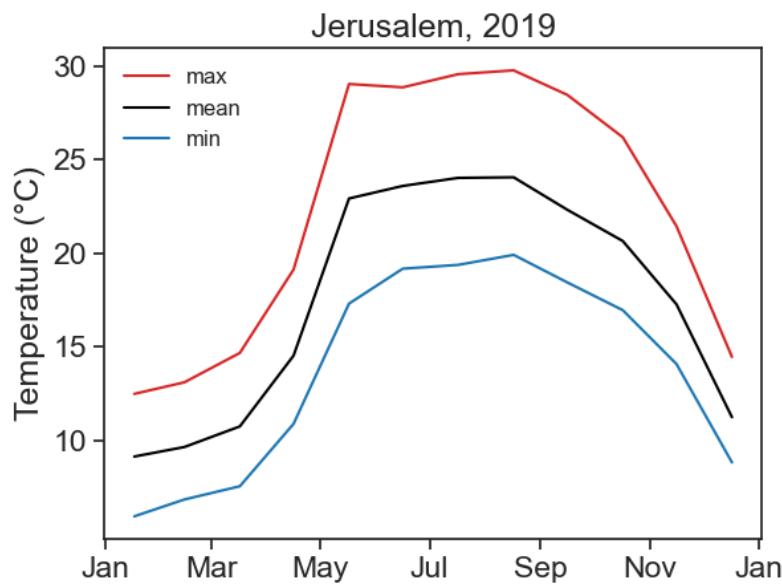
```
df_month['min temp'] = df_day['min temp'].resample('M').mean()
df_month['max temp'] = df_day['max temp'].resample('M').mean()
df_month
```

	mean temp	min temp	max temp
date			
2019-01-31	9.119937	5.922581	12.470968
2019-02-28	9.629812	6.825000	13.089286
2019-03-31	10.731571	7.532258	14.661290
2019-04-30	14.514329	10.866667	19.113333

	mean temp	min temp	max temp
date			
2019-05-31	22.916894	17.296774	29.038710
2019-06-30	23.587361	19.163333	28.860000
2019-07-31	24.019403	19.367742	29.564516
2019-08-31	24.050822	19.903226	29.767742
2019-09-30	22.313287	18.430000	28.456667
2019-10-31	20.641868	16.945161	26.190323
2019-11-30	17.257153	14.066667	21.436667
2019-12-31	11.224131	8.806452	14.448387

Let's plot...

```
fig, ax = plt.subplots()
ax.plot(df_month['max temp'], color='tab:red', label='max')
ax.plot(df_month['mean temp'], color='black', label='mean')
ax.plot(df_month['min temp'], color='tab:blue', label='min')
ax.set(ylabel='Temperature (°C)',
       yticks=np.arange(10,35,5),
       title="Jerusalem, 2019")
ax.xaxis.set_major_locator(mdates.MonthLocator(range(1, 13, 2), bymonthday=15))
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
ax.legend(fontsize=12, frameon=False);
```



Voilà! You made a beautiful graph!

#### 6.0.0.4 \* discussion

This time we did not put month numbers in the horizontal axis, we now have month names. How did we do this black magic, you ask? See lines 8–10 above. Matplotlib gives you absolute power over what to put in the axis, if you can only know how to tell it to... Wanna know more? [Click here.](#)

# 7 upsampling

In the previous chapter, we resampled from fine temporal resolution to a coarser one. This is also called **downsampling**. We will learn the **upsampling** now: how to go from coarse data to a finer scale.

Sadly, there is no free lunch, and we just can't get data that was not measured. What to do then?

It's best to consider a practical example.

## 7.1 Potential Evapotranspiration using Penman's equation

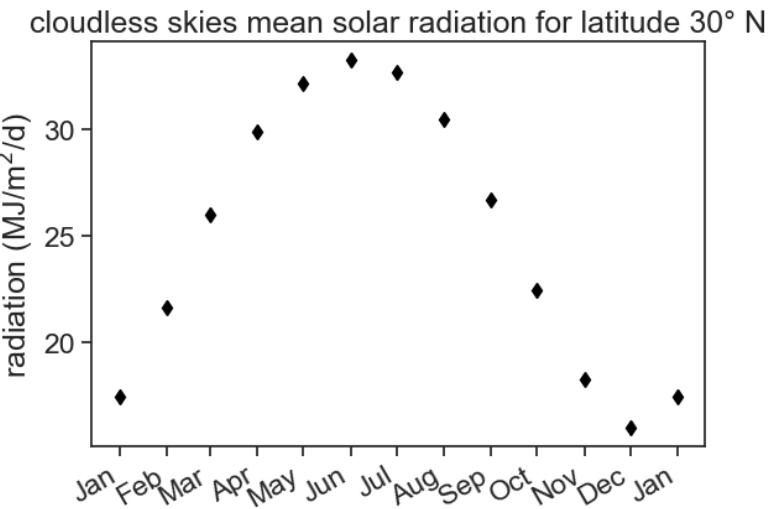
We want to calculate the daily potential evapotranspiration using [Penman's equation](#). Part of the calculation involves characterizing the energy budget on soil surface. When direct solar radiation measurements are not available, we can estimate the energy balance by knowing the “cloudless skies mean solar radiation”,  $R_{so}$ . This is the amount of energy ( $\text{MJ/m}^2/\text{d}$ ) that hits the surface, assuming no clouds. This radiation depends on the season and on the latitude you are. For Israel, located at latitude  $32^\circ \text{ N}$ , we can use the following data for  $30^\circ$ :

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
```

```
dates = pd.date_range(start='2021-01-01', periods=13, freq='MS')
values = [17.46, 21.65, 25.96, 29.85, 32.11, 33.20, 32.66, 30.44, 26.67, 22.48, 18.30, 16.04, 17.46]
df = pd.DataFrame({'date': dates, 'radiation': values})
df = df.set_index('date')
df
```

	radiation
date	
2021-01-01	17.46
2021-02-01	21.65
2021-03-01	25.96
2021-04-01	29.85
2021-05-01	32.11
2021-06-01	33.20
2021-07-01	32.66
2021-08-01	30.44
2021-09-01	26.67
2021-10-01	22.48
2021-11-01	18.30
2021-12-01	16.04
2022-01-01	17.46

```
fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None')
ax.set(ylabel=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")
ax.xaxis.set_major_locator(mdates.MonthLocator())
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates
```



We only have 12 values for the whole year, and we can't use this dataframe to compute daily ET. We need to upsample!

In the example below, we resample the monthly data into daily data, and do nothing else. Pandas doesn't know what to do with the new points, so it fills them with NaN.

```
df_nan = df.asfreq('D')
df_nan.head(33)
```

radiation	
date	
2021-01-01	17.46
2021-01-02	NaN
2021-01-03	NaN
2021-01-04	NaN
2021-01-05	NaN
2021-01-06	NaN
2021-01-07	NaN
2021-01-08	NaN
2021-01-09	NaN
2021-01-10	NaN
2021-01-11	NaN
2021-01-12	NaN
2021-01-13	NaN

	radiation
date	
2021-01-14	NaN
2021-01-15	NaN
2021-01-16	NaN
2021-01-17	NaN
2021-01-18	NaN
2021-01-19	NaN
2021-01-20	NaN
2021-01-21	NaN
2021-01-22	NaN
2021-01-23	NaN
2021-01-24	NaN
2021-01-25	NaN
2021-01-26	NaN
2021-01-27	NaN
2021-01-28	NaN
2021-01-29	NaN
2021-01-30	NaN
2021-01-31	NaN
2021-02-01	21.65
2021-02-02	NaN

## 7.2 Forward/Backward fill

We can forward/backward fill these NaNs:

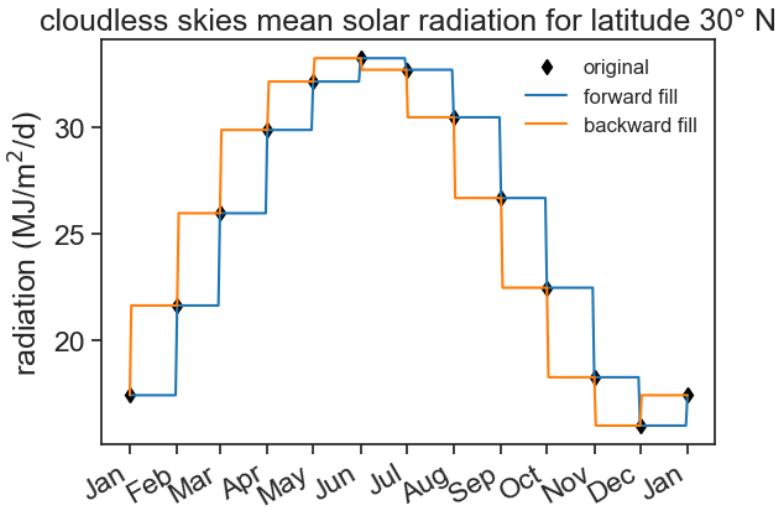
```
df_forw = df_nan.ffill()
df_back = df_nan.bfill()

fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None', label="original")
ax.plot(df_forw['radiation'], color='tab:blue', label="forward fill")
ax.plot(df_back['radiation'], color='tab:orange', label="backward fill")
ax.set(ylab=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")
ax.legend(frameon=False, fontsize=12)
ax.xaxis.set_major_locator(mdates.MonthLocator())
```

```

date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates

```



We first used `asfreq`, and then filled the NaN values with something. Instead of doing this in two steps, we can do the following:

```
df.resample('D').interpolate(method='ffill')
```

```
/var/folders/cn/m5817p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_60150/897677981.py:1: FutureWarning
  df.resample('D').interpolate(method='ffill')
```

radiation	
date	
2021-01-01	17.46
2021-01-02	17.46
2021-01-03	17.46
2021-01-04	17.46
2021-01-05	17.46
...	...
2021-12-28	16.04
2021-12-29	16.04

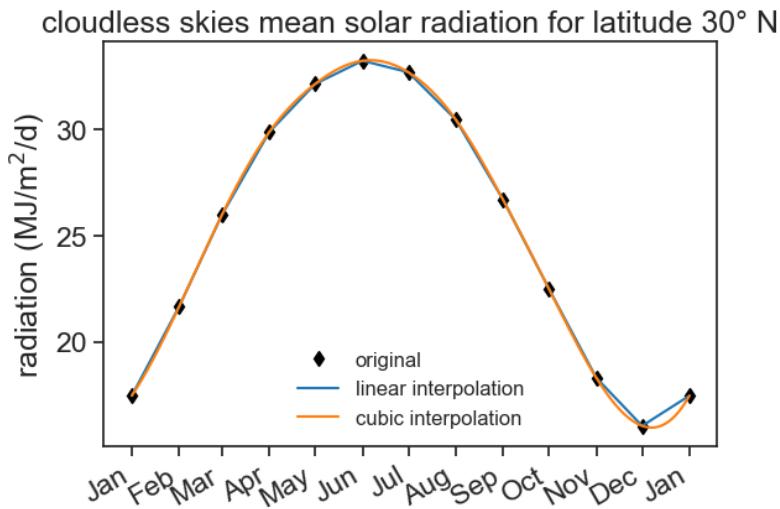
radiation	
date	
2021-12-30	16.04
2021-12-31	16.04
2022-01-01	17.46

This does the job, but I want something better, not step functions. The radiation should vary smoothly from day to day. Let's use other kinds of interpolation.

## 7.3 Interpolation

```
df_linear = df['radiation'].resample('D').interpolate(method='time').to_frame()
df_cubic = df['radiation'].resample('D').interpolate(method='cubic').to_frame()

fig, ax = plt.subplots()
ax.plot(df['radiation'], color='black', marker='d', linestyle='None', label="original")
ax.plot(df_linear['radiation'], color='tab:blue', label="linear interpolation")
ax.plot(df_cubic['radiation'], color='tab:orange', label="cubic interpolation")
ax.set(ylabel=r'radiation (MJ/m$^2$/d)',
       title="cloudless skies mean solar radiation for latitude 30° N")
ax.legend(frameon=False, fontsize=12)
ax.xaxis.set_major_locator(mdates.MonthLocator())
date_form = DateFormatter("%b")
ax.xaxis.set_major_formatter(date_form)
plt.gcf().autofmt_xdate() # makes slanted dates
```



There are many ways to fill NaNs and to interpolate. A nice detailed guide can be [found here](#).

## 7.4 Dealing with missing rows

There is a difference between missing rows and NaN values. The first is when we have a gap in the index, and the second is when we have a value that is missing.

Now we will produce two derivative datasets, one where the month of August is full of NaN values, the other where all the rows of the month of August are removed.

```
df_missing = df_cubic[~df_cubic.index.month.isin([8])]
df_nan = df_cubic.copy()
df_cubic.loc[df_cubic.index.month.isin([8])] = np.nan
# see the difference between the two dataframes
print(df_missing['2021-07-29':])
print(df_nan['2021-07-29':])
```

```
radiation
date
2021-07-29  30.731457
2021-07-30  30.636204
```

```
2021-07-31  30.539054
2021-09-01  26.670000
2021-09-02  26.534642
...
...
2021-12-28  16.995662
2021-12-29  17.103029
2021-12-30  17.216164
2021-12-31  17.335133
2022-01-01  17.460000
```

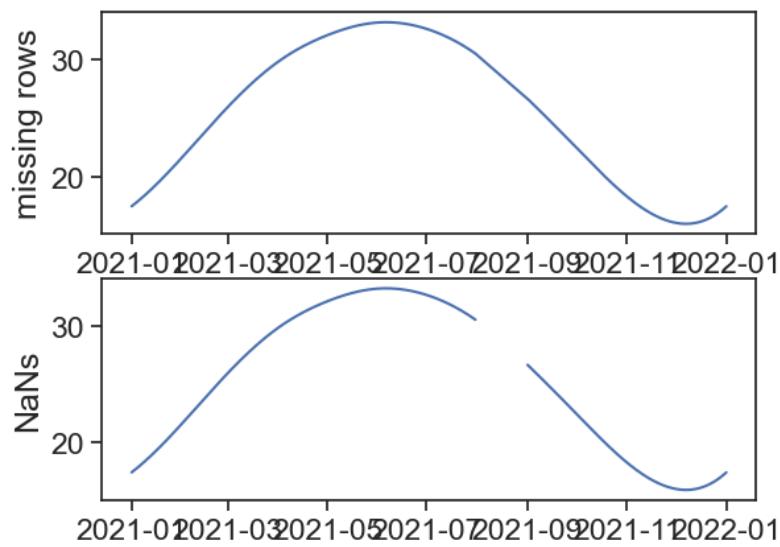
```
[126 rows x 1 columns]
radiation
date
2021-07-29  30.731457
2021-07-30  30.636204
2021-07-31  30.539054
2021-08-01      NaN
2021-08-02      NaN
...
...
2021-12-28  16.995662
2021-12-29  17.103029
2021-12-30  17.216164
2021-12-31  17.335133
2022-01-01  17.460000
```

```
[157 rows x 1 columns]
```

When plotting the data, we see that missing rows don't produce gaps in the lineplot. Depending on the data, we would never know this just by looking at the plot!

```
fig, ax = plt.subplots(2,1)
ax[0].plot(df_missing)
ax[0].set_ylabel("missing rows")
ax[1].plot(df_nan)
ax[1].set_ylabel("NaNs")
```

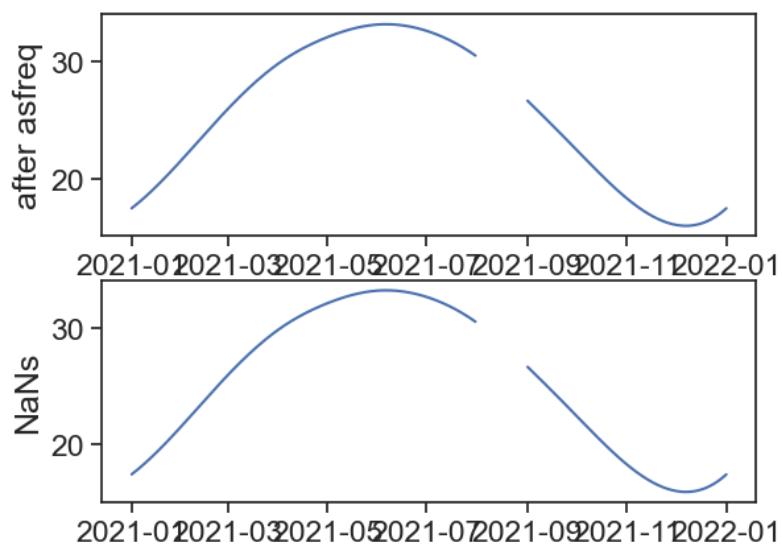
```
Text(0, 0.5, 'NaNs')
```



Use the `asfreq` method to fill the missing rows with NaNs.

```
df_asfreq = df_missing.asfreq('D')
fig, ax = plt.subplots(2,1)
ax[0].plot(df_asfreq)
ax[0].set_ylabel("after asfreq")
ax[1].plot(df_nan)
ax[1].set_ylabel("NaNs")
```

```
Text(0, 0.5, 'NaNs')
```



## 8 interpolation

Interpolation is the act of getting data you don't have from data you already have. We used some interpolation when upsampling, and now it is time to talk about it a little bit more in depth.

There is no one correct way of interpolating, the method you use depends in the end on what you want to accomplish, what are your (hidden or explicit) assumptions, etc. Let's see a few examples.

## 9 practice

Let's try to answer the following questions:

What is the mean temperature for each month?

First we have to divide temperature data by month, and then take the average for each month.

a possible solution

```
df_month = df['temperature'].resample('M').mean()
```

For each month, what is the mean of the daily maximum temperature? What about the minimum?

This is a bit trickier.

1. We need to find the maximum/minimum temperature for each day.
2. Only then we split the daily data by month and take the average.

a possible solution

```
df_day['max temp'] = df['temperature'].resample('D').max()
df_month['max temp'] = df_day['max temp'].resample('MS').mean()
```

What is the average night temperature for every season?

What about the day temperature?

1. We need to filter our data to contain only night times.
2. We need to divide rain data by seasons (3 months), and then take the mean for each season.

a possible solution

```
# filter only night data
df_night = df.loc[((df.index.hour < 6) | (df.index.hour >= 18))]
season_average_night_temp = df_night['temperature'].resample('Q').mean()
```

another possible solution

```
# filter using between_time
df_night = df.between_time('18:00', '06:00', inclusive='left')
season_average_night_temp = df_night['temperature'].resample('Q').mean()
```

What is the daily precipitation?

First we have to divide rain data by day, and then take the sum for each day.

a possible solution

```
daily_precipitation = df['rain'].resample('D').sum()
```

How much rain was there every month?

We have to divide rain data by month, and then sum the totals of each month.

a possible solution

```
monthly_precipitation = df['rain'].resample('M').sum()
```

How many rainy days were there each month?

1. We need to sum rain by day.
2. We need to count how many days are there each month where `rain > 0`.

a possible solution

```
daily_precipitation = df['rain'].resample('D').sum()
only_rainy_days = daily_precipitation.loc[daily_precipitation > 0]
rain_days_per_month = only_rainy_days.resample('M').count()
```

How many days, hours, and minutes were between the last rain of the season (Malkosh) to the first (Yoreh)?

1. We need to divide our data into two: `rainy_season_1` and `rainy_season_2`.
2. We need to find the time of the last rain in `rainy_season_1`.
3. We need to find the time of the first rain in `rainy_season_2`.
4. We need to compute the time difference between the two dates.

a possible solution

```
split_date = '2019-08-01'
rainy_season_1 = df[:split_date] # everything before split date
rainy_season_2 = df[split_date:] # everything after split date
malkosh = rainy_season_1['rain'].loc[rainy_season_1['rain'] > 0].last_valid_index()
yoreh = rainy_season_2['rain'].loc[rainy_season_2['rain'] > 0].first_valid_index()
dry_period = yoreh - malkosh
# extracting days, hours, and minutes
days = dry_period.days
hours = dry_period.components.hours
minutes = dry_period.components.minutes
print(f'The dry period of 2019 was {days} days, {hours} hours and {minutes} minutes.')
```

**i** What was the雨iest morning (6am-12pm) of the year?  
Bonus, what about the雨iest night (6pm-6am)?

1. We need to filter our data to contain only morning times.
2. We need to sum rain by day.
3. We need to find the day with the maximum value.

a possible solution

```
# filter to only day data
morning_df = df.loc[((df.index.hour >= 6) & (df.index.hour < 18))]
morning_rain = morning_df['rain'].resample('D').sum()
rainiest_morning = morning_rain.idxmax()
# plot
morning_rain.plot()
plt.axvline(rainiest_morning, c='r', alpha=0.5, linestyle='--')
```

bonus solution

```
# filter to only night data
df_night = df.loc[((df.index.hour < 6) | (df.index.hour >= 18))]
# resampling night for each day is tricky because the date changes at 12:00. We can do this by
# we shift the time back by 6 hours so all the data for the same night will have the same date
df_shifted = df_night.tshift(-6, freq='H')
night_rain = df_shifted['rain'].resample('D').sum()
rainiest_night = night_rain.idxmax()
# plot
night_rain.plot()
plt.axvline(rainiest_night, c='r', alpha=0.5, linestyle='--')
```

Note: this whole webpage is actually a Jupyter Notebook rendered as html. If you want to know how to make interactive graphs, go to the top of the page and click on “Code”

# 10 FAQ

## 10.1 How to resample by year, but have it end in September?

This is called [anchored offset](#). One possible use to it is to calculate statistics according to the hydrological year that, for example, ends in September.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters

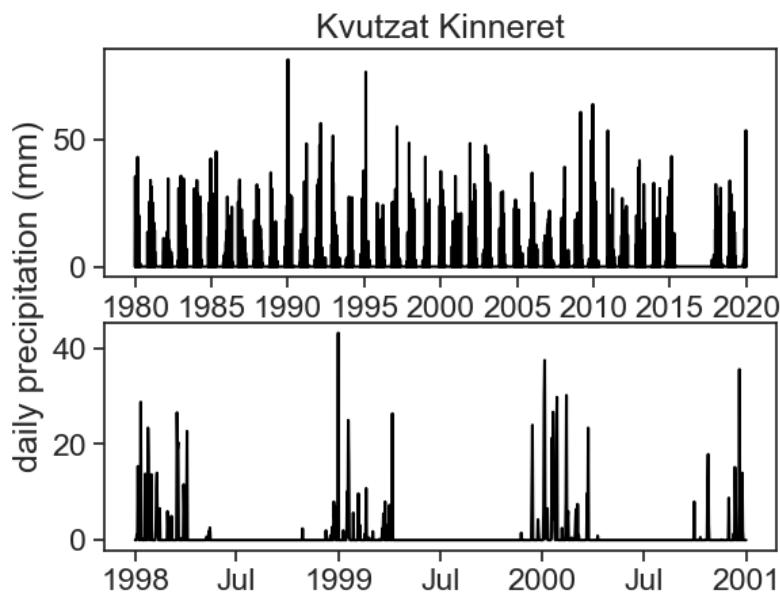
filename = "../archive/data/Kinneret_Kvuza_daily_rainfall.csv"
df = pd.read_csv(filename, na_values=['-'])
df.rename(columns={'Date': 'date',
                  'Daily Rainfall (mm)': 'rain'}, inplace=True)
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date')
df = df.resample('D').asfreq().fillna(0) # asfreq = replace
df
```

	Station	rain
date		
1980-01-02	Kinneret Kvaza 09/1977-08/2023	0.0
1980-01-03	0	0.0
1980-01-04	0	0.0

	Station	rain
date		
1980-01-05	Kinneret Kvaza 09/1977-08/2023	35.5
1980-01-06	Kinneret Kvaza 09/1977-08/2023	2.2
...	...	...
2019-12-26	Kinneret Kvaza 09/1977-08/2023	39.4
2019-12-27	Kinneret Kvaza 09/1977-08/2023	5.2
2019-12-28	Kinneret Kvaza 09/1977-08/2023	1.6
2019-12-29	0	0.0
2019-12-30	Kinneret Kvaza 09/1977-08/2023	0.1

```
fig, ax = plt.subplots(2,1)
ax[0].plot(df['rain'], color='black')
ax[1].plot(df.loc['1998':'2000', 'rain'], color='black')
locator = mdates.AutoDateLocator(minticks=4, maxticks=8)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
fig.text(0.02, 0.5, 'daily precipitation (mm)', va='center', rotation='vertical')
ax[0].set_title("Kvutzat Kinneret")
```

Text(0.5, 1.0, 'Kvutzat Kinneret')



We see a marked dry season during the summer, so let's assume the Hydrological Year ends in September.

```
df_year = df.resample('A-SEP').sum()
df_year = df_year.loc['1980':'2003']
df_year
```

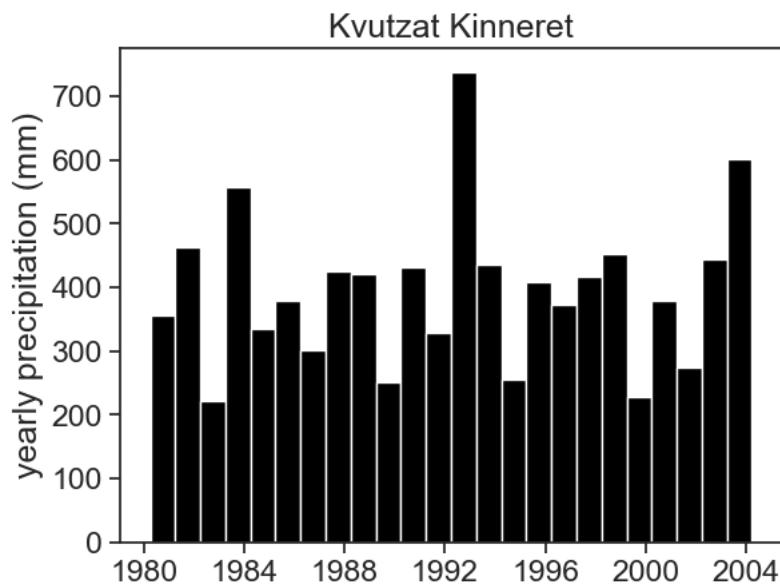
```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_94063/2047090134.py:1: FutureWarning:
```

	rain
date	
1980-09-30	355.5
1981-09-30	463.1
1982-09-30	221.7
1983-09-30	557.1
1984-09-30	335.3
1985-09-30	379.8
1986-09-30	300.7
1987-09-30	424.7
1988-09-30	421.6

date	rain
1989-09-30	251.6
1990-09-30	432.5
1991-09-30	328.3
1992-09-30	738.4
1993-09-30	434.9
1994-09-30	255.4
1995-09-30	408.6
1996-09-30	373.0
1997-09-30	416.2
1998-09-30	451.9
1999-09-30	227.8
2000-09-30	378.9
2001-09-30	273.9
2002-09-30	445.2
2003-09-30	602.4

```
fig, ax = plt.subplots()
ax.bar(df_year.index, df_year['rain'], color='black',
       width=365)
ax.set_ylabel("yearly precipitation (mm)")
ax.set_title("Kvutzat Kinneret")
```

Text(0.5, 1.0, 'Kvutzat Kinneret')



## 10.2 When upsampling, how to fill missing values with zero?

We did that in the example above, like this:

```
df = df.resample('D').asfreq().fillna(0) # asfreq = replace
```

# **Part III**

## **smoothing**

# 11 motivation

This is the temperature for the Yatir Forest (Shani station, see [map](#)), between 2 and 5 of January 2022. Data is in intervals of 10 minutes, and was downloaded from the Israel Meteorological Service.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import requests
import json
import os
# %matplotlib widget
```

```
# read token from file
with open('../archive/IMS-token.txt', 'r') as file:
    TOKEN = file.readline()
# 28 = SHANI station
STATION_NUM = 28
start = "2022/01/01"
end = "2022/01/07"
filename = 'shani_2022_january.json'

# check if the JSON file already exists
# if so, then load file
```

```

if os.path.exists(filename):
    with open(filename, 'r') as json_file:
        data = json.load(json_file)
else:
    # make the API request if the file doesn't exist
    url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/?from={start}&to={end}"
    headers = {'Authorization': f'ApiToken {TOKEN}'}
    response = requests.get(url, headers=headers)
    data = json.loads(response.text.encode('utf8'))

    # save the JSON data to a file
    with open(filename, 'w') as json_file:
        json.dump(data, json_file)
# show data to see if it's alright
# data

df = pd.json_normalize(data['data'], record_path=['channels'], meta=['datetime'])
df['date'] = (pd.to_datetime(df['datetime'])
              .dt.tz_localize(None) # ignores time zone information
              )
df = df.pivot(index='date', columns='name', values='value')
# df

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def centered_dates(ax):
    date_form = DateFormatter("%d %b") # %d 3-letter-Month
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax.xaxis.get_minor_ticks():

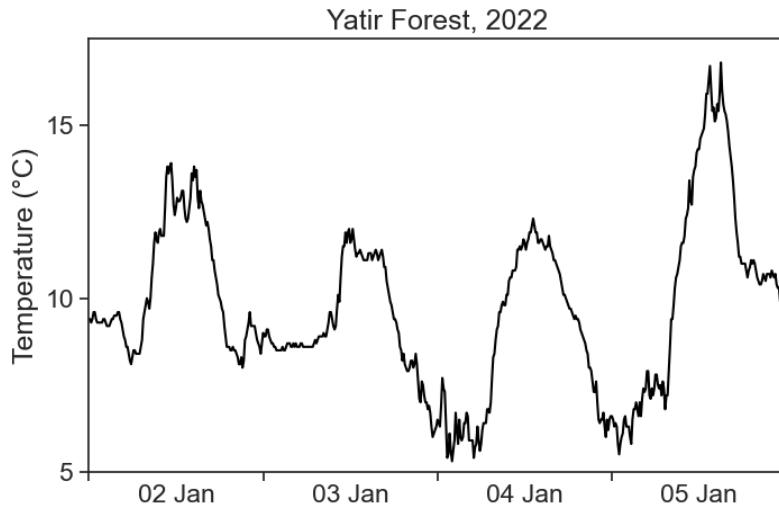
```

```

        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')

fig, ax = plt.subplots(figsize=(8,5))
start = "2022-01-02"
end = "2022-01-05"
df = df.loc[start:end]
ax.plot(df['TD'], color='black')
ax.set(ylim=[5, 17.5],
       xlim=[df.index[0], df.index[-1]],
       ylabel="Temperature (°C)",
       title="Yatir Forest, 2022",
       yticks=[5,10,15])
centered_dates(ax)
fig.savefig("YF-temperature_2022_jan.png", dpi=300)

```



We see that the temperature curve has a rough profile. Can we find ways of getting smoother curves?

We learned how to average over a window with `resample`. Let's try that for a 2-hour window:

```

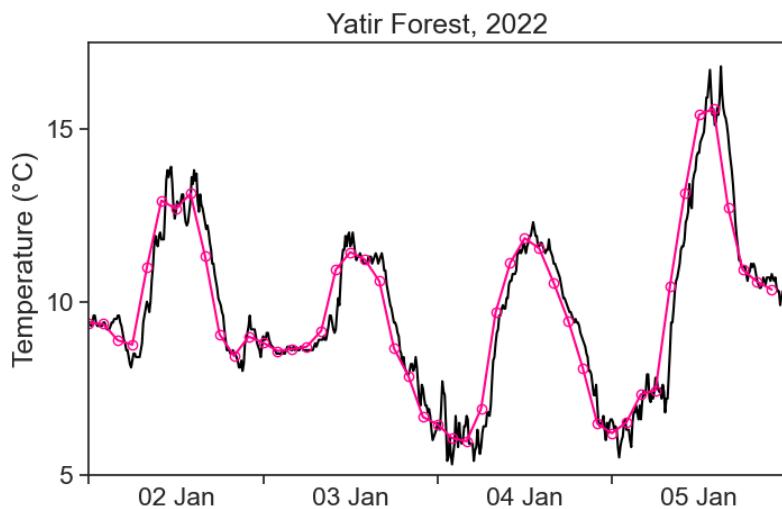
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df['TD'], color='black')

```

```

ax.plot(df['TD'].resample('2H').mean(),
        color='xkcd:hot pink', ls='-' ,
        marker="o", mfc="None")
ax.set(ylim=[5, 17.5],
       xlim=[df.index[0], df.index[-1]],
       ylabel="Temperature (°C)",
       title="Yatir Forest, 2022",
       yticks=[5,10,15])
centered_dates(ax)

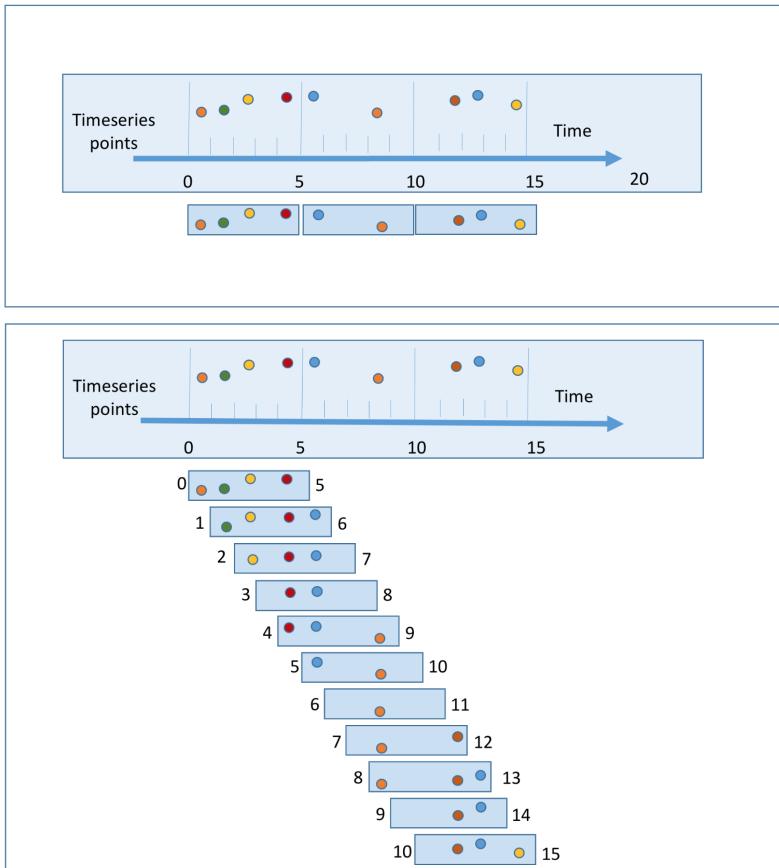
```



The temperature profile now is much smoother, but when using `resample`, we lost temporal resolution. Our original data had 10-minute frequency, and now we have a 2-hour frequency.

How can we get a smoother curve without losing resolution?

## 11.1 Tumbling vs Sliding



## 12 sliding window

This is the temperature for the Yatir Forest, between 2 and 5 of January 2022. Data (download .csv here) is in intervals of 10 minutes, and was downloaded from the Israel Meteorological Service.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker
import os
import warnings
import scipy
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
# %matplotlib widget

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def centered_dates(ax):
    date_form = DateFormatter("%d %b") # %d 3-letter-Month
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
```

```

ax.xaxis.set_major_formatter(ticker.NullFormatter())
# set minor tick labels as defined above
ax.xaxis.set_minor_formatter(date_form)
# completely erase minor ticks, center tick labels
for tick in ax.xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

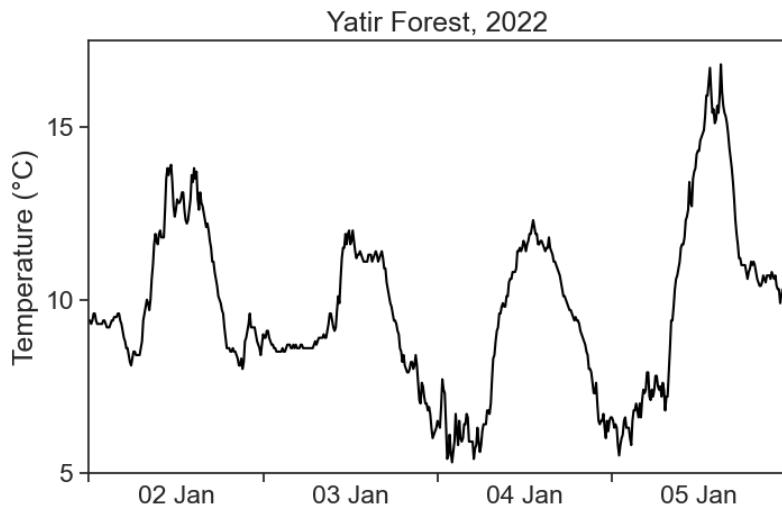
df = pd.read_csv('shani_2022_january.csv', parse_dates=['date'], index_col='date')

fig, ax = plt.subplots(figsize=(8,5))
start = "2022-01-02"
end = "2022-01-05"
df = df.loc[start:end]
ax.plot(df['TD'], color='black')

plot_settings = {
    'ylim': [5, 17.5],
    'xlim': [df.index[0], df.index[-1]],
    'ylabel': "Temperature (°C)",
    'title': "Yatir Forest, 2022",
    'yticks': [5, 10, 15]
}

ax.set(**plot_settings)
centered_dates(ax)

```



We see that the temperature curve has a rough profile. Can we find ways of getting smoother curves?

## 12.1 convolution

Convolution is a fancy word for averaging a time series using a sliding window. We will use the terms **convolution**, **running average**, and **rolling average** interchangeably. See the animation below. We take all temperature values inside a window of width 500 minutes (51 points), and average them with equal weights. The weights profile is called **kernel**.

The pink curve is much smoother than the original! However, the running average cannot describe sharp temperature changes. If we decrease the window width to 200 minutes (21 points), we get the following result.

There is a tradeoff between the smoothness of a curve, and its ability to describe sharp temporal changes.

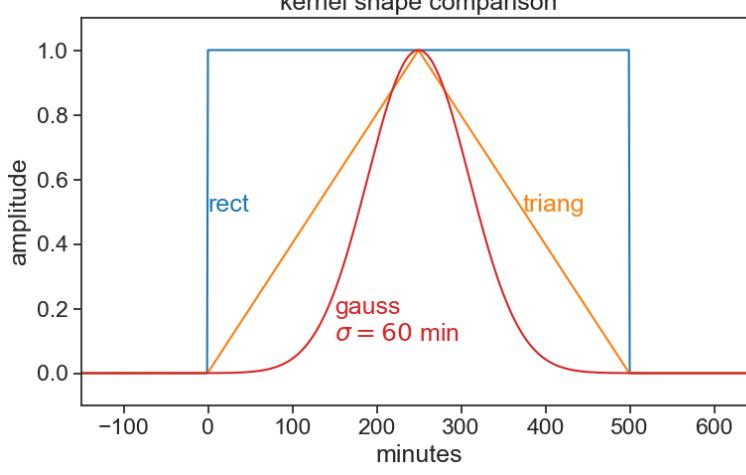
## 12.2 kernels

We can modify our running average, so that values closer to the center of the window have higher weights, and those fur-

ther away count less. This is achieved by changing the weight profile, or the shape of the kernel. We see below the result of a running average using a triangular window of base 500 minutes (51 points).

Things can get as fancy as we want. Instead of a triangular kernel, which has sharp edges, we can choose a smoother gaussian kernel, see the difference below. We used a gaussian kernel with 60-minute standard deviation.

See how the three kernel shapes compare. There are *many* kernels to chose from.



## 12.3 math

The definition of a convolution between signal  $f(t)$  and kernel  $k(t)$  is

$$(f * k)(t) = \int f(\tau)k(t - \tau)d\tau.$$

The expression  $f * k$  denotes the convolution of these two functions. The argument of  $k$  is  $t - \tau$ , meaning that the kernel runs from left to right (as  $t$  does), and at every point the two signals ( $f$  and  $k$ ) are multiplied together. It is the product of the signal with the weight function  $k$  that gives us an average. Because of  $-\tau$ , the kernel is flipped backwards, but this has no effect to

symmetric kernels, like to ones in the examples above. Finally, the actual running average is not the convolution, but

$$\frac{(f * k)(t)}{\int k(t)dt}.$$

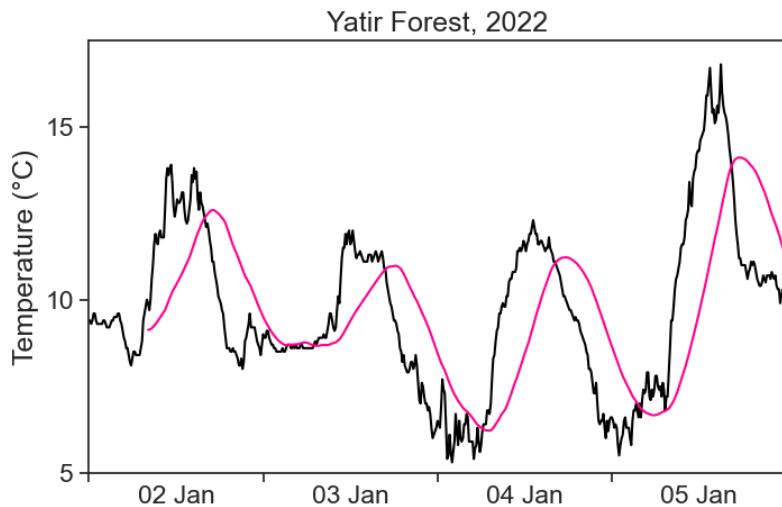
Whenever the integral of the kernel is 1, then the convolution will be identical with the running average.

## 12.4 numerics

Running averages are very common tools in time-series analysis. The `pandas` package makes life quite simple. For example, in order to calculate the running average of temperature using a rectangular kernel, one writes:

```
df['temp_smoothed'] = (
    df['TD'].rolling(window='500min',
                     min_periods=50    # comment this to see what happens
                     )
    .mean()
)

fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df['TD'], color='black')
ax.plot(df['temp_smoothed'], color='xkcd:hot pink')
ax.set(**plot_settings)
centered_dates(ax)
```



The pink curve looks smooth, but why does it lag behind the data?! What's going on?

#### 12.4.1 7-day average of COVID-19 infections

During the COVID-19 pandemic, we would see graphs like this all the time in the news:

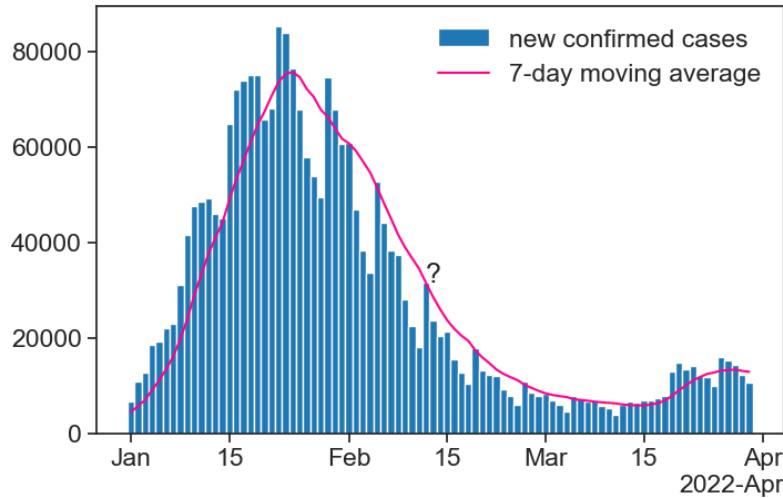
```
# data from https://health.google.com/covid-19/open-data/raw-data?loc=IL
# define the local file path
local_file_path = 'COVID_19_israel.csv'
# check if the local file exists
if os.path.exists(local_file_path):
    # if the local file exists, load it
    covid_IL = pd.read_csv(local_file_path, parse_dates=['date'], index_col='date')
else:
    # if the local file doesn't exist, download from the URL
    url = "https://storage.googleapis.com/covid19-open-data/v3/location/IL.csv"
    covid_IL = pd.read_csv(url, parse_dates=['date'], index_col='date')
    # save the downloaded data to the local file for future use
    covid_IL.to_csv(local_file_path)

df_covid = covid_IL['new_confirmed'].to_frame()
df_covid['7d_avg'] = df_covid['new_confirmed'].rolling(window='7D').mean()
```

```

fig, ax = plt.subplots(figsize=(8,5))
st = '2022-01-01'
en = '2022-03-30'
new_cases = ax.bar(df_covid[st:en].index, df_covid.loc[st:en,'new_confirmed'],
                    color="tab:blue", width=1)
mov_avg, = ax.plot(df_covid.loc[st:en,'7d_avg'],
                    color='xkcd:hot pink')
ax.legend(handles=[new_cases, mov_avg],
           labels=['new confirmed cases', '7-day moving average'],
           frameon=False)
weird_day = "2022-02-12"
weird_day_x = mdates.date2num(dt.datetime.strptime(weird_day, "%Y-%m-%d"))
ax.text(weird_day_x, df_covid.loc[weird_day,'new_confirmed'], "?")
# formating dates on x axis
locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

```



Take a look at the moving average next to the question mark. How can it be that high, when all the bars around that date are lower? Is the calculation right?

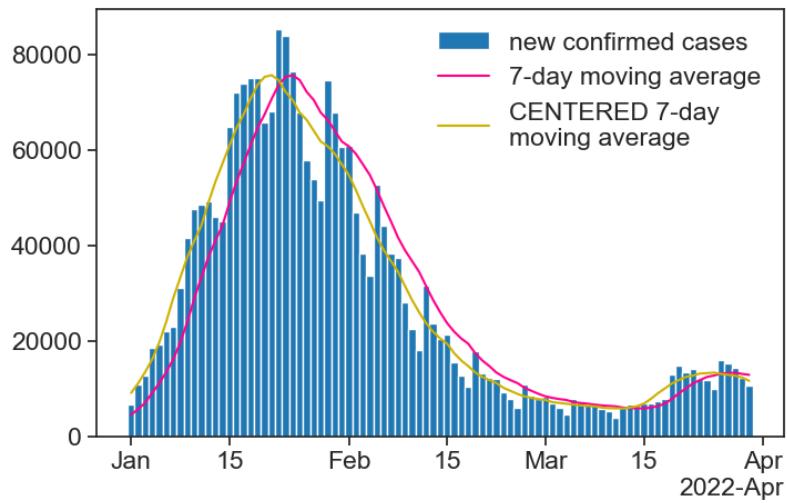
The answer is that the result of the moving average is assigned to the right-most date in the running window. This is reasonable for COVID-19 cases: for a given day, I can only calculate

a 7-day average based on **past** values, I don't know what the future will be.

There is a simple way of assigning the result to the center of the window:

```
df_covid['7d_avg_center'] = (
    df_covid['new_confirmed']
        .rolling(window='7D',
                center=True) # THIS
        .mean()
)
```

```
fig, ax = plt.subplots(figsize=(8,5))
st = '2022-01-01'
en = '2022-03-30'
new_cases = ax.bar(df_covid[st:en].index, df_covid.loc[st:en,'new_confirmed'],
                    color="tab:blue", width=1)
mov_avg, = ax.plot(df_covid.loc[st:en,'7d_avg'],
                    color='xkcd:hot pink')
mov_avg_center, = ax.plot(df_covid.loc[st:en,'7d_avg_center'],
                           color='xkcd:mustard')
ax.legend(handles=[new_cases, mov_avg, mov_avg_center],
           labels=['new confirmed cases',
                   '7-day moving average',
                   'CENTERED 7-day\nmoving average'],
           frameon=False)
# formating dates on x axis
locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)
```



As a rule, we will used a **centered** moving average (`center=True`), unless stated otherwise. Also, only use `min_periods` if you know what you are doing.

### 12.4.2 gaussian

You can easily change the kernel shape by using the `win_type` argument. See how to perform a rolling mean with a gaussian kernel:

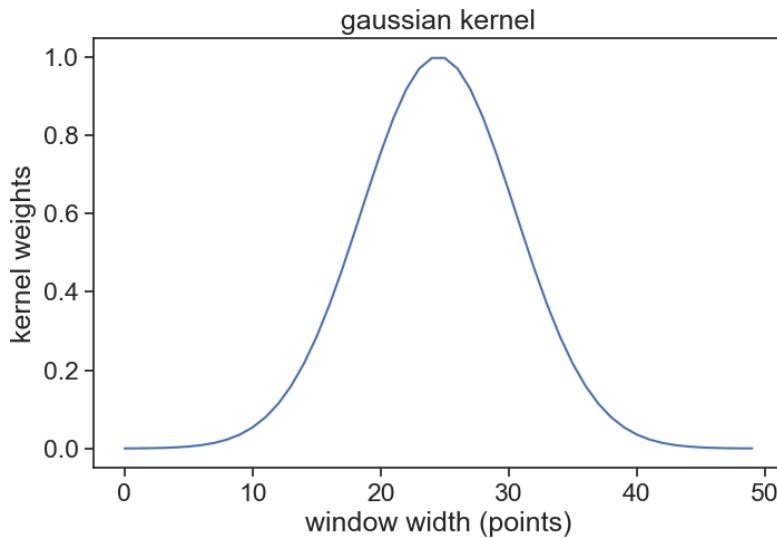
```
(  
df['temperature'].rolling(window=window_width,  
                           center=True,  
                           win_type="gaussian")  
    .mean(std=std_gaussian)  
)
```

where

- `window_width` is an integer, number of points in your window
- `std_gaussian` is the standard deviation of your gaussian, **measured in sample points, not time!**

For instance, if we have measurements every 10 minutes, and our window width is 500 minutes, then `window_width = 500/10 + 1` (first and last included). If we want a standard deviation of 60 minutes, then `std_gaussian = 6`. The gaussian kernel will look like this:

```
window_width = 50 # in points = 500 min
std = 6 # in points = 60 min
fig, ax = plt.subplots(figsize=(8,5))
g = scipy.signal.gaussian(window_width, std)
ax.plot(g)
ax.set(xlabel="window width (points)",
       ylabel="kernel weights",
       title="gaussian kernel");
```



You can take a look at various options for kernel shapes [here](#), provided by the `scipy` package.

#### 12.4.3 triangular

Same idea as gaussian, but simpler, because we don't need to think about standard deviation.

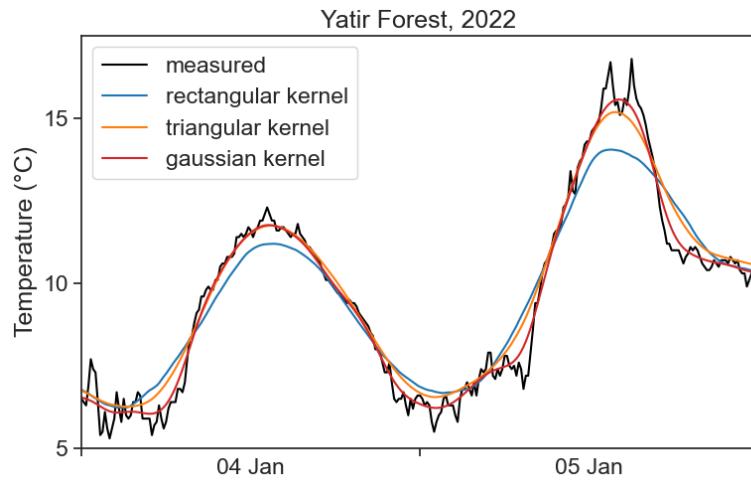
```

(
df['temperature'].rolling(window=window_width,
    center=True,
    win_type="triang")
    .mean()
)

```

## 12.5 which window shape and width to choose?

Sorry, there is not definite answer here... It really depends on your data and what you need to do with it. See below a comparison of all examples in the videos above.



One important question you need to ask is: what are the time scales associated with the processes I'm interested in? For example, if I'm interested in the daily temperature pattern, getting rid of 1-minute-long fluctuations would probably be ok. On the other hand, if we were to smooth the signal so much that all that can be seen are the temperature changes between summer and winter, then my smoothing got out of hand, and I threw away the very process I wanted to study.

All this is to say that you need to know in advance a few things about the system you are studying, otherwise you can't know what is "noise" that can be smoothed away.

## 13 not only averages

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import requests
import json
import os
# %matplotlib widget
```

```
# read token from file
with open('../archive/IMS-token.txt', 'r') as file:
    TOKEN = file.readline()
# 28 = SHANI station
STATION_NUM = 28
start = "2022/01/01"
end = "2022/01/07"
filename = 'shani_2022_january.json'

# check if the JSON file already exists
# if so, then load file
if os.path.exists(filename):
    with open(filename, 'r') as json_file:
        data = json.load(json_file)
else:
```

```

# make the API request if the file doesn't exist
url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/?from={start}&to={end}"
headers = {'Authorization': f'ApiToken {TOKEN}'}
response = requests.get(url, headers=headers)
data = json.loads(response.text.encode('utf8'))

# save the JSON data to a file
with open(filename, 'w') as json_file:
    json.dump(data, json_file)
# show data to see if it's alright
# data

df = pd.json_normalize(data['data'], record_path=['channels'], meta=['datetime'])
df['date'] = (pd.to_datetime(df['datetime'])
              .dt.tz_localize(None) # ignores time zone information
              )
df = df.pivot(index='date', columns='name', values='value')
# let's work only with a few days, and only temperature
start = "2022-01-02"
end = "2022-01-05"
df = df.loc[start:end, 'TD'].to_frame()
df.rename(columns={"TD": "temp"}, inplace=True)
# df

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def centered_dates(ax):
    date_form = DateFormatter("%d %b") # %d 3-letter-Month
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels

```

```

for tick in ax.xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

# creating the dictionary with the desired settings
plot_settings = {
    'ylim': [5, 17.5],
    'xlim': [df.index[0], df.index[-1]],
    'ylabel': 'Temperature (°C)',
    'title': 'Yatir Forest, 2022',
    'yticks': [5, 10, 15]
}

```

Let's see on a graph the average temperature, with an envelope of 1 standard deviation around it:

```

df['mean'] = df['temp'].rolling('3H', center=True).mean()
df['std'] = df['temp'].rolling('3H', center=True).std()

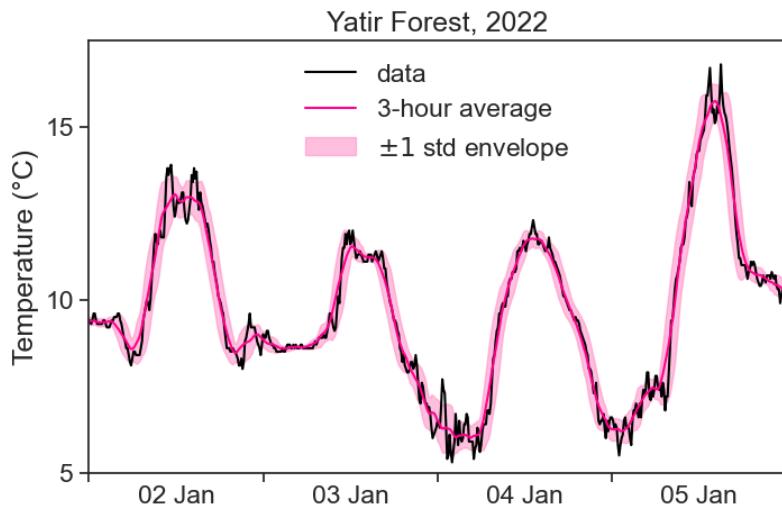
fig, ax = plt.subplots(figsize=(8,5))

plot_std = ax.fill_between(df.index,
                           df['mean'] + df['std'],
                           df['mean'] - df['std'],
                           color="xkcd:pink", alpha=0.5)
plot_data, = ax.plot(df['temp'], color='black')
plot_mean, =ax.plot(df['mean'], color='xkcd:hot pink')

ax.legend([plot_data, plot_mean, plot_std],
          ['data', '3-hour average', r"$\pm$ std envelope"],
          frameon=False)

# applying the settings to the ax object
ax.set(**plot_settings)
centered_dates(ax)
# fig.savefig("YF-temperature_2022_jan.png", dpi=300)

```



### 13.1 Confidence Interval

We can calculate anything we want inside the sliding window. One good example is the **Confidence Interval of the Mean**, given by:

$$CI(\alpha) = Z(\alpha) \cdot SE.$$

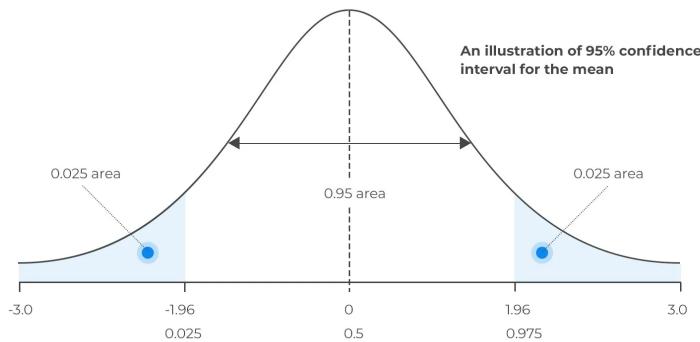
$Z(\alpha)$  is the Z-score corresponding to the chosen confidence level  $\alpha$ . The most commonly used confidence level is 95%, which corresponds to a Z-score of 1.96. What does this mean? This means that we expect to find 95% of the points within  $\pm 1.96$  standard deviations away from the mean.

This is called “ ” - in hebrew.

- $Z(\alpha)$  = Z-score.
- SE = standard error.



## 95% Interval



You can find the Z-score using the following python code:

```
from scipy.stats import norm

confidence_level = 0.95
# 5% outside
out = 1 - confidence_level
# 0.975 of points to the left of right boundary
p = 1 - out/2
# inverse of cdf: 0.975 of the points will be smaller than what distance (in sigma units)?
z_score = norm.ppf(p)
print(f"z-score = {z_score}")
```

**z-score = 1.959963984540054**

If you are still not convinced why we need 0.975 instead of 0.95, read this [excellent response on stackoverflow](#).

SE is the standard error:

$$SE = \frac{\sigma}{\sqrt{N}}.$$

We can write a function to calculate the confidence interval of the mean, and use it with the sliding window:

- $\sigma$  = standard deviation.
- $N$  = number of points.

```

def std_error_of_the_mean(window):
    return window.std() / np.sqrt(window.count())

def confidence_interval(window):
    return z_score * std_error_of_the_mean(window)

df['std_error'] = (
    df['temp'].rolling('3H',
                       center=True)
    .apply(std_error_of_the_mean)
)
df['confidence_int'] = (
    df['temp'].rolling('3H',
                       center=True)
    .apply(confidence_interval)
)

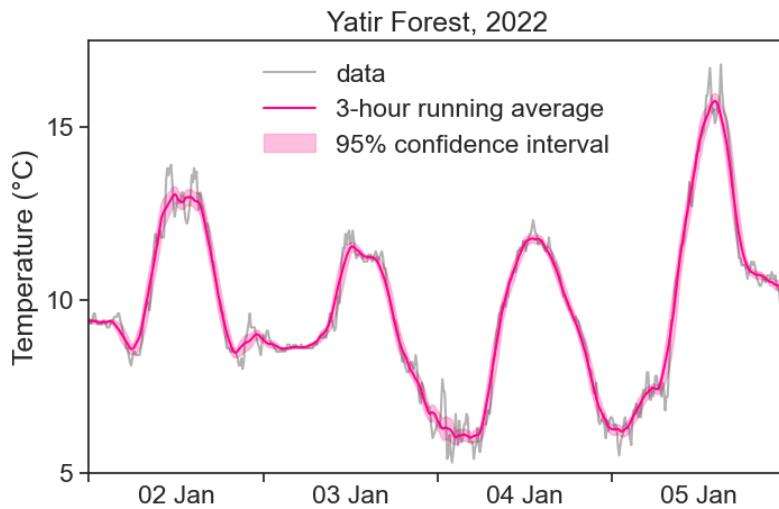
fig, ax = plt.subplots(figsize=(8,5))

plot_std = ax.fill_between(df.index,
                           df['mean'] + df['confidence_int'],
                           df['mean'] - df['confidence_int'],
                           color="xkcd:pink", alpha=0.5)
plot_data, = ax.plot(df['temp'], color='black', alpha=0.3)
plot_mean, =ax.plot(df['mean'], color='xkcd:hot pink')

ax.legend([plot_data, plot_mean, plot_std],
          ['data', '3-hour running average', r"95% confidence interval"],
          frameon=False)

# applying the settings to the ax object
ax.set(**plot_settings)
centered_dates(ax)
# fig.savefig("YF-temperature_2022_jan.png", dpi=300)

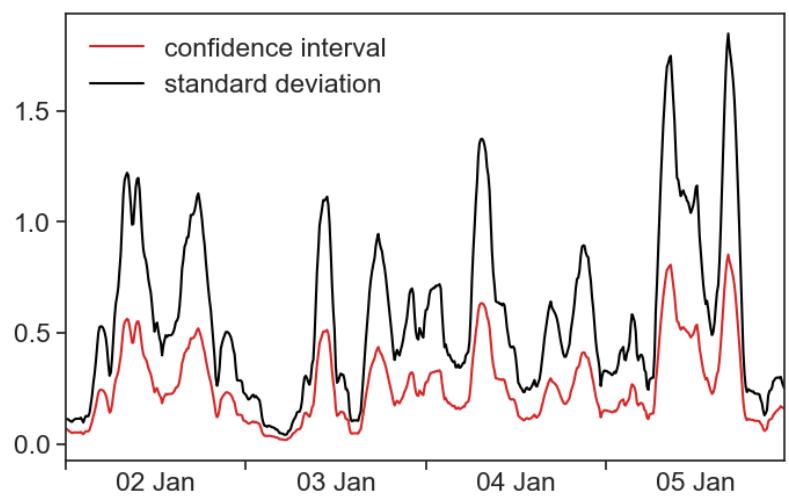
```



When the time series has a regular sampling frequency, all positions of the running window will have the same number of data points in them. Because the Confidence Interval is proportional to the Standard Error, and the SE is proportional to the Standard Deviation ( $\sqrt{N}$  is constant), then the envelope created by the CI is identical to the envelope created by the standard deviation, up to a multiplying constant. Nice.

```
fig, ax = plt.subplots(figsize=(8,5))
plot_ci, = ax.plot(df['confidence_int'], color='tab:red')
plot_std, = ax.plot(df['std'], color="black")
ax.legend([plot_ci, plot_std],
          ['confidence interval', 'standard deviation'],
          frameon=False)

# applying the settings to the ax object
# ax.set(**plot_settings)
ax.set(xlim=[df.index[0], df.index[-1]])
centered_dates(ax)
```



## 14 fit

We will make a little parenthesis to talk about a very important topic: **fitting**.

See below temperature data inside and outside a greenhouse, for a period of about 2 weeks.



```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import altair as alt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
from scipy.optimize import curve_fit
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
# avoid "SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataF
pd.options.mode.chained_assignment = None # default='warn'
# %matplotlib widget

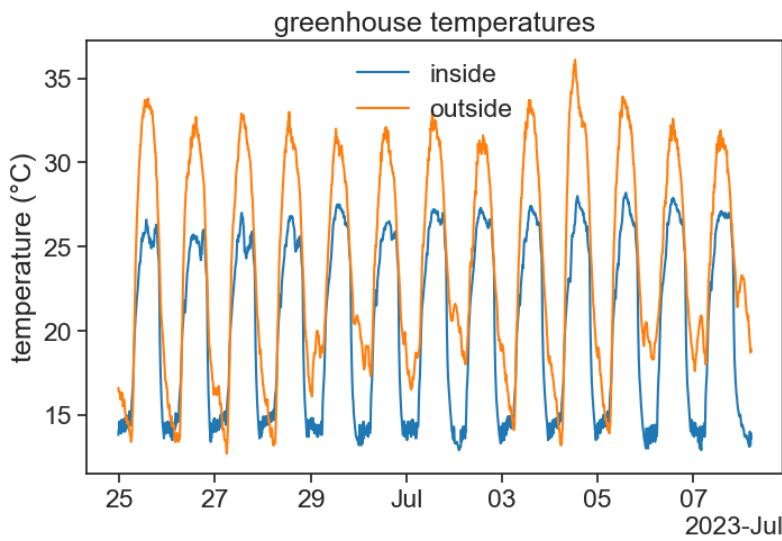
df = pd.read_csv('greenhouse_cooling.csv', index_col='time', parse_dates=True)
# df

fig, ax = plt.subplots(figsize=(8,5))

ax.plot(df['T_in'], c='tab:blue', label='inside')
ax.plot(df['T_out'], c='tab:orange', label='outside')
ax.set(ylabel='temperature (°C)',
       title="greenhouse temperatures")

# formating dates on x axis
locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

ax.legend(frameon=False);
```



Every evening, at 20:00, the air conditioning turns on, and we see a fast decrease in temperature:

```
df_fit = df['2023-06-29 20:10:00':'2023-06-29 22:00:00']

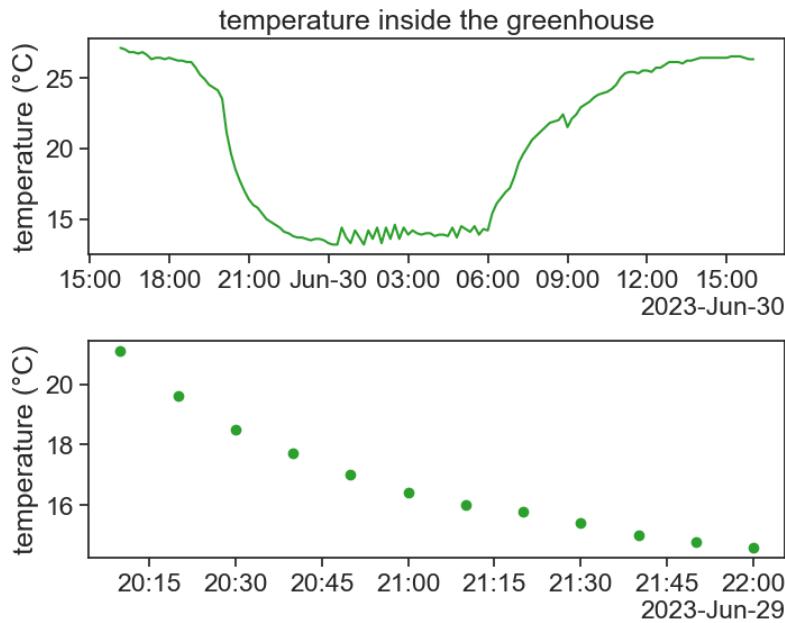
fig, ax = plt.subplots(2, 1, figsize=(8,6))
fig.subplots_adjust(hspace=0.4) # Adjust the vertical space between subplots

ax[0].plot(df.loc['2023-06-29 16:10:00':'2023-06-30 16:00:00', 'T_in'], color='tab:green')
ax[0].set(ylabel='temperature (°C)', title="temperature inside the greenhouse")

ax[1].scatter(df_fit['T_in'].index, df_fit['T_in'], color='tab:green')
ax[1].set(ylabel='temperature (°C)',)

# formating dates on x axis
locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[0].xaxis.set_major_locator(locator)
ax[0].xaxis.set_major_formatter(formatter)

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
```



The AC is able to bring the temperature down, but up to a limit. The AC can work at a maximum given power, and the cooler it is outside, the more effectively the AC will be able to bring down the temperature inside the greenhouse. We can imagine that the AC behaves as an effective external environment to the greenhouse, and the greenhouse cools down according to [Newton's law of cooling](#):

$$\frac{dT}{dt} = r \cdot (T_{\text{env}} - T)$$

The cooling rate is proportional to the difference in temperature between the inside and outside. Assuming  $T_{\text{env}}$  and  $r$  to be constant, the solution of this differential equation is:

$$T(t) = T_{\text{env}} + (T_0 - T_{\text{env}}) e^{-rt}.$$

We want to check if the temperature measured inside the greenhouse behaves like Newton's law of cooling, and if so, what can we say about the cooling coefficient  $r$  and about  $T_{\text{env}}$ .

- $T$  = the greenhouse temperature
- $T_{\text{env}}$  = the outside environment temperature
- $r$  = coefficient of heat transfer.
- $T_0$  = the initial greenhouse temperature

## 14.1 linear fit

The following is a **very short** introduction to curve fitting.  
The natural place to start is with a linear fit.

```
# the "fit" process can't deal with datetimes
# we therefore make a new column 'minutes', that will be used here
df_fit['minutes'] = (df_fit.index - df_fit.index[0]).total_seconds() / 60
# linear Fit (degree 1)
degree = 1
coeffs = np.polyfit(df_fit['minutes'], df_fit['T_in'], degree)
# linear Function
linear_function = np.poly1d(coeffs)
```

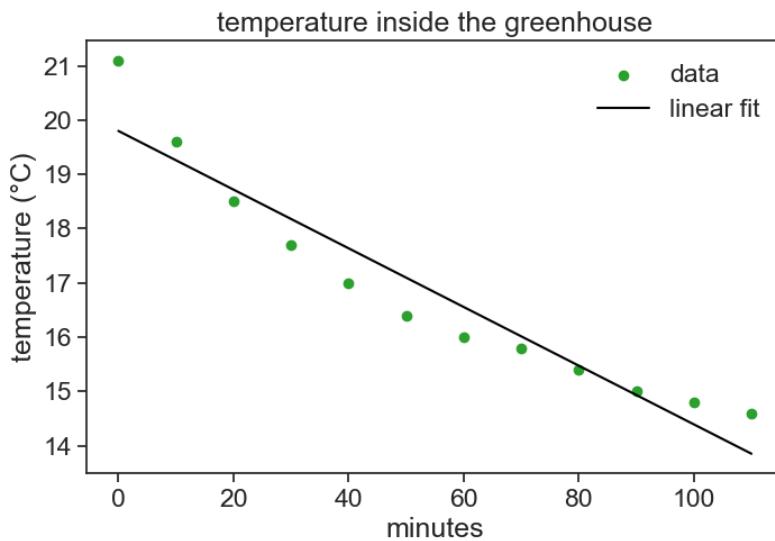
```
fig, ax = plt.subplots(figsize=(8,5))

ax.scatter(df_fit['minutes'],
           color='tab:green', label='data')
ax.plot(df_fit['minutes'], linear_function(df_fit['minutes']),
        color='black', label='linear fit')

ax.set(xlabel='minutes',
       ylabel='temperature (°C)',
       title="temperature inside the greenhouse")

ax.legend(frameon=False)
print(f"starting at {coeffs[1]:.2f} degrees,\nthe temperature decreases by {-coeffs[0]:.2f} de-
```

starting at 19.80 degrees,  
the temperature decreases by 0.05 degrees every minute.



The line above is the “best” straight line that describes our data. Defining the residual as the difference between our data and our model (straight line),

$$e = T_{\text{data}} - T_{\text{model}},$$

the straight line above is the one that **minimizes** the sum of the squares of residuals. For this reason, the method used above to fit a curve to the data is called “least-squares method”.

Can we do better than a straight line?

it minimizes the sum

## 14.2 polynomial fit

$$S = \sum_i e_i^2$$

```
# polynomial fit (degree 2)
degree = 2
coeffs2 = np.polyfit(df_fit['minutes'], df_fit['T_in'], degree)
quad_function = np.poly1d(coeffs2)

# polynomial fit (degree 2)
degree = 3
coeffs3 = np.polyfit(df_fit['minutes'], df_fit['T_in'], degree)
cubic_function = np.poly1d(coeffs3)
```

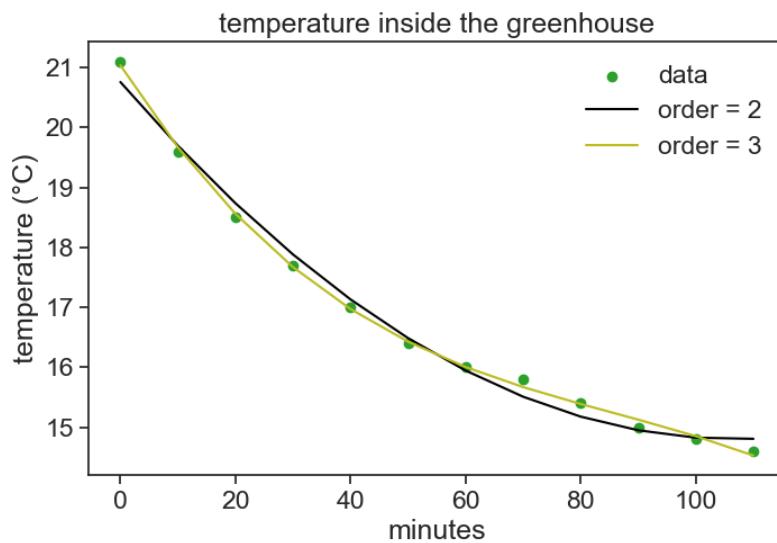
```

fig, ax = plt.subplots(figsize=(8,5))

ax.scatter(df_fit['minutes'], df_fit['T_in'],
           color='tab:green', label='data')
ax.plot(df_fit['minutes'], quad_function(df_fit['minutes']),
        color='black', label='order = 2')
ax.plot(df_fit['minutes'], cubic_function(df_fit['minutes']),
        color='tab:olive', label='order = 3')

ax.set(xlabel='minutes',
       ylabel='temperature (°C)',
       title="temperature inside the greenhouse")
ax.legend(frameon=False)

```



### 14.3 any function you want

Now let's get back to our original assumption, that the greenhouse cools according to Newton's cooling law. We can still use the least-squares method for any function we want!

```

def cooling(t, T_env, T0, r):
    """
    t = time
    other stuff = parameters to be fitted
    """
    return T_env + (T0 - T_env)*np.exp(-r*t)

t = df_fit['minutes'].values
y = df_fit['T_in'].values

T_init = df_fit['T_in'][0]

popt, pcov = curve_fit(f=cooling,
                        xdata=t,
                        ydata=y,
                        p0=(2, T_init, 0.5),
                        )
print(f"the optimal parameters are {popt}")

```

the optimal parameters are [14.01663586 21.0074623 0.02121802]

```

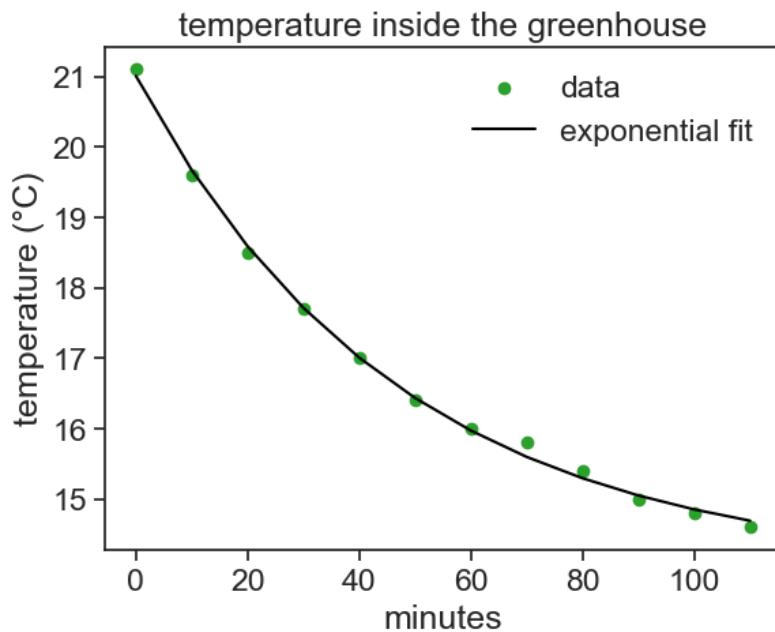
fig, ax = plt.subplots(sharex=True)

ax.scatter(df_fit['minutes'], df_fit['T_in'],
           color='tab:green', label='data')
ax.plot(t, cooling(t, *popt),
         color='black', label='exponential fit')

ax.set(xlabel='minutes',
       ylabel='temperature (°C)',
       title="temperature inside the greenhouse")

ax.legend(frameon=False)

```



That looks really good :)

We can use curve fitting to retrieve important parameters from our data. Let's write a function that executes the fit and returns two of the fitted parameters: `T_env` and `r`.

```
def run_fit(data):
    data['minutes'] = (data.index - data.index[0]).total_seconds() / 60
    t = data['minutes'].values
    y = data['T_in'].values
    T_init = data['T_in'][0]
    popt, pcov = curve_fit(f=cooling,                  # model function
                           xdata=t,                      # x data
                           ydata=y,                      # y data
                           p0=(2, T_init, 0.5),          # initial guess of the parameters
                           )
    return popt[0], popt[2]
```

We now apply this function to several consecutive evenings, and we keep the results in a new dataframe.

```

df_night = df.between_time('20:01', '22:01', inclusive='left')

# group by day and apply the function
# this is where the magic happens.
# if you are not familiar with "groupby", this will be hard to understand
result_series = df_night.groupby(df_night.index.date).apply(run_fit)

# convert the series to a dataframe
result_df = pd.DataFrame(result_series.tolist(), index=result_series.index, columns=['T_env'],
result_df.index = pd.to_datetime(result_df.index)
result_df

```

	T_env	r
2023-06-25	13.275540	0.019354
2023-06-26	13.331949	0.027034
2023-06-27	13.254827	0.018753
2023-06-28	13.392919	0.020449
2023-06-29	14.016636	0.021218
2023-06-30	13.807517	0.021749
2023-07-01	14.994207	0.023504
2023-07-02	14.314220	0.023705
2023-07-03	14.585848	0.019438
2023-07-04	14.377220	0.019504
2023-07-05	14.814939	0.021202
2023-07-06	14.667792	0.022264
2023-07-07	15.535115	0.024421

```

fig, ax = plt.subplots(3,1,sharex=True, figsize=(8,8))

ax[0].plot(df['T_in'], c='tab:blue', label='inside')
ax[0].plot(df['T_out'], c='tab:orange', label='outside')
ax[0].set(ylabel='temperature (°C)',
           title="actual temperatures",
           ylim=[10,45])

# formating dates on x axis
locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[0].xaxis.set_major_locator(locator)

```

```

ax[0].xaxis.set_major_formatter(formatter)

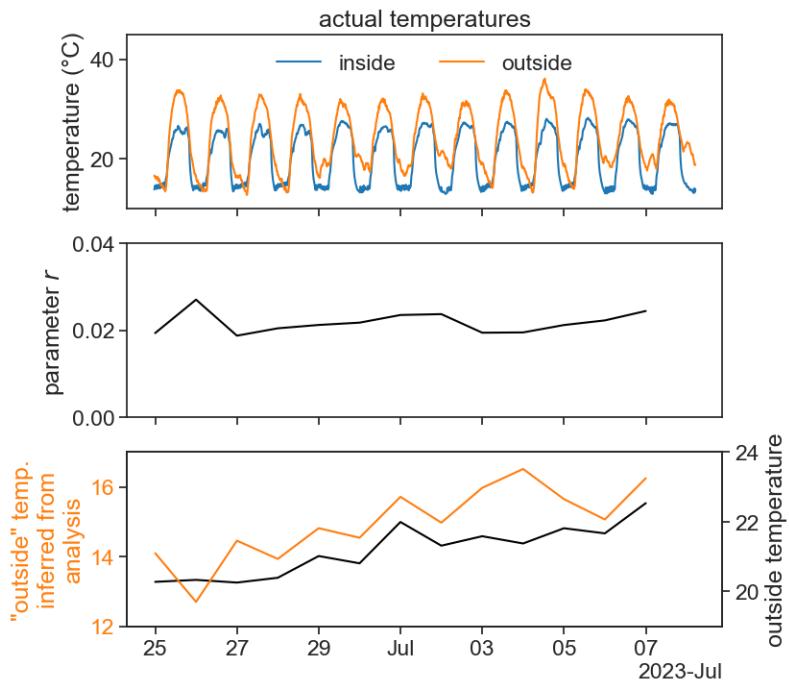
ax[0].legend(ncol=2, loc='upper center', frameon=False)

ax[1].plot(result_df['r'], color='black')
ax[1].set(ylabel=r"parameter $r$",
           ylim=[0, 0.04])

ax[2].plot(result_df['T_env'], color='black')
ax2b = ax[2].twinx()
ax2b.plot(df_night['T_out'].resample('D').mean(), color='tab:orange')
ax[2].set(ylim=[12, 17])
ax2b.set(ylim=[19, 24],
         ylabel='outside temperature')
# color the xticks
for tick in ax[2].get_yticklabels():
    tick.set_color('tab:orange')
# color the xlabel
ax[2].set_ylabel(r'"outside" temp.'+'\ninferred from\nanalysis', color='tab:orange')

Text(0, 0.5, '"outside" temp.\ninferred from\nanalysis')

```



Conclusions:

1. The cooling coefficient  $r$  seems quite stable throughout the two weeks of measurements. This probably says that the greenhouse and AC properties did not change much. For instance, the greenhouse thermal insulation stayed constant, and the AC power output stayed constant.
2. The AC tracks very well the outside temperature! This is to say: the AC works better (more easily) when temperatures outside are low, and vice-versa.

# 15 Savitzky–Golay

The Savitzky-Golay filter, also known as LOESS, smoothes a noisy signal by performing a polynomial fit over a sliding window.

Polynomial fit of order 3, window size = 51 pts

Polynomial fit of order 2, window size = 51 pts

The simulations look different because the order of the polynomial makes a very different impression on us, but in reality the outcome of the two filtering is almost identical:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker
from scipy.signal import savgol_filter
import os
import warnings
import scipy
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
# %matplotlib widget

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def centered_dates(ax):
```

```

date_form = DateFormatter("%d %b") # %d 3-letter-Month
# major ticks at midnight, every day
ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
ax.xaxis.set_major_formatter(date_form)
# minor ticks at noon, every day
ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
# erase major tick labels
ax.xaxis.set_major_formatter(ticker.NullFormatter())
# set minor tick labels as define above
ax.xaxis.set_minor_formatter(date_form)
# completely erase minor ticks, center tick labels
for tick in ax.xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

df = pd.read_csv('shani_2022_january.csv', parse_dates=['date'], index_col='date')
start = "2022-01-02"
end = "2022-01-05"
df = df.loc[start:end]

df['sg_3_51'] = savgol_filter(df['TD'], window_length=51, polyorder=3)
df['sg_2_51'] = savgol_filter(df['TD'], window_length=51, polyorder=2)

fig, ax = plt.subplots(figsize=(8,5))

plot_data, = ax.plot(df['TD'], color='black')
plot_sg2, = ax.plot(df['sg_2_51'], color='xkcd:hot pink')
plot_sg3, = ax.plot(df['sg_3_51'], color='xkcd:mustard')

ax.legend(handles=[plot_data, plot_sg2, plot_sg3],
           labels=['data', 'sg order 2', 'sg order 3'],
           frameon=False)

plot_settings = {
    'ylim': [5, 17.5],
    'xlim': [df.index[0], df.index[-1]],
    'ylabel': "Temperature (°C)",
    'title': "Yatir Forest, 2022",
    'yticks': [5, 10, 15]
}

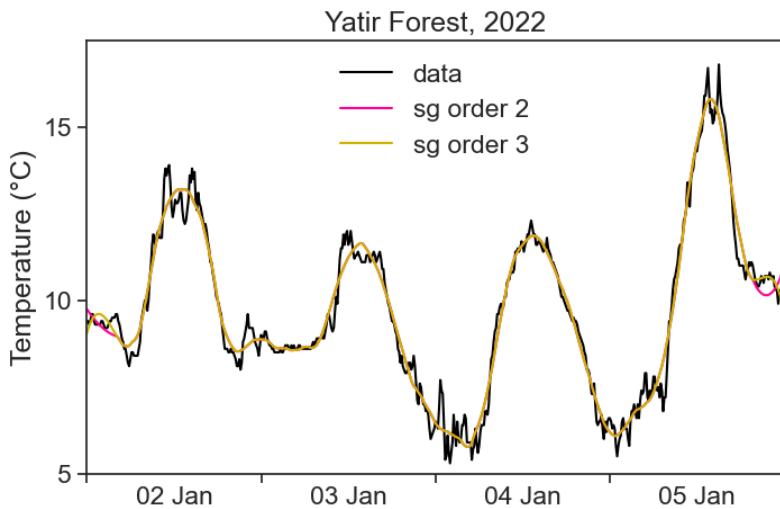
```

```

}

ax.set(**plot_settings)
centered_dates(ax)

```



To really see the difference between window width and polynomial order, we need to play with their ratio,

$$\text{ratio} = \frac{w}{p} = \frac{\text{window width}}{\text{polynomial order}}$$

```

start = "2022-01-02 00:00:00"
end = "2022-01-02 23:50:00"
df = df.loc[start:end]

```

```

# window_length, polyorder
df['sg_1'] = savgol_filter(df['TD'], 5, 3)
df['sg_2'] = savgol_filter(df['TD'], 11, 2)
df['sg_3'] = savgol_filter(df['TD'], 25, 3)

```

```

fig, ax = plt.subplots(figsize=(8,5))

plot_data, = ax.plot(df['TD'], color='black')
plot_sg1, = ax.plot(df['sg_1'], color='xkcd:hot pink')

```

```

plot_sg2, = ax.plot(df['sg_2'], color='xkcd:mustard')
plot_sg3, = ax.plot(df['sg_3'], color='xkcd:royal blue')

ax.legend(handles=[plot_data, plot_sg1, plot_sg2, plot_sg3],
           labels=['data', r'$w/p=1.5$', r'$w/p=5.5$', r'$w/p=8.3$'],
           frameon=False)

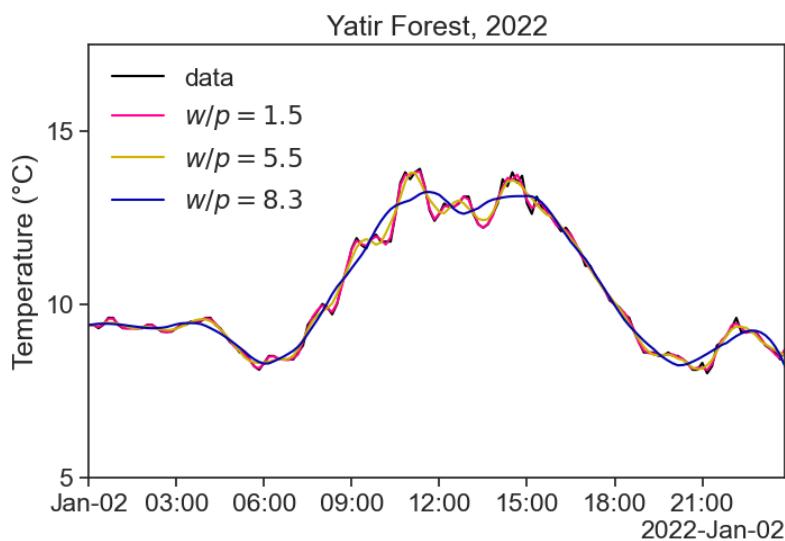
plot_settings = {
    'ylim': [5, 17.5],
    'xlim': [df.index[0], df.index[-1]],
    'ylabel': "Temperature (°C)",
    'title': "Yatir Forest, 2022",
    'yticks': [5, 10, 15]
}

ax.set(**plot_settings)

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)

ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

```



The higher the ratio, the more aggressive the smoothing.

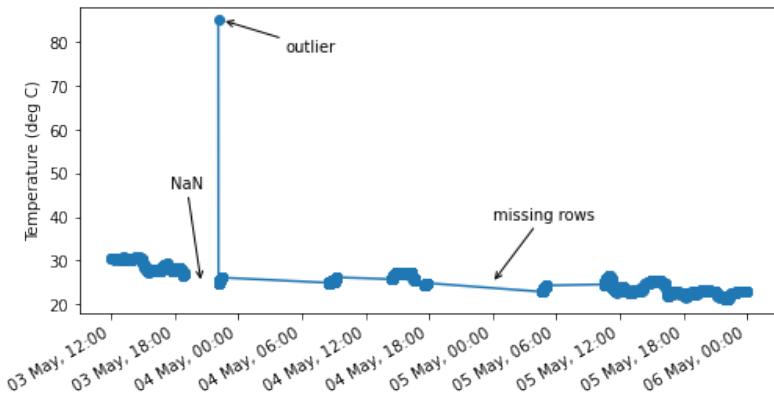
There is **a lot** more about the Savitzky-Golay filter, but for our purposes this is enough. If you want some more discussion about how to choose the parameters of the filter, [read this](#).

# **Part IV**

## **outliers and gaps**

# 16 motivation

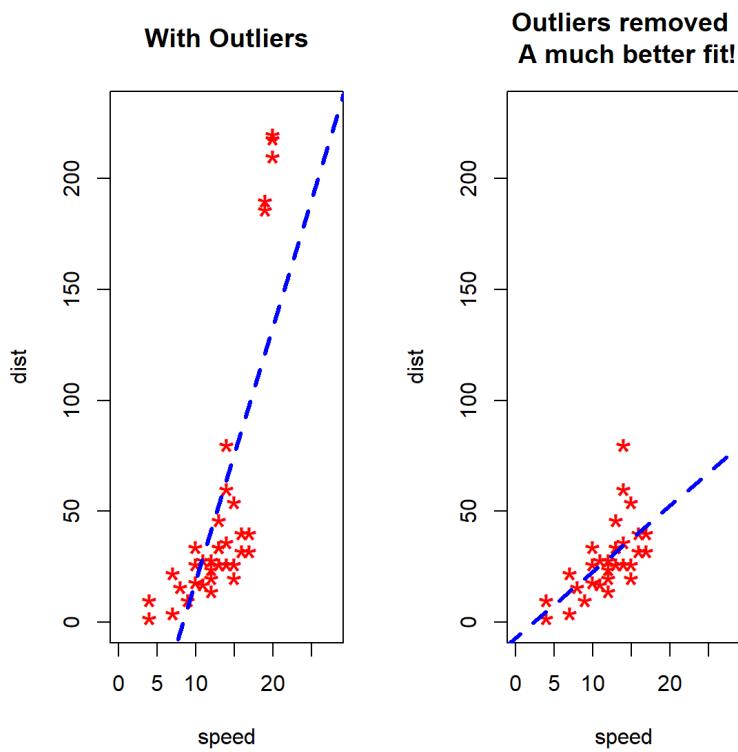
Outliers are observations significantly different from all other observations. Consider, for example, this temperature graph:



While most measured points are between 20 and 30 °C, there is obviously something very wrong with the one data point above 80 °C.

How could such a thing come about? This could be the result of **non-natural causes**, such as measurement errors, wrong data collection, or wrong data entry. On the other hand, this point could have **natural** sources, such as a very hot spark flying next to the temperature sensor.

Identifying outliers is important, because they might greatly impact measures like mean and standard deviation. When left untouched, outliers might make us reach wrong conclusions about our data. See what happens to the slope of this linear regression with and without the outliers.

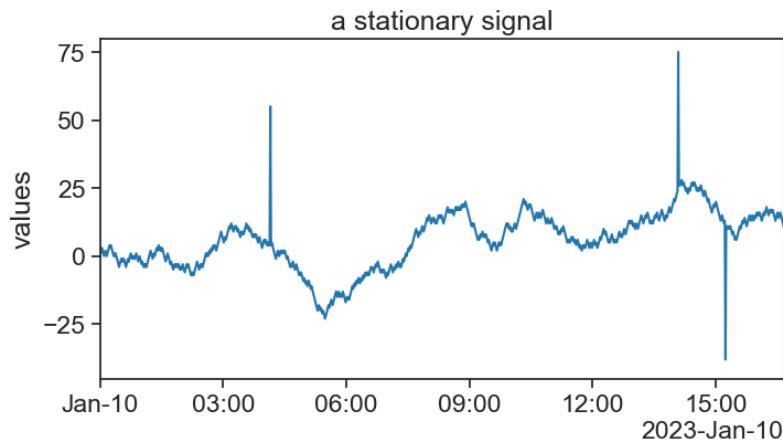


Source: Zhang (2020)

# 17 outlier identification

## 17.1 visual inspection

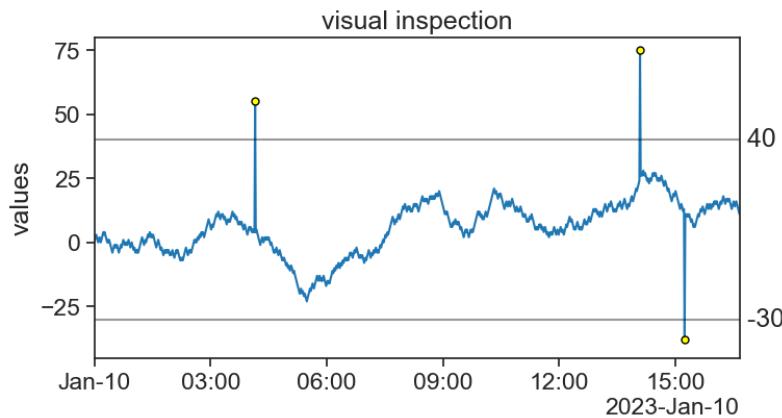
I produced a stationary signal and added to it a few outliers. Can you tell where just by looking at the graph?



The easiest way of identifying the outliers is:

- First plot the time series.
- Choose upper and lower boundaries. Whatever falls outside these boundaries is an outlier.

Easy.



If all you have is this one time series, you're done, congratulations. However, it is often the case that one has very long time series, or a great number of time series to analyze. In this case it is impractical to use the visual inspection method. We would like to devise an algorithm to automate this task.

## 17.2 Z-score

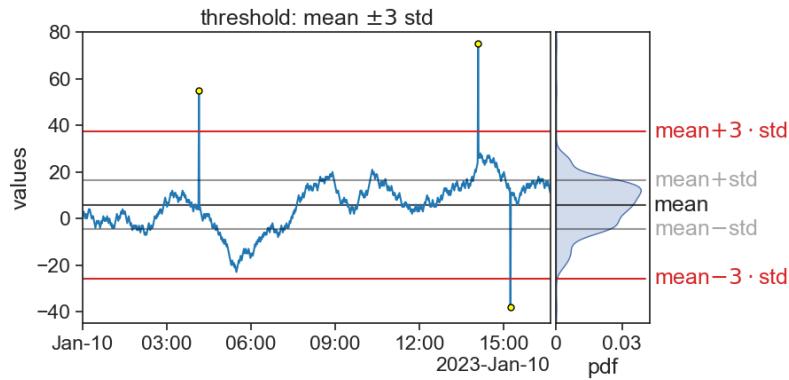
The Z-score is the distance, in units of 1 standard deviation, of a point in the series with respect to the mean:

$$z = \frac{x - \mu}{\sigma},$$

A common choice is to consider an outlier a point whose Z-score is greater than 3, in absolute value. In other words: If a point is more than 3 standard deviations away from the mean, then we call it an outlier.

where

- $x$  = data point,
- $\mu$  = time series mean
- $\sigma$  = time series standard deviation.

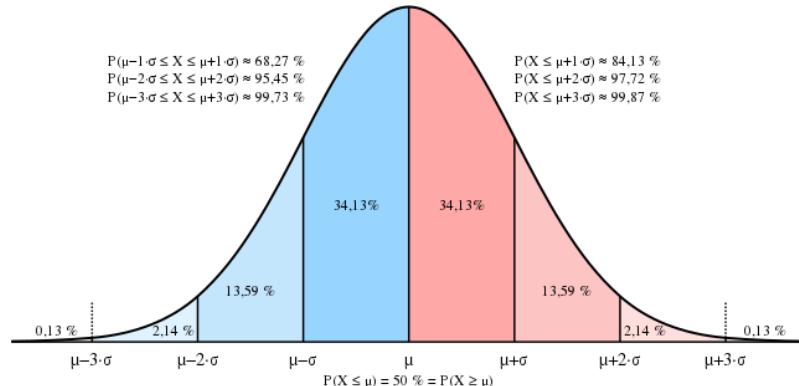


You can now use this algorithm to any number of time series, let the computer do the hard work.

Of course, there is nothing sacred about the number 3. You can choose any Z-score you want to perform an analysis on your own data, depending on your needs.

### 17.2.1 ATTENTION!

For data that is gaussianly distributed, we expect that 99.73% of data to fall within 3 standard deviations from the mean. In other words, 0.27% of points would be considered as *outliers* according to the Z-score method.

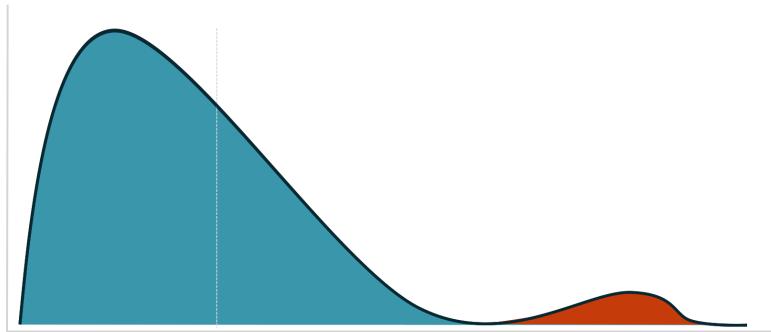


Assume you have a time series gaussianly distributed, with 10k measurements. We would expect to find about 27 outliers in this time series.

Source: [Wikimedia Commons](#)

So what is the problem?!

The thing is, outliers are not supposed to be only data points far from the other points. That's not enough. A better way of understanding outliers is to imagine that our expected measurements are sampled from a given distribution, and every now and then we have measurements that are sampled from **another** distribution.



We should have this in mind always. We wouldn't want to single out good data as something weird. Our true task is to identify which points in our time series were sampled from a different distribution. This can be a very challenging task.

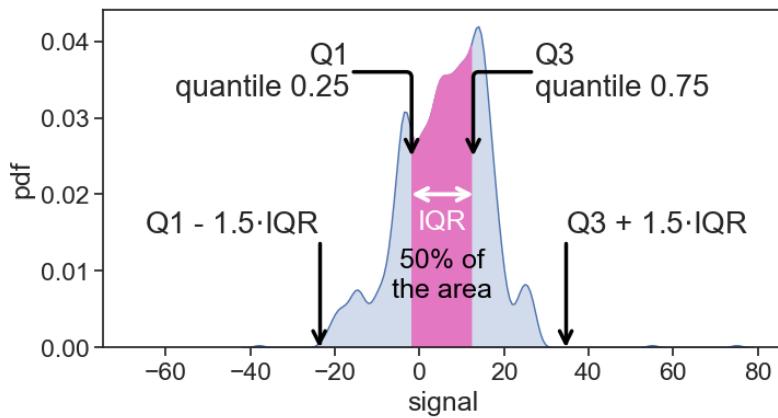
Source: Taylor Wilson's "Dealing with Outliers (Part 1): Ignore Them at Your Peril"

### 17.3 IQR

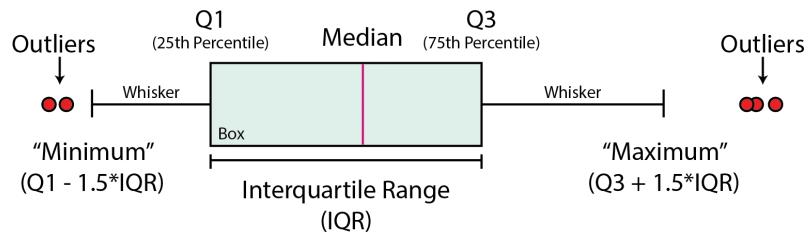
Another super common criterion for identifying outliers is the IQR, or InterQuartile Range.

Take a look at the statistics below of the time series we have been working with so far. The IQR is the distance between the first quartile (Q1) and the third quartile (Q3), where exactly 50% of the data is.

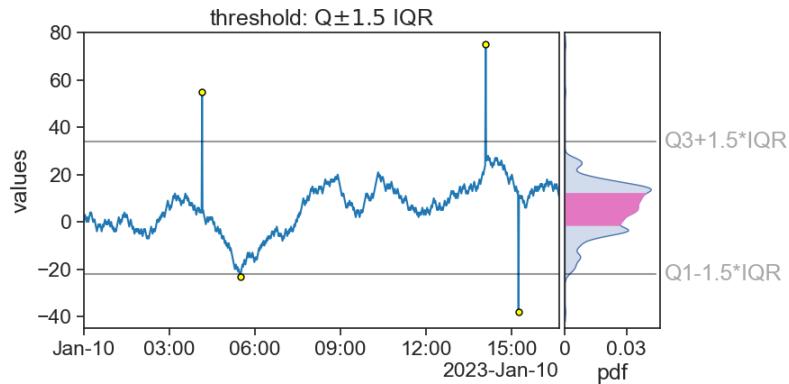
The algorithm here is to determine two thresholds, whose distance is 1.5 times the IQR from Q1 and Q3. Whatever falls outside these two thresholds is an outlier.



We are used to see this in box plots:



Again, the distance 1.5 is not sacred, it's only the most common.  
You might want to choose other values depending on your needs.  
Let's now apply the IQR method to our time series.

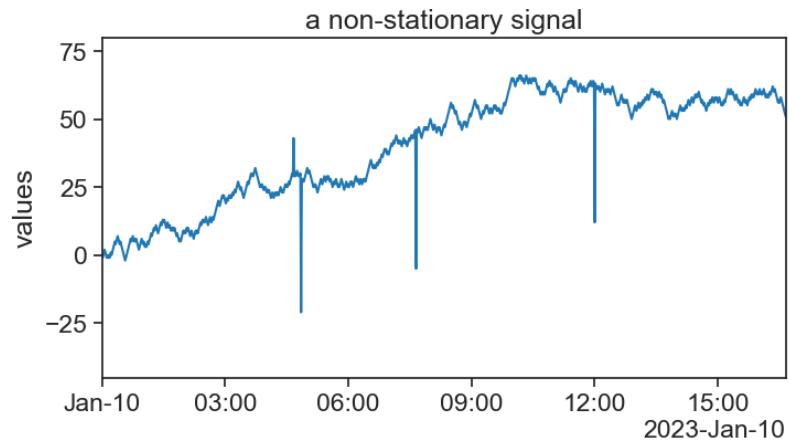


It works pretty well! Notice that now we have an additional outlier (a bit before 06:00). What do we do with that?

Source: McDonald (2022)

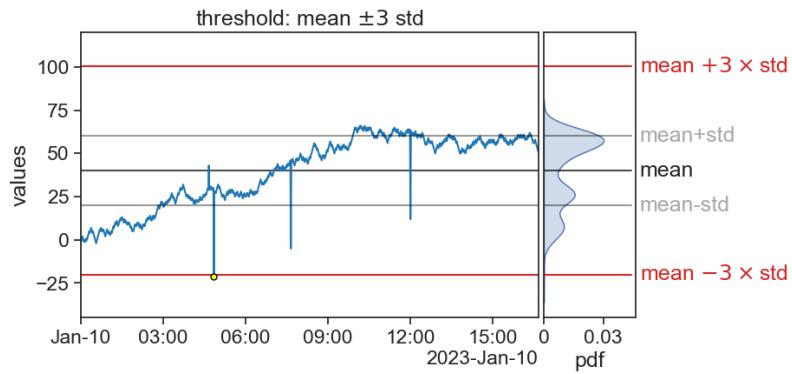
## 17.4 non-stationary time series

I have produced a new time series, one that on average goes up with time. Can you point in the graph where are the outliers?

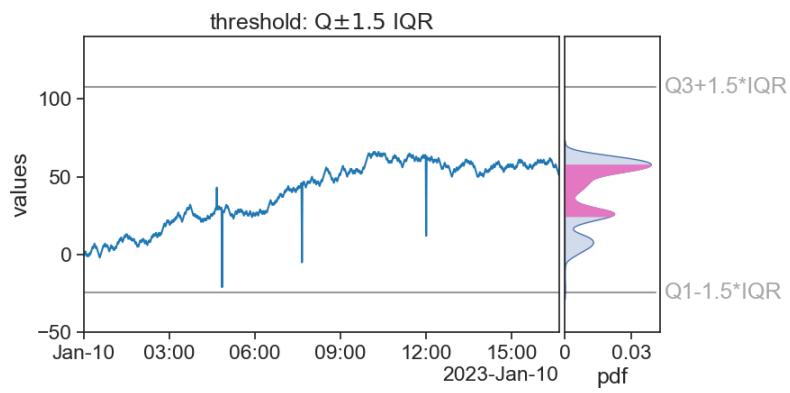


Now, see what happens when we apply the previous two methods to this time series.

### Z-score



### IQR



What happened? Do you have ideas how to solve this?

## 17.5 Sources

# 18 robust analysis

A tool is said to be **robust** if outliers don't influence (much) its results.

The average and standard deviation are **not** robust.

```
import numpy as np
series1 = np.array([0, 1, 2, 3, 4, 5, 6])
series2 = np.array([0, 1, 2, 3, 4, 5, 60])
print(f"series 1: mean={series1.mean():.2f}, std={series1.std():.2f}")
print(f"series 2: mean={series2.mean():.2f}, std={series2.std():.2f}")

series 1: mean=3.00, std=2.00
series 2: mean=10.71, std=20.18
```

On the other hand, the median and IQR are robust:

```
from scipy.stats import iqr
print(f"series 1: median={np.median(series1):.2f}, IQR={iqr(series1):.2f}")
print(f"series 2: median={np.median(series2):.2f}, IQR={iqr(series2):.2f}")

series 1: median=3.00, IQR=3.00
series 2: median=3.00, IQR=3.00
```

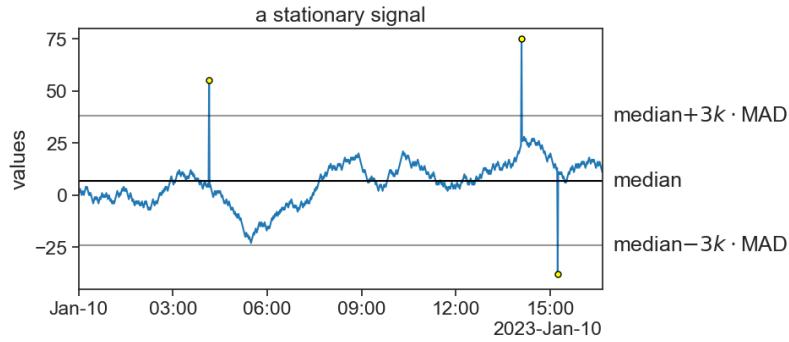
## 18.1 MAD

Another robust method is MAD, the Median Absolute Deviation, given by

$$\text{MAD} = \text{median}(|x_i - \text{median}(x)|),$$

where  $|\cdot|$  is the absolute value.

Applying MAD to the stationary time series from before, yields

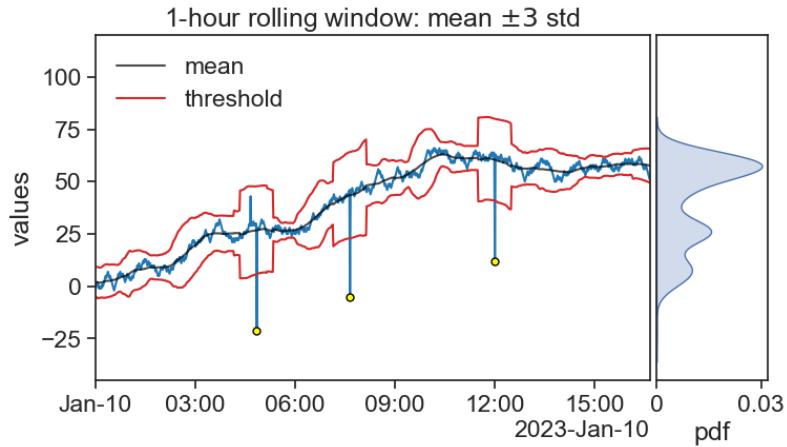


Here, the threshold is the median  $\pm 3k \cdot \text{MAD}$ , where the value  $k = 1.4826$  scales MAD so that when the data is gaussianly distributed,  $3k$  equals 1 standard deviation.

# 19 sliding algorithms

None of the methods learned before seem appropriate to deal with non-stationary data. A simple solution is to apply those methods for sliding windows of “appropriate” widths.

## 19.1 Sliding Z-score

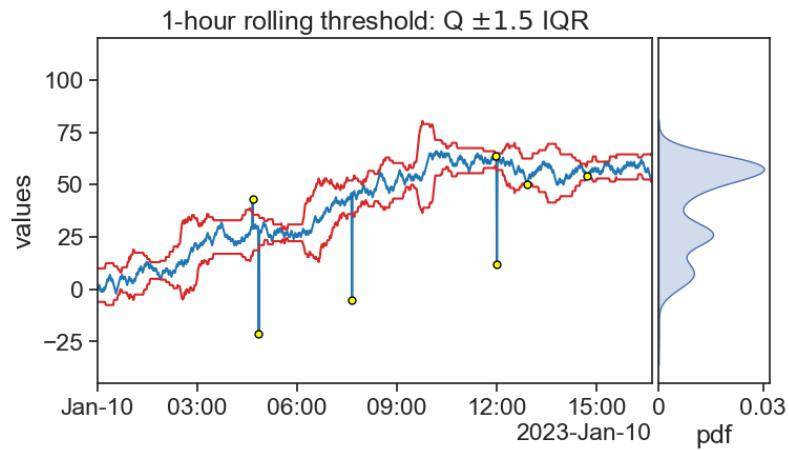


Now the Z-score seems to give really nice results (but not perfect). Maybe playing with the window width and Z-score threshold would give better results?

In any case, we clearly see why the Z-score is not a robust algorithm. See how the standard deviation is sensitive to outliers?

## 19.2 Sliding IQR

Let's see how well the sliding IQR method fares.



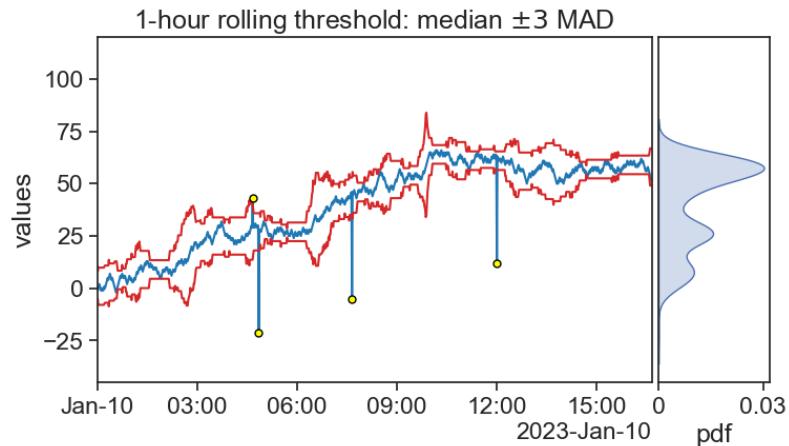
It identified all the outliers, but also found that a few *other* points should be considered outliers? What do you think of that?

See that the threshold does not jump abruptly when the sliding window includes an outlier. In fact, the threshold doesn't even care! This is what it means to be robust.

However, we do see large fluctuations in the threshold. When does this happen? Why?

### 19.3 Sliding MAD

Now it's MAD's time to shine.



Compare this result to the previous two. Which yields best results?

MAD is robust to outliers, and again we see that the threshold envelope widens when there is a rising or falling trend in the data.

## 19.4 Challenges

Now it's your turn to work, I'm tired! Write algorithms for the following outlier identification methods:

1. visual inspection
2. Z-score
3. IQR
4. MAD

Excluding the visual inspection method, write first an algorithm that operates on a full time series, and then write a new version that can work with sliding windows.

# 20 substituting outliers

Ok. We found the outliers. Now what?!

As usual, it depends.

## 20.1 Do nothing

Assuming the outlier indeed happened in real life, and is not the result of faulty data transmission or bad data recording, then excluding an outlier might be the last thing you want to do. Sometimes extreme events do happen, such as a one-in-a-hundred-year storm, and they have a disproportionate weight on the system you are studying. The outliers might actually be the most interesting points in your data for all you know!

In case the outliers are not of interest to you, if you are using **robust** methods to analyze your data, you don't necessarily need to do anything either. For instance, let's say that you want to smooth your time series. If instead of taking the **mean** inside a sliding window you choose to calculate the **median**, then outliers shouldn't be a problem. Test it and see if it's true. Go on.

For many things you need to do (not only smoothing), you might be able to find robust methods. What do you do if you **have** to use a non-robust method? Well, then you can substitute the outlier for two things: NaN or imputed values.

## 20.2 NaN

Substitute outliers for NaN.

NaN means “Not a Number”, and is what you get when you try to perform a mathematical operation like  $0/0$ . It is common to see NaN in dataset rows when data was not collected for some reason.

This might seem like a neutral solution, but it actually can generate problems down the line. See this example:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
from scipy.signal import savgol_filter

# example using numpy
series = np.array([2, 4, 5, np.nan, 8, 15])
mean = np.mean(series)
print(f"the series average is {mean}")
```

the series average is nan

A single NaN in your time series ruins the whole calculation! There is a workaround though:

```
mean = np.nanmean(series)
print(f"the series average is {mean}")
```

the series average is 6.8

You have to make sure what is the behavior of each function you use with respect to NaNs, and if possible, use a suitable substitute.

The same example in `pandas` would not fail:

```

date_range = pd.date_range(start='2024-01-01', periods=len(series), freq='1D')
df = pd.DataFrame({'series': series}, index=date_range)
mean = df['series'].mean()
print(f"the series average is {mean}")

```

the series average is 6.8

## 20.3 impute values

To “impute values” means to fill in the missing value with a guess, an estimation of what this data point “should have been” if it were measured in the first place. Why should we bother to do so? Because many tools that we know and love don’t do well with missing values.

We learned about the Savitzky-Golay filter for smoothing data. See what happens when there is a single NaN in the series:

```

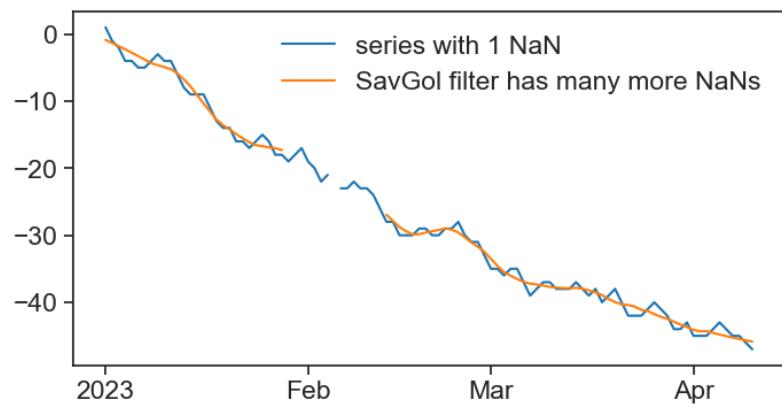
steps = np.random.randint(low=-2, high=2, size=100)
data = steps.cumsum()
date_range = pd.date_range(start='2023-01-01', periods=len(data), freq='1D')
df = pd.DataFrame({'series': data}, index=date_range)
df.loc['2023-02-05', 'series'] = np.nan

df['sg'] = savgol_filter(df['series'], window_length=15, polyorder=2)

def concise(ax):
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

fig, ax = plt.subplots(figsize=(8,4))
ax.plot(df['series'], color="tab:blue", label="series with 1 NaN")
ax.plot(df['sg'], color="tab:orange", label="SavGol filter has many more NaNs")
concise(ax)
ax.legend(frameon=False);

```



We will deal with this topic in the next chapter, “interpolation”. There, we will learn a few methods to fill in missing data, and basic NaN operations you should be acquainted with.

# 21 challenge

## 21.1 importing bad .csv files

Here we will get a taste of what it feels like to work with bad .csv files and how to fix them.

TO DO:

1. create a folder for this challenge, name it whatever you want.
2. download this jupyter notebook and move to that folder.
3. download these 3 .csv files and part 2 notebook:
  1. cleaning1
  2. cleaning2
  3. cleaning3
  4. part\_2
4. move them files into your folder

### 21.1.1 import

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
```

```
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters  
# %matplotlib widget
```

## 21.2 dataset 1

```
df1 = pd.read_csv('cleaning1.csv')  
df1
```

	date	A	B	C	D	E
0	2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000
1	2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300
2	2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504
3	2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084
4	2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579
...	...	...	...	...	...	...
105115	2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984
105116	2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110
105117	2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810
105118	2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432
105119	2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007

Now let's put the column 'date' in the index with datetime format

```
# we can change the format of the column to datetime and then set it as the index.  
df1['date'] = pd.to_datetime(df1['date'])  
df1.set_index('date', inplace=True)  
df1
```

	A	B	C	D	E
date					
2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000
2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300

	A	B	C	D	E
date					
2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504
2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084
2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579
...	...	...	...	...	...
2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984
2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110
2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810
2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432
2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007

If we know that in advance, we can write everything in one command when we import the csv.

```
df1 = pd.read_csv('cleaning1.csv',
                  index_col='date',      # set the column date as index
                  parse_dates=True)     # turn to datetime format
df1
```

	A	B	C	D	E
date					
2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000
2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300
2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504
2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084
2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579
...	...	...	...	...	...
2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984
2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110
2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810
2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432
2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007

Now let's plot all the columns and see what we have.

```
def plot_all_columns(data):
    column_list = data.columns

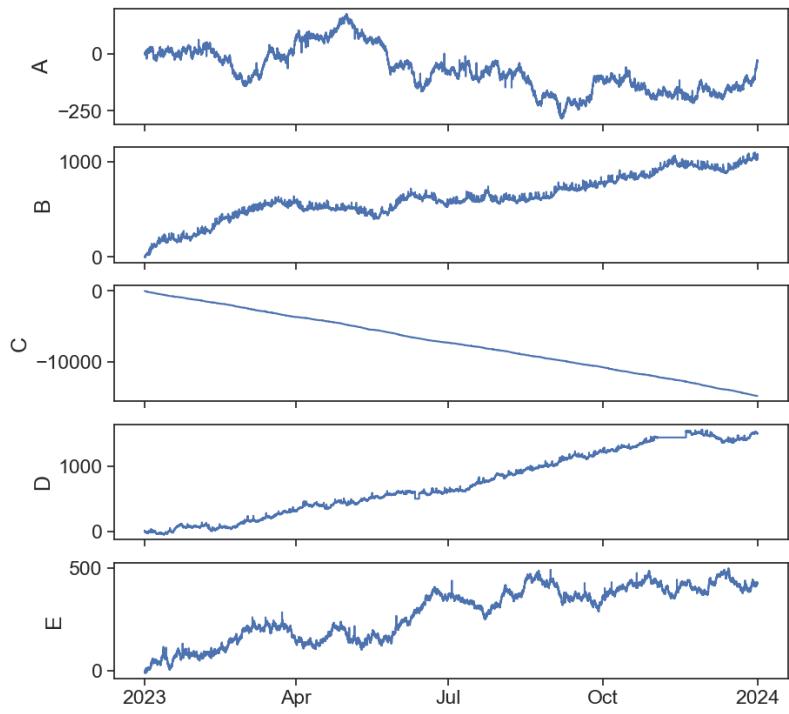
    fig, ax = plt.subplots(len(column_list),1, sharex=True, figsize=(10,len(column_list)*2))

    if len(column_list) == 1:
        ax.plot(data[column_list[0]])
        return
    for i, column in enumerate(column_list):
        ax[i].plot(data[column])
        ax[i].set(ylabel=column)

    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax[i].xaxis.set_major_locator(locator)
    ax[i].xaxis.set_major_formatter(formatter)

    return
```

```
plot_all_columns(df1)
```



Looks like some of this data needs cleaning...

### 21.3 dataset 2

```
df2 = pd.read_csv('cleaning2-.csv')
df2
```

	A	B	date	time
0	0.0	0.0	01012023	00:00:00
1	-2.0275363548598184	0.011984922825112581	01012...	
2	-2.690616715983192	-0.29792822981957684	010120...	
3	-1.9859899758267612	-0.30940867922490206	01012...	
4	-2.290897621584889	-2.8666633353521624	0101202...	
...	...			
8755	-74.51464473079395	293.8680858227996	31122023	...
8756	-74.73805809332175	294.7593463919649	31122023	...
8757	-75.84842465358788	294.07634907736116	31122023...	

---

	A	B	date	time
8758	-77.27218272637339	293.526461290973	31122023	2...
8759	-76.55739976945038	293.35336890454107	31122023...	

---

Something is wrong...

Let's open the actual .csv file and take a quick look.

It seems that the values are seperated by spaces and not by commas ,.

```
df2 = pd.read_csv('cleaning2-.csv', delimiter=' ')
df2
```

---

	A	B	date	time
0	0.000000	0.0	1012023	00:00:00
1	-2.027536	0.011984922825112581	1012023	01:00:00
2	-2.690617	-0.29792822981957684	1012023	02:00:00
3	-1.985990	-0.30940867922490206	1012023	03:00:00
4	-2.290898	-2.8666633353521624	1012023	04:00:00
...	...	...	...	...
8755	-74.514645	293.8680858227996	31122023	19:00:00
8756	-74.738058	294.7593463919649	31122023	20:00:00
8757	-75.848425	294.07634907736116	31122023	21:00:00
8758	-77.272183	293.526461290973	31122023	22:00:00
8759	-76.557400	293.35336890454107	31122023	23:00:00

---

```
# convert the date column to datetime
df2['date_corrected'] = pd.to_datetime(df2['date'])
# df2['date_corrected'] = pd.to_datetime(df2['date'], format='%d%m%Y')

df2
```

---

	A	B	date	time	date_corrected
0	0.000000	0.0	1012023	00:00:00	1970-01-01 00:00:00.001012023
1	-2.027536	0.011984922825112581	1012023	01:00:00	1970-01-01 00:00:00.001012023
2	-2.690617	-0.29792822981957684	1012023	02:00:00	1970-01-01 00:00:00.001012023
3	-1.985990	-0.30940867922490206	1012023	03:00:00	1970-01-01 00:00:00.001012023

---

	A	B	date	time	date_corrected
4	-2.290898	-2.8666633353521624	1012023	04:00:00	1970-01-01 00:00:00.001012023
...	...	...	...	...	...
8755	-74.514645	293.8680858227996	31122023	19:00:00	1970-01-01 00:00:00.031122023
8756	-74.738058	294.7593463919649	31122023	20:00:00	1970-01-01 00:00:00.031122023
8757	-75.848425	294.07634907736116	31122023	21:00:00	1970-01-01 00:00:00.031122023
8758	-77.272183	293.526461290973	31122023	22:00:00	1970-01-01 00:00:00.031122023
8759	-76.557400	293.35336890454107	31122023	23:00:00	1970-01-01 00:00:00.031122023

```
# df2['date_corrected'] = pd.to_datetime(df2['date'][:780], format='%d%m%Y')
# df2['date_corrected'] = pd.to_datetime(df2['date'][780:800], format='%d%m%Y')
# df2['date'][780:800]
```

```
data_types = {'date': str , 'time':str}

# Read the CSV file with specified data types
df2 = pd.read_csv('cleaning2-.csv', delimiter=' ', dtype=data_types)
df2
```

	A	B	date	time
0	0.000000	0.0	01012023	00:00:00
1	-2.027536	0.011984922825112581	01012023	01:00:00
2	-2.690617	-0.29792822981957684	01012023	02:00:00
3	-1.985990	-0.30940867922490206	01012023	03:00:00
4	-2.290898	-2.8666633353521624	01012023	04:00:00
...	...	...	...	...
8755	-74.514645	293.8680858227996	31122023	19:00:00
8756	-74.738058	294.7593463919649	31122023	20:00:00
8757	-75.848425	294.07634907736116	31122023	21:00:00
8758	-77.272183	293.526461290973	31122023	22:00:00
8759	-76.557400	293.35336890454107	31122023	23:00:00

```
df2['date_corrected'] = pd.to_datetime(df2['date'], format='%d%m%Y')
df2
```

	A	B	date	time	date_corrected
0	0.000000	0.0	01012023	00:00:00	2023-01-01
1	-2.027536	0.011984922825112581	01012023	01:00:00	2023-01-01
2	-2.690617	-0.29792822981957684	01012023	02:00:00	2023-01-01
3	-1.985990	-0.30940867922490206	01012023	03:00:00	2023-01-01
4	-2.290898	-2.8666633353521624	01012023	04:00:00	2023-01-01
...	...	...	...	...	...
8755	-74.514645	293.8680858227996	31122023	19:00:00	2023-12-31
8756	-74.738058	294.7593463919649	31122023	20:00:00	2023-12-31
8757	-75.848425	294.07634907736116	31122023	21:00:00	2023-12-31
8758	-77.272183	293.526461290973	31122023	22:00:00	2023-12-31
8759	-76.557400	293.35336890454107	31122023	23:00:00	2023-12-31

```
df2['datetime'] = pd.to_datetime(df2['date'] + ' ' + df2['time'], format='%d%m%Y %H:%M:%S')
df2
```

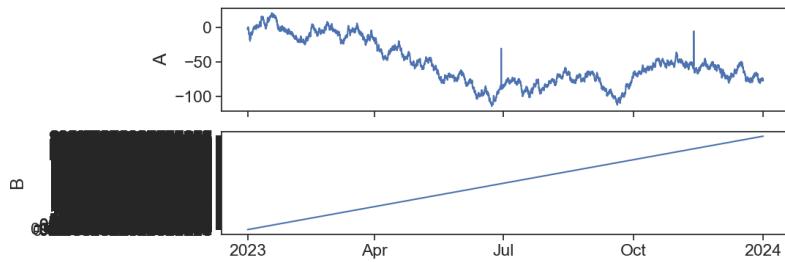
	A	B	date	time	date_corrected	datetime
0	0.000000	0.0	01012023	00:00:00	2023-01-01	2023-01-01 00:00:00
1	-2.027536	0.011984922825112581	01012023	01:00:00	2023-01-01	2023-01-01 01:00:00
2	-2.690617	-0.29792822981957684	01012023	02:00:00	2023-01-01	2023-01-01 02:00:00
3	-1.985990	-0.30940867922490206	01012023	03:00:00	2023-01-01	2023-01-01 03:00:00
4	-2.290898	-2.8666633353521624	01012023	04:00:00	2023-01-01	2023-01-01 04:00:00
...	...	...	...	...	...	...
8755	-74.514645	293.8680858227996	31122023	19:00:00	2023-12-31	2023-12-31 19:00:00
8756	-74.738058	294.7593463919649	31122023	20:00:00	2023-12-31	2023-12-31 20:00:00
8757	-75.848425	294.07634907736116	31122023	21:00:00	2023-12-31	2023-12-31 21:00:00
8758	-77.272183	293.526461290973	31122023	22:00:00	2023-12-31	2023-12-31 22:00:00
8759	-76.557400	293.35336890454107	31122023	23:00:00	2023-12-31	2023-12-31 23:00:00

```
df2.drop(columns=['date', 'time', 'date_corrected'], inplace=True)
df2.set_index('datetime', inplace=True)
df2
```

	A	B
datetime		
2023-01-01 00:00:00	0.000000	0.0

	A	B
datetime		
2023-01-01 01:00:00	-2.027536	0.011984922825112581
2023-01-01 02:00:00	-2.690617	-0.29792822981957684
2023-01-01 03:00:00	-1.985990	-0.30940867922490206
2023-01-01 04:00:00	-2.290898	-2.8666633353521624
...	...	...
2023-12-31 19:00:00	-74.514645	293.8680858227996
2023-12-31 20:00:00	-74.738058	294.7593463919649
2023-12-31 21:00:00	-75.848425	294.07634907736116
2023-12-31 22:00:00	-77.272183	293.526461290973
2023-12-31 23:00:00	-76.557400	293.35336890454107

```
plot_all_columns(df2)
```



What happened in the second ax?

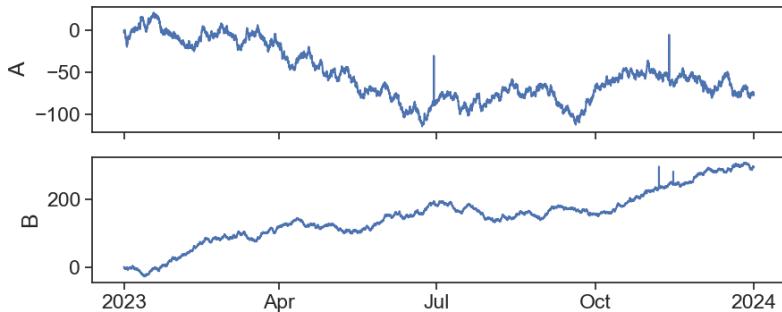
```
df2.dtypes
```

```
A    float64
B    object
dtype: object
```

```
# use pd.to_numeric to convert column 'B' to float
df2['B'] = pd.to_numeric(df2['B'],
                        errors='coerce' # 'coerce' will convert non-numeric values to NaN
)
df2.dtypes
```

```
A    float64  
B    float64  
dtype: object
```

```
plot_all_columns(df2)
```



```
data_types = {'date': str , 'time':str}  
  
# Read the CSV file with specified data types  
df2 = pd.read_csv('cleaning2-.csv', delimiter=' ', dtype=data_types, na_values='-')  
df2['datetime'] = pd.to_datetime(df2['date'] + ' ' + df2['time'], format='%d%m%Y %H:%M:%S')  
df2.drop(columns=['date', 'time'], inplace=True)  
df2.set_index('datetime', inplace=True)  
df2
```

datetime	A	B
2023-01-01 00:00:00	0.000000	0.000000
2023-01-01 01:00:00	-2.027536	0.011985
2023-01-01 02:00:00	-2.690617	-0.297928
2023-01-01 03:00:00	-1.9855990	-0.309409
2023-01-01 04:00:00	-2.290898	-2.8666663
...	...	...
2023-12-31 19:00:00	-74.514645	293.868086
2023-12-31 20:00:00	-74.738058	294.759346
2023-12-31 21:00:00	-75.848425	294.076349
2023-12-31 22:00:00	-77.272183	293.526461
2023-12-31 23:00:00	-76.557400	293.353369

```
df2.to_csv('cleaning2_formated.csv')
```

## 21.4 dataset 3

```
df3 = pd.read_csv('cleaning3.csv')
df3
```

	#
0	# data created by
1	# Yair Mau
2	# for time series data analysis
3	#
4	# time format: unix (s)
...	...
370	6.651300774019661 1703635200.0
371	6.4151748349408715 1703721600.0
372	7.603140054178304 1703808000.0
373	8.668182044560869 1703894400.0
374	8.472767724946076 1703980800.0

Again, let's look at the actual .csv file.

```
df3 = pd.read_csv('cleaning3.csv', comment='#')
df3
```

	A time
0	0.0 1672531200.0
1	-0.03202661701444382 1672617600.0
2	-0.5863508675173621 1672704000.0
3	-1.5759721567247762 1672790400.0
4	-2.7267995149281266 1672876800.0
...	...
360	6.651300774019661 1703635200.0
361	6.4151748349408715 1703721600.0

	A	time
362	7.603140054178304	1703808000.0
363	8.668182044560869	1703894400.0
364	8.472767724946076	1703980800.0

```
df3 = pd.read_csv('cleaning3.csv', comment='#', delimiter=' ')  
df3
```

	A	time
0	0.000000	1.672531e+09
1	-0.032027	1.672618e+09
2	-0.586351	1.672704e+09
3	-1.575972	1.672790e+09
4	-2.726800	1.672877e+09
...	...	...
360	6.651301	1.703635e+09
361	6.415175	1.703722e+09
362	7.603140	1.703808e+09
363	8.668182	1.703894e+09
364	8.472768	1.703981e+09

```
df3.dtypes
```

```
A      float64  
time    float64  
dtype: object
```

Time is in [unix](#).  
[unix converter](#)

```
print(df3['time'][0])
```

1672531200.0

```
df3['time'] = pd.to_datetime(df3['time'], unit='s')
df3.set_index('time', inplace=True)
df3
```

A	
time	
2023-01-01	0.000000
2023-01-02	-0.032027
2023-01-03	-0.586351
2023-01-04	-1.575972
2023-01-05	-2.726800
...	...
2023-12-27	6.651301
2023-12-28	6.415175
2023-12-29	7.603140
2023-12-30	8.668182
2023-12-31	8.472768

```
df3 = pd.read_csv('cleaning3.csv',
                   index_col='time',      # set the column date as index
                   parse_dates=True,      # turn to datetime format
                   comment='#',
                   delimiter=' ')
df3.index = pd.to_datetime(df3.index, unit='s')
df3
```

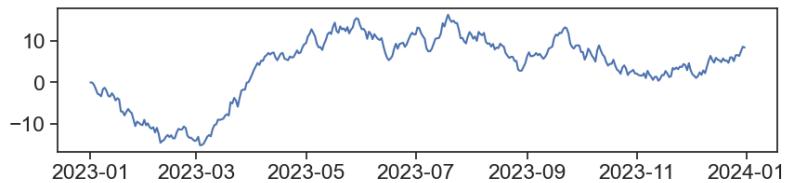
A	
time	
2023-01-01	0.000000
2023-01-02	-0.032027
2023-01-03	-0.586351
2023-01-04	-1.575972
2023-01-05	-2.726800
...	...
2023-12-27	6.651301
2023-12-28	6.415175
2023-12-29	7.603140

---

A	
time	
2023-12-30	8.668182
2023-12-31	8.472768

---

```
plot_all_columns(df3)
```



```
df3.to_csv('cleaning3_formated.csv')
```

## 22 challenge part 2

### 22.1 outliers and missing values

Here you have 3 dataframes that need cleaning. Use the methods learned in class to process the data.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters

# %matplotlib widget

df1 = pd.read_csv('cleaning1.csv',
                   index_col='date',      # set the column date as index
                   parse_dates=True)     # turn to datetime format
df1
```

	A	B	C	D	E
date					
2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000
2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300
2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504
2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084

	A	B	C	D	E
date					
2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579
...	...	...	...	...	...
2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984
2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110
2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810
2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432
2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007

```
df2 = pd.read_csv('cleaning2_formated.csv',
                  index_col='datetime',      # set the column date as index
                  parse_dates=True)        # turn to datetime format
df2
```

	A	B
datetime		
2023-01-01 00:00:00	0.000000	0.000000
2023-01-01 01:00:00	-2.027536	0.011985
2023-01-01 02:00:00	-2.690617	-0.297928
2023-01-01 03:00:00	-1.985990	-0.309409
2023-01-01 04:00:00	-2.290898	-2.866663
...	...	...
2023-12-31 19:00:00	-74.514645	293.868086
2023-12-31 20:00:00	-74.738058	294.759346
2023-12-31 21:00:00	-75.848425	294.076349
2023-12-31 22:00:00	-77.272183	293.526461
2023-12-31 23:00:00	-76.557400	293.353369

```
df3 = pd.read_csv('cleaning3_formated.csv',
                  index_col='time',      # set the column date as index
                  parse_dates=True,      # turn to datetime format
                  )
df3
```

A	
time	
2023-01-01	0.000000
2023-01-02	-0.032027
2023-01-03	-0.586351
2023-01-04	-1.575972
2023-01-05	-2.726800
...	...
2023-12-27	6.651301
2023-12-28	6.415175
2023-12-29	7.603140
2023-12-30	8.668182
2023-12-31	8.472768

## 22.2 cleaning df1 from outliers

### 22.2.1 method 1: rolling standard deviation envelope

Visual inspection of all the data:

```
def plot_all_columns(data):
    column_list = data.columns

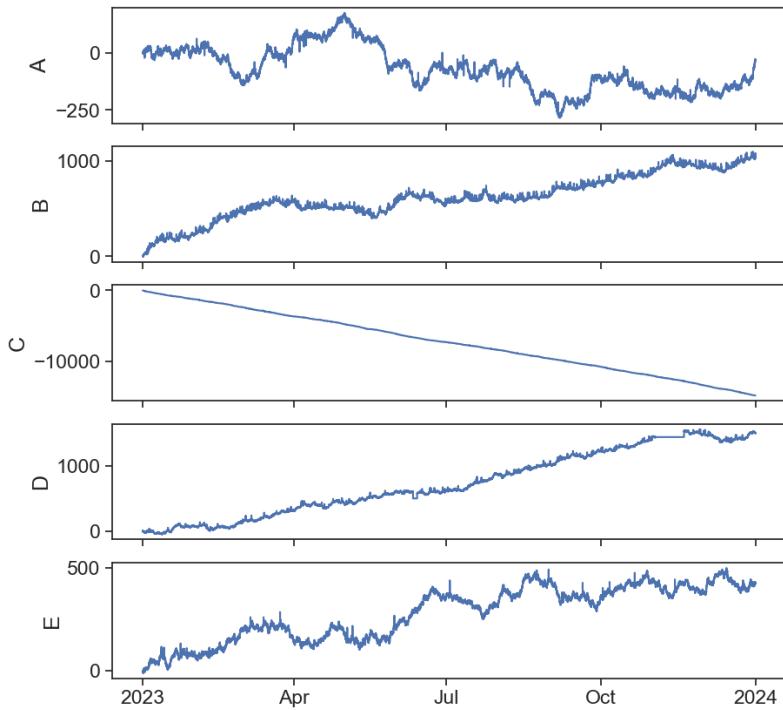
    fig, ax = plt.subplots(len(column_list),1, sharex=True, figsize=(10,len(column_list)*2))

    if len(column_list) == 1:
        ax.plot(data[column_list[0]])
        return
    for i, column in enumerate(column_list):
        ax[i].plot(data[column])
        ax[i].set(ylabel=column)

    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax[i].xaxis.set_major_locator(locator)
    ax[i].xaxis.set_major_formatter(formatter)

    return
```

```
plot_all_columns(df1)
```



Applying the rolling std method on column A:

```
# find the rolling std
df1['A_std'] = df1['A'].rolling(50, center=True, min_periods=1).std()
# find the rolling mean
df1['A_mean'] = df1['A'].rolling(50, center=True, min_periods=1).mean()
# define the k parameter -> the number of standard deviations from the mean which above them we want to filter
k = 2
# finding the top and bottom threshold for each datapoint
df1['A_top'] = df1['A_mean'] + k*df1['A_std']
df1['A_bot'] = df1['A_mean'] - k*df1['A_std']
# creating a mask of booleans that places true if the row is an outlier and false if its not.
df1['A_out'] = ((df1['A'] > df1['A_top']) | (df1['A'] < df1['A_bot']))
# applying the mask and replacing all outliers with nans.
df1['A_filtered'] = np.where(df1['A_out'],
                             np.nan, # use this if A_out is True
                             df1['A']) # otherwise
```

```
df1
```

date	A	B	C	D	E	A_std	A_mean	A
2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000	1.262273	1.520000	4.0
2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300	1.331858	1.423077	4.0
2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504	1.462738	1.296296	4.2
2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084	1.649114	1.142857	4.4
2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579	1.880022	0.965517	4.7
...	...	...	...	...	...	...	...	...
2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984	2.988291	-32.633333	-20
2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110	3.038748	-32.655172	-20
2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810	3.077483	-32.714286	-20
2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432	3.088901	-32.814815	-20
2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007	3.128283	-32.884615	-20

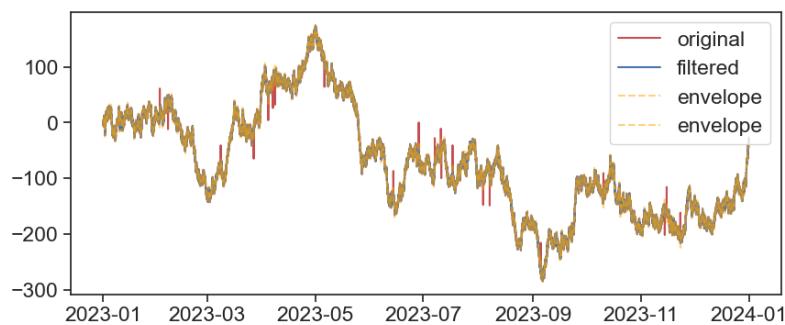
## 22.2.2 plotting the results:

Use %matplotlib widget to visually inspect the results.

```
fig, ax = plt.subplots(figsize=(10,4))

ax.plot(df1['A'], c='r', label='original')
ax.plot(df1['A_filtered'], c='b', label='filtered')
ax.plot(df1['A_bot'], c='orange', linestyle='--', label='envelope', alpha=0.5)
ax.plot(df1['A_top'], c='orange', linestyle='--', label='envelope', alpha=0.5)

ax.legend()
```



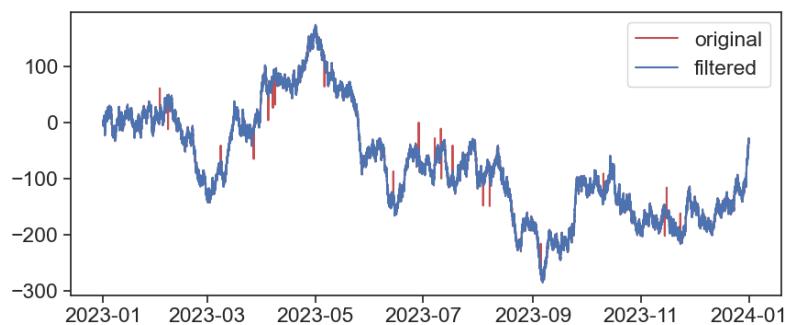
Now let's write a function so we can easily apply it to all our data:

```
def rolling_std_envelop(series, window_size=50, k=2):
    series.name = 'original'
    data = series.to_frame()
    data['std'] = data['original'].rolling(window_size, center=True, min_periods=1).std()
    # find the rolling mean
    data['mean'] = data['original'].rolling(window_size, center=True, min_periods=1).mean()
    # finding the top and bottom threshold for each datapoint
    data['top'] = data['mean'] + k*data['std']
    data['bottom'] = data['mean'] - k*data['std']
    # creating a mask of booleans that places true if the row is an outlier and false if its not
    data['outliers'] = ((data['original'] > data['top']) | (data['original'] < data['bottom']))
    # applying the mask and replacing all outliers with nans.
    data['filtered'] = np.where(data['outliers'],
                                np.nan, # use this if outliers is True
                                data['original']) # otherwise
    return data['filtered']
```

Let's test the new function:

```
fig, ax = plt.subplots(figsize=(10,4))

ax.plot(df1['A'], c='r', label='original')
ax.plot(rolling_std_envelop(df1['A']), c='b', label='filtered')
ax.legend()
```



Now let's reload `df1` so it will be clean (without all the added columns from before) and apply the function on all columns:

```

df1 = pd.read_csv('cleaning1.csv',
                  index_col='date',      # set the column date as index
                  parse_dates=True)     # turn to datetime format

df1_filtered = df1.copy()
df1_filtered

```

	A	B	C	D	E
date					
2023-01-01 00:00:00	0	0.000000	0.000000	0.000000	0.000000
2023-01-01 00:05:00	1	2.245251	-1.757193	1.899602	-0.999300
2023-01-01 00:10:00	2	2.909648	0.854732	2.050146	-0.559504
2023-01-01 00:15:00	3	3.483155	0.946937	1.921080	-0.402084
2023-01-01 00:20:00	2	4.909169	0.462239	1.368470	-0.698579
...	...	...	...	...	...
2023-12-31 23:35:00	-37	1040.909898	-14808.285199	1505.020266	423.595984
2023-12-31 23:40:00	-36	1040.586547	-14808.874072	1503.915566	423.117110
2023-12-31 23:45:00	-37	1042.937417	-14808.690745	1505.479671	423.862810
2023-12-31 23:50:00	-36	1042.411572	-14809.212002	1506.174334	423.862432
2023-12-31 23:55:00	-35	1043.053520	-14809.990338	1505.767197	423.647007

```

columns = df1_filtered.columns

for column in columns:
    filtered_column = rolling_std_envelop(df1_filtered[column], window_size=50, k=2)
    df1_filtered[column] = filtered_column

```

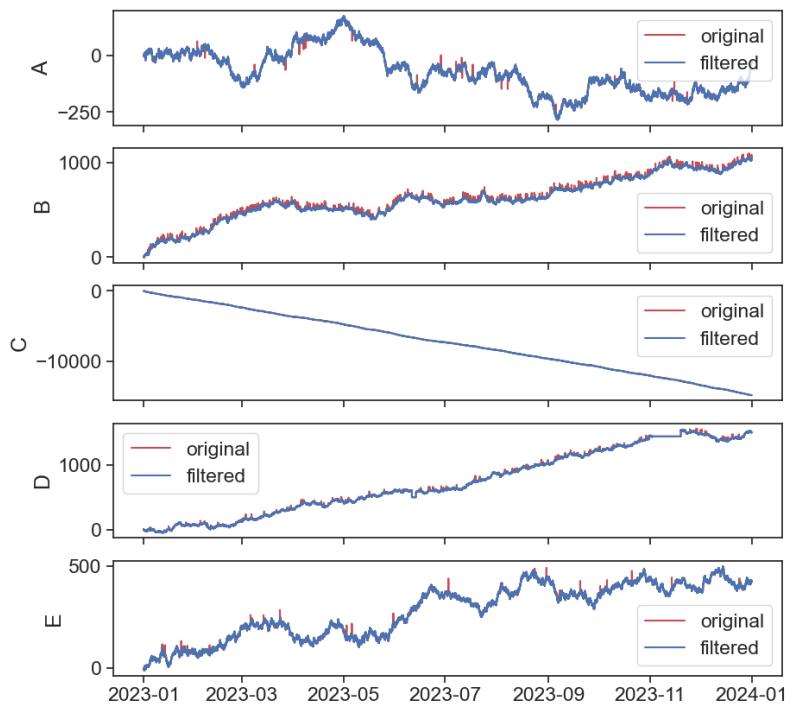
Now let's plot the results:

```

fig, ax = plt.subplots(len(columns), 1, sharex=True, figsize=(10, len(columns)*2))

for i, column in enumerate(columns):
    ax[i].plot(df1[column], c='r', label='original')
    ax[i].plot(df1_filtered[column], c='b', label='filtered')
    ax[i].legend()
    ax[i].set(ylabel=column)

```



## 22.3 plateaus

Now what about the plateaus in the series D?

That's another form of outliers.

The following function will find rows which have more than n equal following values and replace them with NaNs.

```
# function to copy paste:
def conseq_series(series, N):
    """
    part A:
    1. assume a string of 5 equal values. that's what we want to identify
    2. diff produces a string of only 4 consecutive zeros
    3. no problem, because when applying cumsum, the 4 zeros turn into a plateau of 5, that's so far, so good
    part B:
    1. groupby value_grp splits data into groups according to cumsum.
    2. because cumsum is monotonically increasing, necessarily all groups will be composed of n
    equal values, so we can just drop the last row of each group.
    """
    # part A
    # Create a new column 'diff' which contains the difference between consecutive elements
    series['diff'] = np.diff(series)
    # Create a new column 'cumsum' which contains the cumulative sum of the 'diff' column
    series['cumsum'] = np.cumsum(series['diff'])
    # Create a new column 'group' which contains the group index for each row
    series['group'] = series['cumsum'].groupby(series['cumsum']).ngroup()
    # Create a new column 'value_grp' which contains the value of the first row in each group
    series['value_grp'] = series.groupby('group').first()['cumsum']
    # Create a new column 'is_plateau' which contains True if the current row's value is equal to the previous row's value
    series['is_plateau'] = series['cumsum'].shift(1).eq(series['cumsum'])
    # Create a new column 'count' which counts the number of consecutive equal values
    series['count'] = np.where(series['is_plateau'], 1, 0).cumsum()
    # Create a new column 'plateau' which contains the value of the first row in each group
    series['plateau'] = series.groupby('group').first()['cumsum']
    # Replace the original 'cumsum' column with the 'plateau' column
    series['cumsum'] = series['plateau']
    # Drop the 'diff', 'group', 'value_grp', 'is_plateau', and 'count' columns
    series.drop(['diff', 'group', 'value_grp', 'is_plateau', 'count'], axis=1, inplace=True)
    # part B
    # Group by 'value_grp' and drop the last row of each group
    series = series.groupby('value_grp').drop(series.groupby('value_grp').count().last().index[-1], axis=0)
    # Drop the 'value_grp' column
    series.drop('value_grp', axis=1, inplace=True)
    # Return the modified series
    return series
```

```

3. what are those groups made of? of rows of column 'series'. this specific column is not
4. count 'counts' the number of elements inside each group.
5. the real magic here is that 'transform' assigns each row of the original group with the
6. finally, we can ask the question: which rows belong to a string of identical values greater
zehu, you now have a mask (True-False) with the same shape as the original series.

"""

# part A:
sumsum_series = (
    (series.diff() != 0)           # diff zero becomes false, otherwise true
        .astype('int')            # true -> 1 , false -> 0
        .cumsum()                 # cumulative sum, monotonically increasing
)
# part B:
mask_outliers = (
    series.groupby(sumsum_series)
        .transform('count')
        .ge(N)                   # take original series and group it
                                # now count how many are in each group
                                # if row count >= than user-defined N
)
# apply mask:
result = pd.Series(np.where(mask_outliers,
                            np.nan, # use this if mask_outliers is True
                            series), # otherwise
                            index=series.index)

return result

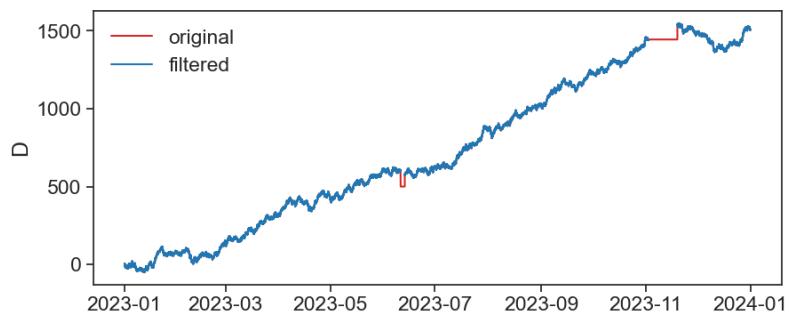
```

Let's apply it to the `df1_filtered` df so we will end with a cleaner signal

```

fig, ax = plt.subplots(figsize=(10,4))
ax.plot(df1_filtered['D'], color="tab:red", label='original')
ax.plot(conseq_series(df1_filtered['D'], 10), c='tab:blue', label='filtered')
ax.set_ylabel('D')
ax.legend(frameon=False)

```



### 22.3.1 TO DO:

**it's not homework but you should definitely do it**

- Try other filtering methods
- Tweak the parameters
- Use other dataframes (1,2,3 and if you have your own so better)
- Write custom filtering functions that you can save and use in your future work.

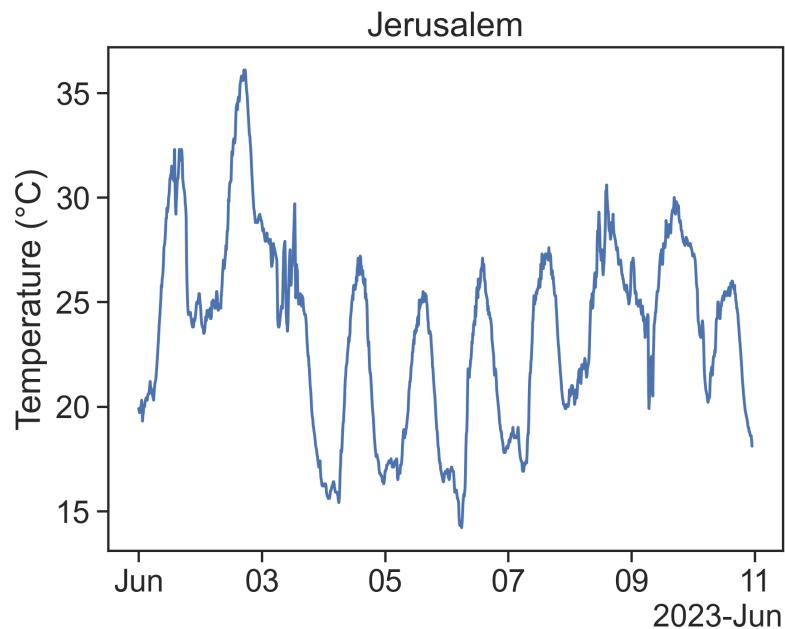
**Yalla have fun**

# **Part V**

## **stationarity**

## 23 motivation

See the temperatures for Jerusalem in a 4-day interval:



### 23.1 questions

- If I know the temperature right now, what does that tell me about the temperature 10 minutes from now? How about 100 minutes? 1000 minutes?
- The same applies to the past: how heavily does past information inform today's measurements?
- **information degradation:** how fast does information from a signal degrade, and gets swamped by noise?

## **23.2 goals**

- discuss what is noise, signal, trend, cycles, etc.
- learn a useful framework to make sense of all the above.
- acquire statistical and time-series tools to analyze my data
- eventually, all the above will be crucial to forecast future states.

Let's go!

# 24 random variables

This lecture is partially based on Brockwell and Davis (2016, chaps. 1, 2). Also [this](#).

A random variable is a mathematical concept used in probability theory and statistics to represent and quantify uncertainty. It is a variable whose possible values are outcomes of a random phenomenon. In other words, it's a variable that can take on different values with certain probabilities associated with each value.

## discrete random variable

There is a countable number of distinct outcomes. The obvious examples are coins and dice, which have 2 and 6 possible outcomes.



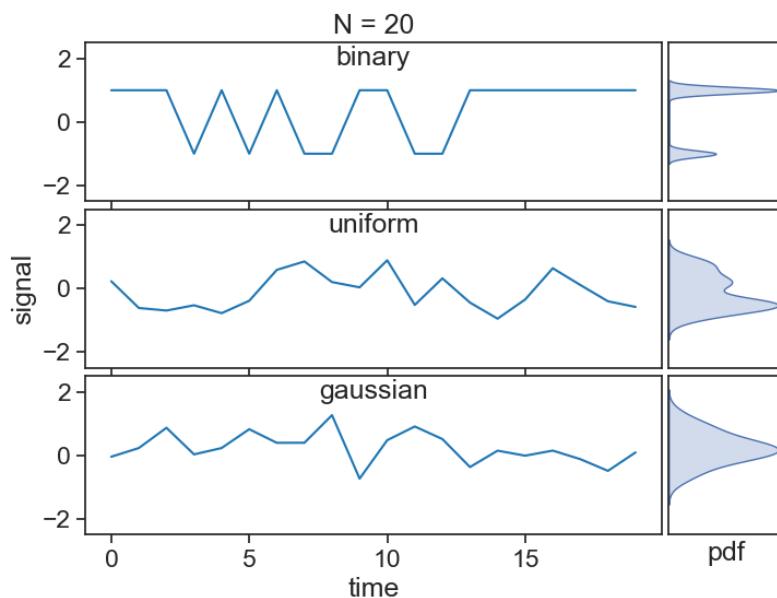
## continuous random variable

Any value within a range is possible. The position of a horizontal game spinner is a good example.

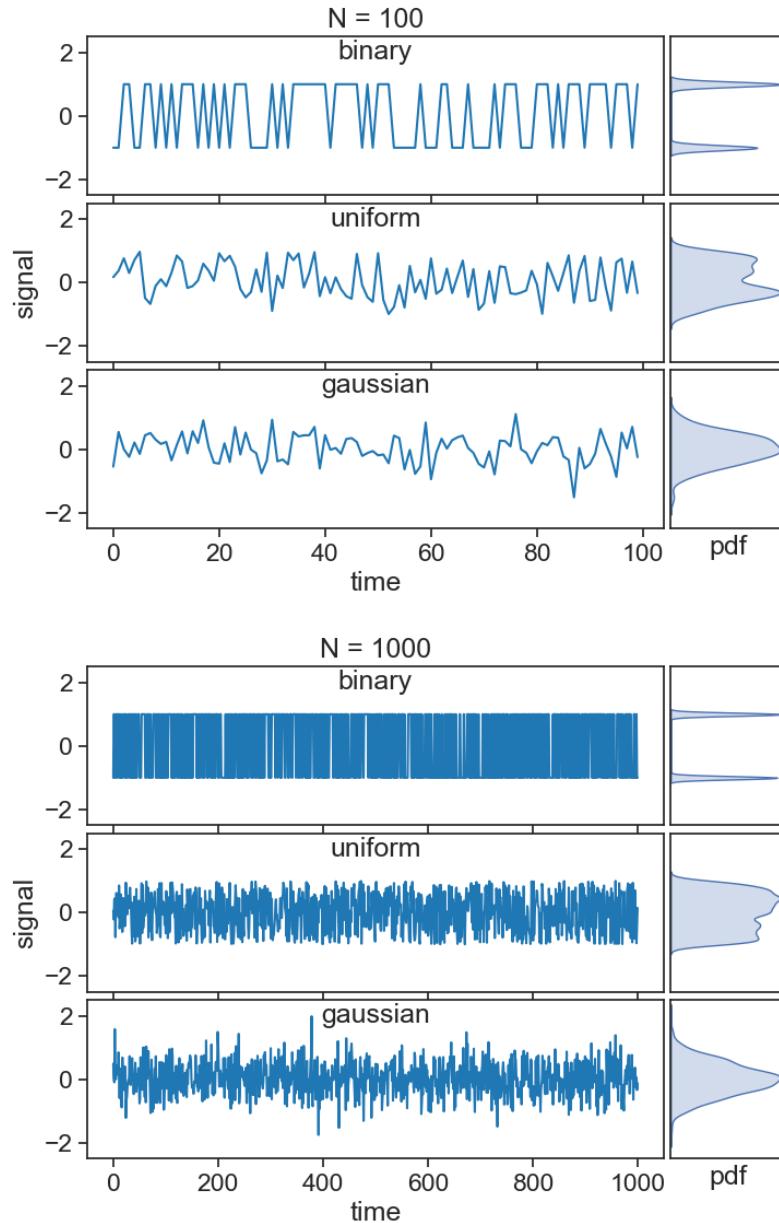


## 24.1 white noise

See below three time series made up by three different stochastic (random) processes. All terms in **each** of the series are **independent and identically distributed (iid)**, meaning that they are uncorrelated and taken from the same underlying distribution.



As we increase the length of the series, the statistics of each series reveal hints of the distributions they were sampled from:



The mathematical way of describing these series is thus:  $\{X\}$  represents the stochastic process (binary, uniform, etc), from which a specific series is randomly drawn:

$$\{x_0, x_1, x_2, \dots\}$$

All of these processes above have zero mean ( $\mu = 0$ ) and a finite variance ( $\sigma^2$ ), which qualify them as **white noise**.

## 24.2 random walk

A random walk  $S_t$  (for  $t = 0, 1, 2, \dots$ ) is obtained by cumulatively summing iid random variables:

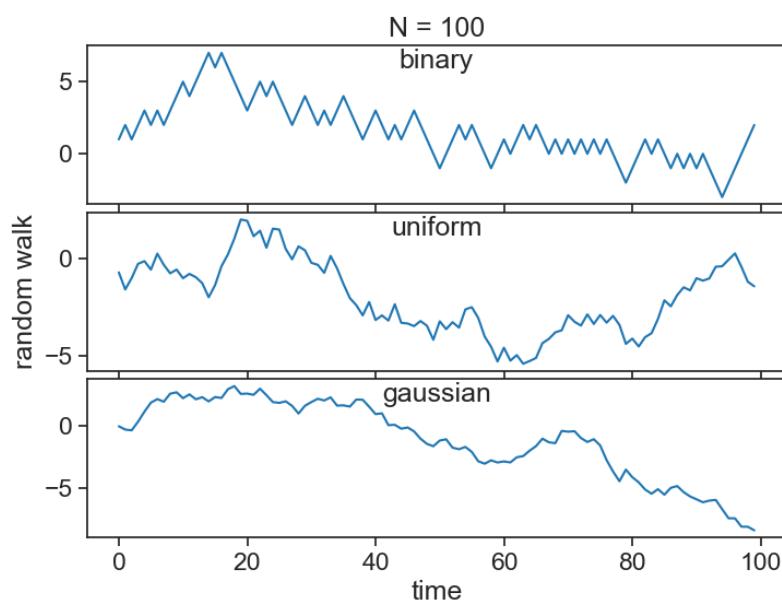
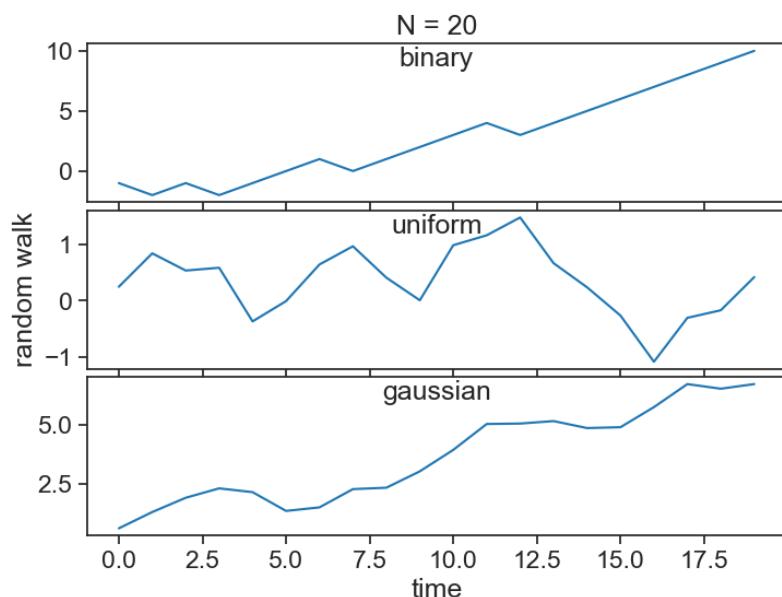
$$S_t = X_1 + X_2 + X_3 + \dots + X_{t-1} + X_t$$

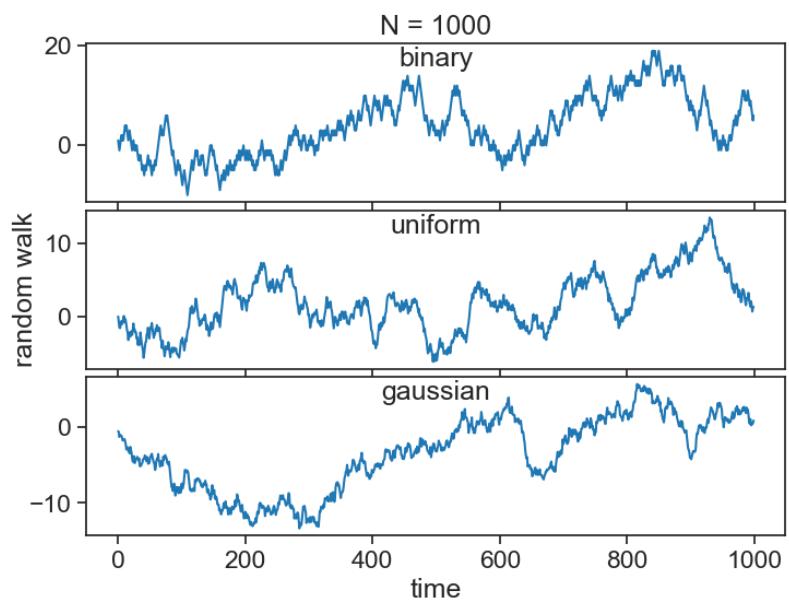
where  $S_0 = 0$ .

In the case of a binary process, you can think of the random walk as the position of a person who takes a step forward every time a coin toss yields heads, and a step backward for tails. Of course, by differencing the random walk, we can recover the original random sequence:

$$X_i = S_i - S_{i-1}.$$

See below the random walks associated with the three white noise processes from before:



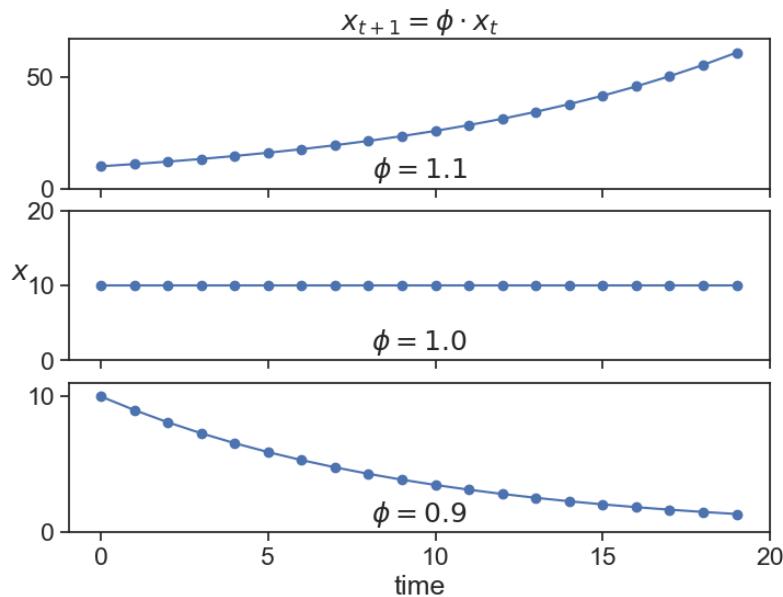


## 25 autoregressive processes

This lecture is partially based on Shumway and Stoffer (2017).

Consider the process

$$x_t = \phi \cdot x_{t-1}$$



These solutions look suspiciously similar to those obtained with the simplest 1st-order-homogenous differential equation:

$$\frac{dx}{dt} = \varphi x$$

whose solution is

$$x(t) = x_0 e^{\varphi t}$$

For the exponential solution, it is clear that  $x(t)$  grows to infinity if  $\varphi > 0$ , and it goes to zero if  $\varphi < 0$ . This connection is not casual, they are the discrete and continuous versions of the same process. Starting from the continuous process

$$\frac{dx}{dt} = \varphi x$$

and using an approximation of the derivative as

$$\frac{dx}{dt} \simeq \frac{x(t + \Delta t) - x(t)}{\Delta t}$$

we have that

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = \varphi x(t).$$

Solving for  $x(t + \Delta t)$  yields

$$x(t + \Delta t) = (1 + \Delta t \varphi)x(t).$$

Calling  $\phi = 1 + \Delta t \varphi$  and modifying the notation a little bit gives

$$X_{t+1} = \phi X_t.$$

## 25.1 AR(1)

Let's add some white noise ( $\varepsilon$ ) to this process.

$$X_t = \phi X_{t-1} + \varepsilon.$$

This is called an Autoregressive Process of order 1, or AR(1). Here, the current value  $x_t$  is dependent on the immediately preceding value  $x_{t-1}$ .

Using separation of variables:

$$\frac{dx}{x} = \varphi dt$$

We now integrate

$$\int \frac{1}{x} dy = \int \varphi dt$$

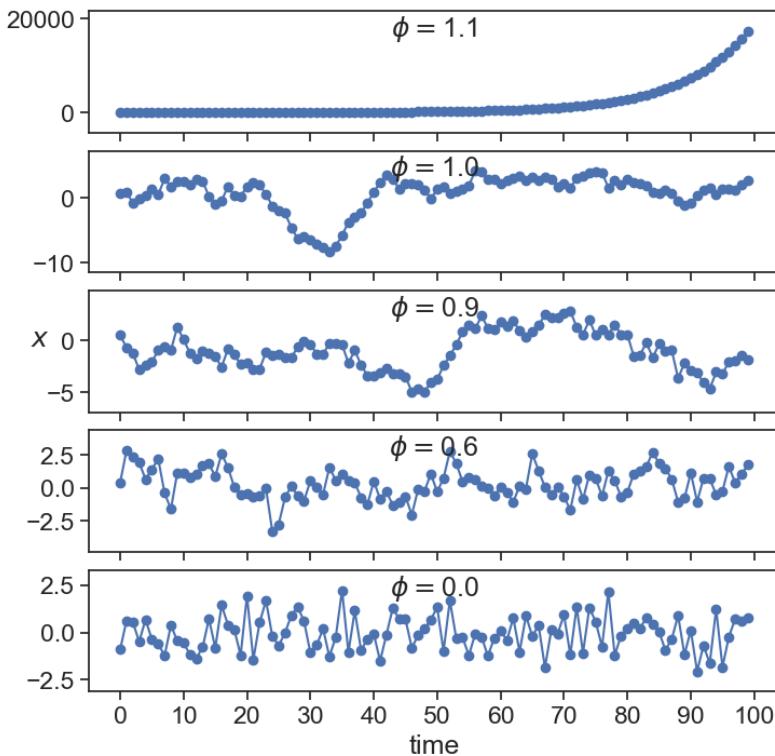
...yielding

$$\ln(x) = \varphi t + C$$

where  $C$  is an integration constant. Exponentiating both sides:

$$x(t) = e^{\varphi t} \cdot e^C = Ce^{\varphi t}$$

If we call the initial condition  $x(0) = x_0$ , we find that  $C = x_0$ , and we finally arrive at the solution.



What can we call these special cases?

- <sup>1</sup>  $\phi = 1.0$
- <sup>2</sup>  $\phi = 0.0$

The time series clearly explodes to infinity if  $\phi > 1$ , and seems to stay bounded for values equal or smaller than 1. We will come back to this observation in a little while, when we discuss stationarity.

## 25.2 AR(2)

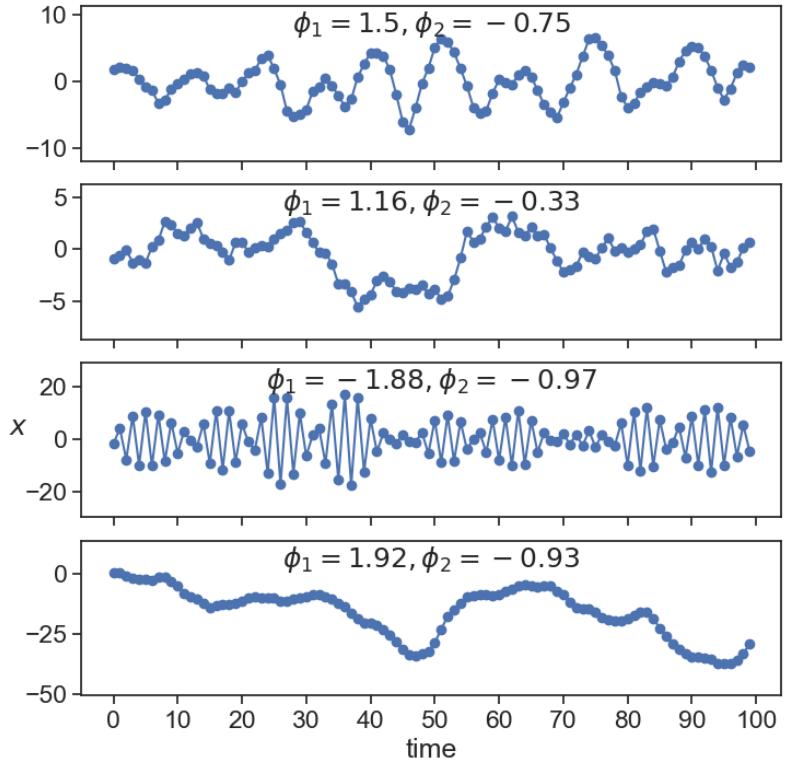
We can define a process that the current state is dependent on the two previous states, each with a different weight.

---

<sup>1</sup>random walk

<sup>2</sup>white noise

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \varepsilon$$



### 25.3 AR(p)

The next thing to do is to generalize, and define an autoregressive process that depends on  $p$  previous states:

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + \varepsilon$$

# 26 autocorrelation

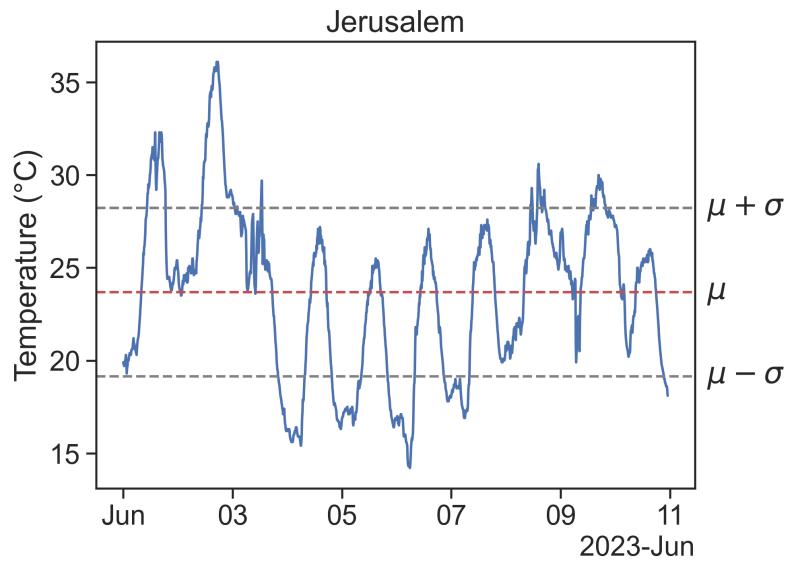
In this section, we will make use of a few fundamental concepts from statistics. Knowing these concepts well is fundamental to make sense of stationarity.

## 26.1 mean and standard deviation

Let's call our time series  $X$ , and its length  $N$ . Then:

$$\text{mean } \mu = \frac{\sum_{i=1}^N X_i}{N}$$
$$\text{standard deviation } \sigma = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$$

The mean and standard deviation can be visualized thus:



## 26.2 expected value

The expected value (or expectation) of a variable  $X$  is given by

$$E[X] = \sum_{i=1}^N X_i p_i.$$

$p_i$  is the weight or probability that  $X_i$  occurs. For a time series, the probability  $p_i$  that a given point  $X_i$  is in the dataset is simply  $1/N$ , therefore we can write the following measures in terms of expected values:

- mean, also called 1st moment:

$$\mu = E[X].$$

- variance, also called 2nd moment:

$$\sigma^2 = E[(X - E[X])^2] = E[(X - \mu)^2].$$

Of course,  $\sigma$  is called the standard deviation.

## 26.3 covariance

The covariance between two time series  $X$  and  $Y$  is given by

$$\begin{aligned}\text{cov}(X, Y) &= E[(X - E[X])(Y - E[Y])] \\ &= E[(X - \mu_X)(Y - \mu_Y)]\end{aligned}$$

Compare this to the definition of the variance, and it is obvious that the covariance  $\text{cov}(X, X)$  of a time series with itself is its variance.

## 26.4 correlation

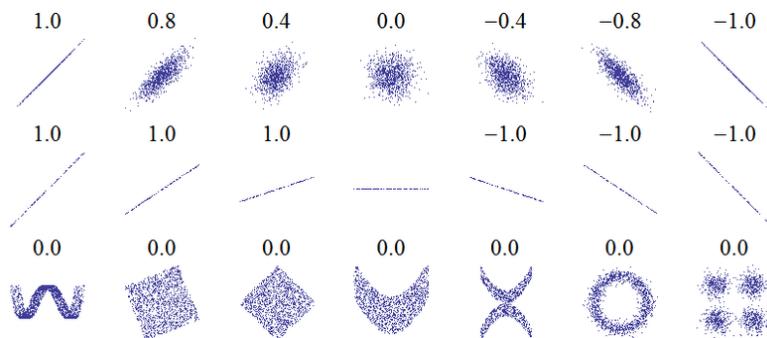
We are almost there. I promise.

The fact that  $\text{cov}(X, X) = \sigma_X^2$  begs us to define a new measure, the correlation:

$$\text{corr}(X, Y) = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}.$$

This is convenient, because now we can say that the correlation of a time series with itself is  $\text{corr}(X, X) = 1$ .

This is also called the Pearson correlation coefficient, and the result has a value between 1 and  $-1$ .



Source: [Wikimedia](#)

## 26.5 autocorrelation

The autocorrelation of a time series  $X$  is the answer to the following question:

if we shift  $X$  by  $\tau$  units, how similar will this be to the original signal?

In other words:

how correlated are  $X(t)$  and  $X(t + \tau)$ ?

The autocorrelation is expressed as

$$\rho_{XX}(\tau) = \frac{E[(X_t - \mu)(X_{t+\tau} - \mu)]}{\sigma^2}$$

The autocorrelation function  $\rho_{XX}(\tau)$  provides a useful measure of the degree of dependence among the values of a time series at different times.

A video is worth a billion words, so let's see the autocorrelation in action:

<https://youtu.be/tpf-tuYHR5w>

A few comments:

- The autocorrelation for  $\tau = 0$  (zero shift) is always 1.  
[Can you prove this? All the necessary equations are above!]

In other disciplines, the autocorrelation is simply the autocovariance, i.e., it is not normalized by dividing by  $\sigma^2$ . In time series it is assumed that the autocorrelation is always normalized, therefore between  $-1$  and  $1$ .

# 27 stationarity

## 27.1 weak stationarity

A time series is called weakly stationary if the following conditions are met:

1. its mean  $\mu$  does not vary in time:

$$\mu_X(t) = \mu_X(t + \tau)$$

for all values of  $t$  and  $\tau$ .

2. its variance is finite for any time  $t$ :

$$\sigma_X^2(t) < \infty.$$

3. The autocorrelation function between two lagged versions of the same time series,  $X(t_1)$  and  $X(t_2)$ , depends only on the difference  $\tau = t_2 - t_1$ .

## 27.2 strict/strong stationarity

As the name suggests, strict stationarity requires much more than the above. It requires that the joint CDF of two lagged versions of  $X$  is the same. It is rare to require such strong terms, we will assume weak stationarity always from now on.

## 27.3 stationarity of AR(p)

Let's write once more the definition of AR(p):

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + \varepsilon$$

We can define the **backward shift operator**  $B$  as

$$BX_t = X_{t-1}$$

Of course, if  $B$  is applied  $p$  times, it shifts  $X$  thus:

$$B^p X_t = X_{t-p}.$$

With that in hand, we can rewrite the definition of AR(p) as follows:

$$x_t = \phi_1 B x_t + \phi_2 B^2 x_t + \cdots + \phi_p B^p x_t + \varepsilon$$

We have many terms with  $x_t$ , so let's group them on the left-hand side:

$$x_t \phi(B) = \varepsilon,$$

where the **characteristic polynomial**  $\phi(B)$  is

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p$$

In order to determine the stationarity of AR(p), we need to find the roots of

$$\phi(B) = 0,$$

called characteristic roots. These roots are often complex numbers.

AR(p) is stationary if ALL the characteristic roots lie OUTSIDE the unit circle.

The reason for this is not obvious. A nice explanation can be found in this [StackExchange response](#). Another good text is Tsay (2010, chap. 2).

Brockwell and Davis (2016, chap. 2 p. 49 and chapter 3 p. 75) explain that, strictly speaking, an AR(p) process is stationary as long as the roots do not lie on the unit circle. However, in the case that the roots lie **inside** the unit circle, the process is **noncausal**, meaning that the present state depends on future states. In reality, everyone just ignores this point, and simply say that we require the roots to lie outside the unit circle to guarantee stationarity.

## 27.4 stationarity of AR(1)

We need to solve the roots of the characteristic equation

$$1 - \phi_1 B = 0.$$

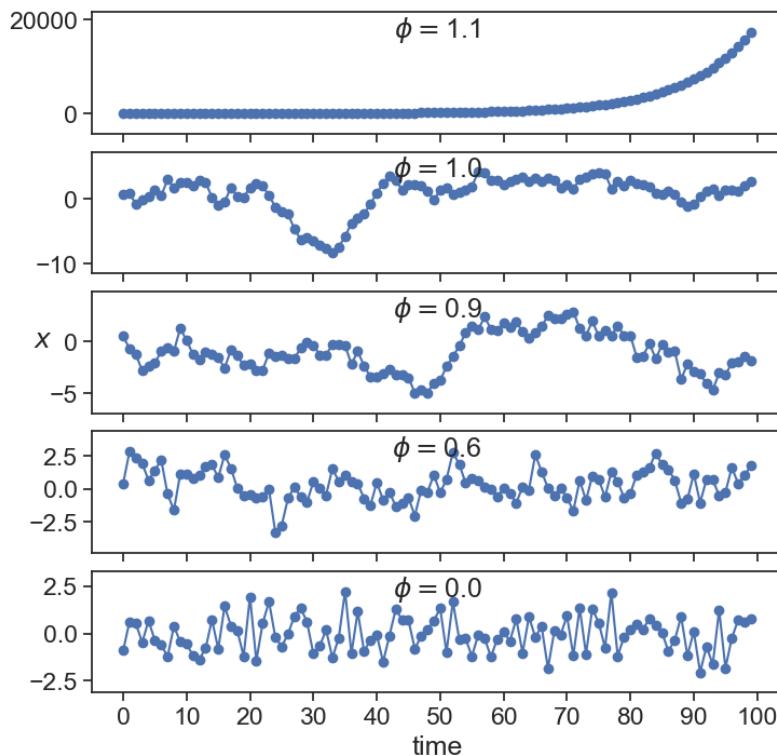
The only root is

$$B = \frac{1}{\phi_1}$$

Because we require that this root is greater than 1 (in absolute value), we have that:

$$\left| \frac{1}{\phi_1} \right| > 1 \quad \rightarrow \quad |\phi| < 1.$$

This result corroborates our conclusion from before:

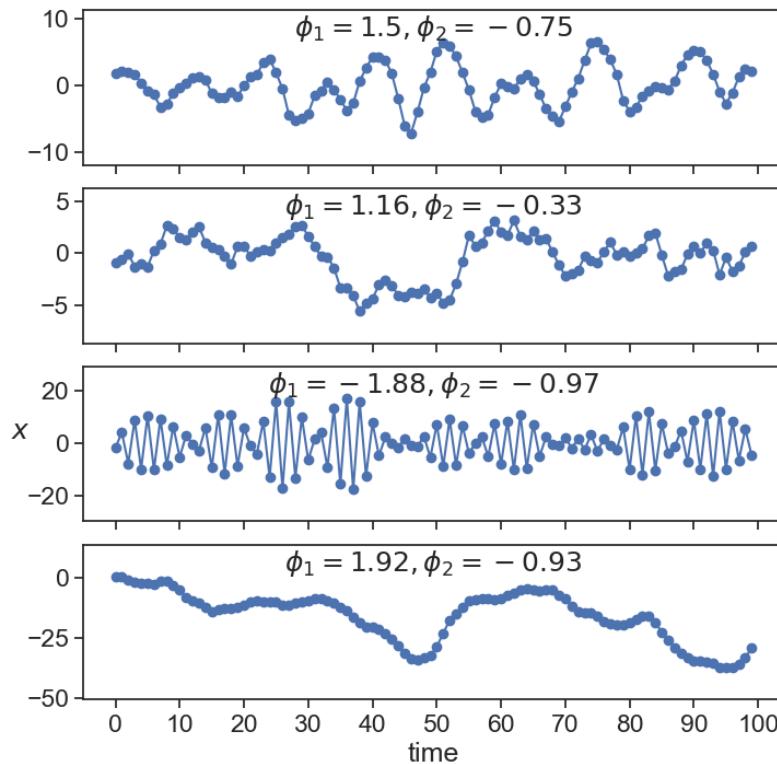


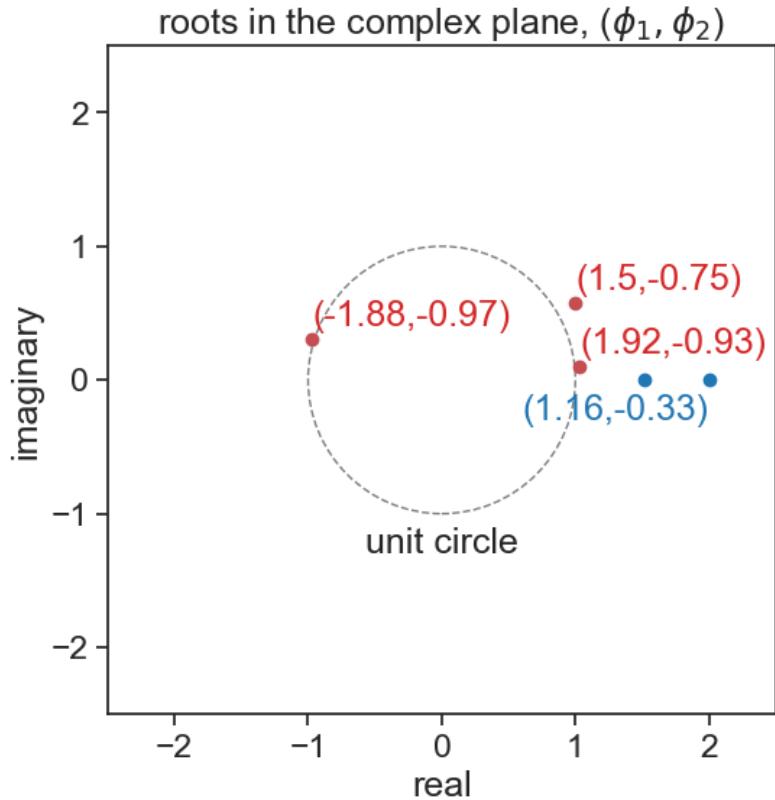
## 27.5 stationarity of AR(2)

We need to solve the roots of the characteristic equation

$$1 - \phi_1 B - \phi_2 B^2 = 0.$$

For the AR(2) time series from before, here are the roots of the characteristic polynomial plotted on the complex plane:





Because complex roots come in pairs, I plotted above only one of the roots, the one with positive imaginary component. The second panel ( $\phi_1 = 1.16, \phi_2 = -0.33$ ) has two real roots, so both are plotted in blue.

The AR(2) time series will have a periodic component if the roots are complex, and the frequency of oscillation is

$$f_0 = \frac{1}{2\pi} \cos^{-1} \left( \frac{\phi_1}{2\sqrt{(-\phi_2)}} \right)$$

Play with the widget below to get a feel of how the complex roots depend on the values of  $\phi_1$  and  $\phi_2$ . In the left panel you can move the point A to choose different  $\phi$  values, and on the right you see the complex roots of the polynomial instantly updated. As long as the point A is inside the blue triangle, the roots will be outside the unit circle, and therefore the process

See an interesting discussion in  
[David Josephs' excellent time series webpage](#)

will be stationary. For  $\phi_2$  values above (below) the red line, the complex roots will be real (complex conjugates).

Conversely, play with the position of one of the complex roots, and see how this influences the value of  $\phi_1, \phi_2$ .

The two roots are easy to find from  $\phi_1, \phi_2$ , you just need to solve

$$1 - \phi_1 B - \phi_2 B^2 = 0$$

for B. What if you choose two roots  $z_1, z_2$  and want to derive from them the value of  $\phi_1, \phi_2$ ? Just use this:

$$\phi_1 = \frac{z_1 + z_2}{z_1 \cdot z_2}$$

$$\phi_2 = -\frac{1}{z_1 \cdot z_2}$$

## 27.6 stationarity of AR(4)

Conceptually, this is exactly like AR(2). In case you ever need to choose AR(4)  $\phi$  values, play with the widget below and see if you can put all the roots outside the unit circle.

Geogebra app made by Yair Mau  
(2024)

## 28 ACF and PACF graphs

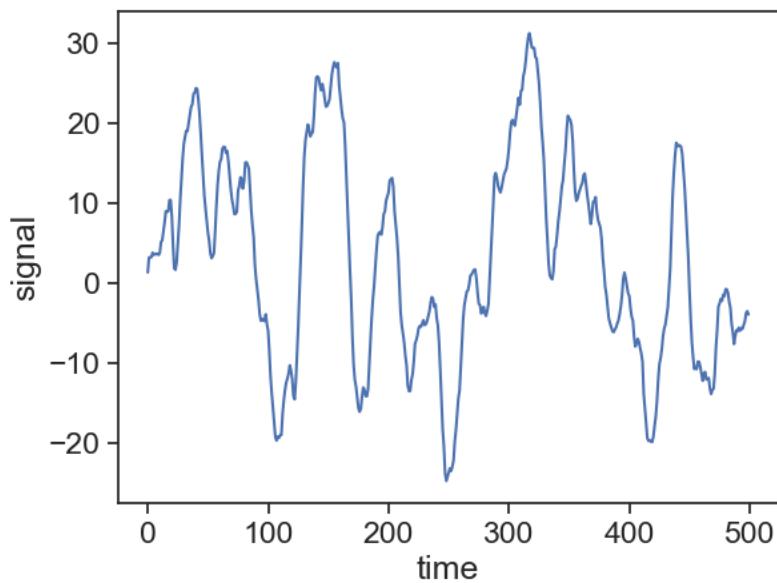
We will now see what is the connection between stationarity and autocorrelation.

Using Python's `statsmodels` package, let's create an AR time series and plot it.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess

phi1 = 1.86
phi2 = -0.87
N = 500
np.random.seed(10) # For reproducibility
ar2_process = ArmaProcess(ar=[1, -phi1, -phi2], ma=[1])
ar2_values = ar2_process.generate_sample(nsample=N)

fig, ax = plt.subplots()
ax.plot(ar2_values)
ax.set(xlabel="time",
       ylabel="signal");
```



## 28.1 ACF

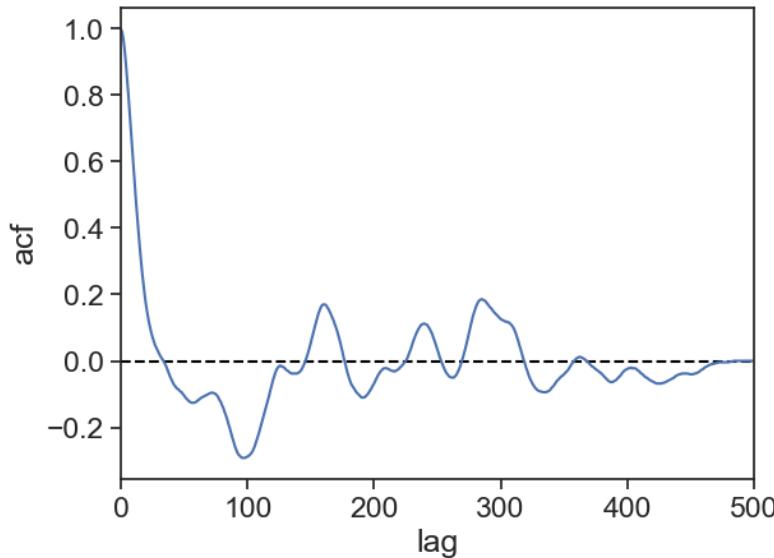
We need to calculate now the autocorrelation function of our series:

$$\rho_{XX}(\tau) = \frac{E[(X_t - \mu)(X_{t+\tau} - \mu)]}{\sigma^2}$$

```
def compute_acf(series):
    N = len(series)
    lags = np.arange(N)
    acf = np.zeros_like(lags)
    series = (series - series.mean()) / series.std()
    for i in lags:
        acf[i] = np.sum(series[i:] * series[:N-i])
    acf = acf / N
    return lags, acf
```

```
fig, ax = plt.subplots()
lags, acf = compute_acf(ar2_values)
ax.plot([0, N], [0]*2, color="black", ls="--")
ax.plot(lags, acf)
ax.set(xlabel="lag",
```

```
ylabel="acf",  
xlim=[0, N]);
```



Notice that the ACF always starts at 1 for zero lag, and it gets closer to zero as the lag increases.

- **Intuitive interpretation:** Two measurements taken within a short time interval (lag) should be similar, therefore their correlation is expected to be high. As we compare measurements from increasing time intervals, they are less and less similar to one another, therefore their correlation goes down.
- **Mathematical interpretation:** Take a look at the code we wrote above. As the lags grows, the length of both arrays keeps shrinking, but we still divide the result by  $N$ . The logical conclusion is that when  $\tau = N$  the ACF will be exactly zero.

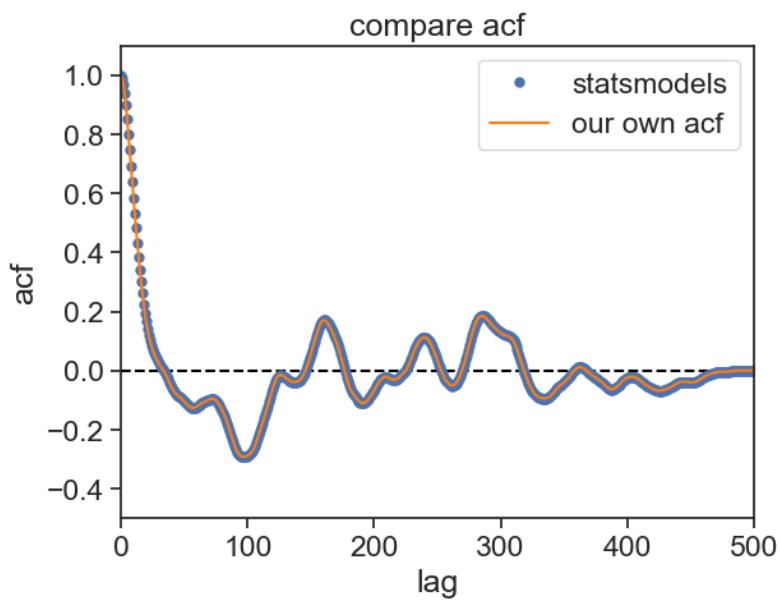
The `statsmodels` package also offers an easy way to plot the ACF, let's compare our calculation with the built-in function:

```
fig, ax = plt.subplots()  
ax.plot([0, N], [0]*2, color="black", ls="--")  
sm.graphics.tsa.plot_acf(ar2_values, lags= N-1, ax=ax, label="statsmodels", alpha=None, use_vl
```

```

ax.plot(lags, acf, color="tab:orange", label="our own acf")
ax.legend()
ax.set(ylim=[-0.5, 1.1],
      xlim=[0, N],
      title="compare acf",
      xlabel="lag",
      ylabel="acf");

```

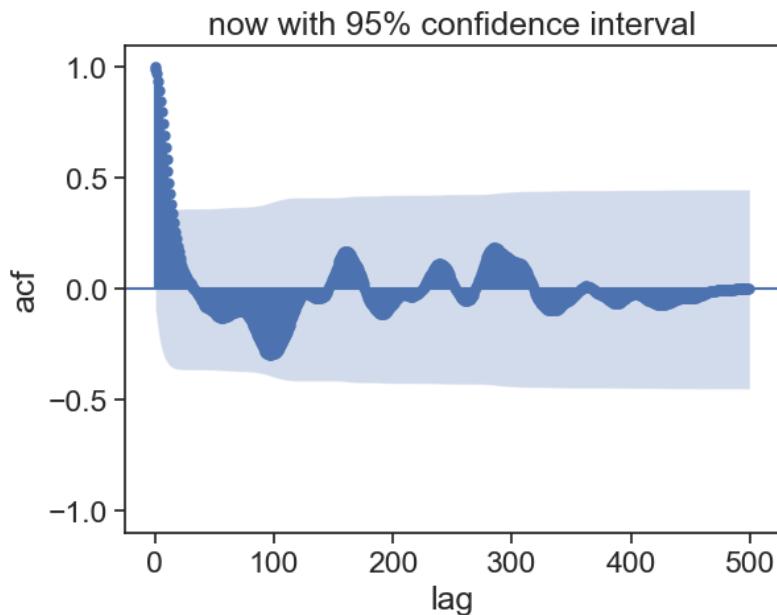


Excellent! From now on we will continue using `statsmodels` functions. We can spice up the ACF graph, by showing an envelope of 95% confidence interval.

```

fig, ax = plt.subplots()
sm.graphics.tsa.plot_acf(ar2_values, lags= N-1, ax=ax, alpha=.05)
ax.set(ylim=[-1.1, 1.1],
      title="now confidence interval")
ax.set(ylim=[-1.1, 1.1],
      title="now with 95% confidence interval",
      xlabel="lag",
      ylabel="acf");

```



If an autocorrelation value at a specific lag falls outside the confidence interval, it suggests that the autocorrelation at that lag is statistically significant. In other words, there is evidence of correlation at that lag. If an autocorrelation value is within the confidence interval, it suggests that the autocorrelation at that lag is not statistically significant, and any observed correlation might be due to random noise. The width of the confidence interval is influenced by the significance level. For a 95% confidence interval, it means that you are 95% confident that the true autocorrelation lies within the interval. If you choose a higher confidence level, the interval will become wider, making it harder to reject the null hypothesis of no correlation.

### 28.1.1 problem?

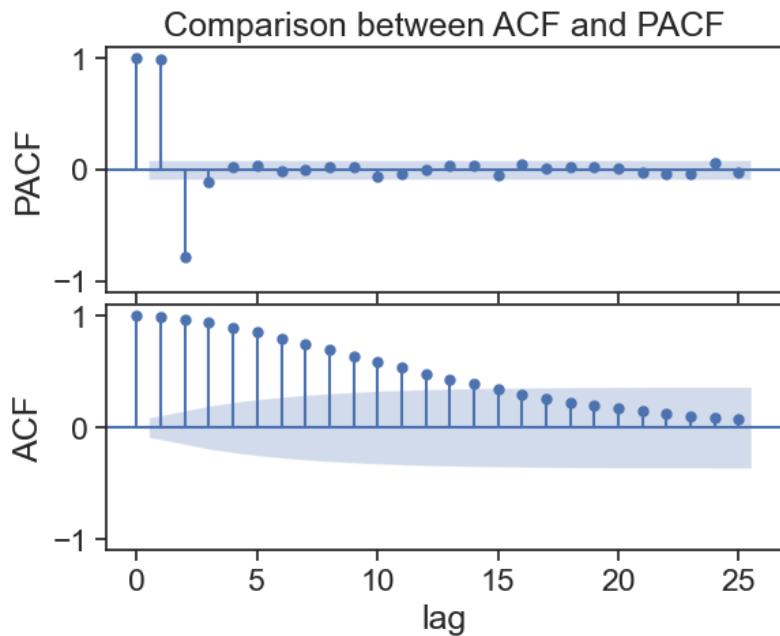
There is something a bit troubling about the ACF graph. We can learn from it how fast the correlation between two points in time falls, but this analysis is not too clean. Assume that the present state  $x_t$  is only dependent on one time step back,  $x_{t-1}$ . Because  $x_{t-1}$  is dependent on  $x_{t-2}$ , the result is that we will find that  $x_t$  is weakly dependent on  $x_{t-2}$ , although the direct dependence is zero.

The Partial ACF (PACF) solves this problem. It removes the intermediate effects between two points, and returns only the direct influence of one time instant on another one lagged by  $\tau$ . Let's see how it looks like for the process above.

## 28.2 PACF

```
fig, (ax1, ax2) = plt.subplots(2,1)
fig.subplots_adjust(hspace=0.05)
sm.graphics.tsa.plot_pacf(ar2_values, lags=25, ax=ax1, alpha=.05)
ax1.set(ylim=[-1.1, 1.1],
        title="now confidence interval")
ax1.set(ylim=[-1.1, 1.1],
        title="Comparison between ACF and PACF",
        xlabel="lag",
        ylabel="PACF");

sm.graphics.tsa.plot_acf(ar2_values, lags= 25, ax=ax2, alpha=.05, title=None)
ax2.set(ylim=[-1.1, 1.1],
        xlabel="lag",
        ylabel="ACF");
```



We see three bars significantly far from the confidence interval. The leftmost shows  $\text{PACF}(\tau = 0) = 1$ , which is expected, so let's not discuss it. The two next bars are the really important ones, they show the greatest correlation. From then on, the correlation for lags greater than 2 is not significant. With PACF's help, we can infer that the original AR processes must have been of order 2.

## 28.3 discussion

What can we say about the following series?

### 28.3.1 sine wave

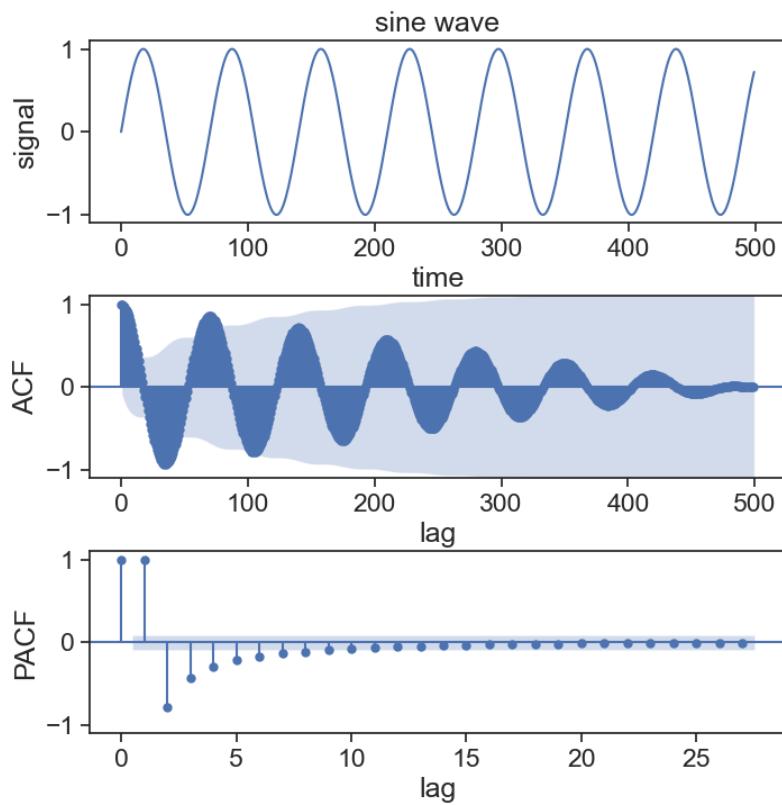
```
N = 500
time = np.arange(N)
period = 70
omega = 2.0 * np.pi / period
signal = np.sin(omega * time)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3,1, figsize=(8,8))
fig.subplots_adjust(hspace=0.4)

ax1.plot(time, signal)
ax1.set(ylabel="signal",
         title="sine wave",
         xlabel="time")

sm.graphics.tsa.plot_acf(signal, ax=ax2, alpha=.05, title=None, lags=N-1)
ax2.set(ylim=[-1.1, 1.1],
        xlabel="lag",
        ylabel="ACF");

sm.graphics.tsa.plot_pacf(signal, ax=ax3, alpha=.05, title=None)
ax3.set(ylim=[-1.1, 1.1],
        ylabel="PACF",
        xlabel="lag");
```



### 28.3.2 white noise

```

N = 500
time = np.arange(N)
signal = np.random.normal(size=N)

fig, (ax1, ax2, ax3) = plt.subplots(3,1, figsize=(8,8))
fig.subplots_adjust(hspace=0.4)

ax1.plot(time, signal)
ax1.set(ylabel="signal",
        title="white noise",
        xlabel="time")

sm.graphics.tsa.plot_acf(signal, ax=ax2, alpha=.05, title=None, lags=25)

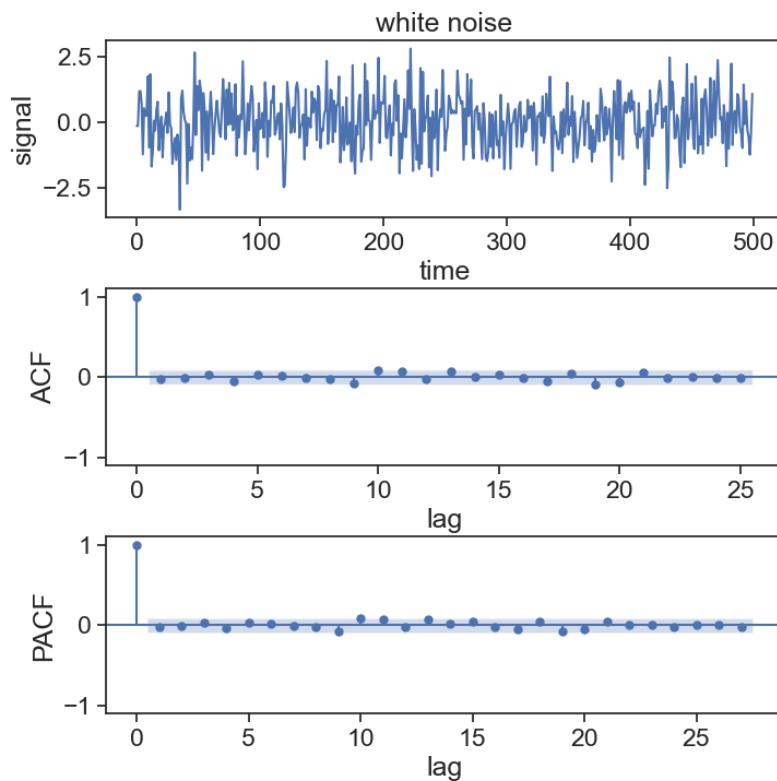
```

```

ax2.set(ylim=[-1.1, 1.1],
        xlabel="lag",
        ylabel="ACF");

sm.graphics.tsa.plot_pacf(signal, ax=ax3, alpha=.05, title=None)
ax3.set(ylim=[-1.1, 1.1],
        ylabel="PACF",
        xlabel="lag");

```



### 28.3.3 random walk

```

N = 500
time = np.arange(N)
signal = np.cumsum(np.random.normal(size=N))

```

```

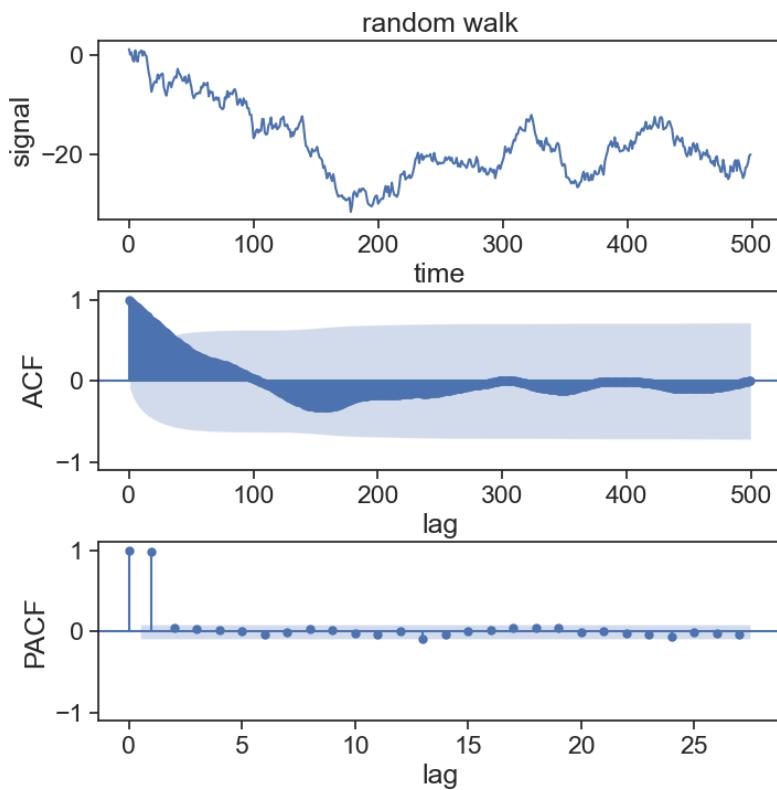
fig, (ax1, ax2, ax3) = plt.subplots(3,1, figsize=(8,8))
fig.subplots_adjust(hspace=0.4)

ax1.plot(time, signal)
ax1.set(ylabel="signal",
         title="random walk",
         xlabel="time")

sm.graphics.tsa.plot_acf(signal, ax=ax2, alpha=.05, title=None, lags=N-1)
ax2.set(ylim=[-1.1, 1.1],
        xlabel="lag",
        ylabel="ACF");

sm.graphics.tsa.plot_pacf(signal, ax=ax3, alpha=.05, title=None)
ax3.set(ylim=[-1.1, 1.1],
        ylabel="PACF",
        xlabel="lag");

```



# 29 from AR to ARIMA

Up to this point, we learned what an AR process is, and how it relates to the concept of stationarity.

Our long term goal is to use these concepts to make forecasts (predictions) about the future. Before we do that, it is useful to talk about a generalization of the AR process, that better resembles real-life data.

## 29.1 AR( $p$ )

An **autoregressive** process  $X$  is one that depend on  $p$  past states:

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \cdots + \phi_p X_{t-p} + \varepsilon$$

From what we already learned, if the complex roots of the polynomial

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p$$

lie **outside** the unit circle, then the AR process is **causal and stationary**.

## 29.2 MA( $q$ )

Similarly, a **moving average** process  $X$  is one that depend on  $q$  past noise steps:

$$X_t = \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_p \varepsilon_{t-q}$$

This has nothing to do with the sliding averages used for smoothing we studied before, it's just the same name for a different concept.

Note that this equation is identical in structure to that of AR(p), but with weights  $\theta$  standing for  $\phi$ , and past noise  $\varepsilon_{t-i}$  standing in for past states  $X_{t-i}$ .

This process also has its characteristic polynomial:

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \cdots - \theta_p B^p$$

The complex roots of this polynomial are also important. As long as the roots are **outside** the unit circle, the MA(q) process will be considered **invertible**, which is to say that it can be transformed into an AR( $\infty$ ) process.

## 29.3 ARMA(p,q)

An ARMA(p,q) process is simply the combination of an AR(p) and an MA(q) process:

The story is of course more complex than that. Using intelligent mathematical tricks (substitutions), one can change the noise term to make roots move from inside the unit circle to the outside, so effectively there shouldn't be any problems as long as there aren't any roots exactly on the unit circle.

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \cdots + \phi_p X_{t-p} \\ + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_p \varepsilon_{t-p}$$

## 29.4 ACF and PACF

The graphs for the autocorrelation and partial autocorrelation functions can be very useful to identify the order  $p$  and  $q$  of an ARMA(p,q) process.

```
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import seaborn as sns
```

```

sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.tsa.stattools import adfuller

np.random.seed(0)
n = 1000 # number of data points
phi_list = np.array([0.8, -0.28, 0.8, -0.36])
ar_coefs = np.insert(-phi_list, 0, 1) # AR coefficients. append 1 at the beginning
ma_coefs = [1] # MA coefficients

arma_process = ArmaProcess(ar_coefs, ma_coefs)
data = arma_process.generate_sample(nsample=n)

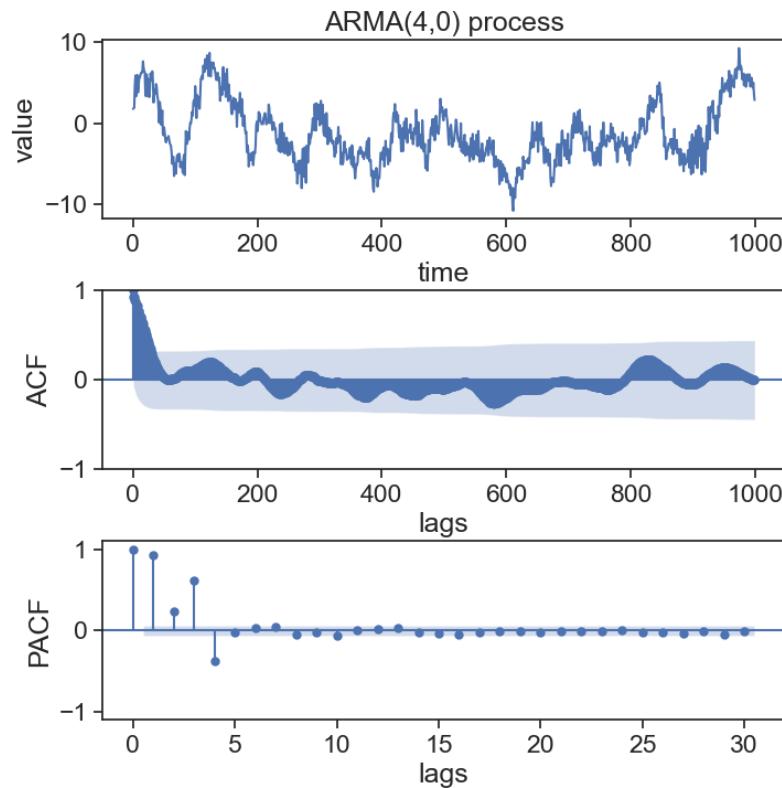
fig, axes = plt.subplots(3, 1, figsize=(8, 8))
fig.subplots_adjust(hspace=0.4) # increase vertical space between panels

ax1 = axes[0]
ax1.plot(data)
ax1.set(xlabel='time',
         ylabel='value',
         title='ARMA(4,0) process',
         )

# plot ACF and PACF graphs
ax2 = axes[1]
sm.graphics.tsa.plot_acf(data, lags=n-1, ax=ax2, title=None)
ax2.set(ylabel="ACF",
         xlabel="lags")

ax3 = axes[2]
sm.graphics.tsa.plot_pacf(data, lags=30, ax=ax3, title=None)
ax3.set(ylim=[-1.1, 1.1],
         ylabel='PACF',
         xlabel="lags");

```



Note that for the ARMA(4,0) process, the last significant PACF value is at lag  $\tau = 4$ .

```

np.random.seed(0)
n = 1000 # number of data points
theta_list = np.array([0.4, -0.3, 0.8])
ma_coefs = np.insert(-theta_list, 0, 1) # MA coefficients. append 1 at the beginning
ar_coefs = [1]

arma_process = ArmaProcess(ar_coefs, ma_coefs)
data = arma_process.generate_sample(nsamp=n)

fig, axes = plt.subplots(3, 1, figsize=(8, 8))
fig.subplots_adjust(hspace=0.4)

ax1 = axes[0]
ax1.plot(data)
ax1.set(xlabel='time',
        title='ARMA(4,0) process')

```

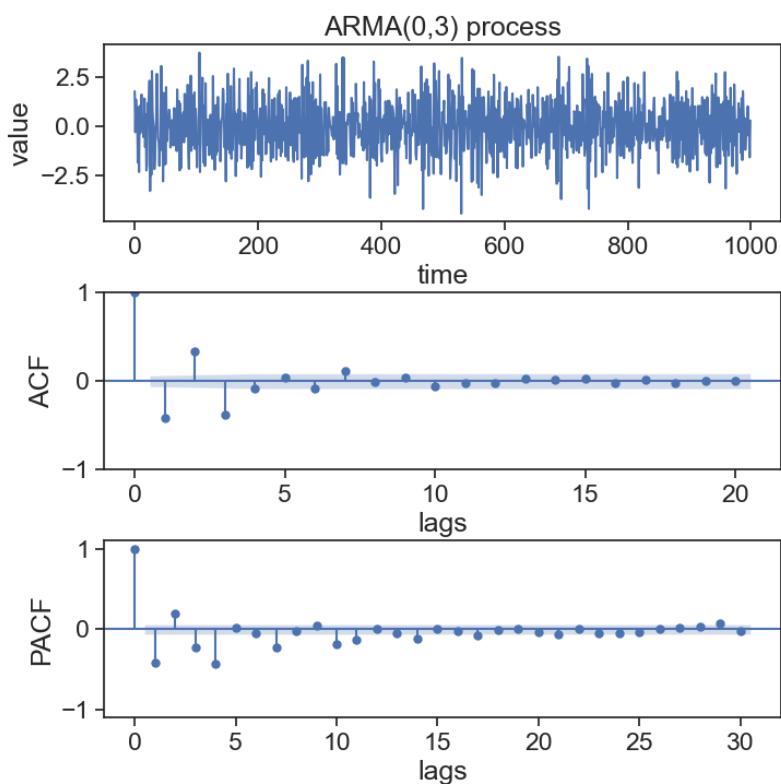
```

        ylabel='value',
        title='ARMA(0,3) process',
    )

# plot ACF and PACF graphs
ax2 = axes[1]
sm.graphics.tsa.plot_acf(data, lags=20, ax=ax2, title=None)
ax2.set(ylabel="ACF",
         xlabel="lags")

ax3 = axes[2]
sm.graphics.tsa.plot_pacf(data, lags=30, ax=ax3, title=None)
ax3.set(ylim=[-1.1, 1.1],
        ylabel='PACF',
        xlabel="lags");

```



For the ARMA(0,3) process, the last significant ACF value is at lag  $\tau = 3$ .

```

np.random.seed(0)
n = 1000 # number of data points
theta_list = np.array([0.4, -0.3, 0.8])
phi_list = np.array([0.8, -0.28, 0.8, -0.36])
ar_coefs = np.insert(-phi_list, 0, 1) # AR coefficients
ma_coefs = np.insert(-theta_list, 0, 1) # MA coefficients

arma_process = ArmaProcess(ar_coefs, ma_coefs)
data = arma_process.generate_sample(nsample=n)

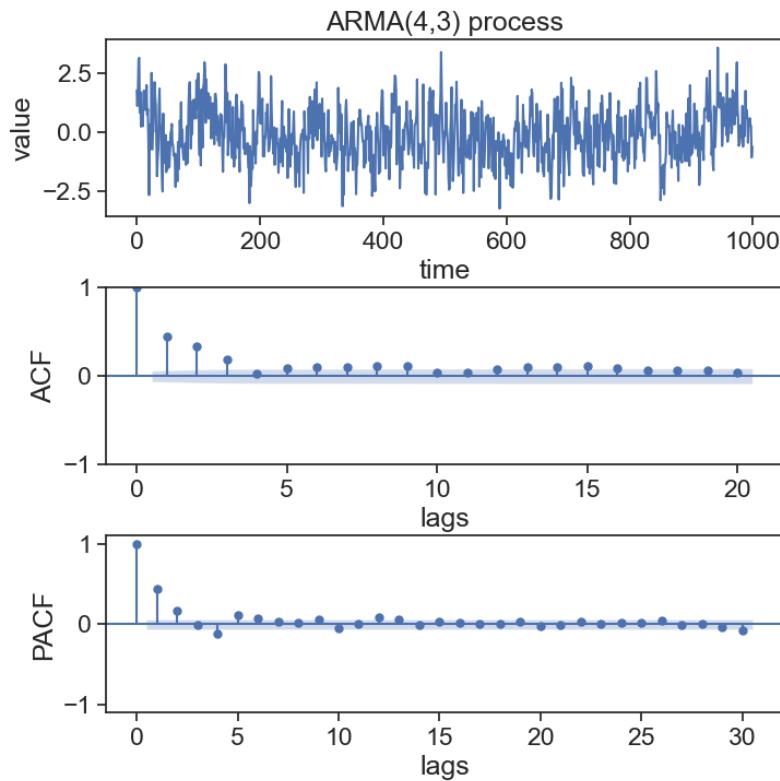
# Create a single figure with panels
fig, axes = plt.subplots(3, 1, figsize=(8, 8))
fig.subplots_adjust(hspace=0.4)

# Plot the ARMA process
ax1 = axes[0]
ax1.plot(data)
ax1.set(xlabel='time',
        ylabel='value',
        title='ARMA(4,3) process',
        )

# Plot ACF and PACF graphs
ax2 = axes[1]
sm.graphics.tsa.plot_acf(data, lags=20, ax=ax2, title=None)
ax2.set(ylabel="ACF",
        xlabel="lags")

ax3 = axes[2]
sm.graphics.tsa.plot_pacf(data, lags=30, ax=ax3, title=None)
ax3.set(ylim=[-1.1, 1.1],
        ylabel='PACF',
        xlabel="lags");

```



This table from Shumway and Stoffer (2017, 108) is useful to sum up what we've learned so far.

	AR(p)	MA(q)	ARMA(p,q)
ACF	gradually goes down	cuts off after lag q	gradually goes down
PACF	cuts off after lag p	gradually goes down	gradually goes down

## 29.5 Non-stationary data and ADF test

The following is partially based on Chatfield (2016, chap. 3, page 63).

What do we do if it turns out that our data is not stationary? Heck, how can we even tell if our data is stationary or not? The

most common stationarity test is the Augmented Dickey–Fuller (ADF) test. This is not a trivial subject that can be completely understood in a few words, so I'll give the very basic intuition here.

A stationary time series has a constant mean  $\mu$ . If at a given instant  $t$  our state  $X_t$  is way above the mean, we would expect that, with a high probability, the next step brings it closer to the mean. This is to say that the difference between two consecutive states  $X_t - X_{t-1}$  depends on the value of  $X_t$ ! Nonstationary time series do not show this behavior: the differences between two time steps do not depend on the state value. The idea described here is for the Dickey-Fuller test. The Augmented Dickey-Fuller test is basically the same, but for time lags  $p$  between states, not only 1.

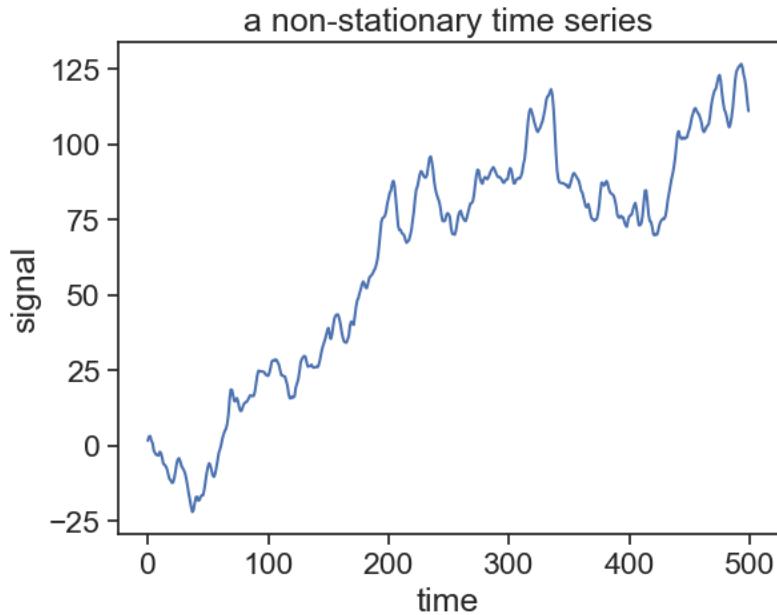
The ADF test has a null hypothesis that the time series is **not stationary**. By applying the test to a given time series, we get a p-value as one of the results. The smaller the p-value, the more evidence we have to reject the null hypothesis, and therefore conclude that our time series is indeed stationary.

Let's see an example:

```
# Generate ARIMA(1,1,2) process with differencing
N = 500
np.random.seed(1)
arima_112_diff = sm.tsa.arma_generate_sample(ar=[1, -0.5], ma=[1, 0.7, 0.3], nsample=N)
arima_112 = np.cumsum(arima_112_diff)

fig, ax = plt.subplots()
ax.plot(arima_112)
ax.set(xlabel="time",
       ylabel="signal",
       title="a non-stationary time series")
plt.show()

result = adfuller(arima_112)
print('p-value: ', result[1])
```



p-value: 0.591478751185507

So what do we do if we have a non-stationary time series? One common solution is to apply successive **differencing** operations, until the outcome becomes stationary.

Let's define the difference operator  $\nabla$  as

$$\nabla X_t = X_t - X_{t-1}.$$

Now recalling that the backward shift operator  $B$  is defined as

$$BX_t = X_{t-1},$$

we can rewrite the difference operator as

$$\begin{aligned}\nabla X_t &= X_t - BX_t \\ &= (1 - B)X_t.\end{aligned}$$

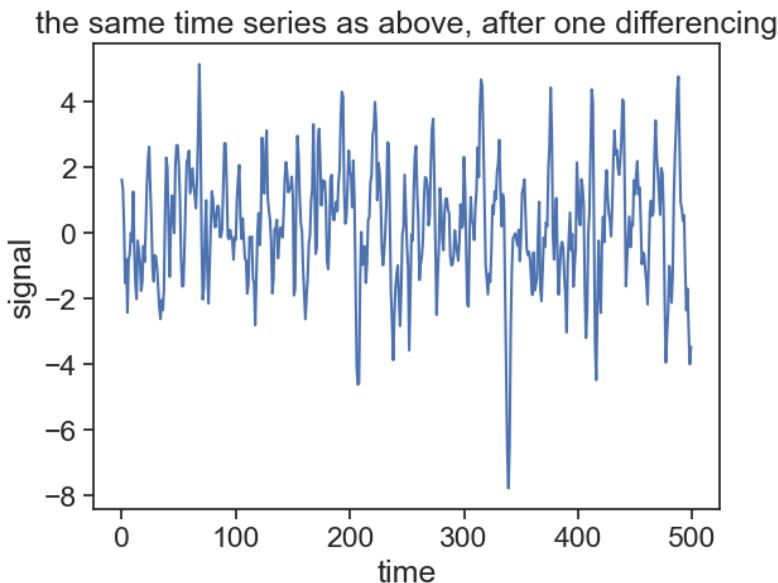
If we apply the difference operator  $d$  times, then we denote this as

$$W_t = \nabla^d X_t = (1 - B)^d X_t.$$

Let's apply the difference operator once to the time series plotted above, and then apply the ADF test.

```
fig, ax = plt.subplots()
ax.plot(arima_112_diff)
ax.set(xlabel="time",
       ylabel="signal",
       title="the same time series as above, after one differencing")
plt.show()

result = adfuller(arima_112_diff)
print('p-value: ', result[1])
```



p-value: 4.7341140554650393e-14

## 29.6 ARIMA(p,d,q)

We are ready to describe an Autoregressive (AR) Integrated (I) Moving Average (MA) process:

$$W_t = \phi_1 W_{t-1} + \phi_2 W_{t-2} + \dots \phi_q W_{t-q} \\ + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \dots \theta_q \varepsilon_{t-q}$$

Rearranging the terms in this equation, we can also express an ARIMA(p,d,q) process as

$$\phi(B)(1 - B)^d X_t = \theta(B)\varepsilon_t.$$

Let's try to put this in a context we already know. We saw that a random walk is the integrated version of a white noise. The random walk can be interpreted as a special case of an AR(1) process for  $\phi = 1$ . However, an AR process is usually called as such when it is stationary. Because a white noise can be understood as an ARMA(0,0) process, and because differencing the random walk yields a white noise, we can say that the white noise is an ARIMA(0,1,0) process.

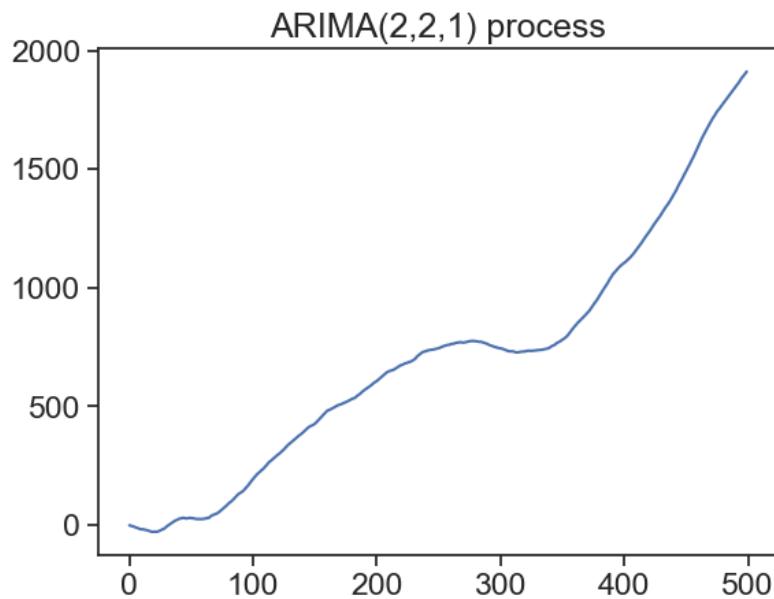
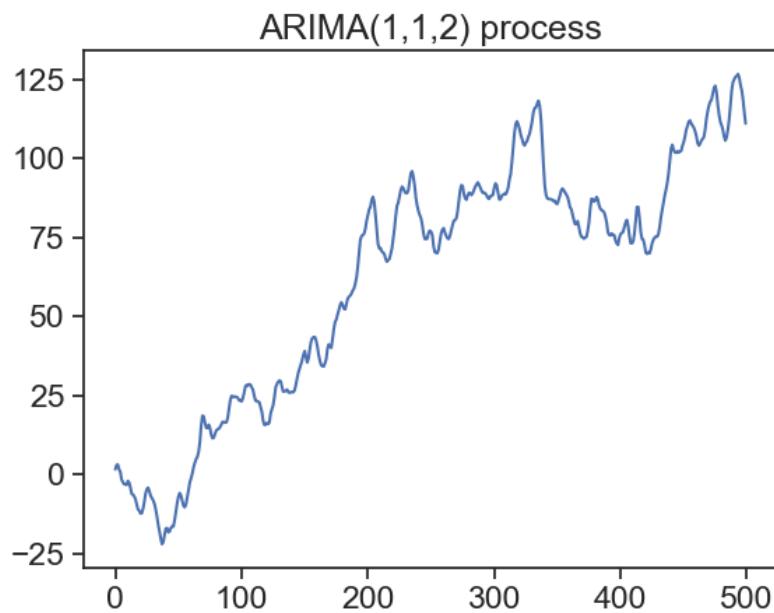
Let's see two examples of ARIMA processes.

```
# Generate ARIMA(2,2,1) process with differencing
arima_221_diff = sm.tsa.arma_generate_sample(ar=[1, -0.18, 0.06], ma=[1, -0.5], nsample=N)
# arima_221 = np.cumsum(arima_221_diff)
arima_221 = np.cumsum(np.cumsum(arima_221_diff)) #

# Plot the ARIMA(1,1,2) process
fig, ax = plt.subplots()
ax.plot(arima_112)
ax.set_title('ARIMA(1,1,2) process')
plt.show()

# Plot the ARIMA(2,2,1) process
fig, ax = plt.subplots()
```

```
ax.plot(arima_221)
ax.set_title('ARIMA(2,2,1) process')
plt.show()
```



We will not fully delve into forecasting right now, but it would be nice to see a real application of ARIMA. If we can reasonably well estimate the parameters associated with a given ARIMA(p,d,q) process, we can use this knowledge to predict future states within a confidence interval. In the simulations below, we see forecasts an ARIMA(2,2,0) process.

```
def arima_forecast(series, ar_coeff):
    s = series.copy()
    phi1 = -ar_coeff[1]
    phi2 = -ar_coeff[2]
    start_index = np.argmax(np.isnan(s))
    for i in np.arange(start_index, len(series)):
        s[i] = phi1 * s[i-1] + phi2 * s[i-2] + np.random.normal()
    return s

np.random.seed(1998)
arima_220_diff = sm.tsa.arma_generate_sample(ar=[1, -0.18, 0.06], ma=[1], nsample=N)
arima_220 = np.cumsum(np.cumsum(arima_220_diff)) #

l = 380
missing = arima_220_diff.copy()
missing[l:] = np.nan

fig, axes = plt.subplots(3, 1, figsize=(8, 8))
fig.subplots_adjust(hspace=0.1)

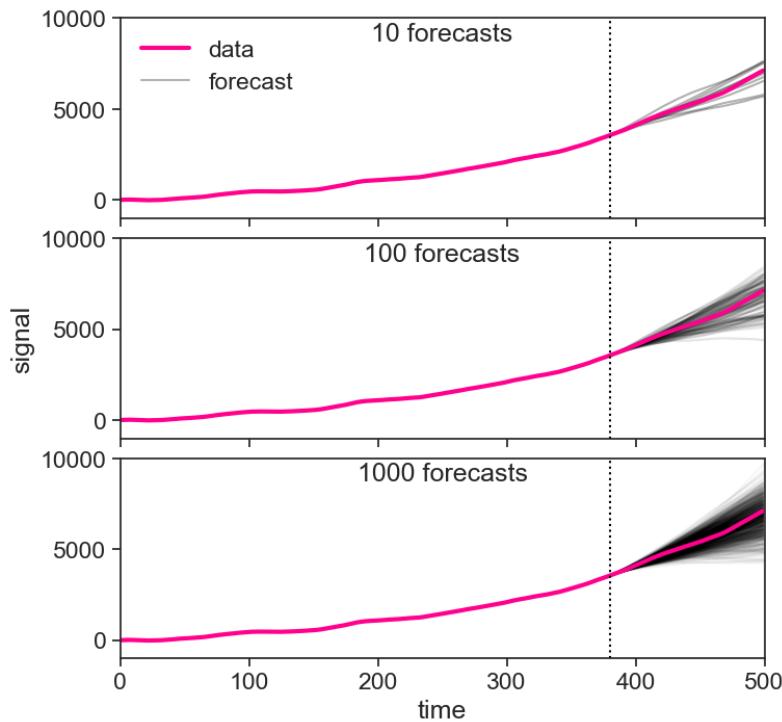
xlim = [0, len(missing)]
ylim = [-1000, 10000]

ax0 = axes[0]
ntries = 10
for i in range(ntries):
    np.random.seed(i)
    try_diff = arima_forecast(missing, [1, -0.18, 0.06])
    t = np.cumsum(np.cumsum(try_diff))
    ax0.plot(t, color="black", alpha=0.3)
ax0.plot(arima_220, color="xkcd:hot pink", lw=3, label="data")
ax0.plot([len(missing), len(missing)+1], [0]*2, color="black", alpha=0.3, label="forecast")
ax0.set(xticklabels=[],
```

```

        xlim=xlim,
        ylim=ylim)
ax0.text(0.5, 0.98, f"{ntries} forecasts",
         transform=ax0.transAxes, ha="center", va="top")
ax0.legend(frameon=False)
ax0.plot([380]*2, ylim, color="black", ls=":")
ax1 = axes[1]
ntries = 100
for i in range(ntries):
    np.random.seed(i)
    try_diff = arima_forecast(missing, [1, -0.18, 0.06])
    t = np.cumsum(np.cumsum(try_diff))
    ax1.plot(t, color="black", alpha=0.1)
ax1.plot(arima_220, color="xkcd:hot pink", lw=3)
ax1.set(xticklabels=[],
        ylabel="signal",
        xlim=xlim,
        ylim=ylim)
ax1.text(0.5, 0.98, f"{ntries} forecasts",
         transform=ax1.transAxes, ha="center", va="top")
ax1.plot([380]*2, ylim, color="black", ls=":")
ax2 = axes[2]
ntries = 1000
for i in range(ntries):
    np.random.seed(i)
    try_diff = arima_forecast(missing, [1, -0.18, 0.06])
    t = np.cumsum(np.cumsum(try_diff))
    ax2.plot(t, color="black", alpha=0.03)
ax2.plot(arima_220, color="xkcd:hot pink", lw=3)
ax2.set(xlabel='time',
        xlim=xlim,
        ylim=ylim)
ax2.text(0.5, 0.98, f"{ntries} forecasts",
         transform=ax2.transAxes, ha="center", va="top")
ax2.plot([380]*2, ylim, color="black", ls=":")

```



We will discuss this later in the course, but estimating the parameters is quite easy:

```
from statsmodels.tsa.arima.model import ARIMA

arima_model = ARIMA(arima_220, order=(2,2,0))
model = arima_model.fit()
print(model.summary())
```

SARIMAX Results			
=====			
Dep. Variable:	y	No. Observations:	500
Model:	ARIMA(2, 2, 0)	Log Likelihood	-703.606
Date:	Tue, 06 Feb 2024	AIC	1413.212
Time:	13:12:24	BIC	1425.844
Sample:	0 - 500	HQIC	1418.170
Covariance Type:	opg		
=====			

	coef	std err	z	P> z	[0.025	0.975]
<hr/>						
ar.L1	0.2171	0.045	4.782	0.000	0.128	0.306
ar.L2	0.0636	0.045	1.401	0.161	-0.025	0.153
sigma2	0.9878	0.072	13.739	0.000	0.847	1.129
<hr/>						
Ljung-Box (L1) (Q):			0.00	Jarque-Bera (JB):		5.16
Prob(Q):			0.99	Prob(JB):		0.08
Heteroskedasticity (H):			1.08	Skew:		0.05
Prob(H) (two-sided):			0.62	Kurtosis:		2.51
<hr/>						

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

## 30 practice

If you don't have it yet, please download here the meteorological data fro 2019 in Jerusalem.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess
np.random.seed(49)

# %matplotlib widget
```

## 31 White noise

If we randomly draw values from the same distribution we will get white noise.

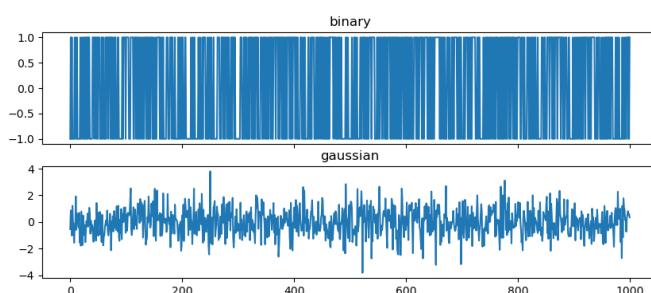
```
# generate binary noise random ones and zeroes with equal distribution
n = 1000
binary_noise = np.random.choice([-1, 1], size=n)

# generate gaussian noise with mean 0 and standard deviation 1
gaussian_noise = np.random.normal(0, 1, n)

# plot:
fig, ax = plt.subplots(2,1, figsize=(10,4), sharex=True)
ax[0].plot(binary_noise)
ax[0].set_title('binary')

ax[1].plot(gaussian_noise)
ax[1].set_title('gaussian')

Text(0.5, 1.0, 'gaussian')
```



## 32 Random walk

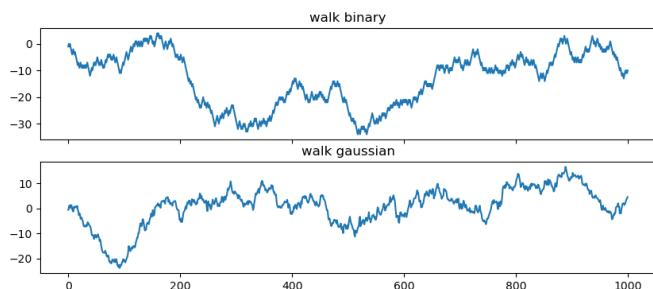
If we cumulatively sum the white noise, we then get a random walk

```
walk_binary = binary_noise.cumsum()
walk_gaussian = gaussian_noise.cumsum()

fig, ax = plt.subplots(2,1, figsize=(10,4), sharex=True)
ax[0].plot(walk_binary)
ax[0].set_title('walk binary')

ax[1].plot(walk_gaussian)
ax[1].set_title('walk gaussian')

Text(0.5, 1.0, 'walk gaussian')
```



### 32.1 Differencing

Given an array

$$a = [a_0, a_1, a_2, \dots, a_{n-1}]$$

the operation performed by `np.diff(a)` can be represented as:

$$\Delta a = [\Delta a_1, \Delta a_2, \dots, \Delta a_{n-1}]$$

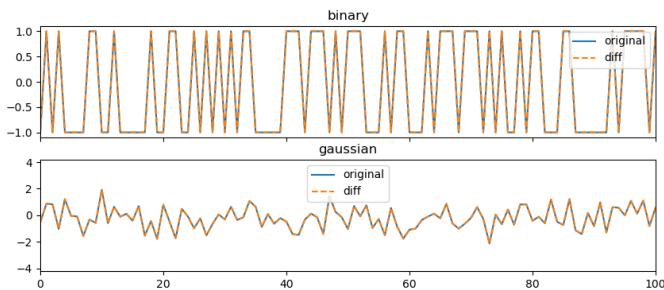
where

$$\Delta a_i = a_i - a_{i-1} \quad \text{for } i = 1, 2, \dots, n-1$$

If we difference the random walk we will get the white noise.

```
fig, ax = plt.subplots(2,1, figsize=(10,4), sharex=True)
ax[0].plot(binary_noise, label='original')
ax[0].plot(np.diff(walk_binary, prepend=0), label='diff', linestyle='--')
ax[0].set_title('binary')
ax[0].legend()
ax[0].set_xlim(0,100)

ax[1].plot(gaussian_noise, label='original')
ax[1].plot(np.diff(walk_gaussian, prepend=0), label='diff', linestyle='--')
ax[1].set_title('gaussian')
ax[1].legend()
ax[1].set_xlim(0,100)
```



Another way of understanding this: the python operations `cumsum` and `diff` are each other's inverse.

## 33 AR(1)

$$X_t = \phi X_{t-1} + \varepsilon.$$

This is called an Autoregressive Process of order 1, or AR(1). Here, the current value  $X_t$  is dependent on the immediately preceding value  $X_{t-1}$ .

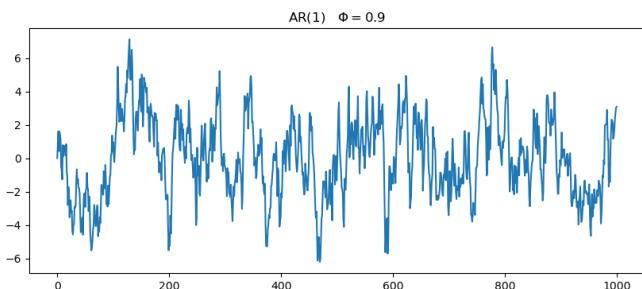
```
# initialize time series array
ar1_series = np.zeros(n)

# set a phi value, in addition to this value you should try phi>1 or phi=0
phi = 0.9

for i in range(1, n):
    ar1_series[i] = phi*ar1_series[i-1] + gaussian_noise[i]

# plot:
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(ar1_series)
ax.set_title(f'AR(1)\t$\Phi={phi}$')
```

Text(0.5, 1.0, 'AR(1)\t\$\Phi=0.9\$')



### 33.1 AR( $p$ )

The next thing to do is to generalize, and define an autoregressive process that depends on  $p$  previous states:

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + \varepsilon$$

```
# Function to generate AR(p) time series
# this function can receive p as an integer and then it will draw random phi values
# or, you can pass p as a np array of the specific phi values you want.
def generate_ar(n, p):
    # Check if p is an integer or an array
    if isinstance(p, int):
        # Generate random coefficients between -1 and 1
        phi = np.random.uniform(-1, 1, size=p)
    elif isinstance(p, np.ndarray):
        phi = p # Use the provided array as coefficients
    else:
        raise ValueError("p should be either an integer or a NumPy array")

    print(phi)
    # Generate white noise
    noise = np.random.normal(0, 1, n)

    # Initialize time series array
    ar_series = np.zeros(n)

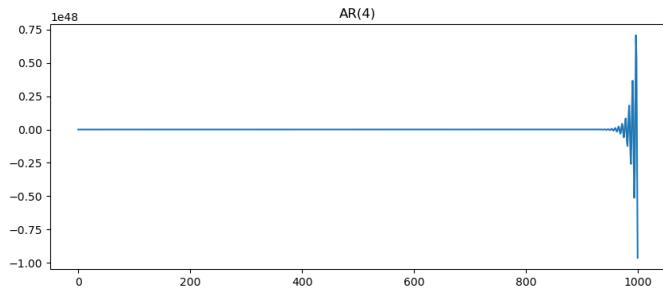
    for i in range(phi.size, n):
        ar_series[i] = np.dot(phi, ar_series[i-phi.size:i]) + noise[i]

    return ar_series

# plot using p as an int
p = 4
ar = generate_ar(n, p)
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(ar)
ax.set_title(f'AR({p})')
```

```
[-0.82679831 -0.50310415 -0.68089179  0.1555622 ]
```

```
Text(0.5, 1.0, 'AR(4)')
```



### 33.1.1 using specific $\phi$ values

In the cell below we can specify specific  $\phi$  values.  
Use the [interactive tool](#) from our website to chose the right values.  
Remember, if one of the roots is inside the unit circle, the series will be **not** stationary.

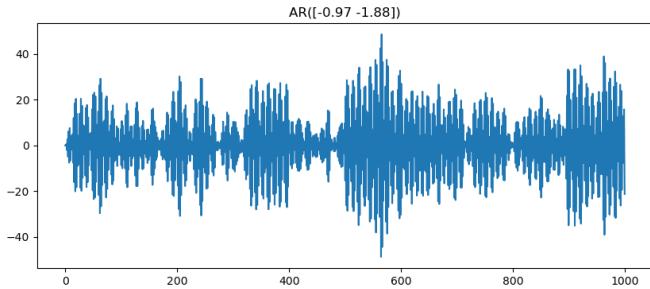
```
# plot using p as an array of phi values
# the order should be [phi2, phi1]
p = np.array([-0.97,-1.88])
# p = np.array([-1.88,-0.97])

ar2 = generate_ar(n, p)
fig, ax = plt.subplots(figsize=(10,4))

ax.plot(ar2)
ax.set_title(f'AR({p})')
```

```
[-0.97 -1.88]
```

```
Text(0.5, 1.0, 'AR([-0.97 -1.88])')
```



### 33.1.2 Weak stationarity

1. its mean  $\mu$  does not vary in time:

$$\mu_X(t) = \mu_X(t + \tau)$$

for all values of  $t$  and  $\tau$ .

2. its variance is finite for any time  $t$ :

$$\sigma_X^2(t) < \infty.$$

3. The autocorrelation function between two lagged versions of the same time series,  $X(t_1)$  and  $X(t_2)$ , depends only on the difference  $\tau = t_2 - t_1$ .

Let's get a feeling by plotting

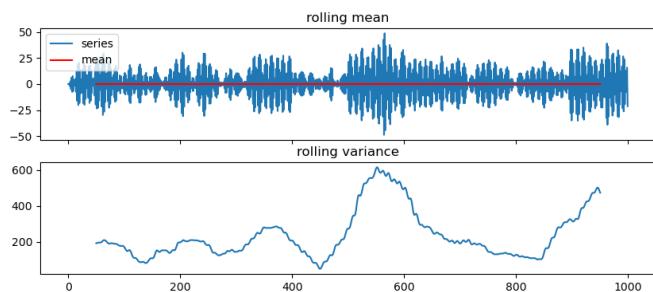
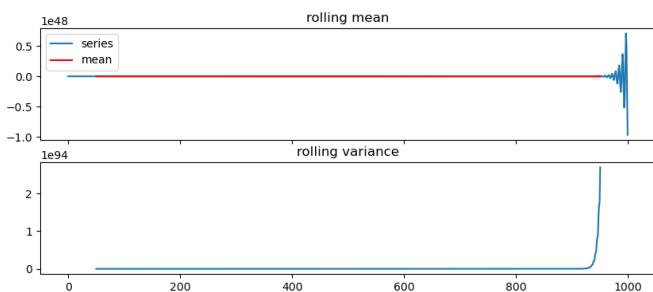
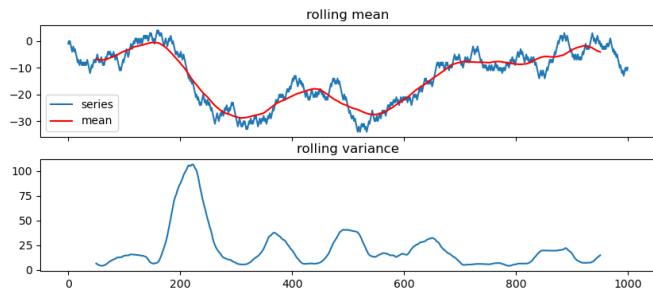
```
def test_stationarity(time_series, window=100):
    series = pd.Series(time_series)
    rolling_var = series.rolling(window=window, center=True).std()**2
    rolling_mean = series.rolling(window=window, center=True).mean()

    fig, ax = plt.subplots(2,1, figsize=(10,4), sharex=True)
    ax[0].plot(series, label='series')
    ax[0].plot(rolling_mean, c='r', label='mean')
    ax[0].legend()
    ax[0].set_title('rolling mean')

    ax[1].plot(rolling_var)
    ax[1].set_title('rolling variance')

    return
```

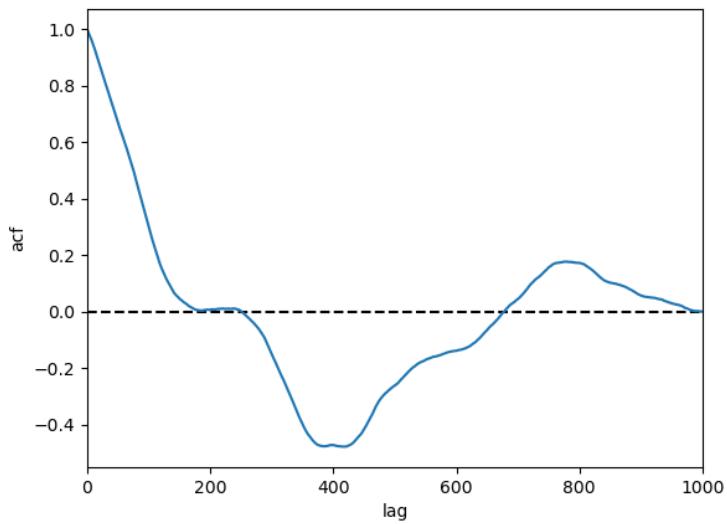
```
test_stationarity(walk_binary)
# test_stationarity(walk_gaussian)
# test_stationarity(ar1_series)
test_stationarity(ar)
test_stationarity(ar2)
```



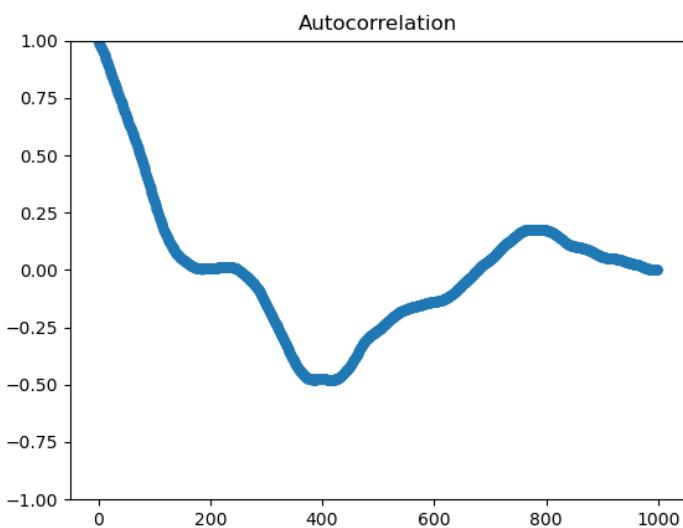
## 34 ACF

```
def compute_acf(series):
    N = len(series)
    lags = np.arange(N)
    acf = np.zeros_like(lags)
    series = (series - series.mean()) / series.std()
    for i in lags:
        acf[i] = np.sum(series[i:] * series[:N-i])
    acf = acf / N
    return lags, acf
```

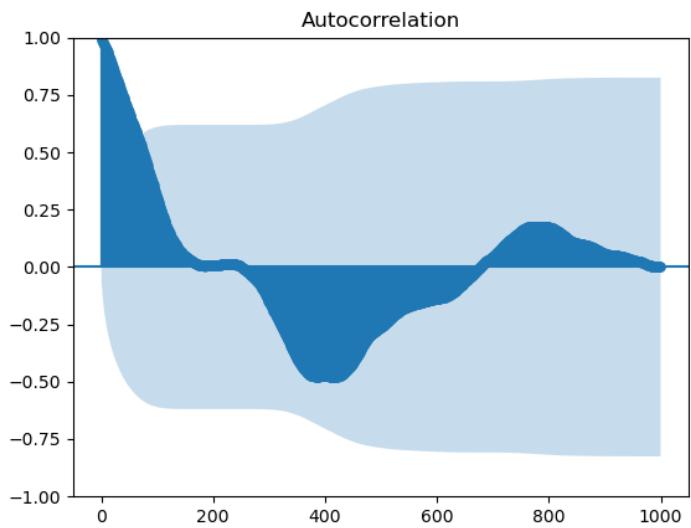
```
# walk_binary, walk_gaussian, ar1_series, ar, ar2
series_to_plot = walk_binary
fig, ax = plt.subplots()
lags, acf = compute_acf(series_to_plot)
ax.plot([0, n], [0]*2, color="black", ls="--")
ax.plot(lags, acf)
ax.set(xlabel="lag",
       ylabel="acf",
       xlim=[0, n]);
```



```
fig, ax = plt.subplots()
sm.graphics.tsa.plot_acf(series_to_plot, lags= n-1, ax=ax, label="statsmodels", alpha=None, use
```

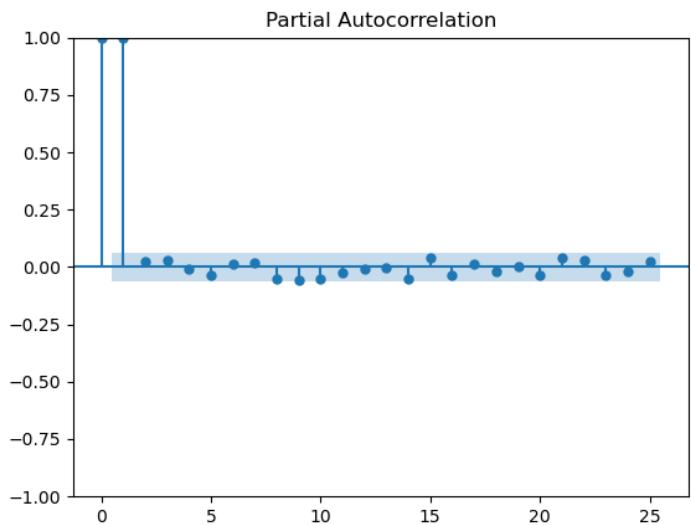


```
fig, ax = plt.subplots()
sm.graphics.tsa.plot_acf(series_to_plot, lags= n-1, ax=ax, alpha=.05);
```



```
fig, ax = plt.subplots()  
sm.graphics.tsa.plot_pacf(series_to_plot, lags=25, ax=ax, alpha=.05);
```

/opt/anaconda3/lib/python3.9/site-packages/statsmodels/graphics/tsaplots.py:348: FutureWarning  
warnings.warn(

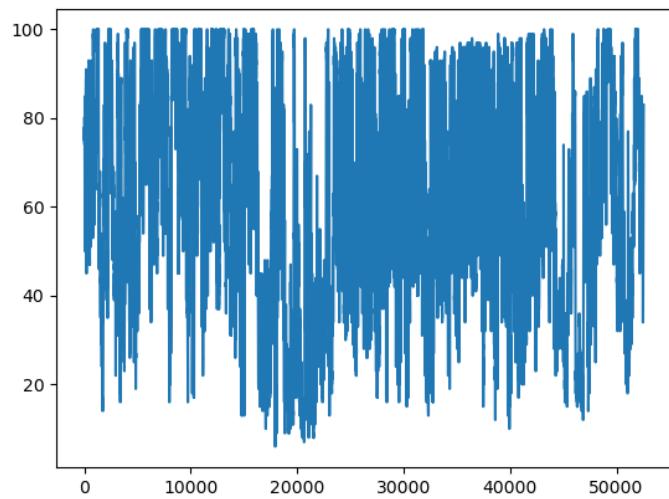


## 34.1 Now let's work with actual data

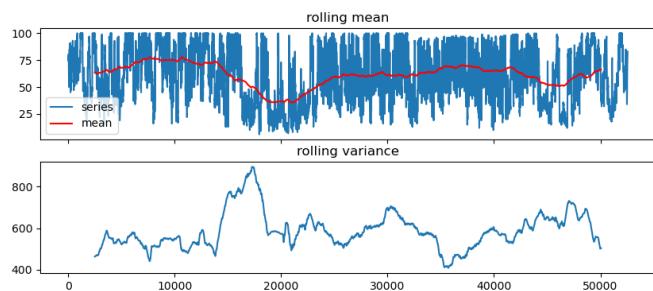
```
filename = "jerusalem2019.csv"
df = pd.read_csv(filename, na_values=['-'])
df.rename(columns={'Temperature (°C)': 'temperature',
                   'Rainfall (mm)': 'rain'}, inplace=True)
df['date'] = pd.to_datetime(df['Date & Time (Winter)'], dayfirst=True)
df = df.set_index('date')
df = df.fillna(method='ffill')
df
```

date	Station	Date & Time (Winter)	Diffused radiation (W/m <sup>2</sup> )	Global rad
2019-01-01 00:00:00	Jerusalem Givat Ram	01/01/2019 00:00	0.0	0.0
2019-01-01 00:10:00	Jerusalem Givat Ram	01/01/2019 00:10	0.0	0.0
2019-01-01 00:20:00	Jerusalem Givat Ram	01/01/2019 00:20	0.0	0.0
2019-01-01 00:30:00	Jerusalem Givat Ram	01/01/2019 00:30	0.0	0.0
2019-01-01 00:40:00	Jerusalem Givat Ram	01/01/2019 00:40	0.0	0.0
...	...	...	...	...
2019-12-31 22:20:00	Jerusalem Givat Ram	31/12/2019 22:20	0.0	0.0
2019-12-31 22:30:00	Jerusalem Givat Ram	31/12/2019 22:30	0.0	0.0
2019-12-31 22:40:00	Jerusalem Givat Ram	31/12/2019 22:40	0.0	0.0
2019-12-31 22:50:00	Jerusalem Givat Ram	31/12/2019 22:50	0.0	0.0
2019-12-31 23:00:00	Jerusalem Givat Ram	31/12/2019 23:00	0.0	0.0

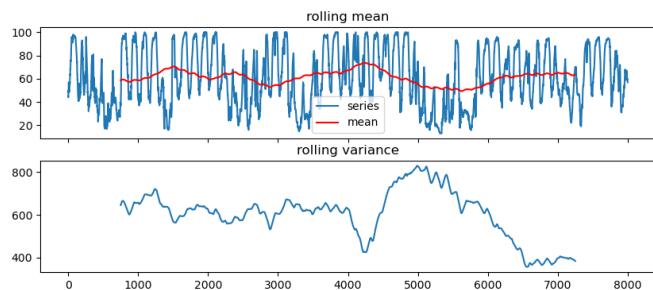
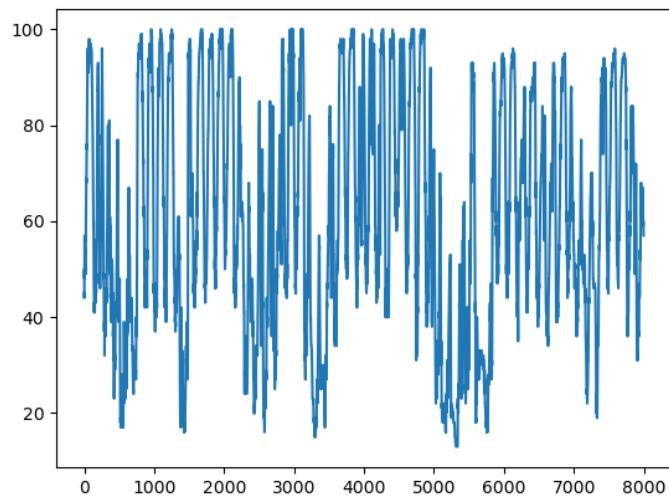
```
# t = df['temperature'].values
t = df['Relative humidity (%)'].values
# t = df['Wind speed (m/s)'].values
# t = df['Wind direction (°)'].values
fig, ax = plt.subplots()
ax.plot(t)
```



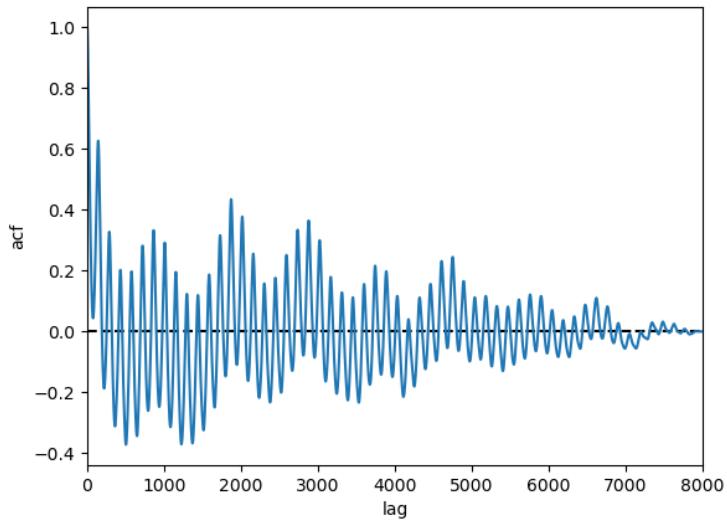
```
test_stationarity(t, window=5000)
```



```
t_stationary = t[27000:35000]
fig, ax = plt.subplots()
ax.plot(t_stationary)
test_stationarity(t_stationary, window=1500)
```

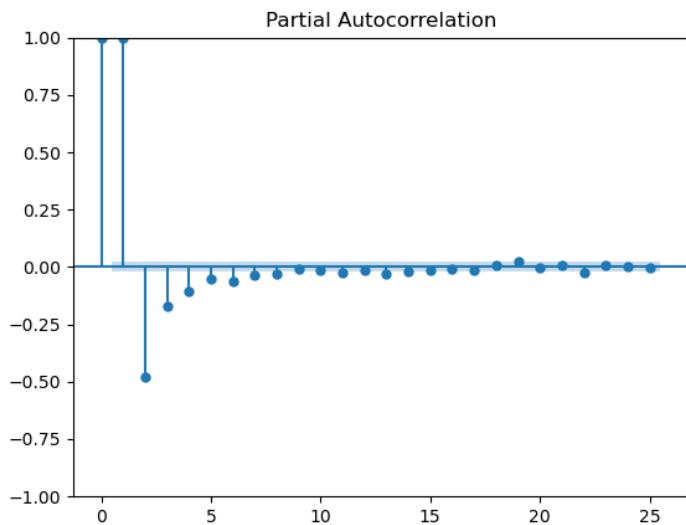


```
series_to_plot = t_stationary
fig, ax = plt.subplots()
lags, acf = compute_acf(series_to_plot)
ax.plot([0, len(series_to_plot)], [0]*2, color="black", ls="--")
ax.plot(lags, acf)
ax.set(xlabel="lag",
       ylabel="acf",
       xlim=[0, len(series_to_plot)]);
```



```
fig, ax = plt.subplots()
sm.graphics.tsa.plot_pacf(series_to_plot, lags=25, ax=ax, alpha=.05);
```

/opt/anaconda3/lib/python3.9/site-packages/statsmodels/graphics/tsaplots.py:348: FutureWarning  
warnings.warn(



## 35 filling missing values

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import random

# %matplotlib widget

def plot_all_columns(data):
    column_list = data.columns

    fig, ax = plt.subplots(len(column_list),1, sharex=True, figsize=(10,len(column_list)*2))

    if len(column_list) == 1:
        ax.plot(data[column_list[0]])
        return
    for i, column in enumerate(column_list):
        ax[i].plot(data[column])
        ax[i].set(ylabel=column)
```

```

locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
formatter = mdates.ConciseDateFormatter(locator)
ax[i].xaxis.set_major_locator(locator)
ax[i].xaxis.set_major_formatter(formatter)

return

def plot_missing_vals(original, new, title=''):
    column_list = original.columns

    fig, ax = plt.subplots(len(column_list), 1, sharex=True, figsize=(10, len(column_list)*2))

    if len(column_list) == 1:
        ax.set_title(title)
        ax.plot(new[column_list[0]], c='r')
        ax.plot(original[column_list[0]])
        return
    for i, column in enumerate(column_list):
        ax[i].plot(original[column_list[0]], c='tab:blue', alpha=0.2)
        ax[i].plot(new[column], c='r')
        ax[i].plot(original[column])
        if i != 0:
            ax[i].scatter(new[column].index, new[column].values, marker='o', s=4, c='r')
            ax[i].scatter(original[column].index, original[column].values, marker='o', s=4)
        else:
            ax[i].set_title(title)
        ax[i].set(ylabel=column)
        # calculate and display MSE
        mse = mean_squared_error(original[column_list[0]], new[column])
        ax[i].text(0.01, 0.95, f'MSE: {mse:.4f}', transform=ax[i].transAxes, fontsize=12, verti
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax[i].xaxis.set_major_locator(locator)
    ax[i].xaxis.set_major_formatter(formatter)

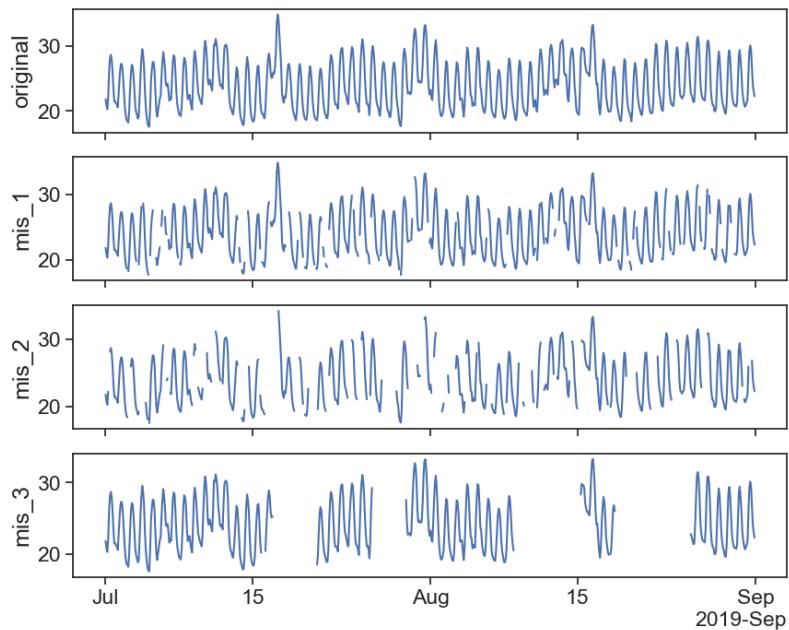
return

```

```
df = pd.read_csv('data_missing.csv', index_col='date', parse_dates=True)
df
```

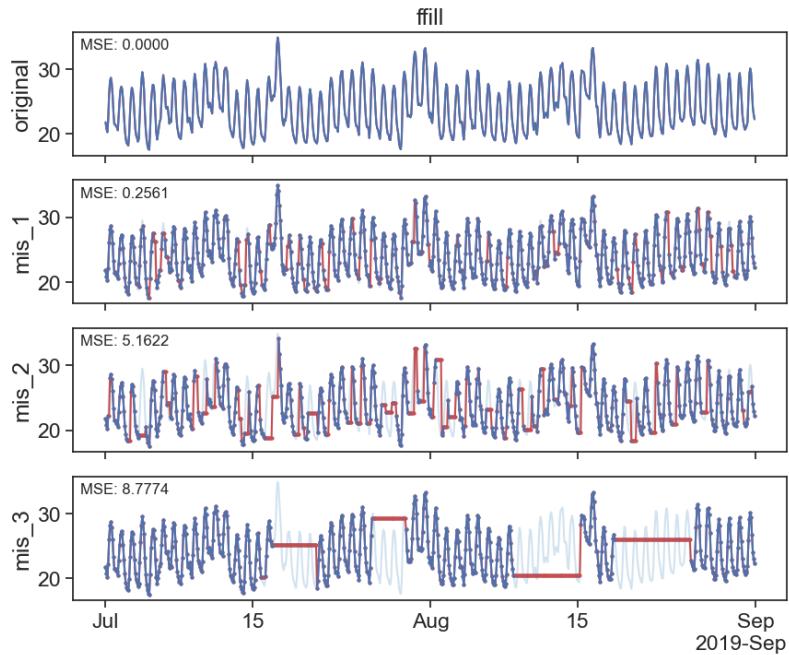
	original	mis_1	mis_2	mis_3
date				
2019-07-01 00:00:00	21.808333	21.808333	21.808333	21.808333
2019-07-01 02:00:00	20.808333	20.808333	20.808333	20.808333
2019-07-01 04:00:00	20.283333	20.283333	20.283333	20.283333
2019-07-01 06:00:00	22.216667	22.216667	22.216667	22.216667
2019-07-01 08:00:00	26.091667	26.091667	NaN	26.091667
...	...	...	...	...
2019-08-31 14:00:00	29.400000	29.400000	NaN	29.400000
2019-08-31 16:00:00	26.808333	26.808333	26.808333	26.808333
2019-08-31 18:00:00	24.050000	24.050000	24.050000	24.050000
2019-08-31 20:00:00	23.008333	23.008333	23.008333	23.008333
2019-08-31 22:00:00	22.275000	22.275000	22.275000	22.275000

```
plot_all_columns(df)
```



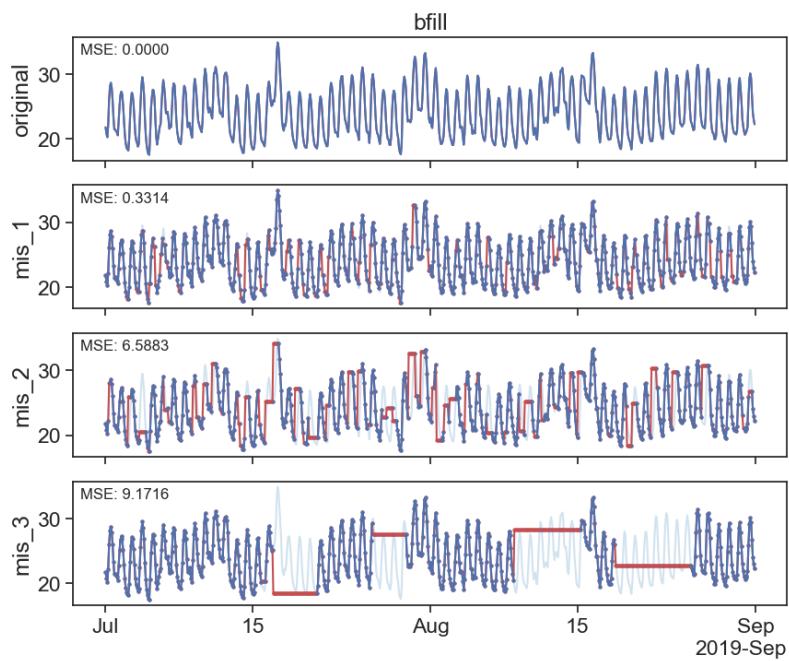
## 35.1 Forward fill

```
df_ffil = df.fillna(method='ffill')
plot_missing_vals(df, df_ffil, title='ffill')
```



## 35.2 Backwrds fill

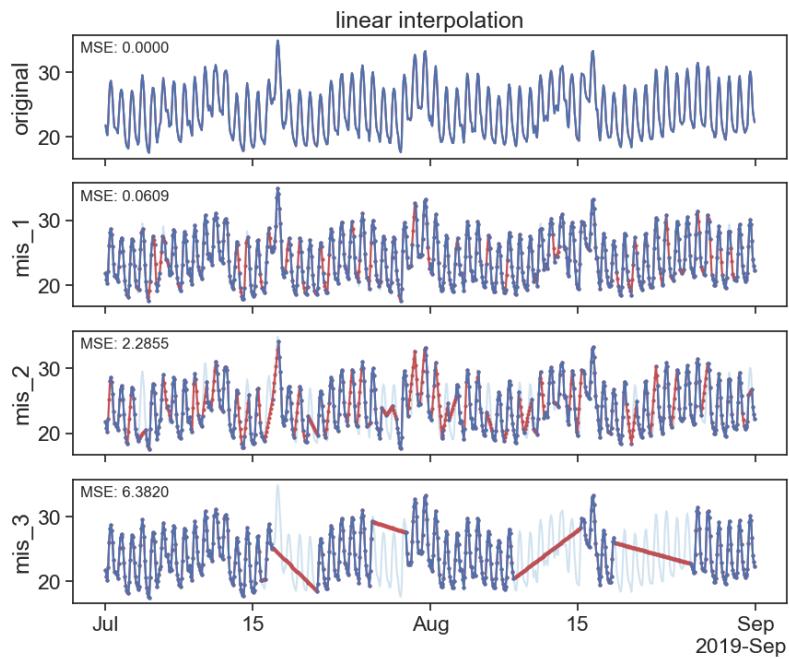
```
df_bfil = df.fillna(method='bfill')
plot_missing_vals(df, df_bfil, title='bfill')
```



### 35.3 Interpolation

#### 35.3.1 linear

```
interpolated_df = df.interpolate(method='linear')
plot_missing_vals(df, interpolated_df, title='linear interpolation')
```

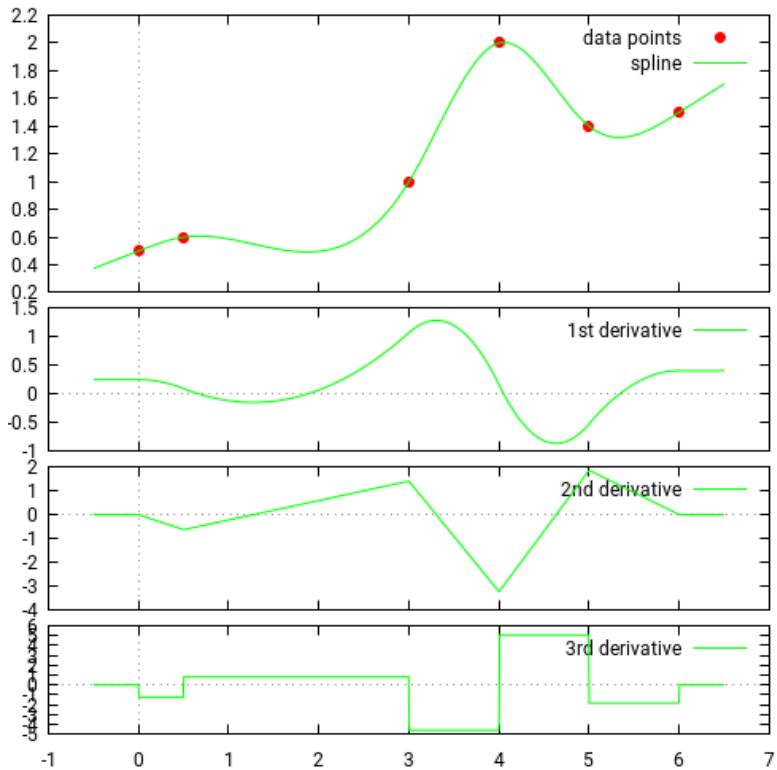


<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html>

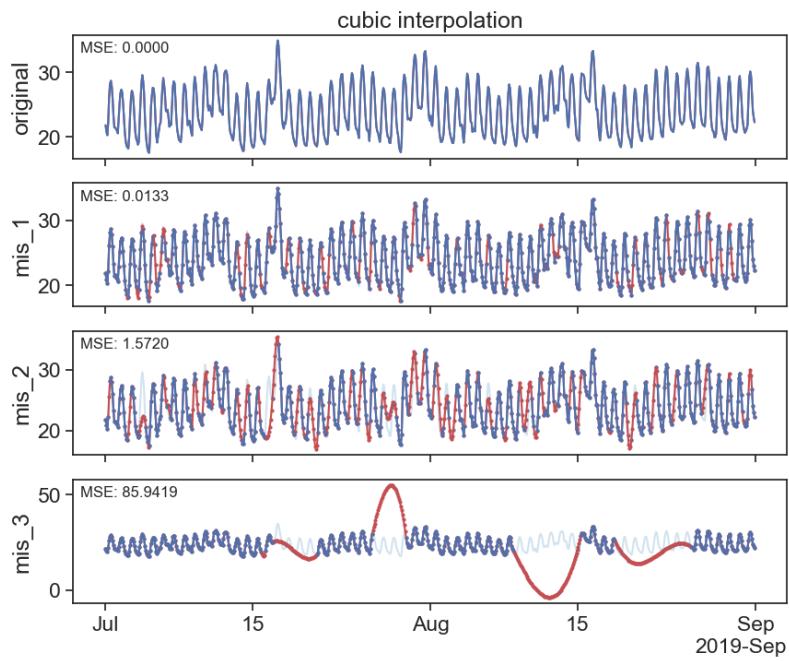
### 35.3.2 cubic splines

Source: <https://docs.scipy.org/doc/scipy/tutorial/interpolate/1D.html#cubic-splines>

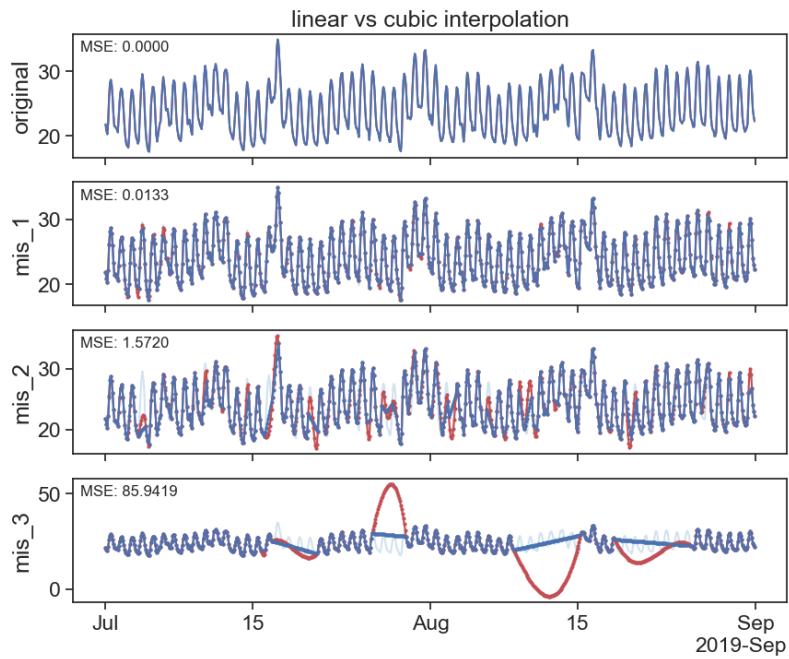
Piecewise linear interpolation produces corners at data points, where linear pieces join. To produce a smoother curve, you can use cubic splines, where the interpolating curve is made of cubic pieces with **matching first and second derivatives**.



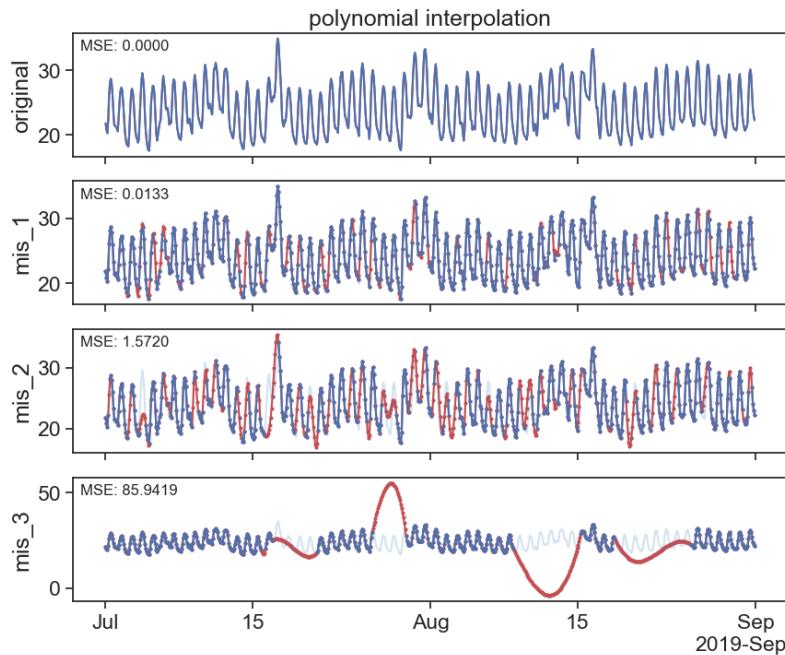
```
interpolated_cubic_df = df.interpolate(method='cubic') Source: https://kluge.in-chemnitz.de/opensource/spline/
plot_missing_vals(df, interpolated_cubic_df, title='cubic interpolation')
```



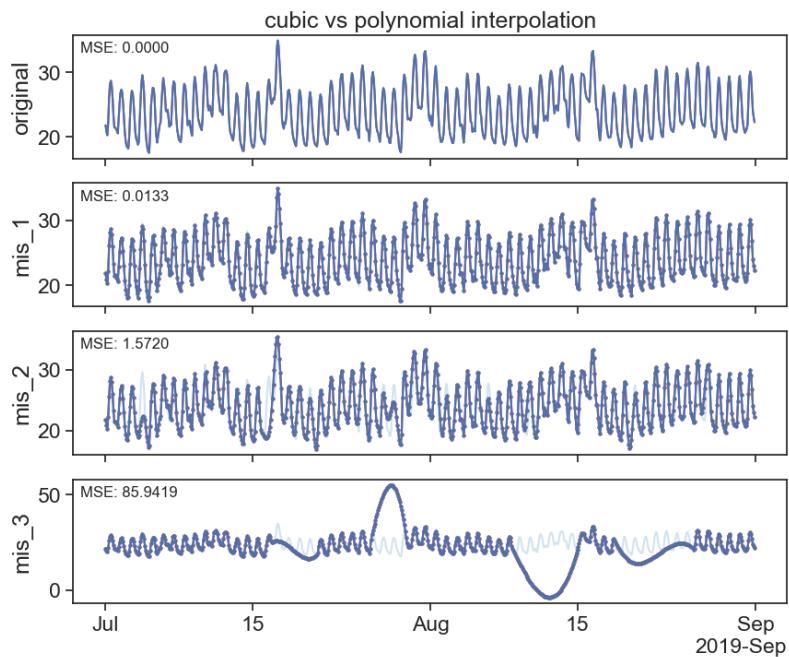
```
plot_missing_vals(interpolated_df, interpolated_cubic_df, title='linear vs cubic interpolation')
```



```
interpolated_poly_df = df.interpolate(method='polynomial', order=3)
plot_missing_vals(df, interpolated_poly_df, title='polynomial interpolation')
```



```
plot_missing_vals(interpolated_cubic_df, interpolated_poly_df, title='cubic vs polynomial interpolation')
```



All available interpolation types can be found at the [pandas documentation](#)

## 35.4 random forest

Here we explore the use of a commonly used Machine learning model - RandomForest

```
def fillna_randomforest(series):
    # Ensure the series index is a datetime object
    if not isinstance(series.index, pd.DatetimeIndex):
        raise ValueError("Index must be a DatetimeIndex")

    # Split series into observed and missing values
    observed = series.dropna()
    missing = series[series.isna()]

    # Extracting time-based features
    def create_features(index):
        return np.array([index.hour, index.day, index.month, index.year]).T
```

```

# Create features for training data
X_train = create_features(observed.index)
y_train = observed.values

# Train the Random Forest regression model
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Create features for missing data and predict
X_missing = create_features(missing.index)
predicted_values = model.predict(X_missing)

# Assign the predicted values to the missing positions
series_filled = series.copy()
series_filled[series_filled.isna()] = predicted_values

return series_filled

```

```

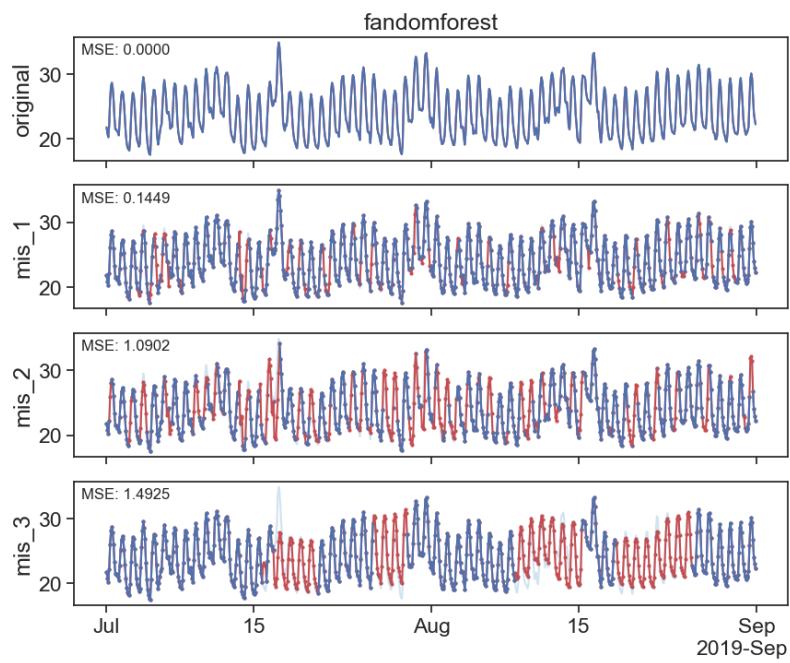
columns = df.columns
df_rf = df.copy()
for i, column in enumerate(columns):
    if i == 0:
        continue
    df_rf[column] = fillna_randomforest(df_rf[column])

```

```

plot_missing_vals(df, df_rf, title='fandomforest')

```



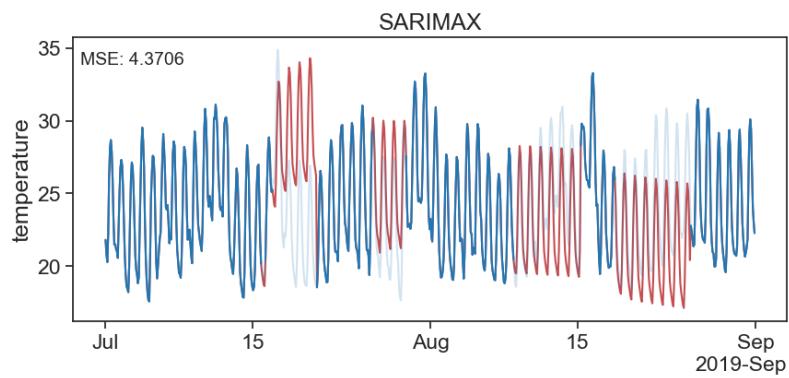
## 36 SARIMAX

```
# Model Building and Imputation
# Build a SARIMA model on the data with gaps
seasonal_period = 24/2
model = SARIMAX(df['mis_3'], order=(2, 1, 0), seasonal_order=(1, 1, 1, seasonal_period), enforce_invertibility=True)
results = model.fit(disp=False)

# Impute missing values one by one
ts_data_filled = df['mis_3'].copy()
missing_indices = df['mis_3'].loc[df['mis_3'].isna()]
for missing_index in missing_indices.index:
    ts_data_filled[missing_index] = results.predict(start=missing_index, end=missing_index)

fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(ts_data_filled, label='Imputed Data', color='r')
ax.plot(df['original'], label='Original Data', color='tab:blue', alpha=0.2)
ax.plot(df['mis_3'], label='Data with Gaps', color='tab:blue', alpha=1)
ax.set_ylabel('temperature')
ax.set_title('SARIMAX')
# calculate and display MSE
mse = mean_squared_error(df['original'], ts_data_filled)
ax.text(0.01, 0.95, f'MSE: {mse:.4f}', transform=ax.transAxes, fontsize=14, verticalalignment='top')
# ax.legend()

locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)
```



The above looks good at the begining of the gap but at the end of the gap it looks bad. That is because we are forcasting and not filling in between the lines...

## 37 SARIMAX cross fade

```
def find_nan_gaps_indexes_datetime(series):
    """
    Find and pair the start and end datetime indexes of gaps in a pandas Series with a datetime index.

    Parameters:
    series (pandas Series): The input pandas Series with a datetime index containing NaN gaps.

    Returns:
    list of tuples: A list of tuples where each tuple contains the start datetime index and end datetime index of a gap.
    """
    is_nan = series.isna().values
    start_indexes = np.where(is_nan & ~np.roll(is_nan, 1))[0]
    end_indexes = np.where(is_nan & ~np.roll(is_nan, -1))[0]

    # If the last gap extends to the end of the Series, add its end index
    if is_nan[-1]:
        end_indexes = np.append(end_indexes, len(series) - 1)

    # Pair start and end datetime indexes together
    gap_pairs = [(series.index[start], series.index[end]) for start, end in zip(start_indexes, end_indexes)]

    return gap_pairs

def forward_reverse_SARIMAX(ts_data, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12), seasonal_periods=12, n_imputations=5, alpha=0.5):
    """
    The function forward_reverse_SARIMAX aims to impute missing values in a time series dataset using a
    SARIMAX model approach, both in the original and reversed order of the data. Initially, it fits a
    SARIMAX model on the provided dataset to predict and fill in the missing values. Then, it applies the
    same model to the time series data, applies another SARIMAX model on this reversed data, and imputes the
    missing values again. After obtaining the imputed datasets from both forward and reverse directions,
    the function iteratively blends the imputed values from both models for each missing point, adjusting
    the weight given to the forward and reverse imputations based on the position within the sequence.
    This blended approach aims to leverage the predictive insights from both the preceding and
    following observations.
    """
    # Fit initial SARIMAX model
    model = SARIMAX(ts_data, order=order, seasonal_order=seasonal_order, seasonal_periods=seasonal_periods)
    model_fit = model.fit()
    ts_imputed = model_fit.predict()

    # Reverse the data
    ts_reversed = ts_data[::-1]

    # Fit second SARIMAX model on reversed data
    model_reversed = SARIMAX(ts_reversed, order=order, seasonal_order=seasonal_order, seasonal_periods=seasonal_periods)
    model_fit_reversed = model_reversed.fit()
    ts_imputed_reversed = model_fit_reversed.predict()

    # Blend the imputed values
    for i in range(n_imputations):
        ts_imputed = alpha * ts_imputed + (1 - alpha) * ts_imputed_reversed
        ts_imputed_reversed = alpha * ts_imputed_reversed + (1 - alpha) * ts_imputed

    # Reverse the blended imputation back to the original order
    ts_imputed = ts_imputed[::-1]

    return ts_imputed
```

```

# data points, potentially providing a more accurate and balanced imputation for the missing

# Model Building and Imputation
# Build a SARIMA model on the data with gaps
model = SARIMAX(ts_data, order=order, seasonal_order=seasonal_order, enforce_stationarity=False)
results = model.fit(disp=False)

# Impute missing values one by one
ts_data_filled = ts_data.copy()
missing_indices = ts_data.loc[ts_data.isna()]
for missing_index in missing_indices.index:
    ts_data_filled[missing_index] = results.predict(start=missing_index, end=missing_index)

# Reverse the time series
ts_data_reversed = ts_data.iloc[::-1]

# Build SARIMAX model on reversed data
model_reversed = SARIMAX(ts_data_reversed, order=order, seasonal_order=seasonal_order, enforce_stationarity=False)
results_reversed = model_reversed.fit(disp=False)

# Impute missing values in reversed data
ts_data_filled_reversed = ts_data_reversed.copy()
missing_indices_reversed = ts_data_reversed.loc[ts_data_reversed.isna()]
for missing_index in missing_indices_reversed.index:
    ts_data_filled_reversed[missing_index] = results_reversed.predict(start=missing_index, end=missing_index)

# Reverse the imputed data back to original order
ts_data_filled_reversed = ts_data_filled_reversed.iloc[::-1]

# Initialize a series to hold the combined predictions
ts_data_combined = ts_data.copy()

# Iterate over each gap
for gap_start, gap_end in find_nan_gaps_indexes_datetime(ts_data):
    # print(gap_start)
    # Calculate the number of periods in the gap
    gap = ts_data[gap_start:gap_end]
    gap_length = len(gap)

```

```

# Iterate over each index in the gap
# for i, index in enumerate(pd.date_range(start=gap_start, end=gap_end, freq=ts_data.index.freq)):
for i, index in enumerate(gap.index):
    forward_weight = (gap_length - i) / gap_length
    backward_weight = i / gap_length
    combined_prediction = (ts_data_filled.at[index] * forward_weight +
                           ts_data_filled_reversed.at[index] * backward_weight)
    ts_data_combined.at[index] = combined_prediction

return ts_data_combined

```

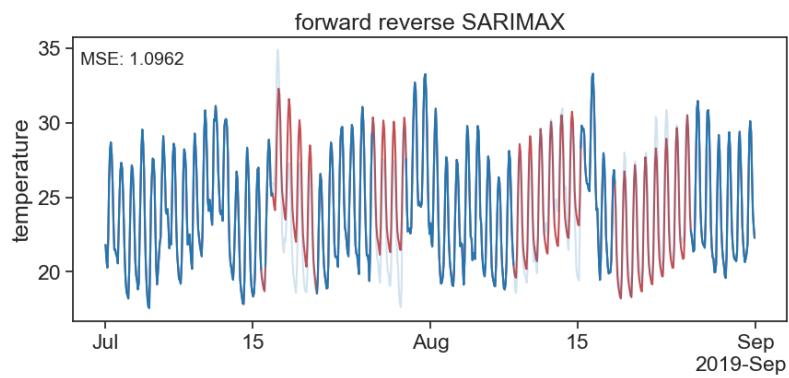
`f_r_SARIMAX = forward_reverse_SARIMAX(df['mis_3'])`

```

fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(f_r_SARIMAX, label='Imputed Data', color='r')
ax.plot(df['original'], label='Original Data', color='tab:blue', alpha=0.2)
ax.plot(df['mis_3'], label='Data with Gaps', color='tab:blue', alpha=1)
ax.set_ylabel('temperature')
ax.set_title('forward reverse SARIMAX')
# calculate and display MSE
mse = mean_squared_error(df['original'], f_r_SARIMAX)
ax.text(0.01, 0.95, f'MSE: {mse:.4f}', transform=ax.transAxes, fontsize=14, verticalalignment='bottom')
# ax.legend()

locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

```



```
# Calculate MSE
mse_fr = mean_squared_error(df['original'], f_r_SARIMAX)
mse_rf = mean_squared_error(df['original'], df_rf['mis_3'])

print("Mean Squared Error sarimax:", mse_fr)
print("Mean Squared Error rand:", mse_rf)
```

Mean Squared Error sarimax: 1.0961749546737887  
Mean Squared Error rand: 1.4924638305799802

# **Part VI**

## **seasonality**

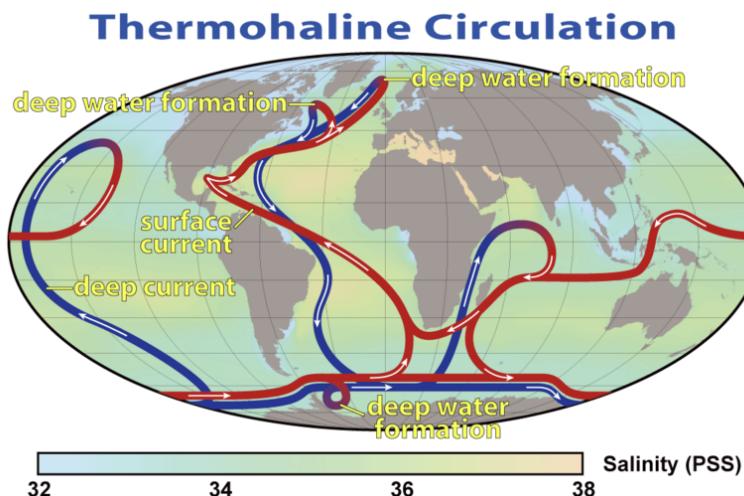
## 38 motivation

This is a bit grim for *motivation*, but it is super important.

This is mind-blowingly bad. Most up-to-date study on Atlantic Ocean currents system shows signals of collapse. This could happen in 2025-2095 with 95% probability. Land to grow wheat may drop by half. Winter temperatures in North Europe could drop between 10 and 30 °C. 1/ <https://t.co/W1uPEEWus3pic.twitter.com/yUKoae9b6S>

— dr. gianluca grimalda (@ GGrimalda) February 10, 2024

Let's discuss what's going on by looking at [these interactive graphs](#).



From Westen, Kliphuis, and Dijkstra (2024):

Classical early warning indicators, such as the increase in the variance and/or the (lag-1) autocorrelation, when applied to sea surface temperature-based time series, suggest that the

Source: [Wikimedia](#)

present-day Atlantic meridional overturning circulation approaches a tipping point before the end of this century (11, 12).

### 38.1 questions

- disconsidering the expected annual cycle, how fast is the ocean warming up?
- what is a *typical* annual oscillation in temperature?
- how can I model processes such as this, where there are both a clear trend and a seasonal component?

# 39 seasonal decomposition from scratch

We will work together, running each cell in this Notebook step by step. Start by downloading the notebook (“Code” button above), and also download these 2 csv files:

- sst\_daily\_world.csv
- sst\_daily\_north.csv

## 39.1 sea surface temperature

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from datetime import datetime as dt
import time
from statsmodels.tsa.stattools import adfuller

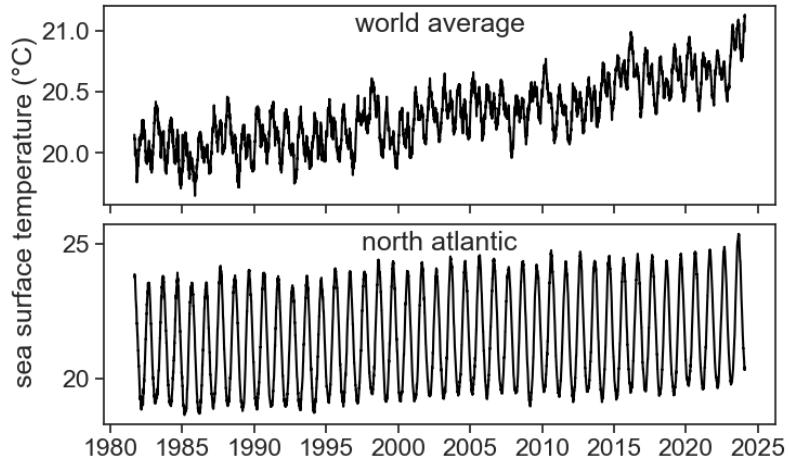
# %matplotlib widget

df_north = pd.read_csv("sst_daily_north_atl.csv", index_col='date', parse_dates=True)
df_world = pd.read_csv("sst_daily_world.csv", index_col='date', parse_dates=True)
```

```

fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True)
fig.subplots_adjust(hspace=0.1) # increase vertical space between panels
ax[0].plot(df_world['sst'], color="black")
ax[1].plot(df_north['sst'], color="black")
fig.text(0.02, 0.5, 'sea surface temperature (°C)', va='center', rotation='vertical')
ax[1].set(yticks=[20, 25])
ax[0].text(0.5, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='center', verticalalignment='top',)
ax[1].text(0.5, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='center', verticalalignment='top',)
pass

```



## 39.2 trend

```

df_north['trend'] = df_north['sst'].rolling('365D', center=True).mean()
df_world['trend'] = df_world['sst'].rolling('365D', center=True).mean()

```

```

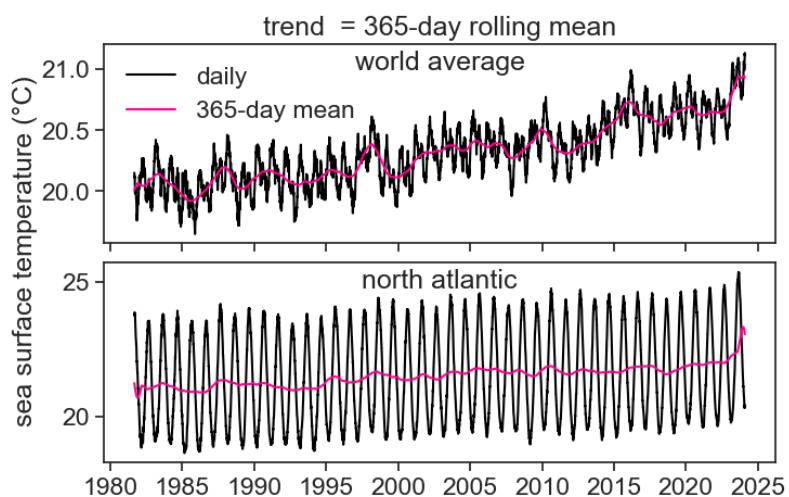
fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True)
fig.subplots_adjust(hspace=0.1) # increase vertical space between panels
ax[0].plot(df_world['sst'], color="black", label="daily")
ax[0].plot(df_world['trend'], color="xkcd:hot pink", label="365-day mean")
ax[1].plot(df_north['sst'], color="black")
ax[1].plot(df_north['trend'], color="xkcd:hot pink")

```

```

fig.text(0.02, 0.5, 'sea surface temperature (°C)', va='center', rotation='vertical')
ax[0].set(title="trend = 365-day rolling mean")
ax[1].set(yticks=[20, 25])
ax[0].text(0.5, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='center', verticalalignment='top',)
ax[1].text(0.5, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='center', verticalalignment='top',)
ax[0].legend(frameon=False)
pass

```



### 39.3 detrend

$$\text{detrended} = \text{signal} - \text{trend}$$

```

df_north['detrended'] = df_north['sst'] - df_north['trend']
df_world['detrended'] = df_world['sst'] - df_world['trend']

```

```

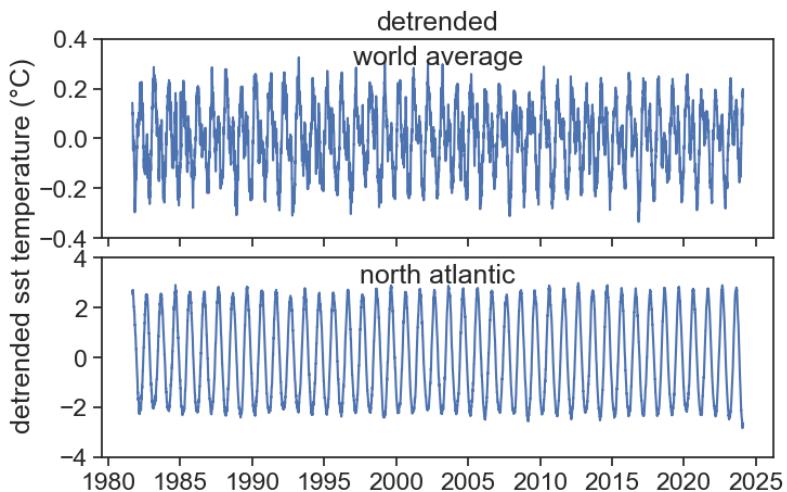
fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True)
fig.subplots_adjust(hspace=0.1) # increase vertical space between panels
ax[0].plot(df_world['detrended'])
ax[1].plot(df_north['detrended'])
fig.text(0.02, 0.5, 'detrended sst temperature (°C)', va='center', rotation='vertical')
ax[0].set(ylim=[-0.4,0.4],

```

```

    title="detrended")
ax[1].set(ylim=[-4,4],)
ax[0].text(0.5, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='center', verticalalignment='top',)
ax[1].text(0.5, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='center', verticalalignment='top',)
pass

```



## 39.4 seasonal component

It is useful to add two new columns to our dataframes: day of year and year:

```

df_north['doy'] = df_north.index.day_of_year
df_north['year'] = df_north.index.year
df_world['doy'] = df_world.index.day_of_year
df_world['year'] = df_world.index.year
df_world

```

	sst	trend	detrended	doy	year
date					
1981-09-01	20.15	20.006721	0.143279	244	1981

date	sst	trend	detrended	doy	year
1981-09-02	20.14	20.007717	0.132283	245	1981
1981-09-03	20.13	20.008703	0.121297	246	1981
1981-09-04	20.13	20.009624	0.120376	247	1981
1981-09-05	20.12	20.010481	0.109519	248	1981
...	...	...	...	...	...
2024-02-04	21.12	20.932460	0.187540	35	2024
2024-02-05	21.11	20.931882	0.178118	36	2024
2024-02-06	21.13	20.931297	0.198703	37	2024
2024-02-07	21.12	20.930707	0.189293	38	2024
2024-02-08	21.12	20.930164	0.189836	39	2024

### 39.4.1 group by

This is an extremely useful method. As the name suggests, it groups data according to some criterion.

```
gb_year_north = df_north.groupby('year')
gb_year_world = df_world.groupby('year')
```

Just like `resample` and `rolling`, `groupby` doesn't do anything after grouping the data, it waits for further instructions.

```
gb_year_world['sst'].mean()
```

year	sst
1981	19.953197
1982	20.051671
1983	20.133260
1984	20.029809
1985	19.924055
1986	19.994055
1987	20.148712
1988	20.108087
1989	20.039863
1990	20.152932
1991	20.145781

```
1992    20.069454
1993    20.080877
1994    20.098274
1995    20.170959
1996    20.118716
1997    20.269945
1998    20.328575
1999    20.117973
2000    20.151940
2001    20.286356
2002    20.321699
2003    20.370849
2004    20.328962
2005    20.412548
2006    20.374110
2007    20.321068
2008    20.312814
2009    20.432301
2010    20.436411
2011    20.314192
2012    20.369317
2013    20.403918
2014    20.525260
2015    20.647151
2016    20.671257
2017    20.607507
2018    20.567808
2019    20.652603
2020    20.678388
2021    20.630658
2022    20.648521
2023    20.889671
2024    21.042564
Name: sst, dtype: float64
```

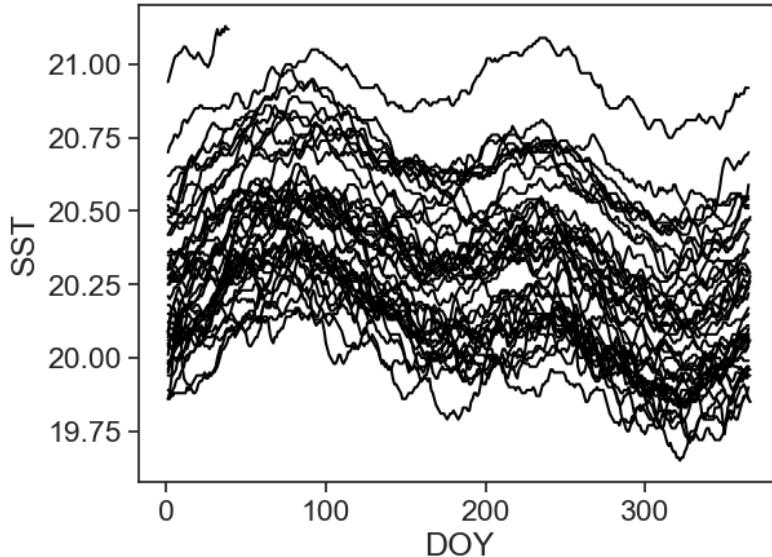
If all you need is the average by years, then the most common way of writing the command would be in one single line:

```
df_world.groupby('year')['sst'].mean()
```

Let's use `groupby` to plot world SST for all years, see how handy

this method is. Also note that groupby return an object that can be iterated upon, as we do below.

```
fig, ax = plt.subplots()
for year, data in gb_year_world:
    ax.plot(data['doy'], data['sst'], color="black")
ax.set(xlabel="DOY", ylabel="SST");
```



With a few more lines of code we can make this look pretty.

```
fig, ax = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
fig.subplots_adjust(hspace=0.1)

# define the segment of the colormap to use
start, end = 0.3, 0.8
base_cmap = plt.cm.hot_r
new_colors = base_cmap(np.linspace(start, end, 256))
new_cmap = mpl.colors.LinearSegmentedColormap.from_list("trunc({n},{a:.2f},{b:.2f})".format(n=
```

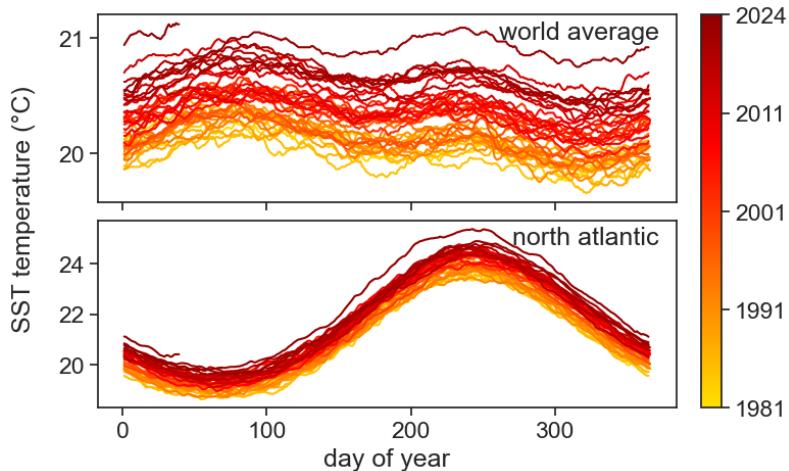
# defining the years for plotting  
years = [year for year, \_ in gb\_year\_world] + [year for year, \_ in gb\_year\_north]  
min\_year, max\_year = min(years), max(years)  
# create a new normalization that matches the years to the new colormap  
norm = mpl.colors.Normalize(vmin=min\_year, vmax=max\_year)

```

# plotting the data with year-specific colors from the new colormap
for year, data in gb_year_world:
    ax[0].plot(data.index.day_of_year, data['sst'], color=new_cmap(norm(year)))
for year, data in gb_year_north:
    ax[1].plot(data.index.day_of_year, data['sst'], color=new_cmap(norm(year)))

# colorbar setup
sm = mpl.cm.ScalarMappable(cmap=new_cmap, norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, ax=ax.ravel().tolist(), orientation='vertical', fraction=0.046, pad=0.04)
ticks_years = [1981, 1991, 2001, 2011, 2024] # Specify years for colorbar ticks
cbar.set_ticks(np.linspace(min_year, max_year, num=len(ticks_years)))
cbar.set_ticklabels(ticks_years)
# adding shared ylabel and xlabel
fig.text(0.02, 0.5, 'SST temperature (°C)', va='center', rotation='vertical')
ax[1].set_xlabel("day of year")
ax[0].text(0.97, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='right', verticalalignment='top',)
ax[1].text(0.97, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='right', verticalalignment='top',);

```



Now let's do the same for the detrended SST:

```

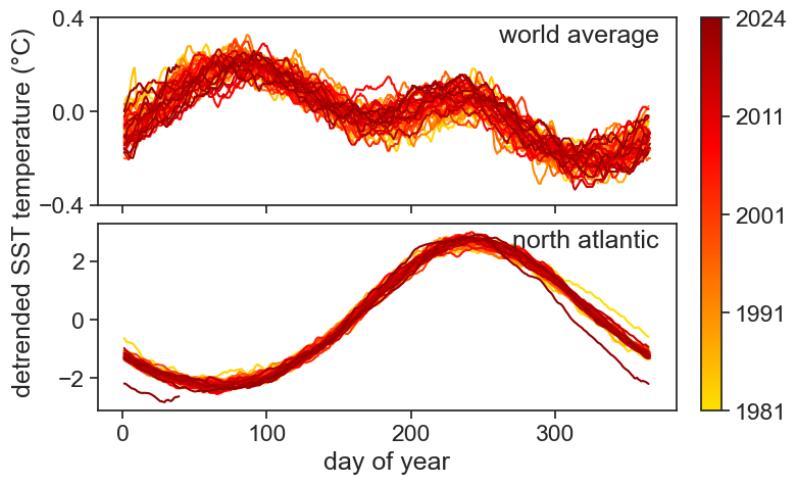
fig, ax = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
fig.subplots_adjust(hspace=0.1)

```

```

# define the segment of the colormap to use
start, end = 0.3, 0.8
base_cmap = plt.cm.hot_r
new_colors = base_cmap(np.linspace(start, end, 256))
new_cmap = mpl.colors.LinearSegmentedColormap.from_list("trunc({n},{a:.2f},{b:.2f})".format(n=
# defining the years for plotting
years = [year for year, _ in gb_year_world] + [year for year, _ in gb_year_north]
min_year, max_year = min(years), max(years)
# create a new normalization that matches the years to the new colormap
norm = mpl.colors.Normalize(vmin=min_year, vmax=max_year)
# plotting the data with year-specific colors from the new colormap
for year, data in gb_year_world:
    ax[0].plot(data.index.day_of_year, data['detrended'], color=new_cmap(norm(year)))
for year, data in gb_year_north:
    ax[1].plot(data.index.day_of_year, data['detrended'], color=new_cmap(norm(year)))
# colorbar setup
sm = mpl.cm.ScalarMappable(cmap=new_cmap, norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, ax=ax.ravel().tolist(), orientation='vertical', fraction=0.046, pad=0.05)
ticks_years = [1981, 1991, 2001, 2011, 2024] # Specify years for colorbar ticks
cbar.set_ticks(np.linspace(min_year, max_year, num=len(ticks_years)))
cbar.set_ticklabels(ticks_years)
# adding shared ylabel and xlabel
fig.text(0.02, 0.5, 'detrended SST temperature (°C)', va='center', rotation='vertical')
ax[0].set(yticks=[-0.4, 0, 0.4])
ax[1].set_xlabel("day of year")
ax[0].text(0.97, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='right', verticalalignment='top',)
ax[1].text(0.97, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='right', verticalalignment='top',);

```



In order to determine what is the seasonal component, we need to calculate the average of the detrended data across all years. Again, `groupby` comes to rescue in a most elegant way:

```
avg_north = df_north.groupby('doy')['detrended'].mean()
avg_world = df_world.groupby('doy')['detrended'].mean()
```

Let's incorporate the averages in the same graphs we saw above:

```
fig, ax = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
fig.subplots_adjust(hspace=0.1)

# define the segment of the colormap to use
start, end = 0.3, 0.8
base_cmap = plt.cm.hot_r
new_colors = base_cmap(np.linspace(start, end, 256))
new_cmap = mpl.colors.LinearSegmentedColormap.from_list("trunc({n},{a:.2f},{b:.2f})".format(n=1,
# defining the years for plotting
years = [year for year, _ in gb_year_world] + [year for year, _ in gb_year_north]
min_year, max_year = min(years), max(years)
# create a new normalization that matches the years to the new colormap
norm = mpl.colors.Normalize(vmin=min_year, vmax=max_year)
# plotting the data with year-specific colors from the new colormap
for year, data in gb_year_world:
    ax[0].plot(data.index.day_of_year, data['detrended'], color=new_cmap(norm(year)))
```

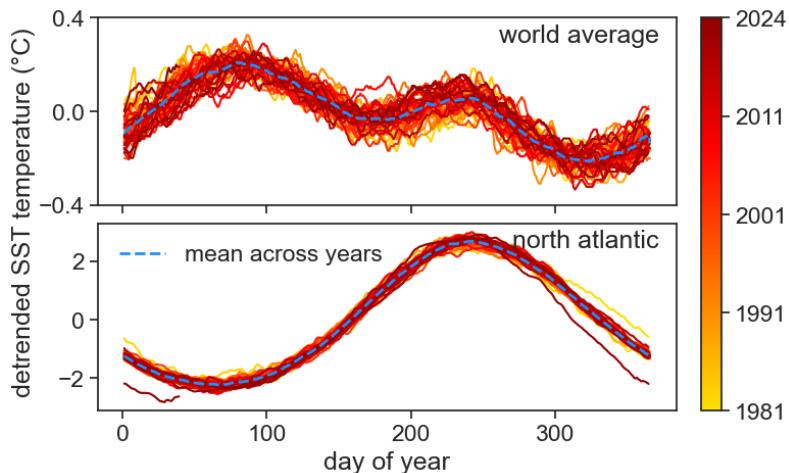
```

for year, data in gb_year_north:
    ax[1].plot(data.index.day_of_year, data['detrended'], color=new_cmap(norm(year)))

ax[0].plot(avg_world, color='dodgerblue', lw=2, ls='--')
ax[1].plot(avg_north, color='dodgerblue', lw=2, ls='--', label="mean across years")
ax[1].legend(loc="upper left", frameon=False)

# colorbar setup
sm = mpl.cm.ScalarMappable(cmap=new_cmap, norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, ax=ax.ravel().tolist(), orientation='vertical', fraction=0.046, pad=0.05)
ticks_years = [1981, 1991, 2001, 2011, 2024] # Specify years for colorbar ticks
cbar.set_ticks(np.linspace(min_year, max_year, num=len(ticks_years)))
cbar.set_ticklabels(ticks_years)
# adding shared ylabel and xlabel
fig.text(0.02, 0.5, 'detrended SST temperature (°C)', va='center', rotation='vertical')
ax[0].set(yticks=[-0.4, 0, 0.4])
ax[1].set_xlabel("day of year")
ax[0].text(0.97, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='right', verticalalignment='top',)
ax[1].text(0.97, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='right', verticalalignment='top',);

```



avg\_north

doy

```

1      -1.258903
2      -1.287413
3      -1.312651
4      -1.345564
5      -1.378712
...
362    -1.127575
363    -1.160508
364    -1.192747
365    -1.220802
366    -1.234997
Name: detrended, Length: 366, dtype: float64

```

`df_north`

date	sst	trend	detrended	doy	year	seasonal	resid
1981-09-01	23.78	21.237432	2.542568	244	1981	2.655598	-0.113030
1981-09-02	23.77	21.225380	2.544620	245	1981	2.650287	-0.105667
1981-09-03	23.80	21.213351	2.586649	246	1981	2.638455	-0.051806
1981-09-04	23.83	21.201237	2.628763	247	1981	2.635235	-0.006472
1981-09-05	23.87	21.188877	2.681123	248	1981	2.629680	0.051443
...	...	...	...	...	...	...	...
2024-02-04	20.37	23.096310	-2.726310	35	2024	-2.018653	-0.707657
2024-02-05	20.41	23.085914	-2.675914	36	2024	-2.034478	-0.641436
2024-02-06	20.42	23.075405	-2.655405	37	2024	-2.052858	-0.602547
2024-02-07	20.41	23.064565	-2.654565	38	2024	-2.066810	-0.587756
2024-02-08	20.42	23.053661	-2.633661	39	2024	-2.076346	-0.557315

The seasonal component is the average across all years, repeated over and over.

```

df_north['seasonal'] = df_north['doy'].map(avg_north)
df_world['seasonal'] = df_world['doy'].map(avg_world)

```

```

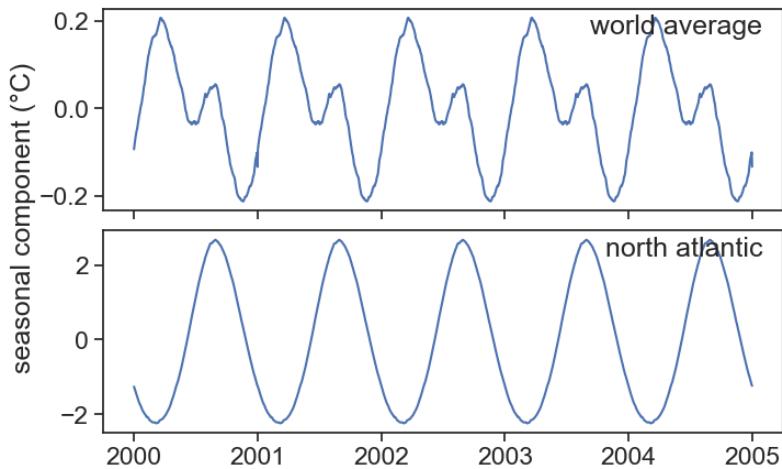
fig, ax = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
fig.subplots_adjust(hspace=0.1)
ax[0].plot(df_world.loc['2000':'2004', 'seasonal'])

```

```

ax[1].plot(df_north.loc['2000':'2004', 'seasonal'])
# adding shared ylabel and xlabel
fig.text(0.02, 0.5, 'seasonal component ( $^{\circ}$ C)', va='center', rotation='vertical')
ax[0].text(0.97, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='right', verticalalignment='top',)
ax[1].text(0.97, 0.97, r"north atlantic", transform=ax[1].transAxes,
           horizontalalignment='right', verticalalignment='top',);

```



## 39.5 residual

$$\text{residual} = \text{signal} - \text{trend} - \text{seasonal}$$

```

df_world['resid'] = df_world['detrended'] - df_world['seasonal']
df_north['resid'] = df_north['detrended'] - df_north['seasonal']

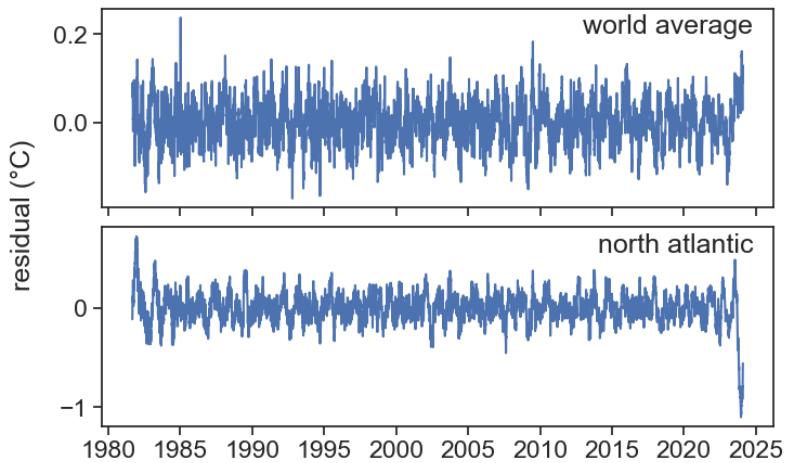
```

```

fig, ax = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
fig.subplots_adjust(hspace=0.1)
ax[0].plot(df_world['resid'])
ax[1].plot(df_north['resid'])
# adding shared ylabel and xlabel
fig.text(0.02, 0.5, 'residual ( $^{\circ}$ C)', va='center', rotation='vertical')
ax[0].text(0.97, 0.97, r"world average", transform=ax[0].transAxes,
           horizontalalignment='right', verticalalignment='top',)

```

```
ax[1].text(0.97, 0.97, r"north atlantic", transform=ax[1].transAxes,  
horizontalalignment='right', verticalalignment='top',);
```



How do we know we have properly decomposed our signal into trend, seasonal and residual? The residual should be stationary. Let's check using the ADF test:

```
result = adfuller(df_world['resid'])  
print('World-average residual\nADF test p-value: ', result[1])  
  
result = adfuller(df_north['resid'])  
print('North-Atlantic residual\nADF test p-value: ', result[1])
```

```
World-average residual  
ADF test p-value: 1.5673285788450267e-25  
North-Atlantic residual  
ADF test p-value: 5.2285789371886856e-18
```

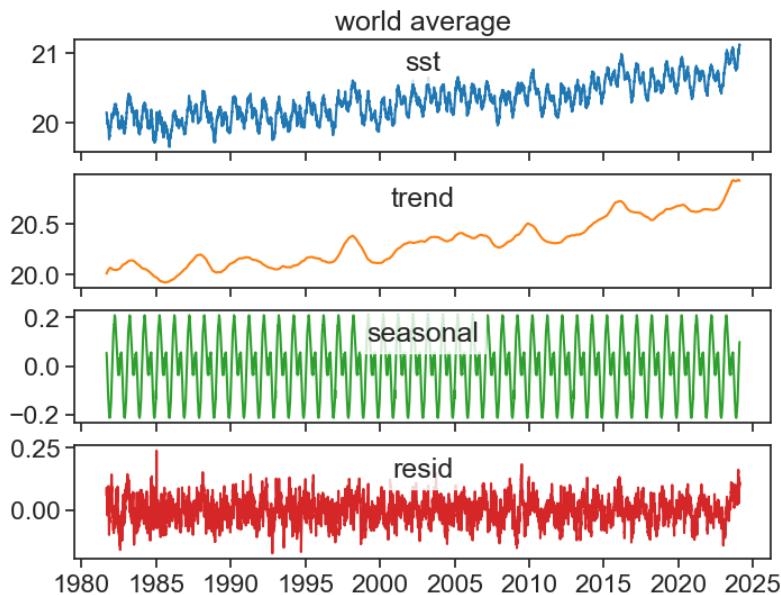
## 39.6 seasonal decomposition

It is customary to plot each component in its own panel:

```

fig, ax = plt.subplots(4, 1, figsize=(8,6), sharex=True)
pos = (0.5, 0.9)
components =["sst", "trend", "seasonal", "resid"]
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red"]
for axx, component, color in zip(ax, components, colors):
    data = getattr(df_world, component)
    axx.plot(data, color=color)
    axx.text(*pos, component, bbox=dict(facecolor='white', alpha=0.8),
              transform=axx.transAxes, ha='center', va='top')
ax[0].set(title="world average");

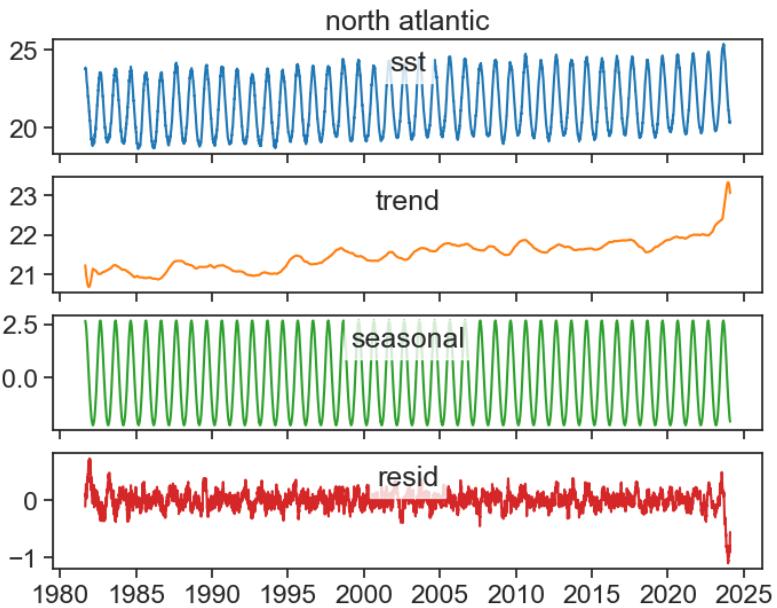
```



```

fig, ax = plt.subplots(4, 1, figsize=(8,6), sharex=True)
pos = (0.5, 0.9)
components =["sst", "trend", "seasonal", "resid"]
colors = ["tab:blue", "tab:orange", "tab:green", "tab:red"]
for axx, component, color in zip(ax, components, colors):
    data = getattr(df_north, component)
    axx.plot(data, color=color)
    axx.text(*pos, component, bbox=dict(facecolor='white', alpha=0.8),
              transform=axx.transAxes, ha='center', va='top')
ax[0].set(title="north atlantic");

```



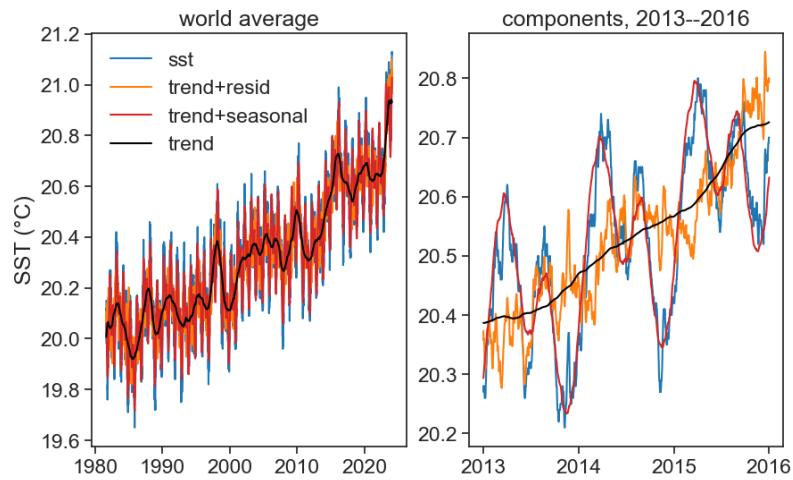
```

fig, ax = plt.subplots(1, 2, figsize=(10,6))
ax[0].plot(df_world['sst'], color="tab:blue", label="sst")
ax[0].plot(df_world['trend'] + df_world['resid'], color="tab:orange", label="trend+resid")
ax[0].plot(df_world['trend'] + df_world['seasonal'], color="tab:red", label="trend+seasonal")
ax[0].plot(df_world['trend'], color="black", label="trend")
ax[0].set(ylabel="SST (°C)",
           title="world average")
date_form = DateFormatter("%Y")
ax[0].xaxis.set_major_formatter(date_form)
ax[0].xaxis.set_major_locator(mdates.YearLocator(10))
ax[0].legend(frameon=False)

start = "2013-01-01"
end = "2016-01-01"
zoom = slice(start, end)
ax[1].plot(df_world.loc[zoom, 'sst'], color="tab:blue", label="sst")
ax[1].plot(df_world.loc[zoom, 'trend'] + df_world.loc[zoom, 'resid'], color="tab:orange", label="trend+resid")
ax[1].plot(df_world.loc[zoom, 'trend'] + df_world.loc[zoom, 'seasonal'], color="tab:red", label="trend+seasonal")
ax[1].plot(df_world.loc[zoom, 'trend'], color="black", label="trend")
date_form = DateFormatter("%Y")
ax[1].xaxis.set_major_formatter(date_form)
ax[1].xaxis.set_major_locator(mdates.YearLocator(1))

```

```
ax[1].set_title("components, 2013--2016");
```



## 40 theory

You might remember that even before this course started, I gave you an [example](#) of how to load data with pandas and how to plot it. Here we will be using the same data, namely, a time series for atmospheric CO<sub>2</sub> measured throughout the years in an obesevatory in Hawaii.

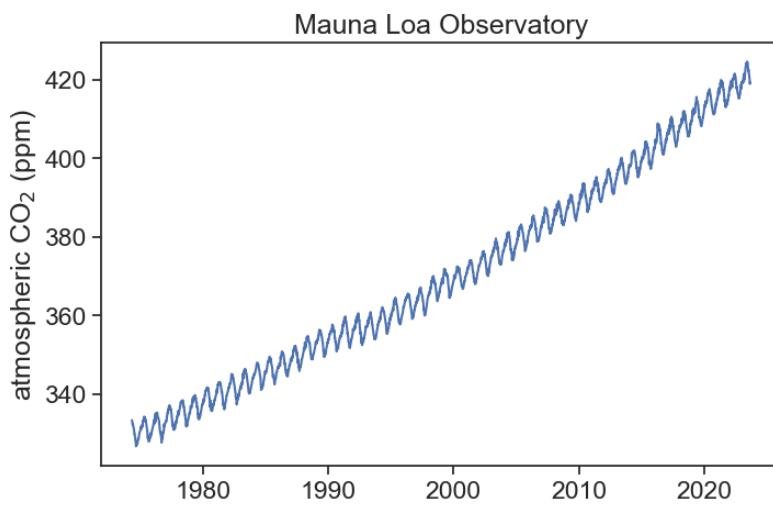
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from datetime import datetime as dt
import time
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import STL
# %matplotlib widget

url = "https://gml.noaa.gov/webdata/ccgg/trends/co2/co2_weekly_mlo.csv"
# df = pd.read_csv(url, header=47, na_values=[-999.99])

# you can first download, and then read the csv
filename = "co2_weekly_mlo.csv"
df = pd.read_csv(filename,
                 comment='#', # will ignore rows starting with #
                 na_values=[-999.99] # substitute -999.99 for NaN (Not a Number), data not av
                 )
df['date'] = pd.to_datetime(df[['year', 'month', 'day']])
```

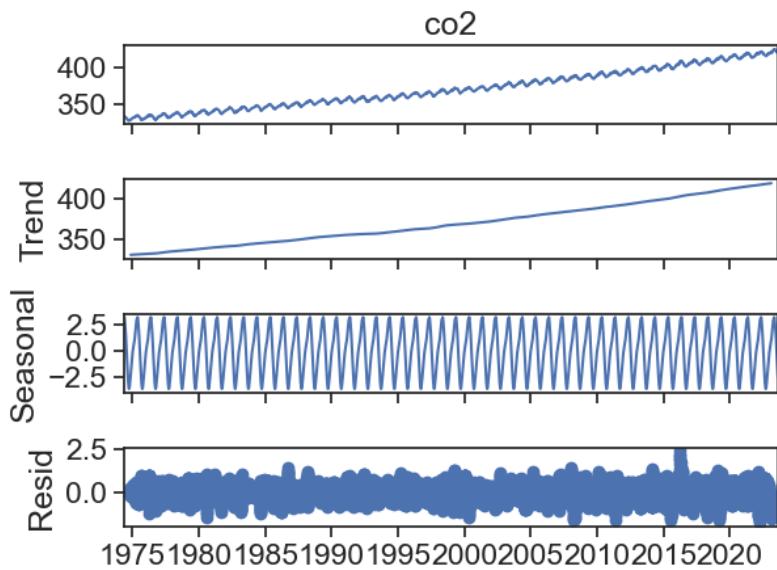
```
df = df.set_index('date')
# there are too many unnecessary columns, let's keep only co2 concentration (ppm)
df = df['average'].to_frame().rename(columns={'average': 'co2'})
# data comes in weekly frequency, let's make it daily and fill gaps with linear interpolation
df = df.resample('D').interpolate(method='linear')
```

```
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df['co2'])
ax.set(ylabel=r'atmospheric CO2 (ppm)',
       title="Mauna Loa Observatory");
```



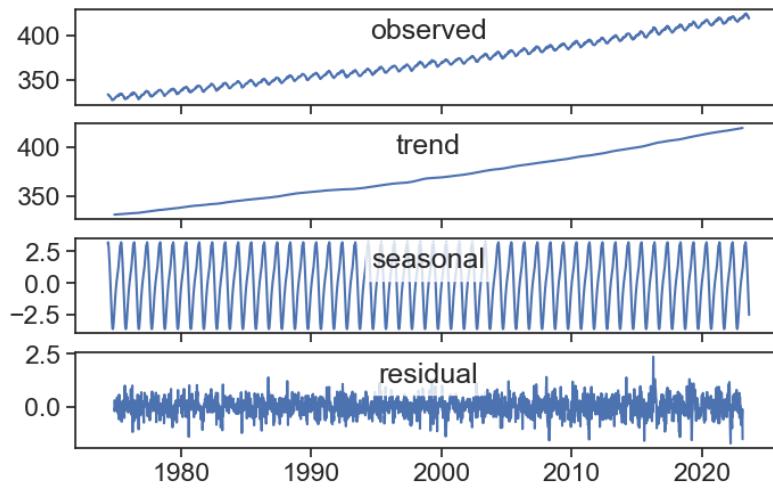
We can use `statsmodels.tsa.seasonal.seasonal_decompose` to decompose this time series.

```
result = seasonal_decompose(df['co2'], period=365)
result.plot();
```



The default plot is a bit ugly, we can make it better.

```
fig, ax = plt.subplots(4,1, figsize=(8,5), sharex=True)
ax[0].plot(result.observed)
ax[0].text(0.5, 0.9, "observed", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[0].transAxes, ha='center', va='top')
ax[1].plot(result.trend)
ax[1].text(0.5, 0.9, "trend", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[1].transAxes, ha='center', va='top')
ax[2].plot(result.seasonal)
ax[2].text(0.5, 0.9, "seasonal", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[2].transAxes, ha='center', va='top')
ax[3].plot(result.resid)
ax[3].text(0.5, 0.9, "residual", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[3].transAxes, ha='center', va='top');
```



## 40.1 additive model

`seasonal_decompose` returns an object with four components:

- observed:  $Y(t)$
- trend:  $T(t)$
- seasonal:  $S(t)$
- resid:  $e(t)$

The default assumption is that the various components are **summed** together to produce the original observed time series:

$$Y(t) = T(t) + S(t) + e(t)$$

This is called the additive model of seasonal decomposition.

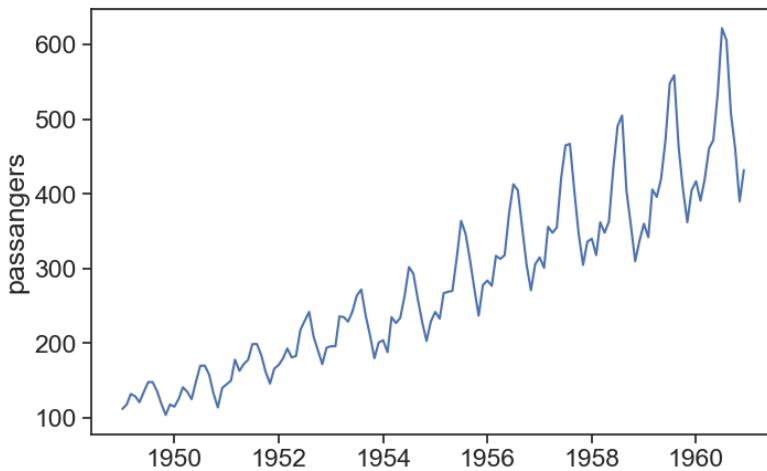
## 40.2 multiplicative model

`seasonal_decompose` returns an object with four components:

Not all time series seem to behave well when decomposed assuming an additive model. See, for instance, the famous time series for passengers of a US airline between 1949 to 1960.

```
df_air = pd.read_csv("airline-passengers.csv")
df_air['date'] = pd.to_datetime(df_air['Month'], format='%Y-%m')
df_air = df_air.set_index('date')

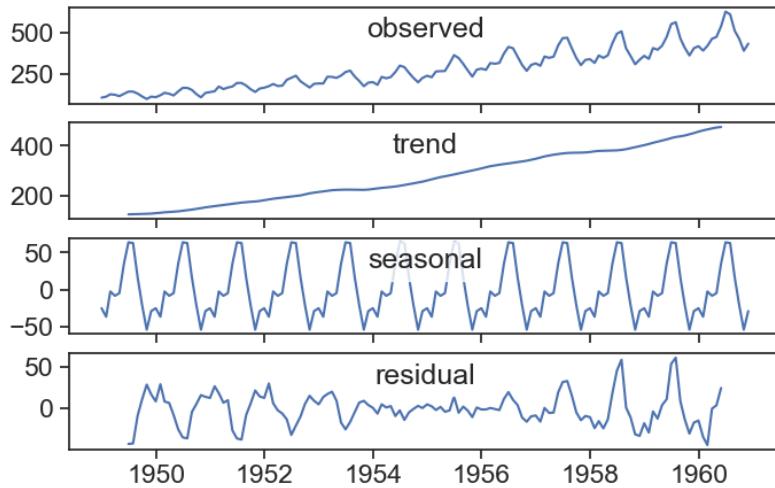
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df_air['Passengers'])
ax.set(ylabel='passangers');
```



```
result = seasonal_decompose(df_air['Passengers'], period=12)

fig, ax = plt.subplots(4,1, figsize=(8,5), sharex=True)
ax[0].plot(result.observed)
ax[0].text(0.5, 0.9, "observed", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[0].transAxes, ha='center', va='top')
ax[1].plot(result.trend)
ax[1].text(0.5, 0.9, "trend", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[1].transAxes, ha='center', va='top')
ax[2].plot(result.seasonal)
ax[2].text(0.5, 0.9, "seasonal", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[2].transAxes, ha='center', va='top')
ax[3].plot(result.resid)
```

```
ax[3].text(0.5, 0.9, "residual", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[3].transAxes, ha='center', va='top');
```



The residual is much larger at the beginning and at the end, and this is probably because the seasonal component increases with time. A good way to overcome this is to use the multiplicative model for seasonal decomposition:

$$Y(t) = T(t) \times S(t) \times e(t)$$

The components are now **multiplied** together to yield the observed time series. Let's test it:

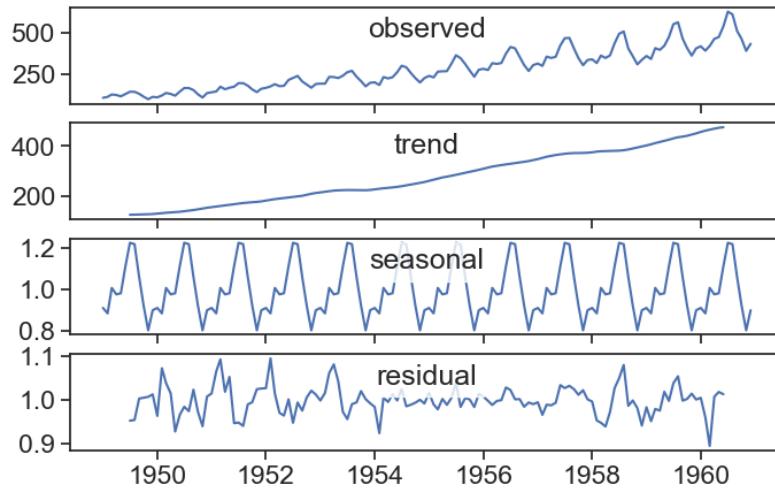
```
result = seasonal_decompose(df_air['Passengers'], period=12, model="multiplicative")

fig, ax = plt.subplots(4,1, figsize=(8,5), sharex=True)
ax[0].plot(result.observed)
ax[0].text(0.5, 0.9, "observed", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[0].transAxes, ha='center', va='top')
ax[1].plot(result.trend)
ax[1].text(0.5, 0.9, "trend", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[1].transAxes, ha='center', va='top')
ax[2].plot(result.seasonal)
ax[2].text(0.5, 0.9, "seasonal", bbox=dict(facecolor='white', alpha=0.8),
```

```

        transform=ax[2].transAxes, ha='center', va='top')
ax[3].plot(result.resid)
ax[3].text(0.5, 0.9, "residual", bbox=dict(facecolor='white', alpha=0.8),
           transform=ax[3].transAxes, ha='center', va='top');

```



## 40.3 STL

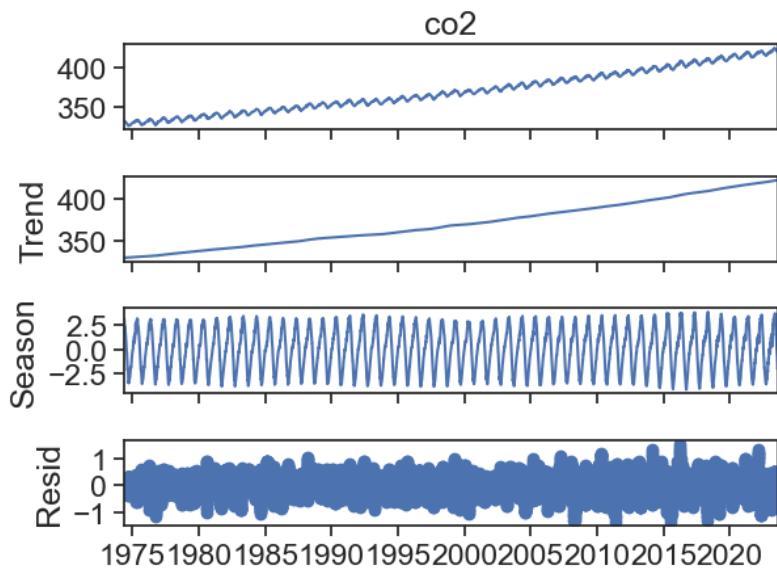
STL stands for “Seasonal and Trend decomposition using LOESS”. LOESS, in turn means “locally estimated scatterplot smoothing”, and it is a spiced up version of the Savitzky-Golay smoothing filter we saw earlier in this course. The method is not trivially understood as the *naive* seasonal decompostion we build together, and the full method is described by Cleveland et al. (1990) [here](#). See in this video how the smoothing method works.

The most striking difference with the plain seasonal decompose method is that the **seasonal** component is not constant in time, it can slightly change and evolve over time.

```

data = df['co2']
res = STL(data, period=365).fit()
res.plot();

```



A possible downside of STL is the fact that it assumes an additive model. If the seasonal oscillations in your data increase with the value of the series, then most probably a multiplicative model would be best for you. How to use STL then? One can take the logarithm of the series, since for a multiplicative model

$$Y(t) = T(t) \times S(t) \times e(t)$$

taking the logarithms yields

$$\log(Y) = \log(T) + \log(S) + \log(e).$$

We can now use STL since our model looks like additive now. After using STL, one can exponentiate the results to reverse the logarithm operation.

## 41 widgets

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from statsmodels.tsa.seasonal import seasonal_decompose
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from datetime import datetime as dt
import time
from statsmodels.tsa.stattools import adfuller
import matplotlib.colors as mcolors
import urllib.request
import json
import plotly.io as pio
pio.renderers.default = "plotly_mimetype+notebook_connected"
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import plotly.io as pio
pio.templates.default = "presentation"

# %matplotlib widget

# helper function to check if a year is a leap year
def is_leap_year(year):
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
```

```

        return True
    elif year % 400 != 0:
        return False
    else:
        return True

def load_and_process_data(type, filename):
    # from https://climatereanalyzer.org/clim/sst_daily/
    # up-to-date URLs for world average and for north atlantic SST
    if type == "world":
        url = "https://climatereanalyzer.org/clim/sst_daily/json/oisst2.1_world2_sst_day.json"
    if type == "north":
        url = "https://climatereanalyzer.org/clim/sst_daily/json/oisst2.1_natlan1_sst_day.json"
    # load JSON data from URL
    with urllib.request.urlopen(url) as response:
        data = json.load(response)
    # Convert JSON data to DataFrame
    data = pd.DataFrame(data)
    # this dataframe is not good to work with, let's make a new one
    # columns are year numbers, rows are temperatures for each day of the year
    years = data['name'].astype(str).tolist()
    values = data['data'].tolist()
    df = pd.DataFrame(values).T
    df.columns = years
    # now let's make a continuous df, with all the data in one column
    # initializing an empty list to hold dataframes before concatenating them
    dfs = []
    # iterating over each column in the original dataframe
    for column in df.columns:
        try:
            # converting the column name to an integer to handle it as a year
            year = int(column)
            # determining the number of days in the year
            days_in_year = 366 if is_leap_year(year) else 365
            # creating a date range for the year
            dates = pd.date_range(start=f'{year}-01-01', end=f'{year}-12-31', periods=days_in_year)
            # creating a temporary dataframe for the year's data
            temp_df = pd.DataFrame({'sst': df[column][:days_in_year].values}, index=dates)
            # adding the temporary dataframe to the list
            dfs.append(temp_df)

```

```

        except ValueError:
            # skipping columns that do not represent a year (e.g., "1982-2011 mean", "plus 2",
            continue

# concatenating all the temporary dataframes into one
df_sst_concat = pd.concat(dfs)
# resetting the index to have a datetime index
df_sst_concat.index = pd.to_datetime(df_sst_concat.index)
df_sst_concat.index.name = 'date'
df_sst_concat.dropna(inplace=True)
# save to file
df_sst_concat.to_csv(filename, index=True)

load_and_process_data("world", "sst_world.csv")
load_and_process_data("north", "sst_north.csv")

df_north = pd.read_csv("sst_north.csv", index_col='date', parse_dates=True)
df_world = pd.read_csv("sst_world.csv", index_col='date', parse_dates=True)

# world
df_world['doy'] = df_world.index.day_of_year
df_world['year'] = df_world.index.year
df_grouped = df_world.groupby('year')

# north
df_north['doy'] = df_north.index.day_of_year
df_north['year'] = df_north.index.year
df_grouped = df_north.groupby('year')

```

## 41.1 range slider

```

blue = px.colors.qualitative.D3[0]
orange = px.colors.qualitative.D3[1]

fig = make_subplots(specs=[[{"secondary_y": True}]])

fig.add_trace(
    go.Scatter(x=list(df_world.index),

```

```

        y=list(df_world['sst']),
        name='world mean',
        line=dict(color=blue),),
    secondary_y=False,
)

fig.add_trace(
    go.Scatter(x=list(df_north.index),
                y=list(df_north['sst']),
                name='north atlantic',
                line=dict(color=orange),),
    secondary_y=True,
)

# Add range slider
fig.update_layout(
    title='sea surface temperature',
    # yaxis_title='temperature (°C)',
    xaxis=dict(
        rangeslider={"visible":True},
        type="date"
    ),
    legend={"orientation":"h",                      # Horizontal legend
            "yanchor":"top",                  # Anchor legend to the top
            "y":1.1,                         # Adjust vertical position
            "xanchor":"center",               # Anchor legend to the right
            "x":0.5,                         # Adjust horizontal position
        },
)
# Set y-axes titles
fig.update_yaxes(
    title_text="temperature (°C)",
    titlefont=dict(
        color=blue
    ),
    secondary_y=False,)
fig.update_yaxes(
    title_text="temperature (°C)",
    titlefont=dict(

```

```
    color=orange
),
secondary_y=True,)
```

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

## 41.2 temperature vs DOY, with colorbar

```
# Define colormap and color range
# base_cmap = plt.cm.hot_r
# start, end = 0.3, 0.8

# base_cmap = plt.cm.coolwarm
# start, end = 0.3, 0.7

base_cmap = plt.cm.turbo
start, end = 0.1, 0.75

# Create truncated colormap
new_colors = base_cmap(np.linspace(start, end, 256))
new_cmap = mcolors.LinearSegmentedColormap.from_list("trunc({n},{a:.2f},{b:.2f})".format(n=base_cmap.name), new_colors)

# List of years
years = range(1981, 2025)

# Create dictionary mapping years to hexadecimal colors
year_colors = {}
for i, year in enumerate(years):
    # Calculate normalized value for the year
    norm_value = (year - years[0]) / (years[-1] - years[0])

    # Get color from colormap
    rgba_color = new_cmap(norm_value)
```

```

# Convert RGBA to hex
hex_color = mcolors.rgb2hex(rgb_color[:3]) # Exclude alpha channel

# Add to dictionary
year_colors[year] = hex_color

# Print the dictionary
# print(year_colors)

# overwrite last 2 years with hotter colors
year_colors[2023] = plt.cm.colors.to_hex('hotpink')
year_colors[2024] = plt.cm.colors.to_hex('deeppink')

years = list(year_colors.keys())
colors = [year_colors[year] for year in years]

# Define a custom colorscale based on the extracted colors
colorscale = [[i / (len(years) - 1), colors[i]] for i in range(len(years))]

fig = go.Figure()

# iterate over each unique year
for year in df_world.index.year.unique():
    # filter the data for the current year
    data_year = df_world[df_world.index.year == year]

    # add a trace for the current year
    fig.add_trace(go.Scatter(
        x=data_year.index.dayofyear, # x-axis: day of year
        y=data_year['sst'], # y-axis: sea surface temperature
        mode='lines',
        name=str(year), # Name of the trace (year)
        line=dict(color=year_colors[year]),
        hovertemplate='<b>Date</b>: %{text}<br>' +
                     '<b>SST (°C)</b>: %{y}', # Customize hover template
        text=data_year.index.strftime('%Y-%m-%d'), # Convert date to YYYY-MM-DD format
        hoverlabel=dict(namelength=0) # Set namelength to 0 to remove the tag
    ))

# update layout

```

```

fig.update_layout(
    title='Sea Surface Temperature, World Average',
    xaxis_title='day of year',
    yaxis_title='temperature (°C)',
    showlegend=False,
)
# dummy data from which colorbar will be used
colorbar_trace = go.Scatter(x=[None],
                             y=[None],
                             mode='markers',
                             marker=dict(
                                 colorscale=colorscale,
                                 showscale=True,
                                 cmin=1981,
                                 cmax=2024,
                                 colorbar=dict(thickness=15, tickvals=[1981,1991,2001,2011,2024]
                                              #      ticktext=['Low', 'High'], outlinewidth=0
                                              )
                             ),
                             hoverinfo='none'
                           )
fig.add_trace(colorbar_trace)
fig.show()

```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

```

# Assuming df_world is your DataFrame containing the data

# Convert the index to datetime if it's not already in datetime format
df_north.index = pd.to_datetime(df_north.index)

# Create a Plotly figure
fig = go.Figure()

# Iterate over each unique year
for year in df_north.index.year.unique():
    # Filter the data for the current year
    data_year = df_north[df_north.index.year == year]

```

```

# Add a trace for the current year
fig.add_trace(go.Scatter(
    x=data_year.index.dayofyear, # x-axis: day of year
    y=data_year['sst'], # y-axis: sea surface temperature
    mode='lines',
    name=str(year), # Name of the trace (year)
    line=dict(
        color=year_colors[year]
    ),
    hovertemplate='<b>Date</b>: %{text}<br>' +
                  '<b>SST (°C)</b>: %{y}', # Customize hover template
    text=data_year.index.strftime('%Y-%m-%d'), # Convert date to YYYY-MM-DD format
    hoverlabel=dict(nameLength=0) # Set nameLength to 0 to remove the tag
))

# Update layout
fig.update_layout(
    title='Sea Surface Temperature, North Atlantic',
    xaxis_title='day of year',
    yaxis_title='temperature (°C)',
    showlegend=False,
    # height=600, # Adjust the height of the image
    # width=800 # Adjust the width of the image
)
# dummy data from which colorbar will be used
colorbar_trace = go.Scatter(x=[None],
                             y=[None],
                             mode='markers',
                             marker=dict(
                                 colorscale=colorscale,
                                 showscale=True,
                                 cmin=1981,
                                 cmax=2024,
                                 colorbar=dict(thickness=15, tickvals=[1981, 1991, 2001, 2011, 2024],
                                              # ticktext=['Low', 'High'], outlineWidth=0
                                              )
                             ),
                             hoverinfo='none'
)
fig.add_trace(colorbar_trace)

```

```
# Show the plot
fig.show()
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

## 41.3 seasonal decomposition

click on the legend components to turn on/off the different lines.

```
decompose_north = seasonal_decompose(df_north['sst'], period=365)

fig = go.Figure()

fig.add_trace(
    go.Scatter(x=list(df_north.index), y=list(decompose_north.observed), name='observed', visible=True))

fig.add_trace(
    go.Scatter(x=list(df_north.index), y=list(decompose_north.trend), name='trend'))

fig.add_trace(
    go.Scatter(x=list(df_north.index), y=list(decompose_north.trend + decompose_north.seasonal), name='seasonal'))

fig.add_trace(
    go.Scatter(x=list(df_north.index), y=list(decompose_north.trend - decompose_north.resid), name='resid'))

fig.update_layout(
    title='north atlantic SST',
    yaxis_title='temperature (°C)',
    xaxis=dict(range=['2017-01-01', '2023-01-01'],),
    template="presentation",
    legend=dict(
        orientation="h", # Horizontal legend
        yanchor="top", # Anchor legend to the top
        y=-0.1, # Adjust vertical position
        xanchor="center", # Anchor legend to the right
        x=0.5, # Adjust horizontal position
```

```
    ),  
)
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

```
decompose_world = seasonal_decompose(df_world['sst'], period=365)
```

```
fig = go.Figure()
```

```
fig.add_trace(
```

```
    go.Scatter(x=list(df_world.index), y=list(decompose_world.observed), name='observed', visi
```

```
fig.add_trace(
```

```
    go.Scatter(x=list(df_world.index), y=list(decompose_world.trend), name='trend'))
```

```
fig.add_trace(
```

```
    go.Scatter(x=list(df_world.index), y=list(decompose_world.trend + decompose_world.seasonal),
```

```
fig.add_trace(
```

```
    go.Scatter(x=list(df_world.index), y=list(decompose_world.trend - decompose_world.resid), n
```

```
fig.update_layout(
```

```
    title='world average SST',
```

```
    yaxis_title='temperature (°C)',
```

```
    xaxis=dict(range=['2017-01-01', '2023-01-01'],),
```

```
    yaxis=dict(range=[20.2, 21.0],),
```

```
    template="presentation",
```

```
    legend=dict(
```

```
        orientation="h",           # Horizontal legend
```

```
        yanchor="top",            # Anchor legend to the top
```

```
        y=-0.1,                  # Adjust vertical position
```

```
        xanchor="center",         # Anchor legend to the right
```

```
        x=0.5,                   # Adjust horizontal position
```

```
    ),
```

```
)
```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

# **Part VII**

## **time lags**

## **42 motivation**

## 43 cross-correlation

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter
from datetime import datetime as dt
from scipy import signal
import math
import scipy
import statsmodels.api as sm

from ipywidgets import interact, IntSlider
from IPython.display import display, HTML

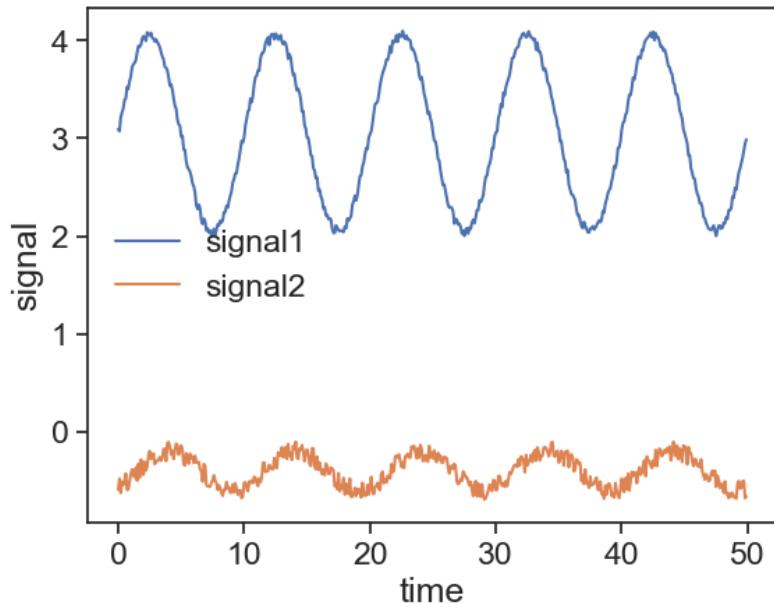
# %matplotlib widget

delta_t = 0.1
time = np.arange(0, 50, delta_t)
N = len(time)
period = 10.0
time_offset = scipy.constants.golden
omega = math.tau / period
signal1 = 3.0 + np.sin(omega*time) + 0.1 * np.random.random(N)
signal2 = -0.5 + 0.2 * np.sin(omega*time - math.tau * (time_offset/period)) + 0.2 * np.random...
```

```

fig, ax = plt.subplots()
ax.plot(time, signal1, label="signal1")
ax.plot(time, signal2, label="signal2")
ax.set(xlabel="time",
       ylabel="signal")
ax.legend(frameon=False);

```



Let's normalize our data:

$$x_n = \frac{x - \mu}{\sigma}.$$

This is identical to the Z-score we learned before. Effectively, our data now has zero mean and unit standard deviation.

```

def norm_data(data):
    # normalize data to have mean=0 and standard_deviation=1
    return (data - data.mean())/ data.std()

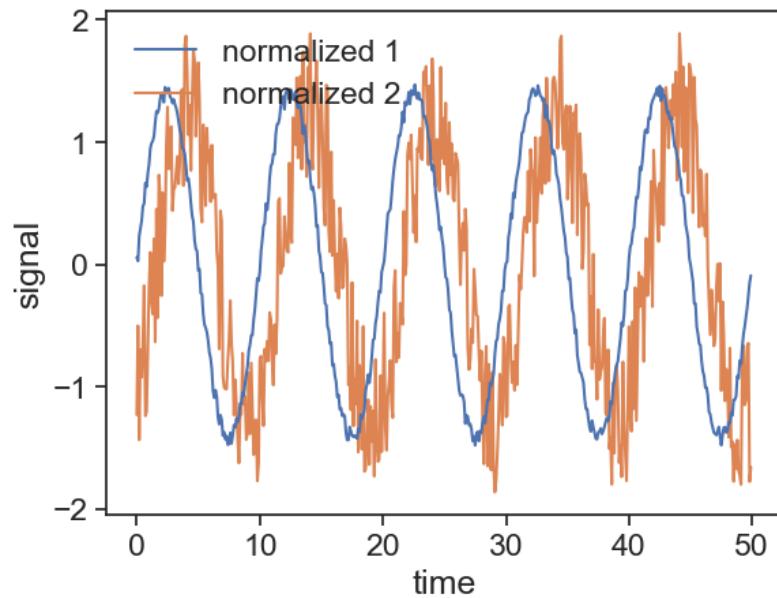
s1 = norm_data(signal1)
s2 = norm_data(signal2)

```

```

fig, ax = plt.subplots()
ax.plot(time, s1, label="normalized 1")
ax.plot(time, s2, label="normalized 2", zorder=0)
ax.set(xlabel="time",
       ylabel="signal")
ax.legend(frameon=False)

```



The definition of cross correlation is very similar to autocorrelation:

$$\rho_{XY}(\tau) = \frac{E[(X_t - \mu)(Y_{t+\tau} - \mu)]}{\sigma_X \sigma_Y}$$

If the autocorrelation helps us check how similar a signal is to a lagged version of itself, the cross correlation measures how a signal  $X$  is similar to a lagged version of another signal  $Y$ .

Let's write this ourselves:

```

def cross_corr(s1, s2):
    """
    we assume s1 and s2 to have the same length

```

```

"""
N = len(s1)
lags_left = np.arange(-(N-1), 1)
lags_right = np.arange(1, N)
cc_left = np.array([np.sum(s2[-i:]*s1[:i]) for i in np.arange(1, len(lags_left)+1)])
cc_right = np.array([np.sum(s2[:-i]*s1[i:]) for i in np.arange(1, len(lags_right)+1)])
lags = np.hstack([lags_left, lags_right])
cc = np.hstack([cc_left, cc_right]) / N
return lags, cc

```

Of course, there are available functions that accomplish the same, let's compare them with our code:

```

corr_np = signal.correlate(s1, s2) / N
lags = signal.correlation_lags(len(s1), len(s2))

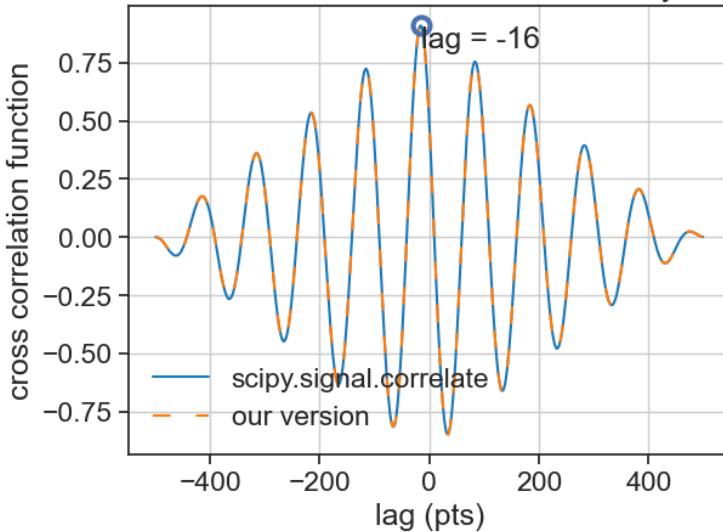
my_lag, my_cc = cross_corr(s1, s2)

max_index = np.argmax(corr_np)
fig, ax = plt.subplots()
ax.plot(lags, corr_np, color="tab:blue", label = "scipy.signal.correlate")
ax.plot(my_lag, my_cc, 'tab:orange', dashes=[7, 10], label = "our version")
ax.legend(loc="lower left", frameon=False)

ax.plot([lags[max_index]], [corr_np[max_index]], ls="None", marker="o", mfc="None", mew=3, ms=100)
ax.text(lags[max_index], corr_np[max_index],
        f"lag = {lags[max_index]}", va="top", ha="left")
ax.grid(True)
ax.set(xlabel="lag (pts)",
       ylabel="cross correlation function",
       title=f"s2 is best correlated with s1 when shifted by {lags[max_index]} pts");

```

s2 is best correlated with s1 when shifted by -16 pts



```
nann = np.zeros(N) * np.nan
s1_padded = np.hstack([nann, s1, nann])
N_padded = len(s1_padded)
lag_padded = np.arange(N_padded) - N
s2_padded = np.hstack([nann, nann, s2, nann, nann])
```

```
# Define a function to plot the time series with a shift
def plot_time_series(shift=-500):
    sh = shift + N + 1
    fig, ax = plt.subplots(2, 1, figsize=(8, 6))
    fig.subplots_adjust(hspace=0.3) # increase vertical space between panels
    ax[0].plot(lag_padded * delta_t, s1_padded, label="normalized 1")
    ax[0].plot(lag_padded * delta_t, s2_padded[2*N-sh:-sh], label="normalized 2", zorder=0)
    ax[0].set(xlabel="time",
              ylabel="normalized signal",
              xlim=[delta_t*lag_padded.min(), delta_t*lag_padded.max()])
    ax[1].plot(lags, corr_np, color="black", alpha=0.2)
    ax[1].plot(lags[:sh], corr_np[:sh], lw=3, color="xkcd:hot pink")
    ax[1].set(xlabel="lag (pts)",
              ylabel="CCF",
              xlim=[lags.min(), lags.max()],
              ylim=[-1,1])
```

```
plt.show()

# Use interact to create the widget
interact(plot_time_series, shift=IntSlider(min=-500, max=499, step=1, value=-500))

# Add custom CSS to change the length of the slider
display(HTML("""
<style>
.widget-slider {
    width: 100%;
}
</style>
"""))

interactive(children=(IntSlider(value=-500, description='shift', max=499, min=-500), Output()))

<IPython.core.display.HTML object>
```

## 44 dynamic time warping

There are beautiful explanations our there of how the Dynamic Time Warping algorithm works. An excellent source is [this tutorial](#).

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import matplotlib.gridspec as gridspec

def fill_dtw_cost_matrix(abs_diff):
    N1, N2 = abs_diff.shape
    cost_matrix = abs_diff.copy() # make a copy
    # prepend row full of nan
    cost_matrix = np.vstack([np.full(N1, np.nan), cost_matrix])
    # prepend col full of nan
    cost_matrix = np.hstack([np.full((N2+1, 1), np.nan), cost_matrix])
    # make origin zero
    cost_matrix[0, 0] = 0
    for i in range(1, N1+1):
        for j in range(1, N2+1):
            south      = cost_matrix[i-1, j]
            west       = cost_matrix[i, j-1]
            southwest = cost_matrix[i-1, j-1]
            # print(([south, west, southwest]))
            # print(np.nanmin([south, west, southwest]))
            cost_matrix[i, j] = cost_matrix[i, j] + np.nanmin([south, west, southwest])
```

I also used as sources for this lecture the following pages:

- Abhishek Mishra's Medium page
- Romain Tavenard's DTW [tutorial](#) for a matplotlib

Other good sources: GenT Warper, Vatsal, Roger Jang, International Audio Laboratories Erlangen, The Practical Quant, Meinard Müller: Dynamic Time Warping, Chapter 4 of Information Retrieval for Music and Motion, Essi Alizadeh, Elena Tsiporkova

```

    return cost_matrix

def find_min_path(m):
    N1, N2 = m.shape
    path=[[N1-1, N2-1]]
    i,j = path[-1][0], path[-1][1]
    while True:
        south = m[i-1, j]
        west = m[i, j-1]
        southwest = m[i-1, j-1]
        if southwest <= min(south, west):
            i = i-1
            j = j-1
        elif south <= west:
            i = i-1
        else:
            j = j-1
        path.append([i,j])
        if (i==1) & (j==1):
            break
    return path

```

```

def plot_series(s1, s2):
    fig, ax = plt.subplots()

    ax.plot(s1, color="tab:blue", lw=3)
    ax.plot(s2, color="tab:orange", lw=3)

    i = len(s1)//2
    for j in range(i-2,i+2):
        ax.plot([i, j], [s1[i], s2[j]], color="black", alpha=0.3, ls="--")
        ax.text((i+j)/2, s2[j] + (s1[i]-s2[j])/2, f"{{np.abs(s1[{i}]-s2[{j}]):.2f}}" )

    ax.text(-0.5, s1[0], "P[i]", color="tab:blue")
    ax.text(-0.5, s2[0], "Q[i]", color="tab:orange")

    ax.set(xticks=np.arange(0,len(s1)),
           xlim=[-1, len(s1)],
           title=fr"absolute vertical distance between $P[{i}]$ and $Q[{i}]$");

```

```

def plot_abs_diff(s1, s2, c):
    fig = plt.figure(1, figsize=(8, 8))
    gs = gridspec.GridSpec(2, 2, width_ratios=[1,len(s2)], height_ratios=[len(s1),1])
    gs.update(left=0.10, right=0.90,top=0.90, bottom=0.10,
              hspace=0.05, wspace=0.05
              )

    ax_left = plt.subplot(gs[0, 0])
    ax_bottom = plt.subplot(gs[1, 1])
    ax_matrix = plt.subplot(gs[0, 1])

    text_dict = {'ha':'center', 'va':'center', 'color':"orangered",
                 'fontsize':24, 'weight':'bold'}

    s1_reshaped = s1.reshape(len(s1), 1)
    s2_reshaped = s2.reshape(1, len(s2))

    # Plot the matrix
    ax_matrix.imshow(c, cmap='Blues', origin="lower")
    ax_matrix.set(xticks=[],
                  yticks=[],
                  title="absolute difference: "+r"\|P[i]-Q[j]\|")
    # https://stackoverflow.com/a/33829001
    for (j,i),label in np.ndenumerate(c):
        ax_matrix.text(i, j, f"{label:.2f}", **text_dict)

    # Plot the array
    ax_left.imshow(s1_reshaped, cmap='Greys', origin="lower")
    ax_bottom.imshow(s2_reshaped, cmap='Greys')

    ax_left.set(xticks=[],
                yticks=np.arange(len(s1)),
                xlabel="P[i]"
                )
    ax_left.xaxis.set_label_position("top")

    ax_bottom.set(yticks=[],
                  xticks=np.arange(len(s2)),
                  )
    ax_bottom.yaxis.set_label_position("right")

```

```

ax_bottom.set_ylabel("Q[j]", rotation="horizontal", labelpad=20)
ax_left.tick_params(axis='y', which='both', left=False)
ax_bottom.tick_params(axis='x', which='both', bottom=False)

for (j,i),label in np.ndenumerate(s1_reshaped):
    ax_left.text(i, j, f"{label:.2f}", **text_dict)

for (j,i),label in np.ndenumerate(s2_reshaped):
    ax_bottom.text(i, j, f"{label:.2f}", **text_dict)

# fig.savefig("abs_diff_3.png")

def plot_cost_and_path(m, p):
    fig, ax = plt.subplots(figsize=(10,10))

    m_plot = ax.imshow(m, origin="lower")
    plt.colorbar(m_plot, fraction=0.046, pad=0.04)

    text_dict = {'ha':'center', 'va':'center', 'color':"deepskyblue",
                'fontsize':24, 'weight':'bold'}

    # https://stackoverflow.com/a/33829001
    for (j,i),label in np.ndenumerate(m):
        if np.isnan(label):
            label = r"\times"
        else: label = f"{label:.2f}"
        ax.text(i,j,label, **text_dict, zorder=10)

    ax.plot(p[:,1], p[:,0], color=[0.6]*3)
    ax.set(title="path of smallest total distance")

def plot_warped_series(s1, s2, p):
    fig, ax = plt.subplots()

    Q_offset = 4
    ax.plot(s1, color="tab:blue", lw=3)
    ax.plot(s2-Q_offset, color="tab:orange", lw=3)

    p1 = p - 1

```

```

for (i,j) in p1:
    ax.plot([i, j], [s1[i], s2[j]-Q_offset], color="black", alpha=0.3, ls="--")

    ax.text(-1, s1[0], "P[i]", color="tab:blue")
    ax.text(-1, s2[0]-Q_offset, "Q[i]", color="tab:orange")

ax.set(xticks=np.arange(0,len(s1)),
       xlim=[-1, len(s1)],
       # yticks=np.arange(-2,2),
       title=r"absolute vertical distance between $P[1]$ and $Q[i]$");

def normalize(series):
    return (series-series.mean()) / series.std()

def impose_causality(m):
    N1, N2 = m.shape
    lower_right = np.tril_indices(n=N1, m=N2, k=-4)
    cost_matrix[lower_right] = np.nan

s1_a = np.array([1,5,4,8,4,7,6,5,3,3])
s2_a = np.array([1,2,3,5,5,2,6,4,5,5]) - 4
c_a = np.abs(np.subtract.outer(s1_a, s2_a))
##Call DTW function
m_a = fill_dtw_cost_matrix(c_a)
p_a = np.array(find_min_path(m_a))

s1_b = np.array([1, 3, 4, 1, 2])-2
s2_b = np.array([-1, 0, 2, 3, 0])-1
# s1_b = normalize(s1_b)
# s2_b = normalize(s2_b)
c_b = np.abs(np.subtract.outer(s1_b, s2_b))
##Call DTW function
m_b = fill_dtw_cost_matrix(c_b)
p_b = np.array(find_min_path(m_b))

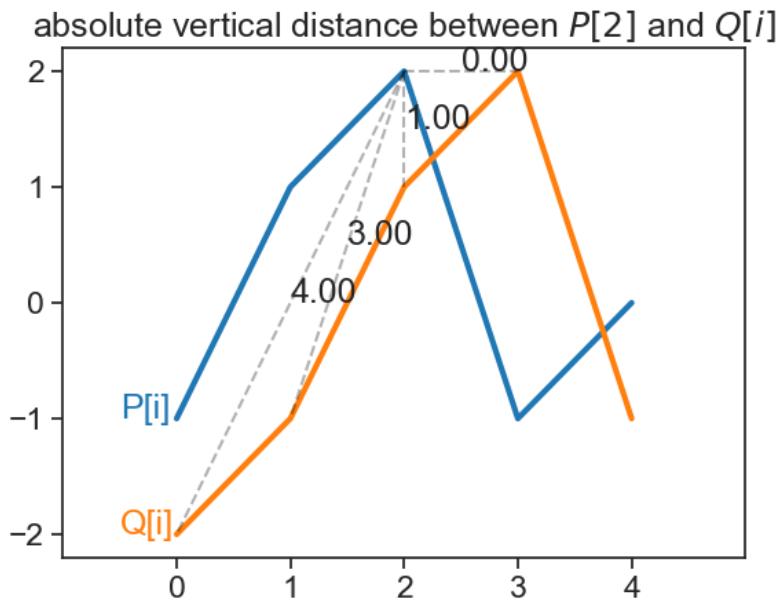
s1_c = np.array([1, 3, 4, 1, 2, 2, 2, 4, 3, 3])
s2_c = np.array([-1, 0, 2, 3, 0, 2, 3, 1, 1, 0])
s1_c = normalize(s1_c)
s2_c = normalize(s2_c)

```

```
c_c = np.abs(np.subtract.outer(s1_c, s2_c))
##Call DTW function
m_c = fill_dtw_cost_matrix(c_c)
p_c = np.array(find_min_path(m_c))
```

Let's plot two time series,  $P$  and  $Q$ .

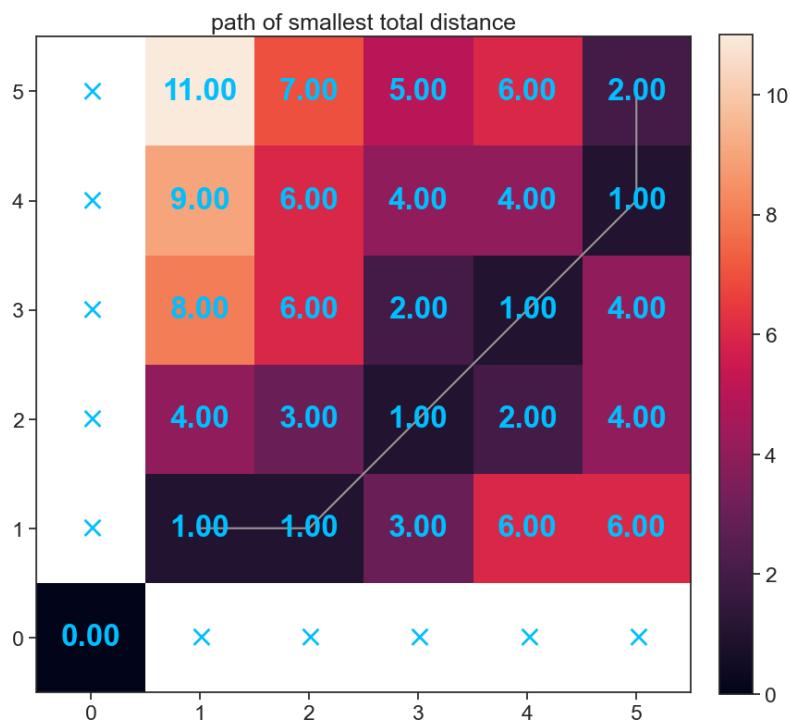
```
plot_series(s1_b, s2_b)
```



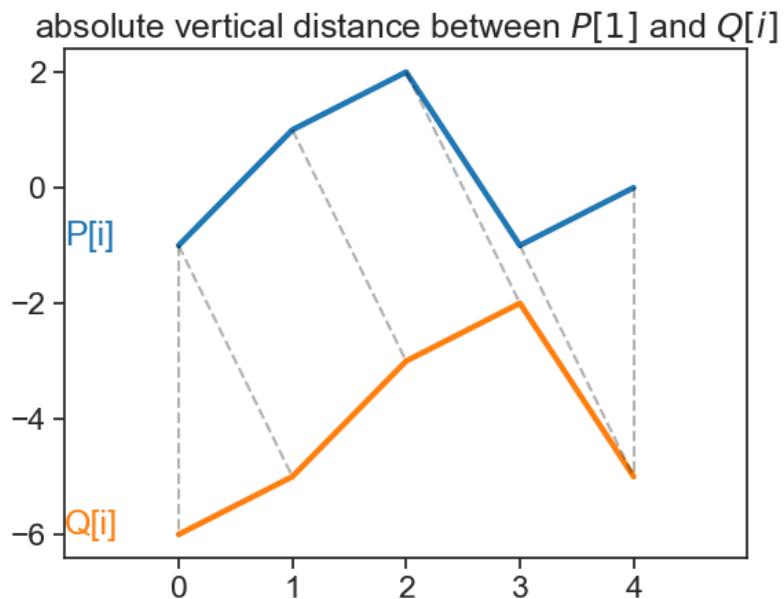
```
plot_abs_diff(s1_b, s2_b, c_b)
```



```
plot_cost_and_path(m_b, p_b)
```

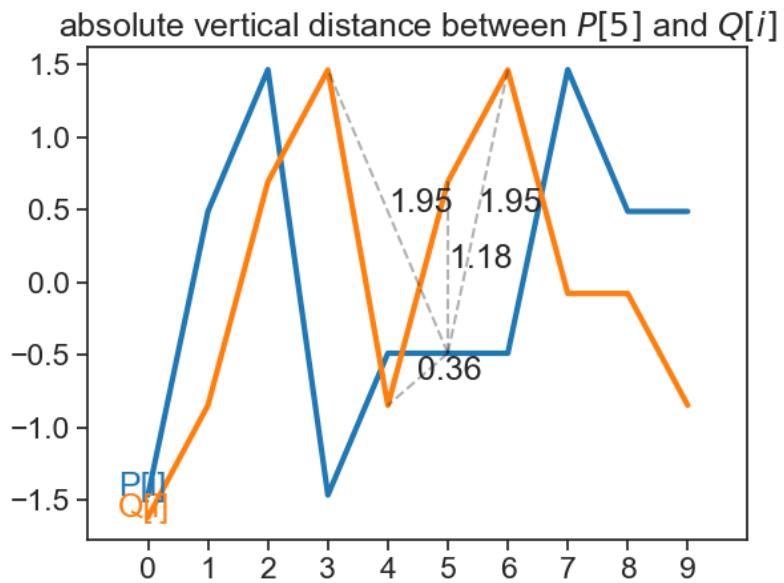


```
plot_warped_series(s1_b, s2_b, p_b)
```



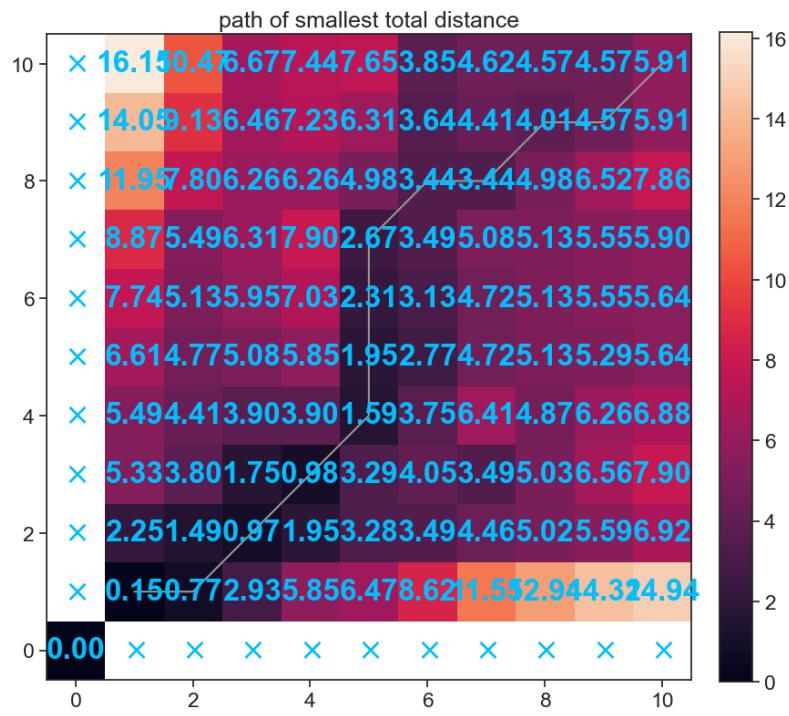
Let's see another example.

```
plot_series(s1_c, s2_c)
```



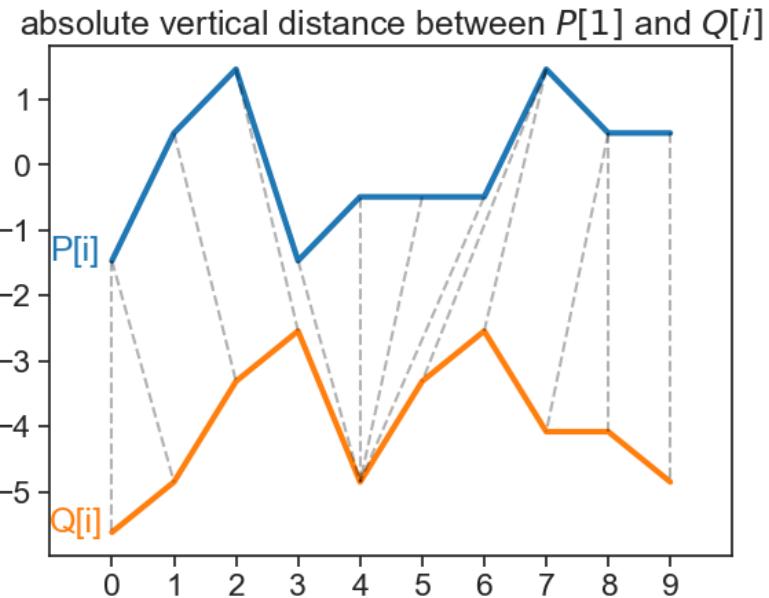
These are exactly the same series from before, with additional data points.

```
plot_cost_and_path(m_c, p_c)
```



Now the path meanders both above and below the diagonal, meaning that the second time series sometimes lags behind (bottom) and sometimes leads ahead (top) the first time series,

```
plot_warped_series(s1_c, s2_c, p_c)
```



## 44.1 causality

```
s1_c = np.array([1, 3, 4, 1, 2, 2, 2, 4, 3, 3])
s2_c = np.array([-1, 0, 2, 3, 0, 2, 3, 1, 1, 0])
s1_c = normalize(s1_c)
s2_c = normalize(s2_c)
c_c = np.abs(np.subtract.outer(s1_c, s2_c))
##Call DTW function
m_c = fill_dtw_cost_matrix(c_c)

N1, N2 = m_c.shape
noncausal = np.tril_indices(n=N1,m=N2, k=-1)
m_c[noncausal] = np.nan

# p_c = np.array(find_min_path(m_c))
```

IndexError: index -12 is out of bounds for axis 1 with size 11

```
m_c
```

```
array([[ 0.          ,         nan,         nan,         nan,         nan,
       nan,         nan,         nan,         nan,         nan,
       nan], [
       nan,  0.15153451,  0.76923077,  2.92538857,  5.85077714,
      6.46847341,  8.62463121, 11.55001978, 12.93694681, 14.32387384,
     14.94157011], [
       nan,         nan,  1.48563839,  0.97358843,  1.94717685,
      3.28128073,  3.48563839,  4.45922681,  5.02409993,  5.58897304,
     6.92307692], [
       nan,         nan,         nan,  1.74513084,  0.97590007,
      3.28590403,  4.05282315,  3.48795004,  5.02872322,  6.56487311,
     7.898977 ], [
       nan,         nan,         nan,         nan,  3.90128864,
      1.59359634,  3.74975414,  6.41333861,  4.87487707,  6.2618041 ,
     6.87950036], [
       nan,         nan,         nan,         nan,         nan,
      1.95180015,  2.77385406,  4.72334256,  5.13436952,  5.28590403,
     5.64410784], [
       nan,         nan,         nan,         nan,         nan,
       nan,  3.13205787,  4.72334256,  5.13436952,  5.54539648,
     5.64410784], [
       nan,         nan,         nan,         nan,         nan,
       nan,         nan,  5.08154637,  5.13436952,  5.54539648,
     5.90360029], [
       nan,         nan,         nan,         nan,         nan,
       nan,         nan,         nan,  4.98283502,  6.5236082 ,
     7.85540044], [
       nan,         nan,         nan,         nan,         nan,
       nan,         nan,         nan,         nan,  4.57180806,
     5.90591194], [
       nan,         nan,         nan,         nan,         nan,
       nan,         nan,         nan,         nan,         nan,
     5.90591194]])
```

```
plot_abs_diff(s1_b, s2_b, c_b)
```

```

##Fill DTW Matrix
def fill_dtw_cost_matrix_causal(abs_diff):
    N1, N2 = abs_diff.shape
    cost_matrix = abs_diff.copy() # make a copy
    # prepend row full of nan
    cost_matrix = np.vstack([np.full(N1, np.nan), cost_matrix])
    # prepend col full of nan
    cost_matrix = np.hstack([np.full((N2+1,1), np.nan), cost_matrix])
    # make origin zero
    cost_matrix[0, 0] = 0

    for i in range(1, N1+1):
        for j in range(1, N2+1):
            south      = cost_matrix[i-1, j]
            west       = cost_matrix[i, j-1]
            southwest = cost_matrix[i-1, j-1]
            # print(([south, west, southwest]))
            # print(np.nanmin([south, west, southwest]))
            cost_matrix[i, j] = cost_matrix[i, j] + np.nanmin([south, west, southwest])
    return cost_matrix

```

```

##Call DTW function
m_a_causal = fill_dtw_cost_matrix_causal(c_a)
p_a_causal = np.array(find_min_path(m_a))

```

```
/var/folders/kv/9cqw3y_s6c75xmgqm9n0t5d40000gn/T/ipykernel_29281/239877799.py:21: RuntimeWarning:
cost_matrix[i, j] = cost_matrix[i, j] + np.nanmin([south, west, southwest])
```

```

fig, ax = plt.subplots(figsize=(10,10))

m = ax.imshow(m_a_causal, origin="lower")
# plt.colorbar(m, ax=ax, shrink=1.0)
plt.colorbar(m,fraction=0.046, pad=0.04)

text_dict = {'ha':'center', 'va':'center', 'color':"deepskyblue",
            'fontsize':24, 'weight':'bold'}

# https://stackoverflow.com/a/33829001
for (j,i),label in np.ndenumerate(m_a_causal):

```

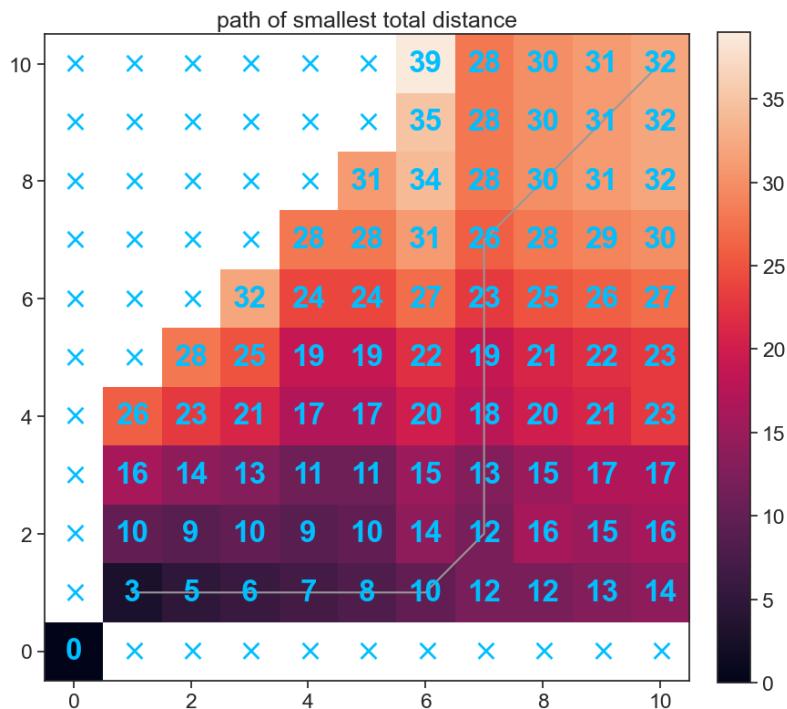
```

if np.isnan(label):
    label = r"\$\\times\$"
else: label = f"{label:.0f}"
ax.text(i,j,label, **text_dict, zorder=10)

ax.plot(p_a_causal[:,1], p_a_causal[:,0], color=[0.6]*3)

ax.set(title="path of smallest total distance")

```



```

fig, ax = plt.subplots()

ax.plot(s1_a, color="tab:blue", lw=3)
ax.plot(s2_a, color="tab:orange", lw=3)

p1 = p_a_causal - 1

for (i,j) in p1:
    ax.plot([i, j], [s1_a[i], s2_a[j]], color="black", alpha=0.3, ls="--")

```

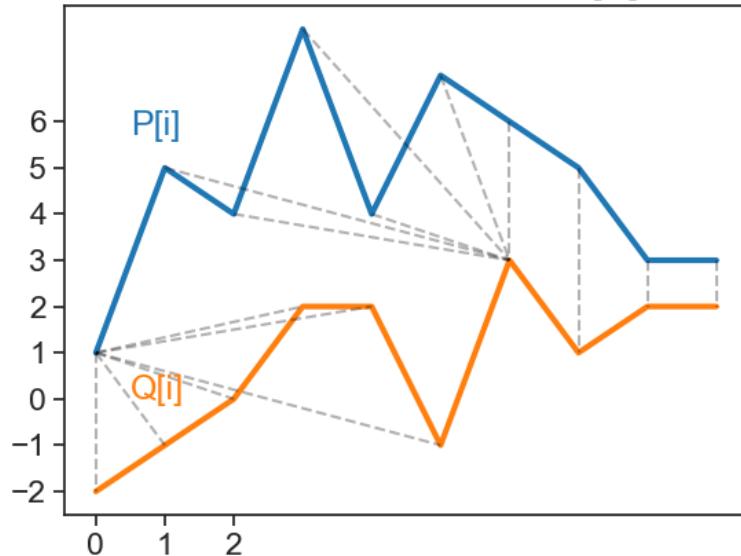
```

ax.text(0.5, 5.7, "P[i]", color="tab:blue")
ax.text(0.5, 0.0, "Q[i]", color="tab:orange")

ax.set(xticks=np.arange(0,3),
       yticks=np.arange(-2,7),
       title=r"absolute vertical distance between $P[1]$ and $Q[i]$");

```

absolute vertical distance between  $P[1]$  and  $Q[i]$



```

a = 1.0*np.arange(16).reshape(4, 4)
# iu1 = np.triu_indices(4)
iu1 = np.tril_indices(n=4,m=5, k=-1)
a[iu1] = np.nan
a

```

```

array([[ 0.,  1.,  2.,  3.],
       [nan,  5.,  6.,  7.],
       [nan, nan, 10., 11.],
       [nan, nan, nan, 15.]])

```

## 45 time lags practice

Download this zip file before you start.

hummus 1

hummus 2

hummus 1 warped

hummus 2 warped

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.ensemble import RandomForestRegressor
import concurrent.futures
from datetime import datetime, timedelta
from scipy.signal import savgol_filter
from scipy import signal
from statsmodels.tsa.seasonal import seasonal_decompose

from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import datetime as dt
import matplotlib.ticker as ticker

# %matplotlib widget
```

The dataset we will work with has two columns:

1. `tree` - dendrometer data of a Jerusalem pine tree
2. `solar_elevation` - solar elevation at the site of the tree.  
As the sun rises the elevation increases and it sets as the elevation decreases. When elevation is negative then the sun is below the horizon.

The sample rate of the data 30min per row.

```
df = pd.read_csv('dendrometer.csv', index_col='time', parse_dates=True)
df
```

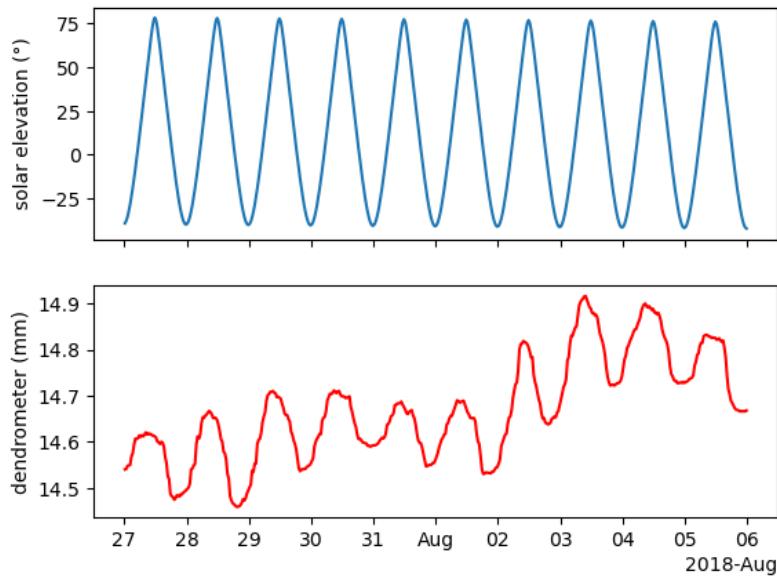
time	tree	solar_elevation
2018-07-27 00:15:00	14.539567	-38.908466
2018-07-27 00:45:00	14.540535	-37.459337
2018-07-27 01:15:00	14.547314	-35.097718
2018-07-27 01:45:00	14.548283	-31.935865
2018-07-27 02:15:00	14.549250	-28.097681
...	...	...
2018-08-05 21:45:00	14.666420	-33.782927
2018-08-05 22:15:00	14.665451	-37.139744
2018-08-05 22:45:00	14.666420	-39.685352
2018-08-05 23:15:00	14.665451	-41.290907
2018-08-05 23:45:00	14.667387	-41.861365

```
fig, ax = plt.subplots(2,1, sharex=True)

ax[0].plot(df['solar_elevation'])
ax[1].plot(df['tree'], c='r')

ax[0].set_ylabel('solar elevation (°)')
ax[1].set_ylabel('dendrometer (mm)')

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
```



The tree contracts as water is leaving the trunk for transpiration. This usually happens during the day. Then the tree expands back at night when there is no transpiration. This is a slow process and there is some lag between the atmospheric conditions and the tree movement. We will quantify this lag.

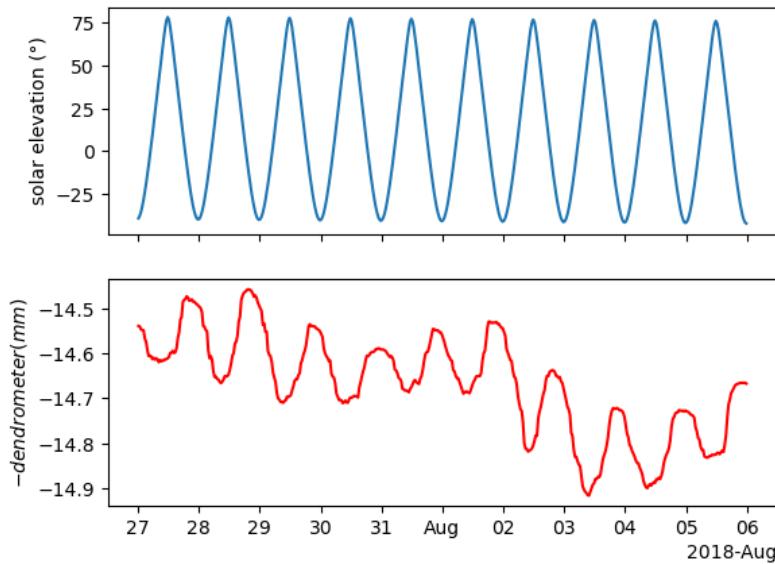
The first thing we will do is multiply the dendrometer data by -1 so it will move in the same direction as the sun elevation. This is just to make things more intuitive when applying cross correlation.

```
fig, ax = plt.subplots(2,1, sharex=True)

ax[0].plot(df['solar_elevation'])
ax[1].plot(df['tree']*-1, c='r')

ax[0].set_ylabel('solar elevation (°)')
ax[1].set_ylabel('$- dendrometer (mm)$')

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
```



The dendrometer gives us the tree's circumference, but in this case we care more about the rate of change - is the tree expanding or contracting? how fast is it moving?

We can answer these questions by computing the derivative of the dendrometer. A simple way to get a derivative from a time series is to compute the differences, as we learned before using the function `.diff()`. Another thing worth mentioning is that the original time series is not stationary, as we learned, differencing can aid in making it stationary.

```
df['tree_diff'] = df['tree'].diff()

fig, ax = plt.subplots(2,1, sharex=True)

ax[0].plot(df['solar_elevation'])
ax[1].plot(-1*df['tree_diff'], c='r')

ax[0].set_ylabel('solar elevation (°)')
ax[1].set_ylabel(r'$-\frac{\Delta \text{dendo}}{\Delta t} (\text{mm})$')

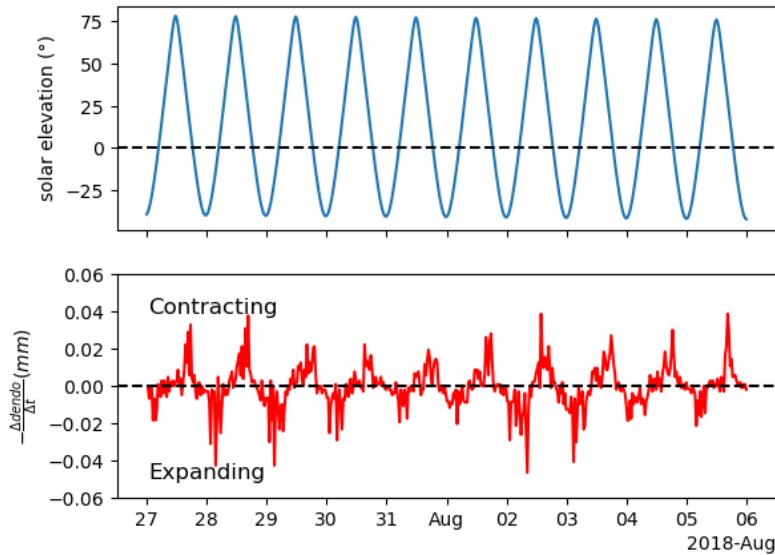
ax[0].axhline(0, c='black', ls='--')
ax[1].axhline(0, c='black', ls='--')
```

```

ax[1].text(df.index[1], 0.037, s='Contracting', ha='left', va='bottom', fontsize=12)
ax[1].text(df.index[1], -0.052, s='Expanding', ha='left', va='bottom', fontsize=12)
ax[1].set_ylim(-0.06,0.06)

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)

```



Now the time series is stationary but the results are a bit noisy. That is because there is some noise in the original signal that are more apparent after computing the differences. If we want a smoother derivative, we need to apply smoothing to the data before. A practical solution is using the `scipy.signal.savgol_filter()` which has a derivative argument. By default, `deriv=0` so we don't get the derivative. But we can change it to whatever derivative order we want. We want the first derivative so we will use `deriv=1`. That way we get both smoothing and derivative using one function.

```
df['tree_der'] = savgol_filter(df['tree'], 24, 3, deriv=1)*-1
```

```

fig, ax = plt.subplots(2,1, sharex=True)

ax[0].plot(df['solar_elevation'])
ax[1].plot(df['tree_der'], c='r')

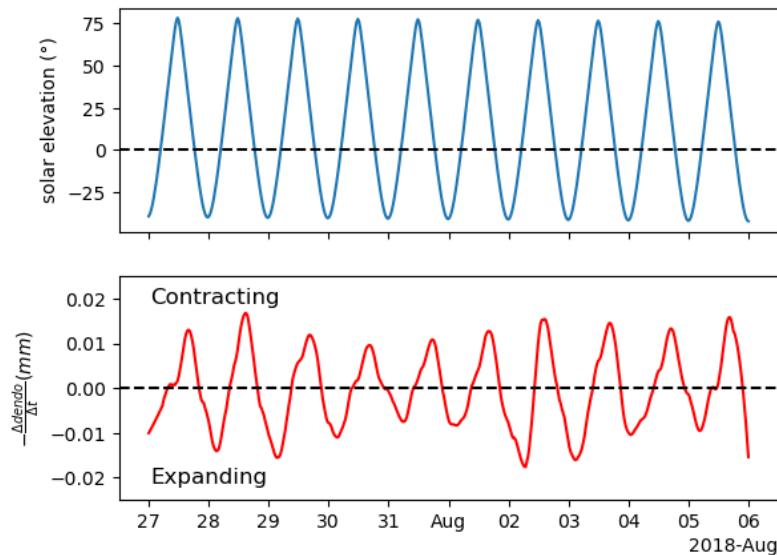
ax[0].set_ylabel('solar elevation (°)')
ax[1].set_ylabel(r'$-\frac{\Delta \text{dendo}}{\Delta t} (\text{mm})$')

ax[0].axhline(0, c='black', ls='--')
ax[1].axhline(0, c='black', ls='--')

ax[1].text(df.index[1], 0.018, s='Contracting', ha='left', va='bottom', fontsize=12)
ax[1].text(df.index[1], -0.022, s='Expanding', ha='left', va='bottom', fontsize=12)
ax[1].set_ylim(-0.025,0.025)

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)

```



Now the signal is nice and smooth.  
The next thing we want to do is apply standardization as the values of the two time series are very different.

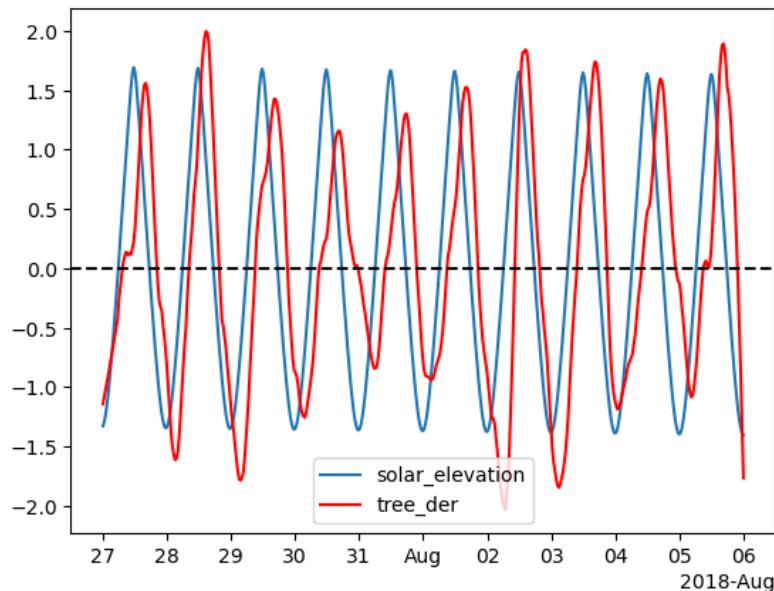
```
# creating a new standardized df
df_z = (df - df.mean())/df.std()
```

Now that we applied standardization we can plot them on the same ax.

```
fig, ax = plt.subplots()

ax.plot(df_z['solar_elevation'], label='solar_elevation')
ax.plot(df_z['tree_der'], c='r', label='tree_der')
ax.axhline(0, c='black', ls='--')
ax.legend()

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)
```



Now that the data is all processed we can work on finding the lag. lets apply cross correlation.

```

series1 = df_z['solar_elevation']
series2 = df_z['tree_der']

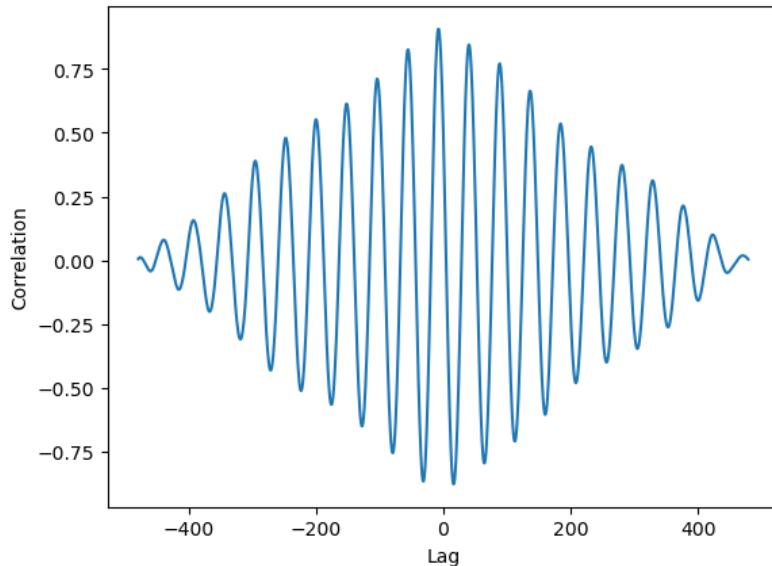
correlation = signal.correlate(series1, series2, mode='full', method='fft')
lags = signal.correlation_lags(len(series1), len(series2))
correlation /= (len(series1))

result_series = pd.Series(correlation, index=lags)

fig, ax = plt.subplots()

ax.plot(result_series)
ax.set(xlabel='Lag', ylabel='Correlation')

```



We know that the tree is lagging (not the opposite) and we know that the lag can't be more than 24hrs (48 points in our case). So let's limit the plot to the relevant range.

```

filtered_series = result_series[(result_series.index > -48) & (result_series.index <= 0)]

fig, ax = plt.subplots()

ax.plot(filtered_series)

```

```

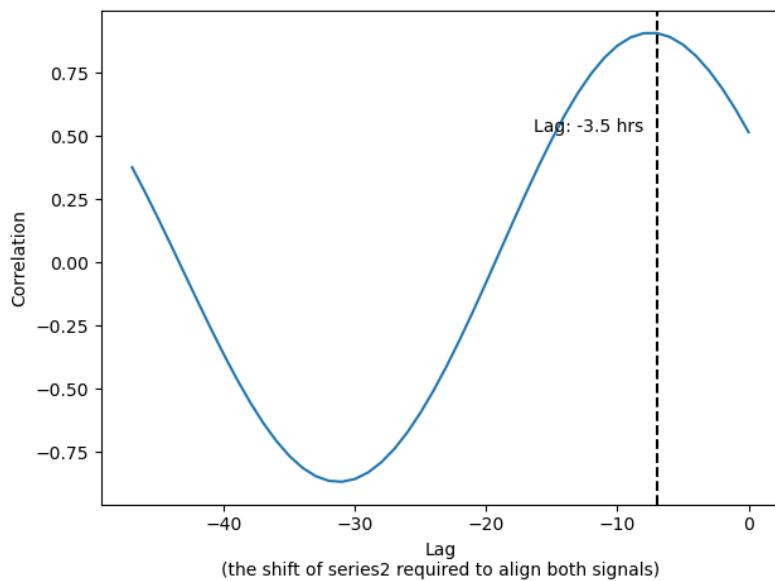
# ax.axvline(filtered_series.idxmax(), c='black', ls='--')

max_index = filtered_series.idxmax()
ax.axvline(max_index, c='black', ls='--')

max_value = filtered_series[max_index]
ax.text(max_index-1, 0.5, f'Lag: {max_index/2} hrs', ha='right', va='bottom')

ax.set(xlabel='Lag\n(the shift of series2 required to align both signals)', ylabel='Correlation')
plt.tight_layout()

```



Note that we got a lag, we can shift the data and inspect it.

```

# Shift column B back by 3.5 hours
df_z['tree_der_shifted'] = df_z['tree_der'].shift(int(max_index))

fig, ax = plt.subplots()

ax.plot(df_z['solar_elevation'], label='solar_elevation')
ax.plot(df_z['tree_der_shifted'], c='r', label='tree_der_shifted')
ax.axhline(0, c='black', ls='--')
ax.legend()

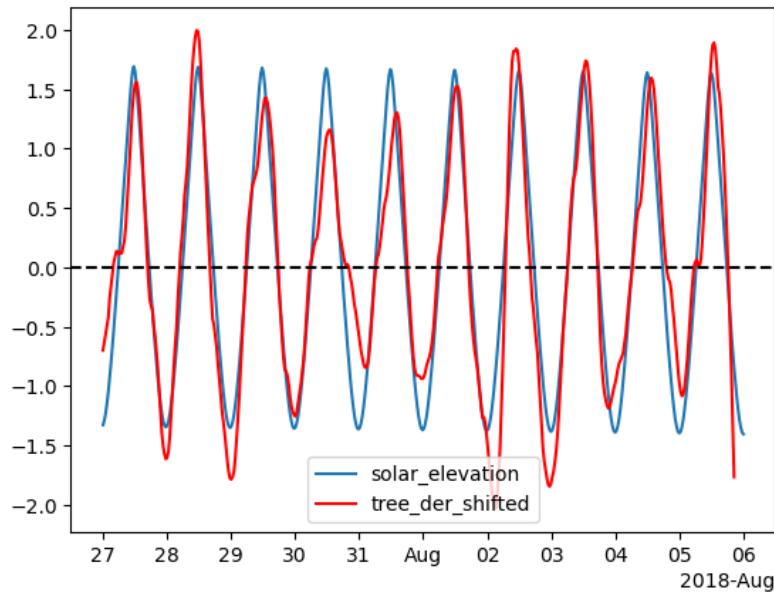
```

```

# ax.set_xlim(pd.Timestamp('2018-07-30'), pd.Timestamp('2018-08-01'))

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

```



Any thoughts of the results? Is it perfect? What is the resolution of our shift?

We can increase the resolution to get a more accurate shift by upsampling the data.

```

df_z_resampled = df_z.resample('T').interpolate()

series1 = df_z_resampled['solar_elevation']
series2 = df_z_resampled['tree_der']

correlation = signal.correlate(series1, series2, mode='full', method='fft')
lags = signal.correlation_lags(len(series1), len(series2))
correlation /= (len(series1))

result_series = pd.Series(correlation, index=lags)

```

```

filtered_series = result_series[(result_series.index > -60*12) & (result_series.index < 60*12)]

fig, ax = plt.subplots()

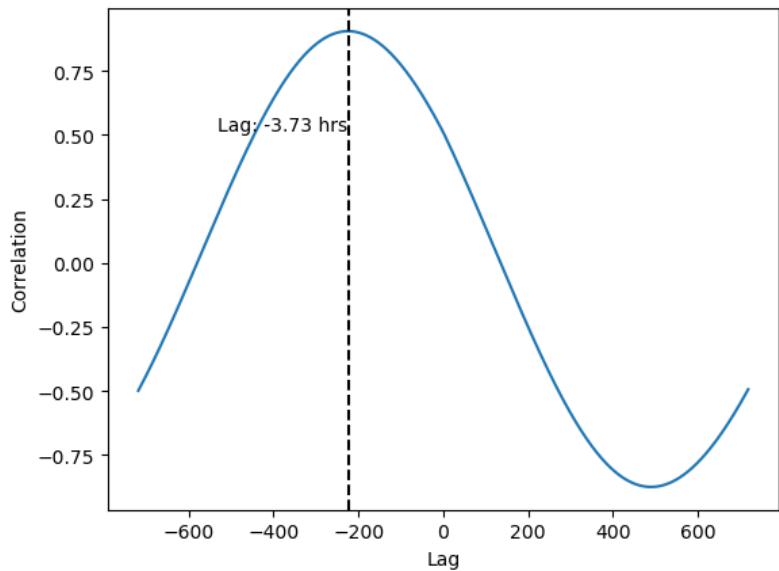
ax.plot(filtered_series)
# ax.axvline(filtered_series.idxmax(), c='black', ls='--')

max_index = filtered_series.idxmax()
ax.axvline(max_index, c='black', ls='--')

max_value = filtered_series[max_index]
ax.text(max_index-1, 0.5, f'Lag: {max_index/60:.2f} hrs', ha='right', va='bottom')

ax.set(xlabel='Lag', ylabel='Correlation')

```



Now we see that the lag is actually greater.

```

# Shift column B back by 3.5 hours
df_z_resampled['tree_der_shifted'] = df_z_resampled['tree_der'].shift(max_index)

fig, ax = plt.subplots()

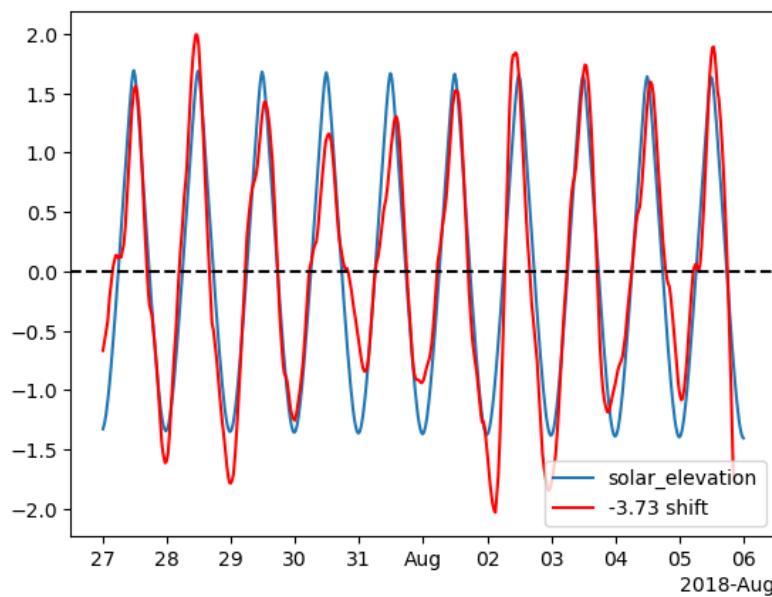
```

```

ax.plot(df_z_resampled['solar_elevation'], label='solar_elevation')
ax.plot(df_z_resampled['tree_der_shifted'], c='r', label='-3.73 shift')
ax.axhline(0, c='black', ls='--')
ax.legend()
# ax.set_xlim(pd.Timestamp('2018-07-30'), pd.Timestamp('2018-08-01'))

locator = mdates.AutoDateLocator(minticks=7, maxticks=11)
formatter = mdates.ConciseDateFormatter(locator)
ax.xaxis.set_major_locator(locator)
ax.xaxis.set_major_formatter(formatter)

```



## 46 Dynamic Time Warping (DTW)

We can apply DTW to our data to find alignments between the time series.

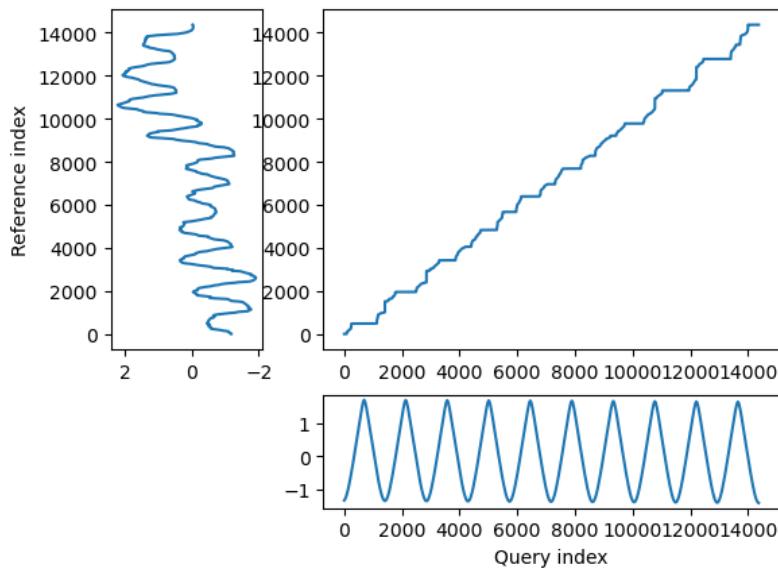
```
from dtw import *

alignment = dtw(df_z_resampled['solar_elevation'], df_z_resampled['tree'], keep_internals=True)
```

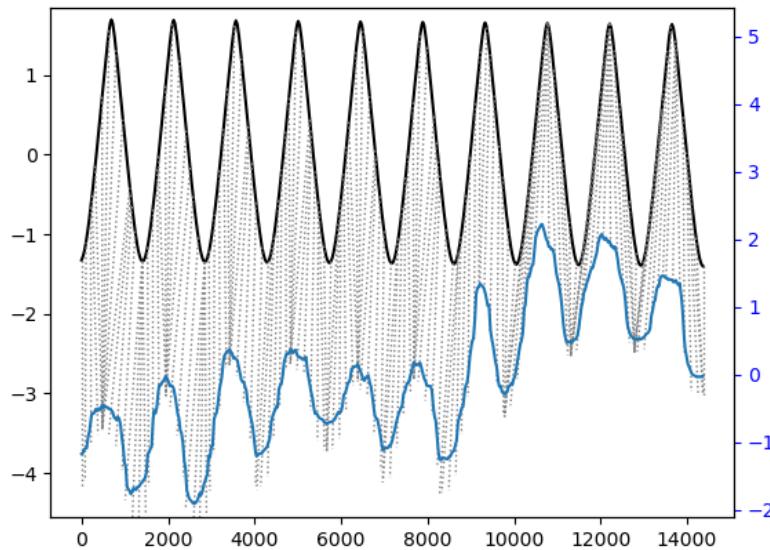
Importing the dtw module. When using in academic works please cite:

T. Giorgino. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package J. Stat. Soft., doi:10.18637/jss.v031.i07.

```
alignment.plot('threeway')
```



```
## Align and plot with the Rabiner-Juang type VI-c unsmoothed recursion
alignment.plot(type="twoway", offset=-3, match_indices=150)
```



As you can see it aligned the peaks pretty well. But in general, cross correlation was better for this kind of analysis.

## 46.1 Sound and DTW

Dynamic Time Warping (DTW) excels in aligning two time series that may vary in their rhythm or speed. This characteristic makes it particularly useful in applications such as speech recognition, where the goal is to identify specific words or phrases within audio recordings, despite variations in how different speakers might pronounce them. For instance, consider the task of recognizing the word “backpack” in various sound recordings. You might have a reference recording of someone clearly saying “backpack,” which serves as your baseline. When you attempt to find occurrences of this word in recordings of other people speaking, you’ll likely encounter variations in the rhythm — some may say it faster or slower than your reference, or even separate the words - “Back pack”.

DTW addresses this challenge by allowing a flexible, non-linear alignment between the time series. This flexibility enables DTW to effectively match sequences that have the same underlying patterns but are expressed over different time scales. Therefore, even if different speakers say “backpack” with varying speeds, DTW can align these variations with the reference, making it possible to accurately recognize the word across different speech rhythms.

#### 46.1.1 “I like to eat Hummus”

That's the sentence we will work with today. We have two recordings of me saying it. In every recording I say it a bit different. We will use DTW to align the words of the two recordings and even warp the every recording to sound like the other.

Start by listening to these two files: 1. `hummus1.wav` 1. `hummus2.wav`

Sound files are basically time series data. So we can import them as np arrays.

```
from scipy.io import wavfile  
  
samplerate, data1 = wavfile.read('hummus1.wav')  
  
samplerate
```

48000

Sample rate is the number of samples per second. 48kHz is a standard sample rate for audio files.

```
data1.shape
```

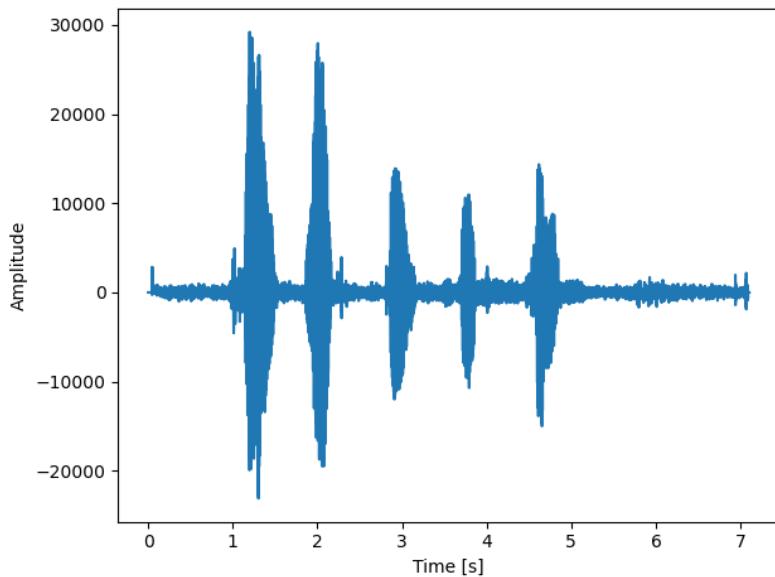
(340992, 2)

Note that there are two channels, as the recording is stereo. We will work with only the first channel (left)

```
length = data1.shape[0] / samplerate
time = np.linspace(0., length, data1.shape[0])

fig, ax = plt.subplots()
ax.plot(time, data1[:, 0])
ax.set_xlabel("Time [s]")
ax.set_ylabel("Amplitude")

plt.tight_layout()
```



Now lets import the second recording

```
samplerate, data2 = wavfile.read('hummus2.wav')
```

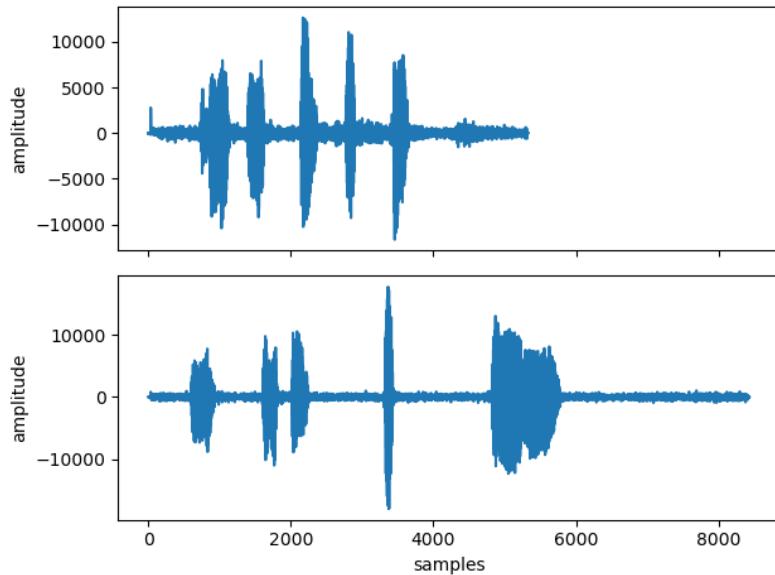
As you can see above these time series are huge, 7 seconds of data translate to 336,000 data points (and that's only one channel). In order to speed up computation time we will resample the data so make the series smaller. **Note, this will drastically reduce the audio quality**

```
resample_factor = 64
hummus1 = data1[:, 0]
hummus2 = data2[:, 0]
hummus1_rs = signal.resample(hummus1, int(len(hummus1)/resample_factor))
hummus2_rs = signal.resample(hummus2, int(len(hummus2)/resample_factor))
```

```
fig, ax = plt.subplots(2,1, sharex=True)
ax[0].plot(hummus1_rs)
ax[1].plot(hummus2_rs)

ax[1].set(xlabel='samples', ylabel='amplitude')
ax[0].set( ylabel='amplitude')

plt.tight_layout()
```



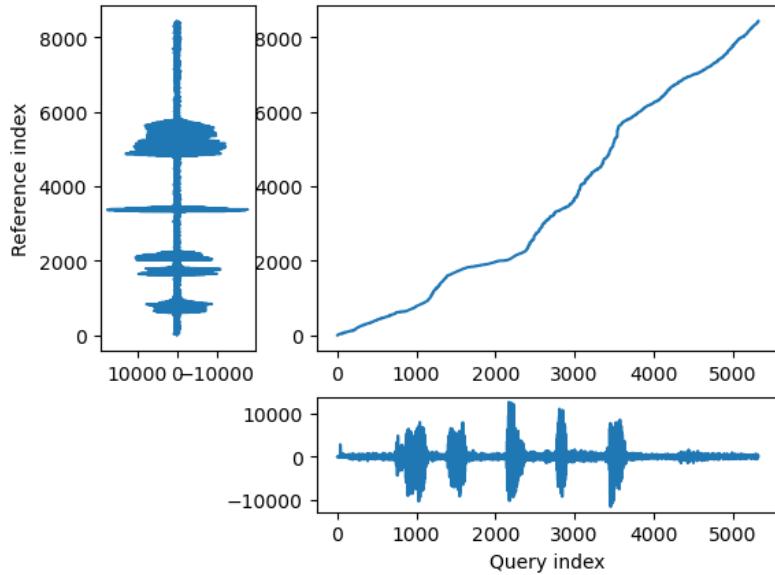
Implementation of DTW will be by using the dtw-python package. Install dtw-python in anaconda using `conda install conda-forge::dtw-python`. Documentation can be found [here](#)

```
from dtw import *
```

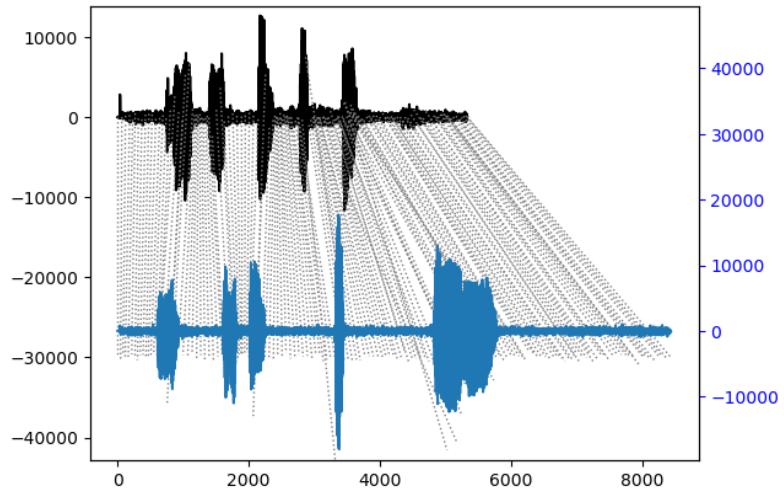
```
# aligning the two time series
alignment = dtw(hummus1_rs, hummus2_rs, keep_internals=True)
```

The package has built in functions for plotting the results.

```
alignment.plot('threeway')
```



```
## Align and plot with the Rabiner-Juang type VI-c unsmoothed recursion
alignment.plot(type="twoway", offset=-30000, match_indices=150)
```



In the few cells below we warp each recording to match the rhythm of the other recording.

first we get the warped index. If we are warping A to match the rhythm of B, then the `warped_index` will be an array in the length of B where each position has the matching index of A.

```
hummus1_warped_index = warp(alignment, index_reference=False)
hummus2_warped_index = warp(alignment, index_reference=True)
```

```
hummus1_warped_index
```

```
array([    0,     0,     0, ..., 5325, 5326, 5327])
```

```
len(hummus1_warped_index)
```

8431

Below is a function for upsampling as we want to go back to the original sample rate so we can hear back the warped recording.

```

def upsampling(array, factor):
    # initialize the upsampled array with NaNs
    us = np.full(int(len(array) * factor), np.nan)
    # assign original array values to their new positions
    us[::factor] = array*factor
    # find indices of the NaNs and the non-NaN (original values)
    nans = np.isnan(us)
    non_nans = ~nans
    # interpolate using np.interp
    # for x-coordinates, use np.arange(len(us)) for full array
    # for xp-coordinates (where we have actual values), use the non-NaN indices
    # for fp-coordinates (actual values to interpolate), use the non-NaN values in us
    us[nans] = np.interp(np.flatnonzero(nans), np.flatnonzero(non_nans), us[non_nans])
    return us.astype(int)

```

```

hummus1_warped_index_us = upsampling(hummus1_warped_index, resample_factor)
hummus2_warped_index_us = upsampling(hummus2_warped_index, resample_factor)

```

```
hummus1_warped_index_us.shape
```

(539584,)

Below, we replace the index values of `warped_index` with their corresponding amplitude values.

```

hummus1_warped = hummus1[hummus1_warped_index_us]
hummus2_warped = hummus2[hummus2_warped_index_us]

```

```

complet = [hummus1, hummus2_warped, hummus2, hummus1_warped]
labels = ['hummus1', 'hummus2_warped', 'hummus2', 'hummus1_warped']

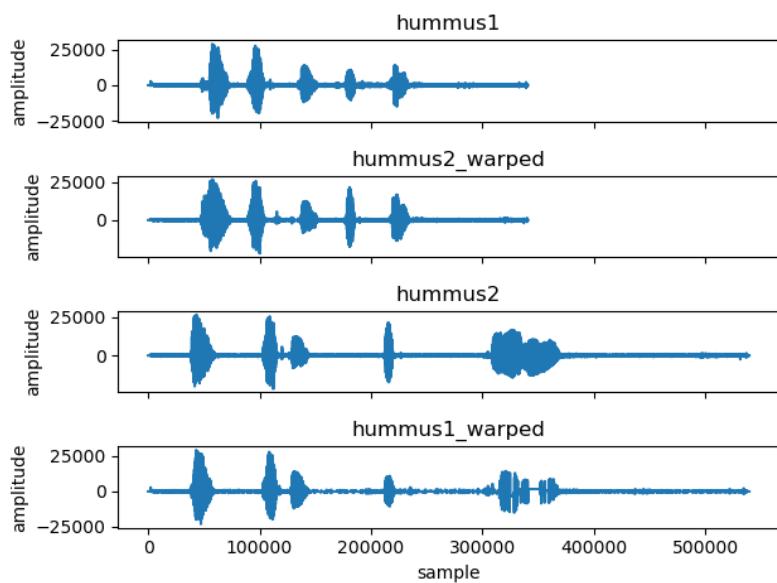
fig, ax = plt.subplots(4,1, sharex=True)

for i, hummus in enumerate(complet):
    ax[i].plot(hummus)
    ax[i].set_title(labels[i])
    ax[i].set_ylabel('amplitude')
    print()

```

```
ax[3].set_xlabel('sample')

plt.tight_layout()
```



As you can see above the DTW algorithm did a great job.  
Now lets export it back to an audio file and listen. Don't forget  
that we downsampled and then upsampled so the quality will  
be bad, **but what matters now is the rhythm**.

```
from scipy.io.wavfile import write

write("hummus1_warped.wav", samplerate, hummus1_warped.astype(np.int16))
write("hummus2_warped.wav", samplerate, hummus2_warped.astype(np.int16))
```

# **Part VIII**

# **frequency**

## 47 motivation

I used multiple sources for this lesson. The most noteworthy are these.

[Jez Swanson's "An Interactive Introduction to Fourier Transforms".](#)

Truly amazing website. Start here.

[Liz O'Gorman's "Decomposing Fourier transforms — an introduction to time-frequency decomposition"](#)

This is excellent. A balanced combination of theory and coding for the beginner.

# 48 Fourier transform

## 48.1 basic wave concepts

The function

$$f(t) = B \sin(2\pi ft) \quad (48.1)$$

has two basic characteristics, its amplitude  $B$  and frequency  $f$ .

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
sns.set(style="ticks", font_scale=1.5)

# %matplotlib widget

# Configure Matplotlib to use LaTeX font
plt.rcParams.update({
    "xtick.labelsize": 14,
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": ["Computer Modern Roman"]
})
```

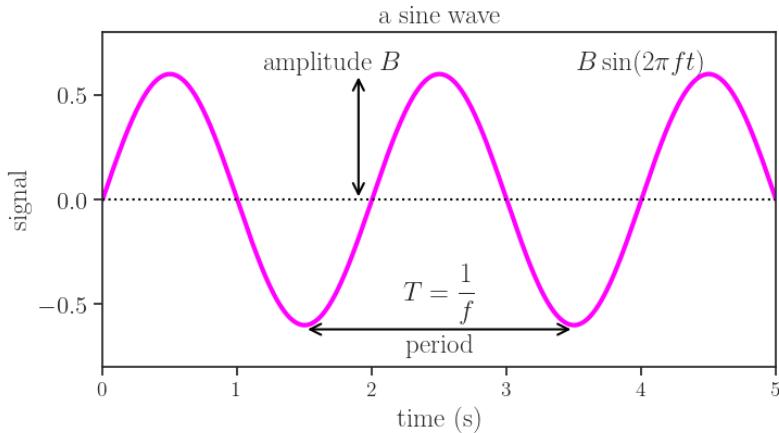
```
T = 2.0 # s
f = 1.0 / T
n_periods = 2.5
dt = 0.01
B = 0.6
t = np.arange(0, T*n_periods + dt, dt)
```

```

s = B * np.sin(2.0 * np.pi * f * t)
c = B * np.cos(2.0 * np.pi * f * t)

fig, ax = plt.subplots(figsize=(8,4))
ax.plot(t, s, color="magenta", lw=3)
ax.plot(t, 0*t, color="black", ls=":")
ax.annotate("", xy=(1.9, 0.0), xycoords='data',
            xytext=(1.9, 0.6), textcoords='data',
            color="black",
            arrowprops=dict(arrowstyle "<->",
                            connectionstyle="arc3",
                            color="black",
                            lw=1.5),
            )
ax.annotate("", xy=(1.5, -0.62), xycoords='data',
            xytext=(3.5, -0.62), textcoords='data',
            color="black",
            arrowprops=dict(arrowstyle "<->",
                            connectionstyle="arc3",
                            color="black",
                            lw=1.5),
            )
ax.text(1.7, 0.62, r"amplitude $B$",
        ha="center")
ax.text(2.5, -0.73, r"period", ha="center")
ax.text(2.5, -0.48, r"$T=\displaystyle\frac{1}{f}$", ha="center")
ax.text(4.0, 0.62, r"$B\sin(2\pi ft)$", ha="center")
ax.set(xlim=[t[0], t[-1]],
       ylim=[-0.8, 0.8],
       title="a sine wave",
       xlabel="time (s)",
       ylabel="signal");
# fig.savefig("sine1.png", dpi=300, bbox_inches="tight")

```



In the figure above, the amplitude  $B = 0.6$  and we see that the distance between two peaks is called period,  $T = 2$  s. The frequency is defined as the inverse of the period:

$$f = \frac{1}{T}. \quad (48.2)$$

When time is in seconds, then the frequency is measured in Hertz (Hz). For the graph above, therefore, we see a wave whose frequency is  $f = 1/(2 \text{ s}) = 0.5 \text{ Hz}$ .

In the figure below, we see what happens when we vary the values of the frequency and amplitude.

```
t20b = mpl.colormaps['tab20b']
t20c = mpl.colormaps['tab20c']

fig, ax = plt.subplots(3, 1, figsize=(8,4), sharex=True)
fig.subplots_adjust(
    hspace=0.0, wspace=0.02)
A = 0.4

s = A * np.sin(2.0 * np.pi * (1/2.0) * t)
ax[0].plot(t, s, color=t20b.colors[16], lw=3)
s = A * np.sin(2.0 * np.pi * (1/0.8) * t) - 1
ax[0].plot(t, s, color=t20b.colors[17], lw=3)
s = A * np.sin(2.0 * np.pi * (1/0.3) * t) - 2
ax[0].plot(t, s, color=t20b.colors[18], lw=3)
```

```

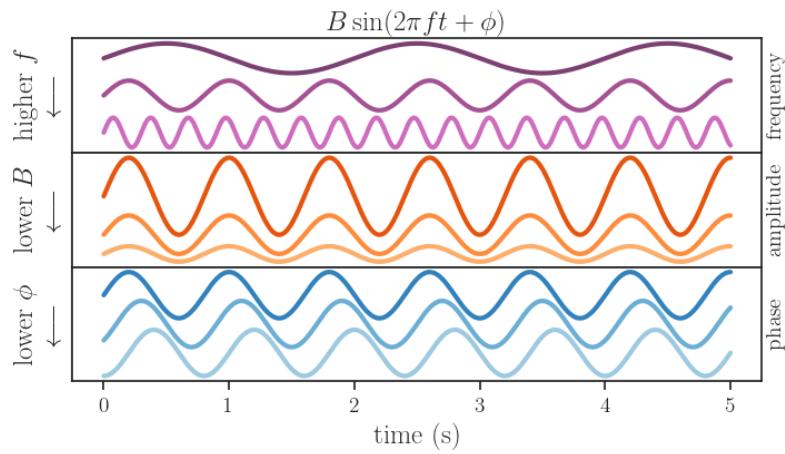
s = 1 * np.sin(2.0 * np.pi * (1/0.8) * t)
ax[1].plot(t, s, color=t20c.colors[4], lw=3)
s = 0.5 * np.sin(2.0 * np.pi * (1/0.8) * t) - 1
ax[1].plot(t, s, color=t20c.colors[5], lw=3)
s = 0.2 * np.sin(2.0 * np.pi * (1/0.8) * t) - 1.5
ax[1].plot(t, s, color=t20c.colors[6], lw=3)

s = A * np.sin(2.0 * np.pi * (1/0.8) * t)
ax[2].plot(t, s, color=t20c.colors[0], lw=3)
s = A * np.sin(2.0 * np.pi * (1/0.8) * t - np.pi/4) - 0.5
ax[2].plot(t, s, color=t20c.colors[1], lw=3)
s = A * np.sin(2.0 * np.pi * (1/0.8) * t - 2*np.pi/4) - 1
ax[2].plot(t, s, color=t20c.colors[2], lw=3)

ax[0].text(1.02, 0.5, "frequency", transform=ax[0].transAxes,
            horizontalalignment='center', verticalalignment='center',
            fontweight="bold", fontsize=14,
            rotation=90)
ax[1].text(1.02, 0.5, "amplitude", transform=ax[1].transAxes,
            horizontalalignment='center', verticalalignment='center',
            fontweight="bold", fontsize=14,
            rotation=90)
ax[2].text(1.02, 0.5, "phase", transform=ax[2].transAxes,
            horizontalalignment='center', verticalalignment='center',
            fontweight="bold", fontsize=14,
            rotation=90)

ax[0].set(title=r"$B\sin(2\pi ft+\phi)$",
           yticks=[],
           ylabel=r"higher $f$"+r"\longleftarrow")
ax[1].set(yticks=[],
           ylabel=r"lower $B$"+r"\longleftarrow")
ax[2].set(xlabel="time (s)",
           yticks=[],
           ylabel=r"lower $\phi$"+r"\longleftarrow");
# fig.savefig("sine2.png", dpi=300, bbox_inches="tight")

```



The graph above introduces two new characteristics of a wave, its phase  $\phi$ , and its offset  $B$ . A more general description of a sine wave is

$$f(t) = B \sin(2\pi ft + \phi) + B_0. \quad (48.3)$$

The offset  $B_0$  moves the wave up and down, while changing the value of  $\phi$  makes the sine wave move left and right. When the phase  $\phi = 2\pi$ , the sine wave will have shifted a full period, and the resulting wave is identical to the original:

$$B \sin(2\pi ft) = B \sin(2\pi ft + 2\pi). \quad (48.4)$$

All the above can also be said about a cosine, whose general form can be given as

$$A \cos(2\pi ft + \phi) + A_0 \quad (48.5)$$

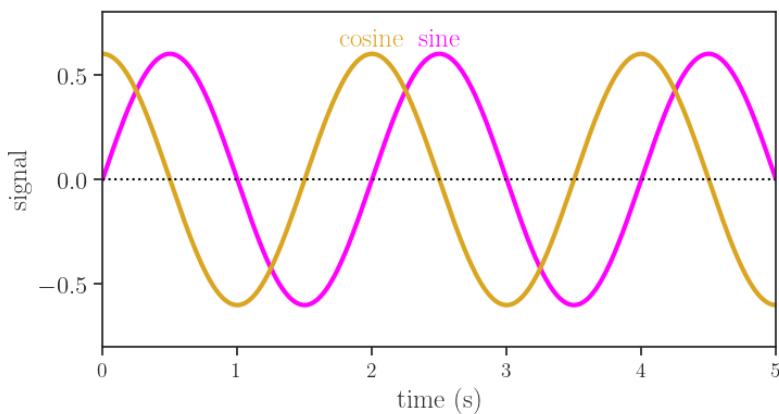
One final point before we jump into the deep waters is that the sine and cosine functions are related through a simple phase shift:

$$\cos\left(2\pi ft + \frac{\pi}{2}\right) = \sin(2\pi ft)$$

```

fig, ax = plt.subplots(figsize=(8,4))
ax.plot(t, s, color="magenta", lw=3)
ax.plot(t, c, color="goldenrod", lw=3)
ax.plot(t, 0*t, color="black", ls=":")
ax.text(2.5, 0.64, "sine", ha="center", color="magenta")
ax.text(2.0, 0.64, "cosine", ha="center", color="goldenrod")
ax.set(xlim=[t[0], t[-1]],
       ylim=[-0.8, 0.8],
       xlabel="time (s)",
       ylabel="signal");

```



## 48.2 Fourier's theorem

Fourier's theorem states that

Any periodic signal is composed of a superposition of pure sine waves, with suitably chosen amplitudes and phases, whose frequencies are harmonics of the fundamental frequency of the signal.

## 48.3 Fourier series

a periodic function can be described as a sum of sines and cosines.

The most common way of representing the Fourier series is

$$f(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt) \quad (48.6)$$

Let's see some excellent visual demonstrations. The classic examples are usually the square function and the sawtooth function:

Source: <https://www.geogebra.org/m/tkajbzmg>

Source: <https://www.geogebra.org/m/k4eq4fkr>

We can take advantage of complex numbers to rewrite the Fourier series in a more compact and elegant way:

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i \frac{n}{P} t}$$

for a periodic function between  $-\frac{P}{2} \leq t \leq \frac{P}{2}$ , and the coefficients are given by

$$c_n = \frac{1}{P} \int_{-\frac{P}{2}}^{\frac{P}{2}} f(t) e^{-2\pi i \frac{n}{P} t} dt.$$

If you are not familiar with complex numbers, or you need a refresher, visit [Dennis Sun's excellent “Introduction to Probability” webpage](#).

The series expressed as a sum of sines and cosines could be translated into an expression of a complex exponential by taking advantage of Euler's formula:

$$e^{ix} = \cos(x) + i \sin(x)$$

for a periodic function  $f(t)$  in the interval  $-\pi < t < \pi$ , where

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) dt \\ a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos(nt) dt \\ b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin(nt) dt \end{aligned} \quad (48.7)$$

## 48.4 Fourier transform

This is a generalization of a Fourier series, but for non-periodic signals. If we take the limit  $P \rightarrow \infty$  in the equations above, we have that

$$f(t) = \int_{-\infty}^{\infty} F(k) e^{2\pi i k t} dk,$$

where  $F(k)$  now takes the role of the coefficients from before, and it is given by

$$F(k) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i k t} dk.$$

If  $t$  is in seconds, the frequency  $k$  is given in Hertz (Hz).

See the following animations to visualize the theorem in action.

Source: [https://en.wikipedia.org/wiki/File:Fourier\\_series\\_and\\_transform.gif](https://en.wikipedia.org/wiki/File:Fourier_series_and_transform.gif)

Source: [https://commons.wikimedia.org/wiki/File:Fourier\\_synthesis\\_square\\_wave\\_animated.gif](https://commons.wikimedia.org/wiki/File:Fourier_synthesis_square_wave_animated.gif)

Source: [https://commons.wikimedia.org/wiki/File:Sawtooth\\_Fourier\\_Animation.gif](https://commons.wikimedia.org/wiki/File:Sawtooth_Fourier_Animation.gif)

Source: [Wikimedia](#)

## 48.5 fun calculation of a Fourier series

Let's calculate the Fourier series of the periodic function  $f(t)$  over the domain  $-\pi < t < \pi$ :

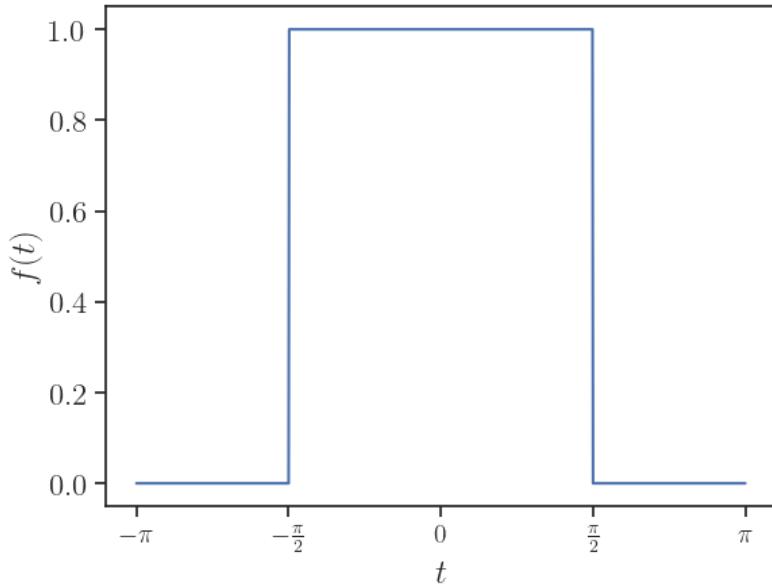
$$f(t) = \begin{cases} 1, & \text{if } -\frac{\pi}{2} < t < \frac{\pi}{2} \\ 0, & \text{otherwise} \end{cases}$$

```

pi = np.pi
t = np.linspace(-pi,pi,1001)
f = np.zeros(len(t))
mask = np.where(np.abs(t)<pi/2)
f[mask] = 1.0

fig, ax = plt.subplots()
ax.plot(t, f)
ax.set(xlabel=r"$t$",
       ylabel=r"$f(t)$",
       xticks=[-pi,-pi/2,0,pi/2,pi],
       xticklabels=[r'$-\pi$',r'$-\frac{\pi}{2}$',r'$0$',r'$\frac{\pi}{2}$',r'$\pi$']);

```



Using the equations of the Fourier series (Equation 48.6 and Equation 48.7), we can calculate the first coefficient  $a_0$ :

$$\begin{aligned}
a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) dt \\
&= \frac{1}{\pi} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} 1 \cdot dt \\
&= \frac{1}{\pi} [t]_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \\
&= \frac{1}{\pi} \left[ \frac{\pi}{2} - \left( -\frac{\pi}{2} \right) \right] \\
&= \frac{1}{\pi} [\pi] \\
&= 1
\end{aligned}$$

The coefficients  $a_n$  are:

$$\begin{aligned}
a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos(nt) dt \\
&= \frac{1}{\pi} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \cos(nt) dt \\
&= \frac{1}{\pi} \left[ \frac{\sin(nt)}{n} \right]_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \\
&= \frac{1}{n\pi} \left[ \sin\left(n\frac{\pi}{2}\right) - \sin\left(-n\frac{\pi}{2}\right) \right] \\
&\text{because } \sin(-a) = -\sin(a) \\
&= \frac{2}{n\pi} \sin\left(n\frac{\pi}{2}\right)
\end{aligned}$$

Finally, the coefficients  $b_n$  are:

$$\begin{aligned}
b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin(nt) dt \\
&= \frac{1}{\pi} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \sin(nt) dt \\
&= \frac{1}{\pi} \left[ -\frac{\cos(nt)}{n} \right]_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \\
&= -\frac{1}{n\pi} \left[ \cos\left(n\frac{\pi}{2}\right) - \cos\left(-n\frac{\pi}{2}\right) \right] \\
&\quad \text{because } \cos(-a) = \cos(a) \\
&= -\frac{1}{n\pi} \left[ \cos\left(n\frac{\pi}{2}\right) - \cos\left(n\frac{\pi}{2}\right) \right] \\
&= 0
\end{aligned}$$

Let's put all this together:

$$\begin{aligned}
f(t) &= \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt) \\
&= \frac{1}{2} + \sum_{n=1}^{\infty} \frac{2}{n\pi} \sin\left(n\frac{\pi}{2}\right) \cos(nt)
\end{aligned}$$

Note that for even values of  $n = 2k$ , we have

$$\sin\left(2k\frac{\pi}{2}\right) = \sin(k\pi) = 0$$

We are left only with the odd values of  $n$ , so let's apply the following substitution:

$$n \rightarrow 2k + 1$$

Then our series for  $f(t)$  can be expressed as

$$f(t) = \frac{1}{2} + \sum_{k=0}^{\infty} \frac{2}{(2k+1)\pi} \sin\left((2k+1)\frac{\pi}{2}\right) \cos((2k+1)t)$$

Finally, note that when  $k = 0, 2, 4, 6, \dots$

$$\sin\left((2k+1)\frac{\pi}{2}\right) = 1$$

and when  $k = 1, 3, 5, 7, \dots$

$$\sin\left((2k+1)\frac{\pi}{2}\right) = -1$$

so we can simply write

$$\sin\left((2k+1)\frac{\pi}{2}\right) = (-1)^k$$

for any value of  $k = 0, 1, 2, 3, \dots$

The result of all this is

$$f(t) = \frac{1}{2} + \sum_{k=0}^{\infty} \frac{2}{(2k+1)\pi} (-1)^k \cos((2k+1)t)$$

Wasn't this fun?

# 49 DFT and FFT

## 49.1 DFT

When studying the frequencies of a time series, we don't actually calculate the Fourier transform, but the Discrete Fourier Transform (DFT).

continuous Fourier:

$$F(k) = \int_{-\infty}^{\infty} x(t)e^{-2\pi i k t} dt$$

discrete Fourier:

$$F_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k \frac{n}{N}}$$

Clearly, in the discrete version, time  $t$  becomes the index  $n$  and frequency  $f$  becomes the index  $k$ . The Fourier transform yields a complex number for each value of the frequency  $\xi_k$ .

Let's see this in action. We will calculate the Fourier transform of the function:

$$x(t) = 3 \sin(2\pi k_1 t) + \frac{1}{2} \cos(2\pi k_2 t) + 2$$

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
```

```

import matplotlib.gridspec as gridspec
import math
import scipy

# plt.rcParams.update({
#     "text.usetex": True,
#     "font.family": "serif",
#     "font.serif": "Computer Modern",
# })

# %matplotlib widget

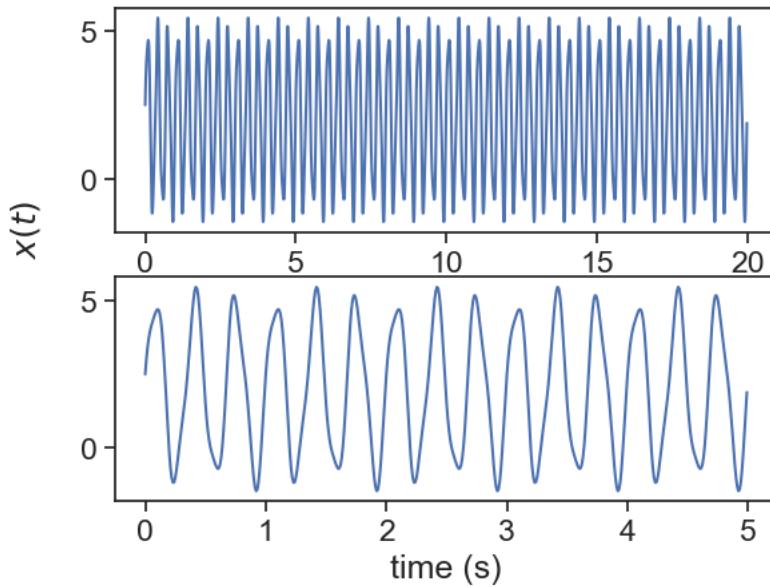
```

```

fig, ax = plt.subplots(2,1)
k1 = 3.0 # Hz = 1/s
k2 = 7.0 # Hz = 1/s
dt = 0.01
t = np.arange(0,20,dt)
N = len(t)
tau = math.tau
x = 3.0 * np.sin(tau * k1 * t) + 0.5*np.cos(tau * k2 * t) + 2.0
ax[0].plot(t, x)
ax[1].plot(t[:int(5/dt)], x[:int(5/dt)])
fig.text(0,0.5, r"$x(t)$", rotation="vertical")
ax[0].set_title(r"$x(t) = 3 \sin(2\pi k_1 t) + \frac{1}{2} \cos(2\pi k_2 t) + 2$", pad=20)
ax[1].set(xlabel=r"time (s)")

```

$$x(t) = 3\sin(2\pi k_1 t) + \frac{1}{2}\cos(2\pi k_2 t) + 2$$



```

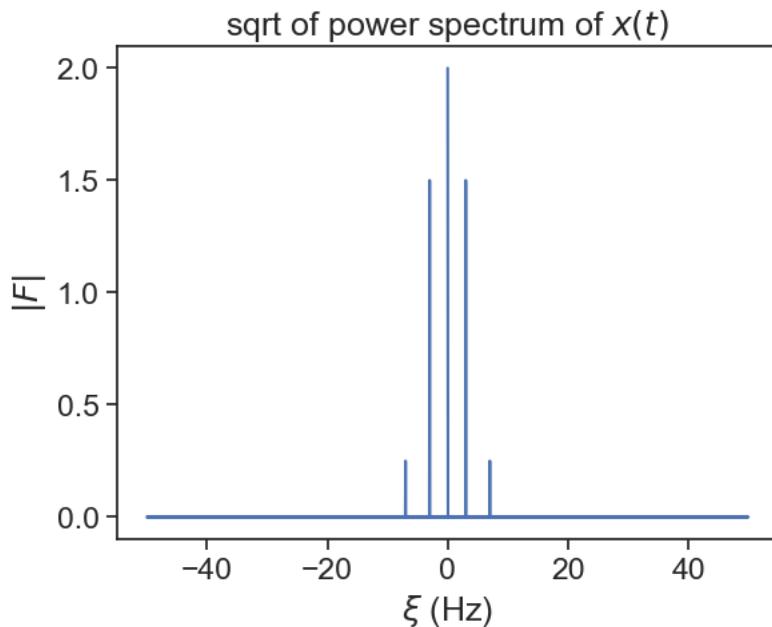
def my_dft(signal, dt):
    N = len(signal)
    k_index = np.arange(N)
    n = np.arange(N)
    F = []
    for ki in k_index:
        expon = np.exp(-2.0*np.pi*1j * ki * n / N)
        Fk = np.sum(signal * expon)
        F.append(Fk)
    k = build_k(N, dt)
    return k, np.array(F)/N

def build_k(N, dt):
    if N%2 == 0:
        k = np.hstack([np.arange(0,N//2),
                      np.arange(-N//2,0)])
    else:
        k = np.hstack([np.arange(0,(N-1)//2+1),
                      np.arange(-(N-1)//2,0)])
    return k / (dt*N)

```

```
k_dft, dft = my_dft(x, dt)
dft_abs = np.abs(dft)
```

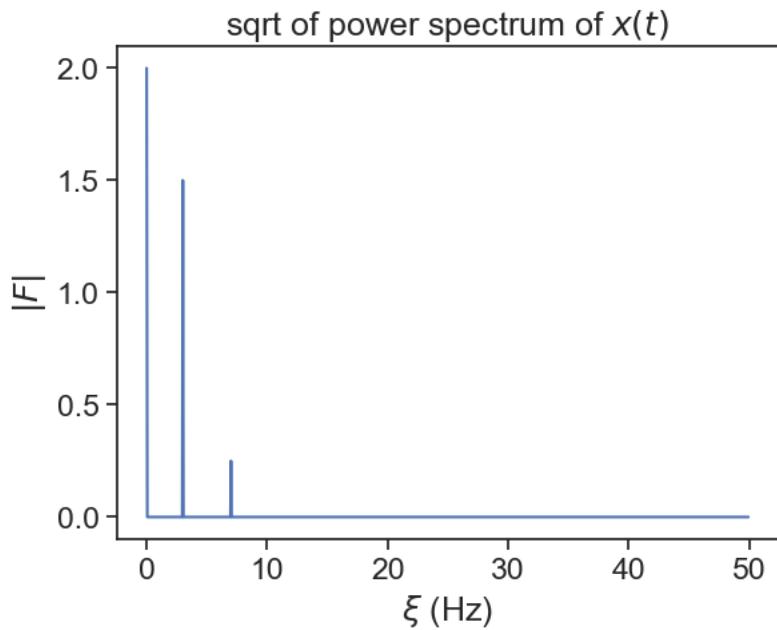
```
fig, ax = plt.subplots()
ax.plot(k_dft, dft_abs)
ax.set(xlabel=r"\xi$ (Hz)",
       ylabel=r"$|F|$",
       title=r"sqrt of power spectrum of $x(t)$");
```



Why does this work the way it does?

The power spectrum of any real-valued signal is symmetric between positive and negative frequencies, so let's plot only the positive frequencies:

```
halfN = N//2
fig, ax = plt.subplots()
ax.plot(k_dft[:halfN], dft_abs[:halfN])
ax.set(xlabel=r"\xi$ (Hz)",
       ylabel=r"$|F|$",
       title=r"sqrt of power spectrum of $x(t)$");
```



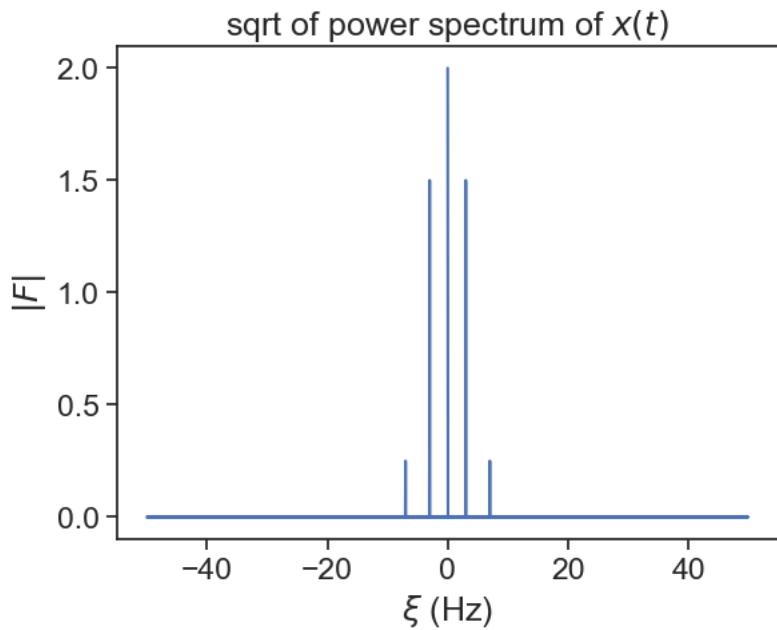
## 49.2 FFT

The Fast Fourier Transform is a very efficient algorithm to calculate the DFT. Derek Muller (Veritasium) made a great video on this topic.

We can from now on use `scipy`'s implementation of the FFT.

```
fft = scipy.fft.fft(x) / N
k_fft = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)
```

```
fig, ax = plt.subplots()
ax.plot(k_fft, fft_abs)
ax.set(xlabel=r"\xi$ (Hz)",
       ylabel=r"$|F|$",
       title=r"sqrt of power spectrum of $x(t)$");
```

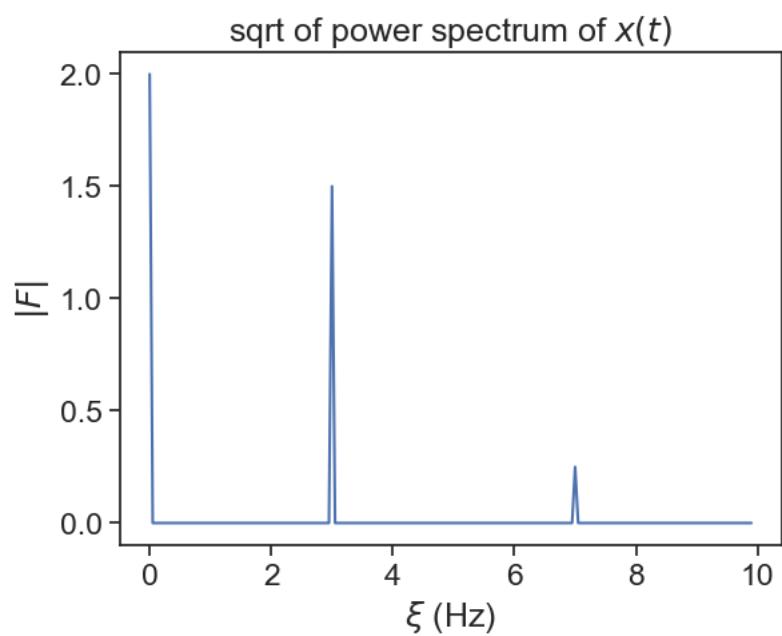


We have the exact same result as before :)

Let's zoom in.

```
dk = 1 / (t.max() - t.min())
n = int(10/dk)

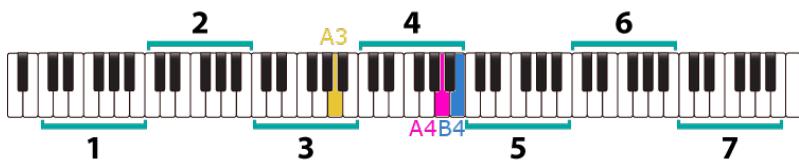
fig, ax = plt.subplots()
ax.plot(k_fft[:n], fft_abs[:n])
ax.set(xlabel=r"\xi$ (Hz)",
       ylabel=r"$|F|$", 
       title=r"sqrt of power spectrum of $x(t)$");
```



# 50 sound & music

Sound source: [University of Iowa Electronic Music Studios](#)

Listen to these three notes on the piano



A3  
A4  
B4

Image modified from:  
[piano-music-theory.com](#)

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
import aifc
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)

import plotly.io as pio
pio.renderers.default = "plotly_mimetype+notebook_connected"
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
pio.templates.default = "presentation"
import math
import scipy
from scipy.io import wavfile

# %matplotlib widget
```

```

def open_aiff(filename):
    data = aifc.open(filename)
    Nframes = data.getnframes()
    str_signal = data.readframes(Nframes)
    # d = np.fromstring(str_signal, numpy.short).byteswap()
    d = np.frombuffer(str_signal, dtype=np.int16).byteswap()
    # if there are two channels, take only one of them
    if data.getnchannels() == 2:
        d = d[::2]
    N = len(d)
    dt = 1.0 / data.getframerate()
    time = np.arange(N) * dt
    return time, d

def open_wav_return_fft(filename, max_freq=None):
    sample_rate, signal = wavfile.read(filename)
    time = np.arange(len(signal)) / sample_rate
    dt = time[1] - time[0]
    N = len(time)
    signal = normalize(signal)
    fft = scipy.fft.fft(signal) / N
    k = scipy.fft.fftfreq(N, dt)
    fft_abs = np.abs(fft)
    if max_freq == None:
        max_freq = k.max()
    dk = 1 / (time.max() - time.min())
    n = int(max_freq/dk)
    return k[:n], fft_abs[:n]

def normalize(signal):
    return (signal - signal.mean()) / signal.std()

def fft_abs(signal, time, max_freq=None):
    dt = time[1] - time[0]
    N = len(time)
    fft = scipy.fft.fft(signal) / N
    k = scipy.fft.fftfreq(N, dt)
    fft_abs = np.abs(fft)
    if max_freq == None:

```

```

    max_freq = k.max()
    dk = 1 / (time.max() - time.min())
    n = int(max_freq/dk)
    return k[:n], fft_abs[:n]

tmin = 2.00
tmax = 2.20

timeA4, pianoA4 = open_aiff('files/Piano.mf.A4.aiff')
indices = np.where((timeA4 > tmin) & (timeA4 < tmax))
timeA4 = timeA4[indices]
pianoA4 = normalize(pianoA4[indices])

timeA3, pianoA3 = open_aiff('files/Piano.mf.A3.aiff')
indices = np.where((timeA3 > tmin) & (timeA3 < tmax))
timeA3 = timeA3[indices]
pianoA3 = normalize(pianoA3[indices])

timeB4, pianoB4 = open_aiff('files/Piano.mf.B4.aiff')
indices = np.where((timeB4 > tmin) & (timeB4 < tmax))
timeB4 = timeB4[indices]
pianoB4 = normalize(pianoB4[indices])

# blue, orange, green, red, purple, brown, pink, gray, olive, cyan = px.colors.qualitative.D3[

color_A3 = "#e7c535"
color_A4 = "#ff00c3"
color_B4 = "#327fce"

fig = make_subplots()

fig.add_trace(
    go.Scatter(x=list(timeA3),
                y=list(pianoA3),
                name='piano A3',
                line=dict(color=color_A3),
                visible='legendonly'),
)

```

```

fig.add_trace(
    go.Scatter(x=list(timeA4),
                y=list(pianoA4),
                name='piano A4',
                line=dict(color=color_A4),
                ),
)

fig.add_trace(
    go.Scatter(x=list(timeB4),
                y=list(pianoB4),
                name='piano B4',
                line=dict(color=color_B4),
                visible='legendonly'),
)

# Add range slider
fig.update_layout(
    title='3 notes on the piano',
    xaxis=dict(
        rangeslider={"visible":True},
        type="linear"
    ),
    legend={"orientation":"h",           # Horizontal legend
            "yanchor":"top",       # Anchor legend to the top
            "y":1.1,              # Adjust vertical position
            "xanchor":"center",   # Anchor legend to the right
            "x":0.5,              # Adjust horizontal position
    },
)

# Set y-axes titles
fig.update_yaxes(
    title_text="signal",
    range=[-3,3])
fig.update_xaxes(
    title_text="time (s)",)

```

Unable to display output for mime type(s): text/html

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html
```

- When we zoom in, both A3 and A4 have the same period, but different shapes.
- A4 and B4 have slightly different periods (A4 is longer).

## 50.1 power spectrum

We can plot the absolute value of the Fourier transform of each signal, see below.

```
kA4, abs_fft_A4 = fft_abs(pianoA4, timeA4, max_freq=4e3)
kA3, abs_fft_A3 = fft_abs(pianoA3, timeA3, max_freq=4e3)
kB4, abs_fft_B4 = fft_abs(pianoB4, timeB4, max_freq=4e3)
```

```
fig = make_subplots()

fig.add_trace(
    go.Scatter(x=list(kA3),
                y=list(abs_fft_A3),
                name='piano A3',
                line=dict(color=color_A3),
                visible='legendonly'),
)

fig.add_trace(
    go.Scatter(x=list(kA4),
                y=list(abs_fft_A4),
                name='piano A4',
                line=dict(color=color_A4),),
)

fig.add_trace(
    go.Scatter(x=list(kB4),
                y=list(abs_fft_B4),
                name='piano B4',
                line=dict(color=color_B4),
                visible='legendonly'),
)
```

```

# Add range slider
fig.update_layout(
    title='3 notes on the piano',
    legend={"orientation":"h",
            "yanchor":"top",
            "y":1.1,
            "xanchor":"center",
            "x":0.5,
            },
)
# Set y-axes titles
fig.update_yaxes(
    title_text="abs(FFT)",
    range=[-0.1,1])
fig.update_xaxes(
    title_text="frequency (Hz)",)

```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

We call the graph above a power spectrum. “Spectrum” refers to the frequencies, and “power” means that we square the signal. Strictly speaking, we see above the square root of the power spectrum, but I’ll still call it power spectrum.

## 50.2 harmonics

Why do we see peaks at regular intervals in the power spectrum??

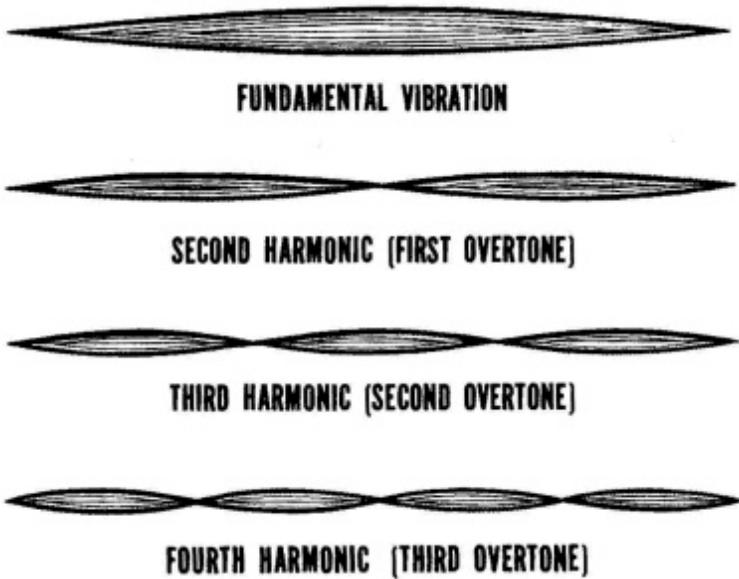
We call the lowest peak in the spectrum the fundamental frequency. Note that there are other high frequency peaks that follow the fundamental, at regular intervals. These are called overtones, and they are multiples of the fundamental frequency. The fundamental and the overtones are called together the harmonics.

The energy associated with a wave is proportional to the square of the wave’s amplitude. When we computed the `fft` of the signal, we divided it by `N`, which is akin to dividing by the whole time duration of the signal. For this reason we are dealing with power, which is energy / time. Finally, it should be noted that the square of a complex number  $z = a + ib$  is given by

$$|z|^2 = z \cdot z^*,$$

where  $z^* = a - ib$  is its complex conjugate:

$$z \cdot z^* = (a + ib)(a - ib) = a^2 + b^2.$$



The sound produced by musical instruments is the outcome of vibrations in the body of the instrument. Often, these vibrations are **standing waves**, and that is the reason why we see such a strong overtone signature in the power spectrum.

Image source: [N.H. Crowhurst](#)

### 50.3 timbre

---

instrument recording of A4

piano

---

image



---

instrument recording of A4

image

---



flute



trumpet



violin

---

```
tmin = 1.00
tmax = 2.50

time_list = []
signal_list = []
instrument_list = ['flute', 'piano', 'trumpet', 'violin']
file_list = ['flute-C4.wav',
             'piano-C4.wav',
             'trumpet-C4.wav',
             'violin-C4.wav'
         ]
```

```

k_list = []
abs_fft_list = []
for i, filename in enumerate(file_list):
    k, abs_fft = open_wav_return_fft(f'files/{filename}', max_freq=3.5e3)
    k_list.append(k)
    abs_fft_list.append(abs_fft)

fig = make_subplots()

for i, instrument in enumerate(instrument_list):
    vis = 'legendonly'
    if instrument == 'piano': vis = True
    fig.add_trace(
        go.Scatter(x=list(k_list[i]),
                    y=list(abs_fft_list[i]),
                    name=f'{instrument}',
                    # line=dict(color=color_A3),
                    visible=vis #'legendonly'
                    ),
    )

# Add range slider
fig.update_layout(
    # title='3 notes on the piano',
    legend={"orientation": "h",                      # Horizontal legend
            "yanchor": "top",                     # Anchor legend to the top
            "y": 1.1,                           # Adjust vertical position
            "xanchor": "center",                 # Anchor legend to the right
            "x": 0.5,                           # Adjust horizontal position
            },
)

# Set y-axes titles
fig.update_yaxes(
    title_text="abs(FFT)",
    range=[-0.05, 0.4])
fig.update_xaxes(
    title_text="frequency (Hz"),)

```

Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

This is a nice video about the connection between fibre and the harmonic series.

## 50.4 linear vs. logarithmic scale

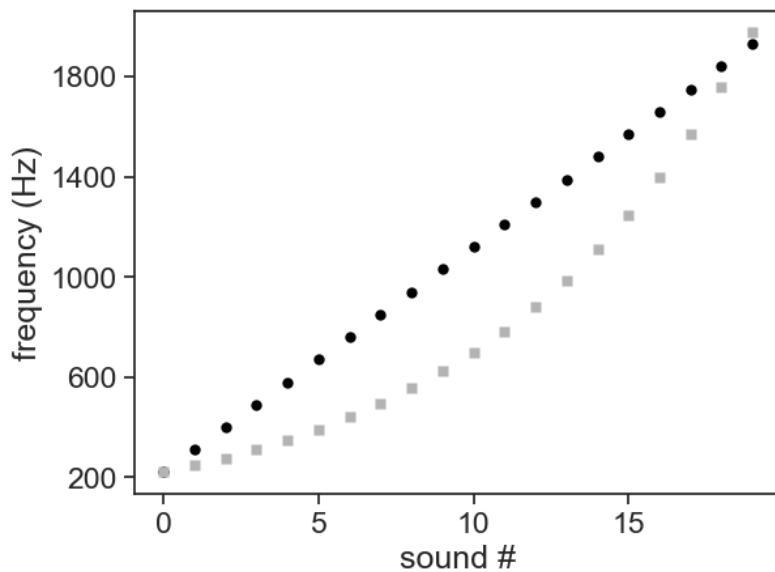
Listen to these two sequences of sounds.

1

2

See below a graph representing the sequence of frequencies for these two recordings. Which recording sounds more regular, like climbing steps of equal size?

```
fr_linear = np.arange(220.0, 220.0 + 20*90, 90)
alpha = 2 ** (2.0/12)
fr_exp = np.array(
    [220.0 * alpha ** i for i in range(20)])
fig, ax = plt.subplots()
ax.plot(fr_linear, 'o', mfc="black", mec="None")
ax.plot(fr_exp, 's', mfc=[0.7]*3, mec="None")
ax.set(xlabel="sound #",
       ylabel="frequency (Hz)",
       yticks=np.arange(200,2001,400))
pass
```



Probably, the most common tuning standard in western music is A440, meaning that the note corresponding to the A4 on the piano must have a fundamental frequency equal to 440 Hz. Go to the graph shown in [power spectrum](#), zoom in, and find out if the piano was “properly” tuned.

The A3 note has its fundamental frequency at half of that, namely 220 Hz.

Because there are 12 half-steps between A3 and A4, can you figure out a rule how to find the frequency of any note on the piano?

Dividing an octave in 12 equal half-steps is called “equal temperament”. Multiplying the A3 frequency 12 times by an unknown factor  $y$  should give us the frequency of A4:

$$\begin{aligned} 220 \times y^{12} &= 440 \\ y^{12} &= 2 \\ \therefore y &= \sqrt[12]{2} \end{aligned}$$

We can now easily check out if the fundamental frequency for B3 as shown in the graph agrees with this rule:

```

factor = 2**(1/12)
B4 = 440 * factor**2 # two half steps above A4
print(f"B4 = {B4:.2f} Hz")

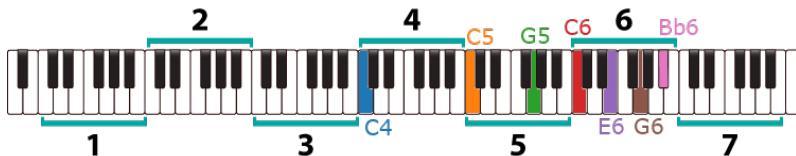
```

B4 = 493.88 Hz

Our pitch perception is logarithmic, meaning that when we hear a string of frequencies that increase exponentially, we perceive it as increasing with regular steps. Sound 2 corresponds to the exponential orange dots.

## 50.5 chords

# of harmonic	frequency (Hz)	pitch	interval
1	261.626	C4	fundamental
2	523.252	C5	octave
3	523.252	G5	perfect fifth
4	1046.504	C6	octave
5	1308.130	E6	major third
6	1569.755	G6	perfect fifth



```

from palettable.tableau import Tableau_10
colors = Tableau_10.hex_colors

N = 7
harmonics = np.arange(1, N+1)
pitches = 261.626 * harmonics
names = ['C4', 'C5', 'G5', 'C6', 'E6', 'G6', 'Bb6']

fig, ax = plt.subplots()
ax.bar(harmonics, pitches, color=colors[:N])

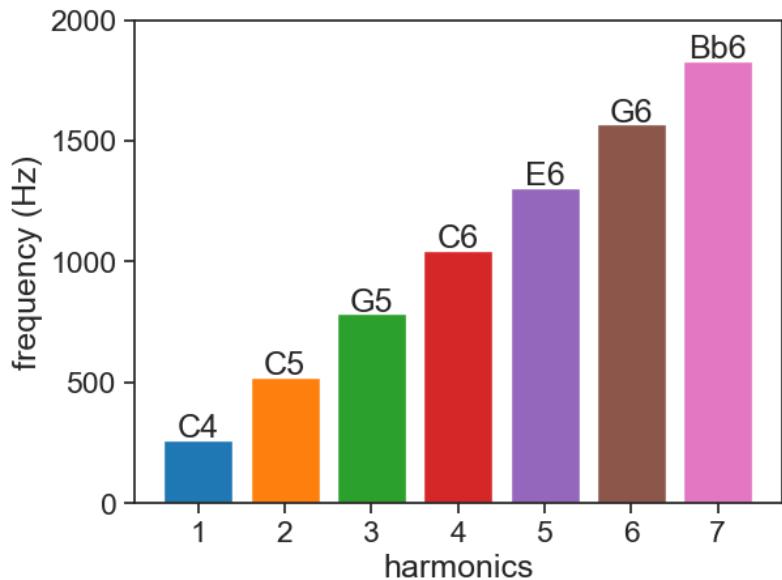
```

```

for i,h in enumerate(harmonics):
    ax.text(h, pitches[i]+10, names[i], ha="center")

ax.set(xlabel="harmonics",
       ylabel="frequency (Hz)",
       xticks=harmonics,
       yticks=np.arange(00,2001,500))
pass

```



C (major chord, C + E + G = do-mi-sol)

C7 (C + E + G + Bb = do-mi-sol-si bemol)

Dissonant sound:

C + C#

# **51 frequencies**

## **51.1 spectrum**

In the 17th century, Isaac Newton studied how white light can be decomposed into the colors of the rainbow. He called the decomposed colors “**spectrum**”, after the latin word for image. Later, the word spectrum started to be used in other contexts involving a range of frequencies, even when not related to light or colors.



Source: [Smithsonian Libraries](#)

## 51.2 resolution

The Discrete Fourier Transform helps us decompose a discrete time series into a list of (complex) weights for a range of frequencies, or spectrum. Its formula is given by

$$F_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k \frac{n}{N}},$$

where  $F_k$  is the array representing the Fourier transform  $F(\xi)$ , and  $x_n$  is the array representing the time series  $f(t)$ . All the information contained in  $f(t)$  is fully preserved in  $F(\xi)$ .

What are the lowest and highest frequencies that can be measured?

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import matplotlib.gridspec as gridspec
import math
import scipy
# %matplotlib widget

fig, ax = plt.subplots(2, 1, sharex=True)

N = 8
dt = 0.2
T1 = N * dt
T2 = 2 * dt
time = dt * np.arange(N)
f1 = np.cos(2.0*np.pi*time/T1)
f2 = np.cos(2.0*np.pi*time/T2)

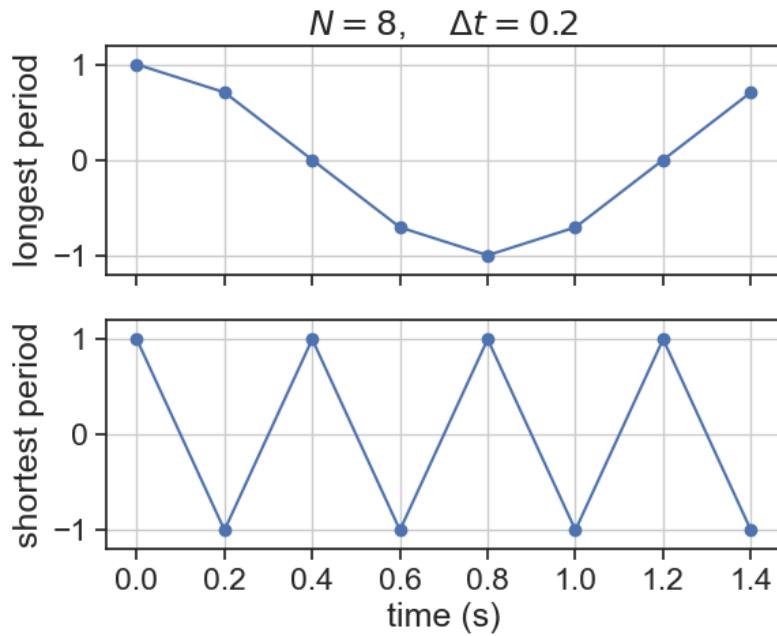
ax[0].plot(time, f1, '-o')
ax[1].plot(time, f2, '-o')

ax[0].set(ylim=[-1.2, 1.2],
          ylabel="longest period",
          title="$N=8$, $\Delta t=0.2$")
ax[1].set(xticks=time,
```

```

    xlabel="time (s)",
    ylim=[-1.2, 1.2],
    ylabel="shortest period")
ax[0].grid()
ax[1].grid()

```



The longest possible period that we can measure is the length of the time series itself:

$$T_{\text{long}} = N \times \Delta t$$

The shortest possible period is twice the time resolution:

$$T_{\text{short}} = 2 \times \Delta t$$

Let's translate this into frequency. Because frequency is the inverse of period ( $\xi = \frac{1}{T}$ ), the smallest possible frequency is the inverse of the longest possible period, and vice versa.

$$\xi_{\text{small}} = \frac{1}{T_{\text{long}}} = \frac{1}{N \times \Delta t}$$

$$\xi_{\text{large}} = \frac{1}{T_{\text{short}}} = \frac{1}{2 \times \Delta t}$$

There is a duality between the variables time ( $t$ ) and frequency ( $\xi$ ). It is important to understand how to convert from one the other in the context of DFTs:

```
fig = plt.figure(1, figsize=(8,4))

gs = gridspec.GridSpec(2, 1, width_ratios=[1], height_ratios=[1,1])
gs.update(left=0.10, right=0.90,top=0.90, bottom=0.10,
           hspace=0.05, wspace=0.05
          )
ax_t = plt.subplot(gs[0, 0])
ax_k = plt.subplot(gs[1, 0])

text_dict = {'ha':'center', 'va':'center', 'color':"orangered",
            'fontsize':24, 'weight':'bold'}
N = 8
t_vector = np.arange(N)
k_vector = np.fft.fftfreq(N, d=1.0)
t_vector = t_vector.reshape(1, len(t_vector))
k_vector = N * k_vector.reshape(1, len(k_vector))

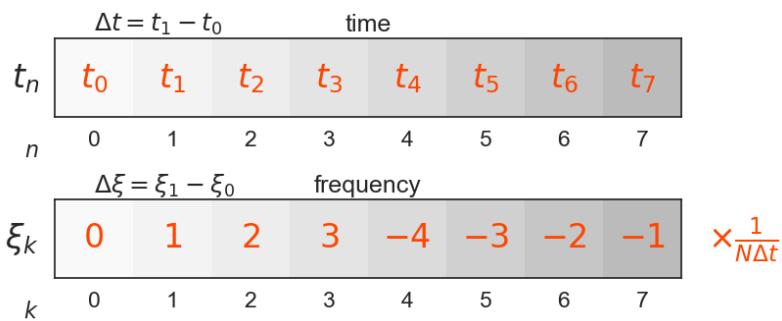
ax_t.imshow(t_vector, cmap='Greys', vmin=-1, vmax=20, origin="lower")
ax_k.imshow(t_vector, cmap='Greys', vmin=-1, vmax=20, origin="lower")

for (j,i),label in np.ndenumerate(t_vector):
    ax_t.text(i, j, fr"${t}_{\{label\}}$".format(label), **text_dict)
for (j,i),label in np.ndenumerate(k_vector):
    ax_k.text(i, j, fr"${\xi}_{\{label\}}$".format(label), **text_dict)
ax_t.set(yticks=[], xticks=np.arange(N), title="time")
# ax_t.set_ylabel(r"$t_n$", rotation="horizontal", ha="center")
ax_t.text(-0.7, 0, r"$t_n$", ha="right", va="center", fontsize=24)
ax_t.text(-0.7, -1.0, r"$n$", ha="right", fontsize=16)
ax_k.text(-0.7, 0, r"${\xi}_k$".format(label), ha="right", va="center", fontsize=24)
ax_k.text(-0.7, -1.0, r"$k$".format(label), ha="right", fontsize=16)
```

```

ax_k.set(yticks=[], xticks=np.arange(N), title="frequency")
# ax_k.yaxis.set_label_position("right")
# ax_k.set_ylabel(r"\times\frac{1}{N\Delta t}", rotation="horizontal", labelpad=20)
ax_k.text(8.3, 0, r"\times\frac{1}{N\Delta t}", **text_dict)
ax_t.tick_params(axis='x', which='both', bottom=False)
ax_k.tick_params(axis='x', which='both', bottom=False)
ax_t.text(0, 0.6, r"\Delta t = t_1 - t_0")
ax_k.text(0, 0.6, r"\Delta \xi = \xi_1 - \xi_0")
pass

```



Why is there such a weird pattern for  $\xi$ , where first we have the positive values, and after that the negative? Let's see again the equation for the DFT:

$$F_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k \frac{n}{N}},$$

What happens to the location of  $x_1$  as we increase the index  $k$ ?

```

fig, ax = plt.subplots(1, figsize=(6, 6), subplot_kw=dict(polar=True))

N = 8
k = np.arange(8)
angles = 2.0 * np.pi * k * 1 / N

# for i in range(N):
#     ax[i].plot([theta[i]]*2, [0, 1], lw=2, color="tab:red")

```

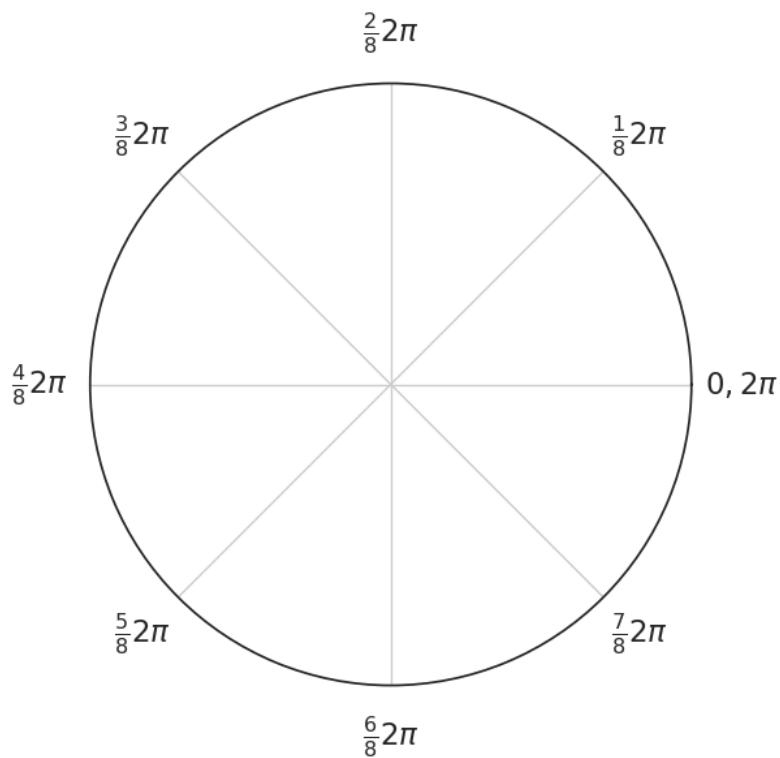
```

# stems = ax.stem(theta, r)
# for artist in stems.get_children():
#     artist.set_clip_on(False)

theta_ticks = 2.0*np.pi * np.arange(8)/8
theta_tick_labels = [r"$0,2\pi$",
                     r"\frac{1}{8}2\pi",
                     r"\frac{2}{8}2\pi",
                     r"\frac{3}{8}2\pi",
                     r"\frac{4}{8}2\pi",
                     r"\frac{5}{8}2\pi",
                     r"\frac{6}{8}2\pi",
                     r"\frac{7}{8}2\pi",
                     ]
                     ]

ax.set(yticks=[],
       xticks=theta_ticks,
       ticklabels=theta_tick_labels,
       ylim=(0, 1))
# # ax.set_xticks(theta_ticks, theta_tick_labels, pad=10)
ax.tick_params(axis='x', which='major', pad=15)
# ax.grid(False)
# # ax.set_clip_on(False)

```



```

N = 8
r = [1] * N
k = np.arange(8)
theta = -2.0 * np.pi * k * 1 / N #np.arange(1,N+1) * 2.0 * np.pi / N

fig, ax = plt.subplots(N, 1, figsize=(30, 10), subplot_kw=dict(polar=True))

for i in range(N):
    ax[i].plot([theta[i]]*2, [0, 1], lw=2, color="tab:red")
    ax[i].set(yticks=[], 
               xticks=theta_ticks,
               xticklabels=[],
               ylim=(0, 1))
    ax[i].text(np.pi, 1.5, fr"$k={i}$", ha="right", va="center")

ax[0].set(title=r"location of $x_1$")

for i in range(4,8):

```

```

ax[i].text(0, 1.5, fr"also $k={i-8}$", ha="left", va="center")

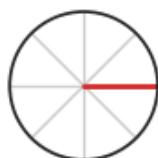
# stems = ax.stem(theta, r)
# for artist in stems.get_children():
#     artist.set_clip_on(False)

# theta_ticks = 2.0*np.pi * np.arange(8)/8
# theta_tick_labels = [r"$0,2\pi$",
#                      r"\frac{1}{8}2\pi",
#                      r"\frac{2}{8}2\pi",
#                      r"\frac{3}{8}2\pi",
#                      r"\frac{4}{8}2\pi",
#                      r"\frac{5}{8}2\pi",
#                      r"\frac{6}{8}2\pi",
#                      r"\frac{7}{8}2\pi",
#                      ]
# # ax.set_xticks(theta_ticks, theta_tick_labels, pad=10)
# ax.tick_params(axis='x', which='major', pad=15)
# ax.grid(False)
# # ax.set_clip_on(False)

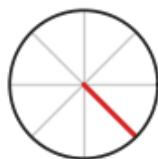
```

location of  $x_1$

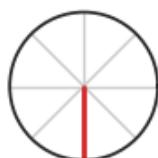
$k = 0$



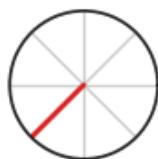
$k = 1$



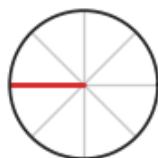
$k = 2$



$k = 3$

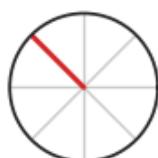


$k = 4$



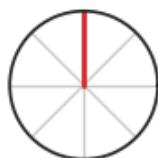
also  $k = -4$

$k = 5$



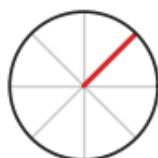
also  $k = -3$

$k = 6$



also  $k = -2$

$k = 7$



also  $k = -1$

## 51.3 Nyquist–Shannon sampling theorem

This part is partially based on [All About Circuits](#).

We saw above that when sampling at a  $\Delta t$ , the shortest possible period is  $2\Delta t$ . This is the Nyquist rate.

### Theorem

If a system uniformly samples an analog signal at a rate that exceeds the signal's highest frequency by at least a factor of two, the original analog signal can be perfectly recovered from the discrete values produced by sampling.

Conversely, we can negate the statement above, saying that:

If we sample an analog signal at a frequency that is lower than the Nyquist rate, we will not be able to perfectly reconstruct the original signal.

As an example, let's sample a cosine wave whose frequency is 1 Hz at various sampling rates.

```
dt_list = [0.01, 0.1, 0.2, 0.5, 0.53, 1.0, 1.2, 1.5]

fig, ax = plt.subplots(len(dt_list), 1, figsize=(8, 15), sharex=True)

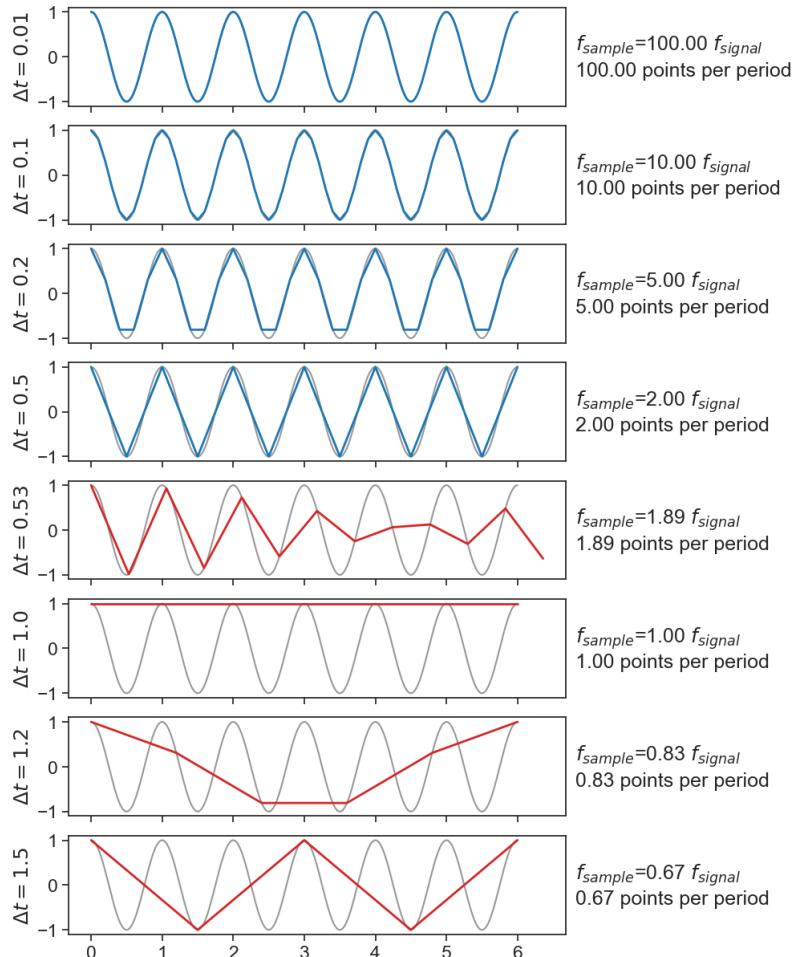
T_total = 6.0 # seconds

func = lambda t: np.cos(2.0 * np.pi * t)

t_high = np.arange(0, T_total, 0.001)
signal = func(t_high)

for i, dt in enumerate(dt_list):
    time = np.arange(0, T_total+dt, dt)
    ax[i].plot(t_high, signal, color="black", alpha=0.4)
    c = "tab:blue"
    if dt > 0.5: c = "tab:red"
    ax[i].plot(time, func(time), lw=2, color=c)
    ax[i].set(ylabel=fr"\Delta t = {dt}$")
    text = r"$f_{sample}"+rf"={1/dt:.2f}"+r" $f_{signal}"+f"\n{1/dt:.2f} points per period"
    ax[i].text(0, -0.1, text)
```

```
ax[i].text(1.02, 0.50, text, transform=ax[i].transAxes,
           horizontalalignment='left', verticalalignment='center',)
```



Sampling at a  $\Delta t$  greater than 0.5 exemplifies a phenomenon called **aliasing**.

## 52 filtering

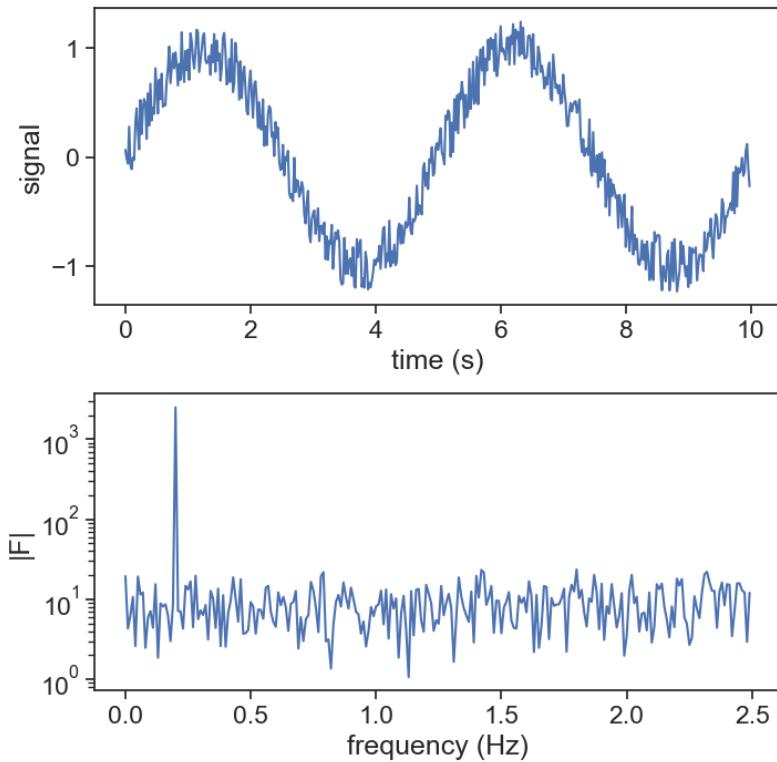
Let's say we have a signal with some noise that we want to filter out. How would we do that using Fourier transforms?

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import math
import scipy
# %matplotlib widget

dt = 0.02
time = np.arange(0,100,dt)
N = len(time)
period = 5
signal = np.sin(2.0*np.pi*time/period) + 0.5 * (np.random.random(N)-0.5)

fft = scipy.fft.fft(signal)
xi = scipy.fft.fftfreq(N, dt)

fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.30)
ax[0].plot(time[:N//10], signal[:N//10])
ax[1].plot(xi[:N//20], np.abs(fft[:N//20]))
ax[1].set_yscale('log')
ax[0].set_xlabel("time (s)",
                 ylabel="signal")
ax[1].set_xlabel("frequency (Hz)",
                 ylabel="|F|");
```



We can get rid of the fast variations in the signal by eliminating all high frequencies in the Fourier transform. The simplest way to do that is determining a cut-off frequency, and zeroing out the Fourier transform corresponding to frequencies higher than the cut off.

```
cutoff = 1.0
# apply low pass filter
mask = np.where( (xi>cutoff) | (xi<-cutoff) )
fft_filtered = fft.copy()
fft_filtered[mask] = 0.0
# reconstitute signal using inverse Fourier transform
signal_filtered = np.fft.ifft(fft_filtered)
```

```
fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.30)
ax[0].plot(time[:N//10], signal[:N//10], color="tab:blue", label="original")
```

Also eliminate frequencies lower than negative cut off! This is important!

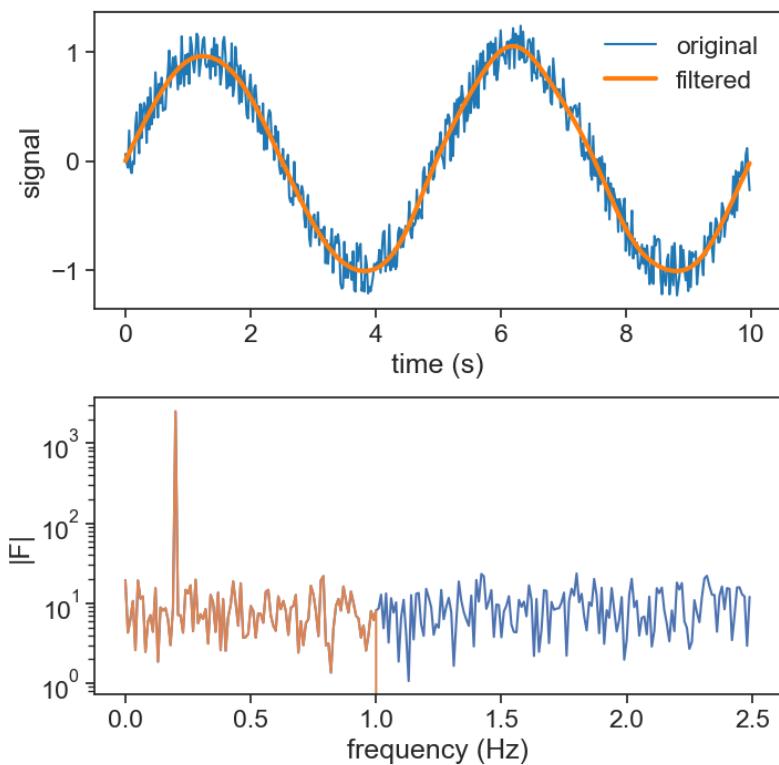
```

ax[0].plot(time[:N//10], signal_filtered[:N//10], color="tab:orange", lw=3, label="filtered")
ax[0].legend(frameon=False)
ax[0].set(xlabel="time (s)",
           ylabel="signal")

ax[1].plot(xi[:N//20], np.abs(fft[:N//20]))
ax[1].plot(xi[:N//20], np.abs(fft_filtered[:N//20]))

ax[1].set(xlabel="frequency (Hz)",
           ylabel="|F|");
ax[1].set_yscale('log');

```



We applied above a “low pass” filter, because we let all the low frequencies pass, and eliminated the higher frequencies. Within signal processing, there are four primary types of filters commonly employed, each serving a unique purpose in shaping the signal’s characteristics.

- 1. Lowpass Filter:** A lowpass filter allows signals with a frequency lower than a certain cutoff frequency to pass through while attenuating (reducing) the components of the signal that have frequencies higher than this cutoff frequency. It's used to remove high-frequency noise or to extract the low-frequency components of a signal.
- 2. Highpass Filter:** A highpass filter does the opposite of a lowpass filter. It allows signals with a frequency higher than a certain cutoff frequency to pass while attenuating signals with frequencies lower than the cutoff frequency. This type of filter is useful for removing low-frequency noise or to isolate high-frequency components.
- 3. Bandpass Filter:** A bandpass filter allows signals within a certain frequency range (between a lower and an upper cutoff frequency) to pass through while attenuating signals outside this range. It's used to isolate a specific frequency band from a broader spectrum of frequencies.
- 4. Bandstop Filter (Notch Filter):** A bandstop filter, also known as a notch filter, attenuates signals within a specific frequency range while allowing signals outside this range to pass through relatively unaffected. This type of filter is useful for eliminating unwanted frequencies or noise from a signal without significantly affecting the other components of the signal.

```

fig, ax = plt.subplots(2, 2, figsize=(8,8), sharex=True, sharey=True)

fr = np.linspace(0,10,101)

plot_dict = {'lw':3, 'color':'tab:red'}

# panel (0,0)
cutoff_lp = 5.0
gain_lp = np.ones_like(fr)
mask = np.where(fr>cutoff_lp)
gain_lp[mask] = 0.0
ax[0,0].plot(fr, gain_lp, **plot_dict)
ax[0,0].text(2.5,1.05, "passband", ha="center", fontsize=14)
ax[0,0].text(7.5,1.05, "stop band", ha="center", fontsize=14)

```

```

# panel (0,1)
cutoff_hp = 5.0
gain_hp = np.ones_like(fr)
mask = np.where(fr<cutoff_hp)
gain_hp[mask] = 0.0
ax[0,1].plot(fr, gain_hp, **plot_dict)
ax[0,1].text(2.5,1.05, "stop band", ha="center", fontsize=14)
ax[0,1].text(7.5,1.05, "passband", ha="center", fontsize=14)

cutoff_1 = 4.0
cutoff_2 = 6.0
gain_bp = np.ones_like(fr)
mask = np.where( (fr<cutoff_1) | (fr>cutoff_2) )
gain_bp[mask] = 0.0
ax[1,0].plot(fr, gain_bp, **plot_dict)

gain_bs = np.ones_like(fr)
mask = np.where( (fr>cutoff_1) & (fr<cutoff_2) )
gain_bs[mask] = 0.0
ax[1,1].plot(fr, gain_bs, **plot_dict)

ax[0,0].set(ylabel="gain",
             ylim=[-0.1, 1.2],
             yticks=[0,1],
             xticks=[],
             title = "low pass")
ax[0,1].set(title = "high pass")
ax[1,0].set(ylabel="gain",
             xlabel="frequency",
             title = "band pass"
            )
ax[1,1].set(xlabel="frequency",
             title = "band stop"
            )

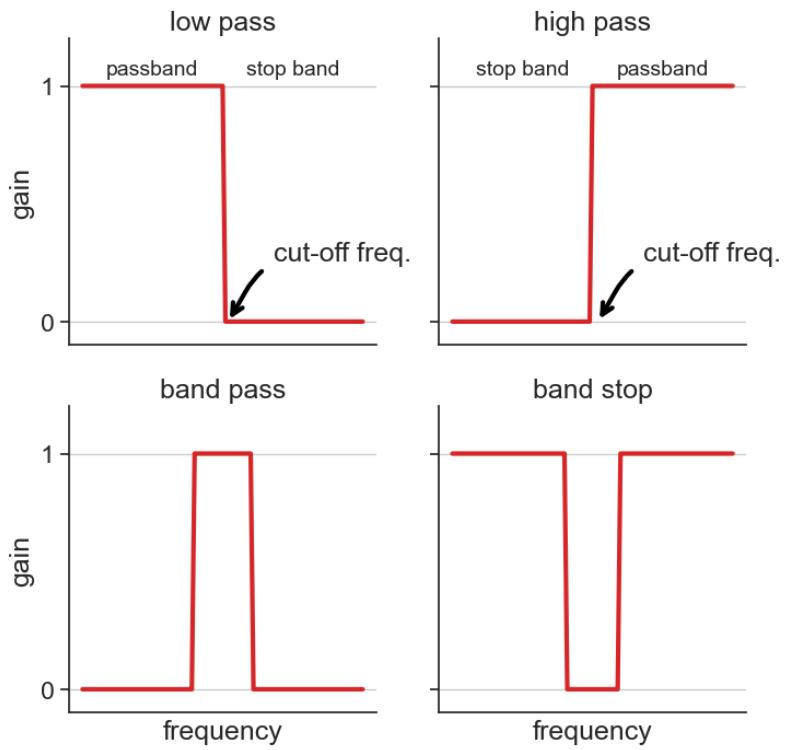
ax[0,0].spines[['right', 'top']].set_visible(False)
ax[0,1].spines[['right', 'top']].set_visible(False)
ax[1,0].spines[['right', 'top']].set_visible(False)
ax[1,1].spines[['right', 'top']].set_visible(False)
ax[0,0].grid()

```

```
ax[0,1].grid()
ax[1,0].grid()
ax[1,1].grid()

ax[0,0].annotate(
    "cut-off freq.",
    xy=(cutoff_lp+0.2, 0),  xycoords='data',
    xytext=(30, 40), textcoords='offset points',
    bbox=dict(boxstyle="round4,pad=.5", fc="white"),
    arrowprops=dict(arrowstyle="->",
                    color="black", lw=3,
                    connectionstyle="angle,angleA=0,angleB=60,rad=40"))

ax[0,1].annotate(
    "cut-off freq.",
    xy=(cutoff_hp+0.2, 0),  xycoords='data',
    xytext=(30, 40), textcoords='offset points',
    bbox=dict(boxstyle="round4,pad=.5", fc="white"),
    arrowprops=dict(arrowstyle="->",
                    color="black", lw=3,
                    connectionstyle="angle,angleA=0,angleB=60,rad=40"));
```



## 52.1 decibels

It is quite easy to apply a low pass filter as in the figure above, but that introduces problems down the road (more on that later). Usually, softer functions are used as filters, such as the ubiquitous **first-order low pass frequency response**:

$$\text{Gain} = \frac{1}{\sqrt{1 + (\xi/\xi_c)^2}}$$

```
fig, ax = plt.subplots(1, 2, figsize=(8,4))

fr = np.linspace(0,5000,1001)
plot_dict = {'lw':3, 'color':'tab:red'}
cutoff_lp = 500.0
```

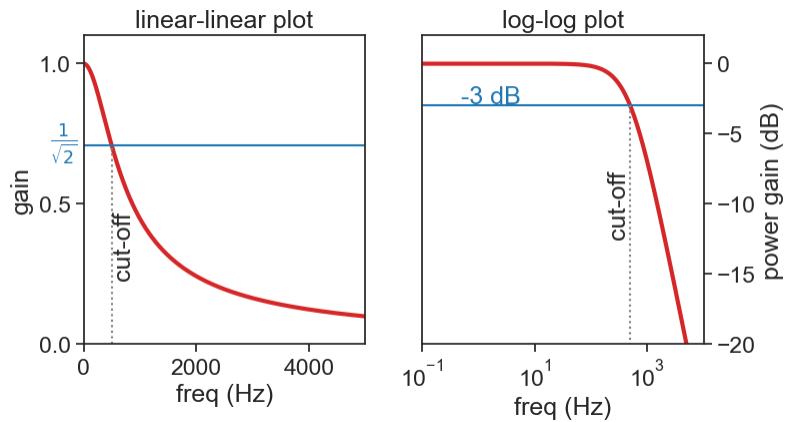
For example, this appears as the response of a simple [RC electric circuit](#). Also, this is a particular instance of the [Butterworth filter](#) (order  $n = 1$ ).

```

H = 1 / np.sqrt(1.0 + (fr/cutoff_lp)**2)
dB = 20.0 * np.log10(H)
ax[0].plot(fr, H, **plot_dict)
ax[0].set(xlim=[fr.min(), fr.max()],
           ylim=[0,1.1],
           yticks=[0,0.5,1.0],
           xlabel="freq (Hz)",
           ylabel="gain",
           title="linear-linear plot")
xlim_lin = ax[0].get_xlim()
ax[0].plot(xlim_lin, [1/np.sqrt(2)]*2, color="tab:blue")
ax[0].text(-100, 1/np.sqrt(2), r"\frac{1}{\sqrt{2}}", ha="right", color="tab:blue")
ax[0].text(cutoff_lp, 0.5/np.sqrt(2), "cut-off", rotation="vertical", ha="left", va="center")
ax[0].plot([cutoff_lp]*2, [0, 1/np.sqrt(2)], color="tab:gray", ls=":")
# ax[0].grid()

ax[1].plot(fr, dB, **plot_dict)
ax[1].set_xscale("log")
ax[1].yaxis.tick_right()
ax[1].yaxis.set_label_position("right")
ax[1].set(ylabel="power gain (dB)",
           xlabel="freq (Hz)",
           xlim=[1e-1,1e4],
           ylim=[-20,2],
           title="log-log plot")
# ax[1].grid()
xlim_log = ax[1].get_xlim()
ax[1].plot(xlim_log, [-3]*2, color="tab:blue")
ax[1].text(0.5, -3+0.2, "-3 dB", color="tab:blue")
ax[1].text(cutoff_lp, -10, "cut-off", rotation="vertical", ha="right", va="center")
ax[1].plot([cutoff_lp]*2, [-3, -30], color="tab:gray", ls=":")

```



The cut-off frequency is roughly at the elbow in the graph on the right, and traditionally it is identified with a power gain of -3 dB. **What does that mean?!**

Decibels are **defined** as the following expression of two powers:

$$\text{dB} = 10 \log_{10} \left( \frac{P}{P_0} \right),$$

where

- $P$  is the power being measured
- $P_0$  is a reference power level

Because powers are the square of the amplitude of a signal (let's call this amplitude  $V$ ), we can rewrite the definition as:

$$\text{dB} = 10 \log_{10} \left( \frac{V^2}{V_0^2} \right) = 20 \log_{10} \left( \frac{V}{V_0} \right),$$

The expression above is the famous one.

The cut-off frequency is usually considered that for which the ratio of powers drops to 1/2:

$$\frac{P}{P_0} = \frac{1}{2} \rightarrow \left( \frac{V}{V_0} \right)^2 = \frac{1}{2} \rightarrow \frac{V}{V_0} = \frac{1}{\sqrt{2}}$$

That is the value that we see on the graph on the left. What about the value on the graph on the right? Let's substitute  $\frac{P}{P_0} = \frac{1}{2}$  in the definition of the decibel:

$$\text{dB} = 10 \log_{10} \left( \frac{1}{2} \right) \simeq -3,$$

Voilà! -3 dB is the value for the cut-off we see on the right!

```
10*np.log10(0.5)
```

-3.010299956639812

# 53 convolution theorem

Way back then, we learned about sliding windows and how to take advantage of them for smoothing a signal. We did not give much emphasis to this [then](#), but the mathematical operation of sliding a window with a given kernel over a signal is called **convolution**. Let's recall what is the definition of convolution.

The definition of a convolution between signal  $f(t)$  and kernel  $k(t)$  is

$$(f * k)(t) = \int f(\tau)k(t - \tau)d\tau.$$

The expression  $f * k$  denotes the convolution of these two functions. The argument of  $k$  is  $t - \tau$ , meaning that the kernel runs from left to right (as  $t$  does), and at every point the two signals ( $f$  and  $k$ ) are multiplied together. It is the product of the signal with the weight function  $k$  that gives us an average. Because of  $-\tau$ , the kernel is flipped backwards, but this has no effect to symmetric kernels.

## 53.1 theorem

The Fourier transform of a convolution is the product of the Fourier transforms of each convolved functions.

In mathematical language:

$$F[f * g] = F[f] \cdot F[g]$$

This means that if we want to smooth a function  $f$  by averaging a sliding window of a given kernel  $g$ , we can instead compute the Fourier transform of  $f$  and  $g$  separately, multiply them together, and finally perform an inverse Fourier transform of this product:

$$\text{smooth function} = F^{-1} \left[ F[\text{signal}] \cdot F[\text{kernel}] \right]$$

### Why is this useful?

1. Sometimes it is computationally more efficient to compute Fourier transforms than to compute running averages. This is not obvious at all, but there are very fast algorithms that can calculate the Fourier transform in a heartbeat. As we will see later, the `numpy.convolve` tool calculates running averages (convolutions) the usual way for short arrays, and automatically switches its method when it evaluates that using the Fourier transform might be faster (usually for arrays longer than 500 points).
2. Thinking about the Fourier signature of different kernels can be extremely useful to achieve precise goals, such as filtering out specific frequencies from your signal, or to avoid undesirable effects of using a sliding window when you don't exactly know what you are doing.

## 53.2 kernels and their spectral signatures

```
import numpy as np
from scipy import signal
from scipy.fft import fft, fftshift
import matplotlib.pyplot as plt
import scipy
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
import matplotlib.dates as mdates
import warnings
warnings.filterwarnings("ignore")
```

```

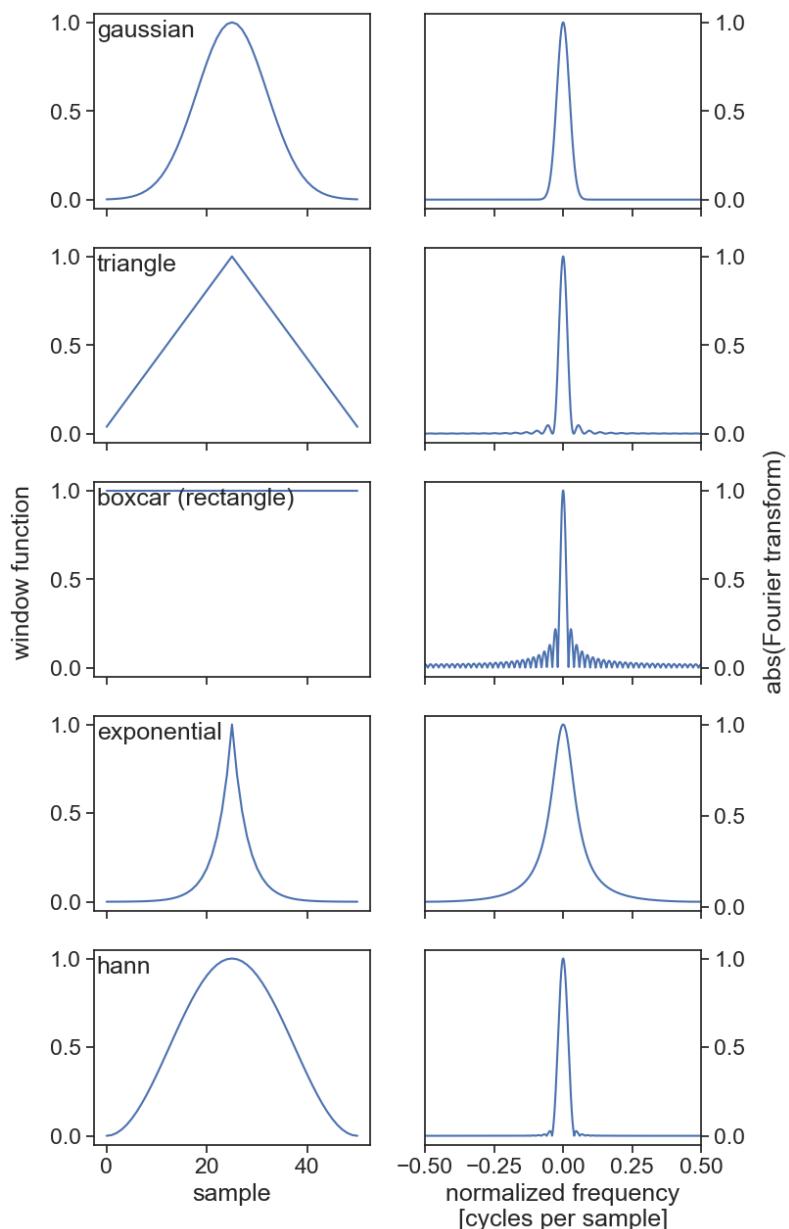
def plot_window_response(name, func, args, ax_left, ax_right, resp="None"):
    window = func(51, **args)
    ax_left.plot(window)
    ax_left.set(ylim=[-0.05, 1.05])
    ax_left.text(0.01, 0.97, name, transform=ax_left.transAxes,
                 horizontalalignment='left', verticalalignment='top')
    A = fft(window, 2048) / (len(window)/2.0)
    freq = np.linspace(-0.5, 0.5, len(A))
    if resp == "dB": response = 20 * np.log10(np.abs(fftshift(A / abs(A).max())))
    else: response = np.abs(fftshift(A / abs(A).max()))
    ax_right.plot(freq, response)
    ax_right.set(xlim=[-0.5, 0.5,])
    if resp == "dB": ax_right.set(ylim=[-120, 0])
    ax_right.yaxis.tick_right()

names = ["gaussian", "triangle", "boxcar (rectangle)", "exponential", "hann"]
args = [{'std':7}, {}, {}, {'tau':3.0}, {}]
windows = [signal.windows.gaussian, signal.windows.triang, signal.windows.boxcar, signal.windows.hann]

n = 5
fig, ax = plt.subplots(n, 2, figsize=(8,3*n), sharex='col')

for i in range(n):
    plot_window_response(names[i], windows[i], args[i], ax[i,0], ax[i,1])
fig.text(0.02, 0.5, 'window function', va='center', rotation='vertical')
fig.text(0.98, 0.5, 'abs(Fourier transform)', va='center', rotation='vertical')
ax[n-1,0].set(xlabel="sample")
ax[n-1,1].set(xlabel="normalized frequency\n[cycles per sample]");

```



```

n = 5
fig, ax = plt.subplots(n, 2, figsize=(8,3*n), sharex='col')

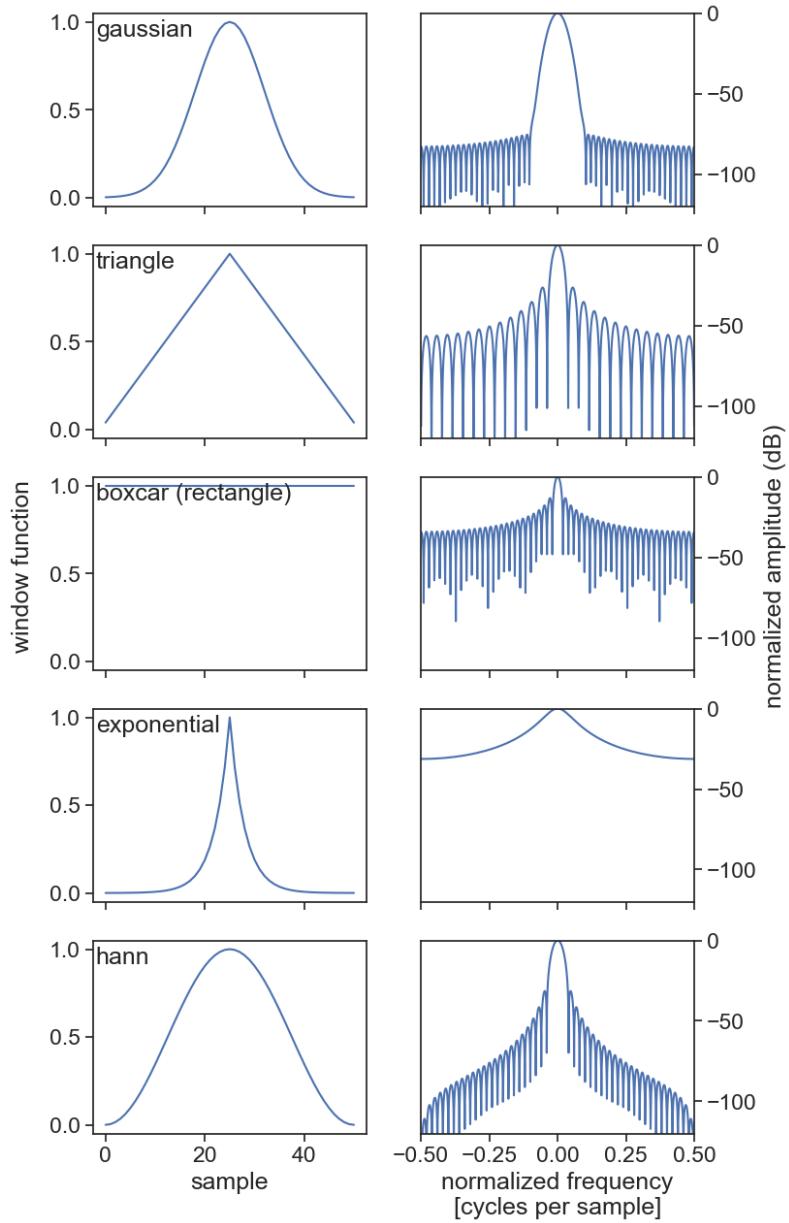
for i in range(n):
    plot_window_response(names[i], windows[i], args[i], ax[i,0], ax[i,1], resp="dB")

```

```

fig.text(0.02, 0.5, 'window function', va='center', rotation='vertical')
fig.text(0.99, 0.5, 'normalized amplitude (dB)', va='center', rotation='vertical')
ax[n-1,0].set(xlabel="sample")
ax[n-1,1].set(xlabel="normalized frequency\n[cycles per sample]");

```

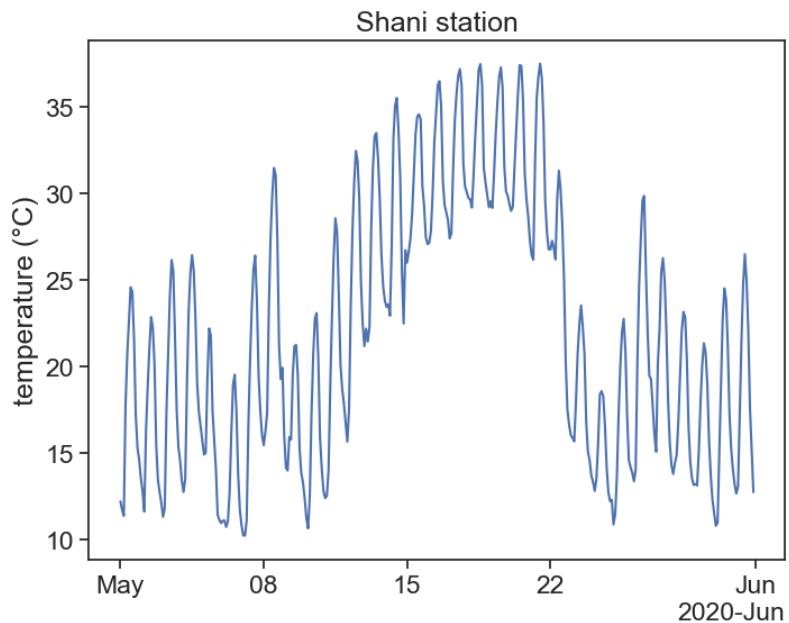


### 53.3 theorem in action

Let's apply the convolution theorem to a real-life time series. See below the temperature of the Shani station over the month of May 2020.

```
def concise(ax):
    """
    Let python choose the best xtick labels for you
    """
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

df = pd.read_csv('shani_temperature.csv', index_col='datetime', parse_dates=True)
df = df.loc['2020-05-01':'2020-05-31', 'T'].to_frame()
fig, ax = plt.subplots(1, 1, figsize=(8,6))
# df['T'].plot(ylabel="temperature (°C)",
#                 xlabel="",
#                 title="Shani station")
ax.plot(df['T'])
ax.set(ylabel="temperature (°C)",
       title="Shani station")
concise(ax);
```



```
# according to the dataframe, temperature is sampled every 2 hours
dt = 2.0 # hours
N = len(df)
time = np.arange(N) * dt
fft_orig = scipy.fft.fft(df['T'].values)
# this shifted version is useful only for plotting, it looks nicer
fft = scipy.fft.fftshift(fft_orig)
xi = scipy.fft.fftfreq(N, dt)
xi = scipy.fft.fftshift(xi)
fft_abs = np.abs(fft)

# boxcar (rectangular) window
width_hours = 24
width_points = int(width_hours/dt)
boxcar = signal.windows.boxcar(width_points)
# integral of boxcar must be 1.0 if we want to take an average
boxcar = boxcar / np.sum(boxcar)
# we need an array of the same size of the original signal
boxcar_array = np.zeros(N)
# boxcar is placed at the beginning
boxcar_array[:width_points] = boxcar
# if we were to leave the boxcar array as above, the average would not be centered
```

```

# let's move the boxcar a half width to the left
boxcar_array = np.roll(boxcar_array, -width_points//2)
# compute Fourier transform of boxcar
fft_boxcar_orig = scipy.fft.fft(boxcar_array)
# shift it only so we can plot it later
fft_boxcar = scipy.fft.fftshift(fft_boxcar_orig)
fft_abs_boxcar = np.abs(fft_boxcar)

# this is the important part
# 1. we multiply the signal's fft with the window's fft
fft_filtered = fft_orig * fft_boxcar_orig
# 2. we apply an inverse Fourier transform. we take only the real part
# because usually the inverse operation yields really small imaginary components
filtered = scipy.fft.ifft(fft_filtered).real

# for comparison's sake, let's apply the usual rolling average
df['rolling1day'] = df['T'].rolling('24H', center=True).mean()

fig, ax = plt.subplots(3, 1, figsize=(8,8), sharex=True)
ax[0].plot(xi, fft_abs, label="temperature", color="tab:blue")
ax[0].set_ylabel("|F|", color="tab:blue", rotation="horizontal", labelpad=10)
ax20 = ax[0].twinx()
ax20.plot(xi, fft_abs_boxcar, label="boxcar", color="tab:orange", alpha=0.5)
ax20.set_ylabel("|F|", color="tab:orange", rotation="horizontal", labelpad=10)
fig.legend(frameon=False, ncol=2, bbox_to_anchor=(0.5,0.95), loc="upper center",)
[t.set_color("tab:blue") for t in ax[0].yaxis.get_ticklines()]
[t.set_color("tab:blue") for t in ax[0].yaxis.get_ticklabels()]
[t.set_color("tab:orange") for t in ax20.yaxis.get_ticklines()]
[t.set_color("tab:orange") for t in ax20.yaxis.get_ticklabels()]

ax[1].plot(xi, fft_abs, label="temperature", zorder=1)
ax[1].set_ylabel("|F|", color="tab:blue", rotation="horizontal", labelpad=10)
ax21 = ax[1].twinx()
ax21.plot(xi, fft_abs_boxcar, label="boxcar", color="tab:orange", alpha=0.5)
ax21.set_ylabel("|F|", color="tab:orange", rotation="horizontal", labelpad=10)
ax[1].set(ylim=[0, 1100])
ax21.set(ylim=[0,1.1])
[t.set_color("tab:blue") for t in ax[1].yaxis.get_ticklines()]
[t.set_color("tab:blue") for t in ax[1].yaxis.get_ticklabels()]
[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklines()]

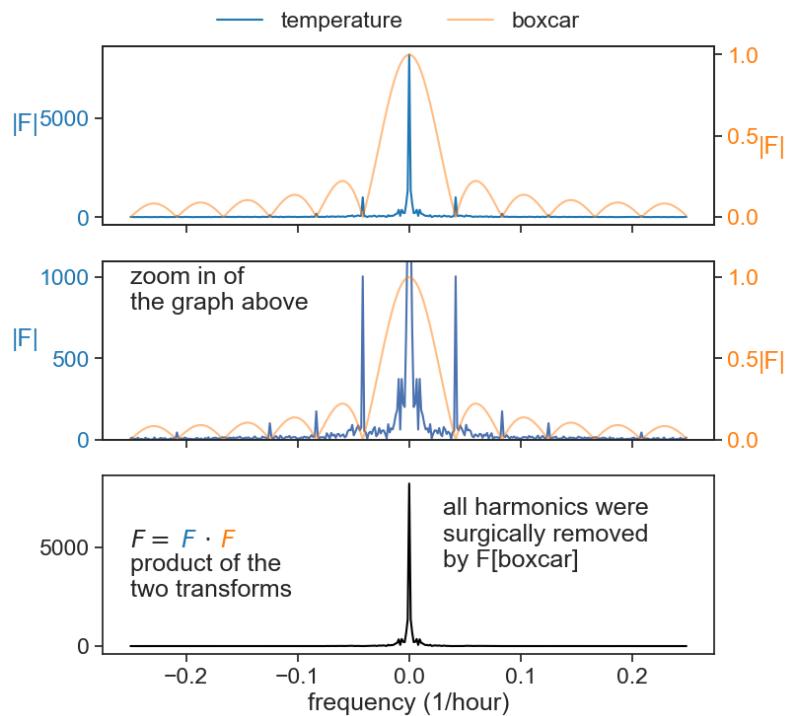
```

```

[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklabels()]
ax21.text(-0.25, 0.80, "zoom in of\nthe graph above")

ax[2].plot(xi, np.abs(scipy.fft.fftshift(fft_filtered)), color="black")
ax[2].text(-0.25, 2500, "product of the\ntwo transforms")
ax[2].text(-0.25, 5000, r"$F=\quad \cdot \quad$")
ax[2].text(-0.205, 5000, r"$F$", color="tab:blue")
ax[2].text(-0.17, 5000, r"$F$", color="tab:orange")
ax[2].text(0.03, 4000, "all harmonics were\nsurgically removed\nby F[boxcar]")
ax[2].set(xlabel="frequency (1/hour)");

```



Let's see how the two methods compare.

```

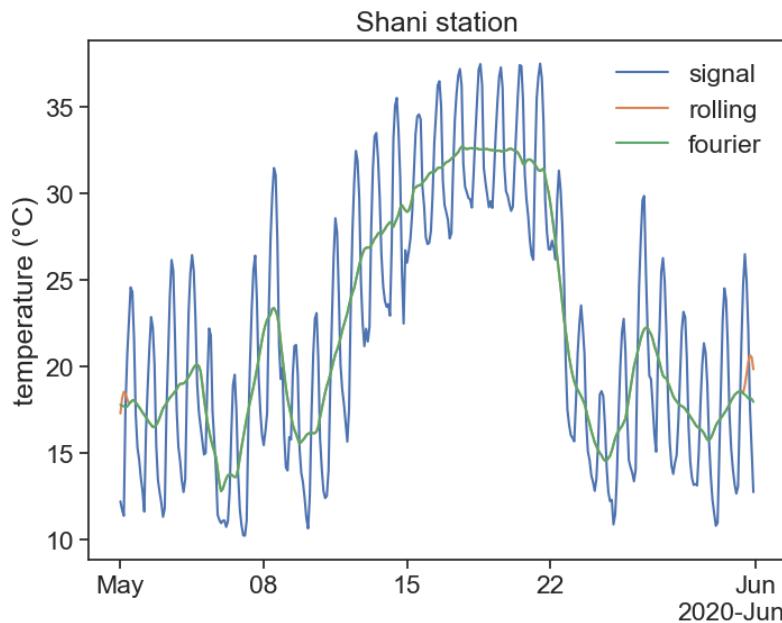
fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['T'], label="signal")
ax.plot(df['rolling1day'], label="rolling")
ax.plot(df.index, filtered, label="fourier")
ax.set(ylabel="temperature (°C)",
       xlabel="",

```

```

    title="Shani station")
ax.legend(frameon=False)
concise(ax);

```



### 53.4 now let's play some more

What would we get if we changed the width of the window? In the example above the boxcar window had a width of 1 day. Let's see what happens when we chose windows of width 2 days and 1.5 days.

```

# boxcar (rectangular) window
width_hours_2days = 48
width_points_2days = int(width_hours_2days/dt)
boxcar_2days = signal.windows.boxcar(width_points_2days)
boxcar_2days = boxcar_2days / np.sum(boxcar_2days)
boxcar_2days_array = np.zeros(N)
boxcar_2days_array[:width_points_2days] = boxcar_2days
boxcar_2days_array = np.roll(boxcar_2days_array, -width_points_2days//2)
fft_boxcar_2days_orig = scipy.fft.fft(boxcar_2days_array)

```

```

fft_boxcar_2days = scipy.fft.fftshift(fft_boxcar_2days_orig)
fft_abs_boxcar_2days = np.abs(fft_boxcar_2days)

width_hours_15days = 36
width_points_15days = int(width_hours_15days/dt)
boxcar_15days = signal.windows.boxcar(width_points_15days)
boxcar_15days = boxcar_15days / np.sum(boxcar_15days)
boxcar_15days_array = np.zeros(N)
boxcar_15days_array[:width_points_15days] = boxcar_15days
boxcar_15days_array = np.roll(boxcar_15days_array, -width_points_15days//2)
fft_boxcar_15days_orig = scipy.fft.fft(boxcar_15days_array)
fft_boxcar_15days = scipy.fft.fftshift(fft_boxcar_15days_orig)
fft_abs_boxcar_15days = np.abs(fft_boxcar_15days)

fft_filtered_20 = fft_orig * fft_boxcar_2days_orig
fft_filtered_15 = fft_orig * fft_boxcar_15days_orig
filtered20 = scipy.fft.ifft(fft_filtered_20).real
filtered15 = scipy.fft.ifft(fft_filtered_15).real

```

We see that when the width is an integer multiple of the fundamental period of oscillation of the signal, the filtering will eliminate all the harmonics. However, if we are not careful with our window width choice (e.g. 36 hours), many of the harmonics will be left untouched!

```

fig, ax = plt.subplots(2, 1, figsize=(8,8), sharex=True)

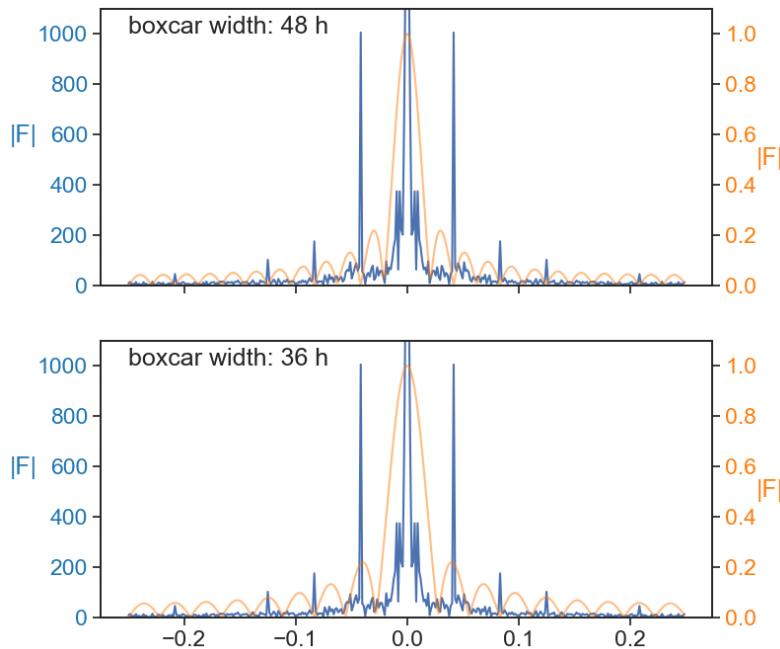
ax[0].plot(xi, fft_abs, label="temperature")
ax[0].set_ylabel("|F|", color="tab:blue", rotation="horizontal", labelpad=10)
ax21 = ax[0].twinx()
ax21.plot(xi, fft_abs_boxcar_2days, label="boxcar", color="tab:orange", alpha=0.5)
ax21.set_ylabel("|F|", color="tab:orange", rotation="horizontal", labelpad=10)
ax[0].set(ylim=[0, 1100])
ax21.set(ylim=[0,1.1])
[t.set_color("tab:blue") for t in ax[0].yaxis.get_ticklines()]
[t.set_color("tab:blue") for t in ax[0].yaxis.get_ticklabels()]
[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklines()]
[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklabels()]
ax[0].text(-0.25, 1000, "boxcar width: 48 h")

```

```

ax[1].plot(xi, fft_abs, label="temperature")
ax[1].set_ylabel("|F|", color="tab:blue", rotation="horizontal", labelpad=10)
ax21 = ax[1].twinx()
ax21.plot(xi, fft_abs_boxcar_15days, label="boxcar", color="tab:orange", alpha=0.5)
ax21.set_ylabel("|F|", color="tab:orange", rotation="horizontal", labelpad=10)
ax[1].set(ylim=[0, 1100])
ax21.set(ylim=[0, 1.1])
[t.set_color("tab:blue") for t in ax[1].yaxis.get_ticklines()]
[t.set_color("tab:blue") for t in ax[1].yaxis.get_ticklabels()]
[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklines()]
[t.set_color("tab:orange") for t in ax21.yaxis.get_ticklabels()]
ax[1].text(-0.25, 1000, "boxcar width: 36 h");

```



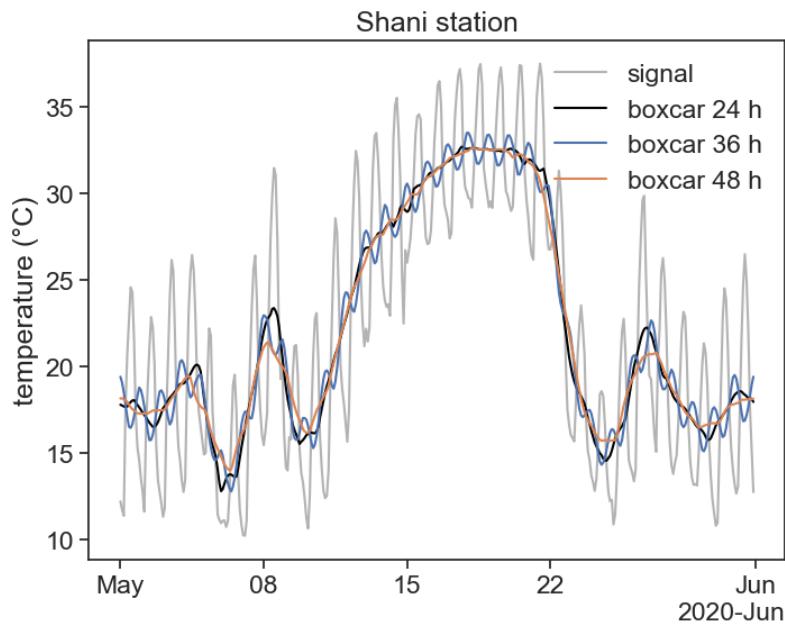
To sum up, let's see what happens when we chose different window widths.

```

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['T'], label="signal", color=[0.7]*3)
ax.plot(df.index, filtered, label="boxcar 24 h", color="black")
ax.plot(df.index, filtered15, label="boxcar 36 h")
ax.plot(df.index, filtered20, label="boxcar 48 h")

```

```
ax.set(ylabel="temperature (°C)",  
       xlabel="",  
       title="Shani station")  
ax.legend(frameon=False, loc="upper right")  
concise(ax);
```



## 54 practice 1

Download this zip file before you start. It contains all required datasets for the fft practice pages.

In this practice we will find the frequency of days and years from temperature data.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.ensemble import RandomForestRegressor
import concurrent.futures
from datetime import datetime, timedelta
from sklearn.cluster import KMeans
import math
import scipy
from scipy.signal import find_peaks

# %matplotlib widget
```

Import data

```
df = pd.read_csv('shani_temperature.csv', index_col='datetime', parse_dates=True)
df
```

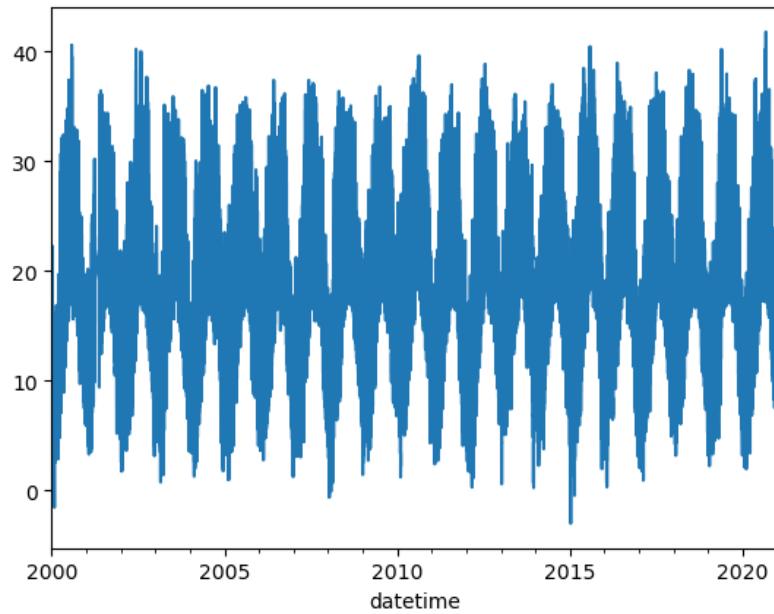
---

	T
datetime	
2000-01-01 00:00:00	16.791667
2000-01-01 02:00:00	16.975000
2000-01-01 04:00:00	16.825000
2000-01-01 06:00:00	17.050000

T	
datetime	
2000-01-01 08:00:00	19.900000
...	...
2020-12-31 14:00:00	17.341667
2020-12-31 16:00:00	14.900000
2020-12-31 18:00:00	13.308333
2020-12-31 20:00:00	12.925000
2020-12-31 22:00:00	12.983333

plot

```
df['T'].plot()
```



#### 54.0.1 Apply FFT

```
dt = 1/12 # 2hr interval is like 1/12 of a day
dt = 2 # 2hr interval is like 1/12 of a day
```

```
N = len(df)
t = np.arange(0,int(len(df)),dt)
x = df['T'].values
# standardize x:
x = (x - np.mean(x))/np.std(x)
```

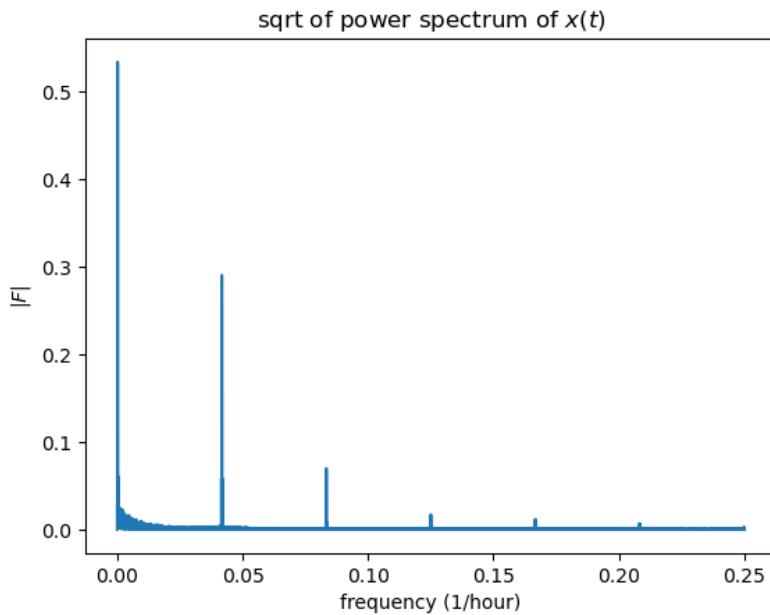
```
fft = scipy.fft.fft(x) / N
k = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)
```

Keep only positive k values

```
fft_abs = fft_abs[k>=0]
k = k[k>=0]
```

```
fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (1/hour)",
       ylabel=r"$|F|$",
       title=r"sqrt of power spectrum of $x(t)$")

peak_freq = k[fft_abs.argmax()]
# print(f'Highest peak at {peak_freq:.5f} per day')
# print(f'If we divide 1 by that value we get {1/peak_freq:.2f} :)')
# print(f'Keep in mind that the resolution is {(np.median(np.diff(k))):.7f} per day')
```

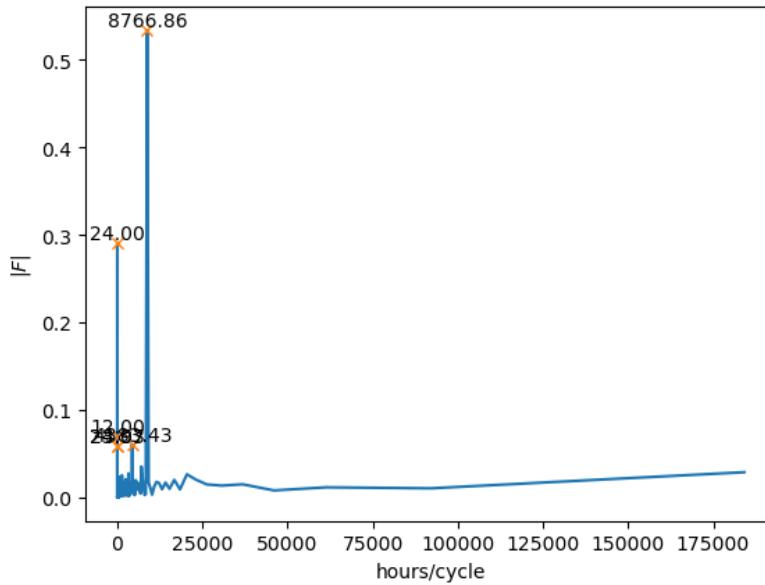


The  $k$  values are in  $\frac{1}{\text{hour}}$  units, which are not that intuitive for finding the frequency of a year or days. If we compute  $1/k$  we will get a more intuitive unit of  $\frac{\text{hour}}{\text{cycle}}$ .

```
peaks, _ = find_peaks(fft_abs, threshold=0.05)
```

```
np.seterr(divide='ignore')

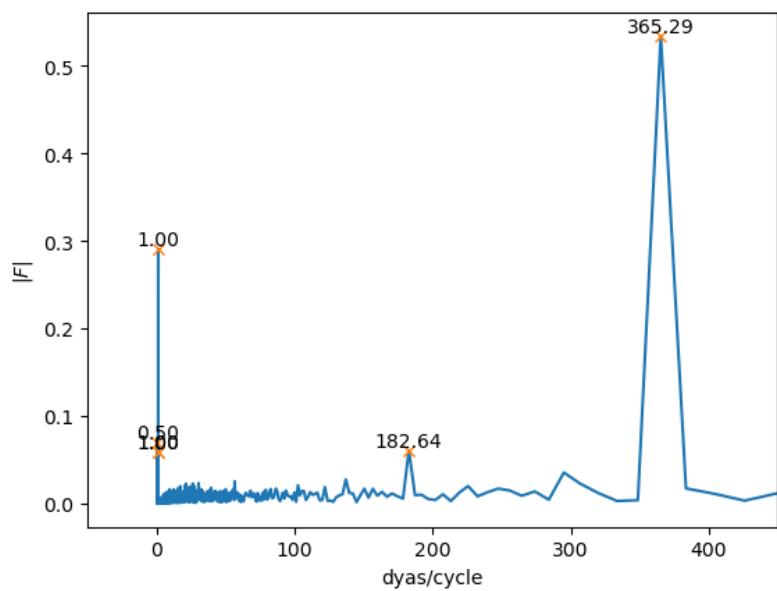
fig, ax = plt.subplots()
ax.plot(1/k, fft_abs)
ax.set(xlabel="hours/cycle",
       ylabel=r"$|F|$")
ax.plot(1/k[peaks], fft_abs[peaks], "x")
# ax.set_xlim(-50,450)
for peak in peaks:
    ax.text(1/k[peak], fft_abs[peak], f'{1/k[peak]:.2f}', ha='center', va='bottom')
None
# print(1/k[peaks])
```



We see a clear peak at 24 indicating that the fft detected a strong signal at 24 hrs per cycle, which is obviously the signal of a day. Now let's modify the units again to make them more intuitive for lower frequencies. We will divide by 24 to get the unit of  $\frac{\text{day}}{\text{cycle}}$ .

```
fig, ax = plt.subplots()
ax.plot(1/k/24, fft_abs)
ax.set(xlabel="days/cycle",
       ylabel=r"$|F|$")
ax.plot(1/k[peaks]/24, fft_abs[peaks], "x")
ax.set_xlim(-50,450)
for peak in peaks:
    ax.text(1/k[peak]/24, fft_abs[peak], f'{1/k[peak]/24:.2f}', ha='center', va='bottom')
None
print(1/k[peaks]/24)
```

[365.28571429 182.64285714	1.0027451	1.	0.99726989
0.5	]		



## 55 practice 2

In this practice we will find the frequency of an interesting oscillation in transpiration measured from a grapevine leaf using a LICOR 6400. We thank [Yotam Zait](#) for providing us this interesting measurement.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import scipy
import seaborn as sns
sns.set(style="ticks", font_scale=1.1)
pd.set_option('mode.chained_assignment', None)
# %matplotlib widget

df = pd.read_csv("grapevine_E.csv",
                  index_col='date', parse_dates=True)
df
```

---

	E
date	
2023-07-10 10:37:30	0.004116
2023-07-10 10:38:30	0.004012
2023-07-10 10:39:30	0.003694
2023-07-10 10:40:30	0.003506
2023-07-10 10:41:30	0.003361
...	...
2023-07-13 20:14:23	0.000059
2023-07-13 20:15:23	0.000069
2023-07-13 20:16:23	0.000061
2023-07-13 20:17:23	0.000063
2023-07-13 20:18:23	0.000061

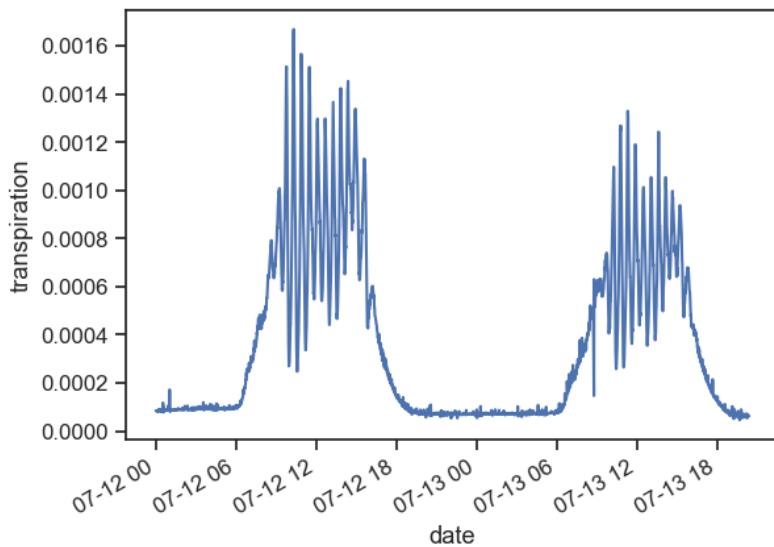
---

```

start = '2023-07-12'
end = '2023-07-13'
df = df[start:end]

fig, ax = plt.subplots()
df['E'].loc[df['E']<4.5e-5] = np.nan
df['E'].ffill(inplace=True)
df['E'].plot()
ax.set(ylabel="transpiration")

```



The measurements shown above suggest that there is a clear oscillatory pattern in the grapevine transpiration. Zooming in, we see that the oscillation period is about once every half hour. Let's find out exactly what the period is.

```

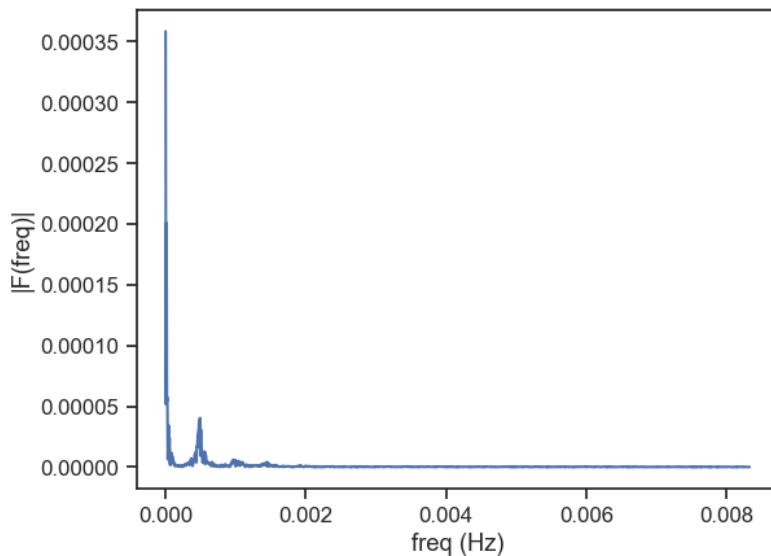
N = len(df)
F = scipy.fft.fft(df['E'].values) / N
n = np.arange(N)
# the basic time unit here is "second"
# measurements every 60 seconds
dt = 60 # seconds
freq = scipy.fft.fftfreq(N, d=dt)

```

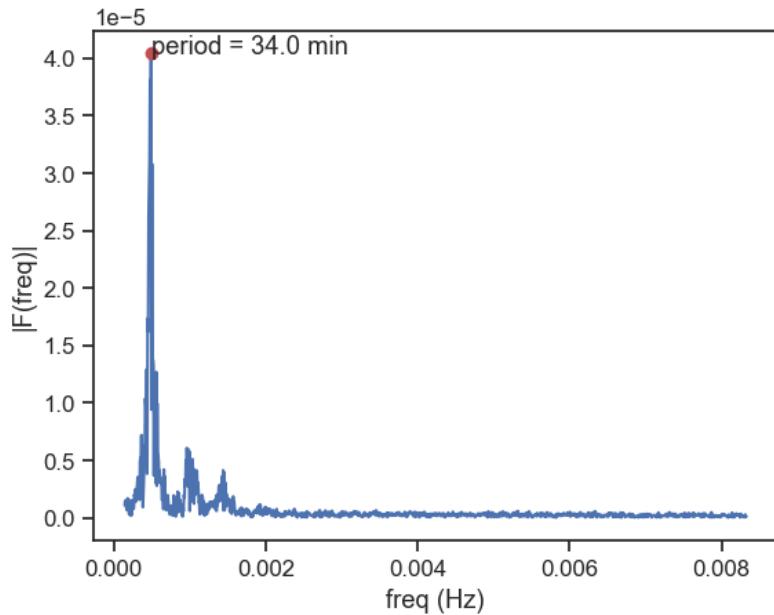
```

# Get the one-sided spectrum
n_oneside = N//2
# get the one side frequency
k_oneside = freq[:n_oneside]
F_oneside = F[:n_oneside]
fig, ax = plt.subplots()
ax.plot(k_oneside, np.abs(F_oneside))
ax.set(xlabel='freq (Hz)',
       ylabel='|F(freq)|')

```



```
fig, ax = plt.subplots()
ax.plot(k_oneside, np.abs(F_oneside))
ax.plot([k_max], [F_max], 'ro')
ax.text(k_max, F_max, f"period = {period / 60:.1f} min")
ax.set(xlabel='freq (Hz)',
       ylabel='|F(freq)|')
```



## 56 practice 3

Here we will find the tempo of a song.

First let's listen to the song:

[Link](#) to the full song on youtube.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from scipy.io import wavfile
import math
import scipy
from scipy.signal import find_peaks

# %matplotlib widget
```

Import audio timeseries

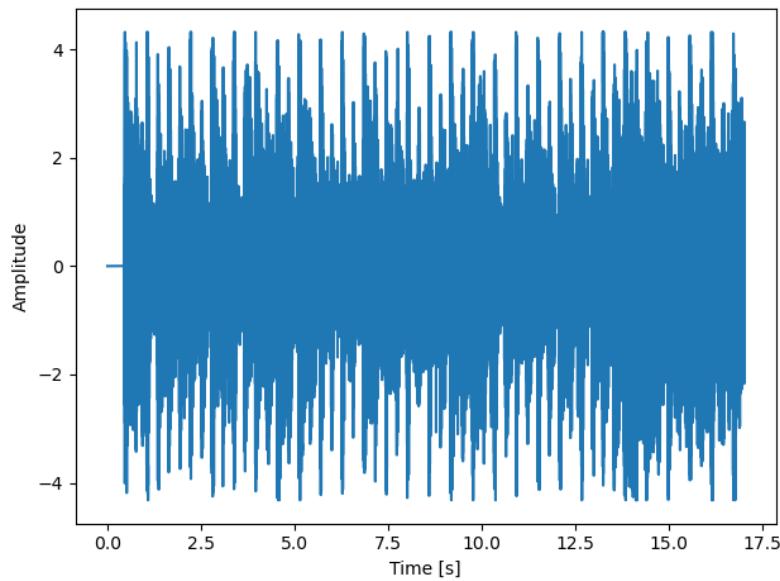
```
samplerate, audio_stereo = wavfile.read('stayin_alive.wav')
audio = audio_stereo[:, 0]
```

Standardize and plot.

```
audio = (audio - np.mean(audio))/np.std(audio)
length = audio.shape[0] / samplerate
time = np.linspace(0., length, audio.shape[0])

fig, ax = plt.subplots()
ax.plot(time, audio)
ax.set_xlabel("Time (s)")
ax.set_ylabel("Amplitude")

plt.tight_layout()
```



Let's see what is the sample rate of the song, that is, how many data points we have per second.

```
samplerate
```

48000

## 56.1 apply FFT

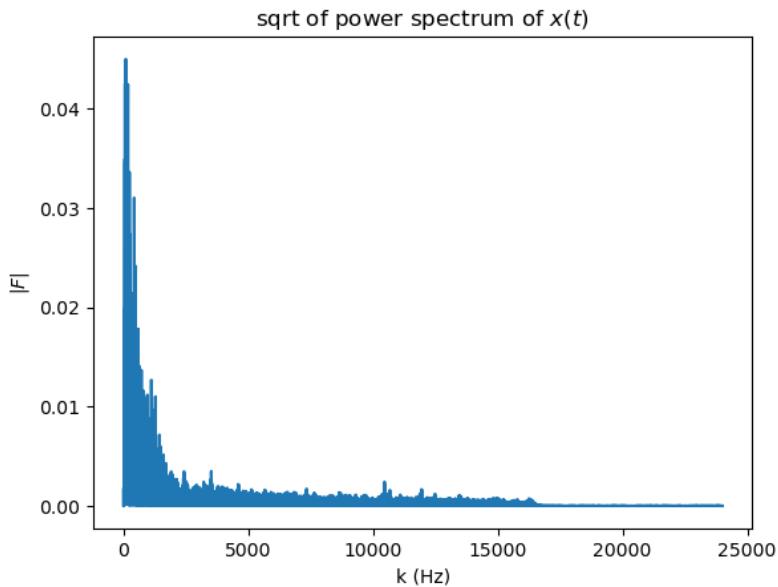
```
dt = 1/samplerate
t = time
N = audio.shape[0]
x = audio

fft = scipy.fft.fft(x) / N
k = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)
```

keep only positive k values

```
fft_abs = fft_abs[k>=0]
k = k[k>=0]
```

```
fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="k (Hz)",
       ylabel="$|F|$", 
       title="sqrt of power spectrum of $x(t)$");
```



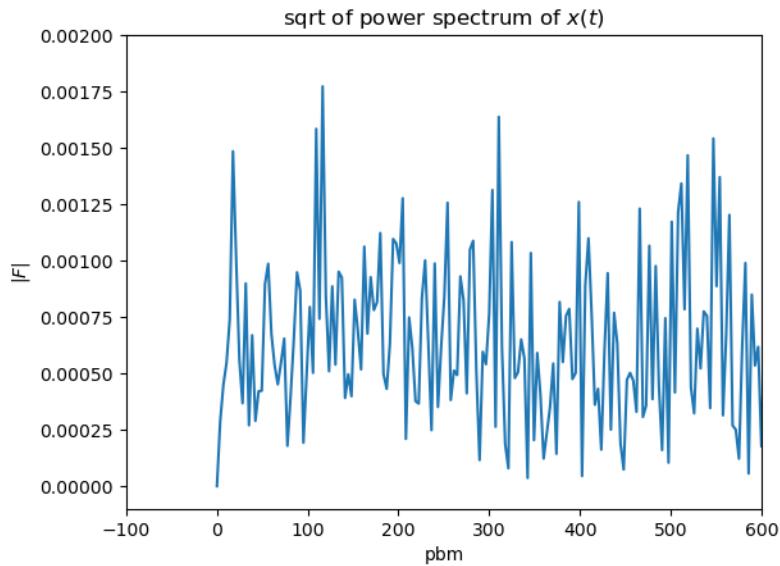
The relevant frequency spectrum for tempo is very low in the 0.5-6 Hz range. Think about it, a clock ticks at 1 Hz, that will be like a slow song. Double that speed will be 2 Hz. Actually, in the music world we don't talk in Hz when dealing with tempo, we use beats per minute (bpm) instead. A clock ticks at 60 bpm, a song double that speed will be at 120 bpm. The conversion factor from Hz to bpm is simple, 1Hz = 60 bpm. So let's zoom in to that range:

```
bpm = k*60
fig, ax = plt.subplots()
ax.plot(bpm, fft_abs)
ax.set(xlabel="pbm",
```

```

    ylabel="$|F|",
    title="sqrt of power spectrum of $x(t)$";
ax.set_xlim(-100,600);
ax.set_ylim(-0.0001,0.002);

```



We don't see any significant peak. That is because the samplerate is very high and it is picking on the actual notes that are being played at every beat making the peak not clear. Lets filter the timeseries to show the beat peaks more clearly. We will do that by resampling the max values of staggered pools of values.

```

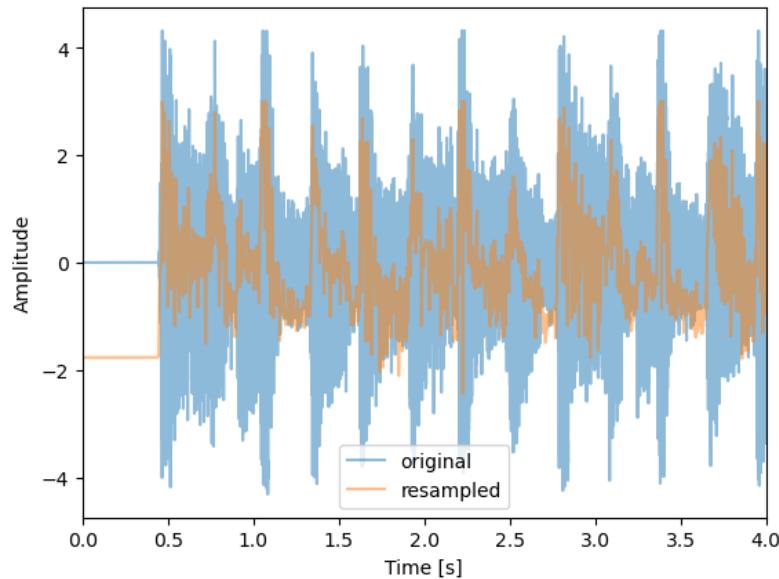
factor = 200
data_reshaped = audio.reshape(int(len(audio)/factor), factor)
max_values = np.max(data_reshaped, axis=1)
max_values = (max_values - np.mean(max_values))/np.std(max_values)
new_samplerate = samplerate/factor
N = max_values.shape[0]
new_length = N / new_samplerate
new_time = np.linspace(0., new_length, N)

```

```

fig, ax = plt.subplots()
ax.plot(time, audio, alpha=0.5, label='original')
ax.plot(new_time, max_values, alpha=0.5, label='resampled')
ax.set_xlabel("Time [s]")
ax.set_ylabel("Amplitude")
ax.set_xlim(0,4)
ax.legend()

```



## 56.2 Apply FFT on resampled timeseries

```

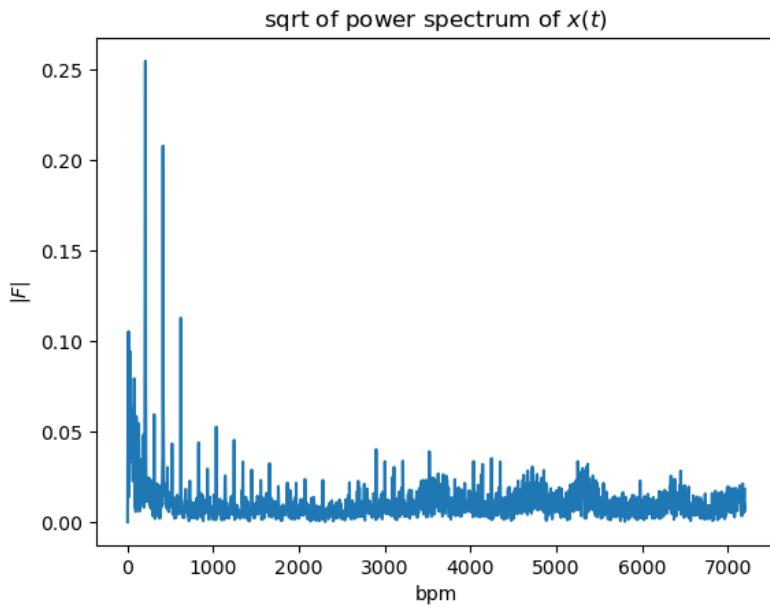
dt = 1/new_samplerate
t = new_time
x = max_values

fft = scipy.fft.fft(x) / N
k = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)

fft_abs = fft_abs[k>=0]
k = k[k>=0]

```

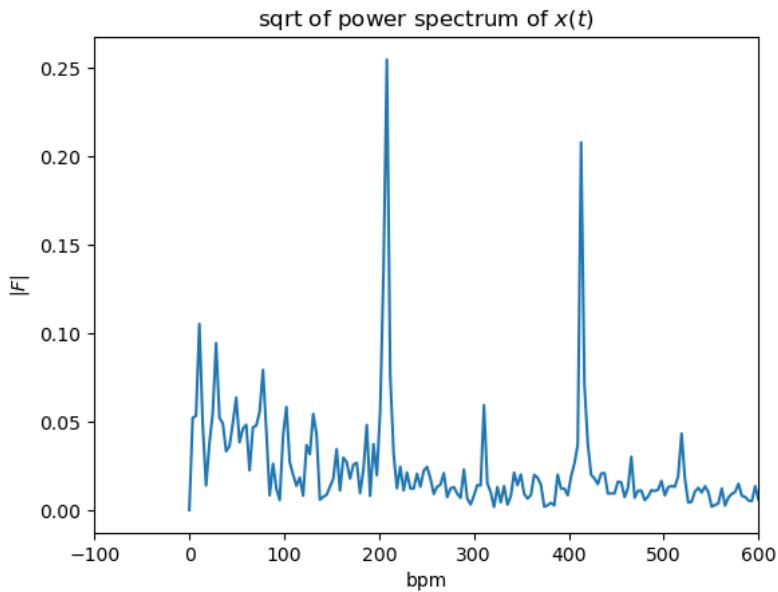
```
bpm = k*60
fig, ax = plt.subplots()
ax.plot(bpm, fft_abs)
ax.set(xlabel="bpm",
       ylabel="$|F|$", 
       title=r"sqrt of power spectrum of $x(t)$");
```



Lets zoom into the relevant range:

```
bpm = k*60
fig, ax = plt.subplots()
ax.plot(bpm, fft_abs)
ax.set(xlabel="bpm",
       ylabel="$|F|$", 
       title=r"sqrt of power spectrum of $x(t)$");
ax.set_xlim(-100,600);
print(f'Highest peak at {bpm[fft_abs.argmax()]:.2f} bpm')
```

Highest peak at 208.24 bpm



We got a strong peak on 208.24 bpm.

Let's check what is the “agreed” bpm of the song, we will do it in two ways:

1. google search: [stayin alive bpm](#)
2. using the site [bpmfinder](#) that computes the bpm for a given youtube link. They don't use fft, you can read about their method [here](#).

## 57 filtering 1

Download this csv file before you start. It contains the dataset used in this notebook.

In this practice we will filter frequencies in our data.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.ensemble import RandomForestRegressor
import concurrent.futures
from datetime import datetime, timedelta
from sklearn.cluster import KMeans
import math
import scipy
from scipy.signal import find_peaks

# %matplotlib widget
```

Import data

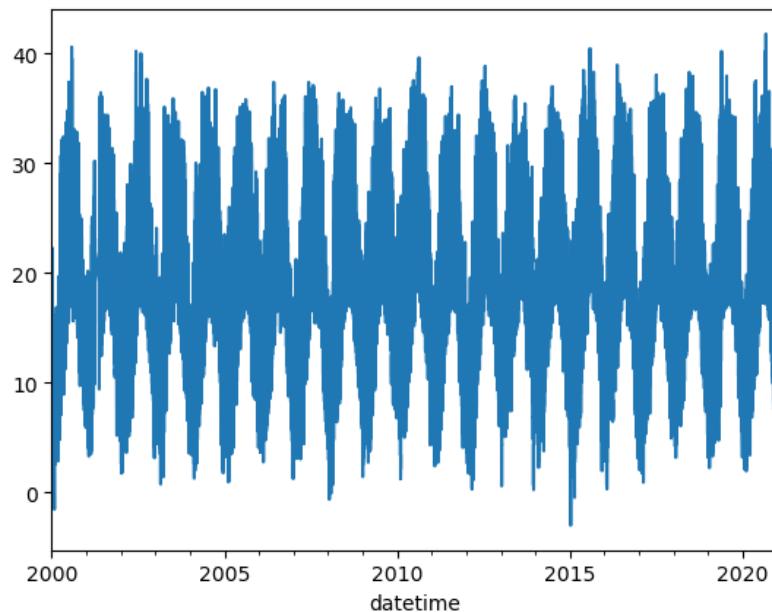
```
df = pd.read_csv('shani_temperature.csv', index_col='datetime', parse_dates=True)
df
```

	T
datetime	
2000-01-01 00:00:00	16.791667
2000-01-01 02:00:00	16.975000
2000-01-01 04:00:00	16.825000
2000-01-01 06:00:00	17.050000
2000-01-01 08:00:00	19.900000

T	
datetime	
...	...
2020-12-31 14:00:00	17.341667
2020-12-31 16:00:00	14.900000
2020-12-31 18:00:00	13.308333
2020-12-31 20:00:00	12.925000
2020-12-31 22:00:00	12.983333

plot

```
df ['T'].plot()
```



### 57.0.1 Apply FFT

```
dt = 2 # 2hr interval is like 1/12 of a day
N = len(df)
t = np.arange(0,int(len(df)),dt)
```

```
x = df['T'].values
# standardize x:
x = (x - np.mean(x))/np.std(x)
```

```
fft = scipy.fft.fft(x) / N
k = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)
```

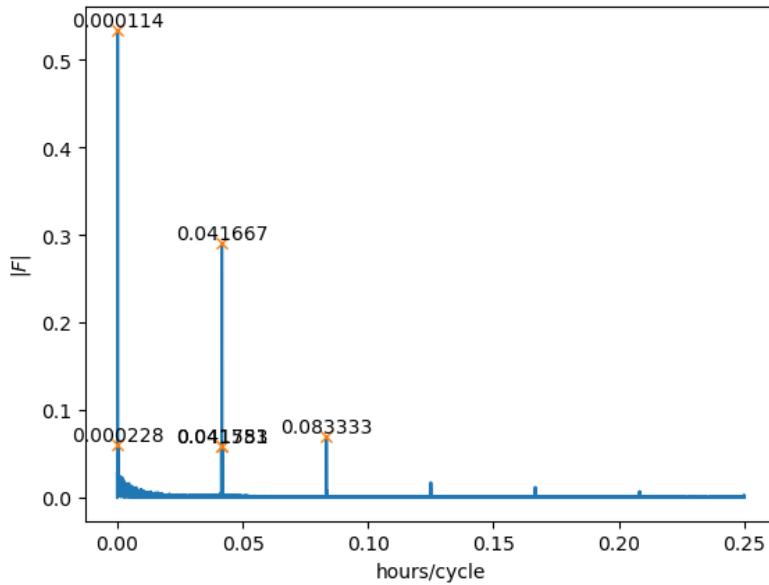
Keep only positive k values

```
fft_abs = fft_abs[k>=0]
k = k[k>=0]
```

```
peaks, _ = find_peaks(fft_abs, threshold=0.05)
```

```
np.seterr(divide='ignore')

fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="hours/cycle",
       ylabel=r"$|F|$")
ax.plot(k[peaks], fft_abs[peaks], "x")
# ax.set_xscale('log')
# ax.set_xlim(-50,450)
for peak in peaks:
    ax.text(k[peak], fft_abs[peak], f'{k[peak]:.6f}', ha='center', va='bottom')
None
# print(1/k[peaks])
```



## 57.1 filtering

By transforming our signal into an array of frequencies, each with its own weight, we've prepared it for manipulation. This array represents the composite frequencies that construct our original signal. Through manipulation—specifically, filtering—we can adjust the weights of these frequencies, often zeroing some out. Filtering allows us to modify the original signal in a controlled manner.

```
def fft_filter(array, condition):
    # this function receives 2 arguments:
    # array - np array of the timeseries that needs filtering
    # condition - np array of a mask of boolean values
    # true values will be kept and false values will be set to 0

    # FFT the signal
    sig_fft = scipy.fft.fft(array)

    # copy the FFT results
    sig_fft_filtered = sig_fft.copy()
```

```
sig_fft_filtered[~condition] = 0

# get the filtered signal in time domain
filtered = scipy.fft.ifft(sig_fft_filtered)
return filtered

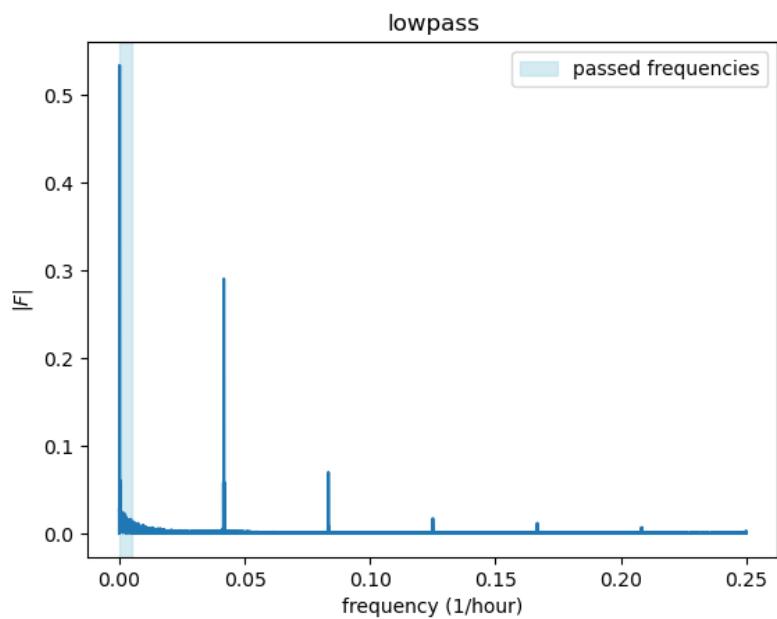
x = df['T'].values
freq = scipy.fft.fftfreq(N, dt)
```

## 57.2 low-pass filter

```
cut_off = 0.005

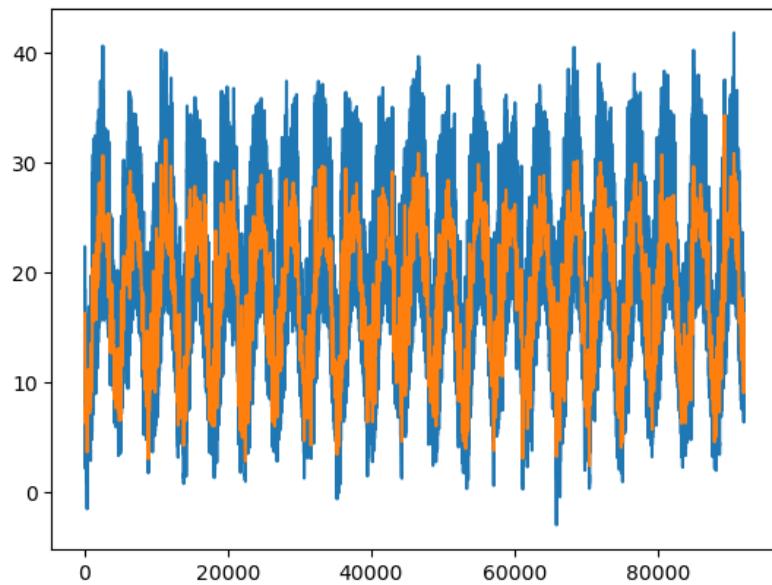
fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (1/hour)",
       ylabel=r"$|F|$",
       title=r"lowpass")

ax.axvspan(0, cut_off, color='lightblue', alpha=0.5, label='passed frequencies')
ax.legend()
```



```
fig, ax = plt.subplots()
ax.plot(x)
condition = np.abs(freq) < cut_off
ax.plot(fft_filter(x, condition))
```

```
/opt/anaconda3/lib/python3.9/site-packages/matplotlib/cbook/__init__.py:1298: ComplexWarning: 0
    return np.asarray(x, float)
```



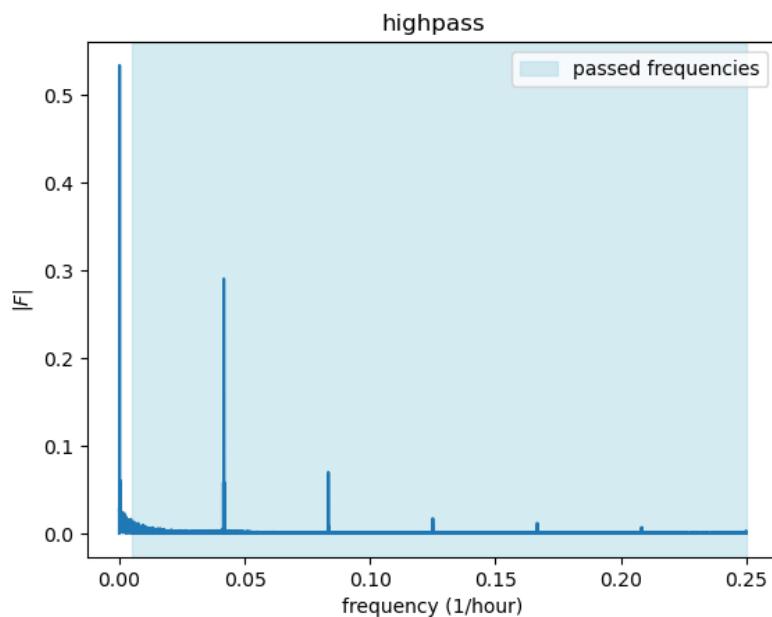
By applying a lowpass filter we removed the high frequency daily signal while keeping the low frequency signal of the yearly oscillation.

### 57.3 high-pass filter

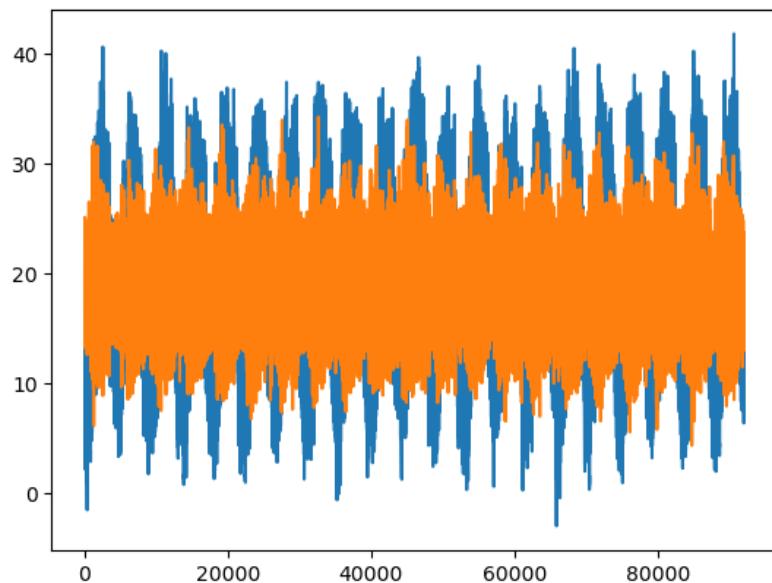
```
cut_off = 0.005

fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (1/hour)",
       ylabel=r"$|F|$",
       title=r"highpass")

# ax.axvspan(0, cut_off, color='lightblue', alpha=0.5)
ax.axvspan(cut_off, np.max(k), color='lightblue', alpha=0.5, label='passed frequencies')
ax.legend()
```



```
fig, ax = plt.subplots()
ax.plot(x)
condition = np.abs(freq) > cut_off
ax.plot(fft_filter(x, condition) + np.mean(x))
```



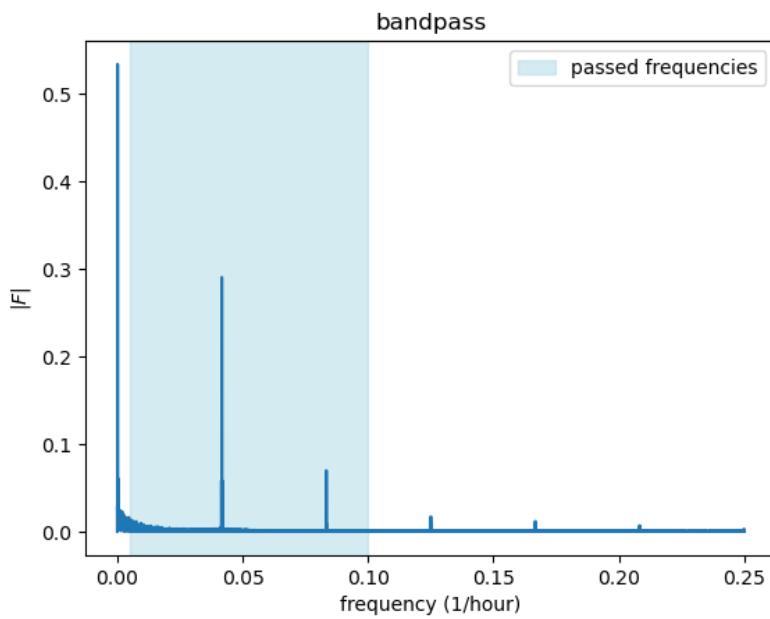
By applying a highpass filter we removed the low frequency signal of the yearly oscillation while keeping the high frequency daily signal

## 57.4 band-pass filter

```
low_thresh = 0.005
high_thresh = 0.1

fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (1/hour)",
       ylabel=r"$|F|$", title=r"bandpass")

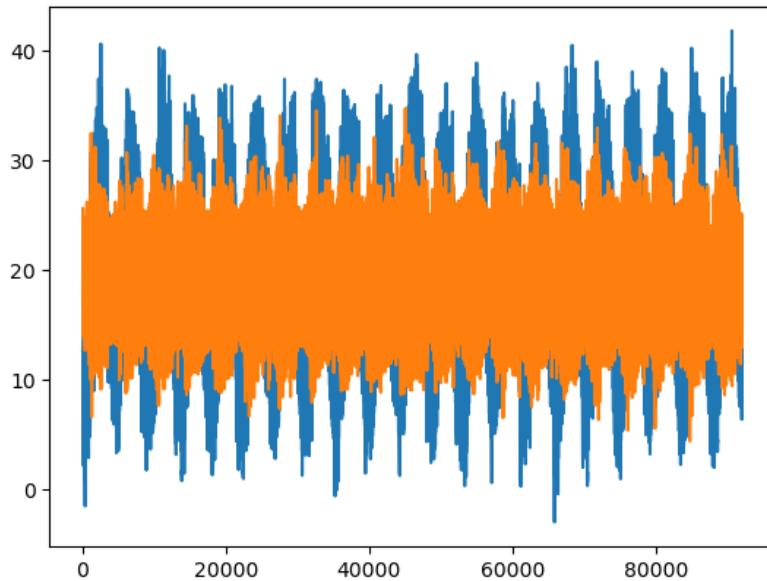
ax.axvspan(low_thresh, high_thresh, color='lightblue', alpha=0.5, label='passed frequencies')
ax.legend()
```



```

fig, ax = plt.subplots()
ax.plot(x)
condition = (np.abs(freq) > low_thresh) & (np.abs(freq) < high_thresh)
ax.plot(fft_filter(x, condition) + np.mean(x))

```



By applying a bandpass filter we removed the low frequency signal of the yearly oscillation, and smoothed the daily signal by removing high frequencies. The results are a smooth daily signal without the yearly seasonal component.

## 57.5 band-stop filter

The fourth common type of filter, alongside low-pass, high-pass, and band-pass filters, is the band-stop filter (also known as a notch filter). A band-stop filter attenuates frequencies within a certain range, allowing frequencies outside of that range to pass through. It's essentially the opposite of a band-pass filter, which allows frequencies within a certain range to pass while attenuating frequencies outside of that range.

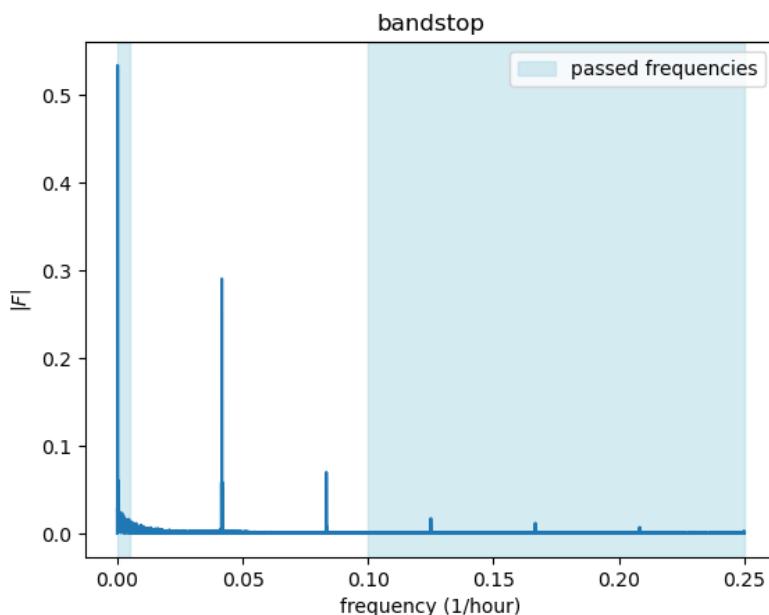
```

low_thresh = 0.005
high_thresh = 0.1

fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (1/hour)",
       ylabel=r"$|F|$", title=r"bandstop")

ax.axvspan(0, low_thresh, color='lightblue', alpha=0.5)
ax.axvspan(high_thresh, np.max(k), color='lightblue', alpha=0.5, label='passed frequencies')
ax.legend()

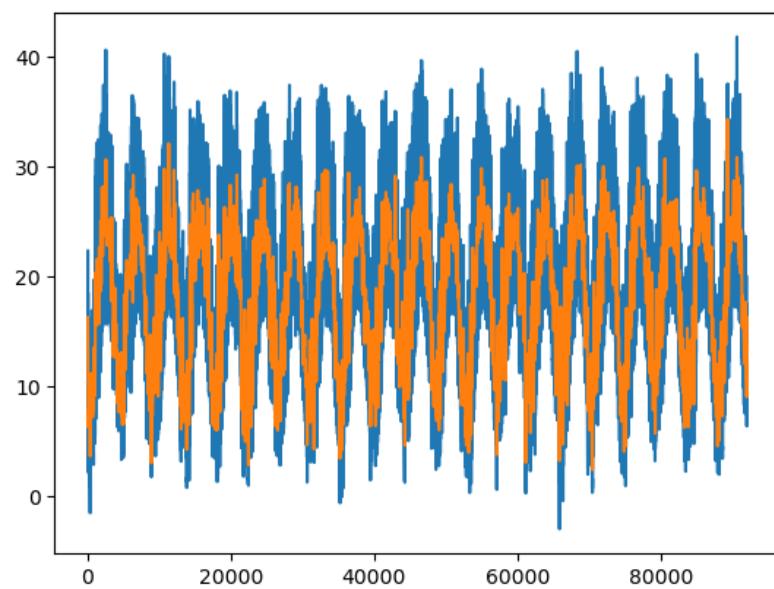
```



```

fig, ax = plt.subplots()
ax.plot(x)
condition = ~(np.abs(freq) > low_thresh) & (np.abs(freq) < high_thresh)
ax.plot(fft_filter(x, condition))

```



By applying a bandstop filter we keep the low frequency signal of the yearly oscillation, and the high frequency noise, but we remove the daily signal by removing the band that corresponds with daily oscillation.

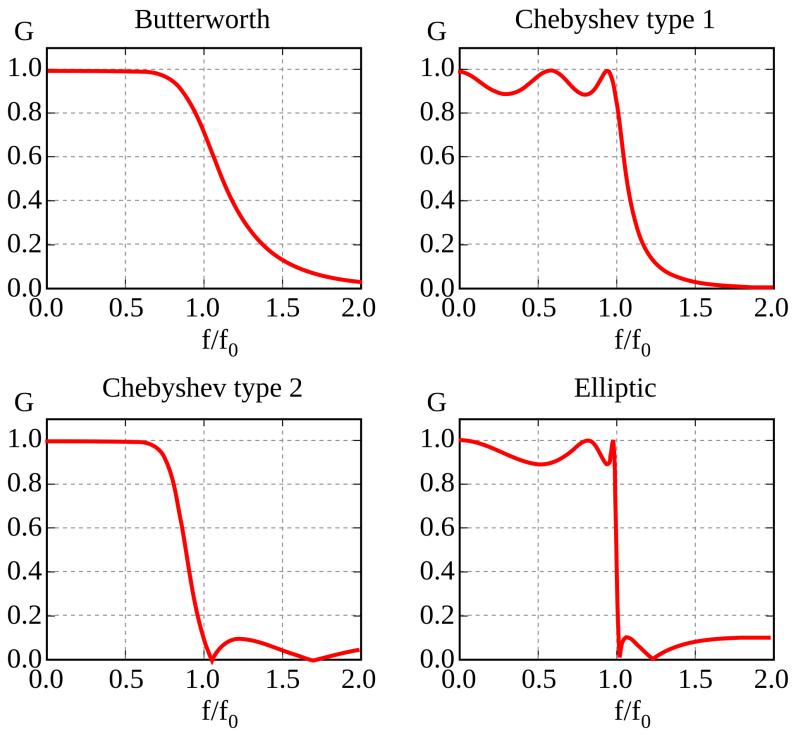
## 58 filtering 2

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.ensemble import RandomForestRegressor
import concurrent.futures
from datetime import datetime, timedelta
from sklearn.cluster import KMeans
import math
import scipy
from scipy.signal import find_peaks
from scipy.signal import sosfiltfilt, butter

# %matplotlib widget
```

The subject of frequency filtering and signal processing is an integral part of our day to day life. It is being used in audio processing, wired and wireless communication, electronics and many more. So, obviously there are many tools in python to apply such filters.

Its Important to note that the manual filtering method we applied in the previous notebook is rarely used as the sharp cuts in the frequencies introduce undesired “ringing” (Gibbs phenomenon) ([more info](#)). Many different filters have been developed to overcome this phenomenon by moderating the “cut”. Here are a few famous ones:

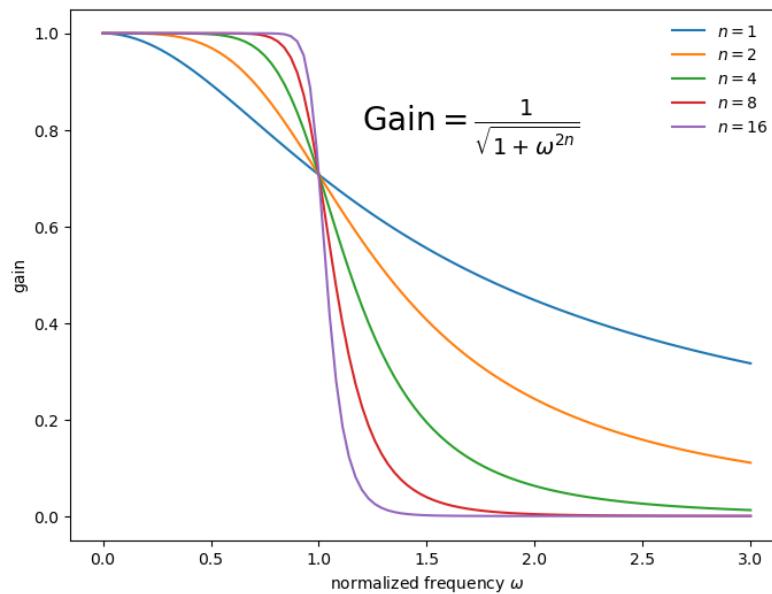


In this notebook, we will be using the butterworth filter from the `scipy.signal.butter` package. Later we will need to specify the order for the filter. What does that mean? The order controls the slope of the decent in the filter as seen here:

```
def bw(omega, n):
    return 1.0 / np.sqrt(1.0 + omega**(2*n))

fig, ax = plt.subplots(1, figsize=(8,6))
omega = np.linspace(0.0,3.0,101)
n_list = [1, 2, 4, 8, 16]
for n in n_list:
    ax.plot(omega, bw(omega, n), label=fr"$n={n}$")
ax.legend(frameon=False)
ax.set(xlabel=r"normalized frequency $\omega$",
       ylabel="gain")
ax.text(1.2, 0.8, r"Gain$=\frac{1}{\sqrt{1+\omega^{2n}}}$", fontsize=20)

Text(1.2, 0.8, 'Gain$=\frac{1}{\sqrt{1+\omega^{2n}}}$')
```



Import data

```
df = pd.read_csv('shani_temperature.csv', index_col='datetime', parse_dates=True)
df
```

T	
datetime	
2000-01-01 00:00:00	16.791667
2000-01-01 02:00:00	16.975000
2000-01-01 04:00:00	16.825000
2000-01-01 06:00:00	17.050000
2000-01-01 08:00:00	19.900000
...	...
2020-12-31 14:00:00	17.341667
2020-12-31 16:00:00	14.900000
2020-12-31 18:00:00	13.308333
2020-12-31 20:00:00	12.925000
2020-12-31 22:00:00	12.983333

plot

```
x = df['T'].values
```

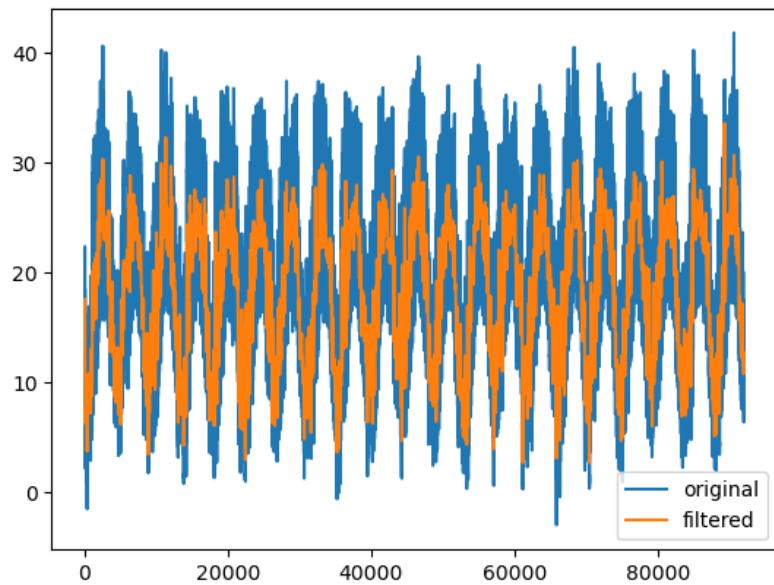
## 58.1 lowpass

```
frequency_sample = 0.5 # in units of 1/hr. we choose 0.5 because we sample every 2hrs.
cut_off = 0.005

sos = butter(4,                      #filter order = how steep is the slope
             cut_off,            # cutoof value in units of the frequency sample
             btype='low',         #type of filter
             output='sos',        # "sos" stands for "Second-Order Sections."
             fs=0.5              #frequency sample = how many samples per 1 unit.
             )

# apply filter to the data:
y = sosfiltfilt(sos, x)

fig, ax = plt.subplots()
ax.plot(x, label='original')
ax.plot(y, label='filtered')
ax.legend()
```



## 58.2 highpass

```

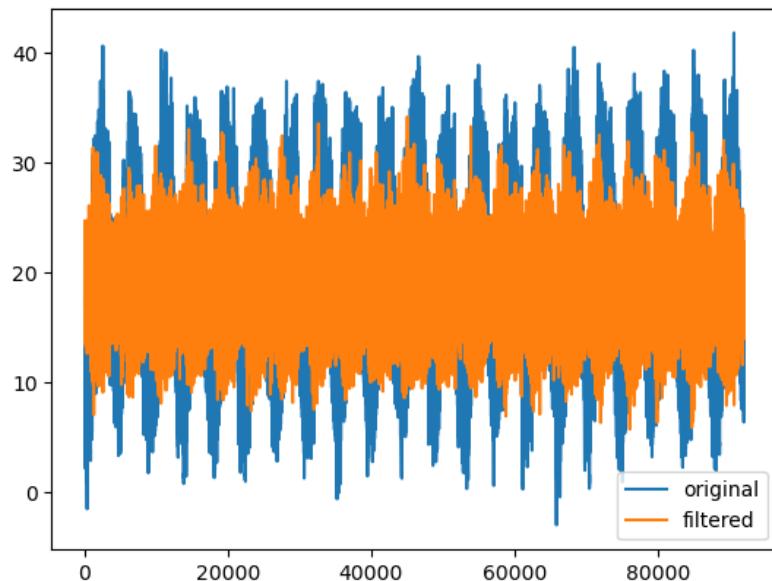
frequency_sample = 0.5 # in units of 1/hr. we choose 0.5 because we sample every 2hrs.
cut_off = 0.005

sos = butter(4,                  #filter order = how steep is the slope
            cut_off,        # cutoff value in units of the frequency sample
            btype='high',   #type of filter
            output='sos',  # "sos" stands for "Second-Order Sections."
            fs=0.5         #frequency sample = how many samples per 1 unit.
            )

# apply filter to the data:
y = sosfiltfilt(sos, x) + np.mean(x)

fig, ax = plt.subplots()
ax.plot(x, label='original')
ax.plot(y, label='filtered')
ax.legend()

```



## 58.3 bandpass

```

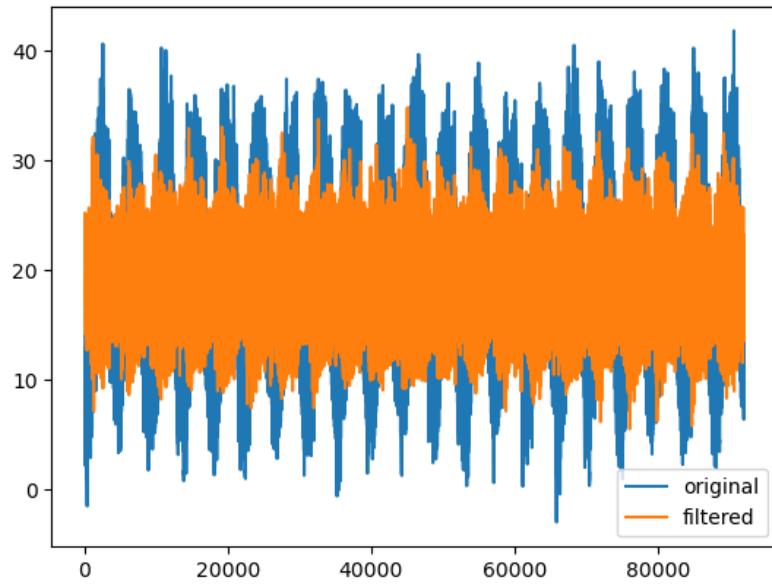
frequency_sample = 0.5 # in units of 1/hr. we choose 0.5 because we sample every 2hrs.
cut_off = 0.005
low = 0.005
high = 0.1

sos = butter(4,                      #filter order = how steep is the slope
             [low, high],        #cutoff value in units of the frequency sample
             btype='band',      #type of filter
             output='sos',      # "sos" stands for "Second-Order Sections."
             fs=0.5            #frequency sample = how many samples per 1 unit.
             )

# apply filter to the data:
y = sosfiltfilt(sos, x) + np.mean(x)

fig, ax = plt.subplots()
ax.plot(x, label='original')
ax.plot(y, label='filtered')
ax.legend()

```



## 58.4 bandstop

```

frequency_sample = 0.5 # in units of 1/hr. we choose 0.5 because we sample every 2hrs.
cut_off = 0.005
low = 0.005
high = 0.1

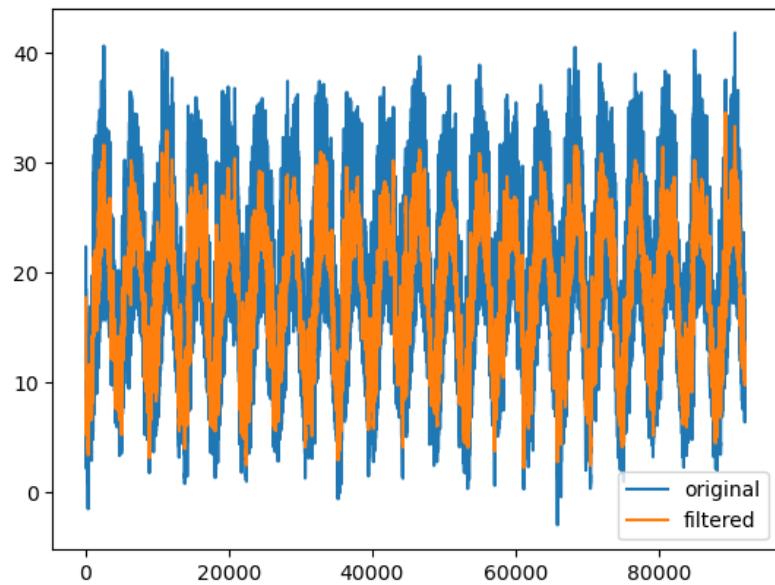
sos = butter(4,                      #filter order = how steep is the slope
            [low, high],        #cutoff value in units of the frequency sample
            btype='bandstop',   #type of filter
            output='sos',      # "sos" stands for "Second-Order Sections."
            fs=0.5            #frequency sample = how many samples per 1 unit.
            )

# apply filter to the data:
y = sosfiltfilt(sos, x)

fig, ax = plt.subplots()

```

```
ax.plot(x, label='original')
ax.plot(y, label='filtered')
ax.legend()
```



## 59 filtering 3

Now lets apply filtering to another dataset. Do you remember the phrase “I like to eat hummus”? When we applied dynamic time warping to that recording we messesed up with the samplerate to make the computation faster but the end result didn’t sound that clear. Now that we know FFT and filtering, lets try to make it sound clearer by removing unwanted frequencies.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.ensemble import RandomForestRegressor
import concurrent.futures
from datetime import datetime, timedelta
from sklearn.cluster import KMeans
import math
import scipy
from scipy import signal
from scipy.io import wavfile
from scipy.signal import find_peaks
from scipy.signal import sosfiltfilt, butter, ellip, cheby1, cheby2

# %matplotlib widget

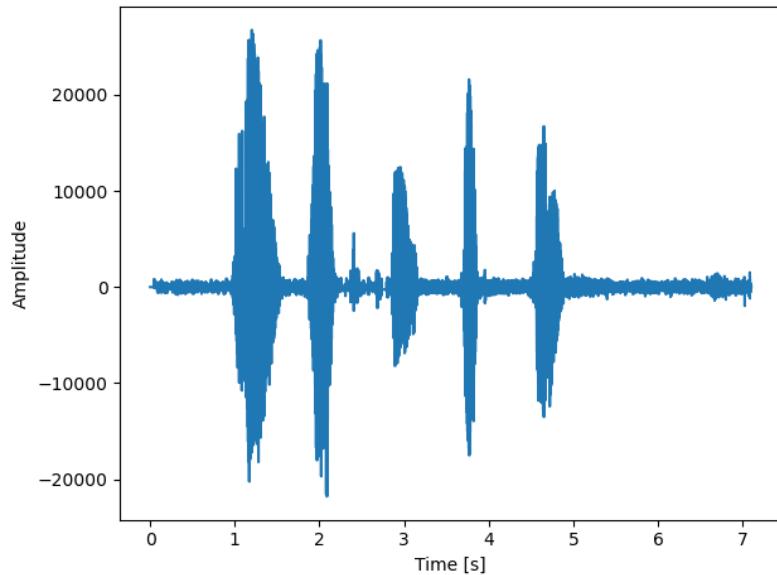
filename = 'hummus2_warped.wav'
samplerate, data1 = wavfile.read(filename)

length = data1.shape[0] / samplerate
time = np.linspace(0., length, data1.shape[0])

fig, ax = plt.subplots()
```

```
ax.plot(time, data1)
ax.set_xlabel("Time [s]")
ax.set_ylabel("Amplitude")

plt.tight_layout()
```



## 59.1 apply FFT

```
x = data1
dt = 1/samplerate
N = len(x)
t=time

fft = scipy.fft.fft(x) / N
k = scipy.fft.fftfreq(N, dt)
fft_abs = np.abs(fft)
```

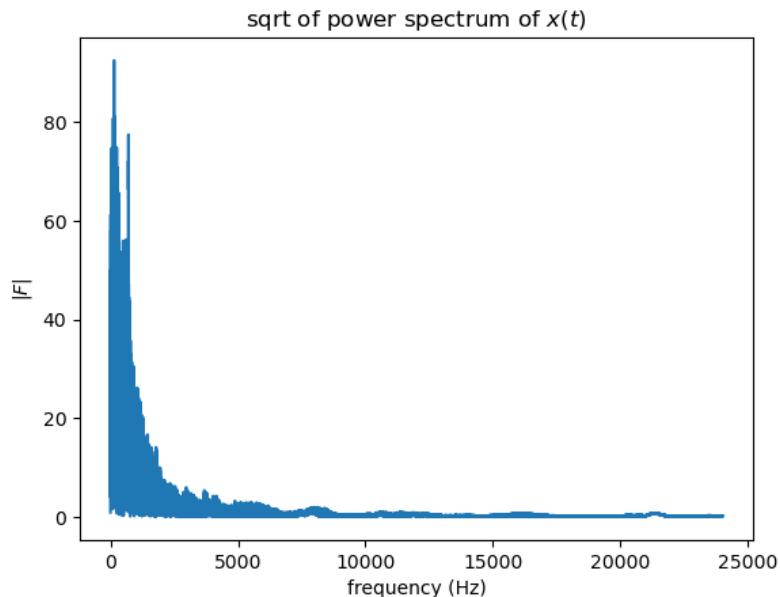
Keep only positive k values

```
fft_abs = fft_abs [k>=0]
k = k [k>=0]
```

```
fig, ax = plt.subplots()
ax.plot(k, fft_abs)
ax.set(xlabel="frequency (Hz)",
       ylabel=r"$|F|$",  

       title=r"sqrt of power spectrum of $x(t)$")
```

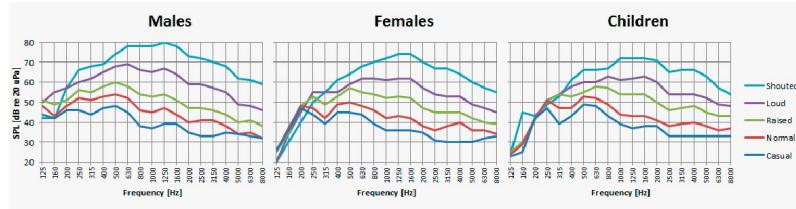
```
[Text(0.5, 0, 'frequency (Hz)'),
 Text(0, 0.5, '$|F|$'),
 Text(0.5, 1.0, 'sqrt of power spectrum of $x(t)$')]
```



## 59.2 apply bandpass filter

In the recording we can hear that there are some undesired low and high frequencies. In general we will choose the band

frequency to match the range of a male adult, which is approximately 300-5000Hz.



```

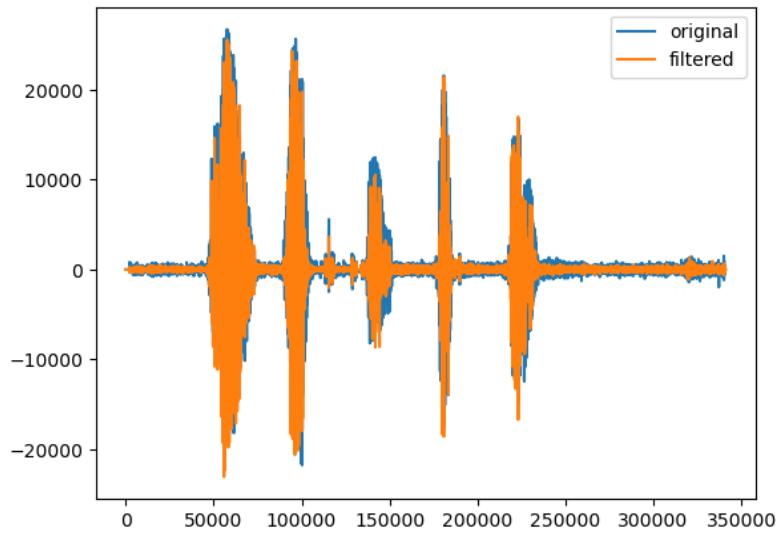
frequency_sample = samplerate
low = 300
high = 3000

sos = butter(4,                  #filter order = how steep is the slope
            [low, high],      # cutoff value in units of the frequency sample
            btype='band',    #type of filter
            output='sos',   # "sos" stands for "Second-Order Sections."
            fs=frequency_sample      #frequency sample = how many samples per 1 unit.
            )

# apply filter to the data:
y = sosfiltfilt(sos, x) + np.mean(x)

fig, ax = plt.subplots()
ax.plot(x, label='original')
ax.plot(y, label='filtered')
ax.legend()

```



```
from scipy.io.wavfile import write
write("filt_" + filename, samplerate, y.astype(np.int16))
```

Lets compare the two:

original

filtered

# **Part IX**

## **rates of change**

## 60 motivation

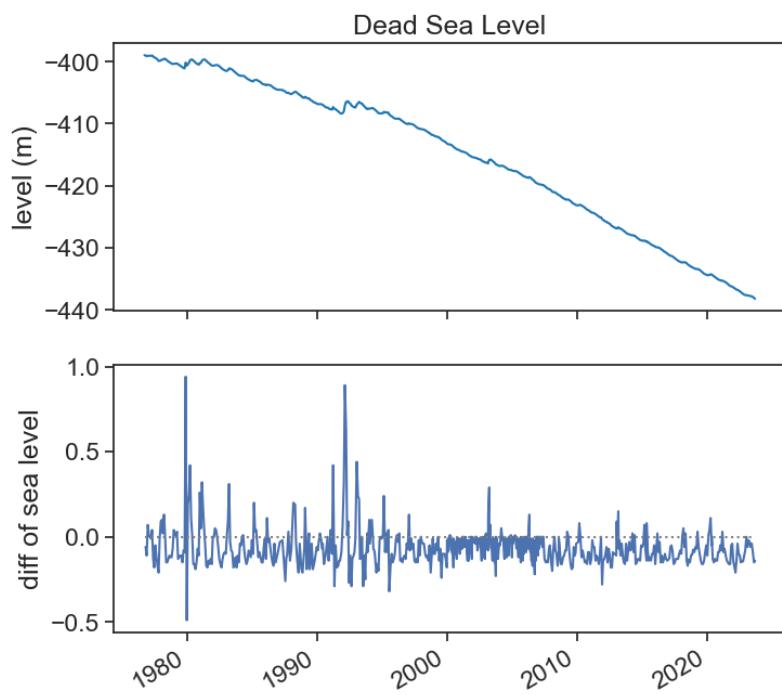
See below a graph of the Dead Sea level in the past decades?

- How fast is it going down on average?
- How fast does it change from month to month?

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
# %matplotlib widget

filename = "../archive/data/dead_sea_level.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date').sort_values(by='date')
# df

fig, ax = plt.subplots(2, 1, figsize=(8,8), sharex=True)
ax[0].plot(df['level'], color="tab:blue")
ax[0].set(title="Dead Sea Level",
           ylabel="level (m)")
ax[1].plot(df['level'].diff())
ax[1].plot(df['level']*0, ls=":", color="gray")
ax[1].set(ylabel="diff of sea level")
plt.gcf().autofmt_xdate() # makes slanted dates
```



We suspect that the operation `diff` has something to do with derivatives, or rates of change.

- What exactly is this connection?
- What are (or should be) the units in the bottom graph?
- Should we care if data is evenly spaced in time?

# 61 finite differences

Definition of a **derivative**:

$$\dot{f} = f'(t) = \underbrace{\frac{d}{dt} f(t)}_{\text{same thing}} = \frac{df(t)}{dt} = \lim_{\Delta t \rightarrow 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

Numerically, we can approximate the derivative  $f'(t)$  of a time series  $f(t)$  as

$$\frac{d}{dt} f(t) = \frac{f(t + \Delta t) - f(t)}{\Delta t} + \mathcal{O}(\Delta t). \quad (61.1)$$

The expression above is called the *two-point forward difference formula*.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
# %matplotlib widget
```

```
t = np.linspace(0,10,101)
def f(t):
    """
    function to plot
    """
    return t**3+t**2

def ftag(t):
```

The expression  $\mathcal{O}(\Delta t)$  means that the error associated with the approximation is proportional to  $\Delta t$ . This is called “**Big O notation**”.

```

"""
derivative
"""

return 3*t**2 +2*t

def line_chord(p1, p2, t):
    """
    given two points p1 and p2, return equation for the line that connects between them
    """
    p1x, p1y = p1
    p2x, p2y = p2
    slope = (p1y-p2y) / (p1x-p2x)
    intercept = (p1x*p2y - p1y*p2x) / (p1x-p2x)
    return slope*t + intercept

def line_tangent(p1, slope, t):
    """
    return equation of the line that passes through p1 with slope "slope"
    """
    p1x, p1y = p1
    intercept = p1y - slope*p1x
    return slope*t + intercept

x1 = 6.0
y1 = f(x1)
x2 = 8.0
y2 = f(x2)
chord = line_chord((x1,y1), (x2,y2), t)
tangent = line_tangent((x1,y1), ftag(x1), t)

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(t, f(t), lw=2, color="black")
ax.plot(t, tangent, color="tab:blue", lw=2, label="derivative")
ax.plot(t, chord, color="tab:orange", lw=2, label="approximation")
ax.legend(frameon=False)

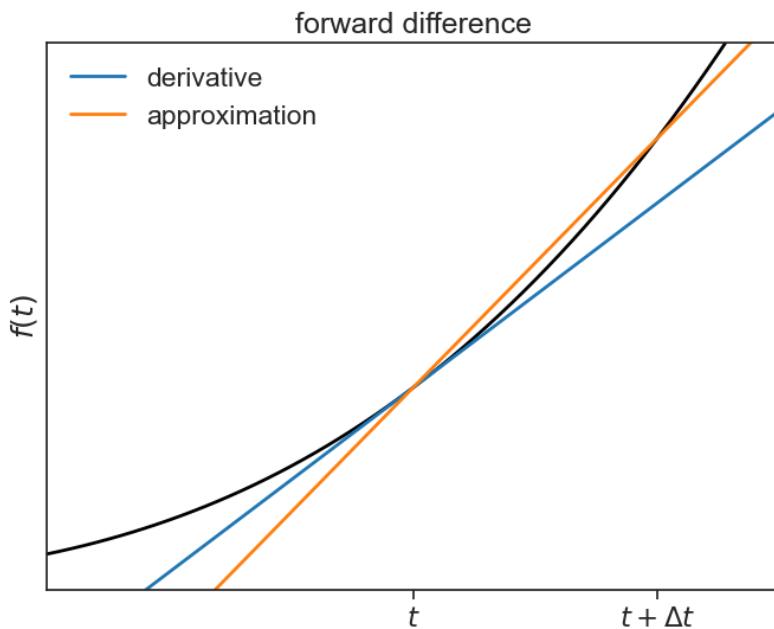
ax.set(xlim=[3, 9],
       ylim=[-10,700],
       ylabel=r"$f(t)$",
       xticks=[6,8],

```

```

yticks=[],
xticklabels=[r'$t$', r'$t+\Delta t$'],
title="forward difference");

```



Likewise, we can define the *two-point backward difference formula*:

$$\frac{df(t)}{dt} = \frac{f(t) - f(t - \Delta t)}{\Delta t} + \mathcal{O}(\Delta t). \quad (61.2)$$

```

x0 = 4.0
y0 = f(x0)
chord2 = line_chord((x1,y1), (x0,y0), t)

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(t, f(t), lw=2, color="black")
ax.plot(t, tangent, color="tab:blue", lw=2, label="derivative")
ax.plot(t, chord2, color="tab:orange", lw=2, label="approximation")
ax.legend(frameon=False)

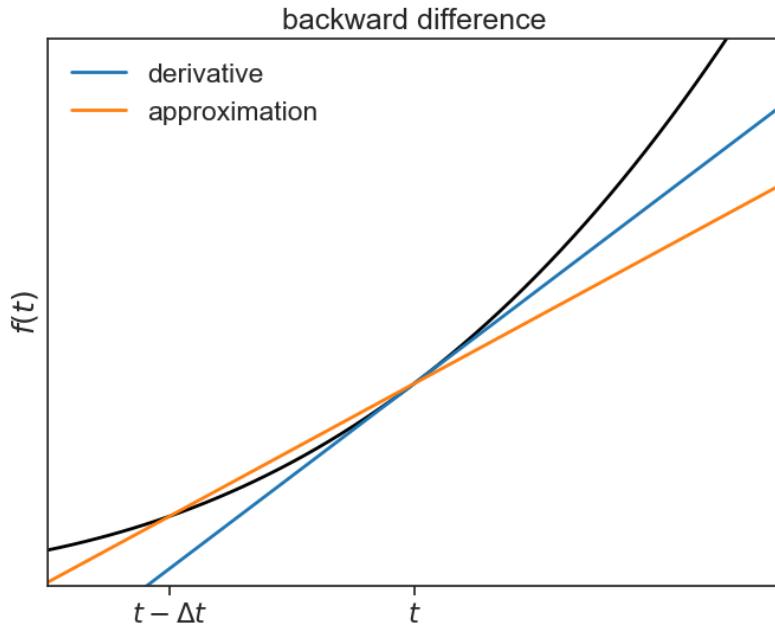
ax.set(xlim=[3, 9],
       ylim=[-10,700],

```

```

ylabel=r"$f(t)$",
xticks=[4,6],
yticks=[],
xticklabels=[r'$t-\Delta t$', r'$t$'],
title="backward difference");

```



If we sum together Equation 61.1 and Equation 61.2 we get:

$$2 \frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t)}{\Delta t} + \frac{f(t) - f(t - \Delta t)}{\Delta t} = \frac{f(t + \Delta t) - f(t - \Delta t)}{\Delta t}. \quad (61.3)$$

Dividing both sides by 2 gives the *two-point central difference formula*:

$$\frac{df(t)}{dt} = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2). \quad (61.4)$$

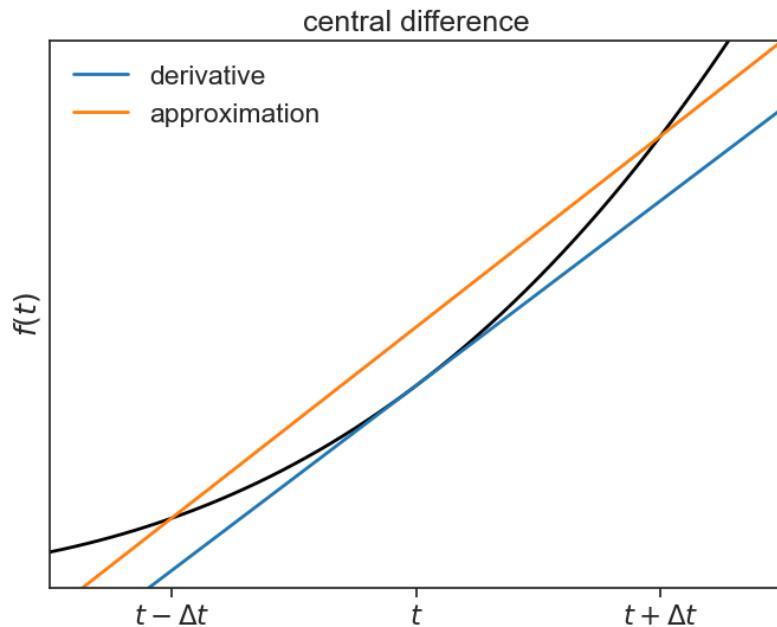
```

chord3 = line_chord((x2,y2), (x0,y0), t)

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(t, f(t), lw=2, color="black")
ax.plot(t, tangent, color="tab:blue", lw=2, label="derivative")
ax.plot(t, chord3, color="tab:orange", lw=2, label="approximation")
ax.legend(frameon=False)

ax.set(xlim=[3, 9],
       ylim=[-10,700],
       ylabel=r"$f(t)$",
       xticks=[4,6,8],
       yticks=[],
       xticklabels=[r'$t-\Delta t$', r'$t$', r'$t+\Delta t$'],
       title="central difference");

```



Let's compare these three methods. For which of them the slope of the orange line is closer to the actual derivative (blue)?

```

fig, ax = plt.subplots(1, 3, figsize=(8,3), sharex=True, sharey=True)

ax[0].plot(t, f(t), lw=2, color="black", label=r"$f(t)$")

```

```

ax[0].plot(t, tangent, color="tab:blue", lw=2, label=r"derivative at $t$")
ax[0].plot(t, chord, color="tab:orange", lw=2, label="approximation")
ax[0].legend(loc="upper left", bbox_to_anchor=(0.0,-0.2), frameon=False)

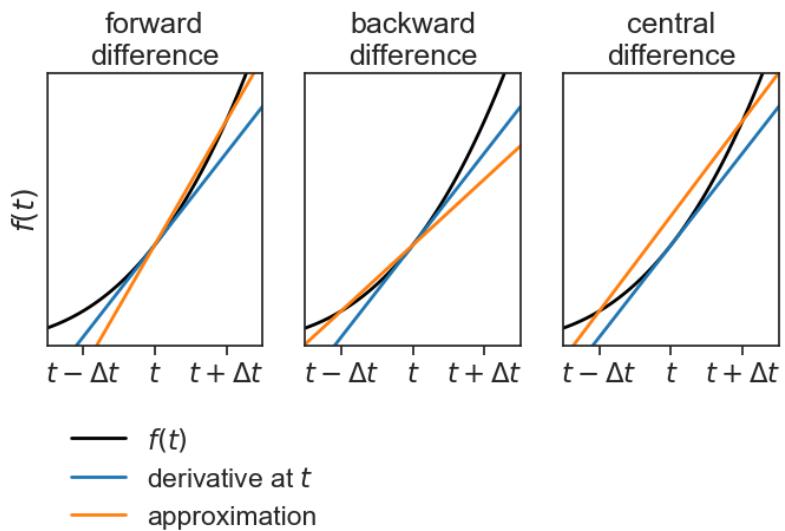
ax[1].plot(t, f(t), lw=2, color="black")
ax[1].plot(t, tangent, color="tab:blue", lw=2, label="derivative")
ax[1].plot(t, chord2, color="tab:orange", lw=2, label="approximation")

ax[2].plot(t, f(t), lw=2, color="black")
ax[2].plot(t, tangent, color="tab:blue", lw=2, label="derivative")
ax[2].plot(t, chord3, color="tab:orange", lw=2, label="approximation")

ax[0].set(title="forward\\ndifference",
           ylabel=r"$f(t)$");
ax[1].set(title="backward\\ndifference");

ax[2].set(xlim=[3, 9],
           ylim=[-10,700],
           xticks=[4,6,8],
           yticks=[],
           xticklabels=[r'$t-\Delta t$', r'$t$', r'$t+\Delta t$'],
           title="central\\ndifference");

```



Two things are worth mentioning about the approximation

above:

1. it is balanced, that is, there is no preference of the future over the past.
2. its error is proportional to  $\Delta t^2$ , it is a lot more precise than the unbalanced approximations :)

The function `np.gradient` calculates the derivative using the central difference for points in the interior of the array, and uses the forward (backward) difference for the derivative at the beginning (end) of the array.

Check out this [nice example](#).

## 61.1 dead sea level

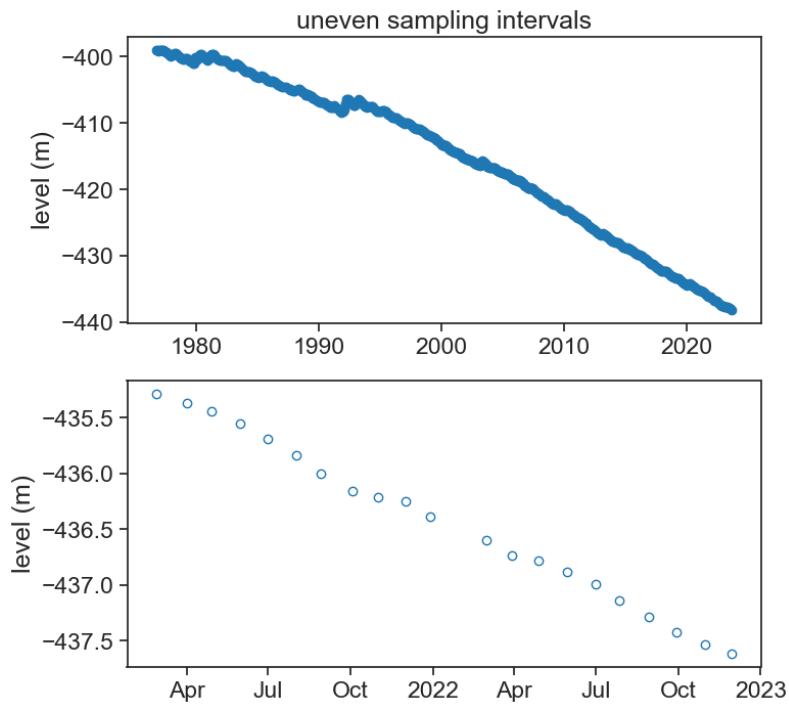
```
filename = "../archive/data/dead_sea_level.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'], dayfirst=True)
df = df.set_index('date').sort_values(by='date')

fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.2)
ax[0].plot(df['level'], '-o', color="tab:blue",)
ax[0].set(title="uneven sampling intervals",
           ylabel="level (m)")
locator = mdates.AutoDateLocator(minticks=5, maxticks=9)
formatter = mdates.ConciseDateFormatter(locator)
ax[0].xaxis.set_major_locator(locator)
ax[0].xaxis.set_major_formatter(formatter)

locator = mdates.AutoDateLocator(minticks=5, maxticks=9)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].plot(df.loc['2021-02-01':'2022-12-01', 'level'], 'o', color="tab:blue", mfc="None")
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)
ax[1].set(ylabel="level (m)");
```

To understand why the error is proportional to  $\Delta t^2$ , one can subtract the Taylor expansion of  $f(t - \Delta t)$  from the Taylor expansion of  $f(t + \Delta t)$ . See this, pages 3 and 4.

The “gradient” usually refers to a first derivative with respect to space, and it is denoted as  $\nabla f(x) = \frac{df(x)}{dx}$ . However, it doesn’t really matter if we call the independent variable  $x$  or  $t$ , the derivative operator is exactly the same.

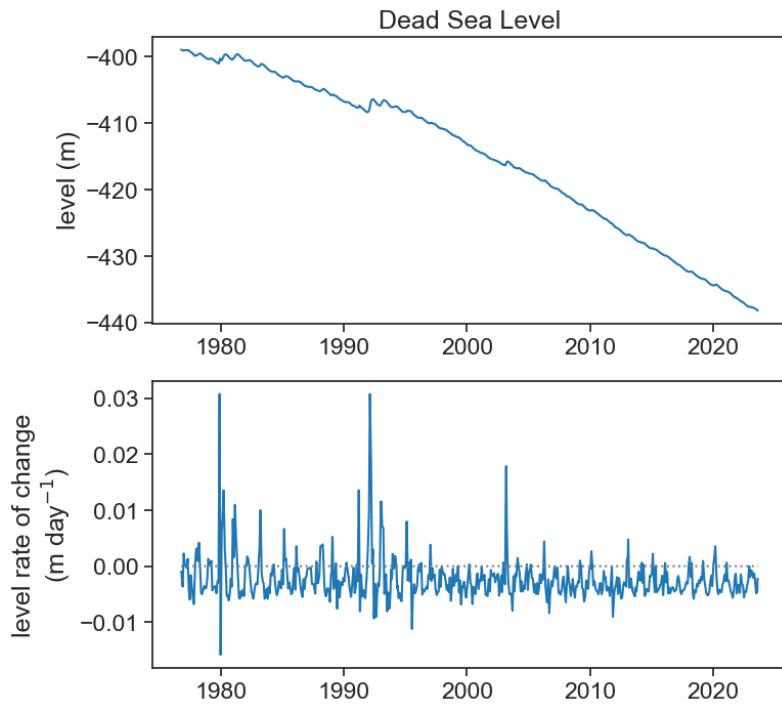


```

df2 = df['level'].resample('D').interpolate('time').to_frame()
df2['level_smooth'] = df2['level'].rolling('30D', center=True).mean()
dt = 1.0 # day
df2['grad'] = np.gradient(df2['level_smooth'].values, dt)

fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.2)
ax[0].plot(df2['level_smooth'], color="tab:blue",)
ax[0].set(title="Dead Sea Level",
          ylabel="level (m)")
ax[1].plot(df2['grad'], color="tab:blue")
ax[1].plot(df2['grad']*0, color="gray", ls=":")
ax[1].set(ylabel="level rate of change\n"+r"(m day$^{-1}$)")

```

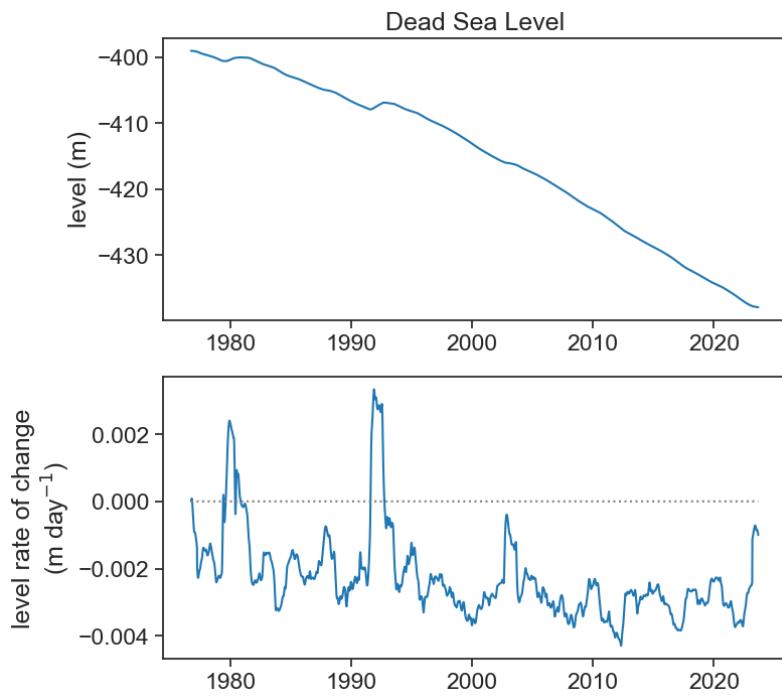


```
df2['level'].to_frame().to_csv()
```

If this is too noisy to your taste, and you are looking for rates of change for longer time scales (e.g., a year), then we can smooth the original signal. First let's smooth it by applying a running average of width 365 days.

```
df2['level_smooth_yr'] = df2['level'].rolling('365D', center=True).mean()
dt = 1.0 # day
df2['grad_yr'] = np.gradient(df2['level_smooth_yr'].values, dt)

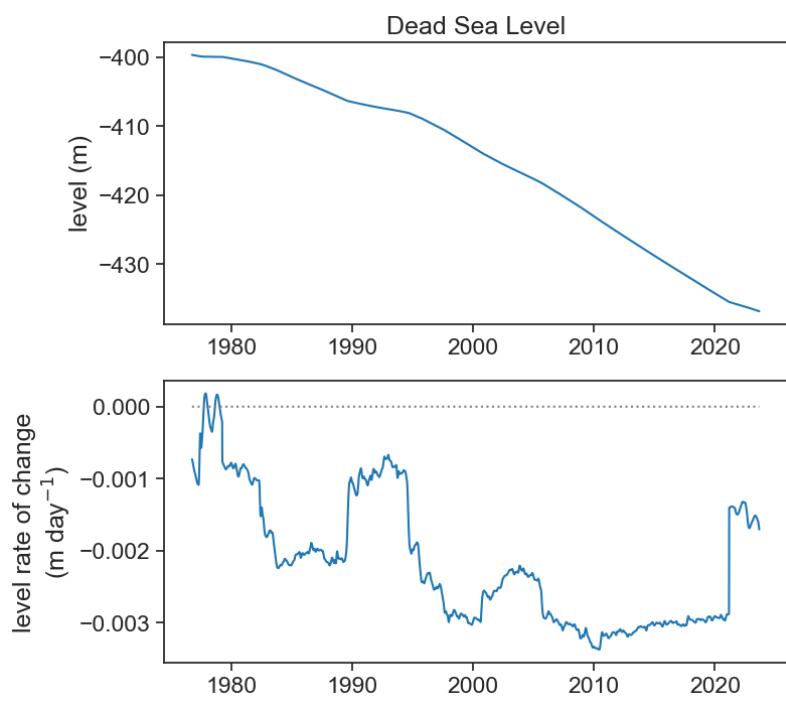
fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.2)
ax[0].plot(df2['level_smooth_yr'], color="tab:blue",)
ax[0].set(title="Dead Sea Level",
           ylabel="level (m)")
ax[1].plot(df2['grad_yr'], color="tab:blue")
ax[1].plot(df2['grad_yr']*0, color="gray", ls=":")
ax[1].set(ylabel="level rate of change\n"+r"(m day^{-1})");
```



Let's see now with a 5-year window.

```
df2['level_smooth_5yr'] = df2['level_smooth'].rolling('1825D', center=True).mean()
dt = 1.0 # day
df2['grad_5yr'] = np.gradient(df2['level_smooth_5yr'].values, dt)

fig, ax = plt.subplots(2, 1, figsize=(8,8))
fig.subplots_adjust(hspace=0.2)
ax[0].plot(df2['level_smooth_5yr'], color="tab:blue",
ax[0].set(title="Dead Sea Level",
           ylabel="level (m)")
ax[1].plot(df2['grad_5yr'], color="tab:blue")
ax[1].plot(df2['grad_5yr']*0, color="gray", ls=":")
ax[1].set(ylabel="level rate of change\n"+r"(m day$^{-1}$))
```



## 62 Fourier-based derivatives

This tutorial is based on Pelliccia (2019).

nice trick: <https://math.stackexchange.com/questions/430858/fourier-transform-of-derivative>

When we learned about Fourier transforms, we saw the following equation:

$$f(t) = \int_{-\infty}^{\infty} F(k)e^{2\pi i k t} dk.$$

What happens if we take the time derivative of the expression above?

$$\begin{aligned}\frac{d}{dt} f(t) &= \frac{d}{dt} \int_{-\infty}^{\infty} F(k)e^{2\pi i k t} dk \\ &= \int_{-\infty}^{\infty} F(k) \frac{d}{dt} e^{2\pi i k t} dk \\ &= \int_{-\infty}^{\infty} F(k) (2\pi i k) e^{2\pi i k t} dk\end{aligned}$$

We found something interesting! The derivative of  $f(t)$  can be calculated by taking the Inverse Fourier Transform of  $(2\pi i k)F(k)$ .

Let's see this in action!

## 62.1 dead sea level

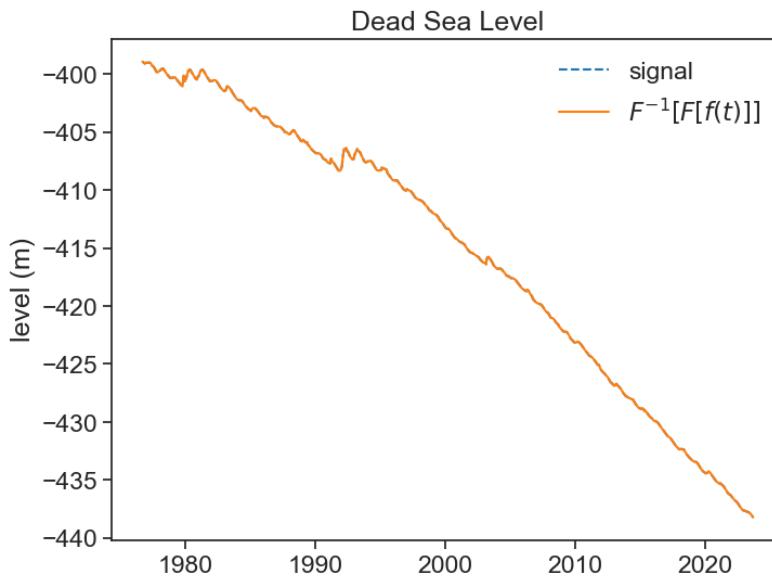
First let's get used to taking the Fourier transform and then reconstituting the signal by taking the inverse Fourier transform.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
# %matplotlib widget

filename = "dead_sea_1d.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')

fft = scipy.fft.fft(df['level'].values)
reconstituted = scipy.fft.ifft(fft).real

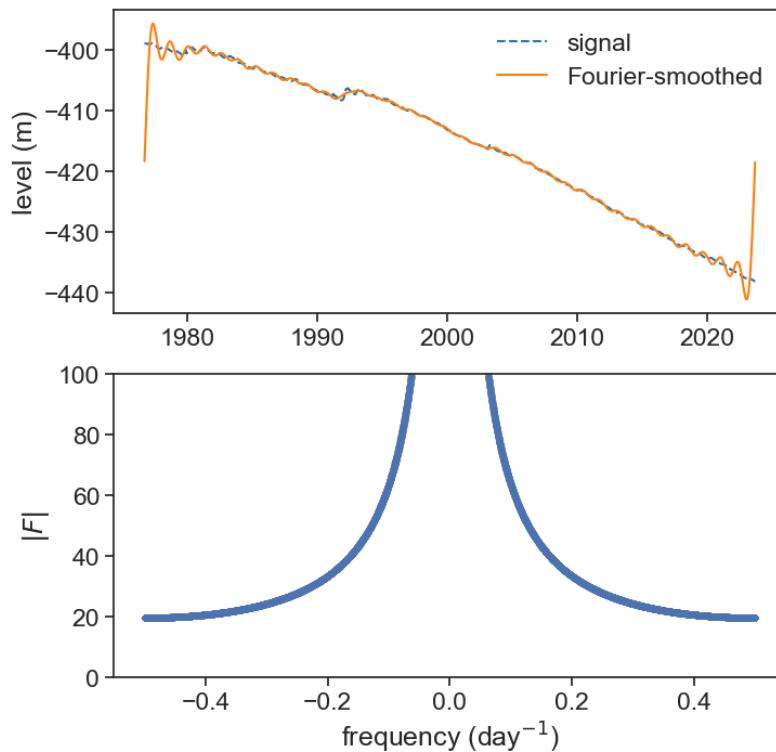
fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['level'], ls="--", color="tab:blue", label="signal")
ax.plot(df['level'].index, reconstituted, color="tab:orange", label=r"$F^{-1}[F[f(t)]]$")
ax.legend(frameon=False)
ax.set(title="Dead Sea Level",
      ylabel="level (m)");
```



Now let's apply some smoothing, since we don't want a choppy derivative.

```
fft = scipy.fft.fft(df['level'].values)
N = len(df['level'])
xi = scipy.fft.fftfreq(N)
cutoff_T = 3000.0 # days
cutoff_xi = 2.0 * np.pi / cutoff_T
mask = np.where(np.abs(xi) > cutoff_xi)
fft_filtered = fft.copy()
fft_filtered[mask] = 0.0
reconstituted = scipy.fft.ifft(fft_filtered).real

fig, ax = plt.subplots(2, 1, figsize=(8,8))
ax[0].plot(df['level'], ls="--", color="tab:blue", label="signal")
ax[0].plot(df['level'].index, reconstituted, color="tab:orange", label="Fourier-smoothed")
ax[0].legend(frameon=False)
ax[0].set(ylabel="level (m)")
ax[1].plot(xi, np.abs(fft), '.')
ax[1].set(ylim=[0,100],
          xlabel=r"frequency (day$^{-1}$)",
          ylabel=r"$|F|$");
```



There's a problem now! We eliminated high frequencies and now the smoothed signal is really bad at the edges.

The reason for this is that one assumption of the `fft` tool is that we have a **periodic signal**. Our signal is clearly not periodic, and when we treat it as it were periodic, we have a **very** broad power spectrum, that has really high values even for the highest frequencies. We need to solve this.

One trick to is to make our signal periodic by subtracting from it a straight line that joins the first and last points. See below.

```
def line_chord(p1, p2, t):
    """
    given two points p1 and p2, return equation for the line that connects between them
    """
    p1x, p1y = p1
    p2x, p2y = p2
    slope = (p1y-p2y) / (p1x-p2x)
```

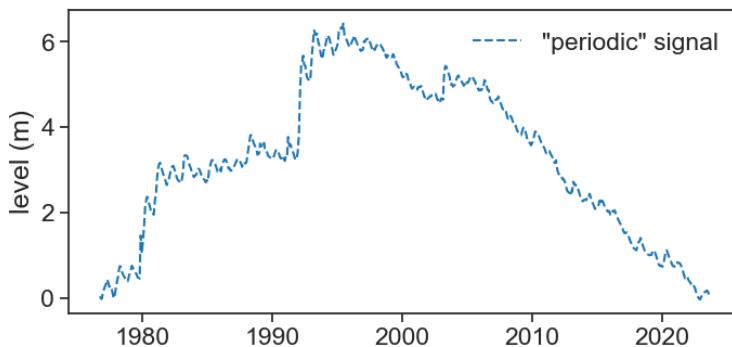
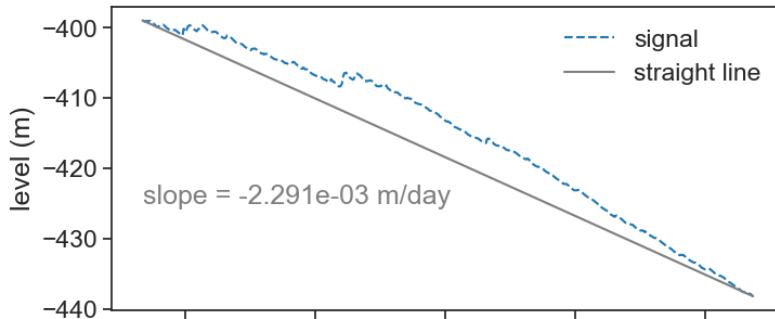
```

intercept = (p1x*p2y - p1y*p2x) / (p1x-p2x)
return slope*tt + intercept

r = np.arange(len(df))
p1 = (r[0], df['level'][0])
p2 = (r[-1], df['level'][-1])
line = line_chord(p1, p2, r)
slope_mperday = (p2[1] - p1[1]) / (p2[0] - p1[0])
df['periodic_level'] = df['level'] - line

fig, ax = plt.subplots(2, 1, figsize=(8,8), sharex=True)
ax[0].plot(df['level'], ls="--", color="tab:blue", label="signal")
ax[0].plot(df.index, line, color="gray", label="straight line")
ax[0].legend(frameon=False)
ax[0].set(ylabel="level (m)")
ax[0].text(df.index[0], -425, f"slope = {slope_mperday:.3e} m/day", color="gray")
ax[1].plot(df['periodic_level'], ls="--", color="tab:blue", label='''"periodic" signal')
ax[1].legend(frameon=False)
ax[1].set(ylabel="level (m)");

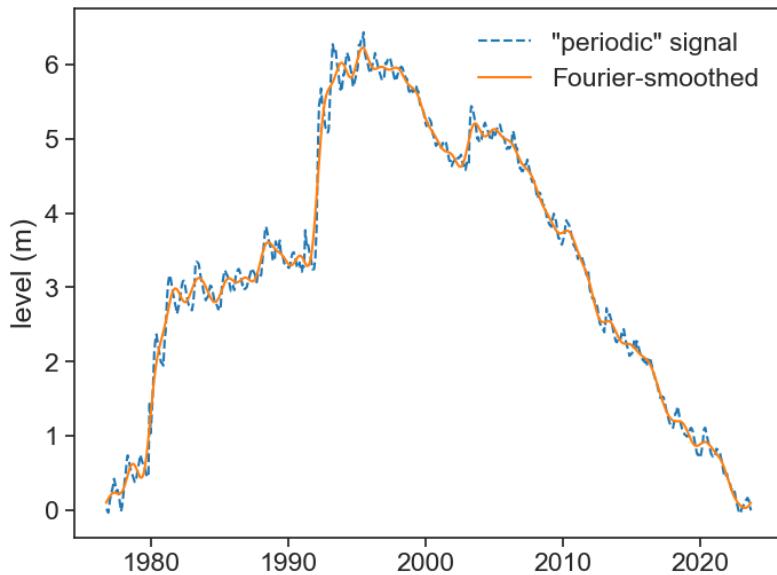
```



We can now check that after removing the highest frequencies from the signal, the smoothed reconstitution doesn't suffer from weird boundary artifacts.

```
fft = scipy.fft.fft(df['periodic_level'].values)
N = len(df)
xi = scipy.fft.fftfreq(N)
cutoff_T = 3000.0
cutoff_xi = 2.0 * np.pi / cutoff_T
mask = np.where(np.abs(xi) > cutoff_xi)
fft_filtered = fft.copy()
fft_filtered[mask] = 0.0
reconstituted = scipy.fft.ifft(fft_filtered).real

fig, ax = plt.subplots(1, 1, figsize=(8,6), sharex=True)
ax.plot(df['periodic_level'], ls="--", color="tab:blue", label='"periodic" signal')
ax.plot(df.index, reconstituted, color="tab:orange", label="Fourier-smoothed")
ax.legend(frameon=False)
ax.set(ylabel="level (m)");
```



That looks great! We can finally calculate the derivative of the smoothed signal by using the nice property we saw at the top of this page.

```

derivative = scipy.fft.ifft(2.0*np.pi*1j*xi*fft_filtered).real
derivative_correct = derivative + slope_mperday

```

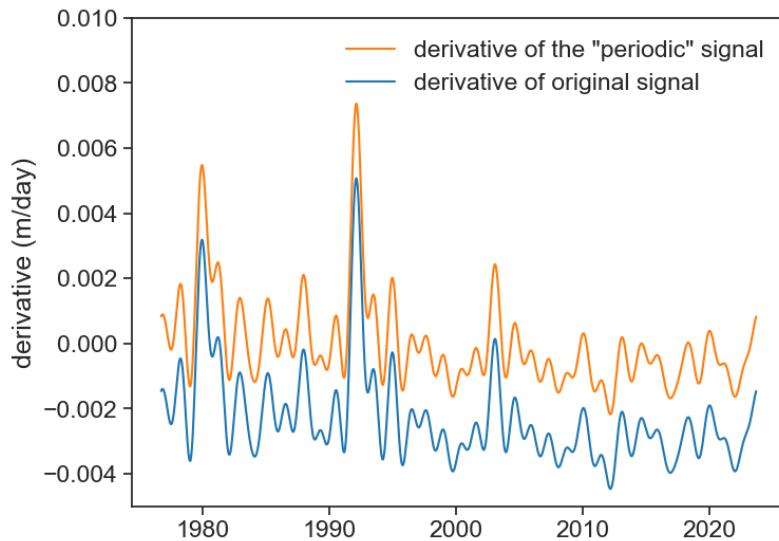
We have to correct for the fact that we subtracted a straight line from our original signal:

$$\begin{aligned}
\frac{d}{dt} \text{signal} &= \frac{d}{dt} [\text{"periodic" signal} + \text{line}] \\
&= \frac{d}{dt} [\text{"periodic" signal}] + \text{slope}
\end{aligned}$$

```

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['level'].index, derivative, color="tab:orange", label='derivative of the "periodic" signal')
ax.plot(df['level'].index, derivative_correct, color="tab:blue", label="derivative of original signal")
ax.legend(frameon=False)
ax.set(ylim=[-0.005, 0.010],
      ylabel="derivative (m/day)");

```



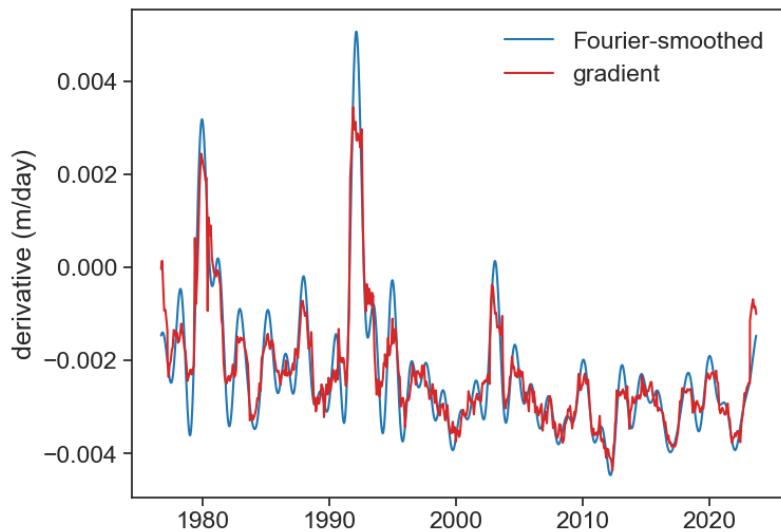
How does our solution compare to the previous method, the gradient?

```

df['level_smooth_yr'] = df['level'].rolling('365D', center=True).mean()
df['grad_yr'] = np.gradient(df['level_smooth_yr'].values)

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['level'].index, derivative_correct, color="tab:blue", label="Fourier-smoothed")
ax.plot(df['grad_yr'], color="tab:red", label="gradient")
ax.legend(frameon=False)
ax.set(ylabel="derivative (m/day)");

```



## 63 LOESS-based derivatives

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
# %matplotlib widget

filename = "dead_sea_1d.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')
```

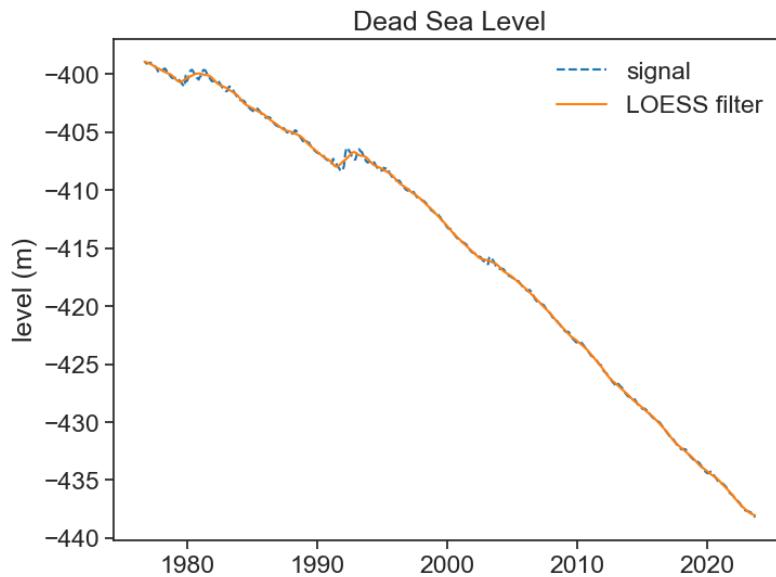
We can use `scipy.signal.savgol_filter` to apply a LOESS filter. By choosing the argument `deriv=1`, we tell it to return the first derivative of the smoothed signal.

```
df['smoothed'] = scipy.signal.savgol_filter(df['level'].values,
                                             window_length=24*30,
                                             polyorder=3)

df['savgol_deriv'] = scipy.signal.savgol_filter(df['level'].values,
                                                 window_length=24*30,
                                                 polyorder=3,
                                                 deriv=1)

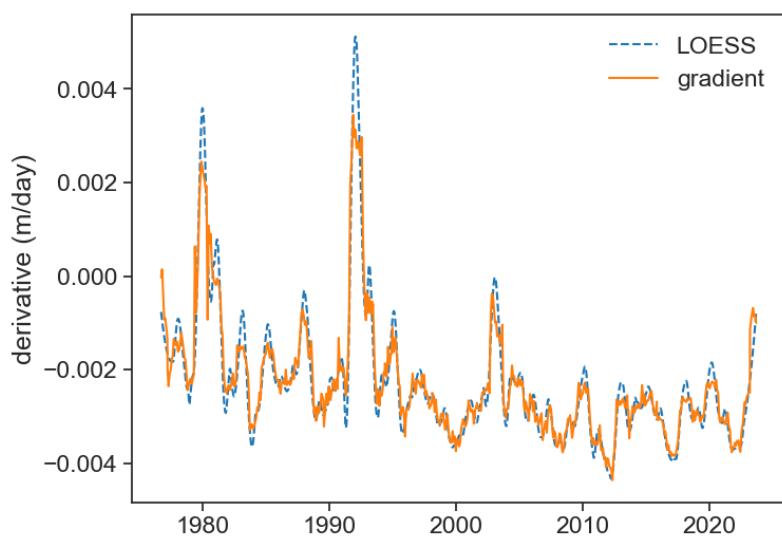
fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['level'], ls="--", color="tab:blue", label="signal")
ax.plot(df['smoothed'], color="tab:orange", label=r"LOESS filter")
```

```
ax.legend(frameon=False)
ax.set(title="Dead Sea Level",
       ylabel="level (m)");
```



```
df['level_smooth_yr'] = df['level'].rolling('365D', center=True).mean()
df['grad_yr'] = np.gradient(df['level_smooth_yr'].values)
```

```
fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['savgol_deriv'], ls="--", color="tab:blue", label="LOESS")
ax.plot(df['grad_yr'], color="tab:orange", label="gradient")
ax.legend(frameon=False)
ax.set(ylabel="derivative (m/day)");
```



## 64 does order matter?

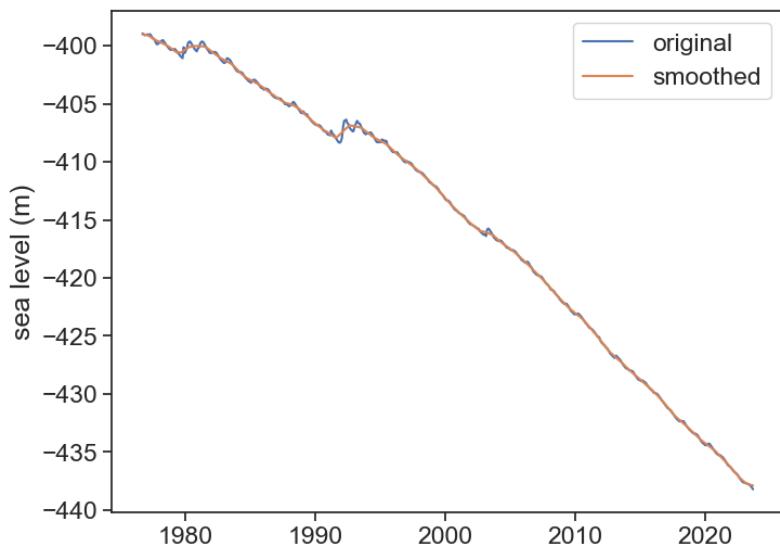
Let's say we want to take the derivative of a signal, but it is too rough, so some smoothing is required. Does the order of operations matter? Which should be done first? Smoothing or differentiation?

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
# %matplotlib widget

filename = "dead_sea_1d.csv"
df = pd.read_csv(filename)
df['date'] = pd.to_datetime(df['date'])
df = df.set_index('date')

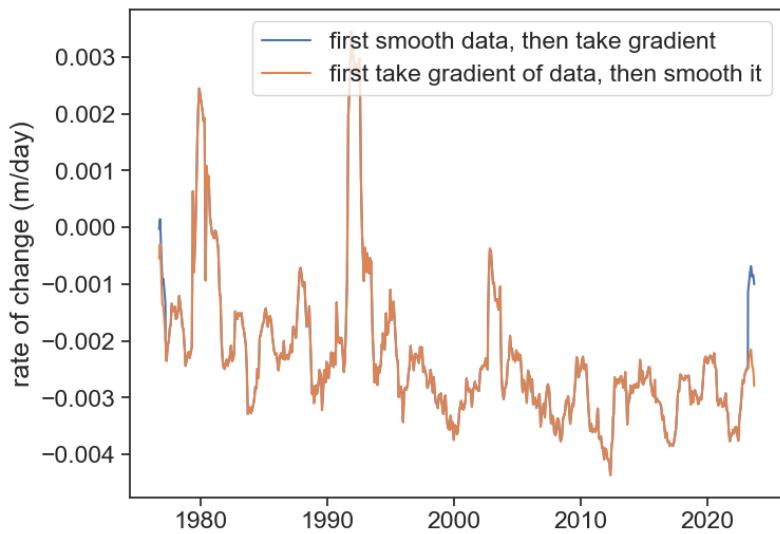
df['grad_of_level'] = np.gradient(df['level'], 1.0)
df['smooth_of_level'] = df['level'].rolling('365D', center=True).mean()
df['grad_of_smooth_of_level'] = np.gradient(df['smooth_of_level'], 1.0)
df['smooth_of_grad_of_level'] = df['grad_of_level'].rolling('365D', center=True).mean()

fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['level'], label="original")
ax.plot(df['smooth_of_level'], label="smoothed")
ax.legend()
ax.set(ylabel="sea level (m)");
```



Now let's see the derivatives.

```
fig, ax = plt.subplots(1, 1, figsize=(8,6))
ax.plot(df['grad_of_smooth_of_level'], label="first smooth data, then take gradient")
ax.plot(df['smooth_of_grad_of_level'], label="first take gradient of data, then smooth it")
ax.legend()
ax.set(ylabel="rate of change (m/day);")
```



# **Part X**

## **forecasting**

## **65 motivation**

## **66 ARIMA**

# **Part XI**

# **assignments**

# 67 Evaluation

All your assignments will be evaluated according to the following criteria:

- **40% Presentation:** How the graphs look, labels, general organization, markdown, and clean code.
- **30% Discussion:** Explain what you did, what you found out, etc.
- **15% Depth of analysis:** Analyze/explore the data with different levels of complexity.
- **10% Replicability:** Ensure your code runs flawlessly.
- **5% Code commenting:** Explain in your code what you are doing; this is good for everyone, especially for yourself.
- **5% Bonus:** For originality, creative problem-solving, or notable analysis.

## 67.1 Penalty for Late Submission

2 points will be deducted for each day after the deadline. Submissions will be accepted up to 7 days after the deadline.

## 67.2 Getting Help from Other Sources

- You can get help from AI tools, websites, books, etc.
- You can exchange ideas and tips with your colleagues, but should **not** do your projects together.
- At a minimum, you should use AI to check the quality of your English text and make the text pleasant to read.
- Ultimately, you, the student, are responsible for the assignment.
- Acknowledge external sources you used in your assignments, cite them, and explain how they helped you.

Above all, I would like you to look at the assignments as an opportunity to *learn*.

# 68 assignment 1

This assignment comes right after the first session, where we discussed resampling. Read the whole instructions.

## 68.1 task

Go to the [IMS website](#), and choose another weather station we have not worked with yet. Download 10-minute data for a full year, any year.

**Make the following graphs:**

1. Daily maximum humidity. Bonus: add another line to the graph, the daily minimum humidity.
2. The number of rainy days for each month.
3. Using global solar radiation data, calculate the number of hours in each day when the sun was up, and plot it. You can use a threshold of  $10 \text{ W/m}^2$  to determine this.

**Make 3 more graphs** of whatever you find interesting. You have the liberty to explore various facets of your dataset that capture your interest. It's essential, however, to maintain a focus on resampling. Each of your plots should effectively showcase and emphasize different aspects or techniques of resampling in your data analysis. To ensure diversity in your visualizations, avoid repetitive themes; for instance, if your first plot illustrates daily wind speed, then your second plot should not simply be a monthly resampling of wind speed. Aim for variety and innovation in each plot to fully explore the potential of resampling in data visualization. You can get ideas of what to plot from these [challenges](#).

You must download this Jupyter Notebook template. Create a zip file with your Jupyter notebook and with the `.csv` you used. Upload this zip file to the moodle task we created.

## 68.2 guidelines

1. Always name the axes and add units when relevant.
2. Always give a title to the plot.
3. Make sure that all axis tick labels (the numbers/dates on the axes) are readable.
4. Include a legend if you have multiple lines, colors, or groups.
5. Use appropriate scales for the axes (linear, logarithmic, etc.) depending on the data's nature.
6. Ensure that the plot is adequately sized for all elements to be clear and visible.
7. Choose colors and markers that are distinguishable, especially for plots with multiple elements.
8. If applicable, include error bars to indicate the variability or uncertainty in the data.
9. Use grid lines sparingly; they should not overshadow the data.

## 68.3 evaluation

Check the [evaluation](#) page to learn about the evaluation criteria.

# 69 assignment 2

## 69.1 Smoothing

In this assignment, you will delve into the application of different smoothing techniques on time series data. Utilizing meteorological data, your task is to create a series of plots that demonstrate the effects of various smoothing methods.

### 69.1.1 1. Comparative Smoothing Methods Analysis

- **Goal:** Showcase three smoothing techniques – Rolling Average, Savitzky-Golay, and Resampling – on the same time series data.
- **Task:** Overlay these methods over the actual data in a single plot. Ensure each method uses the same window size for consistency. **Describe in a few lines the differences you see.**

```
# code goes here
```

### 69.1.2 2. Rolling Average Window Size Impact

- **Goal:** Analyze the effect of varying window sizes on the Rolling Average method.
- **Task:** Produce a plot with three lines, each representing the Rolling Average with a different window size. **Describe in a few lines the differences you see.**

```
# code goes here
```

### 69.1.3 3. Savitzky-Golay Polynomial Order Variation

- **Goal:** Investigate how changing the polynomial order affects the Savitzky-Golay smoothing method.
- **Task:** Create a plot with three lines, where each represents the Savitzky-Golay method with a different polynomial order. **Describe in a few lines the differences you see.**

```
# code goes here
```

### 69.1.4 4. Kernel Shape Influence in Rolling Mean

- **Goal:** Explore the impact of different kernel shapes on the Rolling Mean.
- **Task:** Generate a plot displaying three lines, each using a different kernel shape in the Rolling Mean. We encourage to use unique kernel shapes that we did not showcase in class. See [this list](#) of kernels. **Describe in a few lines the differences you see.**

```
# code goes here
```

### 69.1.5 5. Moving Average with Confidence Interval

- **Goal:** Plot a Moving Average along with a 75% confidence interval.
- **Task:** Design a plot illustrating both the Moving Average and its 75% confidence interval.

```
# code goes here
```

# **70 assignment 3**

In this assignment, you will incorporate all that you have learned until now on real data! Often in homework, teachers (including us) tailor-make a dataset to practice the subjects studied in class. This example is not tailored-made; it's from real life, yet it showcases the importance and relevance of **all** the subjects studied till now.

## **70.1 background**

A tree in Boston is being continuously measured, using an IoT box created by me (Erez).



Sensors connected to this box measure the following

1. Air temperature (C)
2. Air relative humidity (%)
3. Change of tree circumference ( $\mu\text{m}$ ) measured by a dendrometer
4. Battery voltage (V)
5. Internal temperature ( $^{\circ}\text{C}$ ) (inside the box)

The box collects data from sensors every 30 minutes and uploads them to the cloud once a day. The cloud server puts a time stamp on every datapoint, but to make sure things are right: the timestamp at the time of data collection is uploaded as a variable (in units of UNIX at GMT).

You can see the box's dashboard [here](#).

Download the file `boston_raw.csv` [here](#).

Here is a break down of the columns:

- `created_at` - Time stamp added by the cloud
- `entry_id` - Index added by the cloud
- `field1` - Change of tree circumference ( $\mu\text{m}$ )
- `field2` - Air temperature ( $^{\circ}\text{C}$ )
- `field3` - Air relative humidity (%)
- `field4` - Battery voltage (V)
- `field5` - Internal temperature (c) (inside the box)
- `field6` - True timestamp (in units of UNIX at GMT)
- `latitude` - Empty
- `longitude` - Empty
- `elevation` - Empty
- `status` - Empty

## 70.2 analysis

Follow the instructions below to process the data.

**Make sure your text answers are in markdown cells and are numbered. Text answers in Python cells might be ignored.**

Upload to Moodle the completed operational (no errors) Jupyter Notebook file (`.ipynb`), along with the `boston_raw.csv` file.

1. Load `csv` to a dataframe.
2. Rename the columns to short and convenient names that make sense for you.
3. Convert the UNIX time stamp to a human-readable time stamp. Don't forget to convert the time zone from GMT to Boston.
4. Set the new converted timestamp as the index.
5. Sort the entire dataframe based on the chronological order of the index. Here is an example code:  
`df.sort_index(inplace=True)`
6. Plot all columns and explore the data. Zoom in. Do you see gaps? Outliers? Jumps? **Don't process them yet.** Make a list of 3 things you see.  
**Bonus:** Identify any patterns related to outliers/gaps and explain their occurrence.
7. Count how many `nan` values are in each column.
8. Now take a look at the datetime index. As noted earlier, the readings on the device are every 30 minutes, but is that what we see in the data? Do we have a continuous datetime index where every index is 30 min apart? Are there gaps? Are the timestamps consistent? Is there a drift (e.g. sometimes the data comes in at 00:30 and sometimes 00:27 or 00:20, etc..). Explain.
9. Think about a quick and dirty fix to this problem. Then read below:
  - a. An easy fix will be to resample the data at 30T and take the mean. That will ensure consistent timestamps and no gaps (in the index).
  - b. Why quick and “**dirty**”? What is dirty about it? is there a problem? Explain.
  - c. Anyway, we will continue with this. There are other ways but they are a bit more complex.

10. Re-count the number of `nan` values in each column. Explain any changes observed.
11. Now that the index is consistent and without gaps, we want to all data (fix jumps, remove outliers and fill gaps).
  - a. I'll start with a hint regarding the dendrometer that requires some background knowledge you may not know. The dendrometer has a metal band wrapping the trunk of the tree and it needs to be reset when the sensor readings are approaching the limit of  $60,000 \mu\text{m}$ . So you can see it in the data that someone reset the band some day in September. What needs to be done is to **shift up** all the data after the jump to be a continuation of the data before the jump. Plot the dendrometer data after the shift.
12. Outliers: show results of at least 2 methods of outlier identification applied to the data. Compare them and choose the one that did the best work **and explain**. Outliers should be replaced with `nan` values.
13. Filling gaps (missing values): show results of at least 3 methods of gap filling (at least one of them should be advanced for example SARIMAX or Randomforest). Compare them and choose the one that did the best work **and explain**. Differently from what we learned during class, this time we don't have the real data to compare to. **Bonus:** you might find a way to download data from Boston from a meteorological station, and compare to our data.
14. At this point the data should be clean (no outliers or gaps) in all columns and ready for exploration.
15. Add a column of vapor pressure deficit (VPD) based on the air temperature and relative humidity. The formula can be found [here](#).
16. Using the toolkit learned in the course (for example: smoothing, seasonal decompose, detrending, etc..) show the following:

- a. How temperature affects dendrometer readings (hint, this happens in a large time scale of months). Showing it graphically is enough! No need to prove it statistically (for example don't apply correlation).
- b. How VPD affects tree dendrometer readings (hint, this happens in a small time scale of days, during the hotter months). Showing it graphically is enough! No need to prove it statistically (for example don't apply correlation).

# **71 final project**

In this final project, you have the opportunity to explore and analyze a dataset of your choosing. This project is designed to apply and showcase the skills you've acquired in this class.

## **71.1 dataset selection**

Although this class is for “Environment” students and we like environmental data, we are open to all types of time series, be it stock market, audio, economy, demography, etc. The most important thing should be that you find it interesting. So, you are free to choose whatever time series dataset you want as long as it meets the following criteria:

1. Real Data - The dataset must be composed of real, authentic data.
2. Appropriate Size: It should be sufficiently large to support meaningful analysis, with a recommended **minimum** being equal to a 3-year meteorological dataset featuring 4 variables (e.g., temperature, humidity) sampled bi-hourly.
3. Uniqueness: To ensure a diverse range of projects, your dataset should not duplicate another student’s choice exactly. Variations of similar data types (e.g., weather data from different locations or periods) are acceptable.

### **71.1.1 data approval**

The whole project is based on the dataset you choose, and we want to make sure you chose a fitting dataset. So you are required to upload to the moodle a half-page description of your dataset and the basic analyses you will accomplish. It

should include description of the subject, the size (number of rows and columns), the source, and the basic tools you will be using. Should be in .pdf format. If the dataset is not too large you can upload it as well.

**The deadline for this is March 26th.** Late submissions will be penalized with minus 10 points.

## 71.2 Technical Requirements

Your analysis must include techniques from each of the following pools.

**ATTENTION!** You don't need to do each analysis in the order they are listed below. Each analysis should be included in the report according to your best judgement. Where does it fit best? Don't deliver a supermarket list of analyses, try to integrate the different tools inside a coherent story you are telling us.

### 71.2.1 Pool 1: implement at least 3

Resampling

- down sampling
- up sampling

Smoothing

- moving average with a well-chosen kernel
- LOESS (Savitzky-Golay)
- FFT

### 71.2.2 Pool 2: implement at least 3

Outlier identification

- visual
- sliding windows (Z-score, IQR, MAD, etc)

Gap filling

- interpolation
- advanced (SARIMAX, Randomforest, etc.)

### **71.2.3 Pool 3: implement at least 5**

Seasonal decomposition and frequencies

- classic (additive or multiplicative)
- decompose by FFT filtering
- frequency analysis: FFT

Lags

- cross correlation
- DTW

Derivative

- gradient
- using FFT
- using LOESS

### **71.2.4 Pool 4: implement all**

Study your signal.

Don't do that for the whole dataset. Choose a particular signal and characterize it.

- Stationarity
- ACF
- PACF
- ARIMA
  - define order
  - generate synthetic time series

## 71.3 Project Objectives

The primary goal is to derive insights from your dataset using the analytical tools covered in the course.

The purpose of computing is insight, not numbers.  
— Richard Hamming

Make sure you follow the spirit of the quote above. Tell us what is interesting about your dataset, and what questions you wish to answer with the tools learned in this course.

Computers are useless. They can only give you answers.  
— Pablo Picasso

We don't want a mindless string of numbers and graphs. We want a story.

## 71.4 Report Structure

Your final report should include:

- **Introduction:** Describe the chosen dataset, including its source and significance.
- **Exploratory Analysis:** Present an initial exploration of the data.
- **Detailed Analysis:** Divide this section into subsections, each addressing a specific question. For each question, describe the analysis performed, present your findings, and include relevant figures.
- **Conclusions and Pitfalls:** Summarize your findings and describe the difficulties you encountered when analyzing your data, and how you solved them.

### 71.4.1 What to submit

1. A written report in .pdf format.

2. An `.ipynb` file with your complete code. Should be full of comments and executable with no errors.

**NOTE!** This is not a coding class and we will not grade your code, anyway we do want to see it and will appreciate if it is well written and organized.

3. The dataset, preferably in `.csv` format.

#### **71.4.2 When to submit**

Reports are due on April 21st.

#### **71.4.3 How to submit**

Via the course's Moodle (under the 'Final Projects' section).

## **Part XII**

# **technical stuff**

# **72 technical stuff**

## **72.1 operating systems**

I recommend working with UNIX-based operating systems (MacOS or Linux). Everything is easier.

If you use Windows, consider [installing Linux on Windows with WSL](#).

## **72.2 software**

[Anaconda's Python distribution](#)

[VSCode](#)

## **72.3 python packages**

[Kats — a one-stop shop for time series analysis](#)

Developed by Meta

[statsmodels](#) statsmodels is a Python package that provides a complement to scipy for statistical computations including descriptive statistics and estimation and inference for statistical models.

[ydata-profiling](#)

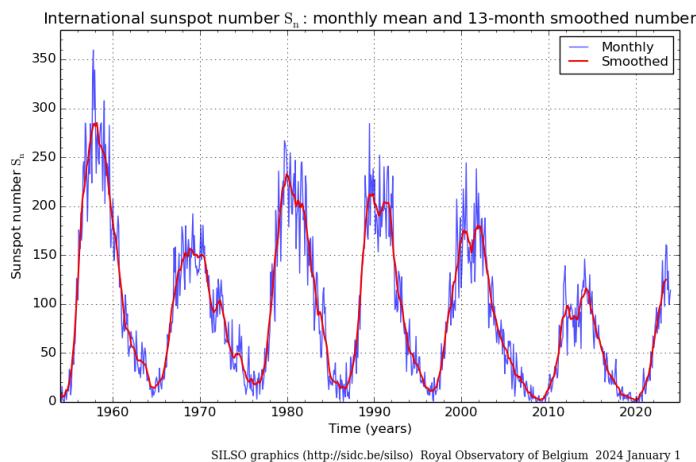
Quick Exploratory Data Analysis on time-series data. [Read also this.](#)

# 73 datasets

where to find data?

## 73.1 Sunspots

The solar cycle produces varying amounts of sunspots throughout the years.



SILSO graphics (<http://sidc.be/silso>) Royal Observatory of Belgium 2024 January 1

[Download data](#) from the Royal Observatory of Belgium.

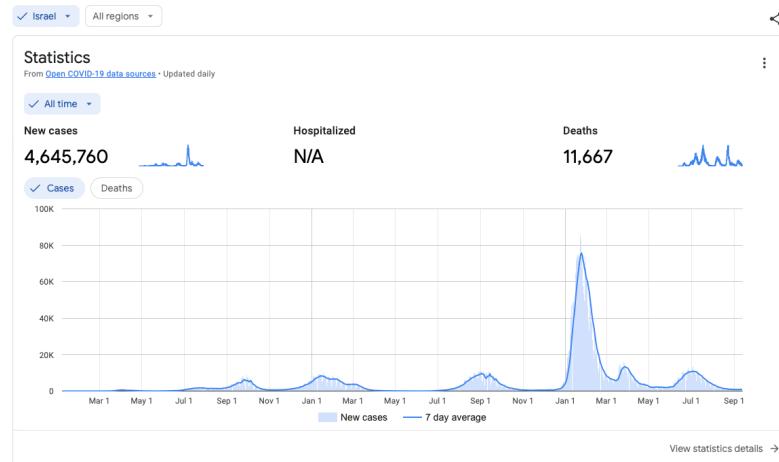
Source: [https://www.sidc.be/  
SILSO/monthlyssnplot](https://www.sidc.be/SILSO/monthlyssnplot)

## 73.2 Covid-19 Open Data

Download the data into your own tools and systems to analyze the virus's spread or decline, investigate COVID-related deaths,

study the effects of different vaccines, and more in 20,000-plus locations worldwide.

#### Data visualizer



[Click here](#) to go to the download page. Choose desired region under section “Understanding the data”.

Source:

<https://health.google.com/covid-19/open-data/explorer>

## 74 date formatting

Here you will find several examples of how to format dates in your plots. Not many explanations are provided.

How to use this page? Find first an example of a plot you like, only then go to the code and see how it's done.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import datetime
from datetime import timedelta
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
import matplotlib.gridspec as gridspec
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker

import pandas as pd

start_date = '2018-01-01'
end_date = '2018-04-30'

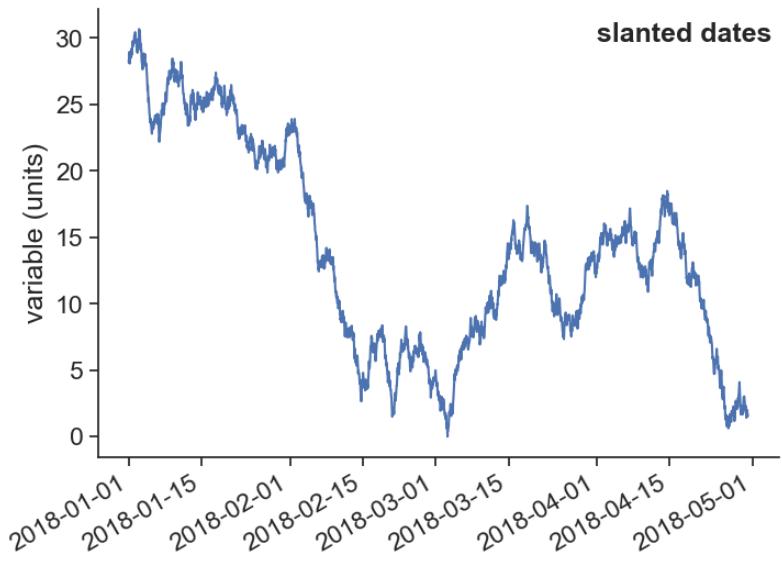
# create date range with 1-hour intervals
dates = pd.date_range(start_date, end_date, freq='1H')
# create a random variable to plot
var = np.random.rand(len(dates)) - 0.51
var = var.cumsum()
var = var - var.min()
# create dataframe, make "date" the index
df = pd.DataFrame({'date': dates, 'variable': var})
df.set_index(df['date'], inplace=True)
df
```

	date	variable
date		
2018-01-01 00:00:00	2018-01-01 00:00:00	28.317035
2018-01-01 01:00:00	2018-01-01 01:00:00	28.120523
2018-01-01 02:00:00	2018-01-01 02:00:00	28.596894
2018-01-01 03:00:00	2018-01-01 03:00:00	28.931941
2018-01-01 04:00:00	2018-01-01 04:00:00	28.561778
...	...	...
2018-04-29 20:00:00	2018-04-29 20:00:00	1.914343
2018-04-29 21:00:00	2018-04-29 21:00:00	1.648757
2018-04-29 22:00:00	2018-04-29 22:00:00	1.992956
2018-04-29 23:00:00	2018-04-29 23:00:00	1.500860
2018-04-30 00:00:00	2018-04-30 00:00:00	1.650439

define a useful function to plot the graphs below

```
def explanation(ax, text, letter):
    ax.text(0.99, 0.97, text,
            transform=ax.transAxes,
            horizontalalignment='right', verticalalignment='top',
            fontweight="bold")
    ax.text(0.01, 0.01, letter,
            transform=ax.transAxes,
            horizontalalignment='left', verticalalignment='bottom',
            fontweight="bold")
    ax.set(ylabel="variable (units)")
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
```

```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
ax.plot(df['variable'])
plt.gcf().autofmt_xdate() # makes slanted dates
explanation(ax, "slanted dates", "")
```



```

fig, ax = plt.subplots(4, 1, figsize=(10, 16),
                      gridspec_kw={'hspace': 0.3})

### plot a ####
ax[0].plot(df['variable'])
date_form = DateFormatter("%b")
ax[0].xaxis.set_major_locator(mdates.MonthLocator(interval=2))
ax[0].xaxis.set_major_formatter(date_form)

### plot b ####
ax[1].plot(df['variable'])
date_form = DateFormatter("%B")
ax[1].xaxis.set_major_locator(mdates.MonthLocator(interval=1))
ax[1].xaxis.set_major_formatter(date_form)

### plot c ####
ax[2].plot(df['variable'])
ax[2].xaxis.set_major_locator(mdates.MonthLocator())
# 16 is a slight approximation for the center, since months differ in number of days.
ax[2].xaxis.set_minor_locator(mdates.MonthLocator(bymonthday=16))
ax[2].xaxis.set_major_formatter(ticker.NullFormatter())
ax[2].xaxis.set_minor_formatter(DateFormatter('%B'))
for tick in ax[2].xaxis.get_minor_ticks():
    tick.tick1line.set_markersize(0)

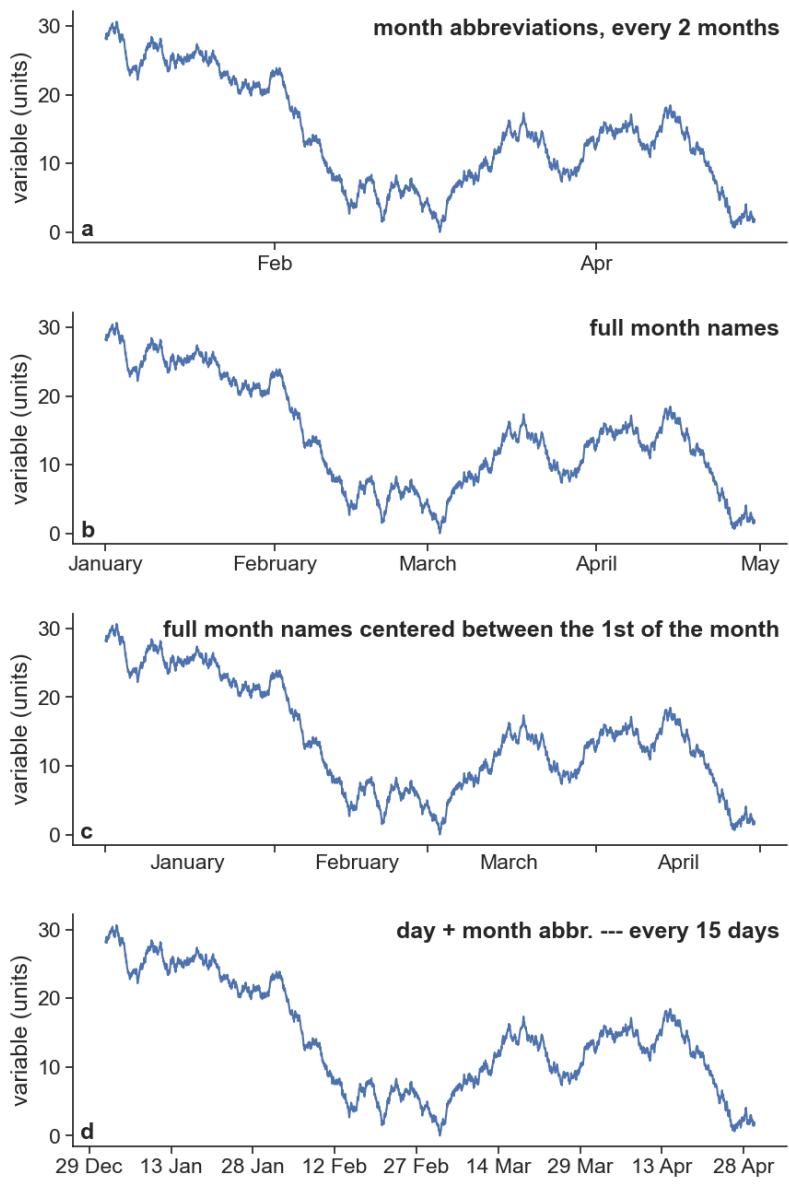
```

```
    tick.tick2line.set_markersize(0)
    tick.label1.set_horizontalalignment('center')

    ### plot d ####
ax[3].plot(df['variable'])
date_form = DateFormatter("%d %b")
ax[3].xaxis.set_major_locator(mdates.DayLocator(interval=15))
ax[3].xaxis.set_major_formatter(date_form)

explanation(ax[0], "month abbreviations, every 2 months", "a")
explanation(ax[1], "full month names", "b")
explanation(ax[2], "full month names centered between the 1st of the month", "c")
explanation(ax[3], "day + month abbr. --- every 15 days", "d")

fig.savefig("dates2.png")
```



```

fig, ax = plt.subplots(4, 1, figsize=(10, 16),
                      gridspec_kw={'hspace': 0.3})

### plot e ####
ax[0].plot(df['variable'])
date_form = DateFormatter("%d/%m")
ax[0].xaxis.set_major_locator(mdates.DayLocator(bymonthday=[5, 20]))

```

```

ax[0].xaxis.set_major_formatter(date_form)

### plot f ####
ax[1].plot(df['variable'])
locator = mdates.AutoDateLocator(minticks=11, maxticks=17)
formatter = mdates.ConciseDateFormatter(locator)
ax[1].xaxis.set_major_locator(locator)
ax[1].xaxis.set_major_formatter(formatter)

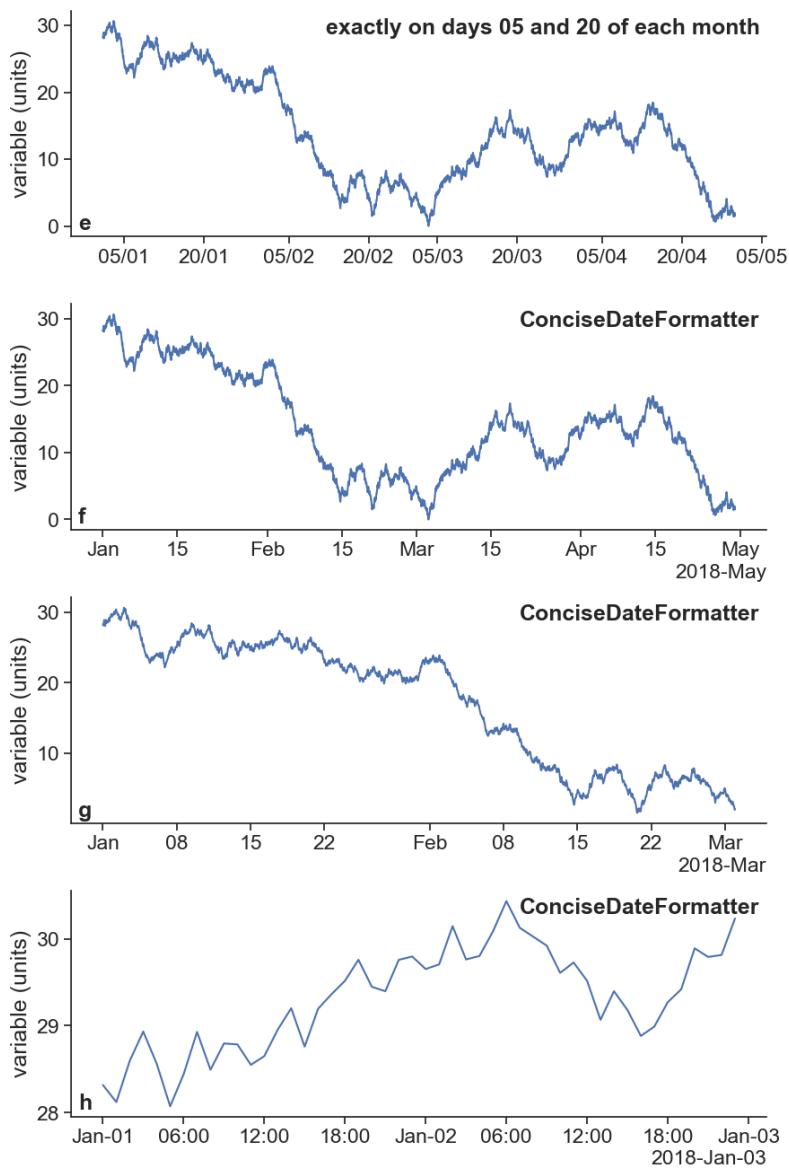
### plot g ####
ax[2].plot(df.loc['2018-01-01':'2018-03-01', 'variable'])
locator = mdates.AutoDateLocator(minticks=6, maxticks=14)
formatter = mdates.ConciseDateFormatter(locator)
ax[2].xaxis.set_major_locator(locator)
ax[2].xaxis.set_major_formatter(formatter)

### plot h ####
ax[3].plot(df.loc['2018-01-01':'2018-01-02', 'variable'])
locator = mdates.AutoDateLocator(minticks=6, maxticks=10)
formatter = mdates.ConciseDateFormatter(locator)
ax[3].xaxis.set_major_locator(locator)
ax[3].xaxis.set_major_formatter(formatter)

explanation(ax[0], "exactly on days 05 and 20 of each month", "e")
explanation(ax[1], "ConciseDateFormatter", "f")
explanation(ax[2], "ConciseDateFormatter", "g")
explanation(ax[3], "ConciseDateFormatter", "h")

fig.savefig("dates3.png")

```



```

fig, ax = plt.subplots(1, 1, figsize=(10, 4),
                      gridspec_kw={'hspace': 0.3})

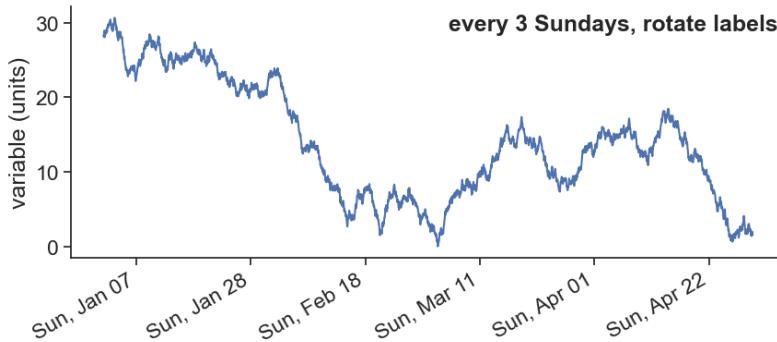
# import constants for the days of the week
from matplotlib.dates import MO, TU, WE, TH, FR, SA, SU
ax.plot(df['variable'])
# tick on sundays every third week

```

```

loc = mdates.WeekdayLocator(byweekday=SU, interval=3)
ax.xaxis.set_major_locator(loc)
date_form = DateFormatter("%a, %b %d")
ax.xaxis.set_major_formatter(date_form)
fig.autofmt_xdate(bottom=0.2, rotation=30, ha='right')
explanation(ax, "every 3 Sundays, rotate labels", "")

```



Code	Explanation
%Y	4-digit year (e.g., 2022)
%y	2-digit year (e.g., 22)
%m	2-digit month (e.g., 12)
%B	Full month name (e.g., December)
%b	Abbreviated month name (e.g., Dec)
%d	2-digit day of the month (e.g., 09)
%A	Full weekday name (e.g., Tuesday)
%a	Abbreviated weekday name (e.g., Tue)
%H	24-hour clock hour (e.g., 23)
%I	12-hour clock hour (e.g., 11)
%M	2-digit minute (e.g., 59)
%S	2-digit second (e.g., 59)
%p	"AM" or "PM"
%Z	Time zone name
%z	Time zone offset from UTC (e.g., -0500)

# 75 sources

## 75.1 books

[from Data to Viz](#)

[Fundamentals of Data Visualization](#), by Claus O. Wilke

[PyNotes in Agriscience](#)

[Forecasting: Principles and Practice \(3rd ed\)](#), by Rob J Hyndman and George Athanasopoulos

[Python for Finance Cookbook 2nd Edition - Code Repository](#)

[Practical time series analysis,: prediction with statistics and machine learning](#), by Aileen Nielsen

The online edition of this book is available for Hebrew University staff and students.

[Time series analysis with Python cookbook : practical recipes for exploratory data analysis, data preparation, forecasting, and model evaluation](#), by Tarek A. Atwan

The online edition of this book is available for Hebrew University staff and students.

[Hands-on Time Series Analysis with Python: From Basics to Bleeding Edge Techniques](#), by B V Vishwas, Ashish Patel

The online edition of this book is available for Hebrew University staff and students.

## 75.2 videos

[Times Series Analysis for Everyone](#), by Bruno Goncalves

This series is available for Hebrew University staff and students.

[Time Series Analysis with Pandas, by Joshua Malina](#) This video is available for Hebrew University staff and students.

### 75.3 references

- Brockwell, Peter J., and Richard A. Davis. 2016. *Introduction to Time Series and Forecasting*. 3rd ed. Springer.
- Chatfield, C. 2016. *The Analysis of Time Series: An Introduction, Sixth Edition*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press.
- Cleveland, Robert B, William S Cleveland, Jean E McRae, and Irma Terpenning. 1990. “STL: A Seasonal-Trend Decomposition.” *J. Off. Stat* 6 (1): 3–73.
- McDonald, Andy. 2022. “Creating Boxplots with the Seaborn Python Library.” *Medium*. Towards Data Science. <https://towardsdatascience.com/creating-boxplots-with-the-seaborn-python-library-f0c20f09bd57>.
- Pelliccia, Daniel. 2019. “Fourier Spectral Smoothing Method.” 2019. <https://nirpyresearch.com/fourier-spectral-smoothing-method/>.
- Shumway, Robert H., and David S. Stoffer. 2017. *Time Series Analysis and Its Applications With R Examples*. 4th ed. Springer. <http://www.stat.ucla.edu/~frederic/415/S23/tsa4.pdf>.
- Tsay, R. S. 2010. *Analysis of Financial Time Series*. Wiley.
- Westen, René M. van, Michael Kliphuis, and Henk A. Dijkstra. 2024. “Physics-Based Early Warning Signal Shows That AMOC Is on Tipping Course.” *Science Advances* 10 (6): eadk1189. <https://doi.org/10.1126/sciadv.adk1189>.
- Zhang, Ou. 2020. “Outliers-Part 3:outliers in Regression.” ouzhang.me. <https://ouzhang.me/blog/outlier-series/outliers-part3/>.

# **Part XIII**

## **behind-the-scenes**

# sliding window video

Import packages and stuff.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import scipy as sp
import json
import requests
import os
import subprocess
from tqdm import tqdm
from scipy import signal

# avoid "SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataF
pd.options.mode.chained_assignment = None # default='warn'
```

Download data from the [IMS](#) using an API.

```
# read token from file
with open('../archive/IMS-token.txt', 'r') as file:
    TOKEN = file.readline()
# 28 = SHANI station
STATION_NUM = 28
start = "2022/01/01"
end = "2022/01/07"
```

```

filename = 'shani_2022_january.json'

# check if the JSON file already exists
# if so, then load file
if os.path.exists(filename):
    with open(filename, 'r') as json_file:
        data = json.load(json_file)
else:
    # make the API request if the file doesn't exist
    url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/?from={start}&to={end}"
    headers = {'Authorization': f'ApiToken {TOKEN}'}
    response = requests.get(url, headers=headers)
    data = json.loads(response.text.encode('utf8'))

    # save the JSON data to a file
    with open(filename, 'w') as json_file:
        json.dump(data, json_file)
# show data to see if it's alright
# data

```

Load and process data.

```

df = pd.json_normalize(data['data'], record_path=['channels'], meta=['datetime'])
df['date'] = (pd.to_datetime(df['datetime'])
              .dt.tz_localize(None)  # ignores time zone information
              )
df = df.pivot(index='date', columns='name', values='value')
df

```

name date	Grad	RH	Rain	STDwd	TD	TDmax	TDmin	TG	TW	Time	WD	V
2022-01-01 00:00:00	0.0	77.0	0.0	10.3	11.2	11.2	11.1	10.7	-9999.0	2354.0	75.0	0
2022-01-01 00:10:00	0.0	77.0	0.0	11.2	11.2	11.2	11.1	10.8	-9999.0	1.0	77.0	8
2022-01-01 00:20:00	0.0	75.0	0.0	10.0	11.4	11.5	11.2	10.9	-9999.0	20.0	80.0	8
2022-01-01 00:30:00	0.0	74.0	0.0	9.6	11.5	11.5	11.4	11.0	-9999.0	22.0	76.0	7
2022-01-01 00:40:00	0.0	73.0	0.0	9.1	11.6	11.7	11.5	11.1	-9999.0	34.0	74.0	6
...	...	...	...	...	...	...	...	...	...	...	...	...
2022-01-06 23:10:00	0.0	36.0	0.0	16.1	11.6	12.0	11.1	6.8	-9999.0	2310.0	144.0	1
2022-01-06 23:20:00	0.0	35.0	0.0	10.1	12.1	12.3	11.9	6.3	-9999.0	2320.0	118.0	1

name	Grad	RH	Rain	STDwd	TD	TDmax	TDmin	TG	TW	Time	WD	Y
date												
2022-01-06 23:30:00	0.0	36.0	0.0	7.1	12.4	12.6	11.9	7.3	-9999.0	2330.0	113.0	
2022-01-06 23:40:00	0.0	37.0	0.0	5.6	12.6	12.7	12.5	7.8	-9999.0	2339.0	119.0	
2022-01-06 23:50:00	0.0	39.0	0.0	11.5	11.9	12.6	11.5	7.1	-9999.0	2341.0	102.0	

Define useful functions.

```
def concise(ax):
    """
    Let python choose the best xtick labels for you
    """
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def center_dates(ax):
    # show day of the month + month abbreviation. see full option list here:
    # https://strftime.org
    date_form = DateFormatter("%d %b")
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')
```

```

def center_dates_two_panels(ax0, ax1):
    # show day of the month + month abbreviation. see full option list here:
    date_form = DateFormatter("%d %b")
    # major ticks at midnight, every day
    ax0.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax1.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax1.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax1.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax1.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax1.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax0.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
    for tick in ax1.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')

```

We don't need the full month, let's cut the dataframe to fewer days.

```

start = "2022-01-01 00:00:00"
end = "2022-01-06 23:50:00"
df = df.loc[start:end]

```

We now redefine a narrower window, this will be the graph's xlims. We leave the dataframe as is, because we will need some data outside the graph's limits.

```

start = "2022-01-02 00:00:00"
end = "2022-01-05 23:50:00"

```

```

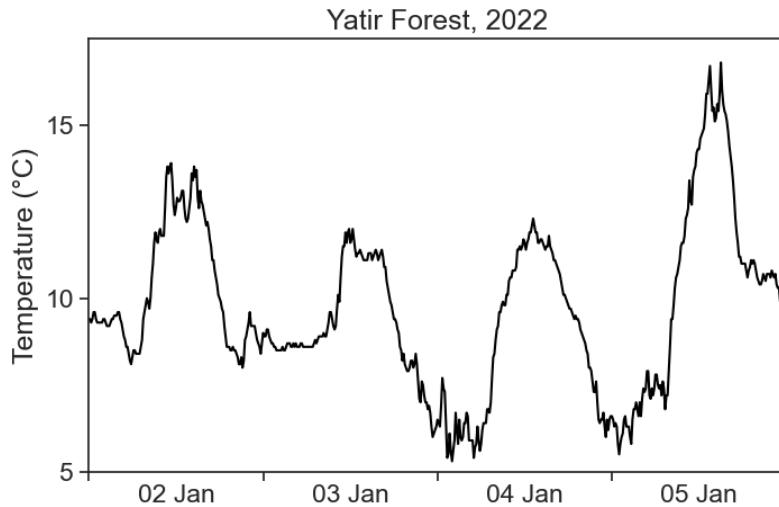
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df.loc[start:end, 'TD'], color='black')
ax.set(ylim=[5, 17.5],

```

```

        xlim=[start, end],
        ylabel="Temperature (°C)",
        title="Yatir Forest, 2022",
        yticks=[5,10,15])
center_dates(ax)
fig.savefig("sliding_YF_temperature_2022.png")

```



Looks good. Let's move on.

## 75.4 Rectangular kernel

```

%matplotlib widget
fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True,
                      gridspec_kw={'height_ratios':[1,0.4], 'hspace':0.1})

class Lines:
    """
    empty class, later will be populated with graph objects.
    this is useful to draw and erase lines on demand.
    """
    pass
lines = Lines()

```

```

# rename axes for convenience
ax0 = ax[0]
ax1 = ax[1]
# sm = df['TD'].rolling(10, center=True).mean()
# ga = df['TD'].rolling(10, center=True, win_type="gaussian").mean(std=100.0)

# set graph y limits
ylim = [3, 22]
# choose here window width in minutes
window_width_min = 200.0
window_width_min_integer = int(window_width_min) # same but integer
window_width_int = int(window_width_min // 10 + 1) # window width in points
N = len(df) # df length
# time range over which the kernel will slide
# starts at "start", minus the width of the window,
# minus half an hour, so that the window doesn't start sliding right away at the beginning of +
# ends an hour after the window has finished sliding
t_swipe = pd.date_range(start=pd.to_datetime(start) - pd.Timedelta(minutes=window_width_min) -
                        end=pd.to_datetime(end) + pd.Timedelta(minutes=60),
                        freq="10min")
# starting time
t0 = t_swipe[0]
# show sliding window on the top panel as a light blue shade
lines.fill_bet = ax0.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                  y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1)
# this is our "boxcart" kernel (a rectangle)
kernel_rect = np.ones(window_width_int)
# calculate the moving average with "kernel_rect" as weights
# this is the same as a convolution, which is just faster to compute
df.loc[:, 'con'] = np.convolve(df['TD'].values, kernel_rect, mode='same') / len(kernel_rect)
# create a new column for the kernel, fill it with zeros
df['kernel_plus'] = 0.0
# populate the kernel column with the window at the very beginning
df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = kernel_rect
# plot kernel on the bottom panel
lines.kernel_line, = ax1.plot(df['kernel_plus'])
# plot temperature on the top panel
ax0.plot(df.loc[start:end, 'TD'], color="black")
# make temperature look gray when inside the sliding window
lines.gray_line, = ax0.plot(df.loc[df['kernel_plus']==1.0, 'TD'],

```

```

        color=[0.6]*3, lw=3)
# calculate the middle of the sliding window
window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
# plot a pink line showing the result of the moving average
# from the beginning to the middle of the sliding window
lines.pink_line, = ax0.plot(df.loc[start:window_middle, 'con'], color="xkcd:hot pink", lw=3)
# emphasize the location of the middle on the window with a circle
lines.pink_circle, = ax0.plot([window_middle], [df.loc[window_middle, 'con']],
                             marker='o', markerfacecolor="None", markeredgecolor="xkcd:dark pink", markeredgewidth=3,
                             markersize=8)
# some explanation
ax0.text(0.99, 0.97, f"kernel: boxcar (rectangle)\nwidth = {window_width_min:.0f} minutes",
         horizontalalignment='right', verticalalignment='top',
         fontsize=14)
# axis tweaking
ax0.set(ylim=ylim,
        xlim=[start, end],
        ylabel="Temperature (°C)",
        yticks=[5,10,15,20],
        title="Yatir Forest, 2022")
ax1.set(ylim=[-0.2, 1.2],
        xlim=[start, end],
        ylabel="kernel"
       )
# adjust dates on both panels as defined before
center_dates_two_panels(ax0, ax1)

def updateSwipe(k, lines):
    """
    updates both panels, given the index k along which the window is sliding
    """
    # left side of the sliding window
    t0 = tSwipe[k]
    # middle position
    window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
    # erase previous blue shade on the top graph
    lines.fillBet.remove()
    # fill again the blue shade in the updated window position
    lines.fillBet = ax0.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                    y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1, color='blue')

```

```

# update pink curve
lines.pink_line.set_data(df[start>window_middle].index,
                         df.loc[start>window_middle, 'con'].values)

# update pink circle
lines.pink_circle.set_data([window_middle], [df.loc[window_middle, 'con']])

# update the kernel in its current position
lines.kernel_rect = np.ones(window_width_int)
df.loc[:, 'kernel_plus'] = 0.0
df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = kernel_rect

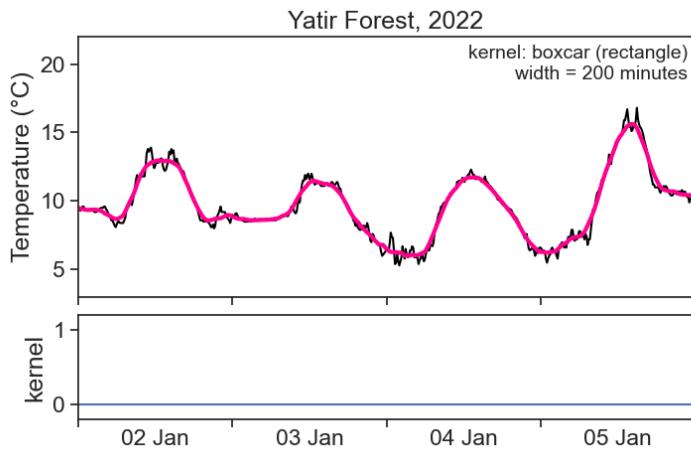
# update gray line
lines.gray_line.set_data(df.loc[df['kernel_plus']==1.0, 'TD'].index,
                         df.loc[df['kernel_plus']==1.0, 'TD'].values)

# update kernel line
lines.kernel_line.set_data(df['kernel_plus'].index, df['kernel_plus'].values)

# create a tqdm progress bar
progress_bar = tqdm(total=len(t_swipe), unit="iteration")
# loop over all sliding indices, update graph and then save it
for fignum, i in enumerate(np.arange(0, len(t_swipe)-1, 1)):
    update_swipe(i, lines)
    fig.savefig(f"pngs/boxcar{window_width_min_integer}/boxcar_{window_width_min_integer}min_{i}.png")
    # update the progress bar
    progress_bar.update(1)
# close the progress bar
progress_bar.close()

```

100% | 604/605 [05:27<00:00, 1.85iteration/s]



Combine all saved images into one mp4 video.

```
# Define the path to your PNG images
pngs_path = f"pngs/boxcar{window_width_min_integer}"
pngs_name = f"boxcar_{window_width_min_integer}min_%03d.png"

# Define the output video file path
video_output = f"output{window_width_min_integer}.mp4"

# Use ffmpeg to create a video from PNG images
# desired framerate. choose 24 if you don't know what to do
fr = 12
# run command
ffmpeg_cmd = f"ffmpeg -framerate {fr} -i {pngs_path}/{pngs_name} -c:v libx264 -vf fps={fr} {video_output}"
subprocess.run(ffmpeg_cmd, shell=True)
```

```
ffmpeg version 6.0 Copyright (c) 2000-2023 the FFmpeg developers
  built with Apple clang version 14.0.3 (clang-1403.0.22.14.1)
configuration: --prefix=/usr/local/Cellar/ffmpeg/6.0 --enable-shared --enable-pthreads --enable
libavutil      58. 2.100 / 58. 2.100
libavcodec     60. 3.100 / 60. 3.100
libavformat    60. 3.100 / 60. 3.100
libavdevice    60. 1.100 / 60. 1.100
libavfilter     9. 3.100 /  9. 3.100
libswscale      7. 1.100 /   7. 1.100
libswresample   4. 10.100 /   4. 10.100
```

```

libpostproc      57. 1.100 / 57. 1.100
Input #0, image2, from 'pngs/boxcar200/boxcar_200min_%03d.png':
  Duration: 00:00:50.33, start: 0.000000, bitrate: N/A
  Stream #0:0: Video: png, rgba(pc), 4800x3000 [SAR 23622:23622 DAR 8:5], 12 fps, 12 tbr, 12 t
Stream mapping:
  Stream #0:0 -> #0:0 (png (native) -> h264 (libx264))
Press [q] to stop, [?] for help
[libx264 @ 0x7fa027f2e300] using SAR=1/1
[libx264 @ 0x7fa027f2e300] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
[libx264 @ 0x7fa027f2e300] profile High 4:4:4 Predictive, level 6.0, 4:4:4, 8-bit
[libx264 @ 0x7fa027f2e300] 264 - core 164 r3095 baee400 - H.264/MPEG-4 AVC codec - Copyleft 200
Output #0, mp4, to 'output200.mp4':
Metadata:
  encoder       : Lavf60.3.100
Stream #0:0: Video: h264 (avc1 / 0x31637661), yuv444p(tv, progressive), 4800x3000 [SAR 1:1 D
  Metadata:
    encoder       : Lavc60.3.100 libx264
  Side data:
    cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: N/A
frame= 604 fps= 23 q=-1.0 Lsize=     1412kB time=00:00:50.08 bitrate= 231.0kbits/s speed=1.91x
video:1404kB audio:0kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.56%
[libx264 @ 0x7fa027f2e300] frame I:3      Avg QP: 9.98  size:135751
[libx264 @ 0x7fa027f2e300] frame P:154    Avg QP:14.75  size:  2507
[libx264 @ 0x7fa027f2e300] frame B:447    Avg QP:22.66  size:  1440
[libx264 @ 0x7fa027f2e300] consecutive B-frames:  1.0%  0.7%  1.0% 97.4%
[libx264 @ 0x7fa027f2e300] mb I  I16..4: 55.5% 38.8%  5.7%
[libx264 @ 0x7fa027f2e300] mb P  I16..4:  0.4%  0.3%  0.0%  P16..4:  0.2%  0.1%  0.0%  0.0%  0.0%
[libx264 @ 0x7fa027f2e300] mb B  I16..4:  0.1%  0.0%  0.0%  B16..8:  1.0%  0.2%  0.0%  direct: 0.0%
[libx264 @ 0x7fa027f2e300] 8x8 transform intra:37.1% inter:49.0%
[libx264 @ 0x7fa027f2e300] coded y,u,v intra: 3.6% 0.4% 0.6% inter: 0.1% 0.0% 0.0%
[libx264 @ 0x7fa027f2e300] i16 v,h,dc,p: 90% 10%  0%  0%
[libx264 @ 0x7fa027f2e300] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 41%  3% 56%  0%  0%  0%  0%  0%  0%
[libx264 @ 0x7fa027f2e300] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 51% 14% 20%  3%  2%  3%  2%  3%  2%
[libx264 @ 0x7fa027f2e300] Weighted P-Frames: Y:0.0% UV:0.0%
[libx264 @ 0x7fa027f2e300] ref P L0: 56.6%  3.8% 28.4% 11.3%
[libx264 @ 0x7fa027f2e300] ref B L0: 85.6% 13.4%  1.0%
[libx264 @ 0x7fa027f2e300] ref B L1: 95.9%  4.1%
[libx264 @ 0x7fa027f2e300] kb/s:228.44

```

```
CompletedProcess(args='ffmpeg -framerate 12 -i pngs/boxcar200/boxcar_200min_%03d.png -c:v libx264 -b:a 128k -t 50 -pix_fmt yuv444p output200.mp4')
```

The following code does exactly as you see above, but it is not well commented. You are an intelligent person, you'll figure this out.

## 75.5 Triangular kernel

```
%matplotlib widget
fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True,
                      gridspec_kw={'height_ratios':[1,0.4], 'hspace':0.1})

class Lines:
    pass
lines = Lines()

ax0 = ax[0]
ax1 = ax[1]
ylim = [3, 22]
window_width_min = 500.0
window_width_int = int(window_width_min / 10) + 1
N = len(df)
t_swipe = pd.date_range(start=pd.to_datetime(start) - pd.Timedelta(minutes=window_width_min) -
                        end=pd.to_datetime(end) + pd.Timedelta(minutes=60),
                        freq="10min")
t0 = t_swipe[200]
window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
# fill between blue shade, plot kernel
lines.fill_bet = ax0.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                  y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1)
half_triang = np.arange(1, window_width_int/2+1, 1)
kernel_triang = np.hstack([half_triang, half_triang[-2::-1]])
kernel_triang = kernel_triang / kernel_triang.max()
df.loc[:, 'con'] = np.convolve(df['TD'].values, kernel_triang, mode='same') / len(kernel_triang)
df['kernel_plus'] = 0.0
df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = kernel_triang
lines.kernel_line, = ax1.plot(df['kernel_plus'], color="tab:blue")
ax0.plot(df.loc[start:end, 'TD'], color="black")
lines.gray_line, = ax0.plot(df.loc[df['kernel_plus']!=0.0, 'TD'],
                           color=[0.6]*3, lw=3)
```

```

lines.pink_line, = ax0.plot(df.loc[start>window_middle, 'con'], color="xkcd:hot pink", lw=3)
lines.pink_circle, = ax0.plot([window_middle], [df.loc[window_middle, 'con']], 
    marker='o', markerfacecolor="None", markeredgecolor="xkcd:dark pink", markeredgewidth=2,
    markersize=8)
ax0.text(0.99, 0.97, f"kernel: triangle\nwidth = {window_width_min:.0f} minutes", transform=ax0.transAxes,
    horizontalalignment='right', verticalalignment='top',
    fontsize=14)
ax0.set(ylim=ylim,
    xlim=[start, end],
    ylabel="Temperature (°C)",
    yticks=[5,10,15,20],
    title="Yatir Forest, 2022")
ax1.set(ylim=[-0.2, 1.2],
    xlim=[start, end],
    ylabel="kernel"
    )
center_dates_two_panels(ax0, ax1)

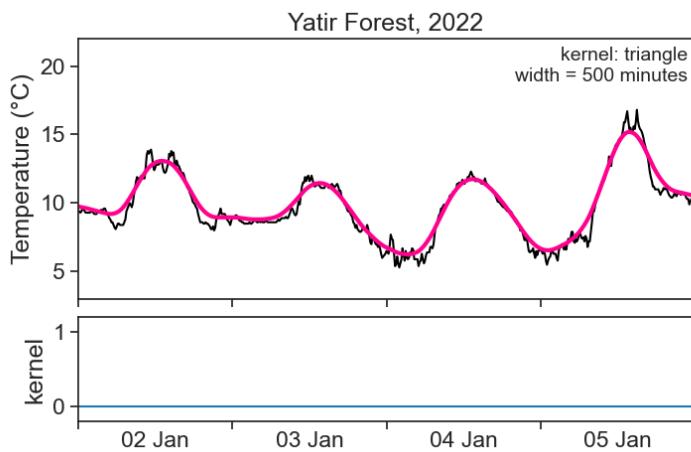
def update_swipe(k, lines):
    t0 = t_swipe[k]
    window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
    lines.fill_bet.remove()
    lines.fill_bet = ax0.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                    y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1, color="xkcd:gray")
    lines.pink_line.set_data(df[start>window_middle].index,
                            df.loc[start>window_middle, 'con'].values)
    lines.pink_circle.set_data([window_middle], [df.loc[window_middle, 'con']])
    lines.kernel_rect = np.ones(window_width_int)
    df['kernel_plus'] = 0.0
    df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = kernel_triangle
    lines.gray_line.set_data(df.loc[df['kernel_plus']!=0.0,'TD'].index,
                            df.loc[df['kernel_plus']!=0.0,'TD'].values)
    lines.kernel_line.set_data(df['kernel_plus'].index, df['kernel_plus'].values)

progress_bar = tqdm(total=len(t_swipe), unit="iteration")
for fignum, i in enumerate(np.arange(0, len(t_swipe)-1, 1)):
    update_swipe(i, lines)
    fig.savefig(f"pngs/triangle/triangle_{fignum:03}.png", dpi=600)
    # update the progress bar
    progress_bar.update(1)

```

```
# close the progress bar
progress_bar.close()
```

100% | 634/635 [05:35<00:00, 1.89iteration/s]



```
# Define the path to your PNG images
pngs_path = "pngs/triangle"
pngs_name = "triangle_%03d.png"
```

```
# Define the output video file path
video_output = "output_triangle.mp4"
```

```
fr = 12
```

```
# run command
```

```
ffmpeg_cmd = f"ffmpeg -framerate {fr} -i {pngs_path}/{pngs_name} -c:v libx264 -vf fps={fr} {video_output}"
subprocess.run(ffmpeg_cmd, shell=True)
```

```
ffmpeg version 6.0 Copyright (c) 2000-2023 the FFmpeg developers
built with Apple clang version 14.0.3 (clang-1403.0.22.14.1)
configuration: --prefix=/usr/local/Cellar/ffmpeg/6.0 --enable-shared --enable-pthreads --enable
libavutil      58. 2.100 / 58. 2.100
libavcodec     60. 3.100 / 60. 3.100
libavformat    60. 3.100 / 60. 3.100
libavdevice    60. 1.100 / 60. 1.100
```



```
CompletedProcess(args='ffmpeg -framerate 12 -i pngs/triangle/triangle_%03d.png -c:v libx264 -v
```

## 75.6 Gaussian kernel

```
%matplotlib widget
fig, ax = plt.subplots(2, 1, figsize=(8,5), sharex=True,
                      gridspec_kw={'height_ratios':[1,0.4], 'hspace':0.1})

class Lines:
    pass
lines = Lines()

ax0 = ax[0]
ax1 = ax[1]
ylim = [3, 22]
window_width_min = 500.0
window_width_int = int(window_width_min / 10) + 1
N = len(df)
t_swipe = pd.date_range(start=pd.to_datetime(start) - pd.Timedelta(minutes=window_width_min) -
                        end=pd.to_datetime(end) + pd.Timedelta(minutes=60),
                        freq="10min")
t0 = t_swipe[0]
window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
# fill between blue shade, plot kernel
half_triang = np.arange(1, window_width_int/2+1, 1)
kernel_triang = np.hstack([half_triang, half_triang[-2::-1]])
kernel_triang = kernel_triang / kernel_triang.max()
df['con'] = np.convolve(df['TD'].values, kernel_triang, mode='same') / len(kernel_triang) * 2
df['kernel_plus'] = 0.0
df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = kernel_triang

# array of minutes. multiply by 10 because data is every 10 minutes

std_in_minutes = 60
g = sp.signal.gaussian(window_width_int, std_in_minutes/10)#, sym=True)
df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = g
gaussian_threshold = np.exp(-2**2) # two sigmas
lines.kernel_line, = ax1.plot(df['kernel_plus'], color="tab:blue")
```

```

window_above_threshold = df.loc[df['kernel_plus'] > gaussian_threshold, 'kernel_plus'].index
lines.fill_bet = ax0.fill_between([window_above_threshold[0], window_above_threshold[-1]],
                                 y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1, color='gray')

# gaussian convolution from here: https://stackoverflow.com/questions/27205402/pandas-rolling-
df.loc[:, 'con'] = np.convolve(df['TD'].values, g/g.sum(), mode='same')
ax0.plot(df.loc[start:end, 'TD'], color="black")
lines.gray_line, = ax0.plot(df.loc[window_above_threshold[0]:window_above_threshold[-1], 'TD'],
                             color=[0.6]*3, lw=3)
lines.pink_line, = ax0.plot(df.loc[start:window_middle, 'con'], color="xkcd:hot pink", lw=3)
lines.pink_circle, = ax0.plot([window_middle], [df.loc[window_middle, 'con']],
                           marker='o', markerfacecolor="None", markeredgecolor="xkcd:dark pink", markeredgewidth=2,
                           markersize=8)
ax0.text(0.99, 0.97, f"kernel: gaussian\nnwidth = {window_width_min:.0f} minutes\nnstd = {std_in_",
         horizontalalignment='right', verticalalignment='top',
         fontsize=14)
ax0.set(ylim=ylim,
        xlim=[start, end],
        ylabel="Temperature (°C)",
        yticks=[5,10,15,20],
        title="Yatir Forest, 2022")
ax1.set(ylim=[-0.2, 1.2],
        xlim=[start, end],
        ylabel="kernel"
       )
gauss = df['TD'].rolling(window=window_width_int, center=True, win_type="gaussian").mean(std=6)
center_dates_two_panels(ax0, ax1)

def updateSwipe(k, lines):
    t0 = t_swipe[k]
    window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
    lines.fill_bet.remove()
    lines.pink_line.set_data(df[start:window_middle].index,
                            df.loc[start:window_middle, 'con'].values)
    lines.pink_circle.set_data([window_middle], [df.loc[window_middle, 'con']])
    lines.kernel_rect = np.ones(window_width_int)
    df['kernel_plus'] = 0.0
    df.loc[t0: t0 + pd.Timedelta(minutes=window_width_min), 'kernel_plus'] = g
    window_above_threshold = df.loc[df['kernel_plus'] > gaussian_threshold, 'kernel_plus'].index
    lines.gray_line.set_data(df.loc[window_above_threshold[0]:window_above_threshold[-1], 'TD'],
                            color=[0.6]*3, lw=3)

```

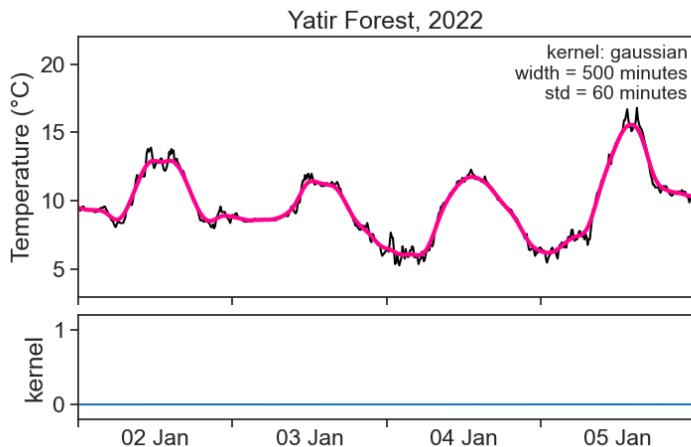
```

        df.loc[window_above_threshold[0]:window_above_threshold[-1], 'TD'] = 1
    lines.kernel_line.set_data(df['kernel_plus'].index, df['kernel_plus'].values)
    window_above_threshold = df.loc[df['kernel_plus'] > gaussian_threshold, 'kernel_plus'].index
    lines.fill_bet = ax0.fill_between([window_above_threshold[0], window_above_threshold[-1]],
                                      y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1, color="tan")

progress_bar = tqdm(total=len(t_swipe), unit="iteration")
for fignum, i in enumerate(np.arange(0, len(t_swipe)-1, 1)):
    update_swipe(i, lines)
    fig.savefig(f"pngs/gaussian/gaussian_{fignum:03}.png", dpi=600)
    progress_bar.update(1)
# close the progress bar
progress_bar.close()

```

100% | 634/635 [05:47<00:00, 1.83iteration/s]



```

# Define the path to your PNG images
pngs_path = "pngs/gaussian"
pngs_name = "gaussian_%03d.png"

# Define the output video file path
video_output = "output_gaussian.mp4"

fr = 12

```

```

# run command
ffmpeg_cmd = f"ffmpeg -framerate {fr} -i {pngs_path}/{pngs_name} -c:v libx264 -vf fps={fr} {video_name}"
subprocess.run(ffmpeg_cmd, shell=True)

ffmpeg version 6.0 Copyright (c) 2000-2023 the FFmpeg developers
  built with Apple clang version 14.0.3 (clang-1403.0.22.14.1)
configuration: --prefix=/usr/local/Cellar/ffmpeg/6.0 --enable-shared --enable-pthreads --enable-libavutil 58. 2.100 / 58. 2.100
               libavcodec 60. 3.100 / 60. 3.100
               libavformat 60. 3.100 / 60. 3.100
               libavdevice 60. 1.100 / 60. 1.100
               libavfilter  9. 3.100 /  9. 3.100
               libswscale   7. 1.100 /  7. 1.100
               libswresample 4. 10.100 /  4. 10.100
               libpostproc 57. 1.100 / 57. 1.100
Input #0, image2, from 'pngs/gaussian/gaussian_%03d.png':
  Duration: 00:00:52.83, start: 0.000000, bitrate: N/A
    Stream #0:0: Video: png, rgba(pc), 4800x3000 [SAR 23622:23622 DAR 8:5], 12 fps, 12 tbr, 12 tbn, 12 tbc
Stream mapping:
  Stream #0:0 -> #0:0 (png (native) -> h264 (libx264))
Press [q] to stop, [?] for help
[libx264 @ 0x7ff6d8907580] using SAR=1/1
[libx264 @ 0x7ff6d8907580] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
[libx264 @ 0x7ff6d8907580] profile High 4:4:4 Predictive, level 6.0, 4:4:4, 8-bit
[libx264 @ 0x7ff6d8907580] 264 - core 164 r3095 baee400 - H.264/MPEG-4 AVC codec - Copyleft 2002-2023
Output #0, mp4, to 'output_gaussian.mp4':
  Metadata:
    encoder         : Lavf60.3.100
  Stream #0:0: Video: h264 (avc1 / 0x31637661), yuv444p(tv, progressive), 4800x3000 [SAR 1:1 DAR 8:5]
    Metadata:
      encoder         : Lavc60.3.100 libx264
    Side data:
      cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: N/A
frame= 634 fps= 21 q=-1.0 Lsize= 1386kB time=00:00:52.58 bitrate= 215.9kbits/s speed=1.77x
video:1378kB audio:0kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.60%
[libx264 @ 0x7ff6d8907580] frame I:3     Avg QP:10.13  size:140267
[libx264 @ 0x7ff6d8907580] frame P:161   Avg QP:14.05  size: 2700
[libx264 @ 0x7ff6d8907580] frame B:470   Avg QP:21.81  size: 1180
[libx264 @ 0x7ff6d8907580] consecutive B-frames: 0.9% 0.6% 0.0% 98.4%
[libx264 @ 0x7ff6d8907580] mb I  I16..4: 53.9% 40.2%  5.9%

```

```
[libx264 @ 0x7ff6d8907580] mb P I16..4: 0.4% 0.3% 0.0% P16..4: 0.2% 0.1% 0.0% 0.0% 0
[libx264 @ 0x7ff6d8907580] mb B I16..4: 0.1% 0.0% 0.0% B16..8: 1.0% 0.1% 0.0% direct:
[libx264 @ 0x7ff6d8907580] 8x8 transform intra:39.0% inter:41.4%
[libx264 @ 0x7ff6d8907580] coded y,u,v intra: 3.0% 0.5% 0.6% inter: 0.0% 0.0% 0.0%
[libx264 @ 0x7ff6d8907580] i16 v,h,dc,p: 91% 9% 0% 0%
[libx264 @ 0x7ff6d8907580] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 40% 5% 55% 0% 0% 0% 0% 0% 0%
[libx264 @ 0x7ff6d8907580] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 45% 17% 20% 4% 3% 4% 2% 3% 2%
[libx264 @ 0x7ff6d8907580] Weighted P-Frames: Y:0.0% UV:0.0%
[libx264 @ 0x7ff6d8907580] ref P L0: 60.0% 3.7% 25.1% 11.2%
[libx264 @ 0x7ff6d8907580] ref B L0: 87.0% 11.7% 1.3%
[libx264 @ 0x7ff6d8907580] ref B L1: 96.7% 3.3%
[libx264 @ 0x7ff6d8907580] kb/s:213.50
```

```
CompletedProcess(args='ffmpeg -framerate 12 -i pngs/gaussian/gaussian_%03d.png -c:v libx264 -v
```

## 75.7 Comparison

Let's plot in one graph the smoothed temperature for each kernel shape we calculated above (rectangular, triangular, gaussian), all of which with a 500-minute-wide window.

```
window_width_min = 500.0
window_width_int = int(window_width_min // 10 + 1)

# rectangular, 500 min
kernel_rect = np.ones(window_width_int)
rect = np.convolve(df['TD'].values, kernel_rect, mode='same') / len(kernel_rect)

# triangular
half_triang = np.arange(1, window_width_int/2+1, 1)
kernel_triang = np.hstack([half_triang, half_triang[-2::-1]])
kernel_triang = kernel_triang / kernel_triang.max()
triang = np.convolve(df['TD'].values, kernel_triang, mode='same') / len(kernel_triang) * 2

# gaussian
gauss = df['TD'].rolling(window=window_width_int, center=True, win_type="gaussian").mean(std=6)
```

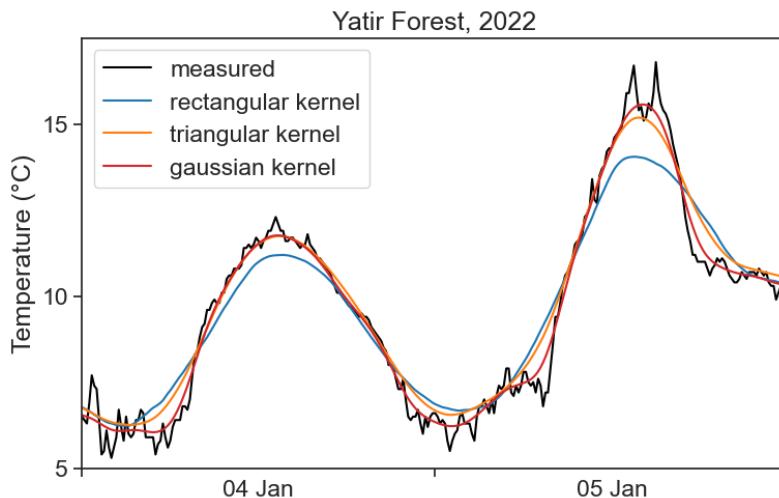
```

fig, ax = plt.subplots(figsize=(8,5))
ax.figure.subplots_adjust(top=0.93, bottom=0.10, left=0.1, right=0.95)

ax.plot(df.loc[start:end, 'TD'], color='black', label="measured")
ax.plot(df.index, rect, color="tab:blue", label="rectangular kernel")
ax.plot(df.index, triang, color="tab:orange", label="triangular kernel")
ax.plot(df.index, gauss, color="tab:red", label="gaussian kernel")
ax.legend()

ax.set(ylim=[5, 17.5],
       xlim=['2022-01-04 00:00:00', '2022-01-05 23:50:00'],
       ylabel="Temperature (°C)",
       title="Yatir Forest, 2022",
       yticks=[5,10,15])
center_dates(ax)
fig.savefig("kernel_comparison.png")

```



```

fig, ax = plt.subplots(figsize=(8,5))
ax.figure.subplots_adjust(top=0.93, bottom=0.15, left=0.1, right=0.95)

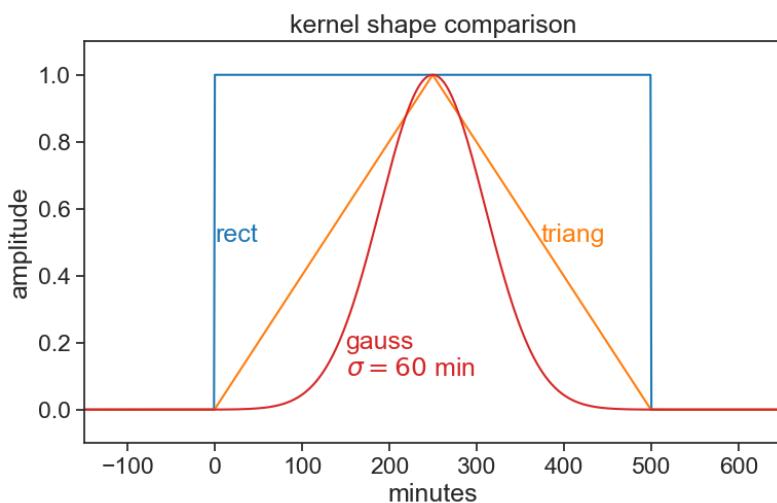
N=500
rec_window = np.zeros(800)
rec_window[150:150+N] = signal.windows.boxcar(N)
tri_window = np.zeros(800)
tri_window[150:150+N] = signal.windows.triang(N)

```

```

gau_window = np.zeros(800)
gau_window[150:150+N] = signal.windows.gaussian(N, std=60)
t = np.arange(-150, 650)
ax.plot(t, rec_window, color="tab:blue")
ax.plot(t, tri_window, color="tab:orange")
ax.plot(t, gau_window, color="tab:red")
ax.text(0, 0.5, "rect", color="tab:blue")
ax.text(373, 0.5, "triang", color="tab:orange")
ax.text(150, 0.1, "gauss\n"+r"$\sigma=60$ min", color="tab:red")
ax.set(ylim=[-0.1, 1.1],
      xlim=[-150, 650],
      ylabel="amplitude",
      xlabel="minutes",
      title="kernel shape comparison")
fig.savefig("kernel_shapes.png")

```



## savgol video

Import packages and stuff.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import scipy as sp
import json
import requests
import os
import subprocess
from tqdm import tqdm
from scipy import signal
from scipy.signal import savgol_filter

# avoid "SettingWithCopyWarning: A value is trying to be set on a copy of a slice from a DataF
pd.options.mode.chained_assignment = None # default='warn'
```

Download data from the [IMS](#) using an API.

```
# read token from file
with open('../archive/IMS-token.txt', 'r') as file:
    TOKEN = file.readline()
# 28 = SHANI station
STATION_NUM = 28
```

```

start = "2022/01/01"
end = "2022/01/07"
filename = 'shani_2022_january.json'

# check if the JSON file already exists
# if so, then load file
if os.path.exists(filename):
    with open(filename, 'r') as json_file:
        data = json.load(json_file)
else:
    # make the API request if the file doesn't exist
    url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/?from={start}&to={end}"
    headers = {'Authorization': f'ApiToken {TOKEN}'}
    response = requests.get(url, headers=headers)
    data = json.loads(response.text.encode('utf8'))

    # save the JSON data to a file
    with open(filename, 'w') as json_file:
        json.dump(data, json_file)
# show data to see if it's alright
# data

```

Load and process data.

```

df = pd.json_normalize(data['data'], record_path=['channels'], meta=['datetime'])
df['date'] = (pd.to_datetime(df['datetime'])
              .dt.tz_localize(None) # ignores time zone information
              )
df = df.pivot(index='date', columns='name', values='value')
df

```

name	Grad	RH	Rain	STDwd	TD	TDmax	TDmin	TG	TW	Time	WD	...
date												...
2022-01-01 00:00:00	0.0	77.0	0.0	10.3	11.2	11.2	11.1	10.7	-9999.0	2354.0	75.0	0
2022-01-01 00:10:00	0.0	77.0	0.0	11.2	11.2	11.2	11.1	10.8	-9999.0	1.0	77.0	8
2022-01-01 00:20:00	0.0	75.0	0.0	10.0	11.4	11.5	11.2	10.9	-9999.0	20.0	80.0	8
2022-01-01 00:30:00	0.0	74.0	0.0	9.6	11.5	11.5	11.4	11.0	-9999.0	22.0	76.0	7
2022-01-01 00:40:00	0.0	73.0	0.0	9.1	11.6	11.7	11.5	11.1	-9999.0	34.0	74.0	6
...	...	...	...	...	...	...	...	...	...	...	...	...

name		Grad	RH	Rain	STDwd	TD	TDmax	TDmin	TG	TW	Time	WD	V
date													
2022-01-06 23:10:00		0.0	36.0	0.0	16.1	11.6	12.0	11.1	6.8	-9999.0	2310.0	144.0	
2022-01-06 23:20:00		0.0	35.0	0.0	10.1	12.1	12.3	11.9	6.3	-9999.0	2320.0	118.0	
2022-01-06 23:30:00		0.0	36.0	0.0	7.1	12.4	12.6	11.9	7.3	-9999.0	2330.0	113.0	
2022-01-06 23:40:00		0.0	37.0	0.0	5.6	12.6	12.7	12.5	7.8	-9999.0	2339.0	119.0	
2022-01-06 23:50:00		0.0	39.0	0.0	11.5	11.9	12.6	11.5	7.1	-9999.0	2341.0	102.0	

Define useful functions.

```
def concise(ax):
    """
    Let python choose the best xtick labels for you
    """
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

# dirty trick to have dates in the middle of the 24-hour period
# make minor ticks in the middle, put the labels there!
# from https://matplotlib.org/stable/gallery/ticks/centered_ticklabels.html

def center_dates(ax):
    # show day of the month + month abbreviation. see full option list here:
    # https://strftime.org
    date_form = DateFormatter("%d %b")
    # major ticks at midnight, every day
    ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
```

```

        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')

def center_dates_two_panels(ax0, ax1):
    # show day of the month + month abbreviation. see full option list here:
    date_form = DateFormatter("%d %b")
    # major ticks at midnight, every day
    ax0.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax1.xaxis.set_major_locator(mdates.DayLocator(interval=1))
    ax1.xaxis.set_major_formatter(date_form)
    # minor ticks at noon, every day
    ax1.xaxis.set_minor_locator(mdates.HourLocator(byhour=[12]))
    # erase major tick labels
    ax1.xaxis.set_major_formatter(ticker.NullFormatter())
    # set minor tick labels as define above
    ax1.xaxis.set_minor_formatter(date_form)
    # completely erase minor ticks, center tick labels
    for tick in ax0.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
    for tick in ax1.xaxis.get_minor_ticks():
        tick.tick1line.set_markersize(0)
        tick.tick2line.set_markersize(0)
        tick.label1.set_horizontalalignment('center')

```

We don't need the full month, let's cut the dataframe to fewer days.

```

start = "2022-01-01 00:00:00"
end = "2022-01-06 23:50:00"
df = df.loc[start:end]

```

We now redefine a narrower window, this will be the graph's xlims. We leave the dataframe as is, because we will need some data outside the graph's limits.

```

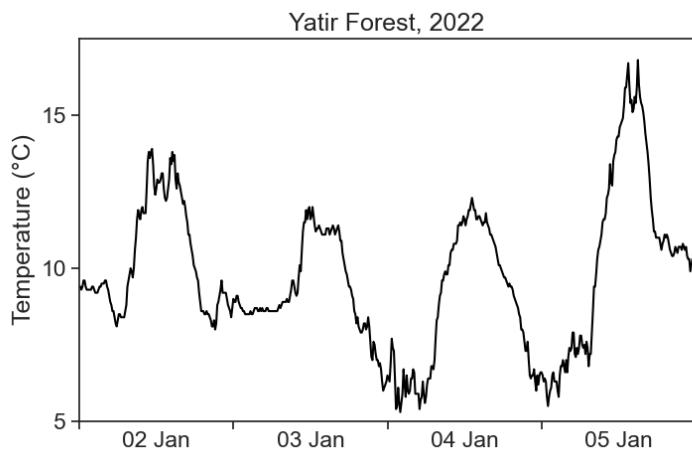
start = "2022-01-02 00:00:00"
end = "2022-01-05 23:50:00"

```

```

fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df.loc[start:end, 'TD'], color='black')
ax.set(ylim=[5, 17.5],
       xlim=[start, end],
       ylabel="Temperature (°C)",
       title="Yatir Forest, 2022",
       yticks=[5,10,15])
center_dates(ax)
# fig.savefig("sliding_YF_temperature_2022.png")

```



Looks good. Let's move on.

## 75.8 Savgol filter

```

# Function to fit and get polynomial values
def fit_polynomial(x, y, degree):
    coeffs = np.polyfit(x, y, degree)
    poly = np.poly1d(coeffs)
    return poly(x), coeffs

# Function to fit and get polynomial values
def poly_coeffs(x, y, degree):
    coeffs = np.polyfit(x, y, degree)
    return coeffs

```

```
%matplotlib widget
fig, ax = plt.subplots(figsize=(8,5))
ax.plot(df.loc[start:end, 'TD'], color='black')

sg = savgol_filter(df['TD'], 13, 2)

i = 500
ax.plot(df.index[:i], sg[:i], color='xkcd:hot pink')

window_pts = 31
p_order = 3

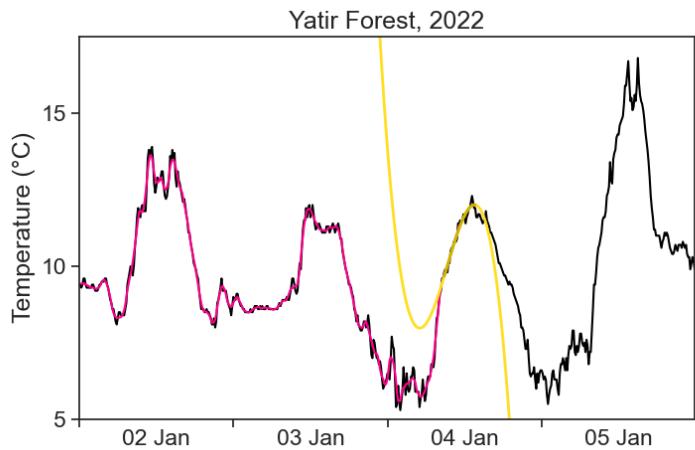
window_x = np.arange(i - window_pts // 2, i + window_pts // 2)
window_y = df['TD'][i - window_pts // 2:i + window_pts // 2]

# Fit and plot polynomial inside the window
fitted_y, coeffs = fit_polynomial(window_x, window_y, p_order)

whole_x = np.arange(len(df))
whole_y = df['TD'].values
poly = np.poly1d(coeffs)
whole_poly = poly(whole_x)

ax.plot(df.index, whole_poly, color='xkcd:sun yellow', lw=2)
# ax.plot(df.index[window_x], fitted_y, color='0.8', lw=3)
ax.plot(df.index[window_x], fitted_y, color='xkcd:mustard', lw=2)

ax.set(ylim=[5, 17.5],
       xlim=[start, end],
       ylabel="Temperature (°C)",
       title="Yatir Forest, 2022",
       yticks=[5,10,15])
center_dates(ax)
# fig.savefig("sliding_YF_temperature_2022.png")
```



```
p_order = 3
```

```
%matplotlib widget
fig, ax = plt.subplots(figsize=(8,5))

class Lines:
    """
    empty class, later will be populated with graph objects.
    this is useful to draw and erase lines on demand.
    """
    pass
lines = Lines()

# set graph y limits
ylim = [3, 22]
# choose here window width in minutes
window_width_min = 500.0
window_width_min_integer = int(window_width_min) # same but integer
window_width_int = int(window_width_min // 10 + 1) # window width in points
N = len(df) # df length
t_swipe = pd.date_range(start=pd.to_datetime(start) - pd.Timedelta(minutes=window_width_min) -
                        end=pd.to_datetime(end) + pd.Timedelta(minutes=60),
                        freq="10min")
# starting time
t0 = t_swipe[0]
ind0 = df.index.get_loc(t0) + window_width_int//2 + 1
```

```

# show sliding window on the top panel as a light blue shade
lines.fill_bet = ax.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                 y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1)

sg = savgol_filter(df['TD'], window_width_int, p_order)
df.loc[:, 'sg'] = sg
# plot temperature
ax.plot(df.loc[start:end, 'TD'], color="black")

# define x,y data inside window to execute polyfit on
window_x = np.arange(ind0 - window_width_int // 2, ind0 + window_width_int // 2)
window_y = df['TD'][ind0 - window_width_int // 2:ind0 + window_width_int // 2].values
# fit and plot polynomial inside the window
fitted_y, coeffs = fit_polynomial(window_x, window_y, p_order)
# get x,y data for the whole array
whole_x = np.arange(len(df))
whole_y = df['TD'].values
poly = np.poly1d(coeffs)
whole_poly = poly(whole_x)

# calculate the middle of the sliding window
window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
# plot a pink line showing the result of the moving average
# from the beginning to the middle of the sliding window
lines.pink_line, = ax.plot(df.loc[start:window_middle, 'sg'], color="xkcd:hot pink", lw=3)

lines.poly_all, = ax.plot(df.index, whole_poly, color='xkcd:sun yellow', lw=2)
lines.poly_window, = ax.plot(df.index[window_x], fitted_y, color='xkcd:mustard', lw=2)

# emphasize the location of the middle on the window with a circle
lines.pink_circle, = ax.plot([window_middle], [df.loc[window_middle, 'sg']],
                             marker='o', markerfacecolor="None", markeredgecolor="xkcd:dark pink", markeredgewidth=2,
                             markersize=8)
# some explanation
ax.text(0.99, 0.97, f"savitzky-golay\nwidth = {window_width_int:.0f} pts\npoly order = {p_order}",
        horizontalalignment='right', verticalalignment='top',
        fontsize=14)
# axis tweaking
ax.set(ylim=ylim,
       xlim=[start, end],

```

```

        ylabel="Temperature (°C)",
        yticks=[5,10,15,20],
        title="Yatir Forest, 2022")
# adjust dates on both panels as defined before
center_dates(ax)

def updateSwipe(k, lines):
    """
    updates both panels, given the index k along which the window is sliding
    """
    # left side of the sliding window
    t0 = t_swipe[k]
    # middle position
    window_middle = t0 + pd.Timedelta(minutes=window_width_min/2)
    ind0 = df.index.get_loc(window_middle)
    # erase previous blue shade on the top graph
    lines.fill_bet.remove()
    # fill again the blue shade in the updated window position
    lines.fill_bet = ax.fill_between([t0, t0 + pd.Timedelta(minutes=window_width_min)],
                                    y1=ylim[0], y2=ylim[1], alpha=0.1, zorder=-1, color='blue')
    # update pink curve
    lines.pink_line.set_data(df[start:window_middle].index,
                             df.loc[start:window_middle, 'sg'].values)
    # update pink circle
    lines.pink_circle.set_data([window_middle], [df.loc[window_middle, 'sg']])
    # define x,y data inside window to execute polyfit on

    window_x = np.arange(ind0 - window_width_int // 2, ind0 + window_width_int // 2)
    window_y = df['TD'][ind0 - window_width_int // 2:ind0 + window_width_int // 2]
    # fit and plot polynomial inside the window
    fitted_y, coeffs = fit_polynomial(window_x, window_y, p_order)
    poly = np.poly1d(coeffs)
    whole_poly = poly(window_x)
    lines.poly_all.set_data(df.index, whole_poly)
    lines.poly_window.set_data(df.index[window_x], fitted_y)

fig.savefig(f"pngs/savgol{window_width_int}/savgol_zero.png", dpi=600)

# create a tqdm progress bar
progress_bar = tqdm(total=len(t_swipe), unit="iteration")

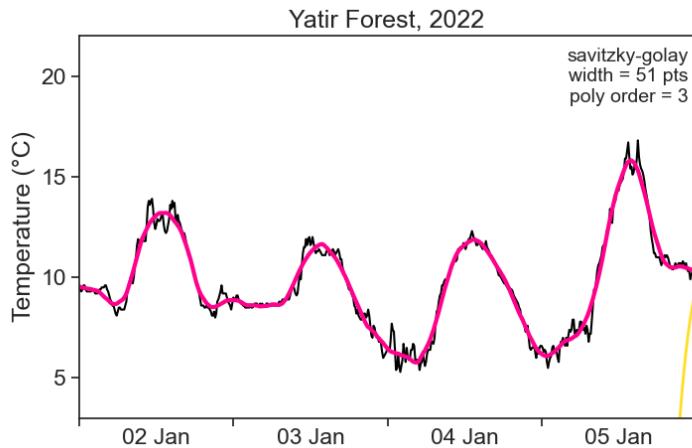
```

```

# loop over all sliding indices, update graph and then save it
for fignum, i in enumerate(np.arange(0, len(t_swipe)-1, 1)):
    update_swipe(i, lines)
    fig.savefig(f"pngs/savgol{window_width_int}/savgol_{window_width_int}_{fignum:03}.png", dpi=300)
    # update the progress bar
    progress_bar.update(1)
# close the progress bar
progress_bar.close()

```

100% | 634/635 [13:07<00:01, 1.24s/iteration]



Combine all saved images into one mp4 video.

```

# Define the path to your PNG images
pngs_path = f"pngs/savgol51"
pngs_name = f"savgol_51_%03d.png"

# Define the output video file path
video_output = f"output_savgol51.mp4"

# Use ffmpeg to create a video from PNG images
# desired framerate. choose 24 if you don't know what to do
fr = 12
# run command

```

```

ffmpeg_cmd = f"ffmpeg -framerate {fr} -i {pngs_path}/{pngs_name} -c:v libx264 -vf fps={fr} {video_name}"
subprocess.run(ffmpeg_cmd, shell=True)

ffmpeg version 6.1.1 Copyright (c) 2000-2023 the FFmpeg developers
  built with Apple clang version 15.0.0 (clang-1500.1.0.2.5)
configuration: --prefix=/usr/local/Cellar/ffmpeg/6.1.1_2 --enable-shared --enable-pthreads --
libavutil      58. 29.100 / 58. 29.100
libavcodec     60. 31.102 / 60. 31.102
libavformat    60. 16.100 / 60. 16.100
libavdevice    60.  3.100 / 60.  3.100
libavfilter     9. 12.100 /  9. 12.100
libswscale      7.  5.100 /  7.  5.100
libswresample   4. 12.100 /  4. 12.100
libpostproc    57.  3.100 / 57.  3.100
Input #0, image2, from 'pngs/savgol51/savgol_51_%03d.png':
  Duration: 00:00:52.83, start: 0.000000, bitrate: N/A
  Stream #0:0: Video: png, rgba(pc, gbr/unknown/unknown), 4800x3000 [SAR 23622:23622 DAR 8:5],
Stream mapping:
  Stream #0:0 -> #0:0 (png (native) -> h264 (libx264))
Press [q] to stop, [?] for help
[libx264 @ 0x7f9cb7906dc0] using SAR=1/1
[libx264 @ 0x7f9cb7906dc0] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
[libx264 @ 0x7f9cb7906dc0] profile High 4:4:4 Predictive, level 6.0, 4:4:4, 8-bit
[libx264 @ 0x7f9cb7906dc0] 264 - core 164 r3108 31e19f9 - H.264/MPEG-4 AVC codec - Copyleft 2005
Output #0, mp4, to 'output_savgol51.mp4':
  Metadata:
    encoder : Lavf60.16.100
  Stream #0:0: Video: h264 (avc1 / 0x31637661), yuv444p(tv, progressive), 4800x3000 [SAR 1:1 DAR 8:5]
    Metadata:
      encoder : Lavc60.31.102 libx264
    Side data:
      cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: N/A
[out#0/mp4 @ 0x7f9cb7806000] video:3513kB audio:0kB subtitle:0kB other streams:0kB global headers:0kB muxer overhead: 0.000000%
frame= 634 fps=3.2 q=-1.0 Lsize= 3521kB time=00:00:52.58 bitrate= 548.5kbits/s speed=0.27x
[libx264 @ 0x7f9cb7906dc0] frame I:3 Avg QP:13.70 size:146882
[libx264 @ 0x7f9cb7906dc0] frame P:232 Avg QP:19.02 size: 7856
[libx264 @ 0x7f9cb7906dc0] frame B:399 Avg QP:24.04 size: 3341
[libx264 @ 0x7f9cb7906dc0] consecutive B-frames: 11.4% 10.7% 10.4% 67.5%
[libx264 @ 0x7f9cb7906dc0] mb I  I16..4: 32.8% 61.0% 6.2%
[libx264 @ 0x7f9cb7906dc0] mb P  I16..4: 0.5% 0.7% 0.3% P16..4: 0.4% 0.2% 0.1% 0.0% 0.0%

```

```
[libx264 @ 0x7f9cb7906dc0] mb B I16..4: 0.1% 0.0% 0.0% B16..8: 1.9% 0.3% 0.0% direct:  
[libx264 @ 0x7f9cb7906dc0] 8x8 transform intra:51.3% inter:35.8%  
[libx264 @ 0x7f9cb7906dc0] coded y,u,v intra: 7.2% 6.5% 4.4% inter: 0.1% 0.1% 0.0%  
[libx264 @ 0x7f9cb7906dc0] i16 v,h,dc,p: 89% 10% 1% 0%  
[libx264 @ 0x7f9cb7906dc0] i8 v,h,dc,ddl,ddr,vr,hd,vl,hu: 31% 3% 65% 0% 0% 0% 0% 0%  
[libx264 @ 0x7f9cb7906dc0] i4 v,h,dc,ddl,ddr,vr,hd,vl,hu: 54% 7% 23% 3% 1% 4% 1% 5% 1%  
[libx264 @ 0x7f9cb7906dc0] Weighted P-Frames: Y:0.0% UV:0.0%  
[libx264 @ 0x7f9cb7906dc0] ref P L0: 58.7% 5.7% 25.2% 10.4%  
[libx264 @ 0x7f9cb7906dc0] ref B L0: 85.9% 11.8% 2.4%  
[libx264 @ 0x7f9cb7906dc0] ref B L1: 96.9% 3.1%  
[libx264 @ 0x7f9cb7906dc0] kb/s:544.58
```

```
CompletedProcess(args='ffmpeg -framerate 12 -i pngs/savgol51/savgol_51_%03d.png -c:v libx264 -v
```

## API to download data from IMS

```
# TOKEN = "f058958a-d8bd-47cc-95d7-7ecf98610e47"
# STATION_NUM = 28 # 28 = "SHANI"
# DATA = 10 # 10 = TDmax (max temperature)
# start = "2022/01/01"
# end = "2022/02/01"
# url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}"
# url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/?from={start}&to={end}"
# # url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/{DATA}/data/11/?from={start}&to={end}"
# headers = {'Authorization': 'ApiToken f058958a-d8bd-47cc-95d7-7ecf98610e47'}
# response = requests.request("GET", url, headers=headers)
# data= json.loads(response.text.encode('utf8'))

# # Save the JSON data to a file
# with open('shani_2022_january.json', 'w') as json_file:
#     json.dump(data, json_file)

# data

# # https://ims.gov.il/he/ObservationDataAPI
# # https://ims.gov.il/sites/default/files/2021-09/API%20explanation.pdf
# # https://ims.gov.il/sites/default/files/2022-04/Python%20API%20example.pdf
# TOKEN = "f058958a-d8bd-47cc-95d7-7ecf98610e47"
# STATION_NUM = 23 # 23 = "JERUSALEM CENTRE"
# DATA = 9 # 9 = TDmax (max temperature)
# start = "2022/01/01"
# end = "2022/02/01"
# url = f"https://api.ims.gov.il/v1/envista/stations/{STATION_NUM}/data/{DATA}/?from={start}&to={end}"
# headers = {'Authorization': 'ApiToken f058958a-d8bd-47cc-95d7-7ecf98610e47'}
# response = requests.request("GET", url, headers=headers)
# data= json.loads(response.text.encode('utf8'))
```

```
# print(url)

# url = "https://api.ims.gov.il/v1/envista/stations/28/data/10/data/11/?from=2022/01/01&to=2022/01/31"
# response = requests.request("GET", url, headers=headers)
# data = json.loads(response.text.encode('utf8'))

# # RH = 8
# # TDmax = 10, max temperature
# # TDmin = 11, min temperature
# url = "https://api.ims.gov.il/v1/envista/stations/28/data/10/?from=2022/01/01&to=2022/01/31"
# response = requests.request("GET", url, headers=headers)
# data = json.loads(response.text.encode('utf8'))

# df = pd.json_normalize(data['data'], record_path=['channels'], meta=['datetime'])
# df['date'] = pd.to_datetime(df['datetime']).dt.tz_localize(None) # ignore time zone information
# df = df.set_index('date')

# df
# data['data']
```

## remove consecutive values

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker
import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
# %matplotlib widget
```

create data, put some defective windows here and there...

```
steps = np.random.randint(low=-2, high=2, size=500)
data = steps.cumsum()
date_range = pd.date_range(start='2023-01-01', periods=len(data), freq='1D')
df = pd.DataFrame({'series': data}, index=date_range)

# make sequence of consecutive values
df.loc['2023-06-05':'2023-07-20', 'series'] = 2
df.loc['2023-10-05':'2023-10-25', 'series'] = -150
```

plot

```
def concise(ax):
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
```

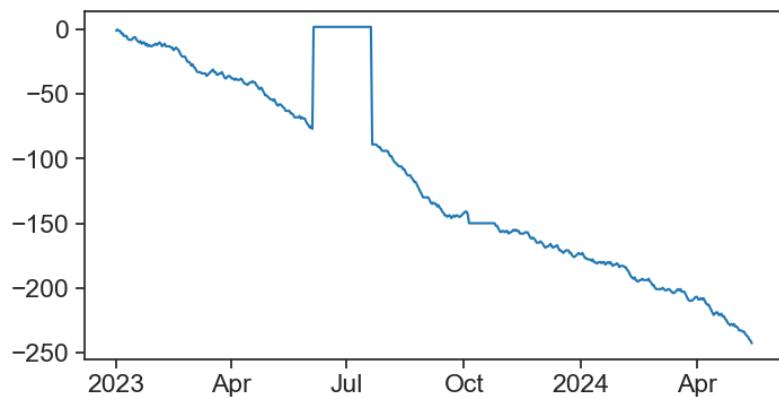
```

    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

fig, ax = plt.subplots(figsize=(8,4))
ax.plot(df['series'], color="tab:blue")
concise(ax)
ax.legend(frameon=False)

```

No artists with labels found to put in legend. Note that artists whose label start with an und



nice function, keep that for future reference

```

# function to copy paste:
def conseq_series(series, N):
    """
    part A:
    1. assume a string of 5 equal values. that's what we want to identify
    2. diff produces a string of only 4 consecutive zeros
    3. no problem, because when applying cumsum, the 4 zeros turn into a plateau of 5, that's
       so far, so good
    part B:
    1. groupby value_grp splits data into groups according to cumsum.
    2. because cumsum is monotonically increasing, necessarily all groups will be composed of
       identical values
    3. what are those groups made of? of rows of column 'series'. this specific column is not
       needed
    4. count 'counts' the number of elements inside each group.
    5. the real magic here is that 'transform' assigns each row of the original group with the
       same 'counts' value
    6. finally, we can ask the question: which rows belong to a string of identical values greater
       than N
    """
    # part A
    # 1
    diff = np.diff(series)
    # 2
    zero_locs = np.where(diff == 0)[0]
    # 3
    cumsum = np.cumsum(np.ones(len(series)))
    cumsum[zero_locs] = 5
    # part B
    # 1
    value_grp = series.groupby(cumsum)
    # 2
    counts = value_grp['series'].count()
    # 3
    counts.name = 'counts'
    # 4
    counts = counts.to_frame()
    # 5
    counts = counts.groupby(cumsum).transform(lambda x: x[0])
    # 6
    return counts

```

```

zehu, you now have a mask (True-False) with the same shape as the original series.

"""
# part A:
sumsum_series = (
    (series.diff() != 0)                      # diff zero becomes false, otherwise true
        .astype('int')                         # true -> 1 , false -> 0
        .cumsum()                             # cumulative sum, monotonically increasing
)
# part B:
mask_outliers = (
    series.groupby(sumsum_series)           # take original series and group it
        .transform('count')                  # now count how many are in each group
        .ge(N)                                # if row count >= than user-defined n
)
# apply mask:
result = pd.Series(np.where(mask_outliers,
                            np.nan,      # use this if mask_outliers is True
                            series),    # otherwise
                   index=series.index)
return result

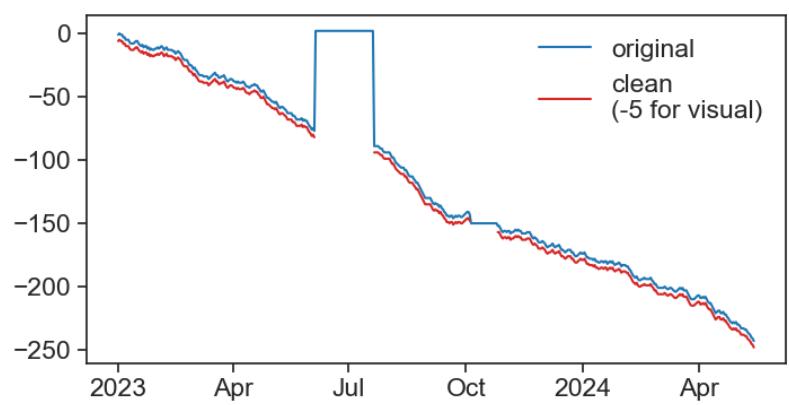
```

plot results. it works :)

```

fig, ax = plt.subplots(figsize=(8,4))
ax.plot(df['series'], color="tab:blue", label='original')
ax.plot(conseq_series(df['series'], 10)-5, c='tab:red', label='clean\n(-5 for visual)')
concise(ax)
ax.legend(frameon=False);

```



## outliers graphs

```
import matplotlib.pyplot as plt
import warnings
import pandas as pd
import numpy as np
import seaborn as sns
sns.set(style="ticks", font_scale=1.5) # white graphs, with large and legible letters
warnings.simplefilter(action='ignore', category=FutureWarning)
import matplotlib.gridspec as gridspec
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
from scipy.stats import median_absolute_deviation
```

<https://www.google.com/imgres?imgurl=https%3A%2F%2Fcxl.com%2Fwp-content%2Fuploads%2F2017%2F01%2Fchart-1.png&tbnid=RClfEFYNWm0WWM&vet=12ahUKEwjsx9KiqtKD>

source: [https://github.com/erykml/medium\\_articles/blob/master/Machine%20Learning/outlier\\_detection\\_ham](https://github.com/erykml/medium_articles/blob/master/Machine%20Learning/outlier_detection_ham)

### 75.9 define functions

```
def random_walk_with_outliers(origin, n_steps, perc_outliers=0.0, outlier_mult=10, seed=42):
    """
    Function for generating a random time series based on random walk.
    It adds a specified percentage of outliers by multiplying the random walk step by a scalar
    Parameters
    -----
    origin : int
        The starting point of the series
    n_steps : int
```

```

    Lenght of the series
perc_outliers : float
    Percentage of outliers to introduce to the series [0.0-1.0]
outlier_mult : float
    Scalar by which to multiply the RW increment to create an outlier
seed : int
    Random seed

Returns
-----
rw : np.ndarray
    The generated random walk series with outliers
indices : np.ndarray
    The indices of the introduced outliers
...
assert (perc_outliers >= 0.0) & (perc_outliers <= 1.0)

#set seed for reproducibility
np.random.seed(seed)

# possible steps
steps = [-1, 1]

# simulate steps
steps = np.random.choice(a=steps, size=n_steps-1)
rw = np.append(origin, steps).cumsum(0)

# add outliers
n_outliers = int(np.round(perc_outliers * n_steps, 0))
indices = np.random.randint(0, len(rw), n_outliers)
rw[indices] = rw[indices] + steps[indices + 1] * outlier_mult

return rw, indices

def concise(ax):
    locator = mdates.AutoDateLocator(minticks=3, maxticks=7)
    formatter = mdates.ConciseDateFormatter(locator)
    ax.xaxis.set_major_locator(locator)
    ax.xaxis.set_major_formatter(formatter)

```

## 75.10 load and process data

```
start = '2023-01-10 00:00:00'
n_steps = 1000
rw39, outlier_ind39 = random_walk_with_outliers(origin=0,
                                                 n_steps=n_steps,
                                                 perc_outliers=0.0031,
                                                 outlier_mult=50,
                                                 seed=39)
date_range = pd.date_range(start, periods=n_steps, freq='1min')
df = pd.DataFrame({'date': date_range, 'signal': rw39}).set_index('date')
start = df.index[0]
end = df.index[-1]

rw40, outlier_ind40 = random_walk_with_outliers(origin=0,
                                                 n_steps=n_steps,
                                                 perc_outliers=0.0031,
                                                 outlier_mult=50,
                                                 seed=40)
df['signal40'] = rw40

df.loc['2023-01-10 04:40:00', 'signal40'] = 43.0

# rw41, outlier_ind41 = random_walk_with_outliers(origin=0,
#                                                 n_steps=n_steps,
#                                                 perc_outliers=0.0031,
#                                                 outlier_mult=50,
#                                                 seed=41)
# df['signal41'] = rw41
```

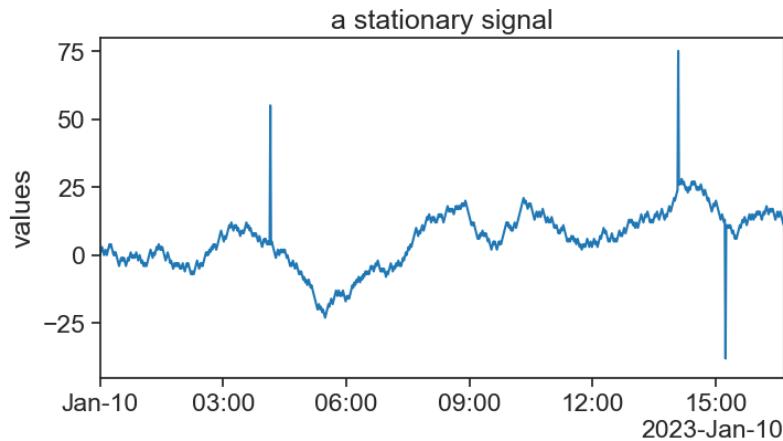
## 75.11 stationary signal

```
fig, ax = plt.subplots(figsize=(8,4))
# plot signal
ax.plot(df['signal'], color="tab:blue")
# make graph look nice
```

```

ax.set(ylabel='values',
       xlim=[start,end],
       title="a stationary signal",
       ylim=[-45, 80])
concise(ax)
fig.savefig("signal_39_stationary.png", bbox_inches='tight')

```



## 75.12 visual inspection

```

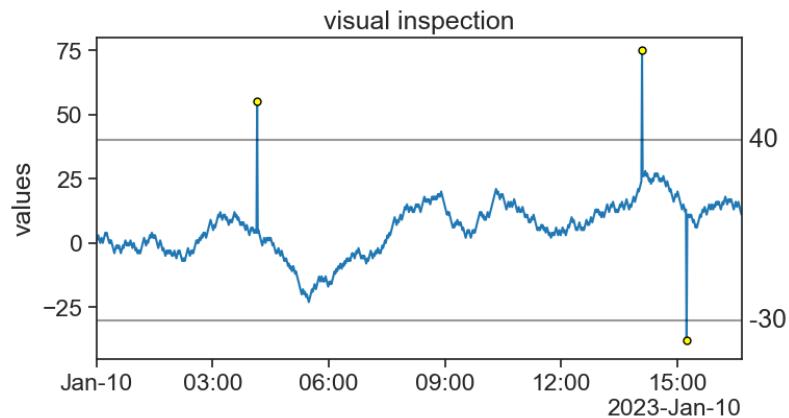
fig, ax = plt.subplots(figsize=(8,4))
# plot signal
ax.plot(df['signal'], color="tab:blue")
# plot horizontal lines
ax.plot([start, end], [40]*2, color="black", alpha=0.4)
ax.text(end, 40, " 40", va="center")
ax.plot([start, end], [-30]*2, color="black", alpha=0.4)
ax.text(end, -30, " -30", va="center")
# find and plot outliers
outliers_index = df.index[(df['signal'] > 40) | (df['signal'] < -30)]
ax.plot(df.loc[outliers_index, 'signal'], ls='None',
        marker='o', markerfacecolor='yellow', markersize=5,
        markeredgecolor="black")
# make graph look nice
ax.set(ylabel='values',

```

```

        xlim=[start,end],
        title="visual inspection",
        ylim=[-45, 80])
concise(ax)
fig.savefig("outliers_visual_inspection.png", bbox_inches='tight')

```



## 75.13 mean +- 3 std

```

fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

avg = df['signal'].mean()
std = df['signal'].std()

# plot signal
ax0.plot(df['signal'], color="tab:blue")

sns.kdeplot(data=df, y='signal', shade=True, ax=ax1)

```

```

pdf_xlim = ax1.get_xlim()
# plot horizontal lines
# mean
ax0.plot([start, end], [avg]*2, color="black", zorder=-10, alpha=0.7)
ax1.plot(pdf_xlim, [avg]*2, color="black", alpha=0.7)
ax1.text(1.1*pdf_xlim[1], avg, "mean", va="center")
# mean + std
ax0.plot([start, end], [avg+std]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [avg+std]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], avg+std, r"mean+$std", va="center", alpha=0.4)
# mean - std
ax0.plot([start, end], [avg-std]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [avg-std]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], avg-std, r"mean$-$std", va="center", alpha=0.4)

n_sigma = 3
# mean + 3std
ax0.plot([start, end], [avg+n_sigma*std]*2, color="tab:red")
ax1.plot(pdf_xlim, [avg+n_sigma*std]*2, color="tab:red")
ax1.text(1.1*pdf_xlim[1], avg+n_sigma*std, r"mean$+3\cdot std", va="center", color="tab:red")
# mean - 3std
ax0.plot([start, end], [avg-n_sigma*std]*2, color="tab:red")
ax1.plot(pdf_xlim, [avg-n_sigma*std]*2, color="tab:red")
ax1.text(1.1*pdf_xlim[1], avg-n_sigma*std, r"mean$-3\cdot std", va="center", color="tab:red")

# find and plot outliers
outliers_index = df.index[(df['signal'] > avg + n_sigma*std) |
                           (df['signal'] < avg - n_sigma*std)
                           ]
ax0.plot(df.loc[outliers_index, 'signal'], ls='None',
         marker='o', markerfacecolor='yellow', markersize=5,
         markeredgecolor="black")
# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-45, 80],
        title=r"threshold: mean $\pm 3\cdot std",
        )
concise(ax0)
ax1.set(xlabel='pdf',

```

```

        ylabel='',
        ylim=[-45, 80],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
    )
fig.savefig("outliers_3sigma.png", bbox_inches='tight')
)

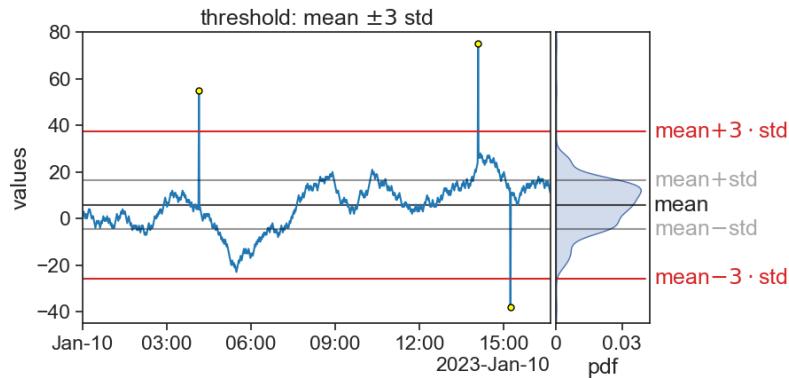
```

/var/folders/c3/7hp0d36n6vv8jc9hm2440\_\_00000gn/T/ipykernel\_73326/653833997.py:6: MatplotlibDep

```

ax0 = plt.subplot(gs[0, 0])

```



## 75.14 IQR

```

# get kdeplot data
fig, ax = plt.subplots(figsize=(8,4))
my_kde = sns.kdeplot(df['signal'], bw_adjust=0.5)
line = my_kde.lines[0]
kde_vals, kde_pdf = line.get_data()
kde_cdf = np.cumsum(kde_pdf) / np.sum(kde_pdf)

def find_nearest(array, value):
    return (np.abs(array - value)).argmin()

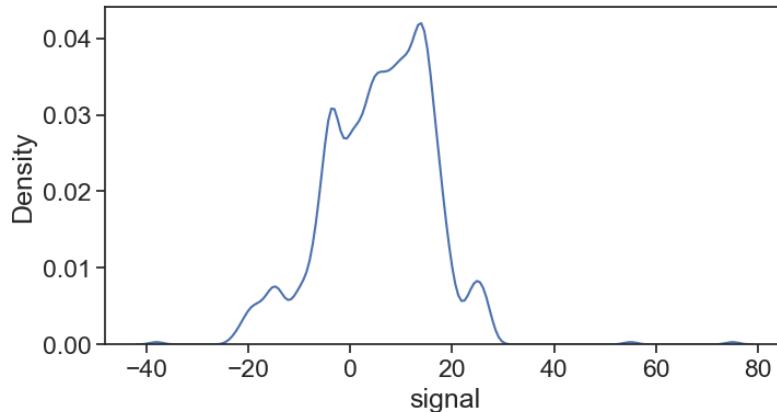
# Find the boundaries where the KDE is 25% and 75% of the area
Q1_index = find_nearest(kde_cdf, 0.25)

```

```

Q1_boundary = kde_vals[Q1_index]
Q3_index = find_nearest(kde_cdf, 0.75)
Q3_boundary = kde_vals[Q3_index]
IQR = Q3_boundary - Q1_boundary

```



```

fig, ax = plt.subplots(figsize=(8,4))

sns.kdeplot(df['signal'], ax=ax, shade=True, bw_adjust=0.5)

ax.fill_between(x=kde_vals[Q1_index:Q3_index],
                 y1=kde_pdf[Q1_index:Q3_index],
                 color="tab:pink"
                 )

h = 0.02
ax.annotate("", 
            xy=(Q1_boundary, h), xycoords='data',
            xytext=(Q3_boundary, h), textcoords='data',
            size=20,
            arrowprops=dict(arrowstyle="<->",
                           connectionstyle="arc3,rad=0.0",
                           shrinkA=0, shrinkB=0,
                           linewidth=2.5
                           ),
            )
ax.annotate("", 
            xy=(Q1_boundary, h), xycoords='data',

```

```

xytext=(Q3_boundary, h), textcoords='data',
size=20,
arrowprops=dict(arrowstyle "<->",
                connectionstyle="arc3,rad=0.0",
                shrinkA=0, shrinkB=0,
                linewidth=2.5
            ),
        )

ax.annotate("Q1\nquantile 0.25",
            xy=(Q1_boundary, 0.025), xycoords='data',
            xytext=(Q1_boundary-IQR, 0.040), textcoords='data',
            size=20,
            ha="right",
            va="top",
            arrowprops=dict(arrowstyle "->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=5",
                            shrinkA=0, shrinkB=0,
                            linewidth=2.5,
                            color="black"
                        ),
        )

ax.annotate("Q3\nquantile 0.75",
            xy=(Q3_boundary, 0.025), xycoords='data',
            xytext=(Q3_boundary+IQR, 0.040), textcoords='data',
            size=20,
            ha="left",
            va="top",
            arrowprops=dict(arrowstyle "->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=5",
                            shrinkA=0, shrinkB=0,
                            linewidth=2.5,
                            color="black"
                        ),
        )

ax.annotate(r"Q3 + 1.5$\cdot$IQR",
            xy=(Q3_boundary+1.5*IQR, 0.00), xycoords='data',
            xytext=(Q3_boundary+1.5*IQR, 0.015), textcoords='data',

```

```

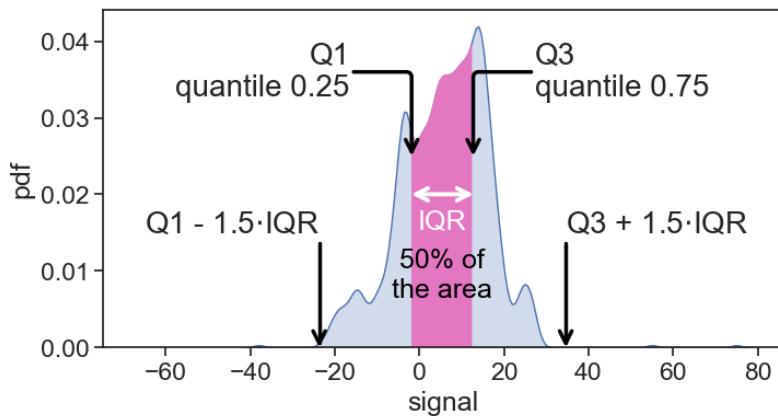
        size=20,
        ha="left",
        arrowprops=dict(arrowstyle="->",
                      connectionstyle="angle,angleA=0,angleB=90,rad=5",
                      shrinkA=0, shrinkB=0,
                      linewidth=2.5,
                      color="black"
                      ),
    )

ax.annotate(r"Q1 - 1.5$\cdot$IQR",
            xy=(Q1_boundary-1.5*IQR, 0.00), xycoords='data',
            xytext=(Q1_boundary-1.5*IQR, 0.015), textcoords='data',
            size=20,
            ha="right",
            arrowprops=dict(arrowstyle="->",
                          connectionstyle="angle,angleA=0,angleB=90,rad=5",
                          shrinkA=0, shrinkB=0,
                          linewidth=2.5,
                          color="black"
                          ),
        )

ax.text(Q1_boundary+IQR/2, 0.018, "IQR",
        ha="center", va="top", color="white")
ax.text(Q1_boundary+IQR/2, 0.013, r"50% of "+"the area",
        ha="center", va="top", color="black")

ax.set(xlim=[Q1_boundary-5*IQR, Q3_boundary+5*IQR],
       ylabel="pdf",)
fig.savefig("IQR_pdf.png", bbox_inches='tight')

```



```

fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

median = df['signal'].quantile(0.50)
Q1 = df['signal'].quantile(0.25)
Q3 = df['signal'].quantile(0.75)
IQR = Q3 - Q1

# plot signal
ax0.plot(df['signal'], color="tab:blue")

my_kde = sns.kdeplot(data=df, y='signal', shade=True, ax=ax1, bw_adjust=0.5)
pdf_xlim = ax1.get_xlim()

kde_q1_idx = find_nearest(kde_vals, Q1)
kde_q3_idx = find_nearest(kde_vals, Q3)
ax1.fill_betweenx(y=kde_vals[kde_q1_idx:kde_q3_idx],
                  x1=kde_pdf[kde_q1_idx:kde_q3_idx],
                  color="tab:pink")

# plot horizontal lines
# Q3 + 1.5 IQR
ax0.plot([start, end], [Q3+1.5*IQR]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [Q3+1.5*IQR]*2, color="black", alpha=0.4)

```

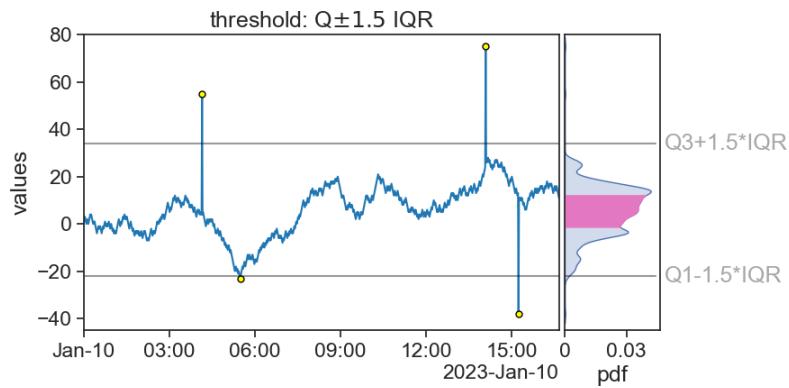
```

ax1.text(1.1*pdf_xlim[1], Q3+1.5*IQR, "Q3+1.5*IQR", va="center", alpha=0.4)
# Q1 - 1.5 IQR
ax0.plot([start, end], [Q1-1.5*IQR]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [Q1-1.5*IQR]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], Q1-1.5*IQR, "Q1-1.5*IQR", va="center", alpha=0.4)

# find and plot outliers
outliers_index = df.index[(df['signal'] > Q3+1.5*IQR) |
                           (df['signal'] < Q1-1.5*IQR)]
                           ]
ax0.plot(df.loc[outliers_index, 'signal'], ls='None',
          marker='o', markerfacecolor='yellow', markersize=5,
          markeredgecolor="black")
# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-45, 80],
        title=r"threshold: Q$\pm$1.5$ \times $ IQR",
        )
concise(ax0)
ax1.set(xlabel='pdf',
        ylabel='',
        ylim=[-45, 80],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
        )
fig.savefig("outliers_1.5IQR.png", bbox_inches='tight')

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_73326/3077230861.py:6: MatplotlibDep
```

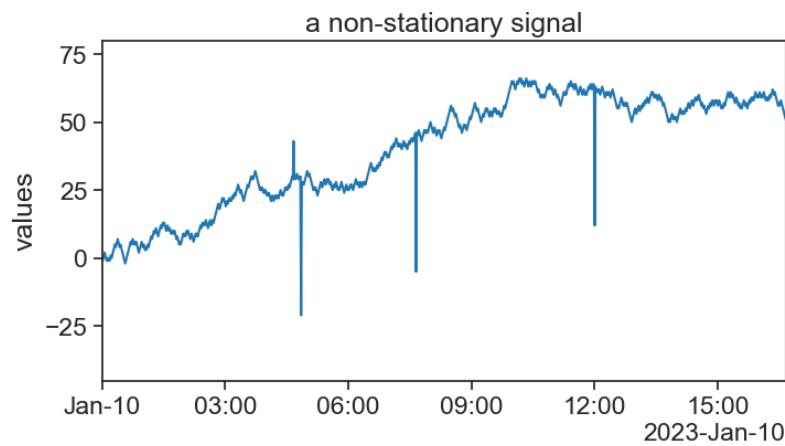


## 75.15 non stationary signal

```

fig, ax = plt.subplots(figsize=(8,4))
# plot signal
ax.plot(df['signal40'], color="tab:blue")
# make graph look nice
ax.set(ylabel='values',
       xlim=[start,end],
       title="a non-stationary signal",
       ylim=[-45, 80])
concise(ax)
fig.savefig("signal_40_non_stationary.png", bbox_inches='tight')

```



## 75.16 running +- 3 std

```
fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

avg = df['signal40'].mean()
std = df['signal40'].std()

# plot signal
ax0.plot(df['signal40'], color="tab:blue")

sns.kdeplot(data=df, y='signal40', shade=True, ax=ax1)

# plot horizontal lines
# mean
ax0.plot([start, end], [avg]*2, color="black", zorder=-10, alpha=0.7)
ax1.plot(pdf_xlim, [avg]*2, color="black", alpha=0.7)
ax1.text(1.1*pdf_xlim[1], avg, "mean", va="center")
# mean + std
ax0.plot([start, end], [avg+std]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [avg+std]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], avg+std, "mean+std", va="center", alpha=0.4)
# mean - std
ax0.plot([start, end], [avg-std]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [avg-std]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], avg-std, "mean-std", va="center", alpha=0.4)

n_sigma = 3
# mean + 3std
ax0.plot([start, end], [avg+n_sigma*std]*2, color="tab:red")
ax1.plot(pdf_xlim, [avg+n_sigma*std]*2, color="tab:red")
ax1.text(1.1*pdf_xlim[1], avg+n_sigma*std, r"mean $+3\sigma", va="center", color="tab:red")
# mean - 3std
```

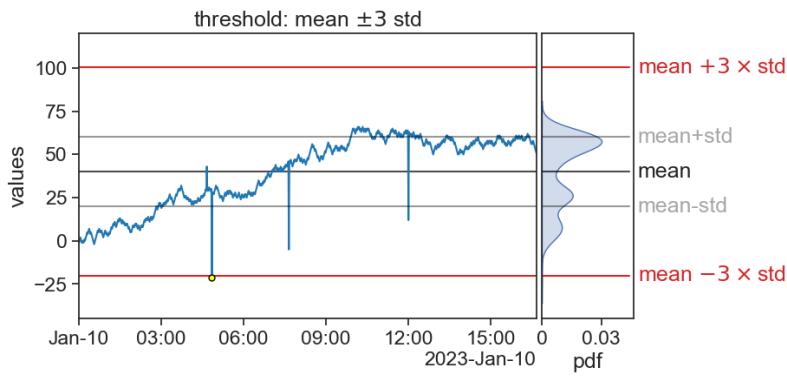
```

ax0.plot([start, end], [avg-n_sigma*std]*2, color="tab:red")
ax1.plot(pdf_xlim, [avg-n_sigma*std]*2, color="tab:red")
ax1.text(1.1*pdf_xlim[1], avg-n_sigma*std, r"mean $-3\sigma$ std", va="center", color="tab:red")

# find and plot outliers
outliers_index = df.index[(df['signal40'] > avg + n_sigma*std) |
                           (df['signal40'] < avg - n_sigma*std)
                          ]
ax0.plot(df.loc[outliers_index, 'signal40'], ls='None',
          marker='o', markerfacecolor='yellow', markersize=5,
          markeredgecolor="black")
# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-45, 120],
        title=r"threshold: mean $\pm 3\sigma$ std",
        )
concise(ax0)
ax1.set(xlabel='pdf',
        ylabel='',
        ylim=[-45, 120],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
       )
fig.savefig("outliers_3sigma_seed40.png", bbox_inches='tight')

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_73326/372016966.py:6: MatplotlibDep
```



```
df['signal40_rol_mean'] = df['signal40'].rolling('60min', center=True).mean()
df['signal40_rol_std'] = df['signal40'].rolling('60min', center=True).std()
```

```
fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

avg = df['signal40'].mean()
std = df['signal40'].std()

# plot signal
ax0.plot(df['signal40'], color="tab:blue")

sns.kdeplot(data=df, y='signal40', shade=True, ax=ax1)

# mean
ax0.plot(df['signal40_rol_mean'], color="black", alpha=0.7, label="mean")
# mean +- 3 std
n_sigma = 3
ax0.plot(df['signal40_rol_mean']+ n_sigma*df['signal40_rol_std'], color="tab:red")
plot_threshold, = ax0.plot(df['signal40_rol_mean']- n_sigma*df['signal40_rol_std'],
                           color="tab:red", label="threshold")

# find and plot outliers
outliers_index = df.index[(df['signal40'] > df['signal40_rol_mean']+ n_sigma*df['signal40_rol_std'])]
```

```

        (df['signal40'] < df['signal40_rol_mean'] - n_sigma*df['signal40_rol_std']
    ]
ax0.plot(df.loc[outliers_index, 'signal40'], ls='None',
          marker='o', markerfacecolor='yellow', markersize=5,
          markeredgecolor="black")
# make graph look nice
ax0.set(ylabel='values',
         xlim=[start,end],
         ylim=[-45, 120],
         title=r"1-hour rolling window: mean  $\pm 3$  std",
         )
concise(ax0)
ax1.set(xlabel='pdf',
         ylabel='',
         ylim=[-45, 120],
         yticks=[],
         xticks=[0, 0.03],
         xticklabels=['0', '0.03']
         )
ax0.legend(frameon=False, loc="upper left")

fig.savefig("outliers_rolling_3std.png", bbox_inches='tight')

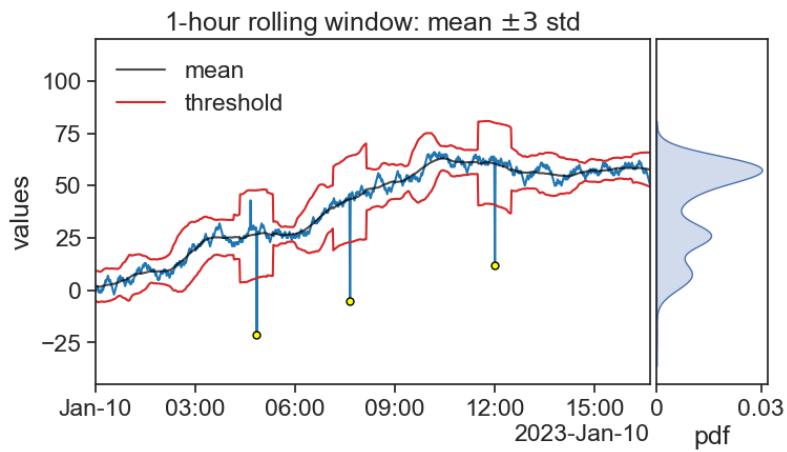
```

/var/folders/c3/7hp0d36n6vv8jc9hm2440\_\_0000gn/T/ipykernel\_73326/142534422.py:6: MatplotlibDep...

```

ax0 = plt.subplot(gs[0, 0])

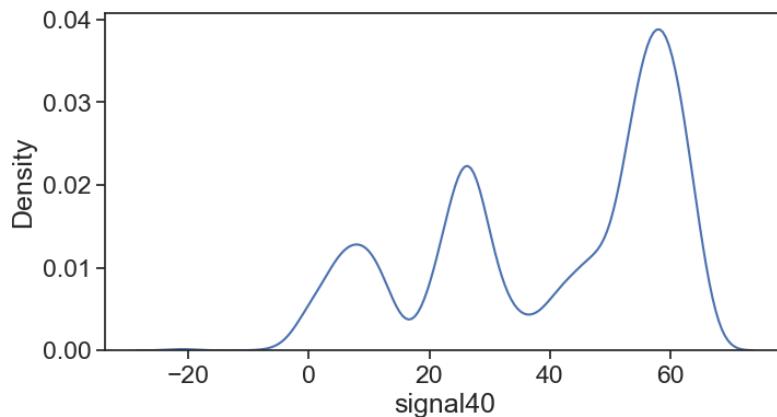
```



## 75.17 running: Q +- IQR

```
fig, ax = plt.subplots(figsize=(8,4))

my_kde = sns.kdeplot(df['signal40'], bw_adjust=0.5)
line = my_kde.lines[0]
kde_vals, kde_pdf = line.get_data()
kde_cdf = np.cumsum(kde_pdf) / np.sum(kde_pdf)
```



```
fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

median = df['signal40'].quantile(0.50)
Q1 = df['signal40'].quantile(0.25)
Q3 = df['signal40'].quantile(0.75)
IQR = Q3 - Q1

# plot signal
ax0.plot(df['signal40'], color="tab:blue")

my_kde = sns.kdeplot(data=df, y='signal40', shade=True, ax=ax1, bw_adjust=0.5)
```

```

pdf_xlim =ax1.get_xlim()

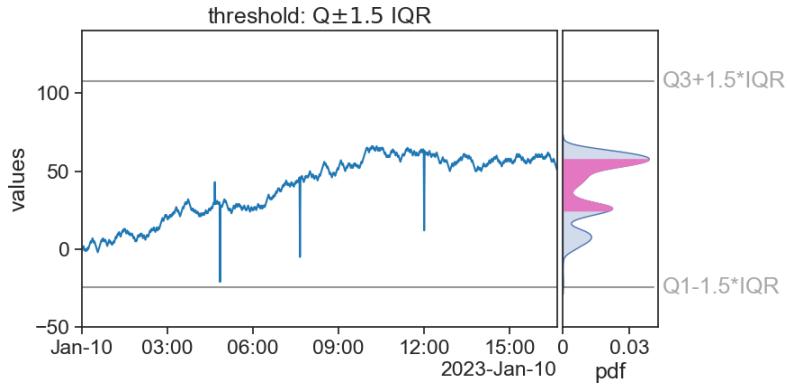
kde_q1_idx = find_nearest(kde_vals, Q1)
kde_q3_idx = find_nearest(kde_vals, Q3)
ax1.fill_betweenx(y=kde_vals[kde_q1_idx:kde_q3_idx] ,
                  x1=kde_pdf[kde_q1_idx:kde_q3_idx] ,
                  color="tab:pink")
# plot horizontal lines
# Q3 + 1.5 IQR
ax0.plot([start, end], [Q3+1.5*IQR]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [Q3+1.5*IQR]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], Q3+1.5*IQR, "Q3+1.5*IQR", va="center", alpha=0.4)
# Q1 - 1.5 IQR
ax0.plot([start, end], [Q1-1.5*IQR]*2, color="black", alpha=0.4)
ax1.plot(pdf_xlim, [Q1-1.5*IQR]*2, color="black", alpha=0.4)
ax1.text(1.1*pdf_xlim[1], Q1-1.5*IQR, "Q1-1.5*IQR", va="center", alpha=0.4)

# find and plot outliers
outliers_index = df.index[(df['signal40'] > Q3+1.5*IQR) |
                           (df['signal40'] < Q1-1.5*IQR)]
                           ]
ax0.plot(df.loc[outliers_index, 'signal40'], ls='None',
         marker='o', markerfacecolor='yellow', markersize=5,
         markeredgecolor="black")
# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-50, 140],
        title=r"threshold: Q$\pm$1.5$~$ IQR",
        )
concise(ax0)
ax1.set(xlabel='pdf',
        ylabel='',
        ylim=[-50, 140],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
        )
fig.savefig("outliers_1.5IQR_seed40.png", bbox_inches='tight')

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_73326/678430211.py:6: MatplotlibDep...
```

```
    ax0 = plt.subplot(gs[0, 0])
```



```
def Q1(window):
    return window.quantile(0.25)
def Q3(window):
    return window.quantile(0.75)
```

```
df['signal40_rol_Q1'] = df['signal40'].rolling('60min', center=True).apply(Q1)
df['signal40_rol_Q3'] = df['signal40'].rolling('60min', center=True).apply(Q3)
df['signal40_rol_IQR'] = df['signal40_rol_Q3'] - df['signal40_rol_Q1']
```

```
fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
           hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

avg = df['signal40'].mean()
std = df['signal40'].std()

# plot signal
ax0.plot(df['signal40'], color="tab:blue")

sns.kdeplot(data=df, y='signal40', shade=True, ax=ax1)
```

```

# median
# ax0.plot(df['signal40_rol_median'], color="black", alpha=0.7, label="median")
# Q1 - 1.5 IQR
threshold_bottom = df['signal40_rol_Q1'] - 1.5*df['signal40_rol_IQR']
ax0.plot(threshold_bottom, color="tab:red")
# Q3 + 1.5 IQR
threshold_top = df['signal40_rol_Q3'] + 1.5*df['signal40_rol_IQR']
ax0.plot(threshold_top, color="tab:red")

# find and plot outliers
outliers_index = df.index[(df['signal40'] > threshold_top) |
                           (df['signal40'] < threshold_bottom)]
ax0.plot(df.loc[outliers_index, 'signal40'], ls='None',
         marker='o', markerfacecolor='yellow', markersize=5,
         markeredgecolor="black")

# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-45, 120],
        title=r"1-hour rolling threshold: Q  $\pm 1.5$  IQR",
        )
concise(ax0)
ax1.set(xlabel='pdf',
        ylabel='',
        ylim=[-45, 120],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
        )

ax0.legend(frameon=False, loc="upper left")

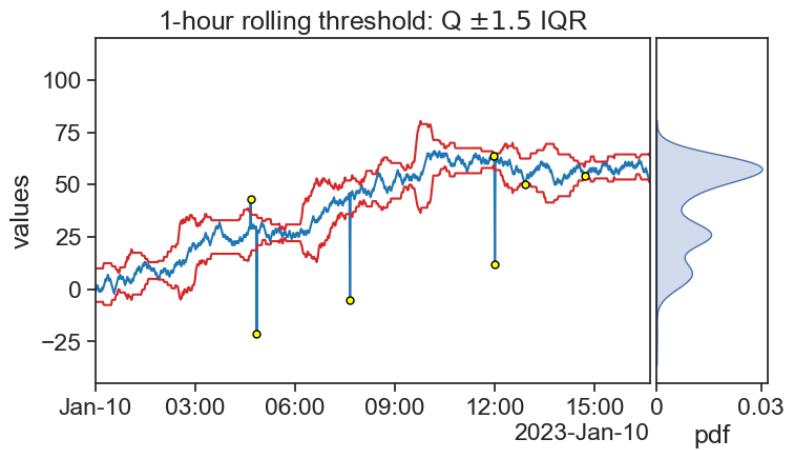
fig.savefig("outliers_rolling_IQR.png", bbox_inches='tight')

```

```

/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_73326/3108172465.py:6: MatplotlibDep
    ax0 = plt.subplot(gs[0, 0])
No artists with labels found to put in legend. Note that artists whose label start with an un

```



## 75.18 Hampel, running MAD

```

# def MAD(window):
#     return np.median(
#         np.abs(
#             window - np.median(window)
#         )
#     )

k = 1.4826 # scale factor for Gaussian distribution
def MAD(window):
    return (window - np.median(window)).abs().median()

df['signal40_rol_mad'] = k * df['signal40'].rolling('60min', center=True).apply(MAD)
df['signal40_rol_median'] = df['signal40'].rolling('60min', center=True).median()

fig, ax = plt.subplots(figsize=(8,4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 0.2], height_ratios=[1])
gs.update(left=0.10, right=0.90, top=0.95, bottom=0.13,
          hspace=0.02, wspace=0.02)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[0, 1])

```

```

# plot signal
ax0.plot(df['signal40'], color="tab:blue")

sns.kdeplot(data=df, y='signal40', shade=True, ax=ax1)

# median
# ax0.plot(df['signal40_rol_median'], color="black", alpha=0.7, label="median")
# Q1 - 1.5 IQR
threshold_bottom = df['signal40_rol_median'] - 3 * df['signal40_rol_mad']
ax0.plot(threshold_bottom, color="tab:red")
# Q3 + 1.5 IQR
threshold_top = df['signal40_rol_median'] + 3 * df['signal40_rol_mad']
ax0.plot(threshold_top, color="tab:red")

# find and plot outliers
outliers_index = df.index[(df['signal40'] > threshold_top) |
                           (df['signal40'] < threshold_bottom)]
ax0.plot(df.loc[outliers_index, 'signal40'], ls='None',
         marker='o', markerfacecolor='yellow', markersize=5,
         markeredgecolor="black")

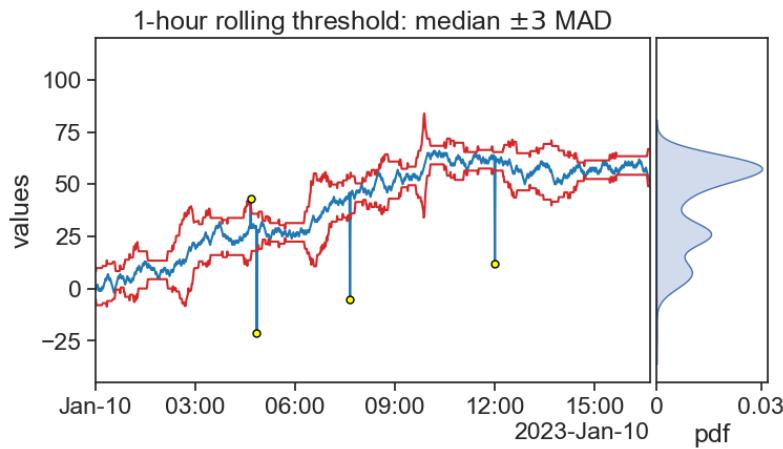
# make graph look nice
ax0.set(ylabel='values',
        xlim=[start,end],
        ylim=[-45, 120],
        title=r"1-hour rolling threshold: median $\pm$ MAD",
        )
concise(ax0)
ax1.set(xlabel='pdf',
        ylabel='',
        ylim=[-45, 120],
        yticks=[],
        xticks=[0, 0.03],
        xticklabels=['0', '0.03']
        )

ax0.legend(frameon=False, loc="upper left")

fig.savefig("outliers_rolling_MAD.png", bbox_inches='tight')

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_73326/3446982039.py:6: MatplotlibDep
    ax0 = plt.subplot(gs[0, 0])
No artists with labels found to put in legend. Note that artists whose label start with an un
```



## 75.19 stationary MAD

```
fig, ax = plt.subplots(figsize=(8,4))
# plot signal
ax.plot(df['signal'], color="tab:blue")

k = 1.4826 # scale factor for Gaussian distribution
mad = median_abs_deviation(df['signal'])
median = df['signal'].median()

xlim = ax.get_xlim()

# median +- 3*k*mad
ax.plot([start, end], [median]*2, color="black")
# ax.text(1.1, median, "median", va="center", transform=ax.transAxes,)

threshold_top = median+3*k*mad
threshold_bottom = median-3*k*mad

ax.plot([start, end], [threshold_bottom]*2, color="black", alpha=0.4)
```

```

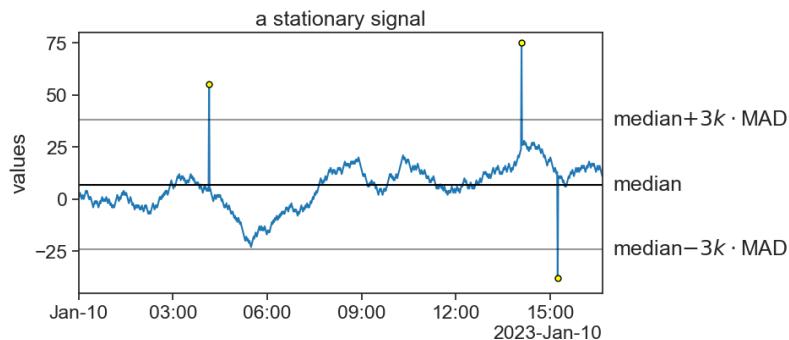
ax.plot([start, end], [threshold_top]*2, color="black", alpha=0.4)

ax.annotate("median", xy=(1.02, median), xycoords='axes fraction', 'data',
            va='center')
ax.annotate(r"median$+3k\cdot MAD", xy=(1.02, threshold_top), xycoords='axes fraction', 'data',
            va='center')
ax.annotate(r"median$-3k\cdot MAD", xy=(1.02, threshold_bottom), xycoords='axes fraction', 'data',
            va='center')

# find and plot outliers
outliers_index = df.index[(df['signal'] > threshold_top) |
                           (df['signal'] < threshold_bottom)]
                           ]
ax.plot(df.loc[outliers_index, 'signal'], ls='None',
        marker='o', markerfacecolor='yellow', markersize=5,
        markeredgecolor="black")

# make graph look nice
ax.set(ylabel='values',
       xlim=[start,end],
       title="a stationary signal",
       ylim=[-45, 80])
concise(ax)
fig.savefig("outliers_MAD_stationary.png", bbox_inches='tight')

```



```
outliers_index
```

```

DatetimeIndex(['2023-01-10 04:09:00', '2023-01-10 14:05:00',
               '2023-01-10 15:14:00'],
              dtype='datetime64[ns]', name='date', freq=None)

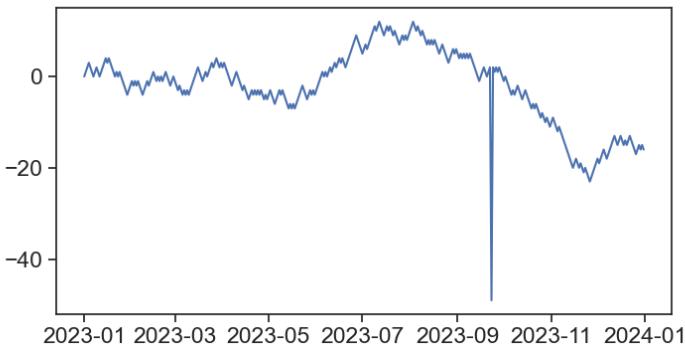
```

## 75.20 save data as csv for later usage

```
start = '2023-01-01 00:00:00'
end = '2023-12-31 23:55:00'
# date_range = pd.date_range(start, end, freq='5min')
date_range = pd.date_range(start, end, freq='1D')
n_steps = len(date_range)
rw39, outlier_ind39 = random_walk_with_outliers(origin=0,
                                                 n_steps=n_steps,
                                                 perc_outliers=0.0031,
                                                 outlier_mult=50,
                                                 seed=39)
# date_range = pd.date_range(start, periods=n_steps, freq='1min')
df = pd.DataFrame({'date': date_range, 'A': rw39}).set_index('date')
```

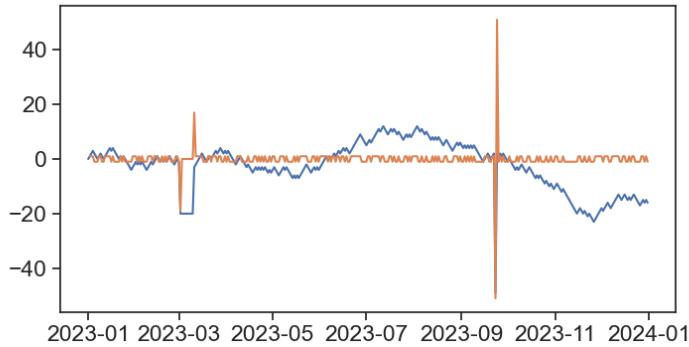
```
%matplotlib widget
fig, ax = plt.subplots(figsize=(8,4))

ax.plot(df['A'])
```



```
df.loc['2023-03-02':'2023-03-10', 'A'] = -20.0
```

```
%matplotlib widget
fig, ax = plt.subplots(figsize=(8,4))
df['Adiff'] = df['A'].diff()
ax.plot(df['A'])
ax.plot(df['Adiff'])
```



```
(df.loc['2023-03-03':'2023-03-06', 'Adiff'] == np.zeros(4)).all()
```

True

```
df.loc['2023-03-03':'2023-03-06', 'Adiff']
```

```
date
2023-03-03    0.0
2023-03-04    0.0
2023-03-05    0.0
2023-03-06    0.0
Name: Adiff, dtype: float64
```

```
np.zeros(4)
```

```
array([0., 0., 0., 0.])
```

```
n_consecutive = 2

def n_zeros(series, N):
    """
    True if all series equals np.zeros(N)
    False otherwise
    """
    return (series == np.zeros(N)).all()
```

```

df['mask1'] = df['Adiff'].rolling(n_consecutive).apply(n_zeros, args=(n_consecutive,))
df['mask'] = 0.0
for i in range(len(df)):
    if df['mask1'][i] == 1.0:
        df['mask'][i-n_consecutive:i+1] = 1.0

```

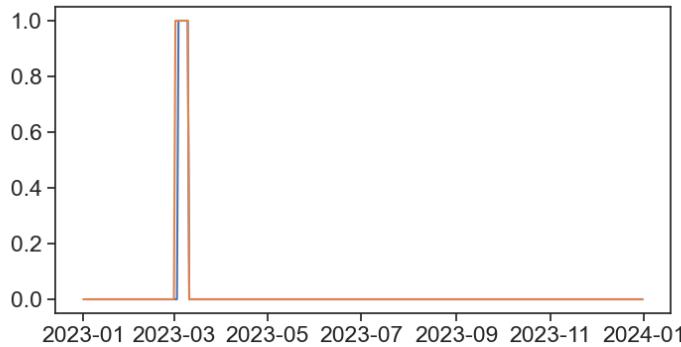
/var/folders/kv/9cqw3y\_s6c75xmgqm9n0t5d40000gn/T/ipykernel\_6180/2921310703.py:5: SettingWithCopyWarning  
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/](https://pandas.pydata.org/pandas-docs/stable/user_guide/)  
df['mask'][i-n\_consecutive:i+1] = 1.0

```

%matplotlib widget
fig, ax = plt.subplots(figsize=(8,4))
ax.plot(df['mask1'])
ax.plot(df['mask'])

```



```

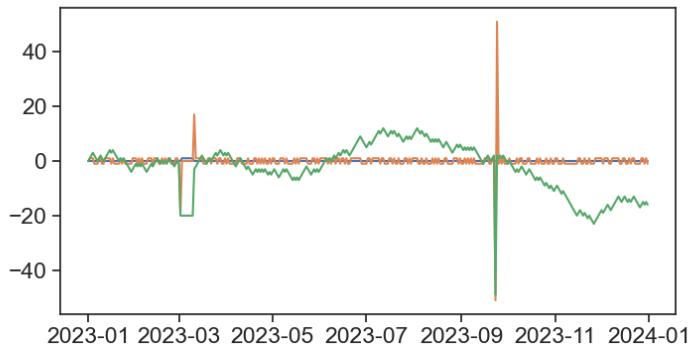
df['mask'] = 0.0
for j in range(len(df)-n_consecutive):
    if (df['Adiff'][j:j+n_consecutive] == np.zeros(n_consecutive)).all():
        df['mask'][j:j+n_consecutive] = 1.0

```

/var/folders/kv/9cqw3y\_s6c75xmgqm9n0t5d40000gn/T/ipykernel\_6180/2364603426.py:4: SettingWithCopyWarning  
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/](https://pandas.pydata.org/pandas-docs/stable/user_guide/)  
df['mask'][j:j+n\_consecutive] = 1.0

```
%matplotlib widget  
fig, ax = plt.subplots(figsize=(8,4))  
# ax.plot(df['mask1'])  
ax.plot(df['mask'])  
ax.plot(df['Adiff'])  
ax.plot(df['A'])
```



## 75.21 generate datasets

```
def random_walk_with_outliers2(origin, steps_dist, outlier_dist, n_steps, perc_outliers=0.0, out  
    '''  
        Function for generating a random time series based on random walk.  
        It adds a specified percentage of outliers by multiplying the random walk step by a scalar  
  
    Parameters  
    -----  
    origin : int  
        The starting point of the series  
    steps_dist : list of int or float  
        step distribution  
    outlier_dist : list of int or float  
        outlier distribution  
    n_steps : int  
        Length of the series  
    perc_outliers : float
```

```

    Percentage of outliers to introduce to the series [0.0-1.0]
outlier_mult : float
    Scalar by which to multiply the RW increment to create an outlier
seed : int
    Random seed

Returns
-----
rw : np.ndarray
    The generated random walk series with outliers
indices : np.ndarray
    The indices of the introduced outliers
...
assert (perc_outliers >= 0.0) & (perc_outliers <= 1.0)

#set seed for reproducibility
np.random.seed(seed)

# possible steps
# steps = [-1, 1]

# simulate steps
steps = np.random.choice(a=steps_dist, size=n_steps-1)
rw = np.append(origin, steps).cumsum(0)

# add outliers
n_outliers = int(np.round(perc_outliers * n_steps, 0))
indices = np.random.randint(0, len(rw), n_outliers)
outlier_jumps = np.random.choice(a=outlier_dist, size=n_outliers)
# rw[indices] = rw[indices] + steps[indices + 1] * outlier_mult
rw[indices] = rw[indices] + outlier_jumps

return rw, indices

rw39test, _ = random_walk_with_outliers2(origin=0,
                                             steps_dist=np.random.normal(size=1000),
                                             outlier_dist=10*np.random.normal(loc=5.0, size=100),
                                             n_steps=n_steps,
                                             perc_outliers=0.0002,
                                             outlier_mult=50,

```

```
seed=206)
```

```
df['B'] = rw39test
```

```
start = '2023-01-01 00:00:00'  
end = '2023-12-31 23:00:00'  
# date_range = pd.date_range(start, end, freq='5min')  
date_range = pd.date_range(start, end, freq='1D')  
n_steps = len(date_range)
```

```
# date_range = pd.date_range(start, periods=n_steps, freq='1min')  
df = pd.DataFrame({'date': date_range, 'A': rw39test}).set_index('date')
```

```
df
```

A	
date	
2023-01-01	0.000000
2023-01-02	-0.032027
2023-01-03	-0.586351
2023-01-04	-1.575972
2023-01-05	-2.726800
...	...
2023-12-27	6.651301
2023-12-28	6.415175
2023-12-29	7.603140
2023-12-30	8.668182
2023-12-31	8.472768

```
df['unix_time'] = df.index.timestamp()
```

```
AttributeError: 'DatetimeIndex' object has no attribute 'timestamp'
```

```
df
```

A	
date	
2023-01-01	0.000000
2023-01-02	-0.032027
2023-01-03	-0.586351
2023-01-04	-1.575972
2023-01-05	-2.726800
...	...
2023-12-27	6.651301
2023-12-28	6.415175
2023-12-29	7.603140
2023-12-30	8.668182
2023-12-31	8.472768

```

rw02, outlier_ind02 = random_walk_with_outliers(origin=0,
                                                n_steps=n_steps,
                                                perc_outliers=0.0001,
                                                outlier_mult=500,
                                                seed=2)

df.loc['2023-01-02 23:00:00':'2023-01-03 03:00:00', 'E'] = np.nan

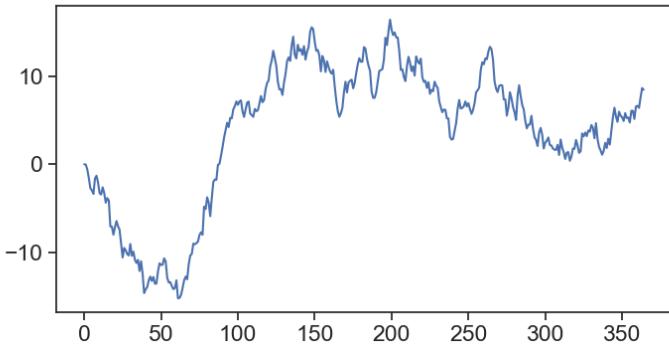
df.iloc[np.random.randint(0, high=len(df), size=600)]['E'] = np.nan

```

/var/folders/kv/9cqw3y\_s6c75xmgqm9n0t5d40000gn/T/ipykernel\_9222/2255542716.py:1: SettingWithCopyWarning  
 A value is trying to be set on a copy of a slice from a DataFrame.  
 Try using .loc[row\_indexer,col\_indexer] = value instead

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/](https://pandas.pydata.org/pandas-docs/stable/user_guide/)  
`df.iloc[np.random.randint(0, high=len(df), size=600)]['E'] = np.nan`

```
%matplotlib widget
fig, ax = plt.subplots(figsize=(8,4))
# ax.plot(df['B'])
ax.plot(rw39test)
```



```

df['date'] = df.index.strftime('%d%m%Y')
df['time'] = df.index.strftime('%H:%M:%S')

df['unix time (s)'] = df.index.view('int64') / 1e9
df

```

	A	date	unix time (s)
date			
2023-01-01	0.000000	2023-01-01	1.672531e+09
2023-01-02	-0.032027	2023-01-02	1.672618e+09
2023-01-03	-0.586351	2023-01-03	1.672704e+09
2023-01-04	-1.575972	2023-01-04	1.672790e+09
2023-01-05	-2.726800	2023-01-05	1.672877e+09
...	...	...	...
2023-12-27	6.651301	2023-12-27	1.703635e+09
2023-12-28	6.415175	2023-12-28	1.703722e+09
2023-12-29	7.603140	2023-12-29	1.703808e+09
2023-12-30	8.668182	2023-12-30	1.703894e+09
2023-12-31	8.472768	2023-12-31	1.703981e+09

```

df.drop(columns=['date'], inplace=True)
df

```

	A	unix time (s)
date		
2023-01-01	0.000000	1.672531e+09
2023-01-02	-0.032027	1.672618e+09
2023-01-03	-0.586351	1.672704e+09
2023-01-04	-1.575972	1.672790e+09
2023-01-05	-2.726800	1.672877e+09
...	...	...
2023-12-27	6.651301	1.703635e+09
2023-12-28	6.415175	1.703722e+09
2023-12-29	7.603140	1.703808e+09
2023-12-30	8.668182	1.703894e+09
2023-12-31	8.472768	1.703981e+09

```
df.loc['2023-10-11 03:00:00', 'B'] = '-'
```

```
df.to_csv('cleaning3.csv', index=False, sep=' ')
```

```
df['A']
```

```
date
2023-01-01 00:00:00      0.000000
2023-01-01 01:00:00     -2.027536
2023-01-01 02:00:00     -2.690617
2023-01-01 03:00:00     -1.985990
2023-01-01 04:00:00     -2.290898
...
2023-12-31 19:00:00    -74.514645
2023-12-31 20:00:00    -74.738058
2023-12-31 21:00:00    -75.848425
2023-12-31 22:00:00    -77.272183
2023-12-31 23:00:00    -76.557400
Name: A, Length: 8760, dtype: float64
```

```
import datetime
import time

dt = datetime.datetime.now()
```

```
timestamp = time.mktime(dt.timetuple())
print(timestamp)
```

1705330442.0

```
timestamp
```

1705330442.0

```
dt
```

```
datetime.datetime(2024, 1, 15, 16, 54, 2, 420872)
```

```
dt.timetuple()
```

```
time.struct_time(tm_year=2024, tm_mon=1, tm_mday=15, tm_hour=16, tm_min=54, tm_sec=2, tm_wday=0)
```

```
pd.to_datetime(1705330442.0, unit='s')
```

```
Timestamp('2024-01-15 14:54:02')
```

```
(df.index.values).apply(lambda x: x.timetuple())
```

```
AttributeError: 'numpy.ndarray' object has no attribute 'apply'
```

```
df['date'] = df.index.strftime('%Y-%m-%d')
```

```
df['date'].apply(lambda x: x.timetuple())
```

```
AttributeError: 'str' object has no attribute 'timetuple'
```

```
df.index.to_pydatetime()
```

```
array([datetime.datetime(2023, 1, 1, 0, 0),
       datetime.datetime(2023, 1, 2, 0, 0),
       datetime.datetime(2023, 1, 3, 0, 0),
       datetime.datetime(2023, 1, 4, 0, 0),
       datetime.datetime(2023, 1, 5, 0, 0),
       datetime.datetime(2023, 1, 6, 0, 0),
       datetime.datetime(2023, 1, 7, 0, 0),
       datetime.datetime(2023, 1, 8, 0, 0),
       datetime.datetime(2023, 1, 9, 0, 0),
       datetime.datetime(2023, 1, 10, 0, 0),
       datetime.datetime(2023, 1, 11, 0, 0),
       datetime.datetime(2023, 1, 12, 0, 0),
       datetime.datetime(2023, 1, 13, 0, 0),
       datetime.datetime(2023, 1, 14, 0, 0),
       datetime.datetime(2023, 1, 15, 0, 0),
       datetime.datetime(2023, 1, 16, 0, 0),
       datetime.datetime(2023, 1, 17, 0, 0),
       datetime.datetime(2023, 1, 18, 0, 0),
       datetime.datetime(2023, 1, 19, 0, 0),
       datetime.datetime(2023, 1, 20, 0, 0),
       datetime.datetime(2023, 1, 21, 0, 0),
       datetime.datetime(2023, 1, 22, 0, 0),
       datetime.datetime(2023, 1, 23, 0, 0),
       datetime.datetime(2023, 1, 24, 0, 0),
       datetime.datetime(2023, 1, 25, 0, 0),
       datetime.datetime(2023, 1, 26, 0, 0),
       datetime.datetime(2023, 1, 27, 0, 0),
       datetime.datetime(2023, 1, 28, 0, 0),
       datetime.datetime(2023, 1, 29, 0, 0),
       datetime.datetime(2023, 1, 30, 0, 0),
       datetime.datetime(2023, 1, 31, 0, 0),
       datetime.datetime(2023, 2, 1, 0, 0),
       datetime.datetime(2023, 2, 2, 0, 0),
       datetime.datetime(2023, 2, 3, 0, 0),
       datetime.datetime(2023, 2, 4, 0, 0),
       datetime.datetime(2023, 2, 5, 0, 0),
       datetime.datetime(2023, 2, 6, 0, 0),
       datetime.datetime(2023, 2, 7, 0, 0),
       datetime.datetime(2023, 2, 8, 0, 0),
       datetime.datetime(2023, 2, 9, 0, 0),
       datetime.datetime(2023, 2, 10, 0, 0),
```

```
datetime.datetime(2023, 2, 11, 0, 0),  
datetime.datetime(2023, 2, 12, 0, 0),  
datetime.datetime(2023, 2, 13, 0, 0),  
datetime.datetime(2023, 2, 14, 0, 0),  
datetime.datetime(2023, 2, 15, 0, 0),  
datetime.datetime(2023, 2, 16, 0, 0),  
datetime.datetime(2023, 2, 17, 0, 0),  
datetime.datetime(2023, 2, 18, 0, 0),  
datetime.datetime(2023, 2, 19, 0, 0),  
datetime.datetime(2023, 2, 20, 0, 0),  
datetime.datetime(2023, 2, 21, 0, 0),  
datetime.datetime(2023, 2, 22, 0, 0),  
datetime.datetime(2023, 2, 23, 0, 0),  
datetime.datetime(2023, 2, 24, 0, 0),  
datetime.datetime(2023, 2, 25, 0, 0),  
datetime.datetime(2023, 2, 26, 0, 0),  
datetime.datetime(2023, 2, 27, 0, 0),  
datetime.datetime(2023, 2, 28, 0, 0),  
datetime.datetime(2023, 3, 1, 0, 0),  
datetime.datetime(2023, 3, 2, 0, 0),  
datetime.datetime(2023, 3, 3, 0, 0),  
datetime.datetime(2023, 3, 4, 0, 0),  
datetime.datetime(2023, 3, 5, 0, 0),  
datetime.datetime(2023, 3, 6, 0, 0),  
datetime.datetime(2023, 3, 7, 0, 0),  
datetime.datetime(2023, 3, 8, 0, 0),  
datetime.datetime(2023, 3, 9, 0, 0),  
datetime.datetime(2023, 3, 10, 0, 0),  
datetime.datetime(2023, 3, 11, 0, 0),  
datetime.datetime(2023, 3, 12, 0, 0),  
datetime.datetime(2023, 3, 13, 0, 0),  
datetime.datetime(2023, 3, 14, 0, 0),  
datetime.datetime(2023, 3, 15, 0, 0),  
datetime.datetime(2023, 3, 16, 0, 0),  
datetime.datetime(2023, 3, 17, 0, 0),  
datetime.datetime(2023, 3, 18, 0, 0),  
datetime.datetime(2023, 3, 19, 0, 0),  
datetime.datetime(2023, 3, 20, 0, 0),  
datetime.datetime(2023, 3, 21, 0, 0),  
datetime.datetime(2023, 3, 22, 0, 0),  
datetime.datetime(2023, 3, 23, 0, 0),
```

```
datetime.datetime(2023, 3, 24, 0, 0),  
datetime.datetime(2023, 3, 25, 0, 0),  
datetime.datetime(2023, 3, 26, 0, 0),  
datetime.datetime(2023, 3, 27, 0, 0),  
datetime.datetime(2023, 3, 28, 0, 0),  
datetime.datetime(2023, 3, 29, 0, 0),  
datetime.datetime(2023, 3, 30, 0, 0),  
datetime.datetime(2023, 3, 31, 0, 0),  
datetime.datetime(2023, 4, 1, 0, 0),  
datetime.datetime(2023, 4, 2, 0, 0),  
datetime.datetime(2023, 4, 3, 0, 0),  
datetime.datetime(2023, 4, 4, 0, 0),  
datetime.datetime(2023, 4, 5, 0, 0),  
datetime.datetime(2023, 4, 6, 0, 0),  
datetime.datetime(2023, 4, 7, 0, 0),  
datetime.datetime(2023, 4, 8, 0, 0),  
datetime.datetime(2023, 4, 9, 0, 0),  
datetime.datetime(2023, 4, 10, 0, 0),  
datetime.datetime(2023, 4, 11, 0, 0),  
datetime.datetime(2023, 4, 12, 0, 0),  
datetime.datetime(2023, 4, 13, 0, 0),  
datetime.datetime(2023, 4, 14, 0, 0),  
datetime.datetime(2023, 4, 15, 0, 0),  
datetime.datetime(2023, 4, 16, 0, 0),  
datetime.datetime(2023, 4, 17, 0, 0),  
datetime.datetime(2023, 4, 18, 0, 0),  
datetime.datetime(2023, 4, 19, 0, 0),  
datetime.datetime(2023, 4, 20, 0, 0),  
datetime.datetime(2023, 4, 21, 0, 0),  
datetime.datetime(2023, 4, 22, 0, 0),  
datetime.datetime(2023, 4, 23, 0, 0),  
datetime.datetime(2023, 4, 24, 0, 0),  
datetime.datetime(2023, 4, 25, 0, 0),  
datetime.datetime(2023, 4, 26, 0, 0),  
datetime.datetime(2023, 4, 27, 0, 0),  
datetime.datetime(2023, 4, 28, 0, 0),  
datetime.datetime(2023, 4, 29, 0, 0),  
datetime.datetime(2023, 4, 30, 0, 0),  
datetime.datetime(2023, 5, 1, 0, 0),  
datetime.datetime(2023, 5, 2, 0, 0),  
datetime.datetime(2023, 5, 3, 0, 0),
```

```
datetime.datetime(2023, 5, 4, 0, 0),  
datetime.datetime(2023, 5, 5, 0, 0),  
datetime.datetime(2023, 5, 6, 0, 0),  
datetime.datetime(2023, 5, 7, 0, 0),  
datetime.datetime(2023, 5, 8, 0, 0),  
datetime.datetime(2023, 5, 9, 0, 0),  
datetime.datetime(2023, 5, 10, 0, 0),  
datetime.datetime(2023, 5, 11, 0, 0),  
datetime.datetime(2023, 5, 12, 0, 0),  
datetime.datetime(2023, 5, 13, 0, 0),  
datetime.datetime(2023, 5, 14, 0, 0),  
datetime.datetime(2023, 5, 15, 0, 0),  
datetime.datetime(2023, 5, 16, 0, 0),  
datetime.datetime(2023, 5, 17, 0, 0),  
datetime.datetime(2023, 5, 18, 0, 0),  
datetime.datetime(2023, 5, 19, 0, 0),  
datetime.datetime(2023, 5, 20, 0, 0),  
datetime.datetime(2023, 5, 21, 0, 0),  
datetime.datetime(2023, 5, 22, 0, 0),  
datetime.datetime(2023, 5, 23, 0, 0),  
datetime.datetime(2023, 5, 24, 0, 0),  
datetime.datetime(2023, 5, 25, 0, 0),  
datetime.datetime(2023, 5, 26, 0, 0),  
datetime.datetime(2023, 5, 27, 0, 0),  
datetime.datetime(2023, 5, 28, 0, 0),  
datetime.datetime(2023, 5, 29, 0, 0),  
datetime.datetime(2023, 5, 30, 0, 0),  
datetime.datetime(2023, 5, 31, 0, 0),  
datetime.datetime(2023, 6, 1, 0, 0),  
datetime.datetime(2023, 6, 2, 0, 0),  
datetime.datetime(2023, 6, 3, 0, 0),  
datetime.datetime(2023, 6, 4, 0, 0),  
datetime.datetime(2023, 6, 5, 0, 0),  
datetime.datetime(2023, 6, 6, 0, 0),  
datetime.datetime(2023, 6, 7, 0, 0),  
datetime.datetime(2023, 6, 8, 0, 0),  
datetime.datetime(2023, 6, 9, 0, 0),  
datetime.datetime(2023, 6, 10, 0, 0),  
datetime.datetime(2023, 6, 11, 0, 0),  
datetime.datetime(2023, 6, 12, 0, 0),  
datetime.datetime(2023, 6, 13, 0, 0),
```

```
datetime.datetime(2023, 6, 14, 0, 0),  
datetime.datetime(2023, 6, 15, 0, 0),  
datetime.datetime(2023, 6, 16, 0, 0),  
datetime.datetime(2023, 6, 17, 0, 0),  
datetime.datetime(2023, 6, 18, 0, 0),  
datetime.datetime(2023, 6, 19, 0, 0),  
datetime.datetime(2023, 6, 20, 0, 0),  
datetime.datetime(2023, 6, 21, 0, 0),  
datetime.datetime(2023, 6, 22, 0, 0),  
datetime.datetime(2023, 6, 23, 0, 0),  
datetime.datetime(2023, 6, 24, 0, 0),  
datetime.datetime(2023, 6, 25, 0, 0),  
datetime.datetime(2023, 6, 26, 0, 0),  
datetime.datetime(2023, 6, 27, 0, 0),  
datetime.datetime(2023, 6, 28, 0, 0),  
datetime.datetime(2023, 6, 29, 0, 0),  
datetime.datetime(2023, 6, 30, 0, 0),  
datetime.datetime(2023, 7, 1, 0, 0),  
datetime.datetime(2023, 7, 2, 0, 0),  
datetime.datetime(2023, 7, 3, 0, 0),  
datetime.datetime(2023, 7, 4, 0, 0),  
datetime.datetime(2023, 7, 5, 0, 0),  
datetime.datetime(2023, 7, 6, 0, 0),  
datetime.datetime(2023, 7, 7, 0, 0),  
datetime.datetime(2023, 7, 8, 0, 0),  
datetime.datetime(2023, 7, 9, 0, 0),  
datetime.datetime(2023, 7, 10, 0, 0),  
datetime.datetime(2023, 7, 11, 0, 0),  
datetime.datetime(2023, 7, 12, 0, 0),  
datetime.datetime(2023, 7, 13, 0, 0),  
datetime.datetime(2023, 7, 14, 0, 0),  
datetime.datetime(2023, 7, 15, 0, 0),  
datetime.datetime(2023, 7, 16, 0, 0),  
datetime.datetime(2023, 7, 17, 0, 0),  
datetime.datetime(2023, 7, 18, 0, 0),  
datetime.datetime(2023, 7, 19, 0, 0),  
datetime.datetime(2023, 7, 20, 0, 0),  
datetime.datetime(2023, 7, 21, 0, 0),  
datetime.datetime(2023, 7, 22, 0, 0),  
datetime.datetime(2023, 7, 23, 0, 0),  
datetime.datetime(2023, 7, 24, 0, 0),
```

```
datetime.datetime(2023, 7, 25, 0, 0),  
datetime.datetime(2023, 7, 26, 0, 0),  
datetime.datetime(2023, 7, 27, 0, 0),  
datetime.datetime(2023, 7, 28, 0, 0),  
datetime.datetime(2023, 7, 29, 0, 0),  
datetime.datetime(2023, 7, 30, 0, 0),  
datetime.datetime(2023, 7, 31, 0, 0),  
datetime.datetime(2023, 8, 1, 0, 0),  
datetime.datetime(2023, 8, 2, 0, 0),  
datetime.datetime(2023, 8, 3, 0, 0),  
datetime.datetime(2023, 8, 4, 0, 0),  
datetime.datetime(2023, 8, 5, 0, 0),  
datetime.datetime(2023, 8, 6, 0, 0),  
datetime.datetime(2023, 8, 7, 0, 0),  
datetime.datetime(2023, 8, 8, 0, 0),  
datetime.datetime(2023, 8, 9, 0, 0),  
datetime.datetime(2023, 8, 10, 0, 0),  
datetime.datetime(2023, 8, 11, 0, 0),  
datetime.datetime(2023, 8, 12, 0, 0),  
datetime.datetime(2023, 8, 13, 0, 0),  
datetime.datetime(2023, 8, 14, 0, 0),  
datetime.datetime(2023, 8, 15, 0, 0),  
datetime.datetime(2023, 8, 16, 0, 0),  
datetime.datetime(2023, 8, 17, 0, 0),  
datetime.datetime(2023, 8, 18, 0, 0),  
datetime.datetime(2023, 8, 19, 0, 0),  
datetime.datetime(2023, 8, 20, 0, 0),  
datetime.datetime(2023, 8, 21, 0, 0),  
datetime.datetime(2023, 8, 22, 0, 0),  
datetime.datetime(2023, 8, 23, 0, 0),  
datetime.datetime(2023, 8, 24, 0, 0),  
datetime.datetime(2023, 8, 25, 0, 0),  
datetime.datetime(2023, 8, 26, 0, 0),  
datetime.datetime(2023, 8, 27, 0, 0),  
datetime.datetime(2023, 8, 28, 0, 0),  
datetime.datetime(2023, 8, 29, 0, 0),  
datetime.datetime(2023, 8, 30, 0, 0),  
datetime.datetime(2023, 8, 31, 0, 0),  
datetime.datetime(2023, 9, 1, 0, 0),  
datetime.datetime(2023, 9, 2, 0, 0),  
datetime.datetime(2023, 9, 3, 0, 0),
```

```
datetime.datetime(2023, 9, 4, 0, 0),  
datetime.datetime(2023, 9, 5, 0, 0),  
datetime.datetime(2023, 9, 6, 0, 0),  
datetime.datetime(2023, 9, 7, 0, 0),  
datetime.datetime(2023, 9, 8, 0, 0),  
datetime.datetime(2023, 9, 9, 0, 0),  
datetime.datetime(2023, 9, 10, 0, 0),  
datetime.datetime(2023, 9, 11, 0, 0),  
datetime.datetime(2023, 9, 12, 0, 0),  
datetime.datetime(2023, 9, 13, 0, 0),  
datetime.datetime(2023, 9, 14, 0, 0),  
datetime.datetime(2023, 9, 15, 0, 0),  
datetime.datetime(2023, 9, 16, 0, 0),  
datetime.datetime(2023, 9, 17, 0, 0),  
datetime.datetime(2023, 9, 18, 0, 0),  
datetime.datetime(2023, 9, 19, 0, 0),  
datetime.datetime(2023, 9, 20, 0, 0),  
datetime.datetime(2023, 9, 21, 0, 0),  
datetime.datetime(2023, 9, 22, 0, 0),  
datetime.datetime(2023, 9, 23, 0, 0),  
datetime.datetime(2023, 9, 24, 0, 0),  
datetime.datetime(2023, 9, 25, 0, 0),  
datetime.datetime(2023, 9, 26, 0, 0),  
datetime.datetime(2023, 9, 27, 0, 0),  
datetime.datetime(2023, 9, 28, 0, 0),  
datetime.datetime(2023, 9, 29, 0, 0),  
datetime.datetime(2023, 9, 30, 0, 0),  
datetime.datetime(2023, 10, 1, 0, 0),  
datetime.datetime(2023, 10, 2, 0, 0),  
datetime.datetime(2023, 10, 3, 0, 0),  
datetime.datetime(2023, 10, 4, 0, 0),  
datetime.datetime(2023, 10, 5, 0, 0),  
datetime.datetime(2023, 10, 6, 0, 0),  
datetime.datetime(2023, 10, 7, 0, 0),  
datetime.datetime(2023, 10, 8, 0, 0),  
datetime.datetime(2023, 10, 9, 0, 0),  
datetime.datetime(2023, 10, 10, 0, 0),  
datetime.datetime(2023, 10, 11, 0, 0),  
datetime.datetime(2023, 10, 12, 0, 0),  
datetime.datetime(2023, 10, 13, 0, 0),  
datetime.datetime(2023, 10, 14, 0, 0),
```

```
datetime.datetime(2023, 10, 15, 0, 0),  
datetime.datetime(2023, 10, 16, 0, 0),  
datetime.datetime(2023, 10, 17, 0, 0),  
datetime.datetime(2023, 10, 18, 0, 0),  
datetime.datetime(2023, 10, 19, 0, 0),  
datetime.datetime(2023, 10, 20, 0, 0),  
datetime.datetime(2023, 10, 21, 0, 0),  
datetime.datetime(2023, 10, 22, 0, 0),  
datetime.datetime(2023, 10, 23, 0, 0),  
datetime.datetime(2023, 10, 24, 0, 0),  
datetime.datetime(2023, 10, 25, 0, 0),  
datetime.datetime(2023, 10, 26, 0, 0),  
datetime.datetime(2023, 10, 27, 0, 0),  
datetime.datetime(2023, 10, 28, 0, 0),  
datetime.datetime(2023, 10, 29, 0, 0),  
datetime.datetime(2023, 10, 30, 0, 0),  
datetime.datetime(2023, 10, 31, 0, 0),  
datetime.datetime(2023, 11, 1, 0, 0),  
datetime.datetime(2023, 11, 2, 0, 0),  
datetime.datetime(2023, 11, 3, 0, 0),  
datetime.datetime(2023, 11, 4, 0, 0),  
datetime.datetime(2023, 11, 5, 0, 0),  
datetime.datetime(2023, 11, 6, 0, 0),  
datetime.datetime(2023, 11, 7, 0, 0),  
datetime.datetime(2023, 11, 8, 0, 0),  
datetime.datetime(2023, 11, 9, 0, 0),  
datetime.datetime(2023, 11, 10, 0, 0),  
datetime.datetime(2023, 11, 11, 0, 0),  
datetime.datetime(2023, 11, 12, 0, 0),  
datetime.datetime(2023, 11, 13, 0, 0),  
datetime.datetime(2023, 11, 14, 0, 0),  
datetime.datetime(2023, 11, 15, 0, 0),  
datetime.datetime(2023, 11, 16, 0, 0),  
datetime.datetime(2023, 11, 17, 0, 0),  
datetime.datetime(2023, 11, 18, 0, 0),  
datetime.datetime(2023, 11, 19, 0, 0),  
datetime.datetime(2023, 11, 20, 0, 0),  
datetime.datetime(2023, 11, 21, 0, 0),  
datetime.datetime(2023, 11, 22, 0, 0),  
datetime.datetime(2023, 11, 23, 0, 0),  
datetime.datetime(2023, 11, 24, 0, 0),
```

```
datetime.datetime(2023, 11, 25, 0, 0),  
datetime.datetime(2023, 11, 26, 0, 0),  
datetime.datetime(2023, 11, 27, 0, 0),  
datetime.datetime(2023, 11, 28, 0, 0),  
datetime.datetime(2023, 11, 29, 0, 0),  
datetime.datetime(2023, 11, 30, 0, 0),  
datetime.datetime(2023, 12, 1, 0, 0),  
datetime.datetime(2023, 12, 2, 0, 0),  
datetime.datetime(2023, 12, 3, 0, 0),  
datetime.datetime(2023, 12, 4, 0, 0),  
datetime.datetime(2023, 12, 5, 0, 0),  
datetime.datetime(2023, 12, 6, 0, 0),  
datetime.datetime(2023, 12, 7, 0, 0),  
datetime.datetime(2023, 12, 8, 0, 0),  
datetime.datetime(2023, 12, 9, 0, 0),  
datetime.datetime(2023, 12, 10, 0, 0),  
datetime.datetime(2023, 12, 11, 0, 0),  
datetime.datetime(2023, 12, 12, 0, 0),  
datetime.datetime(2023, 12, 13, 0, 0),  
datetime.datetime(2023, 12, 14, 0, 0),  
datetime.datetime(2023, 12, 15, 0, 0),  
datetime.datetime(2023, 12, 16, 0, 0),  
datetime.datetime(2023, 12, 17, 0, 0),  
datetime.datetime(2023, 12, 18, 0, 0),  
datetime.datetime(2023, 12, 19, 0, 0),  
datetime.datetime(2023, 12, 20, 0, 0),  
datetime.datetime(2023, 12, 21, 0, 0),  
datetime.datetime(2023, 12, 22, 0, 0),  
datetime.datetime(2023, 12, 23, 0, 0),  
datetime.datetime(2023, 12, 24, 0, 0),  
datetime.datetime(2023, 12, 25, 0, 0),  
datetime.datetime(2023, 12, 26, 0, 0),  
datetime.datetime(2023, 12, 27, 0, 0),  
datetime.datetime(2023, 12, 28, 0, 0),  
datetime.datetime(2023, 12, 29, 0, 0),  
datetime.datetime(2023, 12, 30, 0, 0),  
datetime.datetime(2023, 12, 31, 0, 0)], dtype=object)
```

```
unix = df.index.view('int64') / 1e9
```

```
unix[0]
```

```
1672531200.0
```

```
pd.to_datetime(unix, unit='s')
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
                 '2023-01-05', '2023-01-06', '2023-01-07', '2023-01-08',
                 '2023-01-09', '2023-01-10',
                 ...
                 '2023-12-22', '2023-12-23', '2023-12-24', '2023-12-25',
                 '2023-12-26', '2023-12-27', '2023-12-28', '2023-12-29',
                 '2023-12-30', '2023-12-31'],
                dtype='datetime64[ns]', length=365, freq=None)
```

# make your own website

1. [Install Quarto](#) on your machine.
2. [Install VS Code](#).
3. [Open a GitHub account](#). Choose a good username for you, this will be also the name of your website.
4. [Get acquainted](#) with basic markdown syntax. You will be writing in markdown, this is a good investment of your time.

The more you know about `git` and basic `command line` instructions, the easier all this will be. You don't need to be a jedi master in computers to make this work, this is not hard, I promise. Quarto has a great webpage with detailed explanations. Also, ChatGPT can help you if you're lucky.

## random tips

### extensions

Use icons on your website with [iconify](#) and [fontawesome](#). You can actually use a wide variety of extensions, [check them out](#).

### quarto.yml

You can configure whatever you need in the `_quarto.yml` file. I'll paste here my html formatting for reference.

```
format:  
html:  
  theme:  
    # see all available themes https://bootswatch.com  
    - flatly           # chose whatever theme you find suitable
```

```
- custom.scss      # customize how your website looks
  fontsize: 1.2em    # self explanatory
  # choose a nice highlight style for the code
  highlight-style: breezedark # monokai # breezedark # espresso
  include-in-header:
    - includes.html      # you might need to use css configurations in all your pages
  code-line-numbers: true     # turn on line numbering
  code-tools:
    # if you defined repo-url, this will link your website to it.
    # repo-url: https://github.com/github_username/repository_name/
    source: repo # https://quarto.org/docs/output-formats/html-code.html#code-tools
    callout-icon: false
    fig-align: center      # center images as default
    # the default MathJax rendering option yields ugly results, use katex
    html-math-method: kate
```

## configure your notebook with a suitable header

You could start your jupyter notebook with a markdown cell with this header

```
# this jupyter notebook title
```

but in case you need a lot of control over the details, consider using:

```
---
title: "this jupyter notebook title"
execute:
  # echo: false # chose this if you don't want to see the code at all, just the output
  freeze: auto # re-render only when source changes, VERY useful
format:
  html:
    code-fold: true          # hide code blocks, show them upon click
    code-summary: "Show the code" # rename button to show code block
---
```

## **obvious statement**

This very website is a “Quarto website” project hosted on github. Click on “Code” on the top of the page to go to the github repository. Then copy whatever you want, it’s all open for everyone to see.