

Surface Hydrology

Yair Mau

Table of contents

| | |
|--|-----------|
| About | 7 |
| Syllabus | 7 |
| Course description | 7 |
| Course aims | 7 |
| Learning outcomes | 8 |
| Books and other sources | 8 |
| Course evaluation | 8 |
| | |
| I Introduction | 9 |
| | |
| 1 Water Cycle: Fluxes and Storage | 10 |
| 1.1 How much water is there? Where? | 10 |
| 1.2 The natural water cycle (2019) | 13 |
| 1.3 The new water cycle (2022) | 14 |
| 1.4 Global water distribution | 14 |
| 1.5 Energy drives the hydrologic cycle | 15 |
| 1.6 Components of the water cycle | 15 |
| 1.6.1 Water storage in oceans | 15 |
| 1.6.2 Evaporation / Sublimation | 15 |
| 1.6.3 Evapotranspiration | 20 |
| 1.6.4 Water storage in the atmosphere | 20 |
| 1.6.5 Condensation | 21 |
| 1.6.6 Precipitation | 21 |
| 1.6.7 Water storage in ice and snow | 23 |
| 1.6.8 Snowmelt runoff to streams | 25 |
| 1.6.9 Surface runoff | 25 |
| 1.6.10 Streamflow | 26 |
| 1.6.11 Lakes and rivers | 28 |
| 1.6.12 Infiltration | 31 |
| 1.6.13 Groundwater storage | 32 |
| 1.6.14 Groundwater flow and discharge | 36 |
| 1.6.15 Spring | 39 |

| | |
|--|-----------|
| 2 Exercises | 41 |
| 2.1 download the data | 41 |
| 2.2 import packages | 41 |
| 2.3 import data with pandas | 41 |
| 2.4 rename columns | 42 |
| 2.5 a first plot! | 42 |
| 2.6 how to deal with dates | 43 |
| 2.6.1 date as dataframe index | 44 |
| 2.7 plot again, now with dates | 44 |
| 2.8 we're getting there! the graph could look better | 45 |
| 2.9 make the following figure | 46 |
| 2.10 make another figure | 48 |
| 2.11 one last figure for today | 49 |
| 2.12 homework | 51 |
| 2.12.1 graph 1 | 51 |
| 2.12.2 graph 2 | 52 |
| 2.12.3 graph 3 | 52 |
| II Precipitation | 54 |
| 3 Intra-annual variability of precipitation | 56 |
| 3.1 hydrological year | 58 |
| 3.2 Seasonality Index | 61 |
| 4 Exercises | 68 |
| 4.1 intra-annual variability | 68 |
| 5 Interannual variability of precipitation | 76 |
| 5.1 coefficient of variation | 83 |
| 5.2 climate normals | 84 |
| 5.3 running averages | 84 |
| 6 Exercises | 91 |
| 7 Return Period | 98 |
| 7.1 Bilbao, Spain | 98 |
| 7.2 Today | 99 |
| 7.3 August 1983 | 100 |
| 7.4 How often will such rainfall happen? | 100 |
| 7.5 Return Period | 111 |
| 7.6 Generalized extreme value distribution | 112 |
| 7.6.1 cdf from data | 115 |

| | |
|---|------------|
| 8 Exercises | 123 |
| 8.1 loading data and pre-processing | 123 |
| 8.2 Weibull plotting position | 129 |
| 8.3 fit | 136 |
| III Evapotranspiration | 137 |
| 9 Evapotranspiration | 138 |
| 9.0.1 Potential Evapotranspiration | 139 |
| 9.0.2 Reference-Crop Evapotranspiration | 140 |
| 9.1 Review of methods | 141 |
| 9.2 Thornthwaite | 143 |
| 9.3 Penman | 145 |
| 9.3.1 Psychrometric Constant | 146 |
| 9.3.2 Net Radiation | 147 |
| 9.3.3 Heat Flux Density to the Ground | 148 |
| 9.3.4 Vapor Pressure | 149 |
| 9.3.5 Wind Function | 151 |
| 10 Meaning of “potential” evapotranspiration | 153 |
| 10.0.1 Crop Coefficient | 154 |
| 10.1 Pitfalls | 156 |
| 11 Exercises | 157 |
| 11.1 Download data from the IMS | 157 |
| 11.2 Install and import relevant packages | 158 |
| 11.3 import 10-minute data | 158 |
| 11.4 import radiation data | 159 |
| 11.5 import pan evaporation data | 161 |
| 11.6 calculate penman | 162 |
| 11.7 Thornthwaite | 163 |
| 11.8 Data from NOAA | 167 |
| IV Infiltration | 177 |
| 12 Infiltration | 179 |
| 12.1 Definitions | 179 |
| 12.2 Darcy | 188 |
| 12.3 Richards | 188 |
| 12.3.1 short times | 189 |
| 12.3.2 long times | 189 |
| 12.3.3 Rainfall infiltration | 191 |

| | |
|---|------------|
| 12.4 Horton equation | 193 |
| 12.5 Green & Ampt | 195 |
| 12.6 Best Fit, Least Squares Method | 198 |
| 13 Exercises | 199 |
| 13.1 Tasks | 199 |
| 13.2 Horton's equation | 202 |
| 13.3 What does fit really mean? | 206 |
| 13.3.1 learning rate | 213 |
| 13.3.2 local and global minima | 216 |
| 13.4 Green & Ampt | 216 |
| 13.5 Homework | 221 |
| V Streamflow | 222 |
| 14 Streamflow | 223 |
| 14.1 Watershed - | 224 |
| 14.2 base flow separation | 225 |
| Base flow | 226 |
| Event flow | 226 |
| Total flow | 226 |
| Attention! | 226 |
| 14.3 Urbana, IL | 227 |
| 14.3.1 hyetograph, hydrograph | 228 |
| 14.3.2 notation | 228 |
| 14.3.3 base flow separation | 229 |
| 14.3.4 effective precipitation = effective discharge | 229 |
| 14.3.5 time lags | 232 |
| 15 Exercises | 234 |
| 16 Unit Hydrograph | 243 |
| 16.1 Linear reservoir model | 243 |
| 16.2 Rainfall-Runoff Models | 247 |
| 16.2.1 The Rational Method | 247 |
| 16.2.2 The Soil Conservation Service Curve-Number Method (SCS-CN) | 248 |
| VI summing up | 255 |
| 17 Budyko framework | 256 |
| 17.1 Water and surface energy balances | 256 |
| 17.2 Assumptions | 256 |

| | |
|--|------------|
| 17.3 Question | 257 |
| 17.4 Limits | 257 |
| 17.4.1 Summary: | 257 |
| 17.5 Hypotheses for why dryness index controls so much the partitioning of P into ET and Q | 262 |
| 17.6 Hypotheses for deviations from Budyko curve | 263 |
| 17.6.1 Reasons for falling off the Budyko Curve | 263 |
| 17.6.2 Critique | 263 |
| 18 Spatial distribution - lecture | 264 |
| 18.1 The problem | 264 |
| 18.2 Thiessen method [Voronoi diagram] | 266 |
| 18.3 Inverse distance method | 268 |
| 18.4 Isohyetal method | 269 |
| 18.5 How it is actually done | 269 |
| VII Assignments | 274 |
| Presentation | 275 |
| Evaluation | 275 |
| AI Policy | 275 |
| 19 Assignment - first graphs | 277 |
| 19.1 graph 1 | 277 |
| 19.2 graph 2 | 279 |
| 19.3 graph 3 | 280 |
| 20 Assignment - return period | 282 |
| 21 Assignment - ET | 283 |
| VIII Appendix | 284 |
| 22 Gain full control of date formatting | 286 |
| 23 Plotting guidelines | 295 |
| 23.1 increase fontsize to legible sizes | 295 |
| 23.2 choose colors wisely | 297 |
| 23.3 the best legend is no legend | 302 |
| 24 Summary | 307 |
| References | 308 |

About

Welcome to Surface Hydrology (71630) 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.

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

Syllabus

Course description

This is an introductory course in Surface Hydrology, dealing with some of the major processes in the hydrologic cycle: precipitation, evaporation and transpiration, infiltration, runoff generation and streamflow. The different topics will be treated using mathematical models and practical programming exercises.

Course aims

The course aims at giving the students a quantitative understanding of the main processes in the hydrologic cycle. We will characterize the hydrologic cycle and its fluxes through mass balance equations. The random nature of the various processes will be studied with statistics, time series analysis, return periods, extreme value distributions, etc. We will take a “hands-on approach”, where students will actively engage with the material by analysing data and writing models using Python.

Learning outcomes

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

- Identify the various components of hydrologic budget and their interdependency.
- Describe the various processes in hydrology (precipitation, infiltration, evaporation, etc) in a mathematical language.
- Write computer code to analyze the statistics of hydrologic fluxes, and construct models of hydrological systems.

Books and other sources

- Dingman, S. L. (2015). Physical hydrology (3rd edition). Waveland press.
- Ward, A. D., & Trimble, S. W. (2003). Environmental hydrology. CRC Press.
- Brutsaert, W. (2005). Hydrology: An Introduction. Cambridge University Press.

Course evaluation

There will be some small projects during the semester, all worth 50% of the grade. A final and larger project (50% of the grade) will be due at the end of the semester. All projects will be done in Python (on Jupyter Notebooks).

Part I

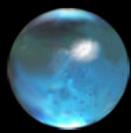
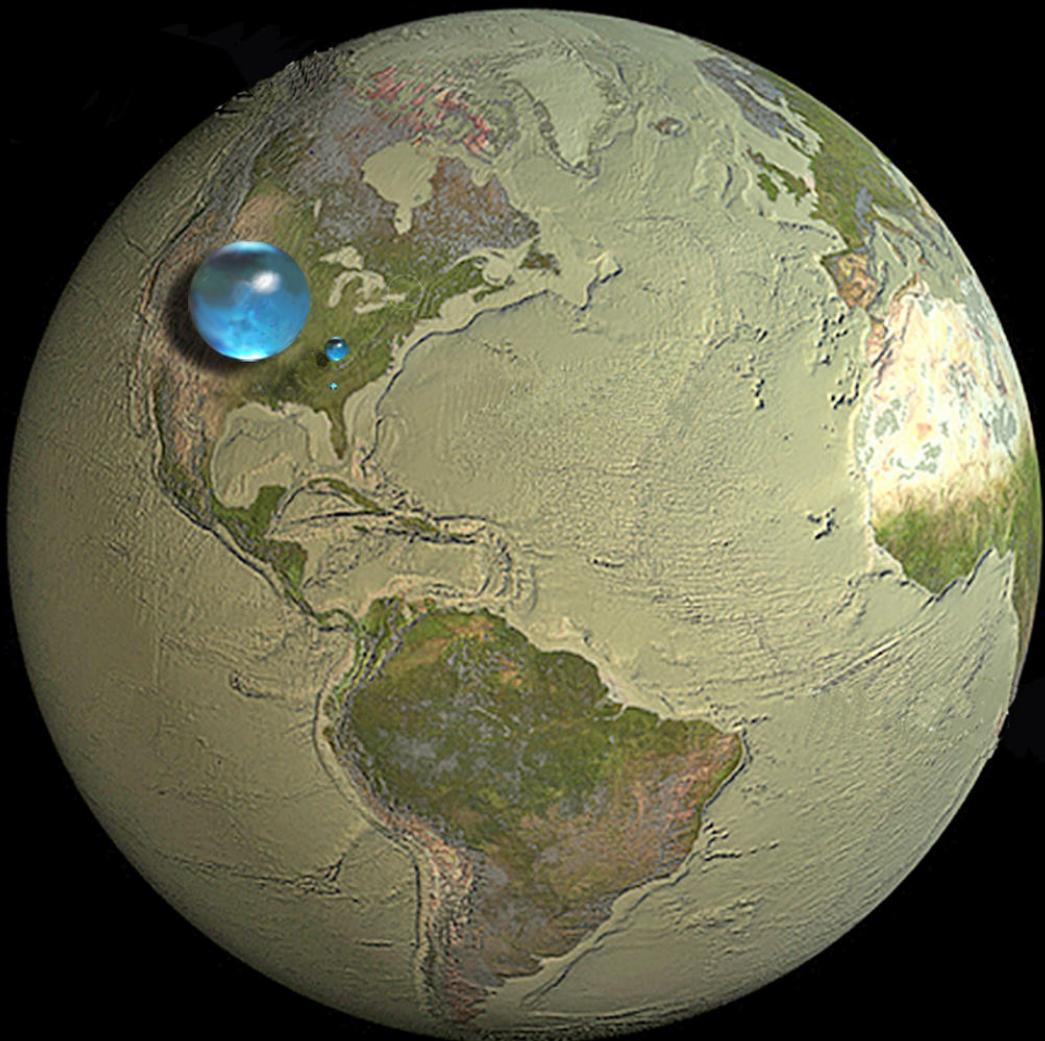
Introduction

1 Water Cycle: Fluxes and Storage

1.1 How much water is there? Where?

<https://youtu.be/2ObMyytxLz8>

The World's Water

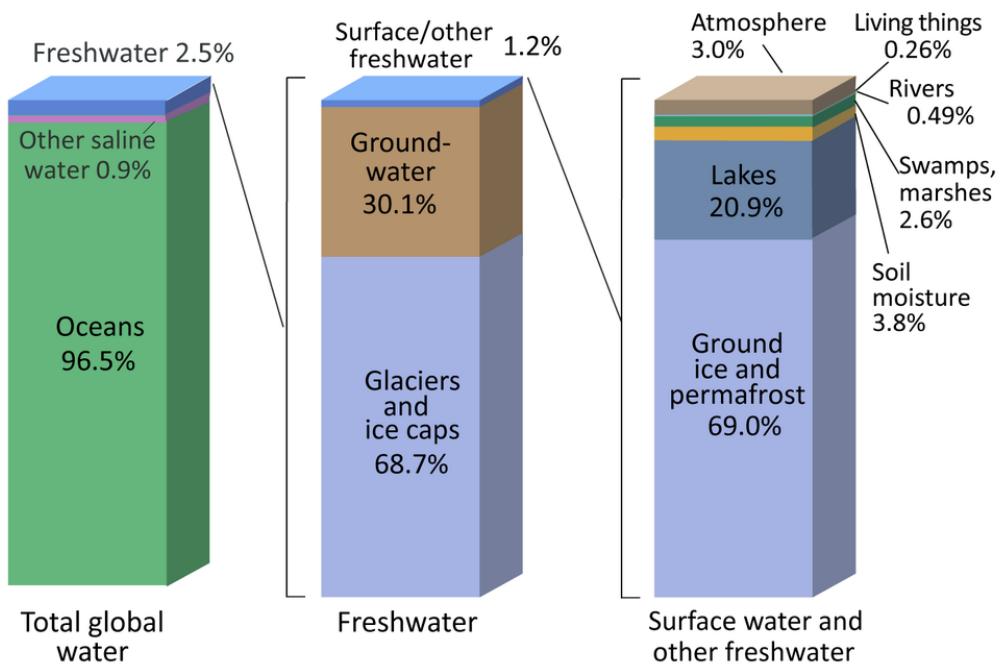


- All water on, in, and above the Earth
 - Liquid fresh water
 - Fresh-water lakes and rivers

Howard Perlman, USGS,
Jack Cook, Woods Hole Oceanographic Institution,
Adam Nieman
Data source: Igor Shiklomanov
<http://ga.water.usgs.gov/edu/earthhowmuch.html>

Figure 1.1: Source: Water Science School (2019c)

Where is Earth's Water?



Credit: U.S. Geological Survey, Water Science School. <https://www.usgs.gov/special-topic/water-science-school>
Data source: Igor Shiklomanov's chapter "World fresh water resources" in Peter H. Gleick (editor), 1993, Water in Crisis: A Guide to the World's Fresh Water Resources. (Numbers are rounded).

Figure 1.2: Source: Water Science School (2018)

1.2 The natural water cycle (2019)

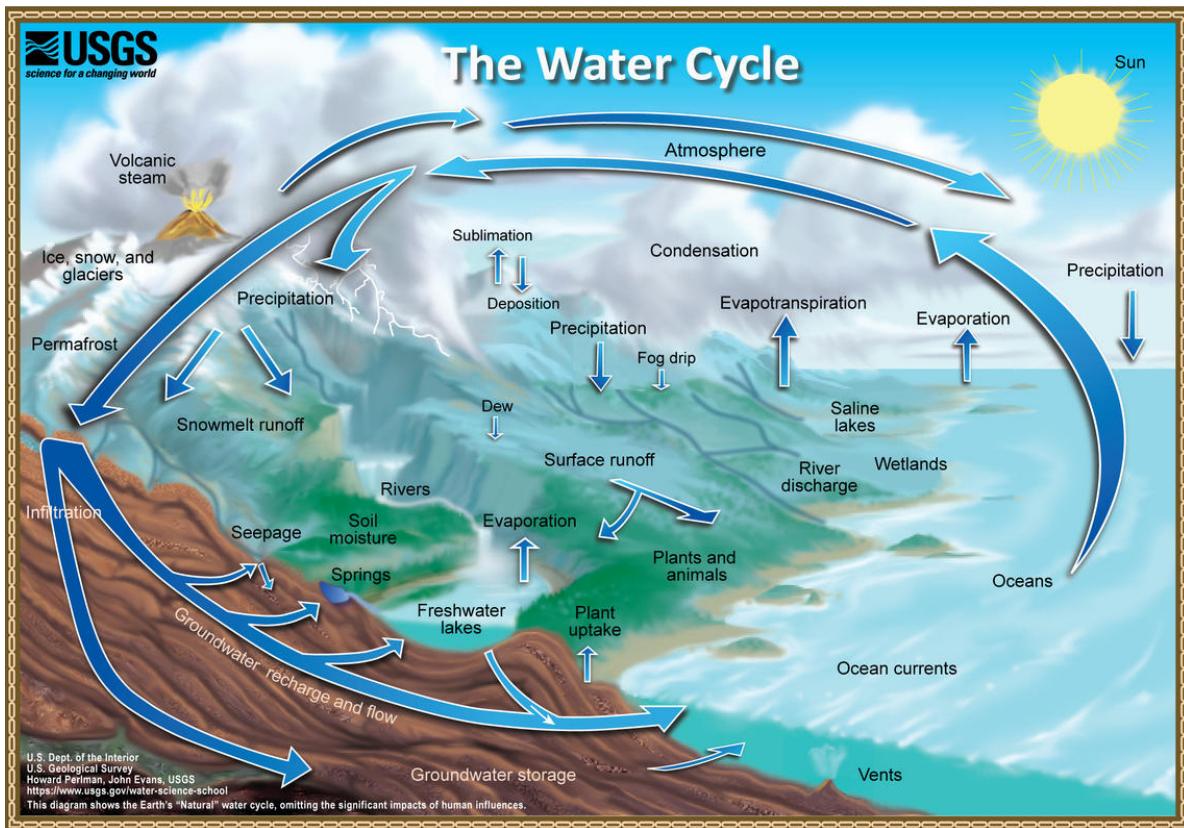
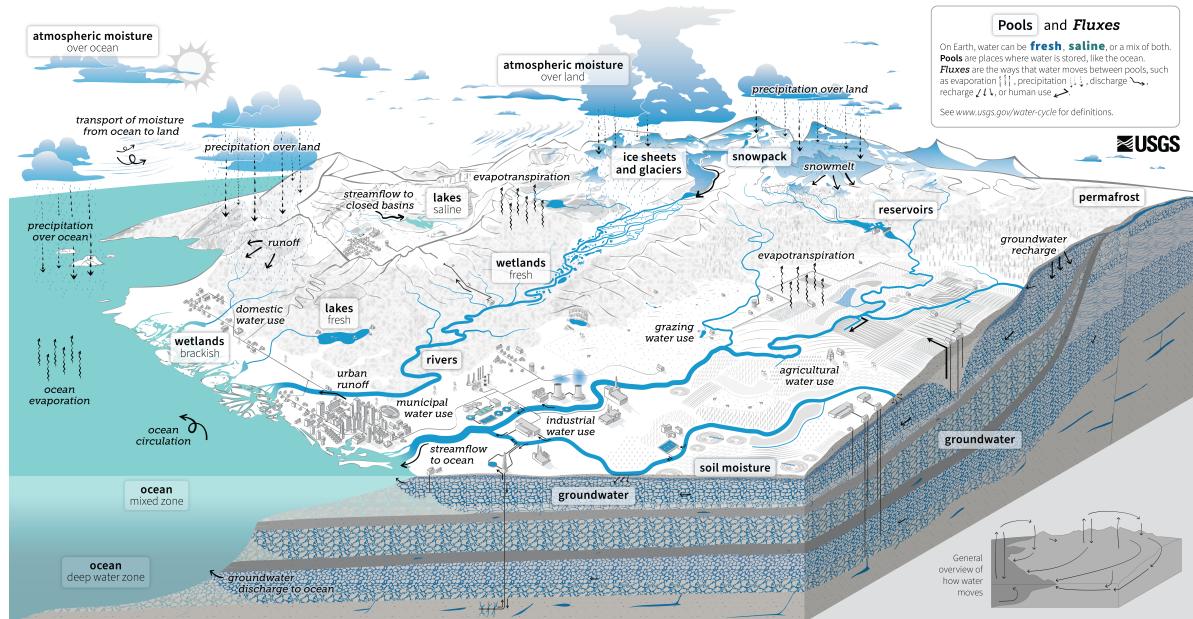


Figure 1.3: Source: Water Science School (2019g)

1.3 The new water cycle (2022)



The Water Cycle

The water cycle describes where water is on Earth and how it moves. Water is stored in the atmosphere, on the land surface, and below the ground. It can be a liquid, a solid, or a gas. Liquid water can be fresh, saline (salty), or a mix (brackish). Water moves between the places it is stored. Water moves at large scales and at very small scales. Water moves naturally and because of human actions. Human water use affects where water is stored, how it moves, and how clean it is.

Pools store water. 96% of all water is stored in **oceans** and is saline. On land, saline water is stored in **saline lakes**. Fresh water is stored in liquid form in **freshwater lakes**, **artificial reservoirs**, **rivers**, and **wetlands**. Water is frozen in freshwater in **ice sheets**, **glaciers**, and in **snowpack** at high elevations or near the Earth's poles. Water vapor is a gas and is stored as **atmospheric moisture** over the ocean and land. In the soil, frozen water is stored as **permafrost** and liquid water is stored as **soil moisture**. Deeper below ground, liquid water is stored as **groundwater** in aquifers, within cracks and pores in the rock.

Fluxes move water between pools. As it moves, water can change form between liquid, solid, and gas. **Circulation** mixes water in the oceans and transports water vapor in the atmosphere. Water moves between the atmosphere and the surface through **precipitation**, **evapotranspiration**, and **sublimation**. Water moves across the surface through **snowmelt**, **runoff**, and **streamflow**. Water moves into the ground through infiltration and **groundwater recharge**. Underground, groundwater flows within aquifers. It can return to the surface through natural **groundwater discharge** into rivers, the ocean, and from springs.

We alter the water cycle. We redirect rivers. We build dams to store water. We drain water from wetlands for development. We use water from rivers, lakes, reservoirs, and groundwater aquifers. We use them to supply irrigation and **agriculture**. We use it for **industrial** activities like thermoelectric power generation, mining, and aquaculture. The amount of water that is available depends on how much water is in each pool (water quantity). It also depends on when and how fast water moves (water timing), how much water we use (water use), and how clean the water is (water quality).

We affect **water quality**. In agricultural and urban areas, irrigation and precipitation wash fertilizers and pesticides into rivers and groundwater. Power plants and factories release heat and chemicals into rivers. Runoff carries chemicals, sediment, and sewage into rivers and lakes. Downstream from these sources, contaminated water can cause harmful algal blooms, spread diseases, and harm habitats. **Climate change** is affecting the water cycle. It is affecting water quality, quantity, timing, and use. It is causing ocean acidification, sea level rise, and more extreme weather. By understanding these impacts, we can work toward using water sustainably.

Figure 1.4: Source: Water Science School (2022)

Interactive chart: [Pools and fluxes in the water cycle](#)

1.4 Global water distribution

Table 1.1: Source: Water Science School (2018). (**Percents are rounded, so will not add to 100**)

| Water source | Volume (km ³) | % of freshwater | % of total water |
|--------------------------------------|---------------------------|-----------------|------------------|
| Oceans, Seas, & Bays | 1,338,000,000 | — | 96.54 |
| Ice caps, Glaciers, & Permanent Snow | 24,064,000 | 68.7 | 1.74 |
| | | | |

| Water source | Volume (km ³) | % of freshwater | % of total water |
|-------------------------|---------------------------|-----------------|------------------|
| Groundwater | 23,400,000 | — | 1.69 |
| Fresh | 10,530,000 | 30.1 | 0.76 |
| Saline | 12,870,000 | — | 0.93 |
| Soil Moisture | 16,500 | 0.05 | 0.001 |
| Ground Ice & Permafrost | 300,000 | 0.86 | 0.022 |
| Lakes | 176,400 | — | 0.013 |
| Fresh | 91,000 | 0.26 | 0.007 |
| Saline | 85,400 | — | 0.006 |
| Atmosphere | 12,900 | 0.04 | 0.001 |
| Swamp Water | 11,470 | 0.03 | 0.0008 |
| Rivers | 2,120 | 0.006 | 0.0002 |
| Biological Water | 1,120 | 0.003 | 0.0001 |

1.5 Energy drives the hydrologic cycle

From Margulis (2019)

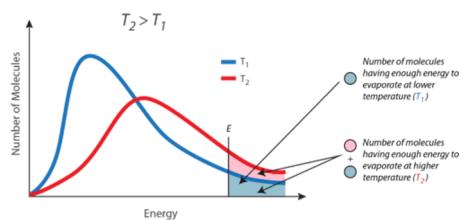
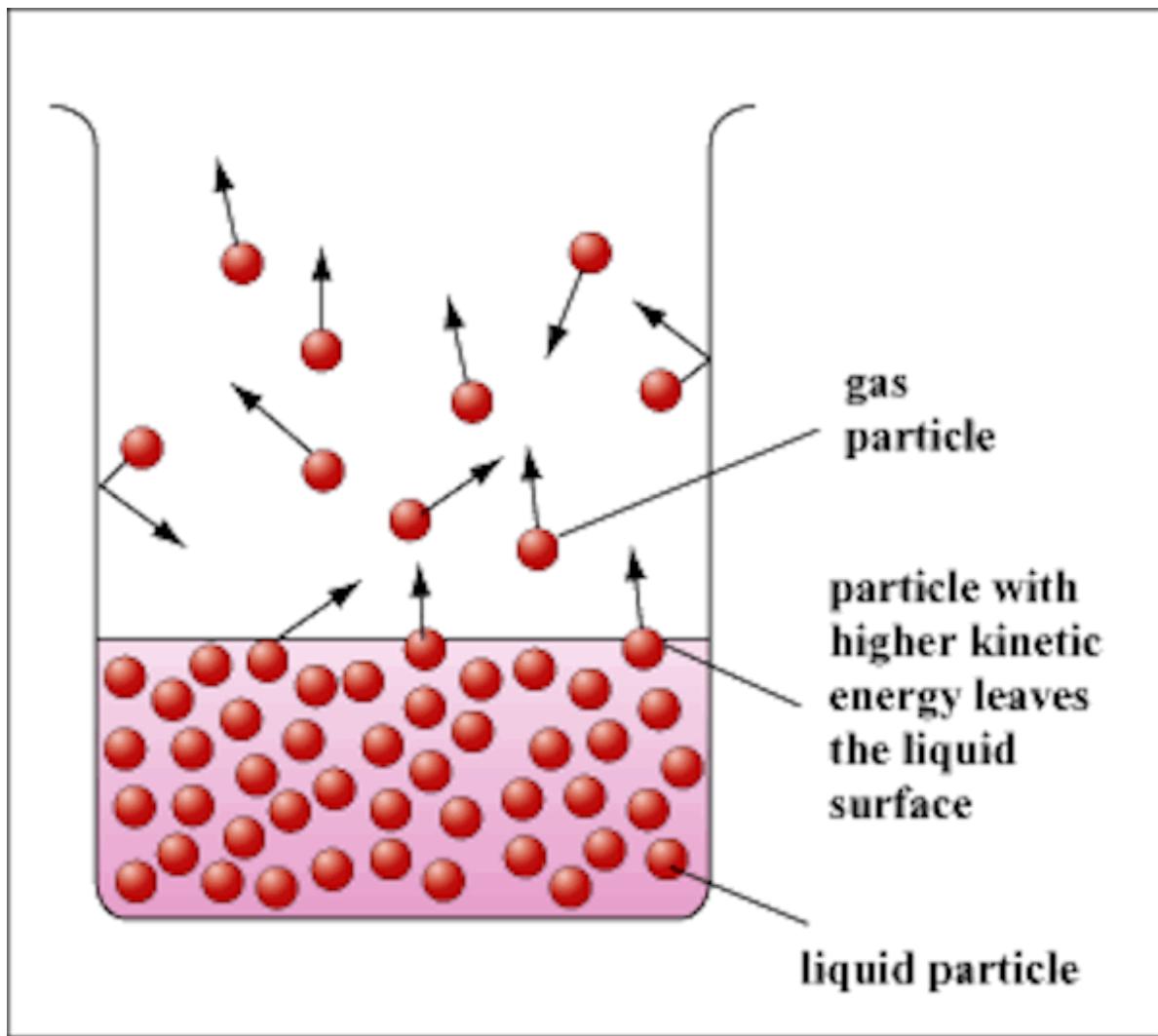
A key aspect of the hydrologic cycle is the fact that it is driven by energy inputs (primarily from the sun). At the global scale, the system is essentially closed with respect to water; negligible water is entering or leaving the system. In other words, there is no external forcing in terms of a water flux. Systems with no external forcing will generally eventually come to an equilibrium state. So what makes the hydrologic cycle so dynamic? The solar radiative energy input, which is external to the system, drives the hydrologic cycle. Averaged over the globe, 342 W m^{-2} of solar radiative energy is being continuously input to the system at the top of the atmosphere. This energy input must be dissipated, and this is done, to a large extent, via the hydrologic cycle. Due to this fact, the study of hydrology is not isolated to the study of water storage and movement, but also must often include study of energy storage and movements.

1.6 Components of the water cycle

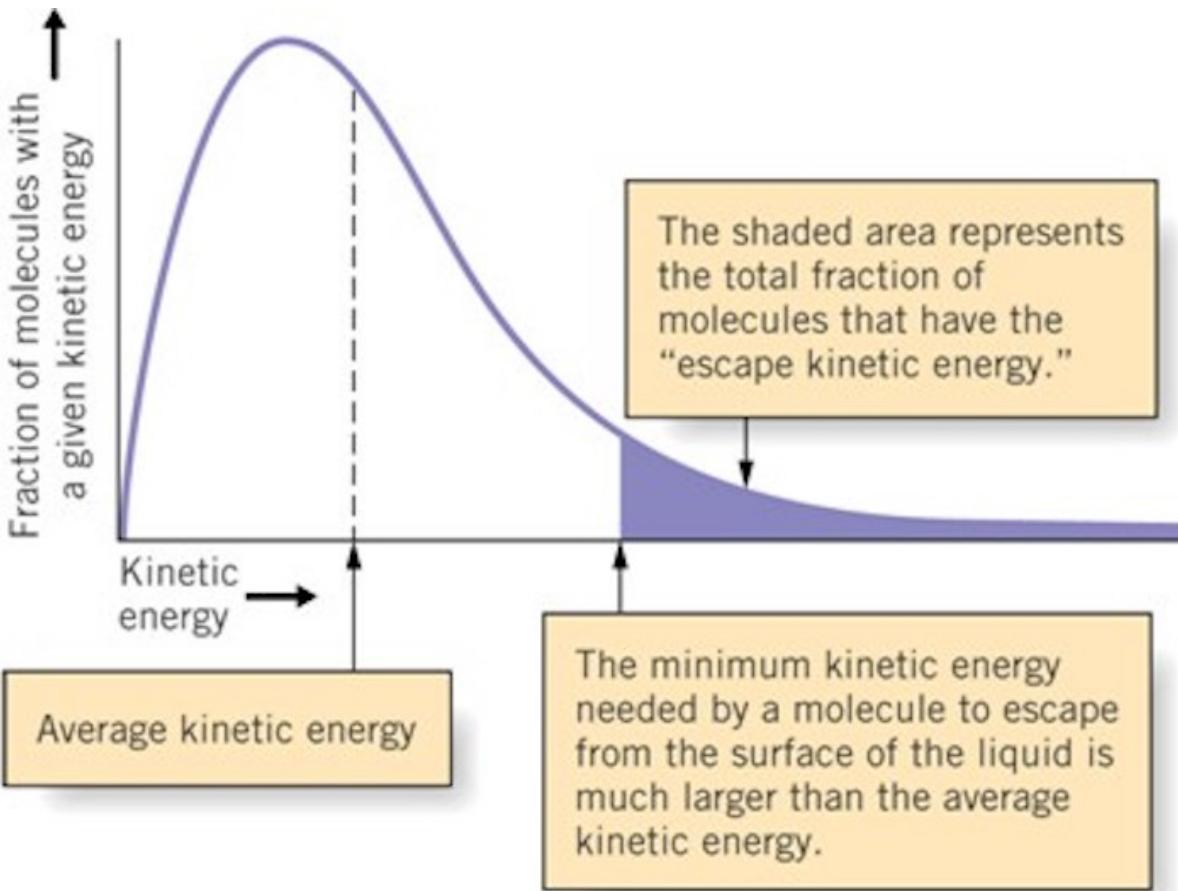
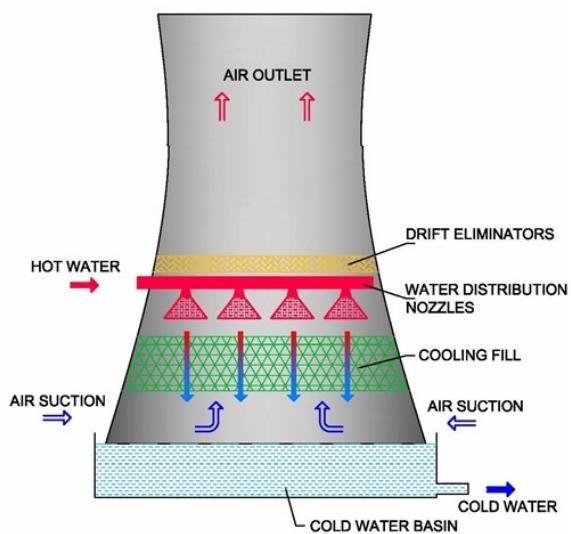
1.6.1 Water storage in oceans

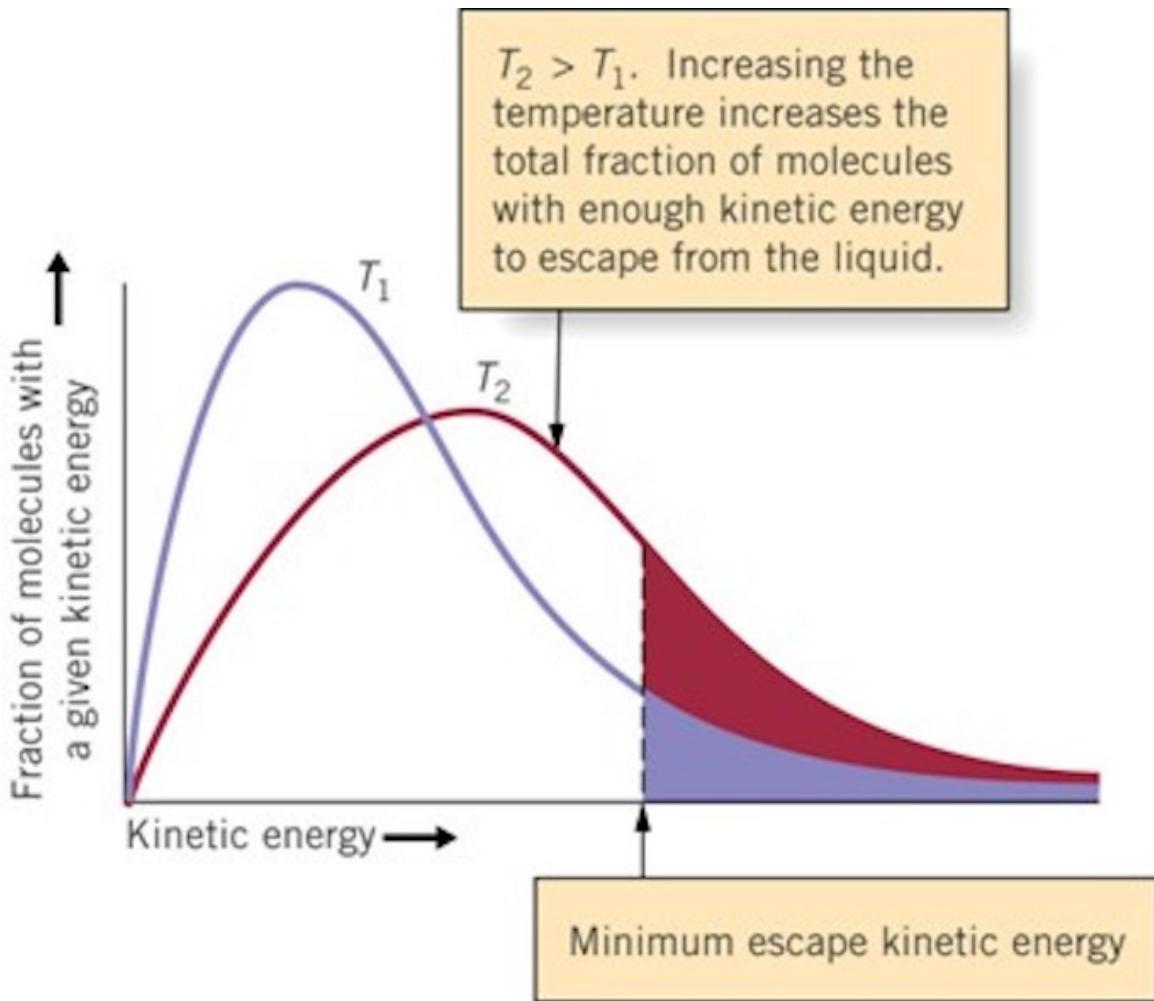
1.6.2 Evaporation / Sublimation

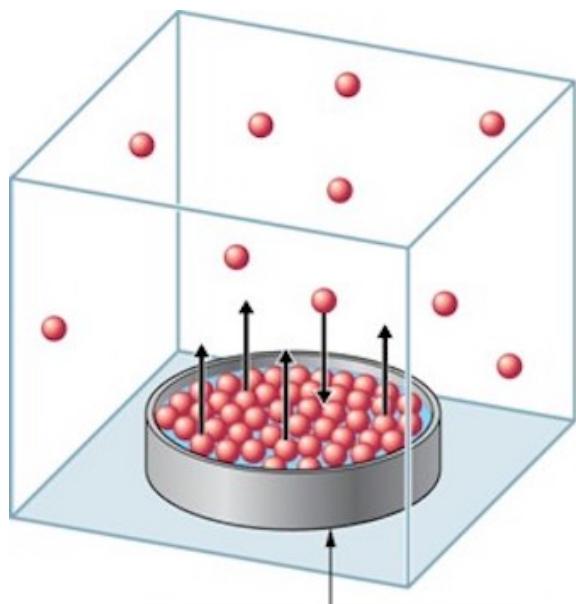
Evaporation → cooling



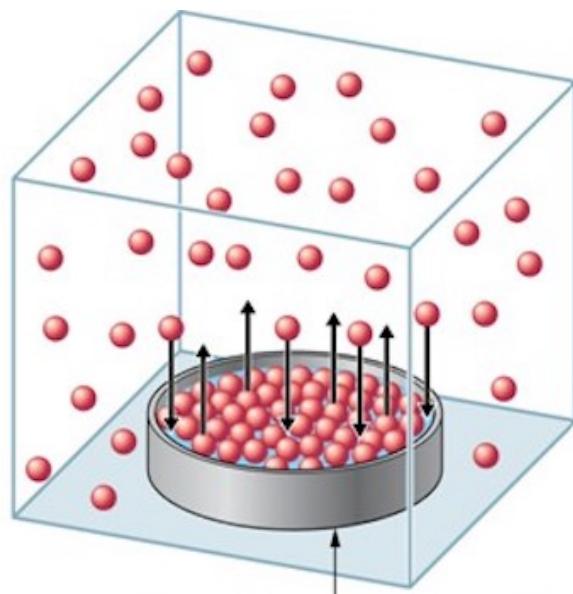
WET COOLING TOWER
NATURAL DRAFT







Before equilibrium:
Rate of evaporation is greater
than the rate of condensation.

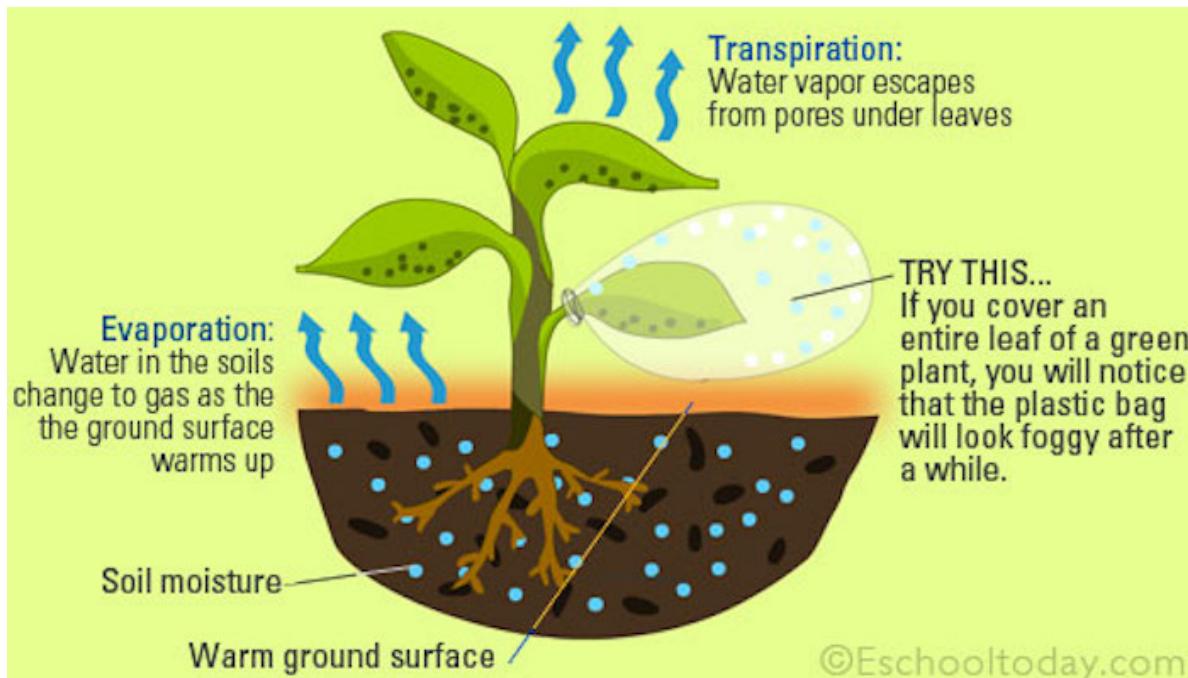


At equilibrium:
Rate of evaporation equals
the rate of condensation.

(a)

(b)

1.6.3 Evapotranspiration



1.6.4 Water storage in the atmosphere



Cumulonimbus cloud over Africa

Picture of cumulonimbus taken from the International Space Station, over western Africa near the Senegal-Mali border.

If all of the water in the atmosphere rained down at once, it would only cover the globe to a

depth of 2.5 centimeters.

$$\begin{array}{ll} \text{amount of water in the atmosphere} & V = 12\,900 \text{ km}^3 \\ \text{surface of Earth} & S = 4\pi R^2; \quad R = 6371 \text{ km} \\ & V = S \times h \\ \text{height} & h = \frac{V}{S} \simeq 2.5 \text{ cm} \end{array}$$

Try to calculate this yourself, and click on the button below to check how to do it.

```
# amount of water in the atmosphere
V = 12900 # km^3
# Earth's radius
R = 6371 # km
# surface of Earth = 4 pi R^2
S = 4 * 3.141592 * R**2
# Volume: V = S * h, therefore
# height
h = V / S # in km
h_cm = h * 1e5 # in cm
print(f"The height would be ~ {h_cm:.1f} cm")
```

The height would be ~ 2.5 cm

1.6.5 Condensation

1.6.6 Precipitation



Figure 1.5: Source: Water Science School (2019f)

Table 1.2: Source: Water Science School (2019e)

| | Intensity (cm/h) | Median diameter (mm) | Velocity of fall (m/s) | Drops s ⁻¹ m ⁻² |
|----------------|---------------------|----------------------------|---------------------------|---------------------------------------|
| Fog | 0.013 | 0.01 | 0.003 | 67,425,000 |
| Mist | 0.005 | 0.1 | 0.21 | 27,000 |
| Drizzle | 0.025 | 0.96 | 4.1 | 151 |
| Light rain | 0.10 | 1.24 | 4.8 | 280 |
| Moderate rain | 0.38 | 1.60 | 5.7 | 495 |
| Heavy rain | 1.52 | 2.05 | 6.7 | 495 |
| Excessive rain | 4.06 | 2.40 | 7.3 | 818 |
| Cloudburst | 10.2 | 2.85 | 7.9 | 1,220 |

1.6.7 Water storage in ice and snow

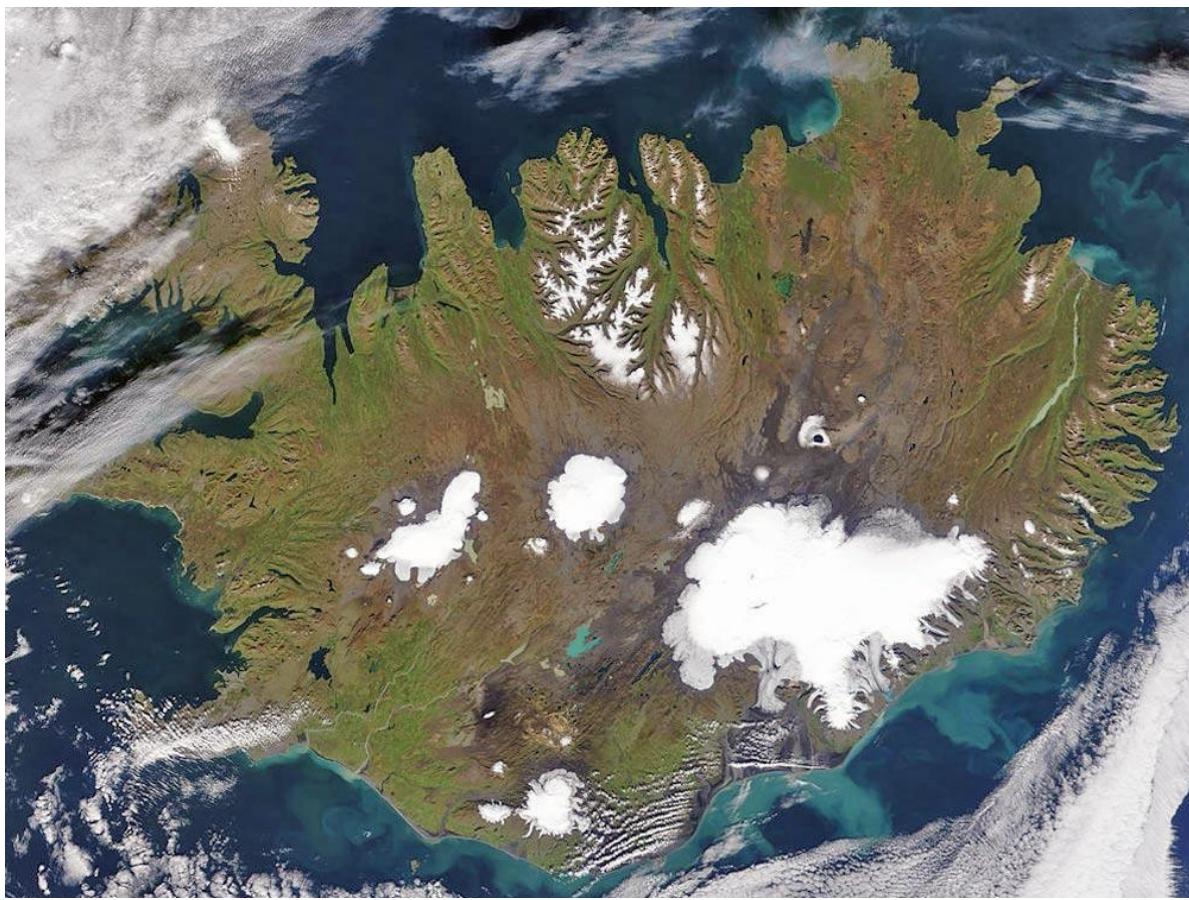


Figure 1.6: Source: Water Science School (2019d)

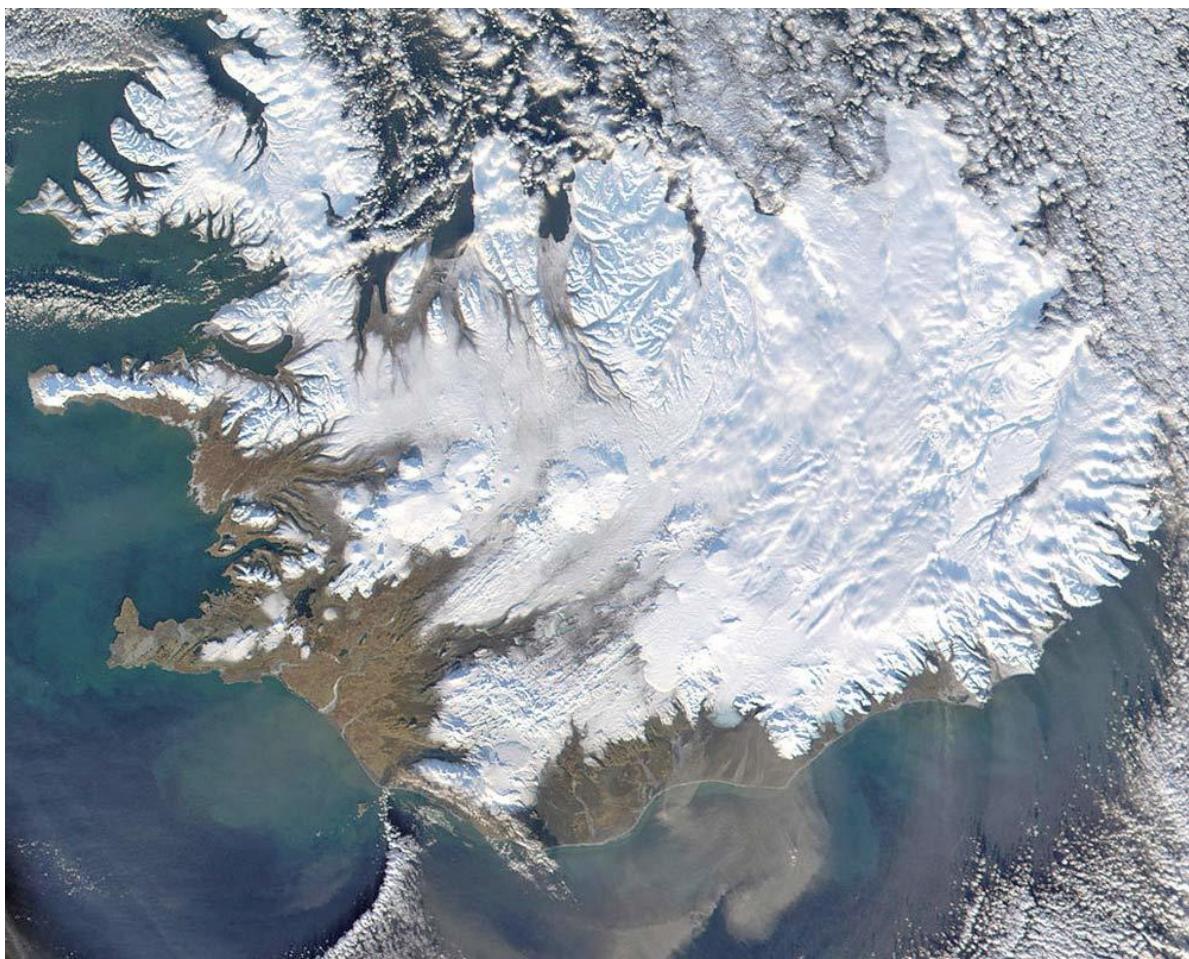


Figure 1.7: Source: Water Science School (2019d)

1.6.8 Snowmelt runoff to streams

1.6.9 Surface runoff



Figure 1.8: Source: (2020)



1.6.10 Streamflow



The Mississippi river basin is very large

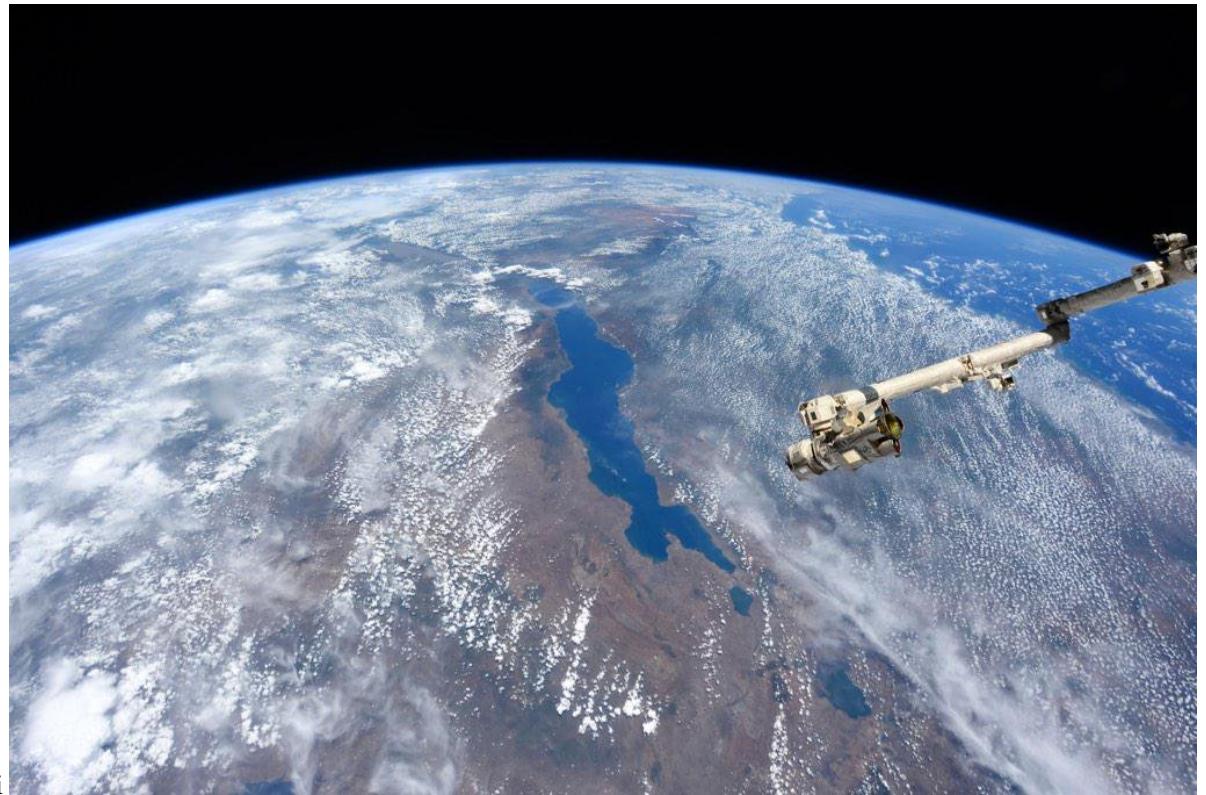


Figure 1.9: Source: Yair Mau

1.6.11 Lakes and rivers



Figure 1.10: Source: dreamstime (2022)



Lake Malawi



Figure 1.11: Source: Fiona Bruce (2015)

1.6.12 Infiltration



Figure 1.12: Source: Suma Groulx (2015)

1.6.13 Groundwater storage

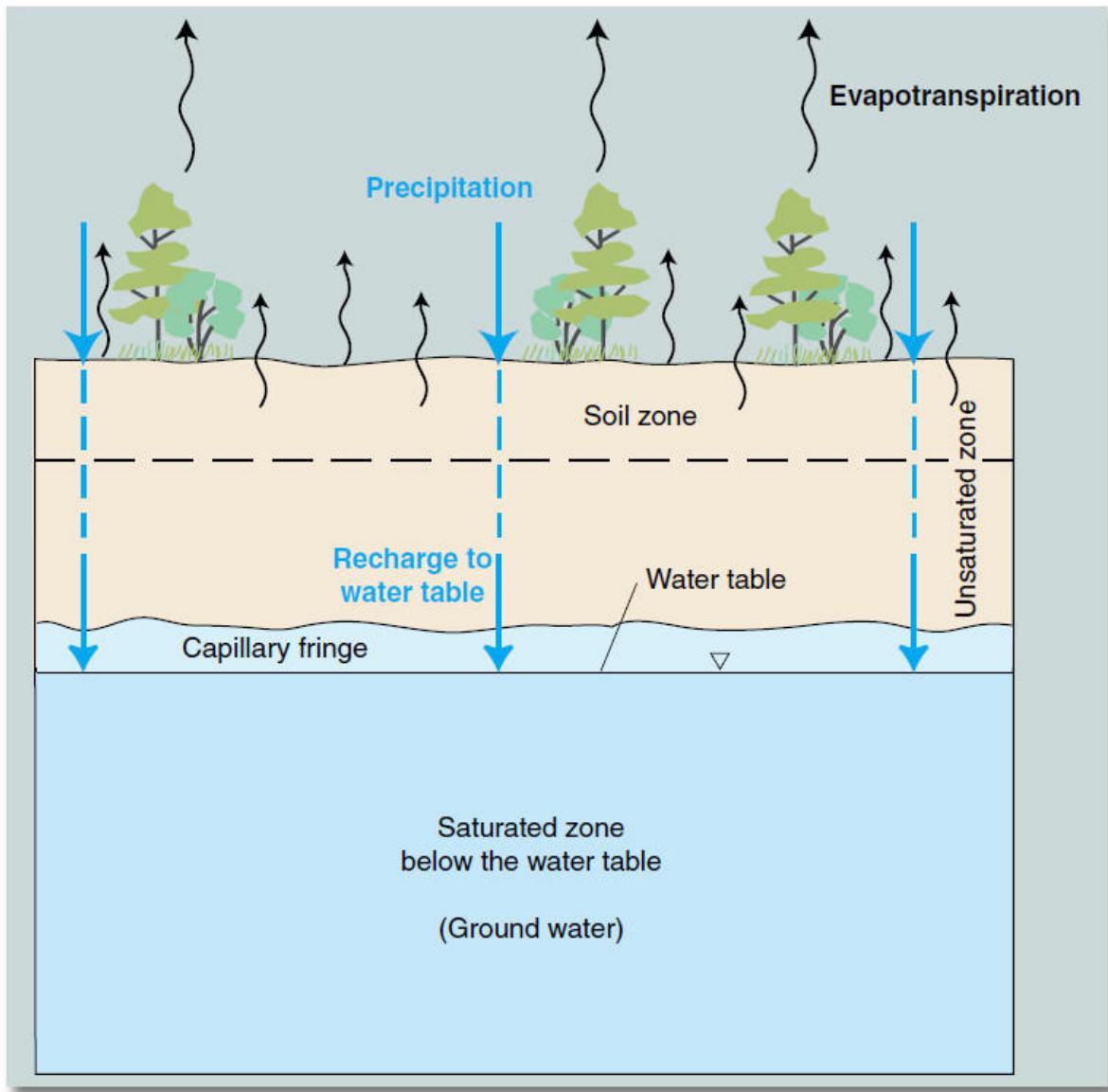


Figure 1.13: Source: Water Science School (2019b)



Figure 1.14: Source: Andrew Amelinckx (2015)

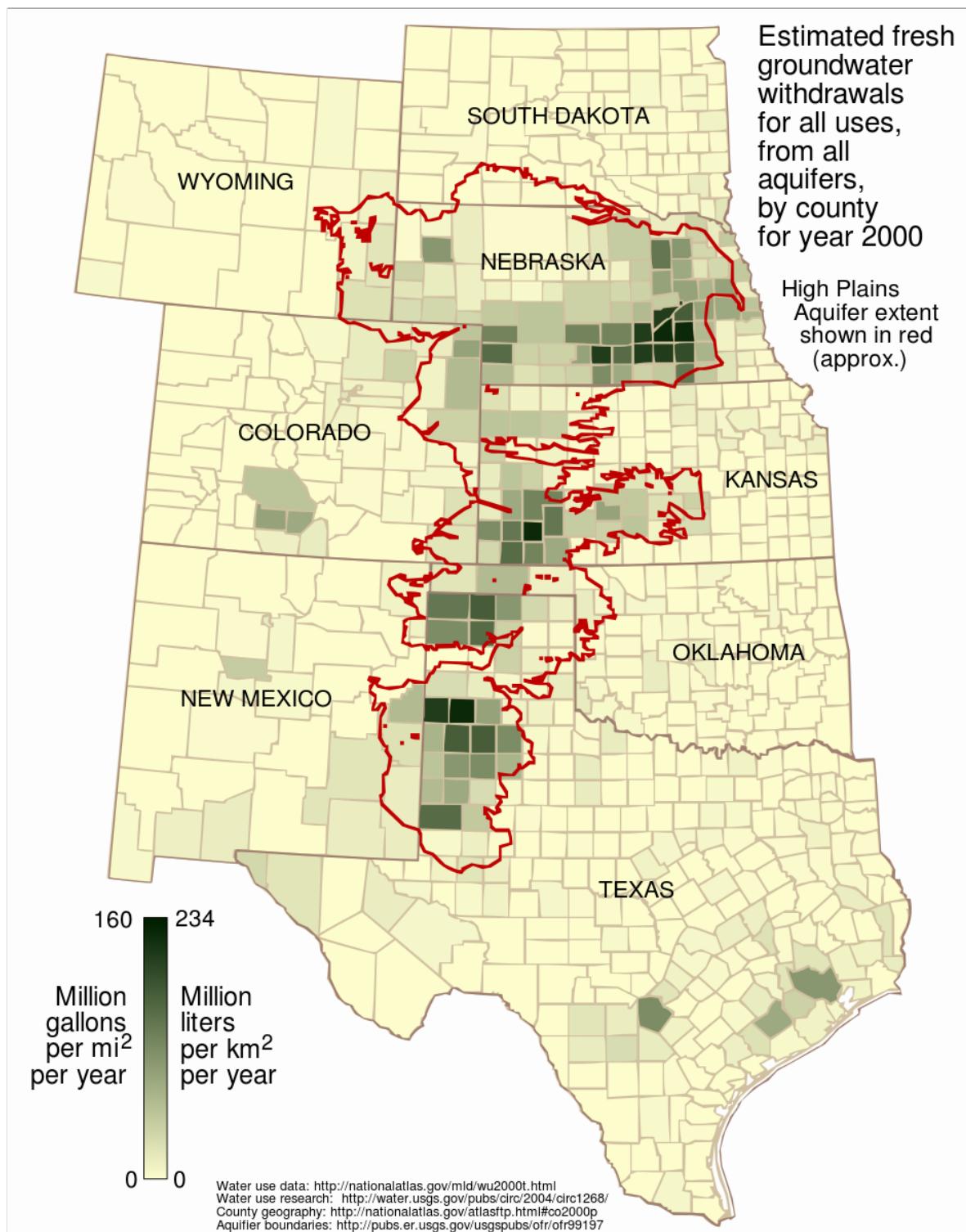


Figure 1.15: Source: Kbh3rd (2009)



Center Pivot irrigation in Nebraska taps the Ogallala Aquifer.

1.6.14 Groundwater flow and discharge

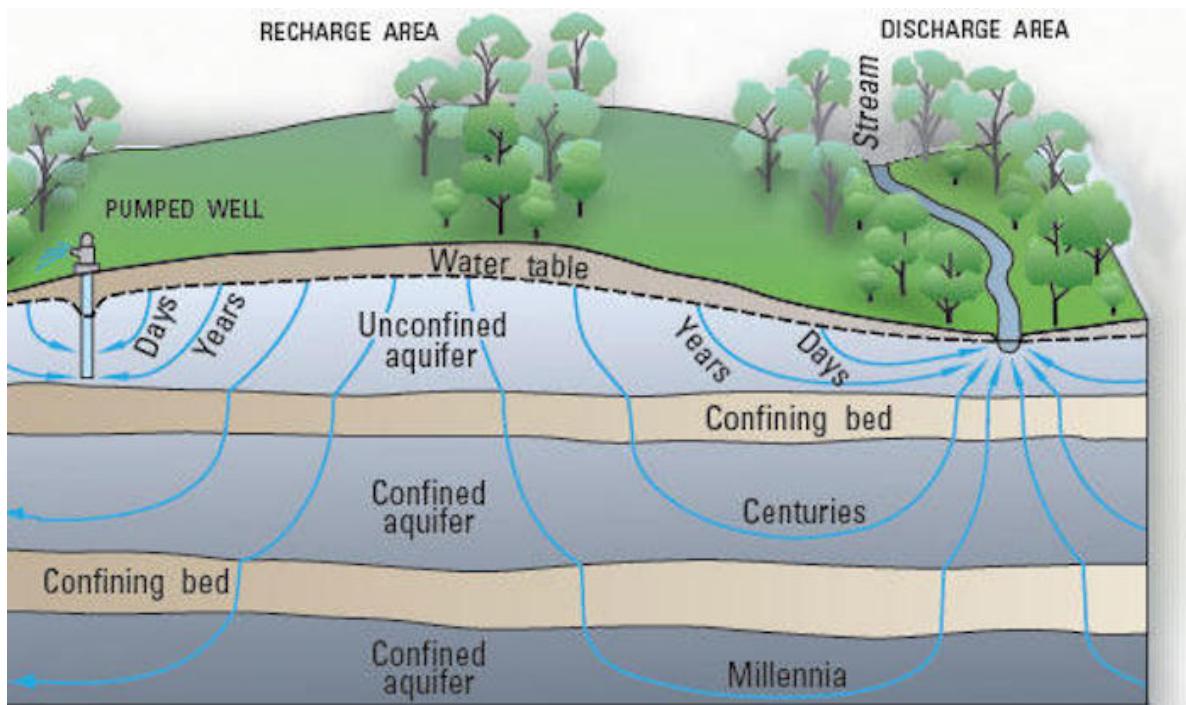


Figure 1.16: Source: Water Science School (2019a)

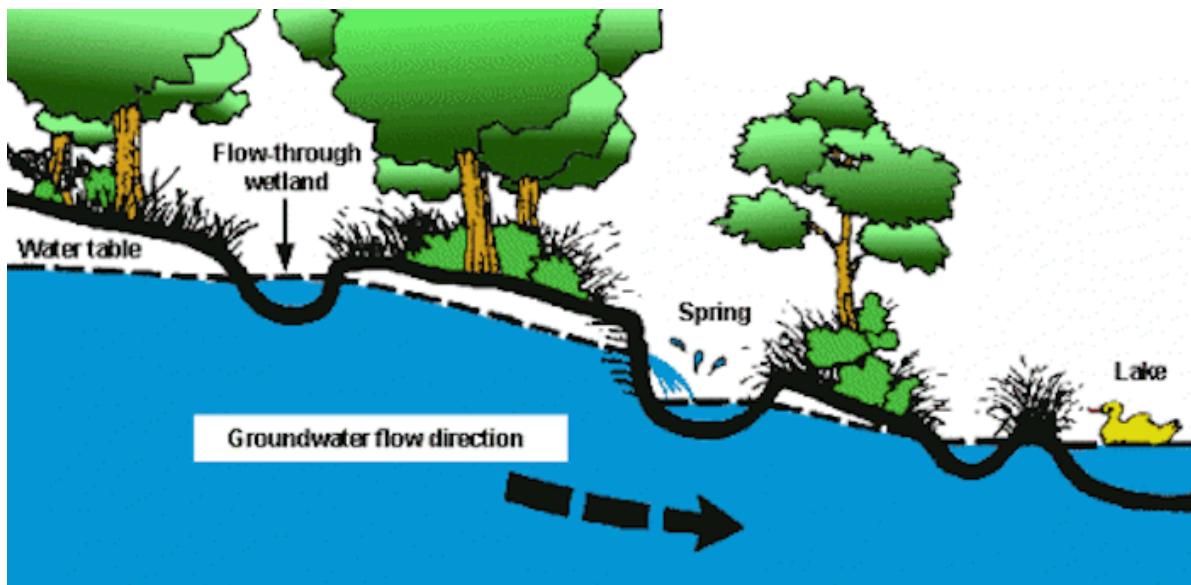


Figure 1.17: Source: Raymond, Lyle S. Jr. (1988)

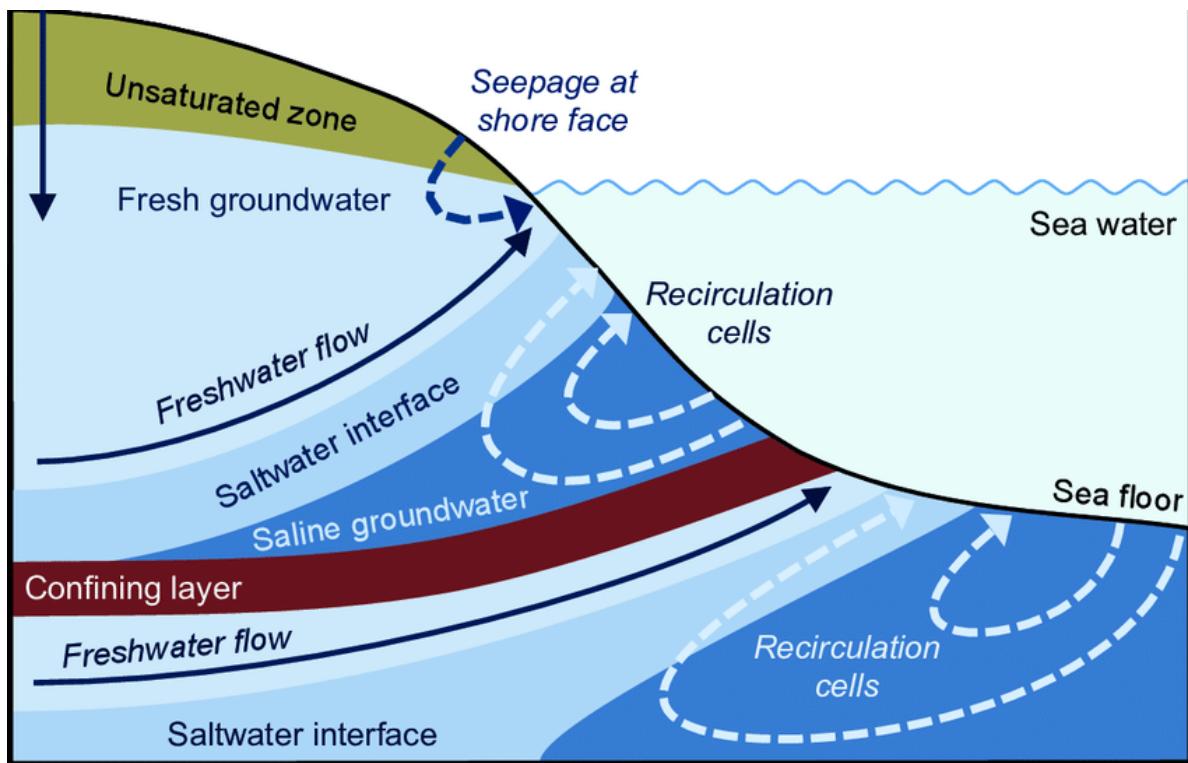


Figure 1.18: Source: Valentí Rodellas (1988)

1.6.15 Spring



Ein Gedi



Thousand Springs, Idaho

2 Exercises

let's have fun plotting some data

2.1 download the data

1. Go to the Faculty of Agriculture's [weather station](#).
2. Click on [here](#) and download data for 1 September 2020 to 28 February 2021, with a 24h interval. Call it `data-sep2020-feb2021`
3. Open the .csv file with Excel, see how it looks like
4. If you can't download the data, just click here.

2.2 import packages

We need to import this data into python. First we import useful packages. **Type** (don't copy and paste) the following lines in the code cell below.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
```

2.3 import data with pandas

Import data from csv and put it in a pandas **dataframe** (a table). Make line 5 the header (column names)

```
df = pd.read_csv("data-sep2020-feb2021.csv",
                  skiprows=4,
                  encoding='latin1',
                  )
df
```

| | Unnamed: 0 | °C | °C.1 | km/h | mm | mm.1 |
|-----|------------|------|------|------|-----|-------|
| 0 | 01/09/20 | 32.8 | 25.3 | 29.7 | 0.0 | 0.0 |
| 1 | 02/09/20 | 33.0 | 24.0 | 28.8 | 0.0 | 0.0 |
| 2 | 03/09/20 | 34.2 | 23.8 | 31.6 | 0.0 | 0.0 |
| 3 | 04/09/20 | 36.3 | 27.3 | 24.2 | 0.0 | 0.0 |
| 4 | 05/09/20 | 34.2 | 26.3 | 22.4 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |
| 176 | 24/02/21 | 20.6 | 9.9 | 28.8 | 0.0 | 481.7 |
| 177 | 25/02/21 | 19.4 | 9.3 | 23.3 | 0.0 | 481.7 |
| 178 | 26/02/21 | 21.3 | 8.0 | 24.2 | 0.1 | 481.8 |
| 179 | 27/02/21 | 23.4 | 9.2 | 30.6 | 0.0 | 481.8 |
| 180 | 28/02/21 | 19.7 | 9.2 | 22.4 | 0.0 | 481.8 |

2.4 rename columns

rename the columns to:

```
date, tmax, tmin, wind, rain24h, rain_cumulative
```

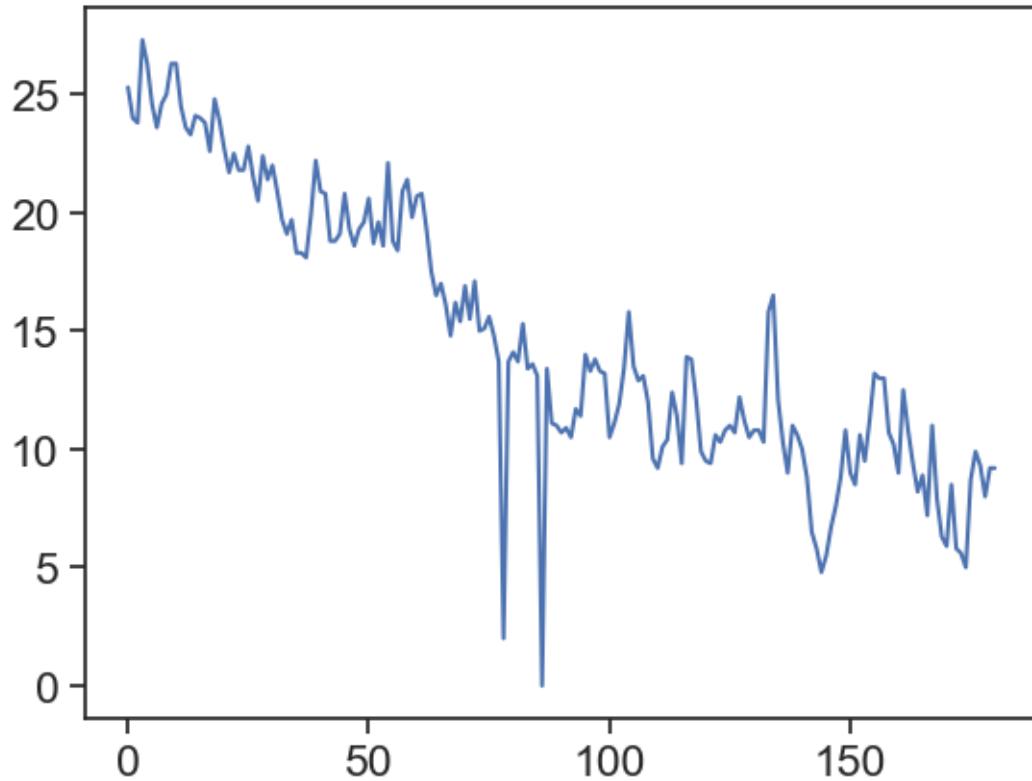
```
df.columns = ['date', 'tmax', 'tmin', 'wind', 'rain24h', 'rain_cumulative']
df
```

| | date | tmax | tmin | wind | rain24h | rain_cumulative |
|-----|----------|------|------|------|---------|-----------------|
| 0 | 01/09/20 | 32.8 | 25.3 | 29.7 | 0.0 | 0.0 |
| 1 | 02/09/20 | 33.0 | 24.0 | 28.8 | 0.0 | 0.0 |
| 2 | 03/09/20 | 34.2 | 23.8 | 31.6 | 0.0 | 0.0 |
| 3 | 04/09/20 | 36.3 | 27.3 | 24.2 | 0.0 | 0.0 |
| 4 | 05/09/20 | 34.2 | 26.3 | 22.4 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |
| 176 | 24/02/21 | 20.6 | 9.9 | 28.8 | 0.0 | 481.7 |
| 177 | 25/02/21 | 19.4 | 9.3 | 23.3 | 0.0 | 481.7 |
| 178 | 26/02/21 | 21.3 | 8.0 | 24.2 | 0.1 | 481.8 |
| 179 | 27/02/21 | 23.4 | 9.2 | 30.6 | 0.0 | 481.8 |
| 180 | 28/02/21 | 19.7 | 9.2 | 22.4 | 0.0 | 481.8 |

2.5 a first plot!

plot the minimum temperature:

```
plt.plot(df['tmin'])
```



2.6 how to deal with dates

We want the dates to appear on the horizontal axis.

Interpret ‘date’ column as a pandas datetime, see how it looks different from before

before: 01/09/20

after: 2020-09-01

```
df['date'] = pd.to_datetime(df['date'], dayfirst=True)  
df
```

| | date | tmax | tmin | wind | rain24h | rain_cumulative |
|---|------------|------|------|------|---------|-----------------|
| 0 | 2020-09-01 | 32.8 | 25.3 | 29.7 | 0.0 | 0.0 |
| 1 | 2020-09-02 | 33.0 | 24.0 | 28.8 | 0.0 | 0.0 |
| 2 | 2020-09-03 | 34.2 | 23.8 | 31.6 | 0.0 | 0.0 |

| | date | tmax | tmin | wind | rain24h | rain_cumulative |
|-----|------------|------|------|------|---------|-----------------|
| 3 | 2020-09-04 | 36.3 | 27.3 | 24.2 | 0.0 | 0.0 |
| 4 | 2020-09-05 | 34.2 | 26.3 | 22.4 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |
| 176 | 2021-02-24 | 20.6 | 9.9 | 28.8 | 0.0 | 481.7 |
| 177 | 2021-02-25 | 19.4 | 9.3 | 23.3 | 0.0 | 481.7 |
| 178 | 2021-02-26 | 21.3 | 8.0 | 24.2 | 0.1 | 481.8 |
| 179 | 2021-02-27 | 23.4 | 9.2 | 30.6 | 0.0 | 481.8 |
| 180 | 2021-02-28 | 19.7 | 9.2 | 22.4 | 0.0 | 481.8 |

2.6.1 date as dataframe index

Make ‘date’ the dataframe’s index (leftmost column, but not really a column!)

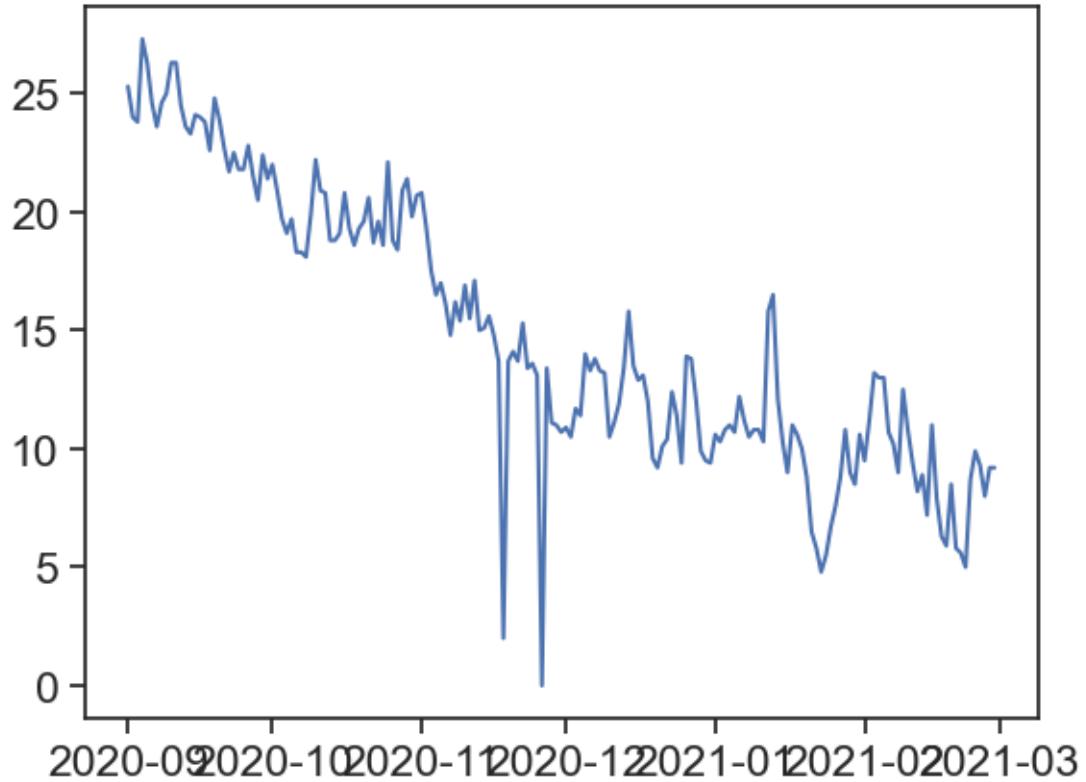
```
df = df.set_index('date')
df
```

| date | tmax | tmin | wind | rain24h | rain_cumulative |
|------------|------|------|------|---------|-----------------|
| 2020-09-01 | 32.8 | 25.3 | 29.7 | 0.0 | 0.0 |
| 2020-09-02 | 33.0 | 24.0 | 28.8 | 0.0 | 0.0 |
| 2020-09-03 | 34.2 | 23.8 | 31.6 | 0.0 | 0.0 |
| 2020-09-04 | 36.3 | 27.3 | 24.2 | 0.0 | 0.0 |
| 2020-09-05 | 34.2 | 26.3 | 22.4 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... |
| 2021-02-24 | 20.6 | 9.9 | 28.8 | 0.0 | 481.7 |
| 2021-02-25 | 19.4 | 9.3 | 23.3 | 0.0 | 481.7 |
| 2021-02-26 | 21.3 | 8.0 | 24.2 | 0.1 | 481.8 |
| 2021-02-27 | 23.4 | 9.2 | 30.6 | 0.0 | 481.8 |
| 2021-02-28 | 19.7 | 9.2 | 22.4 | 0.0 | 481.8 |

2.7 plot again, now with dates

Plot minimum temperature, now we have dates on the horizontal axis

```
plt.plot(df['tmin'])
```

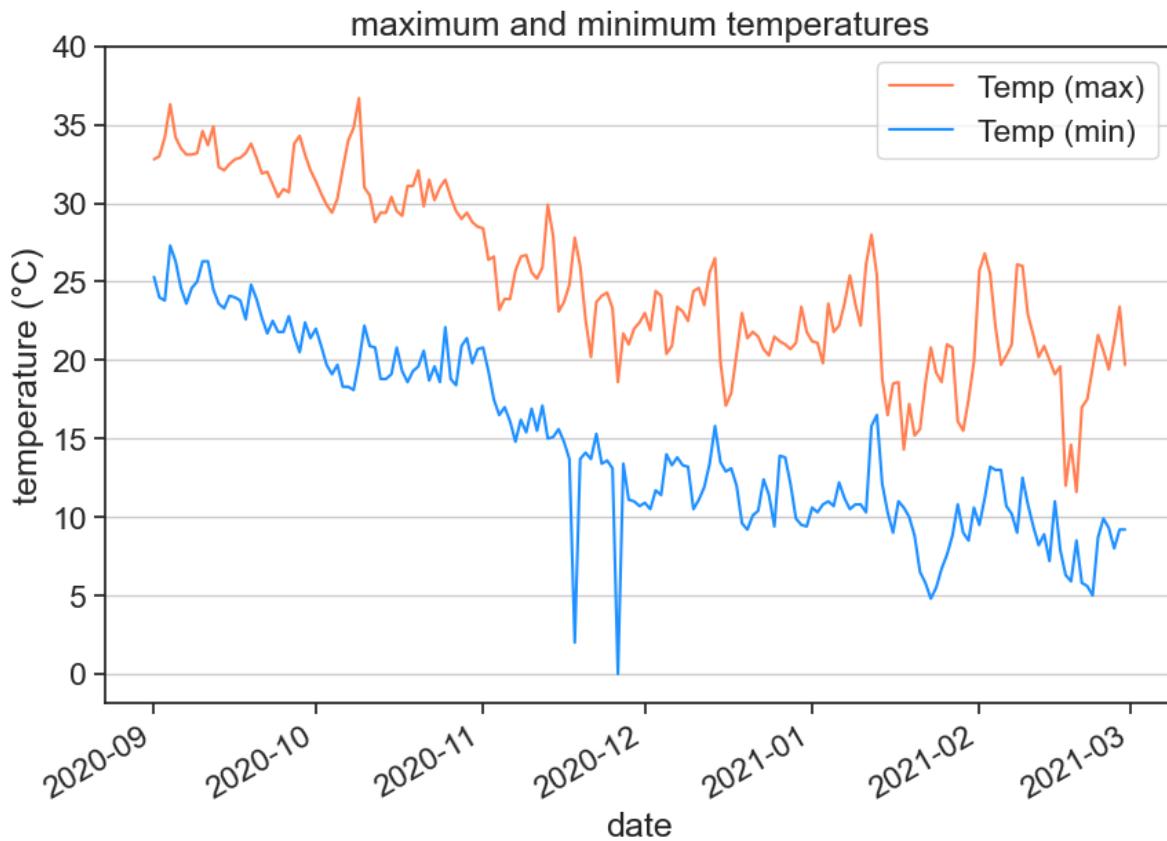


2.8 we're getting there! the graph could look better

Let's make the graph look better: labels, title, slanted dates, etc

```
# creates figure (the canvas) and the axis (rectangle where the plot sits)
fig, ax = plt.subplots(1, figsize=(10,7))
# two line plots
ax.plot(df['tmax'], color="coral", label="Temp (max)")
ax.plot(df['tmin'], color="dodgerblue", label="Temp (min)")
# axes labels and figure title
ax.set_xlabel('date')
ax.set_ylabel('temperature (°C)')
ax.set_title('maximum and minimum temperatures')
# some ticks adjustments
ax.set_yticks(np.arange(0,45,5)) # we can choose where to put ticks
ax.grid(axis='y') # makes horizontal lines
plt.gcf().autofmt_xdate() # makes slanted dates
# legend
```

```
ax.legend(loc='upper right')
# save png figure
plt.savefig("temp_max_min.png")
```



2.9 make the following figure

Use the following function to plot bars for daily rainfall

```
ax.bar(x_array, y_array)
```

Can you write yourself some lines of code that calculate the cumulative rainfall from the daily rainfall?

```

# creates figure (the canvas) and the axis (rectangle where the plot sits)
fig, ax = plt.subplots(1, figsize=(10,7))

# line and bar plots
ax.bar(df.index, df['rain24h'], color="mediumblue", label="daily rainfall")

# there are many ways of calculating the cumulative rain

# method 1, use a for loop:
# rain = df['rain24h'].to_numpy()
# cumulative = rain * 0
# for i in range(len(rain)):
#     cumulative[i] = np.sum(rain[:i])
# df['cumulative1'] = cumulative

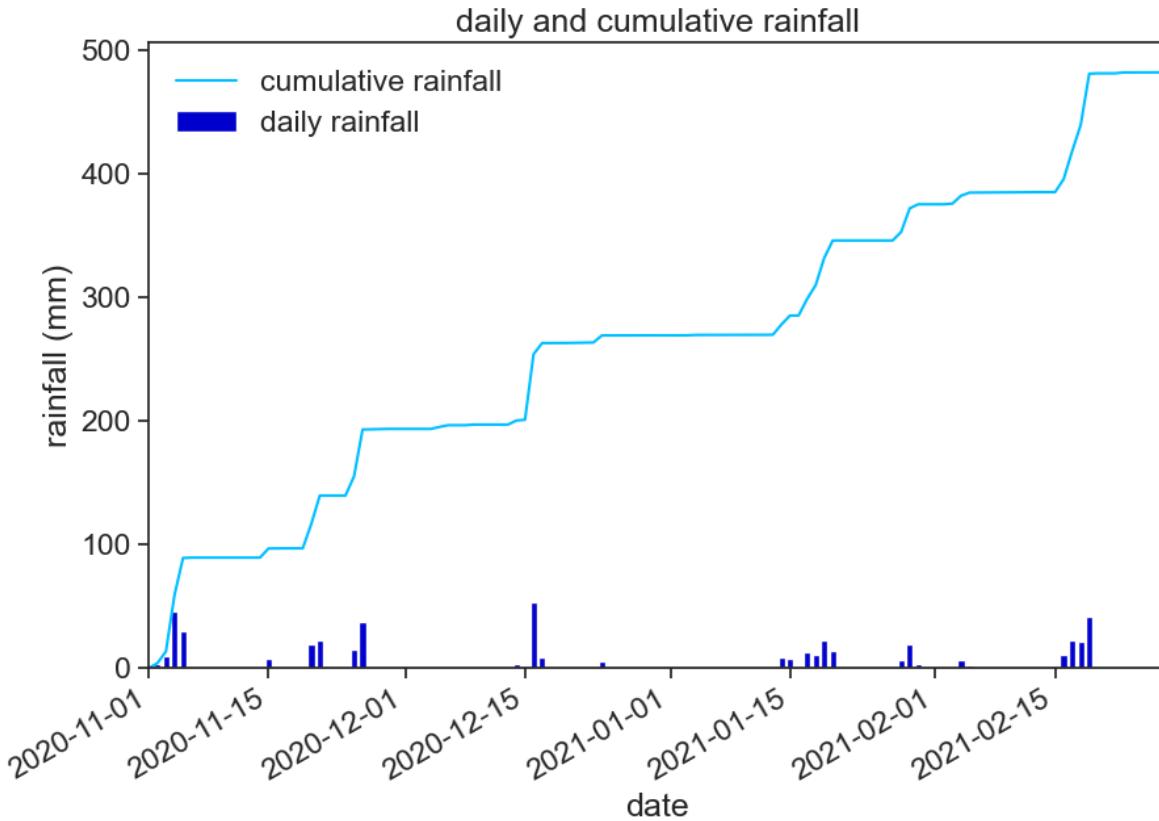
# method 2, use list comprehension:
# rain = df['rain24h'].to_numpy()
# cumulative = [np.sum(rain[:i]) for i in range(len(rain))]
# df['cumulative2'] = cumulative

# method 3, use existing functions:
df['cumulative3'] = np.cumsum(df['rain24h'])

ax.plot(df['cumulative3'], color="deepskyblue", label="cumulative rainfall")
# compare our cumulative rainfall with the downloaded data
# ax.plot(df['rain_cumulative'], 'x')
# axes labels and figure title
ax.set(xlabel='date',
       ylabel='rainfall (mm)',
       title='daily and cumulative rainfall',
       xlim=pd.to_datetime(['2020-11-01','2021-02-28']))
)

# some ticks adjustments
plt.gcf().autofmt_xdate() # makes slanted dates
# legend
ax.legend(loc='upper left', frameon=False)
# save png figure
plt.savefig("cumulative_rainfall.png")

```



2.10 make another figure

In order to choose just a part of the time series, you can use the following:

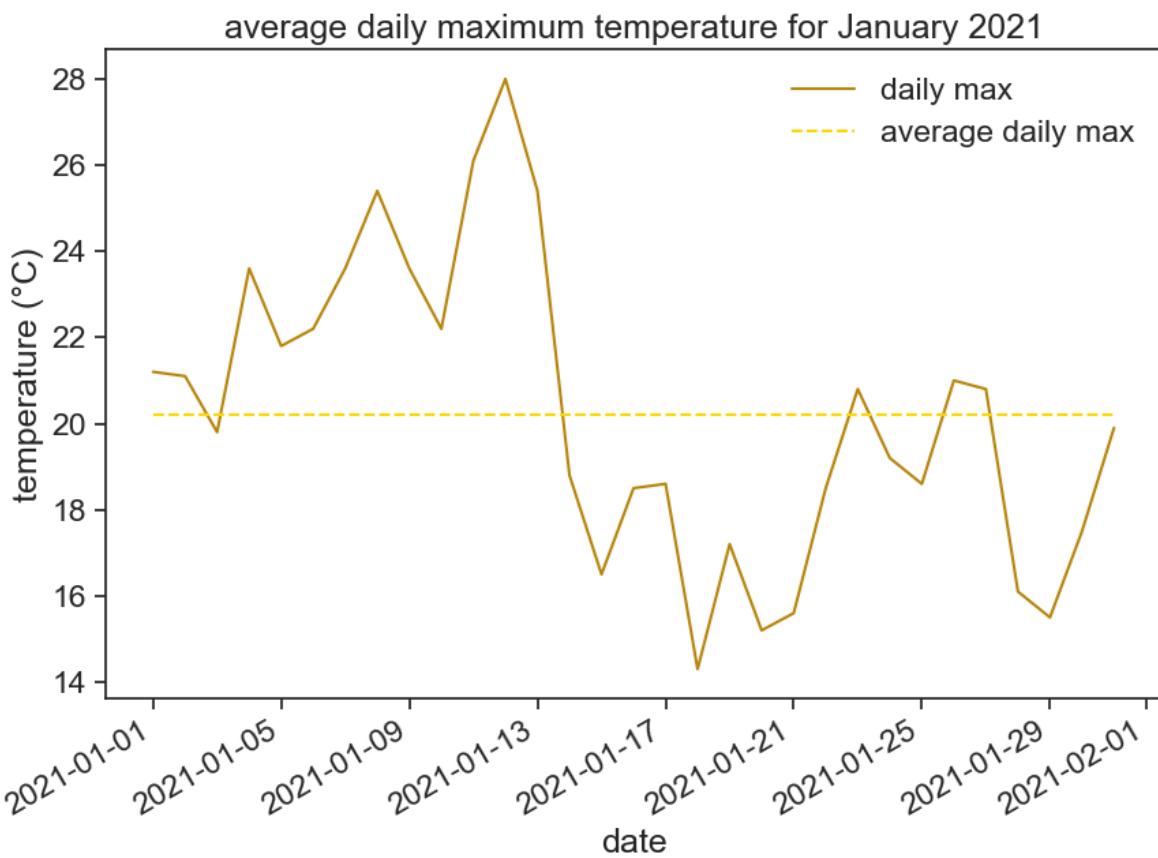
```
start_date = '2021-01-01'  
end_date = '2021-01-31'  
january = df[start_date:end_date]
```

```
# creates figure (the canvas) and the axis (rectangle where the plot sits)
fig, ax = plt.subplots(1, figsize=(10,7))
# define date range
start_date = '2021-01-01'
end_date = '2021-01-31'
january = df.loc[start_date:end_date, 'tmax']
# plots
ax.plot(january, color="darkgoldenrod", label="daily max")
```

```

ax.plot(january*0 + january.mean(), color="gold", linestyle="--", label="average daily max")
# axes labels and figure title
ax.set_xlabel('date')
ax.set_ylabel('temperature (°C)')
ax.set_title('average daily maximum temperature for January 2021')
# some ticks adjustments
plt.gcf().autofmt_xdate() # makes slanted dates
# legend
ax.legend(loc='upper right', frameon=False)
# save png figure
plt.savefig("average_max_temp.png")

```



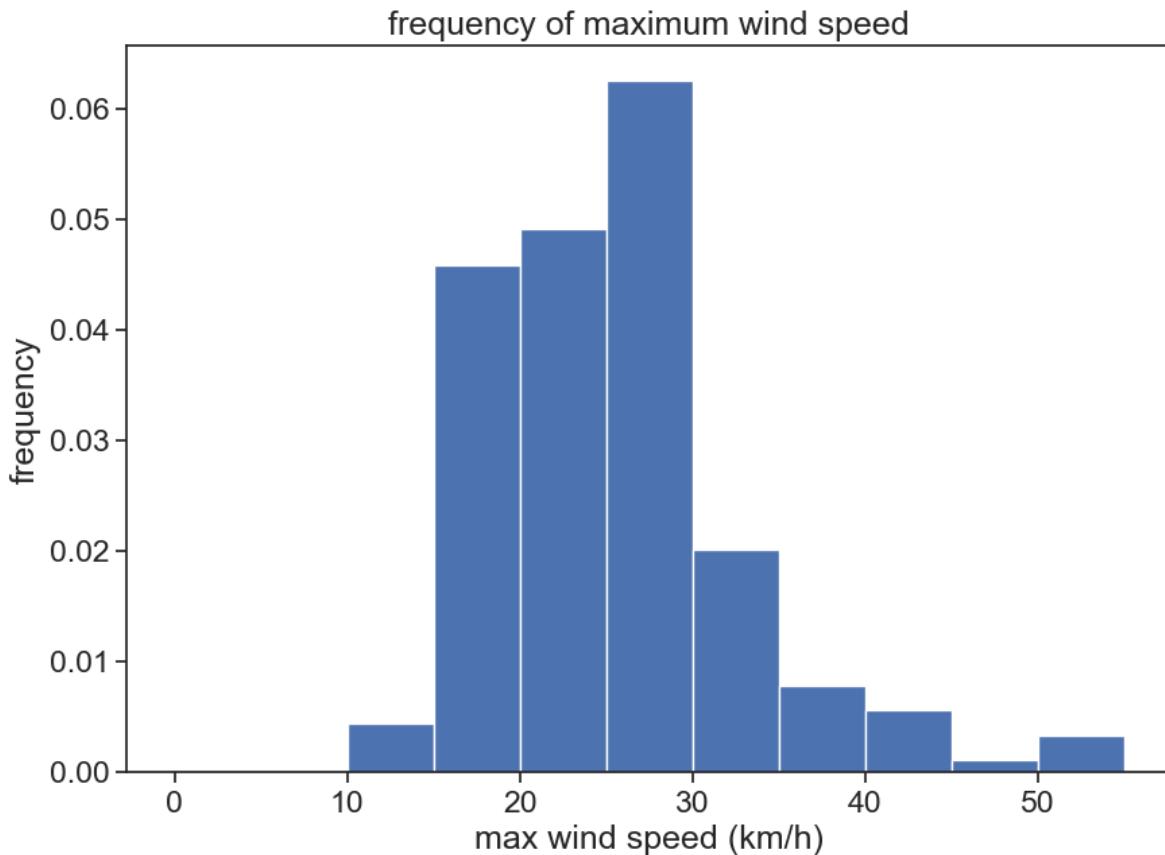
2.11 one last figure for today

Use the following code to create histograms with user-defined bins:

```
b = np.arange(0, 56, 5) # bins from 0 to 55, width = 5
ax.hist(df['wind'], bins=b, density=True)
```

Play with the bins, see what happens. What does `density=True` do?

```
# creates figure (the canvas) and the axis (rectangle where the plot sits)
fig, ax = plt.subplots(1, figsize=(10,7))
# histogram
b = np.arange(0, 56, 5) # bins from 0 to 55, width = 5
ax.hist(df['wind'], bins=b, density=True)
# axes labels and figure title
ax.set(xlabel='max wind speed (km/h)',
       ylabel='frequency',
       title='frequency of maximum wind speed'
      )
# save png figure
plt.savefig("wind-histogram.png")
```



2.12 homework

Go back to the weather station website, download one year of data from 01.01.2020 to 31.12.2020 (24h data). If you can't download the data, just click here.

2.12.1 graph 1

Make one graph with the following:

- daily tmax and tmin
- smoothed data for tmax and tmin

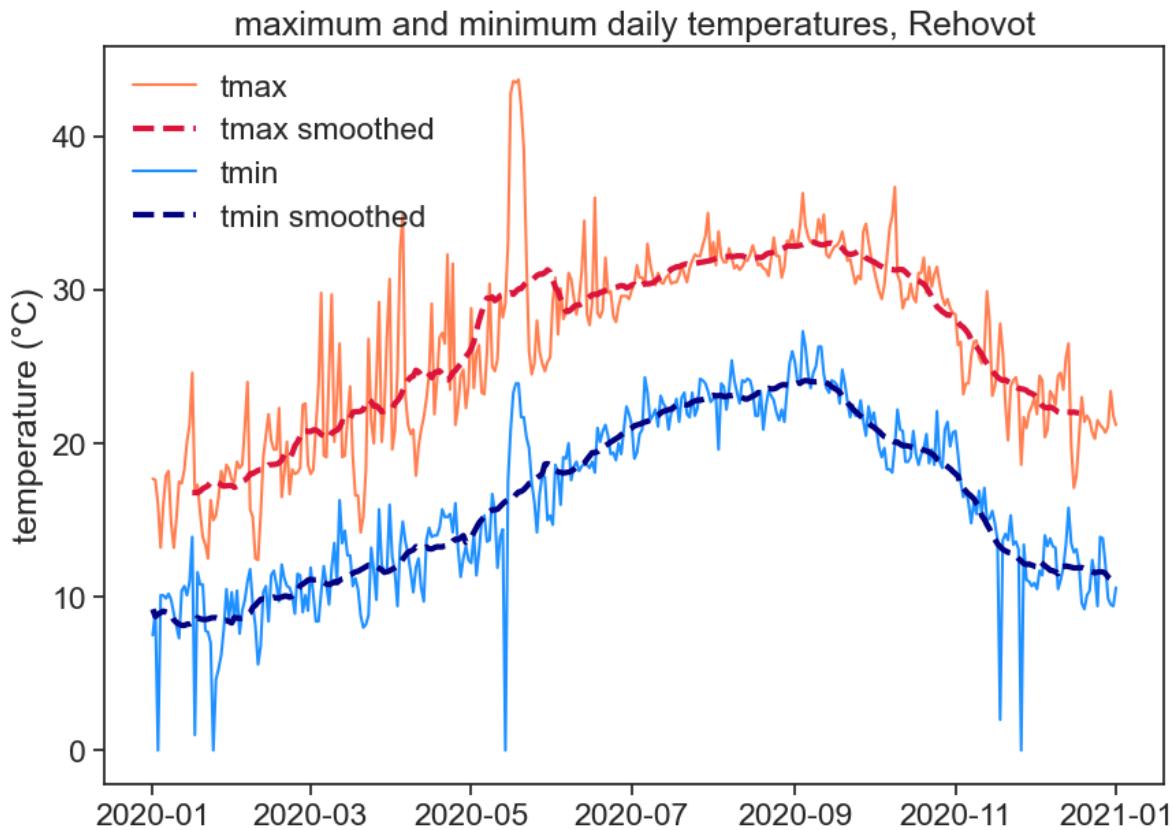
In order to smooth the data with a 30 day window, use the following function:

```
df['tmin'].rolling(30, center=True).mean()
```

This means that you will take the mean of 30 days, and put the result in the center of this 30-day window.

Play with this function, see what you can do with it. What happens when you change the size of the window? Why is the smoothed data shorter than the original data? See the [documentation](#) for `rolling` to find more options.

```
fig, ax = plt.subplots(figsize=(10,7))
col_names = ['date', 'tmax', 'tmin', 'wind', 'rain24h', 'rain_cumulative']
df2 = pd.read_csv("1year.csv",
                  skiprows=5,
                  encoding='latin1',
                  names=col_names,
                  parse_dates=['date'],
                  dayfirst=True,
                  index_col='date'
                 )
tmin_smooth = df2['tmin'].rolling('30D', center=True).mean()
tmax_smooth = df2['tmax'].rolling(30, center=True).mean()
ax.plot(df2['tmax'], label='tmax', color="coral")
ax.plot(tmax_smooth, label='tmax smoothed', color="crimson", linestyle="--", linewidth=3)
ax.plot(df2['tmin'], label='tmin', color="dodgerblue")
ax.plot(tmin_smooth, label='tmin smoothed', color="navy", linestyle="--", linewidth=3)
ax.legend(frameon=False)
ax.set(ylabel='temperature (°C)',
       title='maximum and minimum daily temperatures, Rehovot')
plt.savefig("t_smoothed.png")
```



2.12.2 graph 2

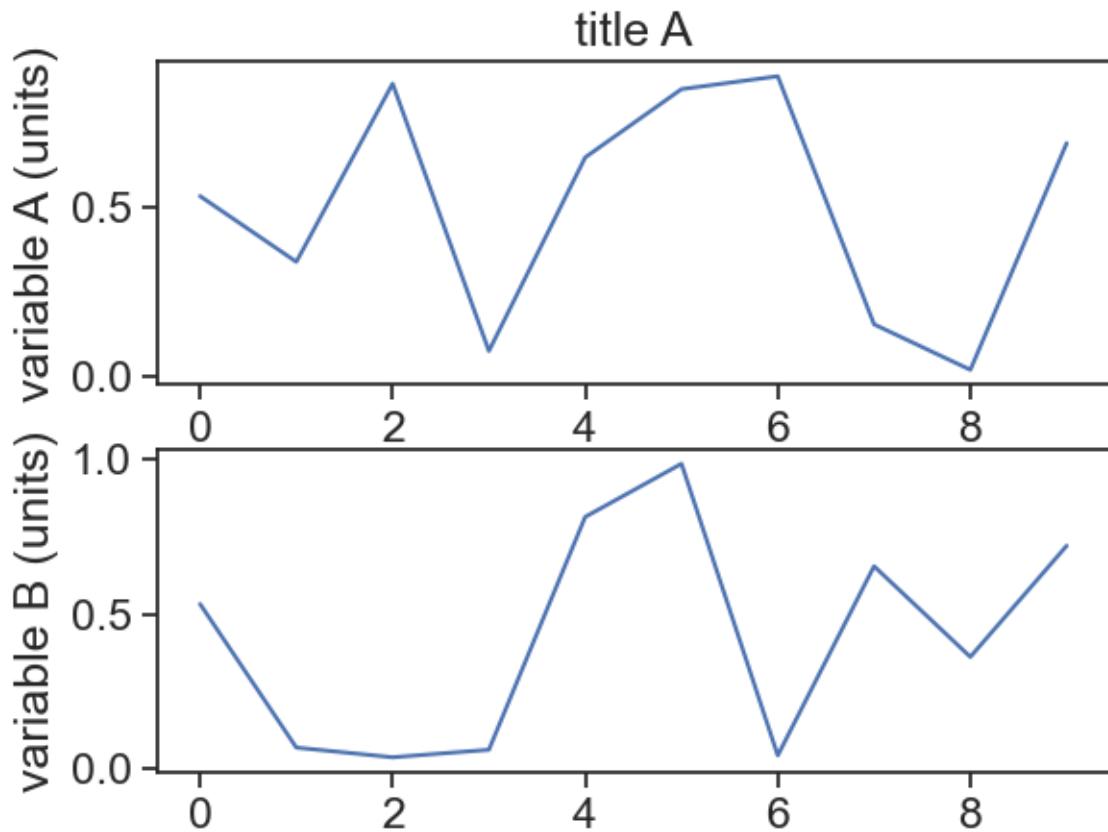
Make another graph that focuses on a part of the year, not the whole thing. We saw before how to do that. Put on this graph two lines, representing two variables of your choosing. Give these lines good colors, and maybe different line styles (solid, dashed, dotted) if you feel fancy.

2.12.3 graph 3

Choose another variable, and make two histograms (each in its own panel), each representing a different time interval. Here is an example how you make subplots:

```
fig, (ax1, ax2) = plt.subplots(2, 1)
N = 10
ax1.plot(np.random.random(N))
ax2.plot(np.random.random(N))
```

```
ax1.set(ylabel='variable A (units)',  
        title='title A'  
       )  
ax2.set(ylabel='variable B (units)',  
        # title='title B'  
       )
```



Of course, I plotted random data as a line plot, your task is to plot histograms instead.

Part II

Precipitation

Here are some of the files we'll use in this module, in case you can't download them from their original repositories.

- [BEN_GURION_monthly.csv](#)
- [BEER_SHEVA_monthly.csv](#)
- [Eilat_daily.csv](#)

3 Intra-annual variability of precipitation

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
# from calendar import month_abbr
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import urllib.request

df_telaviv = pd.read_csv("TEL_AVIV_READING_monthly.csv",
                        sep=",",
                        parse_dates=['DATE'],
                        index_col='DATE'
                       )
df_london = pd.read_csv("LONDON_HEATHROW_monthly.csv",
                        sep=",",
                        parse_dates=['DATE'],
                        index_col='DATE'
                       )
monthly_london = (df_london['PRCP']
                   .groupby(df_london.index.month)
                   .mean()
                   .to_frame()
                  )
monthly_telaviv = (df_telaviv['PRCP']
                     .groupby(df_telaviv.index.month)
                     .mean()
                     .to_frame()
                    )

fig, ax = plt.subplots(figsize=(10,7))

# bar plots
ax.bar(monthly_london.index, monthly_london['PRCP'],
```

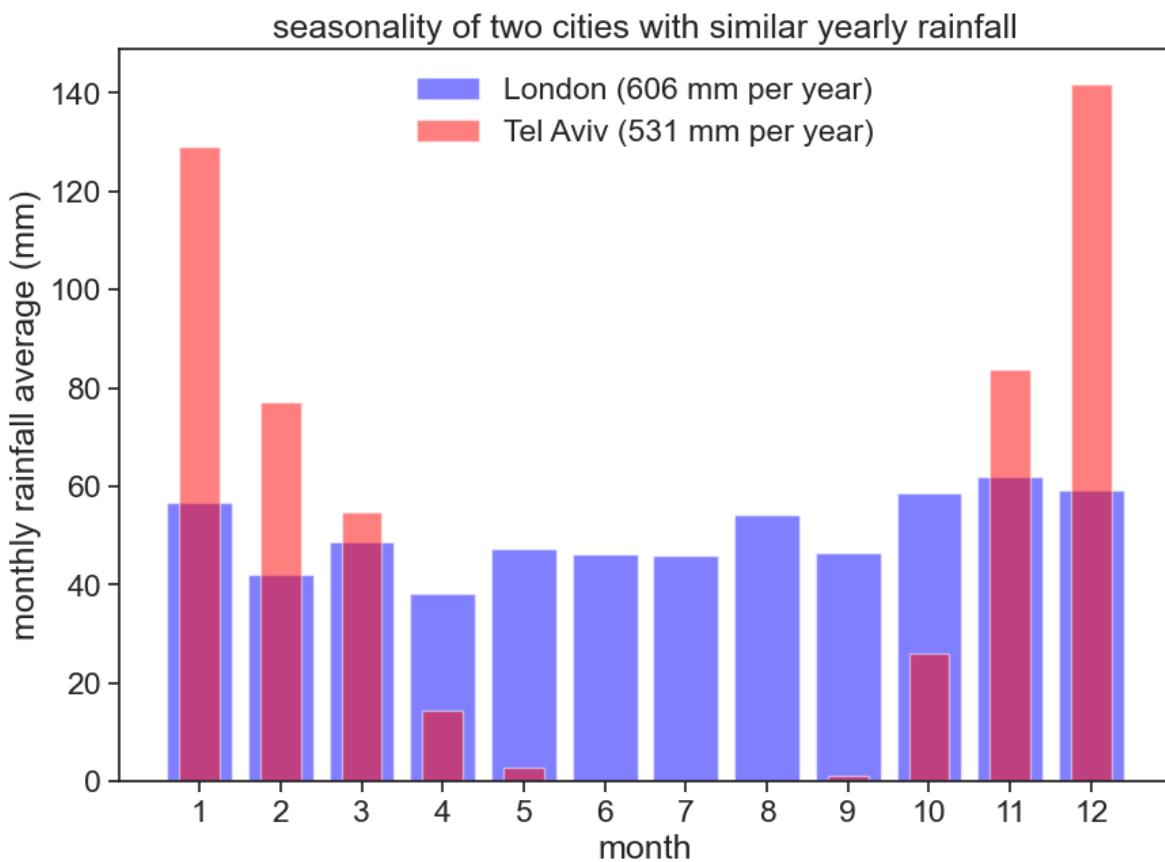
```

alpha=0.5, color="blue", label=f"London ({monthly_london.values.sum():.0f} mm per year)
ax.bar(monthly_telaviv.index, monthly_telaviv['PRCP'],
       alpha=0.5, color="red", width=0.5, label=f"Tel Aviv ({monthly_telaviv.values.sum():.0f} mm per year)

# axes labels and figure title
ax.set(xlabel='month',
       ylabel='monthly rainfall average (mm)',
       title='seasonality of two cities with similar yearly rainfall',
       xticks=monthly_telaviv.index
      )
ax.legend(loc='upper center', frameon=False);

# save figure
# plt.savefig("hydrology_figures/monthly_tel_aviv_london_bars.png")

```



3.1 hydrological year

The hydrological year is time period of 12 months for which precipitation totals are measured. The hydrological year is designated by the calendar year in which it **ends**.

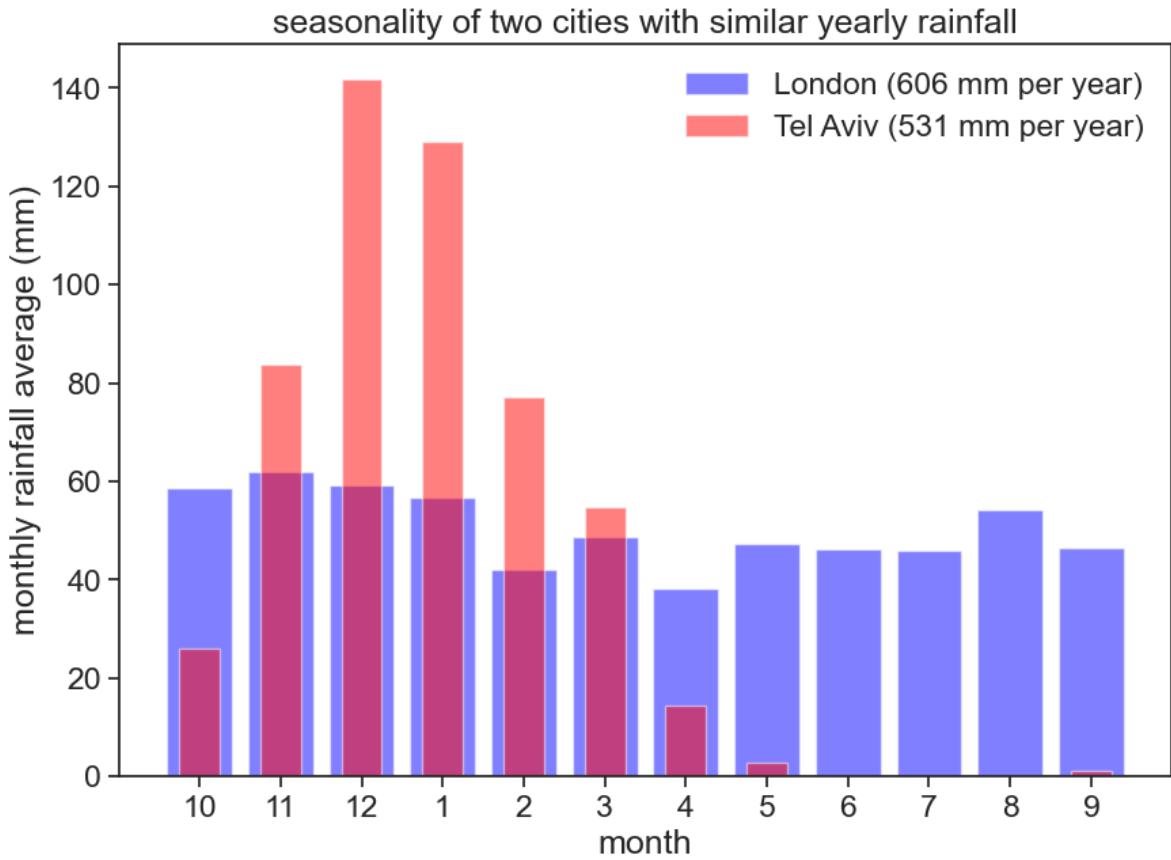
In temperate regions with distinct seasonal patterns, the hydrological year often starts in the fall, when precipitation and streamflow are typically at their lowest levels. This timing ensures that most of the surface runoff during the water year is attributable to the precipitation that fell during the same period.

Let's define the hydrological year for Tel Aviv from 1 October to 30 September.

We will now shift the months according to Tel Aviv's hydrological year.

```
fig, ax = plt.subplots(figsize=(10,7))
Nroll = 3 # number of months to roll
roll_telaviv = np.roll(monthly_telaviv['PRCP'], Nroll)
roll_london = np.roll(monthly_london['PRCP'], Nroll)
roll_months = np.roll(monthly_london.index, Nroll)
# bar plots
ax.bar(monthly_london.index, roll_london,
       alpha=0.5, color="blue", label=f"London ({monthly_london.values.sum():.0f} mm per year")
ax.bar(monthly_telaviv.index, roll_telaviv,
       alpha=0.5, color="red", width=0.5, label=f"Tel Aviv ({monthly_telaviv.values.sum():.0f} mm per year")
# axes labels and figure title
ax.set(xlabel='month',
       ylabel='monthly rainfall average (mm)',
       title='seasonality of two cities with similar yearly rainfall',
       xticks=monthly_london.index,
       xticklabels=roll_months
      )
ax.legend(loc='upper right', frameon=False);

# save figure
# plt.savefig("monthly_tel_aviv_london_bars.png")
```



Another way of representing this data is with polar coordinates:

```
fig = plt.figure(figsize=(10,10))

# radar chart
ax = fig.add_subplot(111, polar=True)          # make polar plot
ax.set_theta_zero_location("N")                 # January on top ("N"orth)
ax.set_theta_direction(-1)                      # clockwise direction
ax.set_rlabel_position(90)                     # radial labels on the right
ax.set_rticks([50,100])                        # two radial ticks is enough
ax.set_rlim(0,150)                            # limits of r axis
angles=np.linspace(0, 2*np.pi, 12, endpoint=False) # divide circle into 12 slices
angles=angles.append(angles[0])                # close loop, otherwise lines will
month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
ax.set_thetagrids(angles[:-1] * 180/np.pi, month_names) # relabel angles with month names

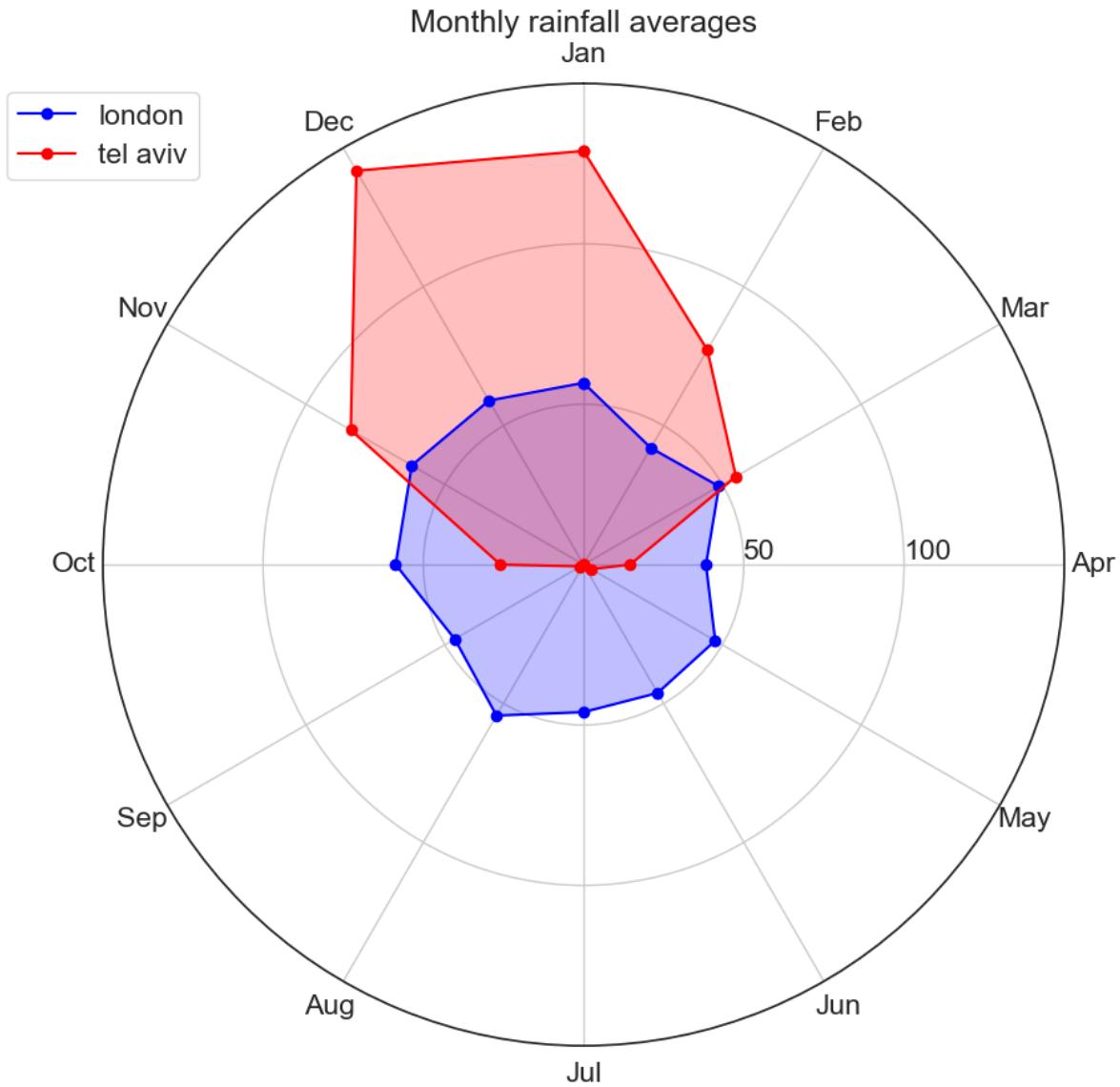
# plot london data
stats_london = np.array(monthly_london['PRCP'].values)      # get london data
```

```
stats_london = np.append(stats_london, stats_london[0])                      # close loop
ax.plot(angles, stats_london, "o-", color='blue', label="london")    # plot line
ax.fill(angles, stats_london, alpha=0.25, color='blue')                      # fill

# plot tel aviv data
stats_telaviv = np.array(monthly_telaviv['PRCP'].values)                  # get tel aviv data
stats_telaviv = np.append(stats_telaviv, stats_telaviv[0])                  # close loop
ax.plot(angles, stats_telaviv, "o-", color='red', label="tel aviv")    # plot line
ax.fill(angles, stats_telaviv, alpha=0.25, color='red')                      # fill

ax.set_title("Monthly rainfall averages")
ax.legend(loc=(-0.1,0.9));  # legend at x=-0.2 so it doesn't overlap with graph

# save figure
# plt.savefig("radar_chart_tel_aviv_london.png")
```



3.2 Seasonality Index

Sources: Ieddris (2010), Walsh and Lawler (1981)

$\langle P \rangle$ = mean annual precipitation

m_i = precipitation mean for month i

$$SI = \frac{1}{\langle P \rangle} \sum_{n=1}^{n=12} \left| m_i - \frac{\langle P \rangle}{12} \right|$$

| <i>SI</i> | Precipitation Regime |
|-----------|---|
| <0.19 | Precipitation spread throughout the year |
| 0.20-0.39 | Precipitation spread throughout the year, but with a definite wetter season |
| 0.40-0.59 | Rather seasonal with a short dry season |
| 0.60-0.79 | Seasonal |
| 0.80-0.99 | Marked seasonal with a long dry season |
| 1.00-1.19 | Most precipitation in <3 months |

Let's write some code to calculate the SI for Tel Aviv and London.

```
def walsh_index(df):
    m = df["PRCP"].values
    R = m.sum()
    SI = np.sum(np.abs(m-R/12)) / R
    return SI

london_index = walsh_index(monthly_london)
telaviv_index = walsh_index(monthly_telaviv)
print("Seasonality index (Walsh and Lawler, 1981)")
print(f"London: {london_index:.2f}")
print(f"Tel Aviv: {telaviv_index:.2f}")
```

```
Seasonality index (Walsh and Lawler, 1981)
London: 0.13
Tel Aviv: 1.00
```

```
fig, ax = plt.subplots(figsize=(10,7))

plt.rcParams['hatch.linewidth'] = 3
roll_telaviv
xlim = [1, 13]
total_telaviv = np.sum(roll_telaviv)
ax.plot(xlim, [total_telaviv/12]*2, color="tab:blue", linewidth=3)
ax.set_xlim(xlim)
```

```

shaded = roll_telaviv - total_telaviv/12
months = monthly_telaviv.index
ax.bar(months, shaded,
       alpha=0.9, color="None", width=1,
       hatch="//", edgecolor='k',
       align='edge', bottom=total_telaviv/12,
       label=f"absolute difference")

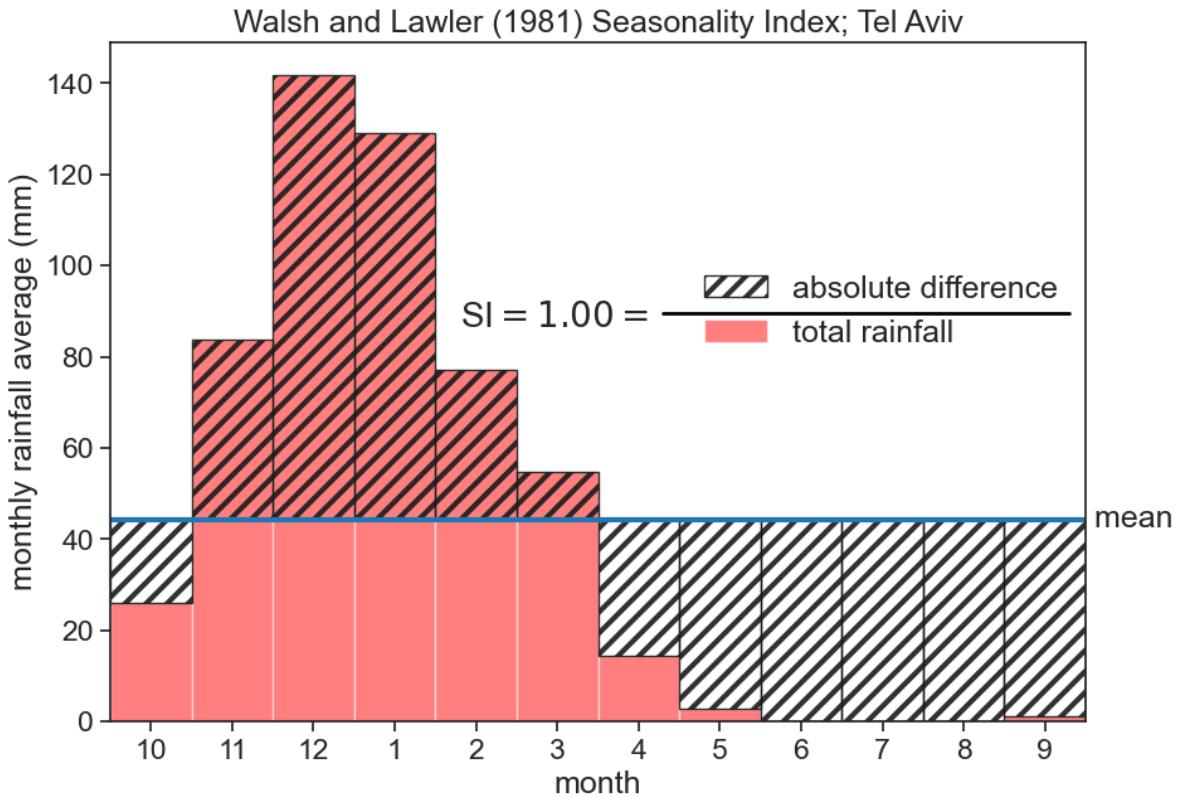
ax.bar(months, roll_telaviv,
       alpha=0.5, color="red", width=1,
       align='edge',
       label=f"total rainfall", zorder=0)

ax.text(5.3, 86.5, r"SI$=1.00=$", fontsize=20)
ax.text(xlim[-1], total_telaviv/12, " mean", va="center")
ax.plot([7.8, 12.8], [89.5]*2, color="black", lw=2)
# axes labels and figure title
ax.set(xlabel='month',
       ylabel='monthly rainfall average (mm)',
       title='Walsh and Lawler (1981) Seasonality Index; Tel Aviv',
       xticks=np.arange(1.5,12.6,1),
       xticklabels=roll_months,
       )

plt.legend(loc='upper right', frameon=False, bbox_to_anchor=(1, 0.7),
           fontsize=18);

# save figure
# plt.savefig("si_walsh_telaviv.png")

```



```

fig, ax = plt.subplots(figsize=(10,7))

plt.rcParams['hatch.linewidth'] = 3
xlim = [1, 13]
total_london = np.sum(roll_london)
ax.plot(xlim, [total_london/12]*2, color="tab:blue", linewidth=3)
ax.set_xlim(xlim)

shaded = roll_london - total_london/12
months = monthly_london.index
ax.bar(months, shaded,
       alpha=0.9, color="None", width=1,
       hatch="//", edgecolor='k',
       align='edge', bottom=total_london/12,
       label=f"absolute difference")

ax.bar(months, roll_london,
       alpha=0.5, color="red", width=1,
       align='edge',

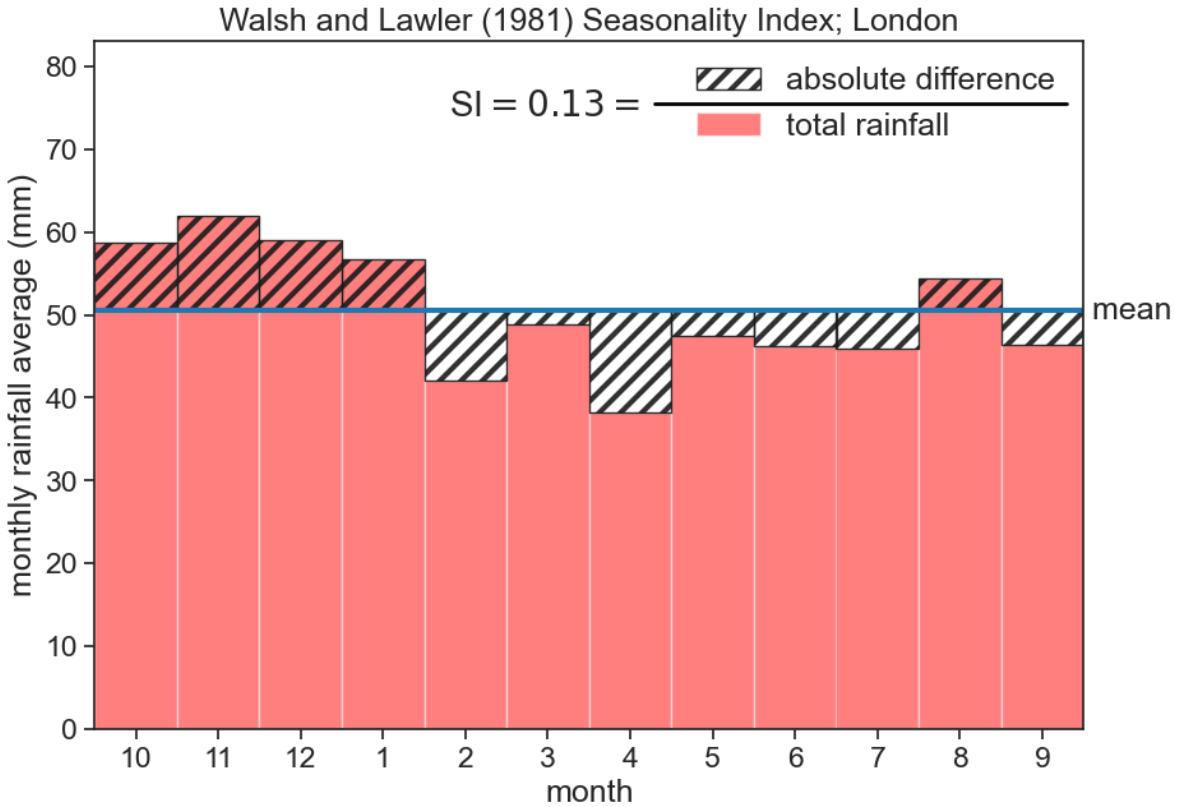
```

```
label=f"total rainfall", zorder=0)

ax.text(5.3, 74, r"SI$=0.13=$", fontsize=20)
ax.text(xlim[-1], total_london/12, " mean", va="center")
ax.plot([7.8, 12.8], [75.5]*2, color="black", lw=2)
# axes labels and figure title
ax.set(xlabel='month',
       ylabel='monthly rainfall average (mm)',
       title='Walsh and Lawler (1981) Seasonality Index; London',
       xticks=np.arange(1.5,12.6,1),
       xticklabels=roll_months,
       ylim=[0,83],
       )

plt.legend(loc='upper right', frameon=False, bbox_to_anchor=(1, 1.005),
           fontsize=18);

# save figure
# plt.savefig("si_walsh_telaviv.png")
```



What is greatest possible value for the Walsh and Lawler seasonality index?

This must be the case when all the rainfall is concentrated in only one month of the year. calculation

Without loss of generality, assume that all the rain in the year (P) is in January: $m_1 = P$ and $m_i = 0$ for other months.

$$\begin{aligned}
 SI &= \frac{1}{P} \sum_{n=1}^{n=12} \left| m_i - \frac{\langle P \rangle}{12} \right| \\
 &= \frac{1}{P} \left| P - \frac{P}{12} \right| + \frac{1}{P} \sum_{n=2}^{n=12} \left| 0 - \frac{P}{12} \right| \\
 &= \frac{11}{12} + 11 \cdot \frac{1}{12} \\
 &= \frac{11}{6} \\
 &= 1.83
 \end{aligned}$$

Can you think of a better seasonality index?

Think of possible problems with Walsh and Lawler's index, then try to fix them :)

What would happen to the index value if we were to randomly shuffle the months?

4 Exercises

Import relevant packages

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import urllib.request
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

4.1 intra-annual variability

Go to NOAA's National Centers for Environmental Information (NCEI)
[Climate Data Online: Dataset Discovery](#)

Find station codes in this [map](#). On the left, click on the little wrench () next to “Global Summary of the Month”, then click on “identify” on the panel that just opened, and click on a station (purple circle). You will see the station’s name, it’s ID, and the period of record. For example, for Ben-Gurion’s Airport in Israel:

BEN GURION, IS

STATION ID: ISM00040180

Period of Record: 1951-01-01 to 2020-03-01

You can download **daily** or **monthly** data for each station. Use the function below to download this data to your computer.

If everything fails and you need easy access to the files we’ll be using today, click here:
[Ben Gurion, Beer Sheva](#).

```
def download_data(station_name, station_code):
    url_daily = 'https://www.ncei.noaa.gov/data/global-historical-climatology-network-daily/'
    url_monthly = 'https://www.ncei.noaa.gov/data/gsom/access/'
    # download daily data - uncomment the next 2 lines to make this work
```

```
# urllib.request.urlretrieve(url_daily + station_code + '.csv',
#                             station_name + '_daily.csv')
# download monthly data
urllib.request.urlretrieve(url_monthly + station_code + '.csv',
                           station_name + '_monthly.csv')
```

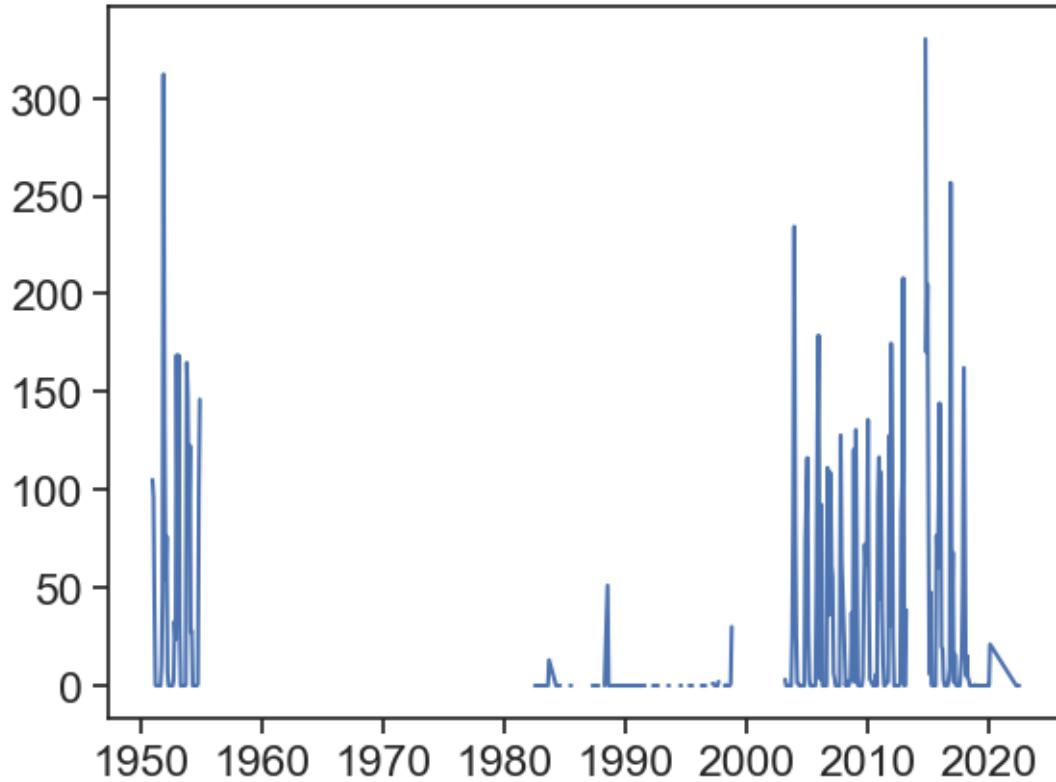
The code above is an example of an API (Application Programming Interface). An API is a set of rules and protocols that allows different software applications to communicate and interact with each other. It acts as an intermediary layer that enables data transmission between different systems or components in a standardized way. For example, when you use a mobile app to book a ride, the app communicates with the ride-sharing service's systems through their API to request and receive data about available drivers, pricing, and booking details.

Now, choose any station with a period of record longer than 30 years, and download its data:

```
download_data('BEN_GURION', 'ISM00040180')
```

Load the data into a datafram, and before you continue with the analysis, plot the rainfall data, to see how it looks like.

```
download_data('BEN_GURION', 'ISM00040180')
df = pd.read_csv('BEN_GURION_monthly.csv', sep=",")
# make 'DATE' the dataframe index
df['DATE'] = pd.to_datetime(df['DATE'])
df = df.set_index('DATE')
plt.plot(df['PRCP']);
```



It doesn't look great for Ben-Gurion airport, lots of missing data! You might need to choose another station...

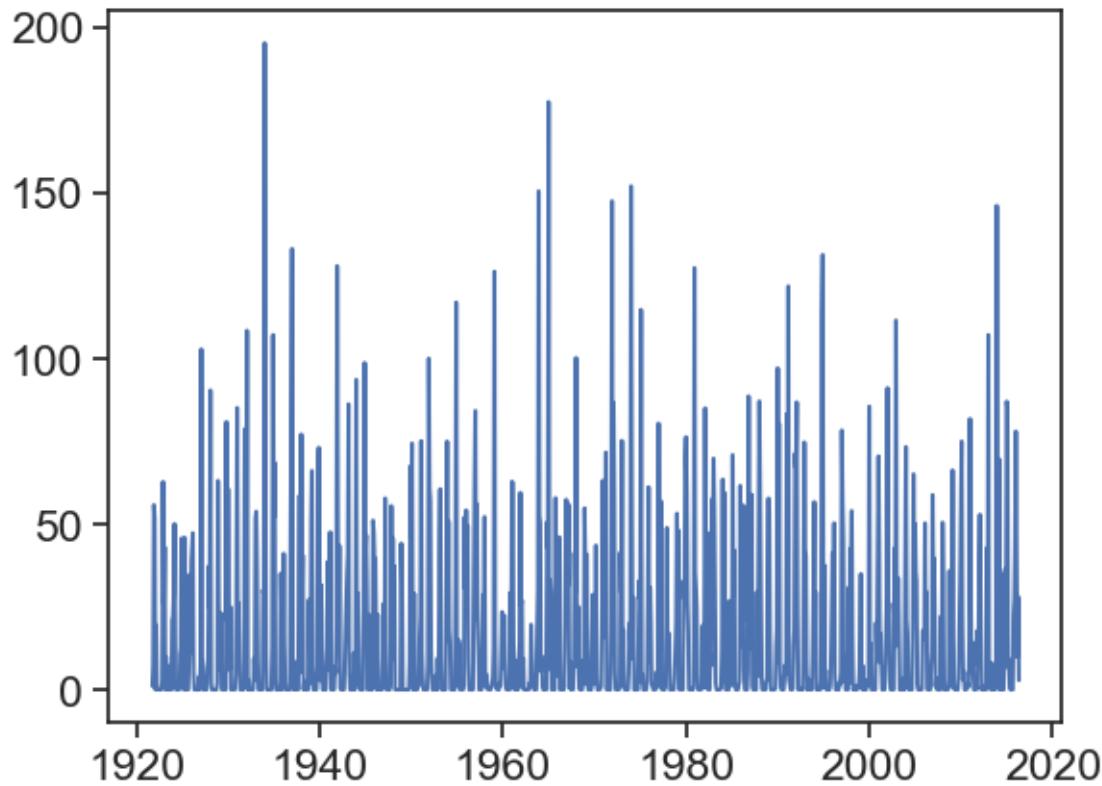
ALWAYS look at your data to see if it looks good.

NEVER mindlessly run code on data you don't know to be good.

*Of course, if you have a lot of data, you can't look at it with your eyes, and smart methods can be devised to increase the chances that everything is alright.

Download data for Beer Sheva, ID IS000051690.

```
download_data('BEER_SHEVA', 'IS000051690')
df = pd.read_csv('BEER_SHEVA_monthly.csv', sep=",")
# make 'DATE' the dataframe index
df['DATE'] = pd.to_datetime(df['DATE'])
df = df.set_index('DATE')
plt.plot(df['PRCP']);
```



That's much better! We need to aggregate all data from each month, so we can calculate monthly averages. How to do that?

```
group_by_month = df['PRCP'].groupby(df.index.month)
df_beersheva = (group_by_month
                 .mean()
                 .to_frame()
                 )
df_beersheva = df_beersheva.reset_index()
df_beersheva.columns = ['month number', 'monthly rainfall (mm)']
df_beersheva
```

| | month number | monthly rainfall (mm) |
|---|--------------|-----------------------|
| 0 | 1 | 48.743158 |
| 1 | 2 | 37.347368 |
| 2 | 3 | 26.551579 |
| 3 | 4 | 9.038947 |
| 4 | 5 | 2.735789 |

| | month number | monthly rainfall (mm) |
|----|--------------|-----------------------|
| 5 | 6 | 0.013830 |
| 6 | 7 | 0.000000 |
| 7 | 8 | 0.002128 |
| 8 | 9 | 0.271277 |
| 9 | 10 | 6.669474 |
| 10 | 11 | 21.850526 |
| 11 | 12 | 41.786316 |

`groupby` is a very powerful tool, but it takes some time to get used to it. We usually make operations on the groupby object like this:

```
df_beersheva = (df['PRCP']
                 .groupby(df.index.month)
                 .mean()
                 .to_frame()
             )
```

If you try to see the object `group_by_month` you won't see anything. This object waits for further instructions to be useful. Another way of understanding what's going on with this operation is to see with our eyes one of the groups:

```
group_by_month.get_group(3)
```

If you want to calculate the same averages using a loop instead of `groupby`, then you can do the following:

```
# choose only the precipitation column
df_month = df['PRCP']
# calculate monthly mean
monthly_mean = np.array([]) # empty array
month_numbers = np.arange(1,13)

for m in month_numbers:      # cycle over months (1, 2, 3, etc)
    this_month_all_indices = (df_month.index.month == m)      # indices in df_month belonging to month m
    this_month_mean = df_month[this_month_all_indices].mean() # this is the monthly mean
    monthly_mean = np.append(monthly_mean, this_month_mean)   # append

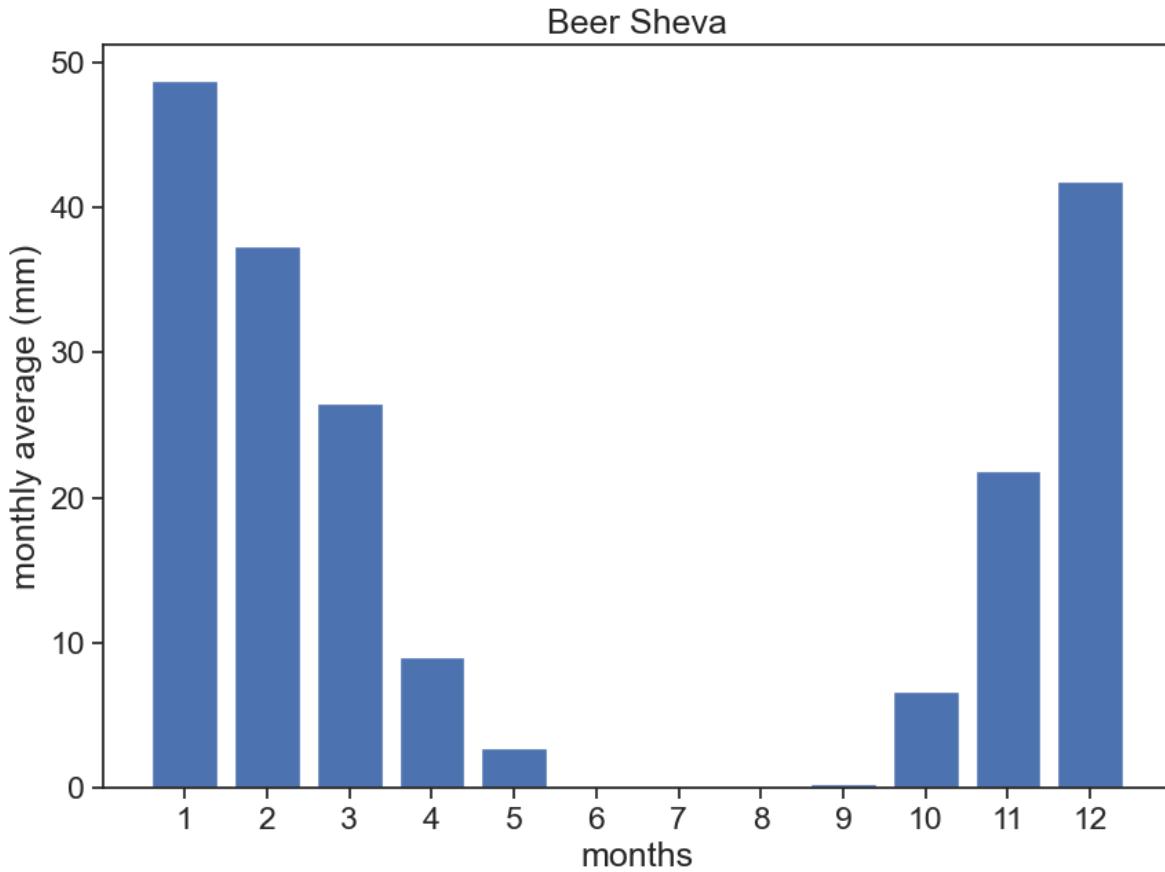
df_beersheva = pd.DataFrame({'monthly rainfall (mm)':monthly_mean,
                             'month number':month_numbers})
```

```
    })
df_beersheva
```

| | monthly rainfall (mm) | month number |
|----|-----------------------|--------------|
| 0 | 48.743158 | 1 |
| 1 | 37.347368 | 2 |
| 2 | 26.551579 | 3 |
| 3 | 9.038947 | 4 |
| 4 | 2.735789 | 5 |
| 5 | 0.013830 | 6 |
| 6 | 0.000000 | 7 |
| 7 | 0.002128 | 8 |
| 8 | 0.271277 | 9 |
| 9 | 6.669474 | 10 |
| 10 | 21.850526 | 11 |
| 11 | 41.786316 | 12 |

Plot the data and see if it makes sense. Try to get a figure like this one.

```
fig, ax = plt.subplots(figsize=(10,7))
ax.bar(df_beersheva['month number'], df_beersheva['monthly rainfall (mm)'])
ax.set(xlabel="months",
       ylabel="monthly average (mm)",
       title="Beer Sheva",
       xticks=df_beersheva['month number'],
       xticklabels=df_beersheva['month number']);
# plt.savefig("beersheva_monthly_average.png")
```



Let's calculate now the Walsh and Lawler Seasonality Index: ledbris (2010), Walsh and Lawler (1981).

Write a function that receives a dataframe like the one we have just created, and returns the seasonality index.

R = mean annual precipitation

m_i = precipitation mean for month i

$$SI = \frac{1}{R} \sum_{n=1}^{n=12} \left| m_i - \frac{R}{12} \right|$$

| SI | Precipitation Regime |
|-----------|---|
| <0.19 | Precipitation spread throughout the year |
| 0.20-0.39 | Precipitation spread throughout the year, but with a definite wetter season |
| 0.40-0.59 | Rather seasonal with a short dry season |

| SI | Precipitation Regime |
|-----------|--|
| 0.60-0.79 | Seasonal |
| 0.80-0.99 | Marked seasonal with a long dry season |
| 1.00-1.19 | Most precipitation in < 3 months |

```
def walsh_index(df):
    mi = df["monthly rainfall (mm)"]
    R = df["monthly rainfall (mm)"].sum()
    SI = np.sum(np.abs(mi - R/12)) / R
    return SI
beersheva_SI = walsh_index(df_beersheva)
print(f"Beer Sheva, SI = {beersheva_SI:.2f}")
```

Beer Sheva, SI = 0.97

5 Interannual variability of precipitation

```
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters() # datetime converter for a matplotlib
from calendar import month_abbr
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import urllib.request
import matplotlib.dates as mdates

ModuleNotFoundError: No module named 'seaborn'

df = pd.read_csv("TEL_AVIV_READING_monthly.csv",
                  sep=",",
                  parse_dates=['DATE'],
                  index_col='DATE'
                 )

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)

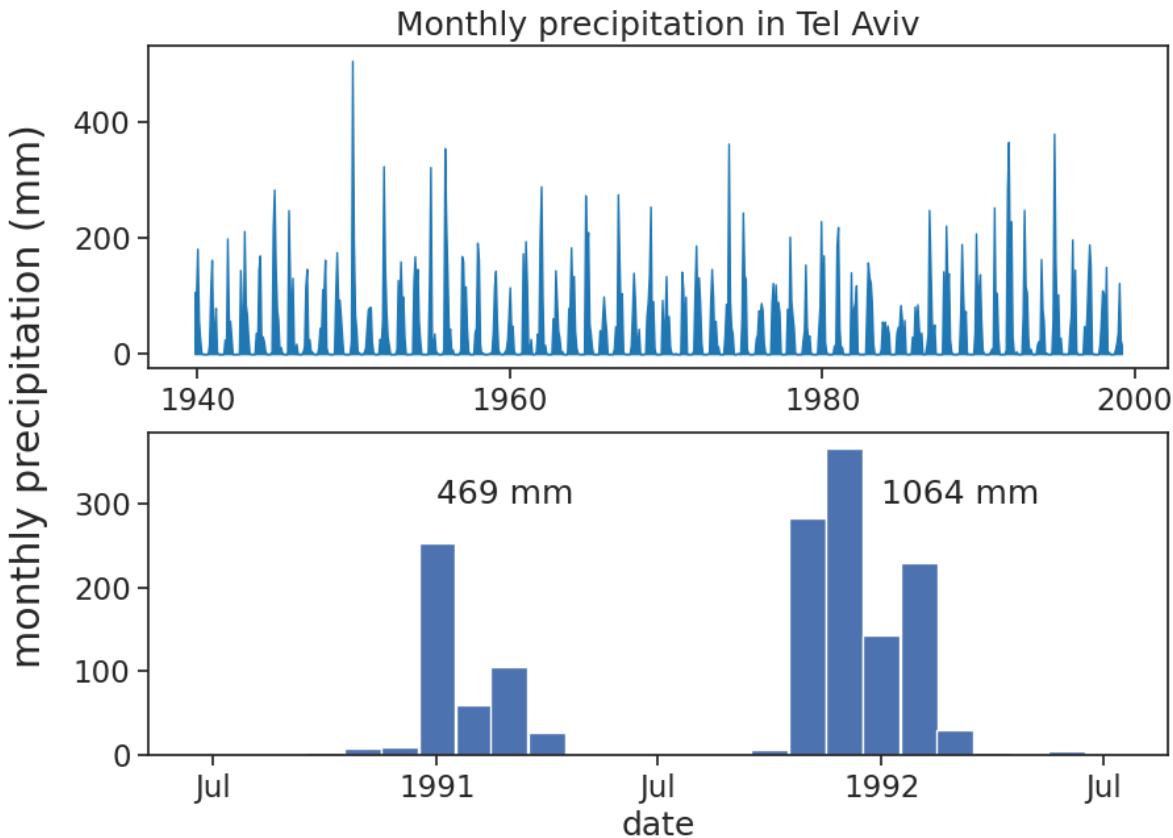
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,7))

# plot precipitation
ax1.fill_between(df.index, df['PRCP'], 0, color='tab:blue')
df_1990_1992 = df.loc['1990-07-01':'1992-07-01']
ax2.bar(df_1990_1992.index, df_1990_1992['PRCP'], width=30)
```

```
# adjust labels, ticks, title, etc
ax1.set_title("Monthly precipitation in Tel Aviv")
# ax2.tick_params(axis='x', rotation=45)
ax2.set_xlabel("date")
concise(ax1)
concise(ax2)

# common y label between the two panels:
fig.supylabel('monthly precipitation (mm)')

# write yearly rainfall
rain_1990_1991 = df.loc['1990-07-01':'1991-07-01','PRCP'].sum()
rain_1991_1992 = df.loc['1991-07-01':'1992-07-01','PRCP'].sum()
ax2.text('1991-01-01', 300, "{:.0f} mm".format(rain_1990_1991))
ax2.text('1992-01-01', 300, "{:.0f} mm".format(rain_1991_1992))
pass
# save figure
# plt.savefig("monthly_tel_aviv_1940-1999.png")
```



Let's aggregate (resample) precipitation according to the hydrological year.

```
# read more about resampling options
# https://pandas.pydata.org/pandas-docs/version/0.12.0/timeseries.html#offset-aliases
# also, annual resampling can be anchored to the end of specific months:
# https://pandas.pydata.org/pandas-docs/version/0.12.0/timeseries.html#anchored-offsets
df_year = df['PRCP'].resample('YE-SEP').sum().to_frame() # yearly frequency, anchored end of
df_year.columns = ['rain (mm)'] # rename 'PRCP' column to 'rain (mm)'
# the last year is the sum of only one month (November), let's take it out
df_year = df_year.iloc[:-1] # exclude last row
```

```
fig, ax = plt.subplots(figsize=(10,7))

# plot YEARLY precipitation
ax.bar(df_year.index, df_year['rain (mm)'],
       width=365, align='edge', color="tab:blue")

# plot mean
```

```

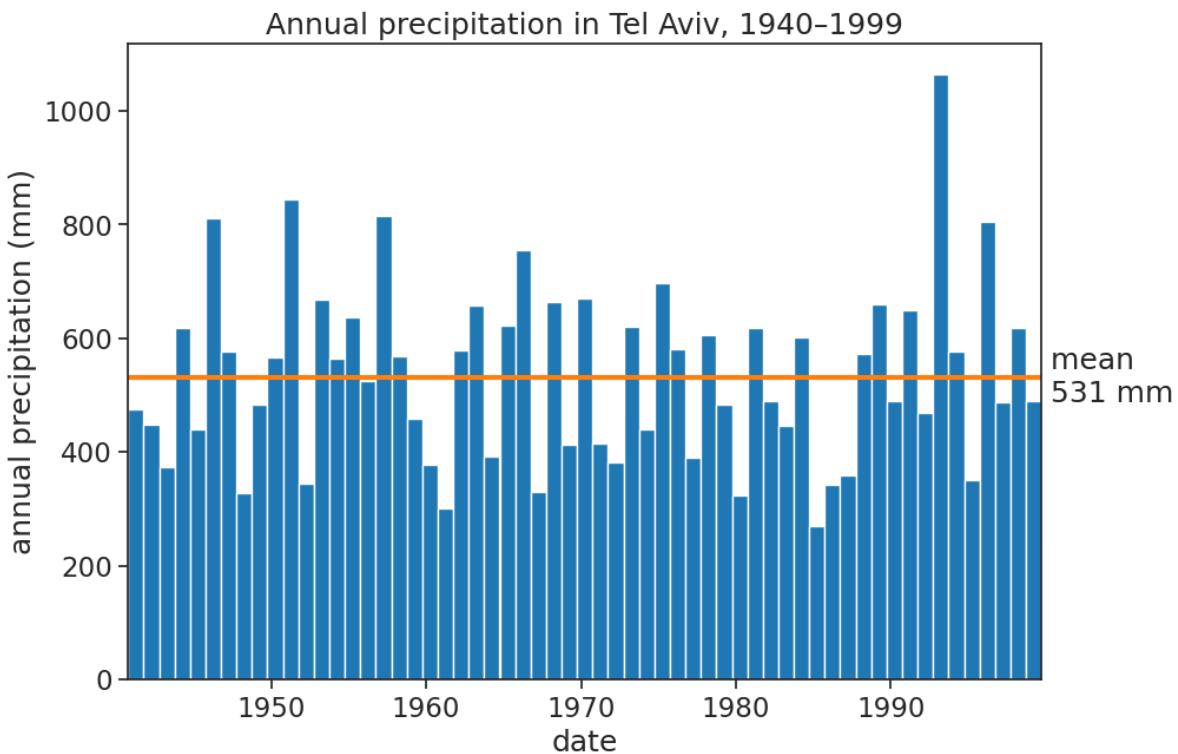
rain_mean = df_year['rain (mm)'].mean()
ax.plot(df_year*0 + rain_mean, linewidth=3, color="tab:orange")

# adjust labels, ticks, title, etc
ax.set(title="Annual precipitation in Tel Aviv, 1940-1999",
       xlabel="date",
       ylabel="annual precipitation (mm)",
       xlim=[df_year.index[0], df_year.index[-1]])
)

# write mean on the right
ax.text(df_year.index[-1], rain_mean, " mean\n {:.0f} mm".format(rain_mean),
        horizontalalignment="left", verticalalignment="center");

# save figure
# plt.savefig("annual_tel_aviv_with_mean.png")

```



```
fig, ax = plt.subplots(figsize=(10,7))
```

```

# calculate mean and standard deviation
rain_mean = df_year['rain (mm)'].mean()
rain_std = df_year['rain (mm)'].std()

# plot histogram
b = np.arange(0, 1101, 100) # bins from 0 to 55, width = 5
ax.hist(df_year, bins=b)

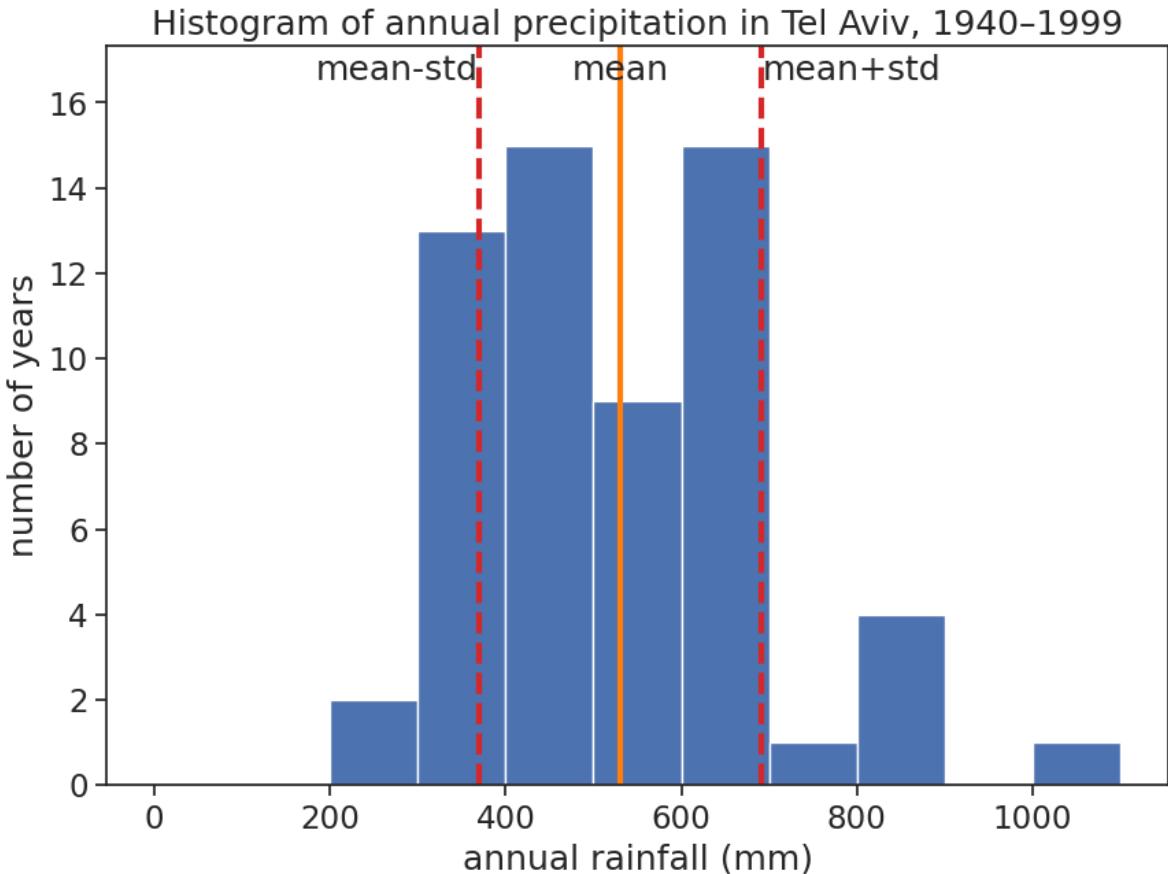
# plot vertical lines with mean, std, etc
ylim = np.array(ax.get_ylimits())
ylim[1] = ylim[1]*1.1
ax.plot([rain_mean]*2, ylim, linewidth=3, color="tab:orange")
ax.plot([rain_mean+rain_std]*2, ylim, linewidth=3, linestyle="--", color="tab:red")
ax.plot([rain_mean-rain_std]*2, ylim, linewidth=3, linestyle="--", color="tab:red")
ax.set_ylimits(ylim)

# write mean, std, etc
ax.text(rain_mean, ylim[1]*0.99, "mean",
        horizontalalignment="center",
        verticalalignment="top",
        )
ax.text(rain_mean+rain_std, ylim[1]*0.99, "mean+std",
        horizontalalignment="left",
        verticalalignment="top",
        )
ax.text(rain_mean-rain_std, ylim[1]*0.99, "mean-std",
        horizontalalignment="right",
        verticalalignment="top",
        )

# adjust labels, ticks, title, limits, etc
ax.set(title="Histogram of annual precipitation in Tel Aviv, 1940-1999",
       xlabel="annual rainfall (mm)",
       ylabel="number of years"
       );

# save figure
# plt.savefig("histogram_tel_aviv_with_mean_and_std.png")

```



```

df_eilat = pd.read_csv("Eilat_monthly.csv",
                      sep=",",
                      parse_dates=['DATE'],
                      index_col='DATE'
                     )
df_year_eilat = df_eilat['PRCP'].resample('YE-SEP').sum().to_frame() # yearly frequency, and
df_year_eilat.columns = ['rain (mm)'] # rename 'PRCP' column to 'rain (mm)'
df_year_eilat = df_year_eilat.iloc[2:-5] # exclude first two and last two years

fig, ax = plt.subplots(figsize=(10,7))

ax.plot(df_year['rain (mm)'], label="Tel Aviv", color="xkcd:hot pink")
ax.plot(df_year_eilat['rain (mm)'], label="Eilat", color="black")
ax.legend(frameon=False)

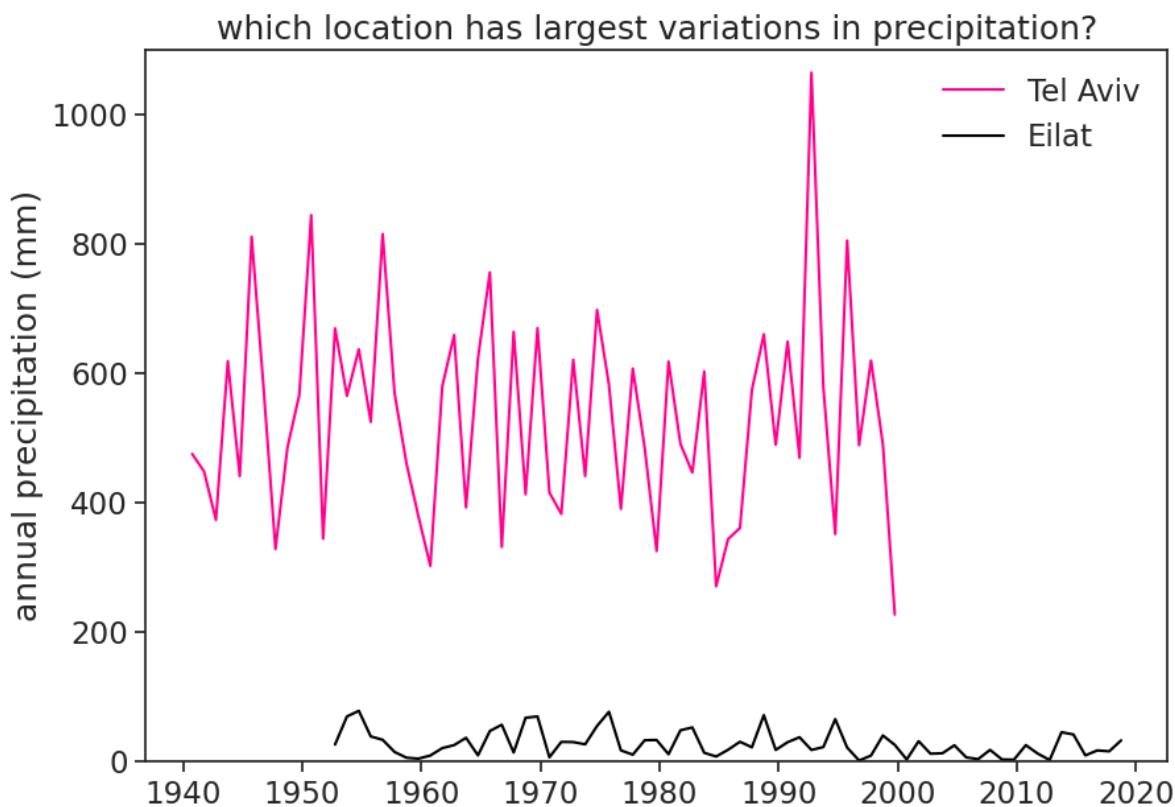
ax.set(ylabel="annual precipitation (mm)",
       title="which location has largest variations in precipitation?",
```

```

    ylim=[0, 1100])

mean_telaviv = df_year['rain (mm)'].mean()
mean_eilat = df_year_eilat['rain (mm)'].mean()
std_telaviv = df_year['rain (mm)'].std()
std_eilat = df_year_eilat['rain (mm)'].std()
cv_telaviv = std_telaviv / mean_telaviv
cv_eilat = std_eilat / mean_eilat

```



```

fig, ax = plt.subplots(1 ,2, figsize=(10,5))

ax[0].plot(df_year['rain (mm)'], label="Tel Aviv", color="xkcd:hot pink")
ax[0].set(ylim=[0,2*mean_telaviv],
          yticks=[0,mean_telaviv,2*mean_telaviv],
          yticklabels=['0', 'mean(Tel Aviv)', r'$2\times$ mean(Tel Aviv)'],
          title="Tel Aviv")
concise(ax[0])

```

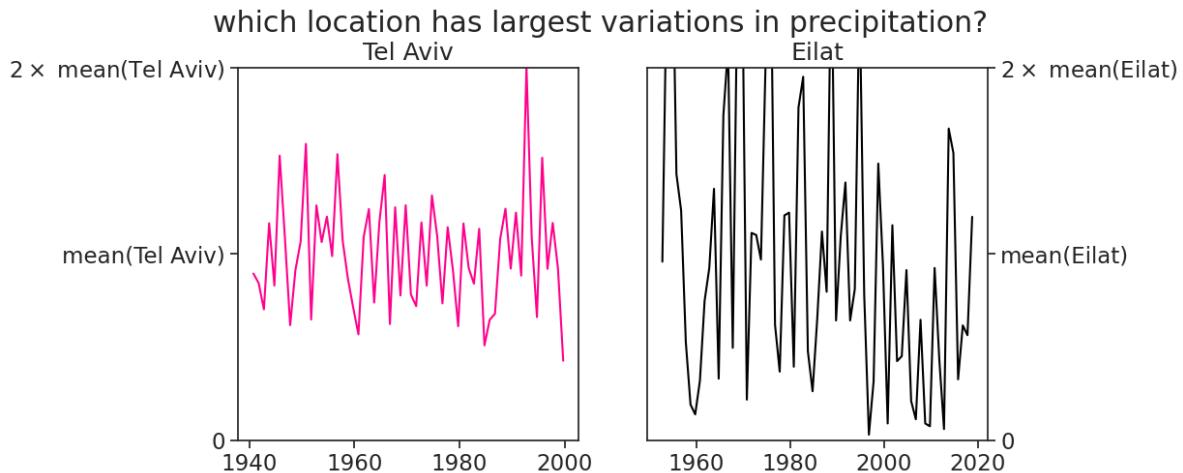
```

ax[1].plot(df_year_eilat['rain (mm)'], label="Eilat", color="black")
ax[1].set(ylim=[0,2*mean_eilat],
           yticks=[0,mean_eilat,2*mean_eilat],
           yticklabels=['0', 'mean(Eilat)', r'$2\times$ mean(Eilat)'],
           title="Eilat")
ax[1].yaxis.set_label_position("right")
ax[1].yaxis.tick_right()
concise(ax[1])

fig.suptitle('which location has largest variations in precipitation?', y=1.0);

```

Text(0.5, 1.0, 'which location has largest variations in precipitation?')



5.1 coefficient of variation

$\langle P \rangle$ = average precipitation
 σ = standard deviation

$$CV = \frac{\sigma}{\langle P \rangle}$$

The coefficient of variation (dimensionless) quantifies the variation (std) magnitude with respect to the mean. In the examples above, although Tel Aviv has a much higher standard deviation in annual precipitation, the spread of precipitation in Eilat is **much larger**, considered relative to its average.

```
print(f"Tel Aviv:\tstd = {std_telaviv:.2f} mm\t\tCV = {cv_telaviv:.2f}")  
print(f"Eilat:\tstd = {std_eilat:.2f} mm\t\tCV = {cv_eilat:.2f}")
```

```
Tel Aviv:    std = 161.19 mm      CV = 0.30  
Eilat:      std = 20.49 mm      CV = 0.77
```

Another way to understand the CV: for gaussian (normal) distributions, 67% of the data lies 1 std from the mean. **Assuming** that the annual rainfall for Tel Aviv and Eilat roughly follows a gaussian distribution, we could say that:

- Tel Aviv: about 67% of the annual precipitation is no more than 30% from the average.
- Eilat: about 67% of the annual precipitation is no more than 77% from the average.

5.2 climate normals

Precipitation averages are usually calculated for time intervals of 30 years. According to the [National Oceanic and Atmospheric Administration — NOAA](#):

Normals serve two purposes: a reference period for monitoring current weather and climate, and a good description of the expected climate at a location over the seasons. They provide a basis for determining whether today's weather is warmer or colder, wetter or drier. They also can be used to plan for conditions beyond the time span of reliable weather forecasts. A 30-year time period was chosen by the governing body of international meteorology in the 1930s, so the first normals were for 1901-1930, the longest period for which most countries had reliable climate records. International normals were called for in 1931-1960 and 1961-1990, but many countries updated normals more frequently, every 10 years, so as to keep them up to date. In 2015 this was made the WMO standard, so all countries will be creating normals for 1991-2020.

5.3 running averages

```
fig, ax = plt.subplots(figsize=(10,7))  
  
ax.plot(df_year['rain (mm)'], color="lightgray")  
  
# windows of length 30 years  
windows = [[1940,1969], [1970,1999]]
```

```

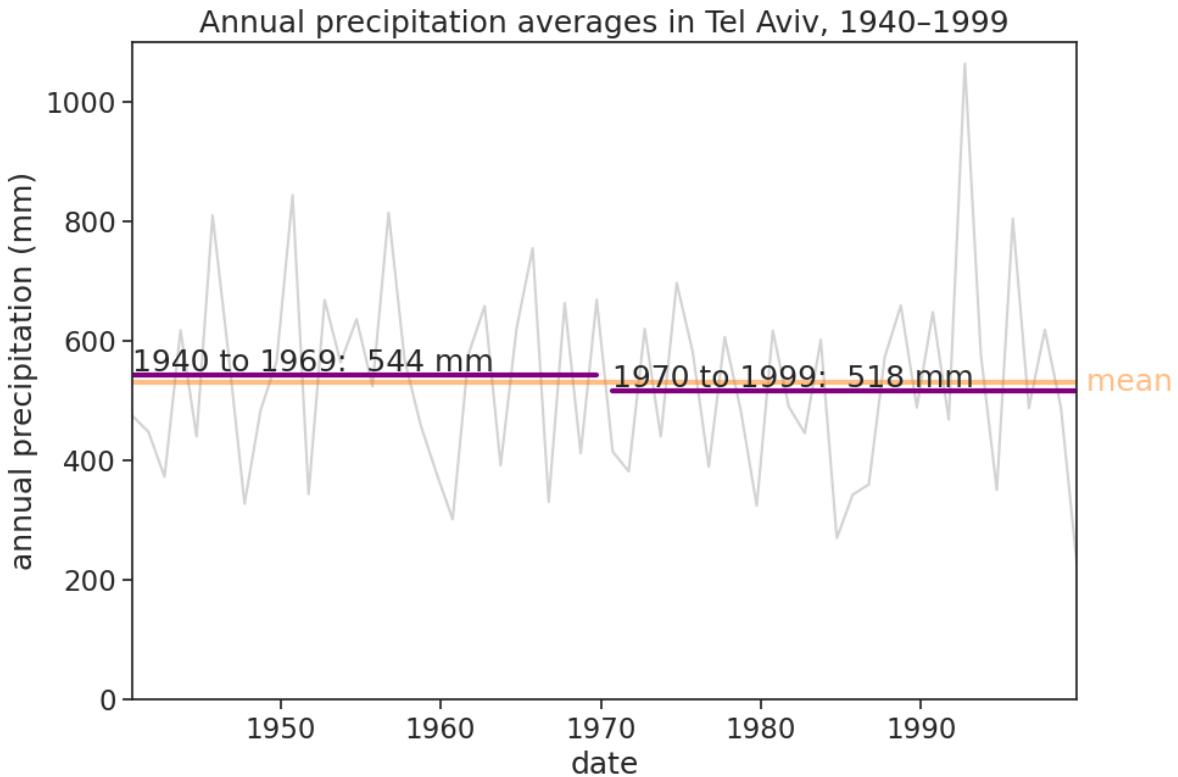
for window in windows:
    start_date = f"{window[0]:d}-09-30"
    end_date = f"{window[1]:d}-09-30"
    window_mean = df_year['rain (mm)'][start_date:end_date].mean()
    ax.plot(df_year[start_date:end_date]*0+window_mean, color="purple", linewidth=3)
    ax.text(start_date, window_mean+5, f'{window[0]} to {window[1]}: {window_mean:.0f} mm',)

# plot mean
rain_mean = df_year['rain (mm)').mean()
ax.plot(df_year*0 + rain_mean, linewidth=3, color="tab:orange", alpha=0.5)
ax.text(df_year.index[-1], rain_mean, " mean".format(rain_mean),
        horizontalalignment="left", verticalalignment="center",
        color="tab:orange", alpha=0.5)

# adjust labels, ticks, title, limits, etc
ax.set(title="Annual precipitation averages in Tel Aviv, 1940-1999",
       xlabel="date",
       ylabel="annual precipitation (mm)",
       xlim=[df_year.index[0], df_year.index[-1]],
       ylim=[0, 1100],
       );

```

save figure
plt.savefig("mean_tel_aviv_2_windows.png")



```

fig, ax = plt.subplots(figsize=(10,7))

# windows of length 30 years
windows = [[x,x+29] for x in [1940,1950,1960,1970]]
for window in windows:
    start_date = f"{window[0]}-09-30"
    end_date = f"{window[1]}-09-30"
    window_mean = df_year['rain (mm)'][start_date:end_date].mean()
    ax.plot(df_year[start_date:end_date]*0+window_mean, color="purple", linewidth=3)
    ax.text(start_date, window_mean+0.5, f"{window[0]} to {window[1]}: {window_mean:.0f} mm"

# plot mean
rain_mean = df_year['rain (mm)'].mean()
ax.plot(df_year*0 + rain_mean, linewidth=3, color="tab:orange", alpha=0.5)
ax.text(df_year.index[-1], rain_mean, " mean".format(rain_mean),
        horizontalalignment="left", verticalalignment="center",
        color="tab:orange", alpha=0.5)

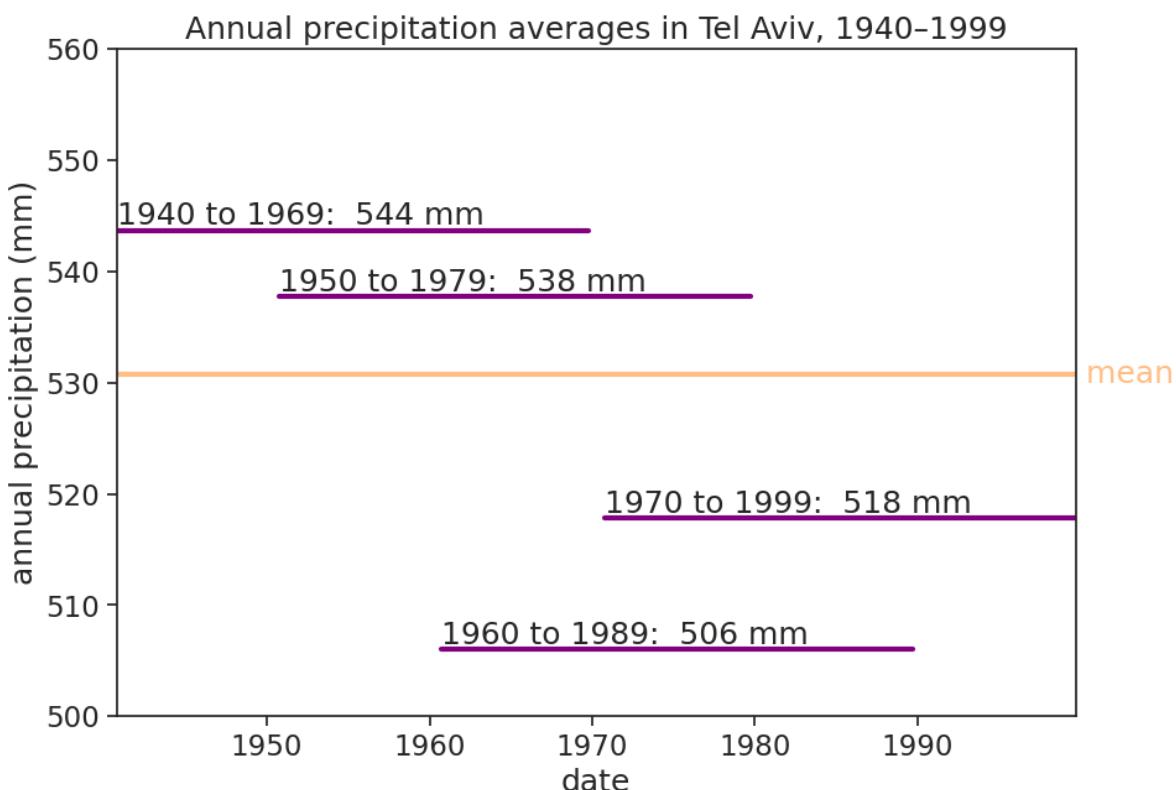
# adjust labels, ticks, title, limits, etc

```

```

ax.set(title="Annual precipitation averages in Tel Aviv, 1940-1999",
       xlabel="date",
       ylabel="annual precipitation (mm)",
       xlim=[df_year.index[0], df_year.index[-1]],
       ylim=[500, 560],
       );
# save figure
# plt.savefig("mean_tel_aviv_2_windows.png")

```



```

import altair as alt

# Custom theme for readability
def readable():
    return {
        "config" : {
            "title": {'fontSize': 16},
            "axis": {

```

```

        "labelFontSize": 16,
        "titleFontSize": 16,
    },
    "header": {
        "labelFontSize": 14,
        "titleFontSize": 14,
    },
    "legend": {
        "labelFontSize": 14,
        "titleFontSize": 14,
    },
    "mark": {
        'fontSize': 14,
        "tooltip": {"content": "encoding"}, # enable tooltips
    },
},
}

alt.themes.register('readable', readable)
alt.themes.enable('readable')

# Altair only recognizes column data; it ignores index values. You can plot the index data by
source = df_year.reset_index()
brush = alt.selection_interval(encodings=['x'])

# T: temporal, a time or date value
# Q: quantitative, a continuous real-valued quantity
# https://altair-viz.github.io/user_guide/encoding.html#encoding-data-types
bars = alt.Chart().mark_bar().encode(
    x=alt.X('DATE:T', axis=alt.Axis(title='date')),
    y=alt.Y('rain (mm):Q', axis=alt.Axis(title='annual precipitation (mm) and average')),
    opacity=alt.condition(brush, alt.OpacityValue(1), alt.OpacityValue(0.2)),
).add_params(
    brush
).properties(
    title='Select year range and drag for rolling average of annual precipitation in Tel Aviv'
).properties(
    width=600,
    height=400
)

line = alt.Chart().mark_rule(color='orange').encode(

```

```

y='mean(rain (mm)) :Q',
size=alt.SizeValue(3)
).transform_filter(
    brush
)

alt.layer(bars, line, data=source)

alt.LayerChart(...)

fig, ax = plt.subplots(figsize=(10,7))

ax.plot(df_year['rain (mm)'], color="lightgray")

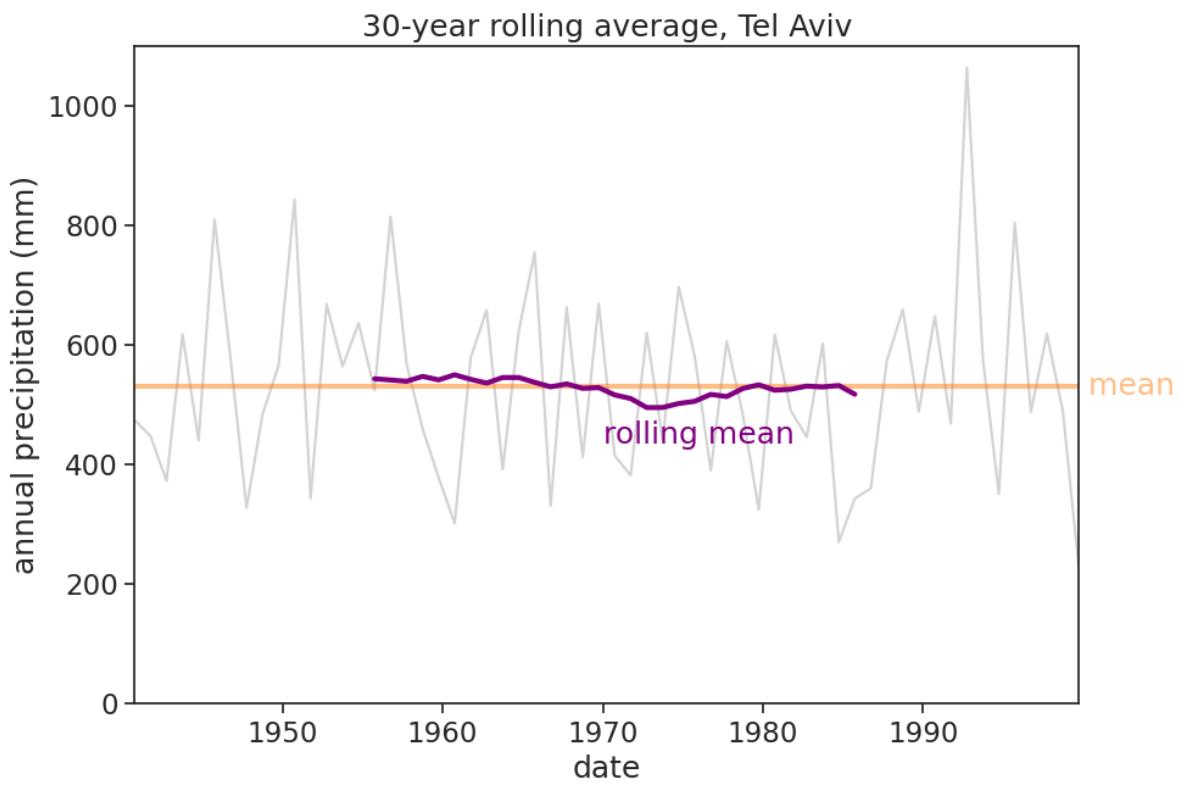
# plot rolling mean
rolling_mean = df_year.rolling(window=30, center=True).mean()
ax.plot(rolling_mean, linewidth=3, color="purple", zorder=5)
ax.text(pd.to_datetime("1970"), 450, "rolling mean".format(rain_mean),
        horizontalalignment="left", verticalalignment="center",
        color="purple",)

# plot mean
ax.plot(df_year*0 + rain_mean, linewidth=3, color="tab:orange", alpha=0.5)
ax.text(df_year.index[-1], rain_mean, " mean".format(rain_mean),
        horizontalalignment="left", verticalalignment="center",
        color="tab:orange", alpha=0.5);

ax.set(title="30-year rolling average, Tel Aviv",
       xlabel="date",
       ylabel="annual precipitation (mm)",
       ylim=[0, 1100],
       xlim=[df_year.index[0], df_year.index[-1]])
);

# save figure
# plt.savefig("rolling_average_tel_aviv.png")

```



6 Exercises

Import relevant packages

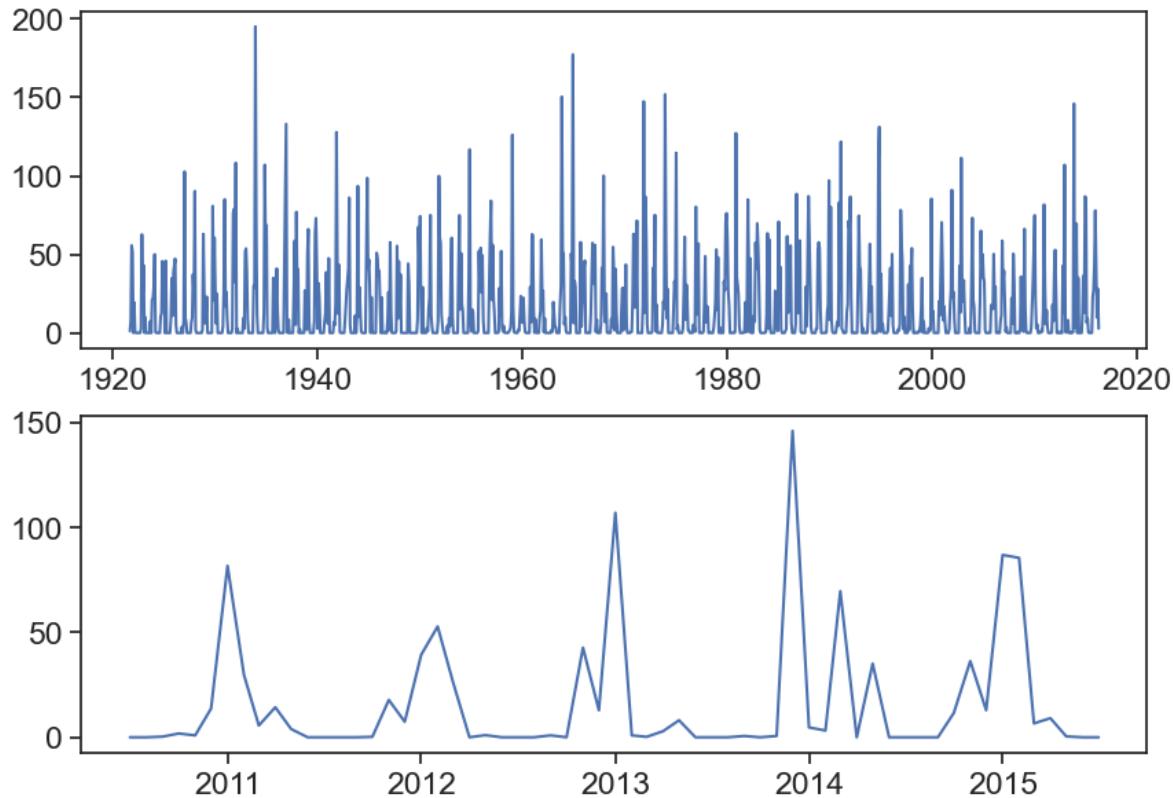
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

Plot monthly rainfall for your station.

Load the data into a dataframe, and before you continue with the analysis, plot the rainfall data, to see how it looks like.

```
df = pd.read_csv('BEER_SHEVA_monthly.csv', sep=",")
# make 'DATE' the dataframe index
df['DATE'] = pd.to_datetime(df['DATE'])
df = df.set_index('DATE')

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,7))
ax1.plot(df['PRCP'])
ax2.plot(df['PRCP']['2010-07-01':'2015-07-01'])
```



How to aggregate rainfall according to the hydrological year? We use the function `resample`.

read more about resampling options:

https://pandas.pydata.org/docs/user_guide/timeseries.html#offset-aliases

also, annual resampling can be anchored to the end of specific months: https://pandas.pydata.org/docs/user_guide/timeseries.html#anchored-offsets

```
# annual frequency, anchored 31 December
df_year_all = df['PRCP'].resample('YE').sum().to_frame()
# annual frequency, anchored 01 January
df_year_all = df['PRCP'].resample('YS').sum().to_frame()
# annual frequency, anchored end of September
df_year_all = df['PRCP'].resample('YE-SEP').sum().to_frame()
# rename 'PRCP' column to 'rain (mm)'
df_year_all.columns = ['rain (mm)']
df_year_all
```

| DATE | rain (mm) |
|------------|-----------|
| 1922-09-30 | 136.6 |
| 1923-09-30 | 144.5 |
| 1924-09-30 | 130.4 |
| 1925-09-30 | 165.3 |
| 1926-09-30 | 188.7 |
| ... | ... |
| 2012-09-30 | 145.7 |
| 2013-09-30 | 175.3 |
| 2014-09-30 | 259.2 |
| 2015-09-30 | 249.3 |
| 2016-09-30 | 257.6 |

You might need to exclude the first or the last line, since their data might have less than 12 months. For example:

```
# exclude 1st row
df_year = df_year_all.iloc[1:]
# exclude last row
df_year = df_year_all.iloc[:-1]
# exclude both 1st and last rows
df_year = df_year_all.iloc[1:-1]
df_year
```

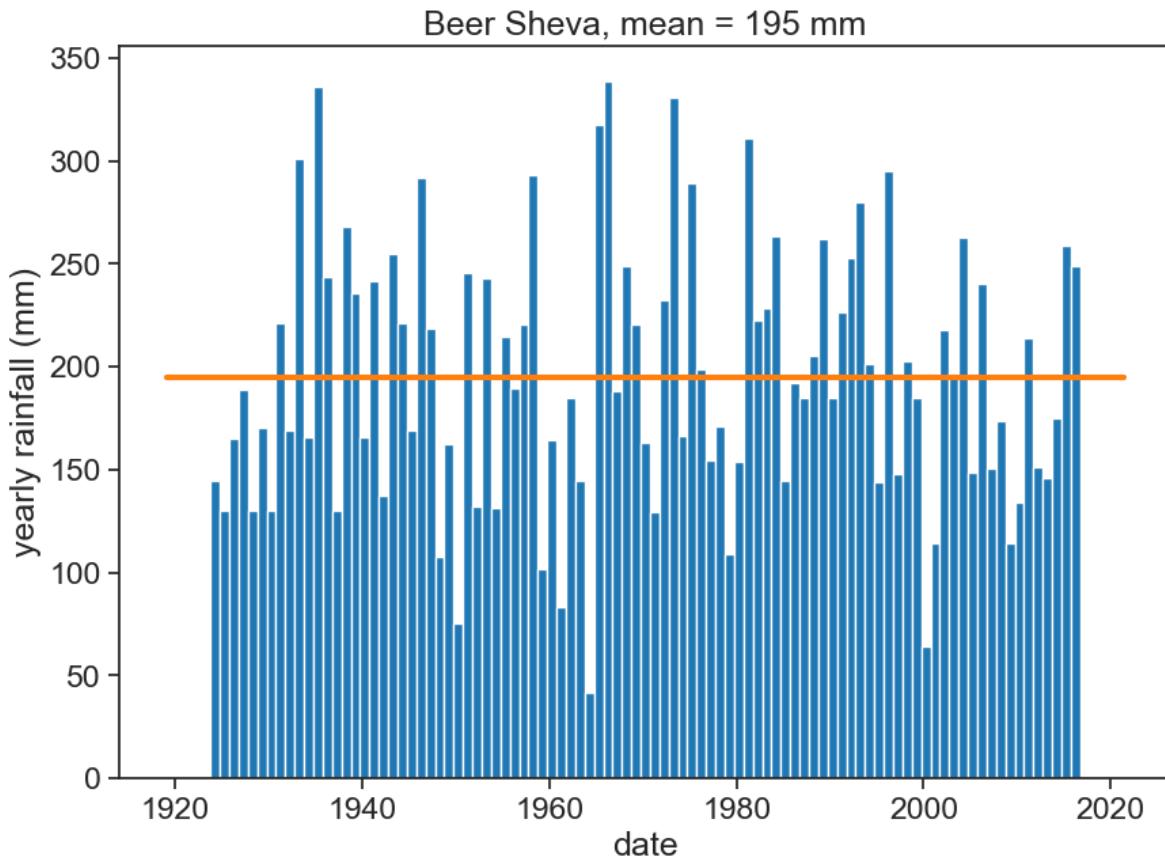
| DATE | rain (mm) |
|------------|-----------|
| 1923-09-30 | 144.5 |
| 1924-09-30 | 130.4 |
| 1925-09-30 | 165.3 |
| 1926-09-30 | 188.7 |
| 1927-09-30 | 130.2 |
| ... | ... |
| 2011-09-30 | 151.6 |
| 2012-09-30 | 145.7 |
| 2013-09-30 | 175.3 |
| 2014-09-30 | 259.2 |
| 2015-09-30 | 249.3 |

Calculate the average annual rainfall. Plot annual rainfall for the whole range, together with the average. You should get something like this:

```
fig, ax = plt.subplots(figsize=(10,7))

# plot YEARLY precipitation
ax.bar(df_year.index, df_year['rain (mm)'],
       width=365, align='edge', color="tab:blue")

# plot mean
rain_mean = df_year['rain (mm)'].mean()
ax.plot(ax.get_xlim(), [rain_mean]*2, linewidth=3, color="tab:orange")
ax.set(xlabel="date",
       ylabel="yearly rainfall (mm)",
       title=f"Beer Sheva, mean = {rain_mean:.0f} mm");
# save figure
# plt.savefig("beersheva_yearly_rainfall_1923_2016.png")
```



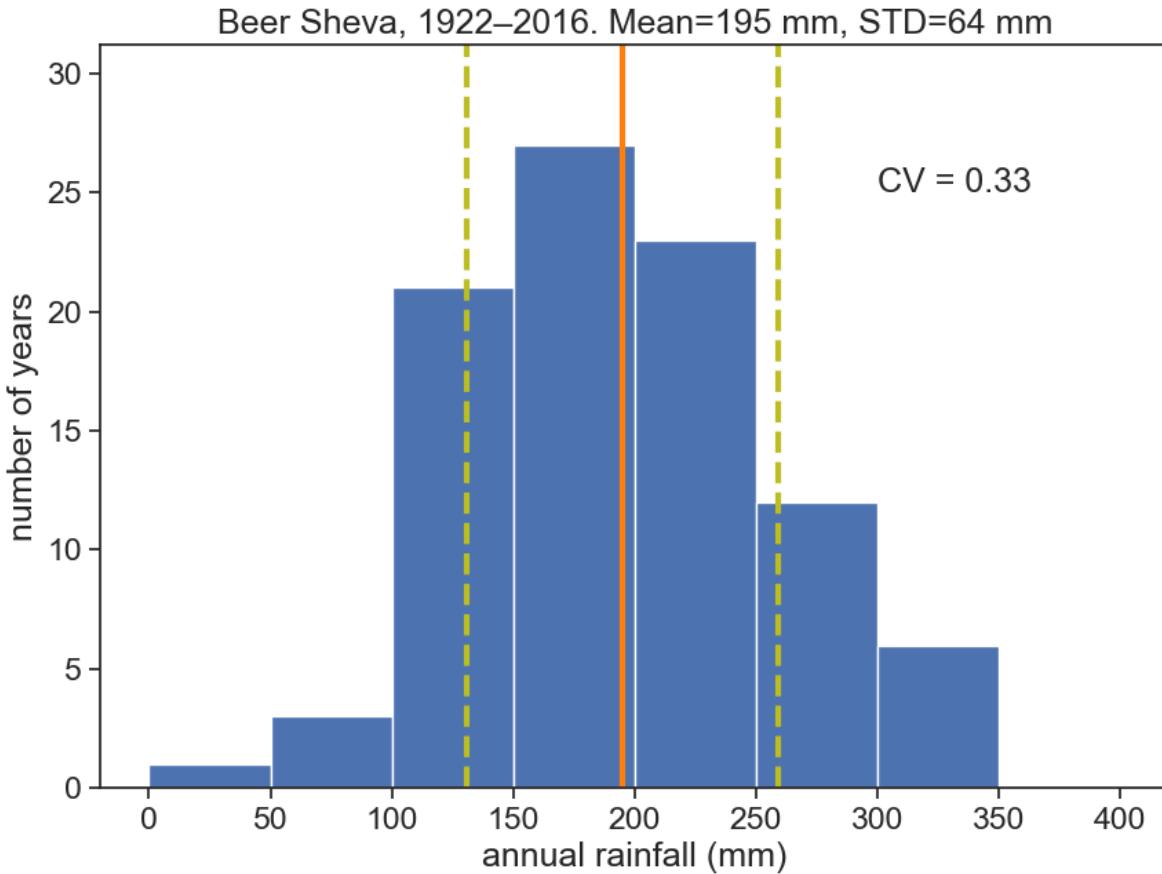
Plot a histogram of annual rainfall, with the mean and standard deviation. Calculate the coefficient of variation. Try to plot something like this:

```
fig, ax = plt.subplots(figsize=(10,7))

# calculate mean, standard deviation, CV
rain_mean = df_year['rain (mm)'].mean()
rain_std = df_year['rain (mm)'].std()
CV = rain_std/rain_mean
# plot histogram
b = np.arange(0, 401, 50) # bins from 0 to 400, width = 50
ax.hist(df_year['rain (mm)'], bins=b)

# plot vertical lines with mean, std, etc
ylim = np.array(ax.get_ylimits())
ylim[1] = ylim[1]*1.1
ax.plot([rain_mean]*2, ylim, linewidth=3, color="tab:orange")
ax.plot([rain_mean+rain_std]*2, ylim, linewidth=3, linestyle="--", color="tab:olive")
ax.plot([rain_mean-rain_std]*2, ylim, linewidth=3, linestyle="--", color="tab:olive")
ax.set(ylim=ylim,
       xlabel="annual rainfall (mm)",
       ylabel="number of years",
       title=f"Beer Sheva, 1922-2016. Mean={rain_mean:.0f} mm, STD={rain_std:.0f} mm")
ax.text(300, 25, f"CV = {CV:.2f}")
# plt.savefig("histogram_beersheva.png")
```

Text(300, 25, 'CV = 0.33')



Calculate the mean annual rainfall for various 30-year intervals

```
##### the hard way #####
# fig, ax = plt.subplots(figsize=(10,7))

# mean_30_59 = df_year.loc['1930-09-30':'1959-09-01','rain (mm)'].mean()
# mean_40_69 = df_year.loc['1940-09-30':'1969-09-01','rain (mm)'].mean()
# mean_50_79 = df_year.loc['1950-09-30':'1979-09-01','rain (mm)'].mean()
# mean_60_89 = df_year.loc['1960-09-30':'1989-09-01','rain (mm)'].mean()
# mean_70_99 = df_year.loc['1970-09-30':'1999-09-01','rain (mm)'].mean()
# mean_80_09 = df_year.loc['1980-09-30':'2009-09-01','rain (mm)'].mean()

# ax.plot([mean_30_59,
#           mean_40_69,
#           mean_50_79,
#           mean_60_89,
#           mean_70_99,
```

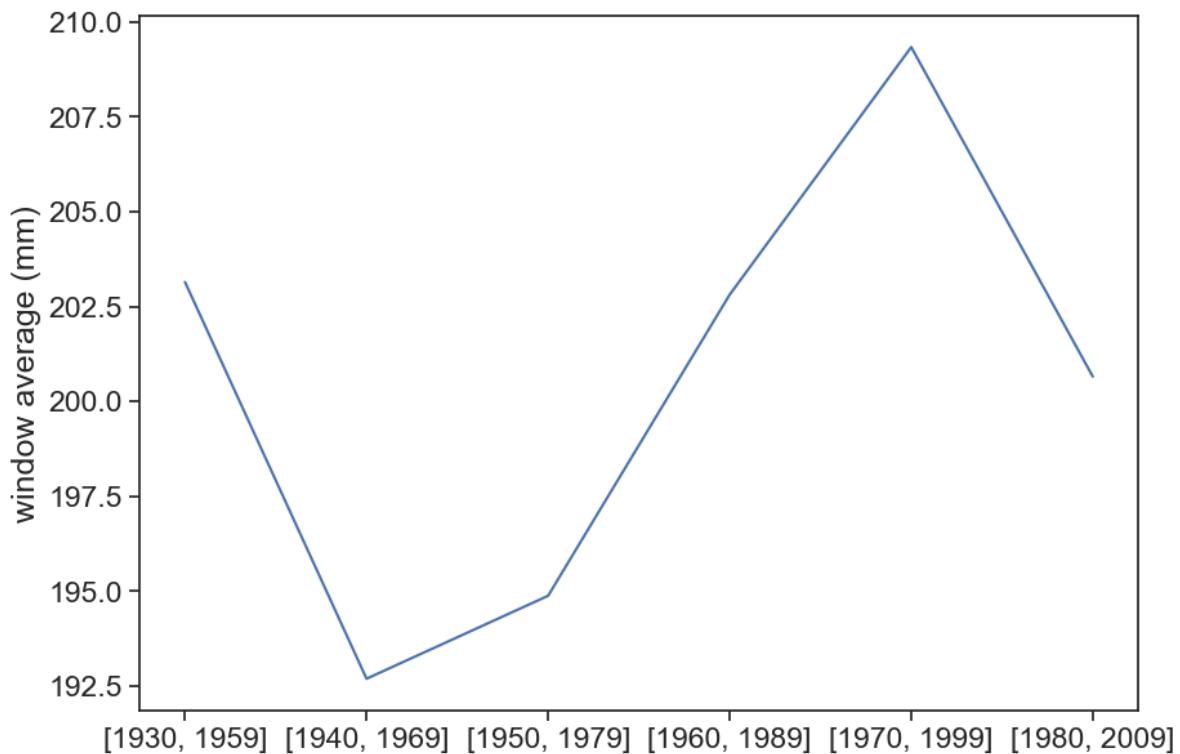
```

#           mean_80_09])

#####
# the easy way #####
fig, ax = plt.subplots(figsize=(10,7))

# use list comprehension
windows = [[x, x+29] for x in [1930,1940,1950,1960,1970,1980]]
mean = [df_year.loc[f'{w[0]}:{d}-09-30':f'{w[1]}:{d}-09-01','rain (mm)'].mean() for w in windows]
ax.plot(mean)
ax.set(xticks=np.arange(len(mean)),
       xticklabels=[str(w) for w in windows],
       ylabel="window average (mm)"
      );

```



7 Return Period

7.1 Bilbao, Spain



7.2 Today



7.3 August 1983



On Friday, August 26, 1983, Bilbao was celebrating its Aste Nagusia or Great Week, the main annual festivity in the city, when it and other municipalities of the Basque Country, Burgos, and Cantabria suffered devastating flooding due to heavy rains. In 24 hours, the volume of water registered 600 liters per square meter. Across all the affected areas, the weather service recorded 1.5 billion tons of water. In areas of Bilbao, the water reached a height of 5 meters (15 feet). Transportation, electricity and gas services, drinking water, food, telephone, and many other basic services were severely affected. 32 people died in Biscay, 4 people died in Cantabria, 2 people died in Alava, and 2 people died Burgos. 5 more people went missing.

7.4 How often will such rainfall happen?

How often does it rain 50 mm in 1 day? What about 100 mm in 1 day? How big is a “once-in-a-century event”?

Let's examine Bilbao's daily rainfall (mm), between 1947 to 2021

```

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import urllib.request
from matplotlib.dates import DateFormatter
import matplotlib.dates as mdates
import altair as alt
alt.data_transformers.disable_max_rows()
from scipy.stats import genextreme

```

```

def download_data(station_name, station_code):
    url_daily = 'https://www.ncei.noaa.gov/data/global-historical-climatology-network-daily/'
    url_monthly = 'https://www.ncei.noaa.gov/data/gsom/access/'
    # download daily data - uncomment to make this work
    urllib.request.urlretrieve(url_daily + station_code + '.csv',
                                station_name + '_daily.csv')
    # download monthly data
    urllib.request.urlretrieve(url_monthly + station_code + '.csv',
                                station_name + '_monthly.csv')
download_data('BILBAO', 'SPE00120611')

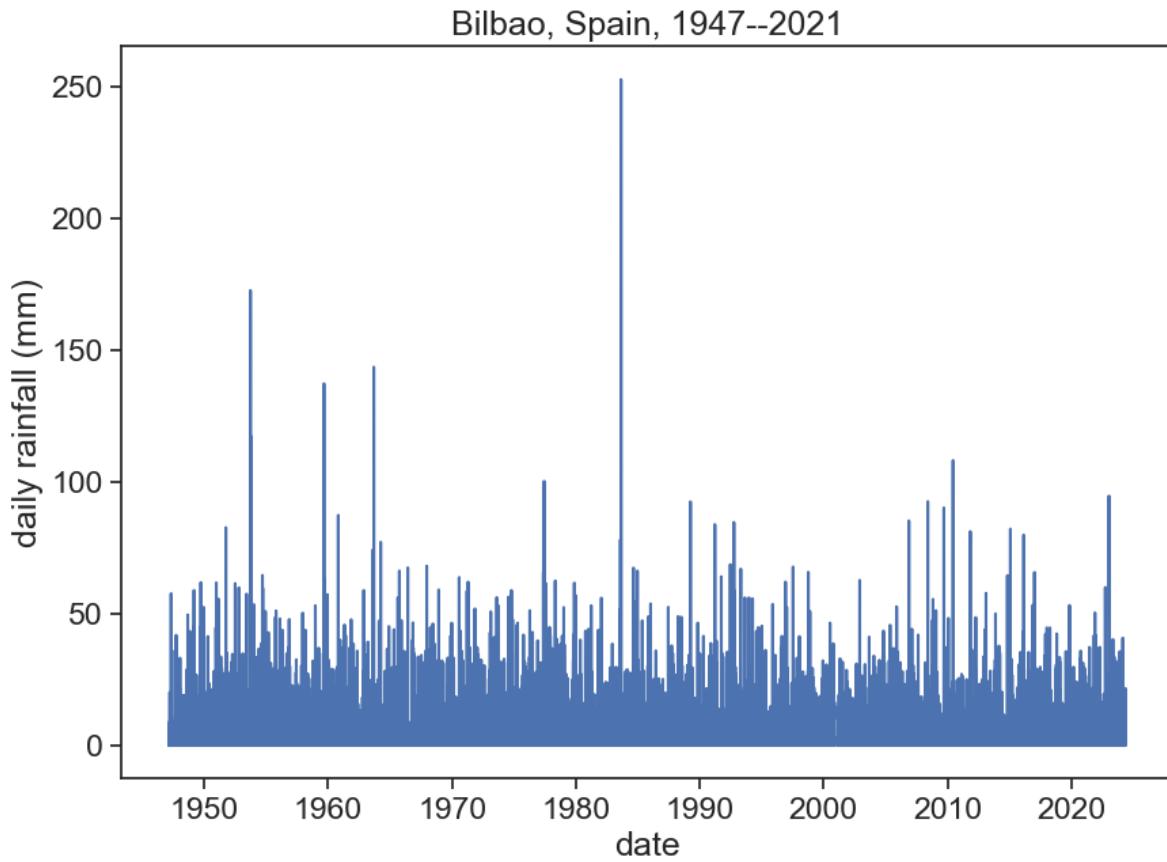
```

```

df = pd.read_csv('BILBAO_daily.csv',
                 sep=',',
                 parse_dates=['DATE'],
                 index_col='DATE')
# IMPORTANT!! daily precipitation data is in tenths of mm, divide by 10 to get it in mm.
df['PRCP'] = df['PRCP'] / 10

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(df['PRCP'])
ax.set(xlabel="date",
       ylabel="daily rainfall (mm)",
       title="Bilbao, Spain, 1947--2021")
ax.annotate("26 August 1983",
            xy=(pd.to_datetime('1983-08-26'), 2500), xycoords='data',
            xytext=(0.7, 0.95), textcoords='axes fraction',
            fontsize=16, va="center",
            arrowprops=dict(facecolor='black', shrink=0.05));

```



On the week of 22-28 August 1983, Bilbao's weather station measured 45 cm of rainfall!

```
fig, ax = plt.subplots(figsize=(10,7))
one_week = df.loc['1983-08-22':'1983-08-28', 'PRCP']
bars = ax.bar(one_week.index, one_week)
ax.set_xlabel("date")
ax.set_ylabel("daily rainfall (mm)")
ax.set_title("Bilbao, Spain, August 1983")

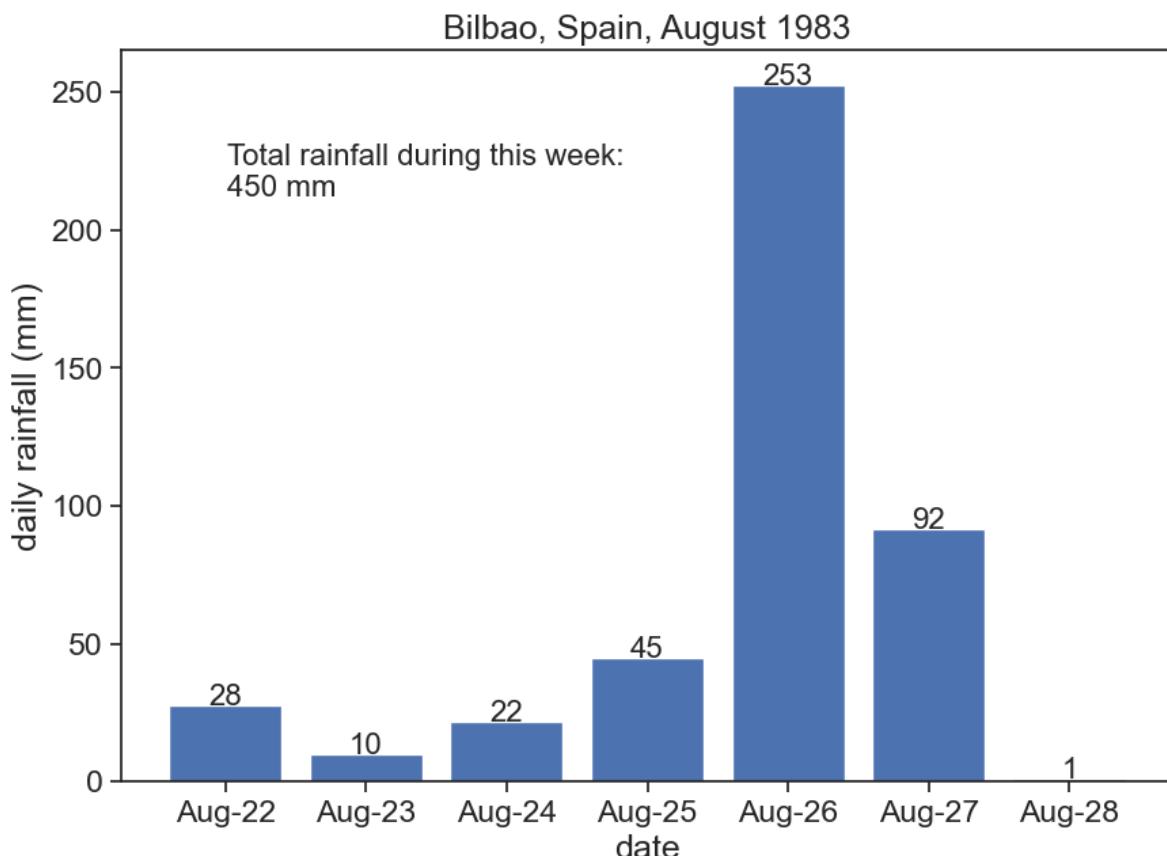
# write daily rainfall
for i in range(len(one_week)):
    ax.text(one_week.index[i], one_week.iloc[i], f"{one_week.iloc[i]:.0f}", ha="center", fontweight='bold')
    # ax.text(one_week.index[i], one_week[i], f"{one_week[i]:.0f}", ha="center", fontsize=16)

ax.text(0.1, 0.8, f"Total rainfall during this week:\n{n{one_week.sum():.0f} mm}",
        transform=ax.transAxes, fontsize=16)
```

```

# Define the date format
# https://docs.python.org/2/library/datetime.html#strftime-and-strptime-behavior
date_form = DateFormatter("%b-%d")
ax.xaxis.set_major_formatter(date_form)
# Ensure a major tick for each day using (interval=1)
# https://matplotlib.org/stable/api/dates_api.html#date-tickers
ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))

```



Let's analyze this data and find out how rare such events are. First we need to find the annual maximum for each hydrological year. Do we see any seasonal patterns with our eyes? Play with the widget below.

```

# Altair only recognizes column data; it ignores index values.
# You can plot the index data by first resetting the index
# I know that I've just made 'DATE' the index, but I want to have this here nonetheless so I
df_new = df.reset_index()
source = df_new[['DATE', 'PRCP']]

```

```

brush = alt.selection_interval(encodings=['x'])

base = alt.Chart(source).mark_line().encode(
    x = 'DATE:T',
    y = 'PRCP:Q'
).properties(
    width=600,
    height=200
)

upper = base.encode(
    alt.X('DATE:T', scale=alt.Scale(domain=brush)),
    alt.Y('PRCP:Q', scale=alt.Scale(domain=(0,100)))
)

lower = base.properties(
    height=60
).add_params(brush)

alt.vconcat(upper, lower)

```

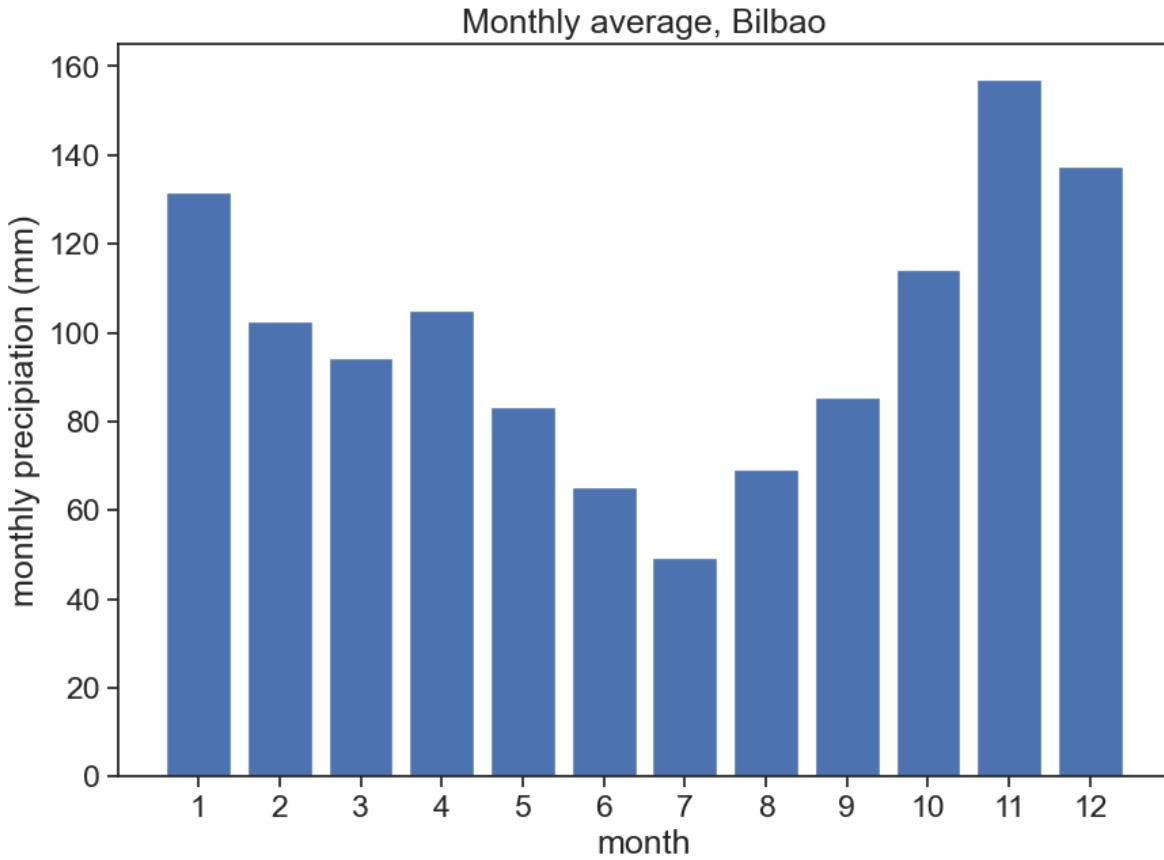
```
alt.VConcatChart(...)
```

Let's do what we learned already in the previous lectures, and plot monthly precipitation averages. For Bilbao, we will consider the hydrological year starting on 1 August.

```

df_month = df['PRCP'].resample('ME').sum().to_frame()
df_month_avg = (df_month['PRCP']
                 .groupby(df_month.index.month)
                 .mean()
                 .to_frame()
                 )
fig, ax = plt.subplots(figsize=(10,7))
ax.bar(df_month_avg.index, df_month_avg['PRCP'])
ax.set(xlabel="month",
       ylabel="monthly precipitation (mm)",
       title="Monthly average, Bilbao",
       xticks=np.arange(1,13));

```



- Top: Histogram of annual daily precipitation maximum events.
- Bottom: The cumulative answers the question: “How many events can be found **below** a given threshold?”

```
max_annual = (df['PRCP'].resample('YE-JUL')
               .max()
               .to_frame()
               )

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,8), sharex=True)

h=max_annual['PRCP'].values

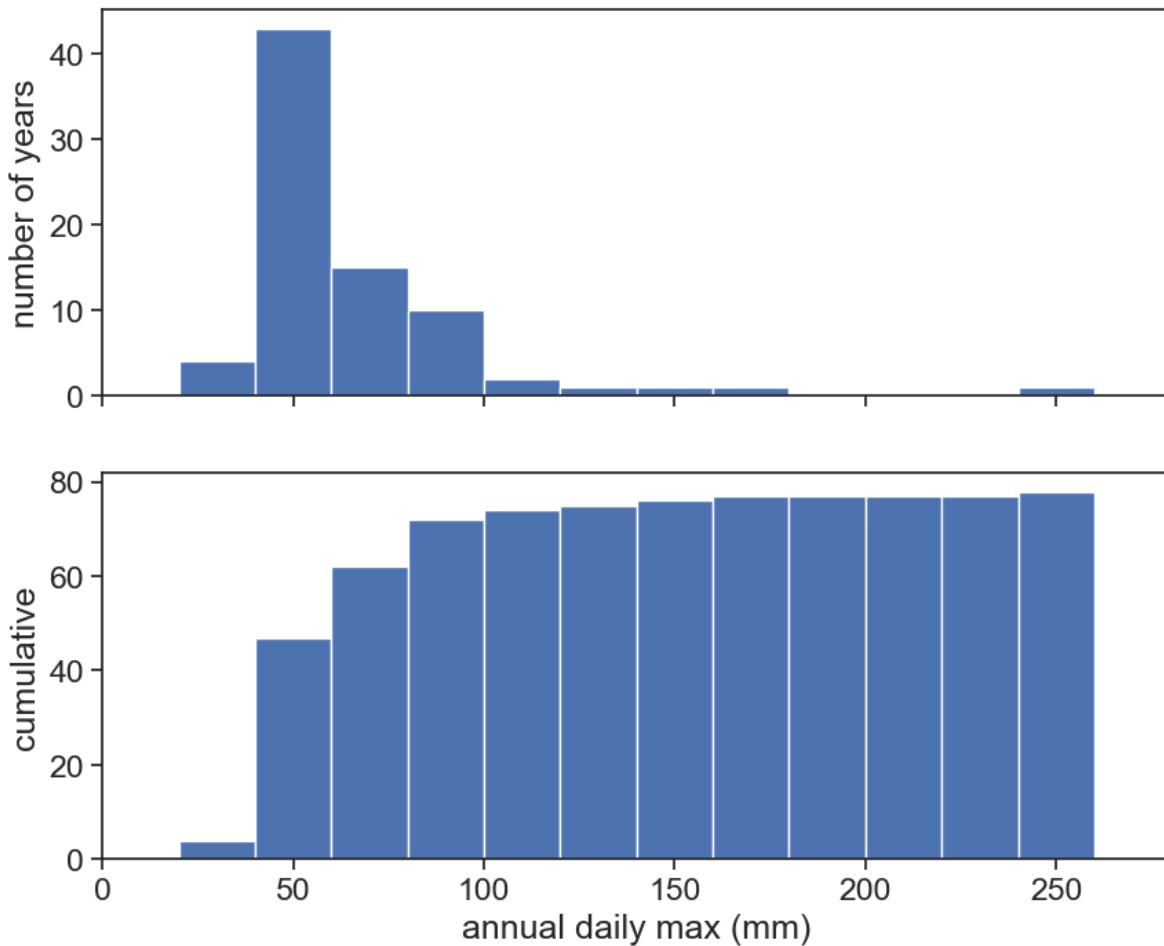
bins=np.arange(0,270,20)
ax1.hist(h, bins=bins)
```

```

ax2.hist(h, bins=bins, cumulative=1)

ax1.set(ylabel="number of years")
ax2.set(xlabel="annual daily max (mm)",
        ylabel="cumulative",
        xlim=[0,280]);

```

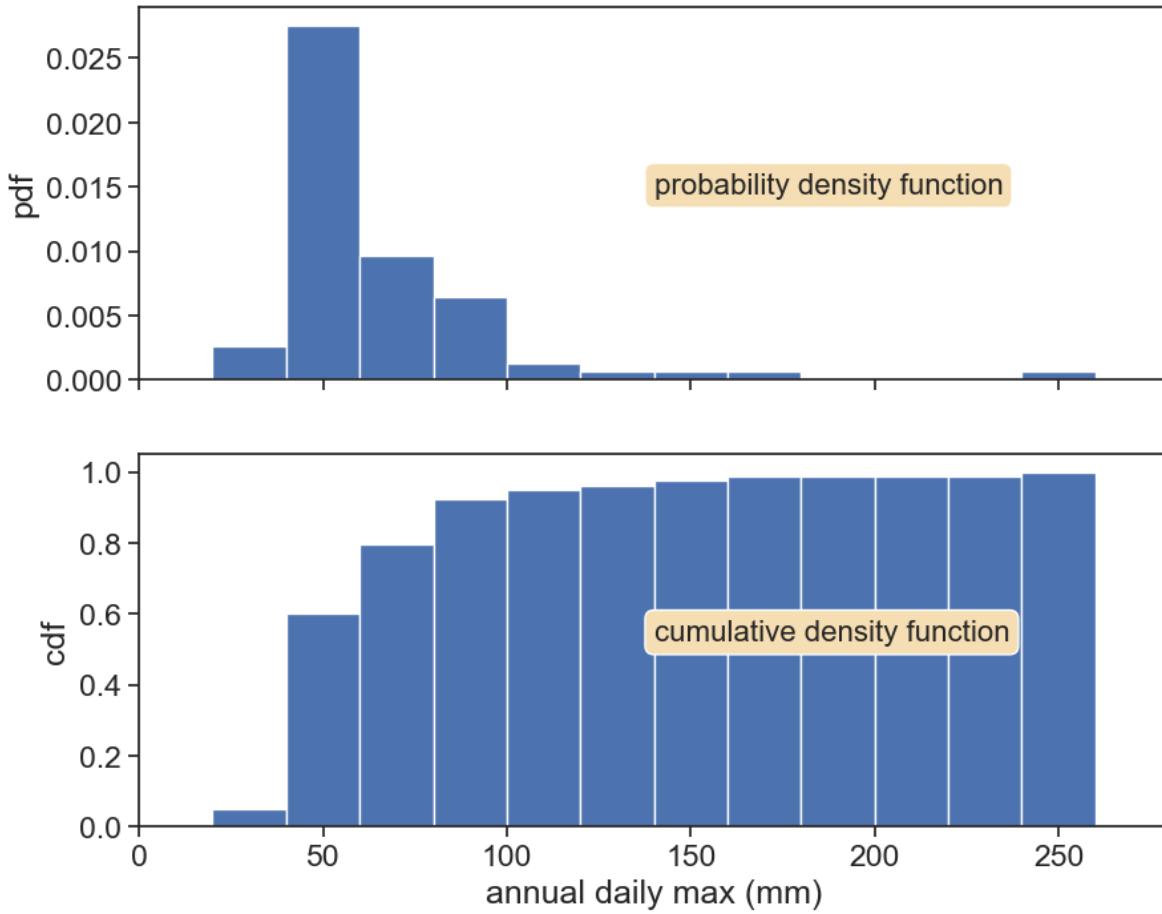


- Top: We can normalize the histogram such that the total area is 1. Now the histogram is called **probability density function (pdf)**. The probability is NOT the pdf, but the area between two thresholds.
- Bottom: The cumulative now becomes a probability between 0 and 1. It is now called **cumulative density function (cdf)**. The cdf answers the question: “What is the probability to choose an event **below** a given threshold?”

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,8), sharex=True)

ax1.hist(h, bins=bins, density=True)
ax2.hist(h, bins=bins, cumulative=1, density=True)

ax1.set(ylabel="pdf")
ax2.set(xlabel="annual daily max (mm)",
        ylabel="cdf",
        xlim=[0,280]
       )
ax1.text(0.5, 0.5, "probability density function",
         transform=ax1.transAxes, fontsize=16,
         bbox=dict(boxstyle='round', facecolor='wheat', alpha=1))
ax2.text(0.5, 0.5, "cumulative density function",
         transform=ax2.transAxes, fontsize=16,
         bbox=dict(boxstyle='round', facecolor='wheat', alpha=1));
```



We are interested in extreme events, and we want to estimate how many years, **on average**, do we have to wait to get an annual maximum **above** a given threshold?

This question is very similar to what we asked regarding the cdf.

We switched the word “below” for “above”. The complementary of the cumulative is called the exceedance (or survival) probability:

$$\text{exceedance, survival} = 1 - \text{cdf}$$

```
def plot_pdf_and_survival(survival):
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,6), sharex=True)

    ax1.hist(h, bins=bins, density=True, histtype='stepfilled', alpha=0.2)
    ax2.hist(h, bins=bins, density=True, histtype='stepfilled', alpha=0.2, cumulative=-1)

    params = genextreme.fit(h)
```

```

rain_max = 200.0
rain = np.arange(0,rain_max)
pdf = genextreme(c=params[0], loc=params[1], scale=params[2]).pdf
cdf = genextreme(c=params[0], loc=params[1], scale=params[2]).cdf
ax1.plot(rain, pdf(rain), 'r-', lw=5, alpha=0.6, clip_on=False)
ax2.plot(rain, 1-cdf(rain), 'r-', lw=5, alpha=0.6, clip_on=False)
ax1.set(ylabel="probability density")
ax2.set(xlabel="annual daily max (mm)",
        ylabel="survival = 1-cdf",
        xlim=[0,200])

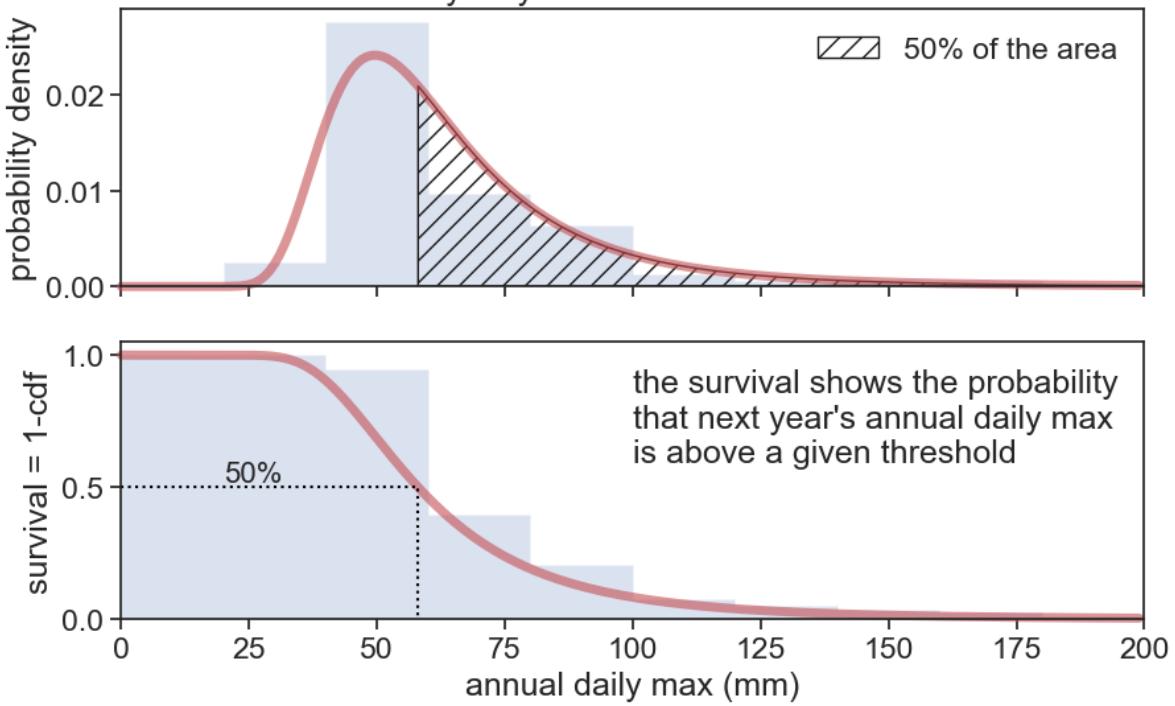
# survival = 0.20
threshold = genextreme(c=params[0], loc=params[1], scale=params[2]).ppf(1-survival)
xfill = np.linspace(threshold, rain_max, 100)
ax1.fill_between(xfill, pdf(xfill), color='None', hatch='//', edgecolor='k', label=f'{survival:.0%}')
ax2.plot([0, threshold, threshold], [survival, survival ,0], color="black", ls=":")
ax2.text(20, survival, f'{100*survival:.0f}%', ha="left", va="bottom", fontsize=12)

ax1.set(title=(f"we'll wait "+ 
               r"\n" + "on average" + "\n" + 
               f"({survival:.0%})" + 
               r"\n" + 
               f"({survival:.2f})" + 
               r"\n" + 
               f" {1/survival:.0f} years\nfor a yearly maximum above {threshold:.0f} mm"
               )
       )
ax1.legend(loc="upper right", frameon=False)
ax2.text(100, 0.95, "the survival shows the probability\nthat next year's annual daily max\nwill be above the current threshold", va="top")

```

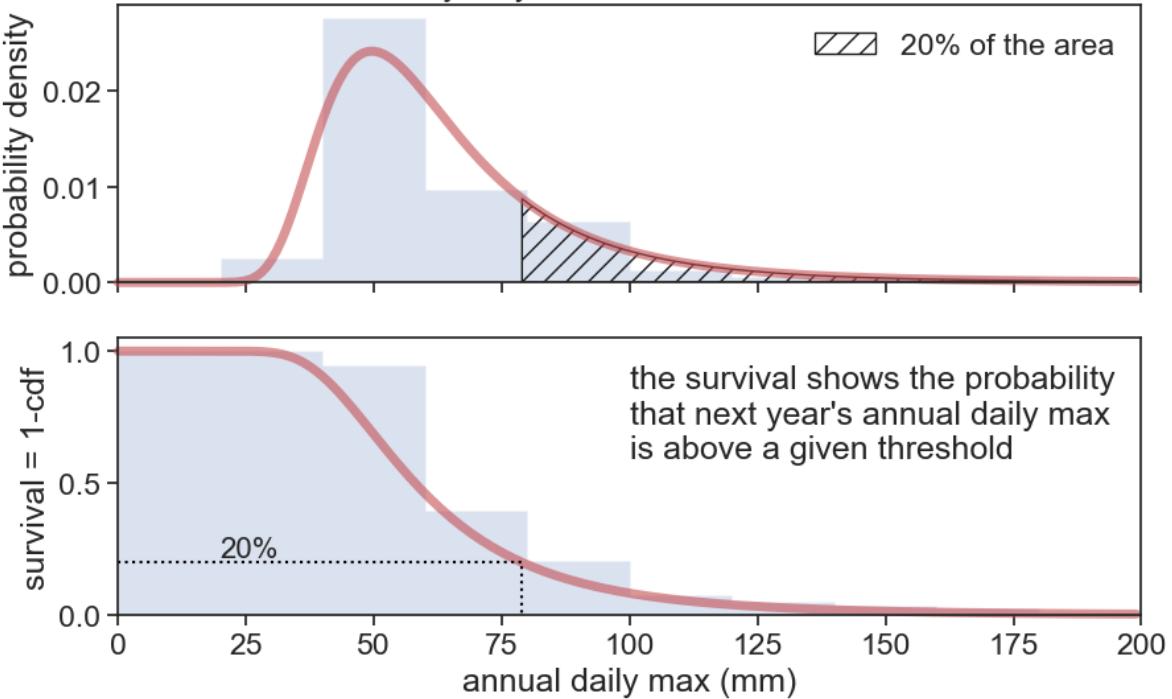
plot_pdf_and_survival(0.5)

we'll wait **on average** $(50\%)^{-1} = (0.50)^{-1} = 2$ years
for a yearly maximum above 58 mm



```
plot_pdf_and_survival(0.2)
```

we'll wait **on average** $(20\%)^{-1} = (0.20)^{-1} = 5$ years
for a yearly maximum above 79 mm



7.5 Return Period

We will follow Brutsaert (2005), page 513.

We call $F(x)$ the CDF of the PDF $f(x)$. $F(x)$ indicates the non-exceedance probability, i.e., the probability that a certain event above x has **not** occurred (or that an event below x has occurred, same thing). The exceedance probability, also called survival probability, equals $1 - F(x)$, and is the probability that a certain event above x *has* occurred. Its reciprocal is the return period:

$$T_r(x) = \frac{1}{1 - F(x)}$$

This return period is the expected number of observations required until x is exceeded once. In our case, we can ask the question: how many years will pass (on average) until we see a rainfall event greater than that of 26 August 1983?

Let's call $p = F(x)$ the probability that we measured once and that an event greater than x has *not* occurred. What is the probability that a rainfall above x will occur only on year number k ?

- it hasn't occurred on year 1 (probability p)
- it hasn't occurred on year 2 (probability p)
- it hasn't occurred on year 3 (probability p)
- ...
- it has occurred on year k (probability $1-p$)

$$P\{k \text{ trials until } X > x\} = p^{k-1}(1-p)$$

Every time the number k will be different. What will be k *on average*?

$$\bar{k} = \sum_{k=1}^{\infty} kP(k) = \sum_{k=1}^{\infty} kp^{k-1}(1-p)$$

Let's open that up:

$$\begin{aligned}\bar{k} &= 1 - p + 2p(1-p) + 3p^2(1-p) + 4p^3(1-p) + \dots \\ \bar{k} &= 1 - p + 2p - 2p^2 + 3p^2 - 3p^4 + 4p^3 - 4p^4 + \dots \\ \bar{k} &= 1 + p + p^2 + p^3 + p^4 + \dots\end{aligned}$$

For $p < 1$, the series converges to

$$1 + p + p^2 + p^3 + p^4 + \dots = \frac{1}{1-p},$$

therefore

$$\bar{k} = \frac{1}{1-p}.$$

We conclude that if we know the exceedance probability, we immediately can say what the return times are. We now need a way of estimating this exceedance probability.

7.6 Generalized extreme value distribution

This part of the lecture was heavily inspired by Alexandre Martinez's excellent [blog post](#).

The Generalized Extreme Value (GEV) distribution is the limit distribution of properly normalized maxima of a sequence of independent and identically distributed random variables ([from Wikipedia](#)).

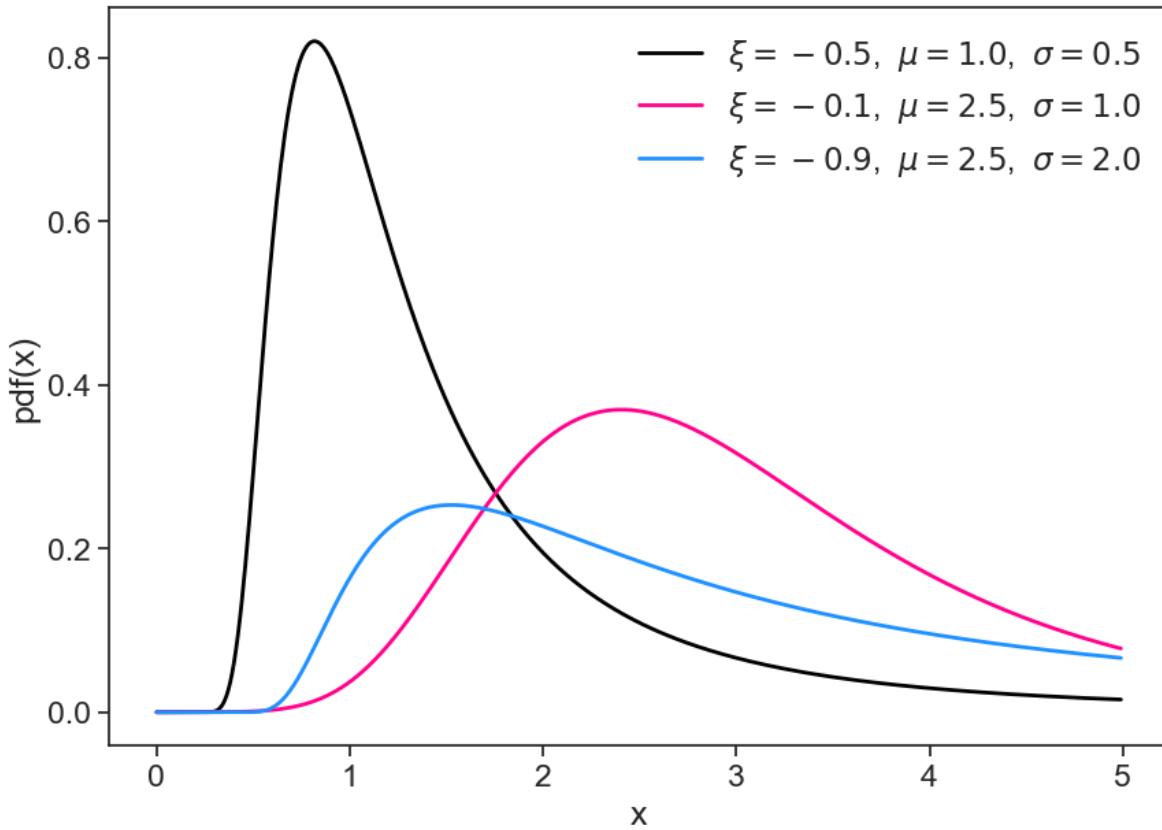
It has three parameters:

- ξ = shape,
- μ = location,
- σ = scale.

See a few examples of how the gev function looks for different parameters.

```
fig, ax = plt.subplots(figsize=(10,7))

shape = [-0.5, -0.1, -0.9]
location = [1.0, 2.5, 2.5]
scale = [0.5, 1.0, 2.0]
color=["black", "xkcd:hot pink", "dodgerblue"]
x = np.arange(0, 5, 0.01)
for i in range(len(shape)):
    pdf = genextreme.pdf(x, shape[i], loc=location[i], scale=scale[i])
    ax.plot(x, pdf, lw=2, color=color[i], label=rf"\xi={shape[i]}, $\mu={location[i]}$, $\\sigma={scale[i]}")
ax.legend(loc="upper right", frameon=False)
ax.set(xlabel="x",
       ylabel="pdf(x)");
```



Our task now is to find the “best curve” that describes our data. We do that by **fitting** the GEV curve, i.e., we need to find the best combination of parameters.

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,8), sharex=True)

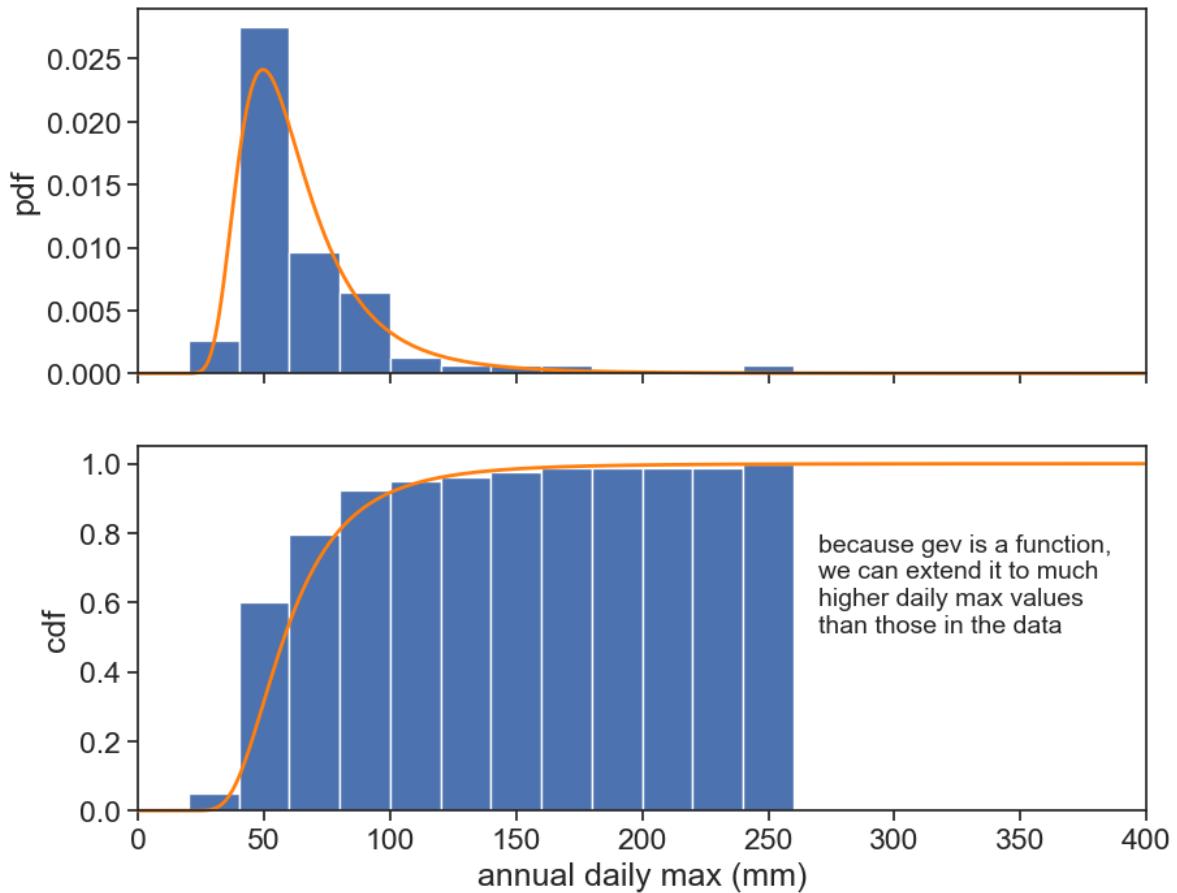
ax1.hist(h, bins=bins, density=True)
ax2.hist(h, bins=bins, cumulative=1, density=True)

params = genextreme.fit(h)
rain_max = 400.0
rain = np.arange(0,rain_max)
pdf = genextreme(c=params[0], loc=params[1], scale=params[2]).pdf
cdf = genextreme(c=params[0], loc=params[1], scale=params[2]).cdf
ax1.plot(rain, pdf(rain), color='tab:orange', clip_on=False, lw=2)
ax2.plot(rain, cdf(rain), color='tab:orange', clip_on=False, lw=2)
ax2.text(270, 0.8, "because gev is a function,\nwe can extend it to much\nhigher daily max va", fontsize=14, va="top")
```

```

ax1.set(ylabel="pdf")
ax2.set(xlabel="annual daily max (mm)",
        ylabel="cdf",
        xlim=[0, 400]
);

```



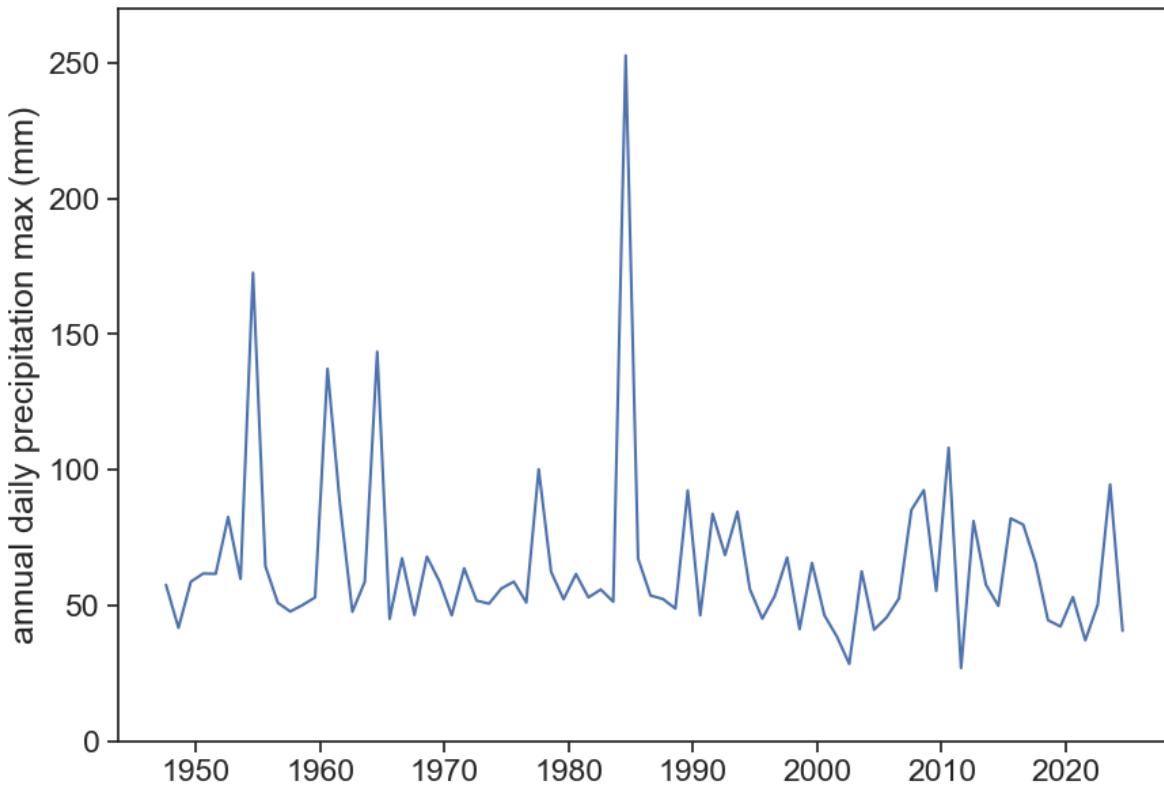
7.6.1 cdf from data

Here is a plot of annual daily precipitation maxima for Bilbao.

```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(max_annual['PRCP'])
ax.set(ylabel="annual daily precipitation max (mm)",
       ylim=[0, 270])

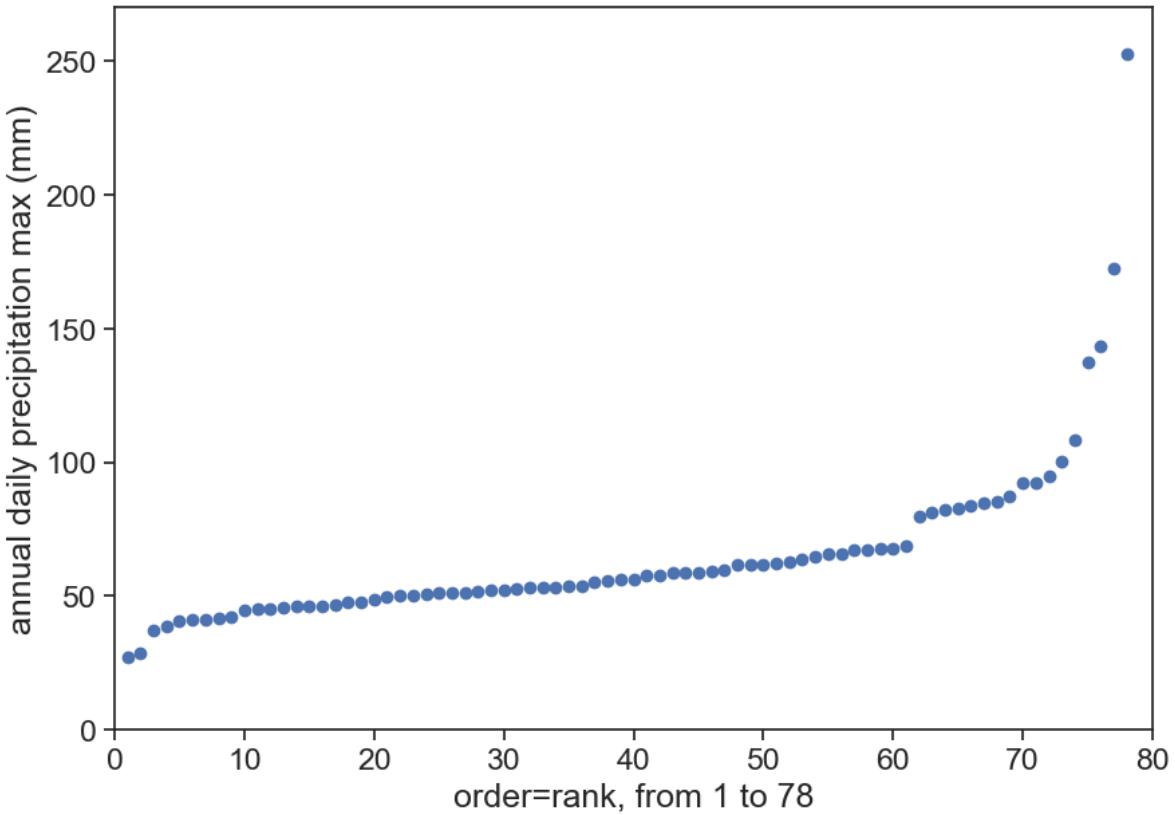
```



There are 74 points here. Let's order them from small to large:

```
# sort yearly max from highest to lowest
max_annual = max_annual.sort_values(by=['PRCP'], ascending=True)
max_annual['rank'] = np.arange(1, len(max_annual) + 1)

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(max_annual['rank'], max_annual['PRCP'], 'o')
ax.set(ylabel="annual daily precipitation max (mm)",
       xlabel=f"order=rank, from 1 to {len(max_annual)}",
       ylim=[0, 270],
       xlim=[0, 80]);
```



```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(max_annual['rank'], max_annual['PRCP'], 'o')

n = len(max_annual['rank'])
m = max_annual['rank']
cdf_fromdata = m / (n+1)

threshold100 = 100 # millimeters
count100 = np.sum(max_annual['PRCP'] < threshold100)
p100 = count100/(n+1)
ax.text(5, 105, f"{count100}/{n+1} = {p100:.0%} of the points\nare below the {threshold100} mm")

ax.plot(max_annual['rank'], threshold100 + 0*max_annual['PRCP'],
        color="tab:orange", lw=2)

threshold60 = 60 # millimeters
count60 = np.sum(max_annual['PRCP'] < threshold60)
p60 = count60/(n+1)
ax.text(5, 65, f"{count60}/{n+1} = {p60:.0%} of the points\nare below the {threshold60} mm")

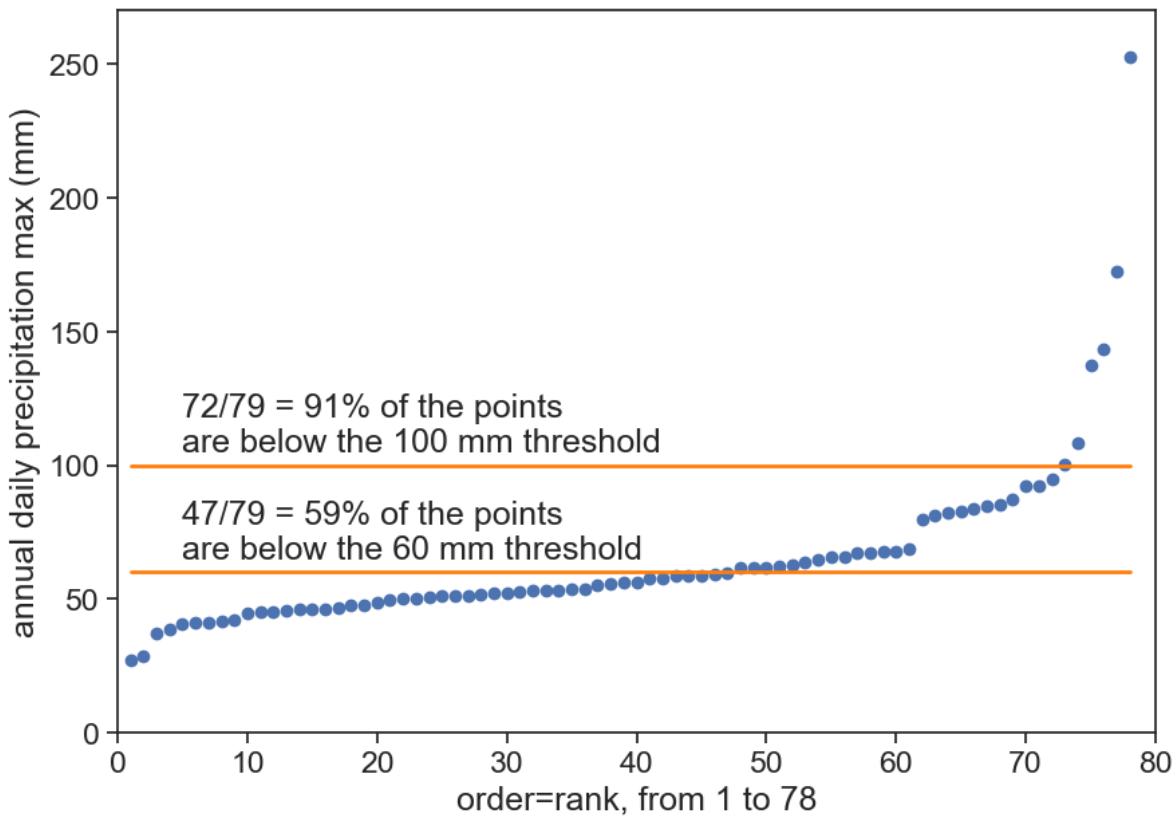
```

```

ax.plot(max_annual['rank'], threshold60 + 0*max_annual['PRCP'],
        color="tab:orange", lw=2)

ax.set(ylab="annual daily precipitation max (mm)",
      xlabel=f"order=rank, from 1 to {len(max_annual)}",
      ylim=[0, 270],
      xlim=[0, 80]);

```



Now, instead of having the order of the event on the horizontal axis, let's make it a fraction from 0 to 1.

```

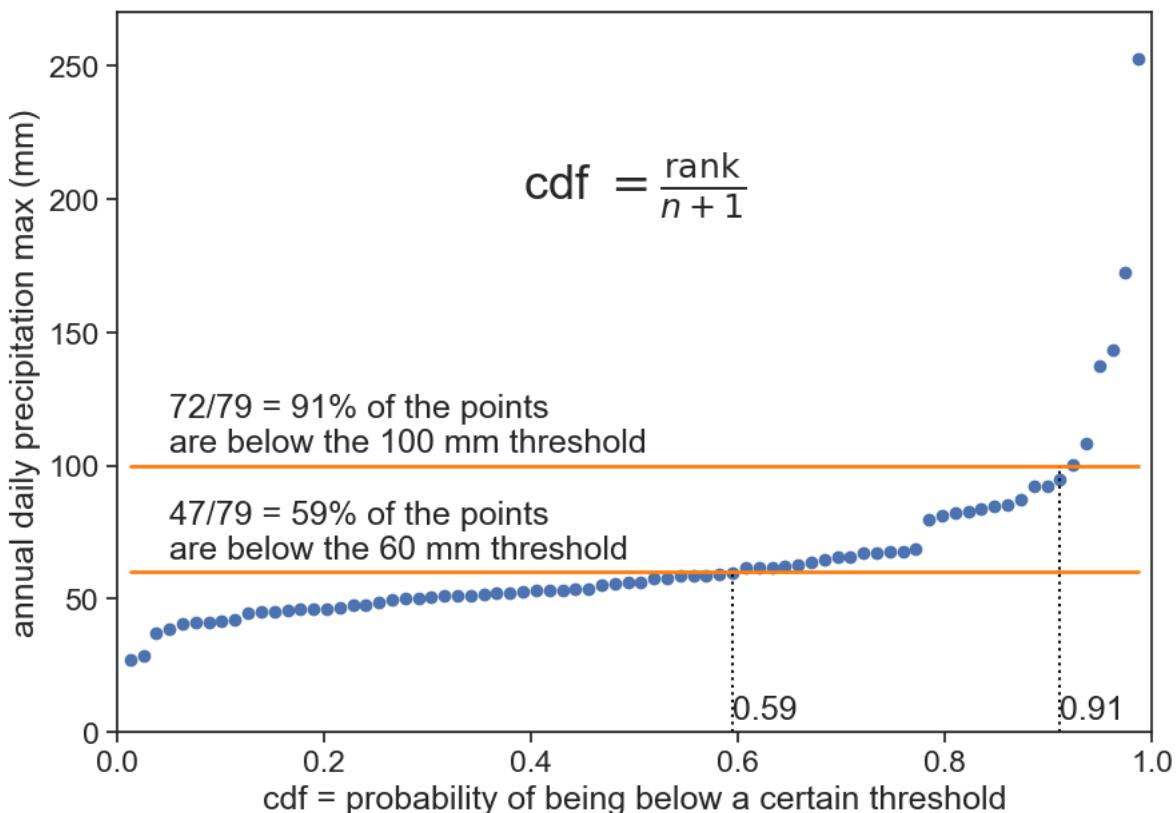
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(cdf_fromdata, max_annual['PRCP'], 'o')
ax.text(0.05, 105, f"{count100}/{n+1} = {p100:.0%} of the points\nare below the {threshold100} mm")
ax.plot(cdf_fromdata, threshold100 + 0*max_annual['PRCP'],
        color="tab:orange", lw=2)
ax.text(0.05, 65, f"{count60}/{n+1} = {p60:.0%} of the points\nare below the {threshold60} mm")
ax.plot(cdf_fromdata, threshold60 + 0*max_annual['PRCP'],
        color="tab:orange", lw=2)

```

```

    color="tab:orange", lw=2)
ax.plot([p100, p100], [0, threshold100], ls=":", color="black")
ax.text(p100, 5, f"{p100:.2f}")
ax.plot([p60, p60], [0, threshold60], ls=":", color="black")
ax.text(p60, 5, f"{p60:.2f}")
ax.text(0.5, 200, r"cdf =\frac{\text{rank}}{n+1}",
       fontsize=26, ha="center")
ax.set(ylabel="annual daily precipitation max (mm)",
       xlabel=f"cdf = probability of being below a certain threshold",
       ylim=[0, 270],
       xlim=[0, 1]);

```



The cdf was calculated using the Weibull plotting position:

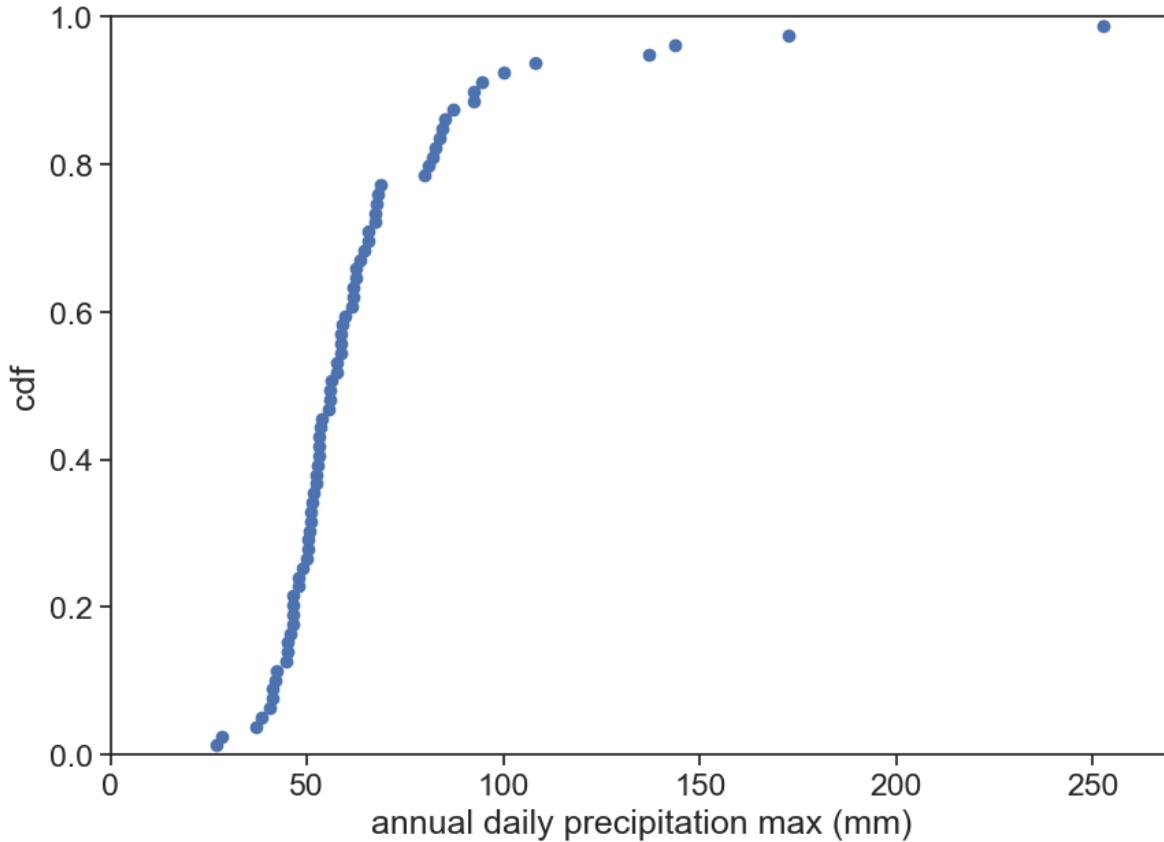
$$P_m = \frac{\text{rank}}{n + 1}$$

In the context of analyzing extreme values in precipitation, using the Weibull plotting position formula above is crucial for accurately estimating the cumulative distribution function (CDF).

This method ensures a more even distribution of plotting positions, correcting the bias that often occurs with small sample sizes when using n alone. By dividing by $n + 1$, each rank's cumulative probability is slightly adjusted, resulting in more realistic and representative plotting positions. This adjustment is particularly important in hydrology and meteorology, where accurate representation of extreme precipitation events is essential for risk assessment and infrastructure planning. The Weibull plotting position thus provides a more reliable tool for understanding and predicting extreme weather patterns.

Now we just need to flip the vertical and horizontal axes, and we're done! We have our cdf!

```
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(max_annual['PRCP'], cdf_fromdata, 'o')
ax.set(xlabel="annual daily precipitation max (mm)",
       ylabel=f"cdf",
       xlim=[0, 270],
       ylim=[0, 1]);
```

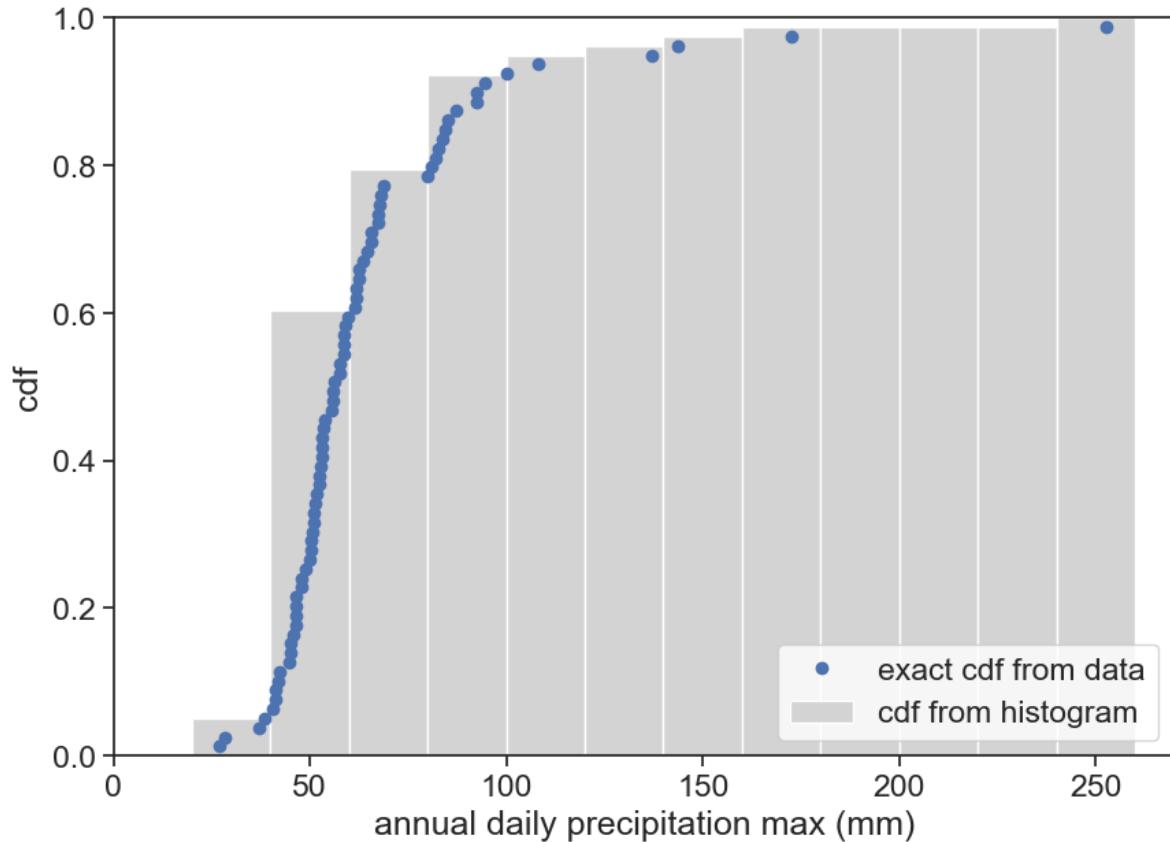


Let's compare this to the cumulative distribution from before, based on the histogram.

```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(max_annual['PRCP'], cdf_fromdata, 'o', label="exact cdf from data")
ax.hist(h, bins=bins, cumulative=1, density=True, histtype="bar", color="lightgray", label="cdf from histogram")
ax.legend(loc="lower right")
ax.set(xlabel="annual daily precipitation max (mm)",
       ylabel=f"cdf",
       xlim=[0, 270],
       ylim=[0, 1]);

```



The highest data point in this graph goes only to 252 mm, corresponding to the highest event recorded in 74 years. We can use the GEV cdf to calculate return times for any desired levels, simply by converting the vertical axis (cdf) to return period, using the equation we found earlier.

$$T_r(x) = \frac{1}{1 - F(x)},$$

where T_r is the return period (in years), and F is the cdf.

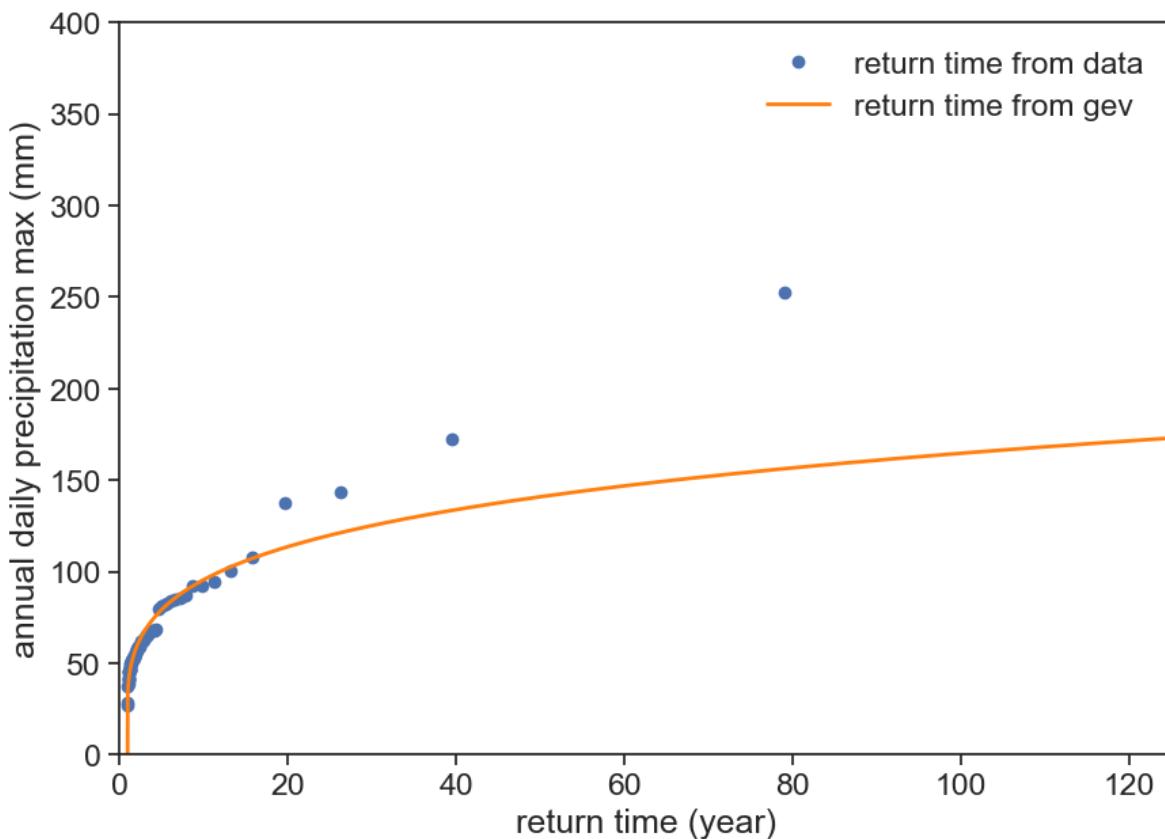
```

fig, ax = plt.subplots(figsize=(10,7))

T = 1 / (1-cdf_fromdata)
ax.plot(T, max_annual['PRCP'], 'o', label="return time from data")

ax.plot(1/(1-cdf(rain)), rain, color='tab:orange', lw=2, label="return time from gev")
ax.legend(loc="upper right", frameon=False)
ax.set(xlabel="return time (year)",
       ylabel=f"annual daily precipitation max (mm)",
       xlim=[0, 125],
       ylim=[0, 400]);

```



The information contained in the last two graphs is exactly the same, but somehow this last graph looks much worse! Why is this so?

8 Exercises

8.1 loading data and pre-processing

Import relevant packages

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
import urllib.request
from scipy.stats import genextreme
from scipy.optimize import curve_fit
```

Go to NOAA's National Centers for Environmental Information (NCEI)
[Climate Data Online: Dataset Discovery](#)

Find station codes in this [map](#). On the left, click on the little wrench next to “Global Summary of the Month”, then click on “identify” on the panel that just opened, and click on a station (purple circle). You will see the station’s name, it’s ID, and the period of record. For example, for Ben-Gurion’s Airport in Israel:

BEN GURION, IS

STATION ID: ISM00040180

Period of Record: 1951-01-01 to 2020-03-01

You can download **daily** or **monthly** data for each station. Use the function below to download this data to your computer. `station_name` can be whatever you want, `station_code` is the station ID.

If everything fails and you need easy access to the files we’ll be using today, click here:
[Eilat daily](#).

```

def download_data(station_name, station_code):
    url_daily = 'https://www.ncdc.noaa.gov/data/global-historical-climatology-network-daily/'
    url_monthly = 'https://www.ncdc.noaa.gov/data/gsom/access/'
    # download daily data - uncomment the following 2 lines to make this work
    urllib.request.urlretrieve(url_daily + station_code + '.csv',
                                station_name + '_daily.csv')
    # download monthly data
    urllib.request.urlretrieve(url_monthly + station_code + '.csv',
                                station_name + '_monthly.csv')

```

Download daily rainfall data for Eilat, Israel. ID: IS000009972

```
download_data('Eilat', 'IS000009972')
```

Then load the data into a dataframe.

IMPORTANT!! daily precipitation data is in tenths of mm, divide by 10 to get it in mm.
How do we know that? It's in the [documentation!](#)

```

df = pd.read_csv('Eilat_daily.csv', sep=",")
# make 'DATE' the dataframe index
df['DATE'] = pd.to_datetime(df['DATE'])
df = df.set_index('DATE')
# IMPORTANT!! daily precipitation data is in tenths of mm, divide by 10 to get it in mm.
df['PRCP'] = df['PRCP'] / 10
df

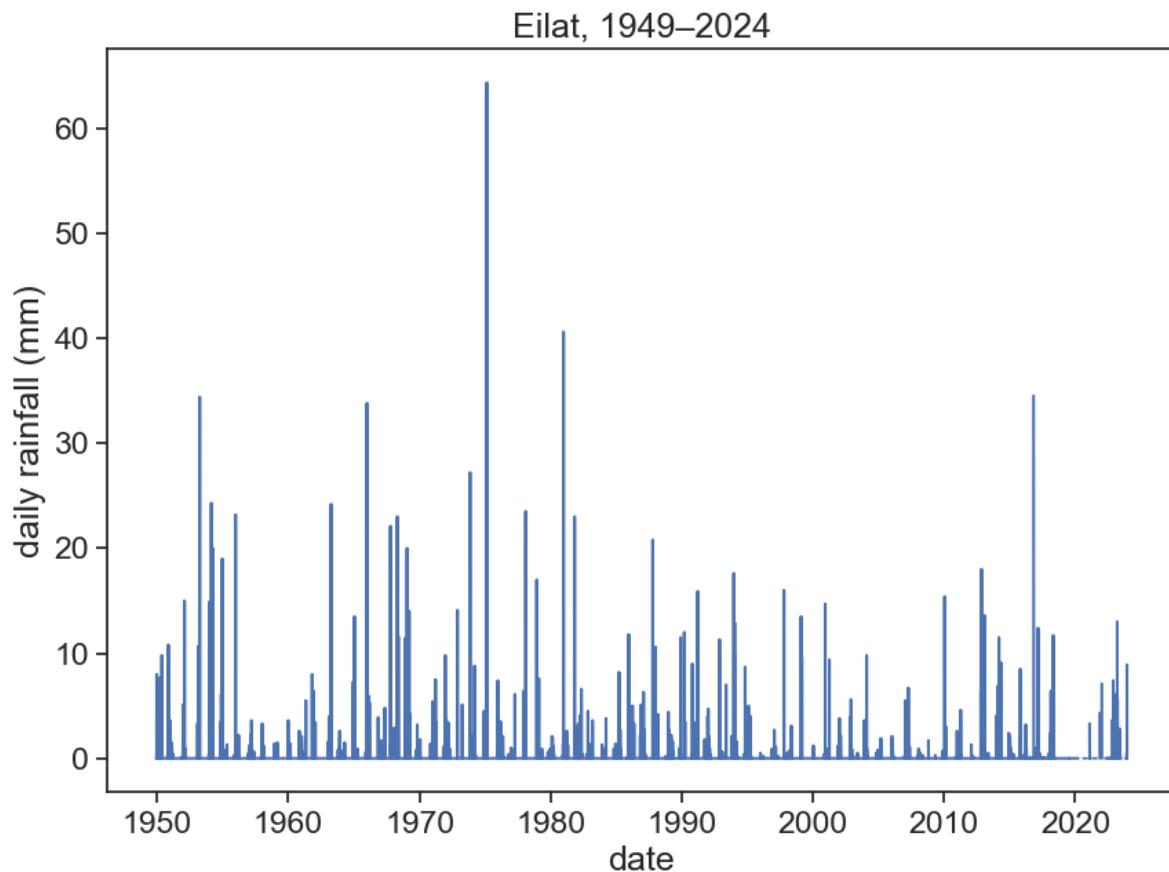
```

| DATE | STATION | LATITUDE | LONGITUDE | ELEVATION | NAME | PRCP | PRCP_ATTR |
|------------|-------------|----------|-----------|-----------|----------|------|-----------|
| 1949-11-30 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1949-12-01 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1949-12-02 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1949-12-03 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1949-12-04 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 2024-01-16 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | NaN | NaN |
| 2024-01-17 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | NaN | NaN |
| 2024-01-18 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | NaN | NaN |
| 2024-01-19 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | NaN | NaN |
| 2024-01-20 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | NaN | NaN |

Plot precipitation data ('PRCP' column) and see if everything is all right.

```
fig, ax = plt.subplots(figsize=(10,7))
ax.plot(df['PRCP'])
ax.set_xlabel("date")
ax.set_ylabel("daily rainfall (mm)")
ax.set_title("Eilat, 1949–2024")
```

```
Text(0.5, 1.0, 'Eilat, 1949–2024')
```



Based on what you see, you might want to exclude certain periods, e.g.:

```
last_date = '2018-08-01'
first_date = '1950-08-01'
df = df.loc[first_date:last_date]
df
```

| DATE | STATION | LATITUDE | LONGITUDE | ELEVATION | NAME | PRCP | PRCP_ATTR |
|------------|-------------|----------|-----------|-----------|----------|------|-----------|
| 1950-08-01 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1950-08-02 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1950-08-03 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1950-08-04 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| 1950-08-05 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „E |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 2018-07-28 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „S |
| 2018-07-29 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „S |
| 2018-07-30 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „S |
| 2018-07-31 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „S |
| 2018-08-01 | IS000009972 | 29.55 | 34.95 | 12.0 | ELAT, IS | 0.0 | „S |

The rainfall data for Eilat is VERY seasonal, it's easy to see that there is no rainfall at all during the summer. We can assume a hydrological year starting on 1 August. If you're not sure, you can plot the monthly means (see last week's lecture) and find what date makes sense best.

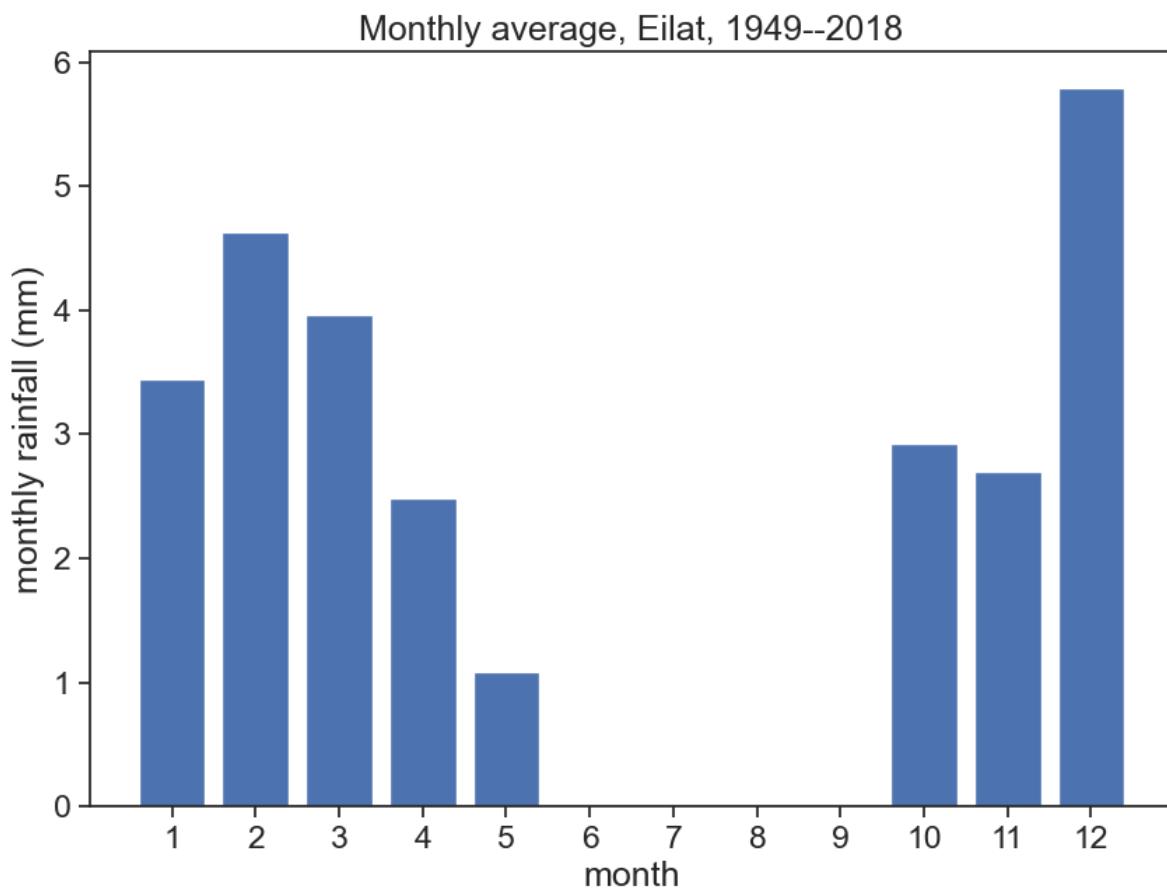
```
df_month = df['PRCP'].resample('M').sum().to_frame()
df_month_avg = (df_month['PRCP']
                 .groupby(df_month.index.month)
                 .mean()
                 .to_frame()
               )
df_month_avg
```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_71663/1784230487.py:1: FutureWarning:
df_month = df['PRCP'].resample('M').sum().to_frame()
```

| PRCP | |
|------|----------|
| DATE | |
| 1 | 3.445588 |
| 2 | 4.629412 |
| 3 | 3.958824 |
| 4 | 2.483824 |
| 5 | 1.086765 |
| 6 | 0.000000 |
| 7 | 0.000000 |

| PRCP | |
|------|----------|
| DATE | |
| 8 | 0.000000 |
| 9 | 0.008824 |
| 10 | 2.923529 |
| 11 | 2.701471 |
| 12 | 5.792647 |

```
fig, ax = plt.subplots(figsize=(10,7))
ax.bar(df_month_avg.index, df_month_avg['PRCP'])
ax.set(xlabel="month",
       ylabel="monthly rainfall (mm)",
       title="Monthly average, Eilat, 1949--2018",
       xticks=np.arange(1,13));
```



Let's resample the data according to the hydrological year (1 August), and we'll keep the maximum value:

```
max_annual = (df['PRCP'].resample('A-JUL')
               .max()
               .to_frame()
               )
max_annual
```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440_/_00000gn/T/ipykernel_71663/299619059.py:1: FutureWarning:
```

```
max_annual = (df['PRCP'].resample('A-JUL')
```

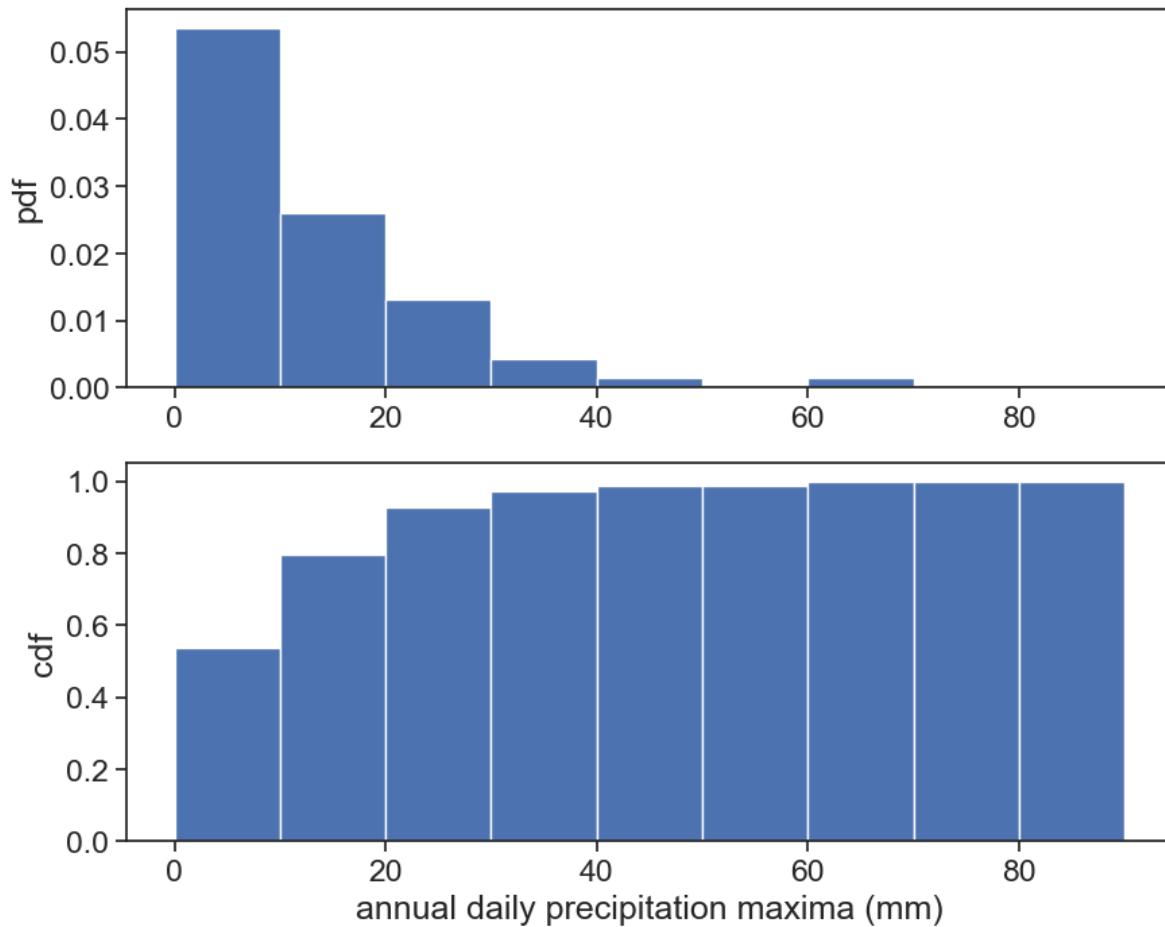
| PRCP | |
|------------|------|
| DATE | |
| 1951-07-31 | 10.8 |
| 1952-07-31 | 15.0 |
| 1953-07-31 | 34.4 |
| 1954-07-31 | 24.3 |
| 1955-07-31 | 19.0 |
| ... | ... |
| 2015-07-31 | 2.4 |
| 2016-07-31 | 8.5 |
| 2017-07-31 | 34.5 |
| 2018-07-31 | 11.7 |
| 2019-07-31 | 0.0 |

Make two graphs: a) the histogram for the annual maximum (pdf) b) the cumulative probability (cdf)

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,8))

h=max_annual['PRCP'].values
ax1.hist(h, bins=np.arange(0,100,10), density=True)
ax2.hist(h, bins=np.arange(0,100,10), cumulative=1, density=True)

ax1.set(ylabel="pdf")
ax2.set(xlabel="annual daily precipitation maxima (mm)",
        ylabel="cdf",
        );
```



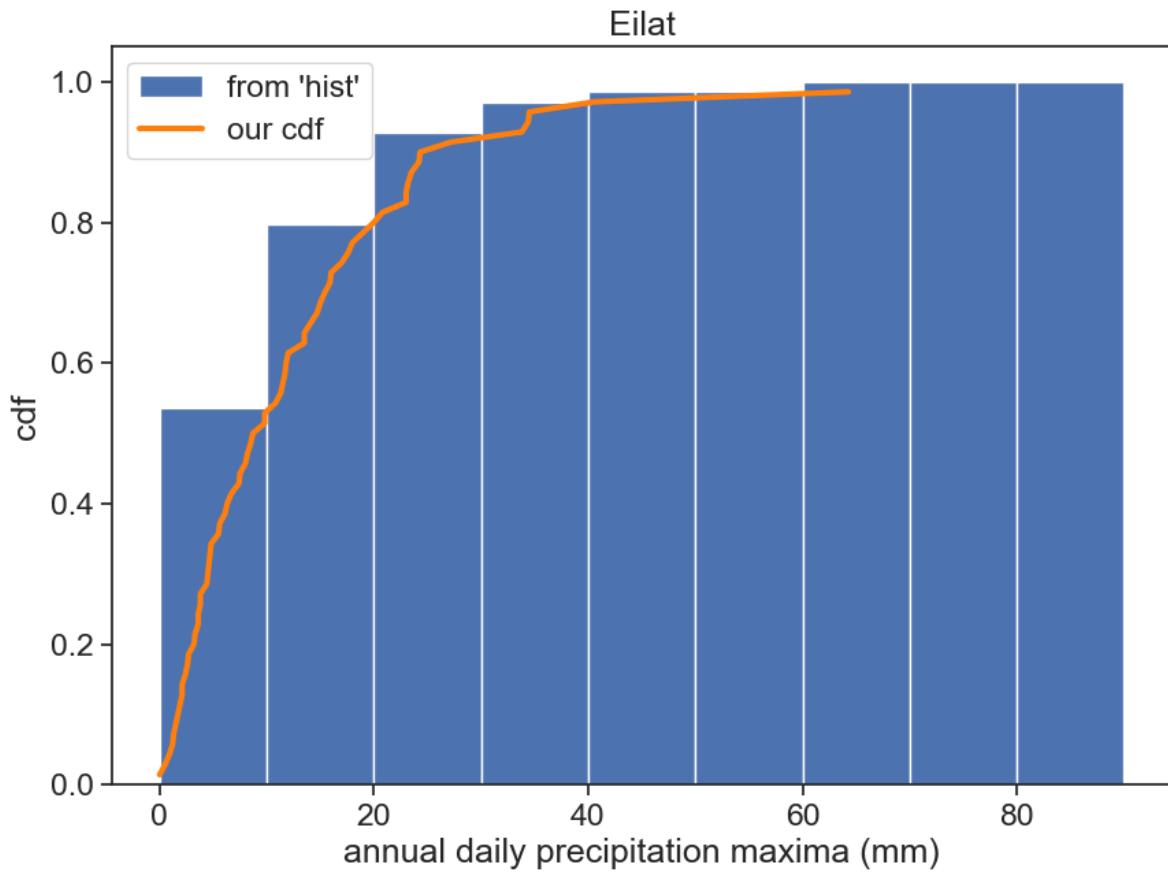
8.2 Weibull plotting position

How to make a cdf by yourself?

```
# sort the annual daily precipitation maxima, from lowest to highest
max_annual['max_sorted'] = np.sort(max_annual['PRCP'])
# let's give it a name, h
h = max_annual['max_sorted'].values
# make an array "order" of size N=len(h), from 1 to N
N = len(h)
rank = np.arange(N) + 1
# make a new array, "rank"
cdf_weibull = rank / (N+1)
```

Plot it next to the cdf that pandas' `hist` makes for you. What do you see?

```
fig, ax = plt.subplots(figsize=(10,7))
ax.hist(h, bins=np.arange(0,100,10), cumulative=1, density=True, label="from 'hist'")
ax.plot(h, cdf_weibull, color="tab:orange", linewidth=3, label="our cdf")
ax.set_ylabel("cdf")
ax.set_xlabel("annual daily precipitation maxima (mm)")
ax.set_title("Eilat")
ax.legend()
```



The generalized extreme value distribution has 3 parameters: shape, location, scale.

Let's get a “best fit” estimate of these parameters for Eilat’s rainfall statistics.

```
params = genextreme.fit(h)
print("Best fit:")
print(f"shape = {params[0]:.2f}\nlocation = {params[1]:.2f}\nscale = {params[2]:.2f}")
```

```

Best fit:
shape = -0.42
location = 6.05
scale = 5.68

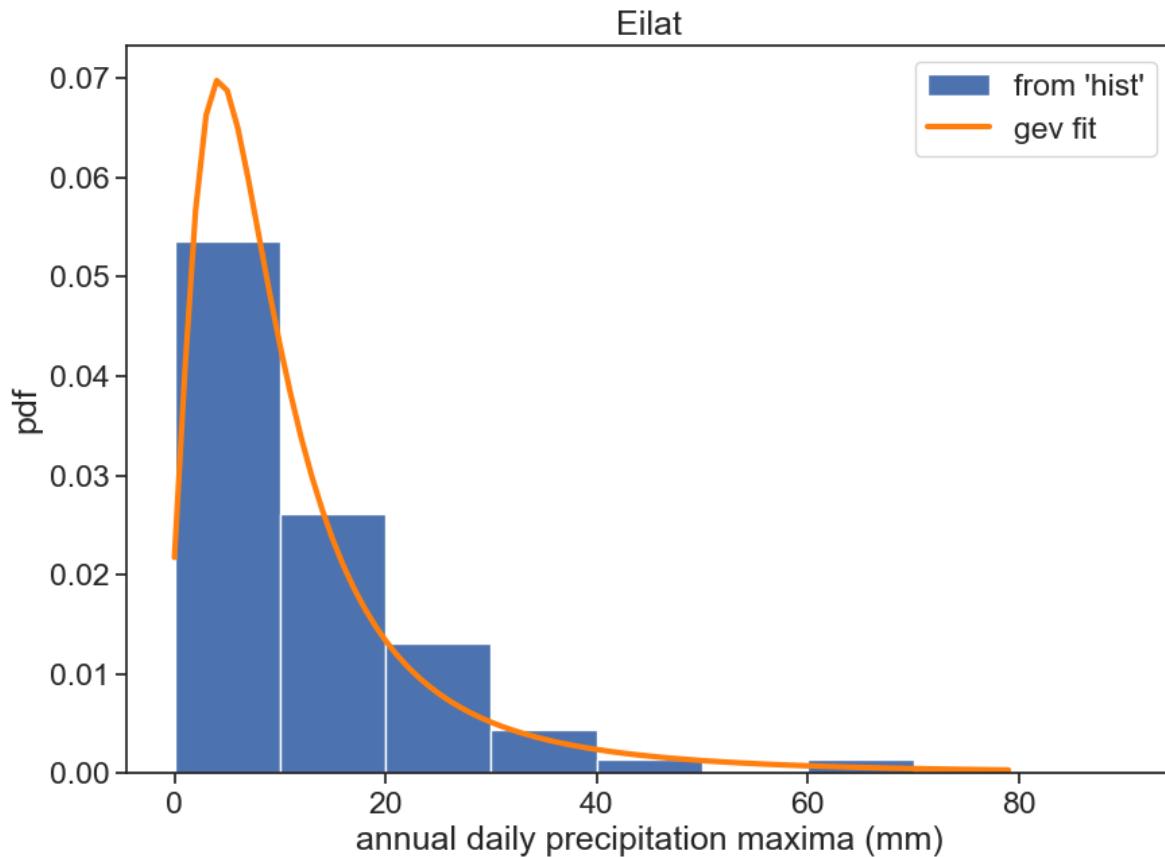
```

Let's see the GEV distribution for these parameters

```

fig, ax = plt.subplots(figsize=(10,7))
ax.hist(h, bins=np.arange(0,100,10), density=True, label="from 'hist'")
rain = np.arange(0,80)
pdf_rain = genextreme(c=params[0], loc=params[1], scale=params[2]).pdf(rain)
ax.plot(rain, pdf_rain, color="tab:orange", lw=3, label="gev fit")
ax.set_ylabel("pdf")
ax.set_xlabel("annual daily precipitation maxima (mm)")
ax.set_title("Eilat")
ax.legend()

```

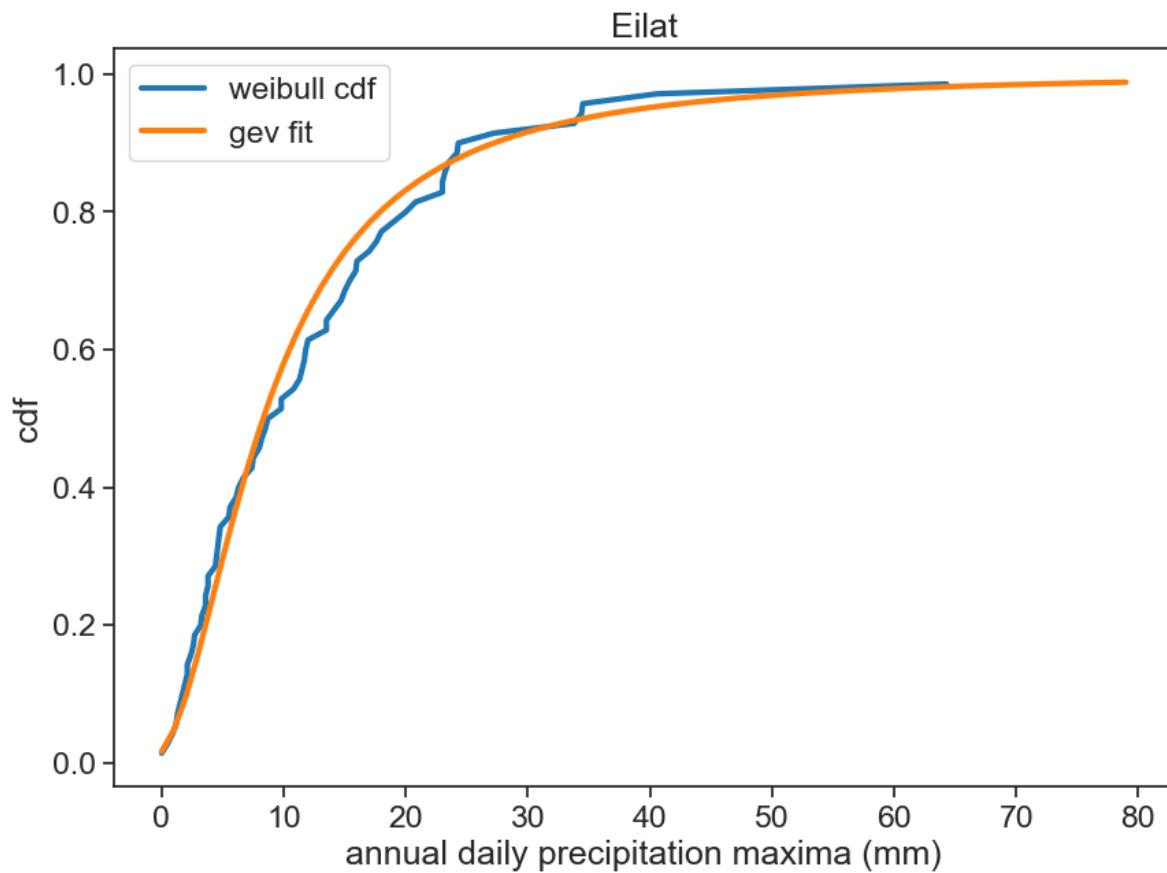


We can do the same for the cdf...

```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(h, cdf_weibull, color="tab:blue", linewidth=3, label="weibull cdf")
rain = np.arange(0,80)
cdf_rain = genextreme(c=params[0], loc=params[1], scale=params[2]).cdf(rain)
ax.plot(rain, cdf_rain, color="tab:orange", lw=3, label="gev fit")
ax.set_ylabel("cdf")
ax.set_xlabel("annual daily precipitation maxima (mm)")
ax.set_title("Eilat")
ax.legend()

```



We are almost there! Remember that the return time are given by:

$$T_r(x) = \frac{1}{1 - F(x)},$$

where F is the cdf.

$$\text{Survival} = 1 - F$$

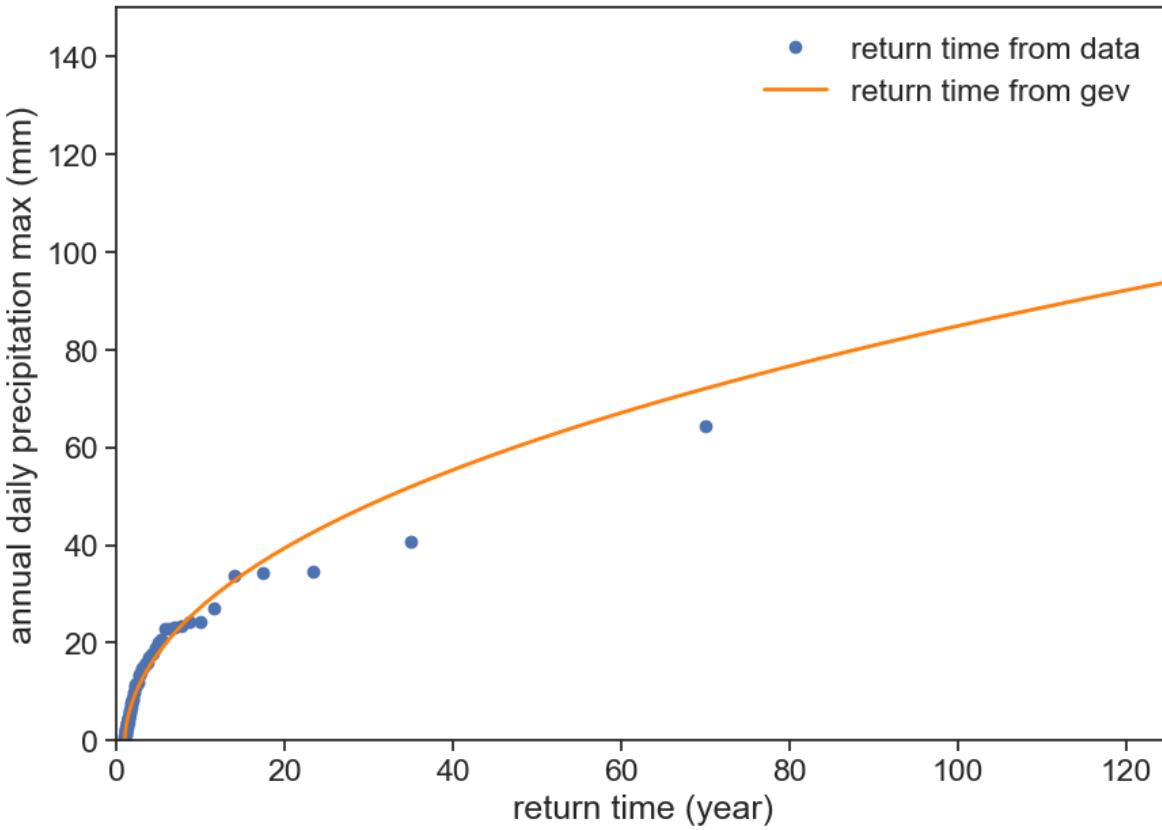
The package that we are using, `scipy.stats.genextreme` has a method called `isf`, or inverse survival function, which is exactly what we want! In order to use it, you have to feed it a “quantile” q or probability. Suppose you want to know how strong is a 1 in a 100 year event, then your return period is 100 (years), and the probability is simply its inverse, $1/100$.

```
fig, ax = plt.subplots(figsize=(10,7))

T = 1 / (1-cdf_weibull)
ax.plot(T, h, 'o', label="return time from data")

rain = np.arange(0,150)
cdf_rain = genextreme(c=params[0], loc=params[1], scale=params[2]).cdf(rain)

ax.plot(1/(1-cdf_rain), rain, color='tab:orange', lw=2, label="return time from gev")
ax.legend(loc="upper right", frameon=False)
ax.set(xlabel="return time (year)",
       ylabel=f"annual daily precipitation max (mm)",
       xlim=[0, 125],
       ylim=[0, 150]);
```



In the code below, we use $1/T$ as the argument for `isf`. Why?

We know that

$$T = \frac{1}{1 - \text{CDF}} = \frac{1}{\text{SF}}$$

If we take the reciprocal of the equation above, SF will become the inverse SF, or ISF:

$$\text{ISF} = \frac{1}{T}$$

The code below was heavily inspired by [this Stack Overflow response](#).

```
# Compute the return levels for several return periods.
return_periods = np.array([5, 10, 20, 50, 100, 500])
return_levels = genextreme.isf(1/return_periods, *params)

print("Return levels:")
```

```

print()
print("Period      Level")
print("(years)    (mm)")

for period, level in zip(return_periods, return_levels):
    print(f'{period:4.0f}  {level:9.2f}')

```

Return levels:

| Period (years) | Level (mm) |
|-------------------|---------------|
| 5 | 17.88 |
| 10 | 27.22 |
| 20 | 39.36 |
| 50 | 61.56 |
| 100 | 84.84 |
| 500 | 173.12 |

You might want to do the opposite: given a list of critical daily max levels, what are the return periods for them? In this case you can use the `sf` method, “survival function”.

```

levels_mm = np.array([20, 50, 100, 200, 300])
return_per = 1/genextreme.sf(levels_mm, *params)

print("Return levels:")
print()
print("Level      Period")
print("(mm)      (years)")

for level,period in zip(levels_mm, return_per):
    print(f'{level:9.2f}  {period:4.0f}')

```

Return levels:

| Level (mm) | Period (years) |
|---------------|-------------------|
| 20.00 | 6 |
| 50.00 | 32 |
| 100.00 | 144 |
| 200.00 | 698 |
| 300.00 | 1798 |

8.3 fit

Not always the fit operation succeeds. Sometimes, the fitted parameters yield curves that do not seem to describe well the pdf or the cdf we are studying. What to do?

1. **ALWAYS** check your parameters. Plot the fitted curve against the experimental data and see with your eyes if it makes sense.
2. If it doesn't make sense, you have to run `fit` again, with some changes. A common problem is that the algorithm chose initial values for the parameters that do not converge to the optimal parameters we are looking for. In this case, one needs to help `fit` by giving it initial guesses for the parameters, like this:

```
location_guess = 6.0
scale_guess = 5.0
shape_guess = -0.5
params = genextreme.fit(data, shape_guess, loc=location_guess, scale=scale_guess)
```

More details on this can be found in the [documentation](#).

Part III

Evapotranspiration

9 Evapotranspiration

Dingman (2015), chapter 6.

Globally, about 62% of the precipitation that falls on the continents is evapotranspirated, amounting to 73 thousand km³/yr. Of this, about 42% (29 thousand km³/yr) is transpiration, and about 3% is open-water evaporation. Most of the remainder is interception loss; soil evaporation is a minor component of the total.

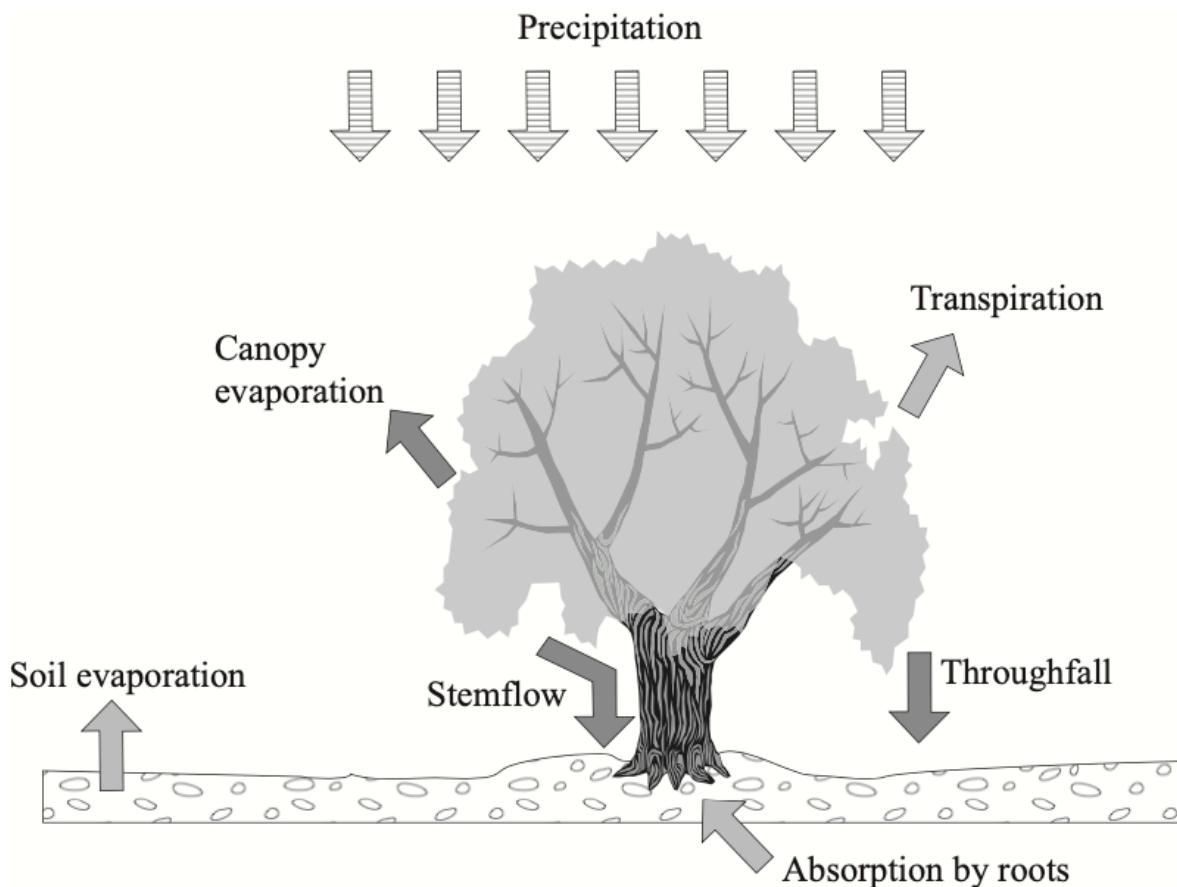


Fig. 5.1 : Principal elements of the interception and evaporation processes.

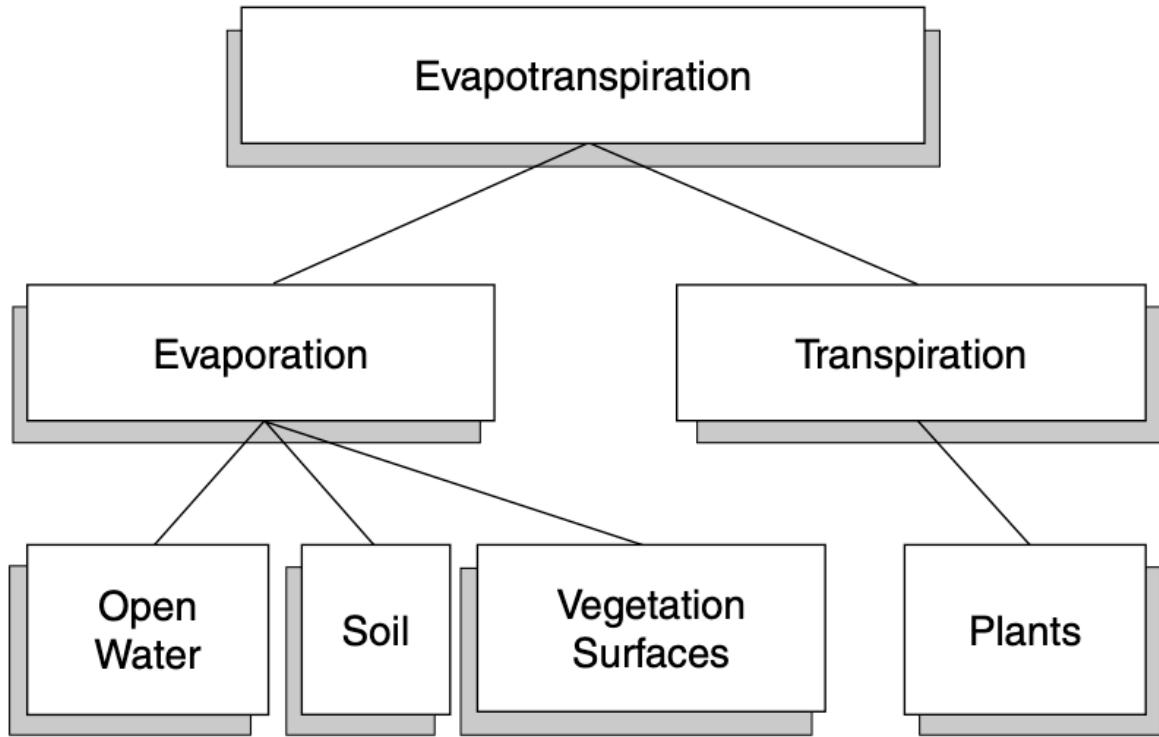


FIGURE 4.1 Evapotranspiration divided into subprocesses.

9.0.1 Potential Evapotranspiration

Potential Evapotranspiration (PET) is the rate at which evapotranspiration would occur from a large area completely and uniformly covered with growing vegetation with access to an unlimited supply of soil water and without advection or heat-storage effects.

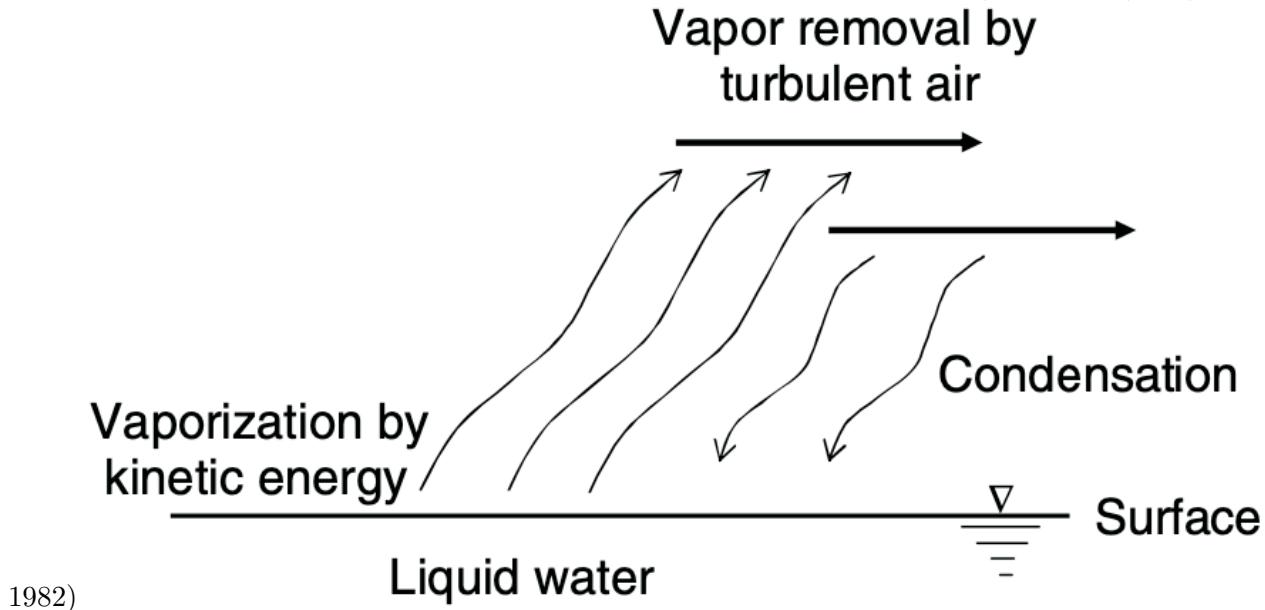
Several characteristics of a vegetative surface have a strong influence on ET rate.

- the albedo of the surface, which determines the net radiation;
- the maximum leaf conductance;
- the atmospheric conductance, largely determined by vegetation height;
- presence or absence of intercepted water.

9.0.2 Reference-Crop Evapotranspiration

Reference-crop evapotranspiration (RET) is the amount of water transpired by a short green crop, completely shading the ground, of uniform height, and never short of water.

The magnitude of PET is often calculated from meteorological data collected under conditions in which the actual ET rate is less than the potential rate. If ET had been occurring at the potential rate, the latent- and sensible-heat exchanges between air and the surface, and hence the air temperature and humidity, would have been considerably different. (Brutsaert (2005)





National Weather Service Class A Evaporation Pan
NOAA / NWS

9.1 Review of methods

There are a variety of ways to estimate evaporative flux in nature. The following table categorizes each method based on the data that must be acquired to apply it:

TABLE 4.3
Minimum Climatic Information Needs
of ET Estimation Methods

| Method | T ^a | RH ^b or e _d ^c | Latitude | Elevation | R _s ^d | u ^e |
|--------------------|----------------|--|----------|-----------|-----------------------------|----------------|
| Penman | x | x | | | x | x |
| Jensen-Haise | x | | | | x | x |
| SCS Blaney-Criddle | x | | | x | | |
| Thornthwaite | x | | | | | |

^a Air temperature.

^b Relative humidity.

^c Actual vapor pressure of the air.

^d Solar radiation.

^e Wind speed.

Figure 9.1: Source: Ward and Trimble (2003)

These methods also vary in the timescales in which they are relevant, typically in correlation with the variety of data needed:

- Thornthwaite and SCS Blaney-Criddle: monthly or seasonal estimations (minimal data)
- Jensen-Haise: 5-day estimates (good enough timescale and data for irrigation scheduling)
- Penman: daily estimates
- Penman-Monteith: hourly estimates (requires a lot of data)

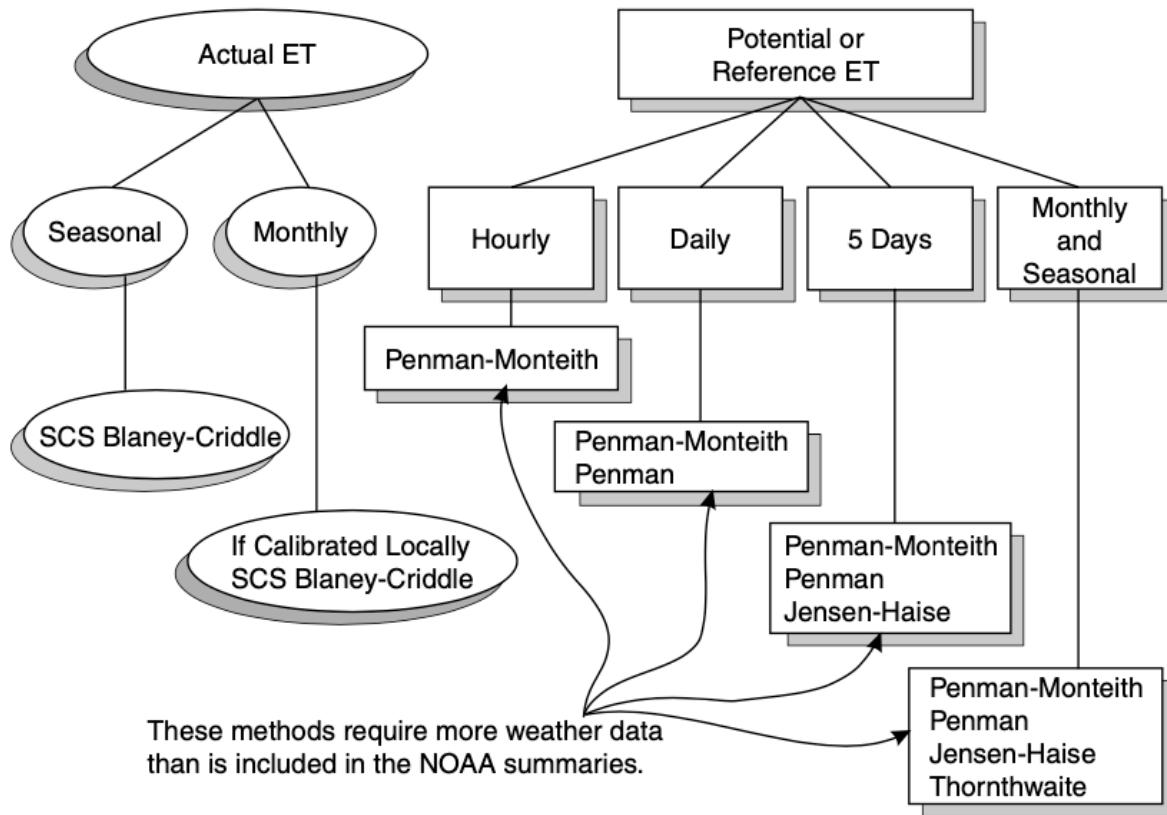


Figure 9.2: Source: Ward and Trimble (2003)

9.2 Thorntwaite

Source: Ward and Trimble (2003), pages 107-108.

Thorntwaite (1948) developed an equation to predict monthly evapotranspiration from mean monthly temperature and latitude data (Equation 4.27). The small amount of data needed is attractive because often ET needs to be predicted for sites where few weather data are available. Based on what we know about ET, we should be skeptical about the general applicability of such a simple equation. Thorntwaite (1948) was not satisfied with the proposed approach: "The mathematical development is far from satisfactory. It is empirical. ... The chief obstacle at present to the development of a rational equation is the lack of understanding of why potential ET corresponding to a given temperature is not the same everywhere."

Taylor and Ashcroft (1972), as cited in Skaggs (1980), provided insight into the answer to Thorntwaite's question. They said:

This equation, being based entirely upon a temperature relationship, has the disadvantage of a rather flimsy physical basis and has only weak theoretical justification. Since temperature and vapor pressure gradients are modified by the movement of air and by the heating of the soil and surroundings, the formula is not generally valid, but must be tested empirically whenever the climate is appreciably different from areas in which it has been tested. ... In spite of these shortcomings, the method has been widely used. Because it is based entirely on temperature data that are available in a large number of localities, it can be applied in situations where the basic data of the Penman method are not available.

M.E. Jensen et al. (1990) warn that Thornthwaite's method is generally only applicable to areas that have climates similar to that of the east central U.S., and it is not applicable to arid and semiarid regions.

Thornthwaite (1948) found that evapotranspiration could be predicted from an equation of the form

$$E = 16 \left[\frac{10 T^{\text{monthly mean}}}{I} \right]^a,$$

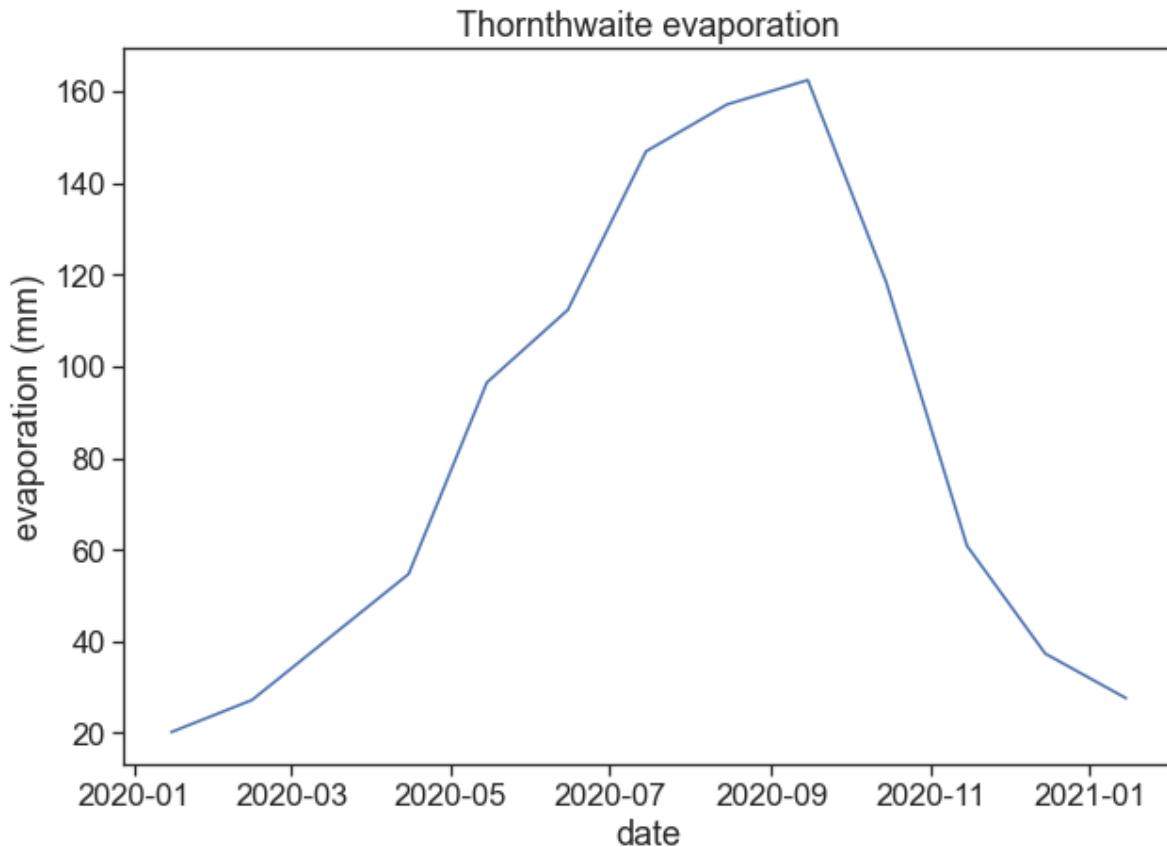
where

$$I = \sum_{i=1}^{12} \left[\frac{T_i^{\text{monthly mean}}}{5} \right]^{1.514},$$

and

$$\begin{aligned} a = & +6.75 \times 10^{-7} I^3 \\ & - 7.71 \times 10^{-5} I^2 \\ & + 1.792 \times 10^{-2} I \\ & + 0.49239 \end{aligned}$$

- E is the monthly potential ET (mm)
- $T^{\text{monthly mean}}$ is the mean monthly temperature in °C
- I is a heat index
- a is a location-dependent coefficient



9.3 Penman

Sources:

Brutsaert (2005), pages 123-127.

Ward and Trimble (2003), subsections 4.5.2, 4.5.3, 4.5.5, 4.6.6.

Allen et al. (1998), “[Crop evapotranspiration - Guidelines for computing crop water requirements - FAO Irrigation and drainage paper 56](#)”

The Penman model is almost entirely a theory-based formula for predicting evaporative flux. It can run on a much finer timescale, and requires a much wider variety of data than most models. In addition to temperature, the Penman functions on measurements of radiation, wind speed, elevation above sea level, vapor-pressure deficit, and heat flux density to the ground. The potential ET (in mm d^{-1}) is given by:

$$E = \frac{1}{\lambda} \left[\frac{\Delta}{\Delta + \gamma} Q_{ne} + \frac{\gamma}{\Delta + \gamma} E_A \right],$$

where Q_n is the available energy flux density

$$Q_n = R_n - G,$$

and E_A is the drying power of the air

$$E_A = 6.43 \cdot f(u) \cdot \text{VPD}.$$

The constituents of the equations above are

- E : potential evapotranspiration (mm d^{-1})
- Δ : slope of the saturation water vapor pressure curve ($\text{kPa } ^\circ\text{C}^{-1}$)
- γ : psychrometric constant ($\text{kPa } ^\circ\text{C}^{-1}$)
- λ : latent heat of vaporization (MJ kg^{-1})
- R_n : net radiation ($\text{MJ m}^{-2}\text{d}^{-1}$)
- G : heat flux density to the ground ($\text{MJ m}^{-2}\text{d}^{-1}$)
- $f(u)$: wind function (dimensionless)
- VPD: vapor pressure deficit (kPa)

and the number 6.43 adjusts the units of E_A so it is in $\text{MJ m}^{-2}\text{d}^{-1}$. In what follows, we will further discuss these constituents.

9.3.1 Psychrometric Constant

The psychrometric constant γ ($\text{kPa } ^\circ\text{C}^{-1}$) relates the partial pressure of water in air to the air temperature:

$$\gamma = \frac{c_p P}{\lambda \cdot MW_{\text{ratio}}}$$

$$P = 101.3 - 0.01055H$$

$$\lambda = 2.501 - 2.361 \times 10^{-3} T$$

- $MW_{\text{ratio}} = 0.622$: ratio molecular weight of water vapor/dry air
- P : atmospheric pressure (kPa). Can be either measured or inferred from station height above sea level (m).
- λ : latent heat of water vaporization (MJ kg^{-1})
- $c_p = 0.001013$: specific heat capacity of moist air ($\text{MJ kg}^{-1} {}^\circ\text{C}^{-1}$)

9.3.2 Net Radiation

Source: Ward and Trimble (2003), page 99.

R_n ($\text{MJ m}^{-2}\text{d}^{-1}$) is net radiation, the balance between net short wave R_s and the long wave R_b components of the radiation:

$$R_n = (1 - \alpha)R_{s\downarrow} - R_{b\uparrow},$$

where α (dimensionless) is the albedo. The net outgoing thermal radiation R_b is given by

$$R_b = \left(a \frac{R_s}{R_{so} + b} \right) R_{bo},$$

where R_{so} is the solar radiation on a cloudless day, and it depends on latitude and day of the year. R_{bo} is given by

$$R_{bo} = \epsilon \sigma T_{Kelvin}^4,$$

where $\sigma = 4.903 \times 10^{-9} \text{ MJ m}^{-2} \text{ d}^{-1} \text{ K}^{-4}$, and ϵ is net net emissivity:

$$\epsilon = -0.02 + 0.261 \exp(-7.77 \times 10^{-4} T_{Celcius}^2).$$

The parameters a and b are determined for the climate of the area:

- $a = 1.0, b = 0.0$ for humid areas,
- $a = 1.2, b = -0.2$ for arid areas,
- $a = 1.1, b = -0.1$ for semihumid areas.

TABLE 4.5
Mean Solar Radiation R_{so} for Cloud

| Latitude | Jan | Feb | Mar | April | Mean Solar R |
|----------|-------|-------|-------|-------|--------------|
| 60° N | 2.51 | 5.99 | 13.82 | 22.32 | |
| 55° N | 4.31 | 8.67 | 16.33 | 24.16 | |
| 50° N | 6.70 | 11.43 | 18.55 | 25.83 | |
| 45° N | 9.34 | 14.36 | 20.64 | 27.21 | |
| 40° N | 12.27 | 17.04 | 22.90 | 28.34 | |
| 35° N | 14.95 | 19.55 | 24.58 | 29.31 | |
| 30° N | 17.46 | 21.65 | 25.96 | 29.85 | |
| 25° N | 19.68 | 23.45 | 27.21 | 30.14 | |
| 20° N | 21.65 | 25.00 | 28.18 | 30.14 | |
| 15° N | 23.57 | 26.50 | 29.01 | 29.85 | |
| 10° N | 25.25 | 27.63 | 29.43 | 29.60 | |
| 5° N | 26.92 | 28.47 | 29.85 | 29.31 | |
| 0° E | 28.18 | 29.18 | 30.02 | 28.47 | |
| 5° S | 28.05 | 29.85 | 29.85 | 27.76 | |
| 10° S | 30.69 | 30.44 | 29.43 | 26.80 | |
| 15° S | 31.53 | 30.69 | 28.76 | 25.54 | |
| 20° S | 32.36 | 30.69 | 27.93 | 23.99 | |
| 25° S | 32.95 | 30.69 | 27.09 | 22.32 | |
| 30° S | 33.37 | 30.44 | 25.96 | 20.81 | |
| 35° S | 33.49 | 29.73 | 24.58 | 18.97 | |
| 40° S | 33.49 | 28.76 | 22.90 | 17.04 | |
| 45° S | 33.49 | 27.76 | 21.23 | 14.95 | |
| 50° S | 32.95 | 26.38 | 19.26 | 12.85 | |
| 55° S | 32.36 | 24.83 | 17.17 | 10.47 | |
| 60° S | 31.53 | 23.15 | 15.07 | 7.83 | |

Note: August values in the Southern Hemisphere months were assumed because when actual days March did not occur.

Source: Jensen, M.E., R.D. Burman, and R.G. Society of Civil Engineers, New York, 1990. Co permission of the American Society of Civil En

We can find below a table for R_{so} , from Ward and Trimble (2003), page 100.

9.3.3 Heat Flux Density to the Ground

The heat flux density to the ground G ($\text{MJ m}^{-2}\text{d}^{-1}$) can be calculated using

$$G = 4.2 \frac{T_{i+1} - T_{i-1}}{\Delta t},$$

where Δt is the time *in days* between midpoints of time periods $i + 1$ and $i - 1$, and T is the

air temperature ($^{\circ}\text{C}$).

This expression is really a finite differences implementation of a time derivative:

$$\frac{dT}{dt} = \lim_{\Delta t \rightarrow 0} \frac{T(t + \Delta t) - T(t - \Delta t)}{2\Delta t}.$$

9.3.4 Vapor Pressure

Source: Ward and Trimble (2003), page 95.

The Vapor Pressure Deficit (VPD, in kPa) is the difference between saturation vapor pressure e_s and actual vapor pressure e_d :

$$\text{VPD} = e_s - e_d.$$

For temperatures ranging from 0 to 50 $^{\circ}\text{C}$, the saturation vapor pressure can be calculated with

$$e_s = \exp \left[\frac{16.78T - 116.9}{T + 237.3} \right],$$

and the actual vapor pressure is given by

$$e_d = e_s \frac{RH}{100},$$

where RH is the relative humidity (%), and the temperature T in the equations above is in degrees Celcius.

We can see below a graph of $e_s(T)$

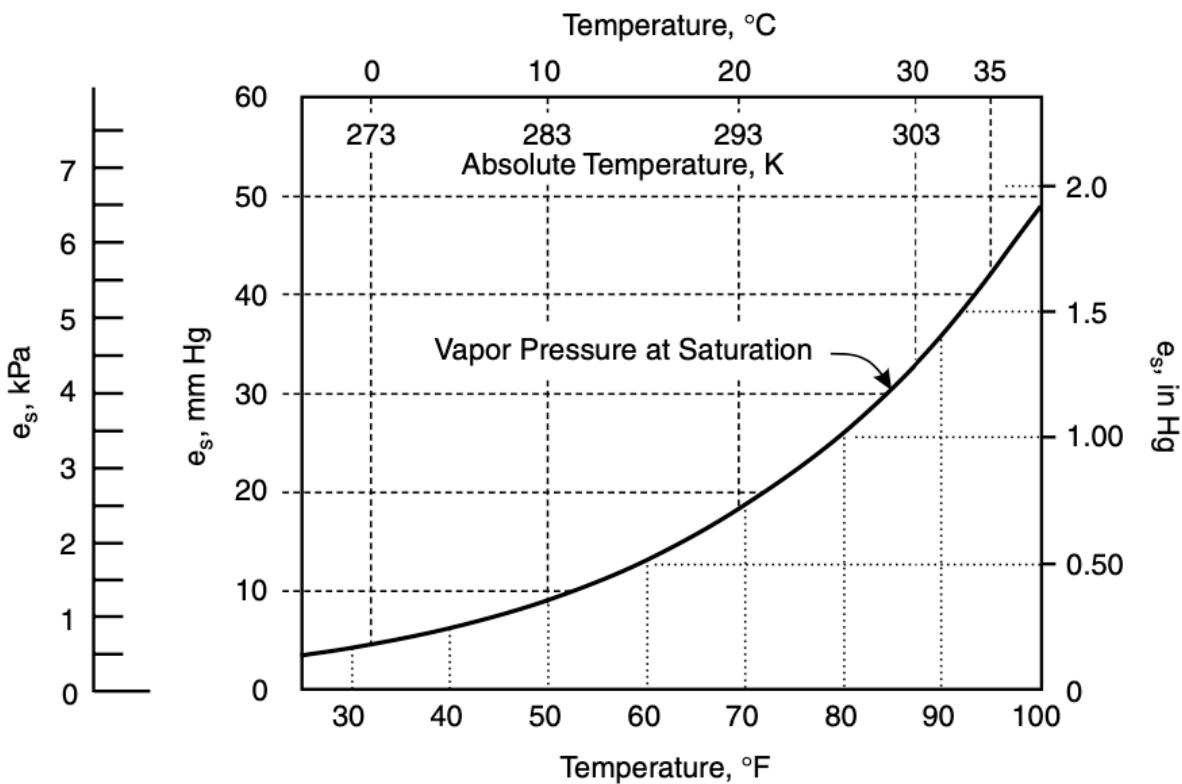


Figure 9.3: Source: Ward and Trimble (2003), page 96

The factor Δ is the slope of $e_s(T)$. See the figure below from Brutsaert, where the saturation vapor pressure is called e^*):

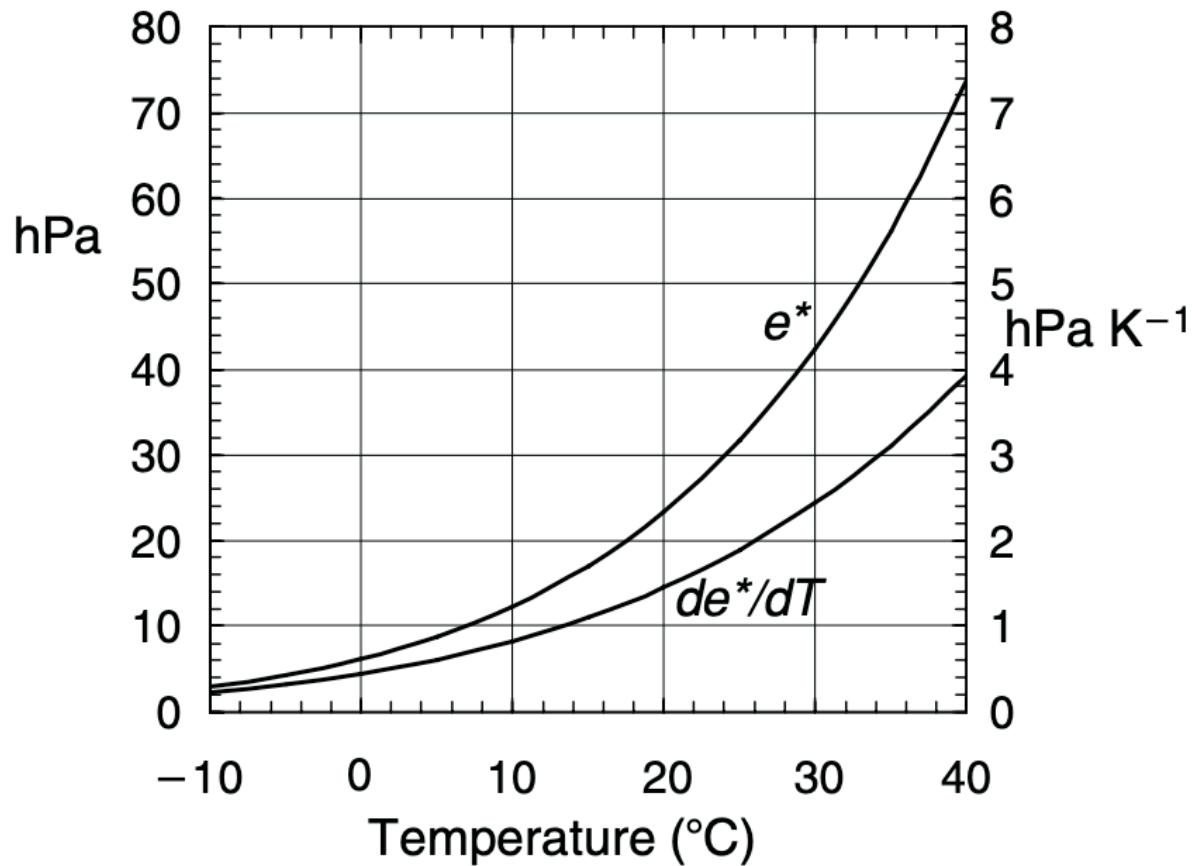


Figure 9.4: Source: Brutsaert (2005), page 28

There are a few ways of defining the function for $\Delta(T)$ ($\text{kPa } ^\circ\text{C}^{-1}$). Ward and Trimble (2003) give the following:

$$\Delta = 0.200 \cdot (0.00738 T + 0.8072)^7 - 0.000116,$$

while differentiating the exponential expression given before yields:

$$\Delta = \frac{de_s}{dT} = e_s(T) \cdot \frac{4098.79}{(T + 237.3)^2}.$$

9.3.5 Wind Function

Source: Ward and Trimble (2003), page 108

$$f(u)=0.26(1.0+0.54\,u_2)$$

10 Meaning of “potential” evapotranspiration

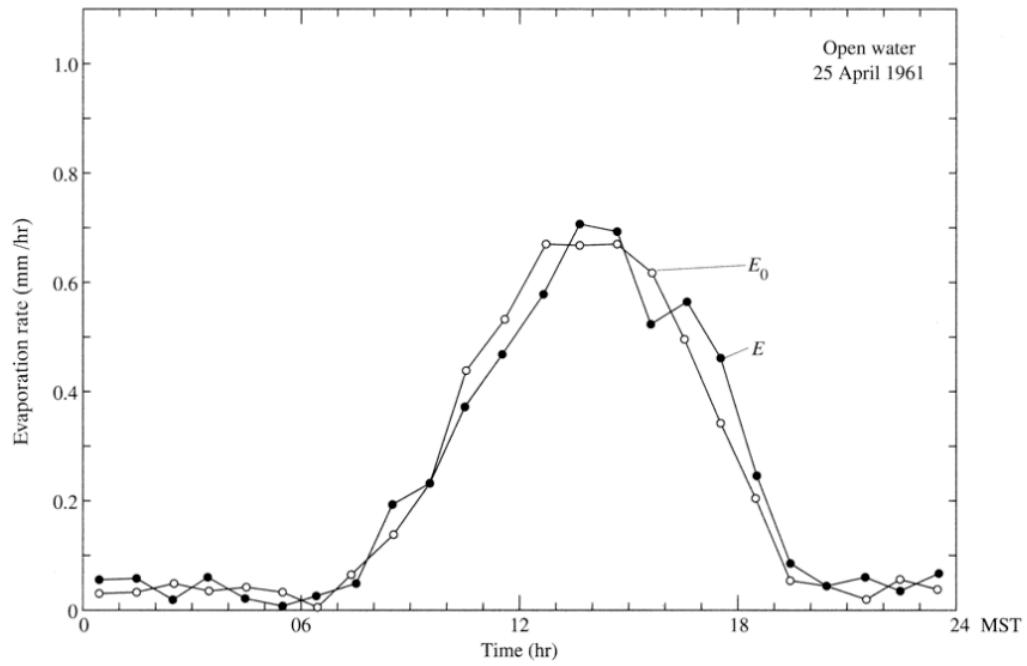


Figure 6.3 Comparison of hourly evaporation rates determined from measurements in a shallow pan (E) with those computed via the combination approach [E_0 , equation (6.36)] at Tempe, Arizona [Van Bavel (1966). Potential evaporation: The combination concept and its experimental verification. *Water Resources Research* 2:455–467, with permission of the American Geophysical Union].

10.0.1 Crop Coefficient

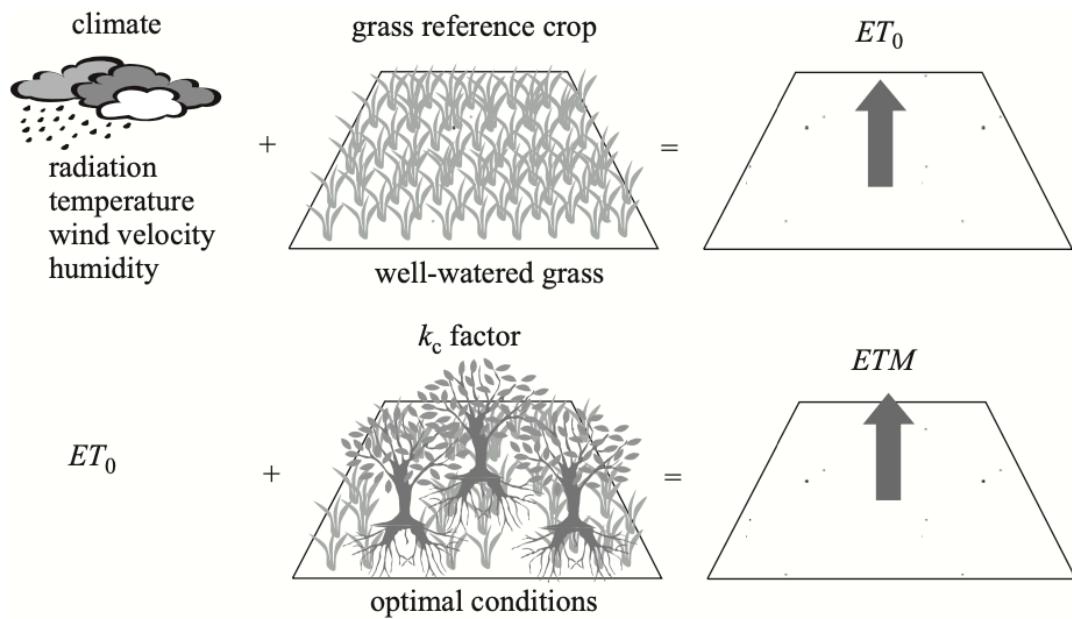


Fig. 5.9 : Water requirements of crops (ETM) and reference evapotranspiration (ET₀) (modified from FAO 1998).

$$E_t = k_c E_{tr, tp}$$

E_t = actual ET

k_c = crop coefficient

E_{tr} = reference crop ET

E_{tp} = potential ET

TABLE 4.7
**Seasonal Consumptive Use Coefficients K for Irrigated Crops
 in the Western U.S.^a**

| Crop | Length of Normal Growing Season ^b | Coefficient K^c |
|---------------------|--|-------------------|
| Alfalfa | Between frosts | 0.80 to 0.90 |
| Bananas | Full year | 0.80 to 1.00 |
| Beans | 3 months | 0.60 to 0.70 |
| Cocoa | Full year | 0.70 to 0.80 |
| Coffee | Full year | 0.70 to 0.80 |
| Corn (maize) | 4 months | 0.75 to 0.85 |
| Cotton | 7 months | 0.60 to 0.70 |
| Dates | Full year | 0.65 to 0.85 |
| Flax | 7 to 8 months | 0.70 to 0.80 |
| Grains, small | 3 months | 0.75 to 0.85 |
| Grains, sorghum | 4 to 5 months | 0.70 to 0.80 |
| Oil seeds | 3 to 5 months | 0.65 to 0.75 |
| Orchard crops | | |
| Avocado | Full year | 0.50 to 0.55 |
| Grapefruit | Full year | 0.55 to 0.65 |
| Orange and lemon | Full year | 0.45 to 0.55 |
| Walnuts | Between frosts | 0.60 to 0.70 |
| Deciduous | Between frosts | 0.60 to 0.70 |
| Pasture crops | | |
| Grass | Between frosts | 0.75 to 0.85 |
| Ladino white clover | Between frosts | 0.80 to 0.85 |
| Potatoes | 3 to 5 months | 0.65 to 0.75 |
| Rice | 3 to 5 months | 1.00 to 1.10 |
| Soybeans | 140 days | 0.65 to 0.70 |
| Sugar beets | 6 months | 0.65 to 0.75 |
| Sugarcane | Full year | 0.80 to 0.90 |
| Tobacco | 4 months | 0.70 to 0.80 |
| Tomatoes | 4 months | 0.65 to 0.70 |
| Vineyard | 5 to 7 months | 0.50 to 0.60 |

^a From USDA (1970).

^b Length of season depends largely on variety and time of year when the crop is grown. Annual crops grown during the winter period may take much longer than if grown in the summer.

^c The lower values of K for use in the Blaney-Criddle formula are for the more humid areas; the higher values are for the more arid climates.

Source: Jensen, M.E. et al., *Evapotranspiration and Irrigation Water Requirements*, ASCE, New York, 1990. ©1990 by ASCE. Reproduced with permission.

10.1 Pitfalls

Different books and papers will present slightly different versions of the Penman equation. Basically, they differ in the units they use for the various components, and one should be vary aware of what inputs any given equation is expecting to get.

11 Exercises

We will calculate evapotranspiration using two methods: Thornthwaite and Penman. After that, we will compare these estimates with measurements of pan evaporation.

11.1 Download data from the IMS

Please follow the instructions below **exactly as they are written**. Go to the [Israel Meteorological Service website](#), and download the following data:

1. 10-min data

- On the navigation bar on the left, choose “10 Minutes Observations”
- Clock: Local time winter (UTC +2)
- Choose the following date range: 01/01/2020 00:00 to 01/01/2021 00:00
- Choose station Bet Dagan
- Select units: Celcius, m/s, KJ/m²
- Under “Select parameters”, choose “Check All”
- Choose option “by station”, then “Submit”
- “Download Result as” CSV, call it `bet-dagan-10min.csv`

2. radiation data

- On the navigation bar on the left, choose “Hourly Radiation”
- Clock: Local time winter (UTC +2)
- Choose the following date range: 01/01/2020 00:00 to 01/01/2021 00:00
- Select hours: Check all hours
- Choose station Bet Dagan
- Select units: KJ/m²
- Under “Select parameters”, choose “Check All”
- “Download Result as” CSV, call it `bet-dagan-radiation.csv`

3. pan evaporation data

- On the navigation bar on the left, choose “Daily Observations”
- Choose the following date range: 01/01/2020 00:00 to 01/01/2021 00:00
- Choose station Bet Dagan Man

- Select units: Celcius
- Under “Select parameters”, choose “Check All”
- “Download Result as” CSV, call it `bet-dagan-pan.csv`

If for some reason you can't download the files following the instructions above, click here:

- `bet-dagan-10min.csv`
- `bet-dagan-radiation.csv`
- `bet-dagan-pan.csv`

11.2 Install and import relevant packages

We will need to use two new packages:

- `pyet`: Estimation of Potential Evapotranspiration
- `noaa_ftp`: Download data from NOAA

If you don't have them installed yet, run this:

```
!pip install pyet
!pip install noaa_ftp
```

Once they are installed, import all the necessary packages for today's exercises.

```
import matplotlib.pyplot as plt
import numpy as np
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_theme(style="ticks", font_scale=1.5)
import pyet
from noaa_ftp import NOAA
```

11.3 import 10-minute data

```

df = pd.read_csv('bet-dagan-10min.csv',
                  # encoding = "ISO-8859-8", # this shows hebrew characters properly
                  na_values=["-"]           # substitute "--" for NaN
                  )
df['timestamp'] = pd.to_datetime(df['Date & Time (Winter)'], dayfirst=True)
df = df.set_index('timestamp')
# choose only relevant columns to us
# if we don't do this the taking the mean will fail because not all columns are numeric. try
df = df[["Temperature (°C)",
          "Wind speed (m/s)",
          "Pressure at station level (hPa)",
          "Relative humidity (%)"]]
# resample to daily data according to "mean"
df = df.resample('D').mean()
# convert hecto pascals (hPa) to kilo pascals (kPa)
df["Pressure (kPa)"] = df["Pressure at station level (hPa)"] / 10.0
df

```

| timestamp | Temperature (°C) | Wind speed (m/s) | Pressure at station level (hPa) | Relative humidity (%) |
|------------|------------------|------------------|---------------------------------|-----------------------|
| 2020-01-01 | 12.375000 | 1.552083 | 1013.263889 | 80.590278 |
| 2020-01-02 | 12.020833 | 2.207639 | 1011.922917 | 85.631944 |
| 2020-01-03 | 12.962500 | 4.763194 | 1013.757639 | 60.756944 |
| 2020-01-04 | 10.849306 | 5.439583 | 1011.581250 | 76.909722 |
| 2020-01-05 | 12.956250 | 4.765278 | 1012.361806 | 79.583333 |
| ... | ... | ... | ... | ... |
| 2020-12-28 | 14.797917 | 2.631915 | 1014.429861 | 58.729167 |
| 2020-12-29 | 14.146528 | 1.493750 | 1015.031944 | 71.215278 |
| 2020-12-30 | 14.186806 | 1.776389 | 1013.234028 | 68.923611 |
| 2020-12-31 | 14.915278 | 1.395833 | 1011.840972 | 75.465278 |
| 2021-01-01 | 11.600000 | 0.700000 | 1011.800000 | 95.000000 |

11.4 import radiation data

```

df_rad = pd.read_csv('bet-dagan-radiation.csv',
                      na_values=["-"]
                      )
df_rad['Date'] = pd.to_datetime(df_rad['Date'], dayfirst=True)

```

```
df_rad = df_rad.set_index('Date')
df_rad
```

| Date | Station | Radiation type | Hourly radiation 05-06 (KJ/m^2) | Hourly radiation 05-06 (MJ/m^2) |
|------------|-------------------------------|----------------|---------------------------------|---------------------------------|
| 2020-01-01 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 10.8 |
| 2020-01-01 | Bet Dagan Rad 01/1991-04/2024 | Direct | 0.0 | 3.6 |
| 2020-01-01 | Bet Dagan Rad 01/1991-04/2024 | Diffused | 0.0 | 10.8 |
| 2020-01-02 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 10.8 |
| 2020-01-02 | Bet Dagan Rad 01/1991-04/2024 | Direct | 0.0 | 3.6 |
| ... | ... | ... | ... | ... |
| 2020-12-31 | Bet Dagan Rad 01/1991-04/2024 | Direct | 0.0 | 0.0 |
| 2020-12-31 | Bet Dagan Rad 01/1991-04/2024 | Diffused | 0.0 | 14.4 |
| 2021-01-01 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 14.4 |
| 2021-01-01 | Bet Dagan Rad 01/1991-04/2024 | Direct | 0.0 | 0.0 |
| 2021-01-01 | Bet Dagan Rad 01/1991-04/2024 | Diffused | 0.0 | 14.4 |

Choose only “Global” radiation. Then sum all hours of the day, and convert from kJ to MJ.

```
df_rad = df_rad[df_rad["Radiation type"] == "Global"]
df_rad['daily_radiation_MJ_per_m2_per_day'] = (df_rad.iloc[:, 2:]      # take all rows, columns
                                              .sum(axis=1) /          # sum all columns
                                              1000                  # divide by 1000
                                              )
df_rad
```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_93264/3322402408.py:2: 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/indexing.html#inplace-evaluation
df_rad['daily_radiation_MJ_per_m2_per_day'] = (df_rad.iloc[:, 2:]      # take all rows, columns
                                              .sum(axis=1) /          # sum all columns
                                              1000                  # divide by 1000
                                              )
df_rad
```

| Date | Station | Radiation type | Hourly radiation 05-06 (KJ/m^2) | Hourly radiation 05-06 (MJ/m^2) |
|------------|-------------------------------|----------------|---------------------------------|---------------------------------|
| 2020-01-01 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 10.8 |
| 2020-01-02 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 10.8 |
| 2020-01-03 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 10.8 |

| Date | Station | Radiation type | Hourly radiation 05-06 (KJ/m^2) | Hourly radiation 07-08 (KJ/m^2) |
|------------|-------------------------------|----------------|---------------------------------|---------------------------------|
| 2020-01-04 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 3.6 |
| 2020-01-05 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 7.2 |
| ... | ... | ... | ... | ... |
| 2020-12-28 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 14.4 |
| 2020-12-29 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 14.4 |
| 2020-12-30 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 21.6 |
| 2020-12-31 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 14.4 |
| 2021-01-01 | Bet Dagan Rad 01/1991-04/2024 | Global | 0.0 | 14.4 |

Now we can add the daily radiation we have just calculated to the first dataframe containing temperature, RH, etc.

```
df['daily_radiation_MJ_per_m2_per_day'] = df_rad['daily_radiation_MJ_per_m2_per_day']
df
```

| timestamp | Temperature (°C) | Wind speed (m/s) | Pressure at station level (hPa) | Relative humidity (%) |
|------------|------------------|------------------|---------------------------------|-----------------------|
| 2020-01-01 | 12.375000 | 1.552083 | 1013.263889 | 80.590278 |
| 2020-01-02 | 12.020833 | 2.207639 | 1011.922917 | 85.631944 |
| 2020-01-03 | 12.962500 | 4.763194 | 1013.757639 | 60.756944 |
| 2020-01-04 | 10.849306 | 5.439583 | 1011.581250 | 76.909722 |
| 2020-01-05 | 12.956250 | 4.765278 | 1012.361806 | 79.583333 |
| ... | ... | ... | ... | ... |
| 2020-12-28 | 14.797917 | 2.631915 | 1014.429861 | 58.729167 |
| 2020-12-29 | 14.146528 | 1.493750 | 1015.031944 | 71.215278 |
| 2020-12-30 | 14.186806 | 1.776389 | 1013.234028 | 68.923611 |
| 2020-12-31 | 14.915278 | 1.395833 | 1011.840972 | 75.465278 |
| 2021-01-01 | 11.600000 | 0.700000 | 1011.800000 | 95.000000 |

11.5 import pan evaporation data

```
df_pan = pd.read_csv('bet-dagan-pan.csv',
                     na_values=['-']                      # substitute "--" for NaN
                     )
df_pan['Date'] = pd.to_datetime(df_pan['Date'], dayfirst=True)
```

```
df_pan = df_pan.set_index('Date')
df_pan
```

| Date | Station | | Maximum Temperature (°C) | Minimum Temperature (°C) |
|------------|---------------|-----------------|--------------------------|--------------------------|
| 2020-01-01 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-01-02 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-01-03 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-01-04 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-01-05 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| ... | ... | ... | ... | ... |
| 2020-12-28 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-12-29 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-12-30 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2020-12-31 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |
| 2021-01-01 | Bet Dagan Man | 01/1964-04/2024 | NaN | NaN |

11.6 calculate penman

We need some data about the Bet Dagan Station. [See here.](#)

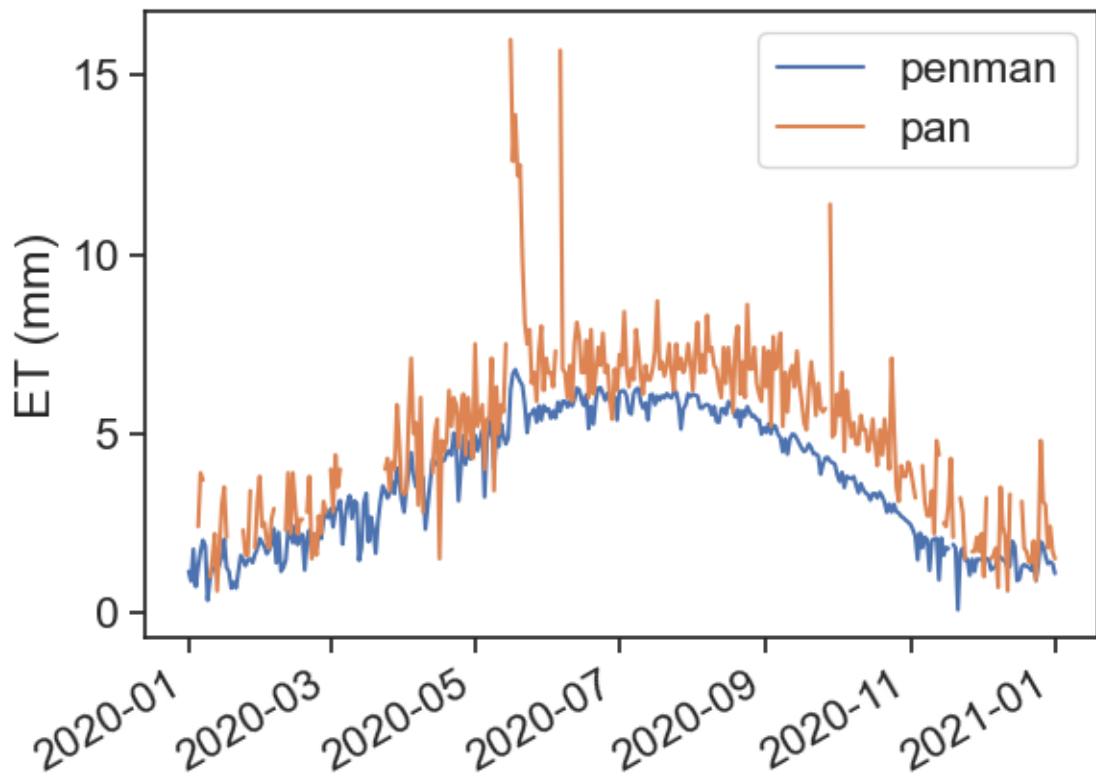
- Latitude: 32.0073°
- Elevation: 31 m

```
# the site elevation [m]
elevation = 31.0
# the site latitude [rad]
latitude = pyet.utils.deg_to_rad(32.0073)
penm = pyet.combination.penman(tmean=df ["Temperature (°C)"],
                                 wind=df ["Wind speed (m/s)"],
                                 pressure=df ["Pressure (kPa)"],
                                 elevation=elevation,
                                 rh=df ["Relative humidity (%)" ],
                                 rs=df ["daily_radiation_MJ_per_m2_per_day"],
                                 lat=latitude,
                                 )
```

```

fig, ax = plt.subplots(1)
ax.plot(penm, label="penman")
ax.plot(df_pan["Daily evaporation type A (mm)"], label="pan")
ax.legend()
plt.gcf().autofmt_xdate() # makes slanted dates
ax.set_ylabel("ET (mm)");

```



11.7 Thornthwaite

$$E = 16 \left[\frac{10 T^{\text{monthly mean}}}{I} \right]^a,$$

where

$$I = \sum_{i=1}^{12} \left[\frac{T_i^{\text{monthly mean}}}{5} \right]^{1.514},$$

and

$$\begin{aligned} a = & 6.75 \times 10^{-7} I^3 \\ & - 7.71 \times 10^{-5} I^2 \\ & + 1.792 \times 10^{-2} I \\ & + 0.49239 \end{aligned}$$

- E is the monthly potential ET (mm)
- $T_{\text{monthly mean}}$ is the mean monthly temperature in °C
- I is a heat index
- a is a location-dependent coefficient

From df, make a new dataframe, df_th, that stores monthly temperatures means. Use resample function.

```
# monthly data
df_th = (df['Temperature (°C)'].resample('MS') # MS assigns mean to first day in the month
          .mean()
          .to_frame()
        )

# we now add 14 days to the index, so that all monthly data is in the middle of the month
# not really necessary, makes plot look better
df_th.index = df_th.index + pd.DateOffset(days=14)
df_th
```

| timestamp | Temperature (°C) |
|------------|------------------|
| 2020-01-15 | 12.484812 |
| 2020-02-15 | 14.044349 |
| 2020-03-15 | 16.371381 |
| 2020-04-15 | 18.476339 |
| 2020-05-15 | 23.177769 |
| 2020-06-15 | 24.666423 |
| 2020-07-15 | 27.380466 |
| 2020-08-15 | 28.099328 |
| 2020-09-15 | 28.421644 |
| 2020-10-15 | 25.058944 |
| 2020-11-15 | 19.266082 |
| 2020-12-15 | 15.915031 |

| | Temperature (°C) |
|------------|------------------|
| timestamp | |
| 2021-01-15 | 11.600000 |

Calculate I , then a , and finally E_p . Add E_p as a new column in df_th.

```
# Preparing "I" for the Thornthwaite equation
I = np.sum(
    (
        df_th['Temperature (°C)'] / 5
    ) ** (1.514)
)

# Preparing "a" for the Thornthwaite equation
a = (+6.75e-7 * I**3
      -7.71e-5 * I**2
      +1.792e-2 * I
      + 0.49239)

# The final Thornthwaite model for monthly potential ET (mm)
df_th['Ep (mm/month)'] = 16 * (
    (
        10 * df_th['Temperature (°C)'] / I
    ) ** a
)
df_th
```

| | Temperature (°C) | Ep (mm/month) |
|------------|------------------|---------------|
| timestamp | | |
| 2020-01-15 | 12.484812 | 20.695370 |
| 2020-02-15 | 14.044349 | 27.765987 |
| 2020-03-15 | 16.371381 | 40.716009 |
| 2020-04-15 | 18.476339 | 55.071668 |
| 2020-05-15 | 23.177769 | 96.997621 |
| 2020-06-15 | 24.666423 | 113.308714 |
| 2020-07-15 | 27.380466 | 147.047919 |
| 2020-08-15 | 28.099328 | 156.877841 |
| 2020-09-15 | 28.421644 | 161.409587 |
| 2020-10-15 | 25.058944 | 117.864618 |
| 2020-11-15 | 19.266082 | 61.138581 |

| timestamp | Temperature (°C) | Ep (mm/month) |
|------------|------------------|---------------|
| 2020-12-15 | 15.915031 | 37.941007 |
| 2021-01-15 | 11.600000 | 17.225130 |

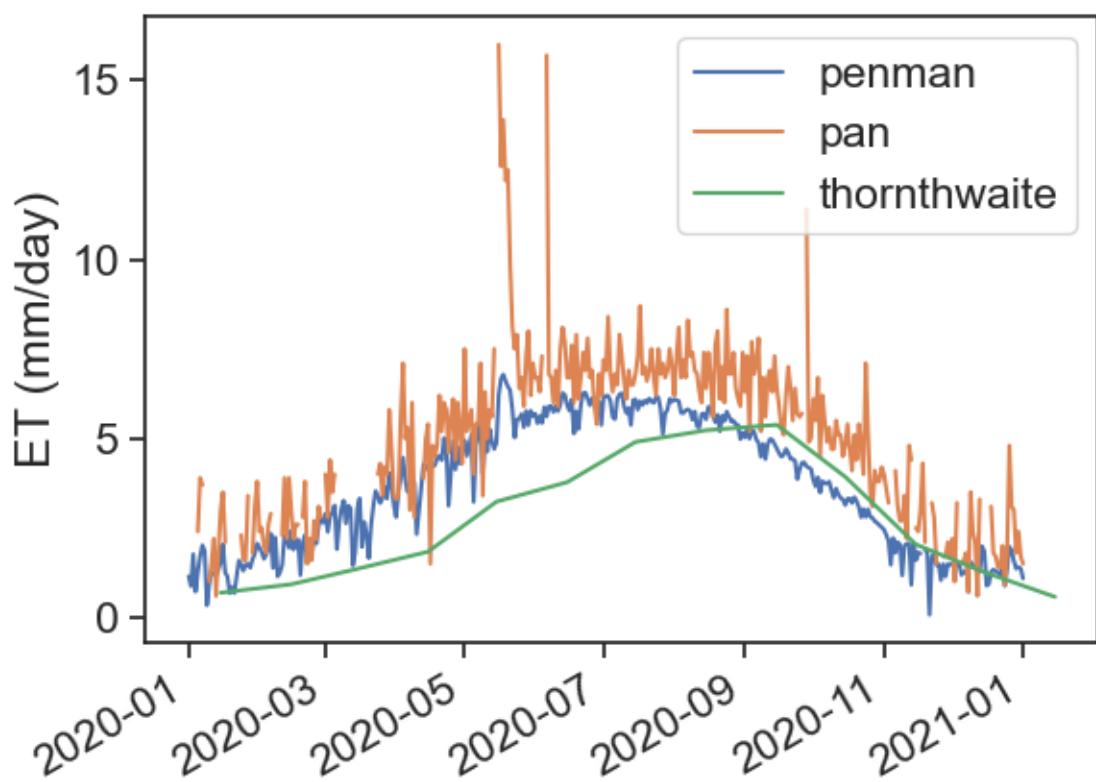
```

fig, ax = plt.subplots(1)
ax.plot(penm, label="penman")
ax.plot(df_pan["Daily evaporation type A (mm)"], label="pan")
ax.plot(df_th['Ep (mm/month)']/30, label="thornthwaite")

ax.legend()
plt.gcf().autofmt_xdate() # makes slanted dates
ax.set_ylabel("ET (mm/day)")

```

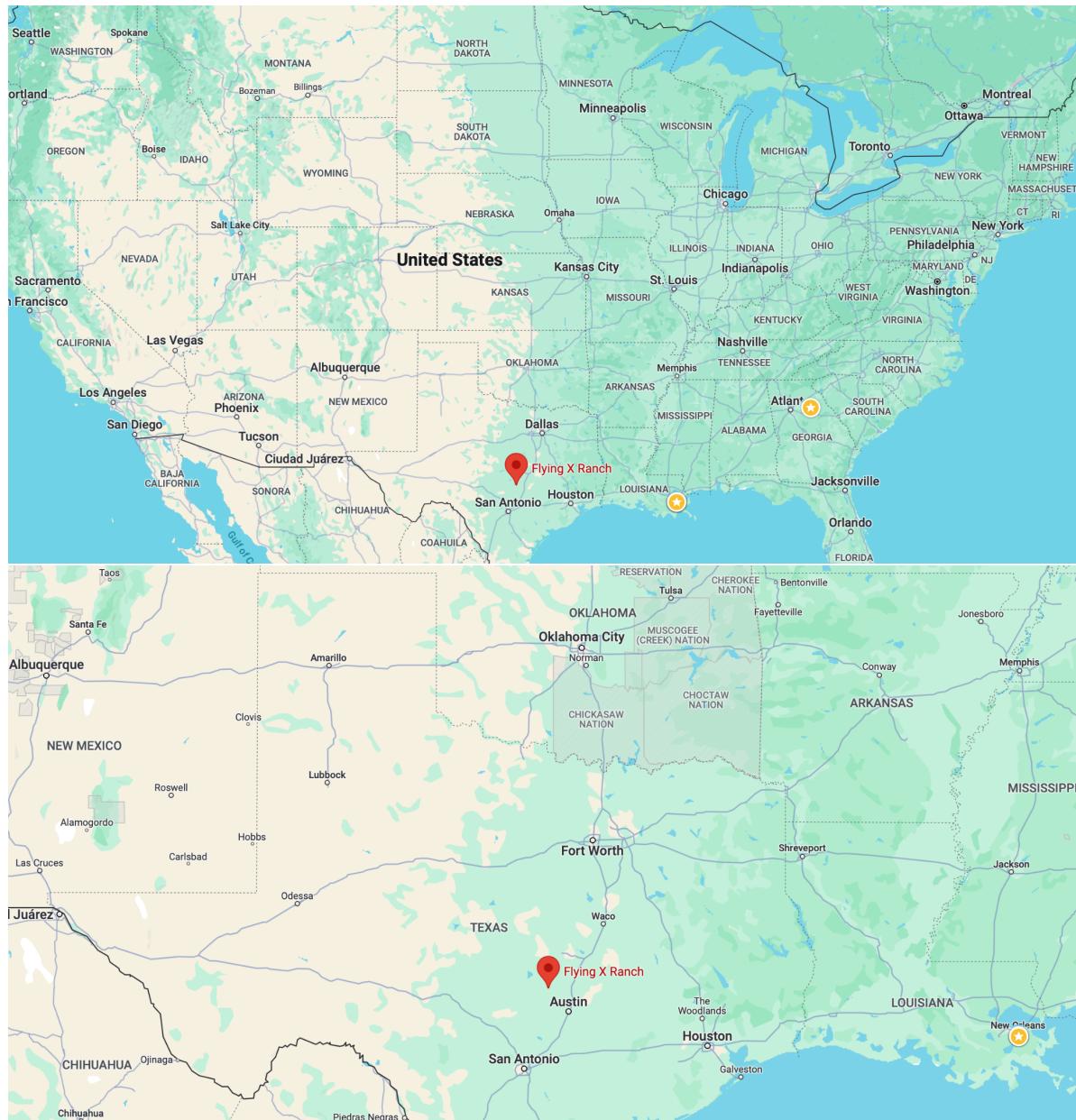
Text(0, 0.5, 'ET (mm/day)')



11.8 Data from NOAA

Let's download data from a different repository. More specifically, we will retrieve sub-hourly data from the U.S. Climate Reference Network. We can see all the sites in [this map](#). Besides the sub-hourly data, we can find [other datasets](#) (monthly, daily, etc).

As an example, we will choose the station near [Austin, Texas](#).



The Austin 33 NW station has coordinates (30.62000, -98.08000) and an elevation of 1361 m above sea level. Its climate is classified as humid subtropical. Rainfall is evenly distributed rainfall throughout the year, averaging around 870 mm per year. The summer is hot and long, where average maximum temperatures reach 35 °C in July and August. The coldest month is January, with average maximum temperatures of 17 °C.

This station lies in the transition zone between the humid regions of the American Southwest and the dry deserts of the American Southwest.

In order to download, we will access the data from the FTP client using the python package `noaa_ftp`.

The `dir` command list everything in the folder:

```
noaa_dir = NOAA("ftp.ncdc.noaa.gov", 'pub/data/uscrn/products/subhourly01').dir()
```

| | | | | | | | | |
|------------|---|-----|-----|-------|-----|----|-------|-------------|
| drwxrwsr-x | 2 | ftp | ftp | 8192 | Oct | 7 | 2020 | 2006 |
| drwxrwsr-x | 2 | ftp | ftp | 8192 | Nov | 10 | 2021 | 2007 |
| drwxrwsr-x | 2 | ftp | ftp | 8192 | Dec | 1 | 2020 | 2008 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | May | 25 | 2021 | 2009 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 10 | 2021 | 2010 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 12 | 2021 | 2011 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 12 | 2021 | 2012 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 15 | 2021 | 2013 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 15 | 2021 | 2014 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 12 | 2021 | 2015 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 12 | 2021 | 2016 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 15 | 2021 | 2017 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 12 | 2021 | 2018 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 24 | 2021 | 2019 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 30 | 2021 | 2020 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Jan | 29 | 2022 | 2021 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Aug | 23 | 2022 | 2022 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Nov | 29 | 2023 | 2023 |
| drwxrwsr-x | 2 | ftp | ftp | 12288 | Apr | 9 | 16:53 | 2024 |
| -rw-rw-r-x | 1 | ftp | ftp | 2157 | Feb | 18 | 2022 | headers.txt |
| -rw-rw-r-x | 1 | ftp | ftp | 456 | Oct | 7 | 2020 | HEADERS.txt |
| -rw-rw-r-x | 1 | ftp | ftp | 14892 | Feb | 18 | 2022 | readme.txt |
| -rw-rw-r-x | 1 | ftp | ftp | 14936 | Sep | 21 | 2021 | README.txt |
| drwxrwsr-x | 2 | ftp | ftp | 8192 | May | 27 | 01:50 | snapshots |

It's worth clicking [here](#) to see this directory with your own eyes. You can use your browser to explore this. Click on `readme.txt` and, well, read it.

Let's download two files:

- sub-hourly data for the year 2022 for Austin, TX.
- the `HEADERS.txt` contains the names of the columns in the csv.

Type the following lines:

```
noaa = NOAA("ftp.ncdc.noaa.gov", 'pub/data/uscrn/products/subhourly01').download('HEADERS.txt')
noaa = NOAA("ftp.ncdc.noaa.gov", 'pub/data/uscrn/products/subhourly01/2022').download('CRNS0101-05-2022-TX_Austin_33_NW.txt')
```

If for some reason you can't download directly from NOAA, click [here](#) to get the files:

- `HEADERS.txt`
- `CRNS0101-05-2022-TX_Austin_33_NW.txt`

```
# Read column names from another file
column_names = pd.read_csv('HEADERS.txt',
                           header=None,
                           sep='\s+',
                           )

# Read CSV file using column names from another file
df = pd.read_csv("CRNS0101-05-2022-TX_Austin_33_NW.txt", # file to read
                  sep='\s+', # use (any number of) white spaces as delimiter between columns
                  names=column_names.iloc[1], # column names from row i=1 of "column_names"
                  na_values=[-99, -9999, -99999], # substitute these values by NaN
                  )

# make integer column LST_DATE as string
df['LST_DATE'] = df['LST_DATE'].astype(str).apply(lambda x: f'{x:0>4}')
# make integer column LST_DATE as string
# pad numbers with 0 from the left, such that 15 becomes 0015
df['LST_TIME'] = df['LST_TIME'].apply(lambda x: f'{x:0>4}')
# combine both DATE and TIME
df['datetime'] = pd.to_datetime(df['LST_DATE'] + df['LST_TIME'], format='%Y%m%d%H%M')
df = df.set_index('datetime')
df
```

| datetime | WBANNO | UTC_DATE | UTC_TIME | LST_DATE | LST_TIME | CRX_VN | LO |
|---------------------|--------|----------|----------|----------|----------|--------|----|
| 2021-12-31 18:05:00 | 23907 | 20220101 | 5 | 20211231 | 1805 | 2.623 | -9 |
| 2021-12-31 18:10:00 | 23907 | 20220101 | 10 | 20211231 | 1810 | 2.623 | -9 |
| 2021-12-31 18:15:00 | 23907 | 20220101 | 15 | 20211231 | 1815 | 2.623 | -9 |
| 2021-12-31 18:20:00 | 23907 | 20220101 | 20 | 20211231 | 1820 | 2.623 | -9 |

| datetime | WBANNO | UTC_DATE | UTC_TIME | LST_DATE | LST_TIME | CRX_VN | LO |
|---------------------|--------|----------|----------|----------|----------|--------|-----|
| 2021-12-31 18:25:00 | 23907 | 20220101 | 25 | 20211231 | 1825 | 2.623 | -9 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 2022-12-31 17:40:00 | 23907 | 20221231 | 2340 | 20221231 | 1740 | 2.623 | -9 |
| 2022-12-31 17:45:00 | 23907 | 20221231 | 2345 | 20221231 | 1745 | 2.623 | -9 |
| 2022-12-31 17:50:00 | 23907 | 20221231 | 2350 | 20221231 | 1750 | 2.623 | -9 |
| 2022-12-31 17:55:00 | 23907 | 20221231 | 2355 | 20221231 | 1755 | 2.623 | -9 |
| 2022-12-31 18:00:00 | 23907 | 20230101 | 0 | 20221231 | 1800 | 2.623 | -9 |

The provided data is not the same as what is provided by the IMS.

- Now we don't have air pressure values, so we need to provide elevation.
- We do have R_n (net radiation), so there is no need to provide latitude.

Attention! According to the headers file, net radiation provided by NOAA is in W m^{-2} , but pyet requires it to be $\text{MJ m}^{-2} \text{ d}^{-1}$. We need to aggregate the downloaded data into daily radiation.

Data comes in 5-minute intervals, so if we have a value $x \text{ W m}^{-2}$ over a 5-minute interval, the total amount of energy is:

$$x \frac{\text{W}}{\text{m}^{-2}} \times 5 \text{ min} = x \frac{\text{J}}{\text{m}^{-2} \cdot \text{s}} \times 5 \cdot 60 \text{ s} = x \cdot 5 \cdot 60 \frac{\text{J}}{\text{m}^{-2}}$$

Let's call $X = \sum_{\text{day}} x$. Then, summing all energy in 1 day:

$$\sum_{\text{day}} x \cdot 5 \cdot 60 \frac{\text{J}}{\text{m}^{-2} \cdot \text{day}} = \sum_{\text{day}} x \cdot 5 \cdot 60 \cdot 10^{-6} \frac{\text{MJ}}{\text{m}^{-2} \cdot \text{day}}$$

Make a new dataframe with daily means for temperature, relative humidity and wind speed.

```
df_TX = df[['AIR_TEMPERATURE',
             'RELATIVE_HUMIDITY',
             'WIND_1_5']].resample('D').mean()
df_TX
```

| datetime | AIR_TEMPERATURE | RELATIVE_HUMIDITY | WIND_1_5 |
|------------|-----------------|-------------------|----------|
| 2021-12-31 | 21.721127 | 79.985915 | 2.113662 |
| 2022-01-01 | 18.336806 | 65.833333 | 2.586840 |

| | AIR_TEMPERATURE | RELATIVE_HUMIDITY | WIND_1_5 |
|------------|-----------------|-------------------|----------|
| datetime | | | |
| 2022-01-02 | -0.222222 | 46.187500 | 3.849757 |
| 2022-01-03 | 4.592708 | 36.927083 | 0.892778 |
| 2022-01-04 | 9.964583 | 41.996528 | 2.428924 |
| ... | ... | ... | ... |
| 2022-12-27 | 6.534375 | 50.871528 | 1.881597 |
| 2022-12-28 | 14.418056 | 65.090278 | 3.289167 |
| 2022-12-29 | 16.878125 | 72.934028 | 2.068750 |
| 2022-12-30 | 13.047917 | 51.006944 | 1.180729 |
| 2022-12-31 | 15.010138 | 58.387097 | 2.590276 |

```
X = df['SOLAR_RADIATION'].resample('D').sum()
df_TX['SOLAR_RADIATION'] = X * 5 * 60 * 1e-6
df_TX
```

| | AIR_TEMPERATURE | RELATIVE_HUMIDITY | WIND_1_5 | SOLAR_RADIATION |
|------------|-----------------|-------------------|----------|-----------------|
| datetime | | | | |
| 2021-12-31 | 21.721127 | 79.985915 | 2.113662 | 0.0000 |
| 2022-01-01 | 18.336806 | 65.833333 | 2.586840 | 7.2870 |
| 2022-01-02 | -0.222222 | 46.187500 | 3.849757 | 14.2536 |
| 2022-01-03 | 4.592708 | 36.927083 | 0.892778 | 14.4309 |
| 2022-01-04 | 9.964583 | 41.996528 | 2.428924 | 14.2056 |
| ... | ... | ... | ... | ... |
| 2022-12-27 | 6.534375 | 50.871528 | 1.881597 | 11.5200 |
| 2022-12-28 | 14.418056 | 65.090278 | 3.289167 | 10.9917 |
| 2022-12-29 | 16.878125 | 72.934028 | 2.068750 | 5.9829 |
| 2022-12-30 | 13.047917 | 51.006944 | 1.180729 | 2.9967 |
| 2022-12-31 | 15.010138 | 58.387097 | 2.590276 | 12.7248 |

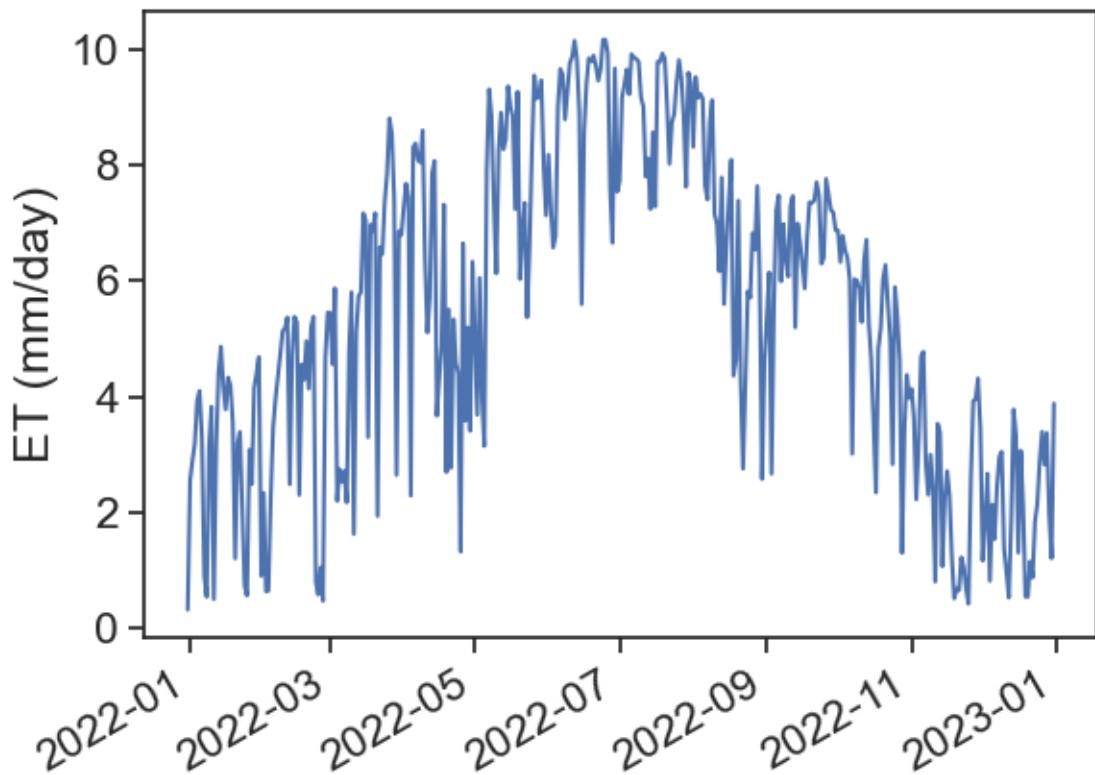
Let's use PYET to calculate the potential ET. This is the time to go to the library's GitHub page and read the functions: <https://github.com/pyet-org/pyet/tree/master>

```
elevation = 358
penm2 = pyet.combination.penman(tmean=df_TX["AIR_TEMPERATURE"],
                                 wind=df_TX["WIND_1_5"],
                                 elevation=elevation,
                                 rh=df_TX["RELATIVE_HUMIDITY"],
                                 rn=df_TX["SOLAR_RADIATION"],
                                 )
```

```

fig, ax = plt.subplots(1)
ax.plot(penm2, label="penman")
plt.gcf().autofmt_xdate() # makes slanted dates
ax.set_ylabel("ET (mm/day)");

```



Does this make sense? How can we know?

What happens if we download data from a station, and we're missing Solar Radiation? What to do? The library PYET can infer solar radiation given the station latitude and the number of daylight hours in the day.

```

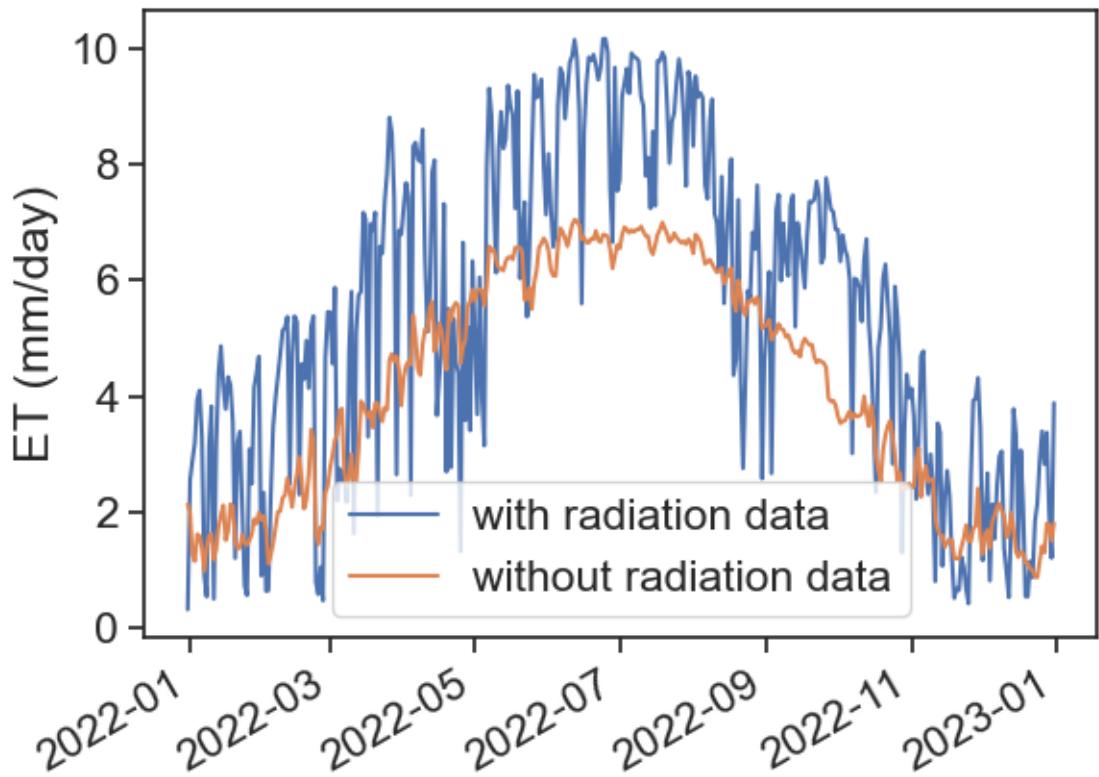
lat = 30.62 # degrees
lat = pyet.utils.deg_to_rad(lat)
df_TX['daylight_hours'] = pyet.meteo_utils.daylight_hours(tmean.index, lat)
penm3 = pyet.combination.penman(tmean=df_TX["AIR_TEMPERATURE"],
                                 wind=df_TX["WIND_1_5"],
                                 elevation=elevation,
                                 rh=df_TX["RELATIVE_HUMIDITY"],
                                 lat=lat,

```

```
n=df_TX['daylight_hours'],
)
```

```
fig, ax = plt.subplots(1)
ax.plot(penm2, label="with radiation data")
ax.plot(penm3, label="without radiation data")
ax.legend()
plt.gcf().autofmt_xdate() # makes slanted dates
ax.set_ylabel("ET (mm/day)")
```

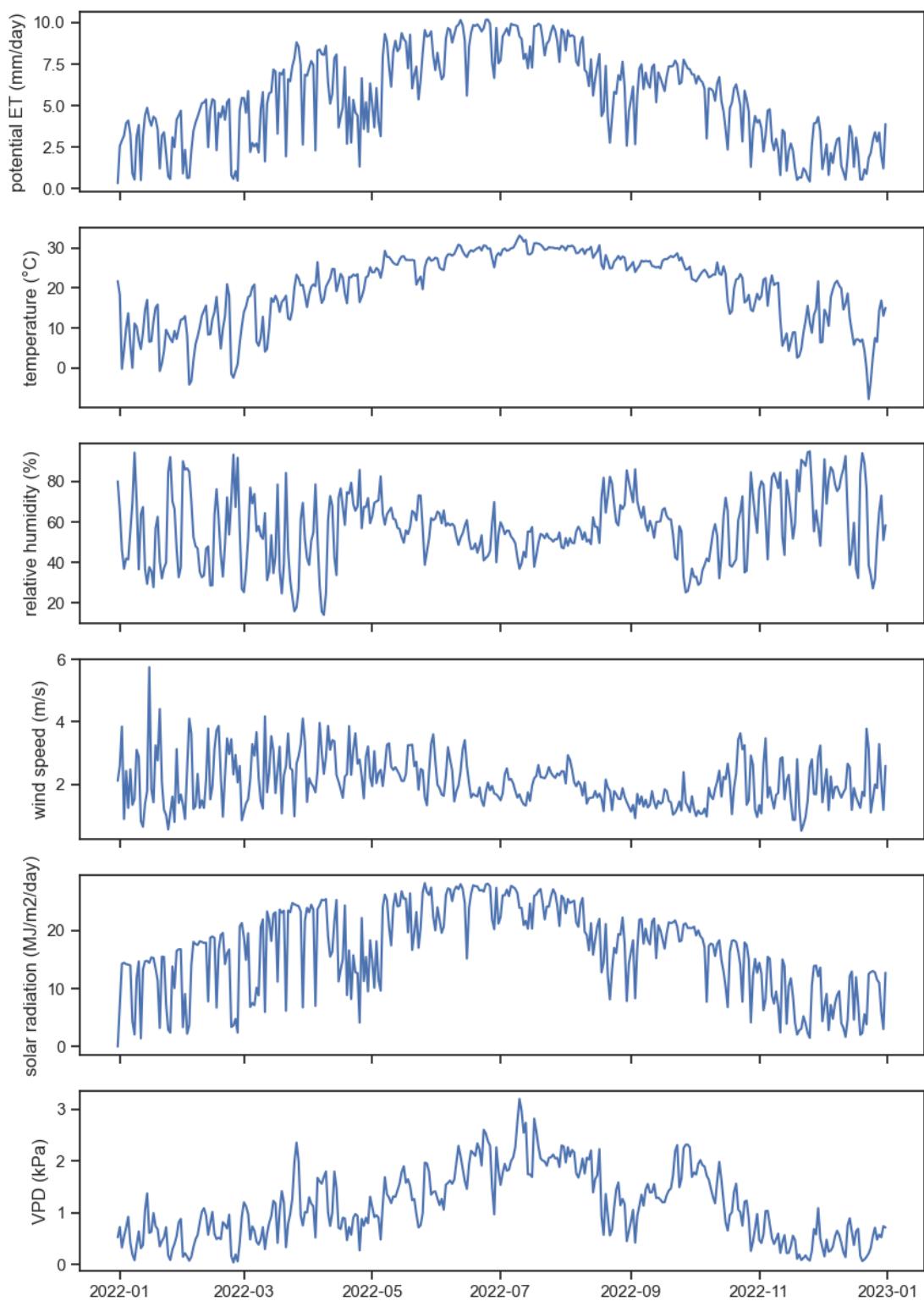
```
Text(0, 0.5, 'ET (mm/day)')
```



```
es = pyet.meteo_utils.calc_es(tmean=df_TX["AIR_TEMPERATURE"])
ea = pyet.meteo_utils.calc_ea(tmean=df_TX["AIR_TEMPERATURE"],
                             rh=df_TX["RELATIVE_HUMIDITY"])
vpd = es - ea
```

```
sns.set_theme(style="ticks", font_scale=1.0)
fig, ax = plt.subplots(6,1, figsize=(10,15), sharex=True)
ax[0].plot(penm2); ax[0].set_ylabel("potential ET (mm/day)")
ax[1].plot(df_TX['AIR_TEMPERATURE']); ax[1].set_ylabel("temperature (°C)")
ax[2].plot(df_TX['RELATIVE_HUMIDITY']); ax[2].set_ylabel("relative humidity (%)")
ax[3].plot(df_TX['WIND_1_5']); ax[3].set_ylabel("wind speed (m/s)")
ax[4].plot(df_TX['SOLAR_RADIATION']); ax[4].set_ylabel("solar radiation (MJ/m2/day)")
ax[5].plot(vpd); ax[5].set_ylabel("VPD (kPa)")
```

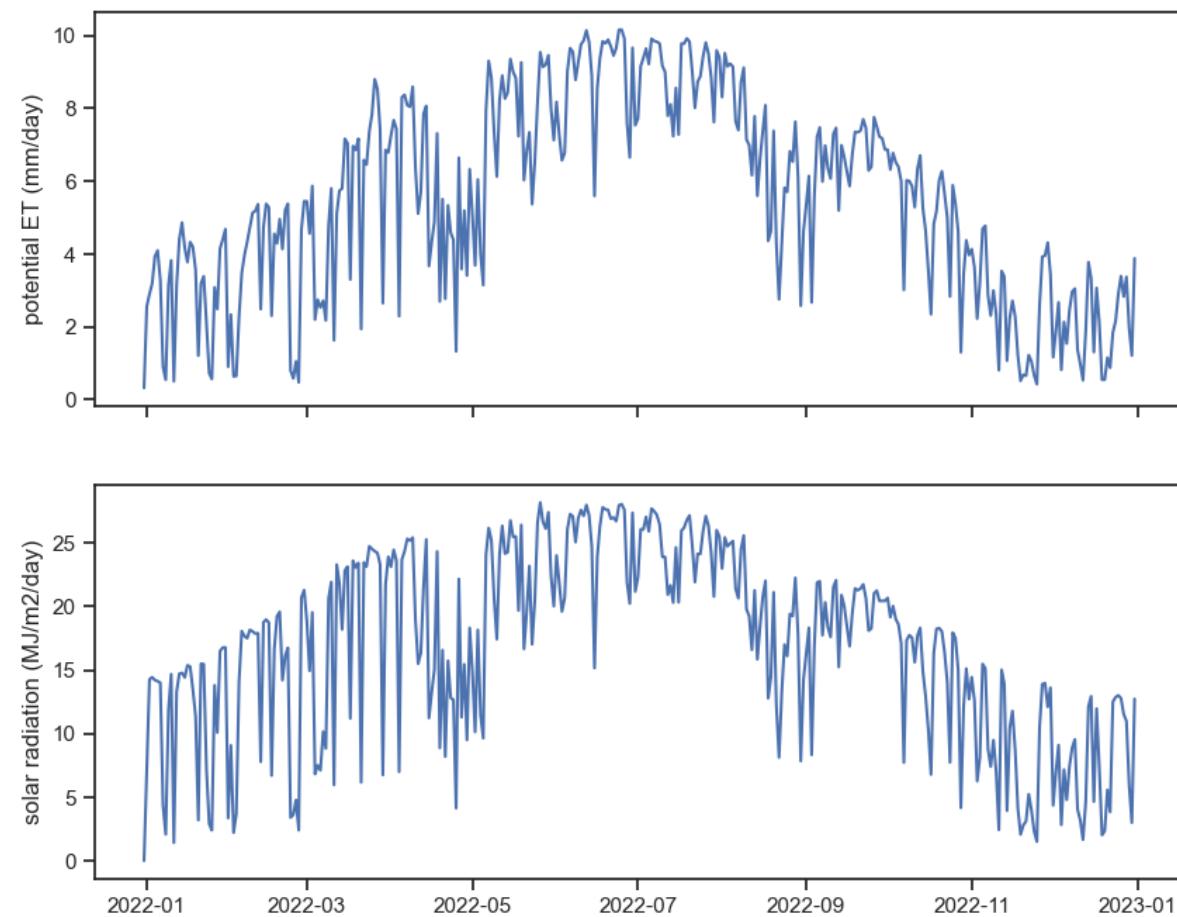
```
Text(0, 0.5, 'VPD (kPa)')
```



Can you see some patterns from the graphs above? Let's focus on the relationship between ET and solar radiation.

```
sns.set_theme(style="ticks", font_scale=1.0)
fig, ax = plt.subplots(2,1, figsize=(10,8), sharex=True)
ax[0].plot(penm2); ax[0].set_ylabel("potential ET (mm/day)")
ax[1].plot(df_TX['SOLAR_RADIATION']); ax[1].set_ylabel("solar radiation (MJ/m2/day)")
```

```
Text(0, 0.5, 'solar radiation (MJ/m2/day)')
```



Part IV

Infiltration

Here are some of the files we'll use in this module, in case you can't download them from their original repositories.

- [input rate 78 mm/h, 16% slope](#)
- [input rate 156 mm/h, 16% slope](#)
- [input rate 234 mm/h, 16% slope](#)
- [input rate 312 mm/h, 16% slope](#)

12 Infiltration

12.1 Definitions

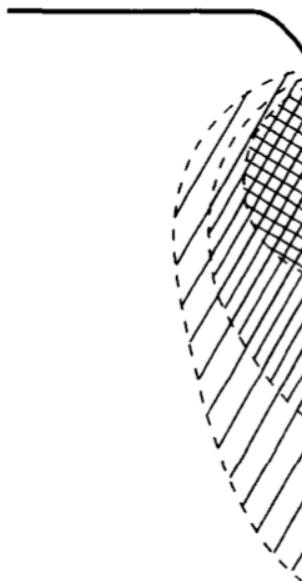
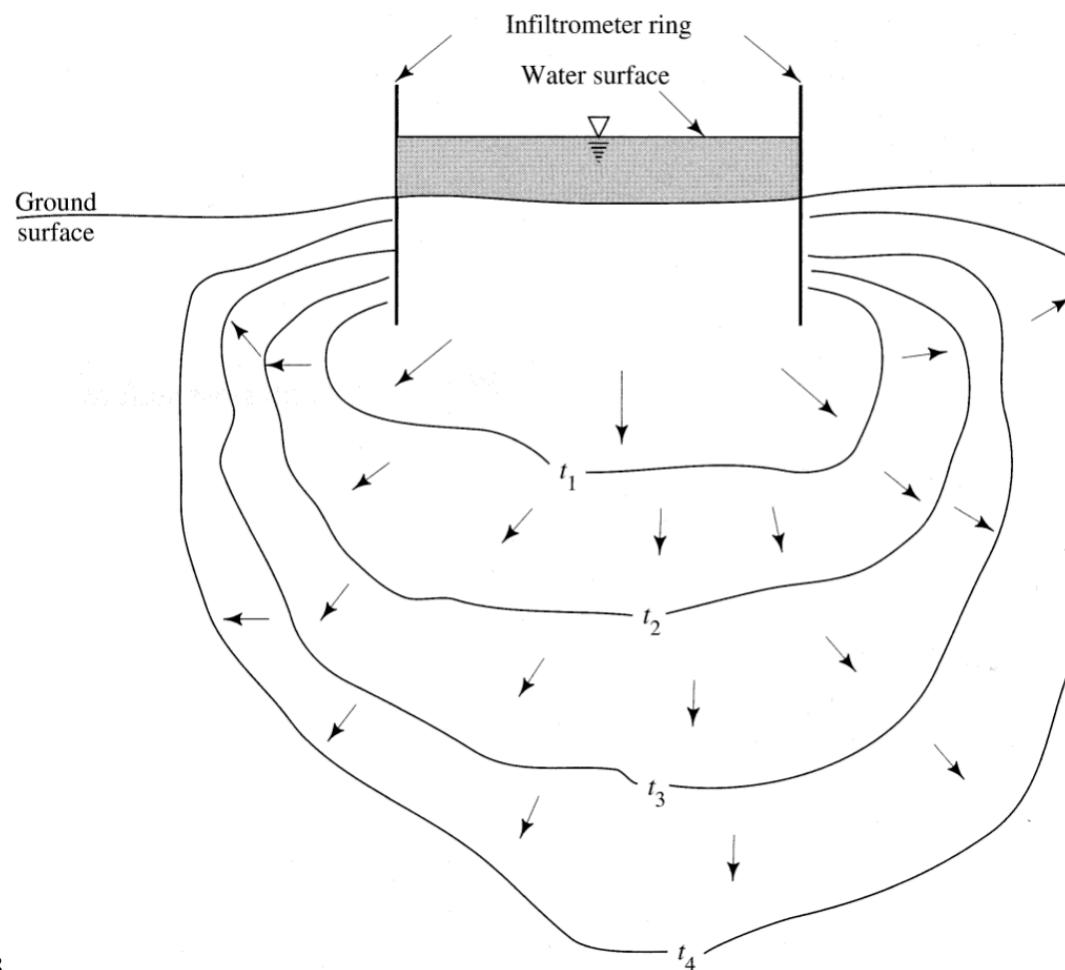


Fig. 14.6. Infiltration from an irrigation source shown after different periods of time ($t_1 < t_2 < t_3$). As infiltration continues, infiltration becomes more uniform in all directions, gradients diminish and the gravitational gradients

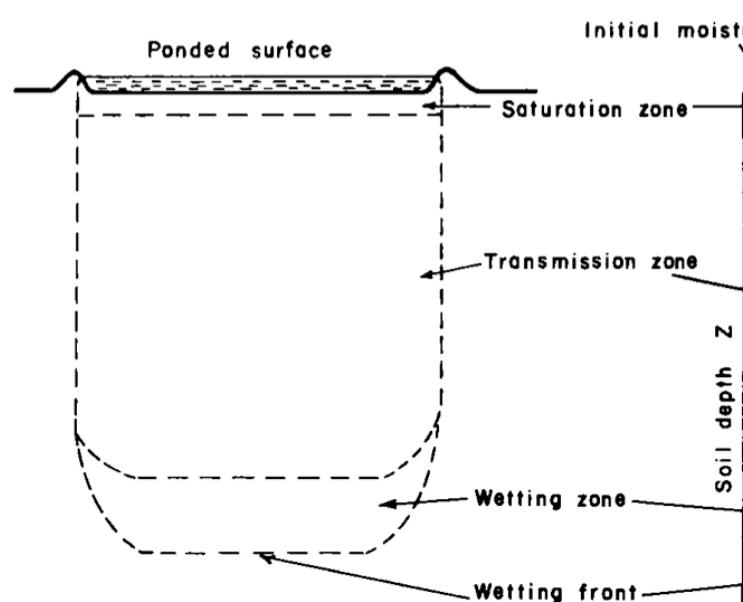
Hillel (2003), Introduction to Environmental Soil Physics, figure 14.6



Dingman (2015), figure 8.13



Dingman (2015), figure 8.14

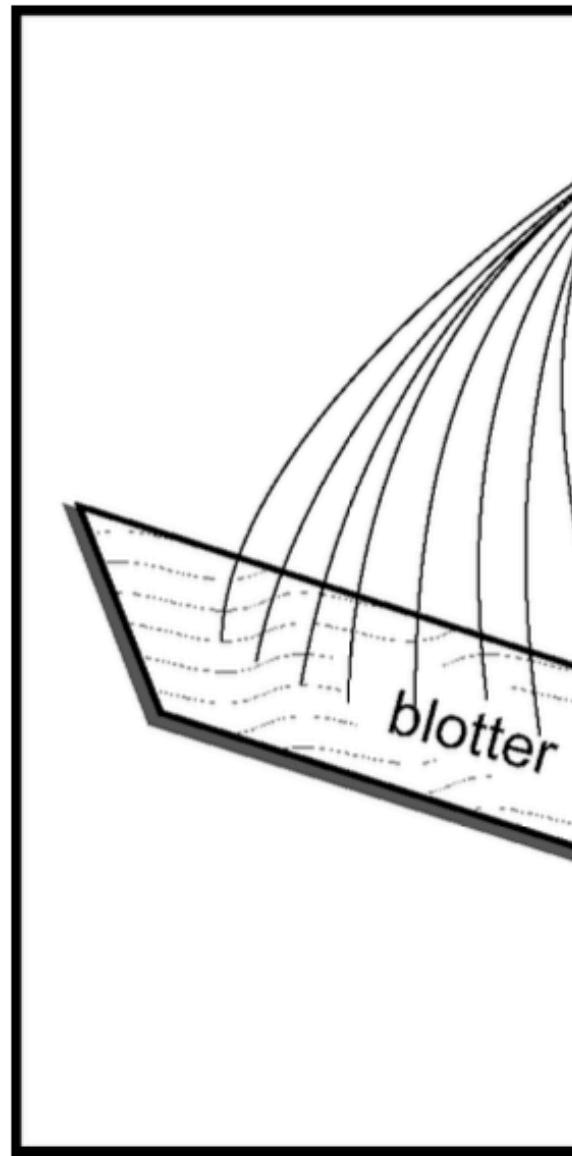


Hillel (2003), Introduction to Soil Physics, figure 12.3

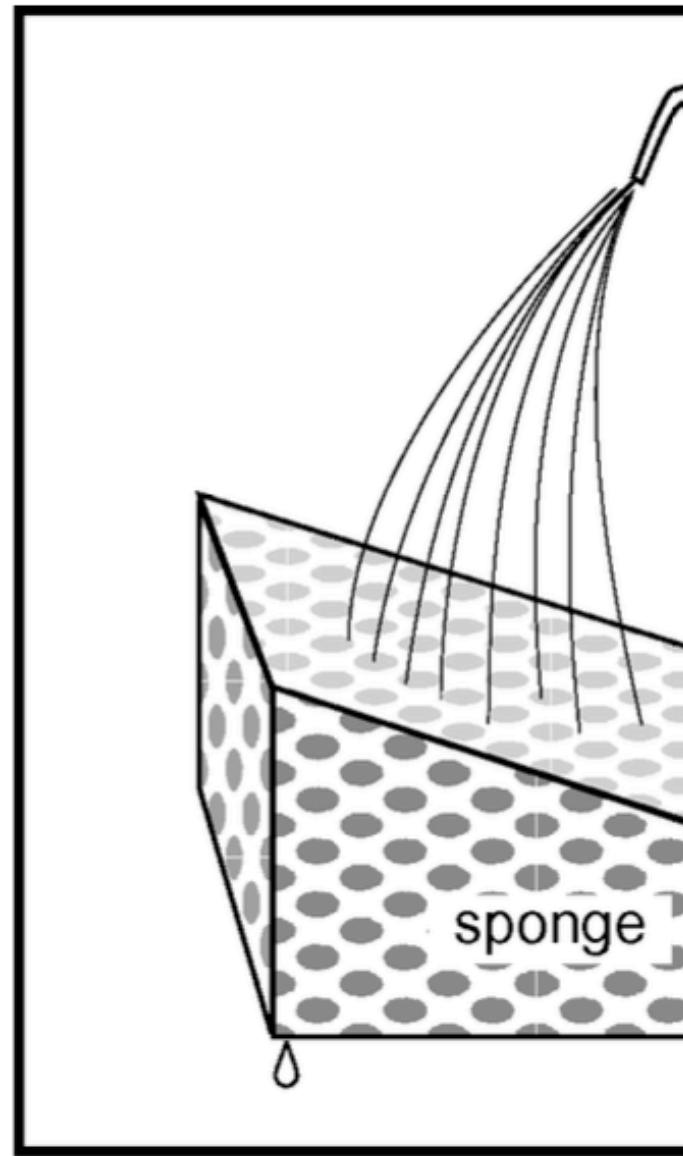
Source: Dingman (2015), page 355

- The **water-input rate**, $w(t)$ [L T^{-1}], is the rate at which water arrives at the surface due to rain, snowmelt, or irrigation. A water-input event begins at time $t = 0$ and ends at $t = T_w$.
- The **infiltration rate**, $f(t)$ [L T^{-1}], is the rate at which water enters the soil from the surface.
- The **infiltrability**, also called **infiltration capacity**, $f^*(t)$ [L T^{-1}], is the maximum rate at which infiltration could occur at any time; note that this changes during the infiltration event.
- The **depth of ponding**, $H(t)$ [L], is the depth of water standing on the surface.

Source: Ward and Trimble (2003), page 63, 64

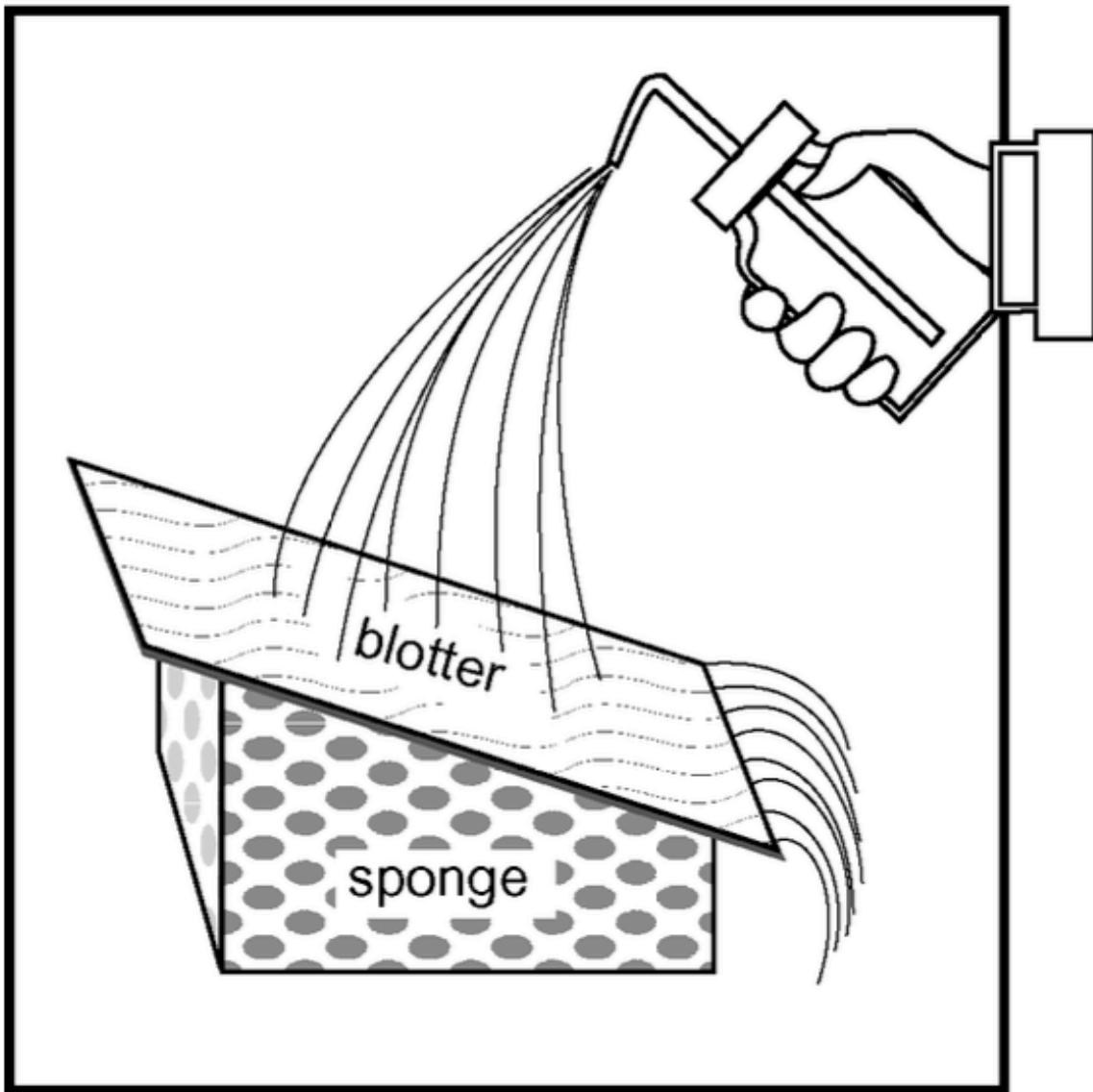


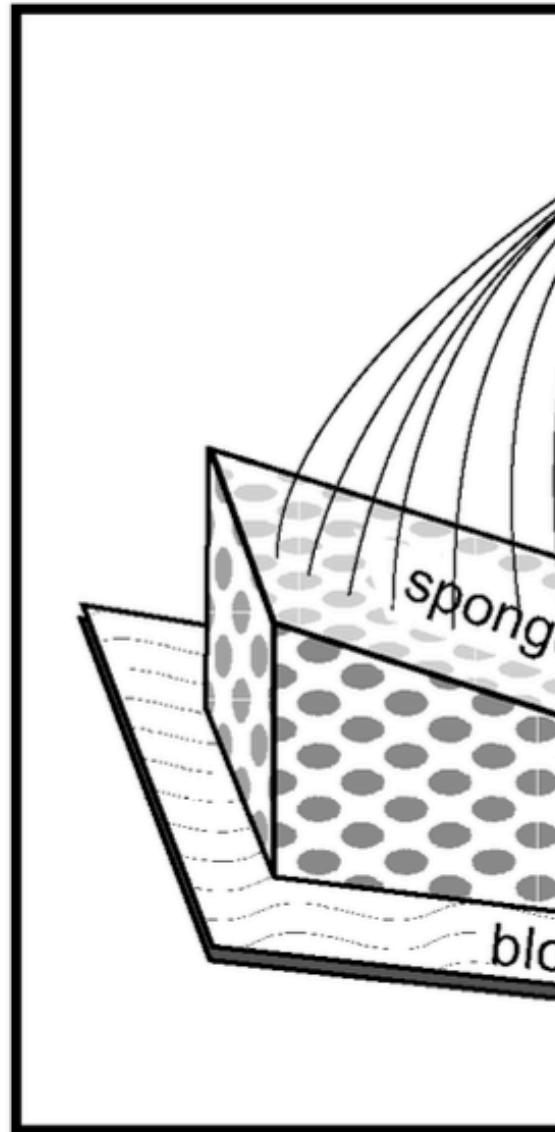
Infiltration capacity of absorbent paper is low, there is much runoff



Infiltration capacity of sponge is high, there is little runoff

Infiltration capacity of the sponge is limited by the overlying layer with low permeability





Infiltration capacity of the sponge is limited by the underlying layer

<https://youtu.be/ego2FkuQwxc>

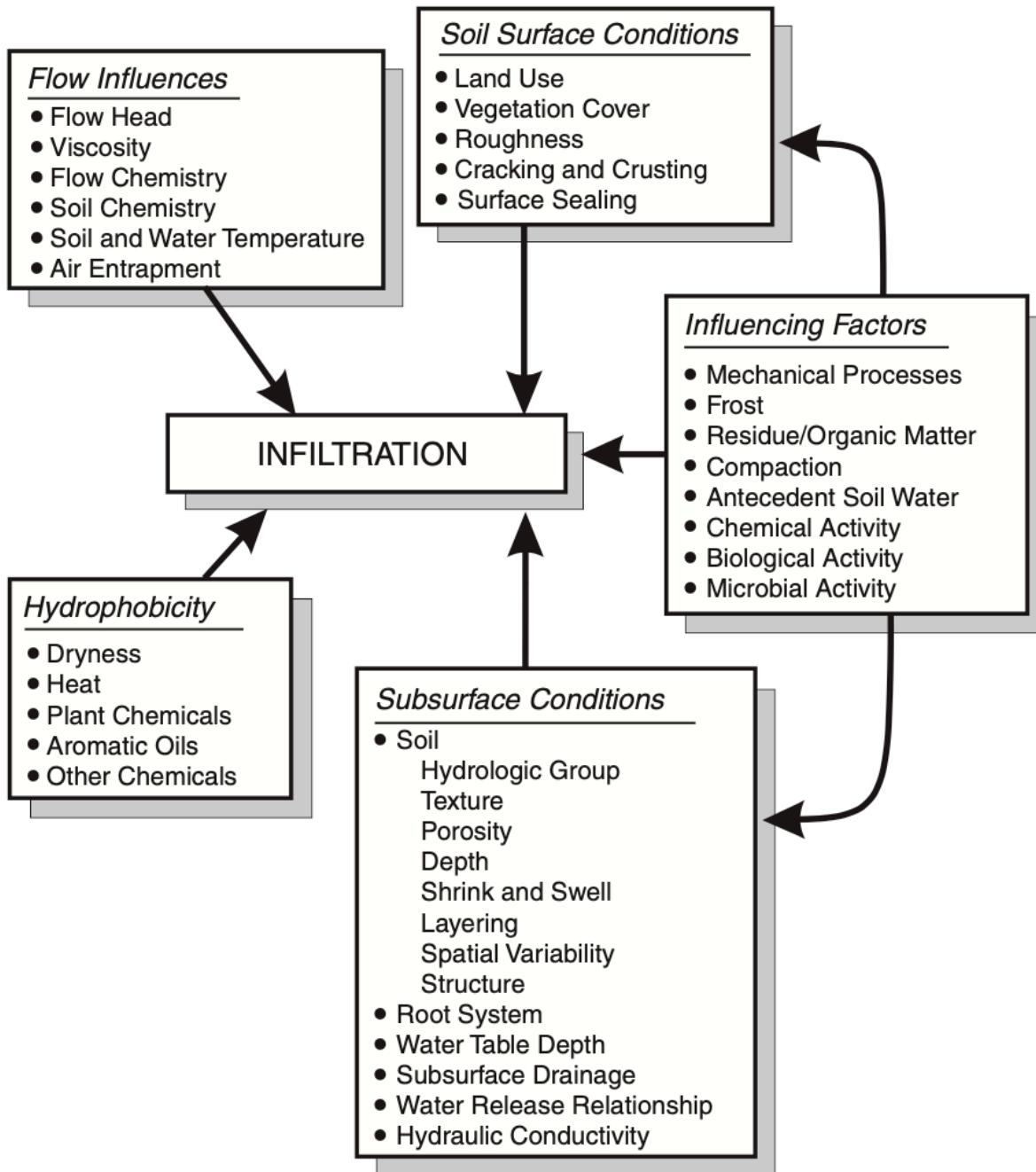


Figure 12.1: Ward and Trimble (2003), page 65

12.2 Darcy

Darcy's equation for vertical flow

$$q = -K \frac{\partial H_{\text{total}}}{\partial z}$$

where the total head $H_{\text{total}} = -H_{\text{suction}} - z_{\text{depth}}$, and

- H_{suction} is the suction head (negative pressure head)
- z_{depth} is the depth, points *downward*.

Substituting:

$$q = K \frac{\partial H_{\text{suction}}}{\partial z} + K$$

Substituting the above into the continuity equation

$$\frac{\partial \theta}{\partial t} = \frac{\partial q}{\partial z}$$

yields the Richards equation.

12.3 Richards

Richards equation:

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z} \left[K(\theta) \frac{\partial H_{\text{total}}}{\partial z} \right]$$

Substituting $H_{\text{total}} = -H_{\text{suction}} - z_{\text{depth}}$ yields:

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z} \left[K(\theta) \left(\frac{\partial (-H_{\text{suction}} - z)}{\partial z} \right) \right]$$

$$\frac{\partial \theta}{\partial t} = - \underbrace{\frac{\partial}{\partial z} \left(K(\theta) \frac{\partial H_{\text{suction}}}{\partial z} \right)}_{\text{matric}} - \underbrace{\frac{\partial K(\theta)}{\partial z}}_{\text{gravitational}}$$

12.3.1 short times

As the water starts to enter the relatively dry soil, the pressure differences in the water at the surface and in the soil are quite large and, as a result, the second term on the right is practically negligible compared to the first one.

$$\frac{\partial \theta}{\partial t} = -\frac{\partial}{\partial z} \left(K(\theta) \frac{\partial H}{\partial z} \right)$$

12.3.2 long times

As illustrated in the figure below (Davidson et al., 1963), after longer times of infiltration, the water content profile near the surface gradually becomes more uniform and it eventually assumes the satiation value, or $\theta \rightarrow \theta_0$; similarly, the pressure in the upper layers of the soil becomes gradually atmospheric, or $H \rightarrow 0$. Hence, their vertical gradients

$$\frac{\partial \theta}{\partial z} \text{ and } \frac{\partial H_{\text{suction}}}{\partial z} \rightarrow 0$$

From Darcy's equation we have that

$$q = K(\theta_0) = K_{\text{sat}}$$

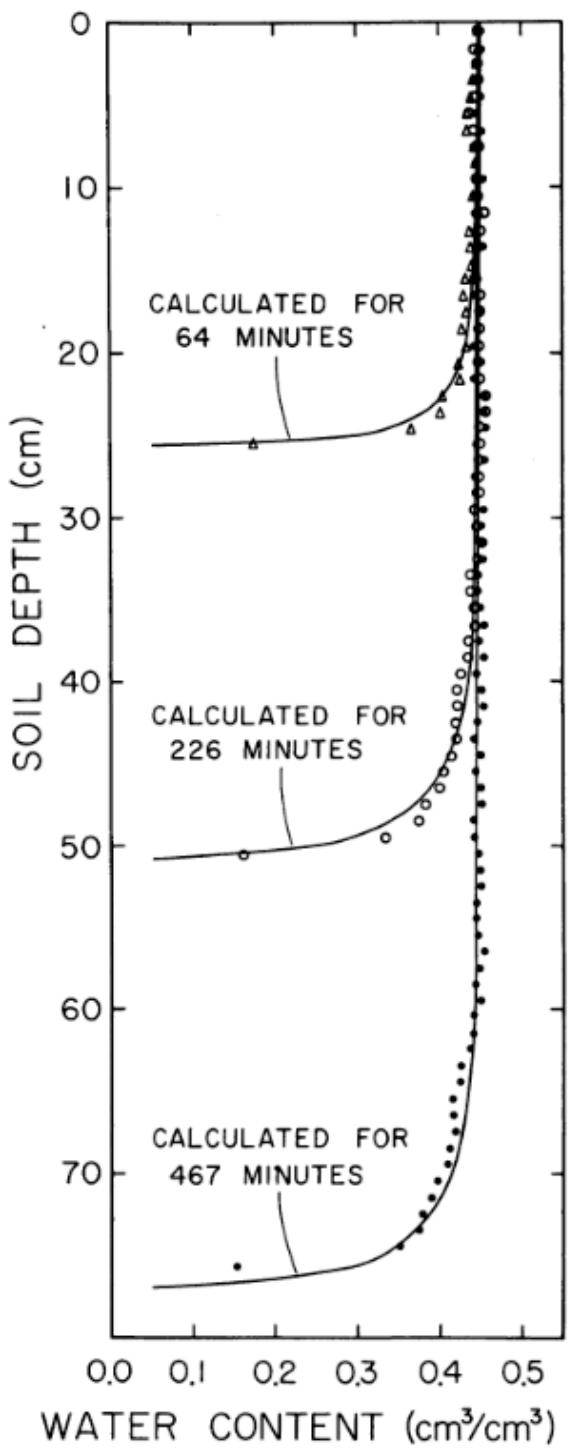
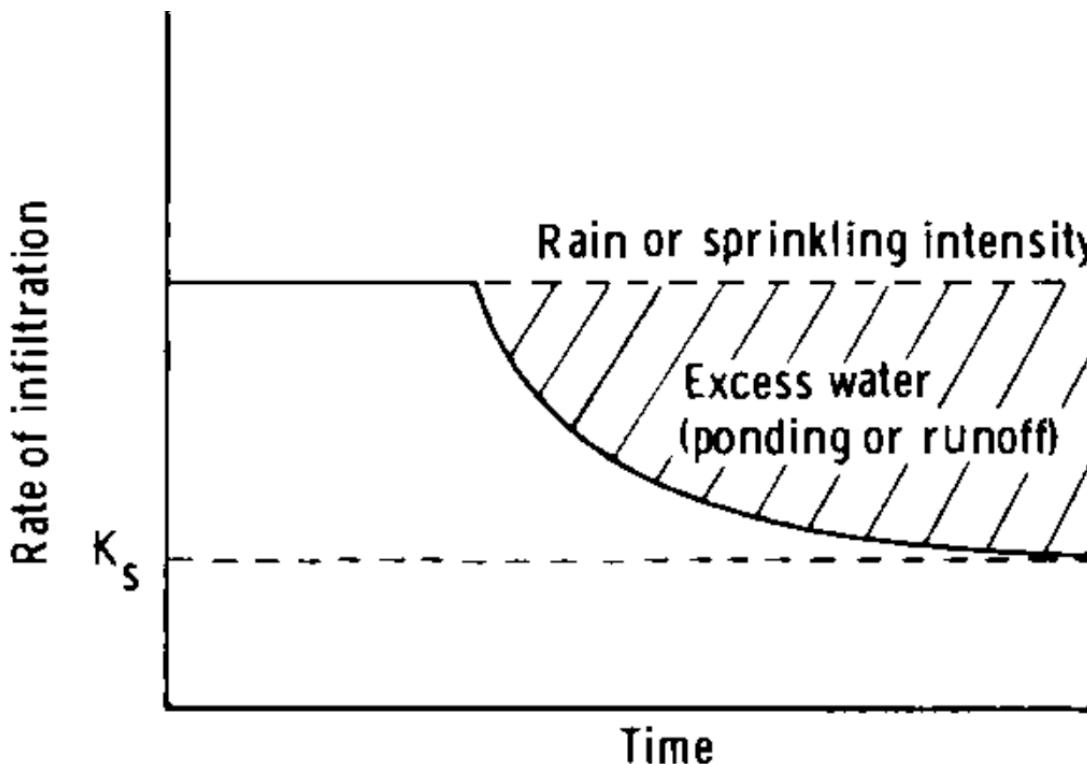


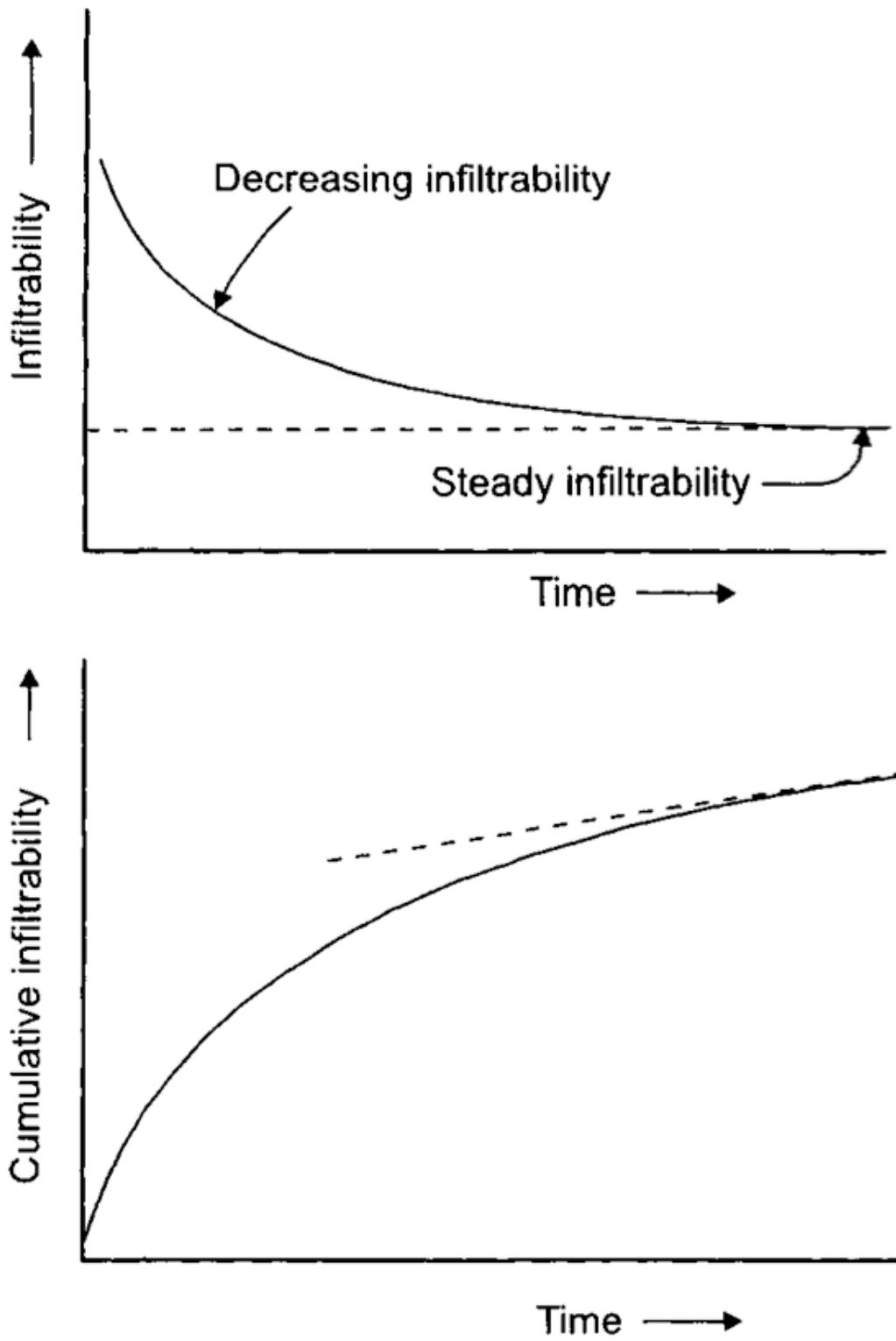
Fig. 19. Calculated and measured soil water profiles for air-dry Columbia soil allowed to wet at $\theta_e = 0.45 \text{ cm}^3/\text{cm}^3$.

12.3.3 Rainfall infiltration

Infiltration rate is equal to rainfall rate, at least at first. If rainfall rate w is lower than K_{sat} , than everything enters the soil, i.e., $f = K_{\text{sat}}$. However, if $w > K_{\text{sat}}$, water content θ will increase at the surface, until it reaches θ_0 , and at that moment, called ponding time t_p , water will begin to accumulate at the surface.



Hillel (2003), figure 12.1



Hillel (2003), figure 12.2

12.4 Horton equation

One of the most widely used models, developed by R.E. Horton (1939), considered to be the father of modern hydrology.

$$f = f_c + (f_0 - f_c)e^{-\beta t}$$

- f : infiltration rate
- f_c : infiltration capacity at large t
- f_0 : initial infiltration capacity
- β : best fit empirical parameter

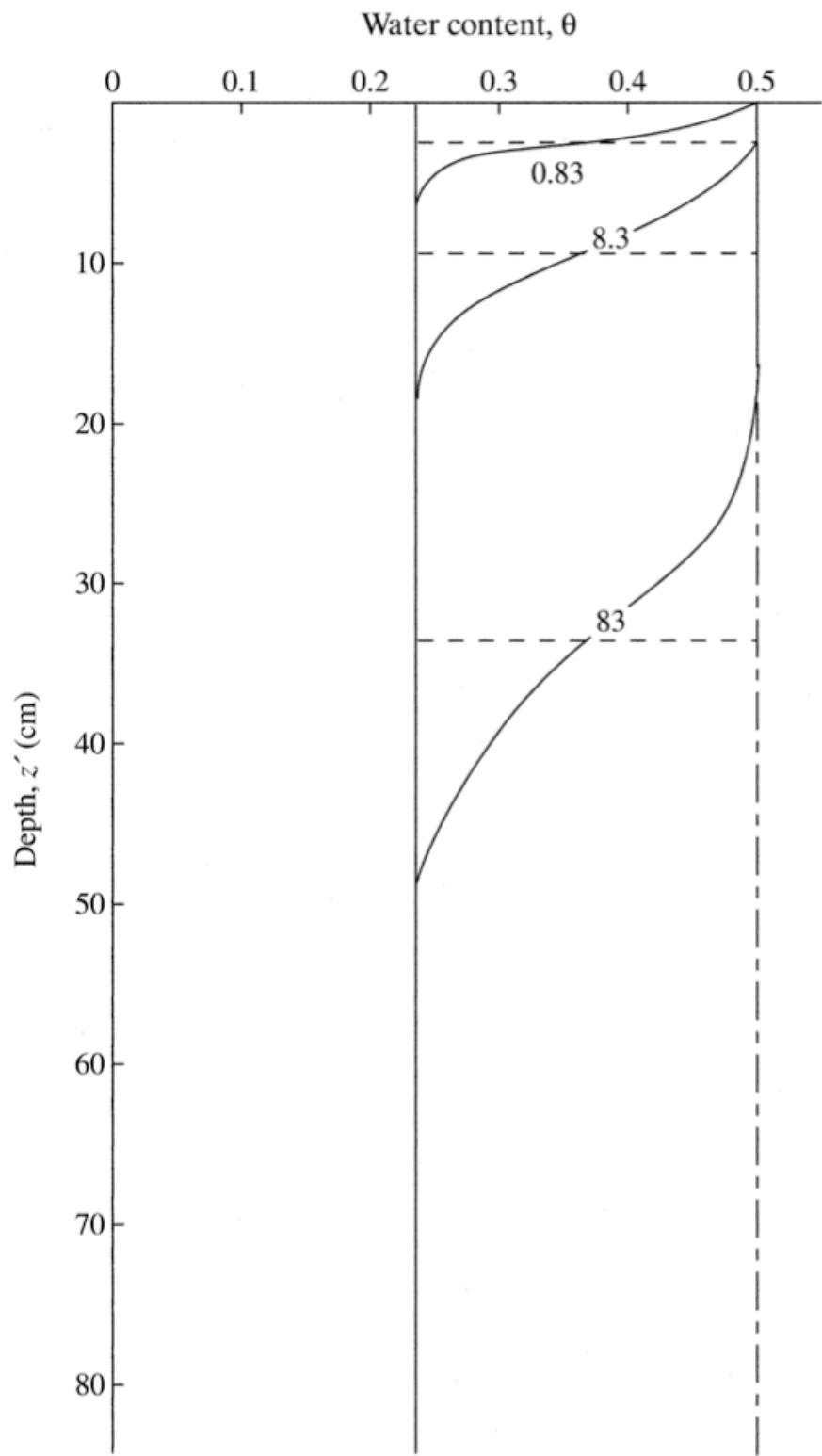
Advantages

- Simple equation
- Usually gives good fit to measured data because it is dependent on three parameters

Disadvantages

- This method has no physical significance, it is not based on any water transport mechanism
- Does not describe infiltration prior to ponding

12.5 Green & Ampt



Dingman (2015), figure 8.11

Assumptions:

- homogeneous soil, infinite depth (no water table)
- horizontal surface
- constant water head equal to zero is maintained at the surface
- uniform water content prior to wetting, $\theta(t = 0, z) = \theta_0$
- moving front is characterized by a constant matric suction, ψ_f

Source: Dingman (2015), page 370

This equation was developed under the scenario of constant rainfall or irrigation on an initially dry soil as a sharp wetting front (such as piston flow). Water penetrates a dry soil with a certain initial moisture content, and wets the layer to a saturated moisture content as it traverses deeper. The connection between soil moisture and infiltration rate is modeled in the Green-Ampt equation:

$$f(t) = K_{\text{sat}} \left[1 + \frac{|\psi_f| \cdot (\phi - \theta_0)}{F(t)} \right]$$

- $f(t)$: infiltration rate
- $F(t)$: cumulative infiltration rate, $F = \int f \, dt$
- ψ_f : effective wetting-front suction
- ϕ : soil porosity
- θ_0 : initial soil water content

The same equation can be simply be rewritten as

$$f = \frac{A}{F} + B$$

where

- $A = K_{\text{sat}} \cdot |\psi_f| \cdot (\phi - \theta_0)$
- $B = K_{\text{sat}}$

The porosity ϕ and the saturated hydraulic conductivity K_{sat} can be estimated from the soil texture. The wetting-front suction ψ_f can be estimated using the Brooks-Corey parameters:

$$|\psi_f| = \frac{2b+3}{2b+6} \cdot |\psi_{ae}|,$$

where ψ_{ae} is the air-entry pressure head. Values for the parameters above can be found in this table:

Table 7.4 Brooks–Corey and Campbell Parameters (Table 7.2) for Various Soil Textures Based on Analysis of 1845 Soils.^a

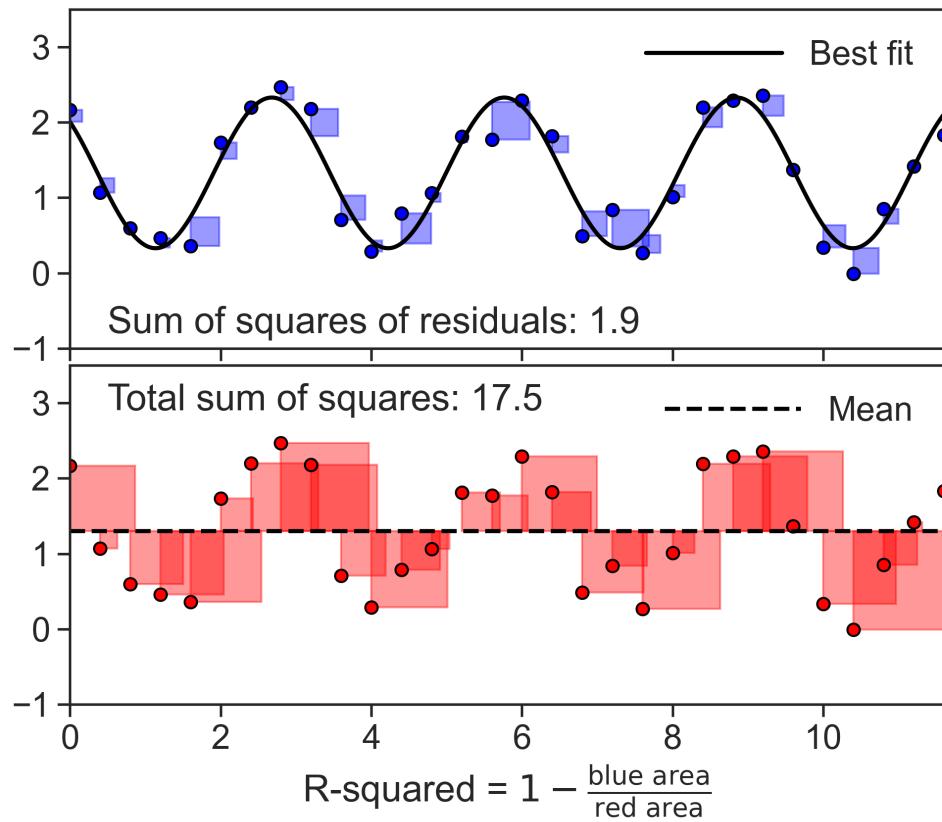
| Soil Texture | ϕ | K_h (cm/s) | $ \psi_{ae} $ (cm) | b |
|---------------------|---------------|-----------------------|--------------------|-------------|
| Sand | 0.395 (0.056) | 1.76×10^{-2} | 12.1 (14.3) | 4.05 (1.78) |
| Loamy sand | 0.410 (0.068) | 1.56×10^{-2} | 9.0 (12.4) | 4.38 (1.47) |
| Sandy loam | 0.435 (0.086) | 3.47×10^{-3} | 21.8 (31.0) | 4.90 (1.75) |
| Silt loam | 0.485 (0.059) | 7.20×10^{-4} | 78.6 (51.2) | 5.30 (1.96) |
| Loam | 0.451 (0.078) | 6.95×10^{-4} | 47.8 (51.2) | 5.39 (1.87) |
| Sandy clay loam | 0.420 (0.059) | 6.30×10^{-4} | 29.9 (37.8) | 7.12 (2.43) |
| Silty clay loam | 0.477 (0.057) | 1.70×10^{-4} | 35.6 (37.8) | 7.75 (2.77) |
| Clay loam | 0.476 (0.053) | 2.45×10^{-4} | 63.0 (51.0) | 8.52 (3.44) |
| Sandy clay | 0.426 (0.057) | 2.17×10^{-4} | 15.3 (17.3) | 10.4 (1.64) |
| Silty clay | 0.492 (0.064) | 1.03×10^{-4} | 49.0 (62.1) | 10.4 (4.45) |
| Clay | 0.482 (0.050) | 1.28×10^{-4} | 40.5 (39.7) | 11.4 (3.70) |

^aValues in parentheses are standard deviations.

Source: Data from Clapp and Hornberger (1978).

12.6 Best Fit, Least Squares Method

Data: $f(x) = p_0 + \cos(p_1x + p_2) + \text{ noise}$



13 Exercises

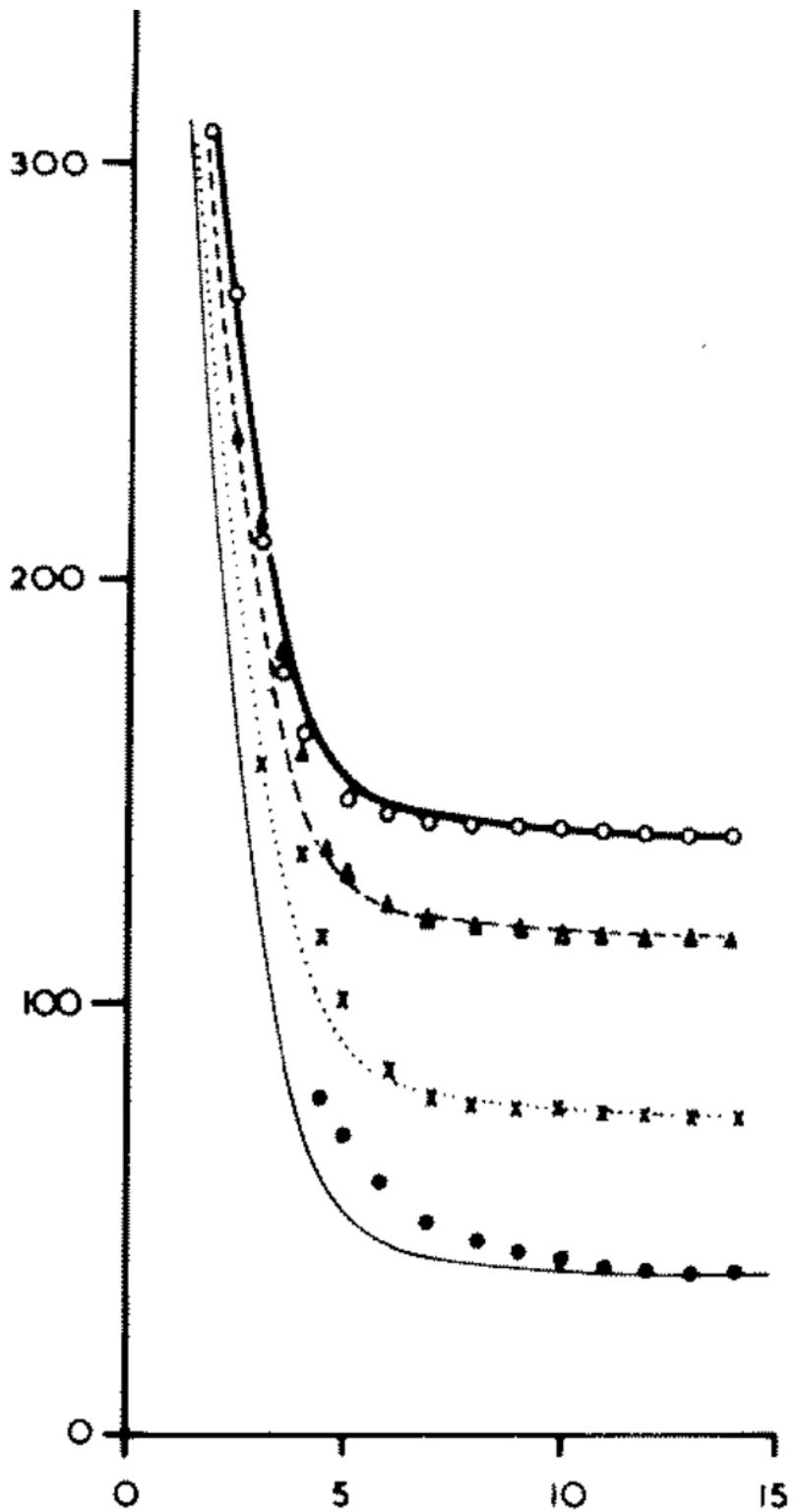
13.1 Tasks

1. Download the paper
Nassif, S. H., and E. M. Wilson, 1975, "The influence of slope and rain intensity on runoff and infiltration", Hydrological Sciences Journal.
2. Google the following: `plot digitizer`
3. Load image "nassif-16percent-slope.png" (see below)
4. Digitize data points from left to right. Create 4 csv files, one for each data set. Call them whatever you want.

Legend:

```
* white circle = 312 mm/h,  
* triangle = 234 mm/h,  
* x = 156 mm/h,  
* black circle = 78 mm/h.
```

The image is the second panel of Fig. 8, from the paper you downloaded.



If for any reason you're having trouble digitizing the data from the graph, download here each csv file I have already prepared.

- input_rate_078mm_per_h_16percent_slope.csv
- input_rate_156mm_per_h_16percent_slope.csv
- input_rate_234mm_per_h_16percent_slope.csv
- input_rate_312mm_per_h_16percent_slope.csv

Import relevant packages

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.optimize import curve_fit
import matplotlib.patches as patches
```

Load all four files you created. Use numpy's function `loadtxt`. Make sure that the first point in each table corresponds to the appropriate rainfall rate. You can normalize the data if it is not.

```
d1 = np.loadtxt("input_rate_078mm_per_h_16percent_slope.csv", delimiter=',')
d2 = np.loadtxt("input_rate_156mm_per_h_16percent_slope.csv", delimiter=',')
d3 = np.loadtxt("input_rate_234mm_per_h_16percent_slope.csv", delimiter=',')
d4 = np.loadtxt("input_rate_312mm_per_h_16percent_slope.csv", delimiter=',')

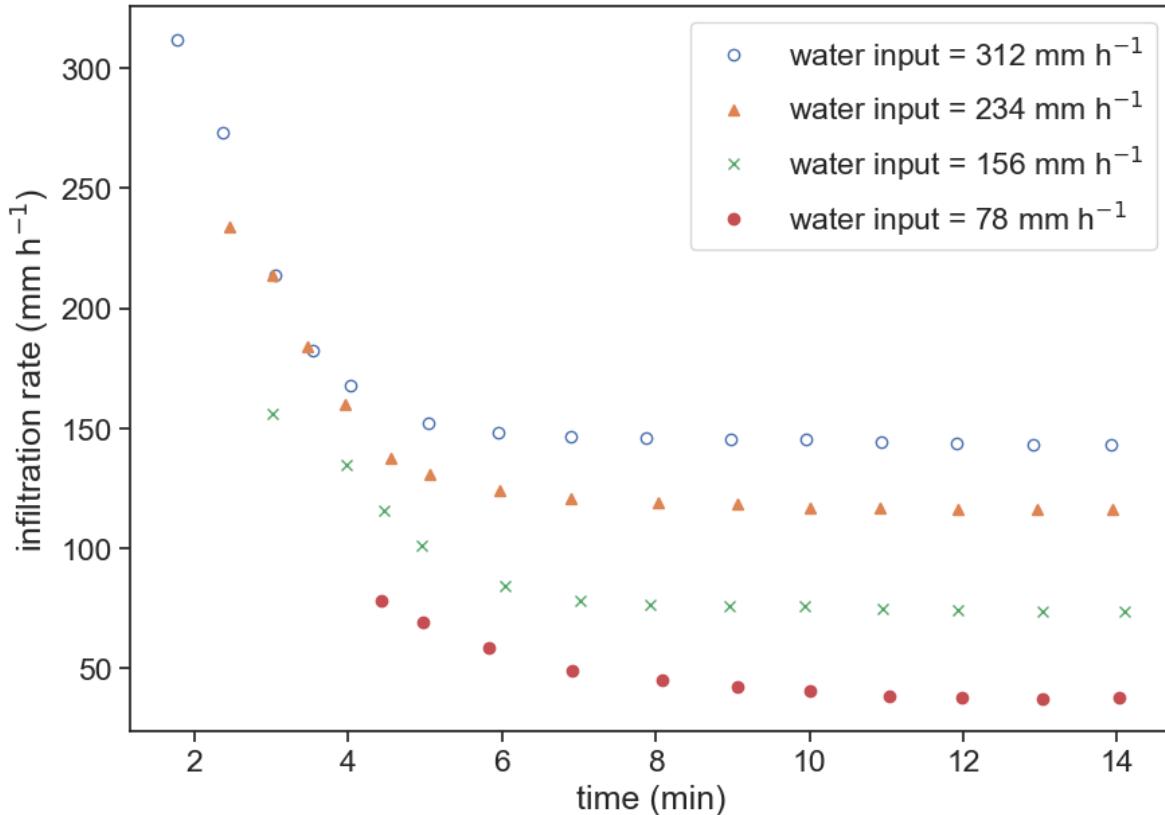
"""
In the digitization process, not all points are necessarily
located in the "correct" location. Because we know the initial
infiltration rate for each data series, we can correct that and
rescale the data so that the first point is exactly where we
expect it to be
"""
d1[:,1] = d1[:,1] * 78 / d1[:,1].max()
d2[:,1] = d2[:,1] * 156 / d2[:,1].max()
d3[:,1] = d3[:,1] * 234 / d3[:,1].max()
d4[:,1] = d4[:,1] * 312 / d4[:,1].max()
```

Reproduce the original figure, make it look good, something like this:

```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(d4[:,0], d4[:,1], 'o', markerfacecolor="None", label=r"water input = 312 mm h$^{-1}$")
ax.plot(d3[:,0], d3[:,1], '^', label=r"water input = 234 mm h$^{-1}$")
ax.plot(d2[:,0], d2[:,1], 'x', label=r"water input = 156 mm h$^{-1}$")
ax.plot(d1[:,0], d1[:,1], 'o', label=r"water input = 78 mm h$^{-1}$")
ax.set(xlabel="time (min)",
       ylabel=r"infiltration rate (mm h$^{-1}$)")
ax.legend(loc="upper right");

```



13.2 Horton's equation

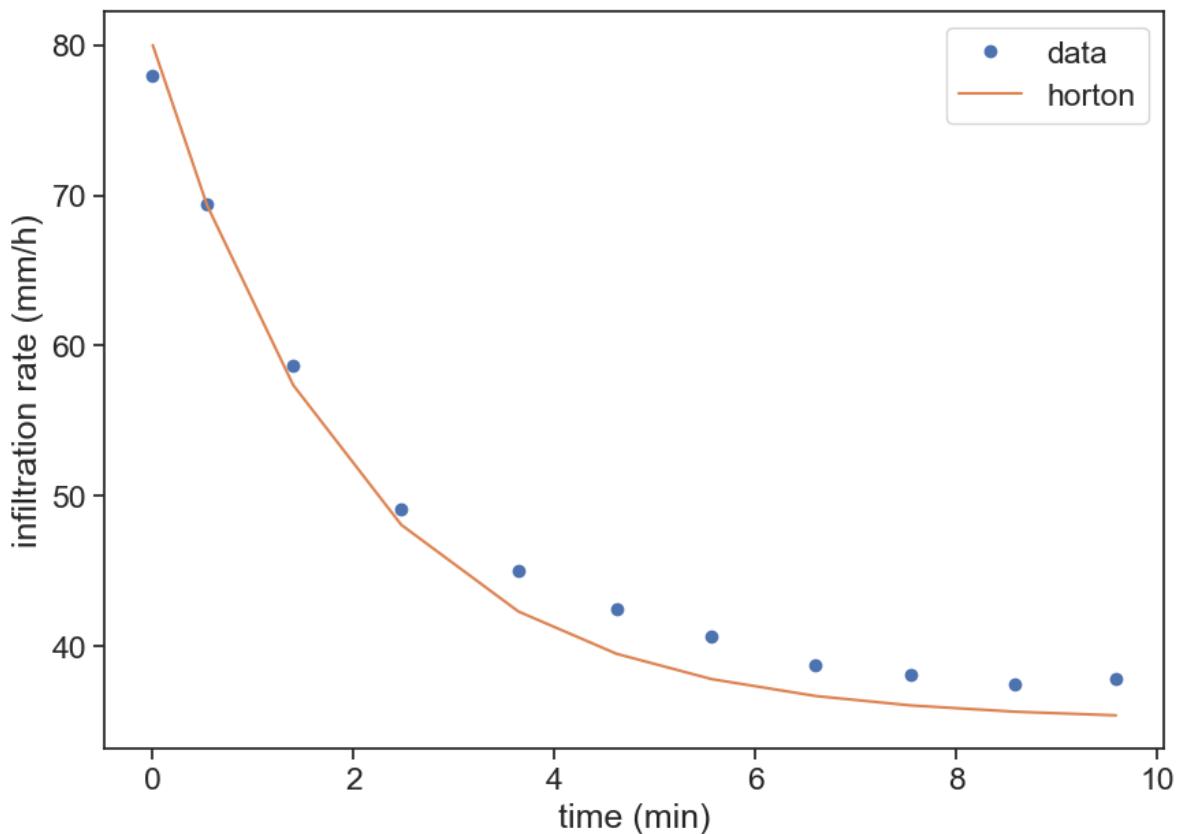
$$f = f_c + (f_0 - f_c)e^{-\beta t}$$

- f : infiltration rate
- f_c : infiltration capacity at large t
- f_0 : initial infiltration capacity
- β : best fit empirical parameter

Write a function called `horton`, that receives time t and the three parameters, and returns the right-hand side of the equation above. Plot one of the data sets, together with a guess of the parameters that should **roughly** fit the data.

```
def horton(t, fc, f0, beta):
    return fc + (f0 - fc) * np.exp(-beta*t)

fig, ax = plt.subplots(figsize=(10,7))
t = d1[:,0]
t = t - t[0]
f = d1[:,1]
ax.plot(t, f, 'o', label="data")
param_guess = [35, 80, 0.5]
ax.plot(t, horton(t, *param_guess), '--', label="horton")
ax.set(xlabel="time (min)",
       ylabel="infiltration rate (mm/h)")
ax.legend(loc="upper right");
```



Find the best fit for the parameters f_c, f_0, β . Calculate the R^2 for each data set.

For the best fit, use scipy's `curve_fit`. Write a function to compute the R-squared of your fit.

```

def best_fit(data):
    t = data[:,0]
    t = t - t[0]
    f = data[:,1]
    # best fit
    popt, pcov = curve_fit(f=horton,
                           xdata=t,
                           ydata=f,
                           p0=(130, 800, 0.5),
                           )
    return [popt, pcov]

def calculate_r_squared(data, popt):
    t = data[:,0]
    t = t - t[0]
    f = data[:,1]
    # Calculate residuals
    residuals = f - horton(t, *popt)
    # You can get the residual sum of squares (ss_res) with
    ss_res = np.sum(residuals**2)
    # You can get the total sum of squares (ss_tot) with
    ss_tot = np.sum((f - np.mean(f))**2)
    # And finally, the r_squared-value with,
    r_squared = 1 - (ss_res / ss_tot)
    return r_squared

def plot_best_fit(data, axis, marker, markercolor):
    # calculate best fit parameters
    popt, pcov = best_fit(data)
    t = data[:,0]
    f = data[:,1]
    # plot data points
    ax.plot(t, f, marker, markerfacecolor=markercolor, markeredgecolor="black")
    # plot best fit line
    r_squared = calculate_r_squared(data, popt)
    labeltext = r"$f_c=$ {:.2f}, $f_0=$ {:.2f}, $\beta=$ {:.2f}, $R^2=$ {:.2f}".format(popt[0],
    ax.plot(t, horton(t-t[0], *popt), color=markercolor, label=labeltext)

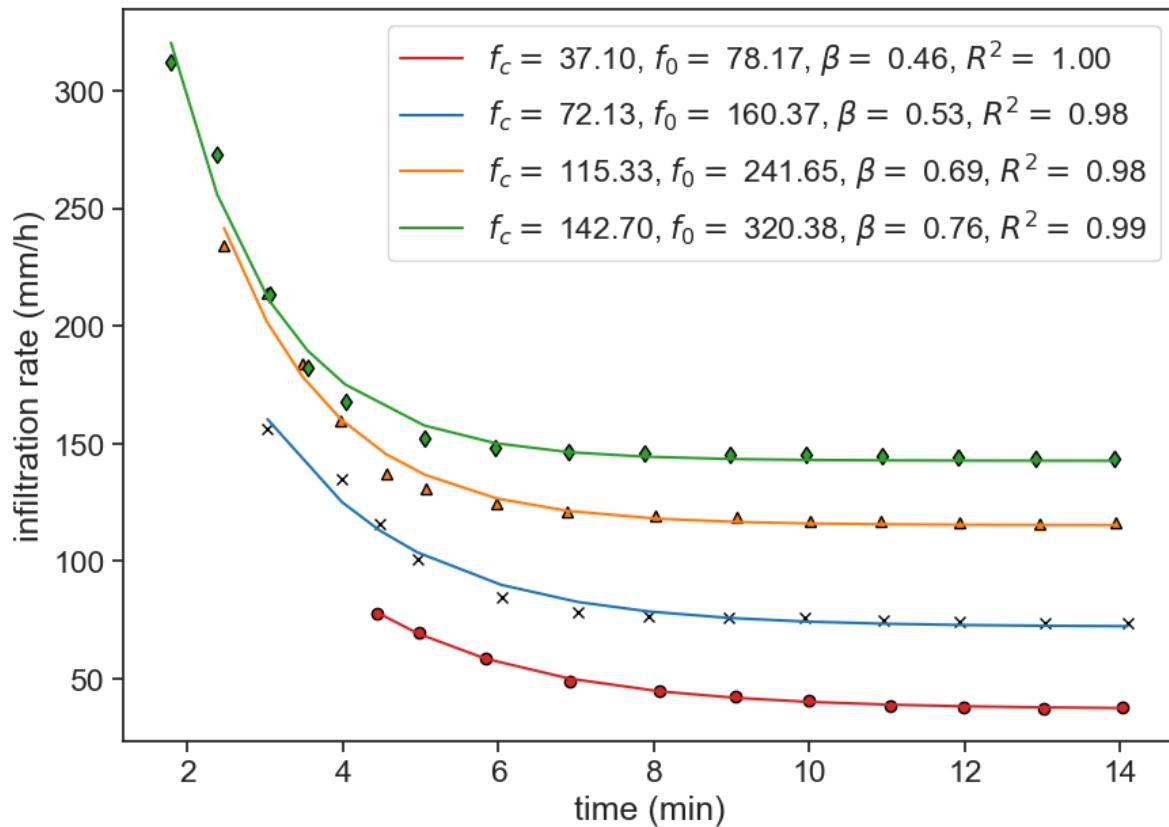
fig, ax = plt.subplots(figsize=(10,7))
plot_best_fit(d1, ax, 'o', "tab:red")

```

```

plot_best_fit(d2, ax, 'x', "tab:blue")
plot_best_fit(d3, ax, '^', "tab:orange")
plot_best_fit(d4, ax, 'd', "tab:green")
ax.set(xlabel="time (min)",
       ylabel="infiltration rate (mm/h)")
ax.legend();

```



Make a graph of the infiltration rate and of the runoff, as a function of time. Use any of the four data sets you have.

```

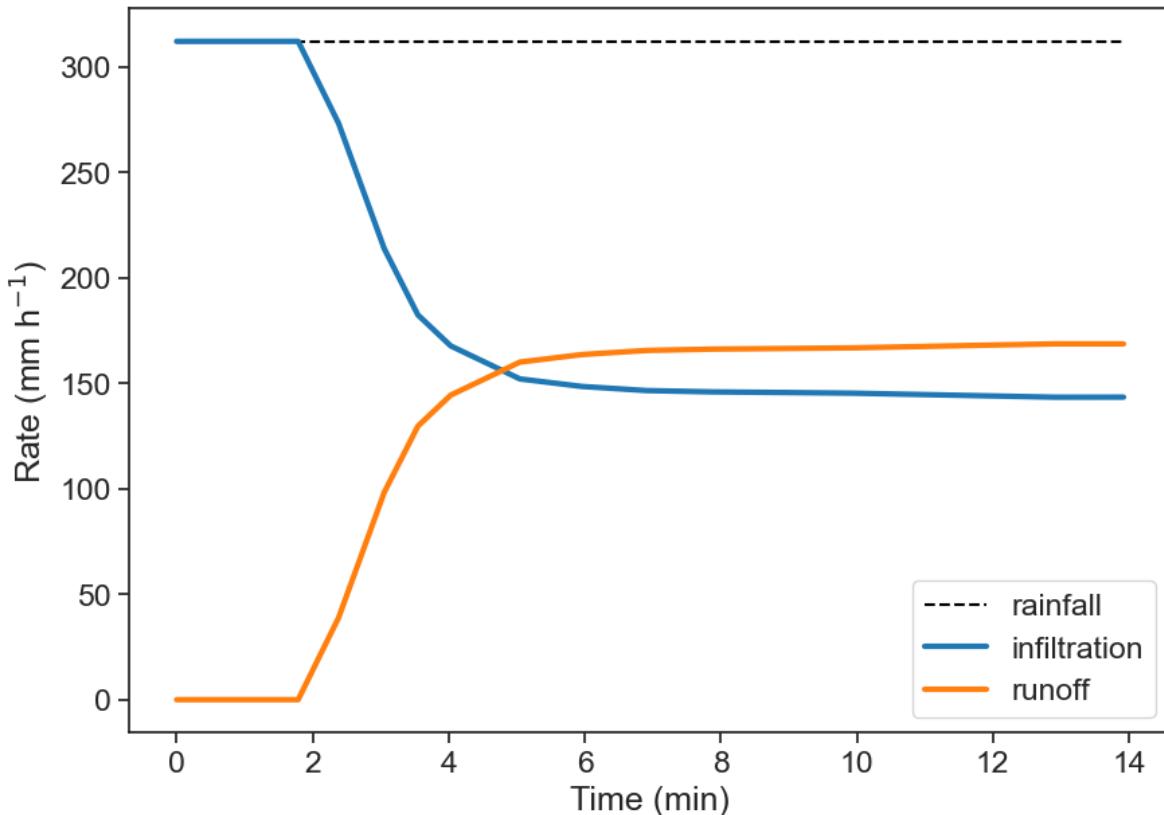
fig, ax = plt.subplots(figsize=(10,7))
data = d4
t = data[:, 0]
f = data[:, 1]
t = np.concatenate([ [0], t])
f = np.concatenate([ [f[0]], f])
runoff = f[0] - f
ax.plot(t, f*0 + f[0], ls="--", color="black", label="rainfall")

```

```

ax.plot(t, f, color="tab:blue", lw=3, label=r"infiltration")
ax.plot(t, runoff, color="tab:orange", lw=3, label=r"runoff")
ax.set(xlabel="Time (min)",
       ylabel=r"Rate (mm h$^{-1}$)")
ax.legend(loc="lower right");

```



13.3 What does fit really mean?

Let's take as an example data (x, y) that look to be organized in a linear trend.

```

N = 30
x_data = np.arange(N)
a = 2.0
b = 30.0
noise = 6 * (np.random.random(N)-0.5)
y_data = a * x_data + b + noise

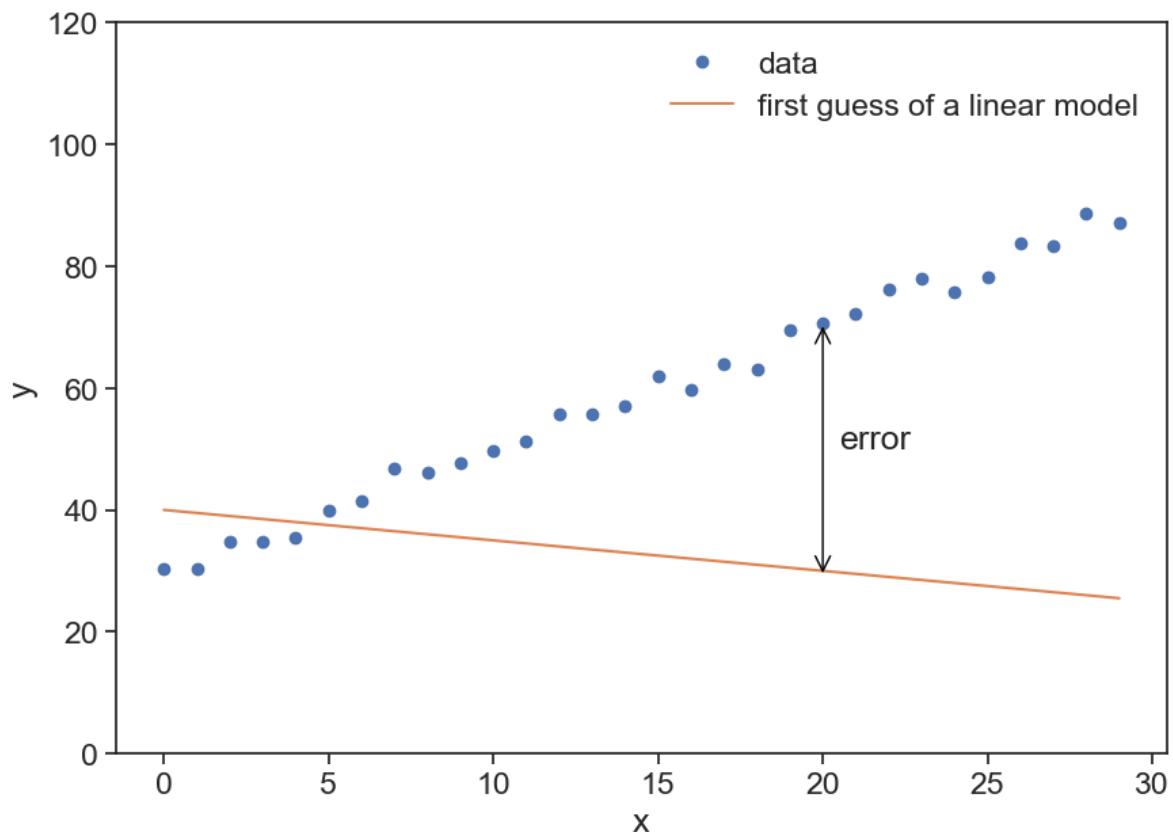
```

```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(x_data, y_data, 'o', label="data")
ax.plot(x_data, -0.5*x_data + 40, label="first guess of a linear model")

ax.annotate("", 
            xy=(x_data[20], y_data[20]), xycoords='data',
            xytext=(x_data[20], 29), textcoords='data',
            size=20,
            arrowprops=dict(arrowstyle="<->",
                            color="black",
                            connectionstyle="arc3"),
            )
ax.text(20.5, 50, "error")
ax.legend(frameon=False)
ax.set(xlabel="x",
       ylabel="y",
       ylim=[0,120]);

```



Our guess doesn't look good. We used the model

$$y = ax + b$$

where $a = -0.5$ and $b = 40$.

We will adjust these two parameters by minimizing the mean error between each data point and the model (straight line). We need to minimize the following expression:

$$\begin{aligned}MSE &= \text{mean}(\text{error}^2) \\MSE &= \frac{1}{N} \sum_{i=1}^N \text{error}^2 \\MSE &= \frac{1}{N} \sum_{i=1}^N (\text{measured} - \text{modelled})^2 \\MSE &= \frac{1}{N} \sum_{i=1}^N [y_i - (a \cdot x_i + b)]^2\end{aligned}$$

The MSE is a function of a and b . For every set of values (a, b) , MSE has value, so you can imagine a surface in the (a, b) plane.

```
def compute_mse(y_measured, y_predicted):
    MSE = np.mean(
        (y_measured - y_predicted)**2
    )
    return MSE

def linear_model(x, a, b):
    return a*x + b

def gradient_descent(x, y, a, b, learning_rate):
    # linear model
    y_predicted = a * x + b
    error = y - y_predicted
    # partial derivatives
    da = np.mean(-2 * x * error)
    db = np.mean(-2 * error)
    # update a and b. minus sign because gradient points to direction
    # of maximal growth, we want to minimize MSE, not maximize
    a = a - da * learning_rate
    b = b - db * learning_rate
```

```

    b = b - db * learning_rate
    mse = compute_mse(y, y_predicted)
    return a, b, mse

```

```

A_vec = np.linspace(-1, 5, 100)
B_vec = np.linspace(0, 60, 100)
A_mesh, B_mesh = np.meshgrid(A_vec, B_vec)
MSE_mesh = np.zeros_like(A_mesh)
for i in range(100):
    for j in range(100):
        Y_predicted_mesh = linear_model(x_data,
                                         A_mesh[i, j],
                                         B_mesh[i, j])
        MSE_mesh[i, j] = compute_mse(y_data, Y_predicted_mesh)

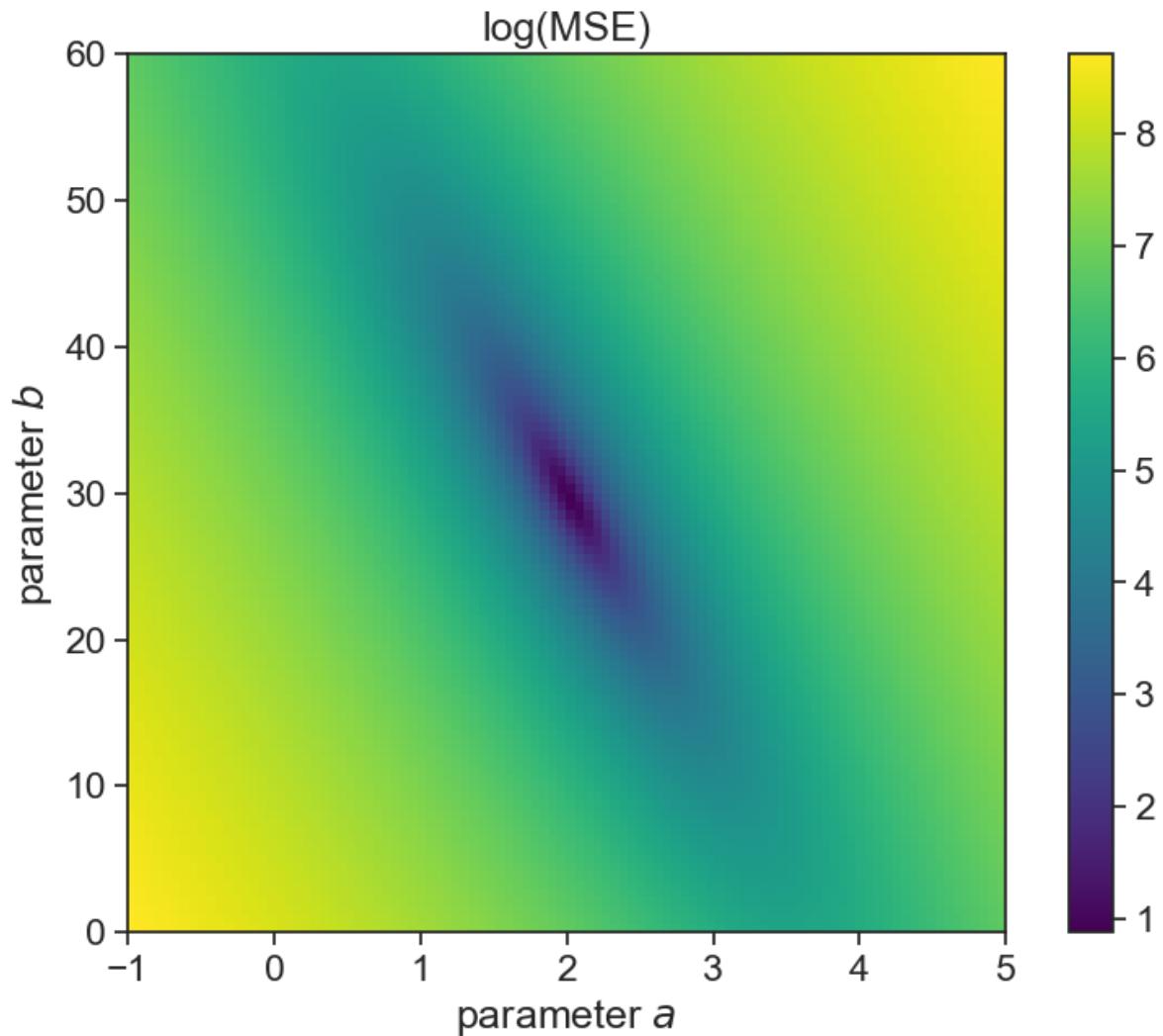
MSE_mesh_log = np.log(MSE_mesh)

```

```

fig, ax = plt.subplots(figsize=(10,7))
A_min = A_vec[0]; A_max = A_vec[-1];
B_min = B_vec[0]; B_max = B_vec[-1];
asp = (A_max - A_min) / (B_max - B_min)
cax = ax.imshow(MSE_mesh_log, extent=[A_min, A_max, B_min, B_max],
                 origin='lower', cmap='viridis', aspect = asp)
fig.colorbar(cax)
ax.set(xlabel=r"parameter $a$",
       ylabel=r"parameter $b$",
       title="log(MSE)");

```



The gradient of MSE is:

$$\nabla MSE = \frac{\partial}{\partial a} MSE \hat{a} + \frac{\partial}{\partial b} MSE \hat{b}$$

Taking the partial derivatives:

$$\begin{aligned}\frac{\partial}{\partial a} MSE &= \frac{1}{N} \sum_{i=1}^N (-2x_i)[y_i - (a \cdot x_i + b)] \\ &= \text{mean}[(-2x_i) \cdot (\text{measured} - \text{modelled})] \\ \frac{\partial}{\partial b} MSE &= \frac{1}{N} \sum_{i=1}^N (-2)[y_i - (a \cdot x_i + b)] \\ &= \text{mean}[(-2) \cdot (\text{measured} - \text{modelled})]\end{aligned}$$

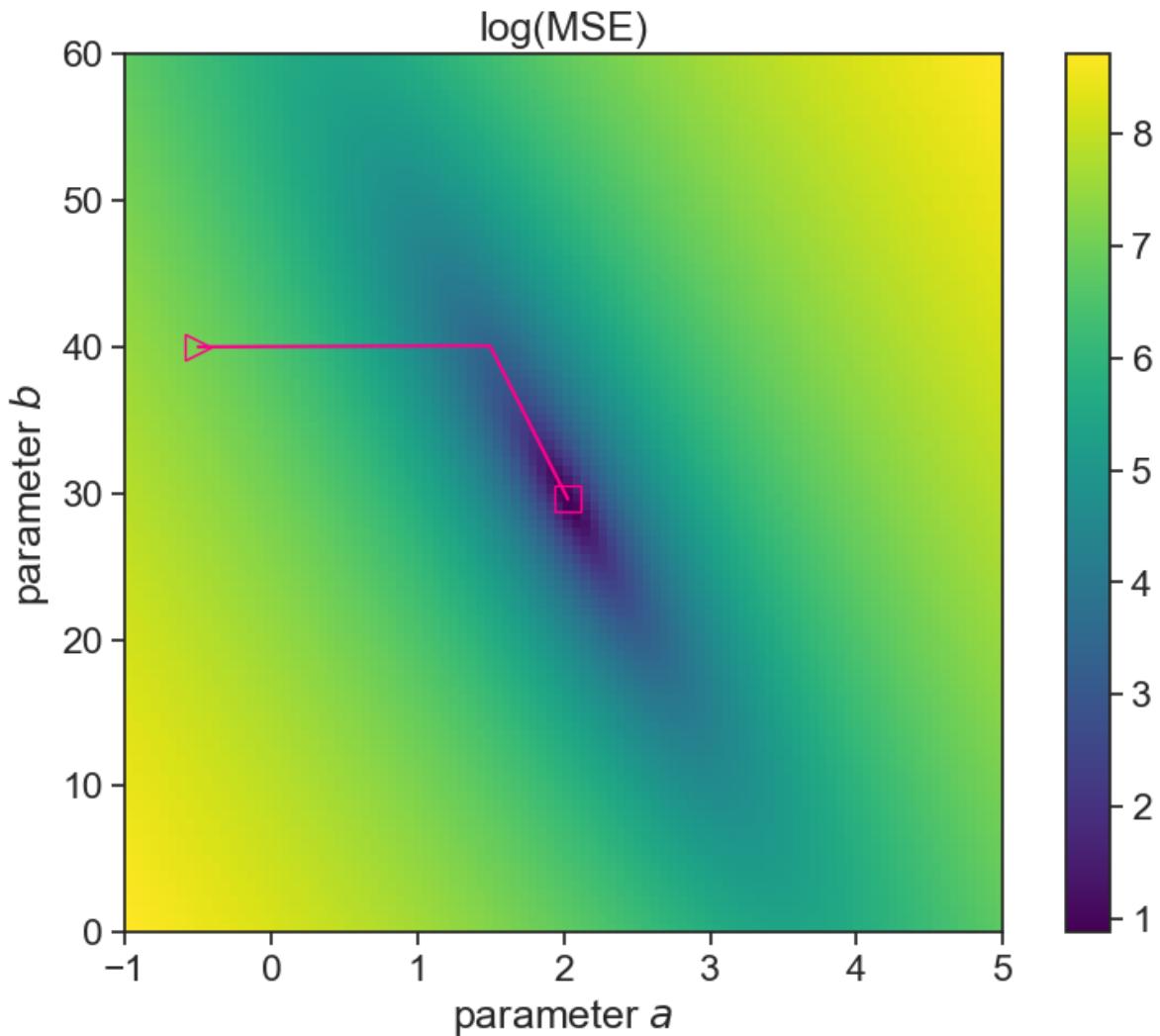
```
lr = 1.0e-3
n_iterations = 30000
a_fit = np.zeros(n_iterations)
b_fit = np.zeros(n_iterations)
mse_fit = np.zeros(n_iterations)
a_fit[0] = -0.5
b_fit[0] = 40.0
for i in np.arange(1, n_iterations):
    a_fit[i], b_fit[i], mse_fit[i] = gradient_descent(x=x_data, y=y_data, a=a_fit[i-1], b=b_fit[i-1])
print(f"best fit: a={a_fit[-1]:.2f}, b={b_fit[-1]:.2f}")
```

best fit: a=2.03, b=29.60

```
fig, ax = plt.subplots(figsize=(10,7))
A_min = A_vec[0]; A_max = A_vec[-1];
B_min = B_vec[0]; B_max = B_vec[-1];
asp = (A_max - A_min) / (B_max - B_min)
cax = ax.imshow(MSE_mesh_log, extent=[A_min, A_max, B_min, B_max],
                 origin='lower', cmap='viridis', aspect = asp)
fig.colorbar(cax)

ax.plot(a_fit, b_fit, color="xkcd:hot pink")
ax.plot(a_fit[0], b_fit[0], linestyle='None', marker='>', markerfacecolor='none', markeredgecolor='black')
ax.plot(a_fit[-1], b_fit[-1], linestyle='None', marker='s', markerfacecolor='none', markeredgecolor='black')

ax.set(xlabel=r"parameter $a$",
       ylabel=r"parameter $b$",
       title="log(MSE)");
```



```

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(x_data, y_data, 'o', label="data")

iter_list = [1, 2, 10, 1000, 30000]
color_list = np.linspace(0.8, 0.1, len(iter_list))

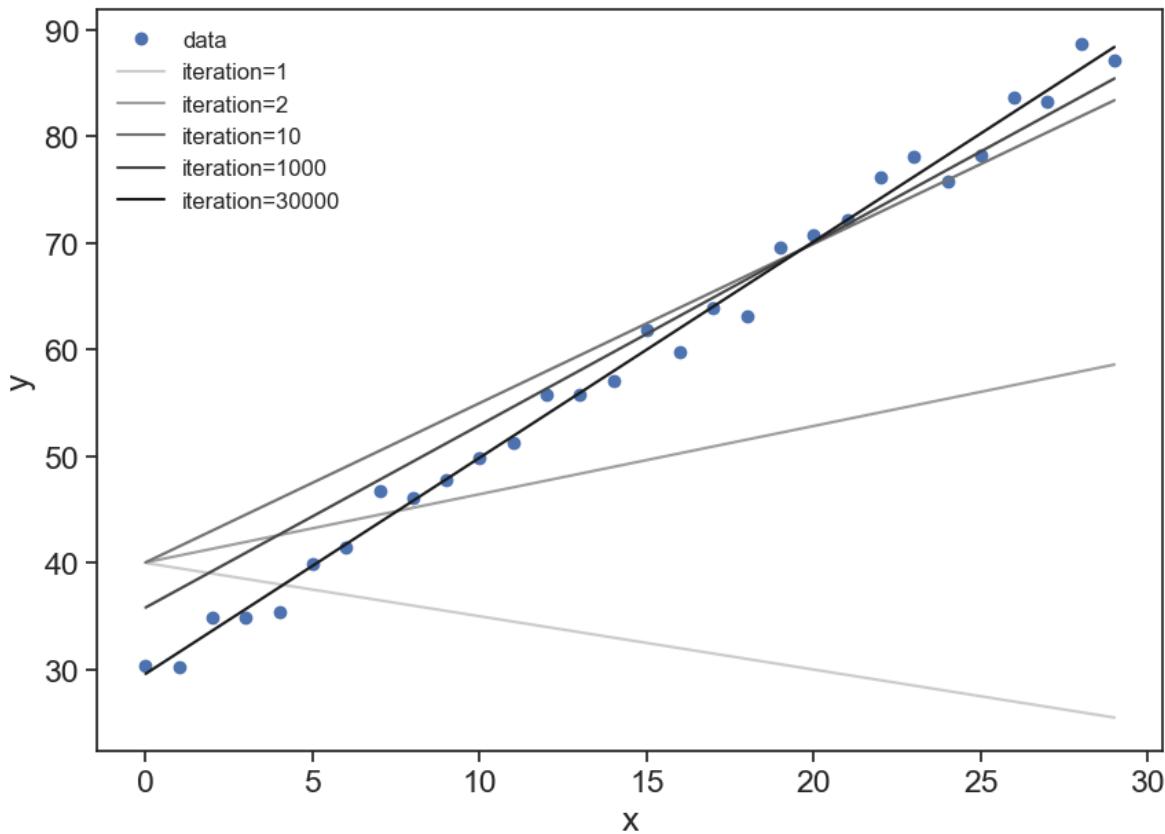
for i in range(len(iter_list)):
    ax.plot(x_data, a_fit[iter_list[i]-1] * x_data + b_fit[iter_list[i]-1],
            label=f"iteration={iter_list[i]:d}",
            color=[color_list[i]]*3
            )
ax.set(xlabel="x",

```

```

    ylabel="y")
ax.legend(frameon=False, fontsize=12);

```



13.3.1 learning rate

The learning rate multiplies the gradient (partial derivatives in each parameter direction), therefore by playing with its values, we can converge slower or faster to the optimal solution we are seeking. There is a tradeoff in play here. If the learning curve is too small, we will have to run a lot of simulations steps to converge to the desired solution. On the other hand, if the learning rate is too large, the algorithm might not be able to converge to the region of the parameter space with the lowest MSE.

```

lr = 1.4e-3
n_iterations = 5000
a_fit_2 = np.zeros(n_iterations)
b_fit_2 = np.zeros(n_iterations)

```

```
mse_fit_2 = np.zeros(n_iterations)
a_fit_2[0] = -0.5
b_fit_2[0] = 40.0
for i in np.arange(1, n_iterations):
    a_fit_2[i], b_fit_2[i], mse_fit_2[i] = gradient_descent(x=x_data, y=y_data, a=a_fit_2[i-1], b=b_fit_2[i-1])
# print(f"best fit: a={a_fit_2[-1]:.2f}, b={b_fit_2[-1]:.2f}")
```

best fit: a=2.01, b=29.87

```
lr = 5e-4
n_iterations = 5000
a_fit_3 = np.zeros(n_iterations)
b_fit_3 = np.zeros(n_iterations)
mse_fit_3 = np.zeros(n_iterations)
a_fit_3[0] = -0.5
b_fit_3[0] = 40.0
for i in np.arange(1, n_iterations):
    a_fit_3[i], b_fit_3[i], mse_fit_3[i] = gradient_descent(x=x_data, y=y_data, a=a_fit_3[i-1], b=b_fit_3[i-1])
# print(f"best fit: a={a_fit_3[-1]:.2f}, b={b_fit_3[-1]:.2f}")
```

best fit: a=1.88, b=32.44

```
lr = 3.5e-3
n_iterations = 5000
a_fit_4 = np.zeros(n_iterations)
b_fit_4 = np.zeros(n_iterations)
mse_fit_4 = np.zeros(n_iterations)
a_fit_4[0] = -0.5
b_fit_4[0] = 40.0
for i in np.arange(1, n_iterations):
    a_fit_4[i], b_fit_4[i], mse_fit_4[i] = gradient_descent(x=x_data, y=y_data, a=a_fit_4[i-1], b=b_fit_4[i-1])
# print(f"best fit: a={a_fit_4[-1]:.2f}, b={b_fit_4[-1]:.2f}")
```

best fit: a=1551.68, b=108.47

```
fig, ax = plt.subplots(figsize=(10,7))

cmap = plt.cm.Blues
colors = [cmap(0.9), cmap(0.6), cmap(0.3)]
```

```

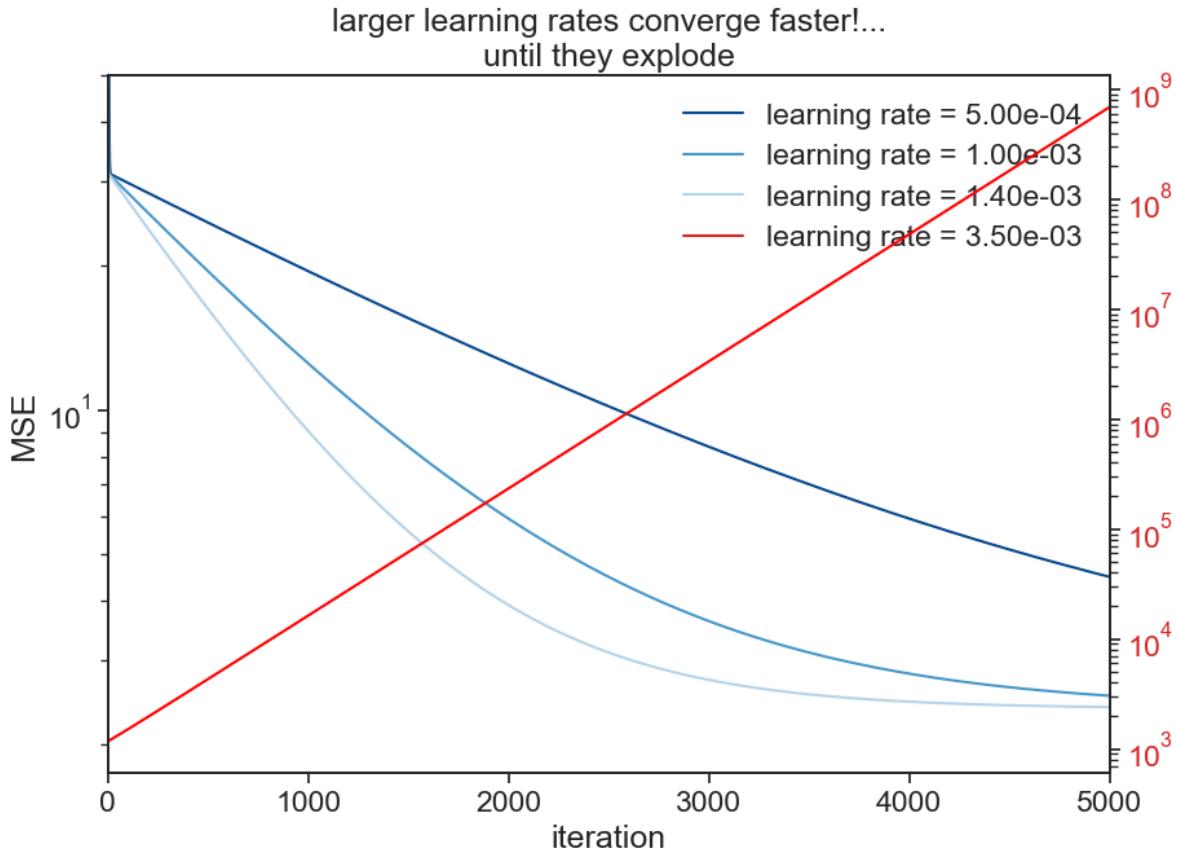
ax.plot((mse_fit_3), label=f"learning rate = {5e-4:.2e}", color=colors[0])
ax.plot((mse_fit), label=f"learning rate = {1.0e-3:.2e}", color=colors[1])
ax.plot((mse_fit_2), label=f"learning rate = {1.4e-3:.2e}", color=colors[2])
ax.plot([], [], label=f"learning rate = {3.5e-3:.2e}", color="tab:red")

ax.set_yscale('log')
ax.set(ylim=[0, 50],
      xlim=[0, 5000],
      xlabel="iteration",
      ylabel="MSE",
      title="larger learning rates converge faster!...\\nuntil they explode")

ax2 = ax.twinx()
ax2.plot((mse_fit_4), color="red")
ax2.set_yscale('log')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax.legend(frameon=False);

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440__00000gn/T/ipykernel_86050/3729129073.py:12: UserWarning:
```



This way of finding the optimal set of parameters is the first and very simplest method. Much more sophisticated methods were developed throughout the years. This basic idea of finding the optimal values for a list of parameters is the basic idea behind machine learning methods.

13.3.2 local and global minima

The landscape we saw in the example above was well behaved, and we easily converged to a reasonable solution. Sometimes the landscape is more convoluted, and we might get stuck on local minima, depending on the initial conditions for the parameters. In that case, we need to help the algorithm by providing it with other initial guesses for the parameters.

13.4 Green & Ampt

$$f = \frac{A}{F} + B$$

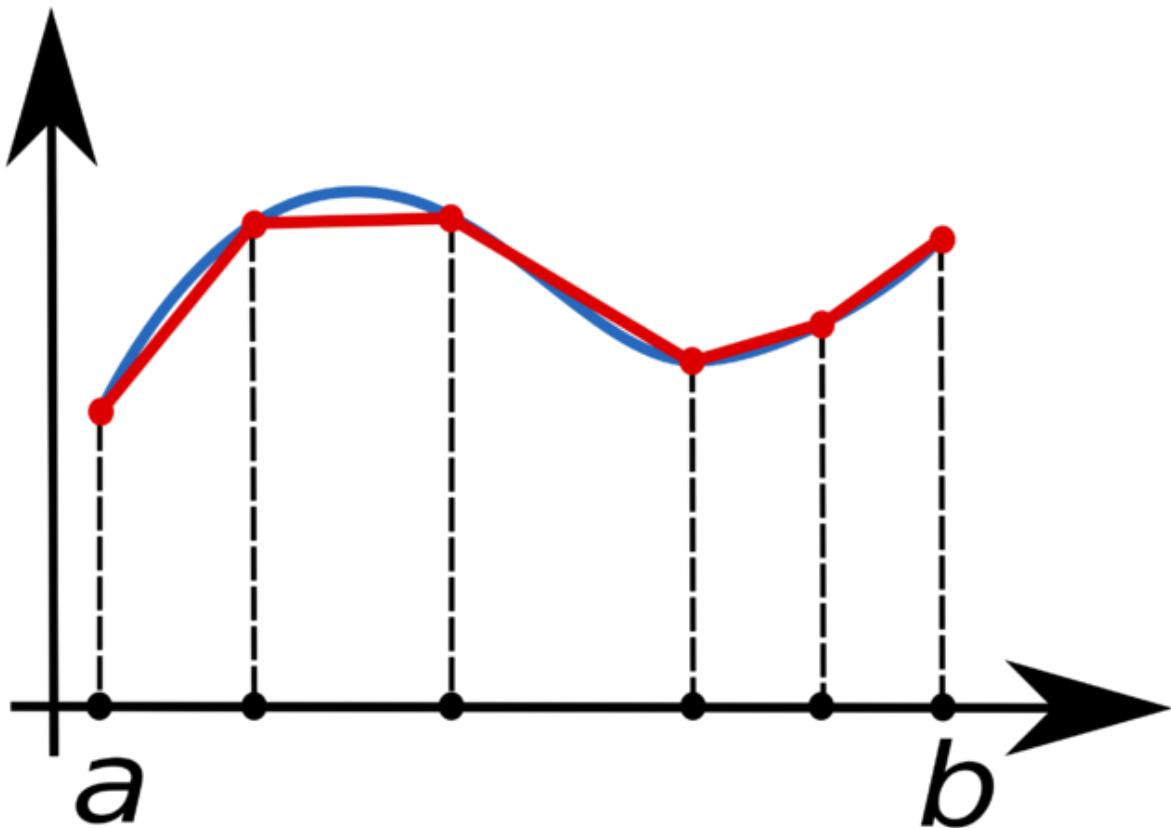
where

- $A = K_{\text{sat}} \cdot |\psi_f| \cdot (\phi - \theta_0)$
- $B = K_{\text{sat}}$

Write a function that calculates the cumulative of the infiltration rate.

$$F(t) = \int_0^t f(t) \, dt$$

Use numpy's `trapz` function, that implements the "trapezoidal rule"



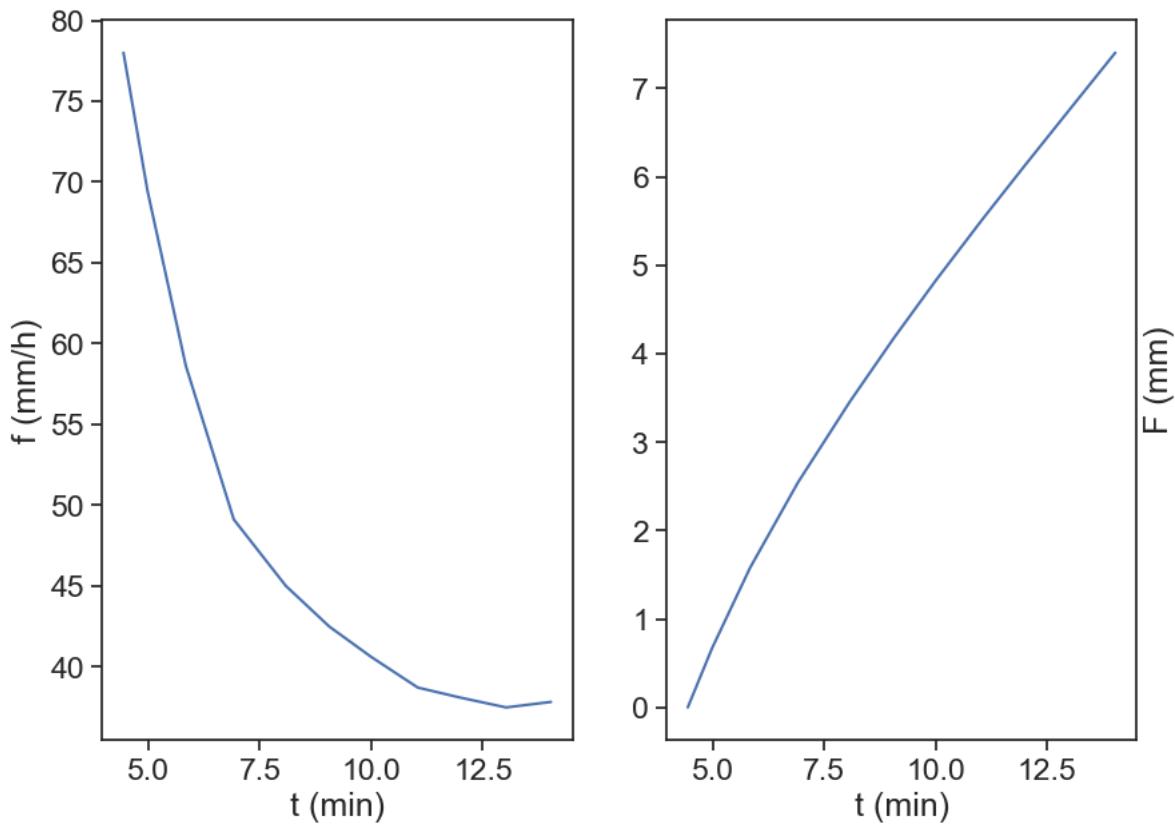
```
def cumulative_F(t, f):
    F = np.array([0])
    t = t/60 # convert minute to hour
    for i in np.arange(2,len(t)+1):
        area = np.trapz(f[:i], t[:i])
        F = np.concatenate([F, [area]])
    return F
```

```

    return F

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,7))
t, f = d1[:,0], d1[:,1]
F = cumulative_F(t, f)
ax1.plot(t, f, label="f, rate")
ax2.plot(t, F, label="F, cumulative")
ax1.set(xlabel="t (min)",
         ylabel="f (mm/h)")
ax2.set(xlabel="t (min)",
         ylabel="F (mm)")
ax2.yaxis.set_label_position("right")

```



Plot f as a function of F . Try to guess A and B that give reasonable results.

```

fig, ax = plt.subplots(figsize=(10,7))
t, f = d1[:,0], d1[:,1]
F = cumulative_F(t, f)

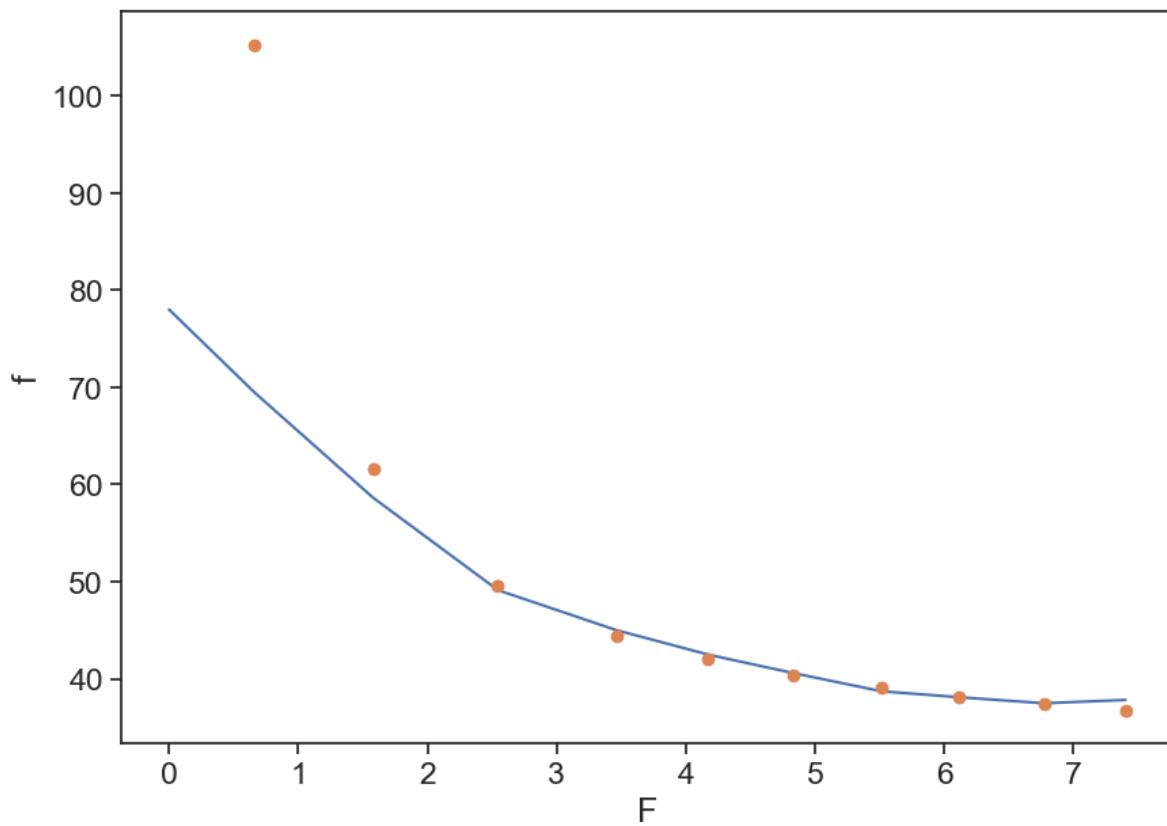
```

```

ax.plot(F, f)
A=50; B=30;
ax.plot(F, A/F + B, 'o')
ax.set(xlabel="F",
       ylabel="f");

```

```
/var/folders/c3/7hp0d36n6vv8jc9hm2440_/_00000gn/T/ipykernel_86050/768544551.py:7: RuntimeWarning:
ax.plot(F, A/F + B, 'o')
```



Use the `curve_fit` to find the optimal values for A and B .

```

def G_and_A(F, A, B):
    return A/F + B

popt, pcov = curve_fit(f=G_and_A,      # model function
                       xdata=F[1:],   # x data
                       ydata=f[1:],   # y data

```

```

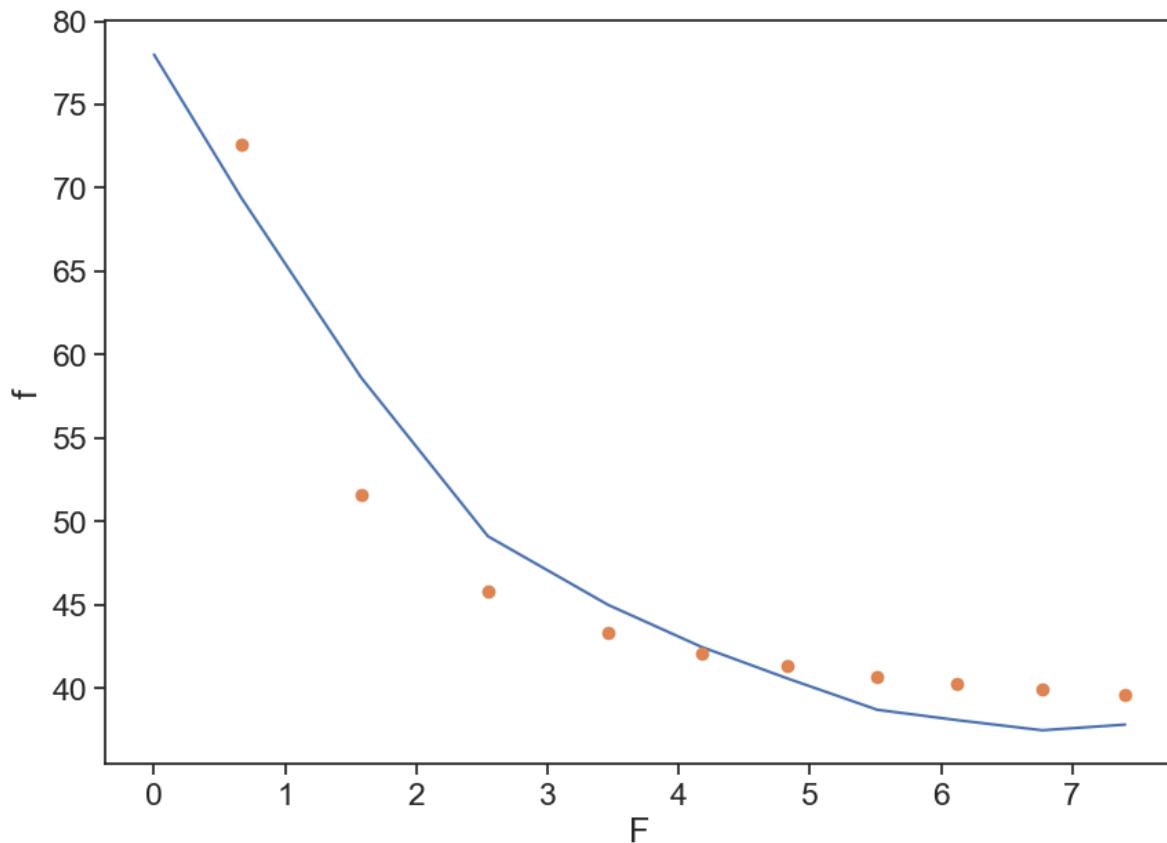
        p0=(50, 30),    # initial guess of the parameters
    )

# popt, pcov = curve_fit(G_and_A, F[1:], f[1:], p0=(50, 30)) # p0 = initial guess
print(popt)

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(F, f)
ax.plot(F[1:], popt[0]/F[1:] + popt[1], 'o')
ax.set(xlabel="F",
       ylabel="f");

```

[24.12368526 36.34242813]



13.5 Homework

Go to [Soil Texture Calculator](#), estimate the texture of “standard soil” in Nassif & Wilson, 1975.

Part V

Streamflow

14 Streamflow

14.1 Watershed -

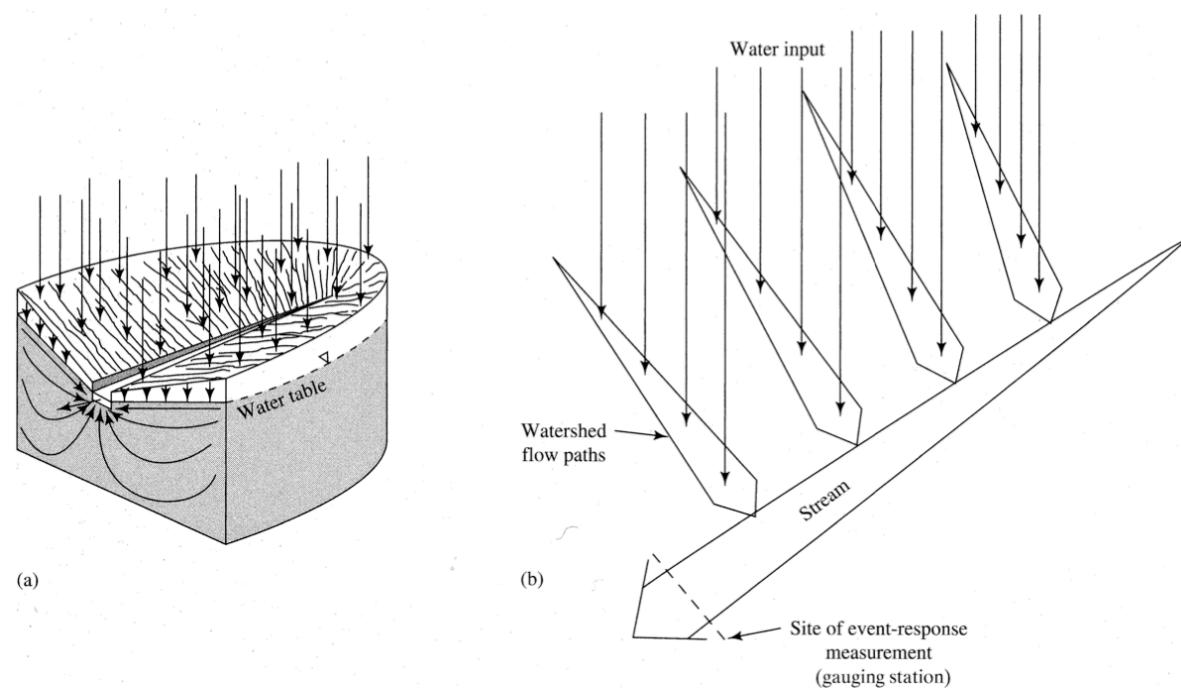


Figure 10.3 (a) Schematic flow paths in a small upland watershed receiving water input. (b) The essence of watershed response as the space- and time-integrated result of flow with lateral inflows.

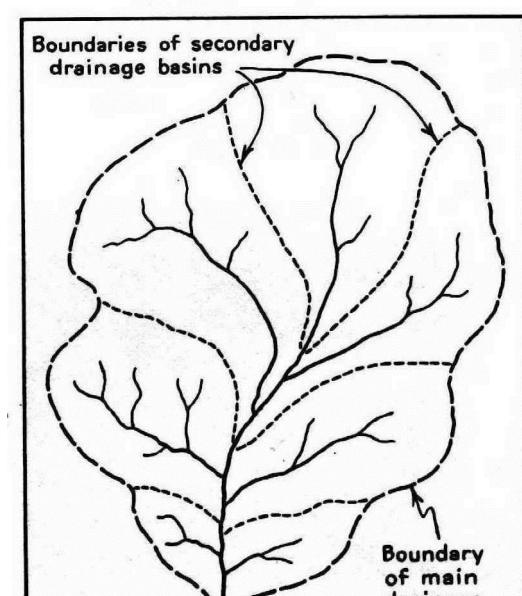
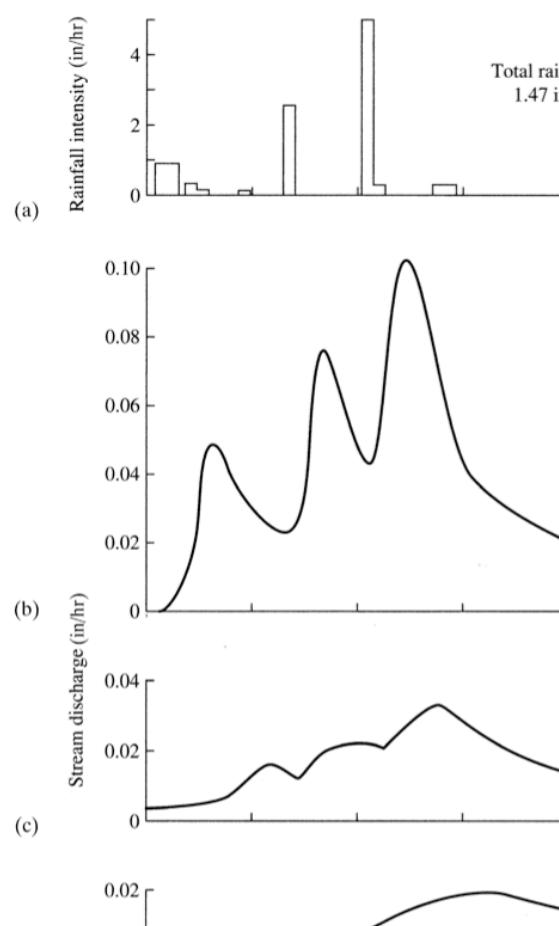


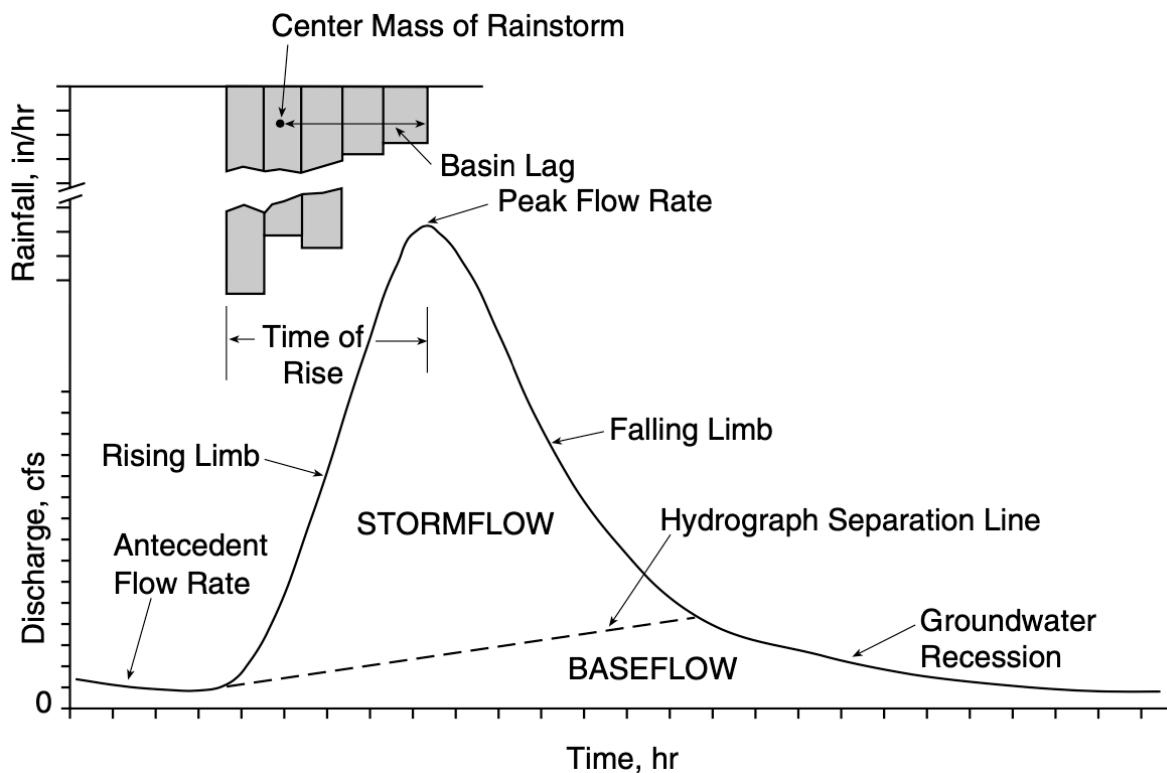
Figure 10.5 Changes in hydrograph shape at a series of gauging stations along the Sleepers River in Danville, Vermont, in response to an intense rainstorm [(a) is a hyetograph]. Note that the left-hand hydrograph ordi-



Watershed response:

- The volume of water appearing in the apparent response hydrograph for a given event is usually only a fraction (often a very small fraction) of the total input. The remainder of the water input ultimately leaves the watershed as:
 1. evapotranspiration;
 2. streamflow that occurs so long after the event that it cannot be associated with that event; or
 3. ground-water outflow from the watershed.
- The water identified as the response to a given event may originate on only a fraction of the watershed; this fraction is called the contributing area.
- The extent of the contributing area may vary from event to event and during an event.
- At least some of the water identified as the response to a given event may be "old water" that entered the watershed in a previous event.

14.2 base flow separation



Base flow

Base flow is the portion of streamflow that is presumed to have entered the watershed in previous events and to be derived from persistent, slowly varying sources. (Ground water is usually assumed to be the main, if not the only, such source.)

Event flow

Event flow (also called direct runoff, storm runoff, quick flow, or storm flow) is considered to be the direct response to a given water-input event.

Total flow

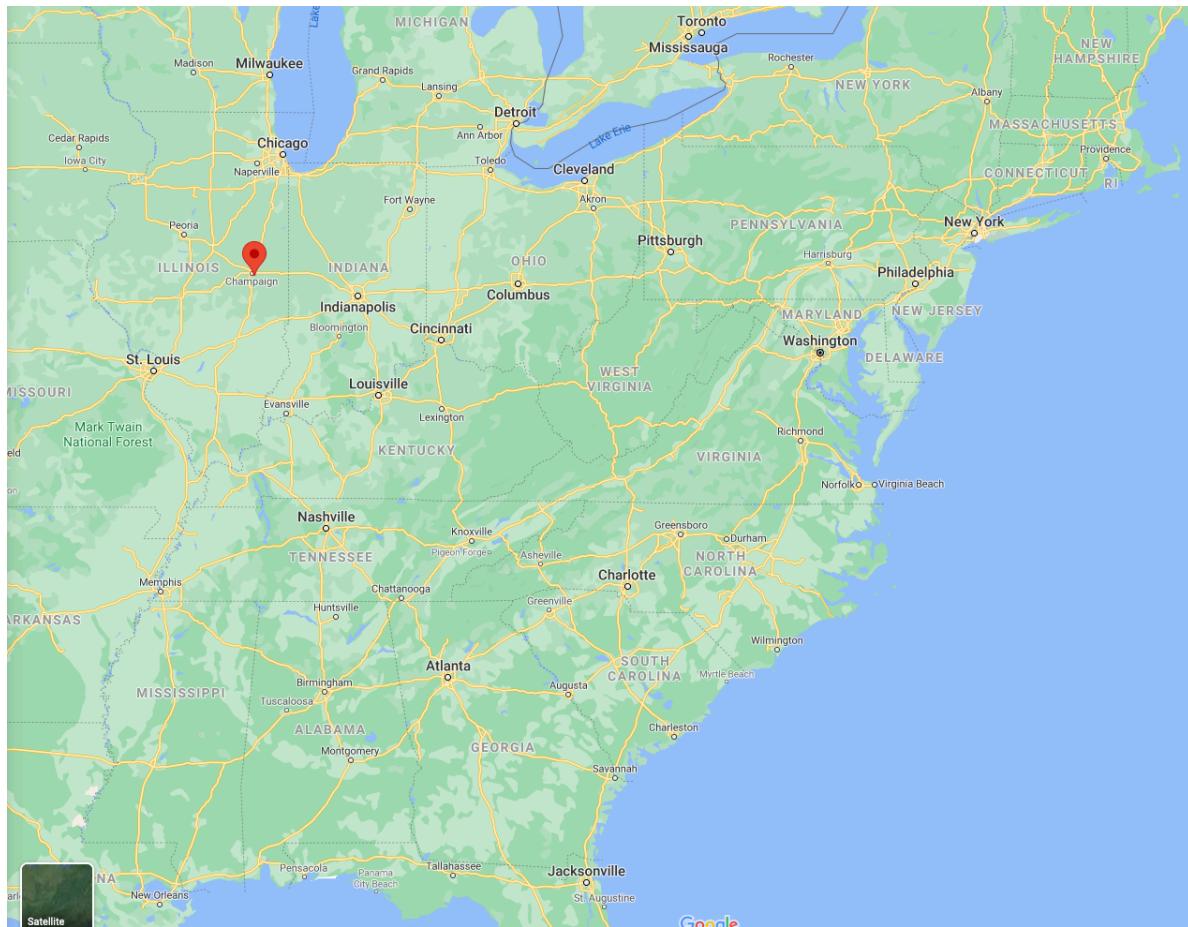
Total flow rate at any instant $q(t)$ is the sum of event-flow rate $q^*(t)$ and base-flow rate $q_{BF}(t)$:

$$q(t) = q^*(t) + q_{BF}(t)$$

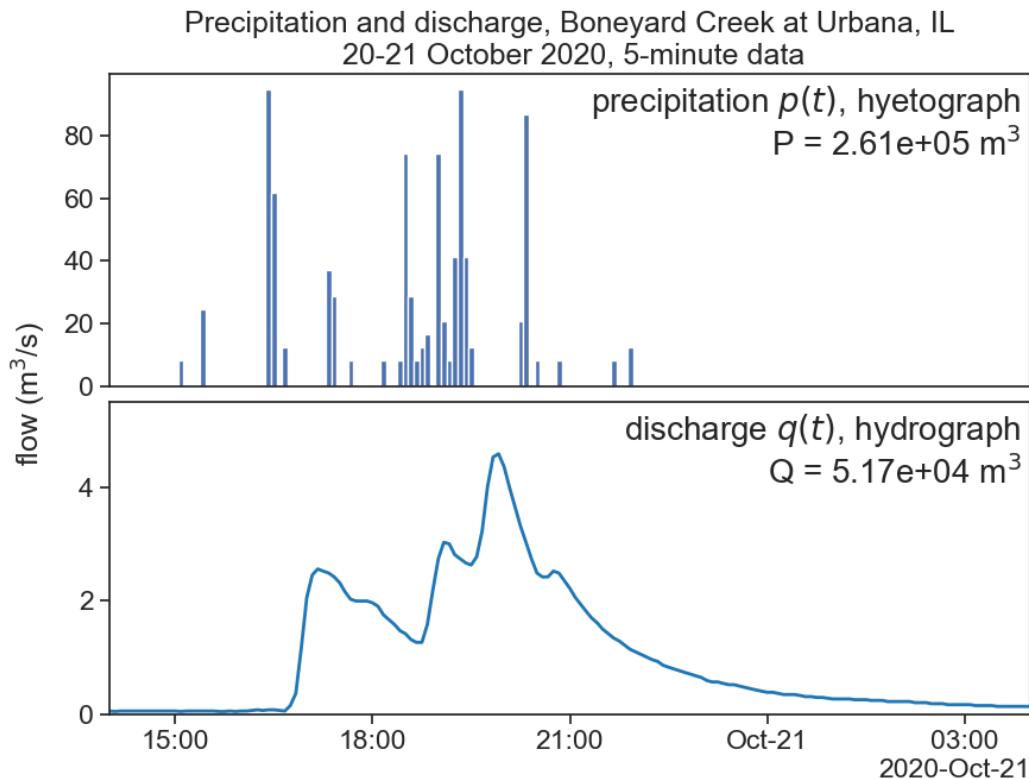
Attention!

Graphical flow separation techniques are heuristic and have no direct scientific basis.

14.3 Urbana, IL



14.3.1 hyetograph, hydrograph

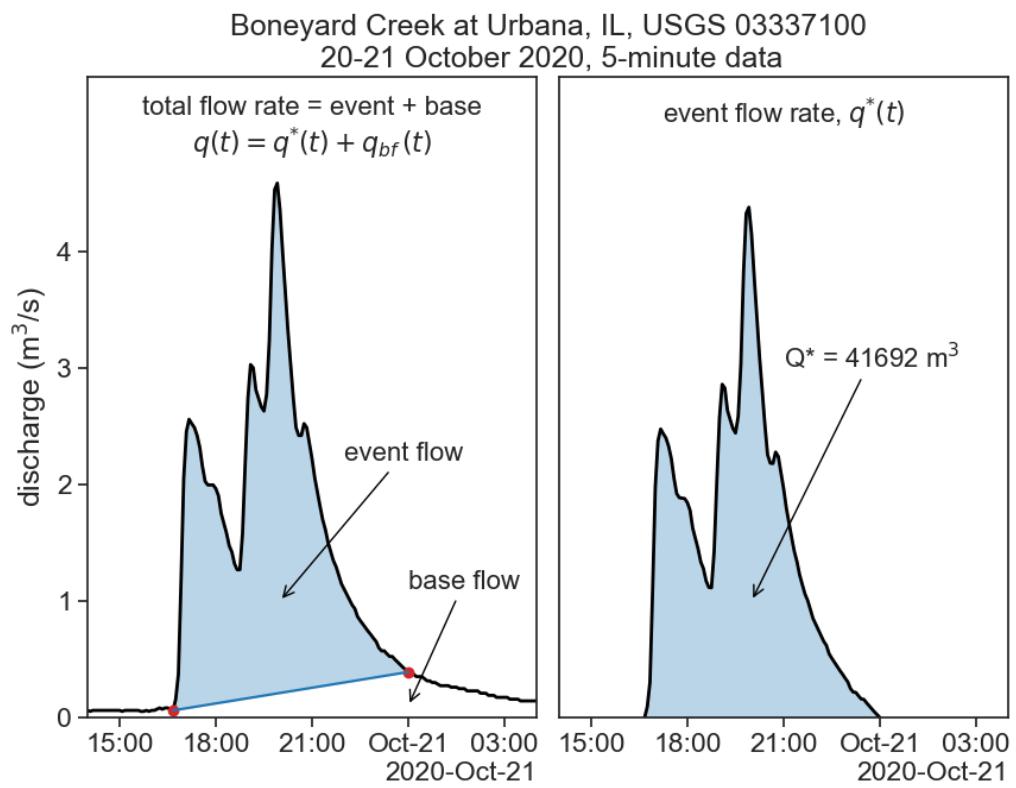


14.3.2 notation

Box 10.1 Notation

| | | |
|---|-------------|--|
| In this chapter, we use the following notation: | $q(t)$ | total streamflow rate as a continuous function of time [L T^{-1}] or [$\text{L}^3 \text{T}^{-1}$] |
| p rate of water input (rain plus snowmelt) in a storm event [L T^{-1}] | $q^*(t)$ | event-flow rate as a continuous function of time [L T^{-1}] or [$\text{L}^3 \text{T}^{-1}$] |
| p^* rate of effective water input (\equiv water that appears as streamflow in response to a water-input event) [L T^{-1}] | $q_{BF}(t)$ | base-flow rate as a continuous function of time [L T^{-1}] or [$\text{L}^3 \text{T}^{-1}$] |
| P total volume of water input in a storm event [L] or [L^3] | Q | total volume of streamflow in a storm event [L^3] or [L] |
| P^* total volume of effective water input in a storm event [L] or [L^3] | Q^* | total volume of event flow in a storm event [L^3] or [L] |

14.3.3 base flow separation



14.3.4 effective precipitation = effective discharge

$$P^* = Q^*$$

Effective precipitation and discharge, Boneyard Creek at Urbana, IL
20-21 October 2020, 5-minute data

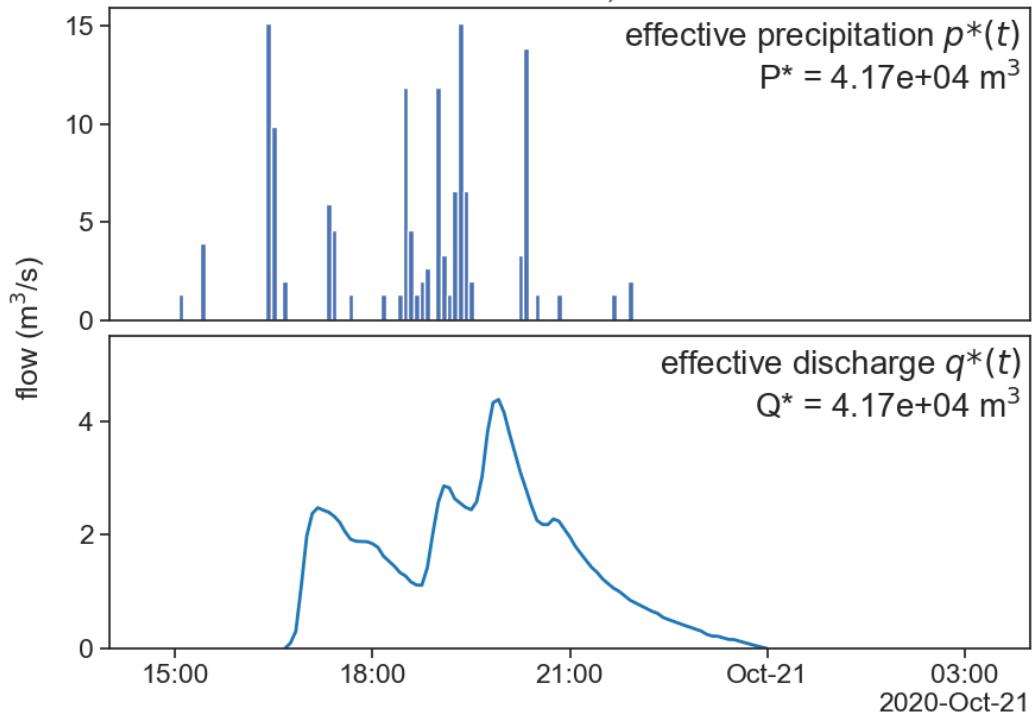
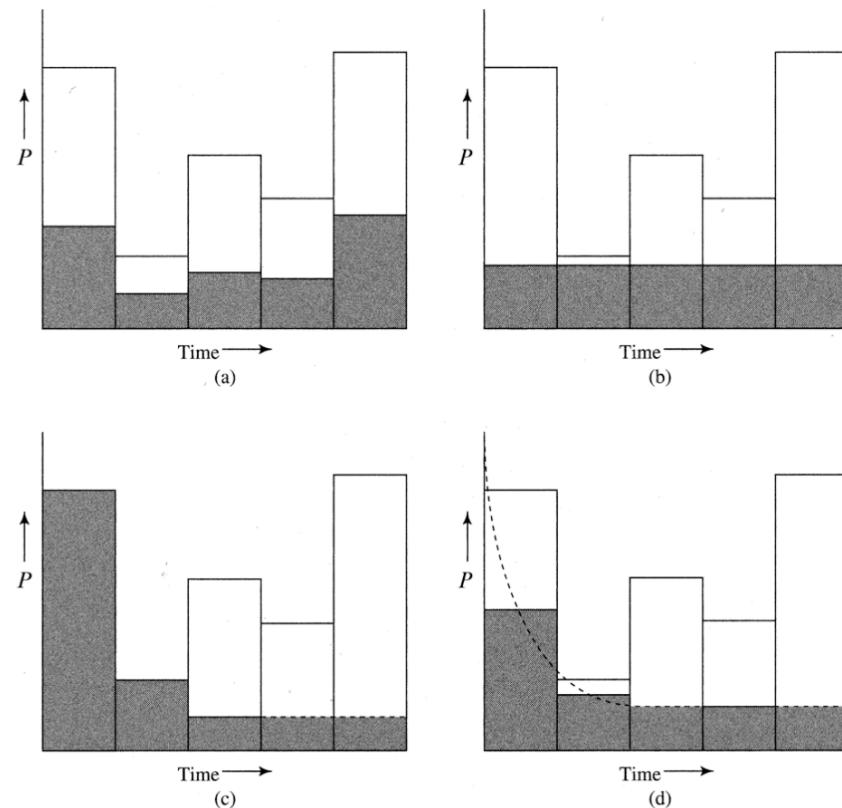


Figure 10.49 Conceptual models for estimating effective water input, P^* , from hyetograph of water input, P . Losses are shaded portions, P^* is unshaded. (a) Losses equal a constant fraction of the water input for each time period. (b) Losses equal a constant rate throughout the event. (c) Losses are given by an **initial abstraction** (which may be a specified amount or all input over an initial time period) followed by a constant rate (which may be zero). (d) Losses are given by an approximation to an infiltration-type curve (dotted line), such as given by the Green-and-Ampt or Philip approach (see chapter 8) [adapted from Pilgrim and Cordery (1992)].



14.3.5 time lags

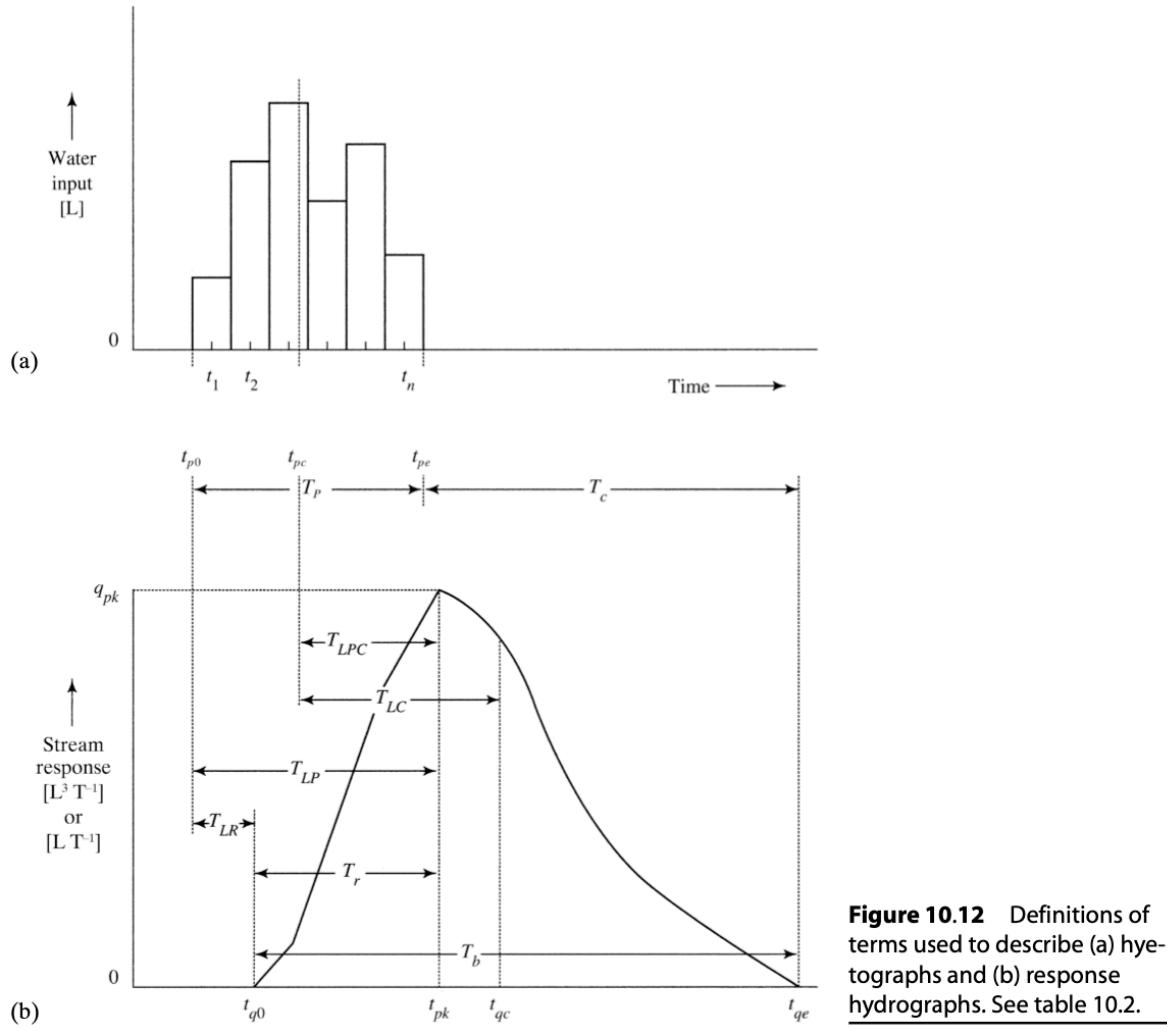
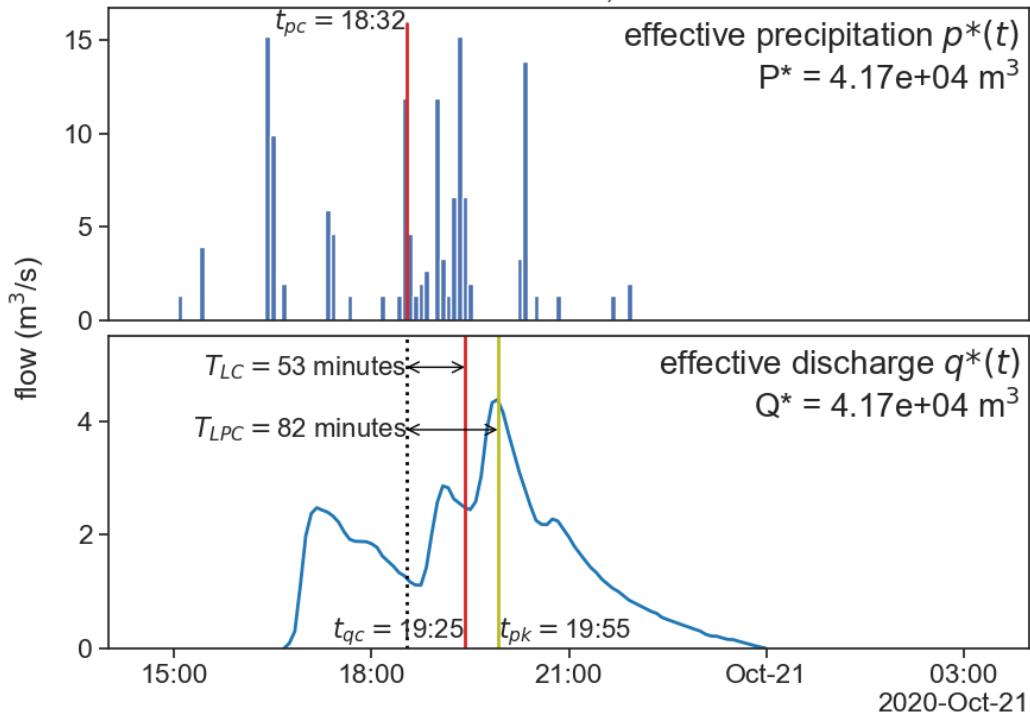


Figure 10.12 Definitions of terms used to describe (a) hyetographs and (b) response hydrographs. See table 10.2.

Table 10.2 Definitions of Terms Used to Describe Hyetographs and Response Hydrographs (see figure 10.12).

| Time Instants | Time Durations |
|---|---|
| t_{p0} ≡ beginning of effective water input | T_p ≡ duration of effective water input = $t_{p0} - t_{pe}$ |
| t_{pc} ≡ centroid of effective water input | T_{LR} ≡ response lag = $t_{q0} - t_{p0}$ |
| t_{pe} ≡ end of effective water input | T_r ≡ time of rise = $t_{pk} - t_{q0}$ |
| t_{q0} ≡ beginning of hydrograph rise | T_{LP} ≡ lag-to-peak = $t_{pk} - t_{p0}$ |
| t_{pk} ≡ time of peak discharge | T_{LPC} ≡ centroid lag-to-peak = $t_{pk} - t_{pc}$ |
| t_{qc} ≡ centroid of response hydrograph | T_{LC} ≡ centroid lag = $t_{qc} - t_{pc}$ |
| t_{qe} ≡ end of response | T_b ≡ time base = $t_{qe} - t_{q0}$ |
| | T_c ≡ time of concentration = $t_{qe} - t_{pe}$ |
| | T_{eq} ≡ time to equilibrium $\approx T_c$ |

precipitation centroid and discharge centroid, Boneyard Creek at Urbana, IL
20-21 October 2020, 5-minute data



It is commonly assumed that $T_{LPC} \simeq 0.60 \cdot T_c$, where T_c is the time of concentration, i.e., the time it takes water to travel from the hydraulically most distant part of the contributing area to the outlet.

The centroid is a weighted-average time, each time instant is multiplied by the amount of flow in that instant.

Time of precipitation centroid:

$$t_{pc} = \frac{\sum_{i=1}^n p_i^* \cdot t_i}{P^*}$$

Time of streamflow centroid:

$$t_{qc} = \frac{\sum_{i=1}^n q_i^* \cdot t_i}{Q^*}$$

15 Exercises

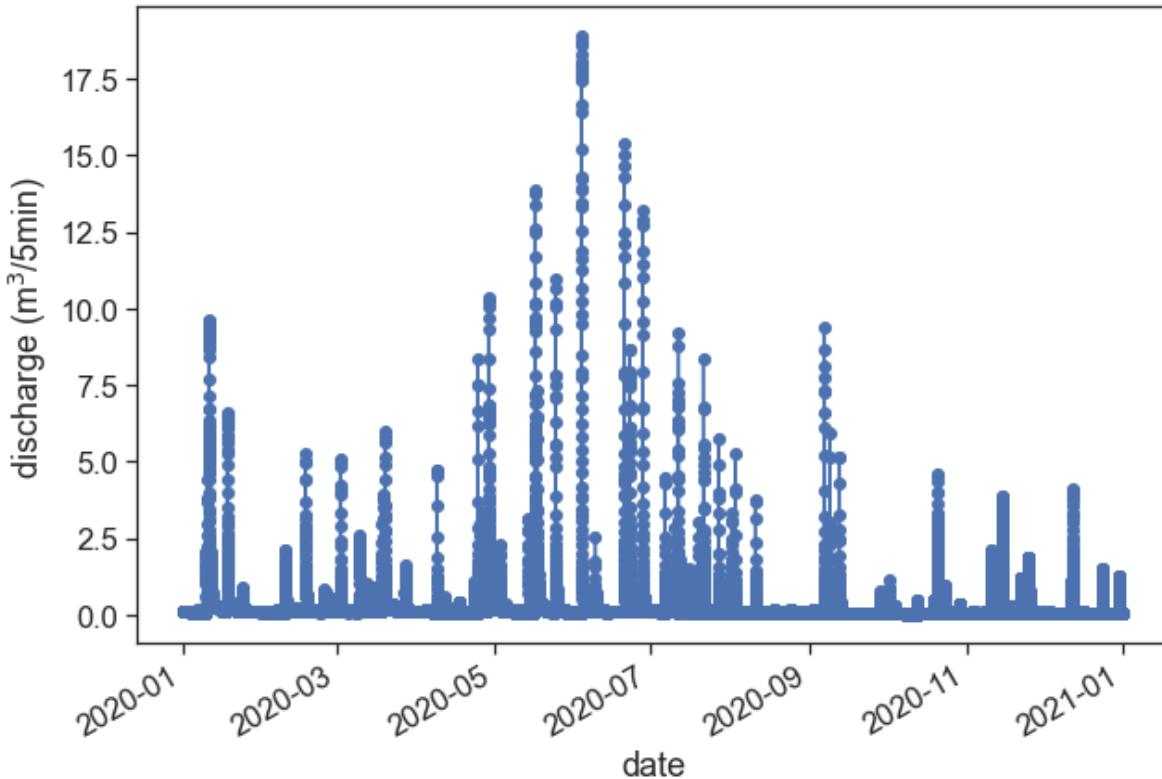
Import relevant packages

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
sns.set(style="ticks", font_scale=1.5)
from ipywidgets import *
```

Import streamflow data from USGS's [National Water Information System](#). We will be using data from [Urbana, IL](#).

```
# Drainage area: 4.78 square miles
data_file = "USGS 03337100 BONEYARD CREEK AT LINCOLN AVE AT URBANA, IL.dat"
df_q_2020 = pd.read_csv(data_file,
                        header=31,                                # no headers needed, we'll do that later
                        delim_whitespace=True,                      # blank spaces separate between columns
                        na_values=["Bkw"]  # substitute these values for missing (NaN) values
)
df_q_2020.columns = ['agency_cd', 'site_no', 'datetime', 'tz_cd', 'EDT', 'discharge', 'code']
df_q_2020['date_and_time'] = df_q_2020['datetime'] + ' ' + df_q_2020['tz_cd'] # combine date and time
df_q_2020['date_and_time'] = pd.to_datetime(df_q_2020['date_and_time'])        # interpret dates as datetime
df_q_2020 = df_q_2020.set_index('date_and_time')                                # make datetime the index
df_q_2020['discharge'] = df_q_2020['discharge'].astype(float)
df_q_2020['discharge'] = df_q_2020['discharge'] * 0.0283168 # convert cubic feet to m3

fig, ax = plt.subplots(figsize=(10,7))
ax.plot(df_q_2020['discharge'], '-o')
plt.gcf().autofmt_xdate()
ax.set(xlabel="date",
       ylabel=r"discharge (m$^3$/5min)");
```



Import sub-hourly (5-min) rainfall data from NOAA's [Climate Reference Network Data](#) website

```

data_file = "Champaign - IL.txt"
df_p_2020 = pd.read_csv(data_file,
                        header=None,                      # no headers needed, we'll do that
                        delim_whitespace=True,            # blank spaces separate between col
                        na_values=["-99.000", "-9999.0"] # substitute these values for miss
)
headers = pd.read_csv("HEADERS_sub_hourly.txt",      # load headers file
                     header=1,                      # skip the first [0] line
                     delim_whitespace=True
)
df_p_2020.columns = headers.columns                # rename df columns with headers co
# LST = local standard time
df_p_2020["LST_TIME"] = [f"{x:04d}" for x in df_p_2020["LST_TIME"]] # time needs padding of
df_p_2020['LST_DATE'] = df_p_2020['LST_DATE'].astype(str)           # convert date into str
df_p_2020['datetime'] = df_p_2020['LST_DATE'] + ' ' + df_p_2020['LST_TIME'] # combine date+ti
df_p_2020['datetime'] = pd.to_datetime(df_p_2020['datetime'])         # interpret datetime
df_p_2020 = df_p_2020.set_index('datetime')                         # make datetime the inde

```

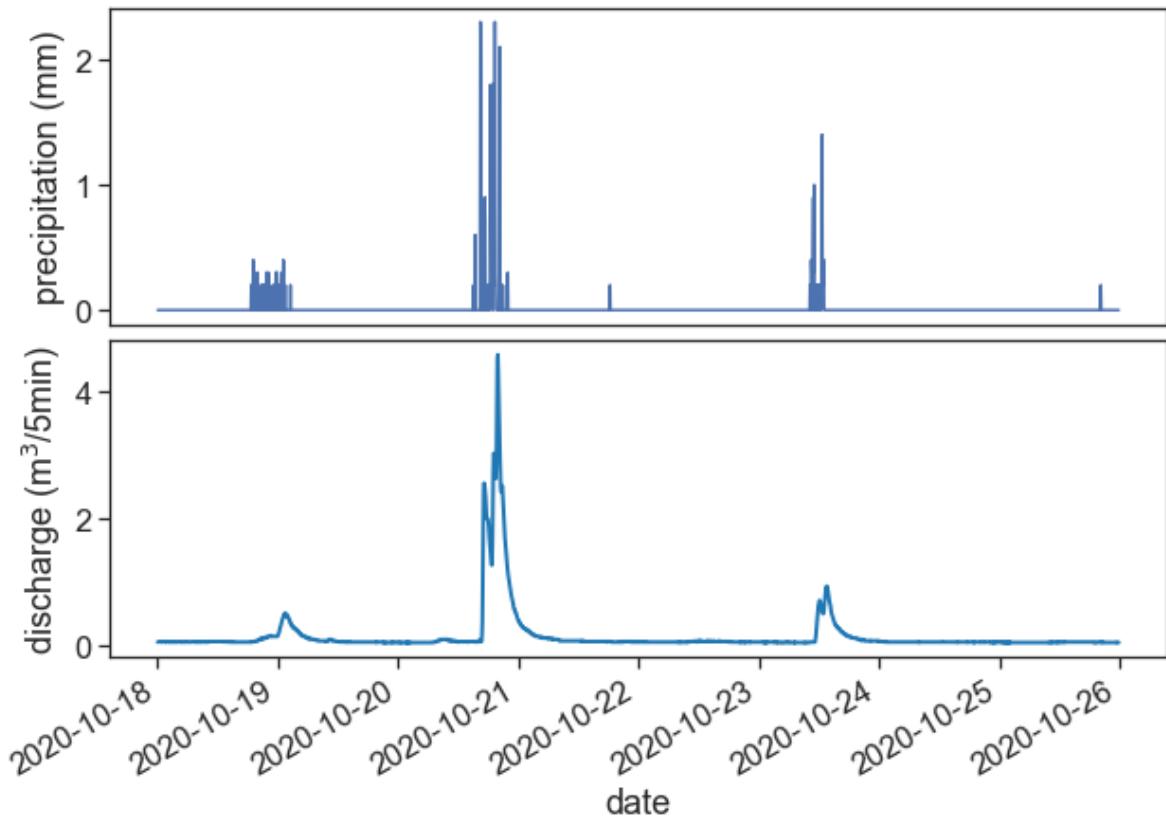
Plot rainfall and streamflow. Does this makes sense?

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,7))
fig.subplots_adjust(hspace=0.05)

start = "2020-10-18"
end = "2020-10-25"
ax1.plot(df_p_2020[start:end] ['PRECIPITATION'])
ax2.plot(df_q_2020[start:end] ['discharge'], color="tab:blue", lw=2)

ax1.set(xticks=[],
        ylabel=r"precipitation (mm)")
ax2.set(xlabel="date",
        ylabel=r"discharge ( $m^3/5min$ )")

plt.gcf().autofmt_xdate() # makes slanted dates
```



Define smaller dataframes for $p(t)$ and $q(t)$, between the dates:

```

start = "2020-10-20 14:00:00"
end = "2020-10-21 04:00:00"

```

Don't forget to convert the units to SI!

Calculate total rainfall P^* and total discharge Q^* , in m³.

```

# Drainage area: 4.78 square miles
area = 4.78 / 0.00000038610 # squared miles to squared meters
start = "2020-10-20 14:00:00"
end = "2020-10-21 04:00:00"

df_p = df_p_2020.loc[start:end]['PRECIPITATION'].to_frame()
df_p_mm = df_p_2020.loc[start:end]['PRECIPITATION'].to_frame()
df_q = df_q_2020.loc[start:end]['discharge'].to_frame()

df_p['PRECIPITATION'] = df_p['PRECIPITATION'].values * area / 1000 # mm to m3 in the whole w
df_p['PRECIPITATION'] = df_p['PRECIPITATION'] / 60 / 5 # convert m3 per 5 min to m3/s

P = df_p['PRECIPITATION'].sum() * 60 * 5
Q = df_q['discharge'].sum() * 60 * 5

print("total precipitation during event: Pstar = {:.1e} m3".format(P.sum()))
print("total streamflow during event: Qstar = {:.1e} m3".format(Q.sum()))

```

```

total precipitation during event: Pstar = 2.6e+05 m3
total streamflow during event: Qstar = 5.2e+04 m3

```

Make another graph of $p(t)$ and $q(t)$, now with SI units.

```

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10,7))
fig.subplots_adjust(hspace=0.05)

start = "2020-10-18"
end = "2020-10-25"
ax1.plot(df_p['PRECIPITATION'])
ax2.plot(df_q['discharge'], color="tab:blue", lw=2)

ax1.set(xticks=[],
        ylabel=r"precipitation (m$^3$/s)",
        title="Precipitation and discharge, Boneyard Creek at Urbana, IL\n 20-21 October 2020")

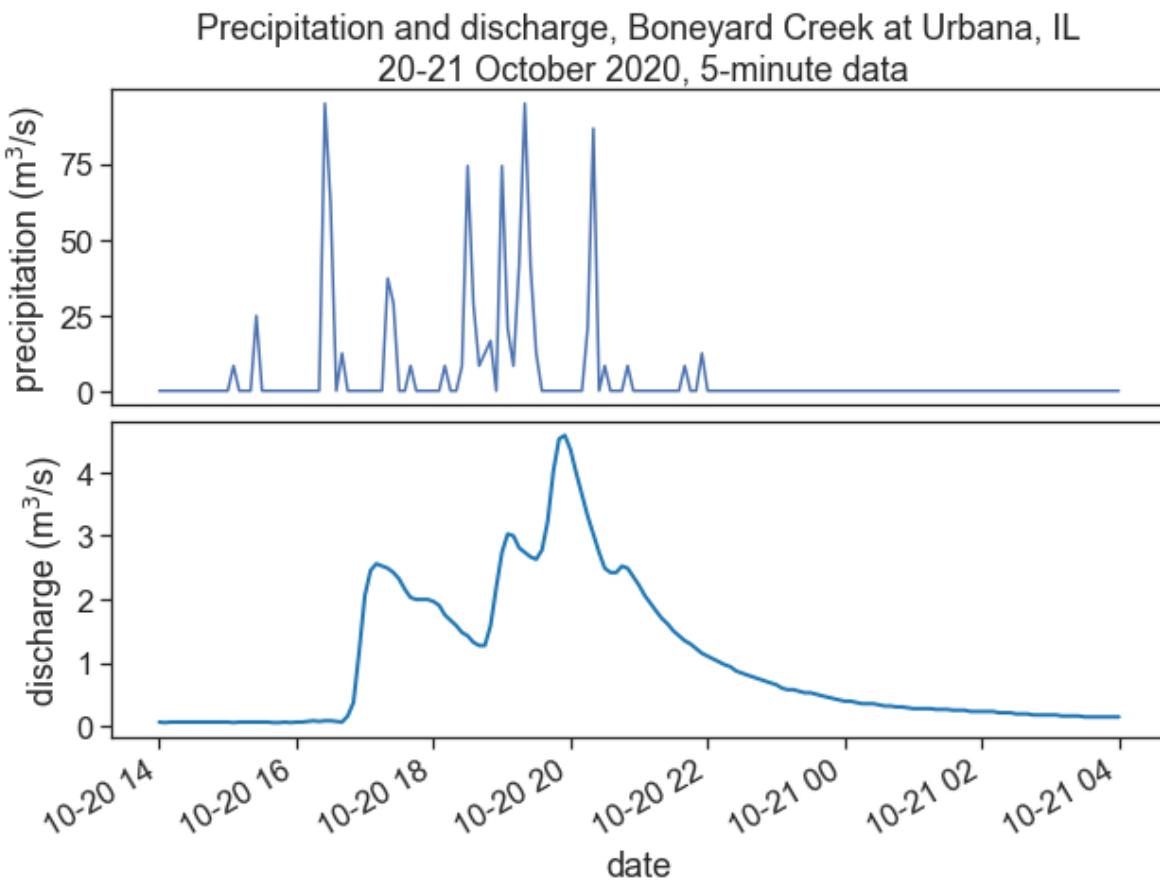
```

```

ax2.set(xlabel="date",
        ylabel=r"discharge (m$^3$/s)")

plt.gcf().autofmt_xdate() # makes slanted dates

```



It's time for base flow separation! Convert $q(t)$ into $q^*(t)$

```

from matplotlib.dates import HourLocator, DateFormatter
import matplotlib.dates as mdates
import matplotlib.ticker as ticker

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,7))
fig.subplots_adjust(wspace=0.05)

ax1.plot(df_q['discharge'], color="black", lw=2)
point1 = pd.to_datetime("2020-10-20 16:40:00")

```

```

point2 = pd.to_datetime("2020-10-21 00:00:00")
two_points = df_q.loc[[point1, point2]]['discharge']
ax1.plot(two_points, 'o', color="tab:red")

new = pd.DataFrame(data=two_points, index=two_points.index)

df_linear = (new.resample("5min") #resample
              .interpolate(method='time') #interpolate by time
              )

ax1.plot(df_linear, color="tab:blue")

df_between_2_points = df_q.loc[df_linear.index]
ax1.fill_between(df_between_2_points.index, df_between_2_points['discharge'],
                  y2=df_linear['discharge'],
                  color="tab:blue", alpha=0.3)

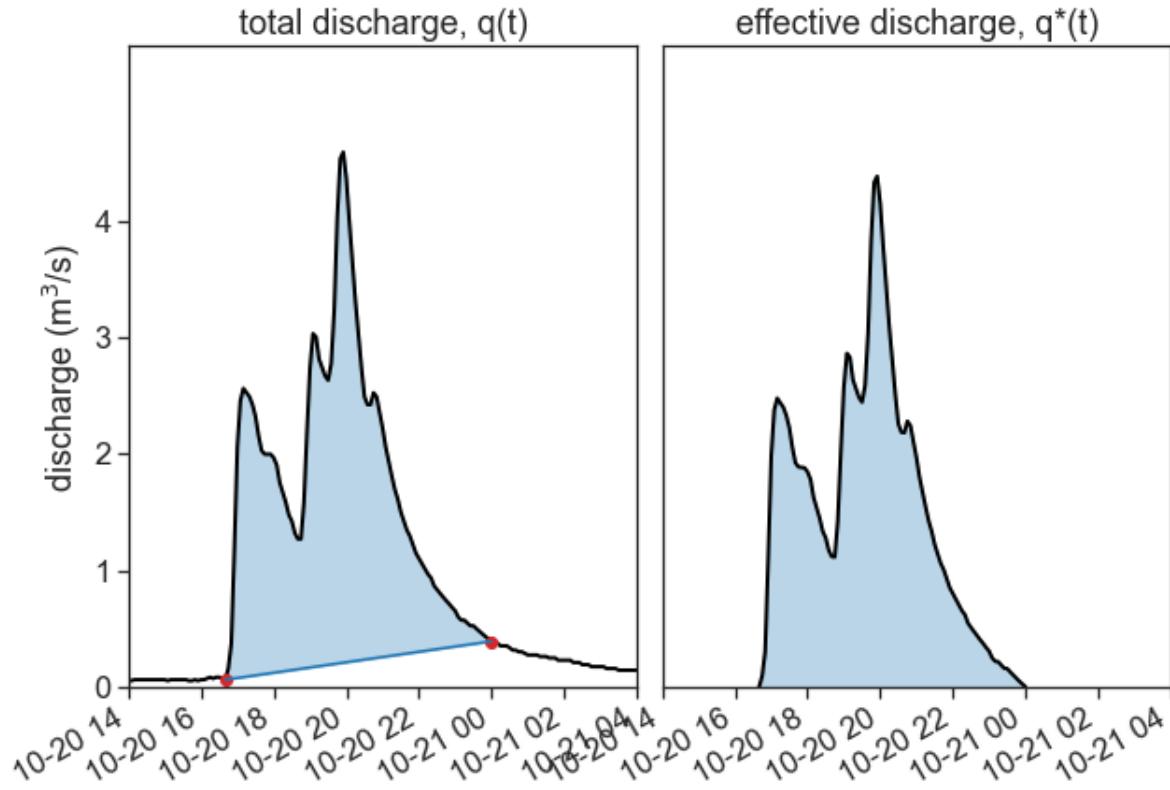
qstar = df_q.loc[df_linear.index]['discharge'] - df_linear['discharge']
Qstar = qstar.sum() * 60 * 5

ax2.plot(qstar, color="black", lw=2)
ax2.fill_between(qstar.index, qstar,
                  y2=0.0,
                  color="tab:blue", alpha=0.3)

ax1.set(xlim=[df_q.index[0],
               df_q.index[-1]],
        ylabel=r"discharge (m$^3$/s)",
        ylim=[0, 5.5],
        yticks=[0,1,2,3,4],
        title="total discharge, q(t)")
ax2.set(yticks=[],
        ylim=[0, 5.5],
        xlim=[df_q.index[0],
               df_q.index[-1]],
        title="effective discharge, q*(t)"
        )

plt.gcf().autofmt_xdate() # makes slanted dates

```



We can calculate p^* now, using

$$P^* = Q^*$$

One of the simplest methods is to multiply $p(t)$ by a fixed constant (<1) to obtain p^* , so that the equation above holds true.

```
ratio = Qstar/ P
pstar = df_p['PRECIPITATION'] * ratio
Pstar = pstar.sum() * 5 * 60
print(f"Qstar / P = {ratio:.2f}")
```

$Qstar / P = 0.16$

Calculate now the centroid (t_{pc}) for effective precipitation p^* and centroid (t_{qc}) of effective discharge q^* . Calculate also the time of peak discharge (t_{pk}). Then, calculate the centroid lag (T_{LC}), the centroid lag-to-peak (T_{LPC}), and the time of concentration (T_c). Use the equations below:

$$T_{LPC} \simeq 0.60 \cdot T_c$$

Time of precipitation centroid:

$$t_{pc} = \frac{\sum_{i=1}^n p_i^* \cdot t_i}{P^*}$$

Time of streamflow centroid:

$$t_{qc} = \frac{\sum_{i=1}^n q_i^* \cdot t_i}{Q^*}$$

Centroid lag:

$$T_{LC} = t_{qc} - t_{pc}$$

Centroid lag-to-peak:

$$T_{LPC} = t_{pk} - t_{pc}$$

Time of concentration:

$$T_{LPC} \simeq 0.60 \cdot T_c$$

```
# pstar centroid
# time of the first (nonzero) rainfall data point
t0 = pstar[pstar != 0.0].index[0]
# time of the last (nonzero) rainfall data point
tf = pstar[pstar != 0.0].index[-1]
# duration of the rainfall event, in minutes
td = (tf-t0) / pd.Timedelta('1 min')
# make time array, add 2.5 minutes (half of dt)
time = np.arange(0, td+1, 5) + 2.5
# create pi array, only with relevant data (during rainfall duration)
pi = pstar.loc[(pstar.index >= t0) & (pstar.index <= tf)]
# convert from m3/5min to m3/s
pi = pi.values * 60 * 5
# time of precipitation centroid
t_pc = (pi * time).sum() / pi.sum()
# add initial time
t_pc = t0 + pd.Timedelta(minutes=t_pc)
t_pc
```

```

# qstar centroid
# time of the first (nonzero) discharge data point
t0 = qstar[qstar != 0.0].index[0]
# time of the last (nonzero) discharge data point
tf = qstar[pstar != 0.0].index[-1]
# duration of the discharge event, in minutes
td = (tf-t0) / pd.Timedelta('1 min')
# make time array, add 2.5 minutes (half of dt)
time = np.arange(0, td+1, 5) + 2.5
# create qi array, only with relevant data (during discharge duration)
qi = qstar.loc[(qstar.index >= t0) & (qstar.index <= tf)]
# convert from m3/5min to m3/s
qi = qi.values * 60 * 5
# time of discharge centroid
t_qc = (qi * time).sum() / qi.sum()
# add initial time
t_qc = t0 + pd.Timedelta(minutes=t_qc)
t_qc

# time of peak discharge
max_discharge = qstar.max()
t_pk = qstar[qstar == max_discharge].index[0]

# centroid lag
T_LC = t_qc - t_pk

# centroid lag-to-peak
T_LPC = t_pk - t_pc

# time of concentration
T_c = T_LPC / 0.60

print(f"T_LC = {T_LC}")
print(f"T_LPC = {T_LPC}")
print(f"T_c = {T_c}")

```

```

T_LC = 0 days 00:53:03.186594
T_LPC = 0 days 01:22:59.857820
T_c = 0 days 02:18:19.763033333

```

16 Unit Hydrograph

16.1 Linear reservoir model

Box 10.3 Linear-Reservoir Model of Watershed Response

We can develop a very simple conceptual model of the response of a watershed to effective precipitation based on: (1) the principle of conservation of mass,

$$p^* - q^* = \frac{dS^*}{dt}; \quad (10B3.1)$$

(section 1.6.2) and (2) the linear-reservoir conceptual model of watershed behavior (section 1.10.2):

$$q^* = \frac{1}{T^*} \cdot S^*, \quad (10B3.2)$$

where p^* is effective rainfall rate ($[L^3 T^{-1}]$ or $[L T^{-1}]$), q^* is event-flow rate ($[L^3 T^{-1}]$ or $[L T^{-1}]$), S^* is storage of event-flow water ($[L^3]$ or $[L]$), t is time, and T^* is a time constant that characterizes the watershed response [T]. Real watersheds are not strictly linear reservoirs, but equation (10B3.2) captures the most basic aspects of watershed response and is mathematically tractable.

Combining equations (10B3.1) and (10B3.2) yields

$$p^* - q^* = T^* \cdot \frac{dq^*}{dt}, \quad (10B3.3)$$

which for constant p^* has the solution

$$q^* = p^* + (q_0^* - p^*) \cdot \exp(-t/T^*), \quad (10B3.4)$$

where q_0^* is the outflow at $t = 0$.

To model an isolated hydrograph rise in response to a constant input beginning at $t = 0$ and lasting until time T_p , we can set $q_0^* = 0$ at $t = 0$ so that equation (10B3.4) becomes

$$q^* = p^* \cdot [1 - \exp(-t/T^*)], t \leq T_p \quad (10B3.5)$$

The peak discharge, q_{pk}^* , occurs when water input ceases, i.e., when $t = T_p$; q_{pk}^* is then given by

$$q_{pk}^* = p^* \cdot [1 - \exp(-T_p/T^*)]. \quad (10B3.6)$$

When input ceases the hydrograph recession begins and we have new conditions: $p^* = 0$ and the "initial" discharge is q_{pk}^* at time $t = T_p$. Thus for the recession equation (10B3.4) becomes

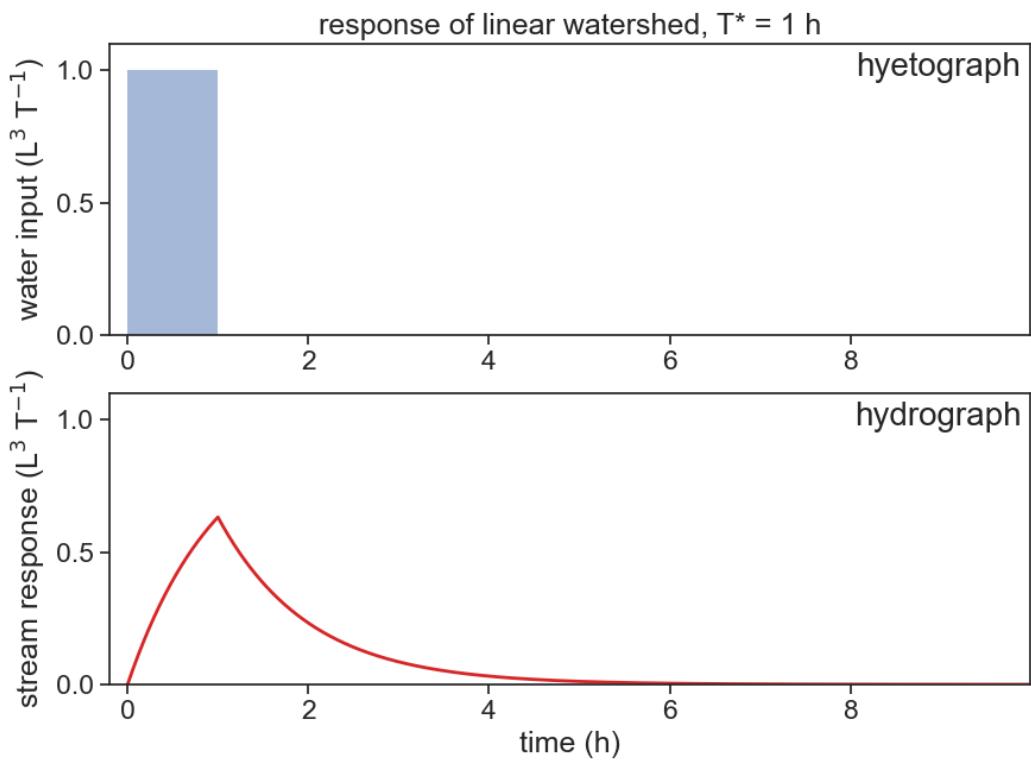
$$q^* = q_{pk}^* \cdot \exp[-(t - T_p)/T^*], t \geq T_p. \quad (10B3.7)$$

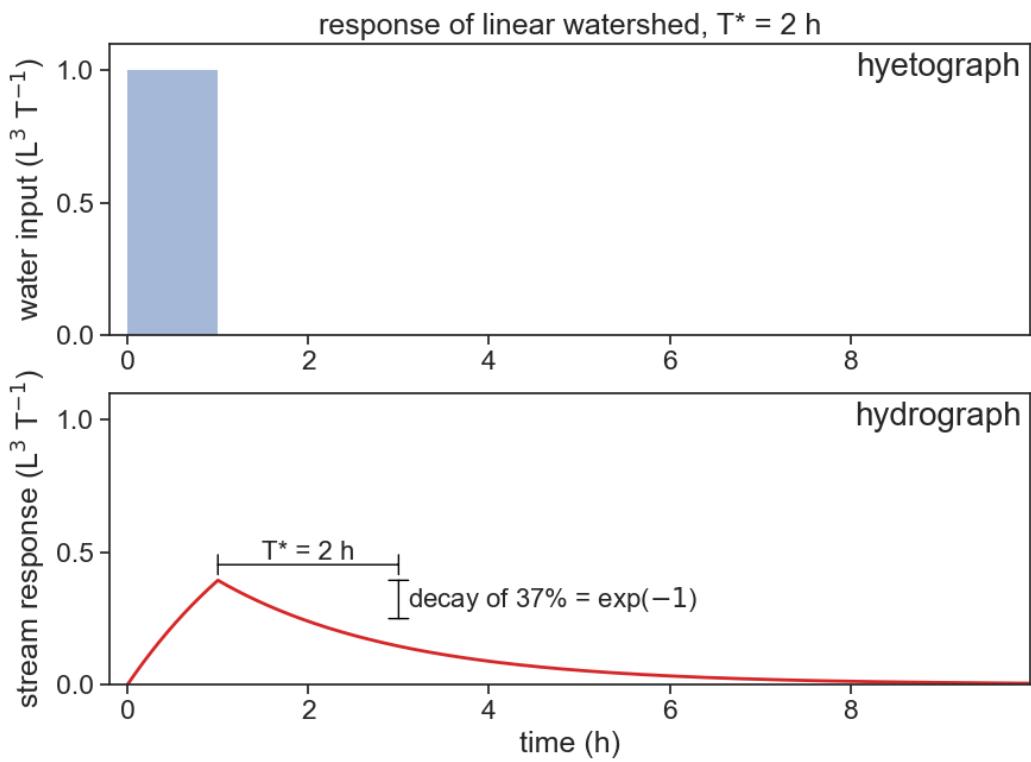
The properties of this model are:

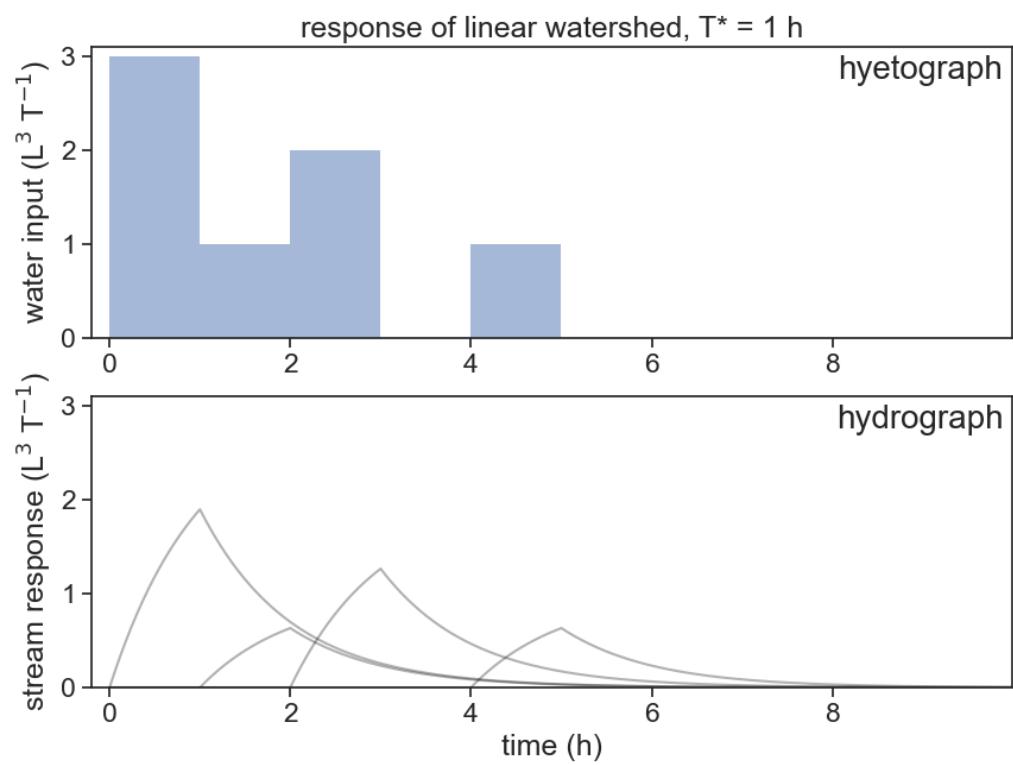
1. The model preserves continuity; i.e., the volume of event response Q^* equals the volume of effective water input, $P^* = p^* \cdot T_p$.
2. T^* can be shown to be equal to the centroid lag of the watershed (i.e., $t_{qc} - t_{pc} = T^*$).
3. The hydrograph rise begins as soon as input begins ($T_{LR} = 0$) and peaks exactly at $t_{pk} = T_p$.
4. Because outflow decreases exponentially after input ceases and approaches zero asymptotically, time of concentration is infinite. However, if we define the time to equilibrium, T_{eq} , as the time it takes for the outflow rate to reach 99% of a constant inflow rate, it can be shown that

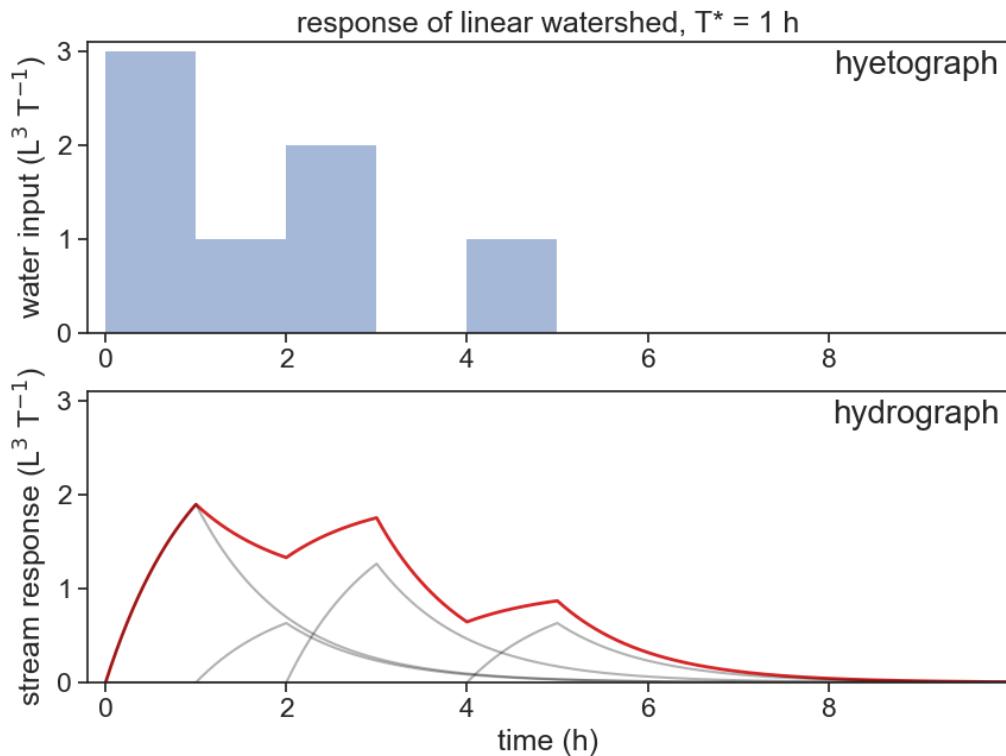
$$T_{eq} = -\ln(0.01) \cdot T^* = 4.605 \cdot T^*. \quad (10B3.8)$$

Figure 16.1: Source: Dingman (2015), page 472









16.2 Rainfall-Runoff Models

16.2.1 The Rational Method

The rational method postulates a simple proportionality between peak discharge, q_{pk} , and rainfall intensity, p^* :

$$q_{pk} = \varepsilon_R \cdot C_R \cdot A_D \cdot p^*$$

- q_{pk} : peak discharge (m^3/s)
- $\varepsilon_R = 0.278$: unit-conversion factor
- C_R : dimensionless runoff coefficient
- A_D : drainage area (km^2)
- p^* : rainfall intensity (mm/h)

Obviously the results obtained with the method are highly sensitive to the value chosen for CR; values range from 0.05 for gently sloping lawns up to 0.95 for highly

urbanized areas of roofs and pavement.

The rational method is widely used in urban drainage design, but Pilgrim and Cordery (1992) caution that there are typically few data available to guide the selection of CR, and that CR for a given watershed may vary widely from storm to storm due to differing antecedent conditions.

16.2.2 The Soil Conservation Service Curve-Number Method (SCS-CN)

Also called NRCS curve number procedure. NRCS = Natural Resources Conservation Service - USDA

$$Q^* = P^* = \frac{(P - S_I)^2}{P - S_I + S_{max}},$$

where

- Q^* : total volume of event flow in a storm event
- P^* : total volume of effective water input in a storm event
- P : total volume of water input in a storm event
- S_I : initial abstraction = amount of rainfall that is retained or absorbed by the land surface before runoff occurs
- S_{max} : potential maximum retention = maximum amount of water that the watershed can retain after runoff starts

The initial abstraction S_I is usually approximated as $0.2 \cdot S_{max}$, therefore:

$$Q^* = P^* = \frac{(P - 0.2 \cdot S_{max})^2}{P + 0.8 \cdot S_{max}}$$

$$S_{max} = 25.4 \left(\frac{1000}{CN} - 10 \right)$$

The number 25.4 is a conversion factor from inches to millimeters.

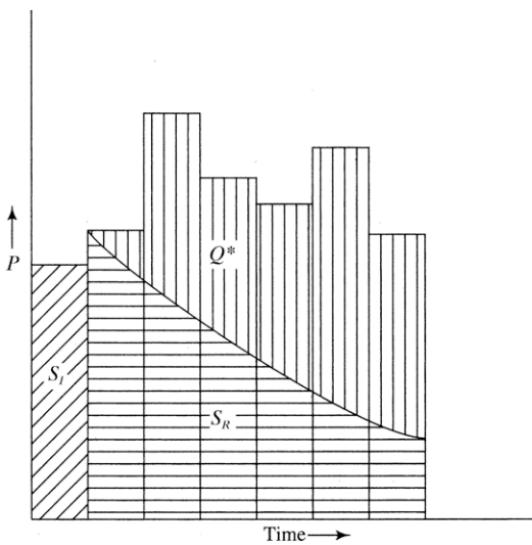


Figure 10.51 Definitions of initial abstraction, S_i , retention, S_R , and event flow, Q^* , in the SCS method.

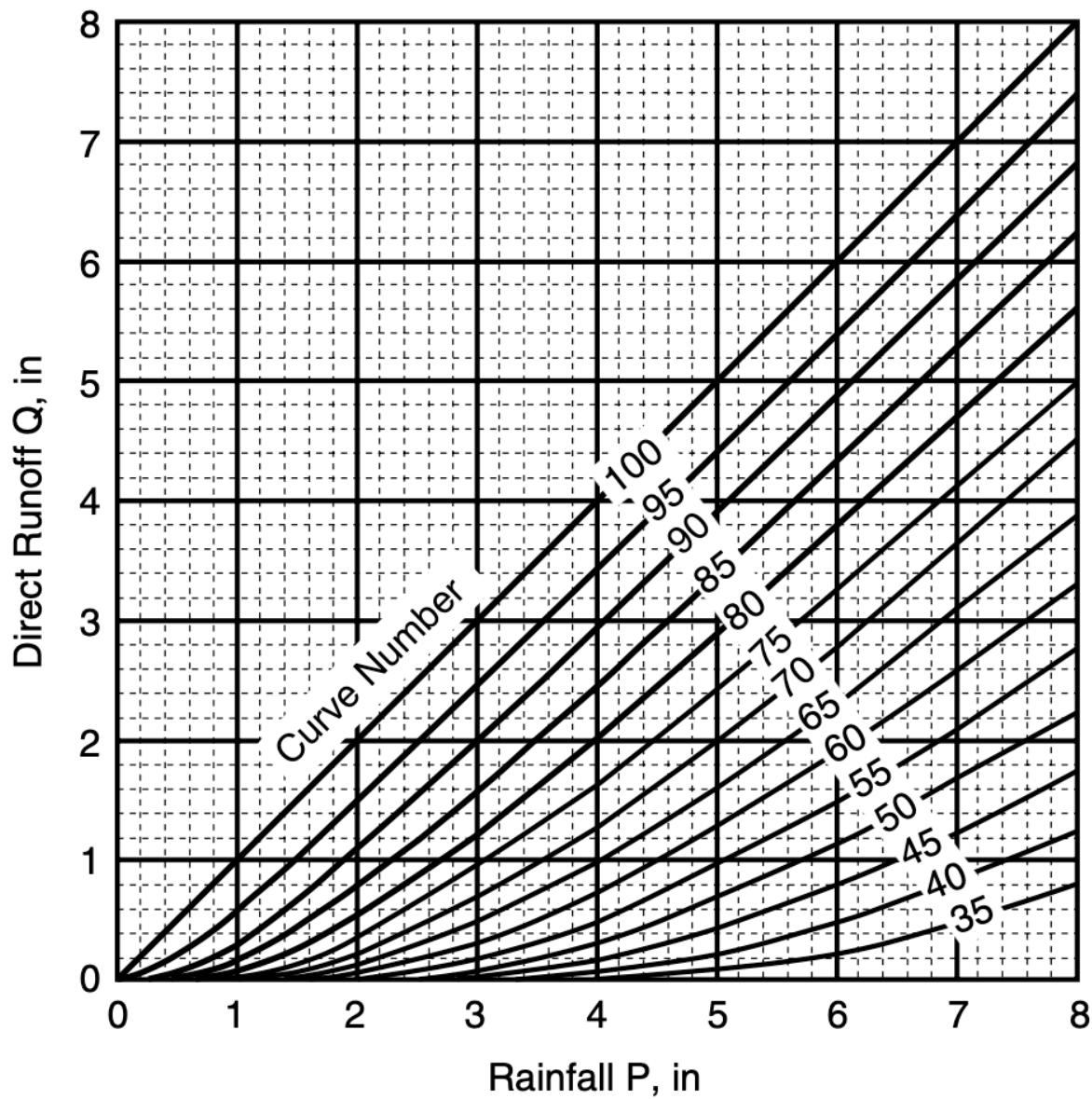


FIGURE 5.15 Relationships of runoff to rainfall based on NRCS curve number method.

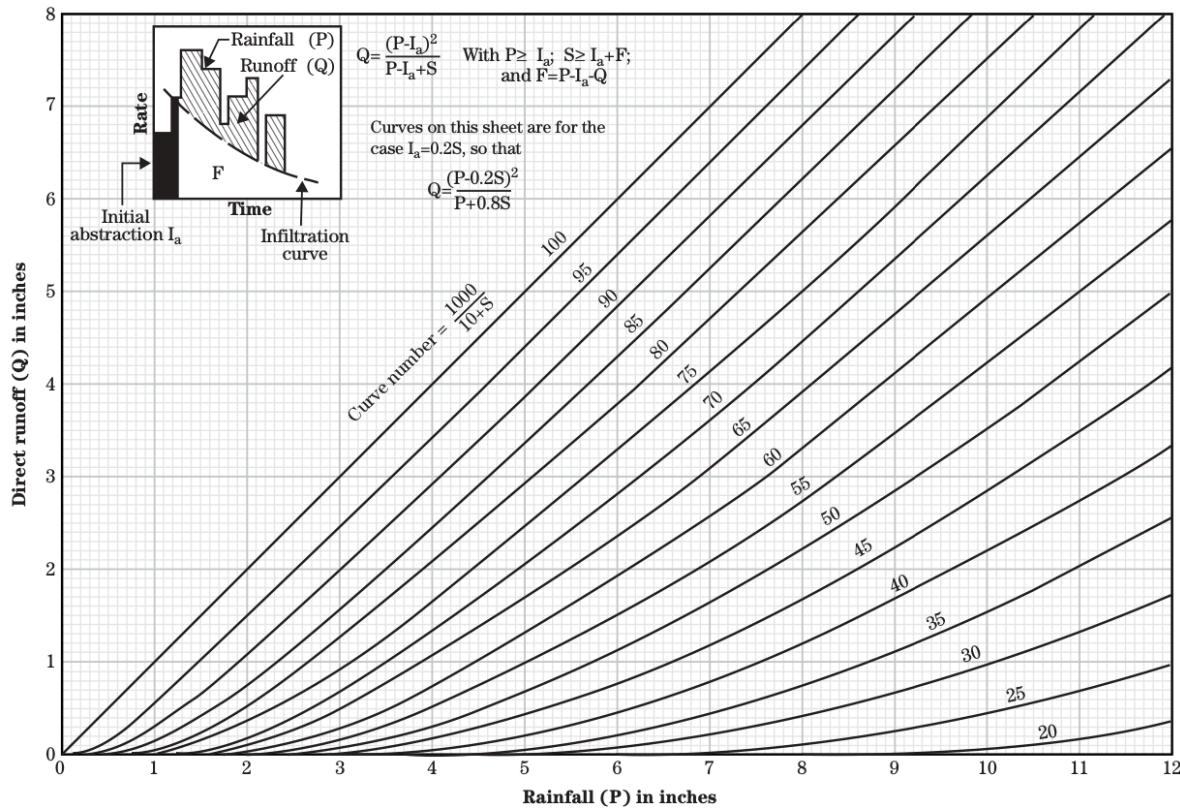


Figure 16.2: Source: United States Department of Agriculture (2004)

The curve number (CN) is a function of the ability of soils to infiltrate water, land use, and the soil water conditions at the start of a rainfall event (antecedent soil water condition). To account for the infiltration characteristics of soils, the NRCS has divided soils into four hydrologic soil groups, which are defined as follows (NRCS, 1984):

- **Group A** (low runoff potential): Soils with high infiltration rates even when thoroughly wetted. These consist chiefly of deep, well-drained sands and gravels. These soils have a high rate of water transmission (final infiltration rate greater than 0.3 in./h).
- **Group B:** Soils with moderate infiltration rates when thoroughly wetted. These consist chiefly of soils that are moderately deep to deep, moderately well drained to well drained with moderately fine to moderately coarse textures. These soils have a moderate rate of water transmission (final infiltration rate 0.15 to 0.30 in./h).
- **Group C:** Soils with slow infiltration rates when thoroughly wetted. These consist chiefly of soils with a layer that impedes downward movement of water or soils with moderately fine to fine texture. These soils have a slow rate of water transmission (final infiltration rate 0.05 to 0.15 in./h).
- **Group D** (high runoff potential): Soils with very slow infiltration rates when thoroughly

wetted. These consist chiefly of clay soils with a high swelling potential, soils with a permanent high water table, soils with a claypan or clay layer at or near the surface, and shallow soils over nearly impervious materials. These soils have a very slow rate of water transmission (final infiltration rate less than 0.05 in./h).

There are also three categories for Antecedent Soil Moisture Condition (AMC):

- **AMC I:** Dormant season antecedent soil moisture less than 0.5 in. Growing season antecedent soil moisture less than 1.4 in.
- **AMC II:** Dormant season antecedent soil moisture between 0.5 and 1.1 in. Growing season antecedent soil moisture between 1.4 and 2.1 in.
- **AMC III:** Dormant season antecedent soil moisture greater than 1.1 in. Growing season antecedent soil moisture greater than 2.1 in.

TABLE 5.1
Curve Numbers for Antecedent Soil Averages

Land Use Description

| |
|--|
| Commercial, row houses and townhouses |
| Fallow, poor condition |
| Cultivated with conventional tillage |
| Cultivated with conservation tillage |
| Lawns, poor condition |
| Lawns, good condition |
| Pasture or range, poor condition |
| Pasture or range, good condition |
| Meadow |
| Pavement and roofs |
| Woods or forest thin stand, poor cover |
| Woods or forest, good cover |
| Farmsteads |
| Residential quarter-acre lot, poor condition |
| Residential quarter-acre lot, good condition |
| Residential half-acre lot, poor condition |
| Residential half-acre lot, good condition |
| Residential 2-acre lot, poor condition |
| Residential 2-acre lot, good condition |
| Roads |

See the table below to find curve numbers for AMC II: *Source:* From NRCS, 1984.

```

P=21
ratio = 4.17e4/2.61e5
CN=86
Smax = 25.4 * (1000/CN - 10)

```

```
Pmin = 0.2 * Smax
Qstar = 0.0
if P > Pmin:
    Qstar = (P - 0.2*Smax)**2 / (P+0.8*Smax)
Qstar/P
```

0.14270006393832066

```
ratio
```

0.15977011494252874

```
Qstar / P
```

0.9148811393863234

```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt

def Qstar_f(pe, CN):
#    Smax = 25.4*(1000/CN - 10)
    Smax = (1000/CN - 10)
#    Smax = (1000/CN - 10) / 25.4
    Qstar = (pe - 0.2*Smax)**2 / (pe+0.8*Smax)
    return Qstar

pe = np.linspace(0,8,101)
# plt.plot(pe, Qstar_f(pe, 35))
plt.plot(pe, Qstar_f(pe, 50))
# plt.plot(pe, Qstar_f(pe, 85))
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Part VI

summing up

17 Budyko framework

Sources used:

Daly et al. (2019), Sposito (2017), Jones et al. (2012), Krajewski et al. (2021), Berghuijs, Gnann, and Woods (2020), Creed and Spargo (2012)

17.1 Water and surface energy balances

For long-term averages:

$$P = ET + Q$$

$$R_n = \lambda_w \cdot ET + H$$

- P : precipitation (L T^{-1} , e.g.: mm/day)
- ET : evapotranspiration (L T^{-1})
- Q : streamflow (L T^{-1})
- R_n : net energy available at soil surface (M T^{-3} , e.g.: W m^{-2})
- λ_w : latent heat of vaporization of water ($\text{M L}^{-1}\text{T}^{-2}$, as defined here, the units will be weird)
- H : sensible heat flux from the surface into the atmosphere (M T^{-3})
- $\lambda_w \cdot ET$: latent heat flux (M T^{-3})

17.2 Assumptions

1. because we are dealing with long-term averages, there are negligible changes of watershed stored water.
2. negligible energy is stored at the soil surface, and heat transfer from soil surface to deeper soil layers (G) averages zero.

17.3 Question

Given measurements of rainfall and meteorological conditions, can we predict the partitioning of P between ET and Q ?

17.4 Limits

For very dry watersheds (deserts, for example), almost all precipitation (P) is lost via evapotranspiration (ET). These watersheds are called water limited.

In wet watersheds, at the annual scale, the sensible heat (H) is directed from the surface to the atmosphere in almost all climatic zones on Earth (meaning: soil heats air). Therefore, H cannot supply much energy to the soil surface, and it is assumed that R_n provides entirely the energy required for evapotranspiration. Dividing the second equation by λ_w , we get $R_n/\lambda_w = ET + H/\lambda_w$. It is clear that the maximum possible ET occurs when all incoming radiation energy R_n is consumed by evapotranspiration ET , and there is negligible sensible heat flux H . As a result, the upper limit of $\lambda_w E$ is R_n , in wet watersheds. In these watersheds, called energy limited, ET tends to the potential evapotranspiration (ET_0).

17.4.1 Summary:

For energy-limited watersheds

- (1) As precipitation $P \rightarrow \infty$, evapotranspiration $ET \rightarrow ET_0$

For water-limited watersheds

- (2) As potential evapotranspiration $ET_0 \rightarrow \infty$, actual evaporation $ET \rightarrow P$

In general, we can write

$$ET = f(P, ET_0)$$

The variables P and ET have the same dimensions ($L T^{-1}$), and we can divide the equation above by P :

$$\frac{ET}{P} = f(D_I),$$

where

$$D_I = \frac{ET_0}{P}$$

is called the **dryness index**. A useful classification is

| Dryness Index | Classification |
|------------------|----------------|
| $D_I < 1.54$ | Humid |
| $1.54 < D_I < 2$ | Dry Subhumid |
| $2 < D_I < 5$ | Semi-arid |
| $5 < D_I < 20$ | Arid |
| $20 < D_I$ | Hyper-arid |

ATTENTION. The dryness index can also be called the “Aridity Index” (AI), **however** sometimes the AI means the inverse of D_I :

$$AI = 1/D_I$$

Be careful to check the definitions.

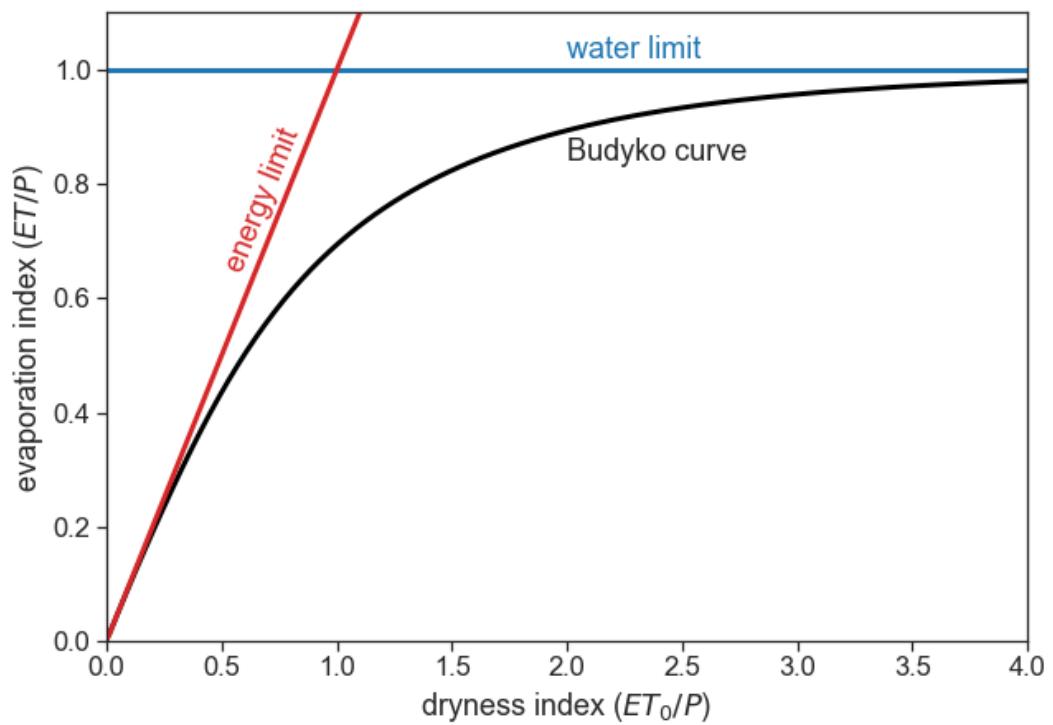
The summary (1) and (2) above can be now represented as:

$$(1) \text{ As } D_I \rightarrow 0, \frac{ET}{P} \rightarrow D_I$$

$$(2) \text{ As } D_I \rightarrow \infty, \frac{ET}{P} \rightarrow 1$$

Budyko (1974), proposed the following equation:

$$\frac{ET}{P} = \left[D_I \tanh \left(\frac{1}{D_I} \right) (1 - e^{-D_I}) \right]^{1/2}$$



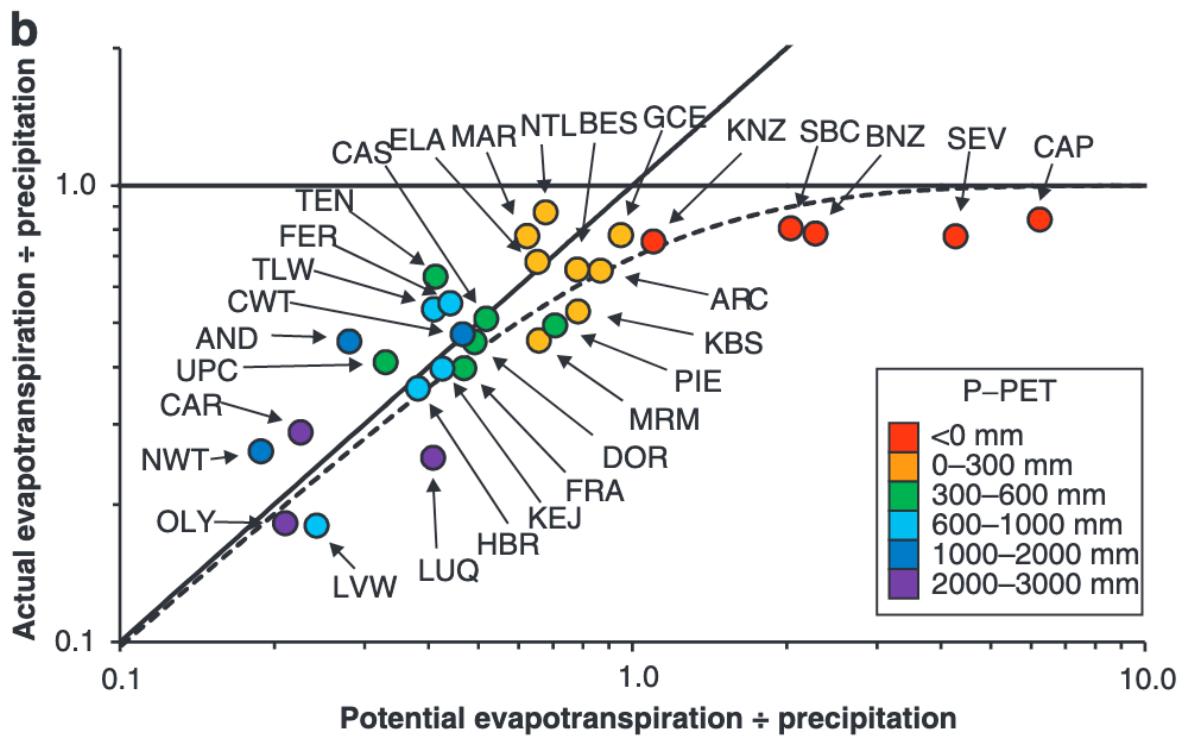


Figure 17.1: Source: Jones et al. (2012)

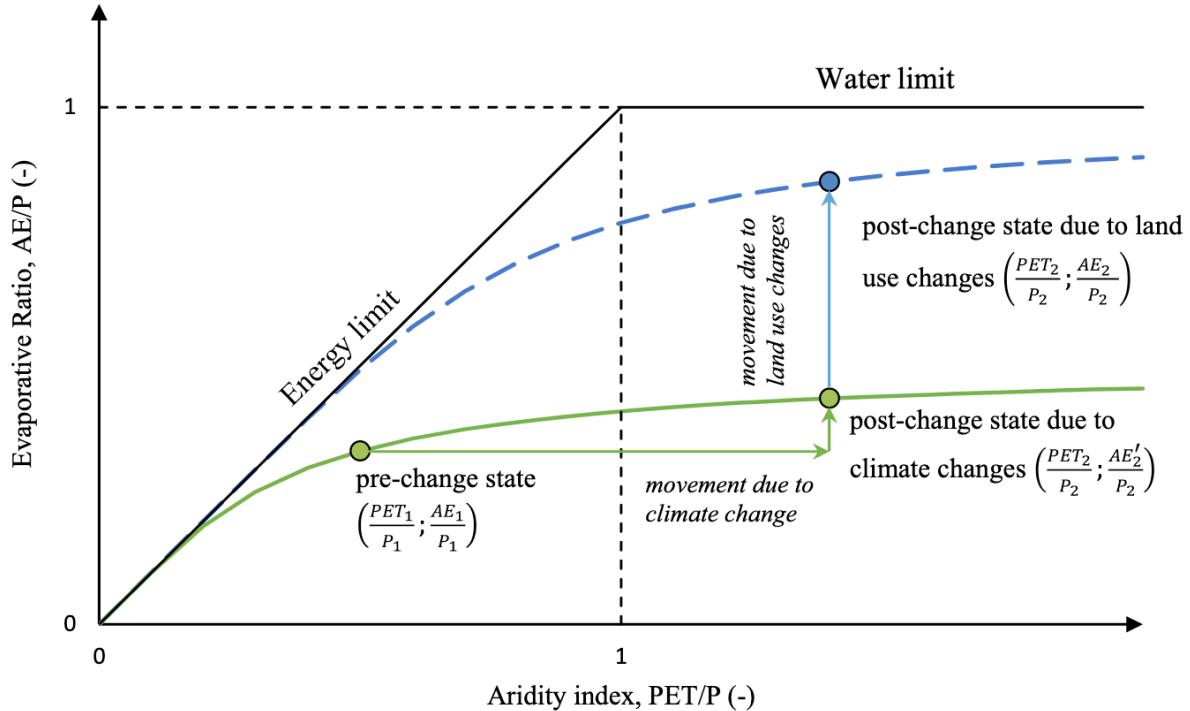


Figure 17.2: Source: Krajewski et al. (2021)

There are many alternatives to Budyko's equation. Many equations have adjustable parameters, such as Fu's equation:

$$\frac{ET}{P} = 1 + D_I - (1 + D_I^w)^{1/w},$$

where $w > 1$. Each catchment has its own specific parameter w , that may represent biophysical/landscape features. There is no consensus regarding the interpretation of w , ranging from an effective empirical parameter, whose relationship to biophysical features can be discerned, to an arbitrary empirical constant with no *a priori* physical meaning. Source: {cite reaver2020reinterpreting %}

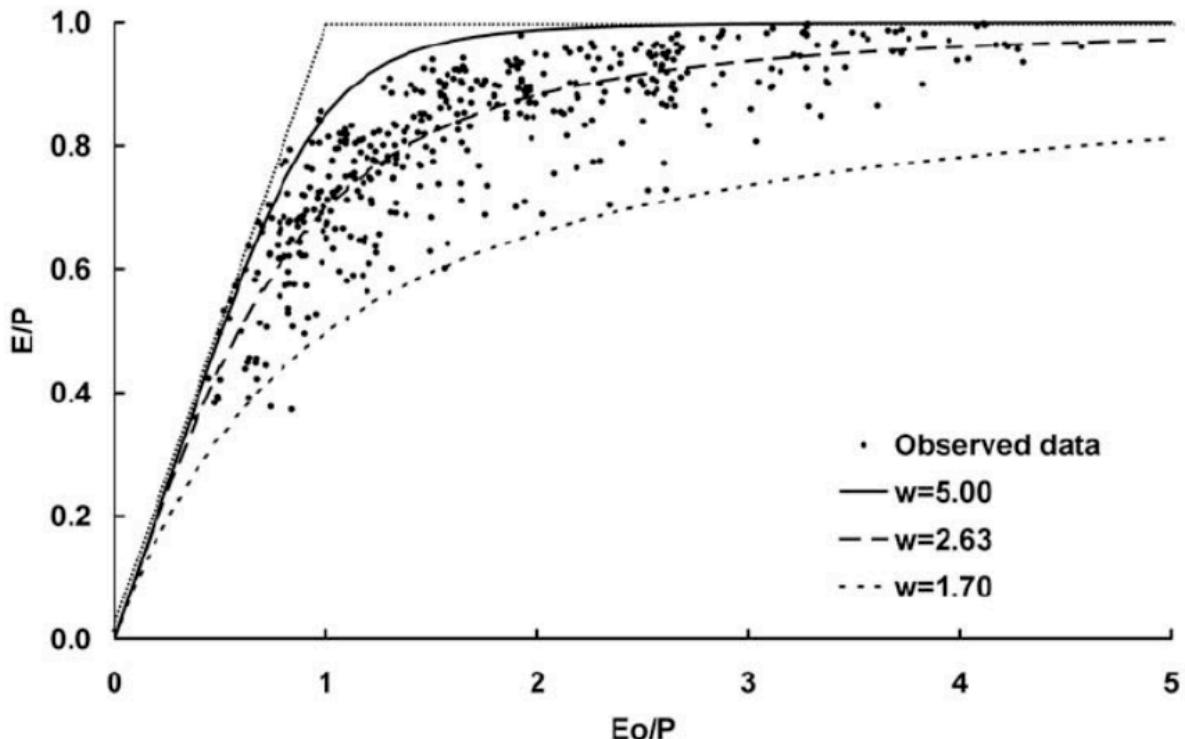


Figure 17.3: Source: Zhang et al. (2004)

17.5 Hypotheses for why dryness index controls so much the partitioning of P into ET and Q

Source: Berghuijs, Gnann, and Woods (2020)

1. The first is that the Budyko curve is accurate because landscape features (e.g., soils and vegetation) coevolve with the local climate in such a manner that precipitation partitioning into streamflow and evapotranspiration converges towards the Budyko curve
2. A second hypothesis is that catchments over time evolve towards the supply and demand limits (rather than towards a curve), because landscapes and their vegetation are unaware of the Budyko curve but do evolve to maximize their use of available resources (including water). However, because limiting factors such as climatic variability exist (which will reduce a catchment's ability to use all water because it cannot fully buffer the highly variable precipitation input), catchments will tend to not reach these limits. This may lead to an (apparent) existence of the Budyko curve which falls relatively close to the demand and supply limits.
3. A third hypothesis is that the existence of a strong universal relationship between aridity and catchment water balances might be explained by an underlying organizing principle

- such as maximum entropy production because the Budyko curve may be consistent with how hydrologic systems optimally partition water and energy
4. A fourth hypothesis is that virtually any landscape and climate combination (also those in heavily disturbed landscapes: e.g., a city, agricultural lands, etc.) will fall near the Budyko curve because climate aridity will dominate precipitation partitioning largely independent of the climate-landscape configuration or any optimization principle.

17.6 Hypotheses for deviations from Budyko curve

Source: Creed and Spargo (2012)

1. Under stationary conditions (naturally occurring oscillations), catchments will fall on the Budyko Curve
2. Under non-stationary conditions (anthropogenic climate change), catchments will deviate from the Budyko Curve in a predictable manner

17.6.1 Reasons for falling off the Budyko Curve

1. Inadequate representation of P and T (Loch Vale)
2. Inadequate representation of ET (Andrews)
3. Inadequate representation of Q (Marcell)
4. Forest conversion (Coweta)
5. Forest disturbance (Luquillo)

17.6.2 Critique

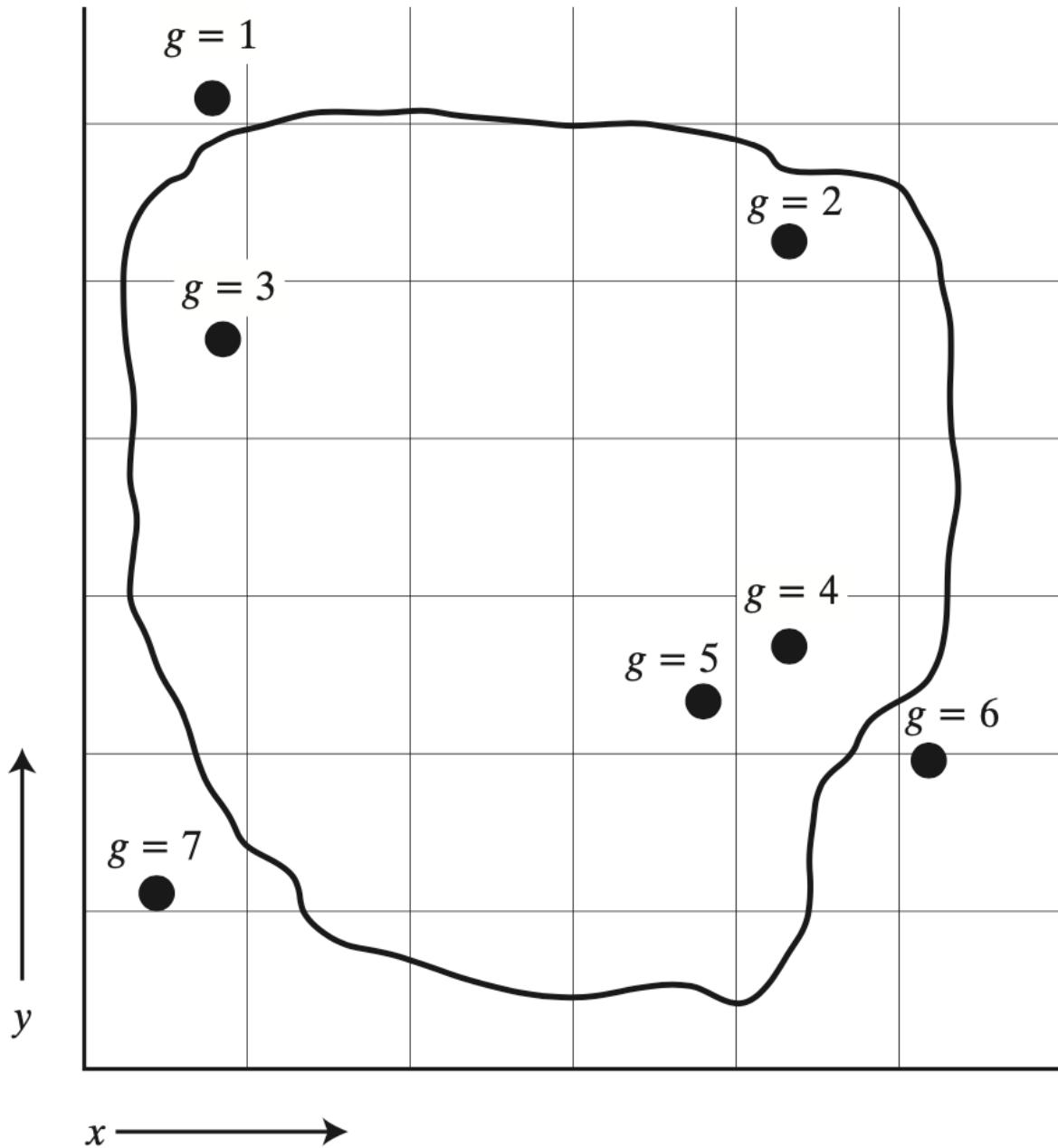
Source: Berghuijs, Gnann, and Woods (2020)

The (mathematical) specifics of such studies vary, but all approaches are founded on the assumption that catchments follow a (parametric) Budyko curve when aridity changes, and that consequently all other movements in the Budyko space are caused by other factors. The validity of this assumption remains mostly untested, which seems surprising given it underpins all of these studies' findings.

18 Spatial distribution - lecture

18.1 The problem

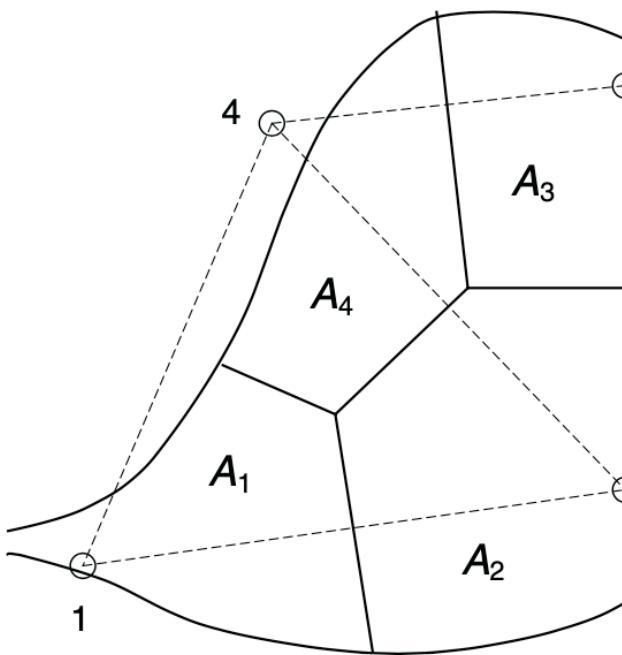
Let's say we want to calculate the average rainfall on a watershed, and we have data available for 7 stations, as shown in the figure below [Dingman, figure 4.26]:



There are a number of methods for calculating the average precipitation.

18.2 Thiessen method [Voronoi diagram]

Fig. 3.11 Sketch illustrating the application of the **Thiessen** polygon method to estimate the subareas A_i assigned to the precipitation gages on the map of a catchment. The subareas are bounded by the boundaries of the catchment and by the lines drawn midway between the stations. The locations of the stations are indicated by the numbered circles.



Brutsaert (2005), Figure 3.11

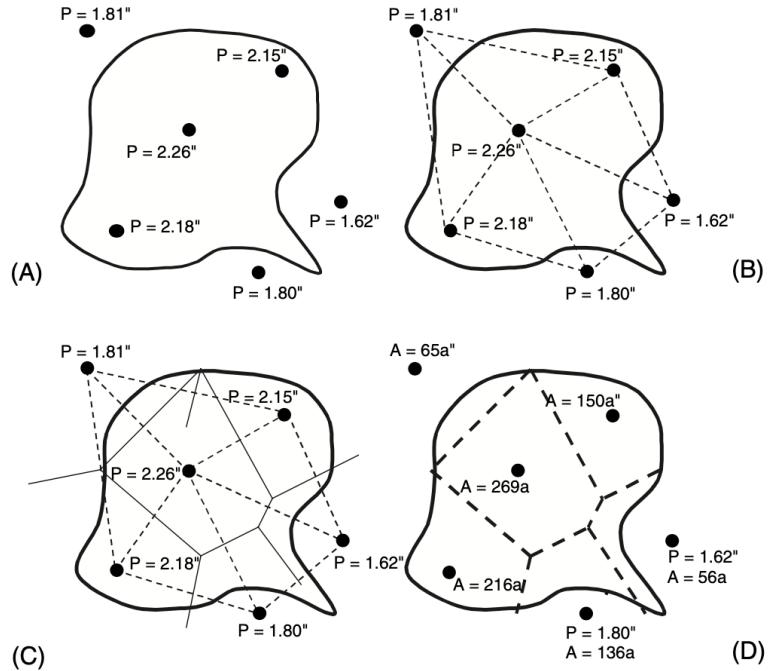


FIGURE 2.15 Use of Thiessen method to find average rainfall: (A) distribution of rain gages in a watershed located in a irregular shape; (B) connection lines drawn between rain gage positions; (C) lines perpendicular to connection lines drawn until they form polygons; and (D) areas measured for each polygon.

How to compute the areas:

Average areal precipitation is a weighted sum:

$$\langle P \rangle = \frac{\sum_i A_i P_i}{\sum_i A_i}$$

A nice way to understand the Thiessen method is depicted in this [gif from Wikipedia](#).

18.3 Inverse distance method

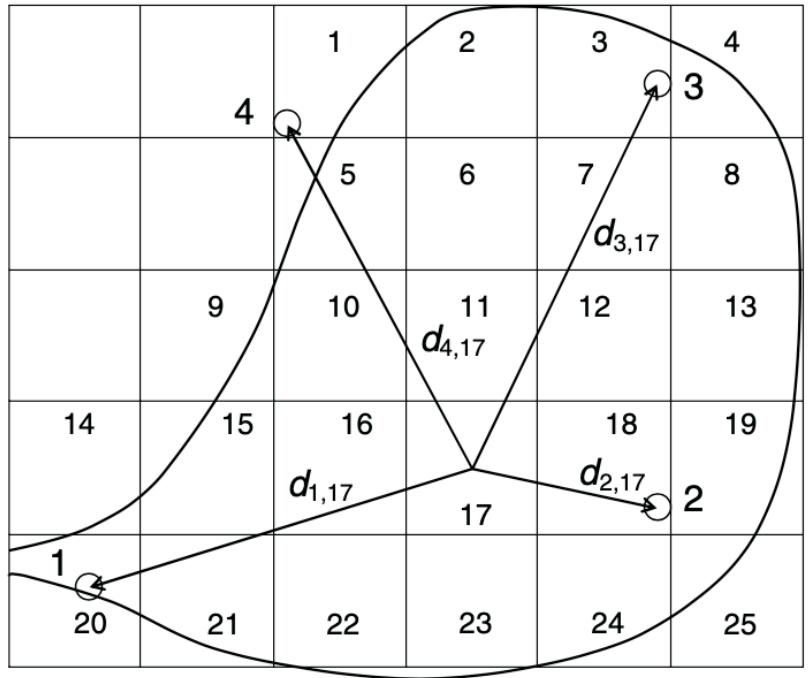


Fig. 3.12 Sketch illustrating the application of the inverse distance method. For example, the precipitation in subarea 17 is determined as $\sum_{i=1}^4 d_{i,17}^{-2} P_i / \sum_{i=1}^4 d_{i,17}^{-2}$. The average precipitation over the catchment is then obtained as the weighted average of all subarea values, as shown in Eq. (3.1). The locations of the stations are indicated by the numbered circles.

Brutsaert (2005), Figure 3.12

The precipitation for square 17 is

$$P_{17} = \frac{\sum_{i=1, \text{all stations}}^4 \frac{P_i}{d_{i,17}^2}}{\sum_{i=1, \text{all stations}}^4 \frac{1}{d_{i,17}^2}}$$

The average precipitation for the whole watershed is the weighted average of all squares, where the weight is their area:

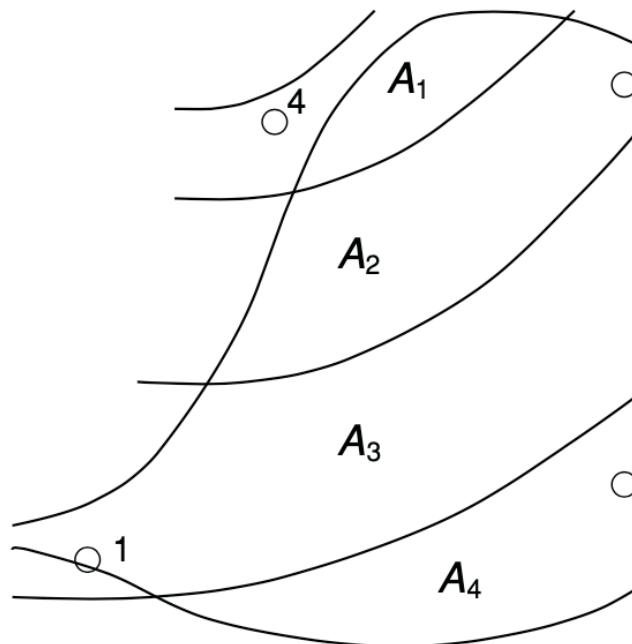
$$\langle P \rangle = \frac{\sum_{j=1, \text{all squares}} A_j P_j}{\sum_{j=1, \text{all squares}} A_j}$$

Brutsaert (2005), page 93:

Dean and Snyder (1977) found that the exponent (for the distance d^{-b}) $b = 2$ yielded the best results in the Piedmont region of the southeastern United States, whereas Simanton and Osborn (1980) concluded from measurements in Arizona that b can range between 1 and 3 without significantly affecting the results.

18.4 Isohyetal method

Fig. 3.13 Sketch illustrating the application of the isohyetal method to estimate the average precipitation over a catchment. The subareas A_i are bounded by the isohyets and by the boundaries of the catchment. The locations of the stations are indicated by the numbered circles.



Brutsaert (2005), Figure 3.12

The same equation of the Thiessen method can be used:

$$\langle P \rangle = \frac{\sum_i A_i P_i}{\sum_i A_i}$$

18.5 How it is actually done

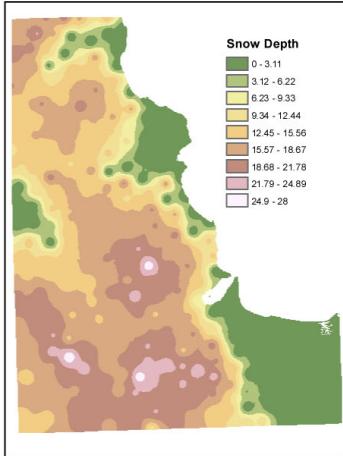
Most often, Geographic Information System (GIS) software is used to analyze spatial data. Two of the most used programs are ArcGIS (proprietary) and QGIS (free).

A good discussion of the different methods can be found on [Manuel Gimond's website](#), *Intro to GIS and Spatial Analysis*.

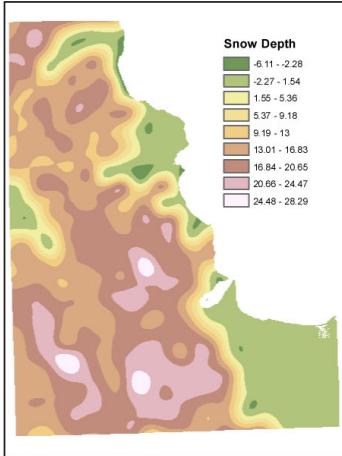
Attention!

Don't mix precision with accuracy. There are many ways of interpolating, just because a result seems detailed, it does not imply that it is accurate! See below three interpolation methods.

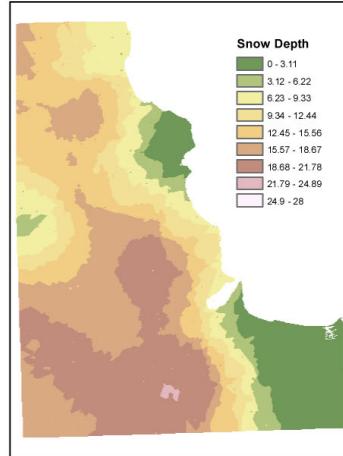
Comparison of Point Interpolation Methods at Default Settings



IDW - This method produces smooth results that still preserves detail in the high and low value locations for this data.

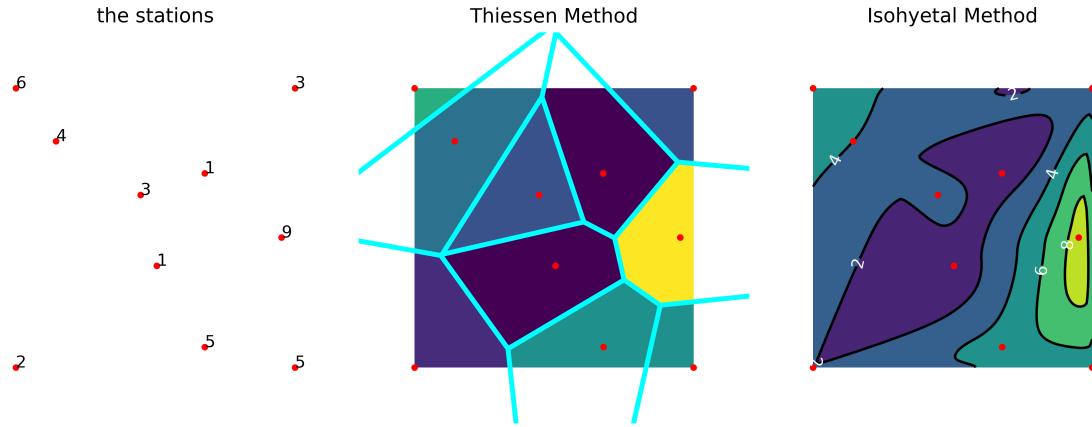


Spline - This method produces very smooth results that perhaps overgeneralize and distort the data. In particular note the presence of negative values created by the spline calculations not in the original data or possible in the real world for this data.



Kriging - This method produces rougher boundaries and several very small areas of influence for values dissimilar from neighbors.

Below you can find a simple Python code that exemplifies some of the methods, producing the following figure:



```

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import griddata
from scipy.spatial import Voronoi, voronoi_plot_2d, ConvexHull

fig, ax = plt.subplots(1, 3, figsize=(10,7))
fig.subplots_adjust(left=0.0, right=1.0, top=0.96, bottom=0.05,
                    hspace=0.02, wspace=0.02)

N = 6
PI = '3141592653589793'
points = np.random.rand(N, 2)
points = np.vstack([points, [0,0], [0,1], [1,0], [1,1]])
values = np.array([int(x) for x in list(PI)])[: (N+4)]
# values = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])

grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]

```

```

grid_z_nearest = griddata(points, values, (grid_x, grid_y), method='nearest')
grid_z_cubic = griddata(points, values, (grid_x, grid_y), method='cubic')

ax[0].plot(points[:,0], points[:,1], 'o', ms=3, markerfacecolor="red", markeredgecolor="red")
ax[0].set_aspect('equal', 'box')
ax[0].set(xlim=[0,1], ylim=[0,1])
ax[0].set_title("the stations")
for i, v in enumerate(values):
    ax[0].text(points[i,0], points[i,1], str(v))

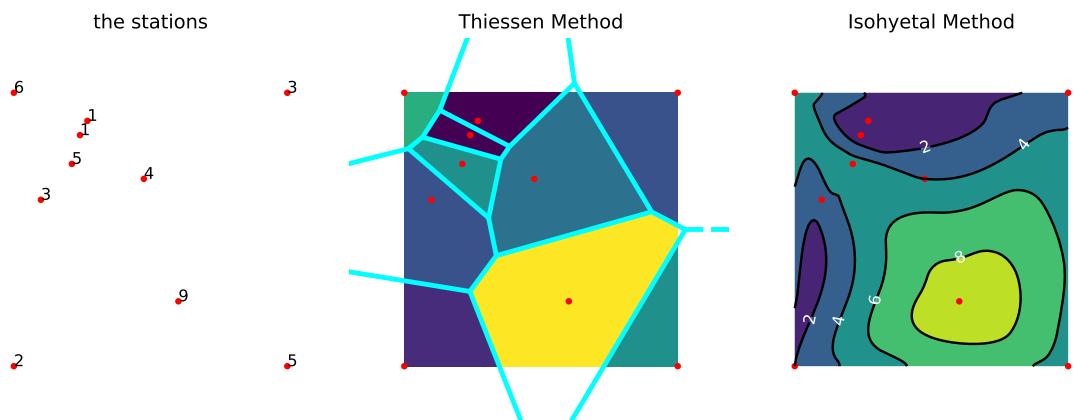
ax[1].imshow(grid_z_nearest.T, extent=(0,1,0,1), origin='lower')
ax[1].plot(points[:,0], points[:,1], 'o', ms=3, markerfacecolor="red", markeredgecolor="red")
vor = Voronoi(points)
voronoi_plot_2d(vor, show_vertices=False, line_colors='cyan',
                 line_width=3, line_alpha=1, point_size=0, ax=ax[1])
ax[1].set_title("Thiessen Method")

ax[2].plot(points[:,0], points[:,1], 'o', ms=3, markerfacecolor="red", markeredgecolor="red")
nlines = int((values.max()-values.min()+1)/2)
ax[2].contourf(grid_x, grid_y, grid_z_cubic, nlines)
cont = ax[2].contour(grid_x, grid_y, grid_z_cubic, nlines, colors="black")
ax[2].clabel(cont, inline=1, colors='white', fmt='%.0f')
ax[2].set_title("Isohyetal Method")

for i, a in enumerate(ax):
    a.set(xlim=[-0.2,1.2], ylim=[-0.2,1.2])
    a.axis('off')
    a.set_aspect('equal', 'box')

fig.savefig("spatial-distribution.png", dpi=500)

```



Part VII

Assignments

The guidelines below are valid for all the assignments.

Presentation

All the assignment must be in **one single** Jupyter Notebook. Use markdown cells to discuss the analysis and results, and in code cells show **all the code** you used to produce the figures and data analysis. Leave only the code necessary for your analysis, delete unnecessary lines you wrote while analyzing your data. Don't forget to **comment your code**, just like we did during exercise sessions. The assignment will be **written in English**.

Evaluation

All your assignments will be evaluated according to the following criteria:

- 40% Presentation. How the graphs look, labels, general organization, markdown, clean code.
- 30% Discussion. This is where you explain what you did, what you found out, etc.
- 15% Depth of analysis. You can analyze/explore the data with different levels of complexity, this is where we take that into consideration.
- 10% Replicability: Your code runs flawlessly.
- 5%: Code commenting. Explain in your code what you are doing, this is good for everyone, especially for yourself!

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, Lex, or whatever else springs up since the time of this writing. You should familiarize yourself with the AI's capabilities and limitations.

At a minimum, you should use AI to check the quality of your English text, and make the text pleasant to read.

Consider the following important points:

- Ultimately, **you**, the student, are responsible for the assignment.
- Don't trust anything the LLM says. Assume everything is wrong unless you either know the answer or can check with another source. You will be responsible for any errors or omissions provided by the tool.

- You can use LLMs to help you write both the text and the code. If you provide minimum effort prompts to the model, you will probably get low quality results. You will need to refine your prompts in order to get good outcomes. Practice a lot.
- Acknowledge the use of AI in your assignment. Be transparent about your use of the tool and the extent of assistance it provided.

The text above was written with the assistance of ChatGPT. The content is mine, ChatGPT checked my English and suggested some improvements.

19 Assignment - first graphs

This assignment relates to the content learned in the [introductory exercise lecture](#).

Go back to the weather station website, download one year of data from 01.01.2020 to 31.12.2020 (24h data). If you can't download the data, just click here.

19.1 graph 1

Make one graph with the following:

- daily `tmax` and `tmin`
- smoothed data for `tmax` and `tmin`

In order to smooth the data with a 30 day window, use the following function:

```
df['tmin'].rolling(30, center=True).mean()
```

This means that you will take the mean of 30 days, and put the result in the center of this 30-day window.

Play with this function, see what you can do with it. What happens when you change the size of the window? Why is the smoothed data shorter than the original data? See the [documentation](#) for `rolling` to find more options.

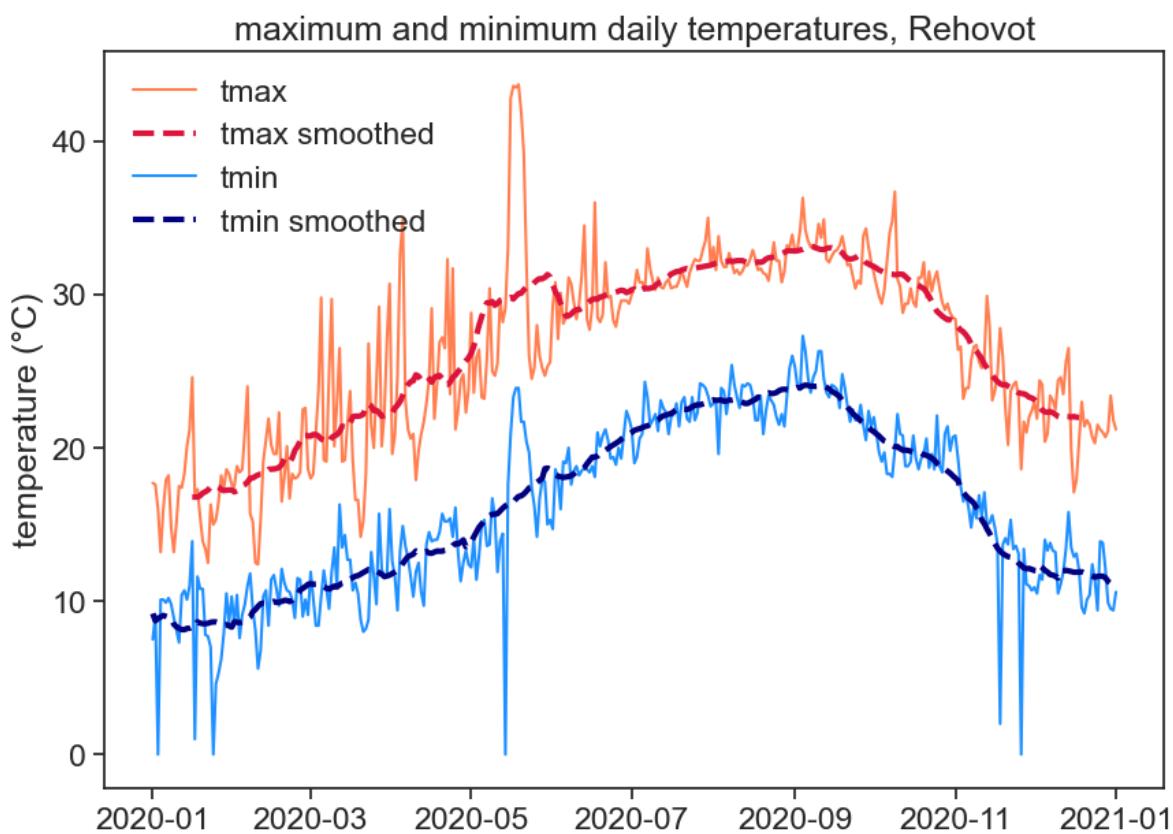
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)

fig, ax = plt.subplots(figsize=(10,7))
col_names = ['date', 'tmax', 'tmin', 'wind', 'rain24h', 'rain_cumulative']
df2 = pd.read_csv("1year.csv",
                  skiprows=5,
                  encoding='latin1',
                  names=col_names,
                  parse_dates=['date'],
                  dayfirst=True,
```

```

        index_col='date'
    )
tmin_smooth = df2['tmin'].rolling('30D', center=True).mean()
tmax_smooth = df2['tmax'].rolling(30, center=True).mean()
ax.plot(df2['tmax'], label='tmax', color="coral")
ax.plot(tmax_smooth, label='tmax smoothed', color="crimson", linestyle="--", linewidth=3)
ax.plot(df2['tmin'], label='tmin', color="dodgerblue")
ax.plot(tmin_smooth, label='tmin smoothed', color="navy", linestyle="--", linewidth=3)
ax.legend(frameon=False)
ax.set(ylabel='temperature (°C)',
       title='maximum and minimum daily temperatures, Rehovot')
)
plt.savefig("t_smoothed.png")

```



19.2 graph 2

Make another graph that focuses on a part of the year, not the whole thing. We saw before how to do that. Put on this graph two lines, representing two variables of your choosing. Give these lines good colors, and maybe different linestyles (solid, dashed, dotted) if you feel fancy. If the two variable you choose have different units, or if they have very different values (e.g. one is between 0 and 1, the other between 100 and 200), then you might find useful the following code. Here we learn how to make another yaxis.

```
# generate some sample data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x) + 10

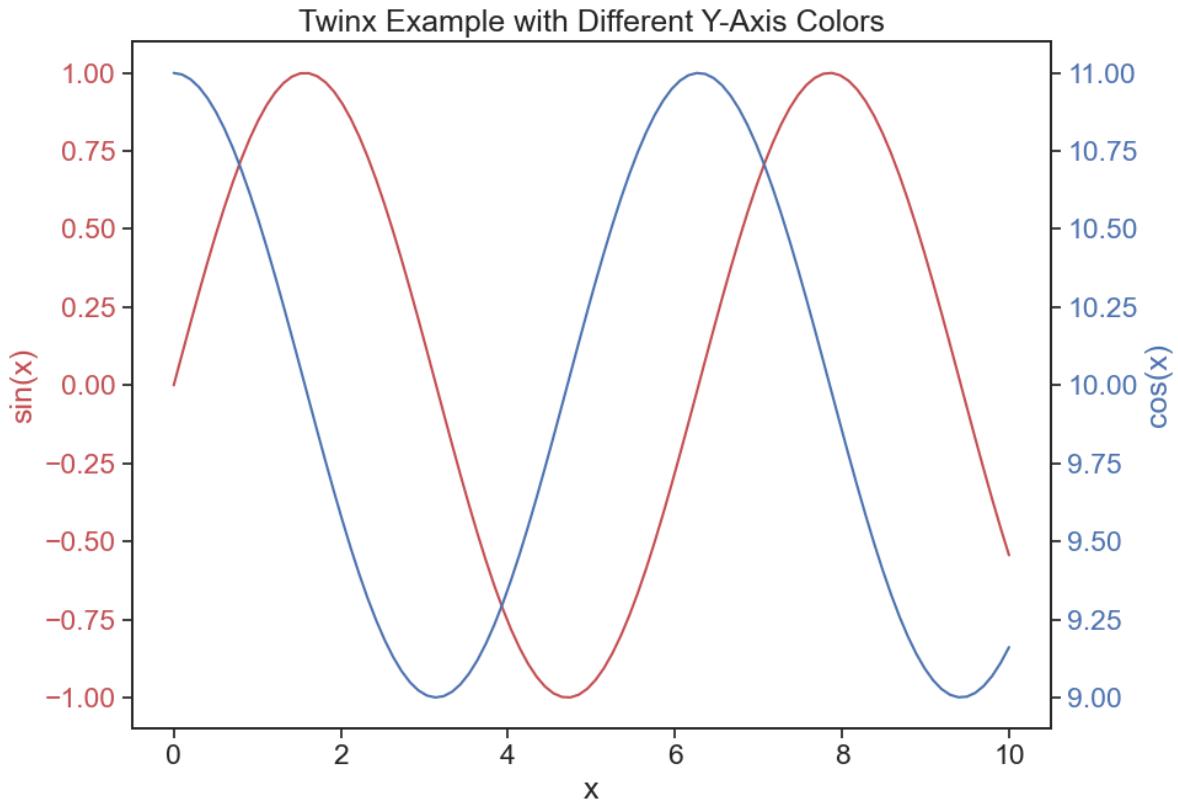
fig, ax1 = plt.subplots(figsize=(10,7))

# create the first plot
ax1.plot(x, y1, 'r-', label='sin(x)')
ax1.set_xlabel('x')
ax1.set_ylabel('sin(x)', color='r')
ax1.tick_params(axis='y', labelcolor='r')

# instantiate a second y-axis that shares the same x-axis
ax2 = ax1.twinx()

# create the second plot
ax2.plot(x, y2, 'b-', label='cos(x)')
ax2.set_ylabel('cos(x)', color='b')
ax2.tick_params(axis='y', labelcolor='b')

# show the plot
plt.title('Twinx Example with Different Y-Axis Colors')
fig.tight_layout() # to ensure the labels don't overlap
```



19.3 graph 3

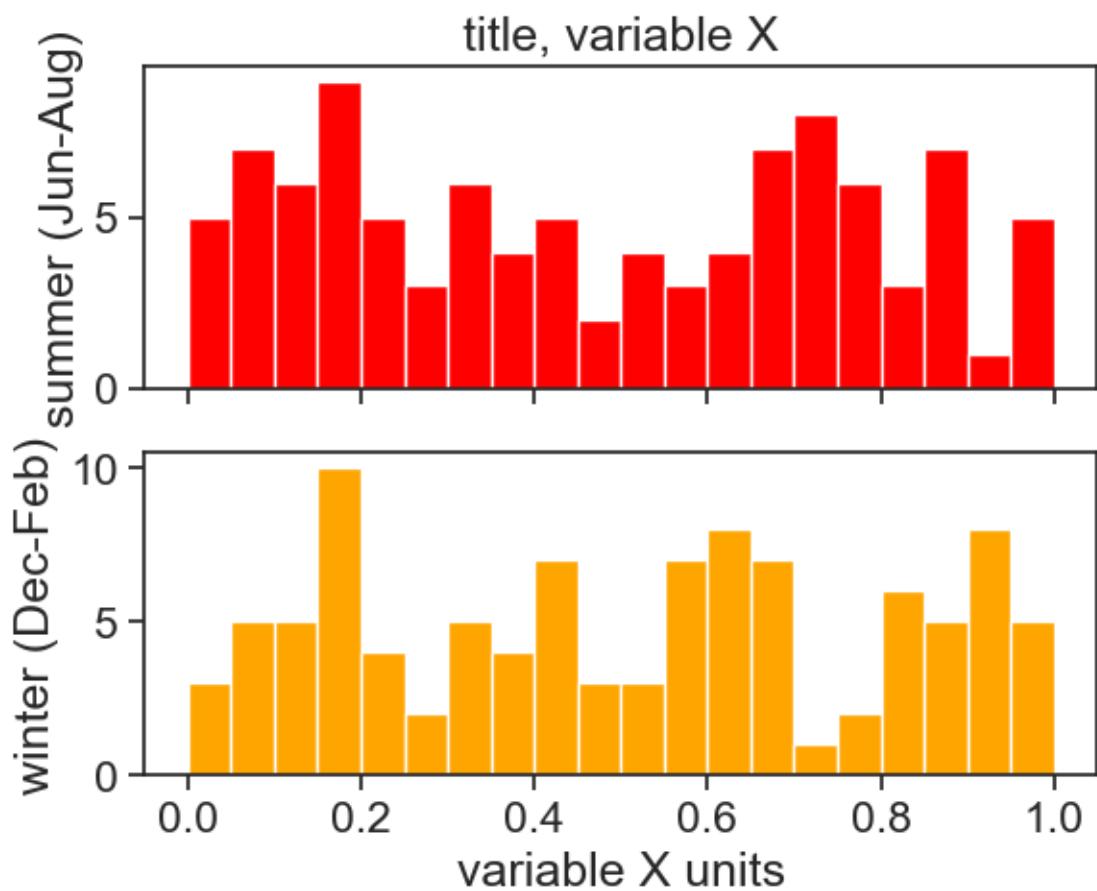
Choose another variable, and make two histograms (each in its own panel), each representing a different time interval. Here is an example how you make subplots:

```
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
N = 100

bins = np.arange(0,1.05, 0.05)

ax1.hist(np.random.random(N), bins=bins, color="red")
ax2.hist(np.random.random(N), bins=bins, color="orange")
ax1.set(ylabel='summer (Jun-Aug)',
         title='title, variable X'
     )
ax2.set(ylabel='winter (Dec-Feb)',
```

```
    xlabel='variable X units',  
);
```



20 Assignment - return period

This assignment relates to the content learned in the lecture on [return period](#).

Choose a new location from NOAA's National Centers for Environmental Information (NCEI) that we have not previously analyzed. It can be anywhere on Earth. Download daily precipitation data for a period **longer than 60 years**, or even longer if available.

In this assignment, you will analyze extreme precipitation events for the chosen location. The required analyses include:

- Show monthly precipitation averages and discuss which hydrological year definition makes the most sense.
- Show the distribution of annual daily maxima and comment on your observations.
- Fit the GEV (Generalized Extreme Value) parameters to your data. Plot both the PDF (Probability Density Function) and CDF (Cumulative Distribution Function) for your data alongside the fitted GEV functions. Does the fit appear reasonable visually?
- Show a graph of annual daily maxima against the return times. Include dots for experimental data and a line for the calculated return times based on the GEV distribution. Do they agree everywhere? Describe your observations.
- Provide a table for the precipitation values corresponding to “one in X-years precipitation events”. Choose values that make sense to you and comment on these results.

You will have two weeks to submit this assignment from the date Yair mentioned it in class.

Best of luck!

21 Assignment - ET

This assignment relates to the content learned in the lecture on [evapotranspiration](#).

Choose a new location from [NOAA](#), one we have not worked with yet. In this assignment, you will analyze meteorological data and produce estimates of potential evapotranspiration using Penman's equation.

- Write an introduction. Explain where is your station, what climate it has, mark it on a map, and give other general information relevant to this project.
- Download three years of data for the station you chose.
- Use the library PYET to calculate potential ET for all the three years you chose.
- Plot graphs of the potential ET and of the other major variables related to the PET computation. ~~sssqzqscbbtyhgt~~
- Do you see interesting patterns in the data?
- Does potential ET change across seasons and across years? Comment on that.

You will have three weeks to submit this assignment from the date Yair mentioned it in class.

Best of luck!

Part VIII

Appendix

Welcome to extra stuff at the end of this book.

22 Gain full control of date formatting

```
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.randint(low=-10, high=11, size=len(dates)).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
```

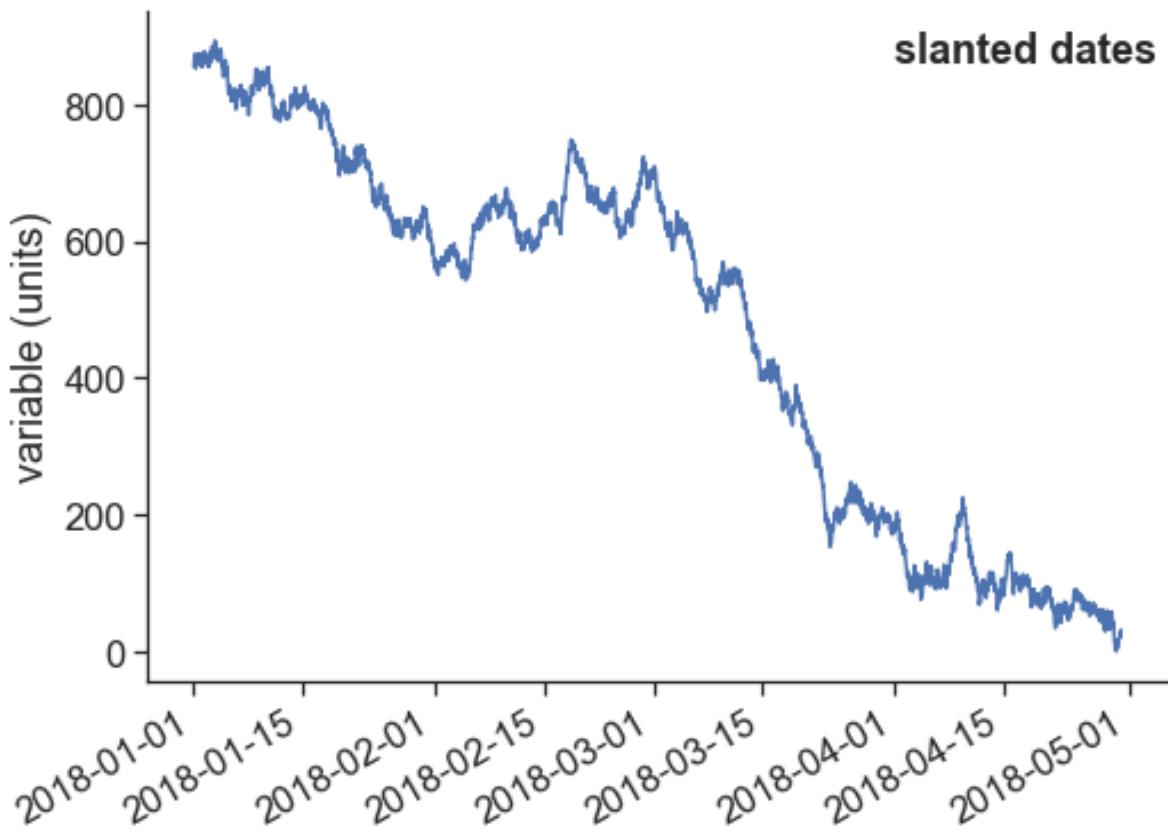
| | date | variable |
|---------------------|---------------------|----------|
| date | | |
| 2018-01-01 00:00:00 | 2018-01-01 00:00:00 | 856 |
| 2018-01-01 01:00:00 | 2018-01-01 01:00:00 | 863 |
| 2018-01-01 02:00:00 | 2018-01-01 02:00:00 | 867 |

| | date | variable |
|---------------------|---------------------|----------|
| date | | |
| 2018-01-01 03:00:00 | 2018-01-01 03:00:00 | 874 |
| 2018-01-01 04:00:00 | 2018-01-01 04:00:00 | 864 |
| ... | ... | ... |
| 2018-04-29 20:00:00 | 2018-04-29 20:00:00 | 20 |
| 2018-04-29 21:00:00 | 2018-04-29 21:00:00 | 20 |
| 2018-04-29 22:00:00 | 2018-04-29 22:00:00 | 27 |
| 2018-04-29 23:00:00 | 2018-04-29 23:00:00 | 23 |
| 2018-04-30 00:00:00 | 2018-04-30 00:00:00 | 32 |

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.savefig("dates1.png")
```



```

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())

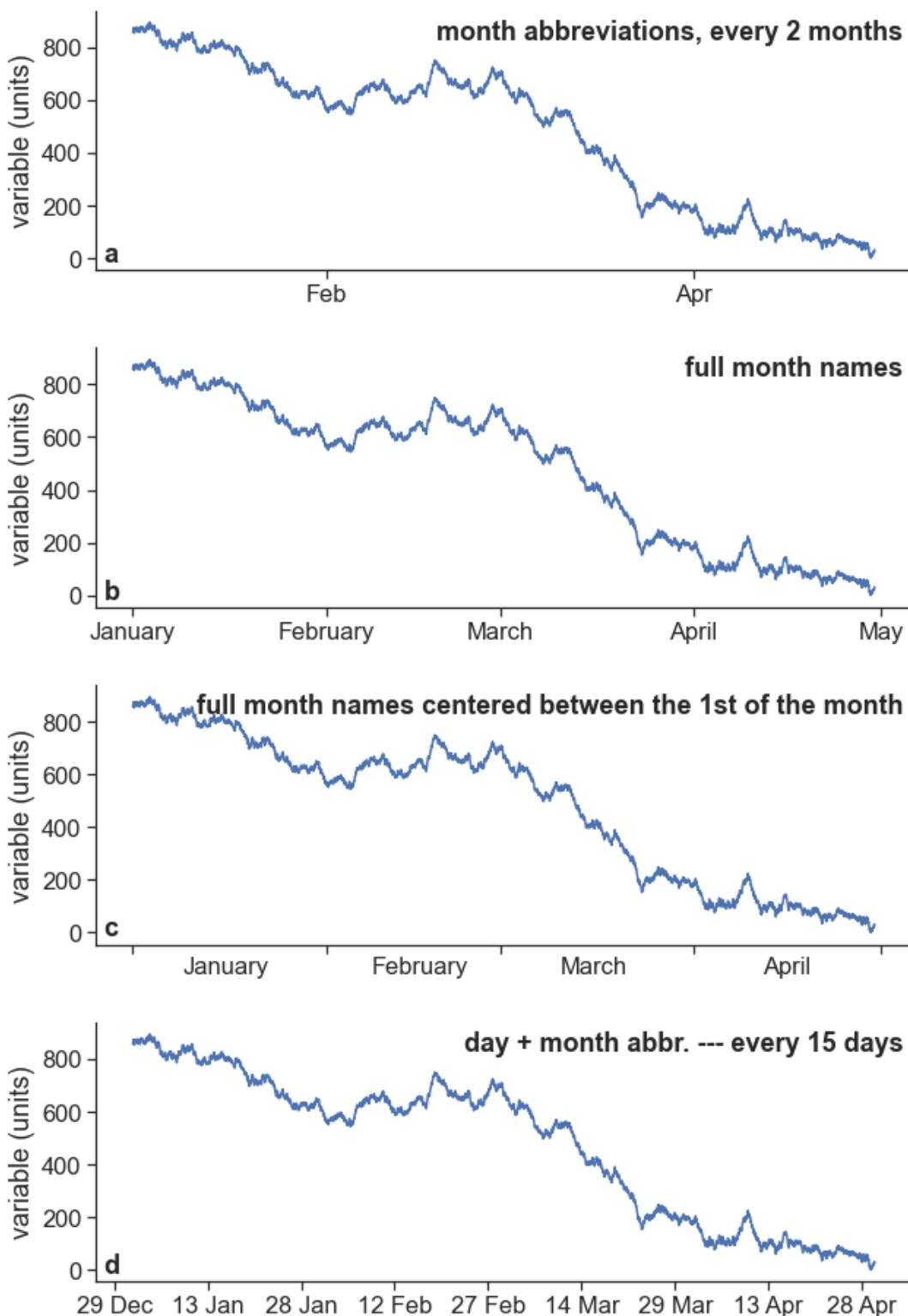
```

```
# 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)

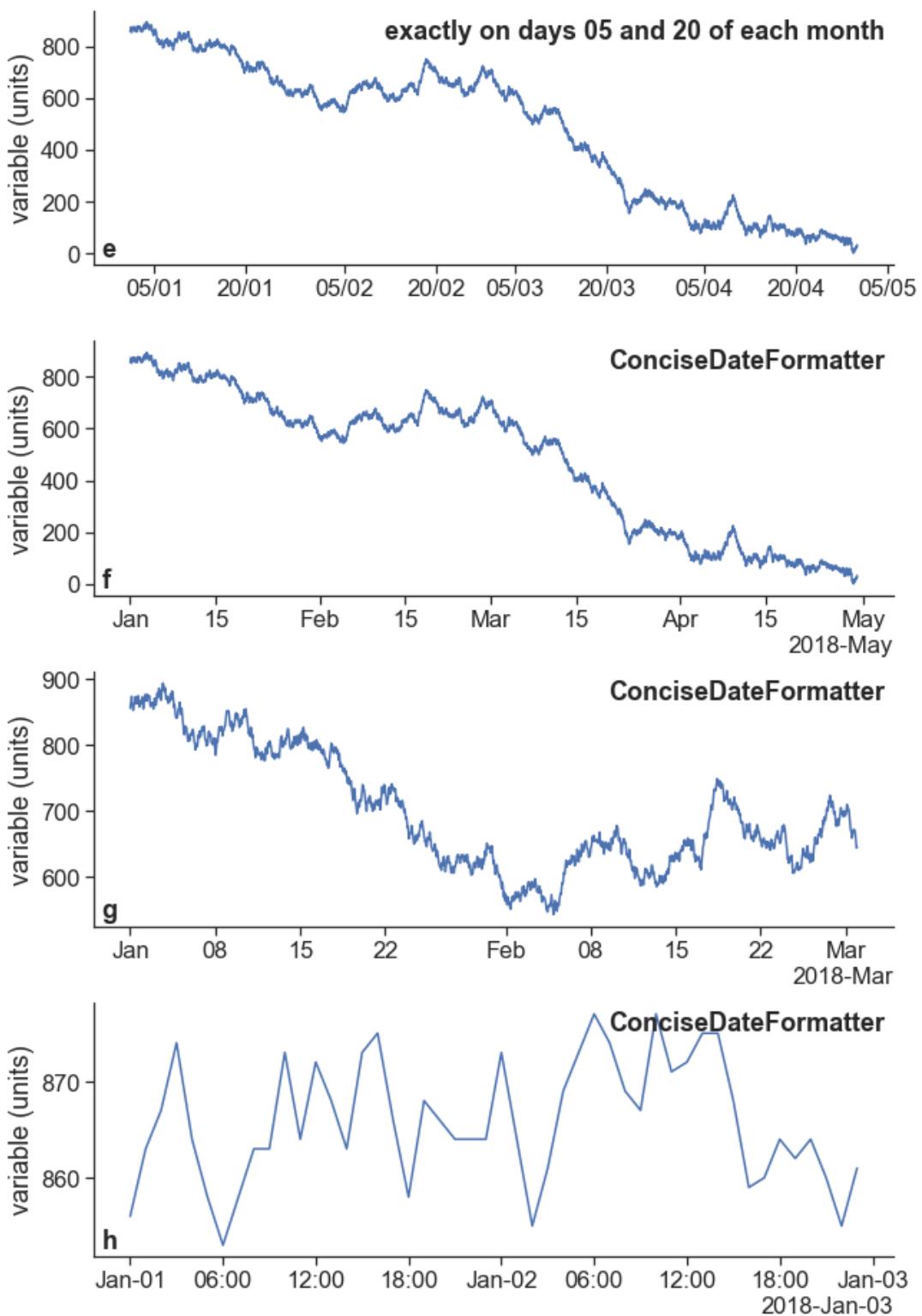
### plot g ####
ax[2].plot(df.loc['2018-01-01':'2018-03-01', 'variable'])
locator = mdates.AutoDateLocator(minticks=6, maxticks=14)
formatter = mdates.ConciseDateFormatter(locator)
ax[2].xaxis.set_major_locator(locator)
ax[2].xaxis.set_major_formatter(formatter)

### 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")

```



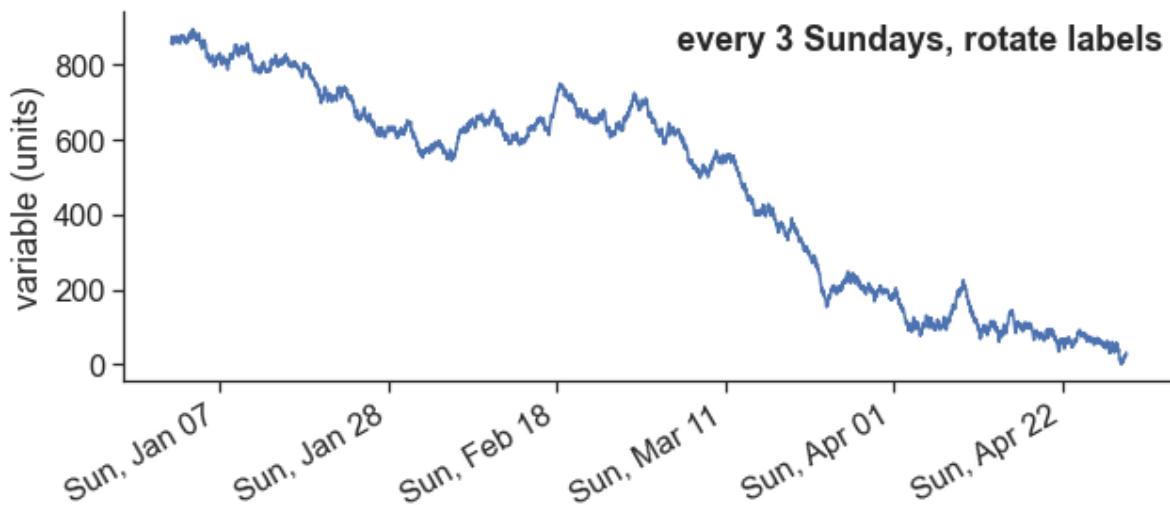
```

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) |

| Code | Explanation |
|------|---|
| %S | 2-digit second (e.g., 59) |
| %p | “AM” or “PM” |
| %Z | Time zone name |
| %z | Time zone offset from UTC (e.g., -0500) |

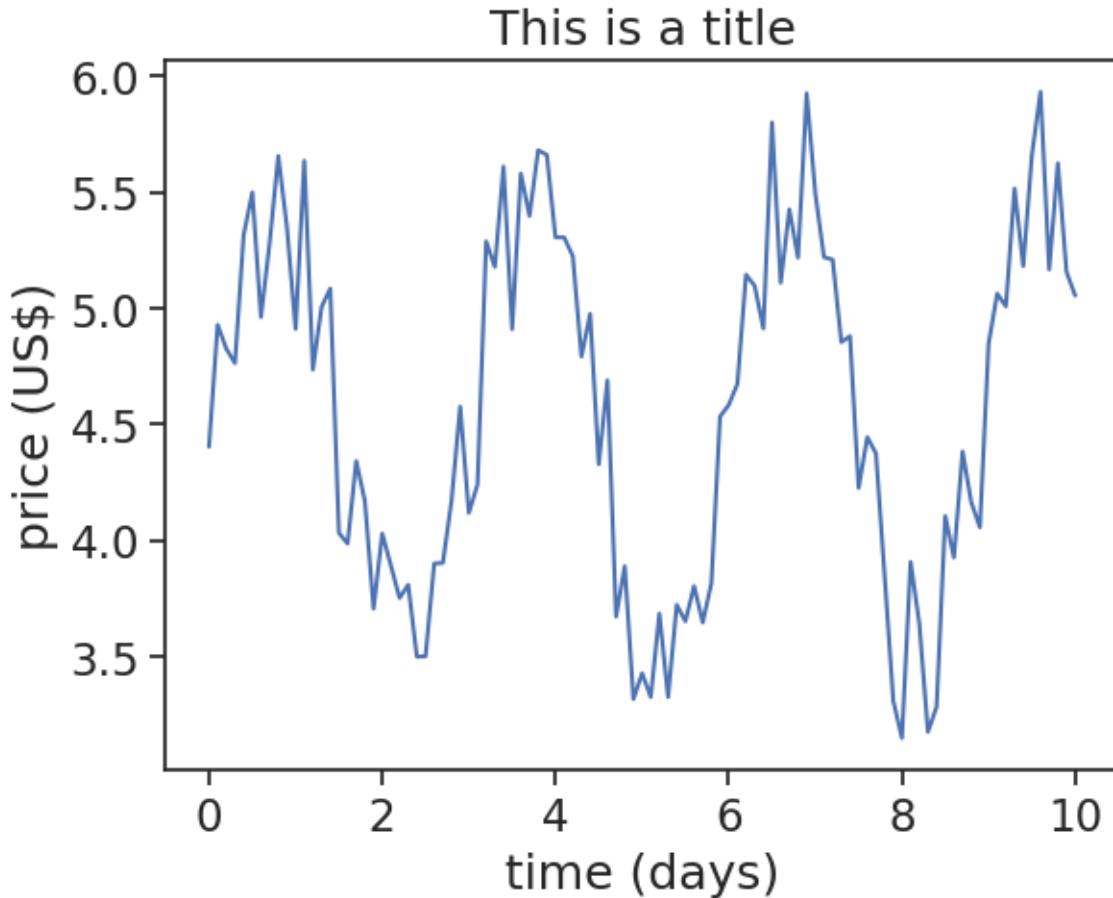
23 Plotting guidelines

23.1 increase fontsize to legible sizes

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

Graph with default matplotlib values:

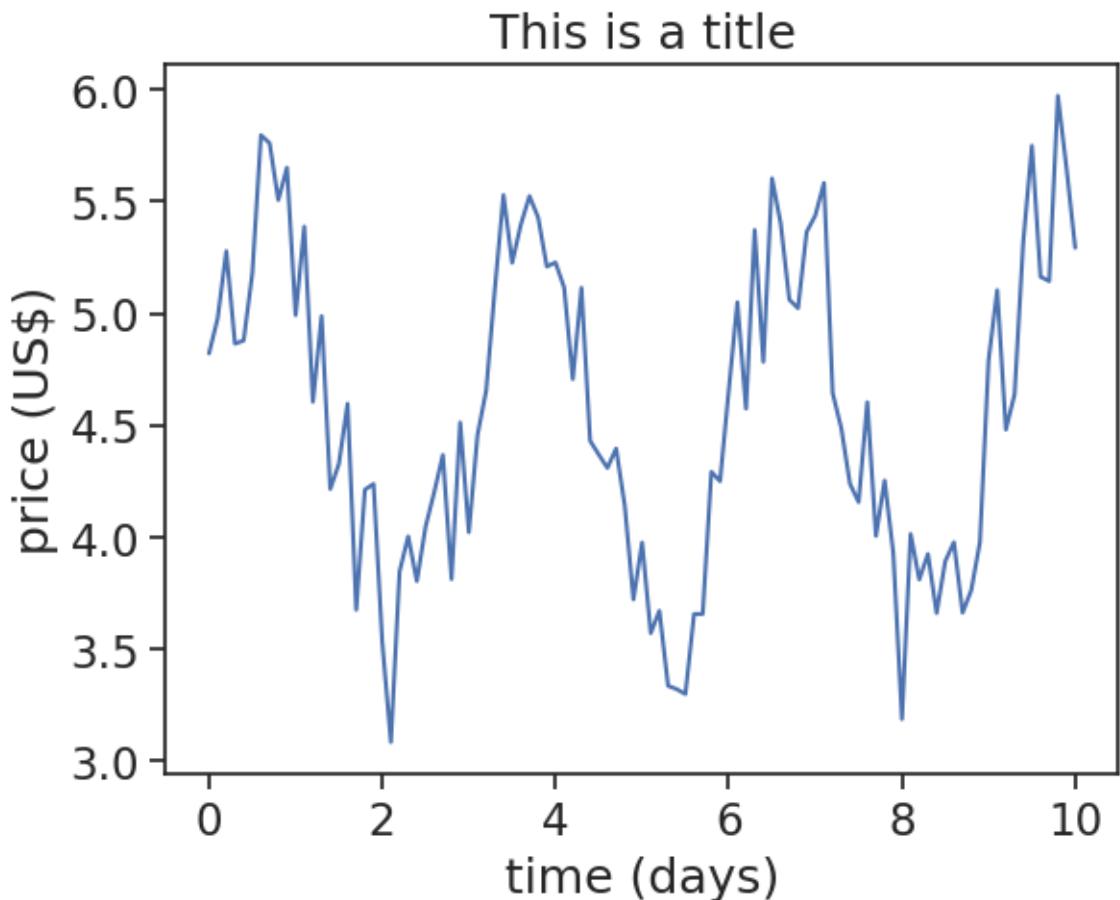
```
t = np.linspace(0, 10, 101)
y = np.sin(2.0*np.pi*t/3) + np.random.random(len(t)) + 4.0
fig, ax = plt.subplots()
ax.plot(t, y)
ax.set(title="This is a title",
       xlabel="time (days)",
       ylabel="price (US$)")
);
```



You can use `seaborn` to easily change plot style and font size:

```
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)

t = np.linspace(0, 10, 101)
y = np.sin(2.0*np.pi*t/3) + np.random.random(len(t)) + 4.0
fig, ax = plt.subplots()
ax.plot(t, y)
ax.set(title="This is a title",
       xlabel="time (days)",
       ylabel="price (US$)"
     );
```



I recommend that you read seaborn's [Controlling figure aesthetics](#).

23.2 choose colors wisely

```
import math
import matplotlib.colors as mcolors
from matplotlib.patches import Rectangle

def plot_colortable(colors, *, ncols=4, sort_colors=True):

    cell_width = 212
    cell_height = 22
    swatch_width = 48
```

```

margin = 12

# Sort colors by hue, saturation, value and name.
if sort_colors is True:
    names = sorted(
        colors, key=lambda c: tuple(mcolors.rgb_to_hsv(mcolors.to_rgb(c))))
else:
    names = list(colors)

n = len(names)
nrows = math.ceil(n / ncols)

width = cell_width * ncols + 2 * margin
height = cell_height * nrows + 2 * margin
dpi = 72

fig, ax = plt.subplots(figsize=(width / dpi, height / dpi), dpi=dpi)
fig.subplots_adjust(margin/width, margin/height,
                    (width-margin)/width, (height-margin)/height)
ax.set_xlim(0, cell_width * ncols)
ax.set_ylim(cell_height * (nrows-0.5), -cell_height/2.)
ax.yaxis.set_visible(False)
ax.xaxis.set_visible(False)
ax.set_axis_off()

for i, name in enumerate(names):
    row = i % nrows
    col = i // nrows
    y = row * cell_height

    swatch_start_x = cell_width * col
    text_pos_x = cell_width * col + swatch_width + 7

    ax.text(text_pos_x, y, name, fontsize=14,
            horizontalalignment='left',
            verticalalignment='center')

    ax.add_patch(
        Rectangle(xy=(swatch_start_x, y-9), width=swatch_width,
                  height=18, facecolor=colors[name], edgecolor='0.7')
    )

```

```
return fig
```

When you plot with matplotlib, the default color order is the following. You can always specify a plot's color by typing something like `color="tab:red`.

```
plot_colortable(mcolors.TABLEAU_COLORS, ncols=2, sort_colors=False);
```



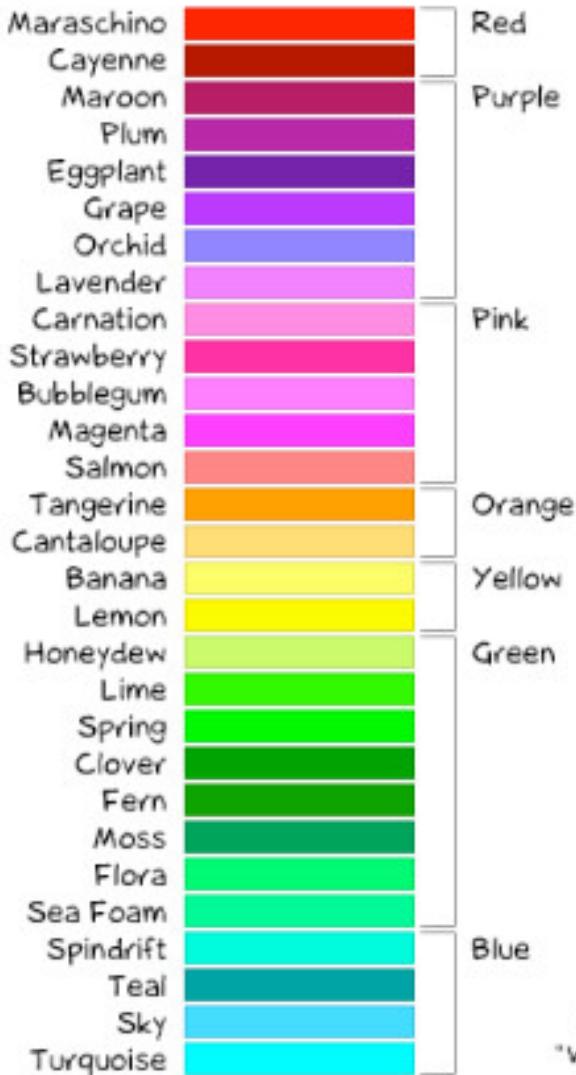
You can write other words as color names, see below.

```
plot_colortable(mcolors.CSS4_COLORS)
plt.show()
```

| | | | |
|-------------|----------------------|-------------------|-----------------|
| black | bisque | forestgreen | slategrey |
| dimgray | darkorange | limegreen | lightsteelblue |
| dimgrey | burlywood | darkgreen | cornflowerblue |
| gray | antiquewhite | green | royalblue |
| grey | tan | lime | ghostwhite |
| darkgray | navajowhite | seagreen | lavender |
| darkgrey | blanchedalmond | mediumseagreen | midnightblue |
| silver | papayawhip | springgreen | navy |
| lightgray | moccasin | mintcream | darkblue |
| lightgrey | orange | mediumspringgreen | mediumblue |
| gainsboro | wheat | mediumaquamarine | blue |
| whitesmoke | oldlace | aquamarine | slateblue |
| white | floralwhite | turquoise | darkslateblue |
| snow | darkgoldenrod | lightseagreen | mediumslateblue |
| rosybrown | goldenrod | mediumturquoise | mediumpurple |
| lightcoral | cornsilk | azure | rebeccapurple |
| indianred | gold | lightcyan | blueviolet |
| brown | lemonchiffon | paleturquoise | indigo |
| firebrick | khaki | darkslategray | darkorchid |
| maroon | palegoldenrod | darkslategrey | darkviolet |
| darkred | darkkhaki | teal | mediumorchid |
| red | ivory | darkcyan | thistle |
| mistyrose | beige | aqua | plum |
| salmon | lightyellow | cyan | violet |
| tomato | lightgoldenrodyellow | darkturquoise | purple |
| darksalmon | olive | cadetblue | darkmagenta |
| coral | yellow | powderblue | fuchsia |
| orangered | olivedrab | lightblue | magenta |
| lightsalmon | yellowgreen | deepskyblue | orchid |
| sienna | darkolivegreen | skyblue | mediumvioletred |
| seashell | greenyellow | lightskyblue | deeppink |
| chocolate | chartreuse | steelblue | hotpink |
| saddlebrown | lawngreen | aliceblue | lavenderblush |
| sandybrown | honeydew | dodgerblue | palevioletred |
| peachpuff | darkseagreen | lightslategray | crimson |
| peru | palegreen | lightslategrey | pink |
| linen | lightgreen | slategray | lightpink |

This reminds me of this cartoon:

Color names if you're a girl...



Color names if you're a guy...

Doghouse Diaries
"We take no as an answer."

For almost all purposes, all these colors should be more than enough.

Be consistent!: if in one plot precipitation is blue and temperature is red, make sure you keep the same colors throughout your assignment.

Be mindful of blind-color people: A good rule of thumb is to avoid red and green shades in the same graph.

I'll put a bunch of links below, this is for my own reference, but you are more than welcome to take a look.

- [ColorBrewer](#)
- [Palettable](#)
- [xkcd colors](#)
- [Colormaps in Matplotlib](#)

23.3 the best legend is no legend

```
t = np.linspace(0, 10, 101)
y1 = np.sin(2.0*np.pi*t/3) + np.random.random(len(t)) + 5.0
y2 = 0.7*np.sin(2.0*np.pi*t/1+1.0) + np.random.random(len(t)) + 2.0
y3 = 0.2*np.sin(2.0*np.pi*t/5+2.0) + 0.2*np.random.random(len(t)) + 3.5

fig, ax = plt.subplots(3, 1, figsize=(8,14))
fig.subplots_adjust(hspace=0.7)

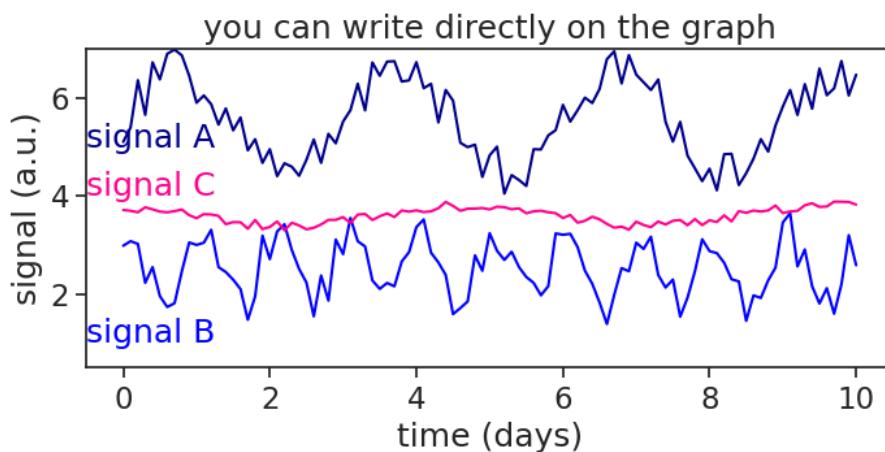
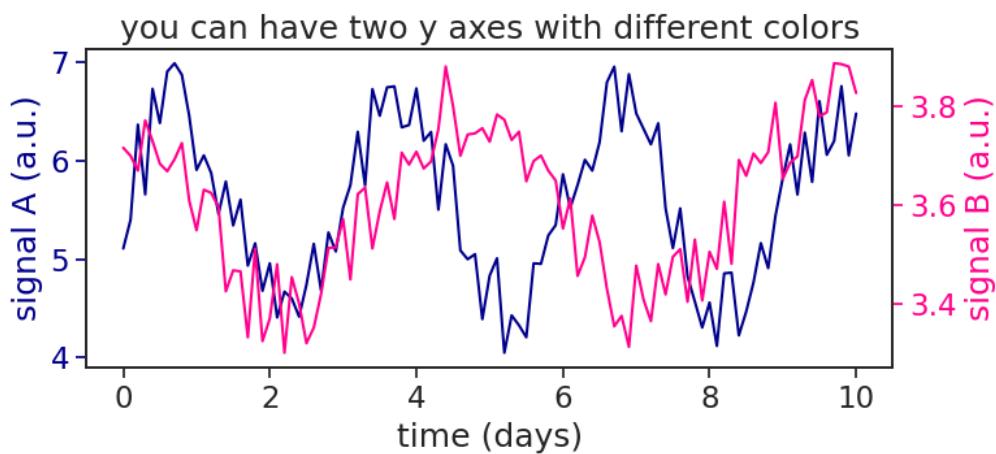
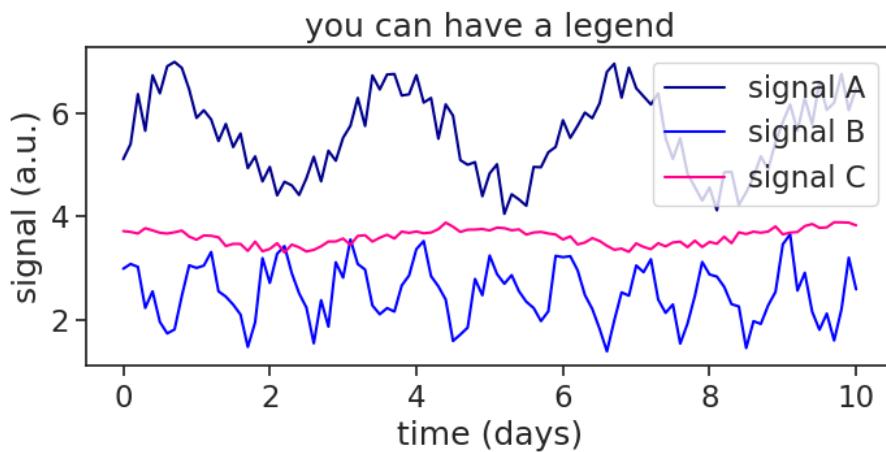
# you can use legends
ax[0].plot(t, y1, color="darkblue", label="signal A")
ax[0].plot(t, y2, color="blue", label="signal B")
ax[0].plot(t, y3, color="xkcd:hot pink", label="signal C")
ax[0].set(title="you can have a legend",
           xlabel="time (days)",
           ylabel="signal (a.u.)"
          )
ax[0].legend()

# you can use an extra y axes
p1, = ax[1].plot(t, y1, color="darkblue")
ax[1].yaxis.label.set_color(p1.get_color())
ax[1].tick_params(axis='y', colors=p1.get_color())
ax[1].set(xlabel="time (days)",
           ylabel="signal A (a.u.)",
           title="you can have two y axes with different colors"
          )
ax1b = plt.twinx(ax[1])
p2, = ax1b.plot(t, y3, color="xkcd:hot pink", label="signal C")
ax1b.set(ylabel="signal B (a.u.)"
          )
```

```
ax1b.yaxis.label.set_color(p2.get_color())
ax1b.tick_params(axis='y', colors=p2.get_color())

# you can write directly on the graph
ax[2].plot(t, y1, color="darkblue", label="signal A")
ax[2].plot(t, y2, color="blue", label="signal B")
ax[2].plot(t, y3, color="xkcd:hot pink", label="signal C")
ax[2].set(xlabel="time (days)",
           ylabel="signal (a.u.)",
           ylim=[0.5,7],
           title="you can write directly on the graph"
)
ax[2].text(-0.5, 5, "signal A", color="darkblue", ha="left")
ax[2].text(-0.5, 1, "signal B", color="blue", ha="left")
ax[2].text(-0.5, 4, "signal C", color="xkcd:hot pink", ha="left")

Text(-0.5, 4, 'signal C')
```



You can also make a colorbar to substitute a legend.

```
num_lines = 6

t = np.linspace(0, 2, 101)

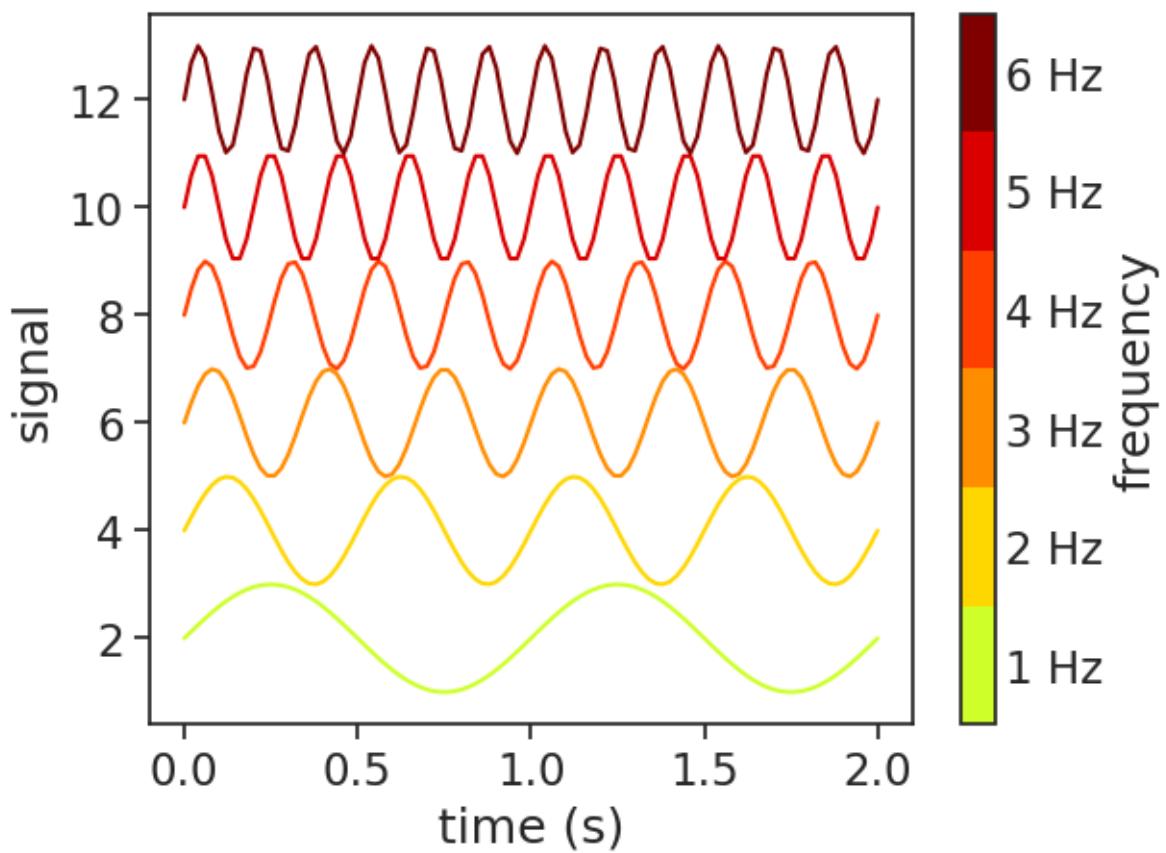
# Get truncated colormap
cmap = plt.colormaps.get_cmap('jet')
bottom = 0.6; top = 1.0
truncated_cmap = mcolors.LinearSegmentedColormap.from_list("truncated_viridis", cmap(np.linspace(bottom, top, num_lines)))

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the lines with increasing frequency
for i in range(num_lines):
    freq = i + 1
    y = np.sin(2.0 * np.pi * t * freq) + 2*freq
    ax.plot(t, y, color=truncated_cmap(i / (num_lines - 1)), label=f'Slope {slope}')

ax.set(xlabel="time (s)",
       ylabel="signal")

# Create a discrete colorbar
boundaries = np.linspace(0.5, num_lines + 0.5, num_lines + 1)
ticks = np.arange(num_lines) + 1
norm = mcolors.BoundaryNorm(boundaries, truncated_cmap.N)
sm = plt.cm.ScalarMappable(cmap=truncated_cmap, norm=norm)
sm.set_array([]) # fake up the array of the scalar mappable
cbar = plt.colorbar(sm, ticks=ticks, boundaries=boundaries, label='frequency', ax=ax)
cbar.ax.tick_params(which='both', size=0)
freq_list = [f"{x+1} Hz" for x in range(num_lines)]
cbar.set_ticklabels(freq_list)
```



24 Summary

In summary, this book has no content whatsoever.

This is Yair making changes to summary.

References

- Allen, Richard G, Luis S Pereira, Dirk Raes, Martin Smith, et al. 1998. “Crop Evapotranspiration-Guidelines for Computing Crop Water Requirements-FAO Irrigation and Drainage Paper 56.” *Fao, Rome* 300 (9): D05109.
- Amazon Waters. 2022. “Amazon Waters.” *Amazon Waters*. <https://amazonwaters.org/basins>.
- Andrew Amelinckx. 2015. “Even Without a Drought, We’re Depleting Groundwater at an Alarming Pace.” *Modern Farmer*. <https://modernfarmer.com/2015/07/ogallala-aquifer-depletion/>.
- Berghuijs, Wouter R, Sebastian J Gnann, and Ross A Woods. 2020. “Unanswered Questions on the Budyko Framework.” *Journal of Hydrology* 265: 164–77.
- Brutsaert, Wilfried. 2005. *Hydrology: An Introduction*. Cambridge University Press.
- Budyko, Mikhail Ivanovich. 1974. “Climate and Life.” (*No Title*).
- Creed, Irena, and Adam Spargo. 2012. “Budyko Guide to Exploring Sustainability of Water Yields from Catchments Under Changing Environmental Conditions.” *London, Ontario. Http://Www. Uwo. Ca/Biology/Faculty/Creed/PDFs/Presentations/PRE116. Pdf*.
- Daly, Edoardo, Salvatore Calabrese, Jun Yin, and Amilcare Porporato. 2019. “Linking Parametric and Water-Balance Models of the Budyko and Turc Spaces.” *Advances in Water Resources* 134: 103435.
- Dingman, S. L. 2015. *Physical Hydrology*. 3rd edition. Waveland Press, Incorporated.
- dreamstime. 2022. “World Map of AFRICA.” *Dreamstime*. <https://www.dreamstime.com/world-map-africa-egypt-libya-ethiopia-arabia-mauritania-nigeria-somalia-namibia-tanzania-madagascar-geographic-xxl-chart-image154799901>.
- Fiona Bruce. 2015. “A Family Holiday in Lake Malawi: Zen and the Art of Paddleboarding.” *The Telegraph*. https://twitter.com/hallaboutafrica/status/1203419359303159809?s=20&t=SkH17UkWrNcXzIqRF0ic_A.
- Hillel, Daniel. 2003. *Introduction to Environmental Soil Physics*. Elsevier.
- James Hall. 2019. “Lake Malawi.” *Twitter*. https://twitter.com/hallaboutafrica/status/1203419359303159809?s=20&t=SkH17UkWrNcXzIqRF0ic_A.
- Jones, Julia A, Irena F Creed, Kendra L Hatcher, Robert J Warren, Mary Beth Adams, Melinda H Benson, Emery Boose, et al. 2012. “Ecosystem Processes and Human Influences Regulate Streamflow Response to Climate Change at Long-Term Ecological Research Sites.” *BioScience* 62 (4): 390–404.
- Kbh3rd. 2009. “High Plains Fresh Groundwater Usage 2000.” *Wikimedia*. https://commons.wikimedia.org/wiki/File:High_plains_fresh_groundwater_usage_2000.svg.

- Krajewski, Adam, Anna E Sikorska-Senoner, Leszek Hejduk, and Kazimierz Banasik. 2021. "An Attempt to Decompose the Impact of Land Use and Climate Change on Annual Runoff in a Small Agricultural Catchment." *Water Resources Management* 35 (3): 881–96.
- leddris. 2010. "Rainfall Seasonality." *Land and Ecosystem Degradation and Desertification Response Information System*. <http://leddris.aegean.gr/ses-parameters/293-rainfall-seasonality.html#:~:text=Rainfall%20seasonality%20index%20is%20a,in%20relation%20to%20water%20availability>.
- Margulis, Steve. 2019. "Introduction to Hydrology. eBook." <https://margulis-group.github.io/textbook/>.
- Marty Friedlander. 2015. "Natural Springs of Israel: Seven Cool Watering Holes to Visit This Summer." *Haaretz*. <https://www.haaretz.com/israel-news/travel/seven-cool-natural-springs-of-israel-1.5388627>.
- National Park Service. 2022. "Mississippi River Facts." *National Park Service*. <https://www.nps.gov/miss/riverfacts.htm>.
- Natural Attractions. 2014. "Ogallala Aquifer." *Nebraska Education on Location*. <https://nebraskaeducationonlocation.org/natural-attractions/ogallala-aquifer/>.
- Raymond, Lyle S. Jr. 1988. "What Is Groundwater?" *Cornell eCommons*. <https://ecommons.cornell.edu/handle/1813/3408>.
- Sposito, Garrison. 2017. "Understanding the Budyko Equation." *Water* 9 (4): 236.
- Suma Groulx. 2015. "Water Infiltration." *Suma Groulx*. <http://sumagroulx.com/water-infiltration/>.
- United States Department of Agriculture, Natural Resources Conservation Service. 2004. *Estimation of Direct Runoff from Storm Rainfall*. National Engineering Handbook, Part 630 Hydrology, Chapter 10. United States Department of Agriculture. <https://directives.sc.egov.usda.gov/OpenNonWebContent.aspx?content=17752.wba>.
- Valentí Rodellas. 1988. "Evaluating Submarine Groundwater Discharge to the Mediterranean Sea by Using Radium Isotopes." *Research Gate*. https://www.researchgate.net/figure/Principal-pathways-for-submarine-groundwater-discharge-to-the-coastal-ocean-including_fig1_274590439.
- Walsh, RPD, and DM Lawler. 1981. "Rainfall Seasonality: Description, Spatial Patterns and Change Through Time." *Weather* 36 (7): 201–8. <https://doi.org/10.1002/j.1477-8696.1981.tb05400.x>.
- Ward, Andy D, and Stanley W Trimble. 2003. *Environmental Hydrology*. 2nd ed. CRC Press.
- Water Science School. 2016. "Water Flowing Underground Can Find Openings Back to the Land Surface." *U.S. Geological Survey*. <https://www.usgs.gov/media/images/water-flowing-underground-can-find-openings-back-land-surface>.
- . 2018. "Where Is Earth's Water?" *U.S. Geological Survey*. <https://www.usgs.gov/special-topics/water-science-school/science/where-earths-water>.
- . 2019a. "Conceptual Groundwater-Flow Diagram." *U.S. Geological Survey*. <https://www.usgs.gov/media/images/conceptual-groundwater-flow-diagram>.
- . 2019b. "Groundwater Is the Area Underground Where Openings Are Full of Water." *U.S. Geological Survey*. <https://www.usgs.gov/media/images/groundwater-area-underground-where-openings-are-full-water>.

- _____. 2019c. "How Much Water Is There on Earth?" *U.S. Geological Survey*. <https://www.usgs.gov/special-topics/water-science-school/science/how-much-water-there-earth>.
- _____. 2019d. "Ice, Snow, and Glaciers and the Water Cycle." *U.S. Geological Survey*. <https://www.usgs.gov/special-topics/water-science-school/science/ice-snow-and-glaciers-and-water-cycle>.
- _____. 2019e. "Precipitation and the Water Cycle." *U.S. Geological Survey*. <https://www.usgs.gov/special-topics/water-science-school/science/precipitation-and-water-cycle>.
- _____. 2019f. "Rain and Precipitation." *U.S. Geological Survey*. <https://www.usgs.gov/special-topics/water-science-school/science/rain-and-precipitation>.
- _____. 2019g. "The Natural Water Cycle." *U.S. Geological Survey*. <https://www.usgs.gov/media/images/natural-water-cycle-jpg>.
- _____. 2022. "The Water Cycle." *U.S. Geological Survey*. <https://www.usgs.gov/media/images/water-cycle-png>.
- Zhang, Lu, Klaus Hickel, WR Dawes, Francis HS Chiew, AW Western, and PR Briggs. 2004. "A Rational Function Approach for Estimating Mean Annual Evapotranspiration." *Water Resources Research* 40 (2).
- . .2020 " ". *Melabes*. <https://www.melabes.co.il/news/51773>.