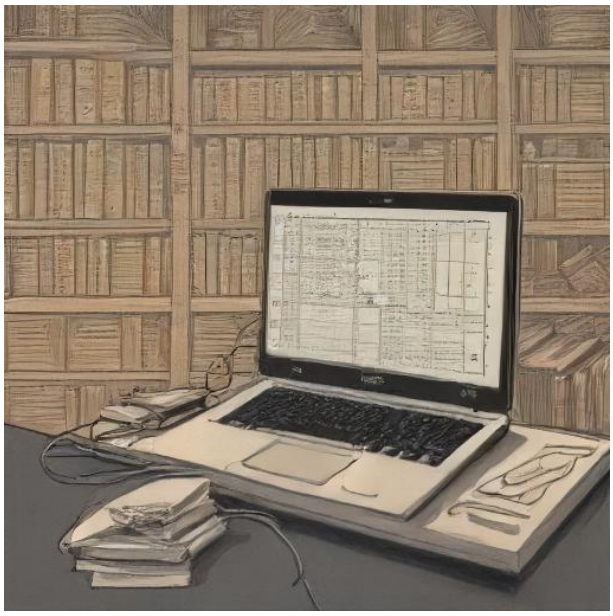
Scientific Computation Lecture Notes



Prasun K. Ray
Department of Mathematics
Imperial College London

Contents

Lecture 1

Introduction

Let's start with a question: what is *scientific computation*? There is no single, standard definition that I'm aware of. However, it is a standard view that scientific computation is fundamentally an interdisciplinary subject which borrows ideas and methods from a range of fields including (but not limited to) computer science, mathematical science, data science, and numerical analysis. Ultimately these ideas and methods are used to produce code or software packages which, in turn, are used to obtain insight into interesting and important scientific problems. Three examples of such problems are 1) understanding brain dynamics, 2) numerical weather prediction, and 3) designing self-driving cars. The human brain contains a network with more than 80 billion neurons, so computers are definitely needed when analyzing how the brain works. Given data about the structure of the brain, we can use tools from scientific computing to partition the brain into components that correspond to distinct functions. We can also use computers to compute numerical solutions to systems of differential equations that model how electrical signals propagate through the brain. Numerical weather prediction relies on the numerical solutions of systems of nonlinear partial differential equations (the Navier-Stokes equations). These equations tell us how the dynamics of the atmosphere and ocean evolve in space and time and give useful information about daily temperature, wind speed, and precipitation. The accuracy of weather forecasts has increased substantially over the last few decades (Figure 1.1). This increase is due in large part to 1) large increases in computational power, 2) increasing availability of data from satellites and weather stations, and 3) improved algorithms which take advantage of 1) and 2). The technology behind self-driving cars has also rapidly evolved in recent years driven by advances in machine learning which build on earlier developments in scientific computing. Specifically, these cars use rapidly spinning cameras to constantly feed images into a computer which must then process the images and decide how to navigate the car. A black and white image can be represented as a matrix where each element of the matrix corresponds to a pixel brightness. Processing images then naturally requires the use of numerical linear algebra and optimization – two areas closely related to both machine learning and scientific computing.

In this class, we will not (quite) learn how to design self-driving cars or predict the weather. The aim instead is to provide a foundation which will allow you to move towards

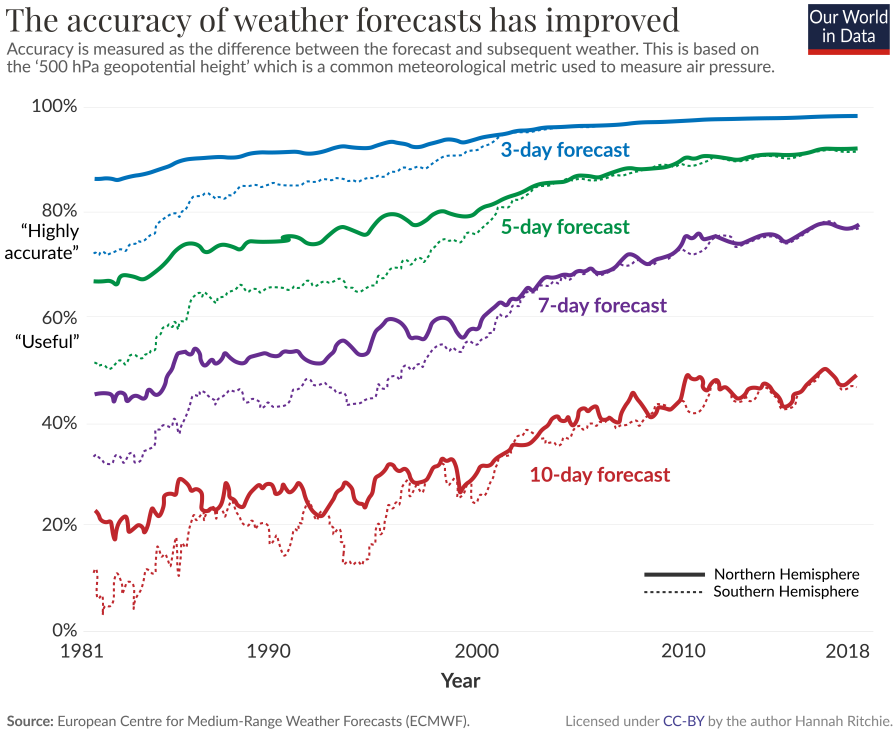


Figure 1.1: Accuracy of weather forecasts since 1981[6]

these kinds of problems. An outline of what we *will* cover is shown below.

Lectures 1-5: Searching and sorting (plus a little DNA)
Lectures 6-8: Complex networks and graph search
Lectures 9-11: Simulating initial value problems (including stochastic differential equations)
Lectures 12-14: Matrix computations: linear systems and data science
Lectures 15-19: Linear and nonlinear data analysis
Lecture 20: Moving beyond this module

Two broad objectives for this class are:

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

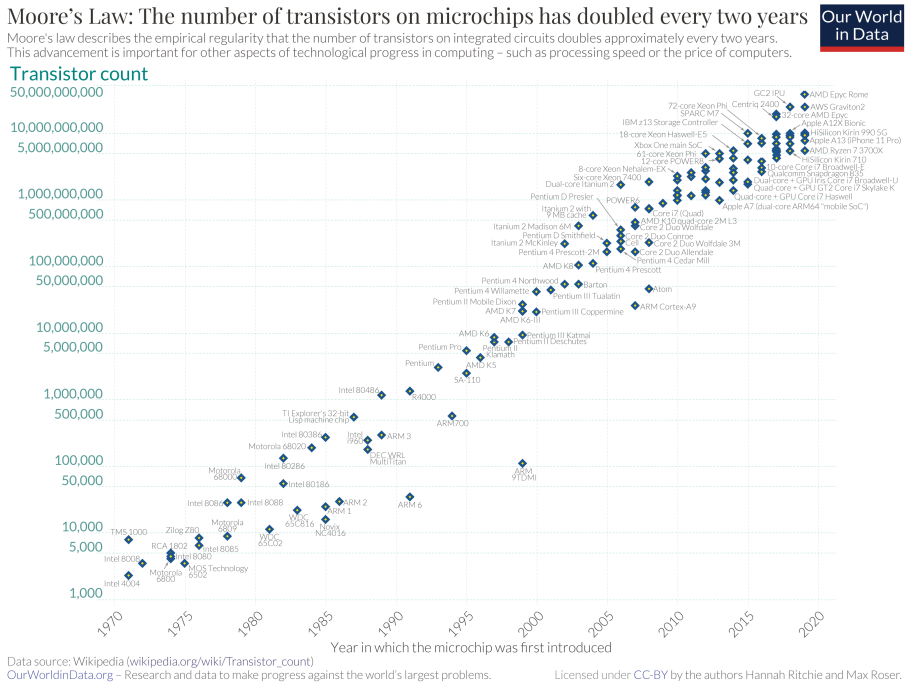


Figure 1.2: Number of transistors on microchips over the years[7]

1.1 Historical perspective

Figure 1.2 shows the increase in the number of transistors on microchips over the last fifty years. We see a linear trend, but note that the vertical axis uses a logarithmic scale. This indicates that that the number of transistors has been increasing at an *exponential* rate, and the figure illustrates Moore's Law which states that the number of transistors on integrated circuits doubles approximately every two years. The number of transistors on a chip correlates with computational power, and we have seen extraordinary increases in computational speed and capacity over the last half-century.

So computational power has been increasing rapidly. What about the efficiency of algorithms used to solve problems on computers? Figure 1.3 shows the speedup of methods used to numerically solve a large sparse system of linear equations corresponding to the discretized 3D Poisson equation. The reference method is banded Gaussian elimination which is a slightly more sophisticated version of the method used to solve linear systems of equations in introductory linear algebra. Banded Gaussian elimination was the method of choice in the 1950s and moving forward 10 years, there is a shift to an iterative method (Gauss-Seidel) which is a thousand times faster. Moving further forwards in time, new, faster algorithms appear until we reach the multigrid method which is seven orders of magnitude faster than Gaussian elimination. This figure is an illustration of a broader trend. For many important problems, algorithmic efficiency has increased substantially in

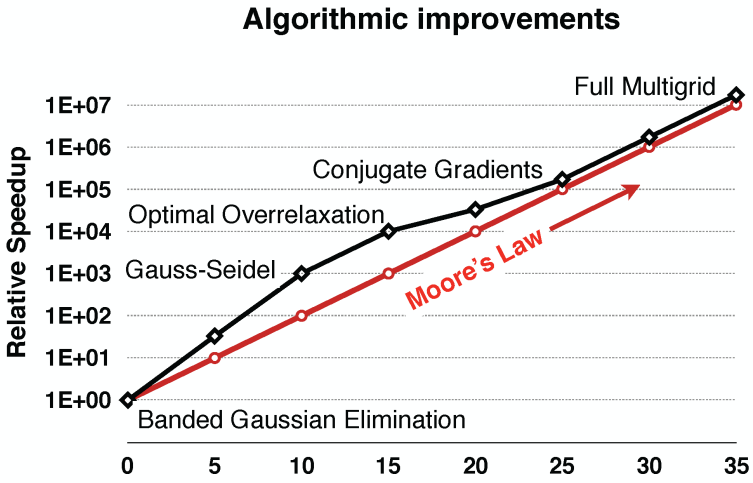


Figure 1.3: Speedup of algorithms for solving large sparse linear systems[8].

parallel with improvements in computational power.

1.2 Python

We will be using Python throughout this course, and an important question to consider now is: *What are the main advantages and disadvantages of using Python for scientific computing?*

Here are a few of the principal 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

And a few important 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

While Python is far from perfect, it is a good place to start. This class assumes either:

- Familiarity with the basics Python as covered in an introductory undergraduate class:

  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 after this lecture! 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 for this class and other projects (Numpy, Scipy, Matplotlib, NetworkX, Pandas, scikit-learn...)

1.3 Searching

The two foundational problems in computer science are searching and sorting. We will begin with searching and the following problem statement:

> Given a sorted list, find a location of a specified item within the list. Return "not found" (or something equivalent) if item is not contained within list.

I am assuming that all elements in the list are comparable, that the $<, >, ==$ operators are meaningful for each distinct pair of elements. Examples of why we might want to solve this problem include finding a name in an alphabetized directory or finding a number in a database. We'll work with, a simple concrete example:

Find "31" in the list below:

2	6	8	23	24	31	32	53	56

What is the simplest way to approach this problem? We can apply *linear search* where we step through the list one element at a time and terminate the search if/when the target item is found. So we would compare 31 with 2 and then 6 and so on until we found the match. This approach can be implemented within a Python function with a few lines of code:

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

When analyzing code or an algorithm, we should consider *correctness* and *cost*. Correctness refers to whether or not we can always expect the method to produce the right answer (given

acceptable input). Cost refers to the number of operations (e.g. additions, multiplications, assignments, comparisons, ...) required by the method, and we usually want to understand how the cost depends on the problem size. The correctness of `linsearch` is straightforward to establish. If the target is in `L`, the loop will find it. If the target is not in `L`, `y==x` will never be satisfied, and -1 will be returned to indicate "not found". What is the cost of linear search for an N-element list? For iterative methods, it is often convenient to first consider the cost per iteration and then determine the number of iterations. Here, the cost depends on where the target is, and we can consider the best, worst, and average cases. In the best case, the target is found with the first comparison, and this is not a particularly useful case to consider. In the worst case, the target is either in the final element of the list or not in the list at all. There will then be N iterations of the for loop. How many operations are there per iteration? For line 3, there are two assignments (for i and y), and one lookup in L. There is also a comparison in line 4, and the total worst-case cost is $4N + 1$ where the $+1$ accounts for the return statement. There's no particular reason to expect the target to be at the end of the list, and on average, we expect $N/2$ iterations and an average cost of $4N/2 + 1 = 2N + 1$ operations. We see that the cost increases linearly with the size of the list. Note that it is not particularly important if you count 3 or 5 operations instead of 4 (if you counted 4000, that would indicate a problem). The key observation is that the functional dependence on the problem size is linear.

Can we do better? We should be able to do better since linear search does not take advantage of the list being sorted. With linear search, there is one comparison each iteration with one element being discarded if a match is not found. Since the list is sorted, we can choose our comparison so that more than one element can be discarded. Specifically, if we compare our target with a median element in the list, we can typically discard about half of the elements in the list. If the target is smaller than the median, we can discard all elements "to the right" of the median which are greater than or equal to the median. Similarly, if the target is larger than the median, we can discard all elements "to the left" which are less than or equal to the median. We will then have a tangibly smaller list to search through. We could apply linear search to this smaller list, but we now know we should instead again compare with a median element. Each iteration, we compare with a median element and discard portions of the list as appropriate until we either find a match or have discarded all elements in the list (i.e. the target is not in the list). Let's apply this to our list of 9 integers above with target, $x = 31$:

1. The middle element is 24 which is smaller than our target (31>24), so we discard the left "half" of the list leaving,

31	32	53	56

2. The new middle element can be either 32 or 53 and the choice is arbitrary. We choose 32 and since 31<32, we now discard 32 and the two elements two its right leaving,

31

3. Now the middle element is the only remaining element. This matches our target, and we conclude the search.

This is an illustration of *binary search*, and a Python implementation is shown below.

```
def bsearch(L,x):
    """find location of x in L, if x is not in L, return -1 """
    #Set initial start and end indices for full list
    istart = 0
    iend = len(L)-1

    #Iterate and contract "active" portion of list
    while istart<=iend: # 1 comparison

        #Set index of middle element
        imid = int(0.5*(istart+iend)) #1 add, 1 mult, 1 round, 1 assignment

        #Compare target w/ middle element, adjust istart or iend as needed
        if x==L[imid]: #1 list lookup, 1 assignment
            return imid
        elif x < L[imid]: #1 list lookup, 1 comparison
            iend = imid-1 #1 subtraction, 1 assignment
        else:
            istart = imid+1

    return -1000
```

Note that portions of the list are not actually being discarded. Instead, `istart` and `iend` are set to keep track of the part of the list that is of interest during a given iteration. I won't provide a detailed argument establishing the correctness of binary search. Instead, I'll simply provide the key observation that can be used to construct such an argument: if the target is within the original list, it will either be the "middle element", or it will be in the sub-list that remains after contraction (convince yourself that a proof of correctness follows). How many operations does binary search require? I count 11 operations per iteration when the target is not found (see the comments in the code). Again, it is not important if your estimate is not exactly the same. There are also four operations above the while loop. How many iterations are there? This depends on if/when the target is found. In the best case, the target is found immediately, and this isn't a helpful case. The worst case is more interesting and more useful. If we start with $N = 8$, then in the worst case, there are iterations for arrays with size, $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, and $2^0 = 1$. What happens if we double the size of the list to $N = 16$? The number of iterations increases by one. So if the list size is $N = 2^p$ where p is a non-negative integer, then the maximum number of iterations is $p + 1 = \log_2 N + 1$. The total worst-case cost is then $11\,(\log_2 N + 1) + constant$ where the constant accounts for operations occurring before the loop and during/after termination of the search. The logarithmic dependence of the cost

on N is key. The worst-case cost for binary search grows more slowly with N than both the average and worst-case costs of linear search, so for large problem sizes, binary search will tend to be more efficient. Note that this general conclusion doesn't rely on N being 2 raised to an integer power, however constructing an estimate for the number of iterations becomes a little more complicated for general N.

A final note: be sure to run and test the linear search and binary search codes provided above. It is also important to carefully read and understand each line of the `bsearch` function.

Lecture 2

Sorting

In order to take advantage of the efficiency of binary search, we need a sorted list. Let's now think about how to produce such a list. Our problem statement:

Given an unsorted list, return a list with the same elements in non-decreasing order.

So given the first list below, we want our algorithm/code to return the second.

23	6	8	32	56	31	2	53	24

2	6	8	23	24	31	32	53	56

2.1 Insertion sort

There are many different sorting algorithms, and we will consider just two. Our first algorithm proceeds as follows:

Step 1: Sort first 2 elements

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

Step i: Sort first $i+1$ elements (knowing that 1^{st} i have been sorted)

For a length-n list, the algorithm terminates after $n-1$ iterations. An example showing the first 6 iterations of the method is shown below. Initially, 6 and 23 are compared and then swapped. Then 8 is compared with 23 and 6 and "placed" between them. During the i^{th} iteration, element $i+1$ is sequentially compared with the i^{th} element, the $i-1^{\text{th}}$ and so on until we know where to place it.

23	6	8	32	56	31	2	53	24	Initial data
6	23	8	32	56	31	2	53	24	Iteration 1
6	8	23	32	56	31	2	53	24	Iteration 2
6	8	23	32	56	31	2	53	24	Iteration 3
6	8	23	32	56	31	2	53	24	Iteration 4
6	8	23	31	32	56	2	53	24	Iteration 5
2	6	8	23	31	32	56	53	24	Iteration 6

This algorithm is known as *insertion sort*, and a Python implementation is shown below.

```
def isort(L):
    """
    Sort list of integers in non-decreasing order
    """

    N = len(L)
    #Iterate i from element 1 to N-1
    for i in range(1,N):
        key = L[i]
        j = i-1
        print("i=",i,"key=",key)
        while j>=0:
            print("j=",j)
            if key>=L[j]:
                ind = j+1
                break
            else:
                ind = j
                j=j-1

        print("ind=",ind)
        L[ind+1:i+1] = L[ind:i]
        L[ind] = key

    return L
```

The i^{th} element of `L` is stored in `key`. This `key` is then compared to the elements in the sorted portion of `L` to determine the index of the location where it should be placed (`ind`). Finally, space is created in `L` for the `key` which is inserted in the correct location (`L[ind]`). Correctness of the algorithm can be established by first showing if the 1^{st} i elements are sorted, then after the i^{th} iteration, the 1^{st} $i+1$ elements are sorted. We will not develop this argument further here. Let's now consider the efficiency of insertion sort. As a shortcut, we will count the number of comparisons per iteration (it is important to be careful that such shortcuts don't neglect important/expensive parts of an algorithm). In the best case, the list is already sorted and $N-1$ comparisons are needed. This isn't a useful case. In

the worst case, the i^{th} iteration requires i comparisons for each $i, i = 1, 2, \ldots, N-1$ and there will then be $\sum_{i=1}^{n-1} i = n(n-1)/2$ comparisons. On average, we can expect around $i/2$ comparisons during the i^{th} iteration, and the total number of comparisons will be approximately $n(n-1)/2$. For general input, we expect the cost to increase quadratically with the problem size, n.

2.2 Merge sort

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 with the following approach:

Step 1: Divide list into left and right halves

Step 2: Sort each half using insertion sort

Step 3: Merge the two halves into single sorted list

For large n, we can approximate the worst-case number of comparisons as $C \approx n^2/2$, so the cost of sorting each half will require about $n^2/8$ comparisons giving a total of $n^2/4$ (in the worst case). The number of comparisons has been reduced, but the viability of this approach depends on the cost of the merging step. This step can be completed in $\mathcal{O}(n)$ operations which means the cost increases linearly with n for large n (a precise definition of $\mathcal{O}(n)$ will be provided in lecture 4). We can expect substantial cost savings for large n due to this linear dependence since we have removed $n^2/4$ comparisons.

2.2.1 Merging

How do we efficiently merge two sorted lists into a single sorted list? Let's look at a specific example:

L:

6	8	23	32

R:

2	24	31	53

Our aim is to merge L and R into a single sorted 8-element list, M. This can be done using an iterative approach where each iteration, an element from L or R is placed in the correct location in M based on a single comparison between the elements in L and R. The first few iterations for our example proceed as follows:

Iteration 1: Compare 6 with 2 and set `M[0]=2` (since $2 < 6$)

Iteration 2: Compare 6 with 24 and set `M[1]=6`

Iteration 3: Compare 23 with 24 and set `M[2]=23`

The key here is to compare the smallest elements in L and R which have not already been placed in M. You should convince yourself that this always finds the "right" value to append to M. Our divide-and-conquer approach is faster than insertion sort, but we

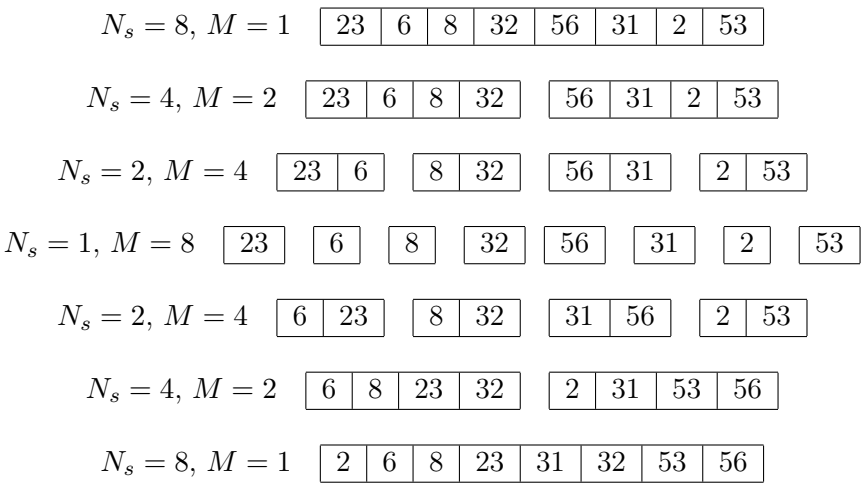


Figure 2.1: Illustration of merge sort

can do better. As with binary search, there is no reason to only divide once. We should keep dividing until the expensive sorting step is as easy as possible. This means we keep dividing until each sub-list contains only 1 element and sorting is trivial. The main part of the algorithm then is to repeatedly merge sub-lists until we have a single sorted list. The full merge sort algorithm is illustrated in Figure 2.1. There is a sequence of divisions until there are 8 1-element lists (note that M corresponds to the number of sub-lists, and N_s is the number of elements in a sub-list). Then there is a sequence of merges, so the 8 1-element lists become 4 sorted 2-element lists. After two more sets of merges, we are left with a single sorted list. How do we implement this in Python? The function *merge* shown below merges two sorted lists.

```
1     def merge(L,R):
2     """
3     Merge sorted lists L and R into single sorted
4     list, M
5     """
6
7     #initialize variables (list sizes, indices, M)
8     nL = len(L)
9     nR = len(R)
10    n = nL+nR
11    indL = 0
12    indR = 0
13    M = []
14
15    #iterate n times adding one element to M per iteration
```

```
    for i in range(n):
        #compare "leftmost" elements of L and R
        if L[indL]<R[indR]:
            #place smaller element in M
            M.append(L[indL])
            #update index of leftmost element of L or R
            indL = indL+1
            #check if end of list has been reached
            if indL==nL:
                M.extend(R[indR:])
                return M
        else:
            M.append(R[indR])
            indR = indR + 1
            if indR==nR:
                M.extend(L[indL:])
                return M

    return M
```

You should read the function carefully and make sure you understand each line. The variables `indL` and `indR` are used to keep track of the locations of the smallest elements in `L` and `R` which have not yet been added to the merged list, `M`. Consider the worst case where there is only one element remaining in one list when we reach the end of the other. First there is an initialization stage where there are 6 assignments, 1 addition, and two length lookups (9 operations in total). Within each iteration of the for loop, there are two variable increments (`i` and either `indL` or `indR`), two comparisons, three list lookups, and one list append (8 operations in total if we count an increment as 1 operation), and the function ends when the last element is appended to `M` using the extend method (two operations). We will have a total of $n-1$ iterations where n is the length of the merged list, so the total number of operations is $C_{merge} \approx 8(n-1)+11 = 8n+3$.

Merge sort can be implemented using a recursive approach. The "base case" for this approach is when the length of the list is one. Then there is nothing to do, and the input list is returned as is. Every other iteration, the input list is split into two parts and the function is called with each part as input, and the output which is returned is then merged. Say we have a two-element list containing 2 and 1. Then the function will call itself twice with one-element lists which will be returned. These one-element lists will then be merged into a sorted two-element list using the merge function above. A Python implementation is shown below.

```
def msort(A):
    """Use merge sort to sort list of integers A
    in non-decreasing order
    """
    n = len(A)
    if n==1:
        return A
    else:
        L = msort(A[:n//2])
        R = msort(A[n//2:])
        return merge(L,R)

    return None
```

What is the cost of merge sort? If we have M length-N_s lists, then there will be $M/2$ merges producing $M/2$ length-$2N_s$ lists. The worst-case cost per merge was estimated to be $8(2N_s) + 3$ operations, so the total cost for these $M/2$ merges will be approximately $(8\,(2N_s) + 3)\,M/2 = 8MN_s + 3M/2$. Noting that $MN_s = N$ and $M \leq N$, we can say that the cost of a set of merges moving from one level to another is, $C < 10N$. If $N = 8$, there are three such moves between levels. If we double N, the number of moves increases by 1. In general, there are $\log_2 N$ moves between levels, and merge sort requires less than $10N \log_2 N$ operations in total in the worst case. Note, however, that we have neglected a detailed operation count of the `msort` function. We can add constants to our upper bound to account for this, cost $< 10(N + c_1) \log_2 N + c_2$ where c_1 and c_2 are small positive integers that become unimportant when N becomes large. We will see later that this estimate implies that the cost of merge sort is $\mathcal{O}(N \log_2 N)$. What this tells us is that the rate at which the cost increases is slower than the quadratic trend we saw for insertion sort, and merge sort is clearly superior for long lists.

Lecture 3

Recursion & sorting; dynamic search & hash functions

This lecture consists of two parts. In the first part, I will provide a few general comments on Python, recursion, sorting, and searching. In the second part, we will consider searching in dynamic datasets and introduce hash functions.

3.1 Getting comfortable with Python

At this stage, you should:

- have command of all of the material in online review lectures - use exercises for self-assessment (solutions are online)
- understand the structure and purpose of functions
- choose an editor + terminal combination for developing code. This can be spyder (distributed w/ anaconda), visual studio + jupyter qtconsole (which is what I use), or pretty much any other options that your prefer. You should use python 3.x (e.g. python 3.11)
- understand the binary search and merge sort codes

For further help with Python, see the list of supplementary material on Blackboard.

3.2 Recursion

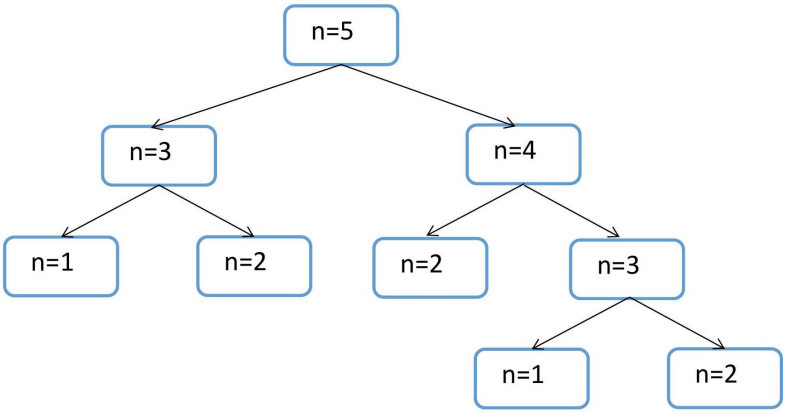
Lab 1 asked you to work with recursive functions. When trying to understand the details of how a recursive function works, it can be helpful to put together a recursion tree. Consider the code below for computing the nth term in the Fibonacci sequence:

```
1     def fib(n):
2     """Find nth term in Fibonacci sequence start from 0,1
3     """
4     print("n=",n)
5     if n==1:
6         return 0
7     elif n==2:
8         return 1
9     else:
10        return fib(n-2) + fib(n-1)
```

Running this function with $n = 5$ gives the following output:

```
fib(5)
n= 5
n= 3
n= 1
n= 2
n= 4
n= 2
n= 3
n= 1
n= 2
```

The recursion tree for fib(5) is shown below:



The tree immediately shows this is an inefficient approach since the function is called with the same input multiple times. Just because we can use recursion doesn't mean we should. We won't go into the details of how the Python interpreter deals with recursive functions, however here are a few key points:

- 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.

The key idea behind merge sort (and binary search) is to "divide and conquer". This is a general approach used in many algorithms. For this approach to be useful, the total work for sub-problems after division has to be sufficiently small relative to the work required for the original problem. So it won't help, for example, when summing the elements in array.

3.3 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, it 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 comparing how well different sorting algorithms work. 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 certain size and/or structure, then choosing a particular sorting algorithm may improve code performance. Understanding how to implement and analyze sorting and searching algorithms provides a foundation which will help us analyze more complicated methods.

3.4 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 (or visitors may ask to have their data 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 $\mathcal{O}(1)$).

Initial proposal: Assign an integer to each unique visitor, and use this as a location in an array. Say that a (struggling) website had 4 visits in the last month, and the visitor were assigned the the following integer ids:

6	8	23	32

Then, we would have a list, `X`, where `X[6]=X[8]=X[23]=X[32]=1`. To check if a new visitor with id $= j$ is new, we check the value of `X[j]`. Speed for search would be constant time, $\mathcal{O}(1)$ since the cost of accessing an element in a list using a specified index is independent of the size of the list. However, this approach has many weaknesses. For example, it requires integer ids. A more significant 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 address this issue and modify the problem so that it is more realistic. 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 a list with space 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). Instead, 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 subject and an active area of research. Let's now consider what the "desirable" properties of a hash function are.

3.5 Hash function

Consider a hash function $H(a) = i$ where i is a non-negative integer, and a is a numeric representation of an IP address. The function should have two desirable properties:

Desirable property 1: The function should always assign a particular IP to the same index on repeat visits.

Desirable property 2: Saw we constrain i so that $0 \leq i \leq N$. Then the probability of two distinct visitors being assigned the same index, i, should be $1/N$.

The first property ensures that we can find repeat visitors. The second desirable property follows from the idea that H should distribute visitors evenly across available spaces. Since there are 256^4 possible IPs, it is impractical to design H so that unique visitors are always assigned unique indices, and when two or more different IPs are assigned the same index, we have a *hash collision*. The idea here is that there should not be a tendency for certain indices to be assigned to a relatively large number of visitors generating many hash collisions.

Consider the following problem setup:

- Represent an IP address as four integers, $a = \{a_1, a_2, a_3, a_4\}$
- Randomly choose four arbitrary integer weights, $\{w_1, w_2, w_3, w_4\}$

Let's say we are expecting about 1000 visitors to our site. 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 $i = \sum_{j=1}^{4} w_j a_j$ 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. For example, there could be a tendency for a_4 to be zero. It turns out that it is better to use a prime number, e.g. $i = \sum_{j=1}^{4} w_j a_j$ rem 997 . It is less likely for patterns in IP addresses to cause a problem, but more importantly, this will allow us to make a connection between the proposed function and desirable property 2.

3.5.1 Aside on modular arithmetic

In order to make such a connection, we will require a few results from modular arithmetic, and this section quickly reviews a few basic definitions. Here, we are assuming that all variables are integers.

- a 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 $r = a$ rem b

Example: $a = 12, b = 5 : a$ rem $b = 2$.
It is often useful to equate integers which have the same remainder, and if $c > 0$:

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

and we then say "a and b are congruent modulo c." For example, if $a = 12, b = 17, c = 5$, then $a \equiv b \pmod{c}$.

> Optional exercise: Show that a rem $c = b$ rem c implies that $c \mid a - b$ where $c \mid a - b$ means that "c divides $a - b$"

3.5.2 Hash function analysis

Our proposed hash function is $h(a) = \sum_{j=1}^{4} w_j a_j$ rem p where p is a prime. How well does this hash function work? Consider the following related question: given two different addresses a and b, what is the probability that $H(a) = H(b)$, and is this probability $\leq \frac{1}{p}$? The answer to this question will tell us if the function tends to place IP addresses in certain locations or if instead it tends to distribute them evenly. In other words, it will tell us if our function satisfies desirable property 2. For convenience, assume that $a_4 \neq b_4$, and note that we still have to specify how the weights are selected. We want to find the probability that the following congruence is satisfied:

$$\sum_{j=1}^{4} w_j a_j \equiv \sum_{j=1}^{4} w_j b_j \pmod{p}.$$

The rules of modular arithmetic tell us that, $a \equiv b \pmod{n}$ implies $a+c \equiv b+c \pmod{n}$, so we can rearrange the congruence as,

$$\sum_{j=1}^{3} w_j (a_j - b_j) \equiv w_4 (b_4 - a_4) \pmod{p}. \tag{3.1}$$

We would like to find an expression for w_4, and then, once we specify how the weights are chosen, we will we be able to find the probability of a hash collision. Ideally, we would like to divide both sides of the congruence by $(b_4 - a_4)$, but this sort of rearrangement is only possible in cases where a multiplicative inverse exists:

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

Using this result, we can then show that if $c \equiv dk \pmod{x}$ then $ck^{-1} \equiv d \pmod{x}$ if k is not a multiple of x and $d < x$. In order to apply this to (3.1), we need 1) p to be a prime, a condition we have already stated, and 2) $(b_4 - a_4)$ should not be a multiple of p. We know that $|b_4 - a_4| < 256$, so we also need to choose a prime larger than 256. First rewriting the congruence as,

$$w_4 (b_4 - a_4) \equiv c \pmod{p}, \tag{3.2}$$

where $c = \sum_{j=1}^{3} w_j (a_j - b_j)$, we can now "solve" the congruence for w_4 giving,

$$w_4 \equiv c(b_4 - a_4)^{-1} \pmod{p}, \tag{3.3}$$

where $(b_4 - a_4)^{-1}$ is the multiplicative inverse of $b_4 - a_4$. We will now argue that if the four weights are four non-negative integers less than p selected uniformly at random, then the hash function will satisfy desirable property 2. With this additional constraint, $w_4 \text{ rem } p = w_4$ (since $w_4 < p$), and (3.3) can be rewritten as,

$$w_4 = c(b_4 - a_4)^{-1} \text{ rem } p. \tag{3.4}$$

The right-hand side is just some integer between 0 and $p-1$, and since w_4 is drawn uniformly at random from this range, the probability of (3.4) being true is $1/p$. 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 (we require p to be a prime larger than 256 and the weights, w_i to be drawn uniformly at random from $\{0, 1, ..., p-1\}$).

> Optional exercise: is it important that we assumed that specifically the 4^{th} 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, and discuss their implementation in Python. We will also see an application in bioinformatics in Lecture 5.

Lecture 4

Hash tables & dictionaries; analyzing computational cost

4.1 Searching

How do we use hash functions to search more efficiently (and beat binary search)? The key 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. This all needs to be done efficiently! Consideration of these steps will lead us to a "proper" understanding of what a Python dictionary is. Let's think about hash tables in the context of the IP address example where a startup is keeping track of unique visitors to its website each month, and:

- Each visitor must be assessed as "new" or "repeat".
- Visitors may ask to have their data deleted.
- Everything must be fast and accurate.

We want to beat binary search when searching to determine if a visitor is new, and the addition/removal of data should also be efficient. What will the general workflow be?

1. Choose the prime (N) for your hash function, and initialize a list (or dictionary) where you will store addresses. This is the hash table. The prime should be a little larger than the number of visitors you expect to process.
2. Given an IP address, compute an index using your hash function.
3. Check if the visitor is new.
4. If new, append the IP address (and additional data) at the corresponding location in the hash table.
5. If not new, update the associated data as needed at the appropriate location in the hash table.

Say that our hash table is a list of lists, `L` (so `L[i]` will initially be an empty list). 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]`. If there are hash collisions, there will be multiple items stored in the list, `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? We consider search, insertion, and deletion separately.

Search : 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 `len(L[i])>1`, iterate through items (if `L[i]==1`, check `L[i]` for match).

The cost of the first two steps is independent of M and N. The third step requires M/N iterations on average, so the 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

The cost of search is the same as above, and the other steps have $\mathcal{O}(1)$ cost, so the overall cost is again $\mathcal{O}(\max(1, M/N))$.

Delete: Remove an IP

This is a little more complicated than Insert, and the cost depends on the details of how the hash table is implemented. After search, you can: set the number of visits to zero ($\mathcal{O}(1)$ cost), 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 $\mathcal{O}(\max(1, M/N))$.

Search methods summary:

Linear search: Cost is $\mathcal{O}(N)$.

Binary search: Cost is $\mathcal{O}(\log_2 N)$ but requires the maintenance of a sorted list/array.

Hash table: Cost is $\mathcal{O}(1)$ for search as well as maintenance provided that the hash function and table are both well-designed!

4.2 Hash tables in Python and dictionaries

What does a hash table 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 strings. Python provides a built-in hash function that returns an integer for (almost) any input:

- For an integer i, `hash(i)=i` (provided that $|i| \lesssim 10^{18}$).

- For two inputs, if `a==b` then `hash(a)=hash(b)`.
  For example, `hash(3.0)=hash(3)=3`.

For non-integers, the function is less predictable:

```
In [1]:  hash ('math')
Out [1]:  -6564006556742056588
In [2]:  hash('maths')
Out [2] :  290530591815711155
```

As noted earlier, we can use a list-of-lists to build a hash table, and the the output of `hash` modulo an appropriate prime could be used to provide location indices. But we don't need to build our own hash table, because Python dictionaries essentially are hash tables! (Technically, they are "associative arrays".) Dictionaries are containers where each element is a key-value pair. A key can be considered to be a a label or ID that is used to identify information stored as the corresponding value. A simple example is shown below:

```
In [1]:  key = "123.45.241.12"
In [2]:  value=[14,1,2022,20]
In [3]:  d =  #initialize dictionary
In [4]:  d[key] = value #insert key/value pair in d
In [5]:  d["123.45.241.12"]
Out[5]:  [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 approach for hash collision is *open addressing* while our previously-described approach where information for multiple keys was stored in the same location is *chaining*. Python will automatically re-size dictionaries as well when they are close to full. Examples of useful dictionary operations and their costs are shown below.

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

```
In [36]:  d=dict()
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 ad
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 operation $\mathcal{O}(N)$

```
In [45]:  for key in d:  # iterate over keys in d
```

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

4.3 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) time complexity in the form $\mathcal{O}(f(N))$ where N is the problem size (e.g. binary search is $\mathcal{O}\left(\log_2(N)\right)$). 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.

When analyzing algorithms, we are primarily interested in the theoretical time complexity, and in particular the asymptotic time complexity 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 time complexity.

4.3.1 Analyzing time complexity

On a basic level, analyzing running time is a matter of counting. For example, a code snippet could have a additions, b assignments, and 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. For example, for linear search with size-N input:

Best case: 1 iteration, (1 assignment, 1 comparison, 1 addition)/iteration $\rightarrow$ 3 operations in total

Average case: $3(N+1)/2$ operations

Worst case: $3N$ operations

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, 1) correctly assess operations involving containers (lists, arrays, dictionaries...) which can contain several elements, and 2) 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 two more complicated examples 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 a single item to the end of a list does not depend on M nor does popping 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 at item to the front.

4.3.2 Asymptotic time complexity

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 $N \geq N_0$, $C(N) \leq af(N)$.

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, there is no need for other approaches. An algorithm that is $\mathcal{O}(N)$ will also be $\mathcal{O}\left(N^2\right)$, 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*

lower-order terms. We ignore the lower-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. The cost, $C(N)$ is $\Omega(f(N))$ if there is a positive constant a, and a positive integer, N_0, where for all $N \geq N_0$, $C(N) \geq af(N)$. This provides a lower bound and describes the best-case scenario. The 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

Pattern search in strings (and gene sequences)

We now shift our focus to searching for patterns within genomes, and we'll begin with a little background science before moving on to the pattern search problem.

5.1 Genetic code

DNA (and RNA) consists of two "strands" arranged in a double helix and connected with covalent bonds (see Figure 5.1). 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, and Thymine (RNA has Uracil in place of Thymine), and we will refer to the bases using their first letter. Crucially, Adenine bonds only with Thymine and Guanine bonds only 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 2^{nd} strand can be constructed so A's pair with T's and C's pair with G's.

Codons consist of three consecutive DNA bases and provide code for synthesizing amino acids. There are 64 possible codons for DNA, but there are only 20 (naturally-occurring) 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 work"). Gene sequencing involves: 1) extracting the sequence of bases from DNA samples, and 2) investigating the proteins or functions associated with codons and their sequences. The second of these steps can be restated as the following computational problem: given a DNA sequence, search for a given pattern. Both the sequence and the pattern can be extremely long, and the number of patterns can also be large. The fruit fly genome contains 139.5 million base pairs, while the human genome contains either 3 or 6 billion base pairs depending on the cell type. Why do we want to (efficiently) search for patterns? In general, trends and patterns in gene sequences can carry important information. For example:

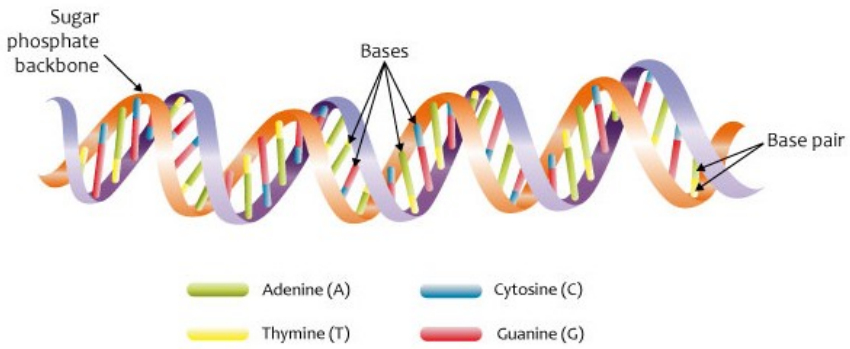


Figure 5.1: DNA structure[2]

- 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.

5.2 Pattern search

Taking the above as motivation, our computational problem is:

> Specify a n-character sequence, S, and a m-character pattern, P.
> Find all locations in S where P occurs

For example:

$$S = \text{ ATGTTGTACCGTATCGG}$$
$$P = \text{ GTA}$$
$$n = 16, m = 3$$

What would you say is the simplest way to solve this problem? Consider the following "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 mismatch.

The code snippet below implements 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,p in enumerate(P):
        if p != S[ind+count]:
            matching=False
            break
        #Update list when match found
        if matching:
            imatch.append(ind)
        print("Match found, i=",ind)
```

Running this code produces the following output:

```
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. The best-case cost is $\mathcal{O}(n)$ and occurs when the first letter of the pattern does not appear in S. This won't happen often, but the cost will be close to $\mathcal{O}(n)$ when the number of matches between the first few letters in the pattern and S is small. The worst-case cost is $\mathcal{O}(mn)$ operations, and the cost will be close to this worst case when there are many matches and/or many near-misses (where close to m characters match).

Can we do better using the search methods we have already discussed? Can we use binary search? We could, 1) first store the $n-m+1$ length- m strings in a numeric form, and then 2) sort the resulting list with $\mathcal{O}\left(n\log_2(n)\right)$ cost. Then the cost is $\mathcal{O}\left(\log_2(n)\right)$ for each search. There could be a cost benefit if $log_2 n < m$, however there is also a cost with extracting the length-m strings from S, and this also requires storing n length-m strings/arrays and for large mn, the memory requirement may be excessive. Can we use a hash table? Again we could store the n length-m strings in a Python dictionary. This will certainly be faster and simpler than the binary search method, but memory usage may still be problematically large when mn is large, and again it is important to consider the cost of extracting the length-m strings which are inserted into the dictionary.

A different approach is to use a *rolling hash function*. Consider the following approach: 1) Compute a hash for the pattern, P and 2) apply the hash 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 $\mathcal{O}(m+n)$, so there is a potential improvement.

We compute the hashes by first rewriting genetic sequences in base 4 using the following transformation: $A \to 0, C \to 1, G \to 2, T \to 3$ and then the base 4 number corresponding to a m-character sequence is converted to base 10.

Example: compute hash for GCTAT

$S = \text{GCTAT} \to X = 21303,$

$H(X) = 2 * 4^4 + 1 * 4^3 + 3 * 4^2 + 0 * 4^1 + 3 * 4^0 = 627.$

Or more generally when X contains n digits, evaluate a $(m-1)^{\text{th}}$ -order polynomial for each length- m sequence of consecutive integers in X. This approach isn't really helpful. The polynomial contains m terms, and there will be $\mathcal{O}(m)$ operations for each of the $(n-m+1)$ sequences (leading to an overall $\mathcal{O}(nm)$ cost). The hash function evaluation is too expensive. The key is to update the hash 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 (if $m = 3$ and $S = \text{AGTCA}$, 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,

$$H(x_i) = x_{i,1}4^{m-1} + x_{i,2}4^{m-2} + \ldots + x_{i,m-1}4 + x_{i,m}. \tag{5.1}$$

Can we use $H(x_i)$ to help us compute $H(x_{i+1})$? We know that the last $m-1$ digits in x_i match the first $m-1$ digits in x_{i+1}, and comparing the expressions for $H(x_{i+1})$ below to that for $H(x_i)$ above,

$$H(x_{i+1}) = x_{i+1,1}4^{m-1} + x_{i+1,2}4^{m-2} + \ldots + x_{i+1,m-1}4 + x_{i+1,m}, \tag{5.2}$$

we see that multiplying the last $m-1$ terms in (5.2) by 4 gives us the first $m-1$ terms in (5.2). This means that we can compute $H(x_{i+1})$ using the following update equation:

$$H(x_{i+1}) = H(x_i) * 4 - x_{i,1}4^m + x_{i+1,m}.$$

The computation of $H(x_j)$ for $j > 1$ will require 4 rather than $\approx 2m$ operations per hash evaluation. The overall cost of pattern search using this rolling hash function will then be $\mathcal{O}(n)$ in both the best and worst cases.

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 tend to be more important for problems in, say, base-26 than in base-4. This problem can be alleviated with the modulo operator. We define $h(x_i) = H(x_i) \text{ rem } q$ with q a large prime number and as before:

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

We can then use rules from modular arithmetic to simplify the calculation of $h(x_{i+1})$ given $h(x_i)$:

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

However, introducing the rem operator to avoid large integers also introduces the possibility of hash collisions where two different strings produce the same hash. 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) and its Python implementation:

1. Convert S and P to base 4 (or 26 or ...) : $S \to X$ and $P \to Y$:

```
def char2base4(S):
    """Convert gene sequence S 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)
```

2. Compute hashes of Y and 1$^{\text{st}}$ 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)
```

3. Iterate through remainder of X and:

- Each iteration, use update equation to compute $h(x_{i+1})$ 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: $h(x_{i+1}) = [h(x_i) * 4 - x_{i,1}(4^m \text{ rem } q) + x_{i+1,m}] \text{ rem } q$

```
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:

- `match(A,B)` has not been provided; it is a function which carries out a character-by-character comparison of equal-length input strings `A` and `B`
- `heval` is using Horner's method for efficiently evaluating polynomials.
- We have pre-computed `bm` 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 of the algorithm is $\mathcal{O}(nm)$; this occurs when there are many hash matches and the subsequent 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 and stored.

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, it was introduced in 1987. There are many applications outside of bioinformatics including plagiarism detection and the "find" function in text editors and browsers.

Lecture 6

Networks & NetworkX; graph search & BFS

6.1 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:

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

Example: The graph below has 6 nodes and 7 links: Node 1 has two edges: $(1, 2)$ and $(1, 5)$.
A graph can be represented by an adjacency matrix, $\mathbf{A}$, where $A_{ij} = 1$ if there is link between nodes i and j and is zero otherwise. The adjacency matrix for our example is:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

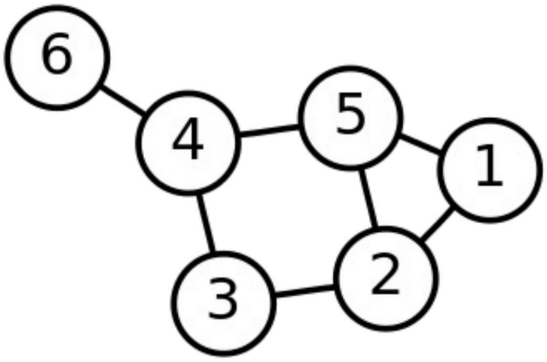
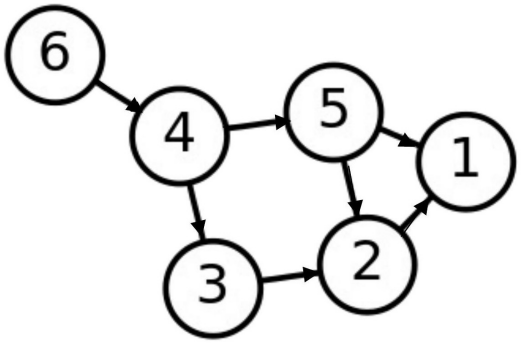


Figure 6.1

This is an example of an undirected graph - each link can be traversed in either direction. Then $A_{ij} = Aji$. $\mathbf{A}$ is symmetric for any undirected graph. 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 other representations of graphs as well such as adjacency lists. 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 the total number of links connected to it, and for the example above, $k_1 = 2, k_5 = 3, \ldots$. Check your understanding: what is k_2?

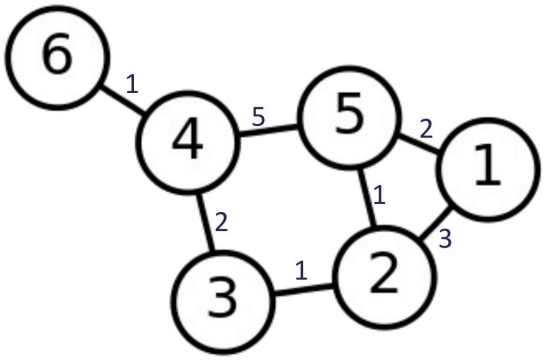
Networks can also be directed as in the graph below:



Then $A_{ij} = 1$ if there is a link to i from j :

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

A will not be symmetric for directed graphs. Networks can be weighted (e.g. weights can correspond to distances or travel times in transportation networks):



The edge weights can be represented with a weight matrix, $\mathbf{W}$:

$$\mathbf{W} = \begin{pmatrix} 0 & 3 & 0 & 0 & 2 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 5 & 1 \\ 2 & 1 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Networks can also be both directed and weighted.

We are generally interested in large complex networks (see table 6.1). 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 in Python. Conveniently, it provides a `Graph` datatype which stores the details of the graph structure (nodes and edges). Let's see how to build the graph in figure 6.1 in NetworkX. First, import the module, and initialize a graph:

```
In [1]:  import networkx as nx
In [2]:  G = nx.Graph()
```

There are numerous methods that can be used to build a graph. We can add one edge at a time using `add_edge`:

```
In [3]:  G.add_edge(1,2)
In [4]:  list(G.edges ())
Out [4]:  [(1,2)]
In [5]:  list(G.nodes())
Out [5]:  [1,2]
```

Table 6.1: Examples of complex networks and their sizes (N is the number of nodes; L is the number of links.) Note that the WWW network is a small portion of the actual world-wide web. Taken from [1] and licensed under CC BY-NC 3.0.

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

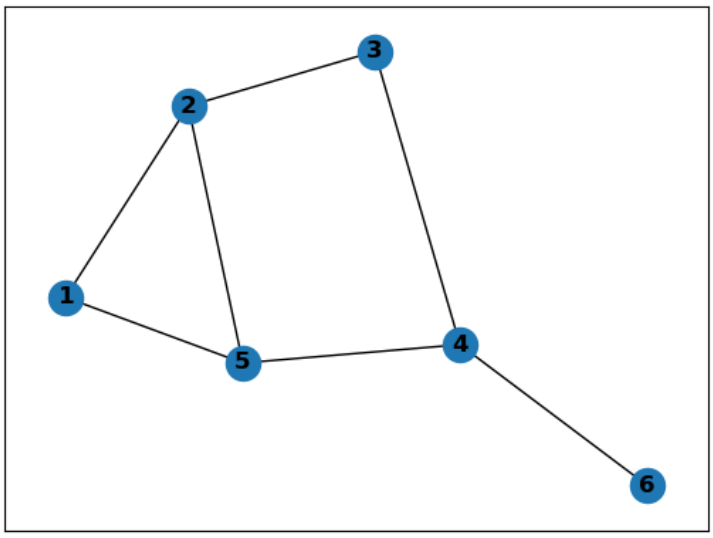
We can also add several edges (or nodes) at once using `add_edges_from`:

```
In [65]:  e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]
In [6]:  G.add_edges_from(e)
In [7]:  list(G.edges())
Out[7]:  [(1, 2), (1, 5), (2, 3), (2, 5), (5, 4), (3, 4), (4, 6)]
In [8]:  list(G.nodes())
Out[8]:  [1, 2, 5, 3, 4, 6]
```

We can use `nx.draw` to visualize the network and check that it is equivalent to the network in figure 6.1:

```
In [9]:  plt.figure()
Out [9]:  <matplotlib.figure.Figure at 0x1515e3fefo>
In [10]:  nx.draw(G, with_labels=True, font_weight='bold')
```

The code above creates the following image:



Note that `nx.draw` sets the node positions randomly, so the layout of the graph will typically change each time the command is run. Given a NetworkX graph, it is straightforward to extract the neighbors of any node:

```
In [11]:  list(G[2])
Out[11]:  [5, 3, 1]
```
or:
```
In [12]:  list(G.adj[2])
Out[12]:  [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 [13]:  nx.shortest_path(G,source=2,target=6)
Out[13]:  [2, 3, 4, 6]
```

Finding shortest paths is very important in the study of graph algorithms and will be discussed in upcoming lectures. Note that the *distance* between two nodes in an unweighted graph is the number of links in a 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

More information about networks:

- See chapter 2 of Network Science: http://networksciencebook.com/chapter/2

6.2 Graph search

Graph search is a class of algorithms for exploring graphs, analyzing their structure, and finding shortest paths. Graph search is important as it is essential for effective navigation, (see figure 6.2), and it also provides important information about network structure (e.g. which nodes are (un)reachable from a given starting node). Many networks of interest have hundreds of thousands of edges (or more), and it is essential that search algorithms are designed and implemented to be efficient.

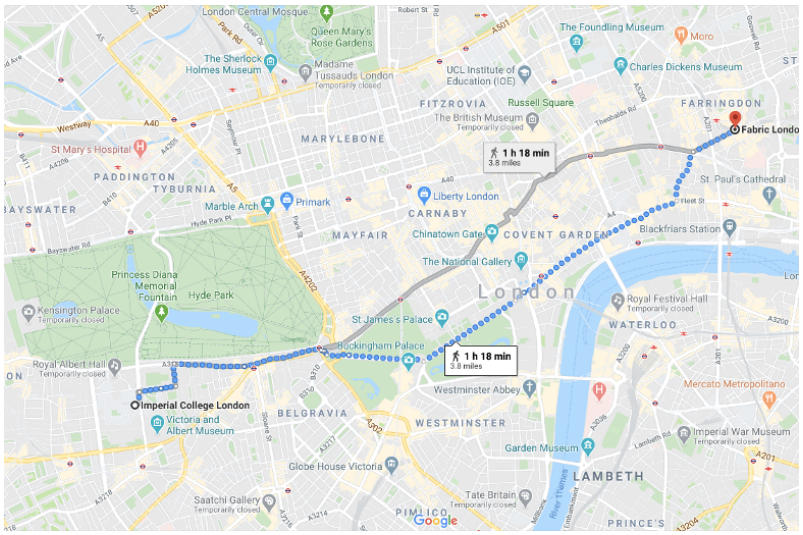


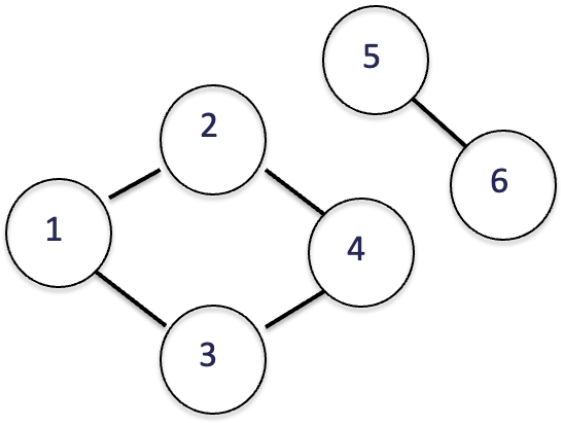
Figure 6.2: Navigation using graph search and Google maps.

We begin by considering a general algorithm for the following problem:

> Given a graph, G, and a source node, s, find all nodes that can be reached from s.

Our general algorithm is:

- 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



Note that this algorithm is under-specified. If there are multiple edges connecting explored and unexplored nodes, we do not have a rule for deciding which edge to select. Let's apply this algorithm to the graph shown above with $s = 1$. Initially, only node 1 is explored and we can select one of two links: $(1, 2)$ or $(1, 3)$. Select $(1, 2)$ (this is arbitrary) and label node 2 as explored. There are again two links we can choose, $(1, 3)$ or $(2, 4)$. Select $(2, 4)$ (another arbitrary choice), and label node 4 as explored. Again there are two links to choose from: $(1, 3)$ and $(4, 3)$. Arbitrarily choosing $(1, 3)$, we label node 3 as explored. There are now no further links connecting unexplored and explored nodes, so the algorithm terminates. Nodes 5 and 6 remain unexplored indicating that they are not reachable from the source node. The algorithm is correct for this specific case, but is it generally correct? Consider the following claim:

> Upon completion of the graph search algorithm, a node is labeled explored if and only if a path exists between it and the source node.

There are 2 cases to consider:

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?

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. An inductive argument can then be used to establish that case 1 cannot occur. For case 2, 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, and case 2 cannot occur.

The implementation of the general algorithm depends on how edges are selected each iteration. One of the following two approaches is typically used:

- 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 ($\mathcal{O}(N + L)$ for a graph with N nodes and L links). 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: 1) how should we store "information" to ensure an efficient search? and 2) what does the answer to 1) mean in terms of a Python implementation?

6.2.1 Breadth-first search

The key elements of a Python implementation of BFS are:

- Input: 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

Returning to our search example with $s = 1$, 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 option. We then pop node 2, and append node 4 giving `Q=[3,4]`. Then we pop node 3 which has no unexplored neighbors, and finally we pop node 4. How does BFS actually look in Python? First, specify the graph and source node, and add the source node to the queue:

```
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)
```

Note that the nodes in the graph are numbered from 1 to 6, and `L2[i]` contains the label for node $i+1$. Adding `print(" n=% d, Q=" %(n), Q)` to the while loop and running the code produces the following output:

$$\begin{aligned} \texttt{n} &= 1, \texttt{Q} = [2, 3] \\ \texttt{n} &= 2, \texttt{Q} = [3, 4] \\ \texttt{n} &= 3, \texttt{Q} = [4] \\ \texttt{n} &= 4, \texttt{Q} = [\,] \end{aligned}$$

Here, `n` is the node removed from the queue and `Q` is the queue after unexplored neighbors of `n` have been added. You should check that the output makes sense for the given graph. What is the cost of BFS? Each reachable node is relabeled as explored, and each (reachable) edge is encountered twice. For a graph with N nodes and L edges, the cost is $\mathcal{O}(L+N)$ (if our queue is managed efficiently).

We can also use BFS to compute the distances to the source for all reachable nodes. BFS iterates through the graph layer by layer, and for our example, we know nodes 2 and 3 will be explored before 4. More generally, when a node is added to the queue, its distance is one greater than the distance of its neighbor being removed from the queue. We just need to maintain a list of distances whose elements are filled in as nodes are added to the queue as shown in the code below:

```
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)
```

After running this code we have:

```
In [5]:  D
Out [5]:  [0,1,1,2,-1000,-1000]
```

We can see that the distances are correct (`D[i]` is the length of a shortest path between the source node and node $i+1$), and a distance of -1000 indicates that the node is unreachable from the source.

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 2^{nd} look at the code. We ignore the cost associated with creating the graph, and the cost of creating the length-N lists is $\mathcal{O}(N)$. When iterating through the neighbors of node n, we treat `G.adj[n]` as a dictionary lookup and then the cost per neighbor is $\mathcal{O}(1)$. What about iterating through the queue and removing nodes? 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 provides the deque datatype. From the online documentation for `deque`: [a deque is] *a list-like container with fast appends and pops on either end.* For a deque, popping from the front and appending to the end are both $\mathcal{O}(1)$, and using this datatype, we can assemble a linear time implementation of BFS in Python.

Lecture 7

DFS and more on Python containers

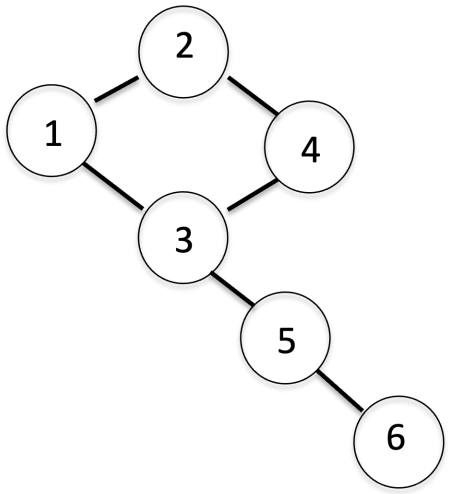
7.1 Depth-first search

Depth-first search (DFS) will also find all reachable nodes from a specified source. BFS requires the management of a queue which uses a first-in, first-out principle, DFS requires a *stack* which uses last-in, first-out. As a result, there is a tendency for DFS to move more quickly away from the source. How much does the code change when we switch to DFS? The problem setup and initialization will be the same, and we can focus on the serach through the graph. For BFS, remove the node from the front of the queue, append unexplored neighbors to the back of the queue. For DFS, remove the node from the end of the stack, append unexplored neighbors to the end of the stack. The modified code is shown below.

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

We use `S` instead of `Q` to indicate the shift from a queue to a stack, and the only meaningful change is in line 3 where the node from the end of the list `S` is being popped. A Python list is just what is needed to manage a stack since it provides fast ($\mathcal{O}(1)$) pops and appends from the list's end. Let's compare DFS and BFS on the graph below. Output from BFS and DFS codes is provided below the graph, and we can compare the order in which nodes are visited. 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




```
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= []
```

of links is $\begin{pmatrix} N \\ 2 \end{pmatrix} = \frac{N(N-1)}{2}$ 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.

7.2 Comments on Python containers

Quiz: what is the output from the code below?

```
1 e = [(1,5),(2,5),(2,3),(3,4),(4,5),(4,6)]
2 G1 = nx.Graph()
3 G1.add_edges_from(e)
4 G2 = G1
5 G2.add_edge(6,7)
6 list(G1.edges())
```

The output is:
`[(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`, 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)]
```

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, and `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).

Lecture 8

Shortest paths in weighted graphs

8.1 Weighted networks and Dijkstra's algorithm

We have only considered unweighted networks so far, however, it is often useful to associate a numerical weight with each edge as in figure 8.1. For example, weights can indicate physical distance or travel time in transportation networks, and they can also be used to represent frequency of interactions between users in online social networks. What is the best way to compute distances and find shortest paths in a weighted network? We can use BFS in some situations. 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, and we will consider the following problem setup:

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

The length of a path in a weighted graph 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. Then, in the undirected graph pictured, the path 5,4,6 has length 6 while $d_{56} = d_{65} = 5$.

The key elements of Dijkstra's algorithm are:

- maintain a priority queue, M, of explored nodes where each node in M is assigned a numerical "priority". Say that the maximum priority is $p_{\max}$
- each iteration remove a node from M with priority $= p_{\max}$, 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

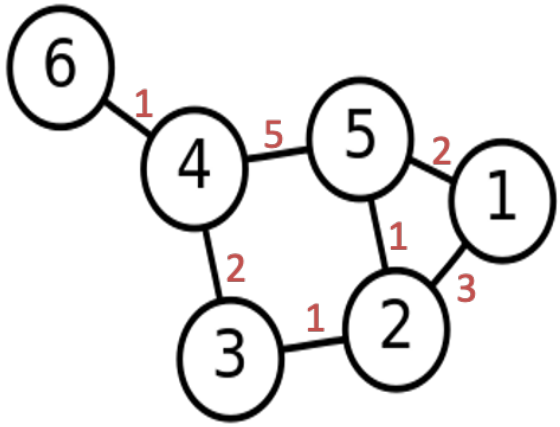


Figure 8.1: Example of weighted graph

– move unexplored neighbors to M, and assign priorities to these nodes

There are strong simlarities to BFS, and the key difference is the need to consider priorities. 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. 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 allows 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 they are. We will sacrifice a little generality and answer the three questions above 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). The answers to the three questions are then:

1. Set $\delta_a = d_{si} + w_{ia}$; this is the length of a shortest path from s to a that includes the link $i - a$.
2. Set $\delta_b^{\text{new}} = \min\left(\delta_b^{old}, d_{si} + w_{ib}\right)$; the rationale for this rule will be explained a little later.

3. For node j, $d_{sj} = \delta_j$ so it (or any other node whose PD matches the minimum PD in M) can be added to the set of "finalized" nodes.

Based on these answers, we can summarize the 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.
>
> Each iteration, find a node in M with the minimum PD (maximum priority) and move it to F. Say this node is node i.
>
> Examine the neighbors of i. For neighbors already in M, update their PDs with: $\delta_b^{\text{new}} = \min\left(\delta_b^{old}, d_{si} + w_{ib}\right)$; for unexplored neighbors, set $\delta_a = d_{si} + w_{ia}$; for neighbors already in F, do nothing.
>
> Continue "exploring" until no reachable nodes remain unexplored.

Consider the graph in Figure 8.1, 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 1^{st} iteration we have: $F = \{(5, 0)\}, M = \{(1, 2), (2, 1), (4, 5)\}$. Continuing on:

2^{nd} iteration: $F = \{(5, 0), (2, 1)\}, M = \{(1, 2), (4, 5), (3, 2)\}$

3^{rd} iteration: $F = \{(5, 0), (2, 1), (1, 2)\}, M = \{(4, 5), (3, 2)\}$

4^{th} iteration: $F = \{(5, 0), (2, 1), (1, 2), (3, 2)\}, M = \{(4, 4)\}$

5^{th} iteration: $F = \{(5, 0), (2, 1), (1, 2), (3, 2), (4, 4)\}, M = \{(6, 5)\}$

Last 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 1^{st} 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 n^{th} iteration and assume that the distances for the $\text{n} - 1$ nodes in F have been set correctly. Let x be a node in M with priority $= p_{\max}$. We will now argue that $\delta_x = d_{sx}$. Figure 8.2 visualizes the state of the search, and there are three important 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

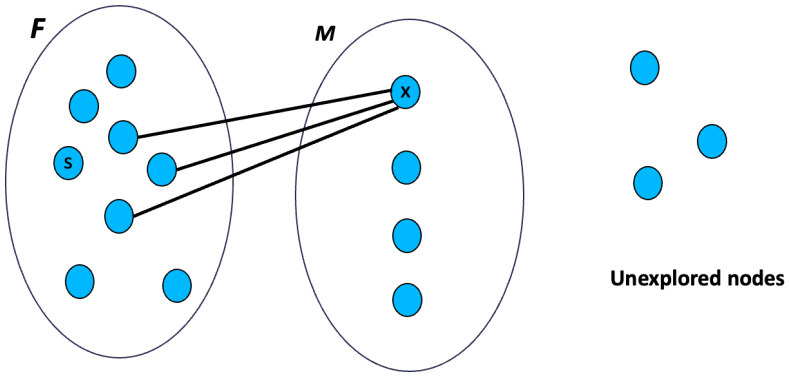


Figure 8.2: This is a sketch of how the graph search problem "looks" after a few iterations of Dijkstra's algorithm.

- A path from s to x must cross at least one link from F to M

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 $i_1, i_2, i_3, \ldots$ are the neighbors of x in F. The shortest path from s to x with all nodes in F except x is:

$$\min\left(d_{si_1} + w_{i_1x}, d_{si_2} + w_{i_2x}, \ldots\right)$$

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.

8.2 Python implementation

The Python implementation can be broadly similar to what we did for BFS, but we have to consider how to manage the priority queue. Specifically, we need to decide which data type to use for the priority queue: a list, array, dictionary, something else? The basic operations needed for queue management are:

- find minimium (provisional distance)
- insert/delete

Let's first try to use a dictionary. Dictionaries 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). We'll also have a NetworkX graph provided as input. The dictionary-based Python implemention is shown below.

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 produces the following output:
`In [56]: Fdict`
`Out[56]: {1: 2, 2: 1, 3: 2, 4: 4, 5: 0, 6: 5}`
The key-value pairs are node:distance, and you should verify that the distances are correct.

8.2.1 Computational cost

What is the computational cost for a graph with N nodes and L edges? Each node will at some point be added and removed from `Mdict` ($\mathcal{O}(N)$ cost). Each edge is examined twice ($\mathcal{O}(L)$ cost). What about the cost of finding the node in the queue with minimum provisional distance? `Mdict` contains at most N nodes each iteration. Then, each iteration, the `dmin` 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 $\mathcal{O}\left(N^2\right)$. The minimum calculation is a bottleneck, and we do not have a linear time implementation. 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 $\mathcal{O}\left(N^2\right)$

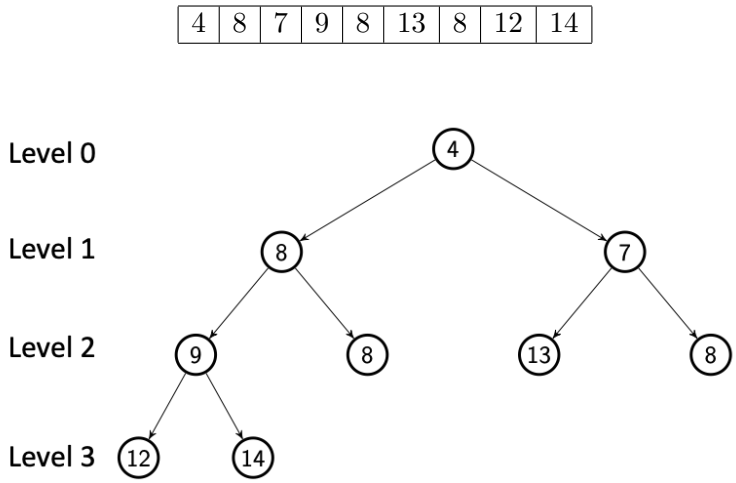


Figure 8.3: Binary tree representation of list ordered as binary heap.

8.3 Binary heaps

Given the bottleneck in our implementation of Dijkstra's algorithm, 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 can provide these operations with cost, $\mathcal{O}(\log_2 N)$. We won't go through the full Dijkstra + heap implementation, but will instead just 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 to the keys of its children. In the example shown in Figure 8.3, there are four levels, and each level (starting at the top and moving down) must be as full as possible. We may also 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. Our task is to insert an element with key $= 5$ in the heap shown in the figure.

Step 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 (Figure 8.4).

Step 2: Compare the new element with its parent and swap them to preserve the heap

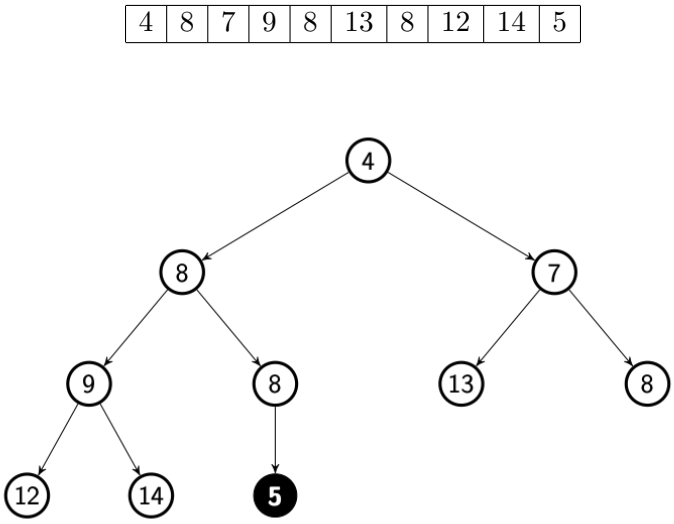


Figure 8.4: Step 1 of insertion

property (Figure 8.5). We continue comparing the new element with parents and swapping

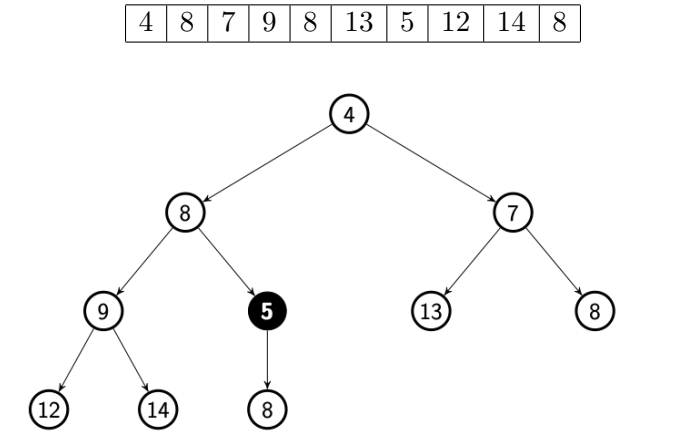


Figure 8.5: Step 2 of insertion

until a parent with a smaller key is encountered (Figure 8.6) You should verify that the heap property is satisfied in the final figure. 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

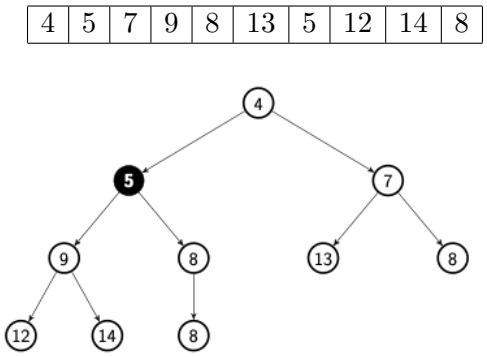


Figure 8.6: Step 3 of insertion

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 $\mathcal{O}(1)$, then the overall time complexity will be $\mathcal{O}(\log_2 N)$ as claimed.

To remove a node with the minimum key ("min-removal"), we first swap the root and "last" element (lowest level, furthest to the right). Then pop the last element. Compare the new root with children, and swap with whichever child is smaller. Continue comparing with children and swapping with smaller child until both children are larger, and the ordering satisfies the heap property. The cost is again $\mathcal{O}(\log_2 N)$ (the reasoning is very similar to that used for insertion).

The heapq module in Python allows us to build ordered lists in linear time and provides $\mathcal{O}(\log_2 N)$ "min-removal" and insertion. It does not provide $\mathcal{O}(\log_2 N)$ key modification (which is needed to update provisional distances). This require additional work, and the use of a binary heap with Dijkstra's method in Python is manageable but not completely straightforward. Assuming we have $\mathcal{O}(\log_2 N)$ 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 and:

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*.

The overall cost of step 1 is $\mathcal{O}(N \log_2 N)$, the overall cost of step 2 is $\mathcal{O}(\mathrm{L}\log_2 N)$, and the overall cost of step 3 is $\mathcal{O}(N \log_2 N)$. You should decide if/when this is better than our initial dictionary-based implementation.

A few final general notes on the graph search problem. First, there are other algorithms available for weighted graphs with negative weights. Second, NetworkX provides shortest-path functions (as do other Python packages), so it is not essential to write your own graph search codes. It *is* important to understand the strengths, weaknesses and cost

of the underlying algorithms used by packages like NetworkX (especially when working on large networks). For graph search and many other problems, it is also important to think about how data is organized (stacks, queues, heaps). And finally, note that not all networks are best represented as NetworkX graphs! A brief overview of the graph search algorithms we have discussed is given below.

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
- $\mathcal{O}\left(N^2\right)$ 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

Lecture 9

Computational science & random walks

9.1 Computational science

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. However, these kinds of problems require huge amounts of computational power, so we will start with simpler 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 we sill take a different approach and use existing functions and Python modules without analyzing them in detail. I want you to build a picture of how scientific reasoning, mathematical models, and computational tools can be understood and used collectively. We will first consider simulations of both stochastic and deterministic dynamical processes which evolve in time from some given initial condition. 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 sort of analysis can take many forms:

- Simple visualization
- Compute 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 we can store "information" for efficient computation.

9.2 Simulating 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 $= 1/2$. 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. However, it is easier to use `np.random.choice`. The code below shows how this function can be used to simulate 10 coin flips:

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

We now have what we need to assemble Python code to simulate random walks. We will set $\Delta x = 1$ for convenience. Here is a simple implementation of an `Nt`-step random walk starting from $X = 0$:

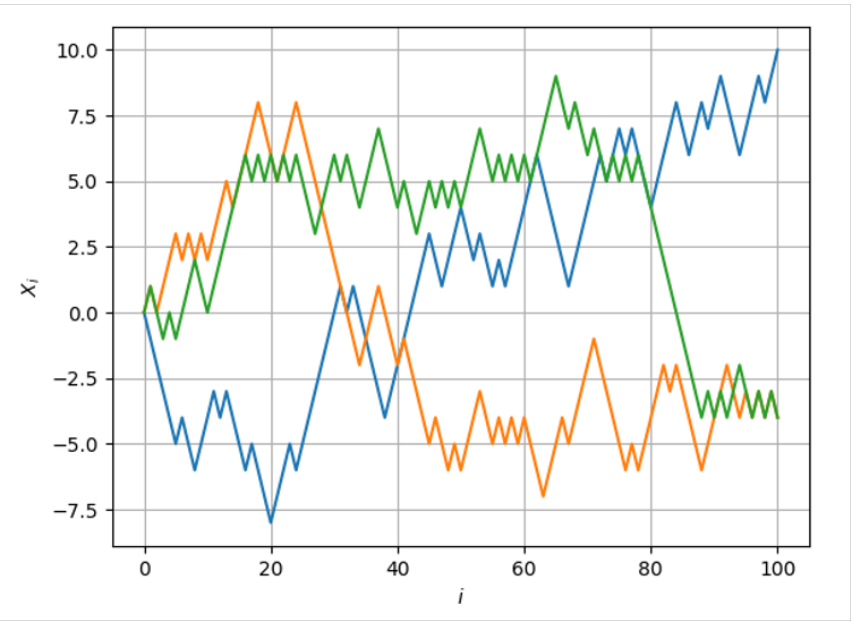


Figure 9.1: Three simulated random walks

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

This code generates trajectories like those shown in Figure 9.1. 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 of the position. The following code simulates M Nt-step random walks:

```
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 $s_i = \sqrt{\overline{X_i^2} - \bar{X}_i^2}$):

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

Note that an overbar indicates averaging over the M simulation. 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 $\overline{X_i^2}$ should converge to $\langle X^2 \rangle$. It

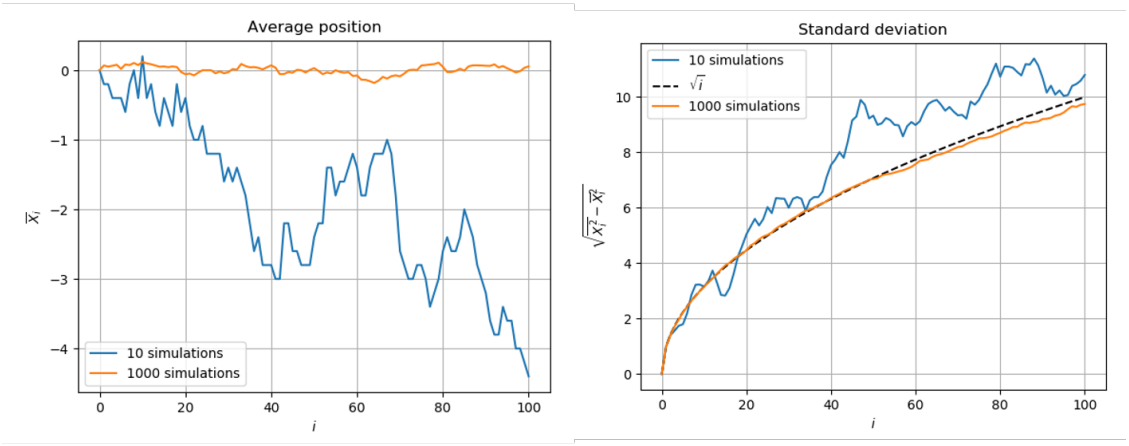


Figure 9.2: Averaged results from several random walk simulations

is straightforward to derive expressions for these expectations using our update equation, $X_{i+1} = X_i + R_i$. Note that:

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

and, using linearity of expectation,

$$\langle X_{i+1} \rangle = \langle X_i \rangle + \langle R_i \rangle = \langle X_i \rangle .$$

From the above, we see that the expected position is constant. If we require, $X_0 = 0$, we have $\langle X_i \rangle = 0$ for all i, and we expect $\bar{X}_i \approx 0$ if M is sufficiently large. Then, squaring both sides of the update equation, taking the expectation of both sides, and using linearity of expectation gives:

$$\left\langle X_{i+1}^2 \right\rangle = \left\langle X_i^2 \right\rangle + 2 \langle X_i R_i \rangle + \left\langle R_i^2 \right\rangle .$$

Note that R_i^2 is always 1, so $\left\langle R_i^2 \right\rangle = 1$. 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, $\left\langle X_{i+1}^2 \right\rangle = \left\langle X_i^2 \right\rangle + 1$, and, by extension, $\left\langle X_{i+1}^2 \right\rangle = i + 1$. It is convenient to restate this as $\left\langle X_i^2 \right\rangle - \langle X_i \rangle^2 = i$ and we expect $s_i \approx \sqrt{i}$ when M is sufficiently large. We use $M = 10$ and $M = 1000$, and the results are shown in Figure 9.2. 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, an outer loop with M iterations, and an inner loop with Nt iterations. Can we make any useful modifications? We have already seen that `np.random.choice` can generate several random numbers at once, so let's pre-compute 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 a 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)
```

The function, *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 using numpy slicing:

```
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)
```

The speed has increased by a further 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 helpful: https://numpy.org/doc/stable/reference/routines.html

Lecture 10

Analyzing deterministic initial value problems

10.1 Simple 2-equation 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). We will analyze the following system:

$$\begin{aligned} \tfrac{dx}{dt} &= ax - bxy \\ \tfrac{dy}{dt} &= -y + bxy \\ x(0) &= x_0, y(0) = y_0. \end{aligned}$$

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:

1. Are there equilibrium solutions where $\frac{dx}{dt} = \frac{dy}{dt} = 0$?
2. What happens if one or more parameters is zero?
3. 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 2000.

Question 1: Are there equilibrium solutions where $\frac{dx}{dt} = \frac{dy}{dt} = 0$?
We need to find solutions of:

$$\begin{aligned} \frac{dx}{dt} &= 0 = ax - bxy \\ \frac{dy}{dt} &= 0 = -y + bxy \end{aligned}$$

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

Question 2: What happens if one or more parameters is zero?
Setting $b = 0$ 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 happily reproduce with the population growing 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 $x = 1/b, y = a/b$. There is more than one way to approach this question. We could consider a Taylor series expansion about this equilibrium which leads to consideration of the Jacobian of the right-hand side of the model equations, but I will take a different route to the same result. First, introduce perturbation series expansions for x and y :

$$x = \frac{1}{b} + \epsilon x_1 + \epsilon^2 x_2 + \cdots$$
$$y = \frac{a}{b} + \epsilon y_1 + \epsilon^2 y_2 + \cdots$$

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

$$\epsilon \frac{dx_1}{dt} = \epsilon a x_1 - \epsilon b \left(x_1 \frac{a}{b} + \frac{1}{b} y_1 \right) + \mathcal{O}\left(\epsilon^2\right)$$
$$\epsilon \frac{dy_1}{dt} = -\epsilon y_1 + \epsilon b \left(x_1 \frac{a}{b} + \frac{1}{b} y_1 \right) + \mathcal{O}\left(\epsilon^2\right)$$

Here, $\mathcal{O}\left(\epsilon^2\right)$ represents terms of the form $f(\epsilon)$ or smaller where $f(\epsilon)/\epsilon^2 \to$ constant when $\epsilon \to 0$. Simplifying the equations and dividing by ϵ gives,

$$\frac{dx_1}{dt} = -y_1 + \mathcal{O}(\epsilon)$$
$$\frac{dy_1}{dt} = a x_1 + \mathcal{O}(\epsilon)$$

Then, we let $\epsilon \to 0$ 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 $\mathbf{v}_1$ and $\mathbf{v}_2$. The general solution to our problem is,

$$\mathbf{z} = c_1\mathbf{v}_1 \exp(i\sqrt{a}t) + c_2\mathbf{v}_2 \exp(i\sqrt{a}t),$$

and the constants c_1 and c_2 are chosen so the solution satisfies the initial conditions. Since the eigenvalues are purely imaginary, in this case, small perturbations will oscillate sinusoidally about the equilibrium.

10.2 Generalized predator-prey system

What does the rabbits and foxes model have to do with computing? Let's think about a problem where we have n species instead of 2. We can then consider a generalized version of our model:

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

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 benefiting 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:

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

This is a linear system of equations (since the x_i terms drop out) and such systems are usually straightforward to solve (e.g. by using *np.linalg.solve*). However, if we replace $a_i x_i$ with a more realistic non-linear term, $f(a_i, x_i)$, we will 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 includes two examples. Note that the Jacobian is not usually explicitly specified for large problems though it is specified in the first example.

How do we analyze small perturbations to an equilibrium state for the generalized model? Let's say that the equilibrium state is $\bar{x}_i, i = 1, 2, \ldots, N$, and we'll write our expansion in a slightly different way from what we did before: $x_i = \bar{x}_i + \epsilon\tilde{x}_i + \mathcal{O}\left(\epsilon^2\right)$. The linearized equations for the perturbations are then,

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

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`. 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]])
```

Returning to the general model, if n linearly independent eigenvectors are found, the general solution can be written as,

$$\tilde{\mathbf{x}} = c_1\mathbf{v}_1\exp\left(\lambda_1 t\right) + c_2\mathbf{v}_2\exp\left(\lambda_2 t\right) + \cdots + c_n\mathbf{v}_n\exp\left(\lambda_2 t\right)$$

Applying the initial conditions then gives a system of linear equations which can be solved using `np.linalg.solve` to find the constants $c_1, c_2, \ldots, c_n$. But are `np.linalg.eig` and `np.linalg.solve` the right tools to use? It depends on the details of the problem. There are multiple Python functions that can be used to compute eigenvalues and eigenvectors. 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 and to understand when they should be used.

10.2.1 Sparse matrices

When deciding which function(s) to use, a key point to consider is the structure of the matrix, $\mathbf{M}$. 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. The package, 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. The following is an example taken from the documentation for scipy.sparse:

```
>>> # Constructing a matrix using ijv 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]])
```

The ith elements in `row` and `column` specify the location of a non-zero element in the matrix, and the ith element in `data` provides the corresponding value (e.g. the (3,3) element in the matrix is 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: 1) we can save memory as we are not storing large numbers of zeros, and 2) 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. Additionally, it is helpful to look carefully at the guidance at the bottom of the main documentation page: https://docs.scipy.org/doc/scipy/reference/sparse.html

10.3 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 usually 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:

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

with initial conditions at $t = 0$ specified. The basic idea is to discretize time, $t = 0, \Delta t, 2\Delta t, \ldots, N_t * \Delta t$, and starting from $x_i(t = 0)$, march forward in time and compute $x_i(\Delta t), x_i(2\Delta t), \ldots x_i\left(N_t * \Delta t\right)$. We can design such an approach using a Taylor series expansion:

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

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

It is important to examine the accuracy and stability of this method:

Accuracy: The approximation error for a single step is $\mathcal{O}\left(\Delta t^2\right)$, however the error will accumulate, and the convention is to say the global error is $\sim \mathcal{O}(\Delta t)$. A smaller time step will give a 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* which means that 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.

Numerical stability is often a substantial concern for nonlinear systems. These systems frequently contain rapidly varying components which affect stability much more than accuracy. In such cases, it is often helpful to use implicit methods:

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

Note that we now have $x_j(t_0 + \Delta t)$ on the RHS rather than $x_j(t_0)$. Returning to accuracy and stability, the global error is again $\sim \mathcal{O}(\Delta t)$, and we know generally that reducing the time step will give more accurate solutions. Applying the method to, $\frac{dx}{dt} = ax, a < 0$, the method is unconditionally stable which means that the method will give a 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. An example illustrating how to use the two Euler methods to find a solution for a system of two ODEs at time $t_0 + \Delta t$ given the solution at $t = t_0$ is shown below.

Example illustrating explicit and implicit Euler methods

Model problem:

$$\begin{aligned}
&\frac{dx_1}{dt} = -ax_2 \\
&\frac{dx_2}{dt} = -bx_1 \\
&x_1(0) = c, x_2(0) = d \\
&a > 0, b > 0
\end{aligned}$$

Explicit Euler update equation:

$$\begin{aligned}
x_1(t_0 + \Delta t) &= x_1(t_0) - a\Delta t x_2(t_0) \\
x_2(t_0 + \Delta t) &= x_2(t_0) - b\Delta t x_1(t_0)
\end{aligned}$$

Implicit Euler update equation:

$$\begin{aligned}
x_1(t_0 + \Delta t) &= x_1(t_0) - a\Delta t x_2(t_0 + \Delta t) \\
x_2(t_0 + \Delta t) &= x_2(t_0) - b\Delta t x_1(t_0 + \Delta t)
\end{aligned}$$

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

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, though we will see the explicit Euler method again when we return to stochastic processes. 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. In these methods, 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 $\left| x^{(m)}_{i,\text{ exact}} - x^{(m)}_{i,\text{ computed}} \right|$ (here $\left| x^{(m)}_i \right|$ is the length of the vector $\left[x^{(m)}_1, x^{(m)}_2, \ldots \right]^T$)
- The user specifies tolerances for the absolute and relative errors $(\epsilon_{abs}, \epsilon_{rel})$
- The method then selects the sub-step size so that $E^{(m)} \leq \epsilon_{abs} + \left| x^{(m)}_{i,computed} \right| \epsilon_{rel}$

The scipy.integrate package 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 a "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. For any of these methods, the user has to specify 1) initial conditions, 2) the time span, and 3) a function which computes the RHS of the the ODEs. The user can also choose to specify error tolerances, the times at which to return the solution, as well as many other parameters. Look at the online documentation!

Lecture 11

Simulating stochastic processes

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 modifications, will be useful here. 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.

11.1 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, $\left\langle X_i^2 \right\rangle - \langle X_i \rangle^2 = i$ 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. For example, $\bar{X}_i \approx \langle X_i \rangle$ 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, $\left\langle X_i^2 \right\rangle - \langle X_i \rangle^2 = i\Delta x^2$. Now, say that each step requires time δt, and $t_i = i\delta t$. We then have, $\left\langle X_i^2 \right\rangle - \langle X_i \rangle^2 = t_i \left(\frac{\Delta x^2}{\delta t} \right)$. 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 is that the variance increases linearly with time. This is a very general result that has been observed for many different stochastic processes.

11.2 Discrete Brownian motion

We should of course think critically about the random walk model. Generally, particles will move in three dimensions rather than one, 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 addresses 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.

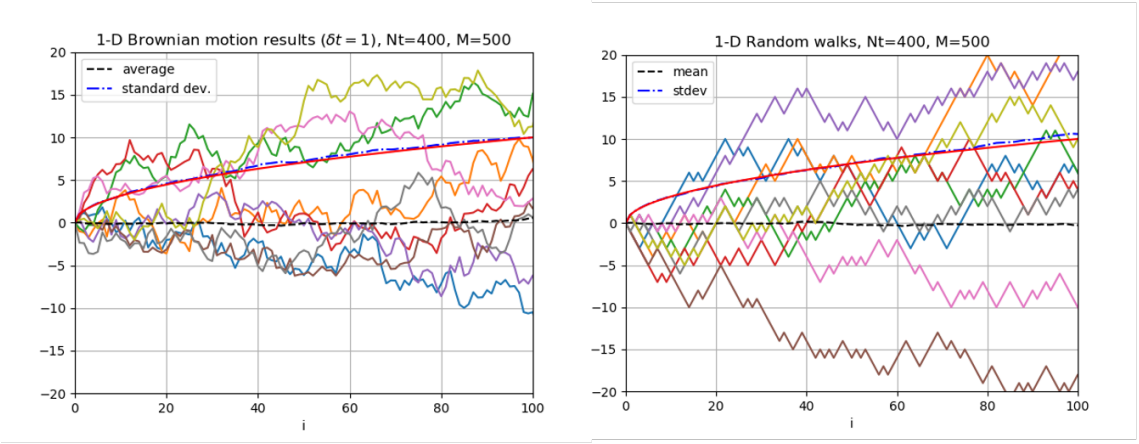


Figure 11.1: Simulations results for Brownian motion (left) and ramdom waks (right)

It is straightforward to modify our random walk code to simulate Brownian motion – essentially, we just replace `np.random.choice` with `np.random.normal`. As we can see in Figure 11.1, the computed results look somewhat similar to what we had with random walks which is unsurprising given the expected values we have found for both models. However, we do see that the Brownian trajectories are less "choppy" than the random walks since a range of stepsizes are used instead of just ± 1. Brownian motion 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?

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:

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

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 stochastic processes. It is natural to think about what happens when $\delta t \to 0$. We could rearrange the 1st equation in Langevin's model 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 $\delta t \to 0$ and $t = t_i$. But what about the second equation, 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 $\delta t \to 0$! 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 Oksendaal (there are many other good books as well).

11.3 Stochastic differential equations

We will now consider a stochastic process over the interval $[0, T]$, and we discretize time as $t_i = i\delta t$ with $T = n\delta t$. 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:

$$X(T) - X(0) = \int_0^T f(X(s))ds + \int_0^T g(X(s))dW(s). \tag{11.1}$$

This tells us how X changes from $t = 0$ to $t = T$ for a model system defined by $f(X)$ and $g(X)$. The first integral is exactly the same as what we have for ODEs (i.e. for $dx/dt = f(x)$). The second equation requires interpretation. We will use the Ito integral:

$$\int_0^T g(X(s))dW(s) = \lim_{\delta t \to 0} \sum_{i=0}^{n} g\left(X\left(t_i\right)\right)\left[W\left(t_{i+1}\right) - W\left(t_i\right)\right] \tag{11.2}$$

with $t_i = i\delta t, T = n\delta t$. Here, $W\left(t_{i+1}\right) - W\left(t_i\right)$ is a "Brownian step":

$$W\left(t_{i+1}\right) - W\left(t_i\right) = \sqrt{\delta t} B_i. \tag{11.3}$$

Ordinary integrals can be defined using Riemann sums where an interval is divided into sub-intervals, and each sub-interval is filled with a rectangle whose height is determined the integrand's value at a point in the sub-interval. For ordinary integrals, it doesn't

matter which point in the sub-interval you choose, however for stochastic integrals, it does matter, and with the Ito integral, the "left" end of the sub-interval is used. Other choices are possible (most notably, the *Stratanovich integral*), but we will only consider the Ito integral here.

Having defined a family of stochastic differential equations, the question now is: how do we compute solutions given the functions $f(X)$ and $g(X)$? 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 previously, and here we will use the *Euler-Maruyama* (E-M) method to compute solutions over the interval, $[0, T]$. As part of this method, we will introduce 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, and in most other situations we can just set $a = 1$, and not worry about δt. Given the solution at time τ_j, we compute the solution at τ_{j+1} using:

$$X_{j+1} = X_j + \Delta t f\left(X_j\right) + g\left(X_j\right)\left[W\left(t_{j+1}\right) - W\left(t_j\right)\right]. \tag{11.4}$$

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

$$W\left(t_i + \delta t\right) = W\left(t_i\right) + \sqrt{\delta t} B_i \quad (W(0) = 0). \tag{11.5}$$

You should notice a clear correspondence between the above two equations and (11.1) and (11.3).

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 ($\mu = 0$). The solution for the full problem is:

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

The appendix provides a sketch of how to verify that this is a solution of the SDE. 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 appendix). 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:

$$X_{j+1} = (1 + \lambda \Delta t) X_j + \mu X_j \left[W\left(t_{j+1}\right) - W\left(t_j\right)\right] \tag{11.6}$$

$$W\left(t_i + \delta t\right) = W\left(t_i\right) + \sqrt{\delta t} B_j \tag{11.7}$$

Note that 4 Brownian motion steps are needed to compute $W\left(t_{j+1}\right)$ given $W\left(t_j\right)$ The basic plan for the code is:

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.

Code for step 1 is shown below:

```
#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
```

Here, `T` is the time span, `l` is the model parameter λ; `mu` is μ, and `X0` is the initial condition. The numerical parameters set: the number of simulations (`M`), the number of Brownian steps (`nt`), the Brownian stepsize δt (`dt`), the number of X updates (`Nt`), the stepsize Δt (`Dt`), and `fac`, a term which appears is the E-M update equation $(1 + \lambda \Delta t)$. For step 2, we have:

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

This is very similar to random walk simulations, but we have not used an `nt`-step loop to update `W`. We have used the cumulative sum function `np.cumsum` instead which is more concise. After applying this function, `W` will contain, `Bterm[0],Bterm[0]+Bterm[1], Bterm[0]+Bterm[1]+Bterm[2], ...` which can be restated as, `Bterm[0],W[0]+Bterm[1], W[1]+Bterm[2], ....` From line 3, we can see that the ith term in `Bterm` corresponds to $\sqrt{\delta t} B_i$ where B_i is sampled from a standard normal distribution. We set $W(t = 0) = 0$, and then the first term in `W` corresponds to $W(t = \delta t)$. More generally, `W[i]` corresponds to $W(t = (i + 1)\delta t)$ which is what we need to update $X(t)$. The code for step 3:

```
#Apply E-M method for M Nt-step simulations
X = np.zeros((Nt+1,M))
X[0,:] = X0
```

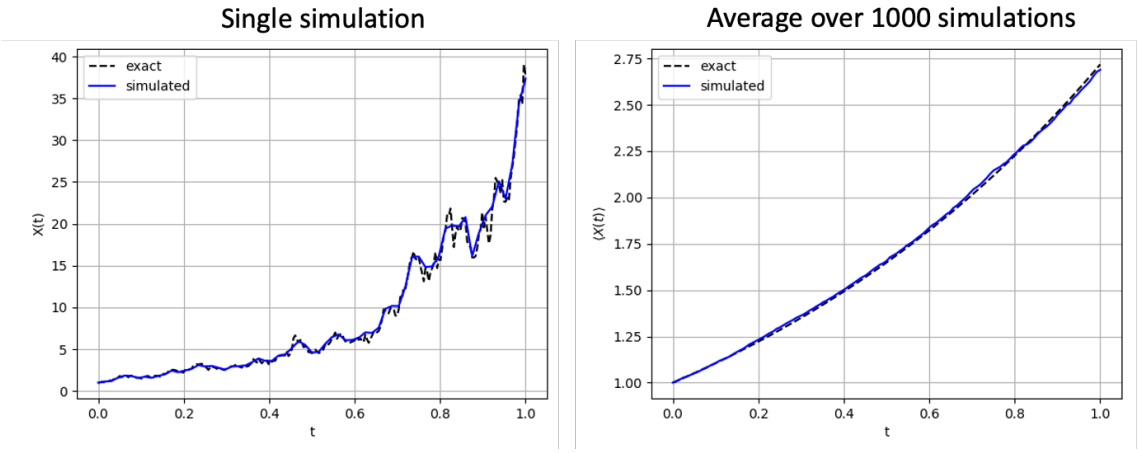


Figure 11.2: Single simulation of model SDE (left) and average over 100 simulations (right).

```
4 dW = W[a::a,:]-W[:-a:a,:]
5 #Iterate over Nt time steps
6 for j in range(Nt):
7     X[j+1,:] = fac*X[j,:]+mu*X[j,:]*dW[j,:] #E-M update equation
```

We must carefully slice `W` when computing `dW` to account fot the fact that $\Delta t = a\delta t$. You should compare line 7 with (11.6) noting how `fac` is defined.

Results for a single simulation are shown in Figure 11.2 on the left while on the right, we compare the ensemble average over 1000 simulations to the expectation, $\langle X(T)\rangle = X_0 \exp(\lambda T)$ These comparisons indicate that the code is correct. Let's now look more closely at the accuracy of the E-M method. We stated previously 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. First define the error as $\epsilon_w(T) = \left|\bar{X}_j - \langle X(T)\rangle\right|$ with $T = j\Delta t$. It can be shown that $\epsilon_w(T) = \mathcal{O}(\Delta t)$. 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 how close the sample average is to the expectation. What about the error of individual simulations? 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 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. Figure Figure 11.3 shows $\epsilon_w(T)$ and our approximation for the "strong error": $\overline{|X_j - X(\tau)|}$. 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

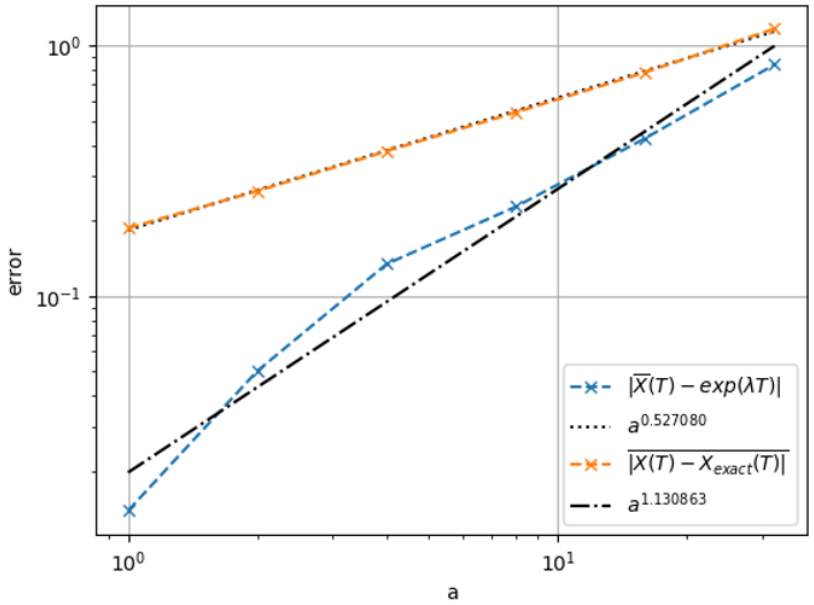


Figure 11.3: Weak and strong error of E-M method applied to model linear SDE.

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 model 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: $\lim_{T\to\infty} \langle X(T)^2 \rangle = 0 \leftrightarrow \mathcal{R}(\lambda) + \frac{1}{2}|\mu|^2 < 0$

asymptotic stability: $P\left(\lim_{T\to\infty} |X(T)| = 0\right) = 1 \leftrightarrow \mathcal{R}\left(\lambda - \frac{1}{2}\mu^2\right) < 0$

These results tell when we can expect the expected value of the exact solution and individual realizations to be bounded at long times. There are two corresponding types of numerical stability:

mean-square numerical stability: $\lim_{j\to\infty} \left\langle X_j^2 \right\rangle = 0 \leftrightarrow |1 + \Delta t\lambda|^2 + \Delta t|\mu|^2 < 1$

asymptotic numerical stability:

$P\left(\lim_{j\to\infty}|X_j|=0\right)=1 \leftrightarrow \langle \log|1+\Delta t\lambda+\sqrt{\Delta t}\mu\mathcal{N}(0,1)|\rangle<0$

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 when the exact solution is bounded if the time step is too large.

11.4 Appendix: Notes on Brownian motion and the model linear SDE

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 $0\le s<t\le T, W(t)-W(s)\sim\sqrt{t-s}\mathcal{N}(0,1)$,
- for $0\le s<t<u<v\le T, W(t)-W(s)$ 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 2^{nd} derivatives. It follows from Ito's lemma that:

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

For our example SDE, $h(t,x)=X(0)\exp\left[\left(\lambda-\frac{1}{2}\mu^2\right)t+\mu x\right]$. 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 $X(T)=X(0)\exp\left[\left(\lambda-\frac{1}{2}\mu^2\right)T+\mu W(T)\right.$ is a solution to this equation.

Lecture 12

Maximum growth

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).

12.1 The maximum growth problem

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 $\mathbf{x}$. If the system was overdetermined with $m > n$, 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 $\mathbf{B} = \mathbf{VSV}^T$ where:

- $\mathbf{V}$ is an orthogonal matrix $\left(\mathbf{V}^T\mathbf{V} = \mathbf{I}\right)$ 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, $\mathbf{A}^T\mathbf{A} = \mathbf{VSV}^T$, and $\mathbf{b}^T\mathbf{b} = \mathbf{x}^T\mathbf{VSV}^T\mathbf{x}$. Now, let $\mathbf{z} = \mathbf{V}^T\mathbf{x}$. We then have $\mathbf{b}^T\mathbf{b} = \mathbf{z}^T\mathbf{Sz}$. Say that the (real) eigenvalues are arranged so that $S_{ii} = \lambda_i$ and $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Then,

$$\mathbf{z}^T\mathbf{Sz} = \lambda_1 z_1^2 + \lambda_2 z_2^2 + \cdots + \lambda z_n^2 \leq \lambda_1 \left(z_1^2 + z_2^T + \cdots z_n^2\right).$$

Here, $\mathbf{z} = [z_1, z_2, \ldots, z_n]^T$. Our inequality can be comncisely stated as, $\mathbf{b}^T\mathbf{b} \leq \lambda_1\mathbf{z}^T\mathbf{z}$. We know that $\mathbf{z}^T\mathbf{z} = \mathbf{x}^T\mathbf{VV}^T\mathbf{x} = \mathbf{x}^T\mathbf{x}$, so we can restate the inequality as, $\mathbf{b}^T\mathbf{b} \leq \lambda_1\mathbf{x}^T\mathbf{x}$ or $|\mathbf{b}|^2 \leq \lambda_1|\mathbf{x}|^2$.

Task 2: Find x so that $|\mathbf{b}|^2 = \lambda_1|\mathbf{x}|^2$ This task is easier, we just let $\mathbf{x} = \mathbf{v}_1$ where $\mathbf{A}^T\mathbf{Av}_1 = \lambda_1\mathbf{v}_1$ (i.e. $\mathbf{v}_1$ is the leading eigenvector of $\mathbf{A}^T\mathbf{A}$). Then, $|\mathbf{x}|^2 = 1$ and $|\mathbf{b}|^2 = |\mathbf{Av_1}|^2 = \lambda_1\mathbf{v}_1^T\mathbf{v}_1 = \lambda_1$ 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:

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{Mx},\ \mathbf{x}(t=0) = \mathbf{x}_0.$$

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 $\mathbf{A}\left(t^*\right)^T\mathbf{A}\left(t^*\right)$, 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, $\mathbf{x}^{(i+1)} = \mathbf{Mx}^{(i)}$. Here, the superscript indicates an iteration number, and $\mathbf{x}^{(p)} = \mathbf{M}^p\mathbf{x}^{(0)}$, so we have another maximum growth problem with $\mathbf{A} = \mathbf{M}^p$.

Previously, when we encountered systems of linear ODEs of the form, $\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{Mx}, \mathbf{x}(t = 0) = \mathbf{x}_0$, 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 $\mathbf{M}$:

$$\mathbf{x}(t) = C_1\mathbf{v}_1e^{\lambda_1 t} + C_2\mathbf{v}_2e^{\lambda_2 t} + \cdots + C_n\mathbf{v}_ne^{\lambda_n t}$$

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:

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

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 $\mathrm{e}^{\lambda_1 t^*} = \mathrm{e}^{-1}$ with $\mathbf{x}(t=0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. This is minimal decay rather than maximum growth!

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

12.2 Computing eigenvalues and eigenvectors

To find solutions to the maximum growth problem, eigenvalues and eigenvectors of $\mathbf{F} = \mathbf{A}^T\mathbf{A}$ need to be computed. Functions like `np.linalg.eig` typically use variations of the QR algorithm which I will now briefly outline. The basic idea behind the method is to 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 $\mathbf{F}$. Here is the 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.
- 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: $O\left(n^3\right)$ if $\mathbf{F}$ is $n \times n$ (consider the cost of computing the product of two $n \times n$ matrices).

The eigenvalues and eigenvectors of $\mathbf{F} = \mathbf{A}^T\mathbf{A}$ can also be found from the singular value decomposition (SVD) of $\mathbf{A}$. Any real $m \times n$ matrix, $\mathbf{A}$, can be decomposed as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^T$, where:

- $\mathbf{U}$ 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 $\mathbf{A}^T\mathbf{A}$
- $\mathbf{\Sigma}$ 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 $\mathbf{A}\mathbf{A}^\mathbf{T}$ ($\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ share the same set of nonzero eigenvalues, $\sigma_1^2 \geq \sigma_2^2 \geq \cdots \geq \sigma_R^2 > 0$ with $R \leq \min(m, n)$).

The σ_i are the *singular values* of $\mathbf{A}$ and are usually arranged in non-increasing order, and the SVD is one of the most useful tools from applied linear algebra. If $\mathbf{A}$ is complex, its SVD is instead, $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^H$ and the transposes in the discussion above become conjugate transposes.

To see how $\mathbf{U}$ is related to $\mathbf{A}\mathbf{A}^T$, start with $\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^T\mathbf{W}\mathbf{\Sigma}^T\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T$, and $\mathbf{\Sigma}\mathbf{\Sigma}^T$ is an $m \times m$ diagonal matrix containing the eigenvalues of $\mathbf{A}\mathbf{A}^T$. We can then recognize $\mathbf{U}$ as the eigenvector matrix in the orthogonal diagonalization of $\mathbf{A}\mathbf{A}^T$.

The columns of $\mathbf{U}$ and $\mathbf{V}$ are also related to each other. Let $\mathbf{u}_i$ be the i^{th} eigenvector of $\mathbf{A}\mathbf{A}^T$ and let $\mathbf{w}_i$ be the i^{th} eigenvector of $\mathbf{A}^T\mathbf{A}$. Then, $\mathbf{A}\mathbf{w}_i = \sigma_i\mathbf{u}_i$. 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 time complexity is also similar: $O\left(m^2n\right) + O\left(n^3\right)$. So the time complexities 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

In [23]: s[0]**2
Out[23]: 3195.4726880276557
```

The SVD calculation is about 40% faster here. Note that `scipy.sparse.linalg` should be

used if the matrices are sparse, and, there is a more efficient version of QR for symmetric matrices which can be faster than the SVD (though it doesn't compute U).

The discussion of maximum growth has generally assumed that the elements of $\mathbf{A}$ are real, but the same ideas apply if its elements are complex-valued. Then, instead of transposes, we use conjugate transposes and take advantage of the properties of Hermitian matrices $\left(\mathbf{F} = \mathbf{A}^H\mathbf{A}, \mathbf{F}^H = \mathbf{F}\right)$ 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.

Notation for matrices and vectors

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 ith vector in contrast with v_i, the ith 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:

$$\sum_{j=1}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, m$$

Lecture 13

Matrix computations and low-rank approximation

In this lecture, we will continue our look at "matrix computations". Motivating applications include:

- linear(ized) dynamical systems:

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

- 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 at long times), 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 $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^{\mathrm{T}}$. With complex $\mathbf{A}$, the transpose is replaced with the conjugate transpose: $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{W}^{\mathrm{H}}$. If $\mathbf{A}$ is $m \times n$, then $\mathbf{U}$ is $m \times m$, $\mathbf{W}$ is $n \times n$, and $\mathbf{\Sigma}$ is a non-negative $m \times n$ diagonal matrix. The matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$ have the same non-zero eigenvalues, and the singular values of $\mathbf{A}$ are the square-roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$.

13.1 Revisiting maximum growth

The 1st columns of $\mathbf{W}$ and $\mathbf{U}$ and the largest singular value in $\mathbf{\Sigma}$ 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.

13.2 Matrices as data tables

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? The first step involves data collection or acquisition. This could mean, lab measurements, accessing online databases (e.g. Kaggle), or obtaining 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. However, we 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 may be "hiding" in the data?

Images are also naturally represented as matrices. Consider this jpeg image of a confused beagle. This corresponds to three 360×326 numpy arrays containing integers between 0 and 255. The three arrays correspond to the colors red, green, and blue, while elements of the matrices with smaller values are "darker" and larger values correspond to "lighter" pixels. 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.

13.3 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 $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$ with $\sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_n = 0$ if $r < n$, then $\text{rank}(\mathbf{A}) = r$. Or in simpler terms, compute the SVD of $\mathbf{A}$, ($\mathbf{A} = \mathbf{U\Sigma W}^T$), and the rank of $\mathbf{A}$ is the number of non-zero elements in $\mathbf{\Sigma}$. 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 $\mathbf{A} = \mathbf{BC}$ as a collection of inner products:

$$A_{ij} = \sum_{k=1}^{n} B_{ik}C_{kj} = \boldsymbol{\beta}_i^T \mathbf{c}_j$$

where $\boldsymbol{\beta}_i^{\mathrm{T}}$ is the ith row of $\mathbf{B}$, and $\mathbf{c}_j$ is the jth column of $\mathbf{C}$. However, it can be useful to view this as a sum of outer products: $\mathbf{A} = \mathbf{b}_1\boldsymbol{\gamma}_1^T + \mathbf{b}_2\boldsymbol{\gamma}_2^T + \cdots + \mathbf{b}_n\boldsymbol{\gamma}_n^T$ 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 $m \times l$:

$$B = \begin{bmatrix} \vdots & \vdots & \vdots & & \vdots \\ \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 & \cdots & \mathbf{b}_l \\ \vdots & \vdots & \vdots & & \vdots \end{bmatrix} = \begin{bmatrix} \cdots & \boldsymbol{\beta_1}^T & \cdots \\ \cdots & \boldsymbol{\beta_2}^T & \cdots \\ & \vdots & \\ \cdots & \boldsymbol{\beta_m}^T & \cdots \end{bmatrix}$$

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, $\mathbf{A} = \mathbf{U\Sigma W}^T$. 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 ith 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 will simply state that it will indeed be r.

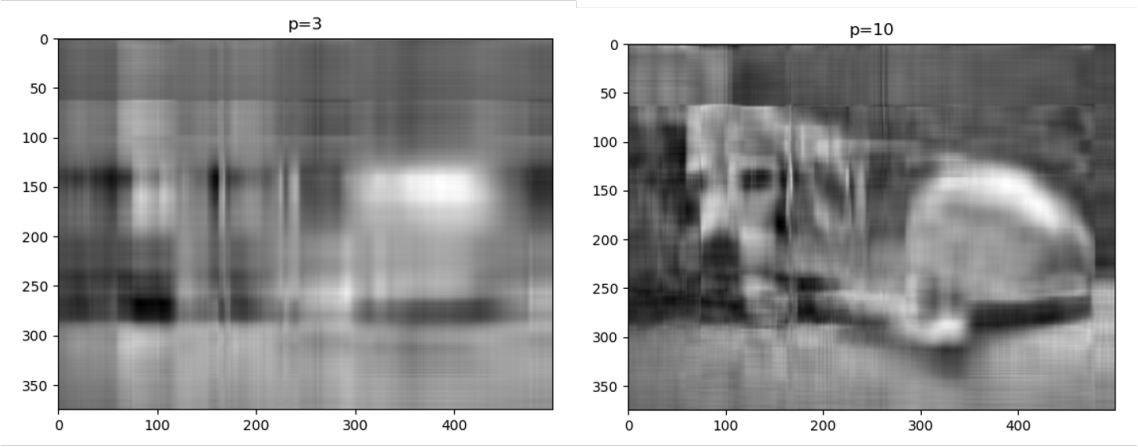


Figure 13.1: Low-rank approximations to image matrix

Recall that the singular values are ordered, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$. 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 [57]: U.shape
In [58]: S.shape
In [59]: WT.shape
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 $\mathbf{A}_p$. As we can see in Figure 13.1, when $p = 3$, too much important "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 Figure 13.2, $p \approx 25$ seems like a sensible next choice, and we can see in Figure 13.3 that we now have a recognizable reproduction of the original 375×500 matrix. It is important to recognize that 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: $|\mathbf{A}|_F^2 = \sum_{i=1}^m \sum_{j=1}^n (A_{ij})^2$,

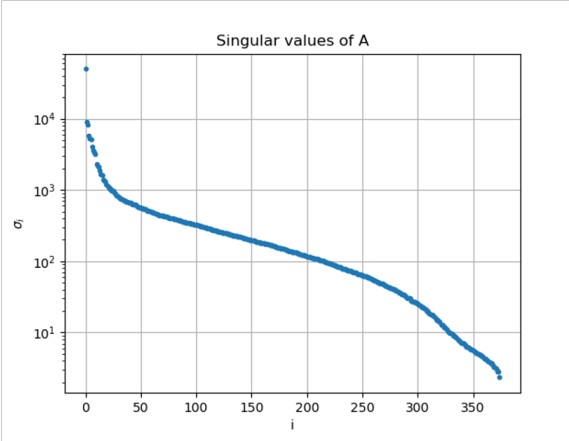


Figure 13.2: Singular values of image matrix

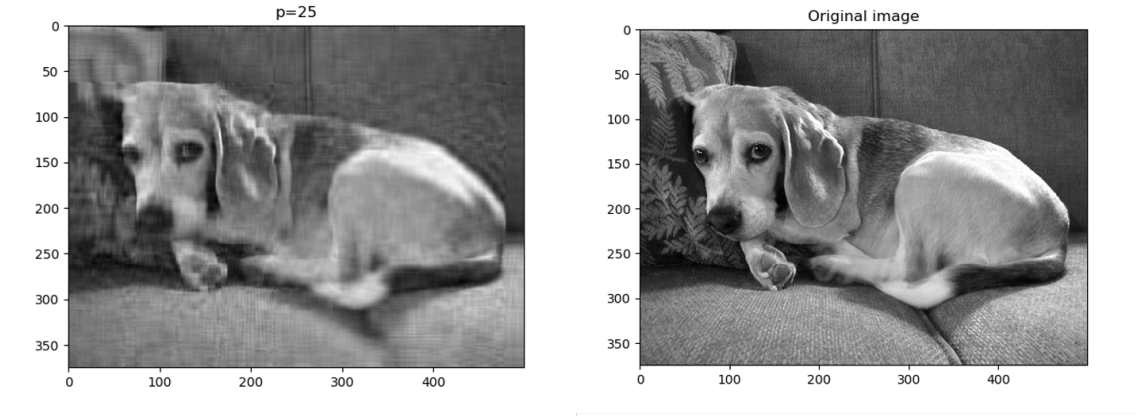


Figure 13.3: Low rank approximation of image matrix (left) and original image (right).

or, in matrix-vector form: $|\mathbf{A}|_F^2 = \text{trace}\left(\mathbf{A}^T\mathbf{A}\right)$, and since the trace of a matrix is equal to the sum of its eigenvalues we know that,

$$|\mathbf{A}|_{\mathrm{F}}^2 = \sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij})^2 = \text{trace}\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right) = \sum_{i=1}^{r}{\sigma_i}^2.$$

What is the Frobenius norm for a rank-1 approximation of $\mathbf{A}$?

$|\mathbf{A}_1|_F = \text{trace}\left(\left(\mathbf{w}_1\sigma_1\mathbf{u}_1^T\right)\left(\sigma_1\mathbf{u}_1\mathbf{w}_1^{\mathrm{T}}\right)\right)$, and the orthogonality of $\mathbf{U}$ tells us $\mathbf{u}_1^T\mathbf{u}_1 = 1$, so $|\mathbf{A}_1|_F = \sigma_1^2\,\text{trace}\left(\mathbf{w}_1\mathbf{w}_1^T\right)$. The orthogonality of $\mathbf{W}$ then gives, $|\mathbf{A}_1|_F = \sigma_1^2$.

With some more (tedious) work, this can be generalized to, $|\mathbf{A}_p|_F^2 = \sum_{i=1}^{p}{\sigma_i}^2$. 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 $|\mathbf{A} - \mathbf{A}_\mathrm{p}|_F$, and here we will rely on the Eckart-Young theorem which we will simply state:

Problem definition: Find $\mathbf{B}$ such that $|\mathbf{A} - \mathbf{B}|_F$ is minimized and $\mathrm{rank}(\mathbf{B}) \leq p$

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

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

Lecture 14

PCA and Recommender systems

14.1 Principal component analysis

Say you are tracking a meteor heading towards Earth. How many variables do you need to be able to decide how worried to be? We would almost certainly want three position coordinates, $(x(t), y(t), z(t))$. Three velocity components, $(u(t), v(t), w(t))$, could be helpful as well, though we could probably estimate the velocity from the position data. But what if the meteor was just moving in a circular orbit? Then we could use polar coordinates (r, θ), and we could get by with a single variable, $\theta(t)$.

Generally, when working on complex problems, we often encounter datasets with a very large number of variables, and this make 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. Principal component analysis (PCA) is a well-known and widely-used method that attempts to address these questions. It creates new variables with certain "desirable properties" from linear combinations of the original variables.

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:

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

Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ with $\mathbf{y}$ and $\mathbf{z}$ defined analogously. It will be convenient to restate the matrix as,

$$\mathbf{A} = \begin{bmatrix} \cdots & \mathbf{x}^T & \cdots \\ \cdots & \mathbf{y}^T & \cdots \\ \cdots & \mathbf{z}^T & \cdots \end{bmatrix}.$$

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:

$$\begin{aligned}\tilde{\mathbf{x}} &= \alpha_1\mathbf{x} + \beta_1\mathbf{y} + \gamma_1\mathbf{z}\\ \tilde{\mathbf{y}} &= \alpha_2\mathbf{x} + \beta_2\mathbf{y} + \gamma_2\mathbf{z}\\ \tilde{\mathbf{z}} &= \alpha_3\mathbf{x} + \beta_3\mathbf{y} + \gamma_3\mathbf{z},\end{aligned}$$

and we can state this more compactly as,

$$\widetilde{\mathbf{A}} = \mathbf{TA} \text{ where } \mathbf{T} = \begin{bmatrix} \alpha_1 & \beta_1 & \gamma_1 \\ \alpha_2 & \beta_2 & \gamma_2 \\ \alpha_3 & \beta_3 & \gamma_3 \end{bmatrix} \text{ and } \widetilde{\mathbf{A}} = \begin{bmatrix} \cdots & \tilde{\mathbf{x}}^T & \cdots \\ \cdots & \tilde{\mathbf{y}}^T & \cdots \\ \cdots & \tilde{\mathbf{z}}^T & \cdots \end{bmatrix}.$$

Our goal now is to construct the transformation matrix, $\mathbf{T}$, so that it satisfies 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. 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. For example, for a large positive covariance, when one variable increases the other tends to increase as well. If the new variables have zero covariance, we are minimizing redundancy, and each new variable should carry different kinds of "information". 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 $\frac{1}{n-1}\left(\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} + \tilde{\mathbf{y}}^T\tilde{\mathbf{y}}\right)$ and so on. The idea here is that high variance indicates importance. Think of speech vs. background noise in an audio recording. The part of the signal corresponding to the speaker's voice will typically have much larger variance than the background noise. If we ensure that the first new variables have the most variance, we may be able to discard some "low-variance" variables leading to a smaller, more interpretable dataset.

So how do we construct our transformation matrix, $\mathbf{T}$ so that the desirable properties are satisfied? 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.

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,

$$\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} = \mathbf{a}^T\mathbf{S}\mathbf{a} = \lambda_1 a_1^2 + \lambda_2 a_2^2 + \cdots + \lambda_r a_r^2$$

where λ_1 is the ith eigenvalue of $\mathbf{A}\mathbf{A}^T$, and $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_r > 0$. 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: $\frac{\tilde{\mathbf{x}}^T\tilde{\mathbf{x}}}{\mathbf{t}_1^T\mathbf{t}_1} \leq \lambda_1$. 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 $\mathbf{A}\mathbf{A}^\mathbf{T}$, we find: $\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} = \mathbf{v}_1^T\mathbf{A}\mathbf{A}^T\mathbf{v}_1 = \mathbf{v}_1^\mathrm{T}\lambda_1\mathbf{v}_1 = \lambda_1$. 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 $\mathbf{A}\mathbf{A}^\mathrm{T}$ 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 $\frac{1}{n-1}(\lambda_1 + \lambda_2) = \frac{1}{n-1}\left(\sigma_1^2 + \sigma_2^2\right)$ where σ_i is the ith singular value of $\mathbf{A}$. We can verify that desirable property 1 is satisfied with this choice:

$$\tilde{\mathbf{x}}^T\widetilde{\boldsymbol{y}} = \mathbf{v}_1^T\mathbf{A}A^T\mathbf{v}_2 = \mathbf{v}_1^T\mathbf{V}\mathbf{S}\mathbf{V}^T\mathbf{v}_2 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & \lambda_2 & 0 \end{bmatrix}^T = 0.$$

These results generalize as you would guess; the kth new variable is $\mathbf{A}^\mathrm{T}\mathbf{v}_k$ 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 $\widetilde{\mathbf{A}} = \mathbf{T}\mathbf{A}$, and from the above statements, we conclude that $\mathbf{T} = \mathbf{V}^T$. Since $\mathbf{V}$ is the orthogonal eigenvector matrix for $\mathbf{A}\mathbf{A}^\mathrm{T}$, $\mathbf{T}$ is just $\mathbf{U}^T$ from the SVD of $\mathbf{A}$.

Try these exercises to check your understanding of this transformation:

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

The derivation above shows that 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 often called the *principal components* of $\mathbf{A}$ (note however that sometimes the new variables are referred to as principal components instead).

14.1.1 Python implementation

Implementing PCA in Python is straightforward. Given a data matrix stored as a numpy array, we remove the mean from each row, use the SVD to compute the transformation

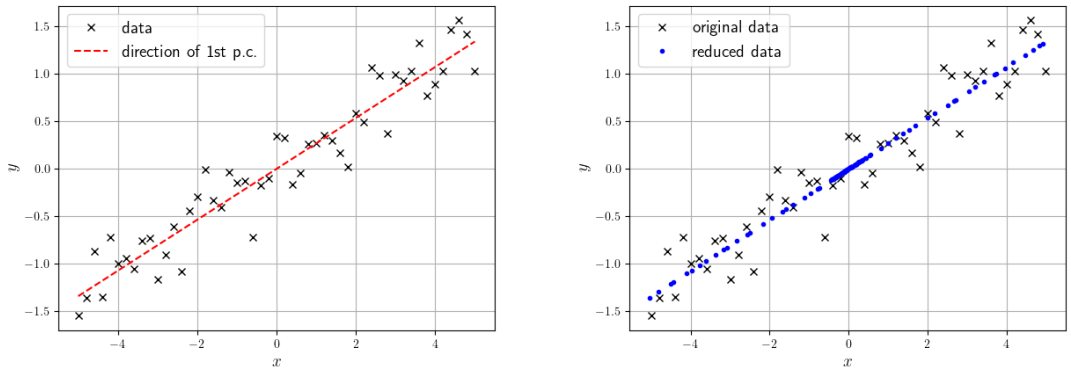


Figure 14.1: PCA results. The figure on the left shows the data and direction of maximum variance. The figure on the right shows the data after dimension reduction.

matrix, $\mathbf{T}$, and then compute the transformed data matrix, $\tilde{\mathbf{A}}$. These steps are shown below for a 2 x 51 matrix whose rows already have zero mean.

```
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 = np.linalg.svd(A)
In [48]: T = U.T
In [49]: Anew = T.dot(A)
In [50]: np.dot(Anew,Anew.T)
Out[50]:
array([[ 8.33602821e+02, -1.18785463e-13],
       [-1.18785463e-13,  1.50729071e+01]])
```

Line 50 outputs the *covariance matrix* for the transformed data. The off-diagonal terms correspond to the covariance of the two new variables and are effectively zero as they should be. We can also see from the diagonal elements that the first new variable has a much larger variance associated with it than the second. Note as well that $\mathbf{S}$ contains the square-roots of the diagonal elements.

The leading eigenvector of $\mathbf{A}\mathbf{A}^T$ (the first column in $\mathbf{U}$) provides information about the direction of maximum variance. Running the following code produces the plot on the left of Figure 14.1:

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

Check your understanding: what is the direction of the 2nd principal component?

PCA provides a straightforward path to dimensionality reduction. For our example with two variables, discard the 2^{nd} row of $\mathbf{A}_{\text{new}}$, and retain the 1^{st} row. Then to visualize the result, transform back to original variables. How do we express reduced data in original variables? Since $\mathbf{U}$ is orthogonal, $\mathbf{A} = \mathbf{U}\widetilde{\mathbf{A}}$. Now view $\mathbf{A}$ as a series of outer products (cf. lecture 13), and retain only the term which uses the first new variable. We then have $\mathbf{A}_{\text{reduced}} = \mathbf{u}_1\widetilde{\mathbf{x}}^T$, where $\mathbf{u}_1$ is the first column of $\mathbf{U}$. The reduced data, in our original variables, is the outer product of the 1^{st} column of $\mathbf{U}$ with the 1^{st} row of $\widetilde{\mathbf{A}}$:

```
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.')
```

Running this code produces the plot on the right of Figure 14.1. We can see that the data has been projected in the direction of maximum variance. Another way to look at this is to go back to our initial presentation of PCA where $\tilde{\mathbf{x}} = \alpha_1\mathbf{x} + \beta_1\mathbf{y}$. Now, $\mathbf{u}_1 = [\alpha_1, \beta_1]^T$. Our reduced data is, $\mathbf{x}_r = \alpha_1\tilde{\mathbf{x}}$, and $\mathbf{y}_r = \beta_1\tilde{\mathbf{x}}$, so $\mathbf{y}_r = \frac{\beta_1}{\alpha_1}\tilde{\mathbf{x}}_r$. This is the line displayed in the plot on the left of Figure 14.1.

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. Going back to our meteor problem, given a circular trajectory in Cartesian coordinates, will PCA give us $\theta(t)$? No, this requires a nonlinear transformation. Many variants of PCA have been developed since it's original introduction more than 100 years ago which extend the method in various ways, and some of these extensions do allow nonlinear transformations to be applied to the data.

14.2 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 for this problem, and we will look at just one based on low-rank matrix factorization. Is problem actually important? There are many applications where it appears including computer graphics, computer vision, machine learning, and recommender systems. We will focus on the last of these. How do Netflix, Amazon, Facebook, etc... decide what to recommend to you? They collect information about what you like (and dislike) as well as information about what everyone else likes. 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 (see Table 14.1).

How do we fill in the missing entries? Several different answers to this question have been developed over the years. One idea is to assume there is a set of high-level concepts

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

Table 14.1: A user-item matrix with missing entries.

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 be used to generate a rating based on the features that the user likes. Let's first think about a simpler case - how do we fill in the missing entries in this matrix: $\mathbf{D} = \begin{bmatrix} 1 & 2 \\ \times & 6 \\ 2 & \times \end{bmatrix}$. We need to set criteria to assess how well we fill in the data. The basic idea is to 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 of the matrix (reminder: Rank($\mathbf{A}$) = dimension of column space of $\mathbf{A}$ = dimension of row space of $\mathbf{A}$). The rank-1 estimate for $\mathbf{D}$ in our example is: $\begin{bmatrix} 1 & 2 \\ 3 & 6 \\ 2 & 4 \end{bmatrix}$. 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", and 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:
  - $\widetilde{\mathbf{R}} = \mathbf{AB}$ (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 $\widetilde{\mathbf{R}}$ will be at most p. The "rank" in our low rank approximation should correspond to the number of item "features". What are features? Figure 14.2 shows movies clustered based on computed "factor vectors" (rows in a feature-item matrix). If $p = 2$, then each column in $\mathbf{B}$ can be interpreted as an (x, y) point, and the figure shows that clusters of these points correspond to recognizable concepts, e.g. strong female lead, critically-acclaimed indie, violent, ...

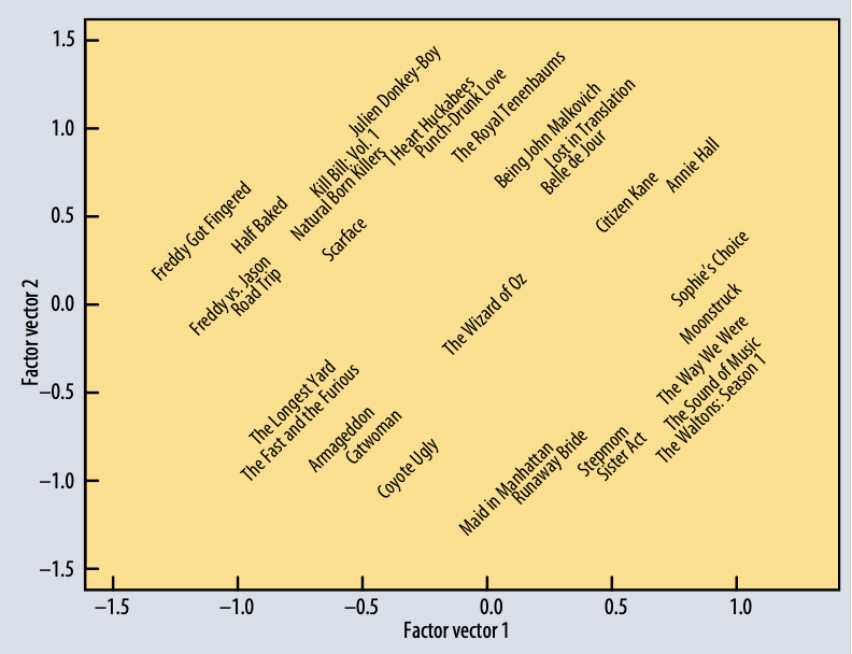


Figure 14.2: Feature vectors generated from Netflix ratings data (taken from [4]).

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" $\widetilde{\mathbf{R}} = \mathbf{AB}$. "Best" will defined with a modified Frobenius norm: $|\mathbf{R} - \widetilde{\mathbf{R}}|^2_{F*} = \sum_{(i,j)\in K} \left(R_{ij} - \tilde{R}_{ij}\right)^2$. Here, K is the 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, $c = |\mathbf{R} - \widetilde{\mathbf{R}}|^*_{\mathrm{F}}$ 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 $k = 1, 2, \ldots, m; l = 1, 2, \ldots p; q = 1, 2, \ldots, n$. 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: $c = \sum_{(i,j)\in K} \left(R_{ij} - \tilde{R}_{ij}\right)^2$, with $\tilde{R}_{ij} = \sum_{s=1}^{p} A_{is}B_{sj}$. Differentiating c with respect to A_{kl} gives:

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

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:

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

We can now adjust the sum so $i = k$ and only j will vary: $\frac{\partial c}{\partial A_{kl}} = -2\sum_{j,(k,j)\in K} \left(R_{kj} - \tilde{R}_{kj}\right) B_{lj}$. Then we use, $\tilde{R}_{kj} = A_{kl}B_{lj} + \sum_{s\neq l} A_{ks}B_{sj}$, and rearrange our equations:

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

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} :

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

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 $p = 3$ produces the following matrix:

	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. For example, 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

Discrete Fourier transform

15.1 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 e.g. $\left(\mathbf{A}^{\mathrm{T}}\mathbf{A}, \mathbf{A}\mathbf{A}^{\mathrm{T}}, \mathbf{A}+\mathbf{A}^{\mathrm{T}}, \ldots\right)$. 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 is a time series: $f\left(t_i\right), t_i = i\Delta t, i = 0, 1, \ldots, n_t - 1$. We could also have data in space (along a line): $f\left(x_i\right), x_i = i\Delta x, i = 0, 1, \ldots, n_x - 1$. How do we extract trends or, more generally, useful "information", from such data? What should we do with a signal that looks like Figure 15.1? The aim here 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? We can think about the frequency spectrum. The underlying idea is to decompose signal into a superposition of waves with different frequencies and amplitudes, and then check which frequencies hold the most "energy".

15.2 Fourier series

It will be useful to first review some properties of Fourier series. The Fourier series of a general function defined on the interval $[-\pi, \pi)$ is:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \exp(ikx) \tag{15.1a}$$

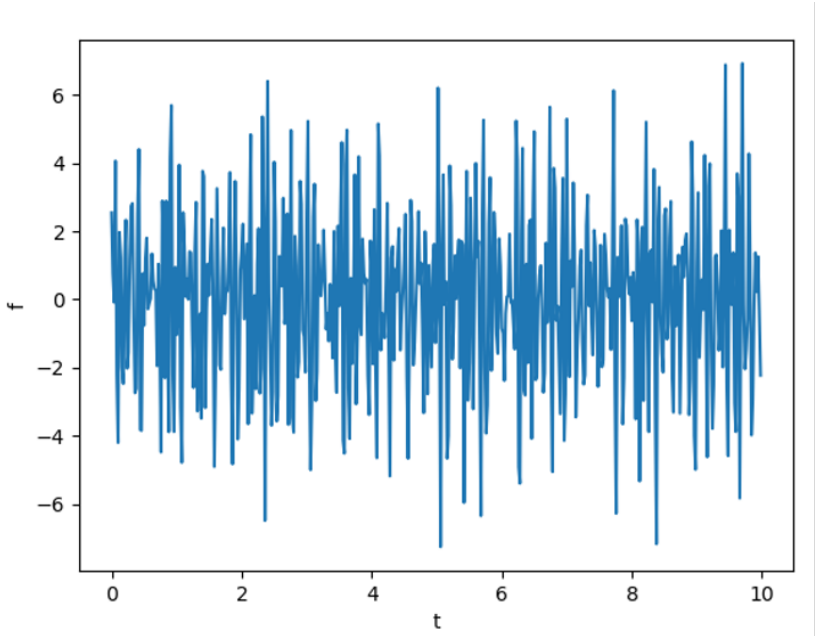


Figure 15.1: An example time series.

$$c_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\exp(-ikx)dx \tag{15.1b}$$

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 usually 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: $c_k \sim \exp(-\mu|k|), \mu$ is a positive constant (this is a very nice property!). We will largely ignore the formal convergence theory and just try to build some useful 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

$$f(x) = x,\ -\pi \leq x < 2\pi \tag{15.2}$$

This function is 2π-periodic, however if we consider its periodic extension to the entire real line, we see that it has a series of discontinuities (Figure 15.2). Due to this lack of smoothness, convergence is slow: $|c_k| \sim 1/|k|$. The plot on the right of Figure 15.2 shows an approximation to the function constructed by retaining the 1^{st} $2N + 1$ terms in the

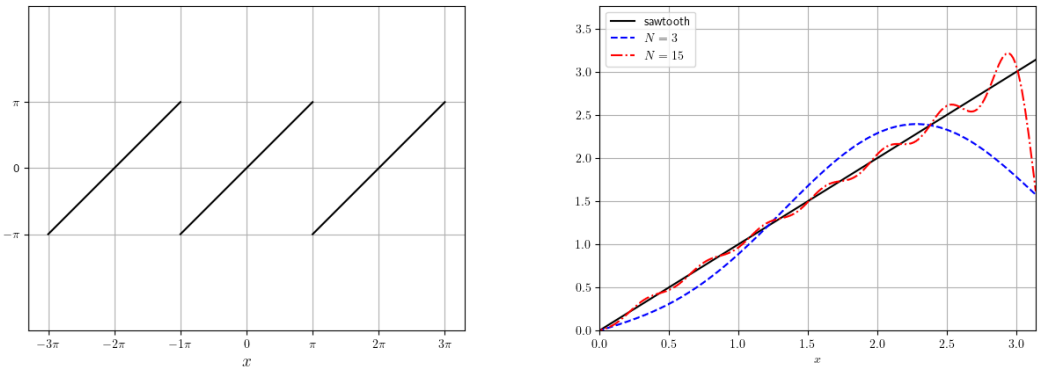


Figure 15.2: The sawtooth function (15.2) is shown on the left, and the partial reconstruction of the function using $2N+1$ Fourier coefficients is on the right.

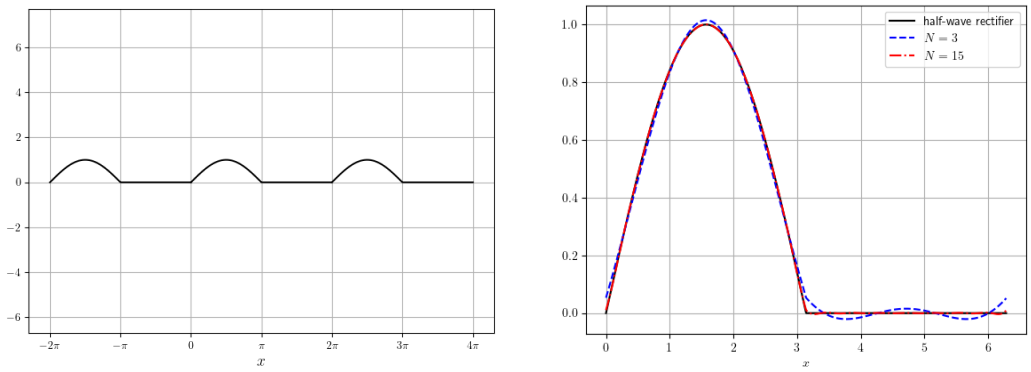


Figure 15.3: The half-wave rectifier (15.3) is on the left, and the partial reconstruction of the function using $2N+1$ Fourier coefficients is on the right.

series expansion:

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

We can see that the quality of the approximation improves with N, but there is always tangible error near $x = \pm\pi$ where the periodic extension of the sawtooth function is discontinuous.

Example 2: half-wave rectifier

$$f(x) = \begin{cases} \sin(x) & \text{if } 0 \le x < \pi \\ 0 & \text{if } \pi \le x < 2\pi \end{cases} \tag{15.3}$$

This function is also 2π-periodic, and it is continuous everywhere. However, the derivative

of its periodic extension has a series of discontinuities (Figure 15.3). Convergence is again slow, $|c_k| \sim \frac{1}{k^2}$, though it is considerably faster than what we saw for the sawtooth function. This improvement is can be seen in the $2N+1 = 31$ term reconstruction shown in Figure 15.3 where it is difficult to visually distinguish between the approximation and the actual function.

These two examples are pointing us towards a more general result.

If:

1. $f(\pi) = f(-\pi), f^{(1)}(\pi) = f^{(1)}(-\pi), \ldots, f^{(m-2)}(\pi) = f^{(m-2)}(-\pi)$,
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.

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 on the interval of interest are closely related to how well it is represented as a Fourier series. The proof is based on repeated application of integration-by-parts to (15.1b).

15.3 Discrete Fourier transform (DFT)

We still have to figure out how these ideas can be applied to discrete data. We'll now work with data on a n-point, equispaced grid:

$$y(t_j), \; t_j = j\Delta t, \; j = 0, 1, \ldots, n-1.$$

We expand the data as a weighted sum of complex exponentials which are periodic over a timespan of length $\tau = n\Delta t$:

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

The second form of the summand is obtained by noticing that $t_j/\tau = j/n$. This form is more convenient to work with as it doesn't require any knowledge of what the timespan or time step are. This expansion is the *inverse discrete Fourier transform* of our data. Given all c_k we can recover the original data. While there is a clear correspondence to (15.1a), there are important differences as well (you should confirm the correspondence). We now have a finite sum with n terms (matching the number of data points, and the arguments of the exponentials are more complicated as we are not restricting the domain width to be

2π. We are decomposing the original data 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, y_j, is sampled from a continuous function, $g(t)$. Then the ideas underlying convergence of Fourier series apply here provided Δt is sufficiently small. In other words, 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:

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

This equation should be compared with (15.1b). Now, an integral has been replaced with a finite sum.

15.3.1 Computing DFTs

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 $\mathcal{O}\left(n^2\right)$. However, discrete Fourier transforms can be computed using a divide-and-conquer approach! The fast-Fourier-transform (FFT) algorithm computes all n coefficients in $\mathcal{O}\left(n\log_2 n\right)$ 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, and this package can be viewed as a subset of the larger scipy package. The function `np.fft.fft` computes the discrete Fourier transform of an input array, but it doesn't give us exactly what we usually want. It instead outputs a numpy array containing:

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

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

$$n * \left[c_{-n/2}, c_{-n/2+1}, \ldots, c_{n/2-1}\right].$$

The function `np.fft.ifft` will output the original data given $n*\left[c_0, c_1, \ldots, c_{n/2-1}, c_{-n/2}, c_{-n/2+1}, \ldots, c_{-1}\right]$ as input (and the cost is $\mathcal{O}\left(n\log_2 n\right)$).

Let's see how these tools work by computing the FFT of the example signal shown in Figure 15.1:

```
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')
```

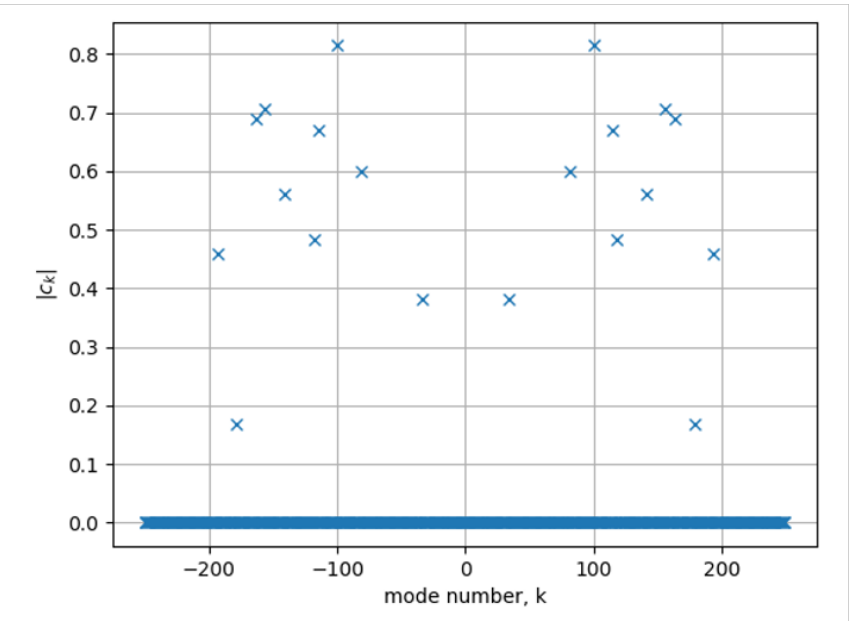


Figure 15.4: Magnitudes of Fourier coefficients obtained from DFT of example signal.

```
plt.ylabel('$|c_k|$')
plt.grid()
```

In the code above, `y` is an array containing the signal data. Applying the DFT to this array gives the Fourier coefficients stored in `c`, and the magnitudes of these coefficients are shown in Figure 15.4. All but twenty of the coefficients are effectively zero. This is because the signal is a superposition of 10 sine waves $w/$ different frequencies and with randomly chosen amplitudes and phases:

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

Since $\sin kx = \frac{e^{ikx} - e^{-ikx}}{2i}$, a single sine wave will have two non-zero coefficients corresponding to $\pm k$. The m^{th} sine wave has frequency $f_m = \frac{r_m}{\tau}$ with $\tau = 10$, and r_m a randomly chosen integer, $1 <= r_m < n/2$. This ensures that the periodic extension of each wave will be smooth and that the data can be represented efficiently with our basis functions. The square of the magnitude of a coefficient, $|c_k|^2$, is the "energy" in a mode with frequency, k/τ. This interpretation follows from a discrete version of Parseval's theorem: $\frac{1}{n}\sum_{j=1}^{n} |y_j|^2 = \sum_{k=1}^{n} |c_k|^2$. Note that for real-valued data, $c_k = -c_{-k}^*$, so when examining the energy, only $k \geq 0$ needs to be considered (and it is more efficient to use `np.ffr.rfft` rather than `np.fft.fft`). Also note that $(n/2 - 1)/\tau = 1/2\Delta t - 1/\tau$ is the highest frequency that can be "resolved" using the n basis functions available. The rule-of-thumb that follows is: > 2 data points per period of the highest-frequency component in the data

are needed for the FFT to accurately "identify" that component. So in our example, we require $2\Delta t < 1/\max(f_m)$.

Let's look at another example. Our data will be 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 $1/f_A = \tau/2$ and $1/f_B = 0.4\tau$ are the periods of the waves. We will set the timespan of the data so that $\tau = 5$, and a figure showing the waves and code for generating them is below:

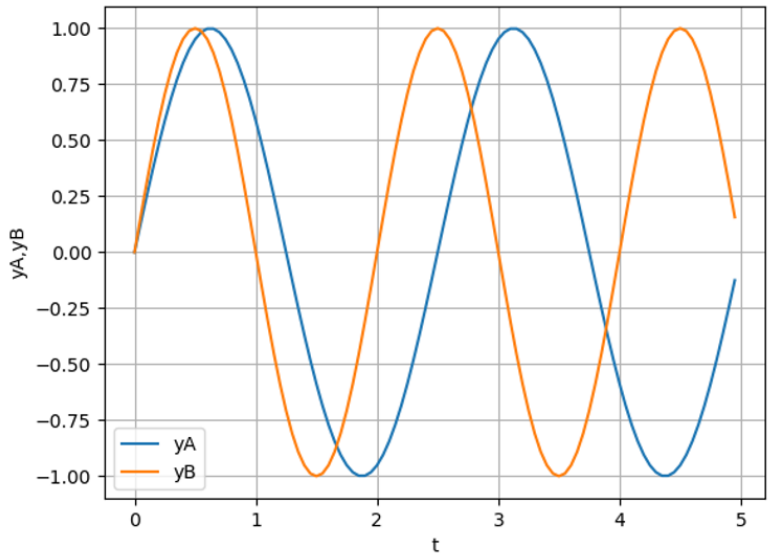


Figure 15.5: Sine waves for DFT example.

```
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)
```

We then compute the DFT of each array:

```
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 in Figure 15.6. We can see that wave A is represented perfectly with modes $k = \pm 2$ (only

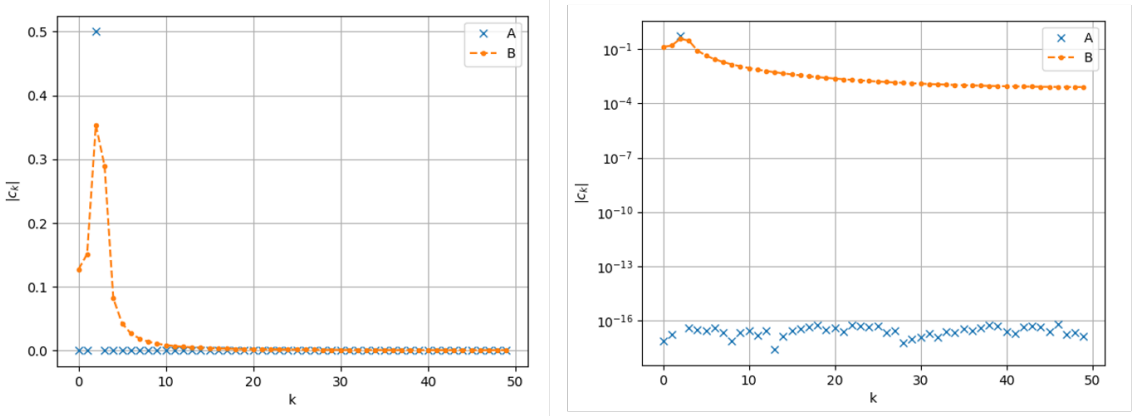


Figure 15.6: Magnitudes of Fourier coefficients ($|c_k|$) for two sine waves. The two plots display the samd data with different vertical axis scales.

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, τ is not an integer multiple of the function's period. On the other hand, The basis functions in the DFT expansion all have periods of the form τ/k (with k an integer) and cannot easily represent wave B. Another way to look at this is to observe that the periodic extension of wave B is not smooth (the derivative is discontinuous at $t = 0$ and $t = 5$). The coefficients decay slowly here for essentially the same reason they decayed slowly for the half-wave rectifier.

There is no reason to generally expect signals to be perfectly periodic for a given τ, 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 application of windowing to wave B is shown in Figure 15.7. The spectrum of the windowed signal has a narrow peak and

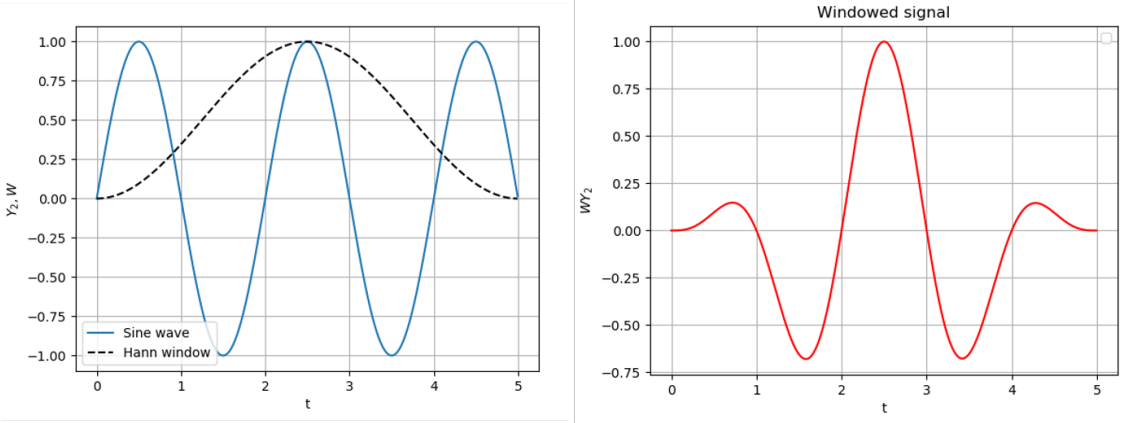


Figure 15.7: The original data and Hann window function are shown on the left. The windowed data is shown on the right.

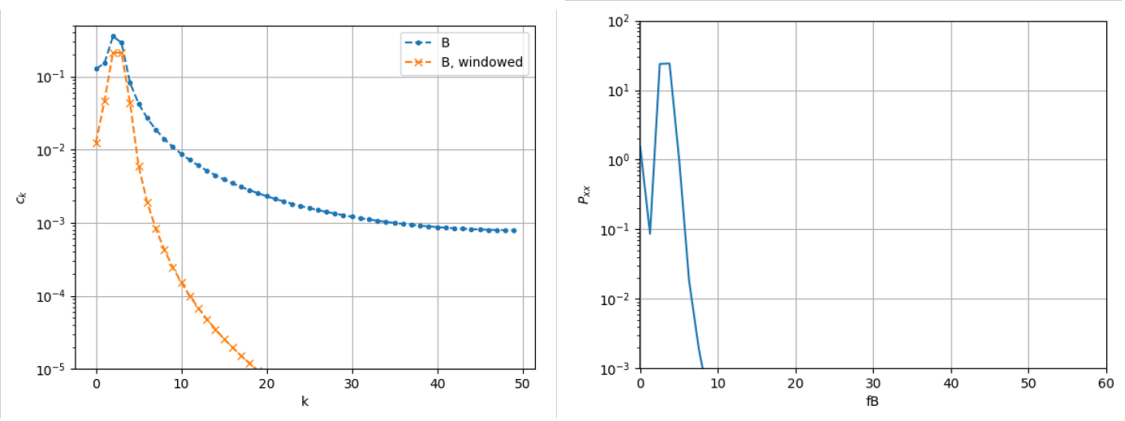


Figure 15.8: Spectrum obtained with the windowed DFT of sine wave B is shown on the left while the power-spectral density computed with Welch's method is shown on the right.

"looks" more like a simple wave (Figure 15.8). However, we can clearly see that there is a loss of energy (how would you quantify this?). There is no perfect solution, however more advanced methods have been developed that reduce this energy loss while preserving the "wave-like" nature of the spectrum.

15.4 Welch's method

Welch's method handles the windowing and DFT computation for us. It 1) breaks the signal up into overlapping segments, 2) windows the signal within each segment, 3) applies the DFT to each segment, and 4) averages the spectra from each segment (typically $|c_k|^2$ is averaged 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 $P_{xx}(f_k) = |c_k|^2 \frac{\Delta t}{n}$. A function implementing Welch's method is available via the `scipy.signal` package. The code below uses this function to compute the power spectral density (PSD) of wave B from our example:

```
In [201]: fB,PxxB = sig.welch(yB,1/dt)
In [202]: plt.semilogy(fB,PxxB)
```

The resulting plot is shown in Figure 15.8. This figure isn't particularly impressive as the example is quite contrived however this is a very widely-used method that has been applied to a broad range of applications.

The fundamental idea in this lecture is to view a signal of n points with step Δt as a distribution of "energy" across a range of frequencies. When analyzing data from this perspective, we need to ensure that the time step in the data is small enough $\Delta t < 1/(2f_{\max})$ to resolve the highest frequency components in the signal. We also need to ensure that the time span is long enough $\tau \ggg 1/f_{\min}$ for important "slow" components are contained within the data. This was not the case in our previous example where the

period of the waves was comparable to τ. Often, slow components contain the most energy when analyzing complex nonlinear systems.

Lecture 16

Numerical differentiation for multiscale problems

Climate modeling, aerodynamics, and material simulation are all highly nonlinear, multiscale, and very complex problems. In each of these examples nonlinearity will generate a broad range of length and time scales. A hurricane can persist for several days while an individual wind gust may only last for less than a minute. Which numerical methods are best for studying such problems? We will focus on the problem of numerical differentiation and consider the analysis and design of finite difference methods for multiscale problems.

Figure 16.1: Image of hurricane Florence from NASA Goddard Space Flight Center from Greenbelt, MD, USA, CC BY 2.0.

Generally, nonlinearity can lead to the generation of a broad range of scales (frequencies and wavelengths). This effect is often balanced by diffusion which tends to dissipate high-

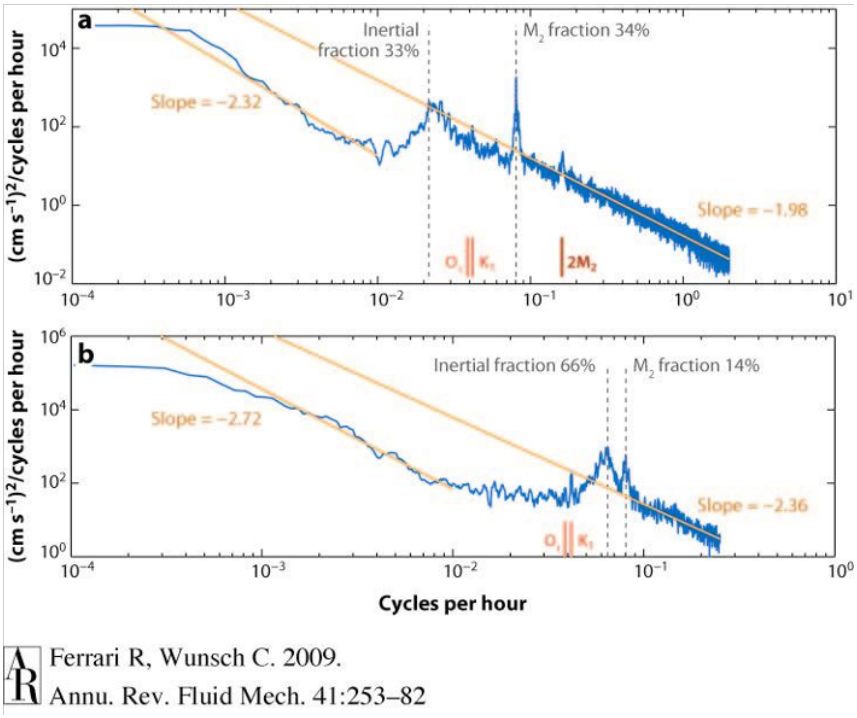


Figure 16.2: Power spectral densities of latitudinal ocean current speeds in the north Atlantic (top) and Southern (bottom) oceans. Taken from [3],

frequency fluctuations. Consider a model for a dynamical system which includes a $u\frac{\partial u}{\partial x}$ term (u could be a wind velocity component), and say at some time, we have $u = \sin(kx)$. Then, the nonlinear term will evaluate to be $u\frac{\partial u}{\partial x} = k\sin(kx)\cos(kx) = \frac{1}{2}k\sin(2kx)$. Nonlinearity generates a function varying twice as fast as our original function. The cumulative effect of nonlinearity can then produce broadband power spectral densities. Figure 16.2 shows two examples based on measurements taken in the Atlantic and Southern oceans. The horizontal axis corresponds to frequency, and for each spectrum, we see two clear peaks marked with vertical dashed lines. The peak at the lower frequency is associated with the Coriolis force caused by the Earth's rotation. The second peak corresponds to diurnal tides. The Coriolis force frequency depends on latitude which is why the peak in the Southern ocean occurs at a higher frequency. Numerical methods for simulations must accurately capture a range of scales. For example, they shouldn't "miss" secondary peaks in spectra.

16.1 Finite difference methods

Say we have data on a n-point, equispaced grid: $f(x_j)$, $x_j = jh$, $j = 0, 1, 2, \ldots, n-1$, and h is the *grid spacing*. Assuming that the data is sampled from a smooth underlying function, we can construct finite difference (FD) methods for estimating f'_j, the derivative

at point j, using the following Taylor series expansions:

$$\begin{aligned} f_{j+1} &= f_j + hf'_j + \frac{h^2}{2}f''_j + \frac{h^3}{6}f'''_j + \cdots \\ f_{j-1} &= f_j - hf'_j + \frac{h^2}{2}f''_j - \frac{h^3}{6}f'''_j + \cdots \end{aligned}.$$

The first equation can be rearranged as, $f'_j = \frac{f_{j+1}-f_j}{h} + \mathcal{O}(h)$ which gives the 1st-order accurate forward FD method, $f'_j \approx \frac{f_{j+1}-f_j}{h}$. We say the method is 1st-order because the truncation error increases as h raised to the first power. The second expansion can be used to derive a 1st-order backward method, $f'_j \approx \frac{f_j-f_{j-1}}{h}$. We can derive a 2nd-order method by combining the expansions. Subtracting the second expansion from the first and rearranging gives,

$$f'_j = \frac{f_{j+1} - f_{j-1}}{2h} + \frac{h^2}{6}f'''_j,$$

and the second-order centered method estimates the derivative as,

$$f'_j \approx \frac{f_{j+1} - f_{j-1}}{2h}.$$

The error is $\mathcal{O}\left(h^2\right)$, but we will see there is more information about the approximation that is "hiding" in this error term. This method is more accurate than the first-order methods and can be computed with the same number of operations (assuming $2h$ is pre-computed and stored). It is also straightforward to implement in Python. Let's use it 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 test function:

```
#generate grid
L = 5
N = 100
x = np.linspace(0,L,N)
h = x[1]-x[0]
hfac = 1/(2*h)

#generate test 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 in Figure 16.3 with $n = 100$. Visually, the agreement is good, but we should quantify the error. Defining a local error as, $\epsilon(x) = \left| \frac{df}{dx}|_{\text{computed}} - \frac{df}{dx}|_{\text{exact}} \right|$, we can see in the figure that the error is reasonably small which suggests that the method works well for this particular value of n. This is just a first step in testing the method and code. 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 $\epsilon(x)$.

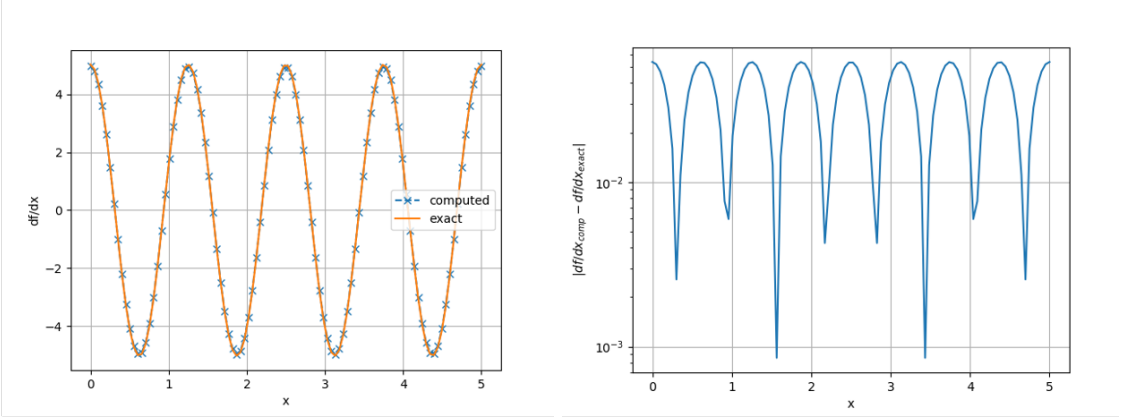


Figure 16.3: The plot on the left compares the exact derivative of the test function with the result computed using 2nd-order centered finite differences. The plot on the right shows the magnitude of the difference between the computed and exact results.

Check your understanding: How will the average of ϵ change if n is doubled to 200?

16.2 Wavenumber analysis

When simulating multiscale problems, a FD method should work well for a broad range of wavelengths. Consider data corresponding to a complex sinusoidal 'wave' with wavenumber, k : $f(x) = \mathrm{e}^{ikx}$ which has derivative, $df/dx = ik\mathrm{e}^{ikx}$. Let's look at the 2^{nd} -order centered finite difference (FD) approximation for df/dx:

$$
\begin{aligned}
&f_{j+1} = e^{ik(x_j+h)} \text{ and } f_{j-1} = e^{ik(x_j-h)}, \text{ so:} \\
&\frac{f_{j+1} - f_{j-1}}{2h} = \frac{e^{ik(x_j+h)} - e^{ik(x_j-h)}}{2h} = \frac{e^{ikx_j}}{2h}\left(e^{ikh} - e^{-ikh}\right) \text{ or,} \\
&\frac{f_{j+1} - f_{j-1}}{2h} = i\left[\frac{\sin(kh)}{h}\right] e^{ikx_j}.
\end{aligned}
$$

Comparing this last equation to $ik\mathrm{e}^{ikx}$, we see that the key question is, how close is $\frac{\sin(kh)}{h}$ to k? It is actually more convenient to ask, how close is $\sin(kh)$ to kh, and Figure 16.4

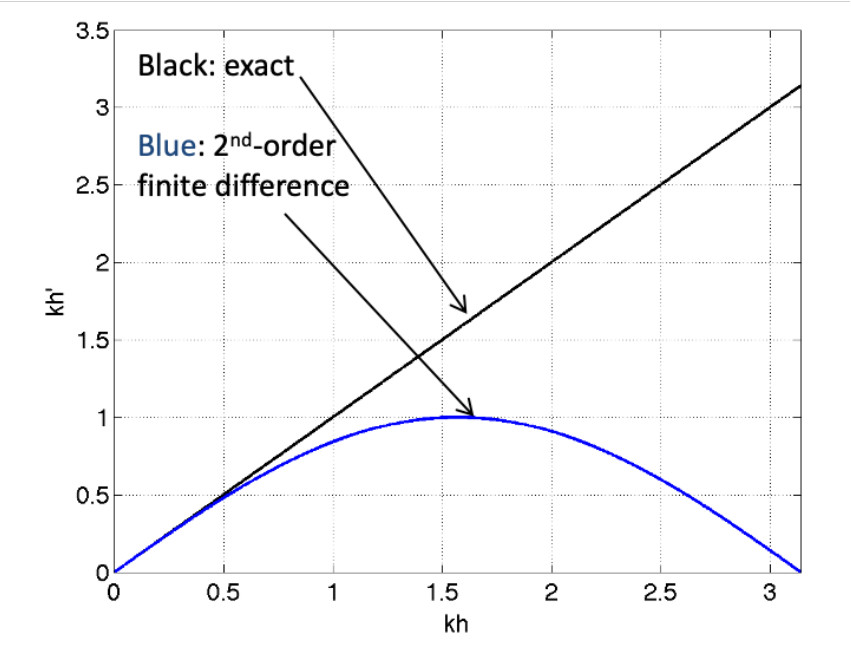


Figure 16.4: Modified wavenumber for 2nd-order centered FD method.

shows $\sin(kh)$ vs. kh. Here, $\sin(kh)$ is the *modified wavenumber* for this FD method, and generally, we will use kh' to indicate modified wavenumbers. 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 $\frac{\lambda}{h} \approx 26$. This means that we need $\approx$ 27 points/wavelength for $\approx$ 1% error. Note that differentiation using DFTs only requires $>$ 2 points/wavelength for the exact solution (this method is discussed in lab 8).

Check your understanding: The 2^{nd} -order 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?

16.3 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 that the explicit Euler method has poor stability properties. The implicit Euler method has much better stability, but requires matrix inversion for systems of ODEs which is more expensive. We can also construct "implicit" finite difference schemes for numerical differentiation. Our 2^{nd}-order explicit finite difference formula is, $f'_j = \frac{f_{j+1}-f_{j-1}}{2h}$. For implicit schemes, we will

work with the following 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, $\mathbf{A}\mathbf{f}' = \mathbf{b}$ 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 oscillations. Let's first think about the truncation error. Using Taylor series and after a lot of arithmetic, we get the following conditions:

2^{nd} order: $a + b + c = 1 + 2\alpha + 2\beta$
4^{th} order: $a + 2^2 b + 3^2 c = 2\frac{3!}{2!}\left(\alpha + 2^2\beta\right)$
6^{th} order: $a + 2^4 b + 3^4 c = 2\frac{5!}{4!}\left(\alpha + 2^4\beta\right)$

The highest possible accuracy is 10^{th} 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 10^{th} -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? Examining Figure 16.5, the 10^{th}-order method is certainly much more accurate than the 2^{nd}-order FD scheme. For the 10^{th} order method, we only need ≈ 4 grid points per wavelength for 1% error. Can we do better? We will apply the following (somewhat arbitrary) constraints:

$kh'(kh = 2.2) = 2.2$
$kh'(kh = 2.3) = 2.3$
$kh'(kh = 2.4) = 2.4$

We are requiring differentiation of the complex exponential function to be exact for these three values of kh. Applying these constraints, and requiring fourth-order accuracy gives the following coefficients:

$$\alpha = 0.5771439, \beta = 0.0896406, a = 1.3025166, b = 0.99335500, c = 0.03750245$$

The modified wavenumber for this scheme is also shown in Figure 16.5. This 4^{th} order scheme requires ≈ 3.4 points/wavelength for 1% error, and it is clear that it performs better for high wavenumbers than the 10th-order scheme. Order of accuracy is important, but it isn't everything!

We also need to consider the cost when comparing methods. What are the operation counts for these schemes? The 2nd-order centered scheme requires N multiplies and N

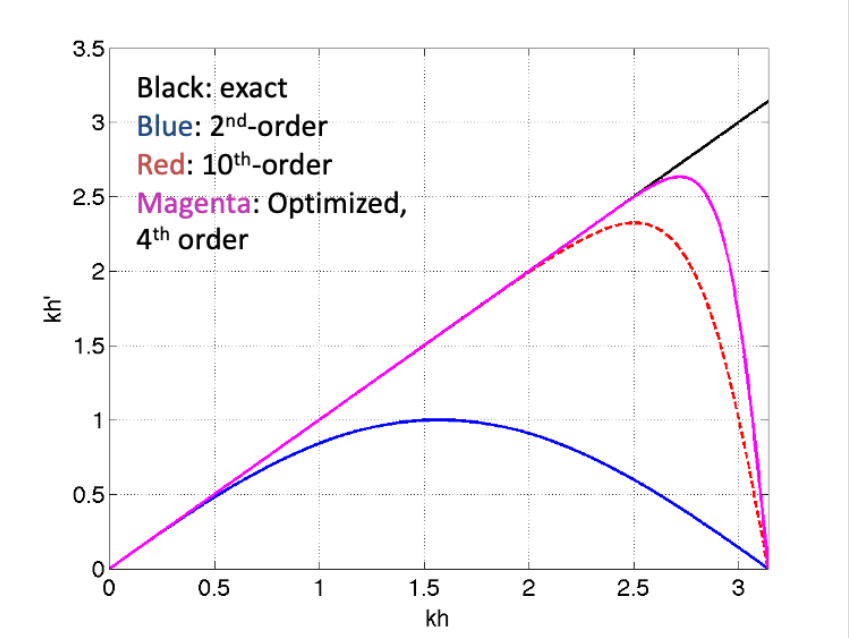


Figure 16.5: Modified wavenumbers for different FD schemes.

additions. For the implicit scheme, the cost (of an efficient solver) is $\mathcal{O}(N)$. For example, the 4th order scheme described above requires the solution of a pentadiagonal linear system, and this requires $7N$ multiplies and $7N$ additions. Spectral differentiation using DFTs has $\mathcal{O}(N \log_2 N)$ cost.
The key question is: which method requires least time for desired accuracy? It is difficult to provide a general answer as the details of a given problem and the desired accuracy are both important and can vary substantially.

16.3.1 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:

$$\begin{bmatrix} f_1 & g_1 & h_1 & & & & & \\ e_2 & f_2 & g_2 & h_2 & & & & \\ d_3 & e_3 & f_3 & g_3 & h_3 & & & \\ & & \cdot & \cdot & \cdot & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & \cdot & \cdot & \cdot & \\ & & & & d_{n-1} & e_{n-1} & f_{n-1} & g_{n-1} \\ & & & & & d_n & e_n & f_n \end{bmatrix}$$

How do we solve this in Python? In general, for problems like this, we can use, *np*.linalg.solve(A, b). However, this isn't an efficient approach as it requires excessive memory (storing

the zeros in $\mathbf{A}$), and it also carries out unnecessary operations (again due to the zeros). We can instead build a sparse banded matrix using scipy.sparse.diags. The first step is to construct the 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, we can construct the sparse matrix, $\mathbf{A}$:

```
A = sp.diags([bgv,agv,zv,agv,bgv],[-2,-1,0,1,2])
print(A.toarray())
```

```
Out[72]:
array([[1.       , 0.5771439, 0.0896406, 0.       , 0.       , 0.       ],
       [0.5771439, 1.       , 0.5771439, 0.0896406, 0.       , 0.       ],
       [0.0896406, 0.5771439, 1.       , 0.5771439, 0.0896406, 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.       ]])
```

Finally, scipy.sparse.linalg.spsolve can be used to solve the sparse system (note that the displayed matrix is not particularly small since N is small).

This approach solves the memory issue, but it is still possible to do better with efficiency. The `scipy.linalg` module has a solver specifically designed for banded matrices: `scipy.linalg.solve_banded` (this, like many scipy functions, is actually a Fortran routine from the Lapack library). This function is a little tricky to use as the matrix contents have to be provided in "matrix diagonal ordered form":

`A_b[u+i-j, j]==A[i, j]`.

Here, `A_b` is a 2-D array required as input to `solve_banded`, and `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. However, there is one further issue that needs to be resolved. 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.

16.3.2 Boundary modifications

At $i = 1$ and $i = n - 2$, we can switch to a 8th-order tridiagonal scheme with:

$$\alpha = \frac{3}{8}, \beta = 0, a = \frac{1}{6}(\alpha + 9), b = \frac{1}{15}(32\alpha - 9), c = \frac{1}{10}(-3\alpha + 1).$$

At $i = 0, n - 1$, we switch to 4th-order "one-sided" schemes:

$$f'_0 + \alpha f'_1 = \frac{1}{h}\left(af_0 + bf_1 + cf_2 + df_3\right)$$
$$f'_{n-1} + \alpha f'_{n-2} = -\frac{1}{h}\left(af_{n-1} + bf_{n-2} + cf_{n-3} + df_{n-4}\right)$$

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.

16.4 Final comments

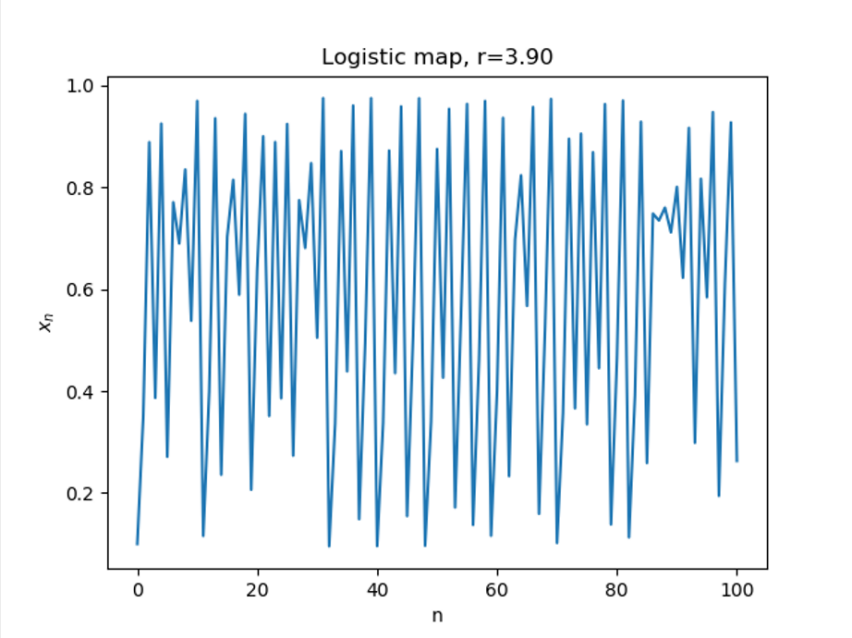
The advantages of optimized implicit FD schemes include 'competitive' efficiency and low memory usage (relative to explicit FD and Fourier). Key questions that follow include how much memory is available, and is there a performance gain from using less memory? Disadvantages include errors in one location potentially "contaminating" results in all locations, and the difficulty in applying the method 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 with the other being quadrature and interpolation. We won't consider the latter two here, but scipy does provide helpful tools in `scipy.integrate` and `scipy.interpolate` for these problems.

Lecture 17

Nonlinear data analysis

We have looked at a few linear methods including DFTs which are based upon a linear superposition of complex exponentials and PCA which applies a linear transformation. However, complex systems are typically nonlinear, and we should think about customized methods designed specifically for nonlinear systems. We previously mentioned nonlinear PCA which is one example, and here, 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 high (Δx or Δ t sufficiently small). However consider the example below. What can we do with this data?



The discrete Fourier transform won't be helpful as the data is not smooth. Instead we could compare extrema and consider the following basic options and compare: 1) peaks

to peaks, 2) troughs to troughs, or peaks to troughs. Let's try the 3rd option. Let x_i be the ith data point. Figure 17.1 shows x_{i+1} vs. x_i, and there is clear "structure" within the data! These figures are showing solutions to the *logistic map* which is defined by,

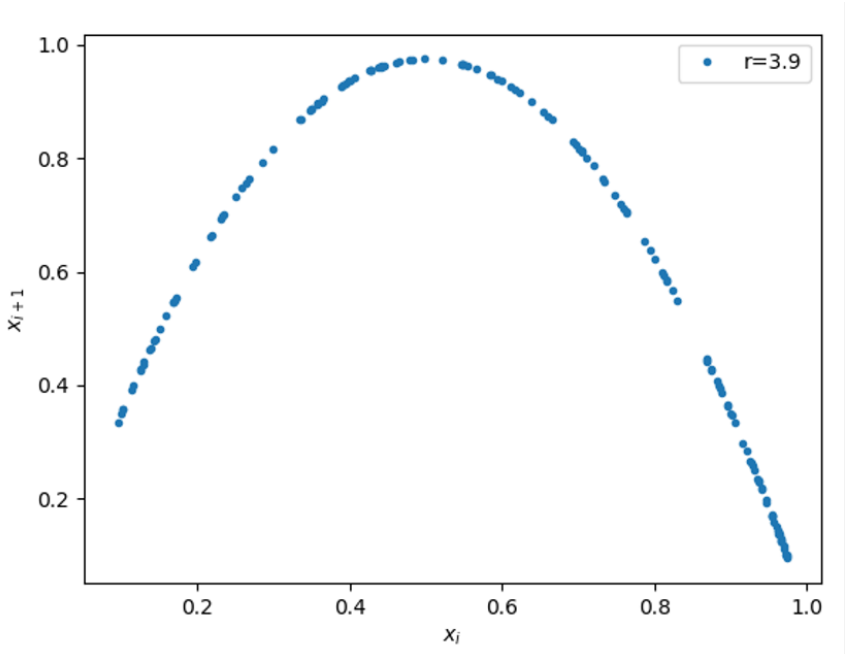


Figure 17.1: Comparing peaks and troughs of logistic map solution wtih $r = 3.9$.

$$x_{i+1} = r x_i \left(1 - x_i\right).$$

The parameter, r, controls the dynamics. For smaller r, the map produces simple periodic behavior, while for $r = 3.9$, the case shown here, the dynamics are chaotic. As time increases, more and more points on the parabola will be visited in an irregular order, and the curve in Figure 17.1 will "fill in". If we reduce r to $r = 3.5$, the dynamics are much simpler and we see period-2 oscillations where $x_{i+4} = x_i$ (Figure 17.2).

When characterizing a solution, it is helpful to think about the set of points visited (after discarding the initial transient). We will focus specifically in on the dimension of this set. The set of points generated when $r = 3.9$ is "almost" a one-dimensional curve while the four points visited when $r = 3.5$ (for $t \gtrsim 10$) could be described as zero-dimensional. How can we make these ideas more precise?

17.1 Fractal dimension

We will borrow ideas from the study of fractals and apply the idea of a *fractal dimension*. Figure 17.3 shows the first few iterations for constructing the von Koch fractal. Each iteration, the curve is broken up into 4 smaller copies of itself. The length of each copy

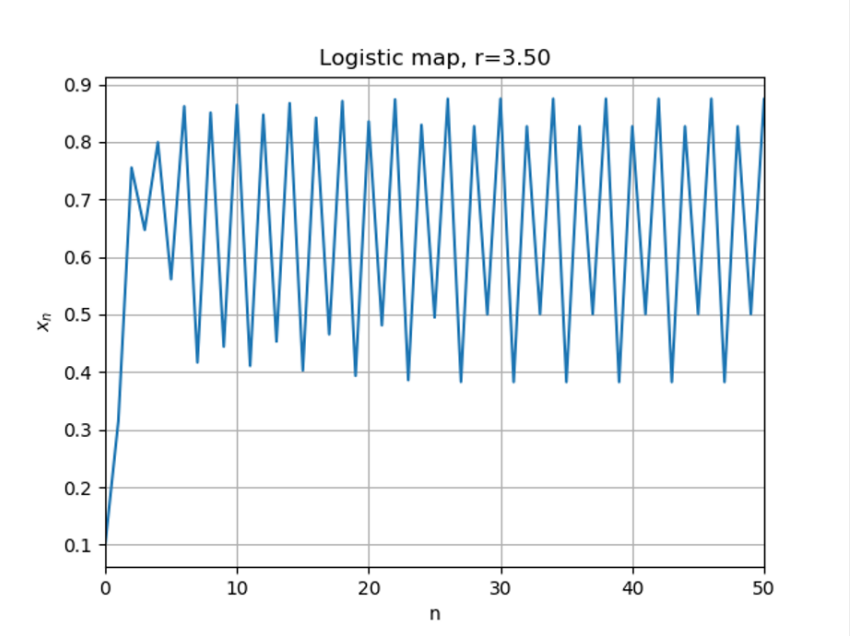


Figure 17.2: x_{i+1} vs. x_i for logistic map with $r = 3.5$

is 1/3 of the original, and the middle two copies are rotated so that the overall width of the object does not change. What is the dimension of this curve? Let's consider a

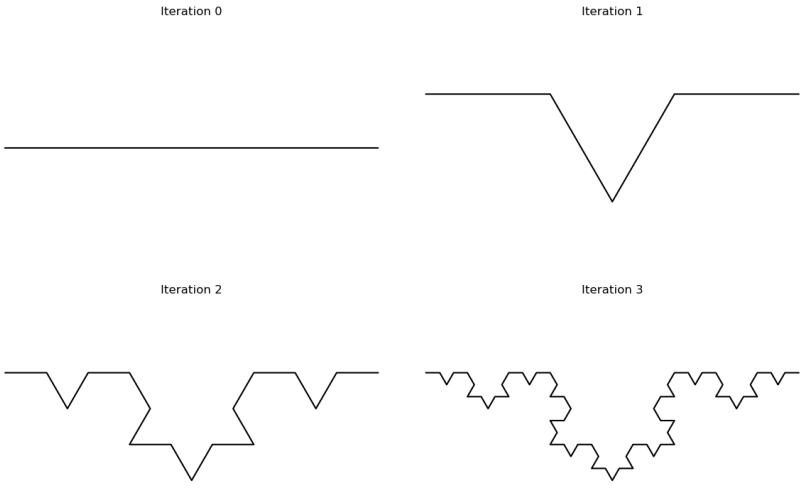


Figure 17.3: First 4 iterations of Koch curve.

simpler example where we know that the dimension should be: a square. We will say that

m is the number of copies that are made during an iteration, and r is the *scale factor*, the factor by which each copy's size is adjusted. Break a square into 4 smaller copies

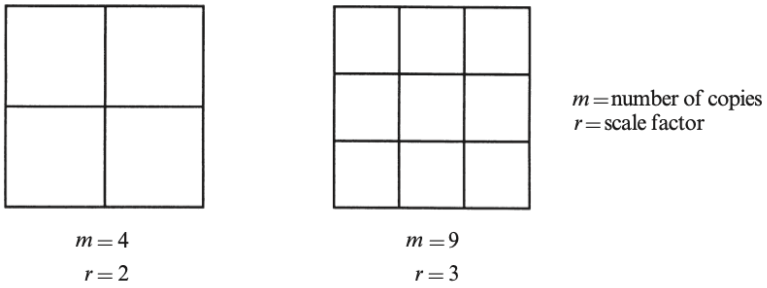


Figure 17.4

($m = 4$). Then, the sides of each copy must be scaled down by a factor of $2(r = 2)$ to ensure that the total area required does not change (Figure 17.4). With $m = 9$, we have $r = 3$, and we see that we can recover the conventional dimension of a square using, $d = \log(m)/\log(r) = 2$ (since $m = r^d$). This equation defines the fractal dimension. For the Koch curve, $m = 4, r = 3, d = \log(4)/\log(3) = 1.261\ldots$, and this result tells us that the fractal is more complex than a simple one-dimensional curve.

We now need to figure out how to compute the fractal dimension given data like the solutions we saw for the logistic map. One approach is to compute the *box dimension*. The box dimension is estimated by finding the number of m-dimensional cubes with edge-

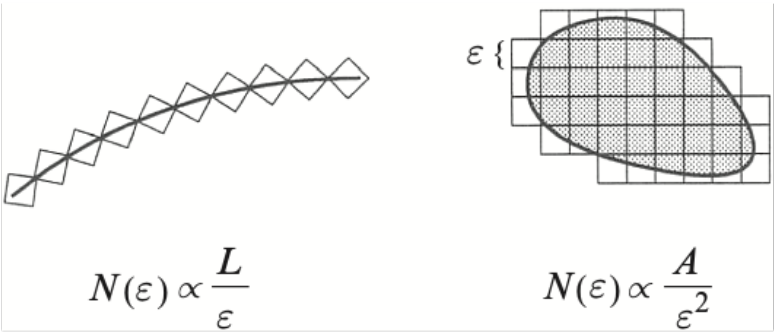


Figure 17.5: Illustrations of box dimension calculations for $d = 1$ and $d = 2$. For

length, ε needed to cover the set of points in the solution, $N(\varepsilon)$. For sets with fractal structure, we expect: $N(\epsilon) \propto 1/\epsilon^d$ where d is the box dimension.

17.1.1 Correlation 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. To compute the

correlation dimension, we first collect n m-dimensional points (vectors) that were "visited" during a process after discarding points at early times which are likely to be strongly influenced by the choice of initial condition: $\{\mathbf{x}_i, i = 1, 2, \ldots, n\}, \mathbf{x}_i \in \mathbb{R}^m$. Then, compute the correlation sum, $C(\varepsilon)$, which is:

(total number of pairs of vectors within distance ε of each other)/(Total number of distinct pairs).

The equivalent formula is:

$$C(\epsilon) = \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} H\left(\epsilon - \|\mathbf{x}_i - \mathbf{x}_j\|\right)$$

Here, H, is the Heaviside function, $H(z) = 0$ if $z <= 0, H(z) = 1$ if $z > 0$, and $\|\mathbf{x}_i - \mathbf{x}_j\|$ is the Euclidian distance between the m-dimensional vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. For points on a fractal-like structure, we expect $C(\epsilon) \sim \epsilon^d$ where d is now the correlation dimension. Generically, we expect that at small ε, there will be few if any pairs of vectors within ε of each other. At sufficiently large ε, almost all pairs of vectors will be within ε of each other. It is the region in between that we care about, and often some trial-and-error is needed to choose a good range of ε.

It is straightforward to construct a plot of $C(\epsilon)$ vs. ϵ for the logistic map using `scipy.spatial.distance.pdist` to compute the distances (look up the documentation!). The required steps and corresponding Python code are shown below:

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 solution into two arrays ($m = 2$) and collect vectors 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 $\frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2}$ 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)
```

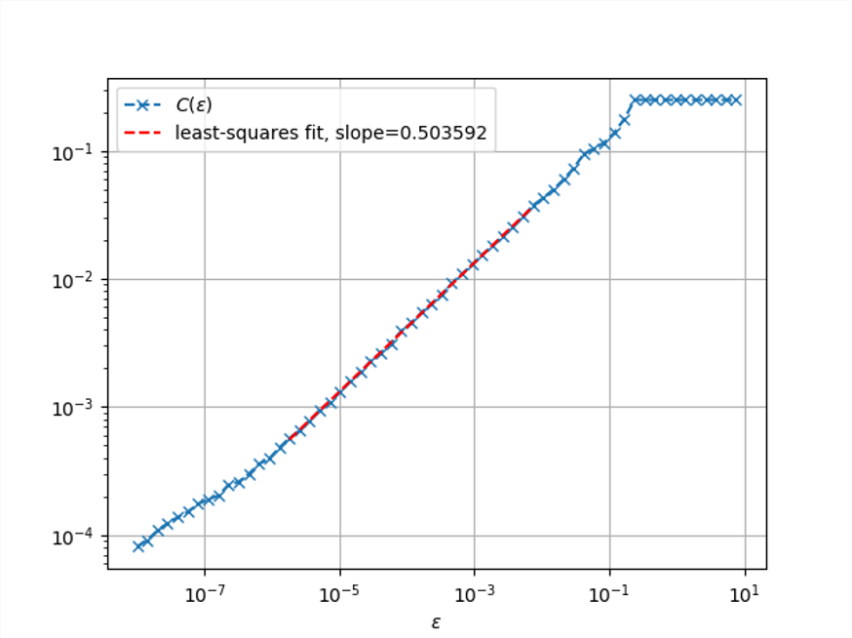


Figure 17.6: Correlation sum for logistic map, $r = 3.5699456$

Results for $r = 3.5699456$ are shown with $n = 8000$ in Figure 17.6. The 1^{st} 15 and last 20 points have been discarded for the best fit calculation, and the fractal dimension is estimated as 0.5 (for this value of r). (The estimate was computed using `np.polyfit`).

17.2 Lorenz system

Can the ideas discussed above be applied to systems of differential equations? Consider the following (famous) example:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x)\\ \frac{dy}{dt} &= rx - y - xz\\ \frac{dz}{dt} &= xy - bz\end{aligned}$$

Here, σ, b, and r are model parameters that must be specified, and in certain parameter ranges, chaotic dynamics are generated. This model, the Lorenz system, was initially developed in the 1960s as a model for atmospheric flows, and the modern field of chaos 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 to really understand what is happening (Figure 17.7). 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.

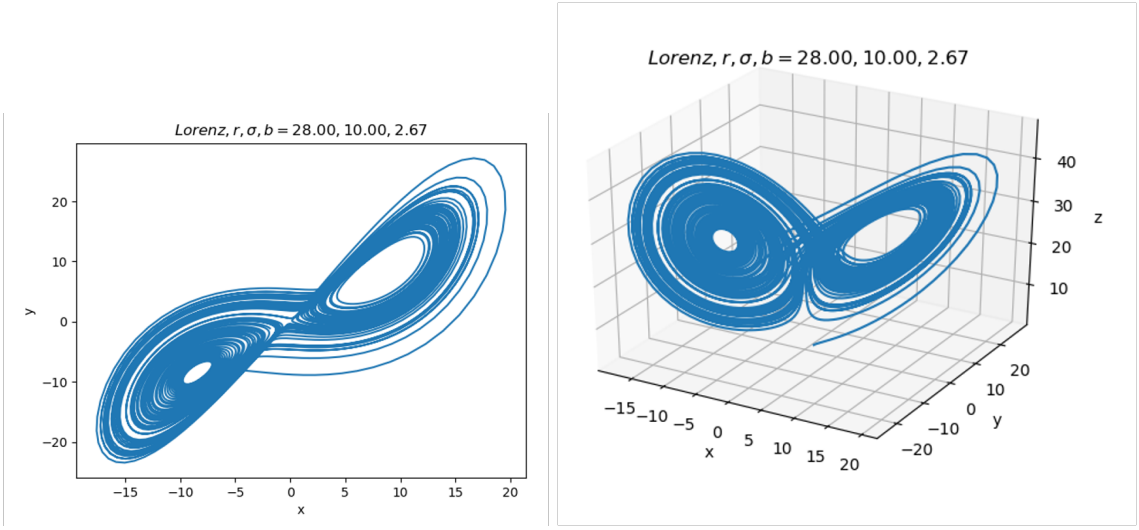


Figure 17.7: Phase plane (left) and 3D phase plot (right) for Lorenz system.

We can compute a correlation sum with n three-dimensional vectors:

$$\mathbf{v}_i = \left[x\left(t_i\right), y\left(t_i\right), z\left(t_i\right)\right]^T, (m = 3, i = 1, 2, \ldots, n)$$

Each point in the plot corresponds to one such vector. Computing the correlation sum after discarding points for $t < 10\ldots$, the dimension is estimated to be approximately 2.05 (Figure 17.8).

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. More complex systems (e.g. larger systems of nonlinear ODEs or nonlinear PDEs) can generate aperiodic dynamics corresponding to high- dimensional chaos.

17.3 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:

$$\mathbf{v}_i = \left[x\left(t_i\right), x\left(t_i - \tau\right), x\left(t_i - 2\tau\right), \ldots, x\left(t_i - (m-1)\tau\right)\right]^T.$$

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.

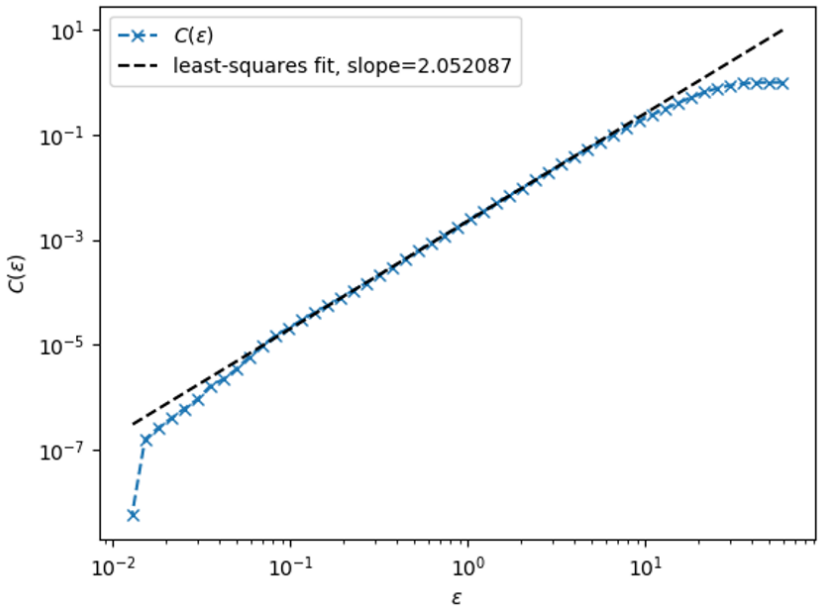


Figure 17.8: Correlation sum for Lorenz system.

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 useful for results from spatiotemporal data (e.g. solutions of PDEs) or from large systems of ODEs. 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.

17.4 Transitions between states

For the logistic map, we have seen that when $r = 3.5$, there are period-2 oscillations where: $x_{i+4} = x_i$, and at $r = 3.9$, we have chaos (Figure 17.9). What other states are there? When do they occur, and for what values of r are there transitions from one state to another? A simple approach to this question is to compute "all" values of x visited for a range of r over a sufficiently large number of iterations - 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 as in the code below:

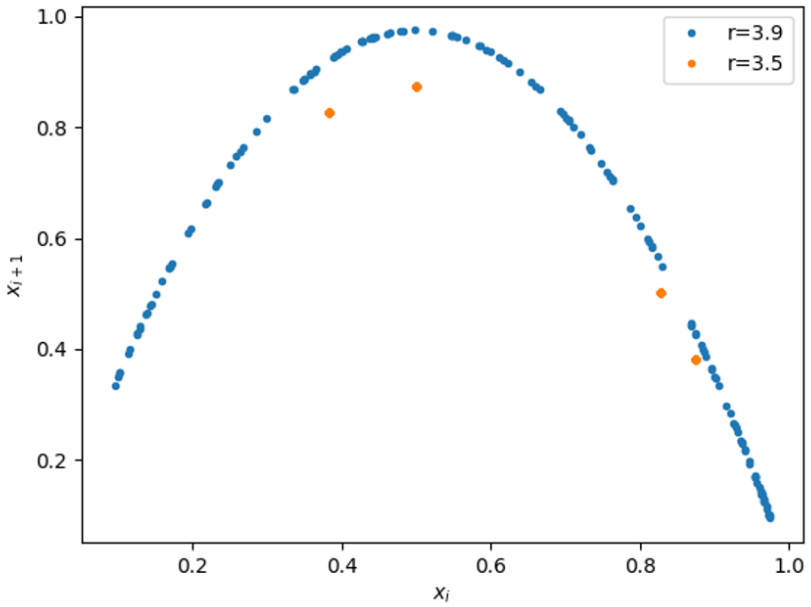


Figure 17.9: x_{i+1} vs. x_i for logistic map.

```
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 in Figure 17.10. This provides a concise global view of dynamics. What is it showing? We see:

- $r < 3.45$: 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" (e.g. around $r = 3.85$)!

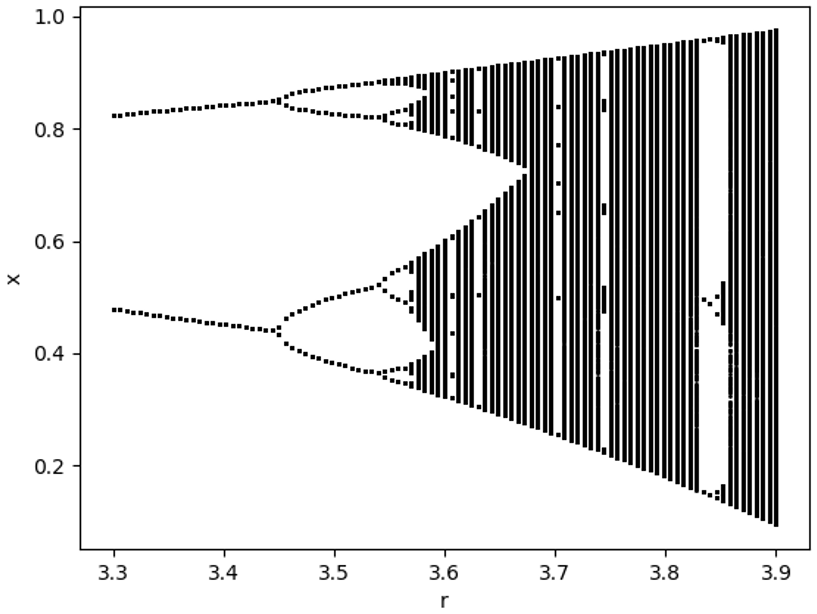


Figure 17.10: Orbit diagram for logistic map

A range of difference equations (maps) show similar behavior. Researchers have found that the map should look like an "upside-down U" (or V) in some sense to produce these trends.

What about differential equations? We have seen that ideas based on the fractal dimension can be carried over to ODEs. Are orbit diagrams for difference and differential equations similar in any sense? Can nonlinear ODEs be viewed as maps in some way?

$$\begin{aligned}\frac{dx}{dt} &= -y - z \\ \frac{dy}{dt} &= x + ay \\ \frac{dz}{dt} &= b + z(x - c)\end{aligned}$$

The Rossler system shown above is the simplest system of ODEs (that I'm aware of) which produces chaos. It has three model parameters, a, b, and c, and 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 in Figure 17.11. 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 by varying c. Computing solutions to the Rossler system is straightforward:

```
f = solve_ivp(RHS,[0,T],f0,t_eval=t,args=(a,b,c),method='BDF')
```

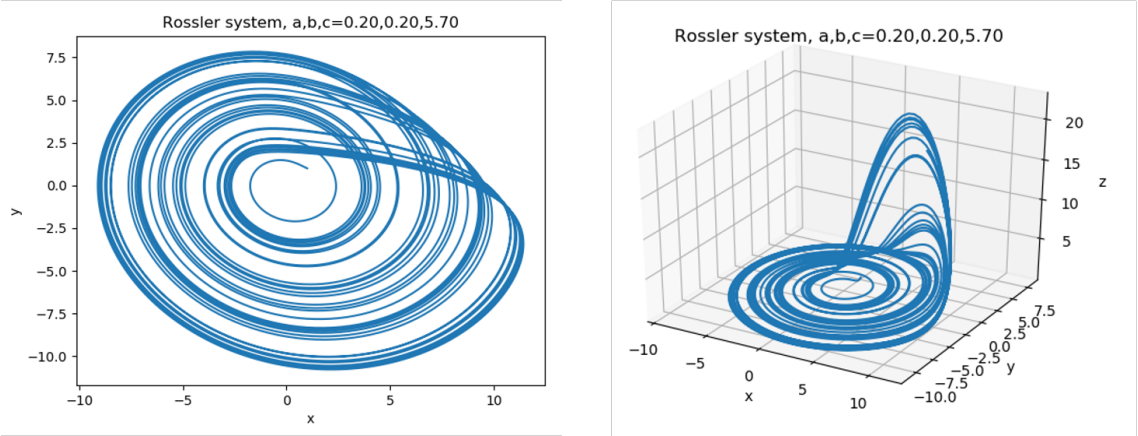


Figure 17.11: Solution trajectories for Rossler system.

Having computed the solution, we then discard the influence of initial condition:

```
f = f[iskip:,:]
```

Next, for each value of c, we extract the local maxima of x:

```
dx = np.diff(x)
d2x = dx[:-1]*dx[1:]
ind = np.argwhere(d2x<0) #locations of extrema
```

The variable `ind+1` contains locations of both maxima and minima, and we now need to separate out the maxima using the odd indices:

```
plot(t[ind[1::2]+1],x[ind[1::2]+1],'x')
```

This last line of code generates the "x"'s in Figure 17.12. Now that we can extract the maxima for a given value of c, we can construct an orbit diagram which shows all possible maxima for a broad range of c. The resulting figure is shown inFigure 17.13. 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 a similar diagram for larger, more complex systems.

Given the qualitative similarities in the orbit diagrams, is there an underlying map that can be extracted for the Rossler system? The idea for extracting such maps comes from Lorenz. Let f_n represent the nth maximum of $x(t)$ (already constructed for orbit diagram). Then plot f_{n+1} vs f_n. We expect *unimodal* maps to be "hiding" behind data generated by chaotic systems and Figure 17.14 indicates the existence of just such a map. There is a similar upside-down V-shaped map for the Lorenz equations, and similar maps have been constructed from measurements as well!

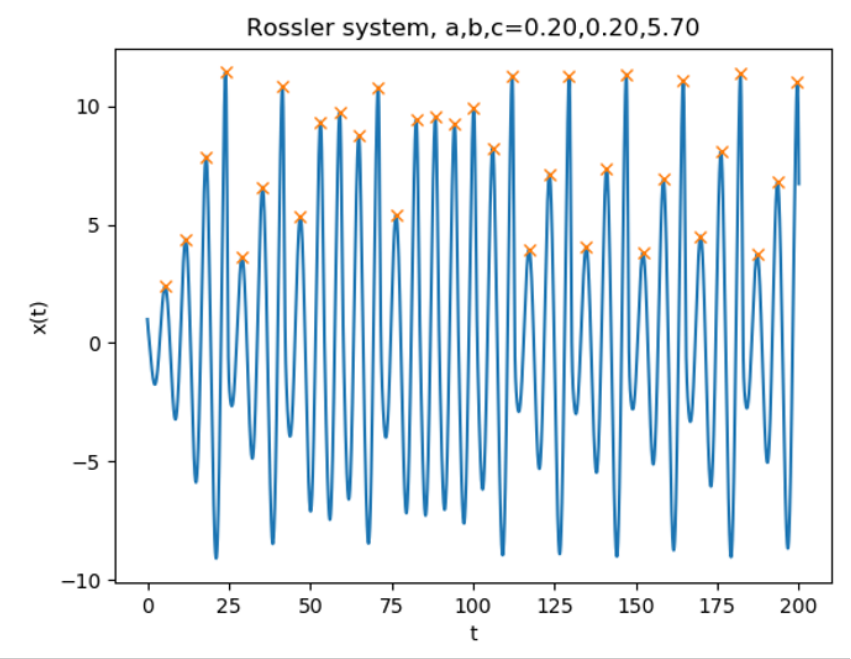


Figure 17.12: Numerical solution of $x(t)$ from Rossler system.

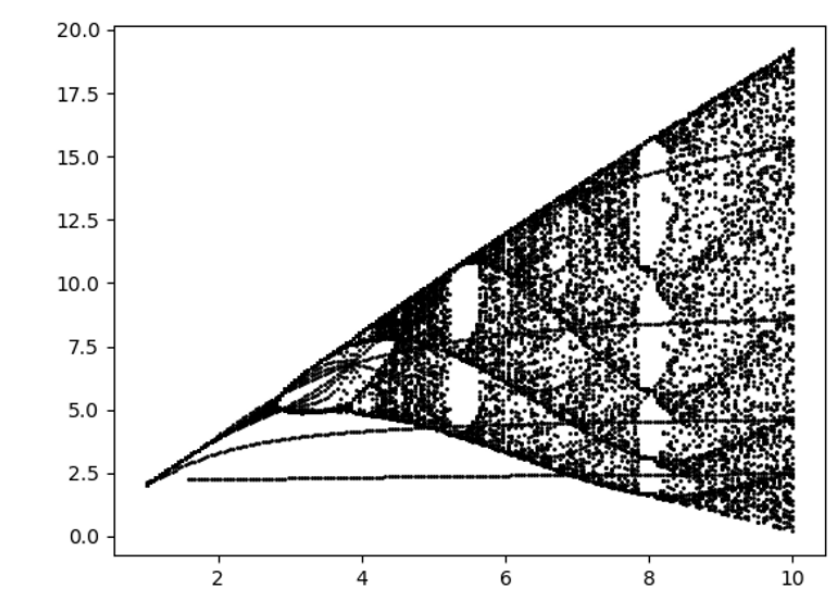


Figure 17.13: Orbit diagram for Rossler system

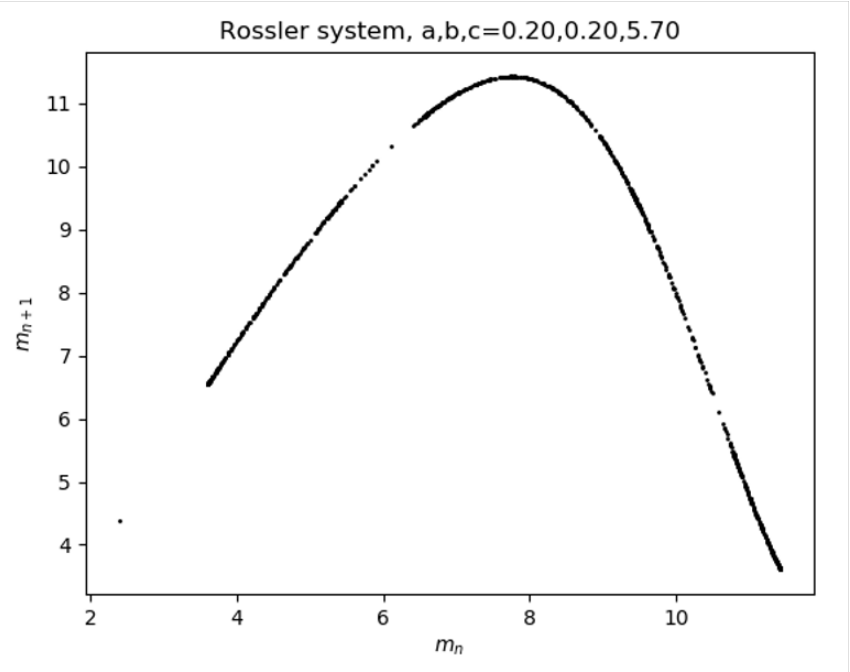


Figure 17.14: Map extracted from maxima of Rossler system.

17.5 Final comments

Another particularly important aspect of chaos that will be discussed elsewhere is sensitivity to initial conditions. Also note that I have ignored randomness when discussing time series analysis – 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

To be added

Lecture 19

Data assimilation

When analyzing complex systems that evolve in time, we often rely on mathematical models (e.g. differential equations) and numerical solutions of these models. However, these models are almost always imperfect representations of the real system, and when comparing simulations with measurements, there will be differences due to both model error and measurement error. We know that both simulation results and measurements are different from reality, but can we use them together to construct an estimate that is closer to the "truth"? We can, by using data assimilation methods where measurement data is assimilated with simulation results.

19.1 Kalman filter

We will look at one particular approach to data assimilation: the *extended Kalman filter*. The basic idea is to combine (imperfect) simulation results with (imperfect) measurements to obtain predictions better than those provided by simulations or measurements alone. A fundamental assumption is that model errors and measurement errors are both normally distributed random variables. Note that "model error" here is defined using the difference between the modeled dynamics and the true dynamics. Various versions of the Kalman filter have been used in a wide array of important applications. The Apollo lunar missions relied on Kalman filters to improve navigation as have many GPS devices.

19.2 Problem setup

We work with the following n-element vectors:

$\mathbf{x_k}$: Simulation solution at time t_k (unknown, deterministic)
$\boldsymbol{y_k}$: Measurements at time t_k (random)
$\mathbf{z_k}$: True physical solution at time t_k (unknown)
$\hat{\mathbf{z}}_{\boldsymbol{k}}$: Estimate for true physical solution at t_k (goal)

We assume that we know the covariance matrix for all measurements, $\boldsymbol{y_k}, k = 1, 2, \ldots$. The true and simulation initial conditions are related by, $\mathbf{z}_0 = \mathbf{x}_0 + \zeta$ where $\zeta \sim N(0, \Sigma_0)$, and the covariance matrix Σ_0 is assumed to be known. The *model* tells us how to update the simulation solution: $\mathbf{x_k} = \boldsymbol{f}(\mathbf{x_{k-1}})$. We also use the model to define the true solution: $\boldsymbol{z_k} = \boldsymbol{f}(\boldsymbol{z_{k-1}}) + \boldsymbol{\eta_k}$. The model error, $\boldsymbol{\eta}_k$, is a normally distributed random variable: $\boldsymbol{\eta}_k \sim \boldsymbol{N}(\mathbf{0}, \boldsymbol{\Sigma_\eta})$. We also have a normally distributed measurement error, $\boldsymbol{\mu}_k = \mathbf{z}_k - \mathbf{y}_k, \boldsymbol{\mu}_k \sim \boldsymbol{N}(\mathbf{0}, \boldsymbol{\Sigma_\mu})$. The measurement covariance matrix, $\boldsymbol{\sigma_\mu}$, is assumed to be known.

For example, we could be interested in atmospheric dynamics and the Lorenz system could be our model:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x),\\ \frac{dy}{dt} &= rx - y - xz,\\ \frac{dz}{dt} &= xy - bz.\end{aligned}$$

Then, $n = 3$, and $\mathbf{x}_k = [x(t_k), y(t_k), z(t_k)]^T$ could be obtained by computing numerical solutions to the system as we have seen before. Measurements of corresponding atmospheric variables would be $\mathbf{y}_k$, and we need to figure out how to use $\mathbf{x}_k$ and $\mathbf{y}_k$ to obtain $\widehat{\mathbf{z}}_k$, an estimate for the true values of the variables at time t_k.

The extended Kalman filter is one tool for constructing such an estimate. We will break the derivation of the Kalman filter into two parts:

1. Derive an update equation for the covariance matrix $\mathbf{C}_k = \left\langle (\mathbf{x}_k - \mathbf{z}_k)(\mathbf{x}_k - \mathbf{z}_k)^T \right\rangle$ which tells us how much the simulation results tends to depart from the true solution

2. Solve an optimization problem (likelihood maximization) which tells us how to combine the simulation results and measurements using the model and measurement variances and covariances.

19.3 Model covariance update

Based on the problem setup and the definition of $\mathbf{C}_k$, we know:

$$\mathbf{C}_k = \left\langle (\mathbf{x_k} - \mathbf{z_k})(\mathbf{x}_k - \mathbf{z}_k)^T \right\rangle = \left\langle [\boldsymbol{f}(\mathbf{x_{k-1}}) - \boldsymbol{f}(\mathbf{z_{k-1}}) - \boldsymbol{\eta_k}][\boldsymbol{f}(\mathbf{x}_{k-1}) - \boldsymbol{f}(\mathbf{z}_{k-1}) - \boldsymbol{\eta}_k]^T \right\rangle . \tag{1}$$

Next, we use a Taylor expansion to relate $\boldsymbol{f}(\mathbf{z}_{k-1})$ to $\boldsymbol{f}(\mathbf{x}_{k-1})$:

$$\boldsymbol{f}(\mathbf{z}_{k-1}) = \boldsymbol{f}(\mathbf{x}_{k-1}) + \left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k-1}} (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) + \text{ higher order terms}$$

A key assumption is that these higher-order terms can be neglected (e.g. the simulation result is "close" to the true result). Substituting the truncated Taylor series expansion into (1) gives...

$$\mathbf{C}_k \approx \left\langle \left[\left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathrm{x}=\mathbf{x}_{k-1}} (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) \right] \left[\left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathrm{x}=\mathbf{x}_{k-1}} (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) \right]^T + \boldsymbol{\eta}_k \left[\left(\left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k-1}} \right) (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) \right] + \left[\left(\left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{\mathbf{k}-1}} \right) (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) \right] \boldsymbol{\eta}_k^T + \boldsymbol{\eta}_k \boldsymbol{\eta}_k^T \right\rangle$$

This equation looks pretty complicated but can be simplified substantially by taking the following steps:

1. Use linearity of expectation to rewrite the RHS as a sum of four expectations
2. Notice that $\frac{d\boldsymbol{f}}{d\mathbf{x}}$ is deterministic
3. Assume that the model error at time t_k is uncorrelated with the simulation and true solution at t_{k-1}:

$\langle \boldsymbol{\eta}_k (\mathbf{x}_{k-1} - \mathbf{z}_{k-1}) \rangle = \langle \boldsymbol{\eta}_k \rangle \langle (\mathbf{x_{k-1}} - \mathbf{z_{k-1}}) \rangle = 0$ (the problem setup sets $\langle \boldsymbol{\eta}_k \rangle = 0$.)

4. Notice that $\left\langle (\mathbf{x}_{k-1} - \mathbf{z}_{k-1})(\mathbf{x_{k-1}} - \mathbf{z_{k-1}})^T \right\rangle = \mathbf{C_{k-1}}$

This leads us to the desired update equation:

$$\mathbf{C}_k \approx \left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k-1}} \mathbf{C}_{k-1} \left(\left.\frac{d\boldsymbol{f}}{d\mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k-1}} \right)^T + \left\langle \boldsymbol{\eta}_k \boldsymbol{\eta}_k^T \right\rangle . \tag{19.1}$$

Note that $\frac{d\boldsymbol{f}}{d\boldsymbol{x}}$ is the $n \times n$ Jacobian matrix for the model, and $\langle \boldsymbol{\eta}_k \boldsymbol{\eta}_k^T \rangle = \boldsymbol{\Sigma}_\eta$ is the diagonal $n \times n$ model covariance matrix which must be specified.

19.4 Kalman gain matrix

The remaining step is to solve an optimization problem (likelihood maximization) which tells us how to construct a linear combination of the simulation results and measurements using the model and measurement variances. The result of the linear combination is $\widehat{\mathbf{z}}_k$:

$$\widehat{\mathbf{z}}_k = \mathbf{x}_k + \mathbf{K}_\mathrm{k} \left(\mathbf{y}_k - \mathbf{x}_k\right)$$

$\mathbf{K}_k$ is the *Kalman gain matrix*, and values close to one place a greater weight on measurements while small gain means that the simulation results receive a larger weight.

The Kalman gain is given by, $\mathbf{K}_k = \mathbf{C}_k \left(\mathbf{C}_k + \langle \boldsymbol{\mu}\boldsymbol{\mu}^T \rangle\right)^{-1}$. Here, $\langle \boldsymbol{\mu}\boldsymbol{\mu}^T \rangle = \boldsymbol{\Sigma}_\mu$ is the diagonal covariance matrix for the measurements which is assumed to be known. We use our update equation to compute $\mathbf{C}_k$.

19.4.1 Likelihood maximization

How is the expression for the Kalman gain derived? Our goal is to find the best estimate of the true solution we can (i.e. the best $\hat{\mathbf{z}}_k$). Given $\mathbf{x}_k$, we will treat $\widehat{\mathbf{z}}_k$ as a multivariate Gaussian random variable which has mean $\mathbf{x}_k$ and covariance matrix, $\mathbf{C}_k : \hat{\mathbf{z}}_k \sim N\left(\mathbf{x}_k, \mathbf{C}_k\right)$. We relate $\hat{\mathbf{z}}_k$ and the measurements via the conditional probability $P\left(\mathbf{y}_k \mid \widehat{\mathbf{z}}_k\right)$ which is required to follow a multivariate Gaussian distribution:

$$P\left(\mathbf{y}_k \mid \widehat{\mathbf{z}}_k\right) = N\left(\mathbf{z}_k, \boldsymbol{\Sigma}_\mu\right).$$

Our goal is to find $\hat{\mathbf{z}}_k$ such that $P(\widehat{\mathbf{z}}_k \mid \mathbf{y}_k)$ is maximized. Bayes' theorem tells us that

$$P(\widehat{\mathbf{z}}_k \mid \mathbf{y}_k) = \frac{P\left(\mathbf{y}_k \mid \widehat{\mathbf{z}}_k\right) P(\hat{\mathbf{z}}_k)}{P(\mathbf{y}_k)}.$$

Note that we can ignore $P(\mathbf{y}_k)$ which is not affected by $\hat{\mathbf{z}}_k$. We then find $\hat{\mathbf{z}}_k$ which maximizes:

$$P\left(\hat{\mathbf{z}}_k \mid \mathbf{y}_k\right) = C \exp\left[-\frac{1}{2}\left(\hat{\mathbf{z}}_k - \mathbf{x}_k\right)^T \mathbf{C}_k^{-1} \left(\hat{\mathbf{z}}_k - \mathbf{x}_k\right)\right] \exp\left[-\frac{1}{2}\left(\mathbf{y}_k - \hat{\mathbf{z}}_k\right)^T \boldsymbol{\Sigma}_\mu^{-1} \left(\mathbf{y}_k - \hat{\mathbf{z}}_k\right)\right]$$

where C is a normalizing constant. It is more convenient to work with the log-likelihood and to find $\hat{\mathbf{z}}_k$ such that the negative log-likelihood $(-\mathcal{L})$ is minimized:

$$-2\mathcal{L} = C\left(\hat{\mathbf{z}}_k - \mathbf{x}_k\right)^T \mathbf{C}_k^{-1} \left(\hat{\mathbf{z}}_k - \mathbf{x}_k\right) + \left(\mathbf{y}_k - \hat{\mathbf{z}}_k\right)^T \boldsymbol{\Sigma}_\mu^{-1} \left(\mathbf{y}_k - \hat{\mathbf{z}}_k\right).$$

Working through the arithmetic, we find the solution stated previously:

$$\widehat{\mathbf{z}}_k = \mathbf{x}_k + \mathbf{K}_\mathrm{k} \left(\mathbf{y}_k - \mathbf{x}_k\right), \tag{19.2a}$$

$$\mathbf{K}_k = \mathbf{C}_k \left(\mathbf{C}_k + \boldsymbol{\Sigma}_\mu\right)^{-1}. \tag{19.2b}$$

Notes:

1. To see how the minimizer is found, it is easier to first consider the scalar problem ($n = 1$) where the covariance matrices are just scalar variances.

2. This presentation of the derivation is not the only way to approach this problem; see the comments at the end of these slides for references that provide different perspectives.

19.5 Algorithm overview

We have nearly all of the results needed to describe the extended Kalman filter method. The initial model covariance matrix, $\mathbf{C}_0$ will be the (specified) diagonal covariance matrix for the (specified) model initial condition. Then, to go from t_0 to t_1:

1. Update model covariance to obtain $\mathbf{C}_1$

2. Compute Kalman gain, $\mathbf{K}_1$

3. Update model solution, $\mathbf{x}_1 = \mathbf{f}(\mathbf{x}_0)$

4. Compute estimate of true solution, $\hat{\mathbf{z}}_1 = \mathbf{x}_1 + \mathbf{K}_1 (\mathbf{y}_1 - \mathbf{x}_1)$

However, we need to modify the method for further time steps. Specifically, the next model solution update should use $\hat{\mathbf{z}}_1$ rather than $\mathbf{x}_1$: $\mathbf{x}_2 = \mathbf{f}(\hat{\mathbf{z}}_1)$ as this is our best estimate for the true solution at t_1. The model covariance, $\mathbf{C}_1$, then should be modified to,

$$\mathbf{C}_1^{(new)} = \left\langle (\hat{\mathbf{z}}_1 - \mathbf{z}_1) (\hat{\mathbf{z}}_1 - \mathbf{z}_1)^T \right\rangle .$$

An expression for this new matrix can be derived using (19.2). We will skip the details and simply state the general result: $\mathbf{C}_k^{(new)} = (\mathbf{I} - \mathbf{K}_k) \mathbf{C}_k$. To derive this result, we make the (very reasonable) assumption that the model and measurement errors are uncorrelated, $\langle \boldsymbol{\eta}_k \boldsymbol{\mu}_k^T \rangle = \langle \boldsymbol{\eta}_k \rangle \langle \boldsymbol{\mu}_k^T \rangle = 0$. We can now describe a general step from t_{k-1} to t_k:

1. Update model covariance to obtain $\mathbf{C}_k$ from $\mathbf{C}_{k-1}^{(new)}$. Note that $\hat{\mathbf{z}}_{k-1}$ should be used when updating the covariance matrix rather than $\mathbf{x}_{k-1}$.

2. Compute Kalman gain, $\mathbf{K}_k$

3. Update model solution, $\mathbf{x}_k = \mathbf{f}(\hat{\mathbf{z}}_{k-1})$

4. Compute estimate of true solution, $\hat{\mathbf{z}}_k = \mathbf{x}_k + \mathbf{K} (\mathbf{y}_k - \mathbf{x}_k)$

5. Compute $\mathbf{C}_k^{(new)}$

19.6 Example: Lorenz system

Let's look at an example using the Lorenz equations. We'll say that the true physical solution is given by the Lorenz system for a particular initial condition. We usually don't know the true solution, but this approach will allow us to evaluate the effectiveness of the filter. We'll generate "measurements" by adding noise to the true solution at a set of equally spaced points in time. The model simulation will be the solution to the Lorenz system which starts with noise added to the initial condition for the true problem. Note that there is no model error in this example ($\mathbf{\Sigma}_\eta = 0$). We know that in chaotic systems, a small change in the initial condition will lead to dramatic differences at later times (lecture 18, sensitivity to initial conditions). Will the Kalman filter allow us to use the "measurements" to stay closer to the true solution for longer?

When applying the Kalman filter, the times t_k are the times when measurements are available (and at t_0 we have the initial condition). Each iteration, we move from t_{k-1} to t_k with the procedure described above. We use `solve_ivp` to compute both the true solution and to advance the model solution each time step (step 3). There is, however, one complication. We have presented a method for discrete models, while the Lorenz system varies continuously in time. This is not an issue when updating the simulation; we just numerically integrate the ODEs from t_{k-1} to t_k. But the model covariance update requires greater care. In particular, the Jacobian in (19.1) **is not** the Jacobian of the right-hand-side of the Lorenz system. We adopt a simple (crude) approach here, and use the explicit Euler method to generate a discrete version of the Lorenz system. For example, the equation for x becomes,

$$x_k = x_{k-1} + \Delta t f(x_{k-1}), \tag{19.3}$$

where $f(x_{k-1}) = \sigma(y_{k-1} - x_{k-1})$, and $\Delta t = t_k - t_{k-1}$. Let $\mathbf{J}(\mathbf{x}_k)^L$ be the Jacobian of the Lorenz system evaluated at time t_k. Then the Jacobian in (19.1) will be

$$\frac{d\boldsymbol{f}}{d\mathbf{x}}|_{\mathbf{x}=\mathbf{x}_{k-1}} = \mathbf{I} + \Delta t \mathbf{J}(\mathbf{x}_{k-1})^L.$$

Implementing the method with this adjustment, we can iterate forward in time. When we advance the model solution, we are generating a *forecast* of the dynamics between t_{k-1} and t_k and then the filter adjusts the results at t_k to generate the initial condition for the next forecast.

Figure 19.1 shows the true solution (solid curve), a simulation with a perturbed initial condition (dotted curve), and the measurements. We can clearly see tangible differences between the two simulations. Figure 19.2 shows just the first variable in the Lorenz system, $x(t)$. Like the previous figure, it includes the true solution, a simulation with a perturbed initial condition, and the measurements. Ita also include the extended Kalman filter estimates for the true solution. We can clearly see that we can do much better than "model-only" by incorporating measurements using the extended Kalman filter.

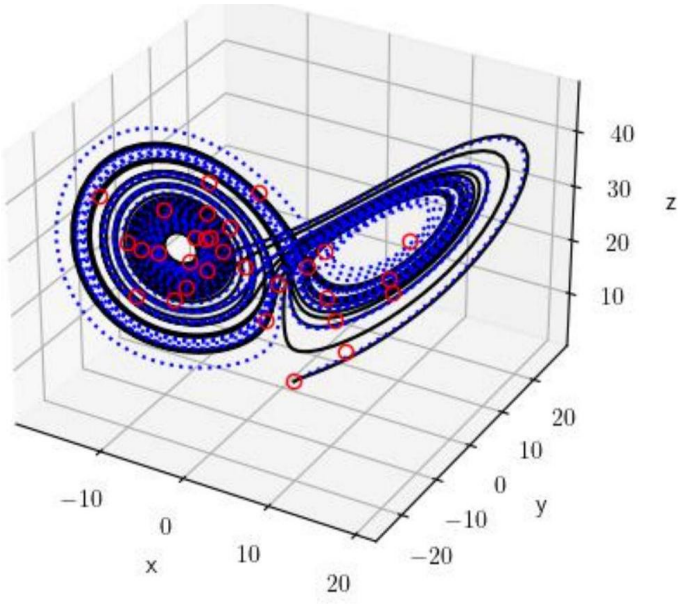


Figure 19.1: Lorenz system solutions and "measurements". Only every 100th measurement is shown.

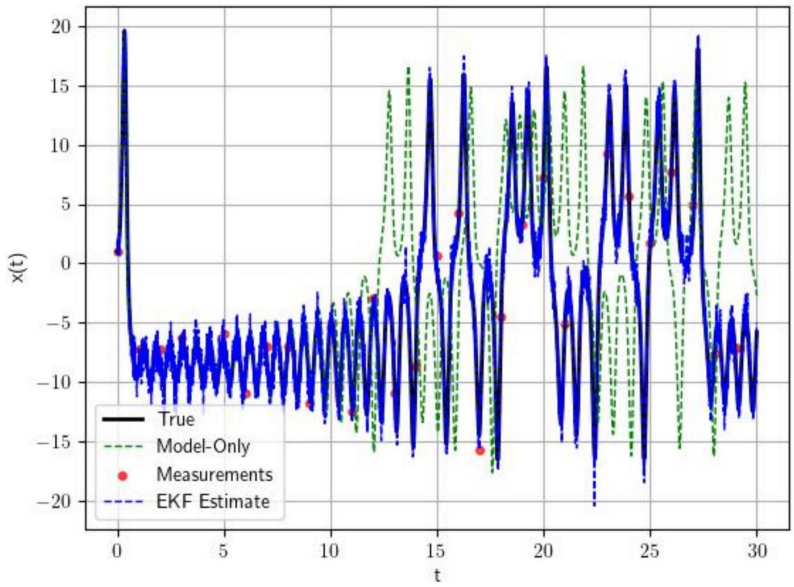


Figure 19.2: Lorenz system solutions, "measurements", and Kalman filter estimate of true solution for $x(t)$. Only every 100th measurement is shown.

Final comments

Much of the material in this lecture is based on Chapter 21 in [5]. Prof. Kutz is a leading expert in data-driven science and engineering. See https://faculty.washington.edu/kutz/KutzBook/page27.html

for links to the book chapter and video lectures. I have taken a few "shortcuts" in the derivations presented here. Notably, I have assumed that the measurements correspond to the state variables while typically, a nonlinear mapping between the measured and state variables is included in the formulation. See Kutz for more detail, and see G.A. Terejanu, Extended Kalman Filter Tutorial for a different perspective on the derivation. Data assimilation is one of the most important areas of current scientific computing research, and the aim of this lecture is just to give a sense of how data assimilation problems can be approached.

Bibliography

[1] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge: Cambridge University Press, 2016. ISBN: 9781107076266 1107076269. URL: http://barabasi.com/networksciencebook/.

[2] *DNA double helix (13081113544)" by Genomics Education Programme is licensed under CC BY 2.0.*

[3] Raffaele Ferrari and Carl Wunsch. "Ocean Circulation Kinetic Energy: Reservoirs, Sources, and Sinks". In: *Annual Review of Fluid Mechanics* 41.Volume 41, 2009 (2009), pp. 253–282. ISSN: 1545-4479. DOI: https://doi.org/10.1146/annurev.fluid.40.111406.102139. URL: https://www.annualreviews.org/content/journals/10.1146/annurev.fluid.40.111406.102139.

[4] Yehuda Koren, Robert Bell, and Chris Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8 (2009), pp. 30–37. DOI: 10.1109/MC.2009.263.

[5] J. Nathan Kutz. *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. USA: Oxford University Press, Inc., 2013. ISBN: 0199660344.

[6] Hannah Ritchie. "Weather forecasts have become much more accurate; we now need to make them available to everyone". In: *Our World in Data* (2024). https://ourworldindata.org/weather-forecasts.

[7] Max Roser, Hannah Ritchie, and Edouard Mathieu. "What is Moore's Law?" In: *Our World in Data* (2023). https://ourworldindata.org/moores-law.

[8] Ulrich Rüde et al. *Research and Education in Computational Science and Engineering*. 2018. arXiv: 1610.02608 [cs.CE]. URL: https://arxiv.org/abs/1610.02608.