

Scientific Computation

Spring 2023

© Prasun K. Ray (2023) These notes are provided for the personal study of students taking Scientific Computation at Imperial College London during the 2022-23 academic year. The distribution of copies in part or whole is not permitted.

Contents

Lecture 1

1.1 Module organization [10](#)

1.2 Overview of scientific computation [13](#)

1.3 Searching and introduction to binary search [24](#)

Lecture 2

2.1 Sorting, insertion sort [32](#)

2.2 merge sort [37](#)

Lecture 3:

3.1 Comments on Python, recursion, searching, and sorting [44](#)

3.2 Searching in dynamic datasets and hash functions [49](#)

Lecture 4:

4.1 Hash tables and Python dictionaries [61](#)

4.2 Analyzing computational cost [71](#)

Contents

Lecture 5:

5.1 Genetic code	80
5.2 The pattern search problem and naïve pattern search	85
5.3 Alternatives to the naïve method and the Rabin Karp algorithm	88

Lecture 6:

6.1 Getting started with networks	99
6.2 Intro to NetworkX	105
6.3 Graph search and breadth-first search	111

Lecture 7:

7.1 Python containers	125
7.2 Depth-first search	128

Contents

Lecture 8:

8.1 Dijkstra's algorithm [133](#)

8.2 Improving Dijkstra's algorithm and binary heaps [150](#)

Lecture 9:

9.1: Computational science [161](#)

9.2: Simulating random walks [165](#)

Lecture 10: Solving deterministic initial value problems

10.1: Simple 2-equation system [175](#)

10.2: Numerical solution of nonlinear systems [182](#)

Lecture 11:

11.1 Random walks and Brownian motion [197](#)

11.2 Stochastic differential equations [205](#)

Contents

Lecture 12:

12.1 Maximum growth [218](#)

12.2 Computation of eigenvalues and singular value decomposition [226](#)

Lecture 13:

13.1 Matrix computations: notation, applications, and the SVD [232](#)

13.2 Low-rank approximation [239](#)

Lecture 14:

14.1 Principal component analysis [250](#)

14.2 Recommender systems and low-rank factorization [262](#)

Lecture 15:

15.1 Data analysis and Fourier series [275](#)

15.2 Discrete Fourier transforms [283](#)

15.3 Windowing and Welch's method [288](#)

Contents

Lecture 16:

16.1 Multiscale science and engineering	295
16.2 Second-order finite difference method and wavenumber analysis	297
16.3 Implicit finite difference methods	303

Lecture 17:

17.1 Linear data analysis, logistic map, fractal dimension, and correlation dimension	317
17.2 Lorenz system, phase plots, and attractor reconstruction	329
17.3 Transitions between states, orbit diagrams, and maps	335

Lecture 1

Module organization
Overview of scientific computation
Searching and binary search

People

Prasun Ray (module leader)

p.ray@imperial.ac.uk

GTAs:

Antonio Matas Gil

Jacob Francis

Module structure

- The module runs weeks 2-11 of term, we have 20 lectures + 9 computer labs
 - Lectures are Mondays 1-1:50pm and Tuesdays 11-11:50am
 - Slides will be posted on Blackboard in advance of each lecture. I will provide collected slides in a single navigable pdf when each assignment is released. These are the “lecture notes” for the class.
 - A small amount of material will be delivered via pre-recorded videos
 - Labs will be Thursdays 11-11:50am. You will be provided with Jupyter notebooks containing self-guided exercises. GTAs and I will be there to answer questions.
 - Office hour ([Teams](#)): Tuesdays 1-1:50 pm

Online resources

- Module Blackboard page: *all module material will be posted here*
- Ed discussion board: <https://edstem.org/us/courses/17138/discussion/>
Ask (and answer) questions on nearly all module-related topics here
 - Questions on Ed will be prioritized over emails
 - You can post questions privately (only instructors/GTAs will see them)
 - You can also post questions where you are anonymous to other classmates (but not to instructors/GTAs)
- Microsoft Teams page: MATH60027/70027/97086 - Scientific Computation (Spring 2022-2023)
 - Meetings will be started here for office hours

Assessment (**tentative**)

3 Programming projects

Project 1: Assigned 2/2, due 10/2 (**20%**)

Project 2: Assigned 21/2, due 6/3 (**30%**)

Project 3, Assigned 11/3, due 24/3 (**50%**)

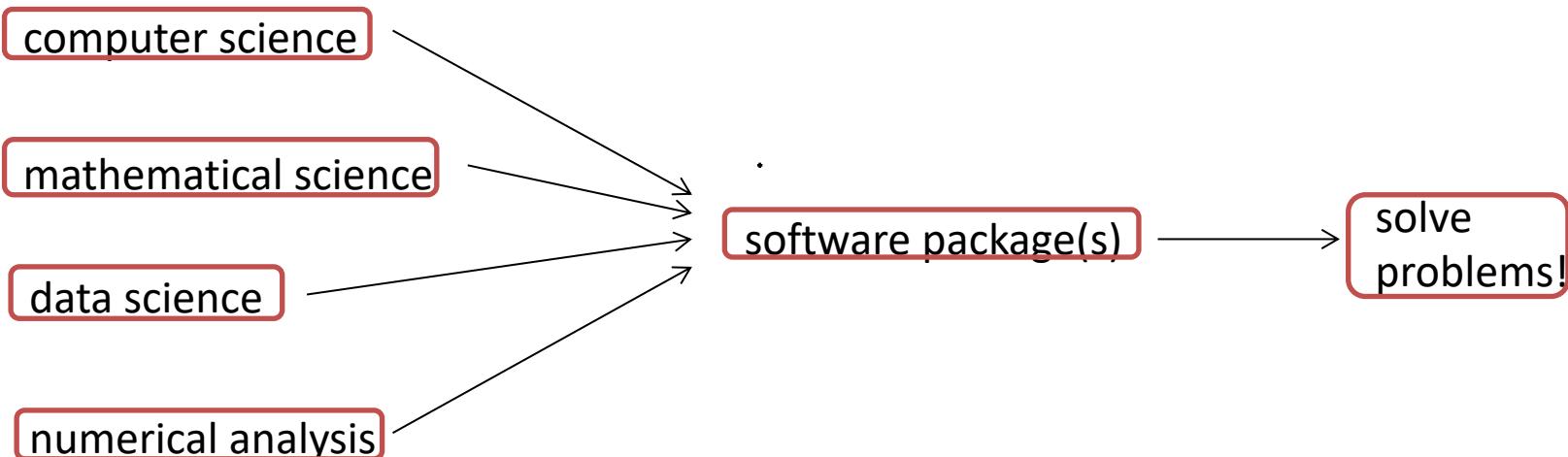
Project 4 (MSci/MSci only): Assigned 20/3, due 5/4 (see adjusted weights below)

Submitting Project 1 commits you to the course

Mastery material (MSci/MSc): Project 4 will be for MSci/MSci students only. The weights for the 3 assignments are different from above; they are: Project 1: **16%**, Project 2: **24%**, Project 3: **40%**, Project 4: **20%**

Scientific computation

- What is “*Scientific computation*”?
 - No single, standard definition (that I’m aware of!)



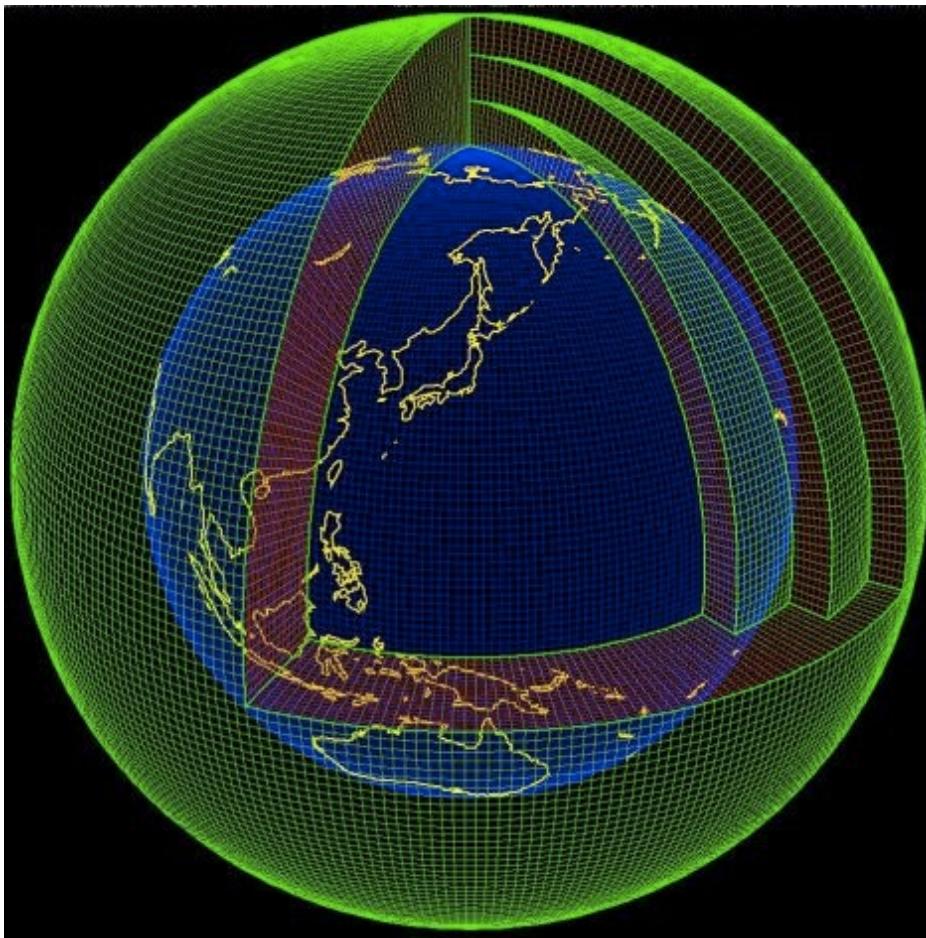
Pause and think

What are some real-world applications of scientific computation?



D.S. Bassett, How You Think: Structural Network Mechanisms of Human Brain Function

- **Example 1:** Brain dynamics
- Graph partitioning
- Biochemistry
- Nonlinear differential equations



<https://www.jma.go.jp/jma/jma-eng/jma-center/nwp>

- **Example 2:** Numerical weather prediction
- Partial differential equations
- Data assimilation
- Large sparse linear systems



- **Example 3:** Self-driving car
- Rapid image processing (images are matrices)
- Numerical linear algebra
- Machine learning
- Mathematical optimization

Approximate syllabus and objectives

Lectures 1-5: Searching and sorting (plus a little DNA)

Lectures 6-8: Complex networks and graph search

Lectures 9-10: Simulating stochastic processes

Lectures 11-14: Matrix computations: linear systems and data science

Lectures 15-19: Linear and nonlinear data analysis

Lecture 20: Moving beyond this module

Objective 1: Deepen your understanding of how science, math, and computing can be used together to attack complex problems

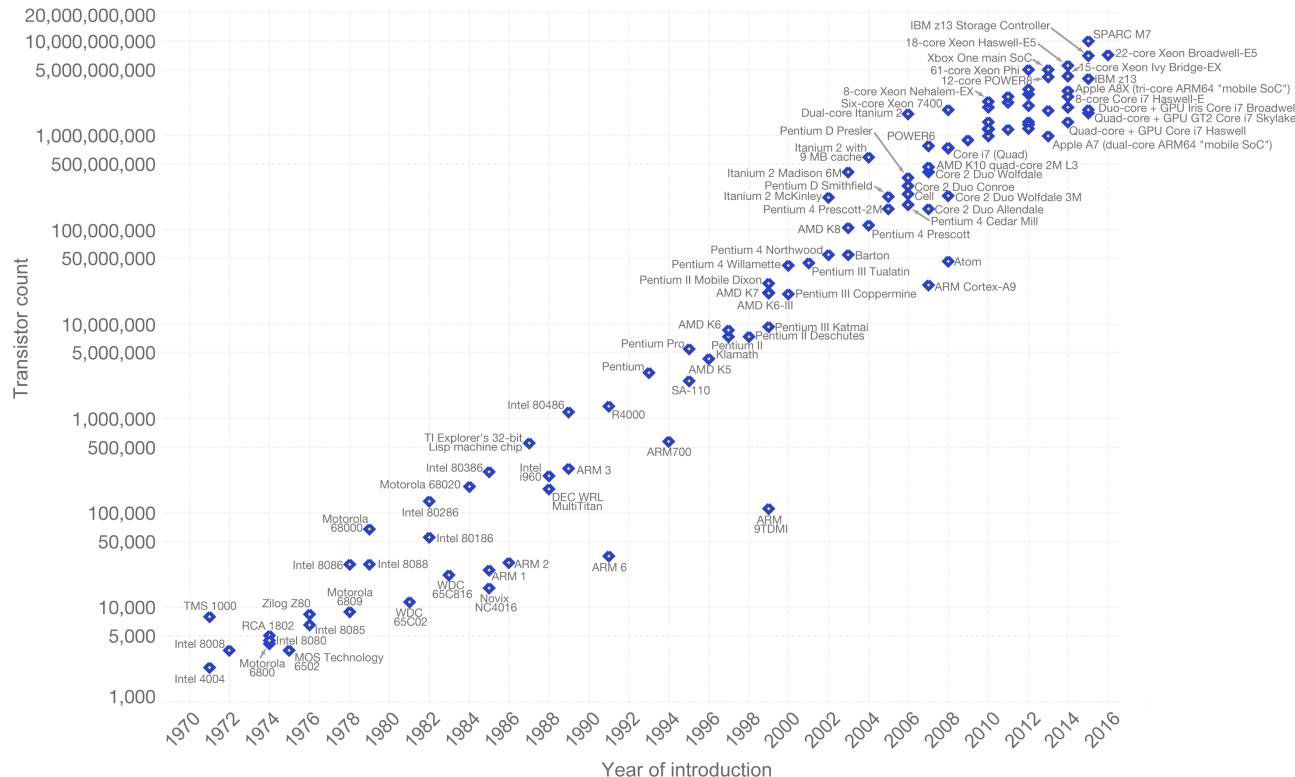
Objective 2: Improve your ability to: develop code, analyze code (and algorithms and numerical methods), and analyze results generated by code

Computing

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

OurWorld
in Data

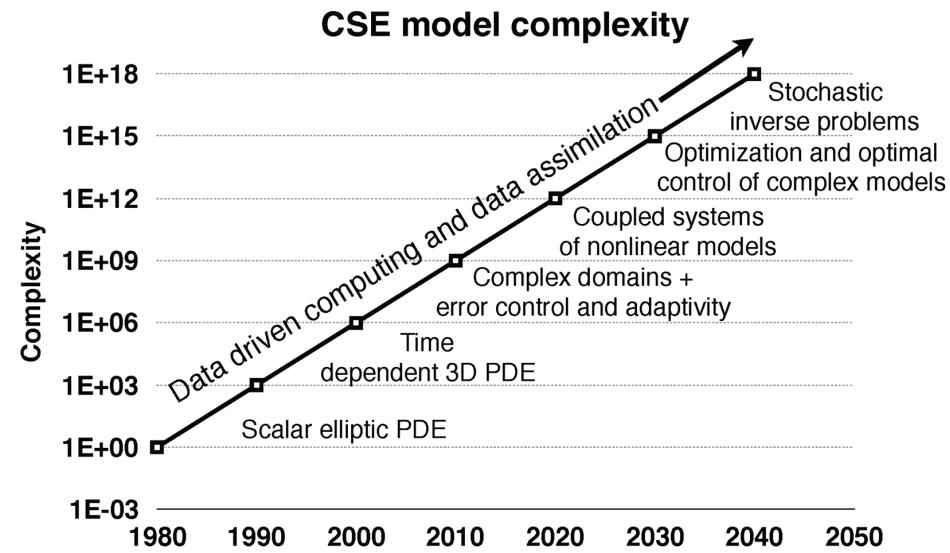
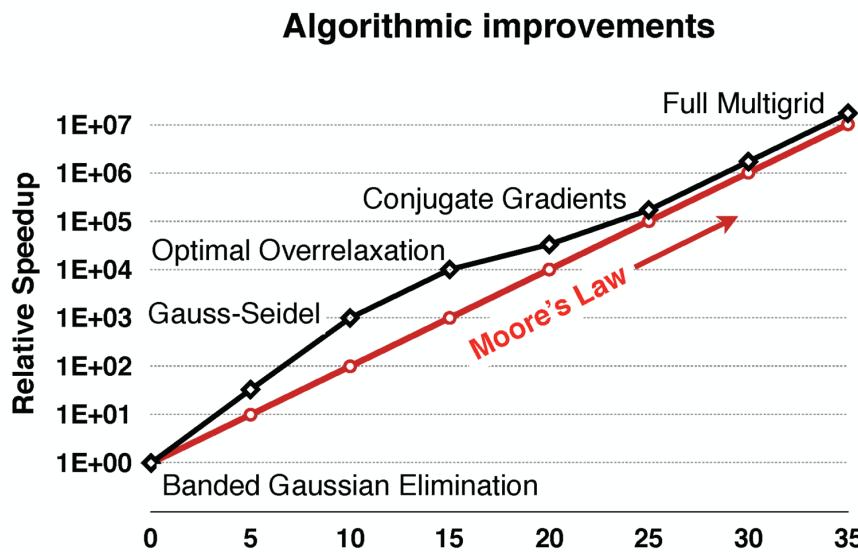


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

- The number of transistors on microchips doubles every two years
- There is an enormous amount of computational power at our fingertips



- Algorithmic efficiency (in some cases) has improved in parallel with computational power
- With this improvement in efficiency and power has come an increase in the complexity of problems being investigated

-
- This course could be called *Scientific computation with Python*
 - Is Python a “good” language for scientific computing?

Pause and think:

What are some of the advantages and disadvantages of using Python?

Python

Advantages:

- Easy to learn, free, very widely-used
- Many powerful tools for scientific computing are available (e.g. numpy, scipy, scikit-learn)
- Also a general-purpose language – well-designed for general software development

Disadvantages:

- Not specifically designed for scientific computing (cf. Matlab, R)
- Python is an *interpreted* language – it will be slower than compiled languages (c, Fortran) in some cases

It's a good place to start!

- This class assumes either:
 - Familiarity with 1st-year computing python:
 - Basic datatypes, loops, functions, numpy arrays, plotting with matplotlib
 - Or good programming experience and a willingness to independently learn the basics of Python+numpy during the first ~2 weeks of the class
- Introductory videos, exercises and references are available on the module Blackboard page: look through the exercises this week!

We will be using Python, and it is recommended that you use/install the standard Anaconda package:
<https://www.anaconda.com/products/individual>



- This will give you the Spyder IDE and all of the packages that you may need (Numpy, Scipy, Matplotlib, NetworkX, Pandas,...)
- For more information on software installation:
<https://imperial-fons-computing.github.io/>
- If you have installation/software/hardware problems, please post your issue on Ed within the computer_issues folder

Searching

- The two foundational problems in computer science are searching and sorting
- **Problem statement:** *Given a sorted list, find a location of a specified item within the list. Return “not found” if item is not contained within list*
- Examples:
 - Find name in directory
 - Find book (by call number) in library
 - Find id number in database
 - Find “31” in the array below

2	6	8	23	24	31	32	53	56
---	---	---	----	----	----	----	----	----

- Note: I am assuming that all elements in the list are comparable, that the `<`, `>`, `==` operators are meaningful for each distinct pair of elements, and I may sometimes refer to the input as an *array* even if the Python list datatype is used

What is the **simplest** way to approach this problem?

-
- Linear (naive) search: Step through list one element at a time, terminate search if/when item is found

```
def linsearch(L,x):
    """find location of x in L, if x is not in L, return -1
    """
    for i,y in enumerate(L):
        if y==x: return i

    return -1
```

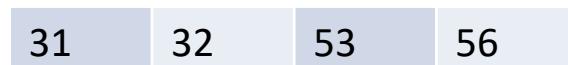
- What is the cost?
 - Each iteration, we 1) increment i , 2) assign a value to y , and 3) compare y with x
 - So we have 3 operations per iteration, on average the cost will be $3N/2$ operations where $N=\text{len}(L)$
 - Can we do better?

-
- To do better, we should take advantage of the list being sorted
 - Linear search discards one element at a time. Can we discard more?
 - We can discard half if we first compare with the median element
 - If $x < L_{\text{median}}$: no need to search in “right” half of list
 - or if $x > L_{\text{median}}$: no need to search in “left” half
 - Can then run linear search on appropriate half
 - But why run linear search? Can again discard half (of what’s left)
 - This is the idea behind *binary search*

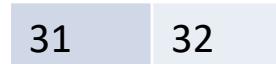
Task: find 31 in list below



$31 > 24$: discard left 'half':



$31 < 53$: discard right half



$31 < 32$: discard right half

(And we should keep track of the locations of the retained elements)

Python implementation: the key is to keep track of start and end indices of list L as it is truncated

```
#Set initial start and end indices for full list
istart = 0
iend = len(L)-1

#Contract "active" portion of list
while istart<=iend:

    imid = int(0.5*(istart+iend))

    if x==L[imid]:
        return imid
    elif x < L[imid]:
        iend = imid-1
    else:
        istart = imid+1

return -1
```

-
- When analyzing algorithms, we should consider their *correctness* and *efficiency*
 - Correctness of binary search: basic idea – if target is within original array, it will either be the “median element”, or it will be in the sub-array that remains after contraction (convince yourself that a proof of correctness follows)
 - Efficiency: How many operations does binary search require? Is it faster than linear search?

Consider a special case of binary search where exactly half of array is discarded each iteration

- If we start with $N = 16$, then in the worst case, there are iterations for arrays with size 16,8,4,2,1
- Each iteration requires ~ 11 operations (how many do you count?)
- Worst case: $11(\log_2 N + 1) + 4$ operations
- Generally, we are interested in situations where N is large and say that the cost is $\mathcal{O}(\log_2 N)$
 - A (somewhat) more formal discussion of “big-O” notation will be provided later

Lecture 2

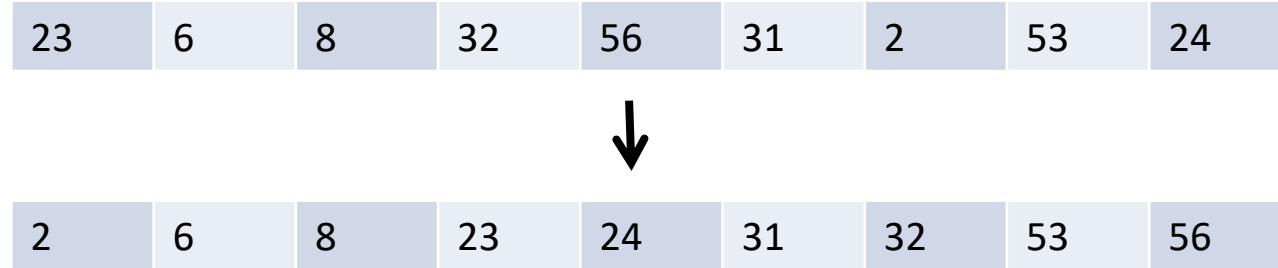
Sorting

insertion sort

merge sort

Sorting

- Our search problem required a sorted list. Let's now think about how to produce a sorted list
- Problem statement: *Given an unsorted list, return a list with the same elements in non-decreasing order*



- Motivation: maintaining sorted lists facilitates fast searches
- There are many different sorting algorithms, we will consider two

Example input:

23	6	8	32	56	31	2	53	24
----	---	---	----	----	----	---	----	----

Algorithm 1

Step 1: Sort first 2 elements

Step 2: Sort first 3 elements (knowing that 1st two have been sorted)

Step N-1: Sort first N elements (knowing that 1st $N - 1$ have been sorted)

-
- Algorithm 1, first 6 iterations:

<table border="1"><tr><td>23</td><td>6</td><td>8</td><td>32</td><td>56</td><td>31</td><td>2</td><td>53</td><td>24</td></tr></table>	23	6	8	32	56	31	2	53	24	Initial data
23	6	8	32	56	31	2	53	24		
<table border="1"><tr><td>6</td><td>23</td><td>8</td><td>32</td><td>56</td><td>31</td><td>2</td><td>53</td><td>24</td></tr></table>	6	23	8	32	56	31	2	53	24	Iteration 1
6	23	8	32	56	31	2	53	24		
<table border="1"><tr><td>6</td><td>8</td><td>23</td><td>32</td><td>56</td><td>31</td><td>2</td><td>53</td><td>24</td></tr></table>	6	8	23	32	56	31	2	53	24	Iteration 2
6	8	23	32	56	31	2	53	24		
<table border="1"><tr><td>6</td><td>8</td><td>23</td><td>32</td><td>56</td><td>31</td><td>2</td><td>53</td><td>24</td></tr></table>	6	8	23	32	56	31	2	53	24	Iteration 3
6	8	23	32	56	31	2	53	24		
<table border="1"><tr><td>6</td><td>8</td><td>23</td><td>32</td><td>56</td><td>31</td><td>2</td><td>53</td><td>24</td></tr></table>	6	8	23	32	56	31	2	53	24	Iteration 4
6	8	23	32	56	31	2	53	24		
<table border="1"><tr><td>6</td><td>8</td><td>23</td><td>31</td><td>32</td><td>56</td><td>2</td><td>53</td><td>24</td></tr></table>	6	8	23	31	32	56	2	53	24	Iteration 5
6	8	23	31	32	56	2	53	24		
<table border="1"><tr><td>2</td><td>6</td><td>8</td><td>23</td><td>31</td><td>32</td><td>56</td><td>53</td><td>24</td></tr></table>	2	6	8	23	31	32	56	53	24	Iteration 6
2	6	8	23	31	32	56	53	24		

Python implementation of algorithm 1:

```
for j in range(1, len(L)):
    #compare L[j] with L[i]
    i = j-1
    key = L[j]

    while i>=0:

        if key<L[i]: #shift from i to i-1
            i=i-1
            if i<0: #unless i is already 0
                L[i+2:j+1] = L[i+1:j]
                L[i+1] = key

        else: #insert key at i+1
            L[i+2:j+1] = L[i+1:j]
            L[i+1] = key
            i = -1
```

The j^{th} element of L is stored in key and it is moved as needed to the appropriate location that ensures the $1^{st} j + 1$ elements of L are sorted.

Algorithm 1 (*insertion sort*)

Correctness: Correctness can be established by first showing if 1st j elements are sorted, then after the j^{th} iteration, 1st $j + 1$ elements are sorted

Speed:

- Best case: array is already sorted, requires $N - 1$ comparisons
- Worst case: j^{th} iteration requires j comparisons for each $j, j = 1, 2, \dots, N - 1$ and there will then be $N(N - 1)/2$ comparisons
- On average, we can expect approximately $N(N - 1)/4$ comparisons
- The cost of the algorithm is $\mathcal{O}(N^2)$ operations

Can we do better?

-
- A “divide and conquer” approach worked well for search
 - Can we do something similar for sorting?
 - Let’s test the basic idea:
 - Divide array into left and right halves
 - Sort each half
 - Then merge the two halves into single sorted list
 - The cost of sorting 2 halves should be half of sorting the full array
 - Merge can be done in $\mathcal{O}(N)$ operations
 - We then save approximately $N^2/8$ comparisons during sorting (on average) but need an extra $\mathcal{O}(N)$ operations during merging
 - We can anticipate substantial savings for ‘large’ N

Merging

- I have claimed that *merge* can be done in $\mathcal{O}(N)$ operations
- How do we efficiently merge the lists L and R below into a sorted array, M?



- We will fill each element of M sequentially

- 1st element, M[0]: compare L[0] with R[0]



- 2nd element: compare L[0] with R[1]



- i^{th} element of M: compare leftmost unassigned element of L with leftmost unassigned element of R

This algorithm requires N comparisons and N assignments as claimed

Python implementation of *merge*:

- Outer loop: add one element from either L or R to M each iteration
- Keep track of leftmost indices in L and R

```
indL,indR=0,0

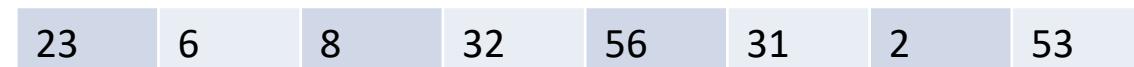
for i in range(n):
    if L[indL]<R[indR]: #add element from L to M
        value = L[indL]
        indL = indL+1
    else: #add element from R to M
        value = R[indR]
        indR = indR+1
    M.append(value)

#Check if all elements in L or R have been assigned
if indL>len(L)-1:
    M.extend(R[indR:])
    break
elif indR>len(R)-1:
    M.extend(L[indL:])
    break
return M
```

Original idea: split list into halves and then merge. But why divide the list only once?

- The sorting step requires $\mathcal{O}(N^2)$ operations, so we want this N to be small as possible
- *Merge sort*: Keep dividing until $N = 1$, then merge multiple times

$$N_s = 8$$



$$N_s = 4$$



$$N_s = 2$$



$$N_s = 1: 8 \text{ 1-element lists}$$

In the above example, N_s is the number of elements in each “sub-list”. How many times do we need to run merge?

Merge Sort

23	6	8	32	56	31	2	53
----	---	---	----	----	----	---	----

First merge pairs of sub-lists when $N_s = 1$:

23 and 6 \rightarrow [6, 23], 8 and 32 \rightarrow [8, 32], 56 and 31 \rightarrow [31, 56], 2 and 53 \rightarrow [2, 53]

Then, merging when $N_s = 2$:

[6, 23] and [8, 32] \rightarrow [6, 8, 23, 32], [31, 56] and [2, 53] \rightarrow [2, 31, 53, 56]

And finally, when $N_s = 4$:

[6, 8, 23, 32] and [2, 31, 53, 56] \rightarrow [2, 6, 8, 23, 31, 32, 53, 56]

What is the cost?

- There are $\log_2 N$ “levels”, $N/(2N_s)$ merges per level, and approximately $16N_s + 3$ operations per merge (in the worst case) so the overall algorithm requires less than $10N \log_2 N$ operations (since $\frac{N}{N_s} \leq N$). Based upon the behavior for large N , we say the algorithm requires $\mathcal{O}(N \log_2 N)$ operations

- We see that the cost of merge sort is $\mathcal{O}(N \log_2 N)$ vs. $\mathcal{O}(N^2)$ for insertion sort, so merge sort is clearly superior for long lists
- How do we implement it in Python?
- We should recognize that after the divide step, each of the sub-lists needs to be sorted
 - We can then apply merge sort to each of these sub-lists except for the case where $N_s = 1$
 - This algorithm is most easily implemented using *recursion* where the function calls itself:

```
def msort(A):
    n = len(A)
    if n==1:
        return A
    else: #call msort on Left and Right lists, then merge
        nh = int(n/2)
        L = msort(A[:nh])
        R = msort(A[nh:])
        M = merge(L,R)
        return M
```

Lecture 3

Comments on Python, recursion, searching, and sorting
Searching in dynamic datasets and hash functions

Python notes

Getting comfortable with python:

- Have command of *all* of the material in online review lectures – use exercises for self-assessment (solutions are online)
- Understand structure and purpose of functions
- Choose an editor + terminal combination for developing code. Can be *spyder* (distributed w/ anaconda), *visual studio*, *atom* + *jupyter qtconsole*. Use python 3.x (e.g. python 3.8)
- Understand binary search and merge sort codes
- Further help: list of supplementary material on Blackboard, office hours

Recursion

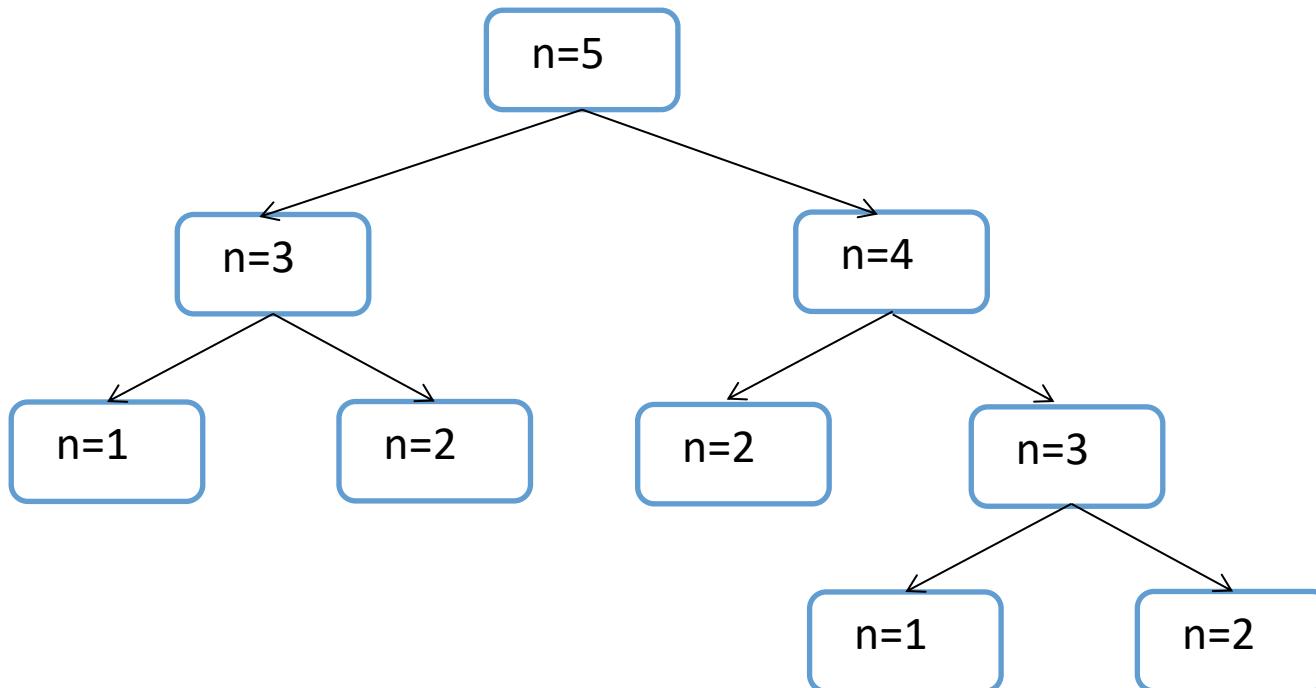
Lab 1 asked you to work with recursive functions

- It can be helpful to put together a *recursion tree*
- Consider the Fibonacci sequence:

```
def fib(n):  
    """Find nth term in Fibonacci sequence start from 0,1  
    """  
    print("n=",n)  
    if n==1:  
        return 0  
    elif n==2:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```

```
In [2]: fib(5)  
n= 5  
n= 3  
n= 1  
n= 2  
n= 4  
n= 2  
n= 3  
n= 1  
n= 2  
Out[2]: 3
```

- The recursion tree for $\text{fib}(5)$ is shown below
- The tree immediately shows this is an *inefficient* solution (can you see why?)



```
In [2]: fib(5)
n= 5
n= 3
n= 1
n= 2
n= 4
n= 2
n= 3
n= 1
n= 2
Out[2]: 3
```

-
- We won't go into the details of how the Python interpreter deals with recursive functions
 - Recursive functions can be inefficient when the number of recursive calls becomes large ("call stack" is too large)
 - This inefficiency depends on how the sequence of calls is stored in memory and how large the stored sequence is
 - Recursion isn't usually necessary, however some find it to be elegant or natural. It is a tool that should be part of a programmer's toolbox and can be useful when analyzing cost and correctness.
 - Key idea here: "divide and conquer"
 - General approach used in many algorithms
 - Need total work for sub-problems after division to be sufficiently small relative to work for original problem
 - E.g. won't help when summing elements in array

Sorting and searching

- Binary search and merge sort are two of three (or so) algorithms which any programmer must master to have finite credibility
- Quick sort is the third – it is a randomized algorithm, and on average is faster than merge sort (though both are $\mathcal{O}(N \log_2 N)$) and used more often
- Many other sorting algorithms are out there: selection, bubble, heap, ...
- See <https://www.toptal.com/developers/sorting-algorithms> for nice visualizations
- In practice, there is no need to code your own sorting routine. In Python we have the sorted and np.sort functions (among others)
 - However, if you know that your input list has a particular size and/or structure, then a different sorting algorithm may improve code performance
 - Understanding how to implement and analyze sorting and searching algorithms is essential to make progress on more complicated problems

Faster search and dynamic datasets

- In lecture 1, we saw that binary search applied to a sorted list requires $\mathcal{O}(\log_2 N)$ operations
- Can we do better?
- And what if the list is *dynamic* (i.e. if data is added/removed over time)?
- Consider the following real-world problem:
 - A startup is keeping track of unique visitors to its website each month
 - Each visitor must be assessed as “new” or “repeat”
 - Visitors who have not visited in 30+ days must be removed
 - Everything must be fast and accurate!
- Our goal: store the data in such a way that search, insertion of new visitors, and deletion of “stale” visitors are all fast (we will aim for $\mathcal{O}(1)$).

Proposal: Assign an integer to each unique visitor, and use this as a location in an array

6	8	23	32
---	---	----	----

- Say that a website had 4 visits in the last month corresponding to the integer *ids* above
- Then we would have an array (or list), X , where $X[6] = X[8] = X[23] = X[32] = 1$
- To check if a new visitor with $\text{id}=y$ is new, we check the value of $X[y]$
- Speed for search would be constant time, $\mathcal{O}(1)$
 - Here, *constant time* means that the cost does not depend on the size of the list
- But this approach has many weaknesses:
 - What about more general identification data? (Not just non-negative integers)
 - And a particularly serious weakness is inefficient memory usage: if one of these visitors had id 2395231, then suddenly our list has to be much, much larger than what we really need

-
- Let's modify our example
 - Now visitors are identified by IP addresses instead of integers
 - IP addresses are four 8-bit numbers which (in base-10) take the form: 251.31.241.80 and identify a user's location on the internet
 - There are 256^4 possible addresses so it's not sensible to maintain an array for all possible addresses
 - We could maintain a sorted list that grows or shrinks but inserting new addresses would require $\mathcal{O}(N)$ operations (in Python)
 - Alternate approach:
 - We estimate the required size for our list and design a *hash function* which takes an IP address as input and provides an index as output
 - The maximum index should be close to our size estimate
 - If the number of distinct visitors is larger than the maximum index, then we have to collect information about distinct visitors in a single location of our list (e.g. by maintaining a list of lists)
 - The design of hash functions is an important and active subject
 - What are the “desirable” properties of a hash function?

Hash function

- Consider a hash function $H(a) = i$
 - i is a non-negative integer
 - a is a numeric representation of an IP address
 - Let's say we expect about 1000 unique visitors each month
- **Desirable property 1:** The function should always assign a particular IP to the same index on repeat visits
- The second desirable property follows from the idea that H should distribute these visitors to about 1000 indices
- Since there are 256^4 possible IPs, it is impractical to design H so that unique visitors are always assigned unique indices
 - When two or more different IPs are assigned the same index, we have a *hash collision*
 - Say we constrain i so that $0 \leq i < N$.
 - **Desirable property 2:** Probability of two distinct visitors being assigned same index, i , should be $1/N$
 - The idea here is that there should not be a tendency for certain indices to be assigned to a relatively large number of visitors

An example hash function *family*:

- Represent an IP address as four integers, $a = \{a_1, a_2, a_3, a_4\}$
- Randomly choose four arbitrary integer weights, $\{w_1, w_2, w_3, w_4\}$
- Is $i = \sum_{j=1}^4 w_j a_j$ a suitable hash function?
- Not quite:
 - This can generate about 256^4 indices, and we only want around 1000
- What about $i = \sum_{j=1}^4 w_j a_j \text{ rem } 1000$?
 - We'll now have the right number of indices
 - But if there are patterns in the IPs, there could be large number of visitors assigned the same index
- It turns out that it is better to use a prime number, e.g. $i = \sum_{j=1}^4 w_j a_j \text{ rem } 997$
This will allow us to make a connection to desirable property 2

Aside on modular arithmetic

Here, we are assuming that all variables are integers

- $a \text{ rem } b$ with $b > 0$ evaluates to the remainder when a is divided by b :
 - If $b > 0$, there is a unique q and r with $0 \leq r < b$ where $a = qb + r$
 - And $r = a \text{ rem } b$
 - Example: $a = 12, b = 5$: $a \text{ rem } b = 2$
- It is often useful to equate integers which have the same remainder, and if $c > 0$:

$$a \equiv b \pmod{c} \text{ if and only if } a \text{ rem } c = b \text{ rem } c,$$

and we then say “ a and b are congruent modulo c ”

- Example: $a = 12, b = 17, c = 5$: $a \equiv b \pmod{c}$
- Optional exercise: show that $a \text{ rem } c = b \text{ rem } c$ implies that $c | a - b$ where $c | a - b$ means that “ c divides $a - b$ ”

Our proposed hash function is now: $H(a) = \sum_{j=1}^4 w_j a_j \text{ rem } 997$

- How well does this function work? To answer this, we consider the question: given two different addresses a and b , what is the probability that $H(a) = H(b)$, and is this probability $\leq \frac{1}{997}$?

Let's assume that $a_4 \neq b_4$ and that the weights are four non-negative integers less than 997 selected uniformly at random

- We want to find the probability that $\sum_{j=1}^4 w_j a_j \equiv \sum_{j=1}^4 w_j b_j \pmod{997}$
- Due to the general property: $a \equiv b \pmod{n}$ implies $a + c \equiv b + c \pmod{n}$, we can rearrange the congruence above as:

$$\sum_{j=1}^3 w_j (a_j - b_j) \equiv w_4 (b_4 - a_4) \pmod{997}$$

- We'll now assume that the 1st 3 weights have been specified. The LHS is then just some integer, c , and we need to "solve" the following congruence for w_4 :

$$w_4 (b_4 - a_4) \equiv c \pmod{997}$$

-
- To solve for w_4 , use the following result from modular arithmetic:

Let p be a prime. If k is not a multiple of p , then there exists an integer $k^{-1} \in \{1, 2, \dots, p - 1\}$ such that: $k \cdot k^{-1} \equiv 1 \pmod{p}$

- Using this result, we can then show that if $c \equiv dk \pmod{p}$ then $ck^{-1} \equiv d \pmod{p}$ if k is not a multiple of p and $d < p$
- In our example, $p = 997$, $k = (b_4 - a_4)$, $d = w_4$, so the condition for a hash collision is:
 $w_4 \equiv (b_4 - a_4)^{-1}c \pmod{p}$
or
 $w_4 \text{ rem } p = (b_4 - a_4)^{-1}c \text{ rem } p$ (Note that $(b_4 - a_4)$ cannot be a multiple of p)
- We required that $w_4 \in \{0, 1, 2, \dots, p - 1\}$ so $w_4 \text{ rem } p = w_4$ and:

$$w_4 = (b_4 - a_4)^{-1}c \text{ rem } p$$

-
- So there is precisely one possible value of w_4 which leads to a hash collision
 - Then, since w_4 is chosen uniformly at random, the probability of a hash collision is $P(H(a) = H(b)) = 1/p$.
 - This is exactly the behavior that we want to satisfy desirable property 2. And the function is deterministic once the weights are selected so desirable property 1 is also satisfied.
 - Optional exercise: is it important that we assumed that the 4th elements in the addresses did not match, $a_4 \neq b_4$?

Next, we will:

- See how to use hash functions to “beat” binary search
- Discuss their implementation in Python
- And next week, we will see an application in bioinformatics

Lecture 4

Hash tables and Python dictionaries
Analyzing computational cost

Videos

This week we have two pre-recorded videos:

- A 13 minute video on analyzing computational cost.
- An optional 12 minute video on the cost of merge sort. This repeats the analysis of merge sort from lecture but provides more detail.

Searching

How do we use hash functions to search more efficiently (and to “beat” binary search)?

- The idea is to store the data in a *hash table* where the index produced by the hash function sets the data location in the table
 - Then, given new data, we compute the index and check the table to see what, if anything, is stored at that location
 - And this all needs to be done efficiently!
- This is essentially the same as what I sketched out last lecture as a list of lists, and we will now review the construction of this data structure
- This work will also lead to a “proper” understanding of what a Python dictionary is!

-
- Let's think about hash tables in the context of the IP address example:

A startup is keeping track of unique visitors to its website each month

- Each visitor must be assessed as “new” or “repeat”
 - Visitors who have not visited in 30+ days must be removed
 - Everything must be fast and accurate
-
- We want to “beat” binary search when deciding if a visitor is new, and the addition/removal of data should also be efficient

What is the general workflow?

0. Choose the prime (N) for your hash function, and initialize a list (or dictionary) where you will store addresses. This is the hash table.
1. Given an IP address, compute an index using your hash function
2. Check if the visitor is new
3. If new: append the IP address (and additional data) at the corresponding location in the hash table
4. If not new, update the associated data as needed at the appropriate location in the hash table

Example: say that our hash table is a list of lists, L . Also say that the address a is assigned $H(a) = i$. Then, we check if a is in $L[i]$, and if it is not, we append it to the list stored at, $L[i]$

- Let's assume that we have stored M IP addresses at N locations (possibly with $M > N$ though this is undesirable)
- With our hash function (or, in general, a well-designed hash function), the expected number of addresses assigned to each index will be M/N IPs

Our initial motivation was to find something faster than binary search. What is the cost of using a hash table?

- *Search* (lookup): Given an IP, find it in the table
 - Evaluate the hash function and obtain an index
 - Check if one or more items are stored at index, i .
 - If $\text{len}(\text{L}[i]) > 1$, iterate through items (if $\text{len}(\text{L}[i]) = 1$, check $\text{L}[i]$ for match)
 - Overall, cost is $\mathcal{O}(\max(1, M/N))$
 - For a well-designed hash table and function, the cost will be close to $\mathcal{O}(1)$!
- *Insert*: Add a new IP to table
 - Search and verify it is new
 - Then append at computed index if IP is not already present
 - Cost is again $\mathcal{O}(\max(1, M/N))$
- *Delete*: Remove an IP
 - This is a little more complicated than *Insert*. After search, you can: set the number of visits to zero ($\mathcal{O}(1)$), replace the entry with an empty list ($\mathcal{O}(1)$), or delete the entry (on average, $\mathcal{O}\left(\max\left(1, \frac{M}{N}\right)\right)$); when including the cost of search each of these approaches is $\mathcal{O}(\max(1, M/N))$

Summary:

- Binary search: Cost is $\mathcal{O}(\log_2 N)$ but requires the maintenance of a sorted list/array
- Hash table: $\mathcal{O}(1)$ for search as well as maintenance provided that the hash function and table are both well-designed!
- What does this look like in Python?
 - Let's move away from the IP problem to a more general task
 - Consider input that may be real numbers or even non-numeric

Python provides a hash function that returns an integer for (almost) any input

- For an integer, i , $\text{hash}(i) = i$ (provided that $|i| \leq 10^{18}$)
- For two inputs, if $a==b$ then $\text{hash}(a)=\text{hash}(b)$
 - For example, $\text{hash}(3.0) = \text{hash}(3) = 3$
- For non-integers, the function is less predictable:

```
In [2]: hash('math')
Out[2]: -6564006556742056588
```

```
In [3]: hash('maths')
Out[3]: 29053059181571155
```

- As noted earlier, we can use a list-of-lists to build a hash table with the output of hash modulo an appropriate prime providing location indices

Python dictionaries

- But we don't have to build our own hash table!
- Python dictionaries *are* hash tables
(technically, they are “associative arrays”)
- Dictionaries are containers where each element is a key-value pair
 - A key can be considered a label or id
 - Example:

```
In [6]: key = "123.45.241.12"
```

```
In [7]: value=[14,1,2022,20]
```

```
In [8]: d = {key:value}
```

```
In [9]: d
Out[9]: {'123.45.241.12': [14, 1, 2022, 20]}
```

```
In [10]: d["123.45.241.12"]
Out[10]: [14, 1, 2022, 20]
```

-
- Python applies a hash function to keys to know where to store them and where to look for them
 - In the case of a hash collision, it generates a new location for the key
 - This is *open addressing* while our previously-described approach is *chaining*
 - It will automatically re-size dictionaries as well when they are close to full

Important dictionary operations

Constant time $\mathcal{O}(1)$:

In [36]: `d = dict()` #initialize a new dictionary

In [37]: `d[key] = value` #associate key with value and store in d

In [38]: `d[key]` #value associated with key in d, raises KeyError if key has not been added to d
Out[38]: [14, 1, 2019, 20]

In [39]: `x=1`

In [40]: `d.get(key,x)` #value associated with key if key is present, otherwise x

In [42]: `key in d` #is key in d?
Out[42]: True

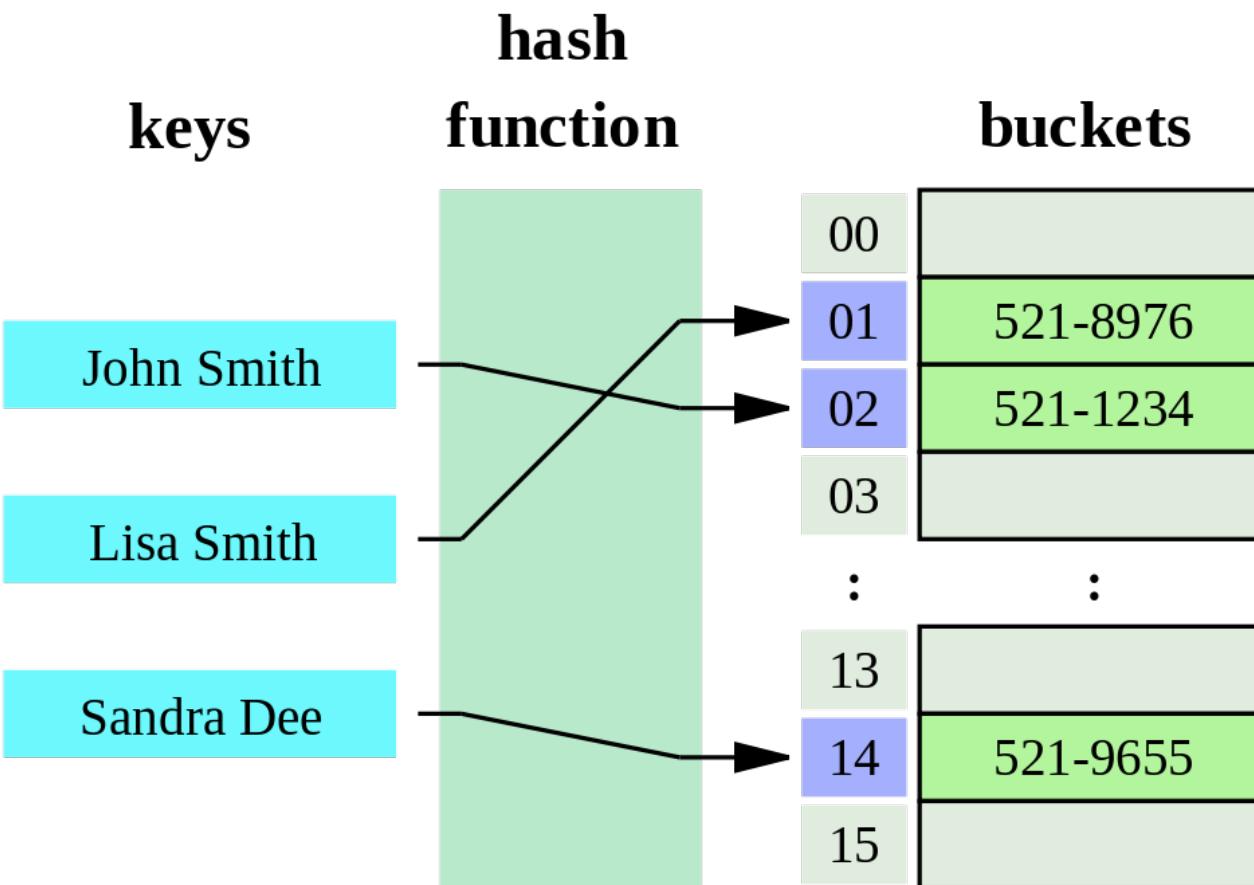
In [43]: `len(d)` #number of key-value pairs in d
Out[43]: 1

In [44]: `del d[key]` #remove key (and its associated value) from d

linear time $\mathcal{O}(N)$: In [45]: `for key in d:` # iterate over keys in d

This is exactly what we need to build and maintain a hash table.

- Check that you understand how the figure below corresponds to how a Python dictionary is built



Analyzing computational cost

- We have discussed the cost of searching and sorting from a theoretical perspective, and I have also listed the costs of a number of dictionary operations
 - These discussions have concluded with statements of the (asymptotic) running time in the form $\mathcal{O}(f(N))$ where N is the problem size
 - E.g. binary search is $\mathcal{O}(\log_2(N))$
 - But what does $\mathcal{O}(f(N))$ really mean?

-
- First, a bit of jargon:
 - “Running time”, “time complexity”, and “computational cost” all refer to the dependence of the number of operations on the problem size (typically when the problem size is “large”). I will use these 3 terms interchangeably.
 - “Wall time” refers to the number of seconds, minutes, or hours you have to wait while code runs (this will be revisited in lab 2)
 - When analyzing algorithms, we are primarily interested in the theoretical running time, and in particular the *asymptotic running time* when the problem size is large
 - It is then the programmer’s responsibility to ensure the wall time of their implementation of an algorithm is consistent with the theoretical running time

Analyzing running time

- On a basic level, analyzing running time is a matter of counting
 - E.g. a additions, b assignments, c comparisons/iteration
 - Here, an assignment is simply setting the value for a variable (e.g. $x=3$), and a comparison is the evaluation of a relational operator (e.g. $x < 3$)
 - Then, with n iterations, cost is $a + b + cn$
 - The number of operations may vary based on input, so we often need to consider worst-case, best-case, and/or average cost

-
- Example: linear search with size- N input
 - Best case: 1 iteration, (1 assignments, 1 comparison, 1 addition)/iteration, 3 operations
 - On average, $3(N + 1)/2$ operations
 - Worst case: $3N$
 - Here, the best case is not particularly helpful, and the worst and average cases provide similar insight into the algorithm's efficiency
 - The key challenges when counting operations are to:
 - correctly assess operations involving containers (lists, arrays, dictionaries...) which can contain several elements
 - Account for the number of iterations in loops (or list comprehensions, ...)

-
- So $x+2$ will be 1 operation if x is just a number, but will be N operations if it is an N -element array
 - Let's look at a more complicated example which we will see later in the module

```
n = Q.pop(0)  
Q.append(v)
```

- Q is a list containing say, M , elements and the question is, how do the costs of the pop and append methods depend on M
 - Appending an item to the end of a list *does not* depend on M (nor does removing an item from the end)
 - However, removing an item from the front of a list *does!* This cost scales linearly with the list size as does the cost of adding an item to the front

Asymptotic running time

We still have the question of what $\mathcal{O}(1)$ or $\mathcal{O}(f(N))$ means where N is an indicator of the problem size

The cost, $C(N)$, is $\mathcal{O}(f(N))$ if there is a positive constant a , and a positive integer, N_0 , where for all $N \geq N_0$, $C(N) \leq a f(N)$

This is “Big-O” notation and establishes an upper bound

- Example:
 - if $C = 11\log_2 N + 8$ (binary search)
then C is $\mathcal{O}(\log_2 N)$
 - Why? $11\log_2 N + 8 \leq 19\log_2 N$ if $N \geq 2$, and we can choose $a = 19$
- This provides an upper bound and describes the worst-case scenario
- If the worst-case scenario is close to the best-case scenario, no need for other approaches
- An algorithm that is $\mathcal{O}(N)$ will also be $\mathcal{O}(N^2)$, but it is bad practice (especially on projects and exams) to not provide a reasonably tight upper bound (e.g. don’t say an $\mathcal{O}(\log N)$ algorithm is $\mathcal{O}(N)$)

-
- The rule of thumb when constructing a big-O estimate is drop leading coefficients and higher order terms
 - We ignore the higher order terms because they become unimportant when the problem size becomes large
 - We ignore the leading coefficients, because we don't really know what the relative cost of distinct operations are
 - E.g. how much slower or faster is the addition of two small integers relative to the multiplication of two real numbers?
 - Additionally, answers to these sorts of questions can depend on the hardware and software/language that you are using
 - This is why I sometimes say things like it “doesn't matter” if you count 8 or 9 or 10 operations for a code snippet

There are two other conventions for describing the cost:

- Cost, $C(N)$ is $\Omega(f(N))$ if there is a positive constant a , and a positive integer, N_0 , where for all $N \geq N_0$, $C(N) \geq a f(N)$

This provides a lower bound and describes the best-case scenario

- Cost is $\Theta(f(N))$ if and only if it is $\Omega(f(N))$ and $\mathcal{O}(f(N))$
- So we have “big-O”, “big-Omega”, and “big-Theta” which are the three standard approaches to characterizing algorithm efficiency
- However, big-O estimates are used most often with the universal understanding that the bound is sensibly tight

Lecture 5

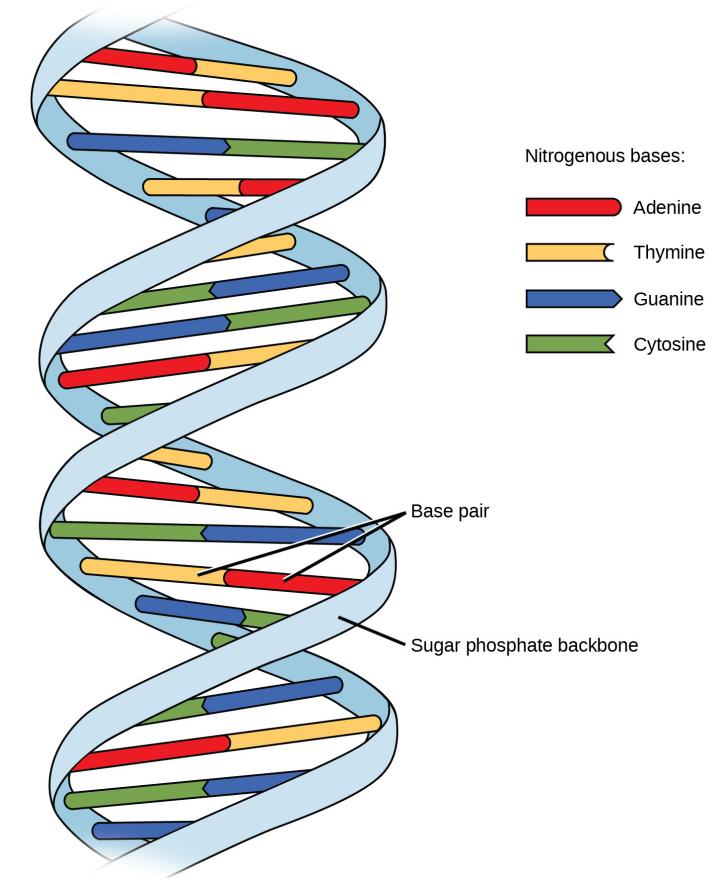
Genetic code

The pattern search problem and naïve pattern search

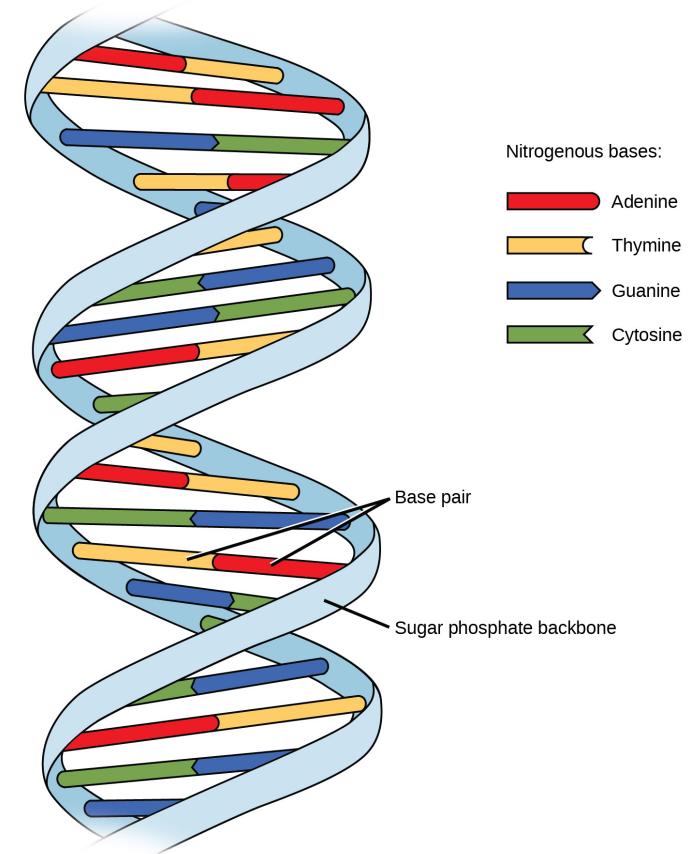
Improving upon the naïve method and the Rabin Karp algorithm

Genetic code

- We now move to thinking about searches within genomes
- First, a little background science...
- DNA (and RNA) consists of two “strands” arranged in a double helix and connected with covalent bonds
- Each strand contains a sequence of nucleotides which consist of 1) a sugar molecule 2) a phosphate group and 3) a nitrogenous base
- There are 4 nitrogenous bases:
 - Adenine
 - Cytosine
 - Guanine
 - Thymine(RNA has Uracil in place of Thymine)



- Adenine bonds with Thymine and Guanine bonds with Cytosine
- So if one strand contains the sequence GCTTCA the other strand will contain CGAAGT in the corresponding location
- During cell division, each daughter cell gets one strand
 - And then the needed 2nd strand can be constructed so A's pair with T's and C's pair with G's



https://upload.wikimedia.org/wikipedia/commons/a/a1/229_Nucleotides-01.jpg

-
- *Codons* consist of three consecutive DNA bases and contain code for synthesizing amino acids
 - 64 possible codons, but there are only 20 essential amino acids specified by DNA
 - Proteins are built from amino acids
 - A protein is a complex molecule whose structure is specially suited for specific tasks or sets of tasks (they “make things go”)
 - Gene sequencing involves:
 - Extracting the sequence of bases from DNA samples
 - Investigating the proteins or functions associated with codons and their sequences

Pattern finding

- A fundamental computational problem:
 - Given a DNA sequence, search for a pattern
 - Both the sequence and the pattern can be extremely long
 - The number of patterns can also be large
- Fruit fly genome: 139.5 million base pairs



- Humans: 3 billion or 6 billion pairs (depending on the cell type)



- How can we *efficiently* search for patterns?

-
- In general, we want to find trends and patterns in gene sequences
 - Examples:
 - ATG is a *start codon* and is present at the beginning of every DNA substring providing code for a protein (in eukaryotes)
 - Frequently occurring patterns point to important portions of a sequence and/or important substrings
 - The relative amount of cytosine and guanine can be used to find where in the sequence replication starts

Pattern search

- Problem setup:
 - Specify a n -character sequence, S , and a m -character pattern, P
 - Find all locations in S where P occurs
- Example:

$S = \text{ATGTTGTACCGTATCGG}$

$P = \text{GTA}$

$n = 16, m = 3$

What would you say is the simplest way to solve this problem?

Naïve pattern search

“Naive” approach:

- Loop through S one character at a time
 - Check for matches with P one character at a time, and break the checking step after first mis-match

-
- The code snippet below contains the core algorithm:

```
#Set sequence
S = 'ATGTTGTACCGTATCGG'
n = len(S)

#Set pattern for search
P = 'GTA'
m = len(P)

for ind in range(0,n-m+1):
    matching=True
    #Compare sub-string to pattern
    for count,indp in enumerate(range(ind,ind+m)):
        if P[count] != S[indp]:
            matching=False
            break
    #Update list when match found
    if matching:
        imatch.append(ind)
        print("Match found, i=",ind)
```

In [15]: run naive_search

Match found, i= 5

Match found, i= 10

-
- What is the cost of the naïve search algorithm? For convenience, let's assume that $n \gg m$ which describes most cases of interest
 - Worst-case cost: $\mathcal{O}(mn)$ operations when there are many near-misses
 - Can we do better?
 - Binary search?
 - First store the $n - m + 1$ length- m strings in a numeric form
 - Then sort the resulting list with $\mathcal{O}(n \log_2(n))$ cost
 - Then $\mathcal{O}(\log_2(n))$ for each search
 - But this requires storing n length- m strings/arrays and for large mn , the memory requirement may be excessive
 - Hash table?
 - Again we could store the n length- m strings
 - This will certainly be faster and simpler than the binary search method, but memory usage may still be problematically large when mn is large

Rolling hash function

- A (partial) solution:
 - Use a *rolling* hash function
 - Compute hash for pattern, P
 - Then apply function sequentially to each length- m substring in S
 - We will then have $n - m + 1$ hash function evaluations and comparisons
 - And memory usage will also be $\mathcal{O}(m + n)$

-
- What is a rolling hash function?
 - First, genetic sequences can be rewritten in base 4
 - A=0, C=1, G=2, T=3
 - A simplistic function – convert sequence from base 4 to base 10
 - Example: $S = \text{GCTAT} \rightarrow X = 21303$
$$H(X) = 2 * 4^4 + 1 * 4^3 + 3 * 4^2 + 0 * 4^1 + 3$$
 - Or more generally, evaluate a $(m - 1)^{\text{th}}$ -order polynomial for each length- m sequence of consecutive integers in X
 - This doesn't really help – there will be $\mathcal{O}(m)$ operations for all $(n - m)$ sequences. The hash function evaluation is too expensive.
 - The key is to update the hash function for each new substring rather than recompute it from scratch

-
- Let x_i be the i^{th} length- m sub-string in S mapped to integers in base 4. So if $m = 3$ and $S=AGTCA$, then $x_3=310$
 - Also let $x_{i,j}$ be the j^{th} number in x_i
 - If $H(x_i)$ is computed as before:

$$H(x_i) = x_{i,1} 4^{m-1} + x_{i,2} 4^{m-2} + \dots + x_{i,m-1} 4 + x_{i,m}$$

- Then:
$$H(x_{i+1}) = H(x_i) * 4 - x_{i,1} 4^m + x_{i+1,m}$$
- So, the computation of $H(x_i)$ for $i > 1$ will require 4 rather than $\sim 2m$ operations per hash evaluation
- There is still one potential problem – when m is large, integers will become large, and arithmetic can become slow (this is programming language dependent)
 - This will of course be more important for problems in, say, base-26 than in base-4
- This problem can be alleviated with the modulo operator...

Rabin Karp algorithm

- So we define $h(x_i) = H(x_i) \text{ rem } q$ with q a large prime number and as before:

$$H(x_i) = x_{i,1}4^{m-1} + x_{i,2}4^{m-2} + \cdots + x_{i,m-1}4 + x_{i,m} \quad (\text{Rabin fingerprint})$$

- We can then use rules from modular arithmetic to simplify the calculation of h :

$$h(x_{i+1}) = [h(x_i) * 4 - x_{i,1}(4^m \text{ rem } q) + x_{i+1,m}] \text{ rem } q$$

- However, introducing the *rem* operator to avoid large integers also introduces the possibility of hash collisions
 - If hashes match, the corresponding strings have to be directly compared (as in naïve search)
 - The worst-case cost then becomes $\mathcal{O}(mn)$
- We can now summarize the *Rabin Karp* algorithm for pattern search (within strings)

-
1. Convert S and P to base 4 (or 26 or ...): $S \rightarrow X$ and $P \rightarrow Y$ in the code below

```
def char2base4(S):
    """Convert gene test_sequence
    string to list of ints
    """
    c2b = {}
    c2b['A']=0
    c2b['C']=1
    c2b['G']=2
    c2b['T']=3
    L=[]
    for s in S:
        L.append(c2b[s])
    return L
```

```
X = char2base4(S)
Y = char2base4(P)
```

-
1. Convert S and P to base 4 (or 26 or ...): $S \rightarrow X$ and $P \rightarrow Y$
 2. Compute hashes of Y and 1st m elements of X and compare:

```
def heval(L,Base,Prime):
    """Convert list L to base-10 number mod Prime
    where Base specifies the base of L
    """
    f = 0
    for l in L[:-1]:
        f = Base*(l+f)

    h = (f + (L[-1])) % Prime
    return h

ind=0
hp = heval(Y,Base,Prime)
imatch=[]
hi = heval(X[:m],Base,Prime)
if hi==hp:
    if match(X[:m],Y):    #Character-by-character comparison
        imatch.append(ind)
```

Note: `match(A,B)` is a function which carries out a character-by-character comparison of equal-length input strings A and B , and `heval` is using *Horner's method* for efficiently evaluating polynomials.

3. Iterate through remainder of X and:

- Each iteration, update $h(x_i)$ and compare to $h(Y)$
- If they match, compare appropriate portion of S with P
- If these also match, add location to list of matches. The second comparison is needed to protect against hash collisions

Update formula:
$$h(x_{i+1}) = [h(x_i) * 4 - x_{i,1}(4^m \text{ rem } q) + x_{i+1,m}] \text{ rem } q$$

Code:

```
bm = (4**m) % q

for ind in range(1,n-m+1):

    #Update rolling hash
    hi = (4*hi - int(X[ind-1])*bm + int(X[ind-1+m])) % q

    if hi==hp: #If hashes match, check if strings match
        if match(X[ind:ind+m],Y): imatch.append(ind)
```

Notes:

- We have pre-computed b_m as it would be inefficient to repeatedly compute it within the main loop. Generally, we should avoid repeatedly computing the same thing.
- The worst-case cost is $\mathcal{O}(nm)$; this occurs when there are many hash matches and the comparison of strings requires $\mathcal{O}(m)$ operations.
- The benefits of this algorithm will be observed when there are many “near-misses”, i.e. many substrings where the 1st a letters match the pattern with a close to but less than m or when there are many (possibly long) patterns whose hashes can be pre-computed

-
- The Rabin-Karp algorithm is one of several algorithms that have been developed for string matching that improve upon the naïve method (in some cases)
 - It isn't "too" old – introduced in 1987
 - There are many applications outside of bioinformatics:
 - Plagiarism detection
 - "Find" function in software applications

Lecture 6

Getting started with networks and NetworkX
Graph search and breadth-first search

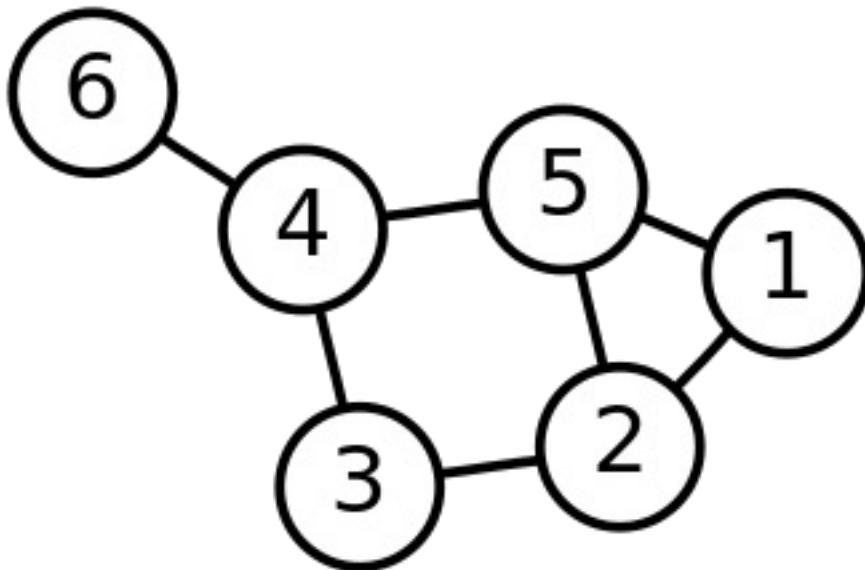
Quick intro to networks

We have thought about “search” from a few different perspectives, and we will now consider one last perspective: searching in networks (graphs)

- Why are we interested in searching in networks?
 - We will see that this allows us to learn about a network’s structure
 - Why do we care about network structure? It is very useful to model many important complex systems as networks. Examples include real-world and online social networks, transportation networks, the human brain, and the world-wide-web
- The next few lectures will focus on learning how to efficiently explore networks and their structure
 - This will lead us to another “classic” algorithm and deepen our understanding of Python
- I’ll first introduce some basic terminology

Networks: basics

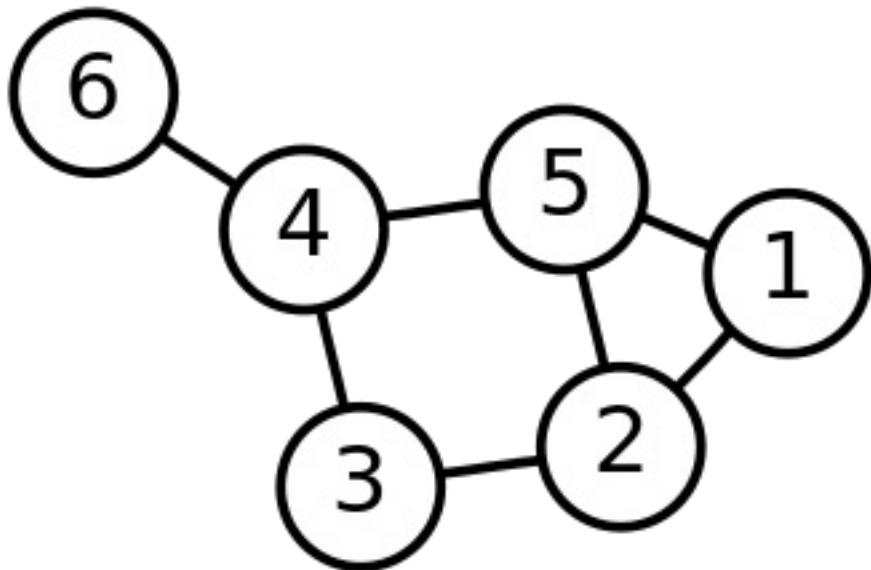
- A network has N nodes and L links between nodes
- Each node has a label, e.g. $1, 2, \dots, N$
- Then a link between node i and j can be represented as (i, j) or $i - j$



Example: This graph has 6 nodes and 7 links

- Node 1 has two edges: $(1, 2)$ and $(1, 5)$
- The graph can be represented by the *adjacency matrix*, \mathbf{A}
- $A_{ij} = 1$ if there is link between nodes i and j and is zero otherwise

Networks: basics



The adjacency matrix for our example is:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

This is an *undirected* graph – each link can be traversed in either direction. Then $A_{ij} = A_{ji}$ and \mathbf{A} is symmetric for any undirected graph.

Networks: basics

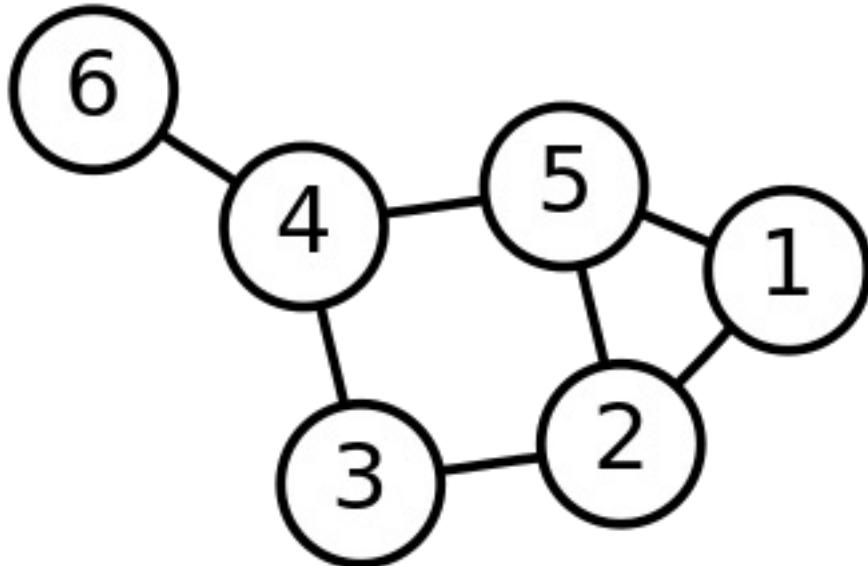
We can also represent connected portions of a graph with an *edge list*. For our example, we would have:

```
elist=[(1,2),(1,5),(2,3),(2,5),(3,4),(4,5),(4,6)]
```

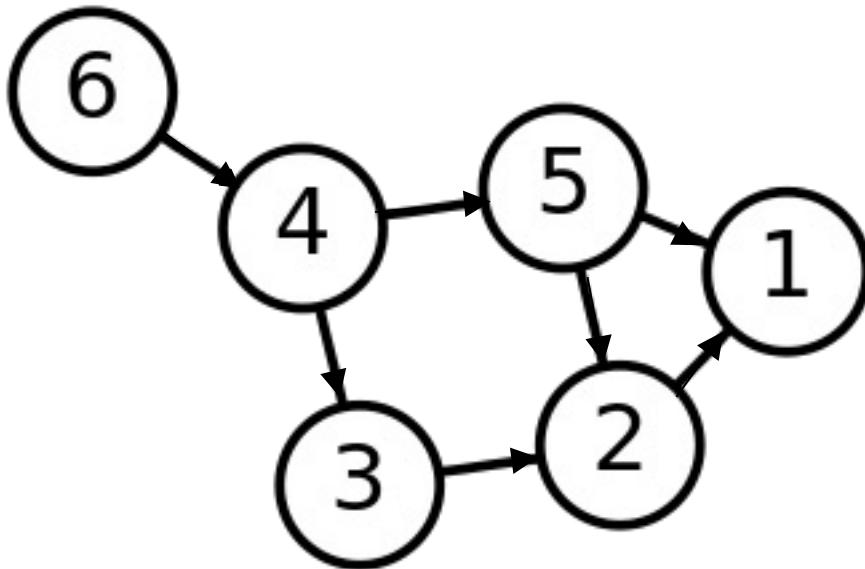
- There are representations of graphs as well. One version of an *adjacency list* is a list of lists where the i^{th} sublist contains the neighbors of node i

- The *degree* of a node is the total number of links connected to it:
 $k_1 = 2, k_5 = 3, \dots$

Check your understanding: what is k_2 ?



Networks: basics

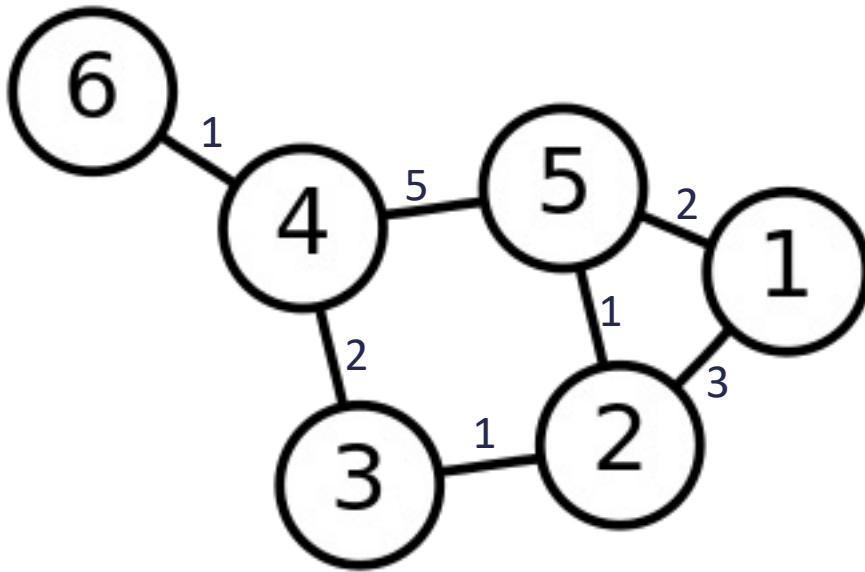


- Networks can also be *directed*
- Then $A_{ij} = 1$ if there is a link *to i from j*:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- \mathbf{A} is then *not* necessarily symmetric.

Networks: basics



Networks can be weighted (e.g. weights can correspond to distances or travel times in transportation networks) with a weight matrix, \mathbf{W} :

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 2 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 5 & 1 \\ 2 & 1 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- Networks can be both directed and weighted

NetworkX

Network	Nodes	Links	Directed / Undirected	N	L	$\langle k \rangle$
Internet	Routers	Internet connections	Undirected	192,244	609,066	6.34
WWW	Webpages	Links	Directed	325,729	1,497,134	4.60
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594	2.67
Mobile-Phone Calls	Subscribers	Calls	Directed	36,595	91,826	2.51
Email	Email addresses	Emails	Directed	57,194	103,731	1.81
Science Collaboration	Scientists	Co-authorships	Undirected	23,133	93,437	8.08
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908	83.71
Citation Network	Papers	Citations	Directed	449,673	4,689,479	10.43
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802	5.58
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930	2.90

Table 2.1

Canonical Network Maps

The basic characteristics of ten networks used throughout this book to illustrate the tools of network science. The table lists the nature of their nodes and links, indicating if links are directed or undirected, the number of nodes (N) and links (L), and the average degree for each network. For directed networks the average degree shown is the average in- or out-degrees $\langle k \rangle = \langle k_{in} \rangle = \langle k_{out} \rangle$ (see Equation (2.5)).

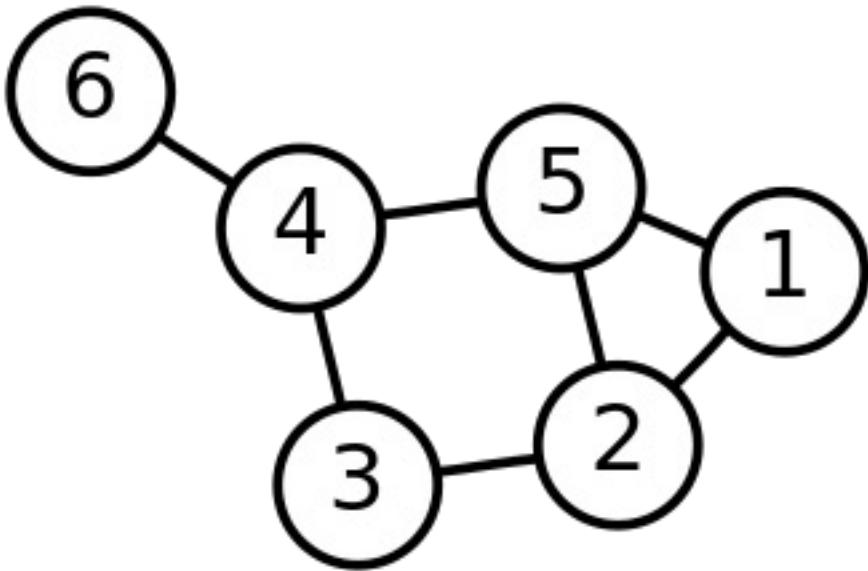
We are generally interested in large *complex* networks

Analysis of such networks can be complicated and expensive (classical example: computing shortest path between pairs of nodes)

The NetworkX package provides a suite of tools for working with complex networks

Conveniently, it provides a *Graph* datatype which stores the details of the graph structure (nodes and edges)

NetworkX: basics



- Let's work with this graph in NetworkX
- First, import the module, and initialize a graph:

```
In [55]: import networkx as nx
```

```
In [56]: G = nx.Graph()
```

- There are numerous methods for building a graph

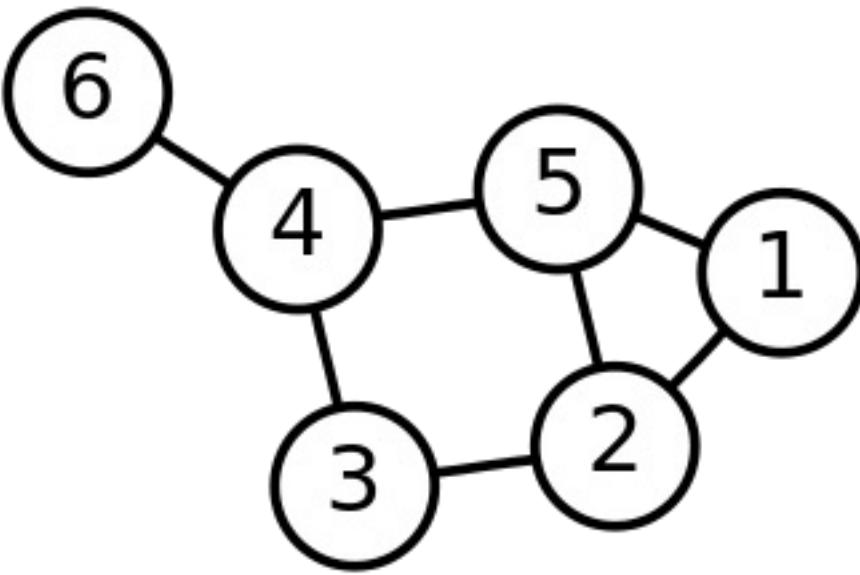
```
In [57]: G.add_edge(1,2)
```

```
In [58]: list(G.edges())
```

```
Out[58]: [(1, 2)]
```

```
In [59]: list(G.nodes())
```

```
Out[59]: [1, 2]
```



We can also add several edges (or nodes) at once:

```
In [65]: e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]
```

```
In [66]: G.add_edges_from(e)
```

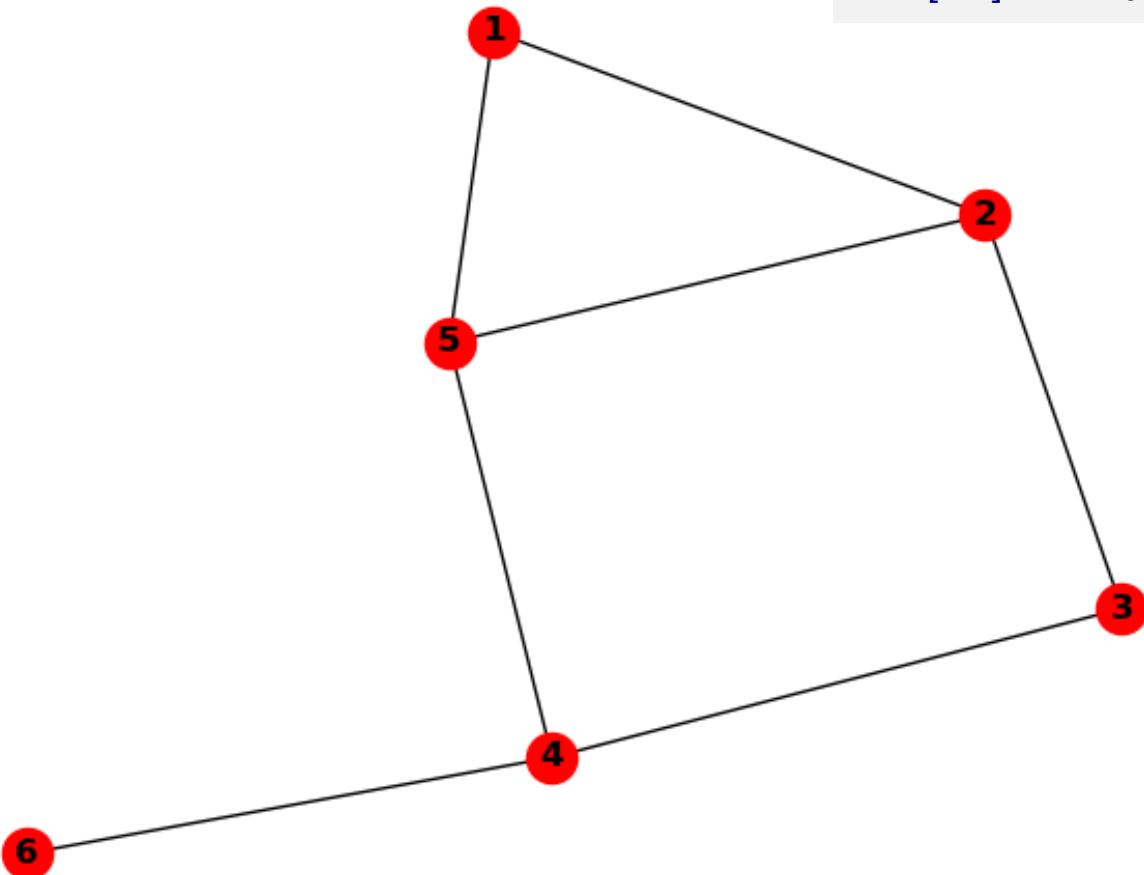
```
In [67]: list(G.edges())
```

```
Out[67]: [(1, 2), (1, 5), (2, 3), (2, 5), (5, 4), (3, 4), (4, 6)]
```

```
In [68]: list(G.nodes())
```

```
Out[68]: [1, 2, 5, 3, 4, 6]
```

Use nx.draw to visualize the network:



```
In [69]: import matplotlib.pyplot as plt
```

```
In [70]: plt.figure()
```

```
Out[70]: <matplotlib.figure.Figure at 0x1515e3fef0>
```

```
In [71]: nx.draw(G, with_labels=True, font_weight='bold')
```

-
- Given a NetworkX graph, it is straightforward to extract the neighbors of any node

```
In [72]: list(G[2])
```

```
Out[72]: [5, 3, 1]
```

or:

```
In [73]: list(G.adj[2])
```

```
Out[73]: [5, 3, 1]
```

- NetworkX contains many, many tools for working with and analyzing networks. An important example for us is `nx.shortest_path` which finds a route between two nodes traversing the fewest possible number of links

```
In [20]: nx.shortest_path(G, source=2, target=6)
```

```
Out[20]: [2, 3, 4, 6]
```

→ Very important in study of algorithms
(*upcoming lectures*)

The *distance* between two nodes is the number of links in the shortest path between the nodes. For this example, the distance between nodes 2 and 6 is 3.

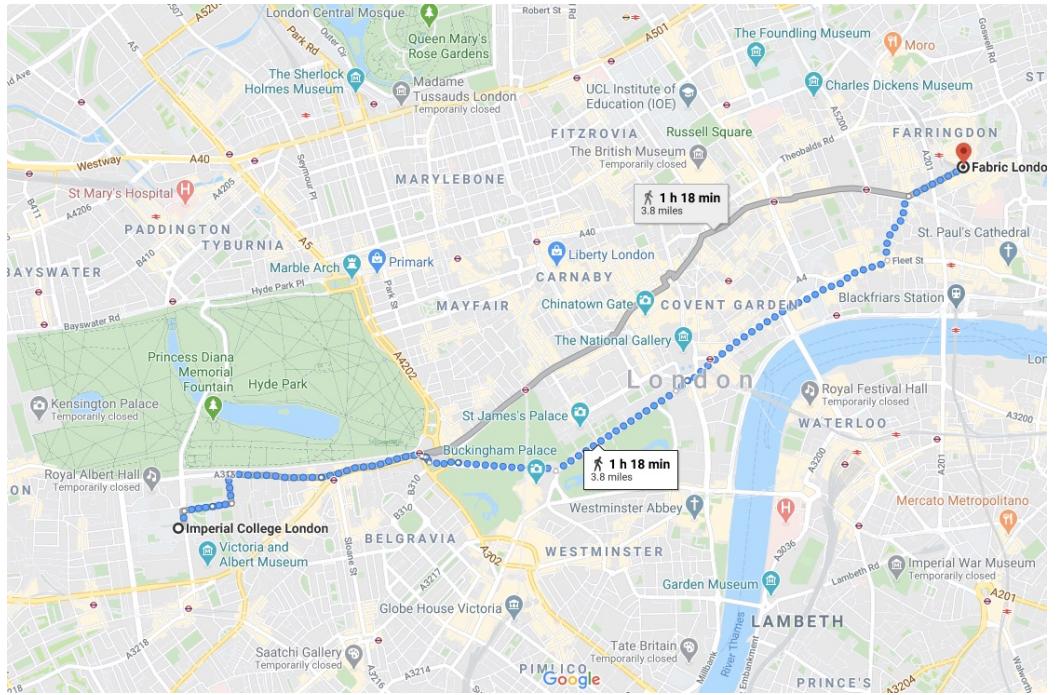
More information about NetworkX:

- Online tutorial: <https://networkx.github.io/documentation/stable/tutorial.html>
- Online reference section: <https://networkx.github.io/documentation/stable/reference/index.html>
- Use NetworkX 2.x (I'm using 2.4/2.9)

More information about networks: see chapter 2 of *Network Science*:
<http://networksciencebook.com/chapter/2>

Graph search

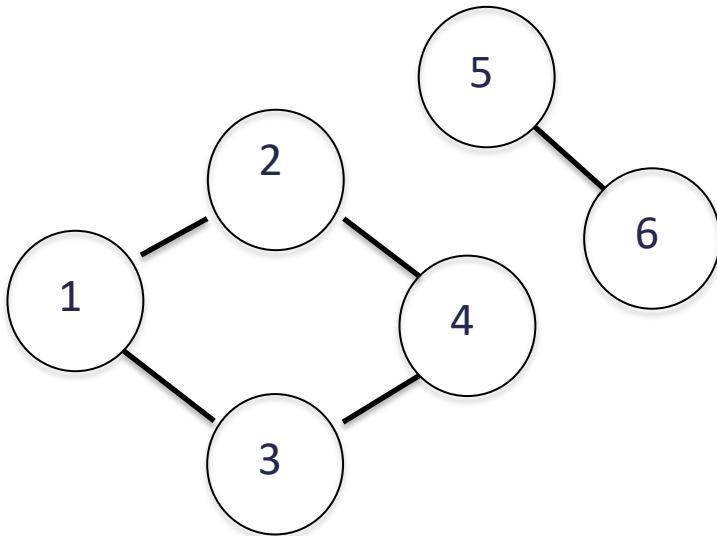
- Graph search is a class of algorithms for finding paths in graphs
- Is graph search important?
 - Essential for effective navigation, e.g. finding shortest paths
 - Also provides basic information about network structure, e.g. which nodes are (un)reachable from a given node?
- Many networks of interest have $1e5+$ edges, it is essential that search algorithms and their implementation are efficient!



- Basic idea: Given a graph, G , and a source node, s , find all other nodes that can be reached from s
- Basic approach:
 - Label s as “explored” and all other nodes as “unexplored”

While there is at least one edge between an explored and unexplored node:

Select one such edge and re-label the unexplored node as explored



Claim: Upon completion, a node is labeled explored if and only if a path exists between it and the source node

There are 2 cases to consider here:

Case 1: Is it possible for a reachable node to be labeled as “unexplored” at the termination of the search?

Case 2: Is it possible for an unreachable node to be labeled as “explored” upon termination?

Case 1: Is it possible for a reachable node to be labeled as “unexplored” at the termination of the search?

- By carefully thinking about the algorithm, we can establish that each unexplored node a distance d from the source will at some point have a link to an explored node with distance $d-1$ and will then be labeled as explored when that link is examined

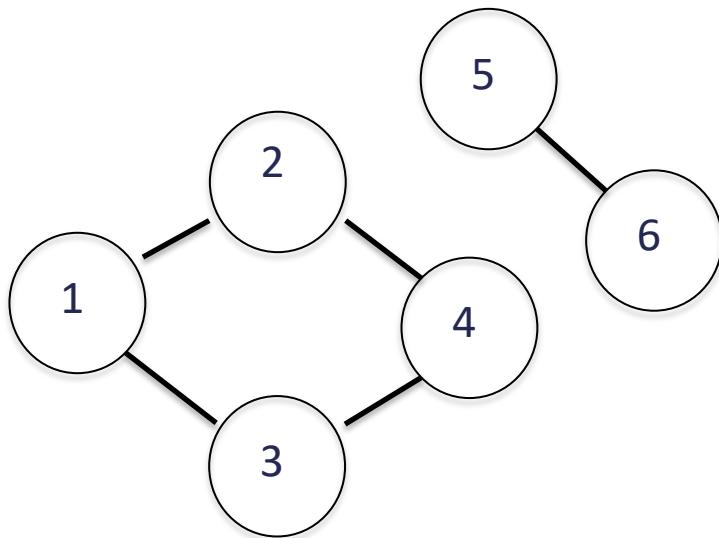
Case 2: Is it possible for an unreachable node to be labeled as “explored” upon termination?

- Here, we can consider the set of all nodes reachable from the source (A) and the set of all nodes reachable from an unreachable node (B). These sets must be disjoint, and there cannot be a link connecting nodes in the two sets. So the search will never encounter a link between an explored node and a node in B.

-
- Basic approach:
 - Label s as “explored” and all other nodes as “unexplored”

While there is at least one edge between an explored and unexplored node:

Select one such edge and re-label the unexplored node as explored



Implementation: Depends on how edges are selected each iteration

- Depth-first search – aggressively move into the graph (1-2, 2-4)
- Breadth-first search – consider one “layer” at a time (1-2, 1-3)

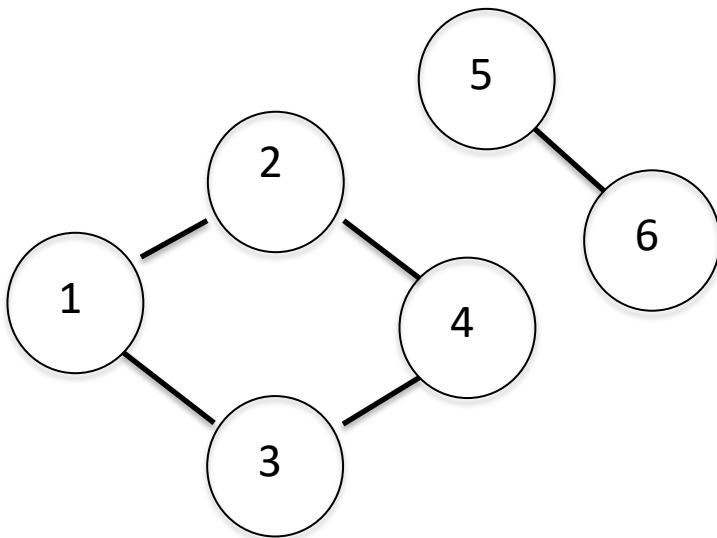
-
- Breadth-first search (BFS) and depth-first search (DFS) can both find all reachable nodes from a source in linear time
 - They also each have distinct applications which only one or the other can be used for
 - We will first focus on BFS, then move to DFS, and finally think about how graph search should be modified for weighted graphs
 - Two points to think about as we go through these algorithms:
 - How should we store “information” to ensure an efficient search?
 - What does this mean in terms of a Python implementation?

Breadth-first search

Python implementation:

- Specify: Graph and source node
- Maintain: 1) list of nodes, 2) list of labels for nodes (-1=unexplored and 1=explored) and 3) *queue* of explored nodes which may have links to unexplored nodes
- Initialize the queue with the source node and mark it as explored
- Remove nodes from the queue in the order they were added (first in, first out)
 - Search through edges of removed node and add unexplored neighbors to queue
 - Label added nodes as explored
 - Terminate search when queue is empty
- Output: two lists described above

-
- For our simple example, the queue will initially be $Q = [1]$
 - Then, we remove node 1 and append its unexplored neighbors giving $Q=[2, 3]$ or $Q=[3, 2]$ depending on the order in which the neighbors of node 1 are stored
 - Let's take the first version, we then pop node 2, and append node 4: $Q=[3, 4]$
 - Then we pop node 3, and finally we pop node 4.



Python implementation

- Specify: Graph and source node

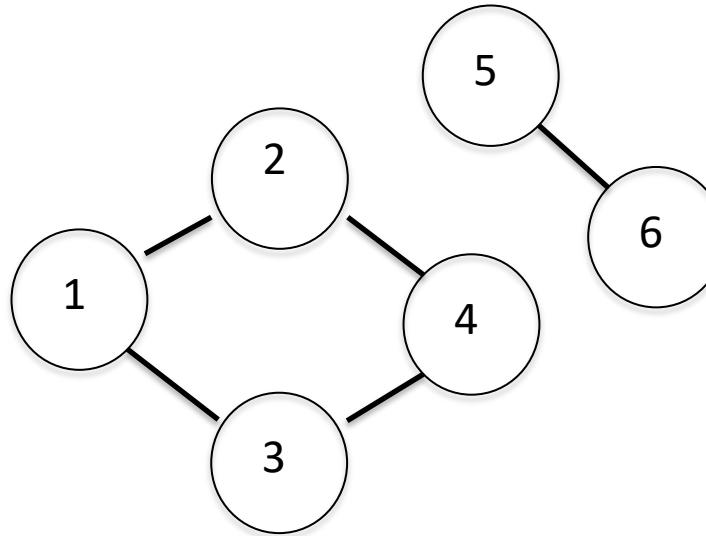
```
G = nx.Graph()
edges = [[1,2],[1,3],[2,4],[3,4],[5,6]]
G.add_edges_from(edges)
s = 1
Q = [s] #Nodes to be explored
```

- Create list of nodes and labels

```
L1 = list(G.nodes())
L2 = [-1 for i in nodes] #labels
L2[s-1]=1 #mark source node as explored
```

- Iterate through nodes in queue (updating Q as appropriate)

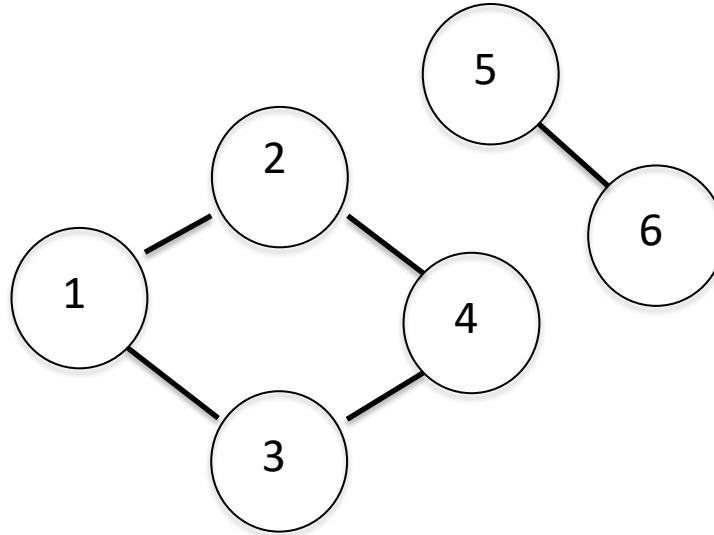
```
while len(Q)>0:
    n = Q.pop(0)
    for v in G.adj[n]: #iterate through neighbors of n
        if L2[v-1]==-1:
            L2[v-1]=1
            Q.append(v)
```



-
- Adding `print("n=%d, Q=%" %(n), Q)` to the while loop and running the code:

```
n=1, Q= [2, 3]
n=2, Q= [3, 4]
n=3, Q= [4]
n=4, Q= []
```

- n is the node removed from the queue and Q is the queue after unexplored neighbors of n have been added (if n is unexplored)
- What is the cost of BFS?
 - Each reachable node is relabeled and each (reachable) edge is encountered twice
 - For a graph with N nodes and L edges, cost is $\mathcal{O}(L + N)$
 - Linear time! (if our queue is managed efficiently)



-
- How can we use BFS to compute distances (from source)?
 - BFS iterates through the graph layer by layer
 - We know nodes 2 and 3 will be searched before 4
 - When a node is added to the queue, its distance is one greater than its neighbor being removed from queue
 - We just need to maintain a list of distances which are filled in as nodes are added to the queue

```
D = [-1000 for i in nodes] #initialize distances to -1000
D[s-1]=0 #Source node has distance zero

while len(Q)>0:
    n = Q.pop(0)
    for v in G.adj[n]: #iterate through neighbors of n
        if L2[v-1]==-1:
            L2[v-1]=1
            D[v-1] = D[n-1]+1 #Set distance of node v
    Q.append(v)
```

- Running this code: In [5]: D
Out[5]: [0, 1, 1, 2, -1000, -1000]

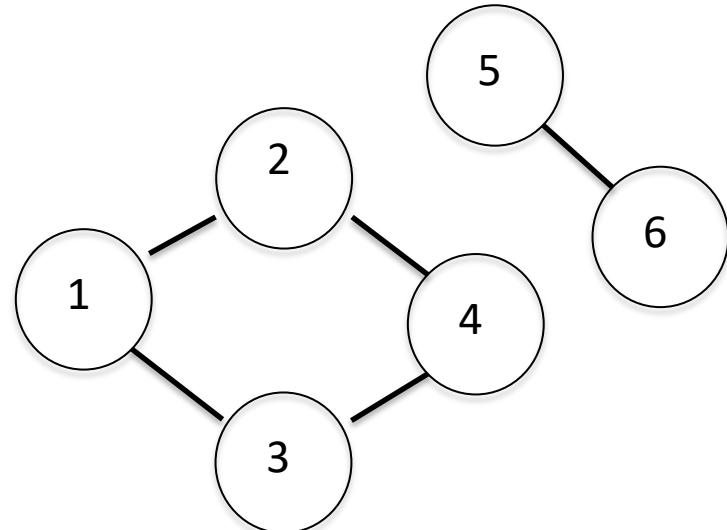
- I've argued that the BFS algorithm is $\mathcal{O}(N + L)$, but is our Python implementation consistent with this estimate?
- Let's take a 2nd look at it...
- Initialization of lists:

```
L1 = list(G.nodes())
L2 = [-1 for i in nodes] #labels
L2[s-1]=1 #mark source node as explored
```

This is $\mathcal{O}(N)$

- Iterating through neighbors:

```
for v in G.adj[n]: #iterate through neighbors of n
    if L2[v-1]==-1:
        L2[v-1]=1
        D[v-1] = D[n-1]+1 #Set distance of node v
        Q.append(v)
```



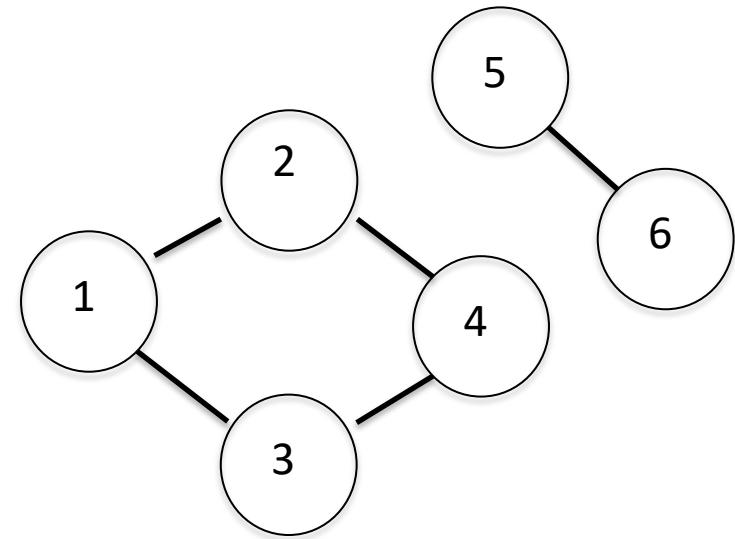
We can treat `G.adj[n]` as a dictionary lookup and then cost per neighbor is $\mathcal{O}(1)$

-
- What about iterating through the queue and removing nodes?

```
while len(Q)>0:  
    n = Q.pop(0)
```

Finding the length of a list is $\mathcal{O}(1)$, but then we have a problem!

- Popping from the front of Q is not $\mathcal{O}(1)$ but rather $\mathcal{O}(|Q|)$ where $|Q|$ is the length of the queue
- This means our algorithm, as implemented cannot be described as $\mathcal{O}(N + L)$
- Fortunately, the collections module contains a *dequeue* datatype
 - From online documentation:
list-like container with fast appends and pops on either end
- Using this datatype, we can put together a linear time implementation in Python



Lecture 7

Python containers
Depth-first search

Comments on Python containers

Quiz: what is the output from the code below?

```
In [11]: e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]
```

```
In [12]: G1 =nx.Graph()
```

```
In [13]: G1.add_edges_from(e)
```

```
In [15]: G2 = G1
```

```
In [16]: G2.add_edge(6,7)
```

```
In [17]: list(G1.edges())
```

Quiz: what is the output from the code below?

```
In [11]: e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]
```

```
In [12]: G1 =nx.Graph()
```

```
In [13]: G1.add_edges_from(e)
```

```
In [15]: G2 = G1
```

```
In [16]: G2.add_edge(6,7)
```

```
In [17]: list(G1.edges())
```

```
Out[17]: [(1, 5), (5, 2), (5, 4), (2, 3), (3, 4), (4, 6), (6, 7)]
```

- Adding an edge to G2 modifies G1. Why? G1 is a reference for data stored in a particular location in memory, and G2=G1 sets G2 to be a reference for the same data
- Then, adding an edge to G2 modifies the data that both G1 and G2 are references for

-
- If we want an “independent copy” of G1 and G2 we should use the .copy method:

```
In [18]: G3 = G1.copy()
```

```
In [19]: G3.add_edge(7,8)
```

```
In [20]: list(G1.edges())
```

```
Out[20]: [(1, 5), (5, 2), (5, 4), (2, 3), (3, 4), (4, 6), (6, 7)]
```

- And the same applies for lists and other mutable containers in Python:

```
In [22]: L1 = [1,2,3]
```

```
In [23]: L2 = L1
```

```
In [24]: L2.append(4)
```

```
In [25]: L1
```

```
Out[25]: [1, 2, 3, 4]
```

- Using L2 = L1.copy() instead of L2=L1 would create an independent copy of L1

- Important note: when working with a container of containers, then copy will not create an independent copy of the elements of the container. copy.deepcopy() should be then used instead of .copy()

-
- We also have to be careful when providing mutable containers as input to functions, as the function may modify the input variable outside of the function where it was called:

```
In [30]: def array_example(x):
...:     x[0]=1
...:     return None    ...:
```

```
In [31]: y = np.zeros(10)
```

```
In [32]: y
Out[32]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

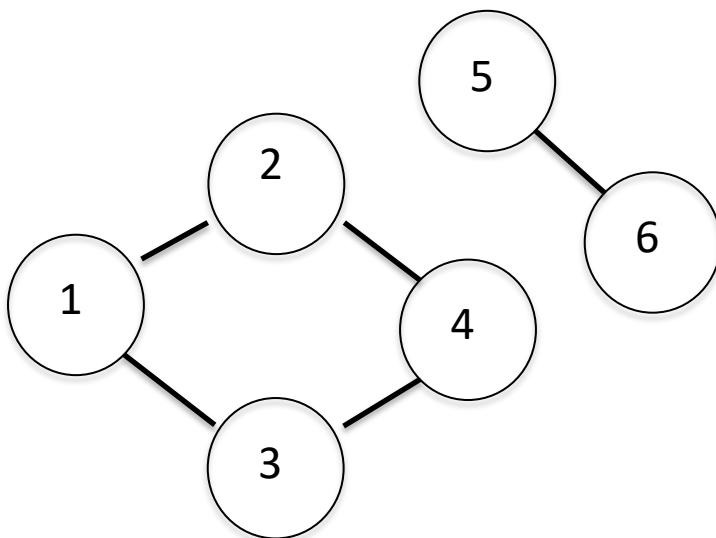
```
In [33]: array_example(y)
```

```
In [34]: y
Out[34]: array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

- For this reason, functions often require tuples as input since tuples are immutable. Strings are also immutable and cannot be modified (without creating a new copy)

Graph search recap

- Basic approach:
 - Label s as “explored” and all other nodes as “unexplored”
 - *While there is at least one edge between an explored and unexplored node:*
Select one such edge and re-label the unexplored node as explored



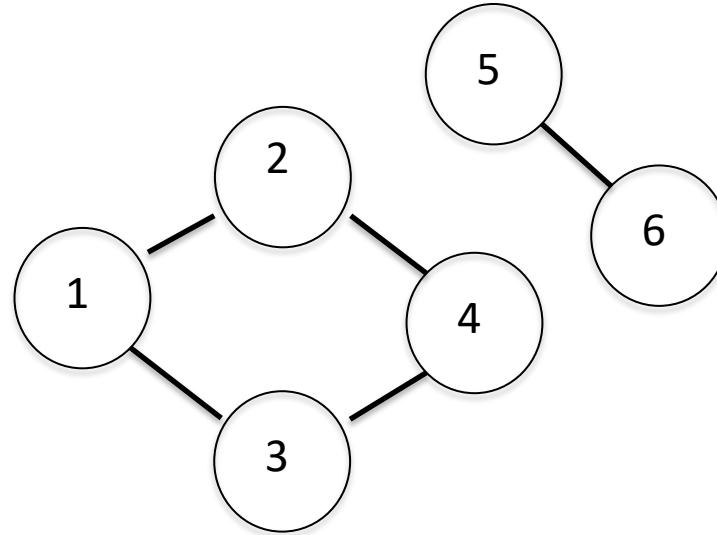
Implementation: Depends on how edges are selected each iteration

- Depth-first search – aggressively move into the graph (1-2, 2-4)
- Breadth-first search – consider one “layer” at a time (1-2, 1-3)

Depth-first search

- What changes for DFS?
- How much does the code change?
- The problem setup and initialization will be the same
- We can then focus on the search through the graph
- BFS: Remove node from front of queue, append unexplored neighbors to back of queue
- DFS: Remove node from end of *stack*, append unexplored neighbors to end of stack

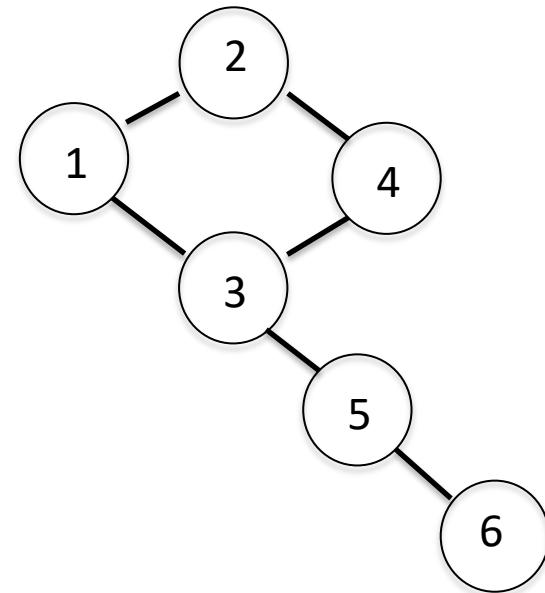
```
#Depth-first search
while len(S)>0:
    n = S.pop() #Only change from BFS
    for v in G.adj[n-1]: #iterate through neighbors of n
        if L[1][v-1]==0:
            L[1][v-1]=1
            S.append(v)
```



-
- Let's compare DFS and BFS on this modified graph
 - Output from codes is provided below, and we can compare the order in which nodes are visited

```
In [16]: run bfs1
n=1,Q= [2, 3]
n=2,Q= [3, 4]
n=3,Q= [4, 5]
n=4,Q= [5]
n=5,Q= [6]
n=6,Q= []
```

```
In [17]: run dfs1
n=1,S= [2, 3]
n=3,S= [2, 4, 5]
n=5,S= [2, 4, 6]
n=6,S= [2, 4]
n=4,S= [2]
n=2,S= []
```



- We can see that DFS finds node 6 faster than BFS though both methods require the same number of iterations
- DFS is used to analyze important properties of directed graphs; for example, DFS can be used to *sort* the nodes in a directed acyclic graph which is useful for planning applications (e.g. solving a Sudoku puzzle)

-
- The cost of DFS, like BFS, is $\mathcal{O}(N + L)$ if the algorithm is implemented efficiently
 - In a network with N nodes, the maximum number of links is $\binom{N}{2} = \frac{N(N-1)}{2}$ or $\approx \frac{N^2}{2}$ for large graphs
 - However, for most large networks, $L \ll \frac{N^2}{2}$, and such networks are described as *sparse*

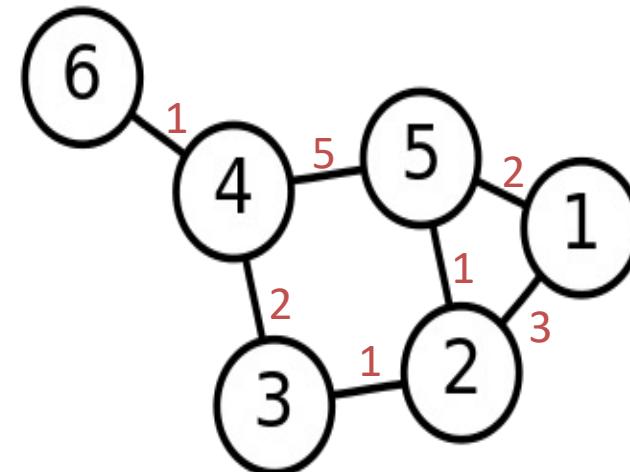
Lecture 8

Dijkstra's algorithm

Improving Dijkstra's algorithm and binary heaps

Weighted networks

- We have only considered unweighted networks so far
- However, it is often useful to associate a numerical weight with each edge
 - E.g. weights can indicate physical distance or travel time
 - And they can represent frequency of interactions between users of online social networks
- What is the best way to compute distances and find shortest paths in a weighted network?
- Can we use BFS? If weights are small positive integers, we can replace a link with weight = n with n links with weight = 1 and then use BFS
 - But weights can be arbitrarily large and may not be integers
- We will now look at Dijkstra's algorithm which is a graph-search method for networks with non-negative weights



distances:

$1 \rightarrow 2: 3$

$1 \rightarrow 4: 6$

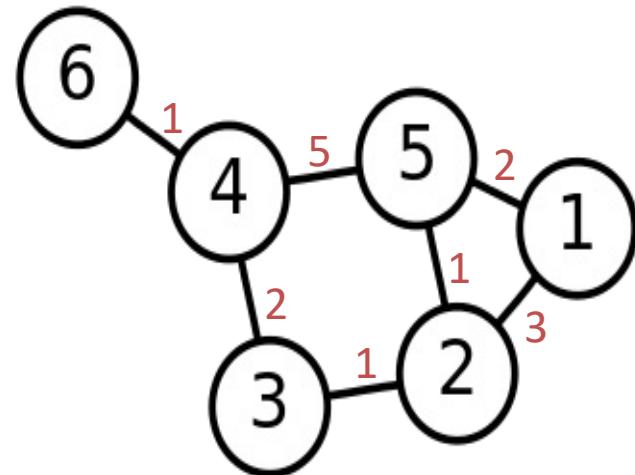
Dijkstra's algorithm

Problem statement:

- Input: Graph (nodes, edges, weights) and source node, s
- Output: Distances from source node to all nodes reachable from source

Notes:

- The length of a path is the sum of the weights of all links crossed on the path.
- Let w_{ij} be the weight of the edge between nodes i and j , and let d_{ij} be distance between nodes i and j
- So, in the undirected graph pictured, the path 5,4,6 has length 6 while $d_{56} = d_{65} = 5$



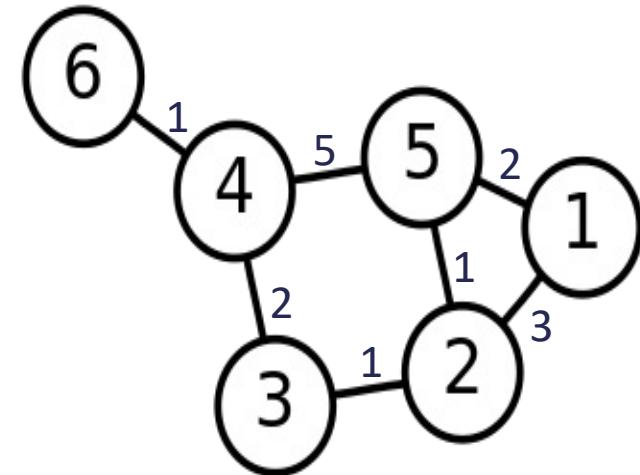
distances:

$1 \rightarrow 2: 3$

$1 \rightarrow 4: 6$

Basic idea:

- Maintain a *priority queue*, M , of explored nodes where each node in M is assigned a numerical “priority”, and say that the maximum priority is p_{max} .
- Each iteration remove a node from M with priority = p_{max} and determine its distance to the source, and move it to F , the set of “finalized” nodes. Then iterate through its neighbors and:
 - for neighbors in M adjust their priority as needed
 - move unexplored neighbors to M , and assign priorities to these nodes
- As we will see, the priority of a node in M can be interpreted as the inverse of a “provisional distance”(PD) to the source, so a higher priority corresponds to a shorter PD.



distances:

$1 \rightarrow 2: 3$

$1 \rightarrow 4: 6$

The main questions we need to consider now are:

- 1) How do we assign a priority to a node when it is first added to M ?
 - 2) How do we update the priority of a node in M when one of its neighbors is moved from M to F ?
 - 3) Why do we move a node with priority = p_{max} from M to F , and what is its distance to the source?
- I will first state the answers to these questions. This will complete the specification of the algorithm and will allow us to see how it works on a small graph
 - We will then examine the correctness of the algorithm using an inductive approach. This will show why the answers to these questions are what has been stated

We will sacrifice a little generality and answer the questions in the context of an example. Let δ_j be the PD of node j (an arbitrary node in M), and the corresponding priority is simply, $1/\delta_j$. Say that node i has just been removed from M and that it has an unexplored neighbor a and an explored neighbor b (with b already in M)

- 1) How do we assign a priority to node a when it is first added to M ?

Answer: $\delta_a = d_{si} + w_{ia}$; this is the length of the shortest path from s to a that includes the link $i - a$

- 2) How do we update the priority of node b to account for i being moved from M to F ? *Answer:* $\delta_b^{new} = \min(\delta_b^{old}, d_{si} + w_{ib})$; the rationale for this rule will be explained a little later

- 3) Why do we move a node with priority $= p_{max}$ from M to F , and what is its distance to the source?

Answer: For any such node, $d_{sj} = \delta_j$ so it can be added to the set of “finalized” nodes.

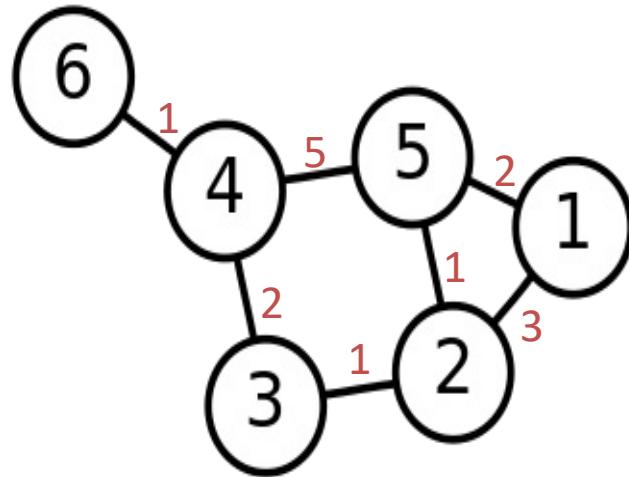
Summary of algorithm

Initialization: Set the PD for the source to be zero, add it to M , and label it as explored. Label all other nodes as unexplored

- 1) Each iteration, find a node in M with the minimum PD (maximum priority) and move it to F . Say this node is node i
- 2) Examine the neighbors of i
- 3) For neighbors already in M , update their PDs with: $\delta_b^{new} = \min(\delta_b^{old}, d_{si} + w_{ib});$
- 4) For unexplored neighbors, set $\delta_a = d_{si} + w_{ia}$
- 5) For neighbors in F , do nothing
- 6) Continue “exploring” until no reachable nodes remain unexplored.

Consider the graph shown, and set $s=5$

- Initially we have $F = \emptyset, M = \{(5,0)\}$ and $(5,0)$ is shorthand here for “node 5 with PD=0”
- Node 5 has the smallest provisional distance (and highest priority), so after the 1st iteration: $F = \{(5,0)\}, M = \{(1,2), (2,1), (4,5)\}$
- 2nd iteration: $F = \{(5,0), (2,1)\}, M = \{(1,2), (4,5), (3,2)\}$
- 3rd iteration: $F = \{(5,0), (2,1), (1,2)\}, M = \{(4,5), (3,2)\}$
- 4th iteration: $F = \{(5,0), (2,1), (1,2), (3,2)\}, M = \{(4,4)\}$
- 5th iteration: $F = \{(5,0), (2,1), (1,2), (3,2), (4,4)\}, M = \{(6,5)\}$
- Final iteration: $F = \{(5,0), (2,1), (1,2), (3,2), (4,4), (6,5)\}$



We see that the algorithm works for a simple (but non-trivial) example. But why does it work? And will it always work? Let's adopt an inductive approach to (partially) address these questions.

- Consider the 1st iteration: The source node is moved from M to F , its distance is finalized as zero, and its neighbors are added to M with initial priorities set as described earlier
 - Clearly the distance of the source node has been set correctly
- Now, consider the nth iteration and assume that the distances for the n-1 nodes in F have been set correctly. Let x be a node in M with priority = p_{max} . We will now argue that $\delta_x = d_{sx}$.

Our discussion on the correctness of the n^{th} iteration will follow 3 steps: 1) provide an interpretation of δ_x , 2) argue that $d_{sx} \leq \delta_x$, and 3) argue that $d_{sx} \geq \delta_x$

Step 1: The provisional distance, δ_x , is the length of a shortest path between the source and x where all nodes on the path are in F except x . Why? Say that nodes i_1, i_2, i_3, \dots are the neighbors of x in F . The shortest path from s to x with all nodes in F except x is:

$$\min(d_{si_1} + w_{i_1x}, d_{si_2} + w_{i_2x}, \dots),$$

and the algorithm ensures that this minimum value is equal to δ_x

Step 2: From step 1, we know that δ_x corresponds to the length of a path between s and x , so $d_{sx} \leq \delta_x$

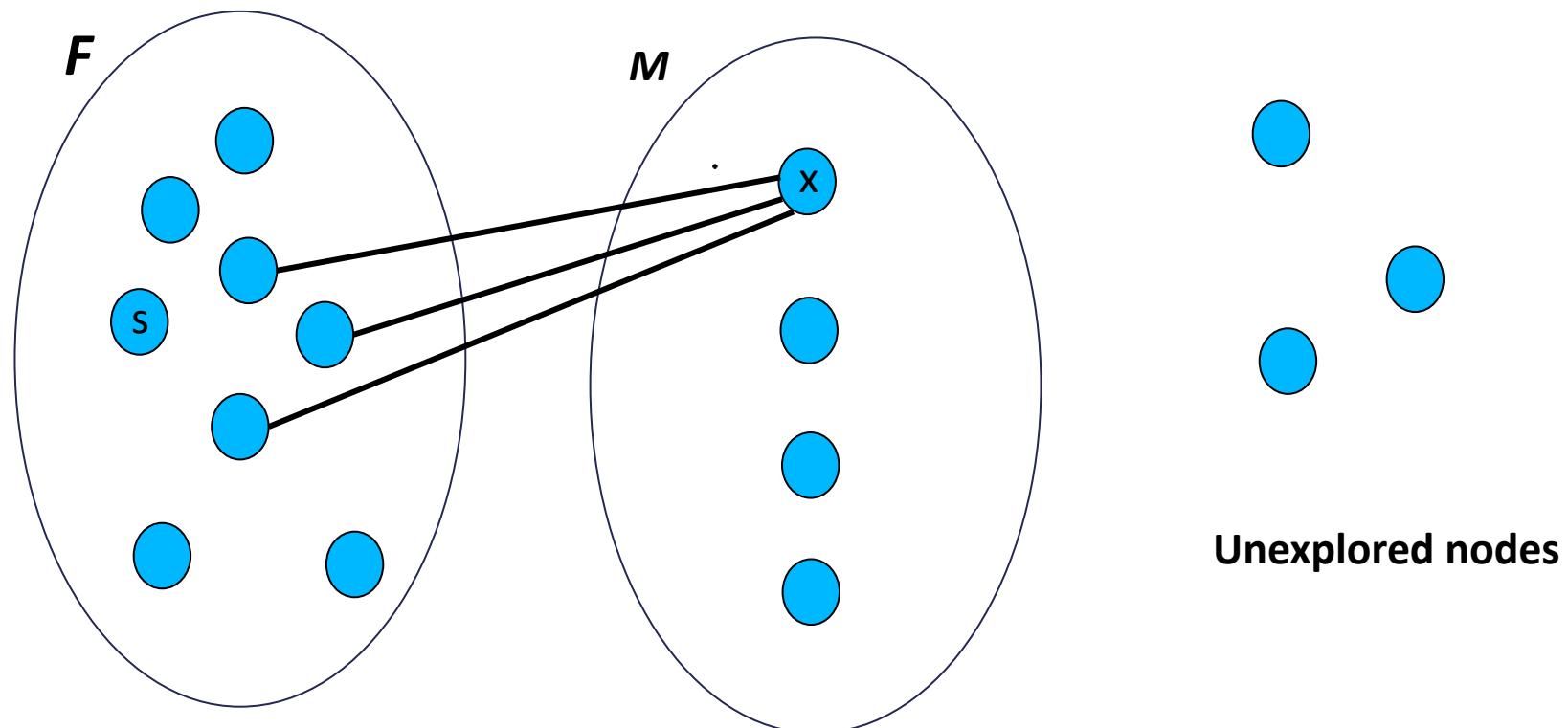
Step 3: Since all neighbors of all nodes in F are either in F or M , all paths from s to x must have at least one edge, $j - y$, connecting a node in F to a node in M . We will now argue no such path can have length less than δ_x

- The length of the shortest path from s to x crossing edge, $j - y$ is $d_{sj} + w_{jy} + d_{yx}$. If $y \in M$, then its PD must satisfy $\delta_y \leq d_{sj} + w_{jy}$ (the PD cannot increase after it is initially set, and if $\delta_y > d_{sj} + w_{jy}$ before j was added to F , it would have then been set to $d_{sj} + w_{jy}$)
 - Since all edge weights are non-negative, $d_{yx} \geq 0$
 - Since x has priority = p_{max} , we know $\delta_x \leq \delta_y$ and:
$$\delta_x \leq \delta_y \leq d_{sj} + w_{jy} \leq d_{sj} + w_{jy} + d_{yx}.$$
 - So δ_x cannot be larger than the length of the shortest path connecting s and x . Combining this observation with the result from step 2, we conclude that: $d_{sx} = \delta_x$

This is a sketch of how the problem “looks” after a few iterations to complement the previous discussion.

General properties:

- There are no links between nodes in F and unexplored nodes.
- Each node in M will have at least one neighbor in F
- A path from s to x must cross at least one link from F to M



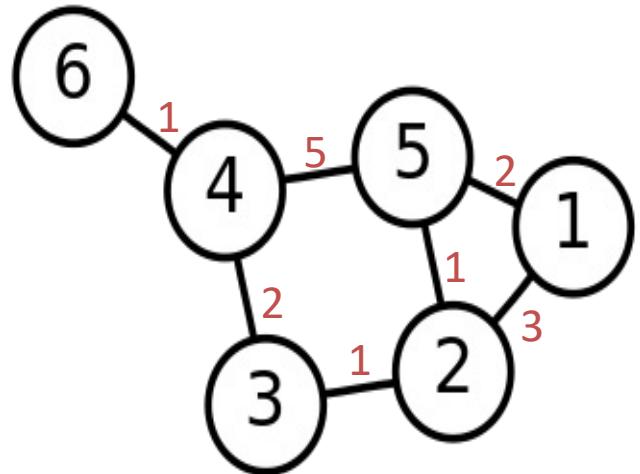
Summary of algorithm

Initialization: Set the PD distance for the source to be zero, add it to M , and label it as explored. Label all other nodes as unexplored

- 1) Each iteration, find the node in M with the minimum PD (highest priority) and move it to F . Say this node is node i
- 2) Examine the neighbors of i
- 3) For neighbors already in M , update their PDs with: $\delta_b^{new} = \min(\delta_b^{old}, d_{si} + w_{ib})$;
- 4) For unexplored neighbors, set $\delta_a = d_{si} + w_{ia}$
- 5) Continue “exploring” until no reachable nodes remain unexplored.

Python implementation

- What data type(s) should we use?
 - list, array, dictionary, something else?
- Basic operations needed:
 - find min
 - insert/delete
- Let's try dictionaries, they are not perfect for this set of operations, but are "ok" and easy to use
- We'll work with two dictionaries – one for finalized explored nodes (where the shortest path length has been determined) and one for neighbors of finalized explored nodes (where provisional distances have been specified)
- And we'll also have a NetworkX graph



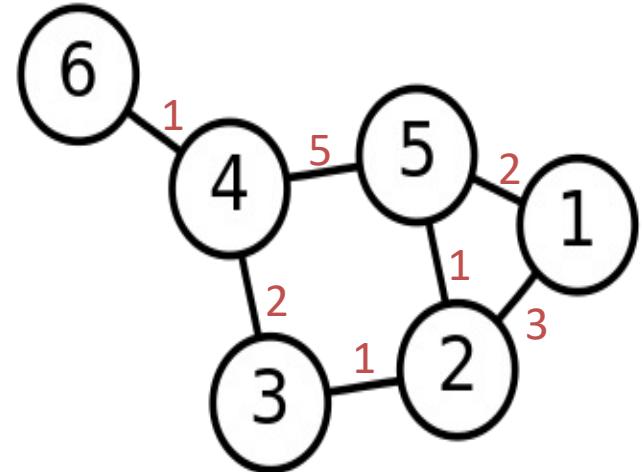
Python implementation

1. Create the graph, initialize the dicts

```
e=[[1,2,3],[1,5,2],[2,5,1],[2,3,1],[3,4,2],[4,5,5],[4,6,1]]  
G = nx.Graph()  
G.add_weighted_edges_from(e)  
  
Mdict = {} #Explored neighbor nodes  
Mdict[5]=0  
Fdict={} #Explored finalized nodes
```

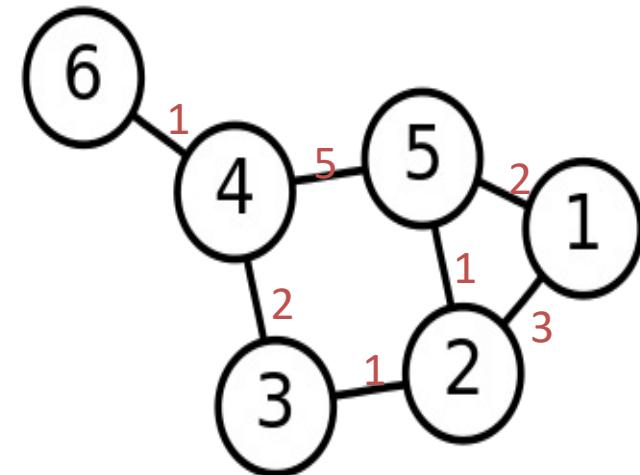
2. Find node in Mdict with smallest provisional distance

```
while len(Mdict)>0:  
    dmin=np.inf  
    for k,v in Mdict.items():  
        if v<dmin:  
            dmin=v #min distance  
            n=k #corresponding node
```



3. Remove ‘smallest-distance’ node and update provisional distances of its unexplored neighbors

```
for m,en,wn in G.edges(n,data='weight'):
    ddist = dmin+wn
    if en in Fdict:
        pass
    elif en in Mdict:
        Mdict[en] = min(Mdict[en],ddist)
    else:
        Mdict[en] = ddist
Fdict[n] = Mdict.pop(n)
```



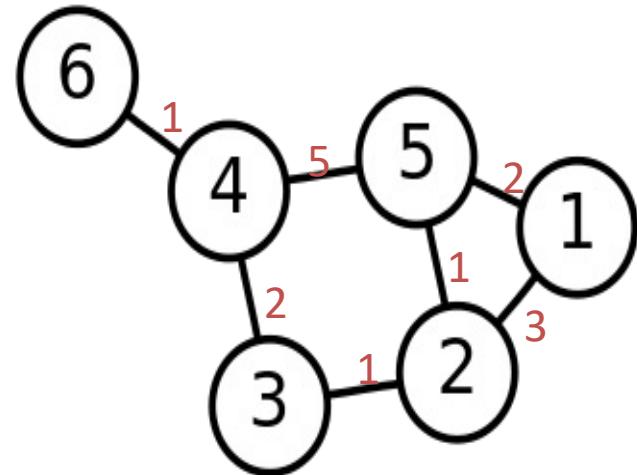
Setting the source to be node 5 and running the code:

```
In [56]: Fdict
Out[56]: {1: 2, 2: 1, 3: 2, 4: 4, 5: 0, 6: 5}
```

The key-value pairs are node:distance

What is the cost for a graph with N nodes and L edges?

- For convenience, say that every node in the graph is reachable from the source
- Each node will at some point be added and removed from $Mdict \rightarrow \mathcal{O}(N)$
- Each edge is examined twice $\rightarrow \mathcal{O}(L)$
- What about the d_{min} calculation?
 - $Mdict$ contains at most N nodes each iteration
 - Then, each iteration, the d_{min} calculation will require $\mathcal{O}(N)$ operations
 - We will have N iterations (each node will at some point be placed in F) so the overall cost is $\mathcal{O}(N^2)$
- The minimum calculation is a bottleneck, and we no longer have linear time. We will focus on this issue next.



Summary of Dijkstra's algorithm:

Maintain: 1) set of “*Finalized*” nodes, F , where distances have been assigned
and 2) set of “*Explored but not finalized*” nodes, M , where provisional distances have been set

Initialization: Place source node in M with distance=0 (alternate approach: all other nodes also in M with provisional distance=Infinity)

At beginning of any iteration:

1. All nodes in M : provisional distance = shortest path length via finalized nodes only
2. Node in M with minimum provisional distance: provisional distance = length of shortest path to source node

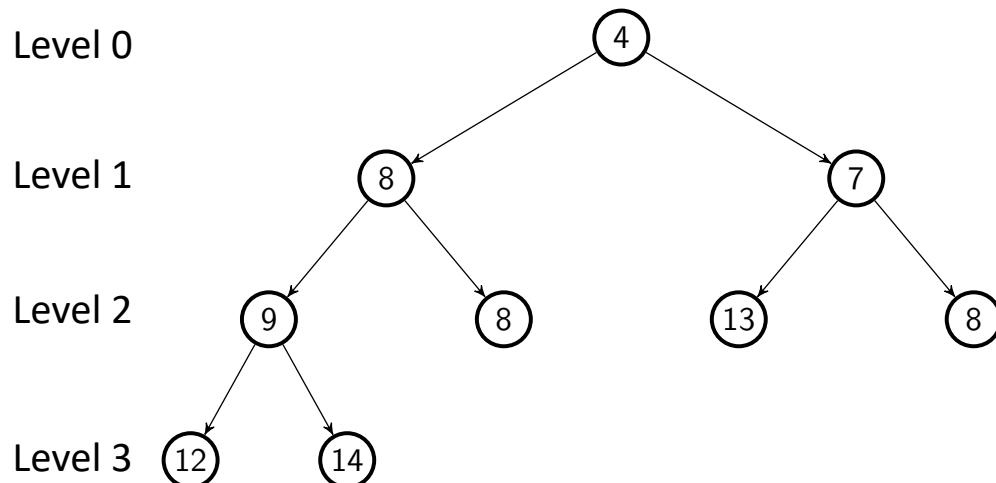
Each iteration: Move a node with minimum provisional distance to F and examine its neighbors

Bottleneck: finding a node with minimum provisional distance; cost is $\mathcal{O}(N^2)$

Binary heap

- It would be useful to arrange data so that it is easy to:
 1. Extract a node (n^*) with the minimum provisional distance (PD)
 2. Update provisional distances
 3. Insert unexplored neighbors of n^*
- A *binary heap* provides these operations with cost, $\mathcal{O}(\log_2 N)$
- We won't go through the full Dijkstra + heap implementation, but will outline the key elements
- A binary heap arranges nodes in a list, H ; the order of the nodes is determined by their PDs with a “smallest-distance” node in $H[0]$
- To understand how binary heaps work, it is better to visualize them as binary trees where each element has 0, 1, or 2 “children”

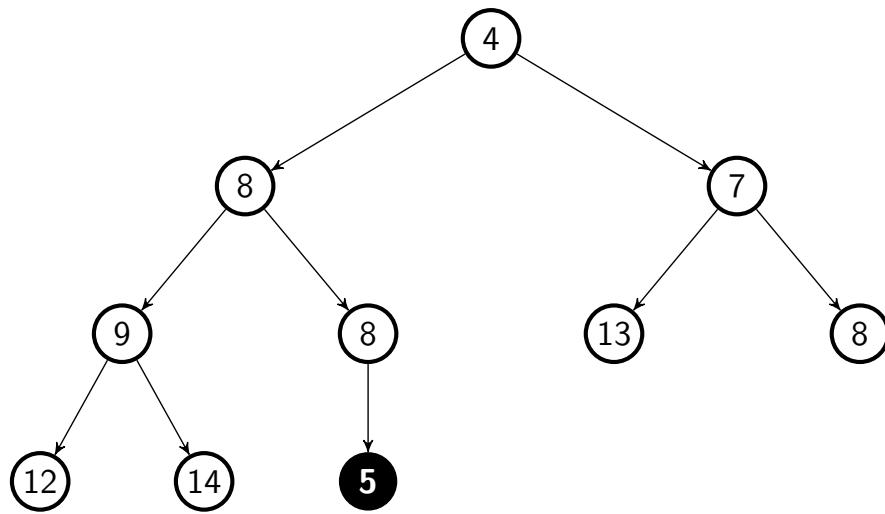
The arrangement of elements in H corresponds to a binary tree with the “heap property”: every parent element has a key less than or equal to the keys of its children.



- In the example, there are 4 levels; and each level (starting at the top and moving down) must be as full as possible
- We may have values associated with the keys (e.g. the node number corresponding to a provisional distance)
- Let’s look at an example that shows how to insert a new element in $\mathcal{O}(\log_2 N)$ time while maintaining the binary tree structure and heap property

$$H = \boxed{4 \quad 8 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14}$$

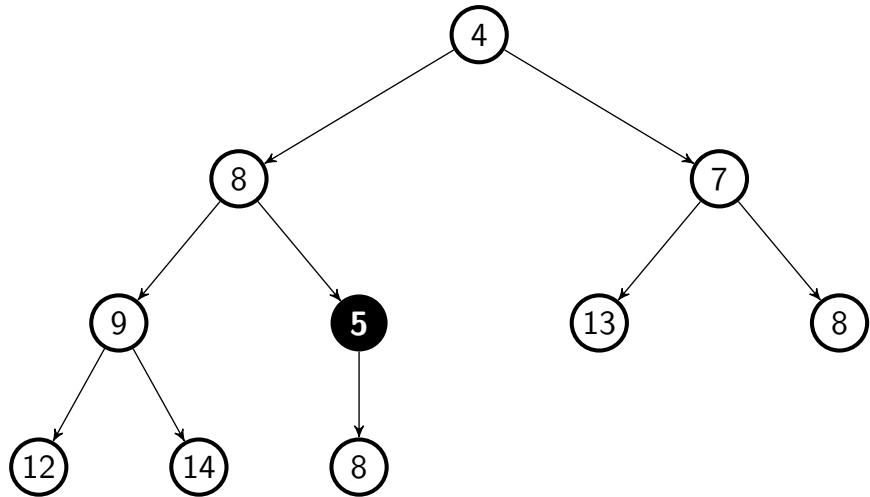
Example: insert an element with key=5 in the heap shown



1. Place new element on bottom level of tree so that it is the “furthest to the right” (or create a new level if the bottom level is full)

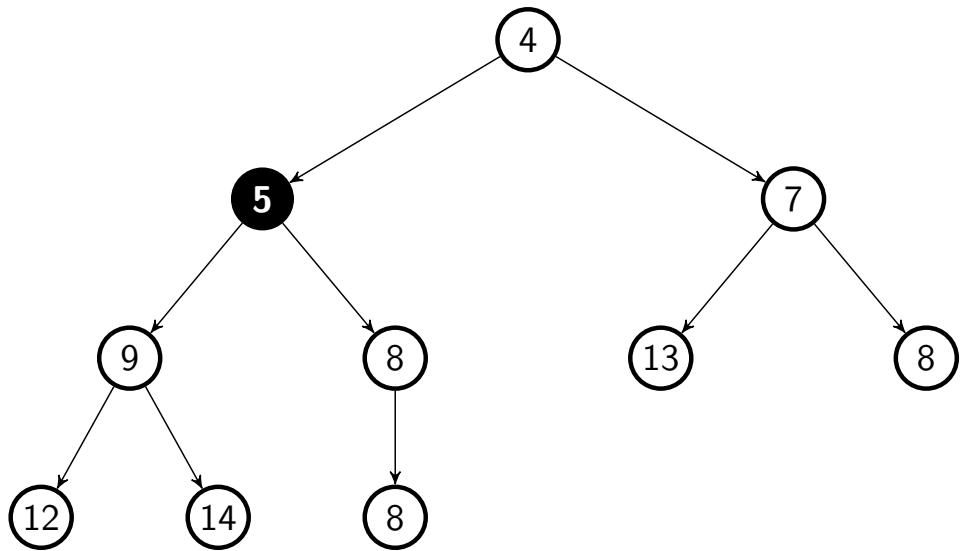
$$H = \boxed{4 \quad 5 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14 \quad 8}$$

Example: insert an element with key=5 in the heap shown



1. Place new element on bottom level of tree so that it is the “furthest to the right” (or create a new level if the bottom level is full)
2. Then we compare the new element with its parent and swap them to preserve the heap property

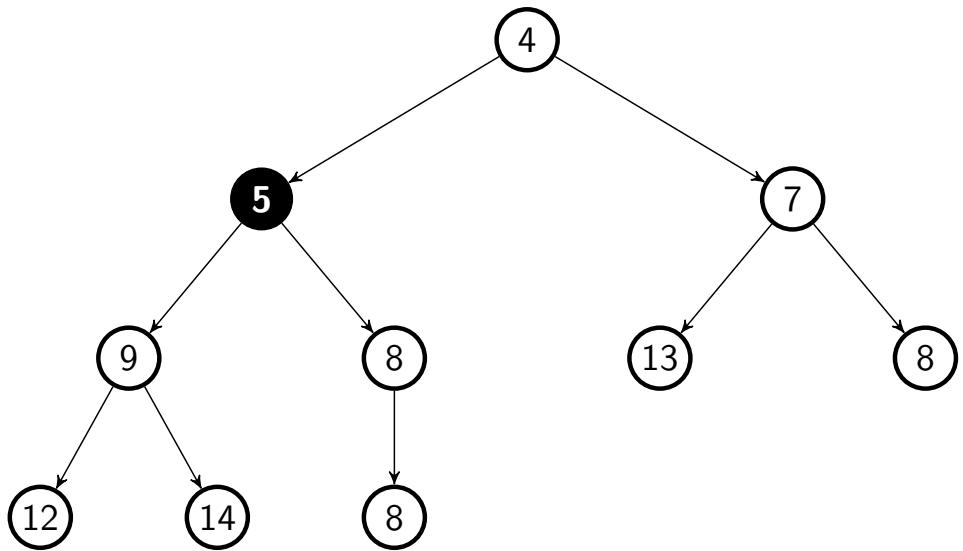
$$H = \boxed{4 \quad 5 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14 \quad 8}$$



Example: insert an element with key=5 in the heap shown

1. Place new element on bottom level of tree so that it is the “furthest to the right” (or create a new level if the bottom level is full)
2. Then we compare the new element with its parent and swap them to preserve the heap property
3. We continue comparing the new element with parents and swapping until a parent with a smaller key is encountered

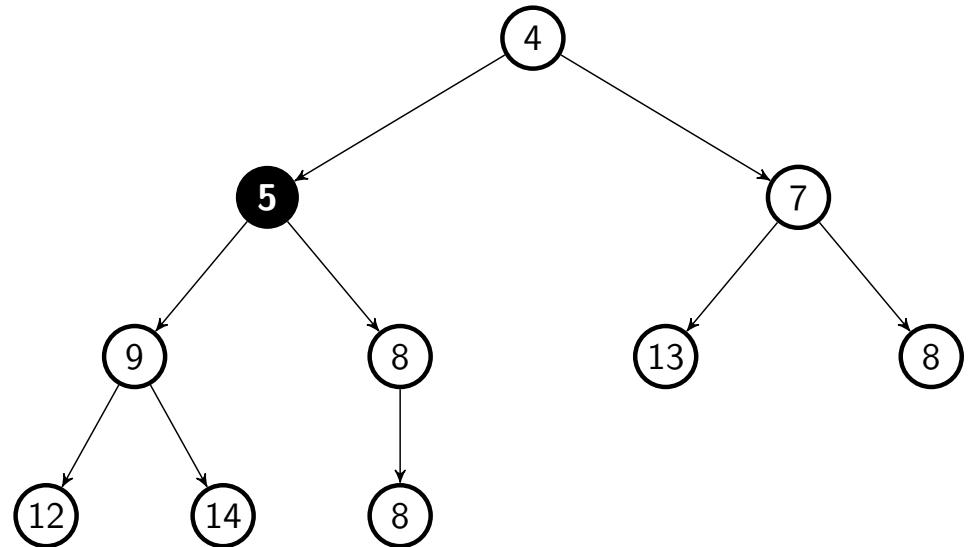
$H = \boxed{4 \quad 5 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14 \quad 8}$



What is the cost of insertion?

- In the worst case the new element has to be moved all the way to the top of the heap
- How many comparisons + swaps would this require?
- Let N be the number of elements prior to insertion. Then the maximum number of swaps+comparisons is $\lfloor \log_2(N + 1) \rfloor$ where $\lfloor \quad \rfloor$ represents the floor function
- So if swaps and comparisons are $O(1)$, then the overall running time will be $O(\log_2 N)$ as claimed.

$$H = \boxed{4 \quad 5 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14 \quad 8}$$



For min-removal:

- we swap the root and “last” element (lowest level, furthest to the right)
- Pop the last element
- Compare root with children, swap with whichever is smaller
- Continue comparing with children and swapping with smaller child until both children are larger
- Cost is again $\mathcal{O}(\log_2 N)$

$$H = \boxed{4 \quad 5 \quad 7 \quad 9 \quad 8 \quad 13 \quad 8 \quad 12 \quad 14 \quad 8}$$

- Python `heapq` builds ordered lists in linear time and provides $\log_2 N$ “min-removal” and insertion
- It does not provide $\log_2 N$ key modification (update provisional distance) – this has to be manually coded, and the use of a binary heap with Dijkstra’s method in Python is manageable but not straightforward
- Assuming we have $\log_2 N$ min-removal, insertion, and key-modification, what is the running time for Dijkstra with heap?

- Start with a heap of neighbors of the source node
 1. Remove n^* and re-structure heap
 2. Adjust weights of neighbors of n^* (if needed) and re-structure heap
 3. Insert unexplored neighbors of n^*

Overall cost of step 1: $\mathcal{O}(N \log_2 N)$

Overall cost of step 2: $\mathcal{O}(L \log_2 N)$

Overall cost of step 3: $\mathcal{O}(N \log_2 N)$

Is this better?

Graph search recap

BFS: search in unweighted graphs

- Maintain *queue* of explored nodes
- $\mathcal{O}(N + L)$ operations *if* items are added to and removed from queue in $\mathcal{O}(1)$ time
- Should use deque rather than list in Python
- Applications include: finding connected components, shortest paths

DFS: search in unweighted graphs

- Maintain *stack* of explored nodes
- $\mathcal{O}(N + L)$ operations *if* items are added to and removed from stack in $\mathcal{O}(1)$ time
- Should use list in Python
- Applications include: finding connected components, topological sort in DAGs

Dijkstra: search in weighted graphs (non-negative weights)

- Maintain *priority queue* of unexplored nodes with provisional distances
- $\mathcal{O}(N^2)$ operations *if* naïve search for highest-priority node is used
- Binary heaps can be used for better performance
- Applications include: finding connected components, shortest paths

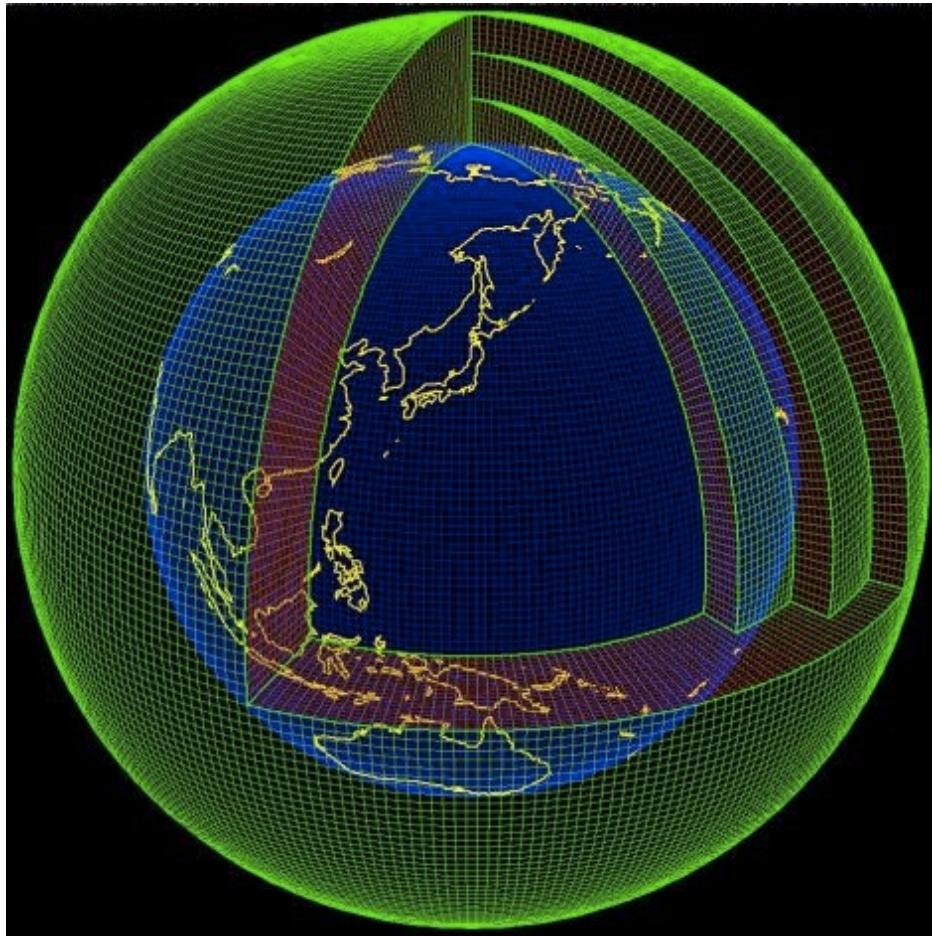
Final notes:

- Other algorithms are available for weighted graphs with negative weights
- NetworkX provides shortest-path functions
 - It is important to understand strengths/weaknesses and cost of underlying algorithms (especially when working on large networks)
 - It is also important to think about how data is organized (stacks, queues, heaps)
 - Also, not all networks are best represented as NetworkX graphs!

Lecture 9

Computational science
Simulating random walks

Computational science



<https://www.jma.go.jp/jma/jma-eng/jma-center/nwp>

- We will now shift our focus from computer science to *computational* science
- We will take a multidisciplinary approach requiring scientific, mathematical, and computational tools and ideas
- Atmospheric and oceanic dynamics were mentioned in lecture 1 and are motivating applications
 - But these kinds of problems require huge amounts of computational power, we will start with simpler problems

-
- A few general comments on how we will be approaching these problems:
 - In the first few weeks of the module, we looked at individual algorithms in detail
 - We are not going to do the same with numerical methods now
 - We already have several modules on numerical methods! So I am taking a different approach and we will use existing functions and modules without analyzing them in detail
 - I want you to build a picture of how scientific reasoning, mathematical models, and computational tools should be understood and used collectively
 - We will later shift our focus to data-centric questions and problems but the general approach will not be too different from what we will now start using

-
- I will ask you to figure out ways to analyze simulation results
 - This can take many forms:
 - Simple visualization
 - Simple statistics: mean, standard deviation, etc...
 - For dynamical processes, how does a system evolve in time?
 - Is there growth or decay? If so, what is the rate of change? Do the results follow a exponential trend? Or a power law? Logarithmic? Something else?
 - Can we draw connections between computational results and the mathematical properties of a relevant model?
 - We will see some simple examples to make these ideas more concrete

-
- Computational science with Python (typically) relies heavily on numpy, scipy, and matplotlib
 - There are a few key points for efficiency
 - Avoid loops and use built-in functions wherever possible
 - Growing or shrinking numpy arrays is expensive – initialize with appropriate size at beginning of problem
 - Typically, efficiency will consider *flops* – the number of floating point operations (additions and multiplications)
 - This can be difficult to estimate, i.e. how many flops does $\sin(3)$ need?
 - We will also continue to think about how to store “information” for efficient computation

Random walks

- We will now think about how to simulate stochastic processes – processes that are influenced by randomness
- One of the simplest examples of a non-trivial stochastic process is a discrete random walk in one dimension:
 - During time Δt , a particle moves a distance $+\Delta x$ or $-\Delta x$ with equal probability = $\frac{1}{2}$
 - Or equivalently, each time step, a walker “flips a coin”, and then takes a step to the right if it is heads and a step to the left if its tails. A random walk is then the result of a sequence of coin flips.
 - This model is simple enough to analyze on paper, but we’ll begin by looking at a Python simulation of random walks
 - We’ll then compare statistics extracted from computations with analytical results

One step of a random walk will be represented as: $X_{i+1} = X_i + R_i$ where R_i is a random number that is $\pm \Delta x$ with equal probability.

- How do we simulate a coin flip?
 - We can generate a random number, x , between 0 and 1 (sampled from a uniform distribution) using `np.random.rand()`, and say $x > 0.5$ corresponds to heads, and $x \leq 0.5$ corresponds to tails.
 - But it is easier to use `np.random.choice`:

```
np.random.choice([-1,1],10)
Out[27]: array([-1, -1, -1, -1,  1,  1,  1, -1, -1])
```

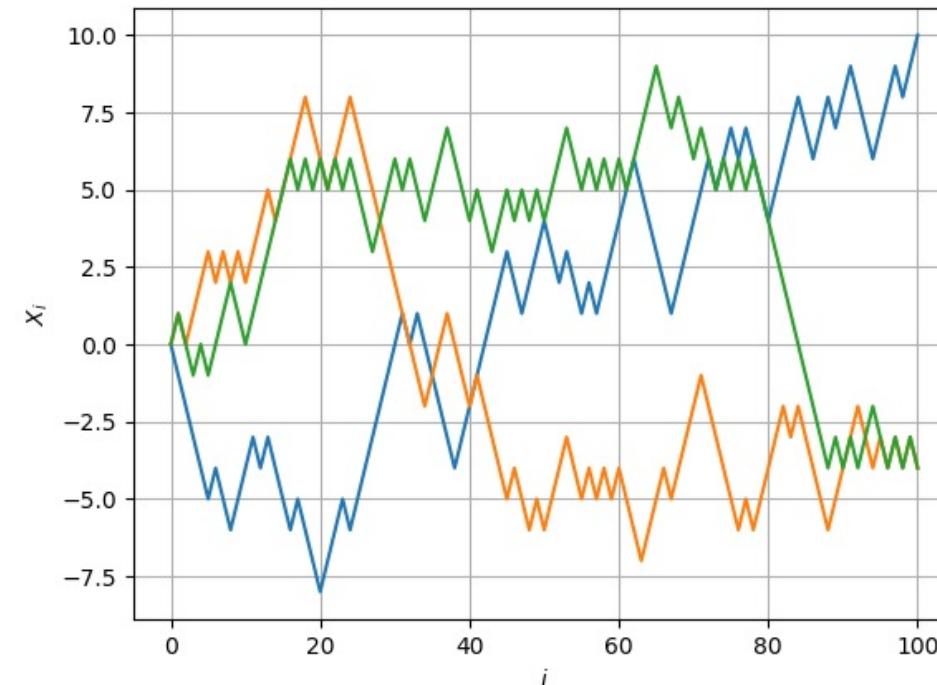
This is a simulation of 10 coin flips.
- Then, it is just a matter of assembling the Python code. We will set $\Delta x = 1$ for convenience.

Random walks

- Here is a simple implementation of an N_t -step random walk starting from $X = 0$:

```
X = np.zeros(Nt+1)
for i in range(Nt):
    X[i+1] = X[i]+np.random.choice([-1,1])
```

- And this code generates trajectories like these:



-
- However, individual trajectories don't tell us much. Instead, we should compute several trajectories and compute statistics like the average position and the standard deviation:

```
X = np.zeros((M,Nt+1))
for j in range(M):
    for i in range(Nt):
        X[j,i+1] = X[j,i]+np.random.choice([-1,1])
```

- And then we compute the average and standard deviation over the M simulations (\bar{X}_i and $s_i = \sqrt{\bar{X}_i^2 - \bar{X}_i^2}$):

```
Xave = X.mean(axis=0)
Xstd = X.std(axis=0)
```

- How do we check if Xave and Xstd are correct? The law of large numbers tells us that Xave should converge to $\langle X \rangle$, the expected value of X , as M is increased. Similarly \bar{X}_i^2 should converge to $\langle X^2 \rangle$. It is straightforward to derive expressions for these expectations.

-
- A random walk is defined using, $X_{i+1} = X_i + R_i$. Note that:

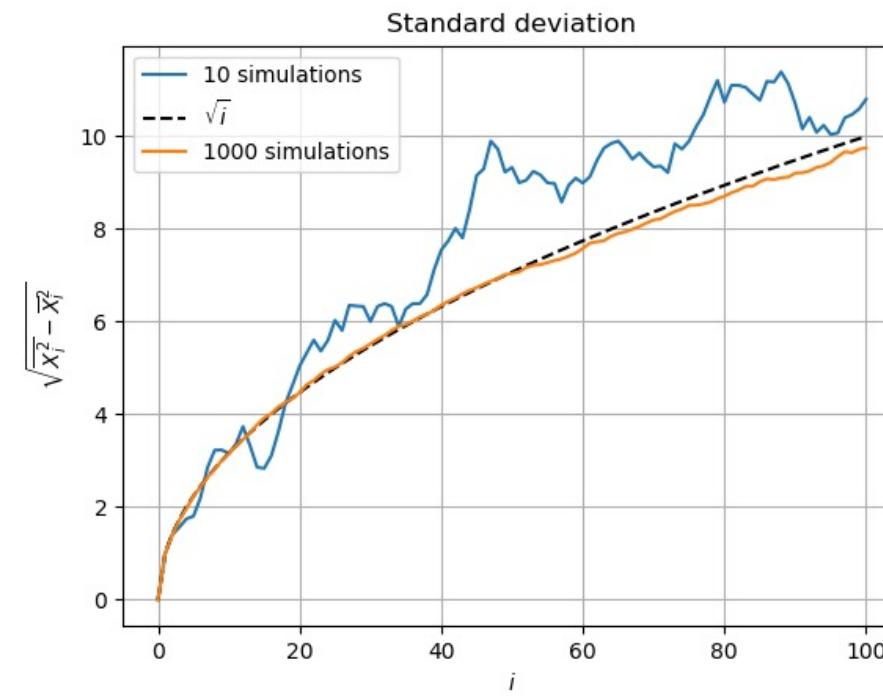
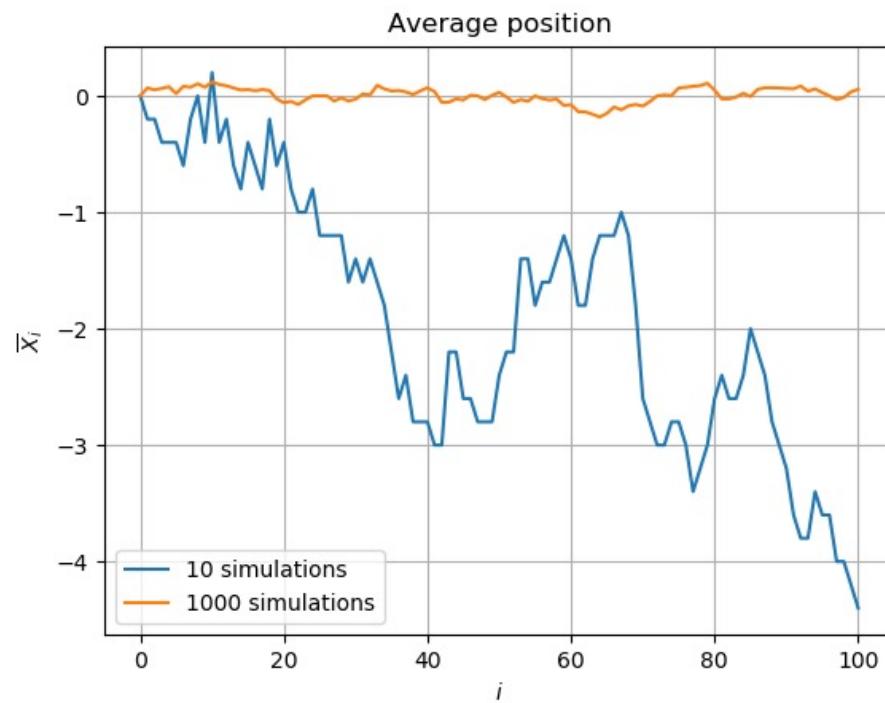
$$\langle R_i \rangle = P(R_i = 1) * 1 + P(R_i = -1) * (-1) = 0$$

- So, $\langle X_{i+1} \rangle = \langle X_i \rangle$. If we require, $X_0 = 0$, and apply linearity of expectation, we find $\langle X_i \rangle = 0$ for all i , and we expect $\bar{X}_i \approx 0$ if M is sufficiently large.
- Similarly, $\langle R_i^2 \rangle = P(R_i = 1) * 1^2 + P(R_i = -1) * (-1)^2 = 1$. Then,

$$\langle X_{i+1}^2 \rangle = \langle X_i^2 \rangle + 2 \langle X_i R_i \rangle + \langle R_i^2 \rangle,$$

and since the “coin flip” is statistically independent of the walker position, $\langle X_i R_i \rangle = \langle X_i \rangle \langle R_i \rangle = 0$. So we have, $\langle X_{i+1}^2 \rangle = \langle X_i^2 \rangle + 1$, and using linearity of expectation, we find, $\langle X_{i+1}^2 \rangle = i + 1$. It is convenient to restate this as $\langle X_i^2 \rangle - \langle X_i \rangle^2 = i$ and we expect $s_i \approx \sqrt{i}$ when M is sufficiently large.

We use $M=10$ and $M=1000$, and the results are below. We see that agreement with theory improves as we increase M , and the results with $M=1000$ indicate that our simulation code is correct



-
- We have a code that appears to be correct and the next thing to think about is its efficiency
 - The key points when working with Numpy are to: avoid loops, vectorize code (using slicing), and use built-in functions as much as possible
 - Looking at our code, we have two loops:

```
X = np.zeros((M,Nt+1))
for j in range(M):
    for i in range(Nt):
        X[j,i+1] = X[j,i]+np.random.choice([-1,1])
```

- Can we make any useful modifications? We have already seen that `np.random.choice` can generate several random numbers at once, so let's precompute all needed random numbers and see if that makes a difference:

```
X = np.zeros((M,Nt+1))
R = np.random.choice([-1,1],(M,Nt))
for j in range(M):
    for i in range(Nt):
        X[j,i+1] = X[j,i]+R[j,i]
```

-
- I've created the function *rwalk1* which uses our original approach:

In [49]: %timeit rwalk1(Nt=100,M=200)

369 ms ± 59.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

- And *rwalk2* implements the modified approach:

In [50]: %timeit rwalk2(Nt=100,M=200)

18.5 ms ± 465 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

- This very simple modification to the code leads to a 20x speedup! And we can do better.
The loop over M simulations can be removed entirely:

```
X = np.zeros((M,Nt+1))
R = np.random.choice([-1,1],(M,Nt))
for i in range(Nt):
    X[:,i+1] = X[:,i]+R[:,i]
```

-
- Running this version:

```
In [56]: %timeit rwalk3(Nt=100,M=200)
```

```
1.54 ms ± 131 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

- And the speed increases by a factor of about 12!
- Numpy is a very powerful package, but it is essential to use it in the right way.
- It is also important to be aware of the built-in functions that are available to us, and I've found the official numpy reference on routines to be pretty useful:
<https://numpy.org/doc/stable/reference/routines.html>

Lecture 10

Solving deterministic initial value problems
Simple 2-equation system
Numerical solution of nonlinear systems

Simple deterministic model

- We have examined a simple stochastic process (random walks), and we will now consider a simple deterministic initial value problem. As with random walks, this will allow us to introduce a few widely-used numerical tools for a problem which can also be analyzed by hand (to an extent).
- The model:

$$\frac{dx}{dt} = ax - bxy$$

$$\frac{dy}{dt} = -y + bxy$$

$$x(0) = x_0, y(0) = y_0$$

Here, a , b , x_0 , and y_0 are non-negative constants which must be specified.

- Let's think about this problem in general terms. We can't write down a solution of the form $x = f(t)$, $y = g(t)$, so what can we do?

-
- We can consider simplifying cases where we *can* write down a solution:
 - Are there *equilibrium solutions* where $\frac{dx}{dt} = \frac{dy}{dt} = 0$?
 - What happens if one or more parameters is zero?
 - What is the behavior of a small perturbation to an equilibrium solution?
 - For a simple model like this one, it is straightforward to work through these questions (as we will do now), but you should also think about what you would do if instead of 2 equations, we had 20, or 200.

Question 1:

- Are there *equilibrium solutions* where $\frac{dx}{dt} = \frac{dy}{dt} = 0$?

- We need to find solutions of:

$$\frac{dx}{dt} = 0 = ax - bxy$$

$$\frac{dy}{dt} = 0 = -y + bxy$$

- By inspection, we can see that $x = y = 0$ is an equilibrium. With a little more work, we can find a second equilibrium: $x = 1/b$, $y = a/b$.

Question 2:

- What happens if one or more parameters is zero?
- $b = 0$ removes the coupling between the two equations giving:

$$\frac{dx}{dt} = ax$$

$$\frac{dy}{dt} = -y$$

- This model is a classic predator-prey system. Removing the coupling means that “rabbits” will reproduce at an exponential rate, $x = x_0 \exp(at)$, while “foxes” will tragically go extinct, $y = y_0 \exp(-t)$

Question 3:

- What is the behavior of a small perturbation to an equilibrium solution?
- Let's focus on the second equilibrium $x = 1/b$, $y = a/b$. The more-formal approach involves introducing a Taylor series expansion about this equilibrium, but I will take a different route to the same result
- First, introduce a power-series expansion for x and y :

$$x = \frac{1}{b} + \epsilon x_1 + \epsilon^2 x_2 + \dots$$

$$y = \frac{a}{b} + \epsilon y_1 + \epsilon^2 y_2 + \dots$$

- Here, $\epsilon \ll 1$, and next, we substitute this expansion into the model equations...

- We find,

$$\epsilon \frac{dx_1}{dt} = \epsilon ax_1 - \epsilon b \left(x_1 \frac{a}{b} + \frac{1}{b} y_1 \right) + \mathcal{O}(\epsilon^2)$$

$$\epsilon \frac{dy_1}{dt} = -\epsilon y_1 + \epsilon b \left(x_1 \frac{a}{b} + \frac{1}{b} y_1 \right) + \mathcal{O}(\epsilon^2)$$

Here, $\mathcal{O}(\epsilon^2)$ represents terms of the form $f(\epsilon)$ or smaller where $f(\epsilon)/\epsilon^2 \rightarrow \text{constant}$ when $\epsilon \rightarrow 0$.

Simplifying the equations and dividing by ϵ gives,

$$\frac{dx_1}{dt} = -y_1 + \mathcal{O}(\epsilon)$$

$$\frac{dy_1}{dt} = ax_1 + \mathcal{O}(\epsilon)$$

- Then, we let $\epsilon \rightarrow 0$ to obtain the governing equations for small perturbations which must be solved

-
- So, we now need to solve, $\frac{d\mathbf{z}}{dt} = \mathbf{A}\mathbf{z}$, where $\mathbf{z} = [x_1, y_1]^T$ and $\mathbf{A} = \begin{bmatrix} 0 & -1 \\ a & 0 \end{bmatrix}$. This matrix has two distinct eigenvalues $\lambda_{1,2} = \pm i\sqrt{a}$ with linearly independent eigenvectors $\mathbf{v}_1, \mathbf{v}_2$.
 - The general solution to our problem is then $\mathbf{z} = c_1 \mathbf{v}_1 \exp(i\sqrt{a}t) + c_2 \mathbf{v}_2 \exp(i\sqrt{a}t)$ and the constants c_1 and c_2 are chosen so the solution satisfies the initial conditions.
 - We can see that in this case, small perturbations will oscillate sinusoidally about the equilibrium. But what does this have to do with computing?

Generalized predator-prey model

- Let's think about a problem where we have n species instead of 2. We can then consider a generalized version of our model:

$$\frac{dx_i}{dt} = a_i x_i + \sum_{j=1}^n B_{ij} x_i x_j, \quad i = 1, 2, \dots, n$$

Here, x_i is a measure of the abundance of species i , a_i is a real number which dictates the success of species i in isolation; B_{ij} is also a real number and determines the nature of the interaction between species i and species j . Usually, $B_{ii} = 0$.

- For our simple 2-species model, we have $B_{ij} = -B_{ji}$ representing a predator-prey interaction, but there are other possibilities. For example, if B_{ij} and B_{ji} are both positive and not equal, then the two species each benefit from the interaction with one benefitting more than the other.
- Let's now think about analyzing this type of n -species model where n is large.

-
- How can we find equilibrium solutions to:

$$a_i x_i + \sum_{j=1}^n B_{ij} x_i x_j = 0, \quad i = 1, 2, \dots, n$$

- This is a linear system of equations (since the x_i terms drop out). Often, however, we have to solve a nonlinear system of equations, and aside from the trivial solution ($x_i = 0$), we will usually need to find solutions numerically. Such solutions can (sometimes) be found using the `root` function in `scipy.optimize`. We won't look at this function in detail here, but let me provide a few notes:
 - There are different methods that `root` can use. The default (`hybr`) is a widely-used and effective method, but `Krylov` may be better for large systems.
 - These methods require the system of equations to be specified via a Python function, and a “guess” for a solution must also be given. An iterative approach then moves from the guess towards a solution one step at a time.
 - The documentation (<https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html#scipy.optimize.root>) includes two examples. Note that the Jacobian is not usually explicitly specified for large problems though it is specified in the first example.

-
- What about small perturbations to an equilibrium state? Let's say that the equilibrium state is $\bar{x}_i, i = 1, 2, \dots, N$, and we'll write our expansion in a slightly different way: $x_i = \bar{x}_i + \epsilon \tilde{x}_i + \mathcal{O}(\epsilon^2)$.
 - The linearized equations for the perturbations are then,

$$\frac{d\tilde{x}_i}{dt} = a_i \tilde{x}_i + \sum_{j=1}^n B_{ij} (\bar{x}_i \tilde{x}_j + \tilde{x}_i \bar{x}_j), \quad i = 1, 2, \dots, n$$

and it is convenient to rewrite these equations in matrix-vector form:

$$\frac{d\tilde{\mathbf{x}}}{dt} = \mathbf{M}\tilde{\mathbf{x}}$$

- We can compute all of the eigenvalues and eigenvectors of \mathbf{M} using `np.linalg.eig`

-
- We can compute all of the eigenvalues and eigenvectors of \mathbf{M} using `np.linalg.eig`
 - For example, for the two species problem:

```
In [2]: a = 1
In [3]: M = np.array([[0,-1],[a,0]])
In [4]: M
Out[4]:
array([[ 0, -1],
       [ 1,  0]])

In [5]: l,v = np.linalg.eig(M)
In [6]: l
Out[6]: array([0.+1.j, 0.-1.j])

In [7]: v
Out[7]:
array([[ 0.70710678+0.j ,  0.70710678-0.j ],
       [ 0. -0.70710678j,  0. +0.70710678j]])
```

-
- And if n linearly independent eigenvectors are found, the general solution can be written as,
$$\tilde{\mathbf{x}} = c_1 \mathbf{v}_1 \exp(\lambda_1 t) + c_2 \mathbf{v}_2 \exp(\lambda_2 t) + \cdots + c_n \mathbf{v}_n \exp(\lambda_n t)$$
 - Applying the initial conditions then gives a system of linear equations which can be solved using `np.linalg.solve` to find the constants c_1, c_2, \dots, c_n
 - But are `np.linalg.eig` and `np.linalg.solve` the right tools to use? It depends on the details of the problem.
 - How do we compute eigenvalues and eigenvectors in Python? Four options (of many) are:
 - `np.linalg.eig`
 - `scipy.sparse.linalg.eigs`
 - `scipy.linalg.eigh`
 - `scipy.sparse.linalg.eigsh`

What are the advantages/disadvantages of these four functions?

Look at the online documentation for these functions to see what they do

Sparse matrices

- When deciding which function(s) to use, the key point to consider is the structure of the matrix, **M**. For example, if many elements in the matrix are zero, we say that the matrix is *sparse* and then tools designed specifically for sparse matrices should be used.
- `scipy.sparse` provides such tools as well as sparse matrix datatypes. The simplest of these takes the following approach: for each non-zero element, store the (i, j) coordinate and the value of that element.
- Example from `scipy.sparse`:

```
>>> # Constructing a matrix using ijk format
>>> from scipy.sparse import coo_matrix
>>> row = np.array([0, 3, 1, 0])
>>> col = np.array([0, 3, 1, 2])
>>> data = np.array([4, 5, 7, 9])
>>> coo_matrix((data, (row, col)), shape=(4, 4)).toarray()
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

Note that `coo_matrix` creates a sparse matrix and `toarray()` converts it to a numpy array.

-
- There are two main advantages to using sparse matrix tools: we can save memory as we are not storing large numbers of zeros, and computations can be considerably more efficient. For example, we won't add a zero to a number or multiply by a zero since we are not storing the zeros.
 - Working in Python (and with `scipy.sparse`), there are a few key points to keep in mind:
 - Building large sparse matrices is non-trivial – see `scipy.sparse` documentation
 - For operations on/with sparse matrices, look at *methods* for a sparse matrix, and look at functions in `scipy.sparse.linalg`.
 - It is important to look carefully at the guidance at the bottom of the main documentation page:
<https://docs.scipy.org/doc/scipy/reference/sparse.html>
 - Next, we will look at how to compute numerical solutions to the full initial value problem without making any simplifying assumptions.

Simulating dynamical processes

- Analytical solutions typically do not exist for systems of interest. For example, model coefficients could vary with time, or the equations could be nonlinear:

$$\frac{dx_i}{dt} = a_i(t)x_i + \sum_{j=1}^n B_{ij}x_i x_j$$

- As we have seen, we can sometimes make useful simplifying approximations, but if we want to solve the “full” problem, we have to turn to computational simulations. The results will be approximate, but typically we can control the error and reduce it to an acceptable level.
- Simulation of systems of nonlinear ODEs are an essential part of modern applied mathematics, science, and engineering

Let's think about simulating generic dynamical processes which can be modeled by equations of the form:

$$\frac{\partial x_i}{\partial t} = \mathcal{F}_i(t, x_1, x_2, \dots, x_n), \quad i \in \{1, 2, \dots, n\}$$

with initial conditions at $t = 0$ specified.

- Basic idea: discretize time, $t = 0, \Delta t, 2\Delta t, \dots, N_t * \Delta t$, and starting from $x_i(t = 0)$ march forward in time and compute $x_i(\Delta t), \dots, x_i(N_t * \Delta t)$
- We can design such an approach using a Taylor series expansion:

$$x_i(t_0 + \Delta t) = x_i(t_0) + \Delta t \frac{dx_i}{dt} \Big|_{t_0} + \mathcal{O}(\Delta t^2)$$

$$x_i(t_0 + \Delta t) \approx x_i(t_0) + \Delta t \mathcal{F}_i(t_0, x_j(t_0)) \quad (\text{explicit Euler method})$$

- Accuracy: The approximation error for a single step is $\mathcal{O}(\Delta t^2)$, however the error will accumulate, and the convention is to say the global error is $\sim \mathcal{O}(\Delta t)$ \rightarrow smaller time step, more accurate solution

-
- In addition to the accuracy of a numerical method, we also need to consider its stability: do numerical solutions stay bounded?
 - Stability: Applying the explicit-Euler method to the test problem, $\frac{dx}{dt} = ax, a < 0$, the numerical solution can be unstable → with sufficiently large Δt , the solution will eventually become unbounded (blows up). If a is imaginary, the method is unstable at long times for *any* Δt
 - Stability is often a substantial concern for nonlinear systems. These systems often contain rapidly varying components which affect stability much more than accuracy
 - In such cases, it is often helpful to use *implicit* methods:

$$x_i(t_0 + \Delta t) \approx x_i(t_0) + \Delta t \mathcal{F}_i(t_0 + \Delta t, x_j(t_0 + \Delta t)) \quad (\text{implicit Euler method})$$

- Note that we now have $x_j(t_0 + \Delta t)$ on the RHS rather than $x_j(t_0)$

-
- Accuracy: global error is again $\sim \mathcal{O}(\Delta t)$ \rightarrow smaller time step, more accurate solution
 - Stability: for $\frac{dx}{dt} = ax, a < 0$, the method is *unconditionally stable* \rightarrow bounded solution at all times for any time-step
 - For a linear system, this method typically requires solution of a system of equations each time step
 - However for a nonlinear system, a system of nonlinear equations must be solved and then further approximations are needed

-
- Simple example:

$$\frac{dx_1}{dt} = -ax_2$$

$$\frac{dx_2}{dt} = -bx_1,$$

$$x_1(0) = c, x_2(0) = d$$

$$a > 0, b > 0$$

- Explicit Euler:

$$x_1(t_0 + \Delta t) = x_1(t_0) - a \Delta t x_2(t_0)$$

$$x_2(t_0 + \Delta t) = x_2(t_0) - b \Delta t x_1(t_0)$$

- Implicit Euler:

$$x_1(t_0 + \Delta t) = x_1(t_0) - a \Delta t x_2(t_0 + \Delta t)$$

$$x_2(t_0 + \Delta t) = x_2(t_0) - b \Delta t x_1(t_0 + \Delta t)$$

or $\begin{bmatrix} 1 & a\Delta t \\ b\Delta t & 1 \end{bmatrix} \begin{bmatrix} x_1(t_0 + \Delta t) \\ x_2(t_0 + \Delta t) \end{bmatrix} = \begin{bmatrix} x_1(t_0) \\ x_2(t_0) \end{bmatrix}$

-
- I have outlined the Euler methods just to give you an idea of how initial value problems are solved numerically; in practice these methods are not widely used for deterministic systems.
 - Many methods have been developed which improve on the accuracy and stability of the Euler methods
 - Variable time-step implicit methods are particularly popular for nonlinear systems
 - A time-step is broken up into sub-steps and the size of the sub-steps is automatically adjusted to ensure a specified accuracy
 - We will not analyze these methods in any detail at all, but here is a sketch:
 - The dependence of the solution on the time step can be used to estimate the error
 - Let $E^{(m)}$ be this estimated error for the m^{th} time step. In other words, $E^{(m)}$ is an estimate for $|x_{i,\text{exact}}^{(m)} - x_{i,\text{computed}}^{(m)}|$ (here $|x_i^{(m)}|$ is the length of the vector $[x_1^{(m)}, x_2^{(m)}, \dots]^T$)
 - The user specifies tolerances for the absolute and relative errors ($\epsilon_{\text{abs}}, \epsilon_{\text{rel}}$)
 - The method then selects the sub-step size so that $E^{(m)} \leq \epsilon_{\text{abs}} + |x_{i,\text{computed}}^{(m)}| \epsilon_{\text{rel}}$

-
- `scipy.integrate` contains a number of functions for solving IVPs
 - The recommended approach is to use `solve_ivp` and to specify the method in the function call
 - For nonlinear problems, it is recommended to initially set method to ‘BDF’ which uses “backward differentiation formula”. This is a highly-robust, highly-sophisticated implicit variable time-step method, but there is no one method that works best for all problems, and some experimentation with different methods is often beneficial.
 - The user has to specify:
 - 1) initial conditions
 - 2) time span
 - 3) function which evaluates RHS of ODEs
 - The user can also choose to specify:
 - 1) error tolerances
 - 2) times at which to return solution
 - 3) many other parameters

Look at the online documentation!

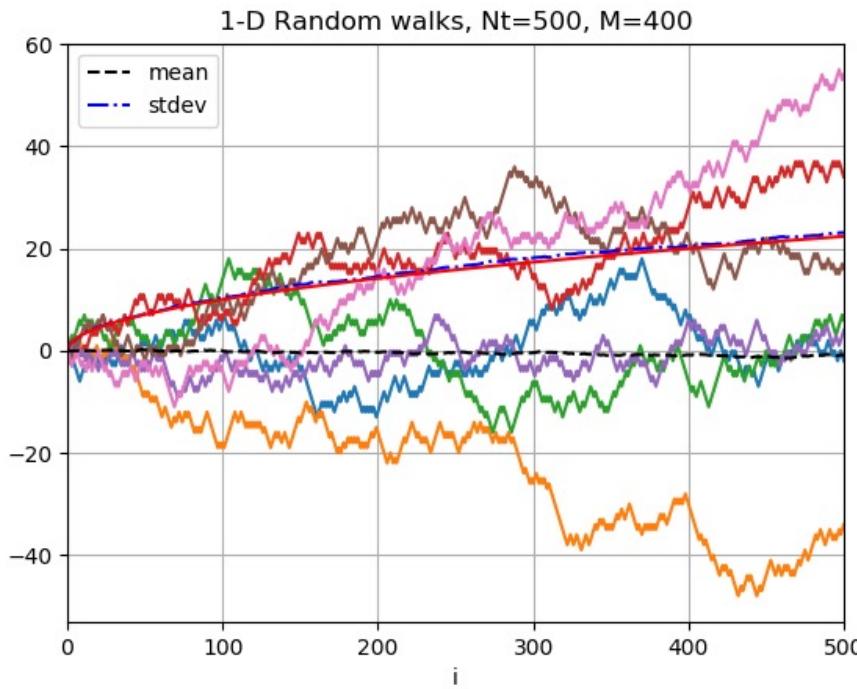
Lecture 11

Random walks and Brownian motion
Stochastic differential equations

Overview

- We've introduced a simple stochastic process (random walks) as well as a fairly simple deterministic problem (predator-prey model) and then considered how to use numpy and scipy to analyze more-complicated deterministic problems
- We'll now think about more-complicated stochastic models starting with "classical" Brownian motion and then moving on to general stochastic differential equations
 - We'll see that the explicit Euler method, with some modest modification, will be useful here
 - And we'll also look more carefully at the accuracy and stability of our numerical simulations than we did for the deterministic case
- Why are we spending this time on stochastic processes? They are relevant to a large number of applications including the spread of heat in your laptop, the diffusion of particles in the air, the spread of Covid through communities, and the behavior of prices in financial markets

Random walks recap



- We have discussed 1-D random walks which are a simple but important and widely-used stochastic model
- The expected position of a random walker after i steps is $\langle X_i \rangle = 0$ while the variance is, $\langle X_i^2 \rangle - \langle X_i \rangle^2 = i$ when the step size is $\Delta x = 1$.
- We have observed that the “ensemble average” of simulation results can be used to approximate expected values. E.g. $\bar{X}_i \approx \langle X_i \rangle$ where $\bar{X}_i = \frac{1}{M} \sum_{j=1}^M X_i^{(j)}$ and M is sufficiently large. ($X_i^{(j)}$ is the i^{th} step in the j^{th} simulation).



- We can generalize our random walk model for steps with length, Δx :
$$X_{i+1} = X_i + \Delta x R_i$$
- Then, $\langle X_i^2 \rangle - \langle X_i \rangle^2 = i\Delta x^2$
- Now, say that each step requires time δt , and $t_i = i\delta t$. We then have,
$$\langle X_i^2 \rangle - \langle X_i \rangle^2 = t_i \left(\frac{\Delta x^2}{\delta t} \right)$$
- Generally, Δx and δt are not independent, and in many applications, it is reasonable (and conventional) to set $\frac{\Delta x^2}{\delta t} = 2\alpha$ where α is the *diffusivity*. The specific value of α depends on the application of interest, e.g. ink particles in water vs. perfume particles in air
- The key takeaway then is that the variance increases linearly with time – a very general result that has been observed for many, many different stochastic processes.

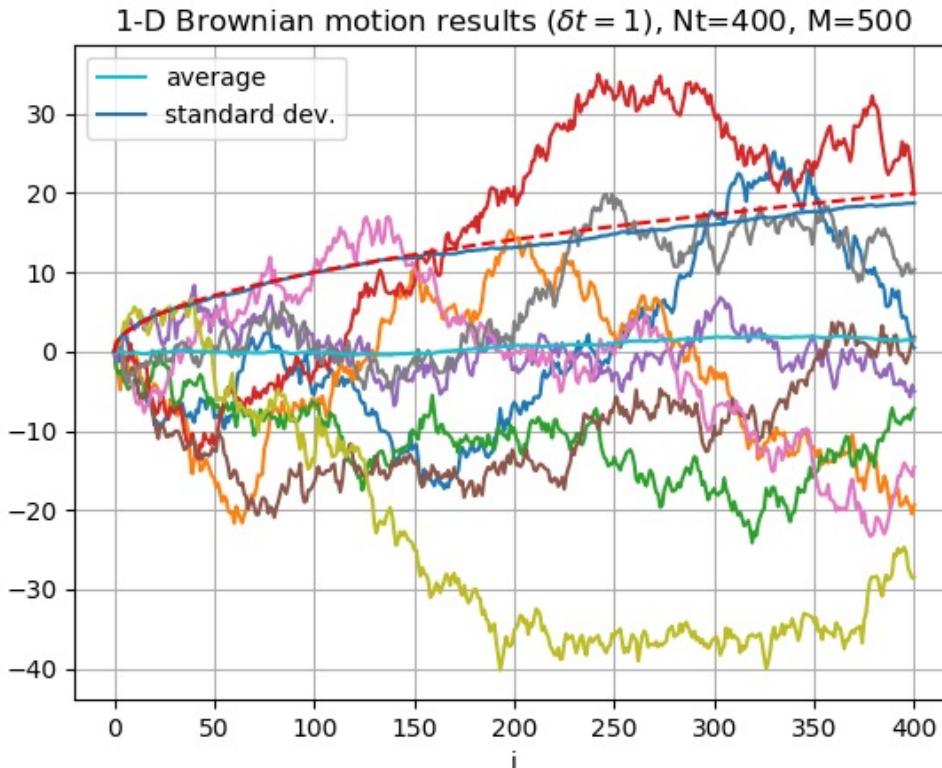
Discrete Brownian motion

- We should of course think critically about the random walk model.
 - Generally, particles will move in three dimensions rather than 1, but this is easily accounted for by simultaneously computing three random walks, one for each of three Cartesian coordinates (X_i, Y_i, Z_i) .
 - A more substantial weakness is the assumption that the walker always moves with a particular stepsize. This weakness is directly addressed in discretized *Brownian motion*
- A timestep of duration δt of one-dimensional Brownian motion is defined by:

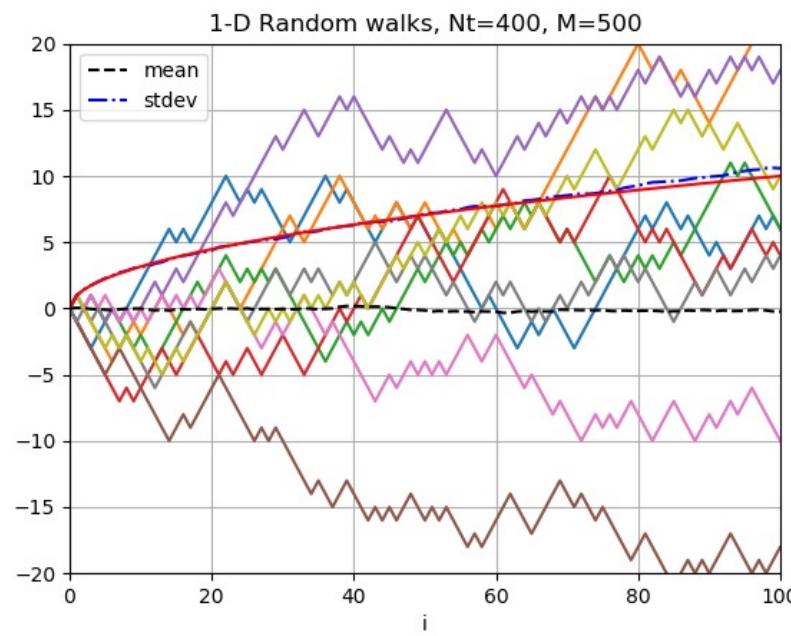
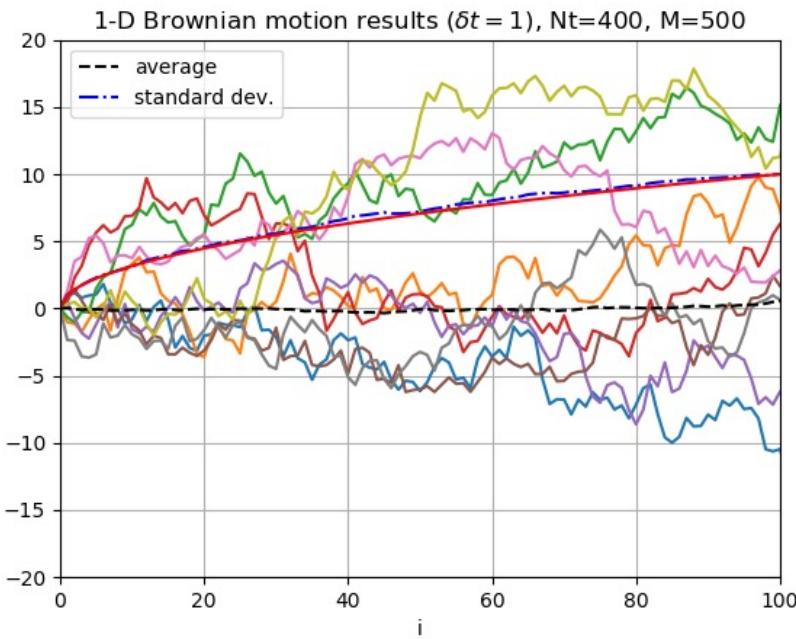
$$X_{i+1} = X_i + \sqrt{2\alpha\delta t}B_i$$

where B_i is a normally-distributed random variable with mean=0 and variance=1: $B_i \sim \mathcal{N}(0,1)$. Frequently, we set $2\alpha = 1$.

- Applying the same methodology we used for random walks, we find: $\langle X_i \rangle = 0$ and $\langle X_i^2 \rangle = 2i\alpha\delta t = 2\alpha t_i$ so the variance, $\langle X_i^2 \rangle - \langle X_i \rangle^2$, again increases linearly with time.



- It is straightforward to modify our random walk code to simulate Brownian motion – essentially, we just replace `np.random.choice` with `np.random.normal`
- The computed results “look” similar to what we had with random walks as we expect based on the expected values. If we zoom in (see next slide), we notice that a range of stepsizes are used instead of just ± 1
- This can be viewed as a more realistic model than random walks, at least if we are thinking about particles moving in air or water, but it is still imperfect – don’t particles (if they are large enough) follow Newton’s laws of motion?



As noted on the previous slide, the difference between the Brownian motion and random-walk steps becomes more apparent when we zoom in

-
- The foundational analysis of Brownian motion was presented by Einstein 1905, and subsequently there has been considerable related work in both statistics and physics.
 - For example, Langevin modified the Brownian motion model to account for Newton's second law of motion:

$$\begin{aligned}X_{i+1} &= X_i + \delta t V_i \\V_{i+1} &= V_i - \gamma \delta t V_i + \sqrt{2\alpha\delta t} B_i\end{aligned}$$

- Here, V_i is the particle velocity at time $t_i = i\delta t$, and $-\gamma V_i$ is a frictional *damping* force whose strength is set by the parameter, γ
- Working in parallel with physicists, mathematicians have built the theory of *Markov processes*.
- It is natural to think about what happens when $\delta t \rightarrow 0$. We could rearrange the 1st equation above as $\frac{(X_{i+1}-X_i)}{\delta t} = V_i$, and if X_i was sufficiently smooth we would have $\frac{dX}{dt} = V$ when $\delta t \rightarrow 0$ and $t = t_i$

-
- But what about the second equation, $V_{i+1} = V_i - \gamma\delta t V_i + \sqrt{2\alpha\delta t}B_i$ and the noise term in particular? We could rearrange this as, $\frac{(V_{i+1}-V_i)}{\delta t} = -\gamma V_i + \sqrt{\frac{2\alpha}{\delta t}} B_i$, however the limit of the second term on the RHS doesn't exist when $\delta t \rightarrow 0!$
 - The rules of *stochastic calculus* help us manage this issue and lead to the consideration of *stochastic differential equations*. I will just state the main results we will use and will generally skip most theoretical details and derivations.
 - If you're interested in the underlying theory, a widely-used reference is *Stochastic Differential Equations* by Oksendal
 - We will now consider a stochastic process over the interval $[0, T]$, and we discretize time as $t_i = i\delta t$ with $T = n\delta t$

Stochastic differential equations

- Our goal now is to interpret and then compute solutions to equations in the following form:

$$dX(t) = f(X(t))dt + g(X(t))dW(t)$$

- This equation doesn't really have a formal meaning. It is just shorthand for an expression involving integrals:

$$X(T) - X(0) = \int_0^T f(X(s))ds + \int_0^T g(X(s))dW(s)$$

- The first integral is exactly the same as what we have for ODEs
- The second equation requires interpretation. We will use the *Ito integral*:

$$\int_0^T g(X(s))dW(s) = \lim_{\delta t \rightarrow 0} \sum_{i=0}^n g(X(t_i)) [W(t_{i+1}) - W(t_i)]$$

with $t_i = i\delta t$, $T = n\delta t$. We will call $W(t_{i+1}) - W(t_i)$ a “Brownian step”:

$$W(t_{i+1}) - W(t_i) = \sqrt{\delta t} B_i$$

-
- Having defined a family of stochastic differential equations, the question now is: how do we compute solutions given the functions $f(X)$ and $g(X)$?
 - Note: we are assuming that $f(X)$ and $g(X)$ are “well-behaved” (continuous, bounded, and square integrable on $[0, \infty)$)
 - We have discussed the explicit Euler method, and here we will use the *Euler-Maruyama* (E-M) method to compute solutions over the interval $[0, T]$
 - We will use a second discretization of time: $\tau_j = j\Delta t$ where $\Delta t = a\delta t$ and a is a positive integer (so $\Delta t \geq \delta t$). This will help us analyze the method’s accuracy here, and in most other situations we can just set $a = 1$.
 - Then, given the solution at time τ_j , we compute the solution at τ_{j+1} using:

$$X_{j+1} = X_j + \Delta t f(X_j) + g(X_j)[W(t_{j+1}) - W(t_j)]$$

The term in the square brackets is computed using a separate Brownian motion simulation:

$$W(t_i + \delta t) = W(t_i) + \sqrt{\delta t} B_i \quad (W(0) = 0)$$

-
- Let's look at a concrete example. Set $f(X) = \lambda X$, $g(X) = \mu X$. Then, our SDE is:

$$X(T) - X(0) = \lambda \int_0^T X(s) ds + \mu \int_0^T X(s) dW(s)$$

- and we expect exponential growth ($\sim \exp(\lambda t)$) in the deterministic case. The solution for the full problem is:

$$X(T) = X(0) \exp\left[\left(\lambda - \frac{1}{2}\mu^2\right)T + \mu W(T)\right]$$

- We can interpret $W(T)$ as the position at time T resulting from Brownian motion with some arbitrary δt (a more formal definition is provided in the slides at the end of this lecture)
- For the E-M method, we will initially choose $\delta t = 10^{-8}$, and $\Delta t = 4\delta t$. Then, our update equations will be:

$$X_{j+1} = (1 + \lambda \Delta t)X_j + \mu X_j [W(t_{j+1}) - W(t_j)]$$

$$W(t_i + \delta t) = W(t_i) + \sqrt{\delta t} B_j$$

Note that 4 Brownian motion steps are needed to compute $W(t_{j+1})$ given $W(t_j)$

-
- The basic steps are:
 - 1) Set model and numerical parameters
 - 2) Simulate Brownian motion and compute the needed $W(t_{j+1}) - W(t_j)$ values
 - 3) Using the E-M update formula, start from the initial condition (we'll use $X_0 = 1$) and march forward in time

Step 1):

```
#set model parameters
T = 1
l = 1
mu = 0.5
X0 = 1

#set numerical parameters
M = 1000
nt = 2**9
dt = T/nt
a = 4
Nt = nt//a
Dt = T/Nt
fac = 1 + l*Dt
```

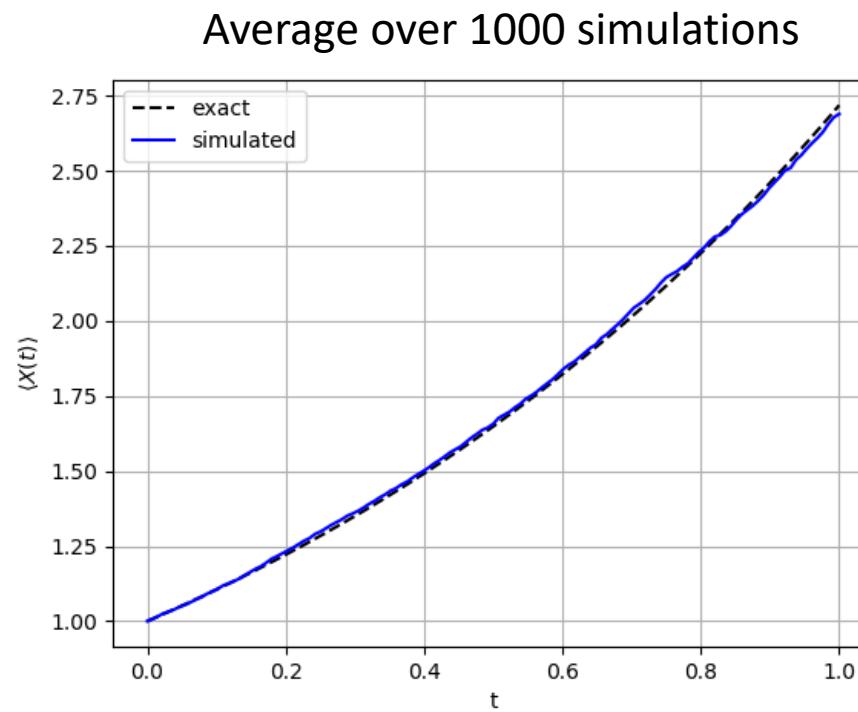
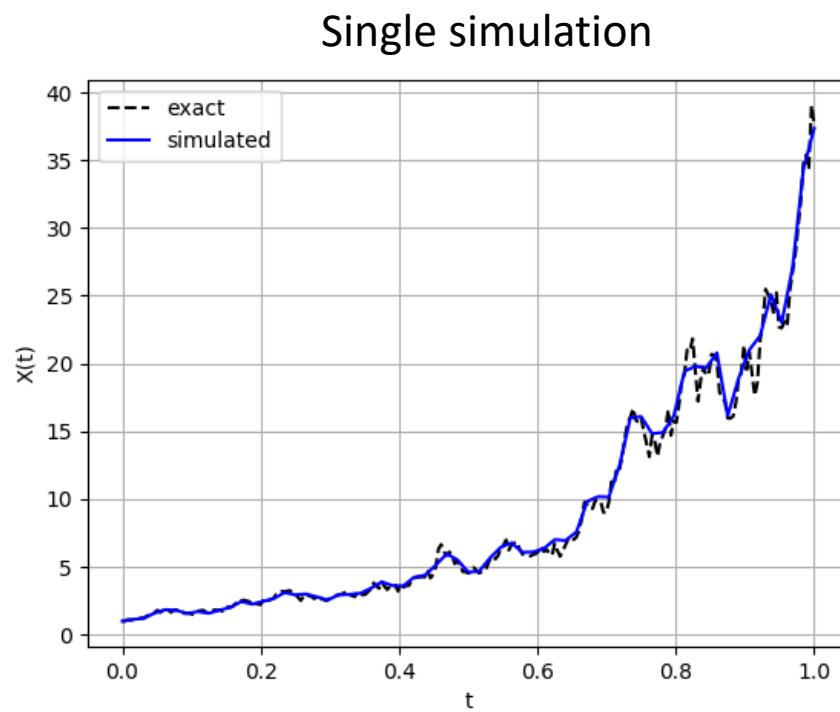
Step 2) This is very similar to random walk simulations, but I have used the cumulative sum function instead of a loop with nt iterations:

```
#M nt-step Brownian motion simulations
B = np.zeros((nt+1,M))
B[1:,:] = np.sqrt(dt)*np.random.normal(size=(nt,M))
W = np.cumsum(B, axis=0) #previously we used a loop with nt iterations
```

Step 3)

```
#Apply E-M method for M Nt-step simulations
X = np.zeros((Nt+1,M))
X[0,:] = X0
dW = W[a::a,:]-W[:a:a,:]
#Iterate over Nt time steps
for j in range(Nt):
    X[j+1,:] = fac*X[j,:]+mu*X[j,:]*dW[j,:] #E-M update equation
```

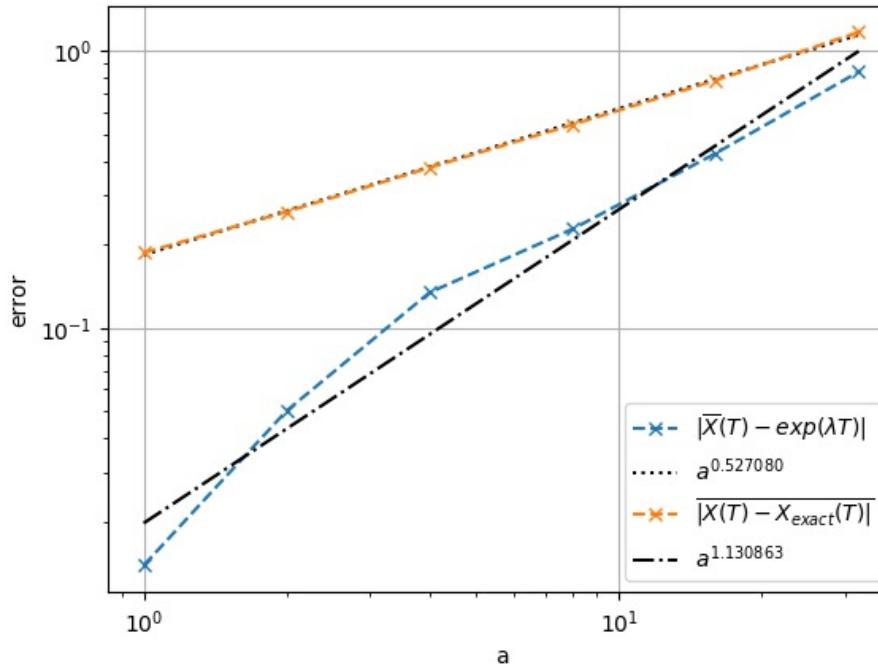
- The figure below on the left shows a single simulation while on the right we compare the ensemble average over 1000 simulations to the expectation, $\langle X(T) \rangle = X_0 \exp(\lambda T)$



- So the code appears to be correct. Let's now look more closely at the accuracy of the E-M method

-
- We stated that the global error for the explicit Euler method decreases linearly with the time step, Δt . Do we see something similar with the E-M method? It depends on how we define the error.
 - Define the error as $\epsilon_w(T) = |\bar{X}_j - \langle X(T) \rangle|$ with $T = j\Delta t$. It can be shown that $\epsilon_w(T) \sim \Delta t$.
 - More precisely, we say that the E-M method has *weak order of convergence* equal to 1. A method has weak order of convergence equal to γ if there exists a constant, C , such that $\epsilon_w(\tau) \leq C\Delta t^\gamma$ for any fixed $\tau \in [0, T]$.
 - The weak order of convergence considers the difference in the expectation. What about the expectation of the difference? A method has *strong order of convergence* equal to γ if there exists a constant, C , such that $\epsilon_s(\tau) = \langle |X_j - X(\tau)| \rangle \leq C\Delta t^\gamma$ for any fixed $\tau = j\Delta t \in [0, T]$ and Δt sufficiently small. The E-M method has strong order of convergence equal to $\frac{1}{2}$.
 - Let's see if our code reproduces these theoretical results. We will use $\delta t = 2^{-9}$, $M = 40000$, $\lambda = 2$, $\mu = 1$, $T = 1$, and $a = 1, 2, 4, 8, 16, 32$. We expect the error to decrease as a decreases.

- The figure below shows $\epsilon_w(T)$ and our approximation for the “strong error”: $|X_j - X(\tau)|$. The ensemble average replaces the expectation in the equation for ϵ_s .



- The estimated rates of convergence (0.53 and 1.13) are close to the theoretical results. Note that this second estimate can fluctuate quite a bit from one set of simulations to the next. A more robust approach for analyzing weak convergence is to set $a = 1$, compute $\epsilon_w(T)$ several times, and then average the results.

-
- The last point we want to consider is the *stability* of the E-M method at long times when applied to our example linear SDE, $X(T) - X(0) = \lambda \int_0^T X(s)ds + \mu \int_0^T X(s)dW(s)$
 - For the explicit Euler method, if we consider the ODE $\frac{dx}{dt} = \lambda x$, the discrete solution is, $x_n = (1 + \lambda \Delta t)^n x_0$, and the condition $|(1 + \lambda \Delta t)| \leq 1$ must be satisfied for the numerical solution to remain bounded
 - So if λ is real and negative and $\Delta t > -\frac{2}{\lambda}$ the numerical solution will “blow up” at long times even though the exact solution decays exponentially. If λ is imaginary, the solution will always blow up!

-
- What is the **analytical** long-time behavior of the linear SDE? There are two types of stability to consider:
 - *mean-square stability*: $\lim_{T \rightarrow \infty} \langle X(T)^2 \rangle = 0 \leftrightarrow \mathcal{R}(\lambda) + \frac{1}{2}|\mu|^2 < 0$
 - *asymptotic stability*: $P\left(\lim_{T \rightarrow \infty} |X(T)| = 0\right) = 1 \leftrightarrow \mathcal{R}\left(\lambda - \frac{1}{2}\mu^2\right) < 0$
 - What about **numerical** stability?
 - mean-square numerical stability: $\lim_{j \rightarrow \infty} \langle X_j^2 \rangle = 0 \leftrightarrow |1 + \Delta t \lambda|^2 + \Delta t |\mu|^2 < 1$
 - asymptotic numerical stability:
$$P\left(\lim_{j \rightarrow \infty} |X_j| = 0\right) = 1 \leftrightarrow \langle \log |1 + \Delta t \lambda + \sqrt{\Delta t} \mu \mathcal{N}(0,1)| \rangle < 0$$
 - The mean-square result should be compared to what we found for the explicit-Euler method.
 - The key point here is that numerical solutions may become unbounded at large times even if the exact solution is bounded if the time step is too large.

A few mathematical details (optional)

- What is $W(t)$ in the solution to an SDE?
 - $W(t)$ is a random variable with $W(0) = 0$ that depends continuously on $t \in [0, T]$ and which satisfies the following conditions:
 - For $0 \leq s < t \leq T$, $W(t) - W(s) \sim \sqrt{t-s}\mathcal{N}(0,1)$
 - For $0 \leq s < t < u < v \leq T$, $W(t) - W(s)$ and $W(v) - W(u)$ are statistically independent
 - Note that the solution to our discretized Brownian motion model satisfies these conditions
- Why is $X(T) = X(0) \exp\left[\left(\lambda - \frac{1}{2}\mu^2\right)T + \mu W(T)\right]$ a solution to:
$$X(T) - X(0) = \lambda \int_0^T X(s)ds + \mu \int_0^T X(s)dW(s)?$$
- To answer this question, we need to use a form of *Ito's lemma* which extends the chain rule from calculus to stochastic processes.

-
- Let $h(t, x)$ be a real-valued function with continuous partial 2nd derivatives. It follows from Ito's lemma that:

$$h(t, W(t)) - h(0, W(0)) = \int_0^t \left[h_t + \frac{1}{2} h_{xx} \right]_{t=s, x=W(s)} ds + \int_0^t [h_x]_{t=s, x=W(s)} dW(s)$$

- For our example SDE, $h(t, x) = X(0) \exp[\left(\lambda - \frac{1}{2}\mu^2\right)t + \mu x]$
- Computing the needed partial derivatives and using the result above gives,

$$h(t, W(t)) - h(0, W(0)) = \lambda \int_0^t h(s, W(s)) ds + \mu \int_0^t h(s, W(s)) dW(s)$$

which is equivalent to $X(T) - X(0) = \lambda \int_0^T X(s) ds + \mu \int_0^T X(s) dW(s)$ and which shows that $X(T) = X(0) \exp[\left(\lambda - \frac{1}{2}\mu^2\right)T + \mu W(T)]$ is a solution to this equation.

Lecture 12

Maximum growth

Computation of eigenvalues and singular value decomposition

Overview

- We will now work through a sequence of problems that will “gently” take us from dynamical processes to data science
- The common threads connecting these problems will be ideas from linear algebra and tools from computational linear algebra (implemented in Numpy and Scipy)

Maximum growth

- How do you interpret: $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m, \mathbf{A} \in \mathbb{R}^{m \times n}$?
 - If \mathbf{A} and \mathbf{b} are known, and $n = m$, this could be a linear system of equations which we want to solve for \mathbf{x}
 - Or if the system is overdetermined and $m > n$, we would typically look for an approximate solution
- If \mathbf{A} and \mathbf{x} are known: this is a linear transformation of \mathbf{x} to \mathbb{R}^m
- What if only \mathbf{A} is known? Again interpreting this as a linear transformation, we can then search for the \mathbf{x} that produces \mathbf{b} with some desirable property
- Let's consider the maximum growth problem where given \mathbf{A} , we find the \mathbf{x} that produces \mathbf{b} with maximum magnitude

-
- As stated, this is not a particularly interesting question. Since the problem is linear, if we scale \mathbf{x} with a constant, then \mathbf{b} will also be scaled.
 - So we modify our problem statement to:

Given a matrix \mathbf{A} , find \mathbf{x} so that $|\mathbf{Ax}|/|\mathbf{x}|$ is maximized

To find this vector \mathbf{x} , we will:

1. Set an upper bound for $|\mathbf{Ax}|/|\mathbf{x}|$
2. Determine how to reach this upper bound

It will be convenient to work with the magnitudes squared, $|\mathbf{Ax}|^2$ and $|\mathbf{x}|^2$, rather than the magnitudes themselves

Task 1: Set an upper bound for $|\mathbf{Ax}|^2$

- Starting from $\mathbf{Ax} = \mathbf{b}$, we know, $\mathbf{b}^T \mathbf{b} = \mathbf{x}^T \mathbf{A}^T \mathbf{Ax}$, and the key point to recognize here is that $\mathbf{A}^T \mathbf{A}$ is symmetric
- This is important because symmetric matrices can be orthogonally diagonalized
- Say that \mathbf{B} is real and symmetric, then $\mathbf{B} = \mathbf{V}\mathbf{S}\mathbf{V}^T$ where:
 - \mathbf{V} is an orthogonal matrix ($\mathbf{V}^T \mathbf{V} = \mathbf{I}$) whose columns are eigenvectors of \mathbf{B}
 - \mathbf{S} is a diagonal matrix with the (real) eigenvalues of \mathbf{B} on its diagonal
- For our problem we have, $\mathbf{A}^T \mathbf{A} = \mathbf{V}\mathbf{S}\mathbf{V}^T$, and $\mathbf{b}^T \mathbf{b} = \mathbf{x}^T \mathbf{V}\mathbf{S}\mathbf{V}^T \mathbf{x}$
- Now, let $\mathbf{z} = \mathbf{V}^T \mathbf{x}$, we then have $\mathbf{b}^T \mathbf{b} = \mathbf{z}^T \mathbf{S} \mathbf{z}$
- Say that the eigenvalues are arranged so that $S_{ii} = \lambda_i$ and $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$
- Then, $\mathbf{z}^T \mathbf{S} \mathbf{z} = \lambda_1 z_1^2 + \lambda_2 z_2^2 + \dots + \lambda_n z_n^2 \leq \lambda_1 (z_1^2 + z_2^2 + \dots + z_n^2)$
 - Here, $\mathbf{z} = [z_1, z_2, \dots, z_n]^T$

-
- Our inequality from the last slide tells us that, $\mathbf{b}^T \mathbf{b} \leq \lambda_1 \mathbf{z}^T \mathbf{z}$
 - We know that $\mathbf{z}^T \mathbf{z} = \mathbf{x}^T \mathbf{V} \mathbf{V}^T \mathbf{x} = \mathbf{x}^T \mathbf{x}$, so, $\mathbf{b}^T \mathbf{b} \leq \lambda_1 \mathbf{x}^T \mathbf{x}$ or $|\mathbf{b}|^2 \leq \lambda_1 |\mathbf{x}|^2$
 - Our 2nd task is to find \mathbf{x} so that $|\mathbf{b}|^2 = \lambda_1 |\mathbf{x}|^2$
 - This task is easier, we just let $\mathbf{x} = \mathbf{v}_1$ where $\mathbf{A}^T \mathbf{A} \mathbf{v}_1 = \lambda_1 \mathbf{v}_1$ (i.e. \mathbf{v}_1 is the leading eigenvector of $\mathbf{A}^T \mathbf{A}$)
 - Then, $|\mathbf{x}|^2 = 1$ and $|\mathbf{b}|^2 = |\mathbf{A} \mathbf{v}_1|^2 = \lambda_1 \mathbf{v}_1^T \mathbf{v}_1 = \lambda_1$ since the eigenvectors in \mathbf{V} are orthonormal.
 - So, if we choose \mathbf{x} to be the leading eigenvector of $\mathbf{A}^T \mathbf{A}$, then \mathbf{Ax} will generate a vector such that, $\frac{|\mathbf{Ax}|}{|\mathbf{x}|} = \sqrt{\lambda_1}$

This is the maximum growth that can be obtained via the transformation $\mathbf{Ax} = \mathbf{b}$, and setting \mathbf{x} to be the leading eigenvector of \mathbf{A} will produce this growth

-
- Why is this maximum growth important?
 - Consider a system of linear ODEs: $\frac{dx}{dt} = \mathbf{M}\mathbf{x}$, $\mathbf{x}(t = 0) = \mathbf{x}_0$
 - We can write the solution to this problem in terms of the *matrix exponential*: $\mathbf{A}(t) = \exp(\mathbf{M}t)$: $\mathbf{x}(t) = \mathbf{A}(t)\mathbf{x}_0$
 - Then we see that the “most dangerous” initial condition which produces the maximum growth at time t^* is the leading eigenvector of $\mathbf{A}(t^*)^T \mathbf{A}(t^*)$ and then the maximum growth will be, $\frac{|\mathbf{x}(t^*)|^2}{|\mathbf{x}_0|^2} = \lambda_1$, the leading eigenvalue of $\mathbf{A}^T \mathbf{A}$
 - A similar situation arises when considering difference equations of the form,
$$\mathbf{x}^{(i+1)} = \mathbf{M}\mathbf{x}^{(i)}$$
 - Here, the superscript indicates an iteration number, and $\mathbf{x}^{(p)} = \mathbf{M}^p \mathbf{x}^{(0)}$, and we have another maximum growth problem with $\mathbf{A} = \mathbf{M}^p$

-
- Previously, when we encountered systems of linear ODEs of the form, $\frac{dx}{dt} = \mathbf{M}\mathbf{x}$, $\mathbf{x}(t = 0) = \mathbf{x}_0$, we constructed solutions in terms of the eigenvalues and eigenvectors of \mathbf{M}
 - Should we have considered the maximum growth problem there? The basic question to consider is whether or not the eigenvectors are orthogonal.
 - To see why, let's assume that the matrix \mathbf{M} is symmetric.
 - Then, the solution to the system above can be written in terms of the (orthogonal) eigenvectors and eigenvalues of \mathbf{M} :

$$\mathbf{x}(t) = C_1 \mathbf{v}_1 e^{\lambda_1 t} + C_2 \mathbf{v}_2 e^{\lambda_2 t} + \cdots + C_n \mathbf{v}_n e^{\lambda_n t}$$

- And with a little work (which I will avoid here), we can show that the maximum growth at time t is simply $e^{\lambda_1 t}$ where λ_1 is the leading eigenvalue of \mathbf{M} and the initial condition that produces maximum growth is the leading eigenvector, \mathbf{v}_1
- For symmetric matrices, all of the needed “information” is stored in the eigenvalues and eigenvectors of the matrix itself; there is no need to separately look at, $\mathbf{A}^T \mathbf{A}$

- Let's look at a simple example:

$$\frac{d\mathbf{x}}{dt} = \mathbf{M}\mathbf{x}, \quad \mathbf{M} = \begin{bmatrix} -1 & a \\ 0 & -2 \end{bmatrix}$$

And we want to find $\mathbf{x}(t = 0)$ such that $|\mathbf{x}(t = 1)|/|\mathbf{x}(t = 0)|$ is maximized.

- The eigenvalues of \mathbf{M} are $\lambda_1 = -1$ and $\lambda_2 = -2$

Case 1: $a = 0$

Here, \mathbf{M} is symmetric with eigenvectors $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and the maximum growth is $e^{\lambda_1 t^*} = e^{-1}$ with $\mathbf{x}(t = 0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. This is minimal decay rather than maximum growth!

Case 2: $a = -30$. Setting $\mathbf{A}(t) = \exp(\mathbf{M}t)$, the leading eigenvalue and eigenvector of $\mathbf{A}^T \mathbf{A}$ are 48.8 and $\begin{bmatrix} 0.0526 \\ -0.999 \end{bmatrix}$ so here, even though the eigenvalues are negative, we have $\frac{|\mathbf{x}(t=1)|}{|\mathbf{x}(t=0)|} = \sqrt{48.8} \approx 7$

Computing eigenvalues

- To find solutions to the maximum growth problem, eigenvalues and eigenvectors of $\mathbf{F} = \mathbf{A}^T \mathbf{A}$ need to be computed
- Implementations like `np.linalg.eig` typically use variations of the *QR algorithm* which I will now briefly outline

Basic idea: iteratively obtain a sequence of matrices all *similar* to \mathbf{F} (so each matrix has the same eigenvalues). The matrices will become almost upper triangular, and the diagonal entries approach the eigenvalues of \mathbf{F} .

Simplest version of the algorithm:

- Factor \mathbf{F} in the form, $\mathbf{F} = \mathbf{Q}^{(1)} \mathbf{R}^{(1)}$ where $\mathbf{Q}^{(1)}$ is orthogonal and $\mathbf{R}^{(1)}$ is upper triangular
- Then set $\mathbf{F}^{(1)} = \mathbf{R}^{(1)} \mathbf{Q}^{(1)}$
- For iteration i , factor $\mathbf{F}^{(i)}$ so $\mathbf{F}^{(i)} = \mathbf{Q}^{(i+1)} \mathbf{R}^{(i+1)}$, and set $\mathbf{F}^{(i+1)} = \mathbf{R}^{(i+1)} \mathbf{Q}^{(i+1)}$
 $\mathbf{F}^{(i)}$ will converge towards an upper triangular form with the eigenvalues of \mathbf{F} on its diagonal
- More sophisticated versions of this algorithm are used in practice, but the cost is still expensive: $O(n^3)$ if \mathbf{F} is $n \times n$

-
- The eigenvalues and eigenvectors of $\mathbf{F} = \mathbf{A}^T \mathbf{A}$ can be found from the *singular value decomposition* (SVD) of \mathbf{A} (which is $m \times n$): $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^T$
 - Any real $m \times n$ matrix, \mathbf{A} , can be decomposed as $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^T$.
 - Here \mathbf{U} is an $m \times m$ orthogonal matrix whose columns are the eigenvectors of $\mathbf{A}\mathbf{A}^T$
 - \mathbf{W} is an $n \times n$ orthogonal matrix whose columns are the eigenvectors of $\mathbf{A}^T\mathbf{A}$
 - Σ is a non-negative diagonal $m \times n$ rectangular matrix. The non-zero values on the diagonal are the square-roots of the non-zero eigenvalues of $\mathbf{A}^T\mathbf{A}$ which are also the square-roots of the non-zero eigenvalues of $\mathbf{A}\mathbf{A}^T$ ($\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ share the same set of nonzero eigenvalues, $\sigma_1^2 \geq \sigma_2^2 \geq \dots \geq \sigma_R^2 > 0$ with $R \leq \min(m, n)$).
 - The σ_i are the *singular values* of \mathbf{A} and are usually arranged in non-increasing order.
 - If \mathbf{A} is complex, its SVD is instead, $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^H$ and the transposes in the discussion above become conjugate transposes.

-
- To connect \mathbf{U} with, \mathbf{AA}^T we start with $\mathbf{AA}^T = \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T\mathbf{W}\boldsymbol{\Sigma}^T\mathbf{U}^T = \mathbf{U}\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T\mathbf{U}^T$ and $\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T$ is an $m \times m$ diagonal matrix containing the eigenvalues of \mathbf{AA}^T . We can then recognize \mathbf{U} as the eigenvector matrix in the orthogonal diagonalization of \mathbf{AA}^T
 - Let \mathbf{u}_i be the i^{th} eigenvector of \mathbf{AA}^T and let \mathbf{w}_i be the i^{th} eigenvector of $\mathbf{A}^T\mathbf{A}$. Then, $\mathbf{Aw}_i = \sigma_i \mathbf{u}_i$. This means that $\sigma_1 \mathbf{u}_1$ is the “final” vector in the maximum growth problem while \mathbf{w}_1 is the initial vector, and σ_1 is the growth
 - The numerical method used to compute the SVD is similar to the QR method, and the asymptotic running time is also similar: $O(m^2n) + O(n^3)$

-
- The running times for the two methods are similar. What do we see in practice?

```
In [9]: A = np.random.randn(800,800)
```

```
In [10]: A.shape
```

```
Out[10]: (800, 800)
```

```
In [11]: %timeit F=np.dot(A.T,A)
```

```
14.1 ms ± 963 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [12]: timeit l,v = np.linalg.eig(F)
```

```
543 ms ± 13.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [13]: timeit u,s,wt = np.linalg.svd(F)
```

```
300 ms ± 1.62 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [22]: l[0]
```

```
Out[22]: 3195.472688027676
```

The SVD calculation is about 40% faster here

Notes:

- `scipy.sparse.linalg` should be used if the matrices are sparse
- There is a more efficient version of QR for symmetric matrices which can be faster than the SVD (though it doesn't compute U)

```
In [23]: s[0]**2
```

```
Out[23]: 3195.4726880276557
```

General comments

- This discussion has assumed that the elements of \mathbf{A} are real, but the same ideas generally apply if it is complex
 - There, instead of transposes, we use conjugate transposes and take advantage of the properties of Hermitian matrices ($\mathbf{F} = \mathbf{A}^H \mathbf{A}$, $\mathbf{F}^H = \mathbf{F}$) which also have real eigenvalues and can be orthogonally diagonalized
- Up to now we have primarily thought about matrices as linear operators appearing in dynamical process problems
- However, in many cases, it is very natural to store numerical data in tables which we can view as matrices
- This then opens the way for using linear algebra and matrix computations to analyze the data

Lecture 13

Matrix computations: applications and the SVD
Low-rank factorization

Notation

- This is a reference slide on the notation we are using
- Scalars: i, j, k, n, m
- Vectors:
 - matrix-vector form: \mathbf{u}, \mathbf{v}
 - index notation: u_i, v_i
 - Occasionally, we will need to number vectors. Then \mathbf{v}_i would be the *i*th vector in contrast with v_i , the *i*th element in a vector
- Matrices:
 - matrix-vector form: $\mathbf{A}, \mathbf{B}, \mathbf{M}$
 - index notation: A_{ij}, B_{ij}, M_{ij}
- So we will say $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m, \mathbf{A} \in \mathbb{R}^{m \times n}$ or:
$$\sum_{j=1}^n A_{ij}x_j = b_i, \quad i = 1, 2, \dots, m$$
- For projects, use whatever sensible convention you are comfortable with

Matrix structure

This week we will continue (and wrap up) our look at “matrix computations”

Motivating applications include:

- linear(ized) dynamical systems:

$$\frac{d\mathbf{x}}{dt} = \mathbf{M}\mathbf{x}, \quad \mathbf{x}(t=0) = \mathbf{x}_0$$

- Data stored in matrices: $\mathbf{A} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \end{bmatrix}$
- If \mathbf{M} or \mathbf{A} is symmetric, its eigenvalues/eigenvectors tell us “everything”

-
- In many applications, \mathbf{A} , is not symmetric.
 - Then, if \mathbf{A} is square, the eigenvalues and eigenvectors of \mathbf{A} still typically contain important information (e.g. if linear dynamical systems exhibit exponential growth)
 - But there may also be important information within $\mathbf{A}^T\mathbf{A}$ (which is always symmetric)
 - The SVD gives us information about the eigenvalues and eigenvectors of both $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$
 - Any real $m \times n$ matrix, \mathbf{A} , can be decomposed as $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^T$
 - With complex \mathbf{A} , the transpose is replaced with the conjugate transpose: $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^H$
 - If \mathbf{A} is $m \times n$, then \mathbf{U} is $m \times m$, \mathbf{W} is $n \times n$, and Σ is a non-negative $m \times n$ diagonal matrix
 - $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ have the same non-zero eigenvalues
 - The *singular values* of \mathbf{A} are the square-roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$

Maximum growth

- The 1st columns of \mathbf{W} and \mathbf{U} and the largest singular value in Σ give us information about maximum growth
- In atmospheric science, it has been hypothesized that cyclogenesis (the formation of storms) may be connected to solutions to the maximum growth problem though there doesn't seem to be a clear consensus on this!
- A prominent weather forecasting center (ECMWF) also considers a similar problem
 - Weather forecasts are based on numerical simulations of systems of nonlinear PDEs
 - Initial conditions are partially provided by observations (weather stations, satellites, etc...). However the data is insufficient, and simulations starting from these conditions often do not produce realistic weather
 - Instead ensembles of simulations are carried out where perturbations are added to the observations. These perturbations are assembled using the leading eigenvectors of $\mathbf{A}^T \mathbf{A}$ for their problem. This tends to stimulate growth and, eventually, realistic dynamics.

- Up to now we have primarily thought about matrices as linear operators appearing in dynamical process problems
- However, in many cases, it is very natural to store numerical data in tables which we can view as matrices
- This then opens the way for using computations to analyze the data
- What is the typical workflow for these kinds of data analysis problems?
- Getting started: Data collection/acquisition
 - Lab measurements
 - Online databases (e.g. Kaggle)
 - Simulation results

-
- We typically work with datasets containing multiple variables
 - For example, temperature, pressure, humidity, wind speed and direction
 - Or, IP address, location, duration of visit, number of visits, pages visited
 - But often don't know in advance if all of these variables are "important" – how many variables do we actually need to capture the essential trends or dynamics?
 - More generally, can we extract coherent structure that is "hiding" in the data?
 - Images are also naturally represented as matrices

- Consider the jpeg image shown →
- This corresponds to three 360×326 numpy arrays containing integers between 0 and 255
- The three arrays correspond to red, green, and blue, while smaller elements are “darker” and larger elements are “lighter”
- Sometimes, an image matrix is “unrolled” into a single column vector. For our example, a black and white image corresponding to a 360×326 matrix would be converted to a 117360-element vector.
 - Then a sequence of images would be a set of unrolled image vectors collected in a matrix
 - Once we have a “clean” dataset, we can apply ideas and tools that are similar to those we encountered for the maximum growth problem



Low-rank approximation

- We will now focus on *low-rank approximations* of datasets represented as matrices.
- We will begin with some of the theoretical background, and then move on to computations and applications
- The rank of an arbitrary matrix \mathbf{A} is the number of linearly independent columns in \mathbf{A} . Or more generally:

$$\text{Rank}(\mathbf{A}) = \text{dimension of column space of } \mathbf{A} = \text{dimension of row space of } \mathbf{A}$$

- If the rank of a matrix is very large, it may be difficult to interpret why the values of its elements are what they are. If we find a matrix with lower rank which is “close” to the original matrix, this will give us a more compact representation of the data that may also be easier to interpret
- How do we compute (or estimate) the rank of a matrix? With the SVD!

-
- If an $m \times n$ matrix \mathbf{A} has r non-zero singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$ with $\sigma_{r+1} = \sigma_{r+2} = \dots = \sigma_n = 0$ if $r < n$, then $\text{rank}(\mathbf{A}) = r$
 - Or in simpler terms, compute the SVD of \mathbf{A} , $\mathbf{A} = \mathbf{U}\Sigma\mathbf{W}^T$, and the rank of \mathbf{A} is the number of non-zero elements in Σ
 - I won't provide a proof, but let's try to build some intuition of why this is the case
 - We will need to think about matrix-matrix multiplication. Usually I think of $\mathbf{A} = \mathbf{B}\mathbf{C}$ as a collection of inner products:
$$A_{ij} = \sum_{k=1}^n B_{ik} C_{kj} = \boldsymbol{\beta}_i^T \mathbf{c}_j$$
where $\boldsymbol{\beta}_i^T$ is the i th row of \mathbf{B} , and \mathbf{c}_j is the j th column of \mathbf{C} .
 - However, it can be useful to view this as a sum of *outer products*: $\mathbf{A} = \mathbf{b}_1\boldsymbol{\gamma}_1^T + \mathbf{b}_2\boldsymbol{\gamma}_2^T + \dots + \mathbf{b}_n\boldsymbol{\gamma}_n^T$
 - Here, \mathbf{b}_i is the i th column in \mathbf{B} , $\boldsymbol{\gamma}_j^T$ is the j th row in \mathbf{C} , and each term in the sum is a rank-1 $n \times n$ matrix

- So if \mathbf{B} is $m \times l$:

$$\mathbf{B} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 & \cdots \mathbf{b}_l \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \cdots & \boldsymbol{\beta}_1^T & \cdots \\ \cdots & \boldsymbol{\beta}_2^T & \cdots \\ \cdots & \boldsymbol{\beta}_3^T & \cdots \\ \vdots & & \vdots \\ \cdots & \boldsymbol{\beta}_m^T & \cdots \end{bmatrix}$$

- Why is $\mathbf{b}_i \boldsymbol{\gamma}_i^T$ a rank-1 matrix? Each column of the matrix is \mathbf{b}_i scaled by an element of $\boldsymbol{\gamma}_i$ which tells us no two columns of the matrix are linearly independent:

For example if $\boldsymbol{\gamma}_1^T = [g_1, g_2]$, then $\mathbf{b}_1 \boldsymbol{\gamma}_1^T = \begin{bmatrix} \vdots & \vdots \\ g_1 \mathbf{b}_1 & g_2 \mathbf{b}_1 \\ \vdots & \vdots \end{bmatrix}$

- Using similar reasoning, the sum of two rank-1 matrices will be at most rank-2
- Let's now return to the SVD, $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T$

-
- The SVD of a matrix can similarly be viewed as a weighted sum of rank-1 matrices:

$$\mathbf{A} = \sigma_1 \mathbf{u}_1 \mathbf{w}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{w}_2^T + \cdots + \sigma_r \mathbf{u}_r \mathbf{w}_r^T$$

where $\sigma_i = \Sigma_{ii}$ is the *i*th singular value

- We can see that the number of non-zero singular values provides a truncation of this sum. Recognizing that the columns of \mathbf{U} are linearly independent, we can also conclude that the rank will be at most r and simply state that it will indeed be r
- Recall that the singular values are ordered, $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$. Then assume that for some p greater than r , that $\sigma_p \ll \sigma_1$.
- We could then guess that a reasonable approximation for \mathbf{A} would be:

$$\mathbf{A} \approx \mathbf{A}_p = \sigma_1 \mathbf{u}_1 \mathbf{w}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{w}_2^T + \cdots + \sigma_p \mathbf{u}_p \mathbf{w}_p^T$$

which is a rank- p approximation.

-
- Let's look at an example. We'll take a matrix corresponding to an image, compute its SVD and construct a rank- p approximation

```
In [55]: U,S,WT = np.linalg.svd(A)
```

```
In [56]: A.shape
```

```
Out[56]: (375, 500)
```

```
In [57]: U.shape
```

```
Out[57]: (375, 375)
```

```
In [58]: S.shape
```

```
Out[58]: (375,)
```

```
In [59]: WT.shape
```

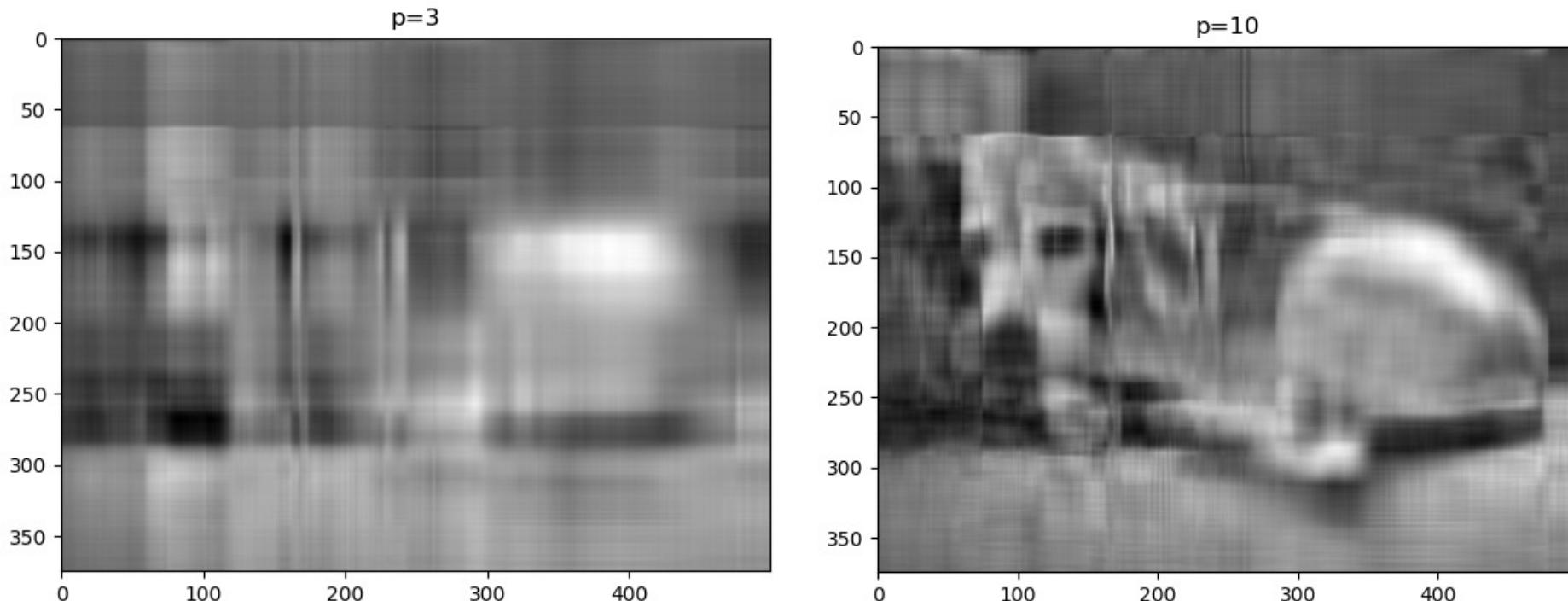
```
Out[59]: (500, 500)
```

```
In [60]: Ap = np.zeros_like(A,dtype=float)
```

```
In [61]: for i in range(p):
```

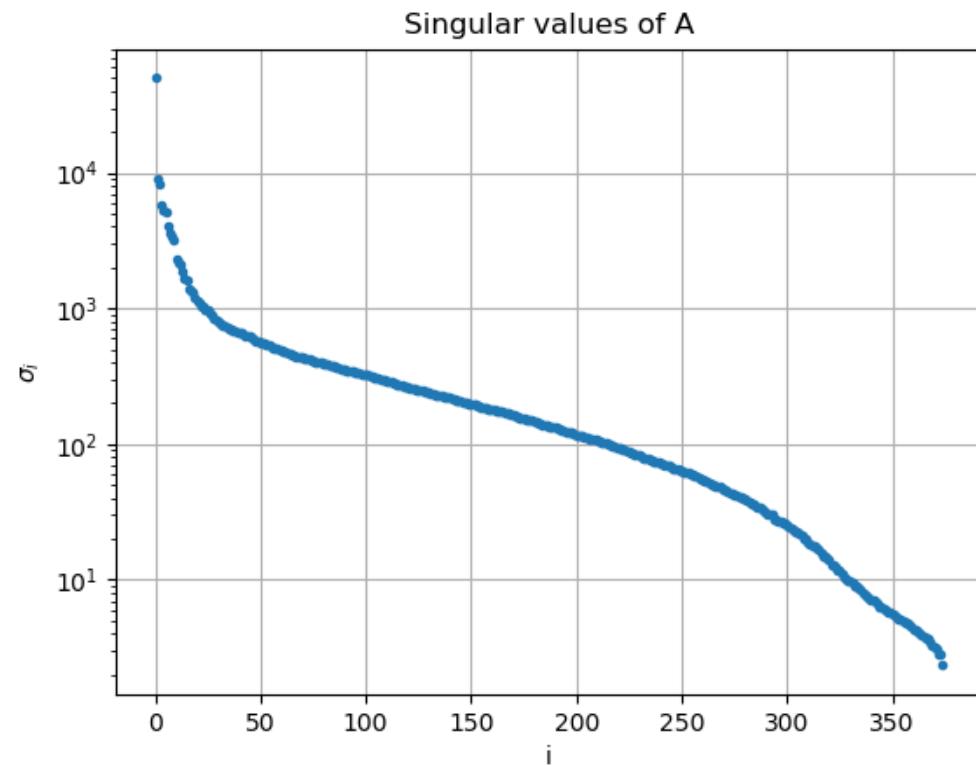
```
...:     Ap = Ap + S[i]*np.multiply.outer(U[:,i],WT[i,:])
```

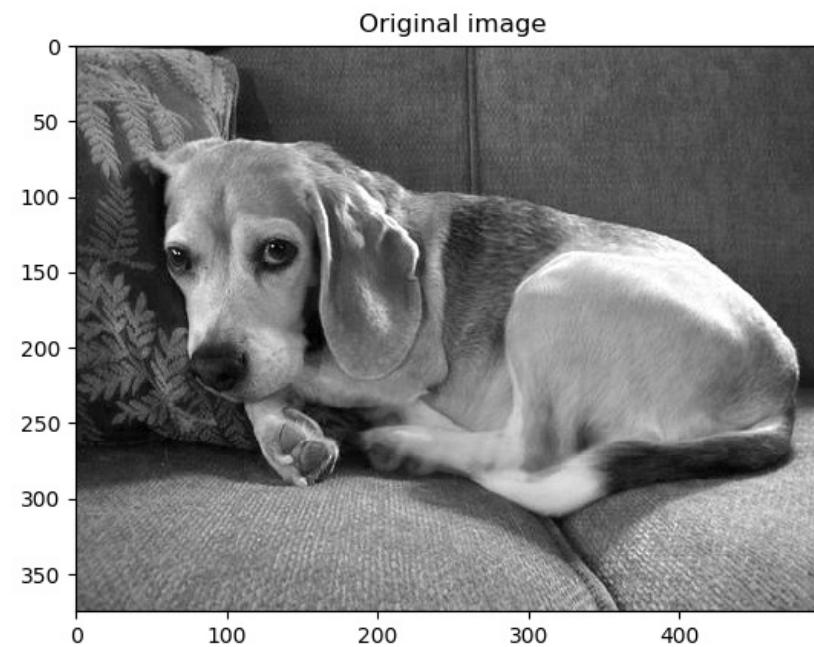
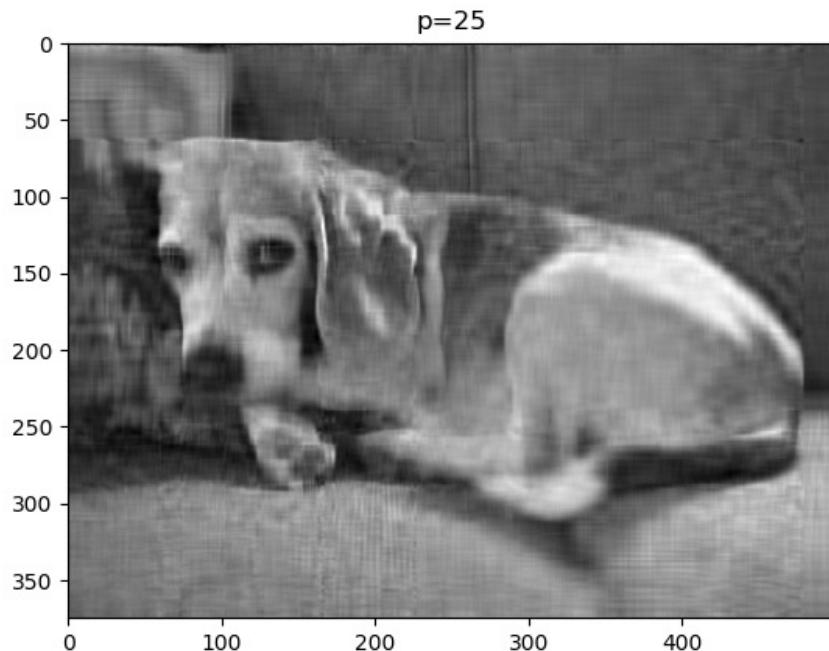
- We'll try a few values of p and display the image corresponding to \mathbf{A}_p



- With $p = 3$, too much “information” has been discarded, however with $p = 10$, we can make out what the image is
- Instead of randomly picking p , we can look to the singular values for guidance

-
- Each σ_i is a weight for a rank-1 matrix, and we should be able to safely discard matrices corresponding to σ_i sufficiently small relative to the largest singular values
 - From this plot, $p \approx 25$ seems like a sensible next choice...





- We have a recognizable reproduction of the original 375×500 matrix
- But this reproduction is constructed with just 25 singular values, 25 columns of \mathbf{U} , and 25 columns of \mathbf{W} !

-
- We have used a heuristic argument to choose p based on the singular values. Can we take a more precise approach? First we need to decide how to assess the quality of an approximation

- We'll work with the *Frobenius norm* of a matrix: $|\mathbf{A}|_F = \sum_{i=1}^m \sum_{j=1}^n (A_{ij})^2$
- Or, in matrix-vector form: $|\mathbf{A}|_F = \text{trace}(\mathbf{A}^T \mathbf{A})$, and since the trace of a matrix is equal to the sum of its eigenvalues:

$$|\mathbf{A}|_F = \sum_{i=1}^m \sum_{j=1}^n (A_{ij})^2 = \text{trace}(\mathbf{A}^T \mathbf{A}) = \sum_{i=1}^r \sigma_i^2$$

- What is the Frobenius norm for a rank-1 approximation of \mathbf{A} ?
 $|\mathbf{A}_1|_F = \text{trace}\left((\mathbf{w}_1 \sigma_1 \mathbf{u}_1^T) (\sigma_1 \mathbf{u}_1 \mathbf{w}_1^T)\right)$, and the orthogonality of \mathbf{U} tells us $\mathbf{u}_1^T \mathbf{u}_1 = 1$, so $|\mathbf{A}_1|_F = \sigma_1^2 \text{trace}(\mathbf{w}_1 \mathbf{w}_1^T)$ and the orthogonality of \mathbf{W} then gives, $|\mathbf{A}_1|_F = \sigma_1^2$.
- With some more (tedious) work, this can be generalized to, $|\mathbf{A}_p|_F = \sum_{i=1}^p \sigma_i^2$. So, we see a connection between the singular values of \mathbf{A} and the Frobenius norm of \mathbf{A}_p , but this isn't quite what we need...

-
- What is actually needed is insight into $\|\mathbf{A} - \mathbf{A}_p\|_F$ and here we will rely on the *Eckart-Young theorem*:

- Problem definition: Find \mathbf{B} such that $\|\mathbf{A} - \mathbf{B}\|_F$ is minimized and $\text{rank}(\mathbf{B}) \leq p$
- Solution (Eckart-Young theorem): $\mathbf{B} = \sigma_1 \mathbf{u}_1 \mathbf{w}_1^T + \sigma_2 \mathbf{u}_2 \mathbf{w}_2^T + \cdots + \sigma_p \mathbf{u}_p \mathbf{w}_p^T$,

with: $\sigma_i, \mathbf{u}_i, \mathbf{w}_i$ taken from the SVD of \mathbf{A} as before,

The approximation error is: $\|\mathbf{A} - \mathbf{B}\|_F = \sum_{i=p+1}^r \sigma_i^2$

- This tells us that the SVD-based rank- p approximation is the *best* low-rank approximation!
- This result provides a foundation for our “image compression” example and SVD-based low-rank approximations more generally.

Lecture 14

Principal component analysis
Recommender systems

Data analysis

- Example: Say you are tracking a meteor heading towards Earth
 - How many variables do you need?
 - Three position coordinates? $x(t), y(t), z(t)$
 - Maybe three velocity components as well? $u(t), v(t), w(t)$



- But what if the meteor is just moving in a circular orbit?
- More generally, when working on complex problems, we often encounter datasets with a very large number of variables, and this makes analysis difficult
- The questions that follow are, how many variables and which combinations of variables are needed to reasonably represent what is “important” in the data

PCA

- Principal component analysis (PCA) is a well-known and widely-used method that attempts to address these questions
- It creates new variables which are linear combinations of the original variables which have certain “desirable properties”
- Let’s look at PCA in the context of our meteor problem
- We measure the meteor coordinates at n times and store them in a $3 \times n$ matrix:

- Data: $\mathbf{A} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \\ z_1 & z_2 & \cdots & z_n \end{bmatrix}$

- Let $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ with y and z defined analogously. Then, $\mathbf{A} = \begin{bmatrix} \cdots & \mathbf{x}^T & \cdots \\ \cdots & \mathbf{y}^T & \cdots \\ \cdots & \mathbf{z}^T & \cdots \end{bmatrix}$

- We also assume that the data has been processed so each row has zero mean, e.g. $\frac{1}{n} \sum_{i=1}^n x_i = 0$

- We now introduce our new variables as linear combinations of the old ones:

$$\tilde{\mathbf{x}} = \alpha_1 \mathbf{x} + \beta_1 \mathbf{y} + \gamma_1 \mathbf{z}$$

$$\tilde{\mathbf{y}} = \alpha_2 \mathbf{x} + \beta_2 \mathbf{y} + \gamma_2 \mathbf{z}$$

$$\tilde{\mathbf{z}} = \alpha_3 \mathbf{x} + \beta_3 \mathbf{y} + \gamma_3 \mathbf{z}$$

Or more compactly, $\tilde{\mathbf{A}} = \mathbf{T}\mathbf{A}$ where $\mathbf{T} = \begin{bmatrix} \alpha_1 & \beta_1 & \gamma_1 \\ \alpha_2 & \beta_2 & \gamma_2 \\ \alpha_3 & \beta_3 & \gamma_3 \end{bmatrix}$ and $\tilde{\mathbf{A}} = \begin{bmatrix} \dots & \tilde{\mathbf{x}}^T & \dots \\ \dots & \tilde{\mathbf{y}}^T & \dots \\ \dots & \tilde{\mathbf{z}}^T & \dots \end{bmatrix}$

- And we have to construct \mathbf{T} to satisfy the following desirable properties:
 - *Desirable property 1:* Each pair of new variables should have zero covariance
 - *Desirable property 2:* The first new variable should have the maximum variance possible for one transformed variable. And continuing on, the first k new variables should have the maximum possible total variance from k transformed variables.
- Let's think about why these properties are desirable...

-
- Why should variables have zero covariance?
 - Here, we are referring to the sample covariance of two (distinct) variables, e.g. $\frac{1}{n-1} \mathbf{x}^T \mathbf{z}$ or $\frac{1}{n-1} \tilde{\mathbf{y}}^T \tilde{\mathbf{z}}$.
 - If the magnitude of the covariance is large, this indicates that the two variables are correlated and are “carrying similar information”
 - So if the new variables have zero covariance, we are minimizing redundancy in this sense
 - Why do we want to maximize the total variance of the 1st k new variables?
 - The variance of \mathbf{x} is $\frac{1}{n-1} \mathbf{x}^T \mathbf{x}$. The total variance of the first 2 new variables is $\frac{1}{n-1} (\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} + \tilde{\mathbf{y}}^T \tilde{\mathbf{y}})$.
 - The idea here is that high variance indicates importance – think of speech vs background noise in an audio recording – and by ensuring that the first new variables have the most variance, we may be able to discard some “low-variance” variables leading to a smaller dataset.

-
- So how do we construct our transformation matrix, \mathbf{T} ?
 - Let's first think about how to construct $\tilde{\mathbf{x}}$ and we will then generalize the results we find
 - Let the first row of \mathbf{T} be $\mathbf{t}_1^T = [\alpha_1, \beta_1, \gamma_1]$. Then $\tilde{\mathbf{x}} = \mathbf{A}^T \mathbf{t}_1$ and we should construct \mathbf{t}_1 so that $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}}$ is maximized
 - However note that for any \mathbf{t}_1 , if we scale the vector with a constant greater than one, this will also increase $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}}$. So, we need a constraint for our transformation parameters:

Constraint for desirable property 2: Maximize the variance of the first k new variables, and constrain each row of \mathbf{T} to have unit length

- So, we need to find \mathbf{t}_1 so that $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}}$ is maximized with $\mathbf{t}_1^T \mathbf{t}_1 = 1$. This is very similar to our maximum growth problem!

-
- The variance (ignoring the $n - 1$ factor) is, $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \mathbf{t}_1^T \mathbf{A} \mathbf{A}^T \mathbf{t}_1$ and we orthogonally diagonalize $\mathbf{A} \mathbf{A}^T = \mathbf{V} \mathbf{S} \mathbf{V}^T$, so: $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \mathbf{t}_1^T \mathbf{V} \mathbf{S} \mathbf{V}^T \mathbf{t}_1$.
 - Then, defining $\mathbf{a} = \mathbf{V}^T \mathbf{t}_1$, we have, $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \mathbf{a}^T \mathbf{S} \mathbf{a} = \lambda_1 a_1^2 + \lambda_2 a_2^2 + \dots + \lambda_r a_r^2$ where λ_1 is the i th eigenvalue of $\mathbf{A} \mathbf{A}^T$, and $\lambda_1 \geq \lambda_2 \dots \geq \lambda_r > 0$
 - So, $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} \leq \lambda_1 \mathbf{a}^T \mathbf{a}$ and since $\mathbf{a}^T \mathbf{a} = \mathbf{t}_1^T \mathbf{t}_1$, we have a useful upper bound: $\frac{\tilde{\mathbf{x}}^T \tilde{\mathbf{x}}}{\mathbf{t}_1^T \mathbf{t}_1} \leq \lambda_1$ and we now need to find \mathbf{t}_1 with unit length such that $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \lambda_1$
 - If we choose \mathbf{t}_1 to be \mathbf{v}_1 , the leading eigenvector of $\mathbf{A} \mathbf{A}^T$: $\tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \mathbf{v}_1^T \mathbf{A} \mathbf{A}^T \mathbf{v}_1 = \mathbf{v}_1^T \lambda_1 \mathbf{v}_1 = \lambda_1$
 - And since $\mathbf{t}_1^T \mathbf{t}_1 = \mathbf{v}_1^T \mathbf{v}_1 = 1$, we have solved our problem!
 - Our first new variable is $\tilde{\mathbf{x}} = \mathbf{A}^T \mathbf{v}_1$ and its variance is the leading eigenvalue of $\mathbf{A} \mathbf{A}^T$ scaled with $n - 1$:
$$\frac{\lambda_1}{n-1}$$

-
- What about our second new variable? It is $\tilde{\mathbf{y}} = \mathbf{A}^T \mathbf{v}_2$. The total variance of these two variables is $\frac{1}{n-1}(\lambda_1 + \lambda_2) = \frac{1}{n-1}(\sigma_1^2 + \sigma_2^2)$ where σ_i is the i th singular value of \mathbf{A}
 - Desirable property 1 is satisfied with this choice:
$$\tilde{\mathbf{x}}^T \tilde{\mathbf{y}} = \mathbf{v}_1^T \mathbf{A} \mathbf{A}^T \mathbf{v}_2 = \mathbf{v}_1^T \mathbf{V} \mathbf{S} \mathbf{V}^T \mathbf{v}_2 = [1 \ 0 \ 0] * [0 \ \lambda_2 \ 0]^T = 0$$
 - And this generalizes as you would guess; the k th new variable is $\mathbf{A}^T \mathbf{v}_k$ and the total variance of the $1^{st} k$ new variables is, $\frac{1}{n-1} \sum_{i=1}^k \sigma_i^2$
 - Our initial goal was to find the full transformation matrix, \mathbf{T} , where $\tilde{\mathbf{A}} = \mathbf{T} \mathbf{A}$ and from the above statements, we conclude that $\mathbf{T} = \mathbf{V}^T$
 - Since \mathbf{V} is the orthogonal eigenvector matrix for $\mathbf{A} \mathbf{A}^T$, \mathbf{T} is just \mathbf{U}^T from the SVD of \mathbf{A}

Check your understanding:

- Why is $\frac{1}{n-1} \text{trace}(\mathbf{A}\mathbf{A}^T)$ the total variance of \mathbf{A} ?
- Why is $\text{trace}(\mathbf{A}\mathbf{A}^T) = \sum_{i=1}^r \sigma_i^2$?
- Why is the total variance of \mathbf{A} equal to the total variance of $\tilde{\mathbf{A}}$?

Notes:

- The singular values of \mathbf{A} are related to the variance of each new variable
- This also provides guidance on the relative “importance” of new variables
- If we discard low-variance new variables, we have a smaller system that is easier to work with. This is *dimensionality reduction*
- The rows of \mathbf{T} are the *principal components* of \mathbf{A}

-
- Python implementation (2 variables):
 1. Set up, solve eigenvalue problem:
 2. Compute transformed data:

```
In [45]: A.shape
```

```
Out[45]: (2, 51)
```

```
In [46]: A.mean(axis=1)
```

```
Out[46]: array([-6.26949473e-16,  
 5.22457894e-17])
```

```
In [47]: U,S,WT = svd(A)
```

```
In [48]: T = U.T
```

```
In [49]: Anew = np.matmul(T,A)
```

```
In [50]: np.dot(Anew,Anew.T)
```

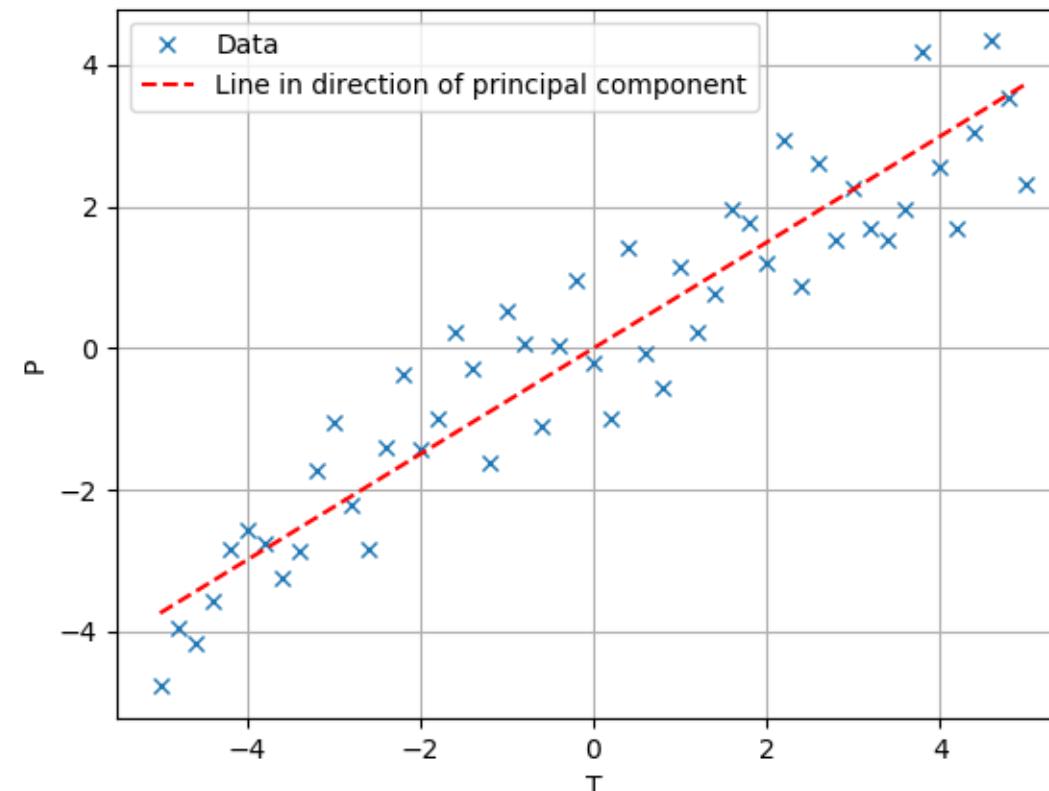
```
Out[50]:  
array([[ 8.33602821e+02, -1.18785463e-13],  
 [-1.18785463e-13,  1.50729071e+01]])
```

- We can see that the first new variable has a much larger variance associated with it than the second, and note that S contains the square-roots of the diagonal terms

3. Compare first eigenvector (corresponding to largest eigenvalue) to data:

In [51]: `plot(t,U[1,0]*t/U[0,0]-5, 'r--')`

- The 1st component points in the direction of maximum variance
- What is the direction of the 2nd component?



- Dimension reduction: discard 2nd row of \mathbf{A}_{new} and retain 1st row. Then to visualize the result, transform back to original variables
- How do we express reduced data in original variables? Recall: $\mathbf{A} = \mathbf{V}\tilde{\mathbf{A}}$; retaining only the first row of $\tilde{\mathbf{A}}$ we have $\mathbf{A}_{reduced} = \mathbf{v}_1 \tilde{\mathbf{x}}^T$

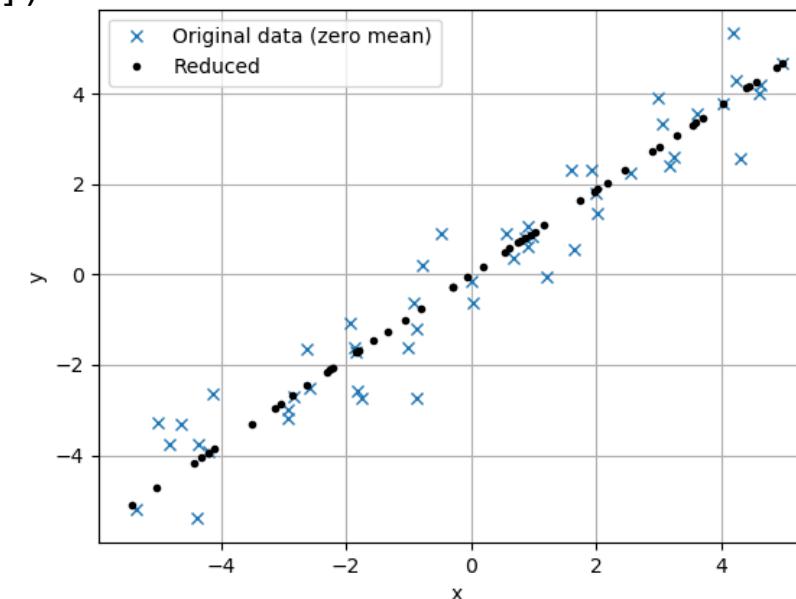
The *reduced* data, in our original variables, is then the outer product of the 1st eigenvector of \mathbf{V} with the 1st row of $\tilde{\mathbf{A}}$:

In [64]: `Areduced=np.multiply.outer(U[:,0],Anew[0,:,:])`

In [65]: `Areduced.shape`

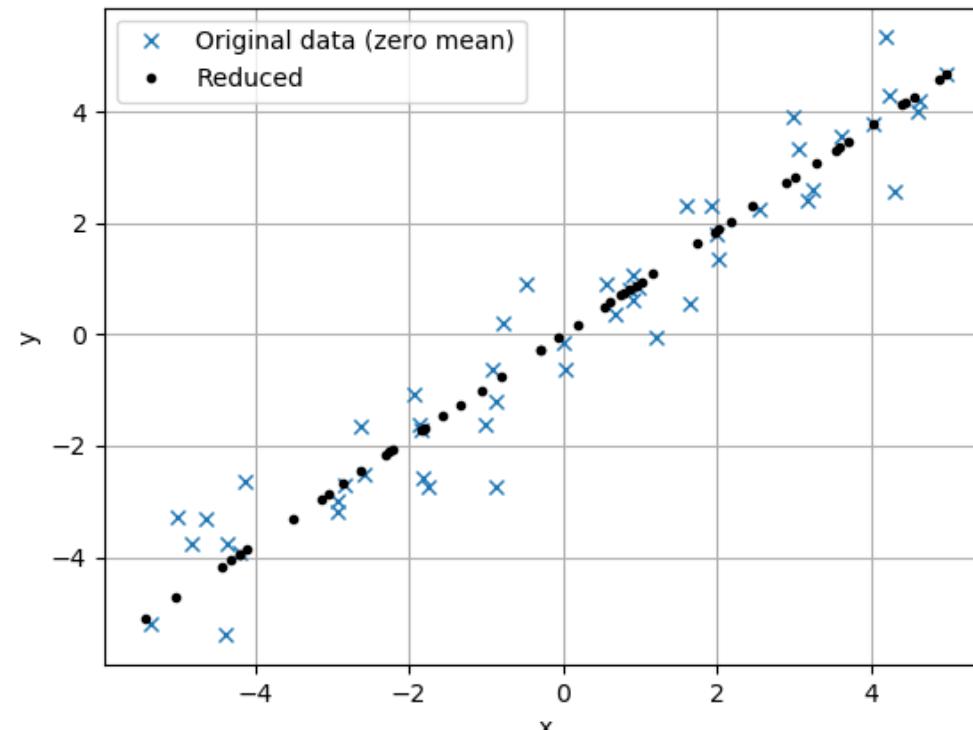
Out [65]: `(2, 51)`

In [66]: `plot(Areduced[0,:],Areduced[1,:],'k.')`



- Dimensionality reduction allows us to reduce redundancy and simplify models for underlying dynamics
- This example with just two variables is not a practical application
- Many problems have 10s, 100s, or even 1000s of variables and it is there where PCA and dimensionality reduction are most useful

- Given a circular trajectory in Cartesian coordinates, will PCA give us $\theta(t)$? No, this requires a *nonlinear* transformation.



Recommender systems

- With PCA, the goal was to extract information from a dataset
- Now, we want to think about filling in information *missing* from a dataset
- There are many different approaches; we will look at just one based on low-rank matrix factorization
- Is this actually useful? It is very important in computer graphics, computer vision, machine learning, and recommender systems
- We will focus on the latter example

-
- How do Netflix, Amazon, Facebook, etc... decide what to recommend to you?
 - They collect:
 - information about what you like (and dislike)
 - information about what everyone else likes
 - And they then attempt to predict what you will spend time and/or money on
 - How do they organize this data? And how do they estimate how much you will like something that is new to you?

A simple way to organize the data is to use a *user-item* or *ratings* matrix

	Suits	Sex Education	Friends	Stranger Things	Killing Eve
Don	5	?	1	?	?
Liz	0	4	5	?	?
Joe	2	?	3	?	5
Bernie	1	5	4	5	?

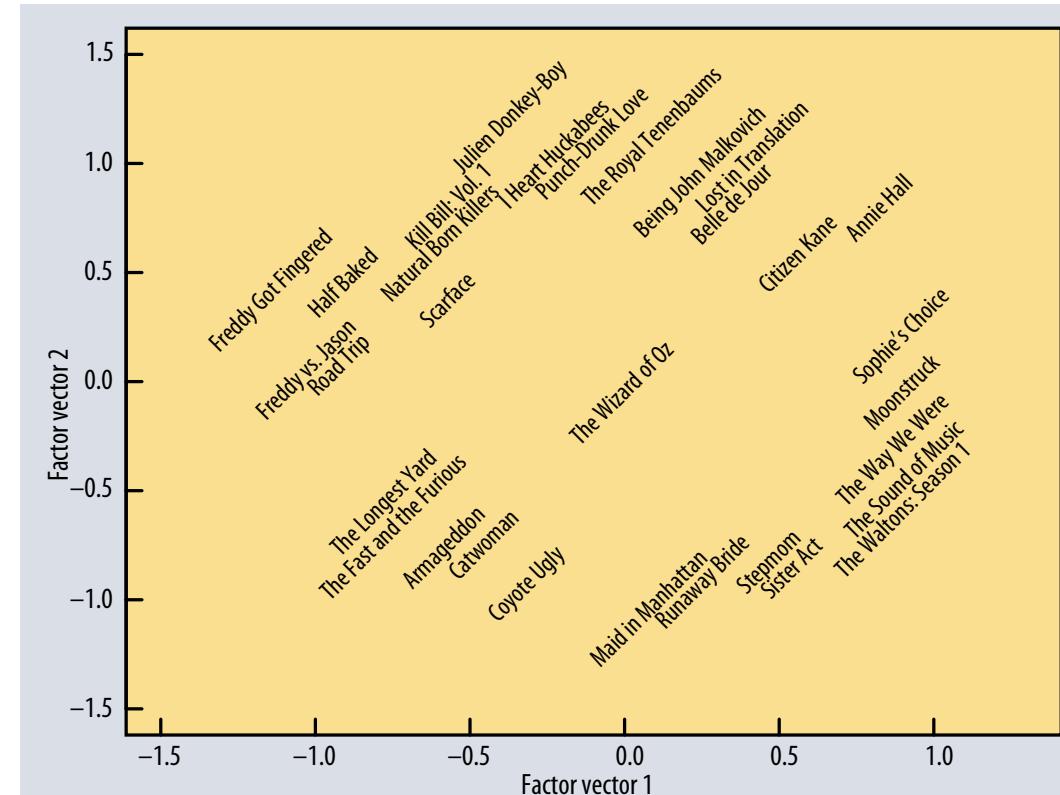
- How do we fill in the missing entries? Several different answers to this question have been developed
- An idea: Assume there is a set of high-level concepts or *features* (e.g. genre, critically-acclaimed, ...) that drive user preferences. Then develop 1) mappings between users and features and 2) mappings between items and features. If a user has not rated an item, the mappings can then be used to generate a rating.

-
- Let's first think about a simpler case – how do we fill in the missing entries in this matrix: $D = \begin{bmatrix} 1 & 2 \\ x & 6 \\ 2 & x \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 6 \\ 2 & 4 \end{bmatrix}$
 - We need to set criteria to assess how well we fill in the data
 - Basic idea: Fill in the data without introducing “new trends”
 - If we maximized variance like in PCA, we would be doing the opposite and inventing trends
 - We could think about minimizing variance, but a simpler closely-related idea is to minimize the *rank*
Reminder: $\text{Rank}(A) = \text{dimension of column space of } A = \text{dimension of row space of } A$
 - The rank-1 estimate for D in our example is shown above

Low-rank factorization

- We won't actually minimize the rank which is a pretty difficult problem, but we will aim for a “small” rank.
- We will then attempt to generate a complete ratings matrix where:
 - Changes to existing entries are “small”
 - The rank of the resulting matrix is also “small”
- The steps of the method are as follows:
 - *Choose* the rank that you want (p)
 - Then for an $m \times n$ incomplete ratings matrix, \mathbf{R} , construct \mathbf{A} , \mathbf{B} where:
 - $\tilde{\mathbf{R}} = \mathbf{AB}$ (this is our approximate full ratings matrix)
 - \mathbf{A} is $m \times p$ (user-feature matrix)
 - \mathbf{B} is $p \times n$ (feature-item matrix)
 - So A_{ij} should indicate the rating of user i for feature j while B_{ij} should indicate the correspondence between feature i and item j , and the rank of $\tilde{\mathbf{R}}$ will be at most p

- The “rank” in our low rank approximation should correspond to item “features”
- What are features? The figure shows movies clustered based on computed “factor vectors” (rows in a feature-item matrix)
- Clusters correspond to recognizable concepts, e.g. strong female lead, critically-acclaimed indie, violent, ...



-
- Let's now state our optimization problem. For a given p , we have to find the \mathbf{A} and \mathbf{B} which give the "best" $\tilde{\mathbf{R}} = \mathbf{AB}$
 - "Best" will be defined with a modified Frobenius norm: $|\mathbf{R} - \tilde{\mathbf{R}}|_F^* = \sum_{(i,j) \in K} (R_{ij} - \tilde{R}_{ij})^2$
 - Here, K is the set of elements in \mathbf{R} with known ratings, and the sum is over the (i,j) locations of the known ratings.
 - Our optimization problem is then to find \mathbf{A} , \mathbf{B} such that the cost, $c = |\mathbf{R} - \tilde{\mathbf{R}}|_F^*$ is minimized
 - So we need to find \mathbf{A} and \mathbf{B} where $\frac{\partial c}{\partial A_{kl}} = \frac{\partial c}{\partial B_{lq}} = 0$ with $k = 1, 2, \dots, m; l = 1, 2, \dots, p; q = 1, 2, \dots, n$
 - Our approach will be to:
 1. Guess \mathbf{A} and \mathbf{B}
 2. For each element of \mathbf{A} and \mathbf{B} update that element so that the derivative of the cost with respect to the element is zero
 3. Repeat step 2 until the change in \mathbf{A} and \mathbf{B} is acceptably small

-
- How do we update the elements of \mathbf{A} and \mathbf{B} ?
 - We will compute the derivative of the cost with respect to a matrix element, set the derivative to zero, and then determine the value of the matrix element which leads to the equation being satisfied
 - Our cost is: $c = \sum_{(i,j) \in K} (R_{ij} - \tilde{R}_{ij})^2$, with $\tilde{R}_{ij} = \sum_{s=1}^p A_{is} B_{sj}$. Differentiating c with respect to A_{kl} :

$$\frac{\partial c}{\partial A_{kl}} = -2 \sum_{(i,j) \in K} (R_{ij} - \tilde{R}_{ij}) \frac{\partial \tilde{R}_{ij}}{\partial A_{kl}}$$
, and $\frac{\partial \tilde{R}_{ij}}{\partial A_{kl}} = \sum_{s=1}^p \frac{\partial A_{is}}{\partial A_{kl}} B_{sj}$

We know that $\frac{\partial A_{is}}{\partial A_{kl}} = \delta_{ik} \delta_{sl}$ where δ_{ik} is the Kronecker delta function and then:

$$\frac{\partial \tilde{R}_{ij}}{\partial A_{kl}} = \sum_{s=1}^p \delta_{ik} \delta_{sl} B_{sj} = \delta_{ik} B_{lj} .$$

Substituting this into the equation for $\frac{\partial c}{\partial A_{kl}}$ gives:

$$\frac{\partial c}{\partial A_{kl}} = -2 \sum_{(i,j) \in K} (R_{ij} - \tilde{R}_{ij}) \delta_{ik} B_{lj}$$

We can now adjust the sum so $i = k$ and only j will vary : $\frac{\partial c}{\partial A_{kl}} = -2 \sum_{j,(k,j) \in K} (R_{kj} - \tilde{R}_{kj}) B_{lj}$

- Then we use, $\tilde{R}_{kj} = A_{kl} B_{lj} + \sum_{s \neq l} A_{ks} B_{sj}$ and rearrange our equations:

$$\frac{\partial c}{\partial A_{kl}} = 0 \rightarrow A_{kl} \sum_{j,(k,j) \in K} B_{lj}^2 = \sum_{j,(k,j) \in K} (R_{kj} - \sum_{s \neq l} A_{ks} B_{sj}) B_{lj}$$

We can use this to update A_{kl} . Going through a similar procedure for $\frac{\partial c}{\partial B_{kl}}$ gives us an update equation for B_{kl} :

$$\frac{\partial c}{\partial B_{kl}} = 0 \rightarrow B_{kl} \sum_{i,(i,l) \in K} A_{ik}^2 = \sum_{i,(i,l) \in K} (R_{il} - \sum_{s \neq k} A_{is} B_{sl}) A_{ik}$$

-
- We now have the expressions we need for our iterative optimization method
 - First, guess **A** and **B** (setting all values to a constant is ok)
 - A “step” will be a process by which we update all terms in these matrices
 - Within one step, we will iterate through **A** and **B** element by element, updating the “current” element using the last two equations on the previous slide
 - Terminate iterations when the change in **A** and **B** from one step to the next is sufficiently small

- Applying this method to our small example with $p = 3$:

	Suits	Sex Education	Friends	Stranger Things	Killing Eve
Don	5	2	1	0	0
Liz	0	4	5	5	5
Joe	2	3	3	3	5
Bernie	1	5	4	5	5

- Here numbers have been rounded to the nearest integer between 0 and 5 for missing entries, and the other entries have been assigned their original values. The known ratings for each user were also scaled to have zero mean prior to the calculation.
- Does this make sense? It was clear from the original matrix that Don's taste is very different from the others, and the completed entries continue this trend.

-
- For this small example, our method can generate ratings that are far outside the desired range, and the cost function can be modified to improve the predictions (e.g. by penalizing matrix entries with large magnitudes).
 - More generally, there are a broad range of matrix factorization approaches
 - Different optimization methods can then be applied for each of these approaches
 - Stochastic gradient descent is popular for very large ratings matrices
 - There are other approaches to recommender systems as well that do not rely on matrix factorization (e.g. collaborative filtering)
 - Also keep in mind that the very best methods are often combinations of different kinds of approaches

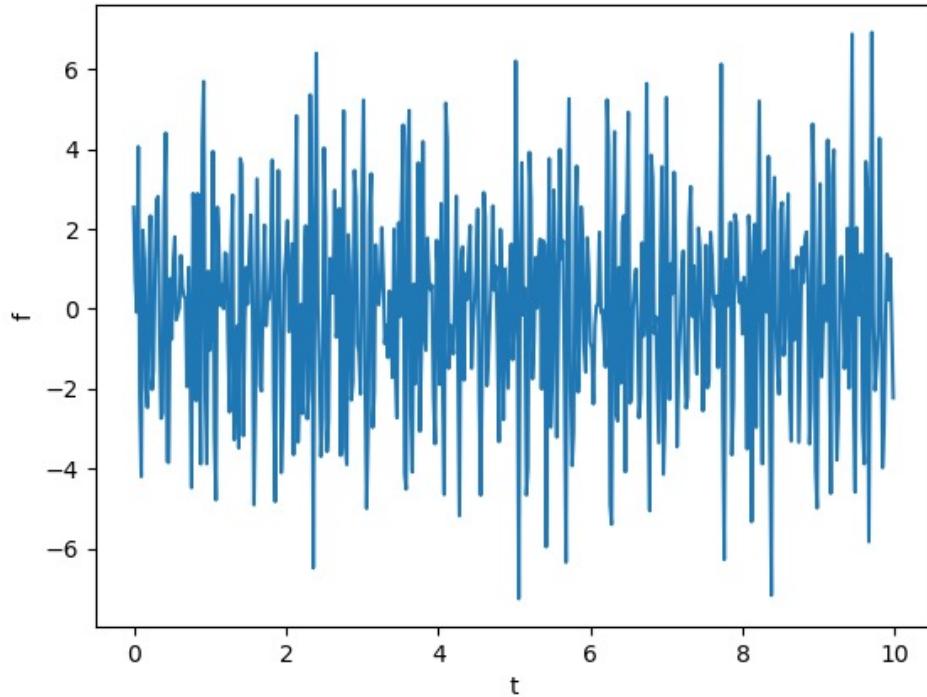
Lecture 15

Data analysis and Fourier series
Discrete Fourier transforms
Windowing and Welch's method

Data analysis

- Over the last three lectures, we have looked at 4 applications which I have collectively called “matrix computations”: maximum growth, low-rank approximation, PCA, and (low-rank) matrix completion
- Each application requires solution of an optimization problem and there were a few ideas/tools that appeared multiple times
- For the first three problems, we took advantage of the properties of symmetric matrices (specifically orthogonal diagonalization), and we also saw that the SVD could be helpful
 - Symmetric matrices often arise naturally (adjacency matrices for undirected graphs), or can be associated with a certain problem or method ($\mathbf{A}^T\mathbf{A}$, \mathbf{AA}^T , $\mathbf{A} + \mathbf{A}^T$, ...)
- Other tools that were relevant more than once include outer products producing low-rank matrices, and the Frobenius norm
- You should think of these applications as additions to your scientific computing “toolbox”

-
- We will now move to something simpler (sort of). We'll look at data arranged in 1D arrays rather than as matrices
 - A typical example, a time series: $f(t_i)$, $t_i = i \Delta t$, $i = 0, 1, \dots, n_t - 1$
 - We could also have data in space (along a line): $f(x_i)$, $x_i = i \Delta x$, $i = 0, 1, \dots, n_x - 1$
 - How do we extract trends or, more generally, “information”, from such data?



- What should we do with a signal that looks like this?
- The aim is to build an understanding of important features of the data, and first steps could be to compute the mean and rms (sample standard deviation)
- But what next?

-
- But what next?
 - We can think about the *frequency spectrum*
 - Underlying idea: decompose signal into superposition of waves with different frequencies and amplitudes
 - And check which frequencies hold the most “energy”
 - It will be useful to first review some useful properties of Fourier series...

Fourier series

- The Fourier series of a general function:

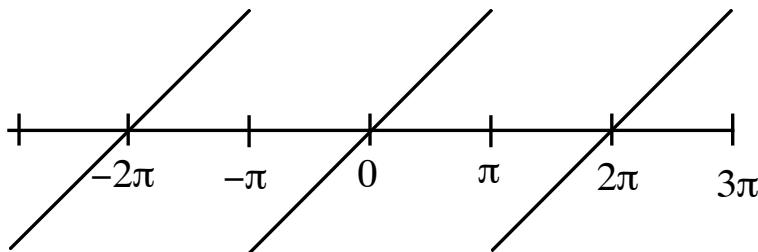
$$f(x) = \sum_{k=-\infty}^{\infty} c_k \exp(ikx)$$

$$c_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \exp(-ikx) dx$$

- Here, the function is expanded as a weighted sum of sines and cosines with period 2π . This can be modified to basis functions with period, l , through a simple change of variables, $y = xl/(2\pi)$
- In practice, Fourier series are only used for functions which are periodic with period equal to the period of the basis functions: $f(x + l) = f(x)$
- For such periodic functions which are also infinitely differentiable and continuous in all of their derivatives on the real line, we have *exponential convergence*: $c_k \sim \exp(-\mu|k|)$, μ is a positive constant (this is a very nice property!)

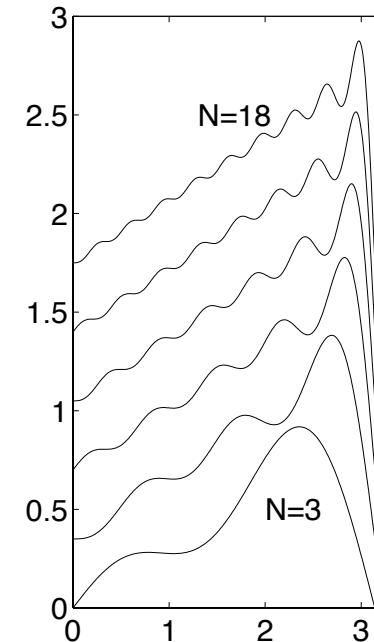
- We will largely ignore the formal convergence theory and try to build intuition
- The rate-of-convergence of the series depends on the smoothness of the function
 - Let's look at a few examples with non-smooth behavior...

- **Example 1: sawtooth function:**

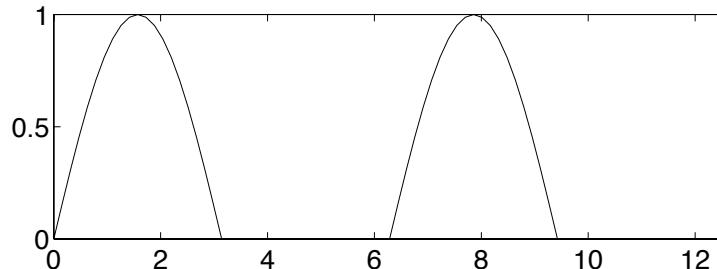


- This function is 2π -periodic, however it has a series of discontinuities
- Convergence is slow: $|c_k| \sim 1/|k|$
- The figure on the right shows an approximation to the function constructed by retaining the 1^{st} $2N + 1$ terms in the series expansion :

$$\tilde{f}(x) = \sum_{k=-N}^N c_k \exp(ikx)$$

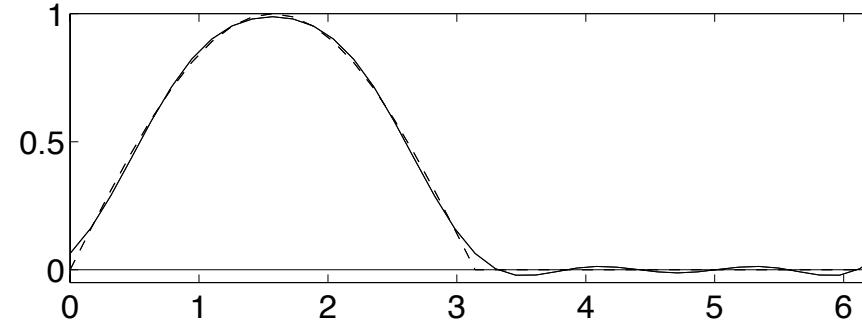


- **Example 2: half-wave rectifier:**



$$f(t) \equiv \begin{cases} \sin(t), & 0 < t < \pi \\ 0, & \pi < t < 2\pi \end{cases}$$

- This function is also 2π -periodic, and it is continuous everywhere. However, its derivative has a series of discontinuities
- Convergence is again slow, $|c_k| \sim \frac{1}{k^2}$
- This figure shows an approximation calculated by retaining the 1st 4 nonzero terms in the Fourier series sum (the solid curve is the approximation):



-
- The examples are pointing us towards a general result:
 - If:
 1. $f(\pi) = f(-\pi)$, $f^{(1)}(\pi) = f^{(1)}(-\pi), \dots, f^{(m-2)}(\pi) = f^{(m-2)}(-\pi)$
 2. $f^{(m)}(x)$ is integrable on $[-\pi, \pi]$

Then:

The coefficients of the Fourier series, $f(x) = \sum_{k=-\infty}^{\infty} c_k \exp(ikx)$, have the upper bound: $|c_k| \leq F/k^m$ for some sufficiently large F independent of k

Notes:

- Here $f^{(m)}(x)$ is the m^{th} derivative of f
- We see clearly that the smoothness of a function and the number of its derivatives that are periodic are closely related to how well it is represented as a Fourier series
- The proof is based on repeated application of integration-by-parts
- We still have to figure out how these ideas can be applied to discrete data

Discrete Fourier transform

- We'll now work with data on a n -point, equispaced grid: $y(t_j)$, $t_j = j \Delta t$, $j = 0, 1, \dots, n - 1$
- And we expand the data as a weighted sum of complex exponentials which are periodic over a timespan of length $\tau = n\Delta t$:

$$y(t_j) = y_j = \sum_{k=-n/2}^{n/2-1} c_k \exp(i2\pi k t_j / \tau) = \sum_{k=-n/2}^{n/2-1} c_k \exp(i2\pi j k / n)$$

- This is the *inverse discrete Fourier transform* of our data. Given all c_k we can recover the original data.
- The original data is decomposed into a sum of “waves” with frequency k/τ and amplitude, $|c_k|$. This amplitude indicates how important the corresponding frequency is.
- Say the data is sampled from a continuous function, $g(t)$. Then the ideas underlying convergence of Fourier series apply here provided Δt is sufficiently small. I.e. the smoothness and periodicity of $g(t)$ control the decay of $|c_k|$ as $|k|$ increases

-
- We can also derive a simple equation for each Fourier coefficient from the definition of the inverse transform. The *discrete Fourier transform* of y is:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j \exp(-i2\pi k t_j / \tau) = \frac{1}{n} \sum_{j=0}^{n-1} y_j \exp(-i2\pi jk / n)$$

- The computation of one c_k requires $\mathcal{O}(n)$ operations, so it would appear that the running time for computing all n coefficients should be $O(n^2)$
- However, this problem can be solved using a divide-and-conquer approach! The *fast-Fourier-transform* (FFT) algorithm computes all n coefficients in $\mathcal{O}(n \log_2 n)$ time.
- FFT functions are available in `np.fft`, and `scipy.fft` (browse through the online reference pages)
- We'll work with the numpy version in this lecture which can be viewed as a subset of the `scipy` package

Python FFT

- `np.fft.fft` computes the discrete Fourier transform of an input array, but it doesn't give us exactly what we want. It outputs a numpy array containing:

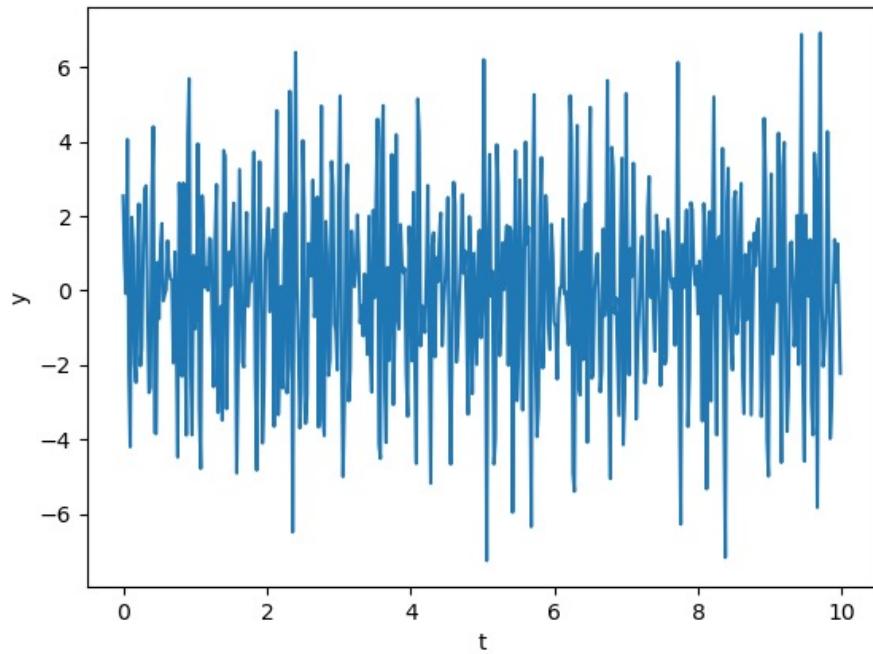
$$n * [c_0, c_1, \dots, c_{n/2-1}, c_{-n/2}, c_{-n/2+1}, \dots, c_{-1}]$$

- It is usually convenient to have the coefficients ordered by their subscripts, and `np.fft.fftshift(np.fft.fft(f))` will produce an array containing,

$$n * [c_{-n/2}, c_{-n/2+1}, \dots, c_{n/2-1}]$$

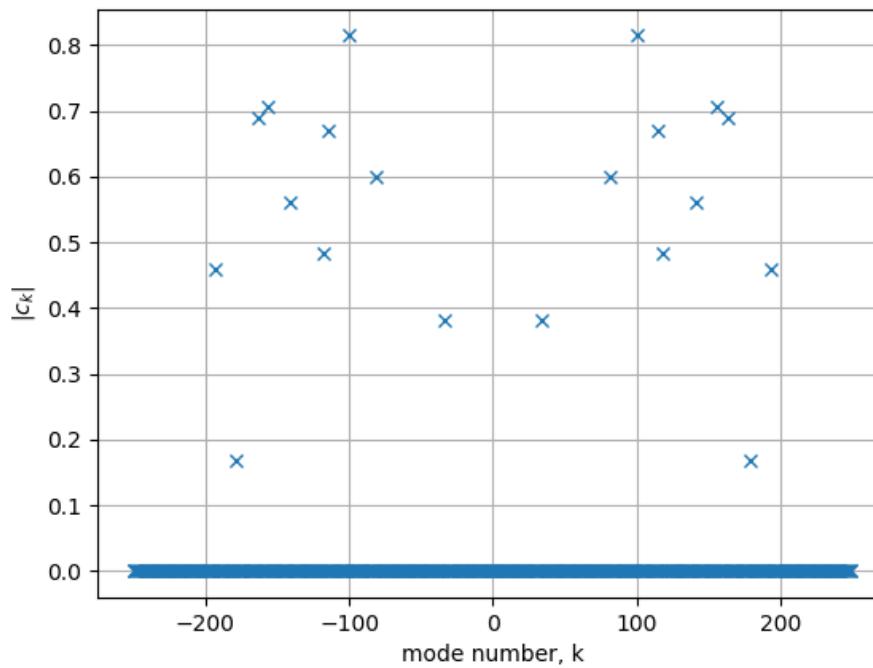
- `np.fft.ifft` will provide the original data given $n * [c_0, c_1, \dots, c_{n/2-1}, c_{-n/2}, c_{-n/2+1}, \dots, c_{-1}]$ as input (and the cost is $\mathcal{O}(n \log_2 n)$).
- Let's compute the FFT of our example signal...

An example



```
c = np.fft.fft(y)
c = np.fft.fftshift(c)/n
k = np.arange(-n/2,n/2)

plt.figure()
plt.plot(k,np.abs(c),'x')
plt.xlabel('mode number, k')
plt.ylabel('$|c_k|$')
plt.grid()
```

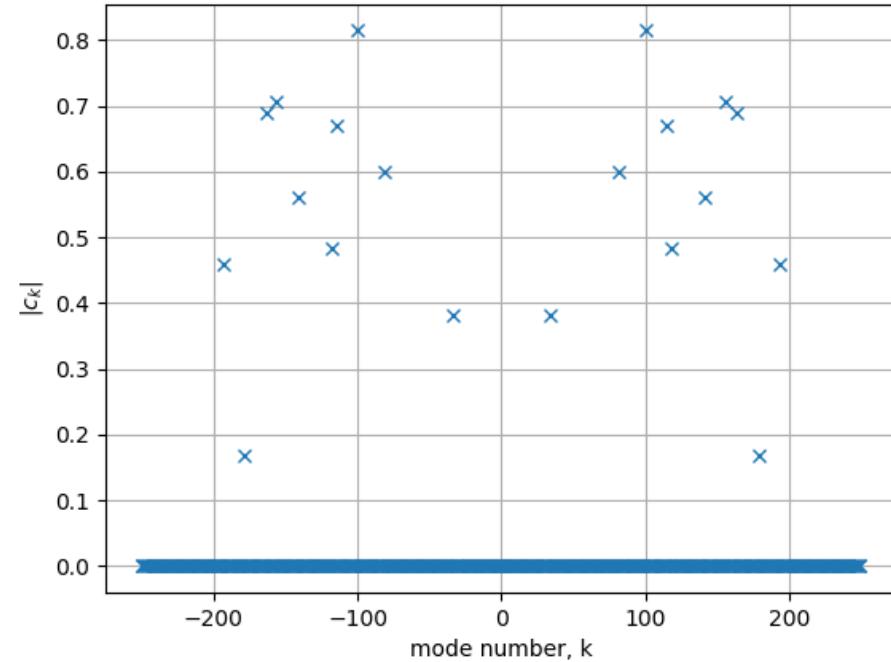


- The signal is a superposition of 10 sine waves w/ different frequencies and with randomly chosen amplitudes and phases:

$$y(t_j) = \sum_{m=1}^{10} a_m \sin(2\pi f_m t_j + \phi_m)$$

- The m^{th} sine wave has frequency $f_m = \frac{r_m}{\tau}$ with $\tau = 10$, and r_m a randomly chosen integer, $1 \leq r_m < n/2$

- $|c_k|^2$ is the “energy” in a mode with frequency, k/τ
- Note that there is a discrete version of Parseval’s theorem: $\frac{1}{n} \sum_{j=1}^n |y_j|^2 = \sum_{k=1}^n |c_k|^2$
- For real-valued data: $c_k = -c_{-k}^*$, so when considering the energy, only $k \geq 0$ needs to be considered (and it is more efficient to use `np.fft.rfft` instead of `np.fft.fft`)
- It is important here that $\tau = n\Delta t$ is an integer multiple of $1/f_m$ for each m as this ensures that the data can be represented efficiently with our basis functions
- $(n/2 - 1)/\tau = 1/2\Delta t - 1/\tau$ is the highest frequency that can be “resolved”
- Rule-of-thumb: >2 data points per period of highest-frequency component are needed for the FFT to accurately “identify” that component. So in our example, we require $2\Delta t < 1/\max(f_m)$



Another example

- Let's take two sine waves with n data points each and with frequencies $f_A = 2/\tau$ and $f_B = 2.5/\tau$. Here, $\tau = n\Delta t$ as before and $1/f_A$ and $1/f_B$ are the periods of the waves:
- Problem setup:

```
In [3]: tau = 5
```

```
In [4]: n = 100
```

```
In [5]: t = np.linspace(0,tau,n+1)
```

```
In [6]: t = t[:-1]
```

```
In [7]: fA = 2/tau
```

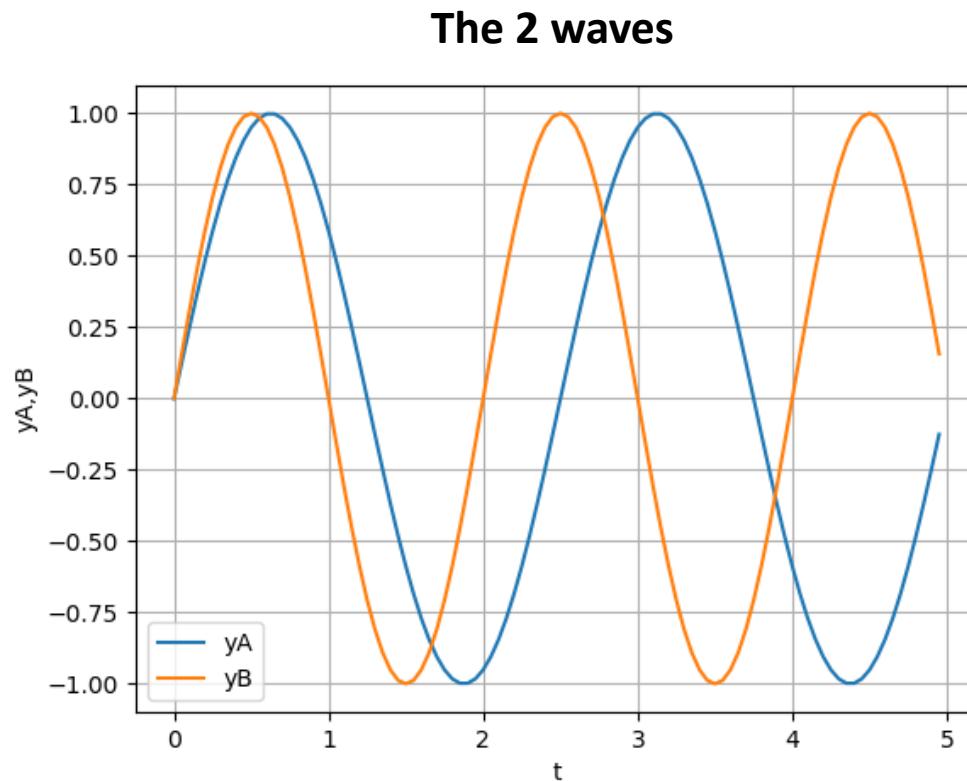
```
In [8]: fB = 2.5/tau
```

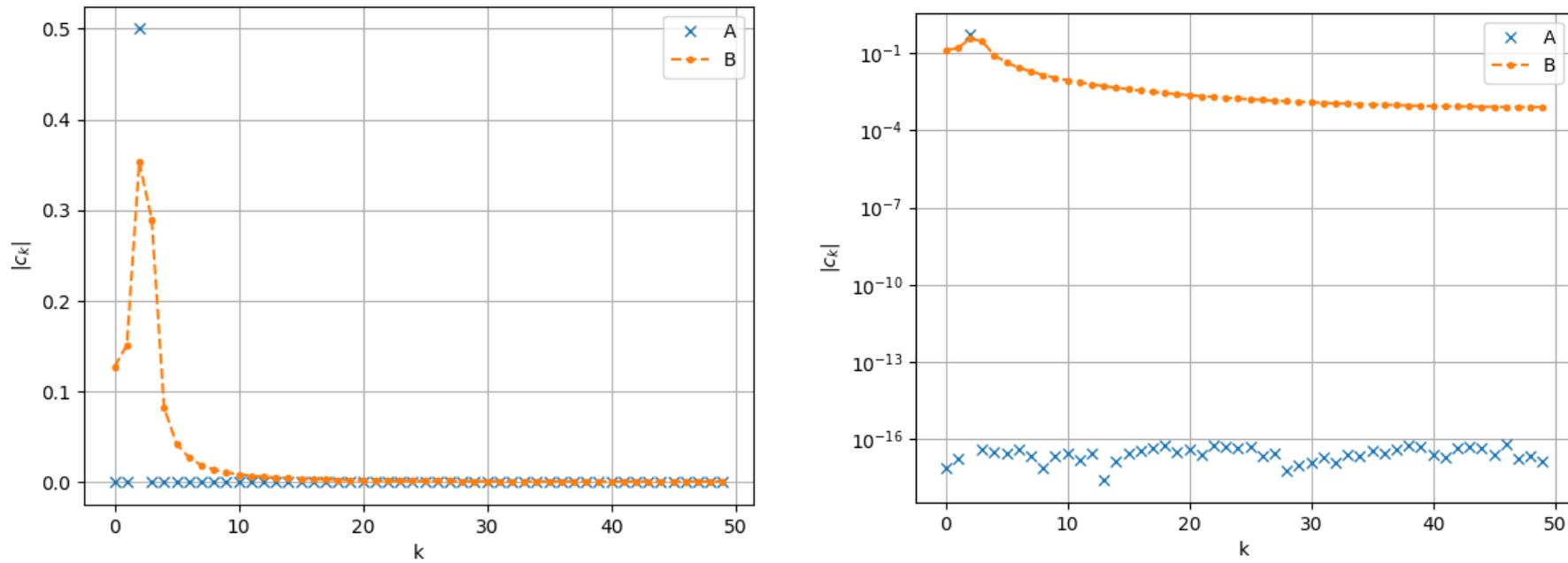
```
In [10]: yA = np.sin(2*np.pi*fA*t)
```

```
In [11]: yB = np.sin(2*np.pi*fB*t)
```

```
In [12]: cA = np.fft.fft(yA)/n
```

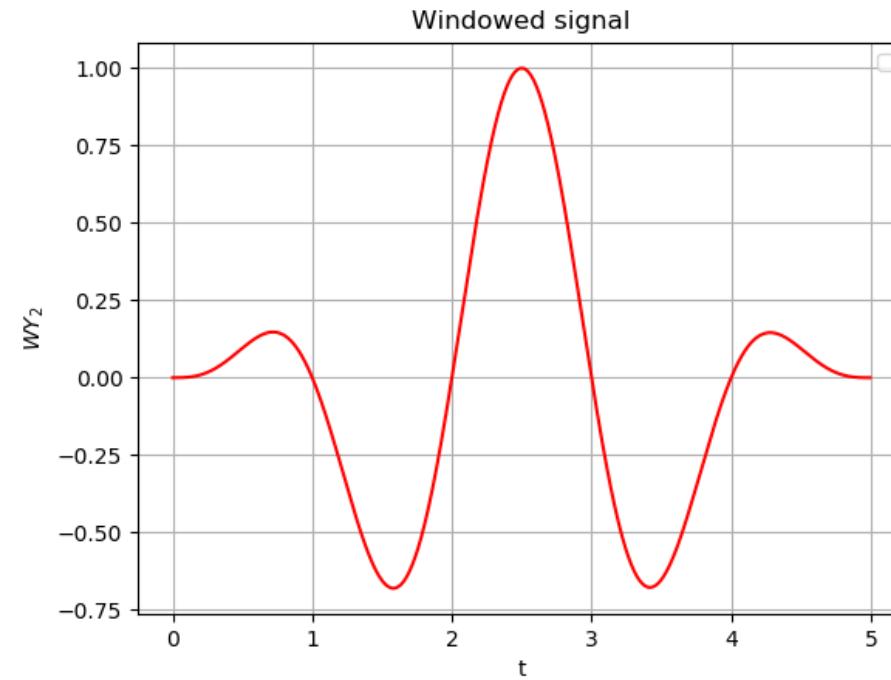
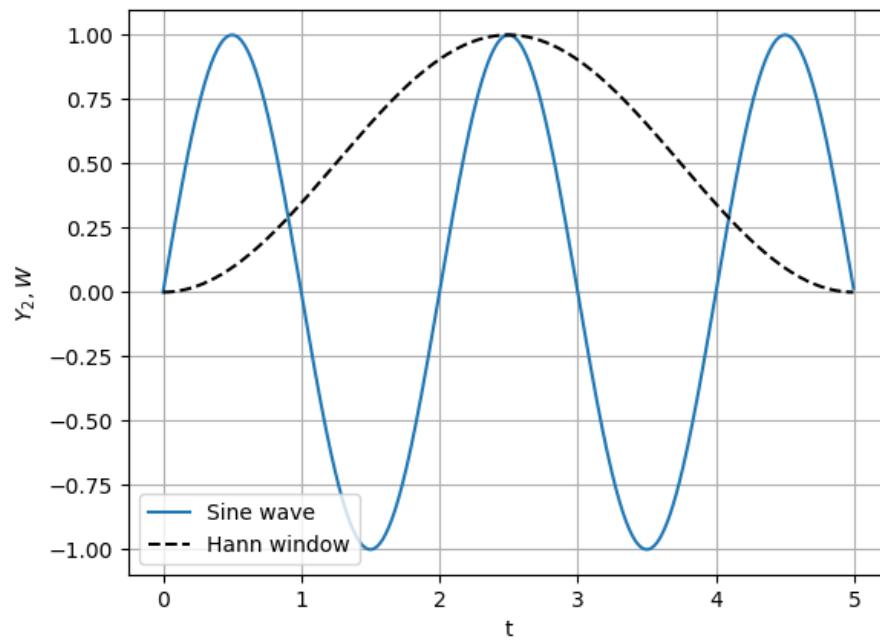
```
In [13]: cB = np.fft.fft(yB)/n
```

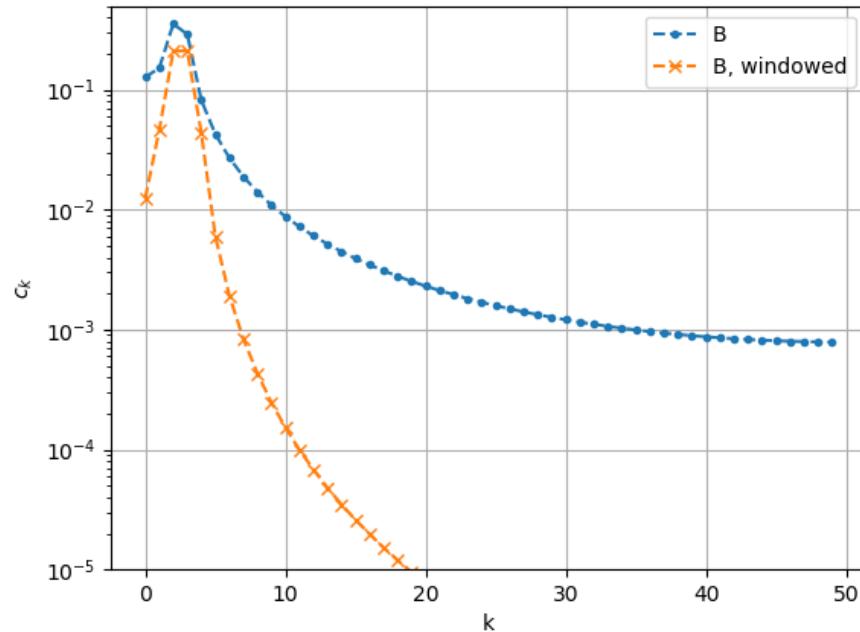




- The magnitudes of the Fourier coefficients, $|c_k|$, $k \geq 0$, for the two waves are shown on these plots (with different axis scales)
- We can see that wave A is represented perfectly with modes $k = \pm 2$ (only the $k = 2$) mode is shown
- Wave B is different. There is a spread of energy across multiple frequencies and there is a much slower decay of the Fourier coefficients. The reason is that for wave B, $y_B(\tau) \neq y_B(0)$, and the basis functions in the expansion all are periodic over a timespan of length τ

- There is no reason to generally expect signals to be periodic, so this is an important issue
- The standard “fix” is to use windowing to force the function towards zero at the endpoints of the time span:





- The spectrum of the windowed signal “looks” more like a simple wave
- However, we can clearly see that there is a loss of energy here (how would you quantify this?)
- There is no perfect solution, more advanced methods have been developed that reduce this energy loss while preserving the “wave-like” nature of the spectrum

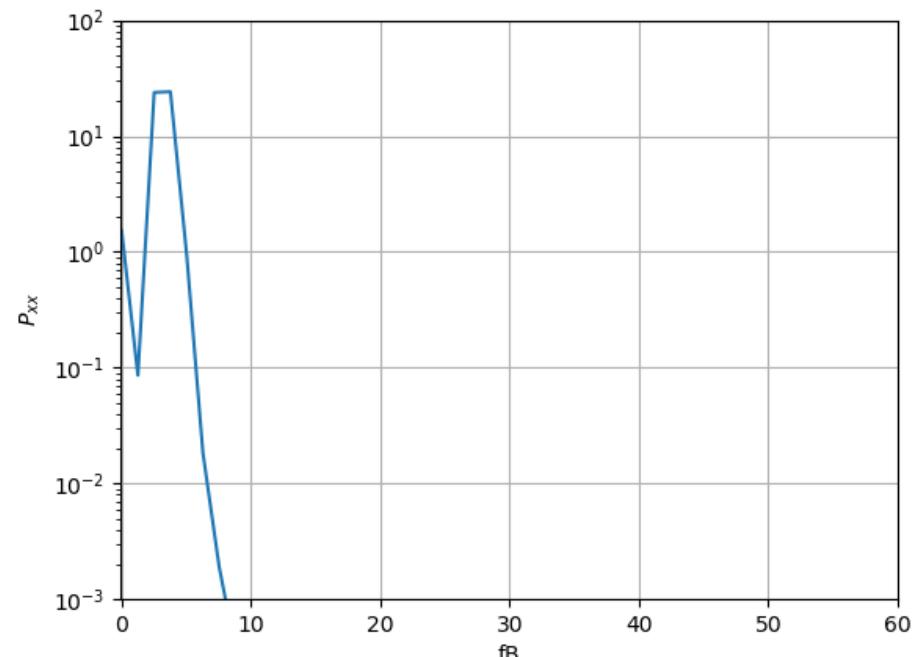
Data analysis

- Signal processing tools exist which:
 - Break the signal up into overlapping segments
 - Window the signal within each segment and compute the spectrum
 - Average the spectra from each segment (typically $|c_k|^2$ rather than $|c_k|$)
- This produces an estimate of the *autospectral density* (also called the *power spectral density*) which, for a given frequency, $f_k = k/\tau$, is $P_{xx}(f_k) = |c_k|^2 \frac{\Delta t}{n}$
- And can be computed using *Welch's method*, `scipy.signal.welch`:

```
In [201]: fB,PxxB = sig.welch(yB,1/dt)
```

```
In [202]: plt.semilogy(fB,PxxB)
```

- This figure isn't particularly impressive as the example is quite contrived however this is a *very* widely-used method for complicated signals



Notes:

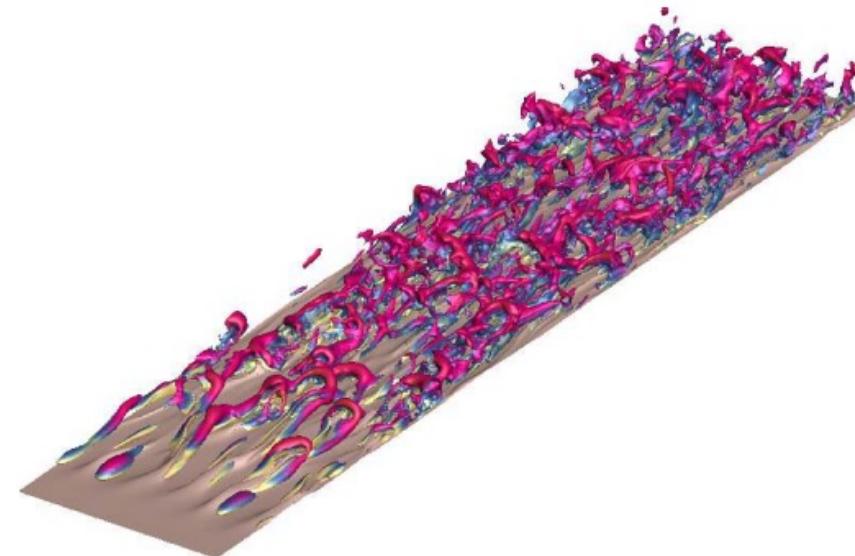
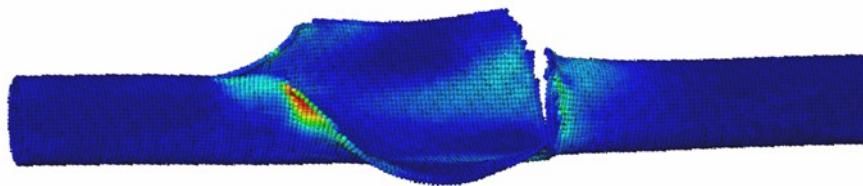
- The fundamental idea here is to view a signal of n points with step Δt as a distribution of “energy” across a range of frequencies
- We need $\Delta t < 1/(2f_{max})$ to *resolve* the highest frequency components
- Also need $\tau \gg 1/f_{min}$ to ensure “slow” components are contained within the signal
 - This was not the case in our previous example
 - Often, slow components contain the most energy

Lecture 16

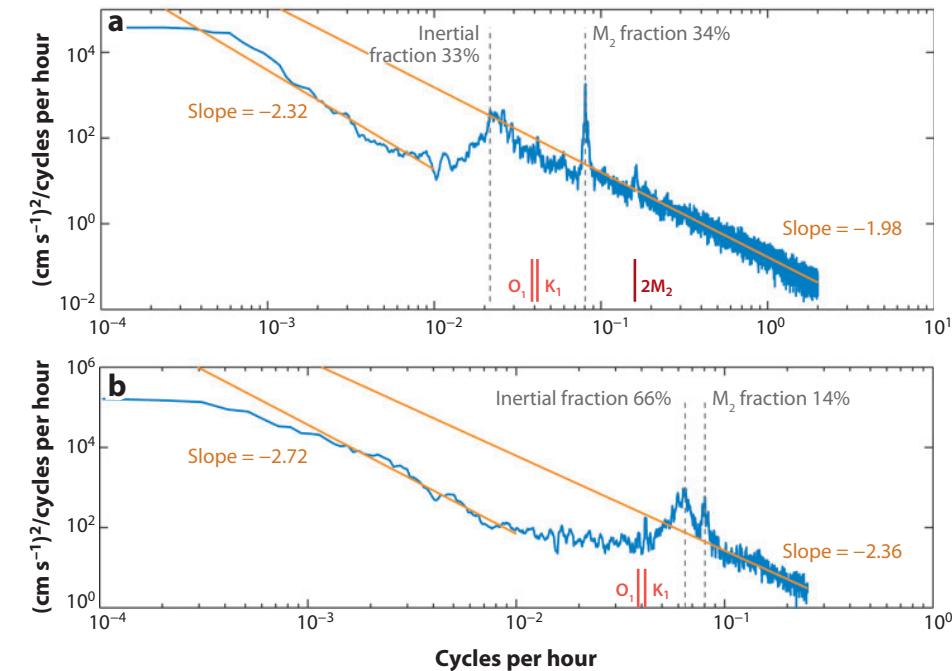
Multiscale science and engineering
2nd-order finite difference method and wavenumber analysis
Implicit finite difference methods

Multiscale science and engineering

- Climate modelling, aerodynamics, and material simulation are all highly nonlinear, multiscale, and very complex
- Which numerical methods are best?
- Today: Analyzing and designing finite difference methods for multiscale problems



- Generally, nonlinearity can lead to the generation of a broad range of scales (frequencies and wavelengths)
 - This effect is often balanced by diffusion which “smooths out” high-frequency fluctuations
- Consider a model with a $u \frac{\partial u}{\partial x}$ term and say at some time, we have $u = \sin(kx)$
 - Then we will have $u \frac{\partial u}{\partial x} = k \sin(kx) \cos(kx) = \frac{1}{2} k \sin(2kx)$ -- nonlinearity generates a function varying twice as fast as our original function
- The cumulative effect of nonlinearity can then produce *broadband* power spectral densities. The figure shows two examples based on measurements taken in the Atlantic and Southern oceans
- Numerical methods for simulations must accurately process a range of scales. E.g. they shouldn't “miss” the secondary peaks in the spectra



Taken from: Ferrara & Wunsch, Annual Review of Fluid Mechanics (2009)

2nd-order finite difference

- We have data on a n -point, equispaced grid: $f(x_j)$, $x_j = jh$, $j = 0, 1, 2, \dots, n - 1$
 - h is the *grid spacing*
- The 2nd-order centered approximation for first derivative follows from:

$$f'_j = \frac{f_{j+1} - f_{j-1}}{2h} + \text{truncation error}$$

- What is the truncation error? We can use Taylor series expansions to answer this →

$$f_{j+1} = f_j + hf'_j + \frac{h^2}{2}f''_j + \frac{h^3}{6}f'''_j + \dots$$

$$f_{j-1} = f_j - hf'_j + \frac{h^2}{2}f''_j - \frac{h^3}{6}f'''_j + \dots$$

- Subtract 2nd equation from first: $f'_j = \frac{f_{j+1} - f_{j-1}}{2h} + \frac{h^2}{6}f'''_j$ and $f'_j \approx \frac{f_{j+1} - f_{j-1}}{2h}$ is the *2nd-order centered finite difference (FD) method*
- We say the “error is $\mathcal{O}(h^2)$ ”, and the method is “second-order accurate”, but we will see there is more information about the approximation that is “hiding” in this error term

-
- This scheme is straightforward to implement in Python. Let's compute the derivative of $\sin(kx)$ with $0 \leq x \leq l$ and we will set $k = \frac{8\pi}{l}$ and $l = 5$
 - Step 1: generate grid and function:

```
#generate grid
L = 5
N = 100
x = np.linspace(0,L,N)
h = x[1]-x[0]
hfac = 1/(2*h)
```

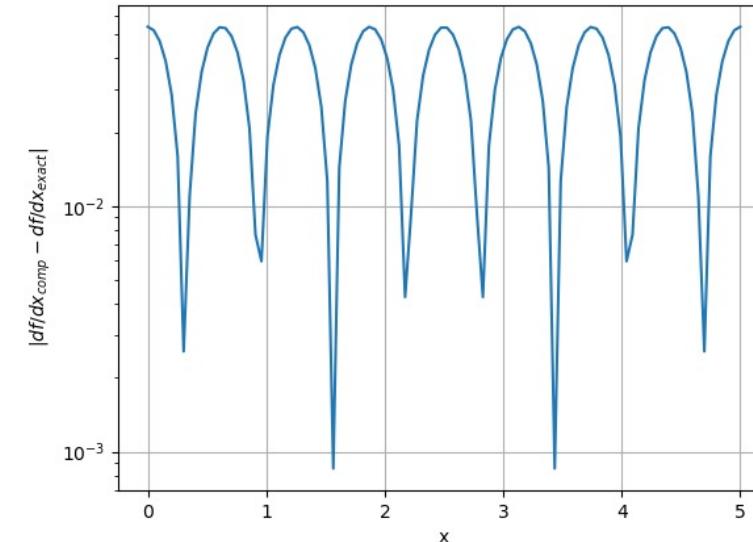
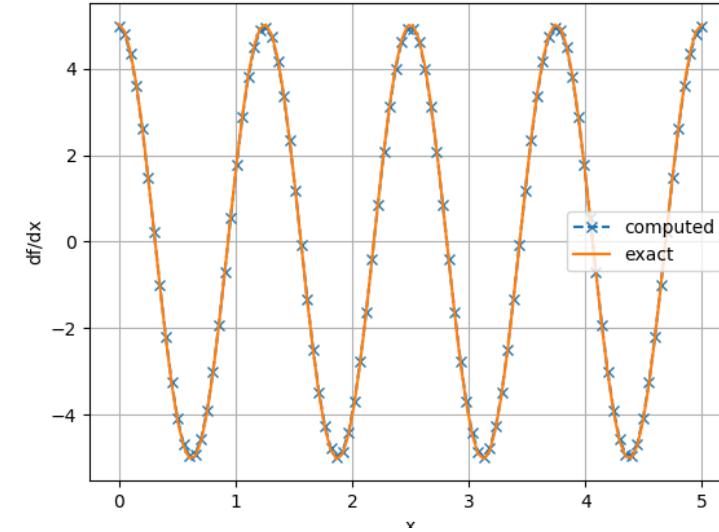
```
#generate function
k = 2*np.pi*4/L
f = np.sin(k*x)
```

- Step 2: compute derivative:

```
#compute derivative
df = np.zeros_like(f)
df[1:N-1] = hfac*(f[2:N]-f[0:N-2])
df[0] = hfac*(f[1] - f[-2])
df[-1] = df[0]
```

- The computed and exact solutions are shown on the right with $n = 100 \rightarrow$
- Visually, the agreement is good, but we should quantify the agreement, and defining a local error as, $\epsilon(x) = \left| \frac{df}{dx}_{computed} - \frac{df}{dx}_{exact} \right|$, this error is shown in the 2nd figure
- Further work would examine the dependence of the error on h and there it would be useful to work with the average and maximum of $\epsilon(x)$

Check your understanding: how will the average of ϵ change if n is doubled to 200?



Wavenumber analysis

- Now consider data corresponding to a complex sinusoidal ‘wave’ with wavenumber, k : $f(x) = e^{ikx}$ which has derivative, $df/dx = ike^{ikx}$
- Let’s look at the 2nd-order centered finite difference (FD) approximation for df/dx :

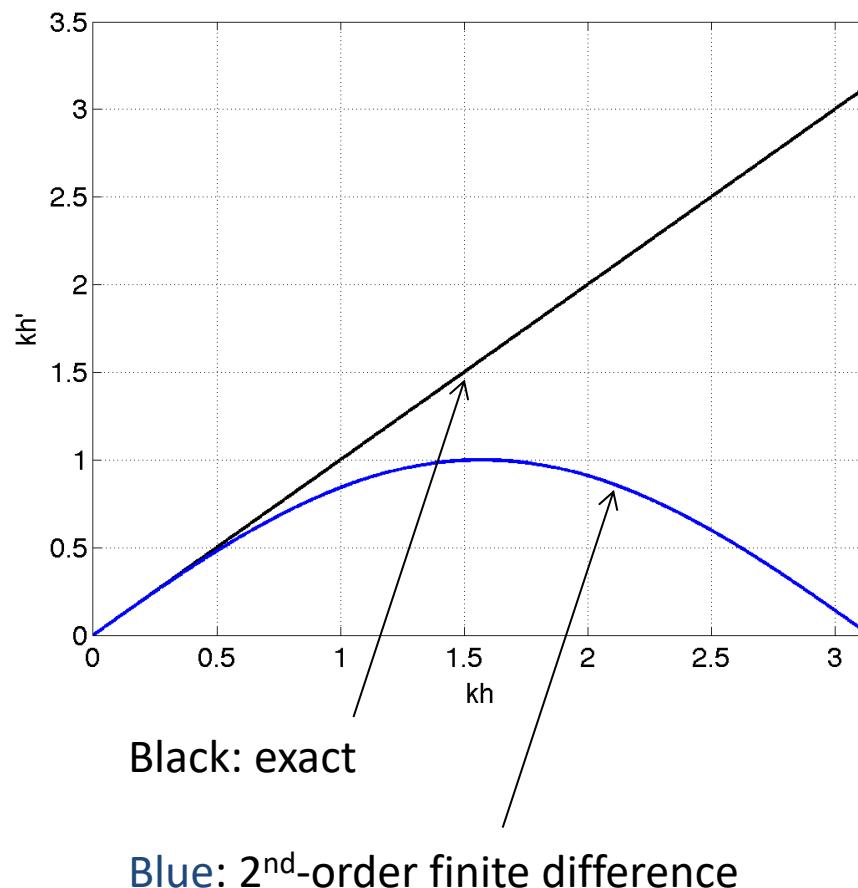
$f_{j+1} = e^{ik(x_j+h)}$ and $f_{j-1} = e^{ik(x_j-h)}$ so:

$$\frac{f_{j+1}-f_{j-1}}{2h} = \frac{e^{ik(x_j+h)} - e^{ik(x_j-h)}}{2h} = \frac{e^{ikx_j}}{2h} (e^{ikh} - e^{-ikh}) \text{ or,}$$

$$\boxed{\frac{f_{j+1}-f_{j-1}}{2h} = i \left[\frac{\sin(kh)}{h} \right] e^{ikx_j}}$$

- Comparing this last equation to ike^{ikx} , we see that the key question is, how close is $\frac{\sin(kh)}{h}$ to k ?
- It is more convenient to ask, how close is $\sin(kh)$ to kh ?

- The figure below shows $\sin(kh)$ vs. kh
- $\sin(kh)$ is the *modified wavenumber* for this finite-difference method, and generally, we will use kh' to indicate a modified wavenumber



- The figure shows that the FD method is only accurate for low wavenumbers (long waves)
- The error is 1% when $kh \approx 0.25$
 - The wavelength is $\lambda = \frac{2\pi}{k}$, and we can restate this as $\frac{\lambda}{h} \approx 26$
- So we need ≈ 27 points/wavelength for $\approx 1\%$ error
- Differentiation using DFTs requires > 2 points/wavelength for the exact solution (this method will be discussed in lab 8)

Check your understanding: the 2nd-order accurate centered finite difference method for the second derivative is: $f_j'' = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2}$

What is the modified wavenumber for this scheme?

FD schemes with better resolution?

- Can we construct a FD scheme that is more accurate for higher wavenumbers (without excessive additional cost)?
- When we looked at time marching methods, we saw :
 - Explicit Euler method has poor stability properties
 - Implicit Euler method: much better stability, but requires matrix inversion → more expensive
- We can also construct “implicit” finite difference schemes for numerical differentiation
- Our 2nd-order explicit finite difference formula is, $f'_j = \frac{f_{j+1} - f_{j-1}}{2h}$
- And for implicit schemes, we will work with the general form,
$$\beta f'_{j-2} + \alpha f'_{j-1} + f'_j + \alpha f'_{j+1} + \beta f'_{j+2} = c \frac{(f_{j+3} - f_{j-3})}{6h} + b \frac{(f_{j+2} - f_{j-2})}{4h} + a \frac{(f_{j+1} - f_{j-1})}{2h}$$
- We now have a system of equations of the form, $\mathbf{Af}' = \mathbf{b}$ where \mathbf{A} is a *banded* (pentadiagonal) matrix

-
- How do we set the coefficients? We should choose them so that the scheme is “accurate” and remember we also would like the method to work well for high-wavenumber data
 - Let’s first think about the truncation error
 - Using Taylor series and after a lot of arithmetic, we get the following conditions →

$$2^{\text{nd}} \text{ order: } a + b + c = 1 + 2\alpha + 2\beta$$

$$4^{\text{th}} \text{ order: } a + 2^2 b + 3^2 c = 2 \frac{3!}{2!} (\alpha + 2^2 \beta)$$

$$6^{\text{th}} \text{ order: } a + 2^4 b + 3^4 c = 2 \frac{5!}{4!} (\alpha + 2^4 \beta)$$

- The highest possible accuracy is 10th order which we get with:

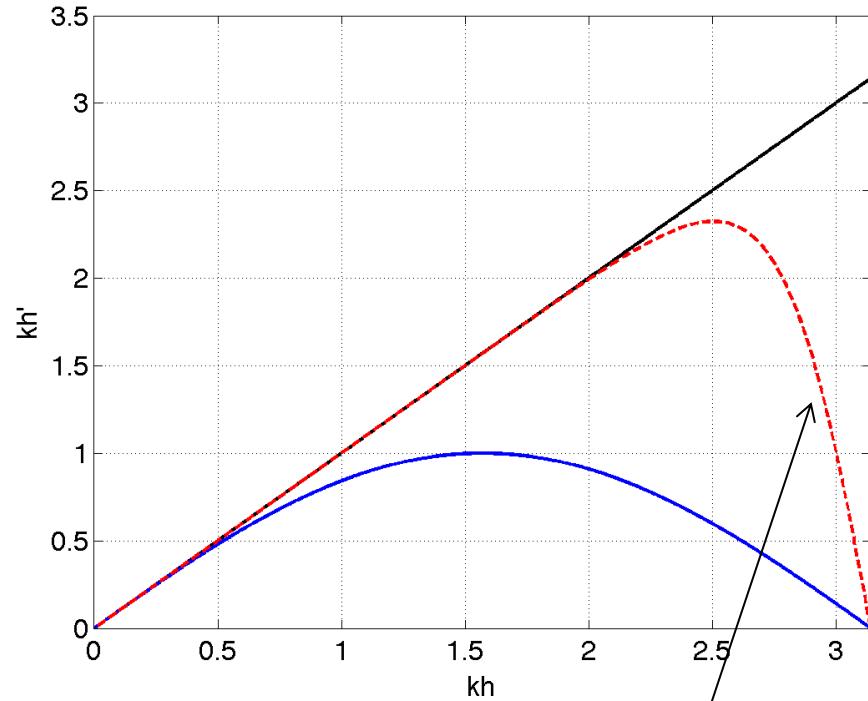
$$\alpha = \frac{1}{2}, \beta = \frac{1}{20}, a = \frac{17}{12}, b = \frac{101}{150}, c = \frac{1}{100}$$

But is this the best scheme?

-
- Let's use wavenumber analysis for guidance, and as before, we assume that $f(x) = e^{ikx}$
 - Then, after working through the arithmetic, we find that the modified wavenumber for our general implicit FD method is:

$$kh' = \frac{a \sin(kh) + (b/2)\sin(2kh) + (c/3)\sin(3kh)}{1 + 2\alpha \cos(kh) + 2\beta \cos(2kh)}$$

- The 10th-order scheme is the best we can do in terms of the order of the truncation error, but does wavenumber analysis also indicate it is best?



Black: exact

Blue: 2nd-order finite difference

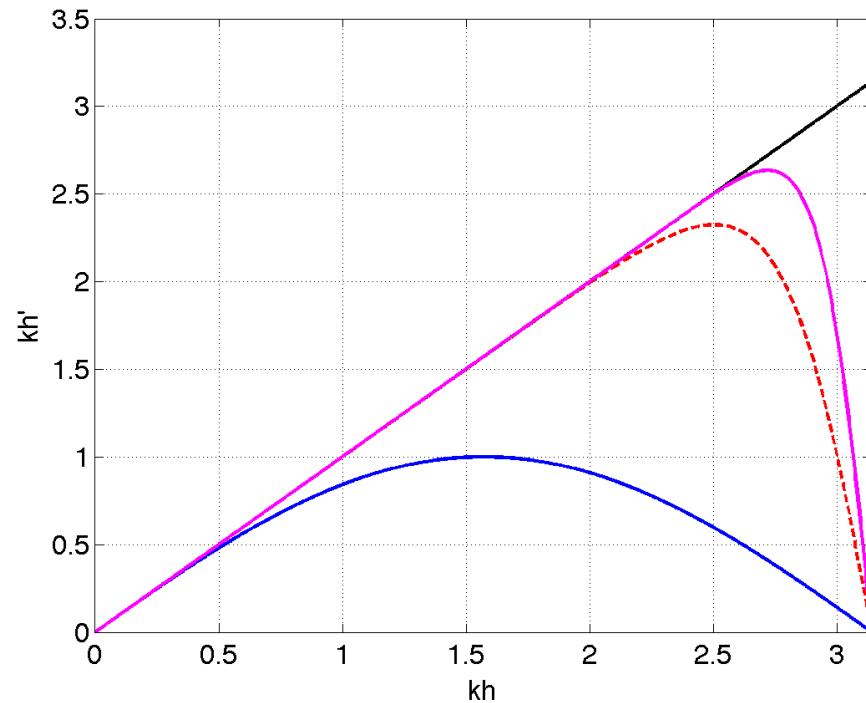
Red: 10th-order finite difference

- The 10th order method is certainly much more accurate than the 2nd order FD scheme
- For the 10th order method, we need ≈ 4 grid points per wavelength for 1% error
- But can we do better?

-
- 4th order schemes have three free constants, let's choose these to optimize the results for high wavenumbers
 - We will apply the following (somewhat arbitrary) constraints:
 - $kh'(kh = 2.2) = 2.2$
 - $kh'(kh = 2.3) = 2.3$
 - $kh'(kh = 2.4) = 2.4$
 - So we are requiring that differentiation of the exponential function should be exact for these three values of kh
 - Applying these constraints, and requiring fourth-order accuracy gives the following coefficients:

$$\alpha = 0.5771439, \beta = 0.0896406, a = 1.3025166, b = 0.99335500, c = 0.03750245$$

- Let's look at the modified wavenumber for this scheme...



- 10^{th} order ≈ 4 points/wavelength for 1% error
- 4^{th} order ≈ 3.4 points/wavelength for 1% error
- Order of accuracy isn't everything! The 4th-order scheme does very well for high wavenumbers (as it was designed to do)

Black: exact

Blue: 2nd-order

Red: 10th-order

Magenta: Optimized, 4th order

-
- What are the operation counts for these schemes?
 - 2nd order FD: N mult + N add
 - 4th order implicit FD: solve pentadiagonal linear system,
 $7N$ mult + $7N$ add
 - Spectral (Fourier): $\sim N \log_2 N$

Key question: which method requires least time for desired accuracy? This depends on the detail of the problem and what the desired accuracy is.

Python implementation

- The optimized 4th order scheme requires the solution of $\mathbf{A}\mathbf{f}' = \mathbf{b}$ and \mathbf{A} is a pentadiagonal matrix; pentadiagonal matrices have the general form:

$$\begin{bmatrix} f_1 & g_1 & h_1 & & \\ e_2 & f_2 & g_2 & h_2 & \\ d_3 & e_3 & f_3 & g_3 & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ & & & \ddots & \vdots \\ & & & d_{n-1} & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & d_n & e_n & f_n & \end{bmatrix}$$

- How do we solve this? In general, for problems like this, we can use, `np.linalg.solve(A,b)`
- But this isn't very efficient
 - It requires excessive memory (storing the zeros in \mathbf{A})
 - And it also requires excessive operations (again due to the zeros)

-
- We can instead build a sparse banded matrix using `scipy.sparse.diags`
 - First construct diagonals:

```
#RHS
a = 1.3025166
b = 0.9935500
c = 0.03750245
```

```
#LHS
ag = 0.5771439
bg = 0.0896406
```

```
#Construct A
zv = np.ones(N)
agv = ag*zv[1:]
bgv = bg*zv[2:]
```

-
- Then construct sparse matrix, A:

```
A = sp.diags([bgv, agv, zv, agv, bgv], [-2,-1,0,1,2])
A.toarray()
```

Out[72]:

```
array([[1.        , 0.5771439, 0.0896406, 0.        , 0.        , 0.        , 0.        ],
       [0.5771439, 1.        , 0.5771439, 0.0896406, 0.        , 0.        , 0.        ],
       [0.0896406, 0.5771439, 1.        , 0.5771439, 0.0896406, 0.        , 0.        ],
       [0.        , 0.0896406, 0.5771439, 1.        , 0.5771439, 0.0896406],
       [0.        , 0.        , 0.0896406, 0.5771439, 1.        , 0.5771439],
       [0.        , 0.        , 0.        , 0.0896406, 0.5771439, 1.        ]])
```

and use `scipy.sparse.linalg.spsolve`

- This would solve the memory issue
- But it is still possible to do better with efficiency

-
- `scipy.linalg` has a solver specifically designed for banded matrices: `scipy.linalg.solve_banded` (this is actually a Fortran routine from the Lapack library)
 - This function is a little tricky to use
 - The matrix has to be provided in “matrix diagonal ordered form”
$$A_b[u + i - j, j] == A[i, j]$$
 - A_b is a 2-D array required as input to `solve_banded`
 - u , is the number of non-zero diagonals *above* the main diagonal (2 for our pentadiagonal matrix)
 - Now we (nearly) have the “best” approach for our problem
 - We need to modify our method for the top two and bottom two rows in the matrix ($i = 0, 1, n - 2, n - 1$ with python-indexing) where there isn’t “space” for 5 coefficients

Boundary modifications

- At $i = 1$ and $i = n - 2$, we can switch to a (8th-order) tridiagonal scheme with:

$$\alpha = \frac{3}{8}, \beta = 0, a = \frac{1}{6}(\alpha + 9), b = \frac{1}{15}(32\alpha - 9), c = \frac{1}{10}(-3\alpha + 1)$$

- And at $i = 0, n - 1$, we switch to 4th-order “one-sided” schemes:

$$f'_0 + \alpha f'_1 = \frac{1}{h}(af_0 + bf_1 + cf_2 + df_3)$$
$$f'_{n-1} + \alpha f'_{n-2} = -\frac{1}{h}(af_{n-1} + bf_{n-2} + cf_{n-3} + df_{n-4})$$

$$\text{with } \alpha = 3, a = -\frac{17}{6}, b = \frac{3}{2}, c = \frac{3}{2}, d = -\frac{1}{6}$$

- This works for *periodic* functions where the RHS can be evaluated using these schemes. For general functions, we would have to modify the scheme at $i = 2, n - 3$ as well

Final notes

- Advantages of optimized implicit FD schemes
 - ‘Competitive’ efficiency
 - Low memory usage (relative to explicit FD and Fourier) – second key question: how much memory is available? Is there a performance gain from using less memory? (low space complexity)
- Disadvantages
 - Errors in one location can “contaminate” results in all locations
 - Difficult to apply to complex geometries (cf. finite element methods)
- There is no single differentiation method which works best for all problems. Ultimately, when choosing one method vs. another, we have to ask, what is the cost for a desired level of accuracy?
- Numerical differentiation is one of three “fundamental” numerical analysis problems
 - Differentiation
 - Quadrature (`scipy.integrate`)
 - Interpolation (`scipy.interpolate`)

Lecture 17

Linear data analysis

Logistic map, fractal dimension, and correlation dimension

Lorenz system, phase plots, and attractor reconstruction

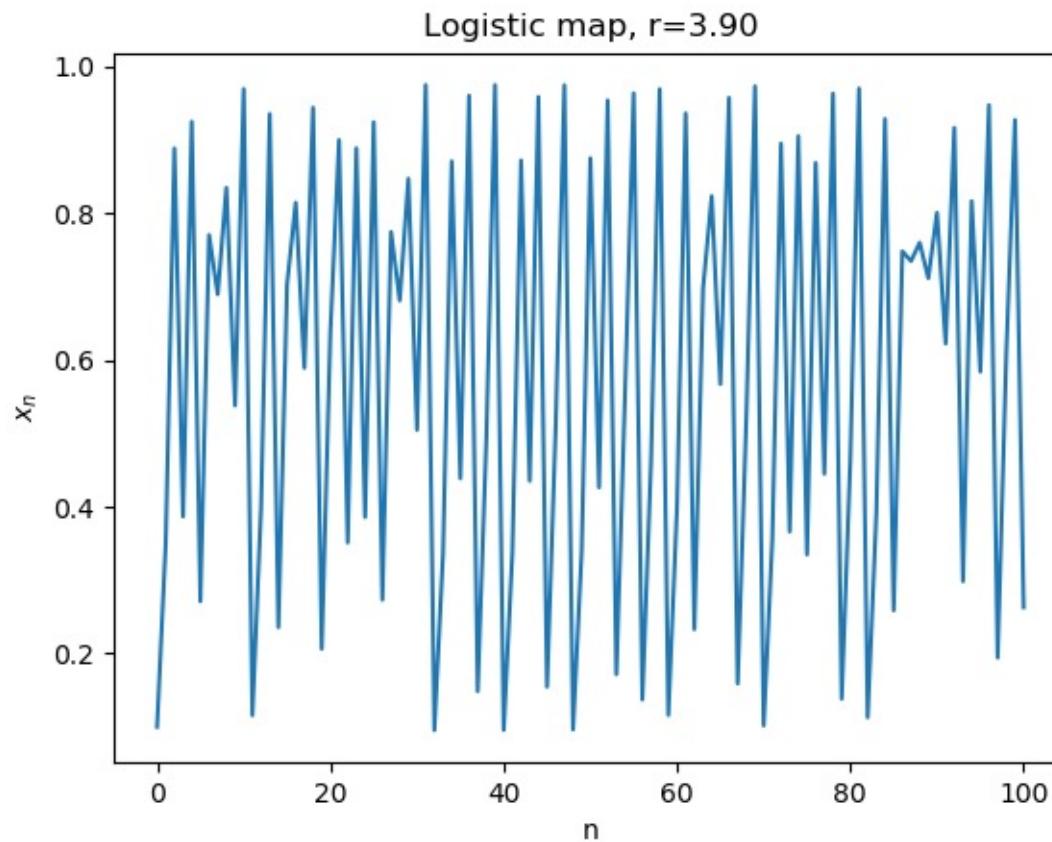
Transitions between states, orbit diagrams, and maps

Linear data analysis

We have looked at a few linear methods:

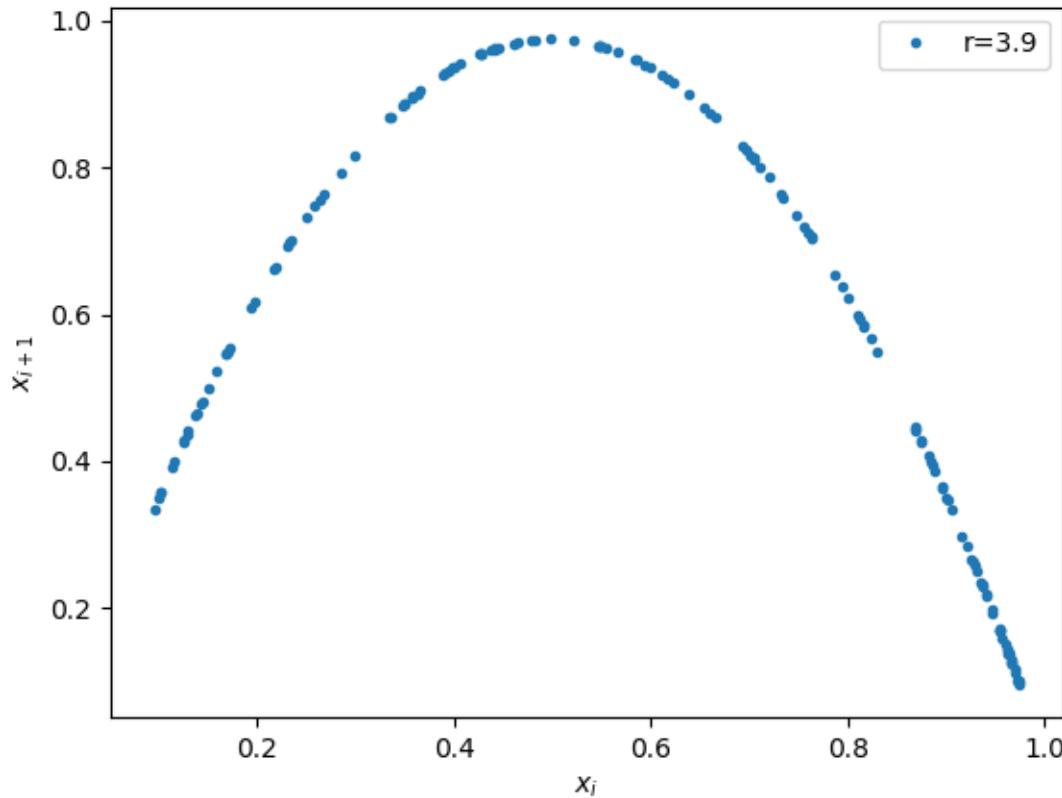
- DFT: linear superposition of complex exponentials
- PCA: linear transformation
- However, given that “complex systems” are typically nonlinear, we need to think about customized methods
 - Nonlinear PCA is one example
 - We will focus on ideas adapted from the study of nonlinear dynamics and chaos

- Fourier (energy or power) spectra are a good place to start with stationary data
 - If the length of the data is sufficiently large
 - And the sampling rate sufficiently large (Δx or Δt sufficiently small)
 - However consider the example below

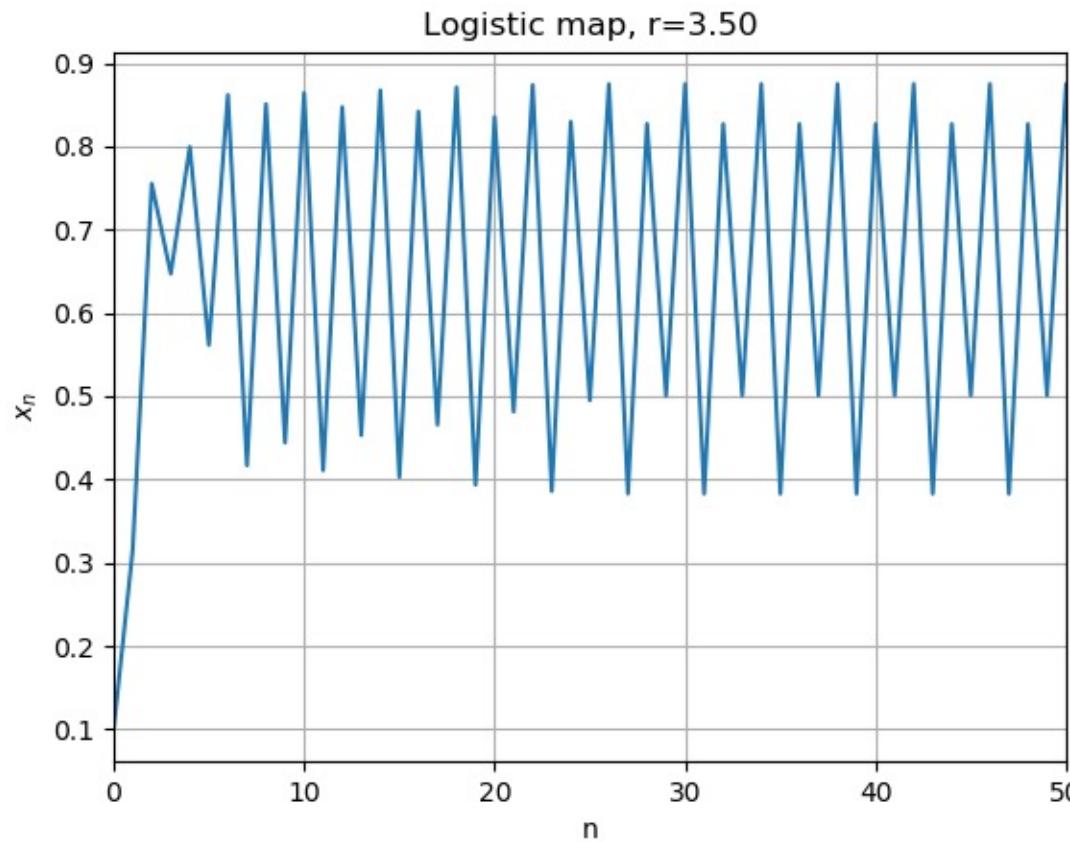


- What can we do with this data?
- FFT won't work – data is not smooth
- Three basic options:
 - Compare peaks to peaks
 - Or troughs to troughs
 - Or peaks to troughs
- Let's try the 3rd option

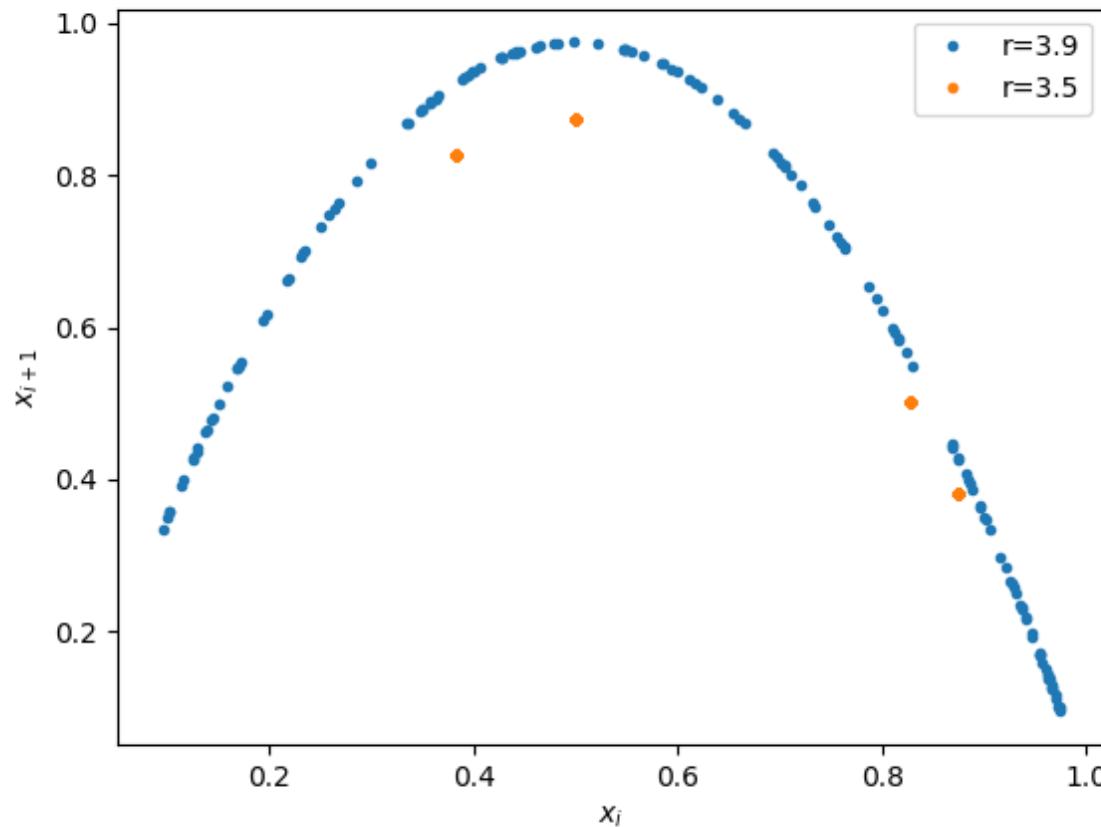
Logistic map



- There is clear “structure” within the data!
- The *logistic map* is:
$$x_{i+1} = rx_i(1 - x_i)$$
- The parameter, r , controls the dynamics.
 - For smaller r , simple periodic behavior
 - For $r = 3.9$, chaotic behavior
 - As time increases, more and more points on the parabola will be visited in an irregular order



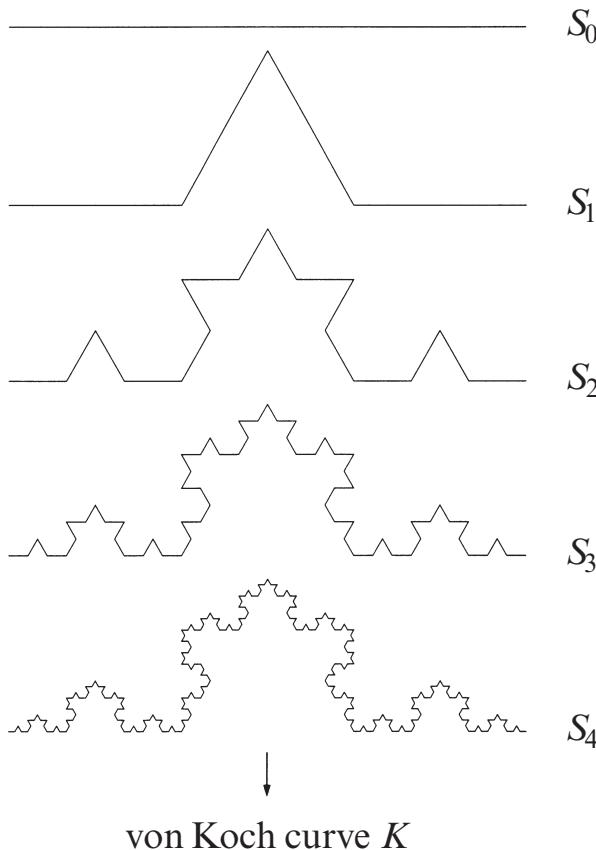
- At $r = 3.5$, there are *period-2* oscillations where: $x_{i+4} = x_i$
- When characterizing a solution, it is helpful to think about the set of points visited (after discarding the initial transient)
- And in particular, the *dimension* of this set



- The set of points generated when $r = 3.9$ is “almost” a one-dimensional curve
- While four points could be described as zero-dimensional
- How can we make these ideas more precise?

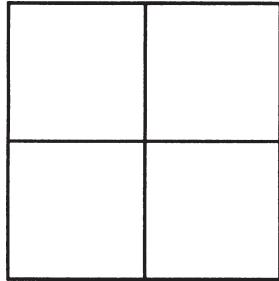
Fractal dimension

- We borrow from the study of fractals where there is a *similarity* or *fractal dimension*

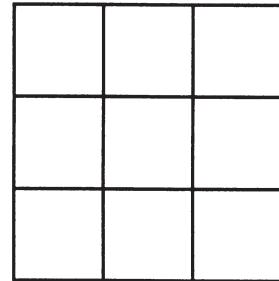


- An example: the Koch curve

- Each “iteration”, the “curve” is broken up into 4 smaller copies of itself. The length of each copy is $1/3$ of the original. What is the dimension of this curve?



$$\begin{aligned}m &= 4 \\r &= 2\end{aligned}$$



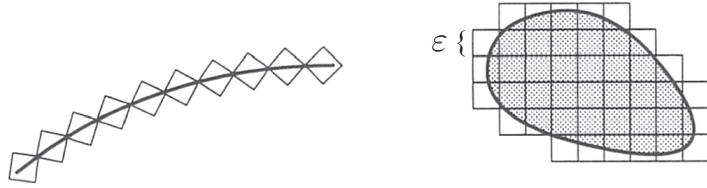
$$\begin{aligned}m &= 9 \\r &= 3\end{aligned}$$

m =number of copies
 r =scale factor

- A simpler example: A square.
 - Break a square into 4 smaller copies ($m = 4$) – the sides of each copy have been scaled down by a factor of 2 ($r = 2$)
 - With $m = 9$, we have $r = 3$
 - And the dimension is $d = \log(m)/\log(r) = 2$
 - For the Koch curve, $m = 4, r = 3, d = \log(4)/\log(3) = 1.261 \dots$

How do we compute the fractal dimension given data?

- One approach is to compute the *box dimension*:



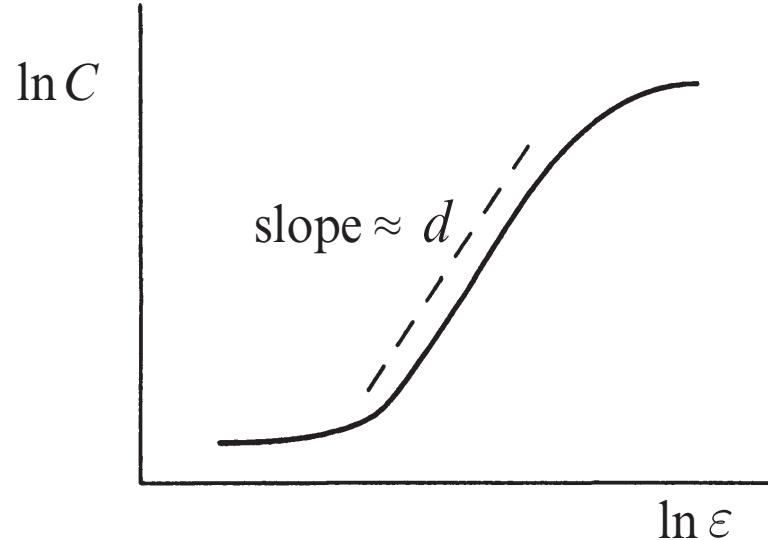
$$N(\varepsilon) \propto \frac{L}{\varepsilon}$$

$$N(\varepsilon) \propto \frac{A}{\varepsilon^2}$$

- Construct m -dimensional cubes with edge-length, ε , find the number of cubes needed to cover the set of points in the solution, $N(\varepsilon)$
- We expect: $N(\varepsilon) \propto 1/\varepsilon^d$ where d is the box dimension

-
- In practice, the box dimension is not used – its computation is too expensive for large high-dimensional sets. The *correlation dimension* is often used instead
 - We collect n m -dimensional points “visited” during a process after discarding the effect of the initial condition: $\{\mathbf{x}_i, i = 1, 2, \dots, n\}, \mathbf{x}_i \in \mathbb{R}^m$
 - The *correlation sum*, $C(\varepsilon)$ is: (total number of pairs of points within distance ε)/(Total number of distinct pairs)
 - $$C(\varepsilon) = \frac{2}{n(n-1)} \sum_{i=1}^n \sum_{j=i+1}^n H(\varepsilon - \|\mathbf{x}_i - \mathbf{x}_j\|)$$
 - Here, H , is the Heaviside function, $H(z) = 0$ if $z \leq 0$, $H(z) = 1$ if $z > 0$
 - $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the Euclidian distance between the m -dimensional vectors \mathbf{x}_i and \mathbf{x}_j
 - For points falling on a fractal-like structure, we expect: $C(\varepsilon) \sim \varepsilon^d$ where d is now the correlation dimension

Generically, we expect:



- At small ε , there will be few if any pairs of points within ε of each other
- At sufficiently large ε , almost all pairs of points will be within ε of each other
- Often some trial-and-error is needed to choose a good range of ε

It is straightforward to construct a plot like this for the logistic map using `scipy.spatial.distance.pdist` to compute the distances (look up the documentation!)

Example: Correlation dimension computation for logistic map

1. Compute solution:

```
for i in range(n0):  
    x[i+1] = r*x[i]*(1-x[i])
```

2. Discard influence of initial condition:

```
y = x[n0//2:]  
n = y.size
```

3. Split into two vectors ($m = 2$) and collect in $\frac{n}{2} \times m$ matrix

```
y1 = y[:-1:2]  
y2 = y[1::2]  
A = np.vstack([y1,y2]).T
```

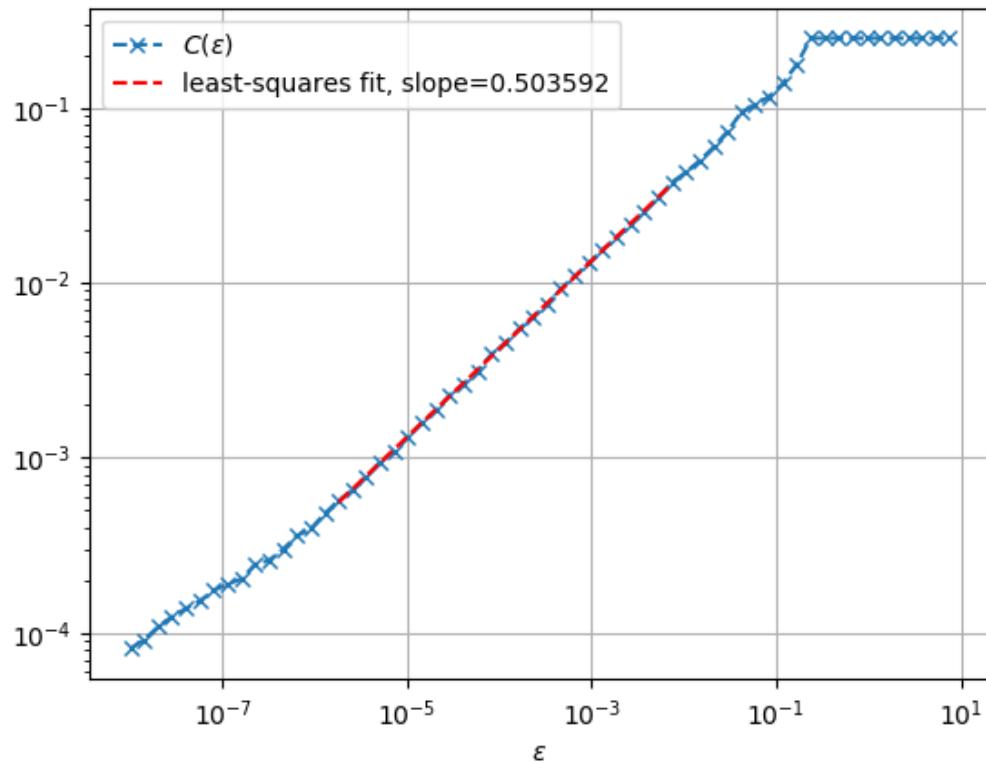
4. pdist will then compute all $\frac{\frac{n}{2}(\frac{n}{2}-1)}{2}$ distances:

```
D = pdist(A)
```

5. Now we just need to pass the computed distances through the Heaviside function for a range of ε ...

`D = D[D<eps[i]] #Discard distances larger than eps. Assumes eps[i+1]<eps[i]`

`C[i] = D.size #Size of new D is Correlation sum (without n/2*(n/2-1)/2 scaling)`



- Results for $r = 3.5699456$ are shown, $n = 8000$
- 1st 15 and last 20 points have been discarded for best fit calculation, fractal dimension is estimated as 0.5 (for this value of r)
- Estimate computed using `np.polyfit`

Lorenz system

A (famous) example:

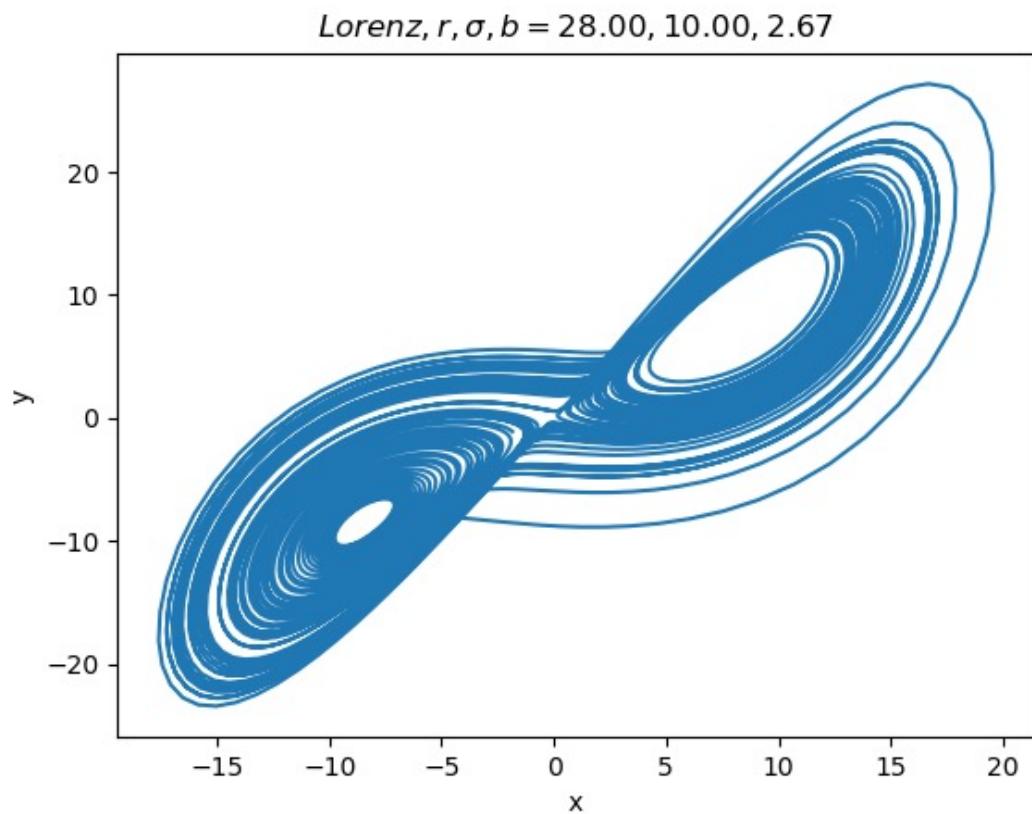
$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = rx - y - xz$$

$$\frac{dz}{dt} = xy - bz$$

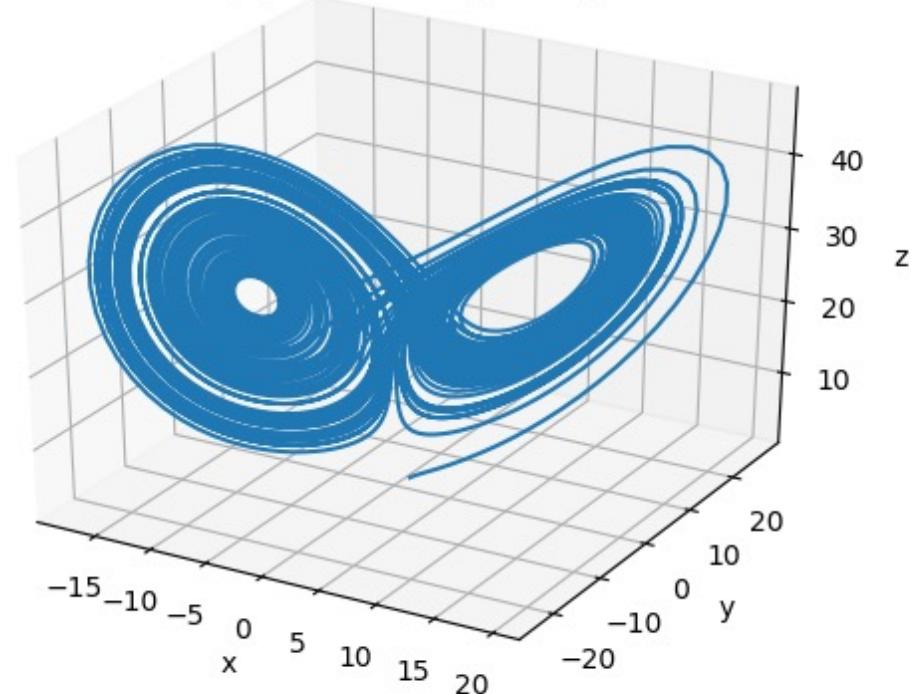
- σ , b , and r are parameters that must be specified. In certain parameter ranges, chaotic dynamics are generated
- Initially developed in 1960s as a model for atmospheric flows
- The modern field of chaotic dynamics emerged from the study of these equations
- We can integrate these equations using `solve_ivp` and we can then examine phase plots constructed using the results

-
- The plot shows the solution trajectory in the $x - y$ plane (a phase plane)
 - It looks like the trajectory is crossing itself (suggesting periodic dynamics), but we need to look at a 3D plot

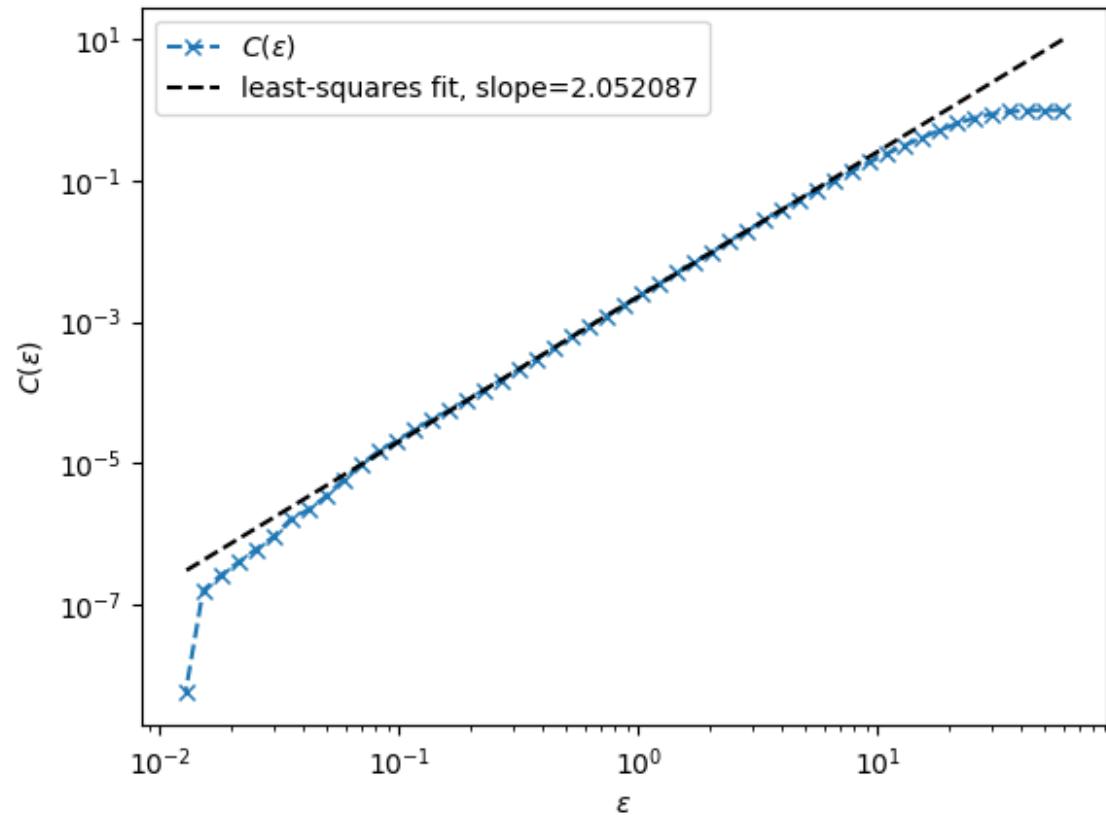


- A three-dimensional plots show how the trajectory “adjusts” in the third coordinate to avoid crossing itself
 - And this indicates that the solution is aperiodic
- We can again compute a correlation sum with n three-dimensional vectors:
$$\mathbf{v}_i = [x(t_i), y(t_i), z(t_i)]^T, (m = 3, i = 1, 2, \dots, n)$$
- Each point in the plot corresponds to one such vector
- Computing the correlation sum after discarding points for $t < 10\dots$

Lorenz, $r, \sigma, b = 28.00, 10.00, 2.67$



- The dimension is estimated to be approximately 2.05
- For an autonomous system of ODEs, a dimension greater than two is necessary for chaos, and our dimension estimate for the Lorenz system indicates *weak or low-dimensional chaos*
- And then aperiodic dynamics for more-complex systems could produce *high-dimensional chaos*

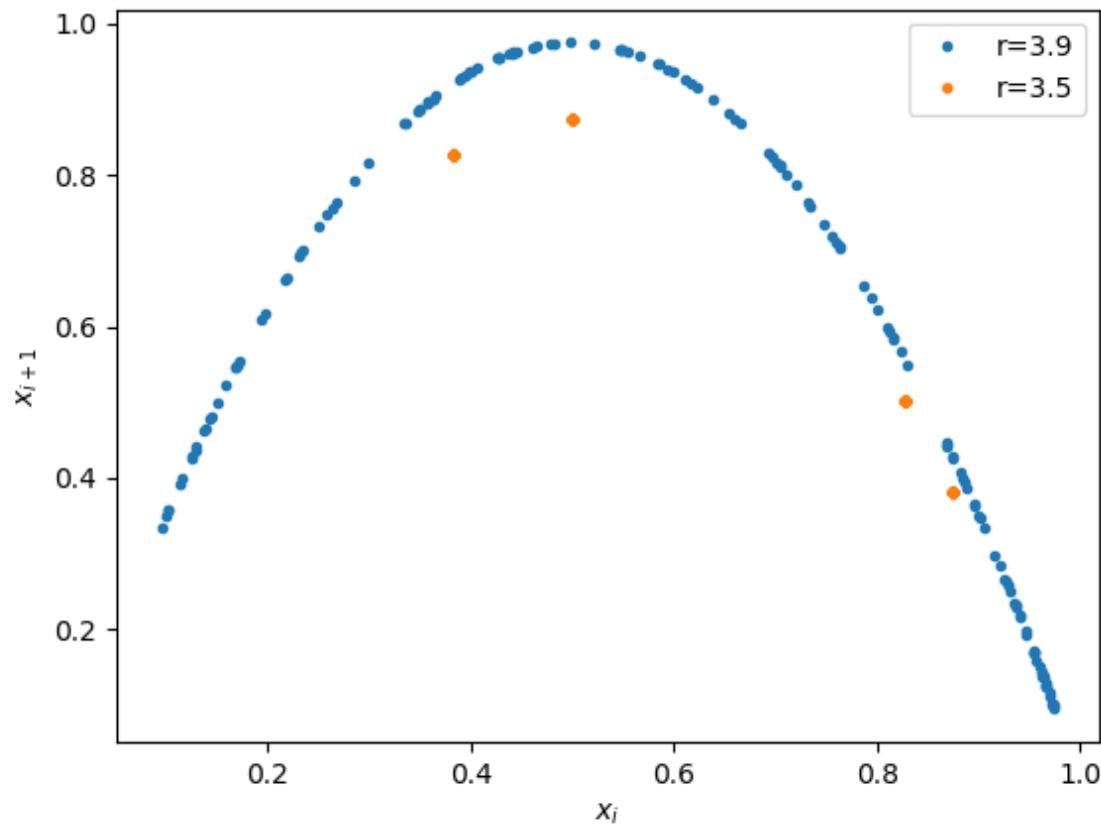


Attractor reconstruction

- For the logistic map and the Lorenz system, we can generate however much data we want, but this is not typically the case when we are analyzing data
- What do we do if we have a single (chaotic) time series?
- Or if we're working with PDEs where time series at locations close to each other tend to be correlated?
- We can then construct an m-dimensional vector at time, t_i , using time delays:
$$\mathbf{v}_i = [x(t_i), x(t_i - \tau), x(t_i - 2\tau), \dots, x(t_i - (m - 1)\tau)]^T$$
- And then after collecting n of these vectors, we can compute the correlation sum as before
- For chaotic systems, it is essential for the coordinates of the vector to be uncorrelated, i.e. the time delay, τ , must be sufficiently large

-
- How do we choose τ ?
 - This typically requires some trial-and-error and iteration
 - Often, a system has a dominant slow time scale (e.g. the time required to go around a loop on the Lorenz attractor)
 - Something like 1/5 of this time scale is a good place to start
 - How do we choose m ?
 - It should be larger than the fractal dimension (but not too much larger!). Ideally, there should not be large changes to the dimension when m is increased
 - Finding a good value again typically requires some manual adjustment/iteration
 - Lab 9 considers the first of these questions
 - Time delays are often the best approach for results from spatiotemporal data (e.g. solutions of PDEs). This is again because of the requirement that “coordinates” in our m -dimensional vector should be uncorrelated, and (sufficiently long) time-delays help ensure this.

Transitions between states



- At $r = 3.5$, there are *period-2 oscillations* where: $x_{i+4} = x_i$
- At $r = 3.9$, we have chaos
- What other states are there?
- For what values of r are there transitions from one state to another??

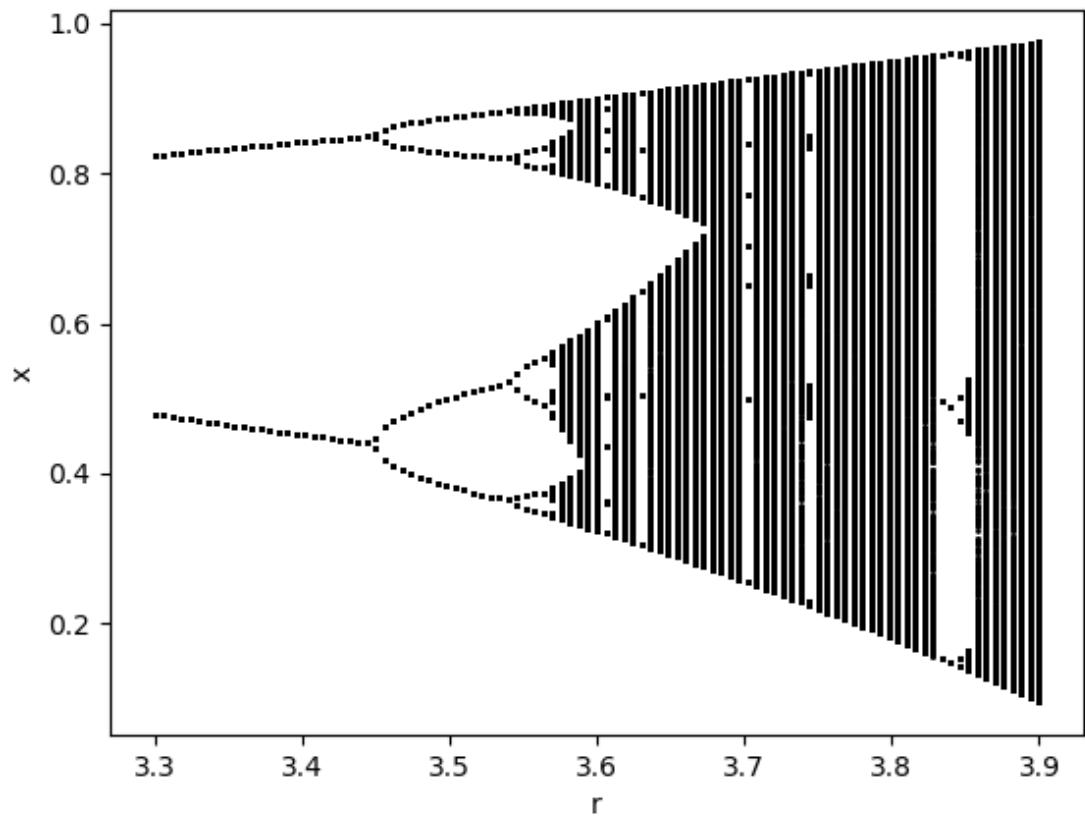
Simple approach: compute “all” values of x visited for a range of r over a sufficiently large number of iterations

Orbit diagram

- We loop over a range of r , compute solutions to the logistic map as before (discarding the 1st half of the data), and plotting all x for each r

```
for r in rarray:  
    #Setup  
    x = np.zeros(N+1)  
    x[0]=x0  
    #Main calculation  
    for i in range(N):  
        x[i+1] = r*x[i]*(1-x[i])  
    x = x[N//2:]  
    rplot = np.ones_like(x)*r  
    plt.plot(rplot,x,'k.',markersize=2)
```

-
- The resulting figure is shown →
 - This provides a concise global view of dynamics
 - What is it showing?:
 - $r < 3.45$: period-1 oscillation
 - $3.45 < r < 3.545$: period-2
 - Then period 4, period 8, ...
 - Until at around $r = 3.5699$ we have aperiodic, chaotic dynamics
 - But for larger r there are periodic “windows”!



Differential equations

- A range of difference equations (maps) show similar behavior
 - The map should look like an “upside-down U” (or V) in some sense
- What about differential equations?
 - We have seen that ideas on fractal dimension can carry over
 - Are orbit diagrams similar in any sense?
 - Can nonlinear ODEs be viewed as maps in some way?

Rössler system

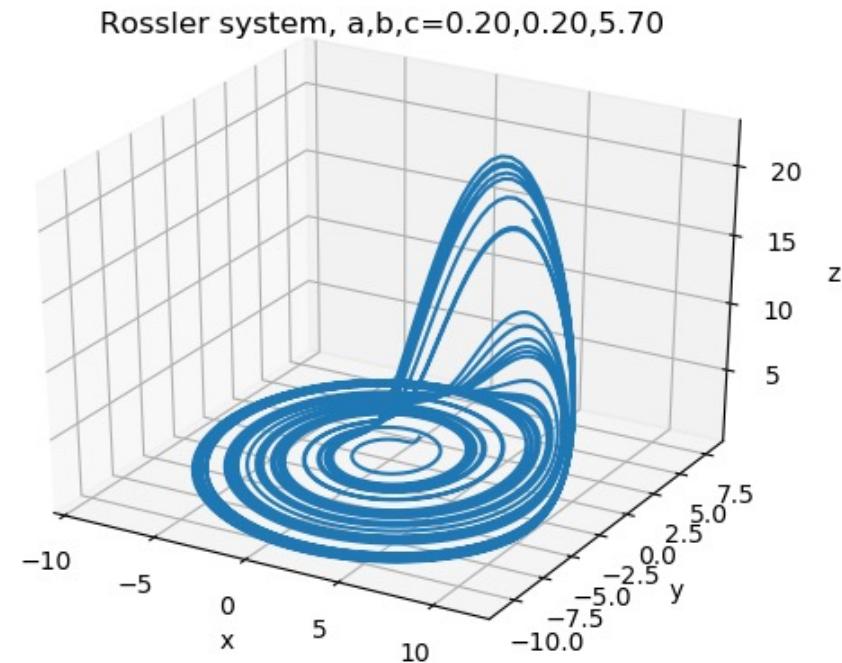
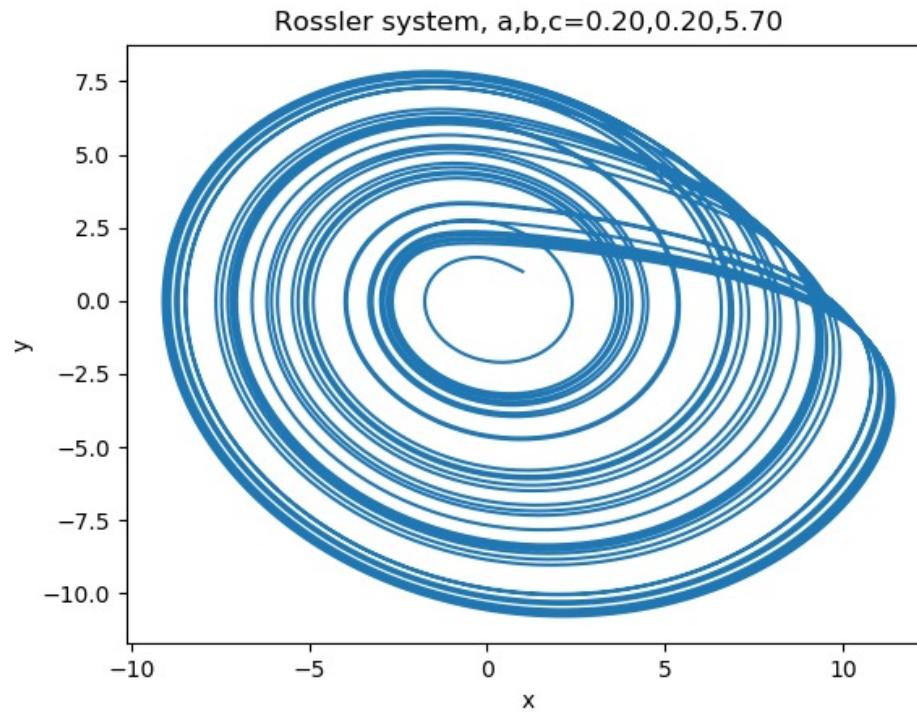
$$\frac{dx}{dt} = -y - z$$

$$\frac{dy}{dt} = x + ay$$

$$\frac{dz}{dt} = b + z(x - c)$$

- The Rössler system is the “simplest” system of ODEs which produce chaos
- a , b , and c are parameters. In certain parameter ranges, chaotic dynamics are generated
- The development of this system was motivated by the Lorenz system and consideration of chaotic maps
- As with the Lorenz system, we can integrate these equations using `solve_ivp`...

-
- Trajectories in the $x - y$ phase plane and the “full” 3-D trajectories are shown below
 - Similar to the Lorenz system, trajectories avoid crossing by moving up in z and the dynamics are aperiodic
 - We can again consider transitions between states, this time varying c



- Basic solution of the Rossler system is straightforward:

```
f = solve_ivp(RHS, [0,T], f0, t_eval=t, args=(a,b,c), method='BDF')
```

- Discard influence of initial condition:

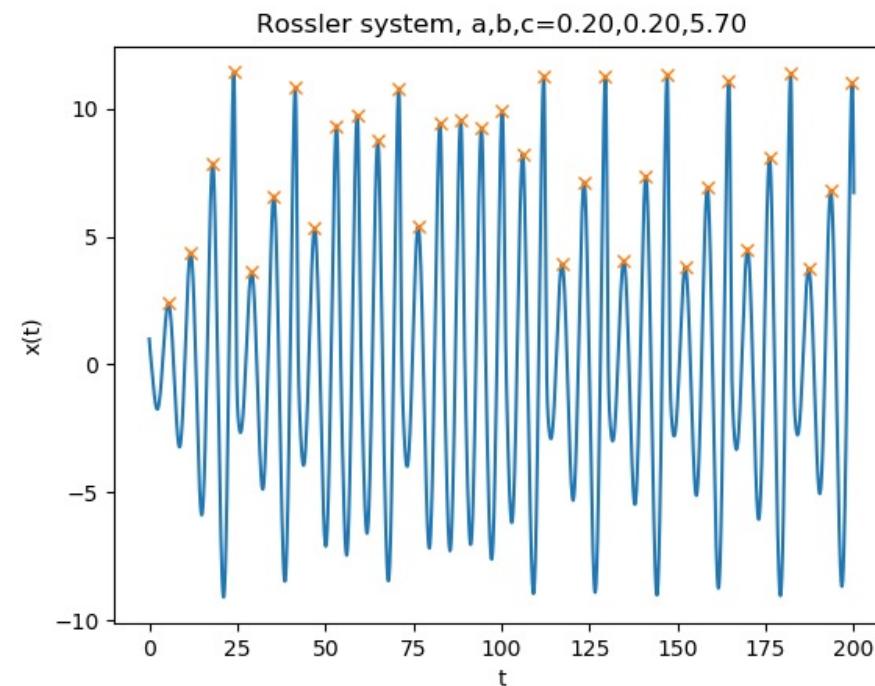
```
f = f[iskip:,:]
```

- Loop over a range of c, but now, we extract local maxima of x :

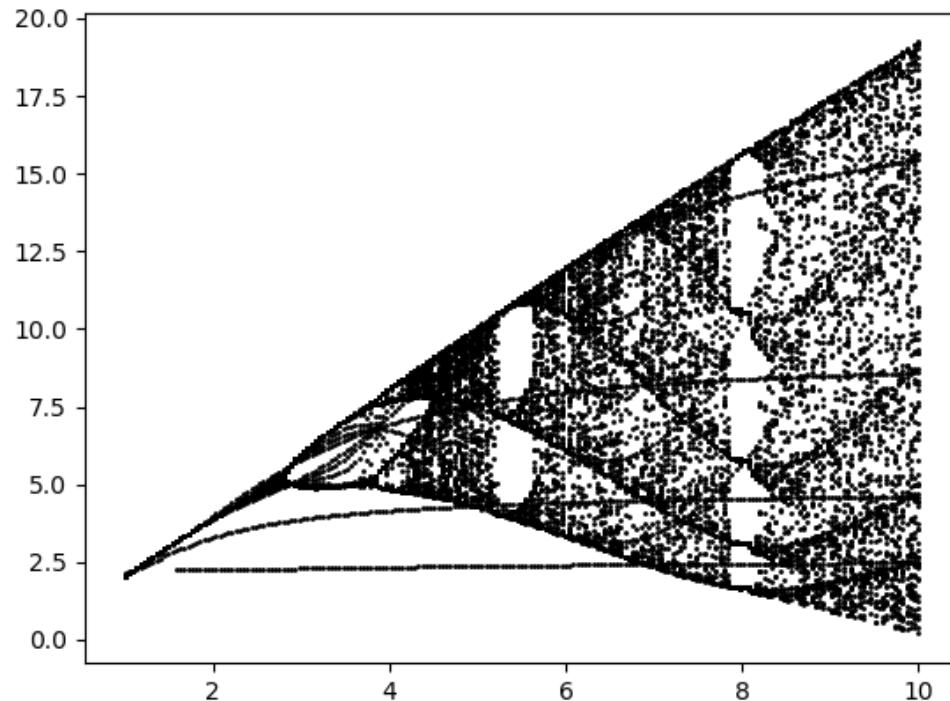
```
dx = np.diff(x)
d2x = dx[:-1]*dx[1:]
ind = np.argwhere(d2x<0) #locations of extrema
```

- $\text{ind}+1$ contains locations of both maxima and minima, and we now need to separate even and odd indices:

```
plot(t[ind[1::2]+1],x[ind[1::2]+1],'x')
```



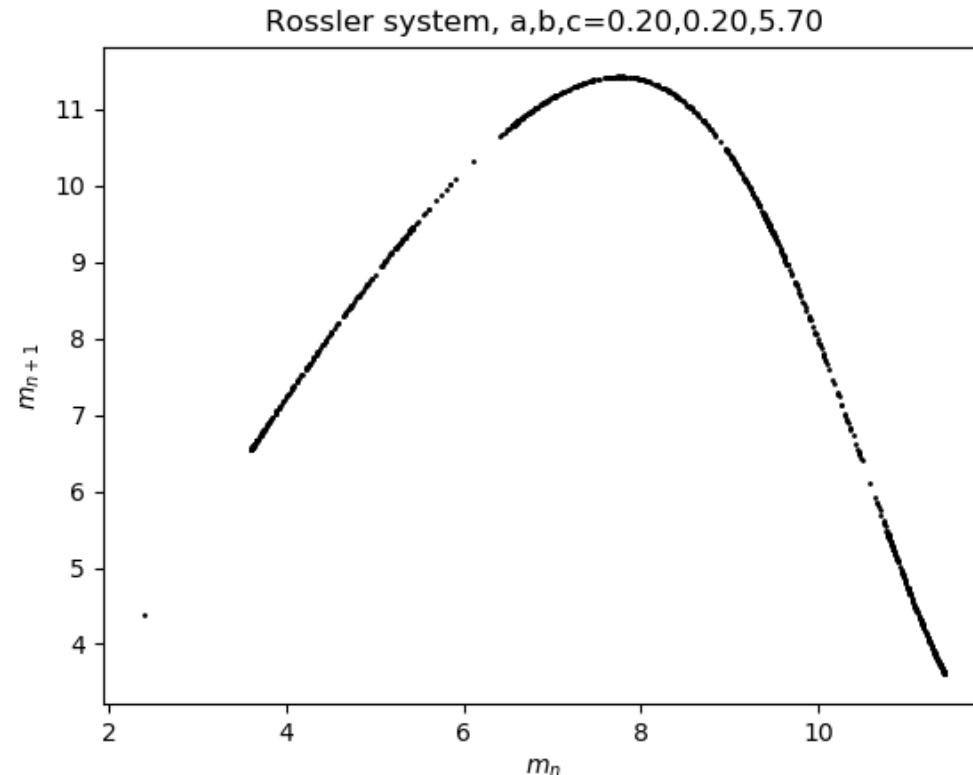
-
- The resulting figure is shown →
 - Again, we have:
 - a sequence of period doublings leading to chaos
 - with subsequent periodic windows
 - This figure can be created with ~20 lines of code in 2-3 minutes
 - However, it can be difficult/expensive to construct for more complex systems



1-D maps

- Given the qualitative similarities in the orbit diagrams, is there an underlying map that can be extracted for the Rossler system?
- This idea comes from Lorenz:
 - Let f_n represent the n th maximum of $x(t)$ (already constructed for orbit diagram)
 - Then plot f_{n+1} vs $f_n \rightarrow$

- We expect *unimodal maps* to be “hiding” behind chaotic dynamics
 - There is a V-map for Lorenz eqns.
- Similar maps have been constructed from measurements as well!



Comments

- Next lecture, we will consider sensitivity to initial conditions, a particularly important aspect of chaos, however you will not be required to carry out the corresponding computations
- Generally, I have ignored randomness when discussing time series – this is an unusual approach!
 - Typically, a set of measurements is affected to some degree by noise or randomness – sometimes the underlying system is fundamentally random -- and a probabilistic framework is natural
 - Often, the level of noise or randomness is small compared to deterministic trends, and then the methods described in this lecture can be applied directly (and they can also be used when the data is noisy in some cases with some additional data processing).

Lecture 18

Chaos and sensitivity to initial conditions
Discrete autocorrelation
Computational libraries

Chaotic dynamics

- Our discussion of chaos last lecture focused on aperiodicity and fractal dimension
 - The solution of a chaotic system corresponds to an object called a “strange attractor” (discarding the adjustment from the initial condition)
 - This object has a non-integer fractal dimension – for continuous systems, this dimension will be larger than two
 - Solution trajectories also never cross each other, so phase plots show increasing densities of points as the solution progresses in time (assuming that the solution remains bounded as $t \rightarrow \infty$)
- A third feature of chaotic dynamics is *sensitive dependence on initial conditions* which we will now briefly discuss
- Note: we are assuming that the system is deterministic and that solutions remain bounded

Lyapunov exponents

- This sensitivity to initial conditions is characterized by *Lyapunov exponents*
 - Given two initial conditions, (x_0, y_0, z_0) and $(x_0 + \epsilon, y_0, z_0)$, consider the distance between the subsequent solutions
 - However in a chaotic system, the distance between the solutions will increase exponentially
 - The rate of exponential growth is the leading Lyapunov exponent.

-
- A precise computation requires consideration of several initial conditions
 - We will just work through a simple example using the Lorenz system:

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = rx - y - xz$$

$$\frac{dz}{dt} = xy - bz$$

- We will first run a simulation from an arbitrary initial condition up to $t = 10$ and take solution at $t = 10$ as initial condition, (x_0, y_0, z_0)
- Then:
 - Run simulation for a “long” time with this initial condition
 - Run simulation for a “long” time with perturbed initial condition and compute distance between the two solutions

-
- Run simulation from arbitrary initial condition up to $t = 10$ and take solution at $t = 10$ as initial condition:

```
T = 10
Nt = 101
t = np.linspace(0,T,Nt+1)
f0 = [1,1,1] #arbitrary initial condition
f = solve_ivp(RHS,f0,[t[0],t[-1]],args=(r,s,b),t_eval=t)
```

- Next, integrate forward using the solution at $t = 10$ as an initial condition:

```
T=40
Nt=2000
t = np.linspace(0,T,Nt+1)
f0new = f[-1,:] #initial condition taken from solution above
f = odeint(RHS,f0new,t,args=(r,s,b))
x,y,z = f[:,0],f[:,1],f[:,2]
```

-
- Repeat previous step using “perturbed” initial condition:

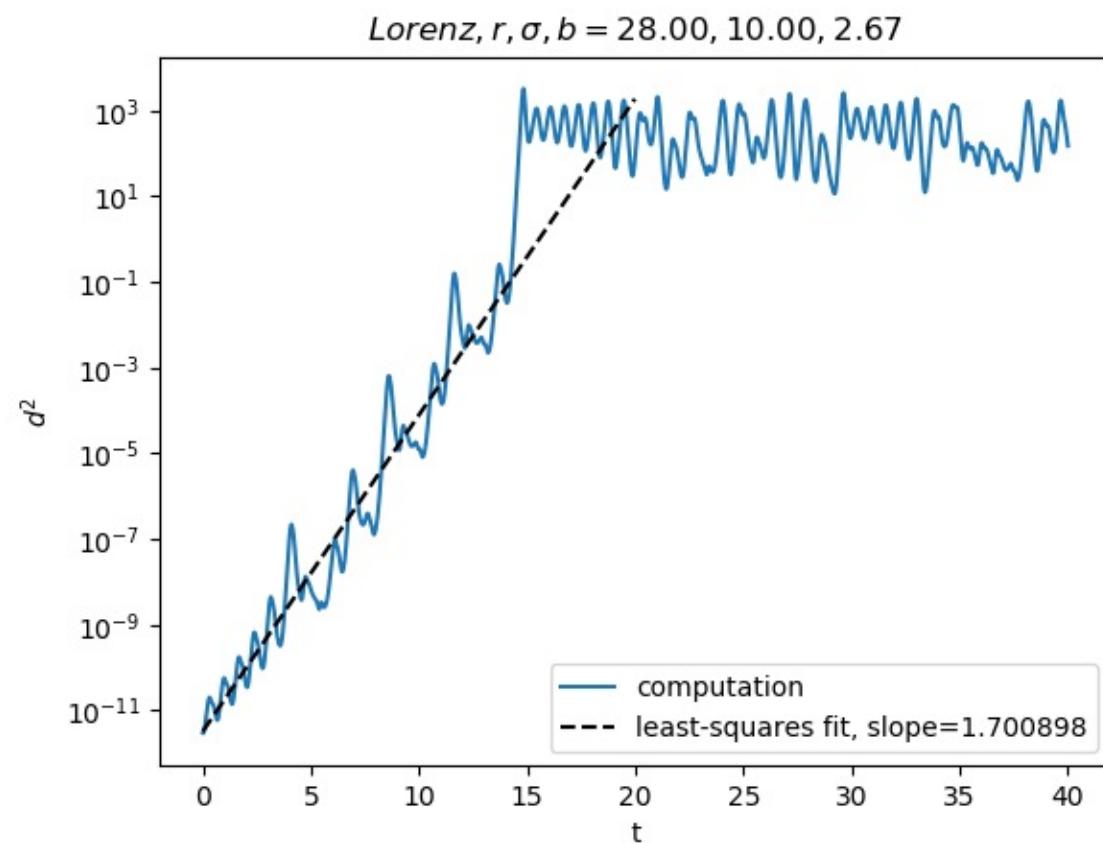
```
f0new[0]=f0new[0]+1e-6 #perturbed initial  
condition  
f2 = odeint(RHS,f0new,t,args=(r,s,b))  
x2,y2,z2 = f2[:,0],f2[:,1],f2[:,2]
```

- And compute distance-squared between the two solutions:

$$d2 = (x2-x)^{**2} + (y2-y)^{**2} + (z2-z)^{**2}$$

- And now we need to plot $d(t)^2$ and analyze the trend

- Note that the distance initially grows exponentially →
- From 3-D phase plots, we know that there will be an upper bound. I.e. we can think of there being an effective diameter for the attractor
- We can estimate the leading Lyapunov exponent using a least-squares fit



The exponent is half of the slope (because distance squared is plotted)

More careful calculations estimate the exponent as ≈ 0.9 for the Lorenz eqns.

-
- Why is this important?
 - It has important consequences for forecasting
 - Small initial errors can lead to substantial errors in predictions at later times
 - This is (partly) why weather forecasts more than 10 days ahead are not very accurate
 - This behavior is often misinterpreted in terms of “butterfly effect”
 - However, the probability of a butterfly initiating a storm is infinitesimally small!

Autocorrelation

- A few final comments on data analysis (for statistically stationary data)
 - It is generally important to be aware of “correlations”, so I will provide a brief informal discussion here – you don’t need to do any correlation calculations!

- For a real-valued function, $x(t)$, the *autocorrelation function* is,

$$R_{xx}(s) = \int_{-\infty}^{\infty} x(t)x(t-s)dt$$

- For periodic functions, the autocorrelation will be relatively large when s matches the period of the function
- The power spectral density function can be expressed as the Fourier transform of the autocorrelation:

$$S_{xx}(f) = \int_{-\infty}^{\infty} R_{xx}(s)\exp(-i2\pi fs)ds$$

-
- The discrete analogues for these functions are what are relevant for us
 - Say we now have n data points: $x(t_j), t_j = j\Delta t, j = 0, 1, 2, \dots, n - 1$
 - Then the power spectral density is approximated as,

$$S_{xx}(f) \approx \Delta t^2 / \tau \left| \sum_{j=0}^{n-1} x_j \exp(-i2\pi f t_j) \right|^2$$

which we can recognize from lecture 15.

- The discrete autocorrelation is,

$$R_{xx}(s) = \sum_{j=0}^{n-1} x_j x_{j-s}, s = 0, 1, 2, \dots,$$

- What do we do when data is “unavailable” (e.g. when $j - s < 0$)?
 - One common approach is to set those values of x_{j-s} to zero

-
- We know how to compute the (discrete) power spectral density. How do we compute the discrete autocorrelation?
 - We can again rely on `scipy.signal` which has the function, `fftconvolve`
 - For large n , the FFT provides the most efficient method for computing discrete convolutions!
 - A simple example, $x(t) = \sin(2t)$, $0 \leq t < 2\pi$

```
In [85]: t = np.linspace(0,2*np.pi,101)[-1]
```

```
In [86]: x = np.sin(2*t)
```

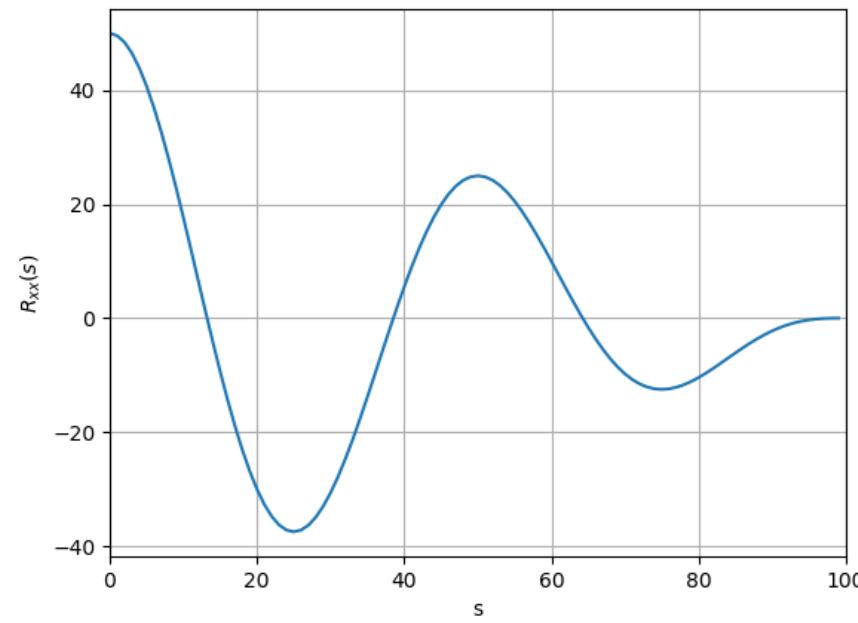
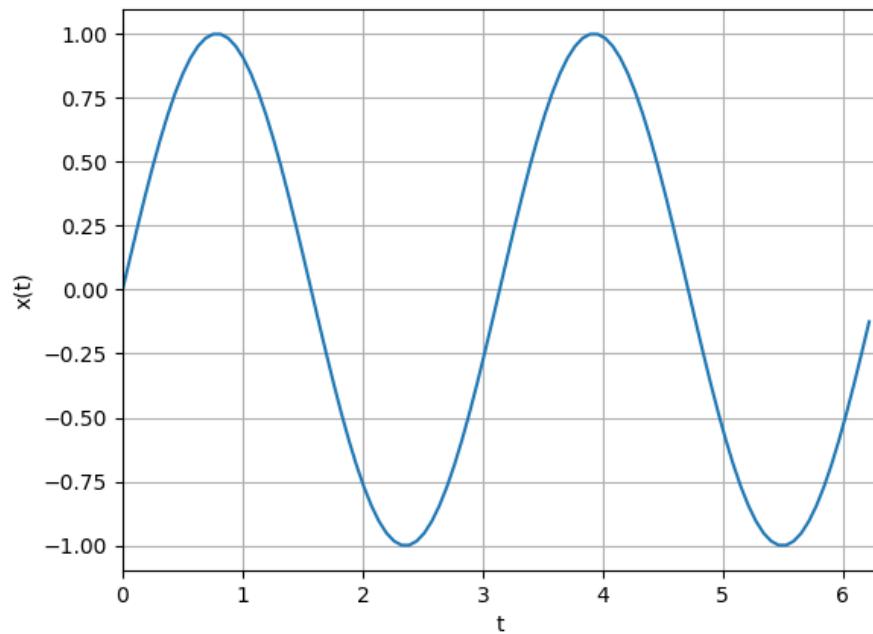
```
In [87]: Rxx = sig.fftconvolve(x,x[::-1])
```

```
In [88]: x.shape  
Out[88]: (100,)
```

```
In [89]: Rxx.shape  
Out[89]: (199,)
```

```
In [90]: Rxx = Rxx[99:]
```

-
- The figure on the left is the sinusoidal “data” and the figure on the right is the autocorrelation
 - The second peak in the autocorrelation occurs at $s = 50$. Here $\Delta t = \frac{\pi}{50}$ so this peak corresponds to a time-lag of π which is the period of the function



Data analysis

- There are many, many other important topics in data analysis and time-series analysis (that's why we have a full module on the subject!)
- See the documentation of `scipy.signal` to get a sense of what you can do computationally with Python

High performance computing

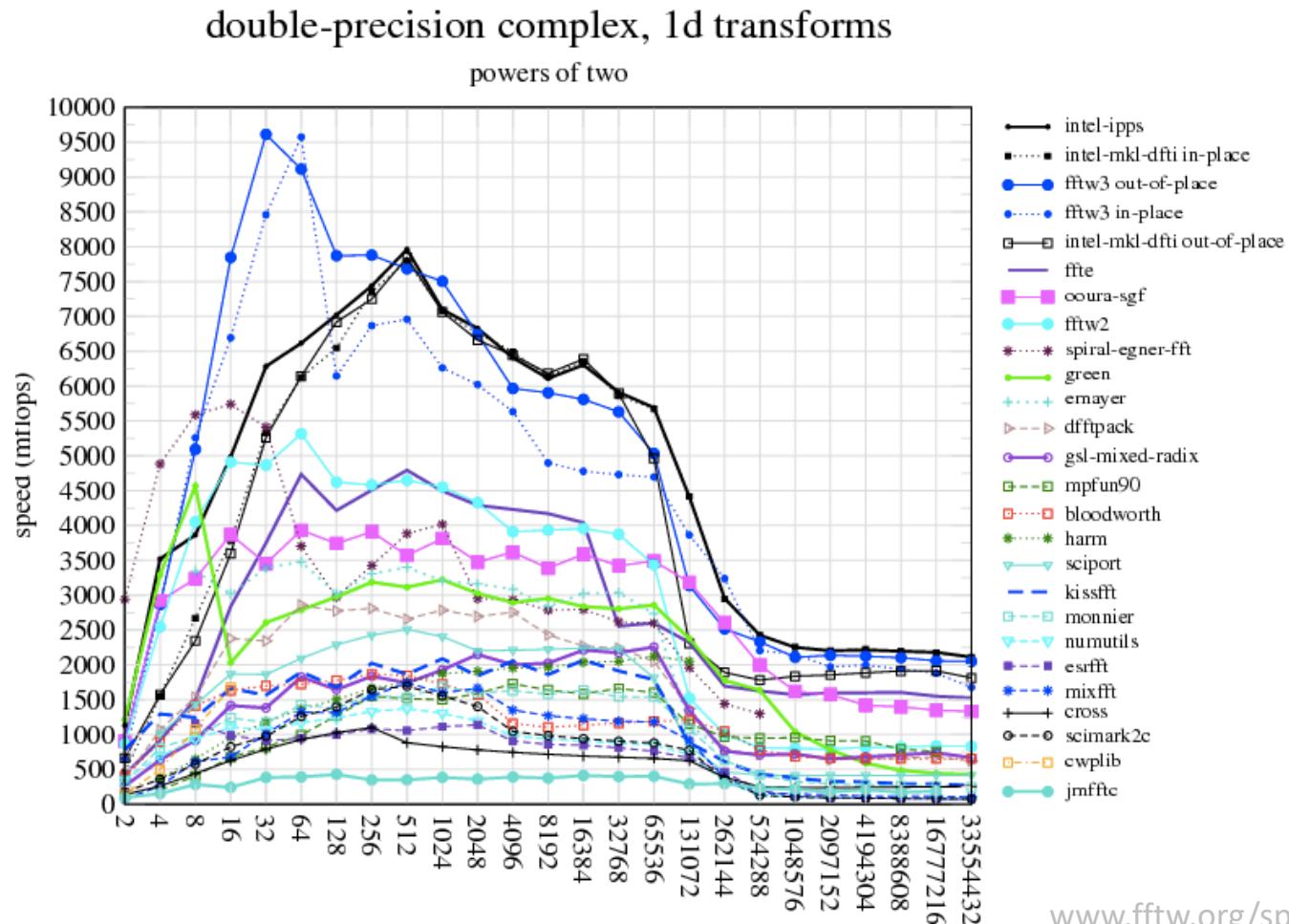
- We have primarily focused on “laptop-friendly” computations
 - But what about larger, more-expensive problems?
 - A few motivating examples are here:
https://www.youtube.com/watch?v=iKR_L0xswdw



Libraries

- Avoid writing your own code whenever possible! (except on your assignments)
- There are many well-established libraries for scientific computing that are freely available.
- We have seen a several examples (np.linalg, scipy.signal, scipy.fft, networkx, ...)
 - Save time: don't have to write/test own code
 - Standard libraries have been extensively tested
 - Libraries often optimized to run fast, difficult to do better
- However there can be some subtleties...

- Sometimes there are many libraries for the same task
- Example: “Right” fast Fourier transform package depends on: compiler, architecture, programmer’s background



- `scipy.fft` uses `fftpack` or something similar (I think)
- `fftw3` is the “fastest Fourier transform in the West” and is available via the `pyfftw` module

Lecture 19

High performance computing
Memory and performance
Compiling Python code
Parallel computing with Python

High performance computing

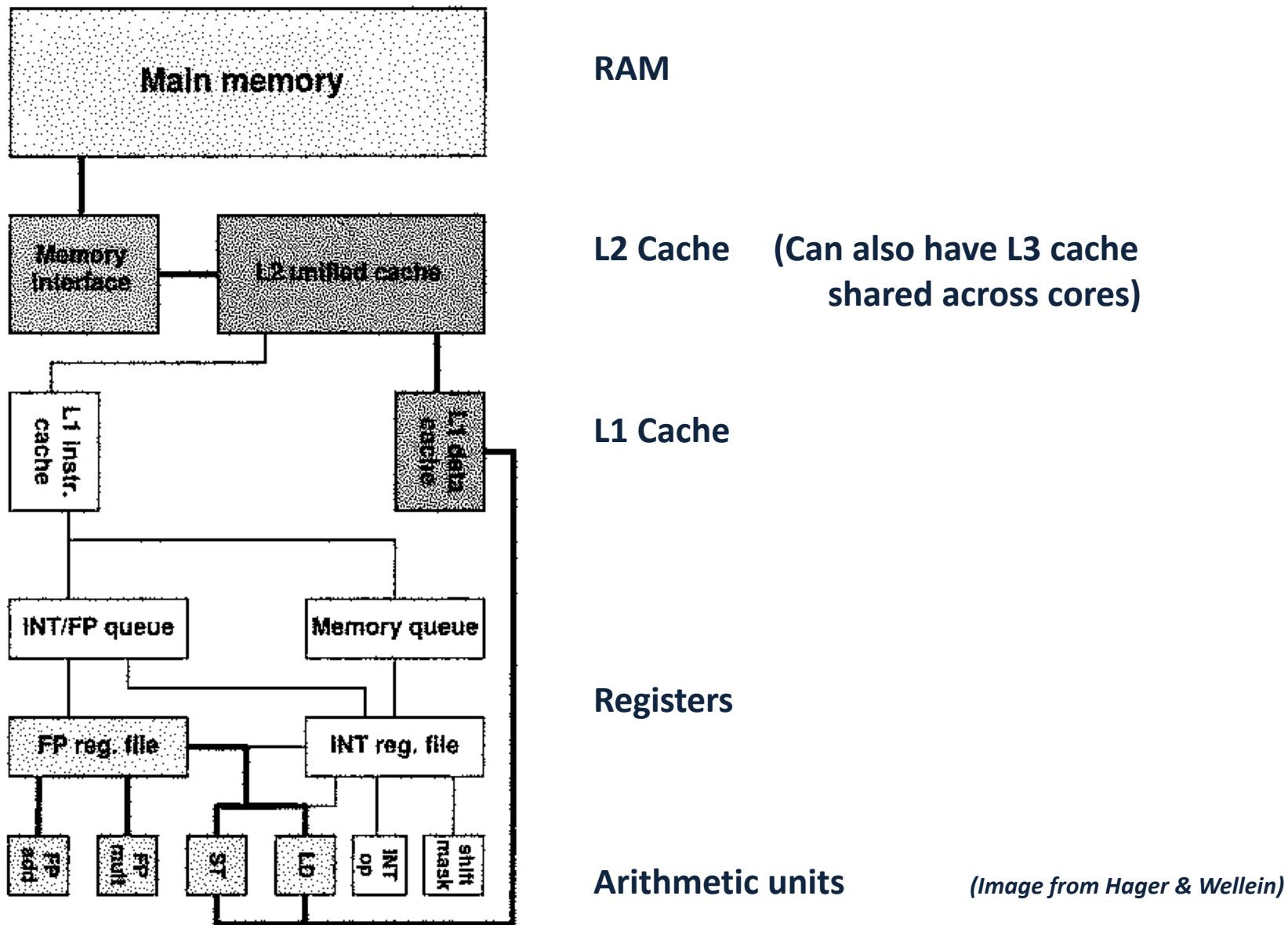
- Following on from last lecture, we will continue to consider what happens/changes when we move to larger, more-expensive problems?



Memory and performance

- Parallel computing is typically required if a computation is either (a) too slow or (b) too big for a single computer
- The second point is related to the amount of memory available, and we will now look at how computer memory is organized and how it affects performance

Schematic of single core processor



-
- CPUs typically have clock speeds of \approx 1-2 GHz
 - Typically, CPUs can produce 2-4 double-precision floating-point results per cycle
 - So, *peak performance is* \approx 4e9 FLOPS, or 4 gigaflops
 - Performance often limited by movement of data in and out of the arithmetic units
 - Need to consider memory hierarchy

-
- Generally, the closer the memory is to the arithmetic units, the faster and smaller the memory
 - Hard drive: very large (≈ 500 gb), *very* slow
 - Main memory (RAM): large (≈ 2 gb), sort-of fast (≈ 1 GHz)
 - All computations, applications, etc... should fit in main memory
 - For my laptop (core i5 processor):
 - L3 Cache \rightarrow 3mb (shared by two cores)
 - $(3e6 \text{ bytes}/8) = 375,000$ double precision numbers $\approx 600 \times 600$ matrix
 - L2 Cache \rightarrow 256 kb (per core)
 - $256000/8 = 32000$ double precision numbers $\approx 180 \times 180$ matrix
 - L1 Cache \rightarrow 64 kb (per core), half for data, half for instructions
 - 4000 double precision numbers

Memory access: data can be moved from registers to arithmetic units once each clock cycle:

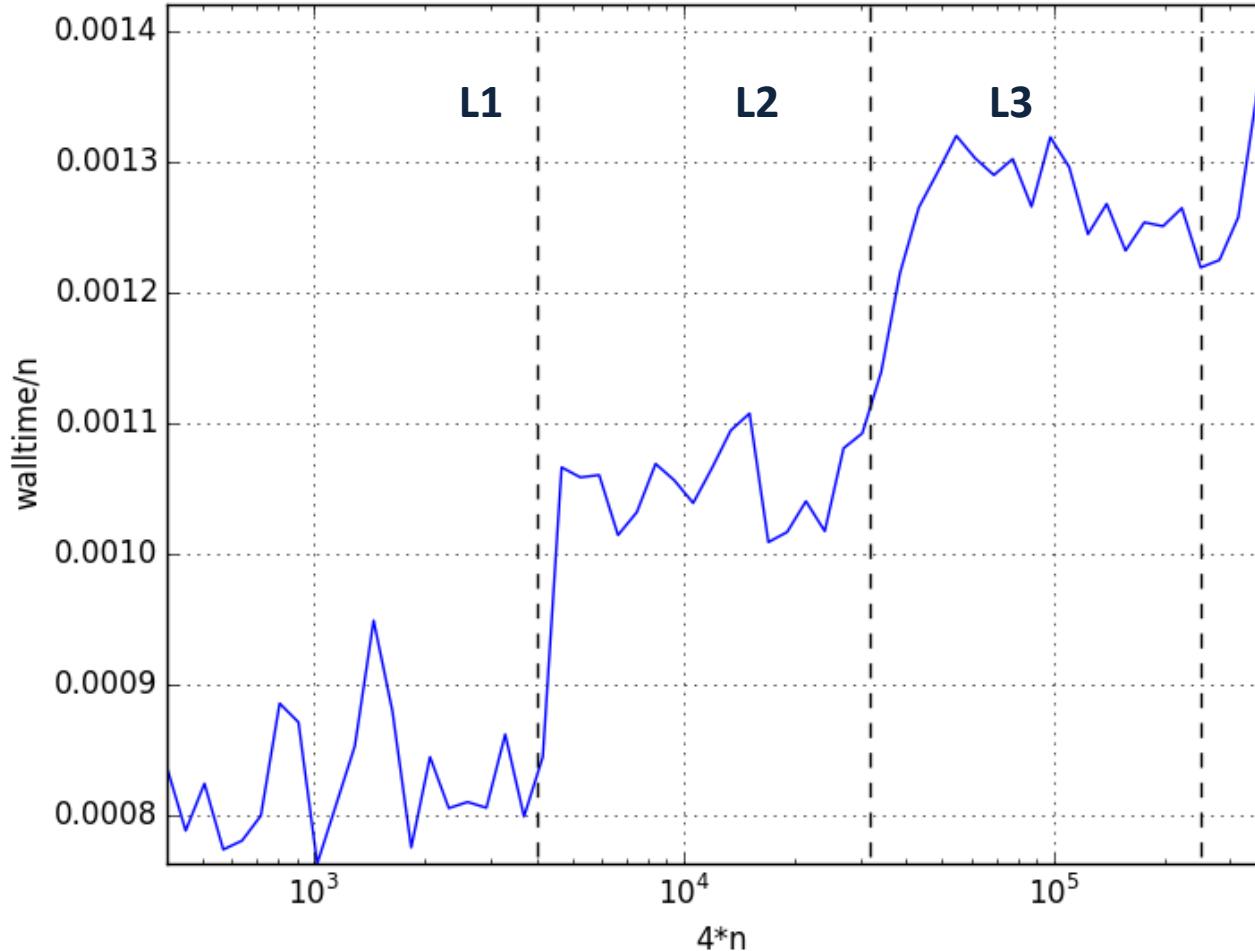
- For my laptop (core i5 processor):
 - Main memory → ~ 800 clock cycles
 - L2 Cache → ~ 11 clock cycles
 - L1 Cache → ~ 4 clock cycles

-
- Consider the following illustrative example
 - Construct three n -element arrays: b,c,d
 - Within a loop, compute $a = b + c * d$
 - Collect timing information as n varies
 - *triad.f90*:

```
real(kind=8), dimension(n) :: a,b,c,d

call system_clock(start)
do j1 = 1,1000000
    do i1 = 1,n
        a(i1) = b(i1) + c(i1) * d(i1)
    end do
    if (a(2) < 0) call dummy(a,b,c,d)
end do
call system_clock(stop,clockrate)
```

Results:



- Vertical lines indicate cache sizes
- Clear performance loss when a cache level becomes full

Temporal locality

Due to the influence of cache size, it is important to think about where data is stored:

- If possible, data in cache should be re-used as much as possible (temporal locality)
 - “cache hit”: needed data is found in cache,
 - “cache miss”: data is not in (nearest cache)
 - old data needs to be moved out, new data moved in
 - data is moved in cache lines (typically 8 or 16 floats)
- Re-using data in cache reduces cache misses and associated performance loss.

Spatial locality

- Since data is passed in “lines,” there can be performance gain when using data which is adjacent in memory
- There can also be a penalty when using data that is scattered in memory
- Consider how matrices are stored in memory

$$A = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 16 \\ 2 & 6 & 10 & 20 \\ 3 & 7 & 11 & 24 \end{bmatrix}$$

Python (and c):

- 0,4,8,12 occupy consecutive locations in memory
- “row-major” ordering
- When using np.array, can force ‘Fortran ordering’

-
- Since data is passed in “lines,” there can be performance gain when using data which is adjacent in memory
 - There can also be a penalty when using data that is scattered in memory
 - Consider how matrices are stored in memory

$$A = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 16 \\ 2 & 6 & 10 & 20 \\ 3 & 7 & 11 & 24 \end{bmatrix}$$

Fortran:

- 0,1,2,3 occupy consecutive locations in memory
- “column-major” ordering

-
- Really only important for very large matrices:
 - Create 20000 x 20000 random matrix
 - Compute statistics by manually looping along rows and columns
 - Use timer to calculate wall time

```
for i in range(0,n):
    av += np.average(H[i,:])
    sum += np.sum(H[i,:])
    var += np.var(H[i,:])
    std += np.std(H[i,:])
```

time: 4.8690969944

```
for i in range(0,n):
    av += np.average(H[:,i])
    sum += np.sum(H[:,i])
    var += np.var(H[:,i])
    std += np.std(H[:,i])
```

time: 132.157668114

- Better to first compute transpose, then loop across a row.

Compiling Python code

- Python is an *interpreted* language
 - The Python interpreter goes through code line-by-line
- Compiled languages use a compiler to optimize blocks of code
 - Code development is slower, but execution time can be much shorter (especially if code contains loops)
 - Python built-in functions usually utilize compiled routines
- Numba allows Python code to be compiled
 - See: http://numba.pydata.org/numba-doc/0.12.2/tutorial_firststeps.html

-
- A simple example – insertion sort:

```
def isort(L):
    """Simple insertion sort code
    """
    for j in range(1, len(L)):
        #compare L[j] with L[i]
        i = j-1
        key = L[j]

        while i>=0:

            if key<L[i]: #shift from i to i-1
                i=i-1
                if i<0:
                    L[i+2:j+1] = L[i+1:j]
                    L[i+1] = key

            else: #insert key at i+1
                L[i+2:j+1] = L[i+1:j]
                L[i+1] = key
                i = -1

        #print("j,L",j,L)

    return L
```

Reminder: with insertion sort, we ensure that after the kth iteration, that the first $k+1$ elements in an input array are sorted

-
- We have a loop with `len(L)-1` iterations, and a compiler could optimize the execution of this loop
 - Compiling this code with numba is fairly straightforward:

```
In [2]: import numba
```

```
In [3]: from isort_numba import isort
```

```
In [4]: isort_jit = numba.jit("void(i4[:])")(isort)
```

- The first argument to `.jit` tells the compiler to expect an array of 4-byte integers as input into the function `isort`

-
- Let's now run `isort` and `isort_jit` and compare their performance:
 - - `In [9]: L = np.random.randint(0,10000,5000)`
 - `In [14]: timeit out1=isort(L)`
6.94 ms ± 165 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
 - `In [15]: timeit out2=isort_jit(L)`
210 µs ± 11.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
 - `In [16]: out1[:5]`
`Out[16]: array([1, 5, 5, 8, 9])`
 - `In [17]: out2[:5]`
`Out[17]: array([1, 5, 5, 8, 9])`
 - The compiled version is about 30 times faster!

Parallel computing with Python

- Our computers have multiple cores. Can we distribute parts of a calculation to these cores and run the computations for these parts in parallel?
- In many cases, numpy and scipy built-in functions already do this (it can depend on your Python installation)
- Several tools exist for parallelizing Python code
- We will look at the joblib module.
- Say that you have a loop where the iterations are independent of each other. E.g. you don't need to have completed the i th iteration to carry out the $i+1$ th iteration. Then joblib allows you to distribute iterations to the cores on your computer

-
- We'll look at a very simple example – computing the square root of the elements in a 10-element array
 - We know we should just use numpy and avoid a loop, but we will parallelize a loop-based implementation as an illustrative example
 - The serial version of the code is:

```
>>> from math import sqrt  
>>> [sqrt(i ** 2) for i in range(10)]  
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Then we have to specify the number of cores to distribute these 10 iterations to using,
`Parallel(n_jobs=num_cores)`

- Parallelizing the code to run on 2 cores:
- ```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```
- And then for sufficiently large n, we expect the parallel version to be faster (theoretically, it could be twice as fast).

- 
- We can use these ideas about cache to improve/optimize code
    - With compiled languages, the compiler can do much of the optimization for you
    - Working with large matrices in interpreted languages requires greater care
    - Most important point: first develop a code that works, then optimize it.