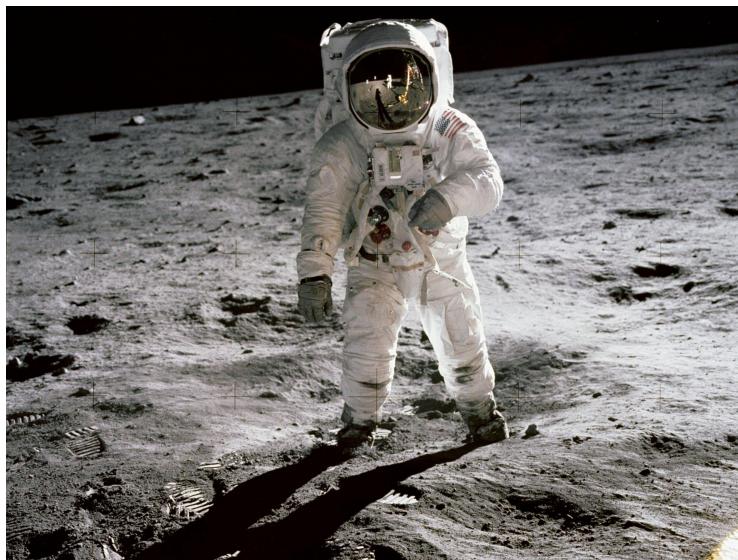


Labs for Foundations of Applied Mathematics

Volume 1
Mathematical Analysis

Jeffrey Humpherys & Tyler J. Jarvis, managing editors



List of Contributors

B. Barker <i>Brigham Young University</i>	R. Dorff <i>Brigham Young University</i>
E. Evans <i>Brigham Young University</i>	B. Ehler <i>Brigham Young University</i>
R. Evans <i>Brigham Young University</i>	M. Fabiano <i>Brigham Young University</i>
J. Grout <i>Drake University</i>	K. Finlinson <i>Brigham Young University</i>
J. Humpherys <i>Brigham Young University</i>	J. Fisher <i>Brigham Young University</i>
T. Jarvis <i>Brigham Young University</i>	R. Flores <i>Brigham Young University</i>
J. Whitehead <i>Brigham Young University</i>	R. Fowers <i>Brigham Young University</i>
J. Adams <i>Brigham Young University</i>	A. Frandsen <i>Brigham Young University</i>
K. Baldwin <i>Brigham Young University</i>	R. Fuhriman <i>Brigham Young University</i>
J. Bejarano <i>Brigham Young University</i>	T. Gledhill <i>Brigham Young University</i>
A. Berry <i>Brigham Young University</i>	S. Giddens <i>Brigham Young University</i>
Z. Boyd <i>Brigham Young University</i>	C. Gigena <i>Brigham Young University</i>
M. Brown <i>Brigham Young University</i>	M. Graham <i>Brigham Young University</i>
A. Carr <i>Brigham Young University</i>	F. Glines <i>Brigham Young University</i>
C. Carter <i>Brigham Young University</i>	C. Glover <i>Brigham Young University</i>
T. Christensen <i>Brigham Young University</i>	M. Goodwin <i>Brigham Young University</i>
M. Cook <i>Brigham Young University</i>	R. Grout <i>Brigham Young University</i>

- D. Grundvig
Brigham Young University
- S. Halverson
Brigham Young University
- E. Hannesson
Brigham Young University
- K. Harmer
Brigham Young University
- J. Henderson
Brigham Young University
- J. Hendricks
Brigham Young University
- A. Henriksen
Brigham Young University
- I. Henriksen
Brigham Young University
- C. Hettinger
Brigham Young University
- S. Horst
Brigham Young University
- R. Howell
Brigham Young University
- E. Ibarra-Campos
Brigham Young University
- J. Larsen
Brigham Young University
- K. Jacobson
Brigham Young University
- R. Jenkins
Brigham Young University
- J. Leete
Brigham Young University
- Q. Leishman
Brigham Young University
- J. Lytle
Brigham Young University
- E. Manner
Brigham Young University
- M. Matsushita
Brigham Young University
- R. McMurray
Brigham Young University
- S. McQuarrie
Brigham Young University
- D. Miller
Brigham Young University
- J. Morrise
Brigham Young University
- M. Morrise
Brigham Young University
- A. Morrow
Brigham Young University
- R. Murray
Brigham Young University
- J. Nelson
Brigham Young University
- C. Noorda
Brigham Young University
- A. Oldroyd
Brigham Young University
- A. Oveson
Brigham Young University
- E. Parkinson
Brigham Young University
- M. Probst
Brigham Young University
- M. Proudfoot
Brigham Young University
- D. Reber
Brigham Young University
- H. Ringer
Brigham Young University
- C. Robertson
Brigham Young University
- M. Russell
Brigham Young University
- R. Sandberg
Brigham Young University
- C. Sawyer
Brigham Young University
- D. Smith
Brigham Young University
- J. Smith
Brigham Young University
- P. Smith
Brigham Young University
- M. Stauffer
Brigham Young University

E. Steadman

Brigham Young University

J. Stewart

Brigham Young University

S. Suggs

Brigham Young University

A. Tate

Brigham Young University

T. Thompson

Brigham Young University

M. Victors

Brigham Young University

E. Walker

Brigham Young University

J. Webb

Brigham Young University

R. Webb

Brigham Young University

J. West

Brigham Young University

R. Wonnacott

Brigham Young University

A. Zaitzeff

Brigham Young University

This project is funded in part by the National Science Foundation, grant no. TUES Phase II
DUE-1323785.

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics Volume 1: Mathematical Analysis* by Humpherys, Jarvis and Evans. The labs focus mainly on important numerical linear algebra algorithms, with applications to images, networks, and data science. The reader should be familiar with Python [VD10] and its NumPy [Oli06, ADH⁺01, Oli07] and Matplotlib [Hun07] packages before attempting these labs. See the Python Essentials manual for introductions to these topics.

©This work is licensed under the Creative Commons Attribution 3.0 United States License. You may copy, distribute, and display this copyrighted work only if you give credit to Dr. J. Humpherys. All derivative works must include an attribution to Dr. J. Humpherys as the owner of this work as well as the web address to

<https://github.com/Foundations-of-Applied-Mathematics/Labs>
as the original source of this work.

To view a copy of the Creative Commons Attribution 3.0 License, visit

<http://creativecommons.org/licenses/by/3.0/us/>
or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.



Contents

Preface	v
I Labs	1
1 Introduction to the Unix Shell	3
2 The Standard Library	17
3 Object-oriented Programming	37
4 Exceptions and File Input/Output	49
5 Linear Transformations	63
6 Linear Systems	75
7 The QR Decomposition	89
8 Least Squares and Computing Eigenvalues	101
9 Image Segmentation	115
10 The SVD and Image Compression	125
11 Facial Recognition	135
12 Data Visualization	143
13 Convolution and Filtering	157
14 Introduction to SymPy	165
15 Differentiation	179
16 Newton's Method	189
17 Conditioning and Stability	197

18	Monte Carlo Integration	209
19	Visualizing Complex-valued Functions	215
20	The PageRank Algorithm	223
21	Profiling	235
22	SQL 1: Introduction	251
23	SQL 2 (The Sequel)	263
24	Iterative Solvers	273
II	Appendices	283
A	Getting Started	285
B	Installing and Managing Python	293
C	NumPy Visual Guide	299
	Bibliography	303

Part I

Labs

1

Unix Shell 1: Introduction

Lab Objective: Unix is a popular operating system that is commonly used for servers and the basis for most open source software. Using Unix for writing and submitting labs will develop a foundation for future software development. In this lab we explore the basics of the Unix shell, including how to navigate and manipulate files, access remote machines with Secure Shell, and use Git for basic version control.

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of the Linux and MacOSX operating systems. Most servers are Linux-based, so knowing how to use Unix shells allows us to interact with servers and other Unix-based machines.

A Unix shell is a program that takes commands from a user and executes those commands on the operating system. We interact with the shell through a terminal (also called a command line), a program that lets you type in commands and gives those commands to the shell for execution.

Note

Windows is not built off of Unix, but it does come with a terminal called PowerShell. This terminal uses a different command syntax. We will not cover the equivalent commands in the Windows terminal, but you could download a Unix-based terminal such as Git Bash or Cygwin to complete this lab on a Windows machine (you will still lose out on certain commands). Alternatively, Windows 10 now offers a Windows Subsystem for Linux, WSL, which is a Linux operating system downloaded onto Windows.

Note

For this lab we will be working in the `UnixShell1` directory provided with the lab materials. If you have not yet downloaded the code repository, follow steps 1 through 6 in the **Getting Started** guide found at <https://foundations-of-applied-mathematics.github.io/> before proceeding with this lab. Make sure to run the `download_data.sh` script as described in step 5 of **Getting Started**; otherwise you will not have the necessary files to complete this lab.

Basic Unix Shell

Shell Scripting

The following sections of the lab will explore several shell commands. You can execute these commands by typing these commands directly into a terminal. Sometimes, though, you will want to execute a more complicated sequence of commands, or make it easy to execute the same set of commands over and over again. In those cases, it is useful to create a script, which is a sequence of shell commands saved in a file. Then, instead of typing the commands individually, you simply have to run the script, and it takes care of running all the commands.

In this lab we will be running and editing a bash script. Bash is the most commonly used Unix shell and is the default shell installed on most Unix-based systems.

The following is a very simple bash script. The command `echo <string>` prints `<string>` in the terminal.

```
#!/bin/bash
echo "Hello World!"
```

The first line, `#!/bin/bash`, tells the computer to use the bash interpreter to run the script, and where this interpreter is located. The `#!` is called the shebang or hashbang character sequence. It is followed by the absolute path to the bash interpreter.

To run a bash script, type `bash <script name>` into the terminal. Alternatively, you can execute any script by typing `./<script name>`, but note that the script must contain executable permissions for this to work. (We will learn more about permissions later in the lab.)

```
$ bash hello_world.sh
Hello World!
```

Navigation

Typically, people navigate computers by clicking on icons to open folders and programs. In the terminal, instead of point and click we use typed commands to move from folder to folder. In the Unix shell, we call folders directories. The file system is a set of nested directories containing files and other directories.

You can picture the file system as an tree, with directories as branches. Smaller branches stem from bigger branches, and all bigger branches eventually stem from the root of the tree. Similarly, in the Unix file system we have a "root directory", where all other directories are nested in. We denote it by using a single slash (/). All absolute paths originate at the root directory, which means all absolute path strings begin with the / character.

Begin by opening a terminal. The text you see in the upper left of the terminal is called the prompt. Before you start creating or deleting files, you'll want to know where you are. To see what directory you are currently working in, type `pwd` into the prompt. This command stands for print working directory, and it prints out a string telling you your current location.

```
~$ pwd
/home/username
```

To see the all the contents of your current directory, type the command `ls`, list segments.

```
~$ ls
Desktop      Downloads      Public      Videos
Documents    Pictures
```

The command `cd`, change directory, allows you to navigate directories. To change to a new directory, type the `cd` command followed by the name of the directory to which you want to move (if you `cd` into a file, you will get an error). You can move up one directory by typing `cd ...`

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing `cd ~` or just `cd`. You can navigate to root by typing `cd /`.

Problem 1. To begin, open a terminal and navigate to the `UnixShell1/` directory provided with this lab. Use `ls` to list the contents. There should be a file called `Shell1.zip` and a script called `unixshell1.sh`.^a

Run `unixshell1.sh`. This script will do the following:

1. Unzip `Shell1.zip`, creating a directory called `Shell1/`
2. Remove any previously unzipped copies of `Shell1/`
3. Execute various shell commands, to be added in the next few problems in this lab
4. Create a compressed version of `Shell1/` called `UnixShell1.tar.gz`.
5. Remove any old copies of `UnixShell1.tar.gz`

Now, open the `unixshell1.sh` script in a text editor. Add commands to the script, within the section for Problem 1, to do the following:

- Change into the `Shell1/` directory.
- Print a string (without using `echo`) telling you the directory you are currently working in.

Test your commands by running the script again and checking that it prints a string ending in the location `Shell1/`.

^aIf the necessary data files are not in your directory, `cd` one directory up by typing `cd ..` and type `bash download_data.sh` to download the data files for each lab.

Documentation and Help

When you encounter an unfamiliar command, the terminal has several tools that can help you understand what it does and how to use it. Most commands have manual pages, which give information about what the command does, the syntax required to use it, and different options to modify the command. To open the manual page for a command, type `man <command>`. Some commands also have an option called `--help`, which will print out information similar to what is contained in the manual page. To use this option, type `<command> --help`.

```
$ man ls
```

```
LS(1)                               User Commands                               LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILEs (the current directory by default).

    -a, --all
        do not ignore entries starting with .
```

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags

When you use `man`, you will see a list of options such as `-a`, `-A`, `--author`, etc. that modify how a command functions. These are called flags. You can use one flag on a command by typing `<command> -<flag>`, like `ls -a`, or combine multiple flags by typing `<command> -<flag1><flag2>`, etc. as in `ls -alt`.

For example, sometimes directories contain hidden files, which are files whose names begin with a dot character like `.bash`. The `ls` command, by default, does not list hidden files. Using the `-a` flag specifies that `ls` should not ignore hidden files. Find more common flags for `ls` in Table 1.1.

Flags	Description
<code>-a</code>	Do not ignore hidden files and folders
<code>-l</code>	List files and folders in long format
<code>-r</code>	Reverse order while sorting
<code>-R</code>	Print files and subdirectories recursively
<code>-s</code>	Print item name and size
<code>-S</code>	Sort by size
<code>-t</code>	Sort output by date modified

Table 1.1: Common flags of the `ls` command.

```
$ ls
file1.py  file2.py

$ ls -a
.  ..  file1.py  file2.py  .hiddenfile.py

$ ls -alt      # Multiple flags can be combined into one flag
total 8
```

```
drwxr-xr-x  2 c c 4096 Aug 14 10:08 .
-rw-r--r--  1 c c     0 Aug 14 10:08 .hiddenfile.py
-rw-r--r--  1 c c     0 Aug 14 10:08 file2.py
-rw-r--r--  1 c c     0 Aug 14 10:08 file1.py
drwxr-xr-x 38 c c 4096 Aug 14 10:08 ..
```

Problem 2. Within the script, add a command using `ls` to print one list of the contents of `Shell1/` with the following criteria:

- Include hidden files and folders
- List the files and folders in long format (include the permissions, date last modified, etc.)
- Sort the output by file size (largest files first)

Test your command by entering it into the terminal within `Shell1/` or by running the script and checking for the desired output.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

```
~$ cd Test/          # navigate to test directory

~/Test$ ls           # list contents of directory
file1.py

~/Test$ mkdir NewDirectory # create a new empty directory

~/Test$ touch newfile.py    # create a new empty file

~/Test$ ls
file1.py  NewDirectory  newfile.py
```

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, use the `-r` flag to recursively copy files contained in the directory. If you try to copy a directory without the `-r`, the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second.

If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

```
~/Test$ ls
file1.py  NewDirectory  newfile.py

~/Test$ mv newfile.py NewDirectory/          # move file into directory
```

```
~/Test$ cp file1.py NewDirectory/          # make a copy of file1 in directory
~/Test$ cd NewDirectory/
~/Test/NewDirectory$ mv file1.py newname.py # rename file1.py
~/Test/NewDirectory$ ls
newfile.py  newname.py
```

When deleting files, use `rm <filename>`, and when deleting a directory, use `rm -r <dir_name>`. The `-r` flag tells the terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to print strings in the terminal describing what it is doing.

When your terminal gets too cluttered, use `clear` to clean it up.

```
~/Test/NewDirectory$ cd ..                  # move one directory up
~/Test$ rm -rv NewDirectory/               # remove a directory and its contents
removed 'NewDirectory/newname.py'
removed 'NewDirectory/newfile.py'
removed directory 'NewDirectory/'

~/Test$ rm file1.py                      # remove a file
~/Test$ ls                               # directory is now empty
~/Test$
```

Commands	Description
<code>clear</code>	Clear the terminal screen
<code>cp file1 dir1</code>	Create a copy of <code>file1</code> and move it to <code>dir1/</code>
<code>cp file1 file2</code>	Create a copy of <code>file1</code> and name it <code>file2</code>
<code>cp -r dir1 dir2</code>	Create a copy of <code>dir1/</code> and all its contents into <code>dir2/</code>
<code>mkdir dir1</code>	Create a new directory named <code>dir1/</code>
<code>mkdir -p path/to/new/dir1</code>	Create <code>dir1/</code> and all intermediate directories
<code>mv file1 dir1</code>	Move <code>file1</code> to <code>dir1/</code>
<code>mv file1 file2</code>	Rename <code>file1</code> as <code>file2</code>
<code>rm file1</code>	Delete <code>file1</code> [-i, -v]
<code>rm -r dir1</code>	Delete <code>dir1/</code> and all items within <code>dir1/</code> [-i, -v]
<code>touch file1</code>	Create an empty file named <code>file1</code>

Table 1.2: File Manipulation Commands

Table 1.2 contains all the commands we have discussed so far. Commonly used flags for some commands are contained in square brackets; use `man` or `--help` to see what these mean.

Problem 3. Add commands to the `unixshell1.sh` script to make the following changes in `Shell1/`:

- Delete the `Audio/` directory along with all its contents
- Create `Documents/`, `Photos/`, and `Python/` directories

- Change the name of the Random/ directory to Files/

Test your commands by running the script and then using `ls` within `Shell1/` to check that each directory was deleted, created, or changed correctly.

Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, you may need to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using wildcards. We will use the * and ? wildcards. The * wildcard represents any string and the ? wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```
$ ls
File1.txt  File2.txt  File3.jpg  text_files

$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt

$ ls
File3.jpg  text_files
```

See Table 1.3 for examples of common wildcard usage.

Command	Description
<code>*.txt</code>	All files that end with <code>.txt</code> .
<code>image*</code>	All files that have <code>image</code> as the first 5 characters.
<code>*py*</code>	All files that contain <code>py</code> in the name.
<code>doc*.txt</code>	All files of the form <code>doc1.txt</code> , <code>doc2.txt</code> , <code>docA.txt</code> , etc.

Table 1.3: Common uses for wildcards.

Problem 4. Within the `Shell1/` directory, there are many files. Add commands to the script to organize these files into directories using wildcards. Organize by completing the following:

- Move all the `.jpg` files to the `Photos/` directory
- Move all the `.txt` files to the `Documents/` directory
- Move all the `.py` files to the `Python/` directory

Working With Files

Searching the File System

There are two commands we can use for searching through our directories. The `find` command is used to find files or directories with a certain name; the `grep` command is used to find lines within files matching a certain string. When searching for a specific string, both commands allow wildcards within the string. You can use wildcards so that your search string matches a broader set of strings.

```
# Find all files or directories in Shell1/ called "final"
# -type f,d specifies to look for files and directories
# . specifies to look in the current directory

$ find . -name "final" -type f,d
$           # There are no files with the exact name "final" in Shell1/

$ find . -name "*final*" -type f,d
./Files/May/finals
./Files/May/finals/finalproject.py
```

```
# Find all within files in Documents/ containing "Mary"
# -r tells grep to search all files with Documents/
# -n tells grep to print out the line number (2)

$ Shell1$ grep -nr "Mary" Documents/
Documents/people.txt:2:female,Mary,31
```

Command	Description
<code>find dir1 -type f -name "word"</code>	Find all files in <code>dir1/</code> (and its subdirectories) called <code>word</code> (-type <code>f</code> is for files; -type <code>d</code> is for directories)
<code>grep "word" filename</code>	Find all occurrences of <code>word</code> within <code>filename</code>
<code>grep -nr "word" dir1</code>	Find all occurrences of <code>word</code> within the files inside <code>dir1/</code> (-n lists the line number; -r performs a recursive search)

Table 1.4: Commands using `find` and `grep`.

Table 1.4 contains basic syntax for using these two commands. There are many more variations of syntax for `grep` and `find`, however. You can use `man grep` and `man find` to explore other options for using these commands.

File Security and Permissions

A file has three levels of permissions associated with it: the permission to read the file, to write (modify) the file, and to execute the file. There are also three categories of people who are assigned permissions: the user (the owner), the group, and others.

You can check the permissions for `file1` using the command `ls -l <file1>`. Note that your output will differ from that printed below; this is purely an example.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
drw-rw-r-- 1 username groupname 373 Aug  5 21:16 Documents
-rwxr-x--x 1 username groupname   27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

The first character of each line denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The next nine characters denote the permissions associated with that file.

For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx`, tell us the owner can read, write, and execute the file. The next three characters, `r-x`, tell us members of the same group can read and execute the file, but not edit it. The final three characters, `--x`, tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are multiple notations used to modify permissions, but the easiest to use when we want to make small modifications to a file's permissions is symbolic permissions notation. See Table 1.5 for more examples of using symbolic permissions notation, as well as other useful commands for working with permissions.

```
$ ls -l script1.sh
total 0
-rw-r--r-- 1 c c 0 Aug 21 13:06 script1.sh

$ chmod u+x script1.sh      # add permission for user to execute
$ chmod o-r script1.sh      # remove permission for others to read
$ ls -l script1.sh
total 0
-rwxr----- 1 c c 0 Aug 21 13:06 script1.sh
```

Command	Description
<code>chmod u+x file1</code>	Add executing (x) permissions to user (u)
<code>chmod g-w file1</code>	Remove writing (w) permissions from group (g)
<code>chmod o-r file1</code>	Remove reading (r) permissions from other other users (o)
<code>chmod a+w file1</code>	Add writing permissions to everyone (a)
<code>chown</code>	change owner
<code>chgrp</code>	change group
<code>getfacl</code>	view all permissions of a file in a readable format.

Table 1.5: Symbolic permissions notation and other useful commands

Running Files

To run a file for which you have execution permissions, type the file name preceded by `./`.

```
$ ./hello.sh
bash: ./hello.sh: Permission denied

$ ls -l hello.sh
```

```
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh  
$ chmod u+x hello.sh      # You can now execute the file  
$ ./hello.sh  
Hello World!
```

Problem 5. Within `Shell1/`, there is a script called `organize_photos.sh`. First, use `find` to locate the script. Once you know the file location, add commands to your script so that it completes the following tasks:

- Moves `organize_photos.sh` to `Scripts/`
- Adds executable permissions to the script for the user
- Runs the script

Test that the script has been executed by checking that additional files have been moved into the `Photos/` directory. Check that permissions have been updated on the script by using `ls -l`.

Accessing Remote Machines

At times you will find it useful to perform tasks on a remote computer or server, such as running a script that requires a large amount of computing power on a supercomputer or accessing a data file stored on another machine.

Secure Shell

Secure Shell (SSH) allows you to remotely access other computers or servers securely. SSH is a network protocol encrypted using public-key cryptography. It ensures that all communication between your computer and the remote server is secure and encrypted.

The system you are connecting to is called the host, and the system you are connecting from is called the client. The first time you connect to a host, you will receive a warning saying the authenticity of the host can't be established. This warning is a default, and appears when you are connecting to a host you have not connected to before. When asked if you would like to continue connecting, select yes.

When prompted for your password, type your password as normal and press enter. No characters will appear on the screen, but they are still being logged. Once the connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type exit.

```
alice@mycomputer:~$ ssh alice27@acme01.byu.edu  
alice27@acme01.byu.edu password:# Type password as normal  
last login 7 Sept 11
```

```
[alice27@byu.local@acme01 ~]$ ls      # Commands are executed on the host
myacmeshire/

[alice27@byu.local@acme01 ~]$ exit  # End a secure connection
logout
Connection to acme01.byu.edu closed.

alice@mycomputer:~$                                         # Commands are executed on the client
```

Secure Copy

To copy files from one computer to another, you can use the Unix command `scp`, which stands for secure copy protocol. The syntax for `scp` is essentially the same as the syntax for `cp`.

To copy a file from your computer to a specific location on a remote machine, use the syntax `scp <file1> <user@remote_host:>file_path`. As with `cp`, to copy a directory and all of its contents, use the `-r` flag.

```
# Make copies of file1 and dir2 in the home directory on acme01.byu.edu
alice@mycomputer:~$ scp file1 alice27@acme01.byu.edu:~
alice@mycomputer:~$ scp -r dir1/dir2 alice27@acme01.byu.edu:~/
```

Use the syntax `scp -r <user@remote_host:>file_path/dir1> <file_path>` to copy `dir1` from a remote machine to the location specified by `file_path` on your current machine.

```
# Make a local copy of dir1 (from acme01.byu.edu) in the home directory
alice@mycomputer:~$ scp -r alice27@acme01.byu.edu:~/dir1 ~
```

Commands	Description
<code>ssh username@remote_host</code>	Establish a secure connection with <code>remote_host</code>
<code>scp file1 user@remote_host:>file_path/</code>	Create a copy of <code>file1</code> on host
<code>scp -r dir1 user@remote_host:>file_path/</code>	Create a copy of <code>dir1</code> and its contents on host
<code>scp user@remote_host:>file_path/file1 file_path2</code>	Create a local copy of file on client

Table 1.6: Basic syntax for `ssh` and `scp`.

Problem 6. On a computer with the host name `acme20.byu.edu` or `acme21.byu.edu`, there is a file called `img_649.jpg`. Secure copy this file to your `UnixShell1/` directory. (Do not add the `scp` command to the script).

To `ssh` or `scp` on this computer, your username is your Net ID, and your password is your typical Net ID password. To use `scp` or `ssh` for this computer, you will have to be on campus using BYU Wifi.

Hint: To use `scp`, you will need to know the location of the file on the remote computer. Consider using `ssh` to access the machine and using `find`. The file is located somewhere in the directory `/sshlab`. Sometimes after logging onto the machine with `ssh`, there will appear to not be any directories you can access. Using the command `cd /` will fix this. When logging on initially, you also may get a message about not having a `myacmeshare`; this is not needed for this lab and the message may be ignored safely.

After secure copying, add a command to your script to copy the file from `UnixShell1/` into the directory `Shell1/Photos/`. (Make sure to leave a copy of the file in `UnixShell1/`, otherwise the file will be deleted when you run the script again.)

Git

Git is a version control system, meaning that it keeps a record of changes in a file. Git also facilitates collaboration between people working on the same code. It does both these things by managing updates between an online code repository and copies of the repository, called clones, stored locally on computers.

We will be using git to submit labs and return feedback on those labs. If git is not already installed on your computer, download it at <http://git-scm.com/downloads>.

Using Git

Git manages the history of a file system through commits, or checkpoints. Each time a new commit is added to the online repository, a checkpoint is created so that if need be, you can use or look back at an older version of the repository. You can use `git log` to see a list of previous commits. You can also use `git status` to see the files that have been changed in your local repository since the last commit.

Before making your own changes, you'll want to add any commits from other clones into your local repository. To do this, use the command `git pull origin master`.

Once you have made changes and want to make a new commit, there are normally three steps. To save these changes to the online repository, first add the changed files to the staging area, a list of files to save during the next commit, with `git add <filename(s)>`. If you want to add all changes that you have made to tracked files (files that are already included in the online repository), use `git add -u`.

Next, save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

Finally, add the changes in this commit to the online repository with `git push origin master`.

```
$ cd MyDirectory/          # Navigate into a cloned repository
$ git pull origin master   # Pull new commits from online repository

### Make changes to file1.py ###

$ git add file1.py          # Add file to staging area
$ git commit -m "Made changes" # Commit changes in staging area
$ git push origin master     # Push changes to online repository
```

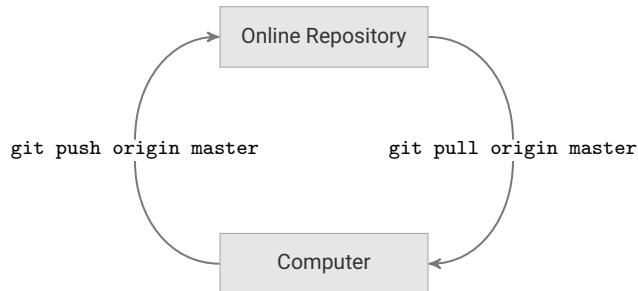


Figure 1.1: Exchanging git commits between the repository and a local clone.

Merge Conflicts

Git maintains order by raising an alert when changes are made to the same file in different clones and neither clone contains the changes made in the other. This is called a merge conflict, which happens when someone else has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have.

Achtung!

When pulling updates with `git pull origin master`, your terminal may sometimes display the following merge conflict message.

```

Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This screen, displayed in vim ([https://en.wikipedia.org/wiki/Vim_\(text_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))), is asking you to enter a message to create a merge commit that will reconcile both changes. If you do not enter a message, a default message is used. To close this screen and create the merge commit with the default message, type `:wq` (the characters will appear in the bottom left corner of the terminal) and press `enter`.

Note

Vim is a terminal text editor available on essentially any computer you will use. When working with remote machines through ssh, vim is often the only text editor available to use. To exit vim, press `esc:wq`. To learn more about vim, visit the official documentation at <https://vimhelp.org>.

Command	Explanation
<code>git status</code>	Display the staging area and untracked changes.
<code>git pull origin master</code>	Pull changes from the online repository.
<code>git push origin master</code>	Push changes to the online repository.
<code>git add <filename(s)></code>	Add a file or files to the staging area.
<code>git add -u</code>	Add all modified, tracked files to the staging area.
<code>git commit -m "<message>"</code>	Save the changes in the staging area with a given message.
<code>git checkout <filename></code>	Revert changes to an unstaged file since the last commit.
<code>git reset HEAD <filename></code>	Remove a file from the staging area, but keep changes.
<code>git diff <filename></code>	See the changes to an unstaged file since the last commit.
<code>git diff --cached <filename></code>	See the changes to a staged file since the last commit.
<code>git config --local <option></code>	Record your credentials (<code>user.name</code> , <code>user.email</code> , etc.).

Table 1.7: Common git commands.

Problem 7. Using git commands, push `unixshell1.sh` and `UnixShell1.tar.gz` to your online git repository. Do not add anything else in the `UnixShell1/` directory to the online repository.

2

The Standard Library

Lab Objective: Python is designed to make it easy to implement complex tasks with little code. To that end, every Python distribution includes several built-in functions for accomplishing common tasks. In addition, Python is designed to import and reuse code written by others. A Python file with code that can be imported is called a module. All Python distributions include a collection of modules for accomplishing a variety of tasks, collectively called the Python Standard Library. In this lab we explore some built-in functions, learn how to create, import, and use modules, and become familiar with the standard library.

Built-in Functions

Python has several built-in functions that may be used at any time. IPython's object introspection feature makes it easy to learn about these functions: start IPython from the command line and use `?` to bring up technical details on each function.

```
In [1]: min?  
Docstring:  
min(iterable, *, default=obj, key=func) -> value  
min(arg1, arg2, *args, *, key=func) -> value
```

With a single iterable argument, return its smallest item. The `default` keyword-only argument specifies an object to return if the provided iterable is empty.

With two or more arguments, return the smallest argument.

Type: builtin_function_or_method

```
In [2]: len?  
Signature: len(obj, /)  
Docstring: Return the number of items in a container.  
Type: builtin_function_or_method
```

Function	Returns
<code>abs()</code>	The absolute value of a real number, or the magnitude of a complex number.
<code>min()</code>	The smallest element of a single iterable, or the smallest of several arguments. Strings are compared based on lexicographical order: numerical characters first, then upper-case letters, then lower-case letters.
<code>max()</code>	The largest element of a single iterable, or the largest of several arguments.
<code>len()</code>	The number of items of a sequence or collection.
<code>round()</code>	A float rounded to a given precision in decimal digits.
<code>sum()</code>	The sum of a sequence of numbers.

Table 2.1: Common built-in functions for numerical calculations.

```
# abs() can be used with real or complex numbers.
>>> print(abs(-7), abs(3 + 4j))
7 5.0

# min() and max() can be used on a list, string, or several arguments.
# String characters are ordered lexicographically.
>>> print(min([4, 2, 6]), min("aXbYcZ"), min('1', 'a', 'A'))
2 X 1
>>> print(max([4, 2, 6]), max("aXbYcZ"), max('1', 'a', 'A'))
6 c a

# len() can be used on a string, list, set, dict, tuple, or other iterable.
>>> print(len([2, 7, 1]), len("abcdef"), len({1, 'a', 'a'}))
3 6 2

# sum() can be used on iterables containing numbers, but not strings.
>>> my_list = [1, 2, 3]
>>> my_tuple = (4, 5, 6)
>>> my_set = {7, 8, 9}
>>> sum(my_list) + sum(my_tuple) + sum(my_set)
45
>>> sum([min(my_list), max(my_tuple), len(my_set)])
10

# round() is particularly useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

See <https://docs.python.org/3/library/functions.html> for more detailed documentation on all of Python's built-in functions.

Problem 1. Write a function that accepts a list L and returns the minimum, maximum, and average of the entries of L in that order as multiple values (separated by a comma). Can you implement this function in a single line?

Namespaces

Whenever a Python object—a number, data structure, function, or other entity—is created, it is stored somewhere in computer memory. A name (or variable) is a reference to a Python object, and a namespace is a dictionary that maps names to Python objects.

```
# The number 4 is the object, 'number_of_students' is the name.
>>> number_of_students = 4

# The list is the object, and 'beatles' is the name.
>>> beatles = ["John", "Paul", "George", "Ringo"]

# Python statements defining a function also form an object.
# The name for this function is 'add_numbers'.
>>> def add_numbers(a, b):
...     return a + b
...
```

A single equals sign assigns a name to an object. If a name is assigned to another name, that new name refers to the same object as the original name.

```
>>> beatles = ["John", "Paul", "George", "Ringo"]
>>> band_members = beatles           # Assign a new name to the list.
>>> print(band_members)
['John', 'Paul', 'George', 'Ringo']
```

To see all of the names in the current namespace, use the built-in function `dir()`. To delete a name from the namespace, use the `del` keyword (**with caution!**).

```
# Add 'stem' to the namespace.
>>> stem = ["Science", "Technology", "Engineering", "Mathematics"]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'stem']

# Remove 'stem' from the namespace.
>>> del stem
>>> "stem" in dir()
False
>>> print(stem)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stem' is not defined
```

Note

Many programming languages distinguish between variables and pointers. A pointer refers to a variable by storing the address in memory where the corresponding object is stored. Python names are essentially pointers, and traditional pointer operations and cleanup are done automatically. For example, Python automatically deletes objects in memory that have no names assigned to them (no pointers referring to them). This feature is called garbage collection.

Mutability

Every Python object type falls into one of two categories: a mutable object, which may be altered at any time, or an immutable object, which cannot be altered once created. Attempting to change an immutable object creates a new object in memory. If two names refer to the same mutable object, any changes to the object are reflected in both names since they still both refer to that same object. On the other hand, if two names refer to the same immutable object and one of the values is “changed,” then one name will refer to the original object, and the other will refer to a new object in memory.

Achtung!

Failing to correctly copy mutable objects can cause subtle problems. For example, consider a dictionary that maps items to their base prices. To make a similar dictionary that accounts for a small sales tax, we might try to make a copy by assigning a new name to the first dictionary.

```
>>> holy = {"moly": 1.99, "hand_grenade": 3, "grail": 1975.41}
>>> tax_prices = holy                      # Try to make a copy for processing.
>>> for item, price in tax_prices.items():
...     # Add a 7 percent tax, rounded to the nearest cent.
...     tax_prices[item] = round(1.07 * price, 2)
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}

# However, dictionaries are mutable, so 'holy' and 'tax_prices' actually
# refer to the same object. The original base prices have been lost.
>>> print(holy)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}
```

To avoid this problem, explicitly create a copy of the object by casting it as a new structure. Changes made to the copy will not change the original object, since they are distinct objects in memory. To fix the above code, replace the second line with the following:

```
>>> tax_prices = dict(holy)
```

Then, after running the same procedure, the two dictionaries will be different.

Problem 2. Determine which Python object types are mutable and which are immutable by repeating the following experiment for an `int`, `str`, `list`, `tuple`, and `set`.

1. Create an object of the given type and assign a name to it.
2. Assign a new name to the first name.
3. Alter the object via only one of the names (for tuples, use `my_tuple += (1,)`).
4. Check to see if the two names are equal. If they are, then changing one name also changes the other. Thus, both names refer to the same object and the object type is mutable. Otherwise, the names refer to different objects—meaning a new object was created in step 2—and therefore the object type is immutable.

For example, the following experiment shows that `dict` is a mutable type.

```
>>> dict_1 = {1: 'x', 2: 'b'}          # Create a dictionary.
>>> dict_2 = dict_1                  # Assign it a new name.
>>> dict_2[1] = 'a'                 # Change the 'new' dictionary.
>>> dict_1 == dict_2              # Compare the two names.
True                           # Both names changed!
```

Print a statement of your conclusions that clearly indicates which object types are mutable and which are immutable.

Achtung!

Mutable objects cannot be put into Python sets or used as keys in Python dictionaries. However, the values of a dictionary may be mutable or immutable.

```
>>> a_dict = {"key": "value"}           # Dictionaries are mutable.
>>> broken = {1, 2, 3, a_dict, a_dict}  # Try putting a dict in a set.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> okay = {1: 2, "3": a_dict}        # Try using a dict as a value.
```

Modules

A module is a Python file containing code that is meant to be used in some other setting, and not necessarily run directly.¹ The `import` statement loads code from a specified Python file. Importing a module containing some functions, classes, or other objects makes those functions, classes, or objects available for use by adding their names to the current namespace.

All import statements should occur at the top of the file, below the header but before any other code. There are several ways to use `import`:

1. `import <module>` makes the specified module available under the alias of its own name.

```
>>> import math                      # The name 'math' now gives
>>> math.sqrt(2)                    # access to the math module.
1.4142135623730951
```

2. `import <module> as <name>` creates an alias for an imported module. The alias is added to the current namespace, but the module name itself is not.

```
>>> import numpy as np              # The name 'np' gives access to the numpy
>>> np.sqrt(2)                     # module, but the name 'numpy' does not.
1.4142135623730951
>>> numpy.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' is not defined
```

3. `from <module> import <object>` loads the specified object into the namespace without loading anything else in the module or the module name itself. This is used most often to access specific functions from a module. The `as` statement can also be tacked on to create an alias.

```
>>> from random import randint    # The name 'randint' gives access to the
>>> r = randint(0, 10000)          # randint() function, but the rest of
>>> random.seed(r)               # the random module is unavailable.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

In each case, the final word of the import statement is the name that is added to the namespace.

Running and Importing

Consider the following simple Python module, saved as `example1.py`.

```
# example1.py

data = list(range(4))
```

¹Python files that are primarily meant to be executed, not imported, are often called scripts.

```

def display():
    print("Data:", data)

if __name__ == "__main__":
    display()
    print("This file was executed from the command line or an interpreter.")
else:
    print("This file was imported.")

```

Executing the file from the command line executes the file line by line, including the code under the `if __name__ == "__main__"` clause.

```

$ python example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.

```

Executing the file with IPython's special `%run` command executes each line of the file and also adds the module's names to the current namespace. **This is the quickest way to test individual functions via IPython.**

```

In [1]: %run example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [0, 1, 2, 3]

```

Importing the file also executes each line,² but only adds the indicated alias to the namespace. Also, code under the `if __name__ == "__main__"` clause is **not** executed when a file is imported.

```

In [1]: import example1 as ex
This file was imported.

# The module's names are not directly available...
In [2]: display()
-----
NameError                                 Traceback (most recent call last)
<ipython-input-2-795648993119> in <module>()
----> 1 display()

NameError: name 'display' is not defined

# ...unless accessed via the module's alias.
In [3]: ex.display()
Data: [0, 1, 2, 3]

```

²Try importing the `this` or `antigravity` modules. Importing these modules actually executes some code.

Problem 3. Create a module called `calculator.py`. Write a function `sum()` that returns the sum of two arguments and a function `product()` that returns the product of two arguments. Also use `import` to add the `sqrt()` function from the `math` module to the namespace. When this file is either run or imported, nothing should be executed.

In your solutions file, import your new custom module. Write a function that accepts two numbers representing the lengths of the sides of a right triangle. Using only the functions from `calculator.py`, calculate and return the length of the hypotenuse of the triangle.

Achtung!

If a module has been imported in IPython and the source code then changes, using `import` again does **not** refresh the name in the IPython namespace. Use `run` instead to correctly refresh the namespace. Consider this example where we test the function `sum_of_squares()`, saved in the file `example2.py`.

```
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x)])
```

In IPython, run the file and test `sum_of_squares()`.

```
# Run the file, adding the function sum_of_squares() to the namespace.
In [1]: %run example2

In [2]: sum_of_squares(3)
Out[2]: 5                                # Should be 14!
```

Since $1^2 + 2^2 + 3^2 = 14$, not 5, something has gone wrong. Modify the source file to correct the mistake, then run the file again in IPython.

```
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x+1)])    # Include the final term.

# Run the file again to refresh the namespace.
In [3]: %run example2
```

```
# Now sum_of_squares() is updated to the new, corrected version.
In [4]: sum_of_squares(3)
Out[4]: 14 # It works!
```

Remember that running or importing a file executes any freestanding code snippets, but any code under an `if __name__ == "__main__"` clause will **only** be executed when the file is run (not when it is imported).

The Python Standard Library

All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the Python standard library. Some commonly standard library modules are listed below, and the complete list is at <https://docs.python.org/3/library/>.

Module	Description
<code>cmath</code>	Mathematical functions for complex numbers.
<code>itertools</code>	Tools for iterating through sequences in useful ways.
<code>math</code>	Standard mathematical functions and constants.
<code>random</code>	Random variable generators.
<code>string</code>	Common string literals.
<code>sys</code>	Tools for interacting with the interpreter.
<code>time</code>	Time value generation and manipulation.

Use IPython's object introspection to quickly learn about how to use the various modules and functions in the standard library. Use `? or help()` for information on the module or one of its names. To see the entire module's namespace, use the `tab` key.

```
In [1]: import math

In [2]: math?
Type:    module
String form: <module 'math' (built-in)>
Docstring:
This module provides access to the mathematical functions
defined by the C standard.

# Type the module name, a period, then press tab to see the module's namespace.
In [3]: math. # Press 'tab'.
      acos()   cos()     factorial()  isclose()   log2()     tan()
      acosh()  cosh()     floor()      isfinite()  modf()     tanh()
      asin()   degrees()  fmod()       isinf()     nan        tau
      asinh()  e          frexp()     isnan()     pi         trunc()
      atan()   erf()      fsum()      ldexp()    pow()
      atan2()  erfc()     gamma()     lgamma()   radians()
      atanh()  exp()      gcd()       log()      sin()
      ceil()   expm1()    hypot()    log10()   sinh()
```

```

copySIGN()  fabs()      inf       log1p()    sqrt()

In [3]: math.sqrt?
Signature: math.sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method

```

The Itertools Module

The `itertools` module makes it easy to iterate over one or more collections in specialized ways.

Function	Description
<code>chain()</code>	Iterate over several iterables in sequence.
<code>cycle()</code>	Iterate over an iterable repeatedly.
<code>combinations()</code>	Return successive combinations of elements in an iterable.
<code>permutations()</code>	Return successive permutations of elements in an iterable.
<code>product()</code>	Iterate over the Cartesian product of several iterables.

```

>>> from itertools import chain, cycle           # Import multiple names.

>>> list(chain("abc", ['d', 'e'], ('f', 'g')))   # Join several
['a', 'b', 'c', 'd', 'e', 'f', 'g']             # sequences together.

>>> for i, number in enumerate(cycle(range(4))):  # Iterate over a single
...     if i > 10:                                # sequence over and over.
...         break
...     print(number, end=' ')
...
0 1 2 3 0 1 2 3 0 1 2

```

A k -combination is a set of k elements from a collection where the ordering is unimportant. Thus the combination (a, b) and (b, a) are equivalent because they contain the same elements. On the other hand, a k -permutation is a sequence of k elements from a collection where the ordering matters. Even though (a, b) and (b, a) contain the same elements, they are counted as different permutations.

```

>>> from itertools import combinations, permutations

# Get all combinations of length 2 from the iterable "ABC".
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]

# Get all permutations of length 2 from "ABC". Note that order matters here.
>>> list(permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

```

Problem 4. The power set of a set A , denoted $\mathcal{P}(A)$ or 2^A , is the set of all subsets of A , including the empty set \emptyset and A itself. For example, the power set of the set $A = \{a, b, c\}$ is $2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

Write a function that accepts an iterable A . Use an `itertools` function to compute and return the power set of A as a list of sets (why couldn't it be a set of sets in Python?). The empty set should be returned as `set()`.

The Random Module

Many real-life events can be simulated by taking random samples from a probability distribution. For example, a coin flip can be simulated by randomly choosing between the integers 1 (for heads) and 0 (for tails). The `random` module includes functions for sampling from probability distributions and generating random data.

Function	Description
<code>choice()</code>	Choose a random element from a non-empty sequence, such as a list.
<code>randint()</code>	Choose a random integer integer over a closed interval.
<code>random()</code>	Pick a float from the interval [0, 1).
<code>sample()</code>	Choose several unique random elements from a non-empty sequence.
<code>seed()</code>	Seed the random number generator.
<code>shuffle()</code>	Randomize the ordering of the elements in a list.

Some of the most common `random` utilities involve picking random elements from iterables.

```
>>> import random

>>> numbers = list(range(1,11))      # Get the integers from 1 to 10.
>>> print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> random.shuffle(numbers)          # Mix up the ordering of the list.
>>> print(numbers)                  # Note that shuffle() returns nothing.
[5, 9, 1, 3, 8, 4, 10, 6, 2, 7]

>>> random.choice(numbers)          # Pick a single element from the list.
5

>>> random.sample(numbers, 4)        # Pick 4 unique elements from the list.
[5, 8, 3, 2]

>>> random.randint(1,10)            # Pick a random number between 1 and 10.
10
```

The Time Module

The `time` module in the standard library include functions for dealing with time. In particular, the `time()` function measures the number of seconds from a fixed starting point, called “the Epoch” (January 1, 1970 for Unix machines).

```
>>> import time
>>> time.time()
1495243696.645818
```

The `time()` function is useful for measuring how long it takes for code to run: record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed.

```
>>> def time_for_loop(iters):
...     """Time how long it takes to iterate 'iters' times."""
...     start = time.time()          # Clock the starting time.
...     for _ in range(int(iters)):
...         pass
...     end = time.time()           # Clock the ending time.
...     return end - start         # Report the difference.
...
>>> time_for_loop(1e5)          # 1e5 = 100000.
0.005570173263549805
>>> time_for_loop(1e7)          # 1e7 = 10000000.
0.26819777488708496
```

The Sys Module

The `sys` (system) module includes methods for interacting with the Python interpreter. For our purposes, we care about the instance that you call a .py file in the command line or in your ipython terminal using either ‘python’ followed by the name of a .py file or ‘%run’ followed by the name of a .py file. One command we will use is `sys.argv`; which returns a list containing the .py file name and all arguments that were passed to the interpreter. This way we can interact with these initial arguments and execute certain parts of our code if the arguments satisfy certain conditions. This will be implemented in problem 5. Note, command line arguments are passed in after the name of the .py file and are separated by spaces like so:

```
# the 'python' command followed by a .py file followed by any arguments
$ python yourProgram.py argument1 argument2 argument3

# the equivalent form in an ipython terminal
In[1]: %run yourProgram.py argument1 argument2 argument3
```

Now we can execute code based on the command line arguments by inserting the use of the `sys` module in our .py file:

```
# example3.py
"""If there are two command line arguments after the .py file name, print a descriptive statement."""
import sys

if len(sys.argv) == 3: # if there's two arguments and the file name
    print("the first command line argument is " + sys.argv[1])
    print("the second command line argument is " + sys.argv[2])
else:
    print("um... I need two after the .py file. This is not two:")
    print(sys.argv)
```

Now provide command line arguments for the program to process.

```
# No extra command line arguments.  
$ python example3.py  
Output:  
um... I need two after the .py file. This is not two:  
['example3.py']  
  
# With two arguments the if statement executes.  
$ python example3.py crunchy juicy  
Output:  
the first command line argument is crunchy  
the second command line argument is juicy
```

Note that the first command line argument is always the filename, so that is always the first element of the `sys.argv` list. This is why we have the if statement execute if the length of the list is 3 even though we want it to execute when there are 2 command line arguments. Also, `sys.argv` is always a list of strings. If a number is provided on the command line, it is converted to a string when it is stored in `sys.argv`. In IPython, command line arguments are specified after the `%run` command.

```
In [1]: %run example3.py 42 too many
Output:
um... I need two after the .py file. This is not two:
['example3.py', '42', 'too', 'many']
```

Another way to get input from the program user is to prompt the user for text. The built-in function `input()` pauses the program and waits for the user to type something. Like command line arguments, the user's input is parsed as a string.

```
>>> x = input("Enter a value for x: ")
Enter a value for x: 20 # Type '20' and press 'enter.'

>>> x
'20' # Note that x contains a string.

>> y = int(input("Enter an integer for y: "))
Enter an integer for y: 16 # Type '16' and press 'enter.'

>>> y
16 # Note that y contains an integer.
```

Problem 5. Shut the box is a popular British pub game that is used to help children learn arithmetic. The player starts with the numbers 1 through 9, and the goal of the game is to eliminate as many of these numbers as possible. At each turn the player rolls two dice, then chooses a set of integers from the remaining numbers that sum up to the sum of the dice roll. These numbers are removed, and the dice are then rolled again. The game ends when none of the remaining integers can be combined to the sum of the dice roll, and the player's final score is the sum of the numbers that could not be eliminated. For a demonstration, see <https://www.youtube.com/watch?v=mwURQC7mjDI>.

Modify your solutions file so that when the file is run with the correct command line arguments (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains two functions that will be useful in your implementation of the game. You do not need to understand exactly how the functions work, but you do need to be able to import and use them correctly. Their functionality is outlined at the beginning of each function declaration. Your game should match the following specifications:

- Require three total command line arguments: the file name (included by default), the player's name, and a time limit in seconds. If there are not exactly three command line arguments, do not start the game.
- Track the player's remaining numbers, starting with 1 through 9.
- Use the `random` module to simulate rolling two six-sided dice. However, if the sum of the player's remaining numbers is 6 or less, roll only one die.
- The player wins if they have no numbers left, and they lose if they are out of time or if they cannot choose numbers to match the dice roll.
- If the game is not over, print the player's remaining numbers, the sum of the dice roll, and the number of seconds remaining. Prompt the user for numbers to eliminate. The input should be one or more of the remaining integers, separated by spaces. If the user's input is invalid, prompt them for input again before rolling the dice again.
(Hint: use `round()` to format the number of seconds remaining nicely.)
- When the game is over, display the player's name, their score, and the total number of seconds since the beginning of the game. Congratulate or mock the player appropriately.

(Hint: **Before you start coding**, write an outline for the entire program, adding one feature at a time. Only start implementing the game after you are completely finished designing it.)

Your game should look similar to the following examples. The characters in red are typed inputs from the user.

```
$ python standard_library.py LuckyDuke 60

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 12
Seconds left: 60.0
Numbers to eliminate: 3 9

Numbers left: [1, 2, 4, 5, 6, 7, 8]
```

```
Roll: 9
Seconds left: 53.51
Numbers to eliminate: 8 1

Numbers left: [2, 4, 5, 6, 7]
Roll: 7
Seconds left: 51.39
Numbers to eliminate: 7

Numbers left: [2, 4, 5, 6]
Roll: 2
Seconds left: 48.24
Numbers to eliminate: 2

Numbers left: [4, 5, 6]
Roll: 11
Seconds left: 45.16
Numbers to eliminate: 5 6

Numbers left: [4]
Roll: 4
Seconds left: 42.76
Numbers to eliminate: 4

Score for player LuckyDuke: 0 points
Time played: 15.82 seconds
Congratulations!! You shut the box!
```

The next two examples show different ways that a player could lose (which they usually do), as well as examples of invalid user input. Use the `box` module's `parse_input()` to detect invalid input.

```
$ python standard_library.py ShakySteve 10

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 7
Seconds left: 10.0
Numbers to eliminate: Seven          # Must enter a number.
Invalid input

Seconds left: 7.64
Numbers to eliminate: 1, 2, 4         # Do not use commas.
Invalid input

Seconds left: 4.55
Numbers to eliminate: 1 2 3          # Numbers don't sum to the roll.
Invalid input
```

```
Seconds left: 2.4
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 8
Seconds left: 0.31
Numbers to eliminate: 8
Game over!                                # Time is up!

Score for player ShakySteve: 30 points
Time played: 11.77 seconds
Better luck next time >:)
```

```
$ python standard_library.py SnakeEyesTom 10000

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Seconds left: 10000.0
Numbers to eliminate: 2

Numbers left: [1, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Game over!                                # Numbers cannot match roll.

Score for player SnakeEyesTom: 43 points
Time played: 1.53 seconds
Better luck next time >:)
```

Additional Material

More Built-in Functions

The following built-in functions are worth knowing, especially for working with iterables and writing very readable conditional statements.

Function	Description
<code>all()</code>	Return <code>True</code> if <code>bool(entry)</code> evaluates to <code>True</code> for every entry in the input iterable.
<code>any()</code>	Return <code>True</code> if <code>bool(entry)</code> evaluates to <code>True</code> for any entry in the input iterable.
<code>bool()</code>	Evaluate a single input object as <code>True</code> or <code>False</code> .
<code>eval()</code>	Execute a string as Python code and return the output.
<code>map()</code>	Apply a function to every item of the input iterable and return an iterable of the results.

```
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive.
>>> numbers = [randint(1,10) for _ in range(5)]

# If all of the numbers are less than 8, print the list.
>>> if all([num < 8 for num in numbers]):
...     print(numbers)
...
[1, 5, 6, 3, 3]

# If none of the numbers are divisible by 3, print the list.
>>> if not any([num % 3 == 0 for num in numbers]):
...     print(numbers)
...
```

Two-Player Shut the Box

Consider modifying your shut the box program so that it pits two players against each other (one player tries to shut the box while the other tries to keep it open). The first player plays a regular round as described in Problem 5. Suppose he or she eliminates every number but 2, 3, and 6. The second player then begins a round with the numbers 1, 4, 5, 7, 8, and 9, the numbers that the first player had eliminated. If the second player loses, the first player gets another round to try to shut the box with the numbers that the second player had eliminated. Play continues until one of the players eliminates their entire list. In addition, each player should have their own time limit that only ticks down during their turn. If time runs out on your turn, you lose no matter what.

Python Packages

Large programming projects often have code spread throughout several folders and files. In order to get related files in different folders to communicate properly, the associated directories must be organized into a Python packages. This is a common procedure when creating smart phone applications and other programs that have graphical user interfaces (GUIs).

A package is simply a folder that contains a file called `__init__.py`. This file is always executed first whenever the package is used. A package must also have a file called `__main__.py` in order to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.py`.

Use the regular syntax to import a module or subpackage that is in the current package, and use `from <subpackage.module> import <object>` to load a module within a subpackage. Once a name has been loaded into a package's `__init__.py`, other files in the same package can load the same name with `from . import <object>`. To access code in the directory one level above the current directory, use the syntax `from .. import <object>`. This tells the interpreter to go up one level and import the object from there. This is called an explicit relative import and cannot be done in files that are executed directly (like `__main__.py`).

Finally, to execute a package, run Python from the shell with the flag `-m` (for “module-name”) and exclude the extension `.py`.

```
$ python -m package_name
```

See <https://docs.python.org/3/tutorial/modules.html#packages> for examples and more details.

3

Object-oriented Programming

Lab Objective: Python is a class-based language. A class is a blueprint for an object that binds together specified variables and routines. Creating and using custom classes is often a good way to write clean, efficient, well-designed programs. In this lab we learn how to define and use Python classes. In subsequent labs we will often create customized classes for use in algorithms.

Classes

A Python class is a code block that defines a custom object and determines its behavior. The `class` key word defines and names a new class. Other statements follow, indented below the class name, to determine the behavior of objects instantiated by the class.

A class needs a method called a constructor that is called whenever the class instantiates a new object. The constructor specifies the initial state of the object. In Python, a class's constructor is always named `__init__()`. For example, the following code defines a class for storing information about backpacks.

```
class Backpack:  
    """A Backpack object class. Has a name and a list of contents.  
  
    Attributes:  
        name (str): the name of the backpack's owner.  
        contents (list): the contents of the backpack.  
    """  
    def __init__(self, name):           # This function is the constructor.  
        """Set the name and initialize an empty list of contents.  
  
    Parameters:  
        name (str): the name of the backpack's owner.  
    """  
        self.name = name                 # Initialize some attributes.  
        self.contents = []
```

An attribute is a variable stored within an object. The `Backpack` class has two attributes: `name` and `contents`. In the body of the class definition, attributes are assigned and accessed via the name `self`. This name refers to the object internally once it has been created.

Instantiation

The `class` code block above only defines a blueprint for backpack objects. To create an actual backpack object, call the class name like a function. This triggers the constructor and returns a new instance of the class, an object whose type is the class.

```
# Import the Backpack class and instantiate an object called 'my_backpack'.
>>> from object_oriented import Backpack
>>> my_backpack = Backpack("Fred")
>>> type(my_backpack)
<class 'object_oriented.Backpack'>

# Access the object's attributes with a period and the attribute name.
>>> print(my_backpack.name, my_backpack.contents)
Fred []

# The object's attributes can be modified after instantiation.
>>> my_backpack.name = "George"
>>> print(my_backpack.name, my_backpack.contents)
George []
```

Note

Every object in Python has some built-in attributes. For example, modules have a `__name__` attribute that identifies the scope in which it is being executed. If the module is being run directly, and not imported, then `__name__` is set to "`__main__`". Therefore, any commands under an `if __name__ == "__main__":` clause are ignored when the module is imported.

Methods

In addition to storing variables as attributes, classes can have functions attached to them. A function that belongs to a specific class is called a method.

```
class Backpack:
    #
    def put(self, item):
        """Add an item to the backpack's list of contents."""
        self.contents.append(item) # Use 'self.contents', not just 'contents'.

    def take(self, item):
        """Remove an item from the backpack's list of contents."""
        self.contents.remove(item)
```

The first argument of each method must be `self`, to give the method access to the attributes and other methods of the class. The `self` argument is only included in the declaration of the class methods, **not** when calling the methods on an instantiation of the class.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")           # my_backpack is passed implicitly to
>>> my_backpack.put("pencils")          # Backpack.put() as the first argument.
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.      # This is equivalent to
>>> my_backpack.take("pencils")        # Backpack.take(my_backpack, "pencils")
>>> my_backpack.contents
['notebook']
```

Problem 1. Expand the `Backpack` class to match the following specifications.

1. Modify the constructor so that it accepts three total arguments: `name`, `color`, and `max_size` (in that order). Make `max_size` a keyword argument that defaults to 5. Store each input as an attribute.
2. Modify the `put()` method to check that the backpack does not go over capacity. If there are already `max_size` items or more, print “No Room!” and do not add the item to the contents list.
3. Write a new method called `dump()` that resets the contents of the backpack to an empty list. This method should not receive any arguments (except `self`).
4. Documentation is especially important in classes so that the user knows what an object’s attributes represent and how to use methods appropriately. Update (or write) the docstrings for the `__init__()`, `put()`, and `dump()` methods, as well as the actual class docstring (under `class` but before `__init__()`) to reflect the changes from parts 1-3 of this problem.

To ensure that your class works properly, write a test function outside of the `Backpack` class that instantiates and analyzes a `Backpack` object.

```
def test_backpack():
    testpack = Backpack("Barry", "black")      # Instantiate the object.
    if testpack.name != "Barry":                # Test an attribute.
        print("Backpack.name assigned incorrectly")
    for item in ["pencil", "pen", "paper", "computer"]:
        testpack.put(item)                      # Test a method.
    print("Contents:", testpack.contents)
    # ...
```

Inheritance

To create a new class that is similar to one that already exists, it is often better to inherit the methods and attributes from an existing class rather than create a new class from scratch. This creates a class hierarchy: a class that inherits from another class is called a subclass, and the class that a subclass inherits from is called a superclass. To define a subclass, add the name of the superclass as an argument at the end of the `class` declaration.

For example, since a knapsack is a kind of backpack (but not all backpacks are knapsacks), we create a special `Knapsack` subclass that inherits the structure and behaviors of the `Backpack` class and adds some extra functionality.

```
# Inherit from the Backpack class in the class definition.
class Knapsack(Backpack):
    """A Knapsack object class. Inherits from the Backpack class.
    A knapsack is smaller than a backpack and can be tied closed.

    Attributes:
        name (str): the name of the knapsack's owner.
        color (str): the color of the knapsack.
        max_size (int): the maximum number of items that can fit inside.
        contents (list): the contents of the backpack.
        closed (bool): whether or not the knapsack is tied shut.

    """
    def __init__(self, name, color, max_size=3):
        """Use the Backpack constructor to initialize the name, color,
        and max_size attributes. A knapsack only holds 3 item by default.

        Parameters:
            name (str): the name of the knapsack's owner.
            color (str): the color of the knapsack.
            max_size (int): the maximum number of items that can fit inside.

        """
        Backpack.__init__(self, name, color, max_size)
        self.closed = True
```

A subclass may have new attributes and methods that are unavailable to the superclass, such as the `closed` attribute in the `Knapsack` class. If methods from the superclass need to be changed for the subclass, they can be overridden by defining them again in the subclass. New methods can be included normally.

```
class Knapsack(Backpack):
    # ...
    def put(self, item):          # Override the put() method.
        """If the knapsack is untied, use the Backpack.put() method."""
        if self.closed:
            print("I'm closed!")
        else:                      # Use Backpack's original put().
            Backpack.put(self, item)
```

```

def take(self, item):           # Override the take() method.
    """If the knapsack is untied, use the Backpack.take() method."""
    if self.closed:
        print("I'm closed!")
    else:
        Backpack.take(self, item)

def weight(self):               # Define a new method just for knapsacks.
    """Calculate the weight of the knapsack by counting the length of the
    string representations of each item in the contents list.
    """
    return sum(len(str(item)) for item in self.contents)

```

Since Knapsack inherits from Backpack, a knapsack object **is** a backpack object. All methods defined in the Backpack class are available as instances of the Knapsack class. For example, the `dump()` method is available even though it is not defined explicitly in the Knapsack class.

The built-in function `issubclass()` shows whether or not one class is derived from another. Similarly, `isinstance()` indicates whether or not an object belongs to a specified class hierarchy. Finally, `hasattr()` shows whether or not a class or object **has** a specified attribute or method.

```

>>> from object_oriented import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")

# A Knapsack is a Backpack, but a Backpack is not a Knapsack.
>>> print(issubclass(Knapsack, Backpack), issubclass(Backpack, Knapsack))
True False
>>> isinstance(my_knapsack, Knapsack) and isinstance(my_knapsack, Backpack)
True

# The put() and take() method now require the knapsack to be open.
>>> my_knapsack.put('compass')
I'm closed!

# Open the knapsack and put in some items.
>>> my_knapsack.closed = False
>>> my_knapsack.put("compass")
>>> my_knapsack.put("pocket knife")
>>> my_knapsack.contents
['compass', 'pocket knife']

# The Knapsack class has a weight() method, but the Backpack class does not.
>>> print(hasattr(my_knapsack, 'weight'), hasattr(my_backpack, 'weight'))
True False

# The dump method is inherited from the Backpack class.
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]

```

Problem 2. Write a `Jetpack` class that inherits from the `Backpack` class.

1. Override the constructor so that in addition to a name, color, and maximum size, it also accepts an amount of fuel. Change the default value of `max_size` to 2, and set the default value of fuel to 10. Store the fuel as an attribute.
2. Add a `fly()` method that accepts an amount of fuel to be burned and decrements the fuel attribute by that amount. If the user tries to burn more fuel than remains, print “Not enough fuel!” and do not decrement the fuel.
3. Override the `dump()` method so that both the contents and the fuel tank are emptied.
4. Write clear, detailed docstrings for the class and each of its methods.

Note

All classes are subclasses of the built-in `object` class, even if no parent class is specified in the class definition. In fact, the syntax “`class ClassName(object):`” is not uncommon (or incorrect) for the class declaration, and is equivalent to the simpler “`class ClassName:`”.

Magic Methods

A magic method is a special method used to make an object behave like a built-in data type. Magic methods begin and end with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, which can be revealed through IPython.

```
In [1]: %run object_oriented.py

In [2]: b = Backpack("Oscar", "green")

In [3]: b.          # Press 'tab' to see standard methods and attributes.
        color      max_size take()
        contents   name
        dump()     put()

In [3]: b.__       # Press 'tab' to see magic methods and hidden attributes.
        __add__()           __getattribute__    __new__()
        __class__            __gt__              __reduce__()
        __delattr__          __hash__            __reduce_ex__()
        __dict__             __init__            __repr__
        __dir__()            __init_subclass__  __setattr__
        __doc__              __le__              __sizeof__
        __eq__               __lt__              __str__
        __format__(...)      __module__         __subclasshook__
        __ge__               __ne__              __weakref__
```

Note

Many programming languages distinguish between public and private variables. In Python, all attributes are public, period. However, attributes that start with an underscore are hidden from the user, which is why magic methods do not show up at first in the preceding code box.

The more common magic methods define how an object behaves with respect to addition and other binary operations. For example, how should addition be defined for backpacks? A simple option is to add the number of contents. Then if backpack A has 3 items and backpack B has 5 items, A + B should return 8. To incorporate this idea, we implement the `__add__()` magic method.

```
class Backpack:
    # ...
    def __add__(self, other):
        """Add the number of contents of each Backpack."""
        return len(self.contents) + len(other.contents)
```

Using the `+` binary operator on two `Backpack` objects calls the class's `__add__()` method. The object on the left side of the `+` is passed in to `__add__()` as `self` and the object on the right side of the `+` is passed in as `other`.

```
>>> pack1 = Backpack("Rose", "red")
>>> pack2 = Backpack("Carly", "cyan")

# Put some items in the backpacks.
>>> pack1.put("textbook")
>>> pack2.put("water bottle")
>>> pack2.put("snacks")

# Add the backpacks together.
>>> pack1 + pack2           # Equivalent to pack1.__add__(pack2).
3
```

Comparisons

Magic methods also facilitate object comparisons. For example, the `__lt__()` method corresponds to the `<` operator. Suppose one backpack is considered “less” than another if it has fewer items in its list of contents.

```
class Backpack(object):
    # ...
    def __lt__(self, other):
        """If 'self' has fewer contents than 'other', return True.
        Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

Using the `<` binary operator on two `Backpack` objects calls `__lt__()`. As with addition, the object on the left side of the `<` operator is passed to `__lt__()` as `self`, and the object on the right is passed in as `other`.

```
>>> pack1, pack2 = Backpack("Maggy", "magenta"), Backpack("Yolanda", "yellow")
>>> pack1 < pack2
False
# Equivalent to pack1.__lt__(pack2).

>>> pack2.put('pencils')
>>> pack1 < pack2
True
```

Comparison methods should return either `True` or `False`, while methods like `__add__()` might return a numerical value or another kind of object.

Method	Arithmetic Operator	Method	Comparison Operator
<code>__add__()</code>	<code>+</code>	<code>__lt__()</code>	<code><</code>
<code>__sub__()</code>	<code>-</code>	<code>__le__()</code>	<code><=</code>
<code>__mul__()</code>	<code>*</code>	<code>__gt__()</code>	<code>></code>
<code>__pow__()</code>	<code>**</code>	<code>__ge__()</code>	<code>>=</code>
<code>__truediv__()</code>	<code>/</code>	<code>__eq__()</code>	<code>==</code>
<code>__floordiv__()</code>	<code>//</code>	<code>__ne__()</code>	<code>!=</code>

Table 3.1: Common magic methods for arithmetic and comparisons. What each of these operations do is up to the programmer and should be carefully documented. For more methods and details, see <https://docs.python.org/3/reference/datamodel.html#special-method-names>.

Problem 3. Endow the `Backpack` class with two additional magic methods:

1. The `__eq__()` magic method is used to determine if two objects are equal, and is invoked by the `==` operator. Implement the `__eq__()` magic method for the `Backpack` class so that two `Backpack` objects are equal if and only if they have the same name, color, and number of contents.
2. The `__str__()` magic method returns the string representation of an object. This method is invoked by `str()` and used by `print()`. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output (that is, construct and return the following string).

```
Owner:      <name>
Color:      <color>
Size:       <number of items in contents>
Max Size:   <max_size>
Contents:   [<item1>, <item2>, ...]
```

(Hint: Use the tab and newline characters '`\t`' and '`\n`' to align output nicely.)

Achtung!

Magic methods for comparison are **not** automatically related. For example, even though the `Backpack` class implements the magic methods for `<` and `==`, two `Backpack` objects cannot respond to the `<=` operator unless `__le__()` is explicitly defined. The exception to this rule is the `!=` operator: as long as `__eq__()` is defined, `A!=B` is `False` if and only if `A==B` is `True`.

Problem 4.

Write a `ComplexNumber` class from scratch.

1. Complex numbers are denoted $a + bi$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. Write the constructor so it accepts two numbers. Store the first as `self.real` and the second as `self.imag`.
2. The complex conjugate of $a + bi$ is defined as $\overline{a + bi} = a - bi$. Write a `conjugate()` method that returns the object's complex conjugate as a new `ComplexNumber` object.
3. Add the following magic methods:
 - (a) Implement `__str__()` so that $a + bi$ is printed out as $(a+bj)$ for $b \geq 0$ and $(a-bj)$ for $b < 0$.
 - (b) The magnitude of $a + bi$ is $|a + bi| = \sqrt{a^2 + b^2}$. The `__abs__()` magic method determines the output of the built-in `abs()` function (absolute value). Implement `__abs__()` so that it returns the magnitude of the complex number.
 - (c) Two `ComplexNumber` objects are equal if and only if they have the same real and imaginary parts. Implement `__eq__()` so that it compares the `ComplexNumber` object with another `ComplexNumber` object and returns a bool indicating whether they are equal.
 - (d) Implement `__add__()`, `__sub__()`, `__mul__()`, and `__truediv__()` appropriately. Each of these should return a new `ComplexNumber` object.

Write a function to test your class by comparing it to Python's built-in `complex` type.

```
def test_ComplexNumber(a, b):
    py_cnum, my_cnum = complex(a, b), ComplexNumber(a, b)

    # Validate the constructor.
    if my_cnum.real != a or my_cnum.imag != b:
        print("__init__() set self.real and self.imag incorrectly")

    # Validate conjugate() by checking the new number's imag attribute.
    if py_cnum.conjugate().imag != my_cnum.conjugate().imag:
        print("conjugate() failed for", py_cnum)

    # Validate __str__().
    if str(py_cnum) != str(my_cnum):
        print("__str__() failed for", py_cnum)
```

```
# ...
```

Additional Material

Static Attributes

Attributes that are accessed through `self` are called instance attributes because they are bound to a particular instance of the class. In contrast, a static attribute is one that is shared between all instances of the class. To make an attribute static, declare it inside of the `class` block but outside of any of the class's methods, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it may be accessed or changed via the class name without even instantiating the class at all.

```
class Backpack:
    # ...
    brand = "Adidas"                      # Backpack.brand is a static attribute.
```

```
>>> pack1, pack2 = Backpack("Bill", "blue"), Backpack("William", "white")
>>> print(pack1.brand, pack2.brand, Backpack.brand)
Adidas Adidas Adidas

# Change the brand name for the class to change it for all class instances.
>>> Backpack.brand = "Nike"
>>> print(pack1.brand, pack2.brand, Backpack.brand)
Nike Nike Nike
```

Static Methods

Individual class methods can also be static. A static method cannot be dependent on the attributes of individual instances of the class, so there can be no references to `self` inside the body of the method and `self` is **not** listed as an argument in the function definition. Thus static methods only have access to static attributes and other static methods. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack:
    # ...
    @staticmethod
    def origin():                  # Do not use 'self' as a parameter.
        print("Manufactured by " + Backpack.brand + ", inc.")
```

```
# Static methods can be called without instantiating the class.
>>> Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed by individual class instances.
>>> pack = Backpack("Larry", "lime")
>>> pack.origin()
Manufactured by Nike, inc.
```

To practice these principles, consider adding a static attribute to the `Backpack` class to serve as a counter for a unique ID. In the constructor for the `Backpack` class, add an instance variable called `self.ID`. Set this ID based on the static ID variable, then increment the static ID so that the next `Backpack` object will have a different ID.

More Magic Methods

Consider how the following methods might be implemented for the `Backpack` class. These methods are particularly important for custom data structure classes.

Method	Operation	Trigger Function
<code>__bool__()</code>	Truth value	<code>bool()</code>
<code>__len__()</code>	Object length or size	<code>len()</code>
<code>__repr__()</code>	Object representation	<code>repr()</code>
<code>__getitem__()</code>	Indexing and slicing	<code>self[index]</code>
<code>__setitem__()</code>	Assignment via indexing	<code>self[index] = x</code>
<code>__iter__()</code>	Iteration over the object	<code>iter()</code>
<code>__reversed__()</code>	Reverse iteration over the object	<code>reversed()</code>
<code>__contains__()</code>	Membership testing	<code>in</code>

See <https://docs.python.org/3/reference/datamodel.html#special-method-names> for more details and documentation on all magic methods.

Hashing

A hash value is an integer that uniquely identifies an object. The built-in `hash()` function calculates an object's hash value by calling its `__hash__()` magic method.

In Python, the built-in `set` and `dict` structures use hash values to store and retrieve objects in memory quickly. If an object is unhashable, it cannot be put in a set or be used as a key in a dictionary. See <https://docs.python.org/3/glossary.html#term-hashable> for details.

If the `__hash__()` method is not defined, the default hash value is the object's memory address (accessible via the built-in function `id()`) divided by 16, rounded down to the nearest integer. However, two objects that compare as equal via the `__eq__()` magic method must have the same hash value. The following simple `__hash__()` method for the `Backpack` class conforms to this rule and returns an integer.

```
class Backpack:
    # ...
    def __hash__(self):
        return hash(self.name) ^ hash(self.color) ^ hash(len(self.contents))
```

The caret operator `^` is a bitwise XOR (exclusive or). The bitwise AND operator `&` and the bitwise OR operator `|` are also good choices to use.

See https://docs.python.org/3/reference/datamodel.html#object.__hash__ for more on hashing.

4

Exceptions and File Input/Ouput

Lab Objective: In Python, an exception is an error detected during execution. Exceptions are important for regulating program usage and for correctly reporting problems to the programmer and end user. An understanding of exceptions is essential to safely read data from and write data to external files. Being able to interact with external files is important for analyzing data and communicating results. In this lab we learn exception syntax and file interaction protocols.

Exceptions

An exception formally indicates an error and terminates the program early. Some of the more common exception types are listed below, along with the kinds of problems they typically indicate.

Exception	Indication
<code>AttributeError</code>	An attribute reference or assignment failed.
<code>ImportError</code>	An <code>import</code> statement failed.
<code>IndexError</code>	A sequence subscript was out of range.
<code>NameError</code>	A local or global name was not found.
<code>TypeError</code>	An operation or function was applied to an object of inappropriate type.
<code>ValueError</code>	An operation or function received an argument that had the right type but an inappropriate value.
<code>ZeroDivisionError</code>	The second argument of a division or modulo operation was zero.

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> [1, 2, 3].fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'
```

Raising Exceptions

Most exceptions are due to coding mistakes and typos. However, exceptions can also be used intentionally to indicate a problem to the user or programmer. To create an exception, use the keyword `raise`, followed by the name of the exception class. As soon as an exception is raised, the program stops running unless the exception is handled properly.

```
>>> if 7 is not 7.0:                      # Raise an exception with an error message.
...     raise Exception("ints and floats are different!")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: ints and floats are different!

>>> for x in range(10):
...     if x > 5:                      # Raise a specific kind of exception.
...         raise ValueError("'x' should not exceed 5.")
...     print(x, end=' ')
...
0 1 2 3 4 5
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
```

Problem 1. Consider the following arithmetic “magic” trick.

1. Choose a 3-digit number where the first and last digits differ by 2 or more (say, 123).
2. Reverse this number by reading it backwards (321).
3. Calculate the positive difference of these numbers ($321 - 123 = 198$).
4. Add the reverse of the result to itself ($198 + 891 = 1089$).

The result of the last step will always be 1089, regardless of the original number chosen in step 1 (can you explain why?).

The following function prompts the user for input at each step of the magic trick, but does not check that the user’s inputs are correct.

```
def arithmagic():
    step_1 = input("Enter a 3-digit number where the first and last "
                  "digits differ by 2 or more: ")
    step_2 = input("Enter the reverse of the first number, obtained "
                  "by reading it backwards: ")
    step_3 = input("Enter the positive difference of these numbers: ")
    step_4 = input("Enter the reverse of the previous result: ")
    print(str(step_3), "+", str(step_4), "= 1089 (ta-da!)")
```

Modify `arithmagic()` so that it verifies the user's input at each step. Raise a `ValueError` with an informative error message if any of the following occur:

- The first number (`step_1`) is not a 3-digit number.
- The first number's first and last digits differ by less than 2.
- The second number (`step_2`) is not the reverse of the first number.
- The third number (`step_3`) is not the positive difference of the first two numbers.
- The fourth number (`step_4`) is not the reverse of the third number.

(Hint: `input()` always returns a string, so each variable is a string initially. Use `int()` to cast the variables as integers when necessary. The built-in function `abs()` may also be useful.)

Handling Exceptions

To prevent an exception from halting the program, it must be handled by placing the problematic lines of code in a `try` block. An `except` block then follows with instructions for what to do in the event of an exception.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     print("Entering try block...")
...     raise Exception("for no reason")
...     print("No problem!")      # This line gets skipped.
... # The 'except' block is executed just after the exception is raised.
... except Exception as e:
...     print("There was a problem:", e)
...
Entering try block...
There was a problem: for no reason
>>> # The program then continues on.
```

In this example, the name `e` represents the exception within the `except` block. Printing `e` displays its error message. If desired, `e` can be raised again with `raise e` or just `raise`.

The try-except control flow can be expanded with two other blocks, forming a code structure similar to a sequence of `if-elif-else` blocks.

1. The `try` block is executed until an exception is raised (if at all).
2. An `except` statement specifying the same kind of exception that was raised in the try block "catches" the exception, and the block is then executed. There may be multiple `except` blocks following a single `try` block (similar to having several `elif` statements following a single `if` statement), and a single `except` statement may specify multiple kinds of exceptions to catch.
3. The `else` block is executed if an exception was **not** raised in the try block.
4. The `finally` block is always executed if it is included.

```

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
...     raise ValueError("The house is on fire!")
... # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     print("caught an exception.")
...     house_on_fire = True
... else:                                # Skipped due to the exception.
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...caught an exception.
The house is on fire: True

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
... except ValueError as e:                 # Skipped because there was no exception.
...     print("caught a ValueError.")
...     house_on_fire = True
... except TypeError as e:                  # Also skipped.
...     print("caught a TypeError.")
...     house_on_fire = True
... else:
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...no exceptions raised.
The house is on fire: False

```

The code in the `finally` block is always executed, even if a `return` statement or an uncaught exception occurs in any block following the `try` statement.

```

>>> def implode():
...     try:                                # Try to return immediately...
...         return
...     finally:                            # ...but 'finally' goes before 'return'.
...         print("Goodbye, world!")
...
>>> implode()
Goodbye, world!

```

See <https://docs.python.org/3/tutorial/errors.html> for more examples.

Achtung!

An `except` statement with no specified exception type catches **any** exception raised in the corresponding `try` block. This approach can mistakenly mask unexpected errors. Always be specific about the kinds of exceptions you expect to encounter.

```
>>> def divider(x, y):
...     try:
...         return x / yy           # The misspelled yy raises a NameError.
...     except:                  # Catch ANY exception.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
y must not equal zero!

>>> def divider(x, y):
...     try:
...         return x / yy
...     except ZeroDivisionError: # Specify an exception type.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divider
NameError: name 'yy' is not defined      # Now the mistake is obvious.
```

Problem 2. A random walk is a path created by a sequence of random steps. The following function simulates a random walk by repeatedly adding or subtracting 1 to a running total.

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    directions = [1, -1]
    for i in range(int(max_iters)):
        walk += choice(directions)
    return walk
```

A `KeyboardInterrupt` is a special exception that can be triggered at any time by entering `ctrl+c` (on most systems) in the keyboard. Modify `random_walk()` so that if the user raises a `KeyboardInterrupt` by pressing `ctrl+c` while the program is running, the function catches the exception and prints “Process interrupted at iteration i ”. If no `KeyboardInterrupt` is raised, print “Process completed”. In both cases, return `walk` as before.

Note

The built-in exceptions are organized into a class hierarchy. For example, the `ValueError` class inherits from the generic `Exception` class. Thus, a `ValueError` is an `Exception`, but an `Exception` is **not** a `ValueError`.

```
>>> try:
...     raise ValueError("caught!")
... except Exception as e:           # A ValueError is an Exception.
...     print(e)
...
caught!                           # The exception was caught.

>>> try:
...     raise Exception("not caught!")
... except ValueError as e:         # A Exception is not a ValueError.
...     print(e)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: not caught!           # The exception wasn't caught!
```

See <https://docs.python.org/3/library/exceptions.html> for the complete list of built-in exceptions and the exception class hierarchy.

File Input and Output

A file object acts as an interface to a file stream, meaning, it allows a program to read from or write to external files. The built-in function `open()` creates a file object. It accepts the name of the file to open and an editing mode. The mode determines the kind of access that the user has to the file. There are four common modes:

`'r'`: **read**. Open an existing file for reading. The file must already exist, or `open()` raises a `FileNotFoundException`. This is the default mode.

`'w'`: **write**. Create a new file or **overwrite an existing file** (careful!) and open it for writing.

`'x'`: **write new**. Create a new file and open it for writing. If the file already exists, `open()` raises a `FileExistsError`. This is a safer form of `'w'` because it never overwrites existing files.

`'a'`: **append**. Open a file for writing and append new data to the end of the file if it already exists.

```
>>> myfile = open("hello_world.txt", 'r')    # Open a file for reading.
>>> print(myfile.read())                   # Print the contents of the file.
Hello,                                     # (it's a really small file.)
World!

>>> myfile.close()                        # Close the file connection.
```

The With Statement

An `IOError` indicates that some input or output operation has failed. A simple `try-finally` control flow can ensure that a file stream is closed safely.

The `with` statement provides an alternative method for safely opening and closing files. Use `with open(<filename>, <mode>) as <alias>:` to create an indented block in which the file is open and available under the specified alias. At the end of the block, the file is automatically and safely closed, even in the event of an exception. This is the preferred file-reading method when a file only needs to be accessed briefly.

```
>>> myfile = open("hello_world.txt", 'r')      # Open a file for reading.
>>> try:
...     contents = myfile.readlines()          # Read in the content by line.
... finally:
...     myfile.close()                        # Explicitly close the file.

# Equivalently, use a 'with' statement to take care of errors.
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...                                         # The file is closed automatically.
```

In both cases, if the file `hello_world.txt` does not exist in the current directory, `open()` raises a `FileNotFoundException`. However, errors in the `try` or `with` blocks do not prevent the file from being safely closed.

Reading and Writing

Open file objects have an implicit cursor that determines the location in the file to read from or write to. After the entire file has been read once, either the file must be closed and reopened, or the cursor must be reset to the beginning of the file with `seek(0)` before it can be read again.

Some of the more important file object attributes and methods are listed below.

Attribute	Description
<code>closed</code>	<code>True</code> if the object is closed.
<code>mode</code>	The access mode used to open the file object.
<code>name</code>	The name of the file.
Method	Description
<code>close()</code>	Close the connection to the file.
<code>read()</code>	Read a given number of bytes; with no input, read the entire file.
<code>readline()</code>	Read a line of the file, including the newline character at the end.
<code>readlines()</code>	Call <code>readline()</code> repeatedly and return a list of the resulting lines.
<code>seek()</code>	Move the cursor to a new position.
<code>tell()</code>	Report the current position of the cursor.
<code>write()</code>	Write a single string to the file (spaces are not added).
<code>writelines()</code>	Write a list of strings to the file (newline characters are not added).

Only strings can be written to files; to write a non-string type, first cast it as a string with `str()`. Be mindful of spaces and newlines to separate the data.

```
>>> with open("out.txt", 'w') as outfile: # Open 'out.txt' for writing.
...     for i in range(10):
...         outfile.write(str(i**2) + ' ') # Write some strings (and spaces).
...
>>> outfile.closed # The file is closed automatically.
True
```

Problem 3. Define a class called `ContentFilter`. Implement the constructor so that it accepts the name of a file to be read.

1. If the file name is invalid in any way, prompt the user for another filename using `input()`. Continue prompting the user until they provide a valid filename.

```
>>> cf1 = ContentFilter("hello_world.txt") # File exists.
>>> cf2 = ContentFilter("not-a-file.txt") # File doesn't exist.
Please enter a valid file name: still-not-a-file.txt
Please enter a valid file name: hello_world.txt
>>> cf3 = ContentFilter([1, 2, 3]) # Not even a string.
Please enter a valid file name: hello_world.txt
```

(Hint: `open()` might raise a `FileNotFoundException`, a `TypeError`, or an `OSError`.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Make sure the file is securely closed.

String Formatting

The `str` class has several useful methods for parsing and formatting strings. They are particularly useful for processing data from a source file and for preparing data to be written to an external file.

Method	Returns
<code>count()</code>	The number of times a given substring occurs within the string.
<code>find()</code>	The lowest index where a given substring is found.
<code>isalpha()</code>	<code>True</code> if all characters in the string are alphabetic (a, b, c, ...).
<code>isdigit()</code>	<code>True</code> if all characters in the string are digits (0, 1, 2, ...).
<code>isspace()</code>	<code>True</code> if all characters in the string are whitespace (" ", '\t', '\n').
<code>join()</code>	The concatenation of the strings in a given iterable with a specified separator between entries.
<code>lower()</code>	A copy of the string converted to lowercase.
<code>upper()</code>	A copy of the string converted to uppercase.
<code>replace()</code>	A copy of the string with occurrences of a given substring replaced by a different specified substring.
<code>split()</code>	A list of segments of the string, using a given character or string as a delimiter.
<code>strip()</code>	A copy of the string with leading and trailing whitespace removed.

The `join()` method translates a list of strings into a single string by concatenating the entries of the list and placing the principal string between the entries. Conversely, `split()` translates the principal string into a list of substrings, with the separation determined by a single input.

```
# str.join() puts the string between the entries of a list.
>>> words = ["state", "of", "the", "art"]
>>> "-".join(words)
'state-of-the-art'

# str.split() creates a list out of a string, given a delimiter.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split('\n')
['One fish', 'Two fish', 'Red fish', 'Blue fish', '']

# If no delimiter is provided, the string is split by whitespace characters.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split()
['One', 'fish', 'Two', 'fish', 'Red', 'fish', 'Blue', 'fish']
```

Can you tell the difference between the following routines?

```
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.read().split('\n')
```

Variables within Strings

In addition to formatting strings using the methods above there may also be times when you wish to insert generalized variable values into the middle of strings. There are two ways to do this: using `.format()` or using f-strings. Of the two, f-strings are generally more readable, concise, faster, and less prone to error. As a result, an explanation of how to use `.format()` has been banished to the "Additional Materials" section, which we know is rarely read, and a short explanation of f-strings will be included here.

We define an f-string with an 'f' precluding the string declaration (i.e f'your words here', or f"your words here"). Within the f-string, curly brackets can be used along with variable names to insert the variable value into the string. Consider the following code which puts an argument defaulting to 'bread and butter pickles' into the sentence: "I think that (blank) are an abomination." Note that f-strings can also be used with multiple variables at once by using multiple sets of curly brackets.

```
# defining function
>>> def abomination(userOpinion='Bread and Butter Pickles'):
...     # notice the use of the f-string
...     return f'I think that {userOpinion} are an abomination'

# calling the function
>>> abomination()
'I think that Bread and Butter Pickles are an abomination.'
```

Problem 4. Add the following methods to the `ContentFilter` class for writing the contents of the original file to new files. Each method should accept the name of a file to write to and a keyword argument `mode` that specifies the file access mode, defaulting to '`w`'. If `mode` is not '`w`', '`x`', or '`a`', raise a `ValueError` with an informative message.

1. `uniform()`: write the data to the outfile with uniform case. Include an additional keyword argument `case` that defaults to "`upper`".
If `case="upper"`, write the data in upper case. If `case="lower"`, write the data in lower case. If `case` is not one of these two values, raise a `ValueError`.
2. `reverse()`: write the data to the outfile in reverse order. Include an additional keyword argument `unit` that defaults to "`line`".
If `unit="word"`, reverse the ordering of the words in each line, but write the lines in the same order as the original file. If `unit="line"`, reverse the ordering of the lines, but do not change the ordering of the words on each individual line. If `unit` is not one of these two values, raise a `ValueError`.
3. `transpose()`: write a "transposed" version of the data to the outfile. That is, write the first word of each line of the data to the first line of the new file, the second word of each line of the data to the second line of the new file, and so on. Viewed as a matrix of words, the rows of the input file then become the columns of the output file, and vice versa. You may assume that there are an equal number of words on each line of the input file.
4. `__str__()`: Also implement the `__str__()` magic method so that printing a `ContentFilter` object yields the following output. You may want to calculate these statistics in the constructor. (Note: Using f-strings will also make this implementation much simpler).

Source file:	<filename>
Total characters:	<The total number of characters in file>
Alphabetic characters:	<The number of letters>
Numerical characters:	<The number of digits>
Whitespace characters:	<The number of spaces, tabs, and newlines>
Number of lines:	<The number of lines>

(Hint: list comprehensions are **very** useful for some of these functions. For example, what does `[line[::-1] for line in lines]` do? What about `sum([s.isspace() for s in data])`?)

Compare your class to the following example.

```
# cf_example1.txt
A b C
d E f
```

```
>>> cf = ContentFilter("cf_example1.txt")
>>> cf.uniform("uniform.txt", mode='w', case="upper")
>>> cf.uniform("uniform.txt", mode='a', case="lower")
>>> cf.reverse("reverse.txt", mode='w', unit="word")
>>> cf.reverse("reverse.txt", mode='a', unit="line")
>>> cf.transpose("transpose.txt", mode='w')
```

```
# uniform.txt
A B C
D E F
a b c
d e f
```

```
# reverse.txt
C b A
f E d
d E f
A b C
```

```
# transpose.txt
A d
b E
C f
```

Additional Material

Custom Exception Classes

Custom exceptions can be defined by writing a class that inherits from some existing exception class. The generic `Exception` class is typically the parent class of choice.

```
>>> class TooHardError(Exception):
...     pass
...
>>> raise TooHardError("This lab is impossible!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.TooHardError: This lab is impossible!
```

This may seem like a trivial extension of the `Exception` class, but it is useful to do because the interpreter never automatically raises a `TooHardError`. Any `TooHardError` must have originated from a hand-written `raise` command, making it easier to identify the exact source of the problem.

Chaining Exceptions

Sometimes, especially in large programs, it is useful to raise one kind of exception just after catching another. The two exceptions can be linked together using the `from` statement. This syntax makes it possible to see where the error originated from and to “pass it up” to another part of the program.

```
>>> try:
...     raise TooHardError("This lab is impossible!")
... except TooHardError as e:
...     raise NotImplementedError("Lab is incomplete") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.TooHardError: This lab is impossible!

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError: Lab is incomplete
```

More String Formatting Tools

Concatenating string values with non-string values can be cumbersome and tedious. The `str` class’s `format()` method makes it easier to insert non-string values into the middle of a string. Write the desired output in its entirety, replacing non-string values with curly braces `{}`. Then use the `format()` method, entering each replaced value in order.

```
# Join the data using string concatenation.
>>> day, month, year = 10, "June", 2017
```

```
>>> print("Is today", day, str(month) + ',', str(year) + "?")
Is today 10 June, 2017?

# Join the data using str.format().
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 10 June, 2017?

# Join the data easier using a f-string.
>>> print(f"Is today {day} {month}, {year}?")
Is today 10 June, 2017?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Consider the following code for printing out a simple progress bar from within a loop.

```
>>> iters = int(1e7)
>>> chunk = iters // 20
>>> for i in range(iters):
...     print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...                                     end='', flush=True)
...
```

Here the string "`\r[{:<20}]`" used in conjunction with the `format()` method tells the cursor to go back to the beginning of the line, print an opening bracket, then print the first argument of `format()` left-aligned with at least 20 total spaces before printing the closing bracket. The `end` parameter can change the default newline added to the output to another ending. Setting to an empty string, `end=''`, the output ends without any whitespace. The `flush` parameter does not need to be changed if the `end` parameter is not changed and it defaults to `False`. As shown in the example above, setting the parameter equal to `True` forces the output to be printed on the terminal before it is complete. If it is `False` with the `end=''` the `print()` function will first build the output with the set ending before printing each iteration rather than printing incrementally.

Printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

```
>>> for i in range(iters):
...     if not i % chunk:
...         print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...                           end='', flush=True)
...
```

See <https://docs.python.org/3/library/string.html#format-string-syntax> for more examples and specific syntax for using `str.format()`. For a more robust progress bar printer, research the `tqdm` module.

Standard Library Modules for I/O

The standard library has other tools for input and output operations. For details on each module, see <https://docs.python.org/3/library/>.

Module	Description
<code>csv</code>	CSV (comma separated value) file writing and parsing.
<code>io</code>	Support for file objects and <code>open()</code> .
<code>os</code>	Communication with the operating system.
<code>os.path</code>	Common path operations such as checking for file existence.
<code>pickle</code>	Create portable serialized representations of Python objects.

5

Linear Transformations

Lab Objective: Linear transformations are the most basic and essential operators in vector space theory. In this lab we visually explore how linear transformations alter points in the Cartesian plane. We also empirically explore the computational cost of applying linear transformations via matrix multiplication.

Linear Transformations

A linear transformation is a mapping between vector spaces that preserves addition and scalar multiplication. More precisely, let V and W be vector spaces over a common field \mathbb{F} . A map $L : V \rightarrow W$ is a linear transformation from V into W if

$$L(a\mathbf{x}_1 + b\mathbf{x}_2) = aL\mathbf{x}_1 + bL\mathbf{x}_2$$

for all vectors $\mathbf{x}_1, \mathbf{x}_2 \in V$ and scalars $a, b \in \mathbb{F}$.

Every linear transformation L from an m -dimensional vector space into an n -dimensional vector space can be represented by an $m \times n$ matrix A , called the matrix representation of L . To apply L to a vector \mathbf{x} , left multiply by its matrix representation. This results in a new vector \mathbf{x}' , where each component is some linear combination of the elements of \mathbf{x} . For linear transformations from \mathbb{R}^2 to \mathbb{R}^2 , this process has the form

$$A\mathbf{x} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{x}'.$$

Linear transformations can be interpreted geometrically. To demonstrate this, consider the array of points H that collectively form a picture of a horse, stored in the file `horse.npy`. The coordinate pairs \mathbf{x}_i are organized by column, so the array has two rows: one for x -coordinates, and one for y -coordinates. Matrix multiplication on the left transforms each coordinate pair, resulting in another matrix H' whose columns are the transformed coordinate pairs:

$$\begin{aligned} AH = A \begin{bmatrix} x_1 & x_2 & x_3 & \dots \\ y_1 & y_2 & y_3 & \dots \end{bmatrix} &= A \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \dots \end{bmatrix} = \begin{bmatrix} A\mathbf{x}_1 & A\mathbf{x}_2 & A\mathbf{x}_3 & \dots \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}'_1 & \mathbf{x}'_2 & \mathbf{x}'_3 & \dots \end{bmatrix} = \begin{bmatrix} x'_1 & x'_2 & x'_3 & \dots \\ y'_1 & y'_2 & y'_3 & \dots \end{bmatrix} = H'. \end{aligned}$$

To begin, use `np.load()` to extract the array from the `npy` file, then plot the unaltered points as individual pixels. See Figure 5.1 for the result.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

# Load the array from the .npy file.
>>> data = np.load("horse.npy")

# Plot the x row against the y row with black pixels.
>>> plt.plot(data[0], data[1], 'k,')

# Set the window limits to [-1, 1] by [-1, 1] and make the window square.
>>> plt.axis([-1,1,-1,1])
>>> plt.gca().set_aspect("equal")
>>> plt.show()
```

Types of Linear Transformations

Linear transformations from \mathbb{R}^2 into \mathbb{R}^2 can be classified in a few ways.

- **Stretch:** Stretches or compresses the vector along each axis. The matrix representation is diagonal:

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}.$$

If $a = b$, the transformation is called a dilation. The stretch in Figure 5.1 uses $a = \frac{1}{2}$ and $b = \frac{6}{5}$ to compress the x -axis and stretch the y -axis.

- **Shear:** Slants the vector by a scalar factor horizontally or vertically (or both simultaneously). The matrix representation is

$$\begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix}.$$

Pure horizontal shears ($b = 0$) skew the x -coordinate of the vector while pure vertical shears ($a = 0$) skew the y -coordinate. Figure 5.1 has a horizontal shear with $a = \frac{1}{2}, b = 0$.

- **Reflection:** Reflects the vector about a line that passes through the origin. The reflection about the line spanned by the vector $[a, b]^T$ has the matrix representation

$$\frac{1}{a^2 + b^2} \begin{bmatrix} a^2 - b^2 & 2ab \\ 2ab & b^2 - a^2 \end{bmatrix}.$$

The reflection in Figure 5.1 reflects the image about the y -axis ($a = 0, b = 1$).

- **Rotation:** Rotates the vector around the origin. A counterclockwise rotation of θ radians has the following matrix representation:

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

A negative value of θ performs a clockwise rotation. Choosing $\theta = \frac{\pi}{2}$ produces the rotation in Figure 5.1.

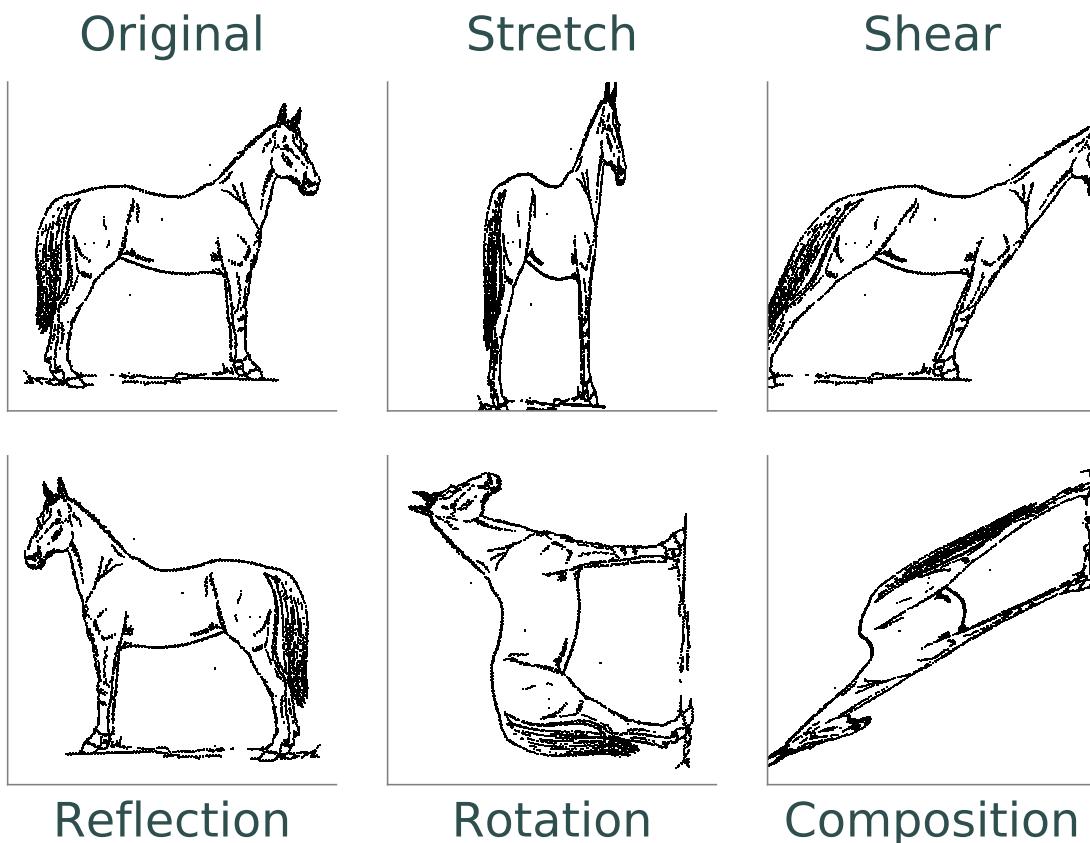


Figure 5.1: The points stored in `horse.npy` under various linear transformations.

Problem 1. Write a function for each type of linear transformation. Each function should accept an array to transform and the scalars that define the transformation (a and b for stretch, shear, and reflection, and θ for rotation). Construct the matrix representation, left multiply it with the input array, and return a transformation of the data.

To test these functions, write a function to plot the original points in `horse.npy` together with the transformed points in subplots for a side-by-side comparison. Compare your results to Figure 5.1.

Compositions of Linear Transformations

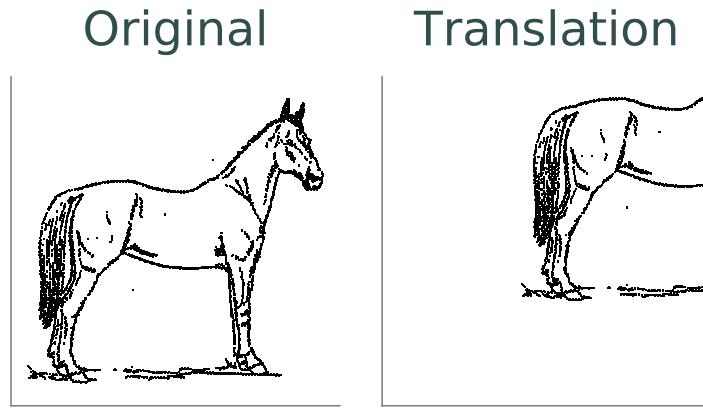
Let V , W , and Z be finite-dimensional vector spaces. If $L : V \rightarrow W$ and $K : W \rightarrow Z$ are linear transformations with matrix representations A and B , respectively, then the composition function $KL : V \rightarrow Z$ is also a linear transformation, and its matrix representation is the matrix product BA .

For example, if S is a matrix representing a shear and R is a matrix representing a rotation, then RS represents a shear followed by a rotation. In fact, any linear transformation $L : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ is a composition of the four transformations discussed above. Figure 5.1 displays the composition of all four previous transformations, applied in order (stretch, shear, reflection, then rotation).

Affine Transformations

All linear transformations map the origin to itself. An affine transformation is a mapping between vector spaces that preserves the relationships between points and lines, but that may not preserve the origin. Every affine transformation T can be represented by a matrix A and a vector \mathbf{b} . To apply T to a vector x , calculate $Ax + \mathbf{b}$. If $\mathbf{b} = \mathbf{0}$ then the transformation is linear, and if $A = I$ but $\mathbf{b} \neq \mathbf{0}$ then it is called a translation.

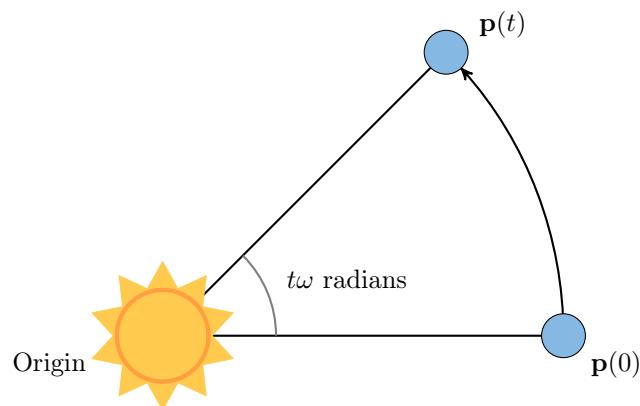
For example, if T is the translation with $\mathbf{b} = [\frac{3}{4}, \frac{1}{2}]^\top$, then applying T to an image will shift it right by $\frac{3}{4}$ and up by $\frac{1}{2}$. This translation is illustrated below.



Affine transformations include all compositions of stretches, shears, rotations, reflections, and translations. For example, if S represents a shear and R a rotation, and if \mathbf{b} is a vector, then $RS\mathbf{x} + \mathbf{b}$ shears, then rotates, then translates \mathbf{x} .

Modeling Motion with Affine Transformations

Affine transformations can be used to model particle motion, such as a planet rotating around the sun. Let the sun be the origin, the planet's location at time t be given by the vector $\mathbf{p}(t)$, and suppose the planet has angular velocity ω (a measure of how fast the planet goes around the sun). To find the planet's position at time t given the planet's initial position $\mathbf{p}(0)$, rotate the vector $\mathbf{p}(0)$ around the origin by $t\omega$ radians. Thus if $R(\theta)$ is the matrix representation of the linear transformation that rotates a vector around the origin by θ radians, then $\mathbf{p}(t) = R(t\omega)\mathbf{p}(0)$.



Composing the rotation with a translation shifts the center of rotation away from the origin, yielding more complicated motion.

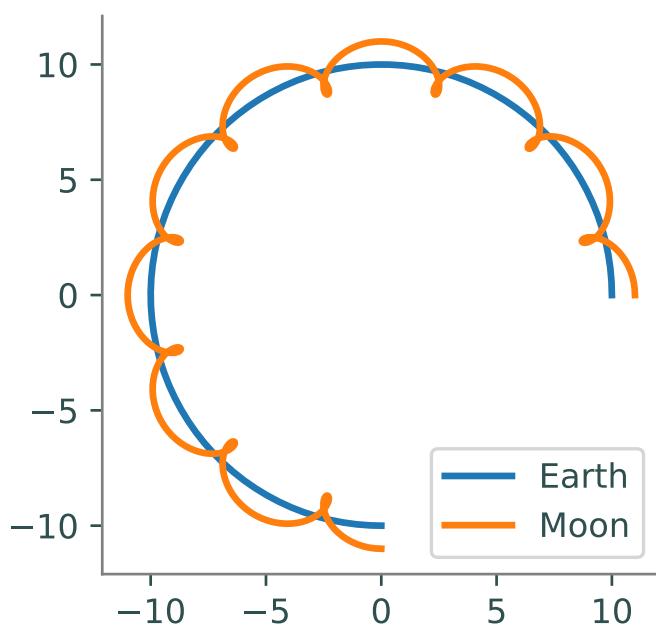
Problem 2. The moon orbits the earth while the earth orbits the sun. Assuming circular orbits, we can compute the trajectories of both the earth and the moon using only linear and affine transformations.

Assume an orientation where both the earth and moon travel counterclockwise, with the sun at the origin. Let $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ be the positions of the earth and the moon at time t , respectively, and let ω_e and ω_m be each celestial body's angular velocity. For a particular time t , we calculate $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ with the following steps.

1. Compute $\mathbf{p}_e(t)$ by rotating the initial vector $\mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_e$ radians.
2. Calculate the position of the moon relative to the earth at time t by rotating the vector $\mathbf{p}_m(0) - \mathbf{p}_e(0)$ counterclockwise about the origin by $t\omega_m$ radians.
3. To compute $\mathbf{p}_m(t)$, translate the vector resulting from the previous step by $\mathbf{p}_e(t)$.

Write a function that accepts a final time T , initial positions x_e and x_m , and the angular momenta ω_e and ω_m . Assuming initial positions $\mathbf{p}_e(0) = (x_e, 0)$ and $\mathbf{p}_m(0) = (x_m, 0)$, plot $\mathbf{p}_e(t)$ and $\mathbf{p}_m(t)$ over the time interval $t \in [0, T]$.

Setting $T = \frac{3\pi}{2}$, $x_e = 10$, $x_m = 11$, $\omega_e = 1$, and $\omega_m = 13$, your plot should resemble the following figure (fix the aspect ratio with `ax.set_aspect("equal")`). Note that a more celestially accurate figure would use $x_e = 400$, $x_m = 401$ (the interested reader should see <http://www.math.nus.edu.sg/aslaksen/teaching/convex.html>).



Timing Matrix Operations

Linear transformations are easy to perform via matrix multiplication. However, performing matrix multiplication with very large matrices can strain a machine's time and memory constraints. For the remainder of this lab we take an empirical approach in exploring how much time and memory different matrix operations require.

Timing Code

Recall that the `time` module's `time()` function measures the number of seconds since the Epoch. To measure how long it takes for code to run, record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed. Additionally, in IPython, the quick command `%timeit` uses the `timeit` module to quickly time a single line of code.

```
In [1]: import time

In [2]: def for_loop():
....:     """Go through ten million iterations of nothing."""
....:     for _ in range(int(1e7)):
....:         pass

In [3]: def time_for_loop():
....:     """Time for_loop() with time.time()."""
....:     start = time.time()           # Clock the starting time.
....:     for_loop()
....:     return time.time() - start   # Return the elapsed time.

In [4]: time_for_loop()
0.24458789825439453

In [5]: %timeit for_loop()
248 ms +- 5.35 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Timing an Algorithm

Most algorithms have at least one input that dictates the size of the problem to be solved. For example, the following functions take in a single integer n and produce a random vector of length n as a list or a random $n \times n$ matrix as a list of lists.

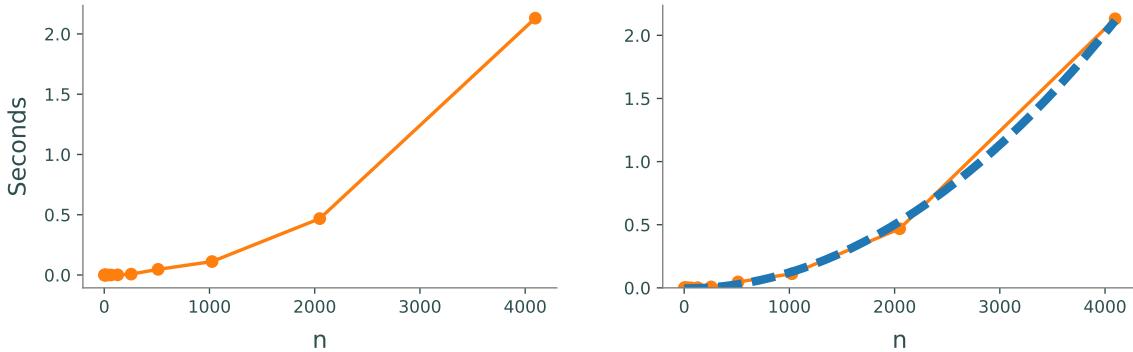
```
from random import random
def random_vector(n):          # Equivalent to np.random.random(n).tolist()
    """Generate a random vector of length n as a list."""
    return [random() for i in range(n)]

def random_matrix(n):          # Equivalent to np.random.random((n,n)).tolist()
    """Generate a random nxn matrix as a list of lists."""
    return [[random() for j in range(n)] for i in range(n)]
```

Executing `random_vector(n)` calls `random()` n times, so doubling n should about double the amount of time `random_vector(n)` takes to execute. By contrast, executing `random_matrix(n)` calls `random()` n^2 times (n times per row with n rows). Therefore doubling n will likely more than double the amount of time `random_matrix(n)` takes to execute, especially if n is large.

To visualize this phenomenon, we time `random_matrix()` for $n = 2^1, 2^2, \dots, 2^{12}$ and plot n against the execution time. The result is displayed below on the left.

```
>>> domain = 2**np.arange(1,13)
>>> times = []
>>> for n in domain:
...     start = time.time()
...     random_matrix(n)
...     times.append(time.time() - start)
...
>>> plt.plot(domain, times, 'g.-', linewidth=2, markersize=15)
>>> plt.xlabel("n", fontsize=14)
>>> plt.ylabel("Seconds", fontsize=14)
>>> plt.show()
```



The figure on the left shows that the execution time for `random_matrix(n)` increases quadratically in n . In fact, the blue dotted line in the figure on the right is the parabola $y = an^2$, which fits nicely over the timed observations. Here a is a small constant, but it is much less significant than the exponent on the n . To represent this algorithm's growth, we ignore a altogether and write `random_matrix(n) $\sim n^2$` .

Note

An algorithm like `random_matrix(n)` whose execution time increases quadratically with n is called $O(n^2)$, denoted by `random_matrix(n) $\in O(n^2)$` . Big-oh notation is common for indicating both the temporal complexity of an algorithm (how the execution time grows with n) and the spatial complexity (how the memory usage grows with n).

Problem 3. Let A be an $m \times n$ matrix with entries a_{ij} , \mathbf{x} be an $n \times 1$ vector with entries x_k , and B be an $n \times p$ matrix with entries b_{ij} . The matrix-vector product $A\mathbf{x} = \mathbf{y}$ is a new $m \times 1$ vector and the matrix-matrix product $AB = C$ is a new $m \times p$ matrix. The entries y_i of \mathbf{y} and c_{ij} of C are determined by the following formulas:

$$y_i = \sum_{k=1}^n a_{ik}x_k \quad c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$$

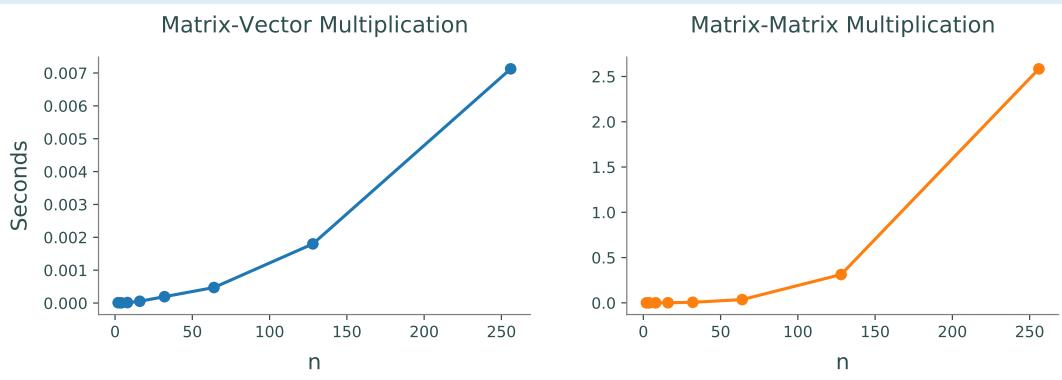
These formulas are implemented below **without** using NumPy arrays or operations.

```
def matrix_vector_product(A, x):      # Equivalent to np.dot(A,x).tolist()
    """Compute the matrix-vector product Ax as a list."""
    m, n = len(A), len(x)
    return [sum([A[i][k] * x[k] for k in range(n)]) for i in range(m)]

def matrix_matrix_product(A, B):        # Equivalent to np.dot(A,B).tolist()
    """Compute the matrix-matrix product AB as a list of lists."""
    m, n, p = len(A), len(B), len(B[0])
    return [[sum([A[i][k] * B[k][j] for k in range(n)])
            for j in range(p)] for i in range(m)]
```

Time each of these functions with increasingly large inputs. Generate the inputs A , \mathbf{x} , and B with `random_matrix()` and `random_vector()` (so each input will be $n \times n$ or $n \times 1$). Only time the multiplication functions, not the generating functions.

Report your findings in a single figure with two subplots: one with matrix-vector times, and one with matrix-matrix times. Choose a domain for n so that your figure accurately describes the growth, but avoid values of n that lead to execution times of more than 1 minute. Your figure should resemble the following plots.



Logarithmic Plots

Though the two plots from Problem 3 look similar, the scales on the y -axes show that the actual execution times differ greatly. To be compared correctly, the results need to be viewed differently.

A logarithmic plot uses a logarithmic scale—with values that increase exponentially, such as $10^1, 10^2, 10^3, \dots$ —on one or both of its axes. The three kinds of log plots are listed below.

- **log-lin:** the x -axis uses a logarithmic scale but the y -axis uses a linear scale.
Use `plt.semilogx()` instead of `plt.plot()`.
- **lin-log:** the x -axis is uses a linear scale but the y -axis uses a log scale.
Use `plt.semilogy()` instead of `plt.plot()`.
- **log-log:** both the x and y -axis use a logarithmic scale.
Use `plt.loglog()` instead of `plt.plot()`.

Since the domain $n = 2^1, 2^2, \dots$ is a logarithmic scale and the execution times increase quadratically, we visualize the results of the previous problem with a log-log plot. The default base for the logarithmic scales on logarithmic plots in Matplotlib is 10. To change the base to 2 on each axis, specify the keyword arguments `base=2`.

Suppose the domain of n values are stored in `domain` and the corresponding execution times for `matrix_vector_product()` and `matrix_matrix_product()` are stored in `vector_times` and `matrix_times`, respectively. Then the following code produces Figure 5.5.

```
>>> ax1 = plt.subplot(121) # Plot both curves on a regular lin-lin plot.
>>> ax1.plot(domain, vector_times, 'b.-', lw=2, ms=15, label="Matrix-Vector")
>>> ax1.plot(domain, matrix_times, 'g.-', lw=2, ms=15, label="Matrix-Matrix")
>>> ax1.legend(loc="upper left")

>>> ax2 = plt.subplot(122) # Plot both curves on a base 2 log-log plot.
>>> ax2.loglog(domain, vector_times, 'b.-', base=2, lw=2)
>>> ax2.loglog(domain, matrix_times, 'g.-', base=2, lw=2)

>>> plt.show()
```

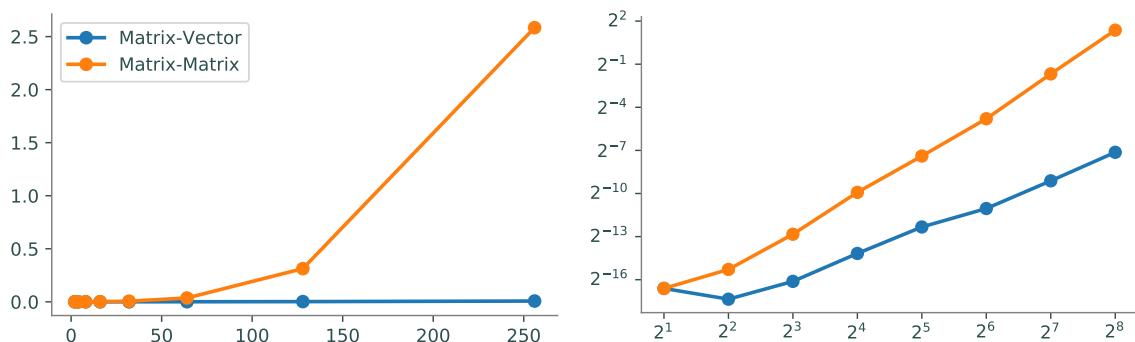


Figure 5.5

In the log-log plot, the slope of the `matrix_matrix_product()` line is about 3 and the slope of the `matrix_vector_product()` line is about 2. This reflects the fact that matrix-matrix multiplication (which uses 3 loops) is $O(n^3)$, while matrix-vector multiplication (which only has 2 loops) is only $O(n^2)$.

Problem 4. NumPy is built specifically for fast numerical computations. Repeat the experiment of Problem 3, timing the following operations:

- matrix-vector multiplication with `matrix_vector_product()`.
- matrix-matrix multiplication with `matrix_matrix_product()`.
- matrix-vector multiplication with `np.dot()` or `@`.
- matrix-matrix multiplication with `np.dot()` or `@`.

Create a single figure with two subplots: one with all four sets of execution times on a regular linear scale, and one with all four sets of execution times on a log-log scale. Compare your results to Figure 5.5.

Note

Problem 4 shows that **matrix operations are significantly faster in NumPy than in plain Python**. Matrix-matrix multiplication grows cubically regardless of the implementation; however, with lists the times grows at a rate of an^3 while with NumPy the times grow at a rate of bn^3 , where a is much larger than b . NumPy is more efficient for several reasons:

1. Iterating through loops is very expensive. Many of NumPy's operations are implemented in C, which are much faster than Python loops.
2. Arrays are designed specifically for matrix operations, while Python lists are general purpose.
3. NumPy carefully takes advantage of computer hardware, efficiently using different levels of computer memory.

However, in Problem 4, the execution times for matrix multiplication with NumPy seem to increase somewhat inconsistently. This is because the fastest layer of computer memory can only handle so much information before the computer has to begin using a larger, slower layer of memory.

Additional Material

Image Transformation as a Class

Consider organizing the functions from Problem 1 into a class. The constructor might accept an array or the name of a file containing an array. This structure would make it easy to do several linear or affine transformations in sequence.

```
>>> horse = ImageTransformer("horse.npy")
>>> horse.stretch(.5, 1.2)
>>> horse.shear(.5, 0)
>>> horse.select(0, 1)
>>> horse.rotate(np.pi/2.)
>>> horse.translate(.75, .5)
>>> horse.display()
```

Animating Parametrizations

The plot in Problem 2 fails to fully convey the system's evolution over time because time itself is not part of the plot. The following function creates an animation for the earth and moon trajectories.

```
from matplotlib.animation import FuncAnimation

def solar_system_animation(earth, moon):
    """Animate the moon orbiting the earth and the earth orbiting the sun.
    Parameters:
        earth ((2,N) ndarray): The earth's position with x-coordinates on the
            first row and y coordinates on the second row.
        moon ((2,N) ndarray): The moon's position with x-coordinates on the
            first row and y coordinates on the second row.
    """
    fig, ax = plt.subplots(1,1)                                     # Make a figure explicitly.
    plt.axis([-15,15,-15,15])                                      # Set the window limits.
    ax.set_aspect("equal")                                         # Make the window square.
    earth_dot, = ax.plot([],[], 'C0o', ms=10)                      # Blue dot for the earth.
    earth_path, = ax.plot([],[], 'C0-')                            # Blue line for the earth.
    moon_dot, = ax.plot([],[], 'C2o', ms=5)                         # Green dot for the moon.
    moon_path, = ax.plot([],[], 'C2-')                            # Green line for the moon.
    ax.plot([0],[0], 'y*', ms=20)                                    # Yellow star for the sun.

    def animate(index):
        earth_dot.set_data(earth[0,index], earth[1,index])
        earth_path.set_data(earth[0,:index], earth[1,:index])
        moon_dot.set_data(moon[0,index], moon[1,index])
        moon_path.set_data(moon[0,:index], moon[1,:index])
        return earth_dot, earth_path, moon_dot, moon_path,
    a = FuncAnimation(fig, animate, frames=earth.shape[1], interval=25)
    plt.show()
```


6

Linear Systems

Lab Objective: The fundamental problem of linear algebra is solving the linear system $A\mathbf{x} = \mathbf{b}$, given that a solution exists. There are many approaches to solving this problem, each with different pros and cons. In this lab we implement the LU decomposition and use it to solve square linear systems. We also introduce SciPy, together with its libraries for linear algebra and working with sparse matrices.

Gaussian Elimination

The standard approach for solving the linear system $A\mathbf{x} = \mathbf{b}$ on paper is reducing the augmented matrix $[A | \mathbf{b}]$ to row-echelon form (REF) via Gaussian elimination, then using back substitution. The matrix is in REF when the leading non-zero term in each row is the diagonal term, so the matrix is upper triangular.

At each step of Gaussian elimination, there are three possible operations: swapping two rows, multiplying one row by a scalar value, or adding a scalar multiple of one row to another. Many systems, like the one displayed below, can be reduced to REF using only the third type of operation. First, use multiples of the first row to get zeros below the diagonal in the first column, then use a multiple of the second row to get zeros below the diagonal in the second column.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 3 & 4 & 5 \end{array} \right] \rightarrow \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array} \right]$$

Each of these operations is equivalent to left-multiplying by a type III elementary matrix, the identity with a single non-zero non-diagonal term. If row operation k corresponds to matrix E_k , the following equation is $E_3E_2E_1A = U$.

$$\left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{array} \right] \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{array} \right] \left[\begin{array}{ccc} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array} \right]$$

However, matrix multiplication is an inefficient way to implement row reduction. Instead, modify the matrix in place (without making a copy), changing only those entries that are affected by each row operation.

```
>>> import numpy as np

>>> A = np.array([[1, 1, 1, 1],
...                 [1, 4, 2, 3],
...                 [4, 7, 8, 9]], dtype=np.float)

# Reduce the 0th column to zeros below the diagonal.
>>> A[1,0:] -= (A[1,0] / A[0,0]) * A[0]
>>> A[2,0:] -= (A[2,0] / A[0,0]) * A[0]

# Reduce the 1st column to zeros below the diagonal.
>>> A[2,1:] -= (A[2,1] / A[1,1]) * A[1,1:]
>>> print(A)
[[ 1.  1.  1.  1.]
 [ 0.  3.  1.  2.]
 [ 0.  0.  3.  3.]]
```

Note that the final row operation modifies only part of the third row to avoid spending the computation time of adding 0 to 0.

If a 0 appears on the main diagonal during any part of row reduction, the approach given above tries to divide by 0. Swapping the current row with one below it that does not have a 0 in the same column solves this problem. This is equivalent to left-multiplying by a type II elementary matrix, also called a permutation matrix.

Achtung!

Gaussian elimination is not always numerically stable. In other words, it is susceptible to rounding error that may result in an incorrect final matrix. Suppose that, due to roundoff error, the matrix A has a very small entry on the diagonal.

$$A = \begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix}$$

Though 10^{-15} is essentially zero, instead of swapping the first and second rows to put A in REF, a computer might multiply the first row by 10^{15} and add it to the second row to eliminate the -1 . The resulting matrix is far from what it would be if the 10^{-15} were actually 0.

$$\begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10^{-15} & 1 \\ 0 & 10^{15} \end{bmatrix}$$

Round-off error can propagate through many steps in a calculation. The NumPy routines that employ row reduction use several tricks to minimize the impact of round-off error, but these tricks cannot fix every matrix.

Problem 1. Write a function that reduces an arbitrary square matrix A to REF. You may assume that A is invertible and that a 0 will never appear on the main diagonal (so only use type III row reductions, not type II). Avoid operating on entries that you know will be 0 before and after a row operation. Use at most two nested loops.

Test your function with small test cases that you can check by hand. Consider using `np.random.randint()` to generate a few manageable tests cases.

The LU Decomposition

The LU decomposition of a square matrix A is a factorization $A = LU$ where U is the **upper** triangular REF of A and L is the **lower** triangular product of the type III elementary matrices whose inverses reduce A to U . The LU decomposition of A exists when A can be reduced to REF using only type III elementary matrices (without any row swaps). However, the rows of A can always be permuted in a way such that the decomposition exists. If P is a permutation matrix encoding the appropriate row swaps, then the decomposition $PA = LU$ always exists.

Suppose A has an LU decomposition (not requiring row swaps). Then A can be reduced to REF with k row operations, corresponding to left-multiplying the type III elementary matrices E_1, \dots, E_k . Because there were no row swaps, each E_i is lower triangular, so each inverse E_i^{-1} is also lower triangular. Furthermore, since the product of lower triangular matrices is lower triangular, L is lower triangular:

$$\begin{aligned} E_k \dots E_2 E_1 A &= U \quad \longrightarrow \quad A = (E_k \dots E_2 E_1)^{-1} U \\ &= E_1^{-1} E_2^{-1} \dots E_k^{-1} U \\ &= LU. \end{aligned}$$

Thus, L can be computed by right-multiplying the identity by the matrices used to reduce U . However, in this special situation, each right-multiplication only changes one entry of L , matrix multiplication can be avoided altogether. The entire process, only slightly different than row reduction, is summarized below.

Algorithm 6.1

```

1: procedure LU Decomposition( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                       $\triangleright$  Store the dimensions of  $A$ .
3:    $U \leftarrow \text{copy}(A)$                                       $\triangleright$  Make a copy of  $A$  with np.copy().
4:    $L \leftarrow I_m$                                           $\triangleright$  The  $m \times m$  identity matrix.
5:   for  $j = 0 \dots n - 1$  do
6:     for  $i = j + 1 \dots m - 1$  do
7:        $L_{i,j} \leftarrow U_{i,j}/U_{j,j}$ 
8:        $U_{i,j:} \leftarrow U_{i,j:} - L_{i,j}U_{j,j:}$ 
9:   return  $L, U$ 

```

Problem 2. Write a function that finds the LU decomposition of a square matrix. You may assume that the decomposition exists and requires no row swaps.

Forward and Backward Substitution

If $PA = LU$ and $A\mathbf{x} = \mathbf{b}$, then $LU\mathbf{x} = PA\mathbf{x} = P\mathbf{b}$. This system can be solved by first solving $L\mathbf{y} = P\mathbf{b}$, then $U\mathbf{x} = \mathbf{y}$. Since L and U are both triangular, these systems can be solved with backward and forward substitution. We can thus compute the LU factorization of A once, then use substitution to efficiently solve $A\mathbf{x} = \mathbf{b}$ for various values of \mathbf{b} .

Since the diagonal entries of L are all 1, the triangular system $L\mathbf{y} = \mathbf{b}$ has the form

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}.$$

Matrix multiplication yields the equations

$$\begin{aligned} y_1 &= b_1, & y_1 &= b_1, \\ l_{21}y_1 + y_2 &= b_2, & y_2 &= b_2 - l_{21}y_1, \\ &\vdots &&\vdots \\ \sum_{j=1}^{k-1} l_{kj}y_j + y_k &= b_k, & y_k &= b_k - \sum_{j=1}^{k-1} l_{kj}y_j. \end{aligned} \tag{6.1}$$

The triangular system $U\mathbf{x} = \mathbf{y}$ yields similar equations, but in reverse order:

$$\begin{aligned} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} &= \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}, \\ u_{nn}x_n &= y_n, & x_n &= \frac{1}{u_{nn}}y_n, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, & x_{n-1} &= \frac{1}{u_{n-1,n-1}}(y_{n-1} - u_{n-1,n}x_n), \\ &\vdots &&\vdots \\ \sum_{j=k}^n u_{kj}x_j &= y_k, & x_k &= \frac{1}{u_{kk}} \left(y_k - \sum_{j=k+1}^n u_{kj}x_j \right). \end{aligned} \tag{6.2}$$

Problem 3. Write a function that, given A and \mathbf{b} , solves the square linear system $A\mathbf{x} = \mathbf{b}$. Use the function from Problem 2 to compute L and U , then use (6.1) and (6.2) to solve for \mathbf{y} , then \mathbf{x} . You may again assume that no row swaps are required ($P = I$ in this case).

SciPy

SciPy [JOP⁺] is a powerful scientific computing library built upon NumPy. It includes high-level tools for linear algebra, statistics, signal processing, integration, optimization, machine learning, and more.

SciPy is typically imported with the convention `import scipy as sp`. However, SciPy is set up in a way that requires its submodules to be imported individually.¹

```
>>> import scipy as sp
>>> hasattr(sp, "stats")           # The stats module isn't loaded yet.
False

>>> from scipy import stats        # Import stats explicitly. Access it
>>> hasattr(sp, "stats")           # with 'stats' or 'sp.stats'.
True
```

Linear Algebra

NumPy and SciPy both have a linear algebra module, each called `linalg`, but SciPy's module is the larger of the two. Some of SciPy's common `linalg` functions are listed below.

Function	Returns
<code>det()</code>	The determinant of a square matrix.
<code>eig()</code>	The eigenvalues and eigenvectors of a square matrix.
<code>inv()</code>	The inverse of an invertible matrix.
<code>norm()</code>	The norm of a vector or matrix norm of a matrix.
<code>solve()</code>	The solution to $Ax = b$ (the system need not be square).

This library also includes routines for computing matrix decompositions.

```
>>> from scipy import linalg as la

# Make a random matrix and a random vector.
>>> A = np.random.random((1000,1000))
>>> b = np.random.random(1000)

# Compute the LU decomposition of A, including pivots.
>>> L, P = la.lu_factor(A)

# Use the LU decomposition to solve Ax = b.
>>> x = la.lu_solve((L,P), b)

# Check that the solution is legitimate.
>>> np.allclose(A @ x, b)
True
```

¹SciPy modules like `linalg` are really packages, which are not initialized when SciPy is imported alone.

As with NumPy, SciPy's routines are all highly optimized. However, some algorithms are, by nature, faster than others.

Problem 4. Write a function that times different `scipy.linalg` functions for solving square linear systems.

For various values of n , generate a random $n \times n$ matrix A and a random n -vector \mathbf{b} using `np.random.random()`. Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

1. Invert A with `la.inv()` and left-multiply the inverse to \mathbf{b} .
2. Use `la.solve()`.
3. Use `la.lu_factor()` and `la.lu_solve()` to solve the system with the LU decomposition.
4. Use `la.lu_factor()` and `la.lu_solve()`, but only time `la.lu_solve()` (not the time it takes to do the factorization with `la.lu_factor()`).

Plot the system size n versus the execution times. Use log scales if needed.

Achtung!

Problem 4 demonstrates that computing a matrix inverse is computationally expensive. In fact, numerically inverting matrices is so costly that there is hardly ever a good reason to do it. Use a specific solver like `la.lu_solve()` whenever possible instead of using `la.inv()`.

Sparse Matrices

Large linear systems can have tens of thousands of entries. Storing the corresponding matrices in memory can be difficult: a $10^5 \times 10^5$ system requires around 40 GB to store in a NumPy array (4 bytes per entry $\times 10^{10}$ entries). This is well beyond the amount of RAM in a normal laptop.

In applications where systems of this size arise, it is often the case that the system is sparse, meaning that most of the entries of the matrix are 0. SciPy's `sparse` module provides tools for efficiently constructing and manipulating 1- and 2-D sparse matrices. A `sparse` matrix only stores the nonzero values and the positions of these values. For sufficiently sparse matrices, storing the matrix as a `sparse` matrix may only take megabytes, rather than gigabytes.

For example, diagonal matrices are sparse. Storing an $n \times n$ diagonal matrix in the naïve way means storing n^2 values in memory. It is more efficient to instead store the diagonal entries in a 1-D array of n values. In addition to using less storage space, this allows for much faster matrix operations: the standard algorithm to multiply a matrix by a diagonal matrix involves n^3 steps, but most of these are multiplying by or adding 0. A smarter algorithm can accomplish the same task much faster.

SciPy has seven sparse matrix types. Each type is optimized either for storing sparse matrices whose nonzero entries follow certain patterns, or for performing certain computations.

Name	Description	Advantages
<code>bsr_matrix</code>	Block Sparse Row	Specialized structure.
<code>coo_matrix</code>	Coordinate Format	Conversion among sparse formats.
<code>csc_matrix</code>	Compressed Sparse Column	Column-based operations and slicing.
<code>csr_matrix</code>	Compressed Sparse Row	Row-based operations and slicing.
<code>dia_matrix</code>	Diagonal Storage	Specialized structure.
<code>dok_matrix</code>	Dictionary of Keys	Element access, incremental construction.
<code>lil_matrix</code>	Row-based Linked List	Incremental construction.

Creating Sparse Matrices

A regular, non-sparse matrix is called full or dense. Full matrices can be converted to each of the sparse matrix formats listed above. However, it is more memory efficient to never create the full matrix in the first place. There are three main approaches for creating sparse matrices from scratch.

- **Coordinate Format:** When all of the nonzero values and their positions are known, create the entire sparse matrix at once as a `coo_matrix`. All nonzero values are stored as a coordinate and a value. This format also converts quickly to other sparse matrix types.

```
>>> from scipy import sparse

# Define the rows, columns, and values separately.
>>> rows = np.array([0, 1, 0])
>>> cols = np.array([0, 1, 1])
>>> vals = np.array([3, 5, 2])
>>> A = sparse.coo_matrix((vals, (rows,cols)), shape=(3,3))
>>> print(A)
(0, 0)    3
(1, 1)    5
(0, 1)    2

# The toarray() method casts the sparse matrix as a NumPy array.
>>> print(A.toarray())
[[3 2 0]          # Note that this method forfeits
 [0 5 0]          # all sparsity-related optimizations.
 [0 0 0]]
```

- **DOK and LIL Formats:** If the matrix values and their locations are not known beforehand, construct the matrix incrementally with a `dok_matrix` or a `lil_matrix`. Indicate the size of the matrix, then change individual values with regular slicing syntax.

```
>>> B = sparse.lil_matrix((2,6))
>>> B[0,2] = 4
>>> B[1,3:] = 9

>>> print(B.toarray())
[[ 0.  0.  4.  0.  0.  0.]
 [ 0.  0.  0.  9.  9.  9.]]
```

- **DIA Format:** Use a `dia_matrix` to store matrices that have nonzero entries on only certain diagonals. The function `sparse.diags()` is one convenient way to create a `dia_matrix` from scratch. Additionally, every sparse matrix has a `setdiags()` method for modifying specified diagonals.

```
# Use sparse.diags() to create a matrix with diagonal entries.
>>> diagonals = [[1,2],[3,4,5],[6]]      # List the diagonal entries.
>>> offsets = [-1,0,3]                  # Specify the diagonal they go on.
>>> print(sparse.diags(diagonals, offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  4.  0.  0.]
 [ 0.  2.  5.  0.]]

# If all of the diagonals have the same entry, specify the entry alone.
>>> A = sparse.diags([1,3,6], offsets, shape=(3,4))
>>> print(A.toarray())
[[ 3.  0.  0.  6.]
 [ 1.  3.  0.  0.]
 [ 0.  1.  3.  0.]]

# Modify a diagonal with the setdiag() method.
>>> A.setdiag([4,4,4], 0)
>>> print(A.toarray())
[[ 4.  0.  0.  6.]
 [ 1.  4.  0.  0.]
 [ 0.  1.  4.  0.]]
```

- **BSR Format:** Many sparse matrices can be formulated as block matrices, and a block matrix can be stored efficiently as a `bsr_matrix`. Use `sparse.bmat()` or `sparse.block_diag()` to create a block matrix quickly.

```
# Use sparse.bmat() to create a block matrix. Use 'None' for zero blocks.
>>> A = sparse.coo_matrix(np.ones((2,2)))
>>> B = sparse.coo_matrix(np.full((2,2), 2.))
>>> print(sparse.bmat([[ A , None,  A ],
                      [None,  B , None]], format='bsr').toarray())
[[ 1.  1.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  1.  1.]
 [ 0.  0.  2.  2.  0.  0.]
 [ 0.  0.  2.  2.  0.  0.]]

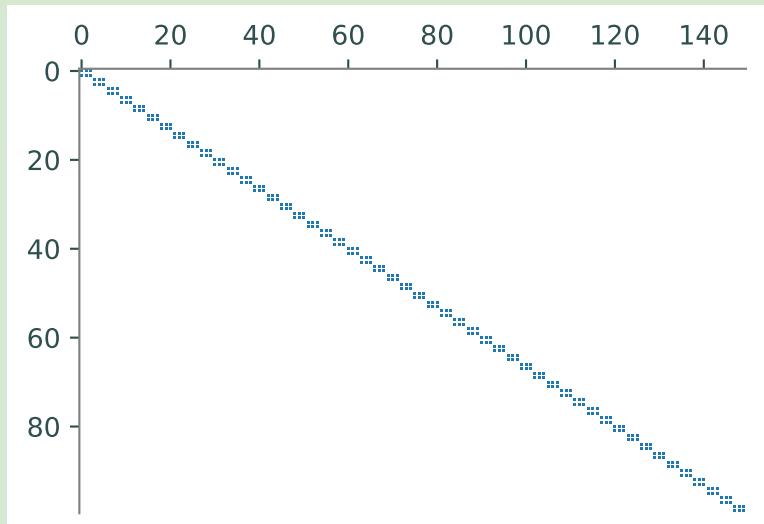
# Use sparse.block_diag() to construct a block diagonal matrix.
>>> print(sparse.block_diag((A,B)).toarray())
[[ 1.  1.  0.  0.]
 [ 1.  1.  0.  0.]
 [ 0.  0.  2.  2.]
 [ 0.  0.  2.  2.]]
```

Note

If a sparse matrix is too large to fit in memory as an array, it can still be visualized with Matplotlib's `plt.spy()`, which colors in the locations of the non-zero entries of the matrix.

```
>>> from matplotlib import pyplot as plt

# Construct and show a matrix with 50 2x3 diagonal blocks.
>>> B = sparse.coo_matrix([[1,3,5],[7,9,11]])
>>> A = sparse.block_diag([B]*50)
>>> plt.spy(A, markersize=1)
>>> plt.show()
```



Problem 5. Let I be the $n \times n$ identity matrix, and define

$$A = \begin{bmatrix} B & I & & \\ I & B & I & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{bmatrix}, \quad B = \begin{bmatrix} -4 & 1 & & \\ 1 & -4 & 1 & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix},$$

where A is $n^2 \times n^2$ and each block B is $n \times n$. The large matrix A is used in finite difference methods for solving Laplace's equation in two dimensions, $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$.

Write a function that accepts an integer n and constructs and returns A as a sparse matrix. Use `plt.spy()` to check that your matrix has nonzero values in the correct places.

Sparse Matrix Operations

Once a sparse matrix has been constructed, it should be converted to a `csr_matrix` or a `csc_matrix` with the matrix's `to csr()` or `to csc()` method. The CSR and CSC formats are optimized for row or column operations, respectively. To choose the correct format to use, determine what direction the matrix will be traversed.

For example, in the matrix-matrix multiplication AB , A is traversed row-wise, but B is traversed column-wise. Thus A should be converted to a `csr_matrix` and B should be converted to a `csc_matrix`.

```
# Initialize a sparse matrix incrementally as a lil_matrix.
>>> A = sparse.lil_matrix((10000,10000))
>>> for k in range(10000):
...     A[np.random.randint(0,9999), np.random.randint(0,9999)] = k
...
>>> A
<10000x10000 sparse matrix of type '<type 'numpy.float64'>' with 9999 stored elements in LInked List format>

# Convert A to CSR and CSC formats to compute the matrix product AA.
>>> Acsr = A.tocsr()
>>> Acsc = A.tocsc()
>>> Acsr.dot(Acsc)
<10000x10000 sparse matrix of type '<type 'numpy.float64'>' with 10142 stored elements in Compressed Sparse Row format>
```

Beware that row-based operations on a `csc_matrix` are very slow, and similarly, column-based operations on a `csr_matrix` are very slow.

Achtung!

Many familiar NumPy operations have analogous routines in the `sparse` module. These methods take advantage of the sparse structure of the matrices and are, therefore, usually significantly faster. However, SciPy's `sparse` matrices behave a little differently than NumPy arrays.

Operation	<code>numpy</code>	<code>scipy.sparse</code>
Component-wise Addition	<code>A + B</code>	<code>A + B</code>
Scalar Multiplication	<code>2 * A</code>	<code>2 * A</code>
Component-wise Multiplication	<code>A * B</code>	<code>A.multiply(B)</code>
Matrix Multiplication	<code>A.dot(B), A @ B</code>	<code>A * B, A.dot(B), A @ B</code>

Note in particular the difference between `A * B` for NumPy arrays and SciPy sparse matrices. Do **not** use `np.dot()` to try to multiply sparse matrices, as it may treat the inputs incorrectly. The syntax `A.dot(B)` is safest in most cases.

SciPy's `sparse` module has its own linear algebra library, `scipy.sparse.linalg`, designed for operating on sparse matrices. Like other SciPy modules, it must be imported explicitly.

```
>>> from scipy.sparse import linalg as spla
```

Problem 6. Write a function that times regular and sparse linear system solvers.

For various values of n , generate the $n^2 \times n^2$ matrix A described in Problem 5 and a random vector \mathbf{b} with n^2 entries. Time how long it takes to solve the system $A\mathbf{x} = \mathbf{b}$ with each of the following approaches:

1. Convert A to CSR format and use `scipy.sparse.linalg.spsolve()` (`spla.spsolve()`).
2. Convert A to a NumPy array and use `scipy.linalg.solve()` (`la.solve()`).

In each experiment, only time how long it takes to solve the system (not how long it takes to convert A to the appropriate format).

Plot the system size n^2 versus the execution times. As always, use log scales where appropriate and use a legend to label each line.

Achtung!

Even though there are fast algorithms for solving certain sparse linear systems, it is still very computationally difficult to invert sparse matrices. In fact, the inverse of a sparse matrix is usually not sparse. There is rarely a good reason to invert a matrix, sparse or dense.

See <http://docs.scipy.org/doc/scipy/reference/sparse.html> for additional details on SciPy's `sparse` module.

Additional Material

Improvements on the LU Decomposition

Vectorization

Algorithm 6.1 uses two loops to compute the LU decomposition. With a little vectorization, the process can be reduced to a single loop.

Algorithm 6.2

```

1: procedure Fast LU Decomposition( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:   for  $k = 0 \dots n - 1$  do
6:      $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$ 
7:      $U_{k+1:,k:} \leftarrow U_{k+1:,k:} - L_{k+1:,k} U_{k,k:}^T$ 
8:   return  $L, U$ 
```

Note that step 7 is an outer product, not the regular dot product ($\mathbf{x}\mathbf{y}^T$ instead of the usual $\mathbf{x}^T\mathbf{y}$). Use `np.outer()` instead of `np.dot()` or `@` to get the desired result.

Pivoting

Gaussian elimination iterates through the rows of a matrix, using the diagonal entry $x_{k,k}$ of the matrix at the k th iteration to zero out all of the entries in the column below $x_{k,k}$ ($x_{i,k}$ for $i \geq k$). This diagonal entry is called the pivot. Unfortunately, Gaussian elimination, and hence the LU decomposition, can be very numerically unstable if at any step the pivot is a very small number. Most professional row reduction algorithms avoid this problem via partial pivoting.

The idea is to choose the largest number (in magnitude) possible to be the pivot by swapping the pivot row² with another row before operating on the matrix. For example, the second and fourth rows of the following matrix are exchanged so that the pivot is -6 instead of 2 .

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix} \xrightarrow{\quad} \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

A row swap is equivalent to left-multiplying by a type II elementary matrix, also called a permutation matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix}$$

For the LU decomposition, if the permutation matrix at step k is P_k , then $P = P_k \dots P_2 P_1$ yields $PA = LU$. The complete algorithm is given below.

²Complete pivoting involves row and column swaps, but doing both operations is usually considered overkill.

Algorithm 6.3

```

1: procedure LU Decomposition with Partial Pivoting( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:    $P \leftarrow [0, 1, \dots, n - 1]$                                  $\triangleright$  See tip 2 below.
6:   for  $k = 0 \dots n - 1$  do
7:     Select  $i \geq k$  that maximizes  $|U_{i,k}|$ 
8:      $U_{k,k} \leftrightarrow U_{i,k}$                                  $\triangleright$  Swap the two rows.
9:      $L_{k,:k} \leftrightarrow L_{i,:k}$                                  $\triangleright$  Swap the two rows.
10:     $P_k \leftrightarrow P_i$                                  $\triangleright$  Swap the two entries.
11:     $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$ 
12:     $U_{k+1:,k} \leftarrow U_{k+1:,k} - L_{k+1:,k} U_{k,k}^T$ 
13:   return  $L, U, P$ 

```

The following tips may be helpful for implementing this algorithm:

1. Since NumPy arrays are mutable, use `np.copy()` to reassign the rows of an array simultaneously.
2. Instead of storing P as an $n \times n$ array, fancy indexing allows us to encode row swaps in a 1-D array of length n . Initialize P as the array $[0, 1, \dots, n]$. After performing a row swap on A , perform the same operations on P . Then the matrix product PA will be the same as $A[P]$.

```

>>> A = np.zeros(3) + np.vstack(np.arange(3))
>>> P = np.arange(3)
>>> print(A)
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

# Swap rows 1 and 2.
>>> A[1], A[2] = np.copy(A[2]), np.copy(A[1])
>>> P[1], P[2] = P[2], P[1]
>>> print(A)                                     # A with the new row arrangement.
[[ 0.  0.  0.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]

>>> print(P)                                     # The permutation of the rows.
[0 2 1]
>>> print(A[P])                                 # A with the original row arrangement.
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

```

There are potential cases where even partial pivoting does not eliminate catastrophic numerical errors in Gaussian elimination, but the odds of having such an amazingly poor matrix are essentially zero. The numerical analyst J.H. Wilkinson captured the likelihood of encountering such a matrix in a natural application when he said, “Anyone that unlucky has already been run over by a bus!”

In Place

The LU decomposition can be performed in place (overwriting the original matrix A) by storing U on and above the main diagonal of the array and storing L below it. The main diagonal of L does not need to be stored since all of its entries are 1. This format saves an entire array of memory, and is how `scipy.linalg.lu_factor()` returns the factorization.

More Applications of the LU Decomposition

The LU decomposition can also be used to compute inverses and determinants with relative efficiency.

- **Inverse:** $(PA)^{-1} = (LU)^{-1} \implies A^{-1}P^{-1} = U^{-1}L^{-1} \implies LUA^{-1} = P$. Solve $LUA_i = \mathbf{p}_i$ with forward and backward substitution (as in Problem 3) for every column \mathbf{p}_i of P . Then

$$A^{-1} = \left[\begin{array}{c|c|c|c} & & & \\ \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{array} \right],$$

the matrix where \mathbf{a}_k is the k th column.

- **Determinant:** $\det(A) = \det(P^{-1}LU) = \frac{\det(L)\det(U)}{\det(P)}$. The determinant of a triangular matrix is the product of its diagonal entries. Since every diagonal entry of L is 1, $\det(L) = 1$. Also, P is just a row permutation of the identity matrix (which has determinant 1), and a single row swap negates the determinant. Then if S is the number of row swaps, the determinant is

$$\det(A) = (-1)^S \prod_{i=1}^n u_{ii}.$$

The Cholesky Decomposition

A square matrix A is called positive definite if $\mathbf{z}^\top A \mathbf{z} > 0$ for all nonzero vectors \mathbf{z} . In addition, A is called Hermitian if $A = A^\mathsf{H} = \overline{A^\top}$. If A is Hermitian positive definite, it has a Cholesky Decomposition $A = U^\mathsf{H}U$ where U is upper triangular with real, positive entries on the diagonal. This is the matrix equivalent to taking the square root of a positive real number.

The Cholesky decomposition takes advantage of the conjugate symmetry of A to simultaneously reduce the columns and rows of A to zeros (except for the diagonal). It thus requires only half of the calculations and memory of the LU decomposition. Furthermore, the algorithm is numerically stable, which means, roughly speaking, that round-off errors do not propagate throughout the computation.

Algorithm 6.4

```

1: procedure Cholesky Decomposition( $A$ )
2:    $n, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{np.triu}(A)$                                       $\triangleright$  Get the upper-triangular part of  $A$ .
4:   for  $i = 0 \dots n - 1$  do
5:     for  $j = i + 1 \dots n - 1$  do
6:        $U_{j,j:} \leftarrow U_{j,j:} - U_{i,j:} \overline{U_{ij}} / U_{ii}$ 
7:        $U_{i,i:} \leftarrow U_{i,i:} / \sqrt{U_{ii}}$ 
8:   return  $U$ 

```

As with the LU decomposition, SciPy's `linalg` module has optimized routines, `la.cho_factor()` and `la.cho_solve()`, for using the Cholesky decomposition.

7

The QR Decomposition

Lab Objective: The QR decomposition is a fundamentally important matrix factorization. It is straightforward to implement, is numerically stable, and provides the basis of several important algorithms. In this lab we explore several ways to produce the QR decomposition and implement a few immediate applications.

The QR decomposition of a matrix A is a factorization $A = QR$, where Q is has orthonormal columns and R is upper triangular. Every $m \times n$ matrix A of rank $n \leq m$ has a QR decomposition, with two main forms.

- **Reduced QR:** Q is $m \times n$, R is $n \times n$, and the columns $\{\mathbf{q}_j\}_{j=1}^n$ of Q form an orthonormal basis for the column space of A .
 - **Full QR:** Q is $m \times m$ and R is $m \times n$. In this case, the columns $\{\mathbf{q}_j\}_{j=1}^m$ of Q form an orthonormal basis for all of \mathbb{F}^m , and the last $m - n$ rows of R only contain zeros. If $m = n$, this is the same as the reduced factorization.

We distinguish between these two forms by writing \widehat{Q} and \widehat{R} for the reduced decomposition and Q and R for the full decomposition.

$$\left[\begin{array}{c|ccccc} & & & & & \\ & & & & & \\ & & & & & \\ \hline & \mathbf{q}_1 & \cdots & \mathbf{q}_n & \mathbf{q}_{n+1} & \cdots & \mathbf{q}_m \\ & & & & & & \\ & & & & & & \\ \hline & & & & & & \\ \end{array} \right] \left[\begin{array}{c|ccc} & & & \\ & r_{11} & \cdots & r_{1n} \\ & \ddots & & \vdots \\ & & & r_{nn} \\ \hline 0 & \cdots & & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & & 0 \\ \hline & & & \\ \end{array} \right] = A \quad (m \times n)$$

QR via Gram-Schmidt

The classical Gram-Schmidt algorithm takes a linearly independent set of vectors and constructs an orthonormal set of vectors with the same span. Applying Gram-Schmidt to the columns of A , which are linearly independent since A has rank n , results in the columns of Q .

Let $\{\mathbf{x}_j\}_{j=1}^n$ be the columns of A . Define

$$\mathbf{q}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|}, \quad \mathbf{q}_k = \frac{\mathbf{x}_k - \mathbf{p}_{k-1}}{\|\mathbf{x}_k - \mathbf{p}_{k-1}\|}, \quad k = 2, \dots, n,$$

$$\mathbf{p}_0 = \mathbf{0}, \quad \mathbf{p}_{k-1} = \sum_{j=1}^{k-1} \langle \mathbf{q}_j, \mathbf{x}_k \rangle \mathbf{q}_j, \quad k = 2, \dots, n.$$

Each \mathbf{p}_{k-1} is the projection of \mathbf{x}_k onto the span of $\{\mathbf{q}_j\}_{j=1}^{k-1}$, so $\mathbf{q}'_k = \mathbf{x}_k - \mathbf{p}_{k-1}$ is the residual vector of the projection. Thus \mathbf{q}'_k is orthogonal to each of the vectors in $\{\mathbf{q}_j\}_{j=1}^{k-1}$. Therefore, normalizing each \mathbf{q}'_k produces an orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$.

To construct the reduced QR decomposition, let \widehat{Q} be the matrix with columns $\{\mathbf{q}_j\}_{j=1}^n$, and let \widehat{R} be the upper triangular matrix with entries

$$r_{kk} = \|\mathbf{x}_k - \mathbf{p}_{k-1}\|, \quad r_{jk} = \langle \mathbf{q}_j, \mathbf{x}_k \rangle = \mathbf{q}_j^\top \mathbf{x}_k, \quad j < k.$$

This clever choice of entries for \widehat{R} reverses the Gram-Schmidt process and ensures that $\widehat{Q}\widehat{R} = A$.

Modified Gram-Schmidt

If the columns of A are close to being linearly dependent, the classical Gram-Schmidt algorithm often produces a set of vectors $\{\mathbf{q}_j\}_{j=1}^n$ that are not even close to orthonormal due to rounding errors. The modified Gram-Schmidt algorithm is a slight variant of the classical algorithm which more consistently produces a set of vectors that are “very close” to orthonormal.

Let \mathbf{q}_1 be the normalization of \mathbf{x}_1 as before. Instead of making just \mathbf{x}_2 orthogonal to \mathbf{q}_1 , make **each** of the vectors $\{\mathbf{x}_j\}_{j=2}^n$ orthogonal to \mathbf{q}_1 :

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_1, \mathbf{x}_k \rangle \mathbf{q}_1, \quad k = 2, \dots, n.$$

Next, define $\mathbf{q}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|}$. Proceed by making each of $\{\mathbf{x}_j\}_{j=3}^n$ orthogonal to \mathbf{q}_2 :

$$\mathbf{x}_k = \mathbf{x}_k - \langle \mathbf{q}_2, \mathbf{x}_k \rangle \mathbf{q}_2, \quad k = 3, \dots, n.$$

Since each of these new vectors is a linear combination of vectors orthogonal to \mathbf{q}_1 , they are orthogonal to \mathbf{q}_1 as well. Continuing this process results in the desired orthonormal set $\{\mathbf{q}_j\}_{j=1}^n$. The entire modified Gram-Schmidt algorithm is described below.

Algorithm 7.1

```

1: procedure Modified Gram-Schmidt( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                       $\triangleright$  Store the dimensions of  $A$ .
3:    $Q \leftarrow \text{copy}(A)$                                           $\triangleright$  Make a copy of  $A$  with np.copy().
4:    $R \leftarrow \text{zeros}(n, n)$                                         $\triangleright$  An  $n \times n$  array of all zeros.
5:   for  $i = 0 \dots n - 1$  do
6:      $R_{i,i} \leftarrow \|Q_{:,i}\|$                                           $\triangleright$  Normalize the  $i$ th column of  $Q$ .
7:      $Q_{:,i} \leftarrow Q_{:,i}/R_{i,i}$ 
8:     for  $j = i + 1 \dots n - 1$  do
9:        $R_{i,j} \leftarrow Q_{:,j}^\top Q_{:,i}$ 
10:       $Q_{:,j} \leftarrow Q_{:,j} - R_{i,j}Q_{:,i}$                                  $\triangleright$  Orthogonalize the  $j$ th column of  $Q$ .
11:    return  $Q, R$ 

```

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n . Use Algorithm 7.1 to compute the reduced QR decomposition of A .

Consider the following tips for implementing the algorithm.

- Use `scipy.linalg.norm()` to compute the norm of the vector in step 6.
- Note that steps 7 and 10 employ scalar multiplication or division, while step 9 uses vector multiplication.

To test your function, generate test cases with NumPy's `np.random` module. Verify that R is upper triangular, Q is orthonormal, and $QR = A$. You may also want to compare your results to SciPy's QR factorization routine, `scipy.linalg.qr()`.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate a random matrix and get its reduced QR decomposition via SciPy.
>>> A = np.random.random((6,4))
>>> Q,R = la.qr(A, mode="economic") # Use mode="economic" for reduced QR.
>>> print(A.shape, Q.shape, R.shape)
(6,4) (6,4) (4,4)

# Verify that R is upper triangular, Q is orthonormal, and QR = A.
>>> np.allclose(np.triu(R), R)
True
>>> np.allclose(Q.T @ Q, np.identity(4))
True
>>> np.allclose(Q @ R, A)
True
```

Consequences of the QR Decomposition

The special structures of Q and R immediately provide some simple applications.

Determinants

Let A be $n \times n$. Then Q and R are both $n \times n$ as well.¹ Since Q is orthonormal and R is upper-triangular,

$$\det(Q) = \pm 1 \quad \text{and} \quad \det(R) = \prod_{i=1}^n r_{i,i}.$$

Then since $\det(AB) = \det(A)\det(B)$,

$$|\det(A)| = |\det(QR)| = |\det(Q)\det(R)| = |\det(Q)| |\det(R)| = \left| \prod_{i=1}^n r_{i,i} \right|. \quad (7.1)$$

¹An $n \times n$ orthonormal matrix is sometimes called unitary in other texts.

Problem 2. Write a function that accepts an invertible matrix A . Use the QR decomposition of A and (7.1) to calculate $|\det(A)|$. You may use your QR decomposition algorithm from Problem 1 or SciPy's QR routine. Can you implement this function in a single line?

(Hint: `np.diag()` and `np.prod()` may be useful.)

Check your answer against `la.det()`, which calculates the determinant.

Linear Systems

The LU decomposition is usually the matrix factorization of choice to solve the linear system $A\mathbf{x} = \mathbf{b}$ because the triangular structures of L and U facilitate forward and backward substitution. However, the QR decomposition avoids the potential numerical issues that come with Gaussian elimination.

Since Q is orthonormal, $Q^{-1} = Q^T$. Therefore, solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving the system $R\mathbf{x} = Q^T\mathbf{b}$. Since R is upper-triangular, $R\mathbf{x} = Q^T\mathbf{b}$ can be solved quickly with back substitution.²

Problem 3. Write a function that accepts an invertible $n \times n$ matrix A and a vector \mathbf{b} of length n . Use the QR decomposition to solve $A\mathbf{x} = \mathbf{b}$ in the following steps:

1. Compute Q and R .
2. Calculate $\mathbf{y} = Q^T\mathbf{b}$.
3. Use back substitution to solve $R\mathbf{x} = \mathbf{y}$ for \mathbf{x} .

QR via Householder

The Gram-Schmidt algorithm orthonormalizes A using a series of transformations that are stored in an upper triangular matrix. Another way to compute the QR decomposition is to take the opposite approach: triangularize A through a series of orthonormal transformations. Orthonormal transformations are numerically stable, meaning that they are less susceptible to rounding errors. In fact, this approach is usually faster and more accurate than Gram-Schmidt methods.

The idea is for the k th orthonormal transformation Q_k to map the k th column of A to the span of $\{\mathbf{e}_j\}_{j=1}^k$, where the \mathbf{e}_j are the standard basis vectors in \mathbb{R}^m . In addition, to preserve the work of the previous transformations, Q_k should not modify any entries of A that are above or to the left of the k th diagonal term of A . For a 4×3 matrix A , the process can be visualized as follows.

$$Q_3 Q_2 Q_1 \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} = Q_3 Q_2 \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix} = Q_3 \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & * \end{bmatrix} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \end{bmatrix}$$

Thus $Q_3 Q_2 Q_1 A = R$, so that $A = Q_1^T Q_2^T Q_3^T R$ since each Q_k is orthonormal. Furthermore, the product of square orthonormal matrices is orthonormal, so setting $Q = Q_1^T Q_2^T Q_3^T$ yields the full QR decomposition.

How to correctly construct each Q_k isn't immediately obvious. The ingenious solution lies in one of the basic types of linear transformations: reflections.

²See the Linear Systems lab for details on back substitution.

Householder Transformations

The orthogonal complement of a nonzero vector $\mathbf{v} \in \mathbb{R}^n$ is the set of all vectors $\mathbf{x} \in \mathbb{R}^n$ that are orthogonal to \mathbf{v} , denoted $\mathbf{v}^\perp = \{\mathbf{x} \in \mathbb{R}^n \mid \langle \mathbf{x}, \mathbf{v} \rangle = 0\}$. A Householder transformation is a linear transformation that reflects a vector \mathbf{x} across the orthogonal complement \mathbf{v}^\perp for some specified \mathbf{v} .

The matrix representation of the Householder transformation corresponding to \mathbf{v} is given by $H_{\mathbf{v}} = I - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top \mathbf{v}}$. Since $H_{\mathbf{v}}^\top H_{\mathbf{v}} = I$, Householder transformations are orthonormal.

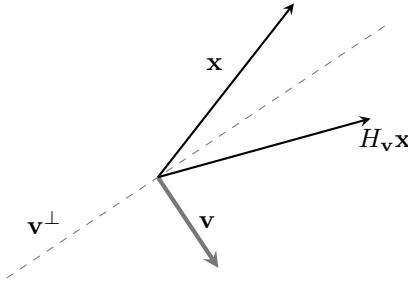


Figure 7.1: The vector \mathbf{v} defines the orthogonal complement \mathbf{v}^\perp , which in this case is a line. Applying the Householder transformation $H_{\mathbf{v}}$ to \mathbf{x} reflects \mathbf{x} across \mathbf{v}^\perp .

Householder Triangularization

The Householder algorithm uses Householder transformations for the orthonormal transformations in the QR decomposition process described on the previous page. The goal in choosing Q_k is to send \mathbf{x}_k , the k th column of A , to the span of $\{\mathbf{e}_j\}_{j=1}^k$. In other words, if $Q_k \mathbf{x}_k = \mathbf{y}_k$, the last $m-k$ entries of \mathbf{y}_k should be 0, i.e.,

$$Q_k \mathbf{x}_k = Q_k \begin{bmatrix} z_1 \\ \vdots \\ z_k \\ z_{k+1} \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{y}_k.$$

To begin, decompose \mathbf{x}_k into $\mathbf{x}_k = \mathbf{x}'_k + \mathbf{x}''_k$, where \mathbf{x}'_k and \mathbf{x}''_k are of the form

$$\mathbf{x}'_k = [z_1 \quad \cdots \quad z_{k-1} \quad 0 \quad \cdots \quad 0]^\top, \quad \mathbf{x}''_k = [0 \quad \cdots \quad 0 \quad z_k \quad \cdots \quad z_m]^\top.$$

Because \mathbf{x}'_k represents elements of A that lie above the diagonal, only \mathbf{x}''_k needs to be altered by the reflection.

The two vectors $\mathbf{x}''_k \pm \|\mathbf{x}''_k\| \mathbf{e}_k$ both yield Householder transformations that send \mathbf{x}''_k to the span of \mathbf{e}_k (see Figure 7.2). Between the two, the one that reflects \mathbf{x}''_k further is more numerically stable. This reflection corresponds to

$$\mathbf{v}_k = \mathbf{x}''_k + \text{sign}(z_k) \|\mathbf{x}''_k\| \mathbf{e}_k,$$

where z_k is the first nonzero component of \mathbf{x}''_k (the k th component of \mathbf{x}_k).

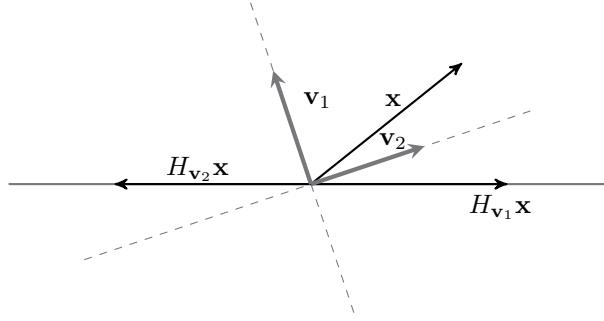


Figure 7.2: There are two reflections that map \mathbf{x} into the span of \mathbf{e}_1 , defined by the vectors \mathbf{v}_1 and \mathbf{v}_2 . In this illustration, $H_{\mathbf{v}_2}$ is the more stable transformation since it reflects \mathbf{x} further than $H_{\mathbf{v}_1}$.

After choosing \mathbf{v}_k , set $\mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}$. Then $H_{\mathbf{v}_k} = I - 2\frac{\mathbf{v}_k\mathbf{v}_k^\top}{\|\mathbf{v}_k\|^2} = I - 2\mathbf{u}_k\mathbf{u}_k^\top$, and hence Q_k is given by the block matrix

$$Q_k = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_{k-1} & \mathbf{0} \\ \mathbf{0} & I_{m-k+1} - 2\mathbf{u}_k\mathbf{u}_k^\top \end{bmatrix}.$$

Here I_p denotes the $p \times p$ identity matrix, and thus each Q_k is $m \times m$.

It is apparent from its form that Q_k does not affect the first $k-1$ rows and columns of any matrix that it acts on. Then by starting with $R = A$ and $Q = I$, at each step of the algorithm we need only multiply the entries in the lower right $(m-k+1) \times (m-k+1)$ submatrices of R and Q by $I - 2\mathbf{u}_k\mathbf{u}_k^\top$. This completes the Householder algorithm, detailed below.

Algorithm 7.2

```

1: procedure Householder( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $R \leftarrow \text{copy}(A)$ 
4:    $Q \leftarrow I_m$                                  $\triangleright$  The  $m \times m$  identity matrix.
5:   for  $k = 0 \dots n-1$  do
6:      $\mathbf{u} \leftarrow \text{copy}(R_{k:,k})$ 
7:      $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$            $\triangleright u_0$  is the first entry of  $\mathbf{u}$ .
8:      $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$                        $\triangleright$  Normalize  $\mathbf{u}$ .
9:      $R_{k:,k} \leftarrow R_{k:,k} - 2\mathbf{u}(\mathbf{u}^\top R_{k:,k})$      $\triangleright$  Apply the reflection to  $R$ .
10:     $Q_{k,:} \leftarrow Q_{k,:} - 2\mathbf{u}(\mathbf{u}^\top Q_{k,:})$          $\triangleright$  Apply the reflection to  $Q$ .
11:   return  $Q^\top, R$ 

```

Problem 4. Write a function that accepts as input a $m \times n$ matrix A of rank n . Use Algorithm 7.2 to compute the full QR decomposition of A .

Consider the following implementation details.

- NumPy's `np.sign()` is an easy way to implement the `sign()` operation in step 7. However, `np.sign(0)` returns 0, which will cause a problem in the rare case that $u_0 = 0$ (which is possible if the top left entry of A is 0 to begin with). The following code defines a function that returns the sign of a single number, counting 0 as positive.

```
sign = lambda x: 1 if x >= 0 else -1
```

- In steps 9 and 10, the multiplication of \mathbf{u} and $(\mathbf{u}^T X)$ is an outer product $(\mathbf{x}\mathbf{y}^T)$ instead of the usual $\mathbf{x}^T \mathbf{y}$). Use `np.outer()` instead of `np.dot()` to handle this correctly.

Use NumPy and SciPy to generate test cases and validate your function.

```
>>> A = np.random.random((5, 3))
>>> Q,R = la.qr(A)                      # Get the full QR decomposition.
>>> print(A.shape, Q.shape, R.shape)
(5,3) (5,5) (5,3)
>>> np.allclose(Q @ R, A)
True
```

Upper Hessenberg Form

An upper Hessenberg matrix is a square matrix that is nearly upper triangular, with zeros below the first subdiagonal. Every $n \times n$ matrix A can be written $A = QHQ^T$ where Q is orthonormal and H , called the Hessenberg form of A , is an upper Hessenberg matrix. Putting a matrix in upper Hessenberg form is an important first step to computing its eigenvalues numerically.

This algorithm also uses Householder transformations. To find orthogonal Q and upper Hessenberg H such that $A = QHQ^T$, it suffices to find such matrices that satisfy $Q^T AQ = H$. Thus, the strategy is to multiply A on the left and right by a series of orthonormal matrices until it is in Hessenberg form.

Using the same Q_k as in the k th step of the Householder algorithm introduces $n - k$ zeros in the k th column of A , but multiplying $Q_k A$ on the right by Q_k^T destroys all of those zeros. Instead, choose a Q_1 that fixes \mathbf{e}_1 and reflects the first column of A into the span of \mathbf{e}_1 and \mathbf{e}_2 . The product $Q_1 A$ then leaves the first row of A alone, and the product $(Q_1 A)Q_1^T$ leaves the first column of $(Q_1 A)$ alone.

$$\begin{array}{c} \left[\begin{array}{cccccc} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{array} \right] \xrightarrow{Q_1} \left[\begin{array}{cccccc} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{array} \right] \xrightarrow{Q_1^T} \left[\begin{array}{cccccc} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{array} \right] \\ A \qquad \qquad \qquad Q_1 A \qquad \qquad \qquad (Q_1 A)Q_1^T \end{array}$$

Continuing the process results in the upper Hessenberg form of A .

$$Q_3 Q_2 Q_1 A Q_1^T Q_2^T Q_3^T = \left[\begin{array}{cccccc} * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{array} \right]$$

This implies that $A = Q_1^T Q_2^T Q_3^T H Q_3 Q_2 Q_1$, so setting $Q = Q_1^T Q_2^T Q_3^T$ results in the desired factorization $A = QHQ^T$.

Constructing the Reflections

Constructing the Q_k uses the same approach as in the Householder algorithm, but shifted down one element. Let $\mathbf{x}_k = \mathbf{y}'_k + \mathbf{y}''_k$ where \mathbf{y}'_k and \mathbf{y}''_k are of the form

$$\mathbf{y}'_k = [z_1 \ \cdots \ z_k \ 0 \ \cdots \ 0]^\top, \quad \mathbf{y}''_k = [0 \ \cdots \ 0 \ z_{k+1} \ \cdots \ z_m]^\top.$$

Because \mathbf{y}'_k represents elements of A that lie above the first subdiagonal, only \mathbf{y}''_k needs to be altered. This suggests using the reflection

$$Q_k = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & H_{\mathbf{v}_k} \end{bmatrix} = \begin{bmatrix} I_k & \mathbf{0} \\ \mathbf{0} & I_{m-k} - 2\mathbf{u}_k\mathbf{u}_k^\top \end{bmatrix}, \text{ where}$$

$$\mathbf{v}_k = \mathbf{y}''_k + \text{sign}(z_k)\|\mathbf{y}''_k\|\mathbf{e}_k, \quad \mathbf{u}_k = \frac{\mathbf{v}_k}{\|\mathbf{v}_k\|}.$$

The complete algorithm is given below. Note how similar it is to Algorithm 7.2.

Algorithm 7.3

```

1: procedure Hessenberg( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $H \leftarrow \text{copy}(A)$ 
4:    $Q \leftarrow I_m$ 
5:   for  $k = 0 \dots n - 3$  do
6:      $\mathbf{u} \leftarrow \text{copy}(H_{k+1:,k})$ 
7:      $u_0 \leftarrow u_0 + \text{sign}(u_0)\|\mathbf{u}\|$ 
8:      $\mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\|$ 
9:      $H_{k+1:,k} \leftarrow H_{k+1:,k} - 2\mathbf{u}(\mathbf{u}^\top H_{k+1:,k})$   $\triangleright$  Apply  $Q_k$  to  $H$ .
10:     $H_{:,k+1} \leftarrow H_{:,k+1} - 2(H_{:,k+1} \mathbf{u})\mathbf{u}^\top$   $\triangleright$  Apply  $Q_k^\top$  to  $H$ .
11:     $Q_{k+1,:} \leftarrow Q_{k+1,:} - 2\mathbf{u}(\mathbf{u}^\top Q_{k+1,:})$   $\triangleright$  Apply  $Q_k$  to  $Q$ .
12:   return  $H, Q^\top$ 

```

Problem 5. Write a function that accepts a nonsingular $n \times n$ matrix A . Use Algorithm 7.3 to compute the upper Hessenberg H and orthogonal Q satisfying $A = QHQ^\top$.

Compare your results to `scipy.linalg.hessenberg()`.

```

# Generate a random matrix and get its upper Hessenberg form via SciPy.
>>> A = np.random.random((8,8))
>>> H, Q = la.hessenberg(A, calc_q=True)

# Verify that H has all zeros below the first subdiagonal and QHQ^T = A.
>>> np.allclose(np.triu(H, -1), H)
True
>>> np.allclose(Q @ H @ Q.T, A)
True

```

Additional Material

Complex QR Decomposition

The QR decomposition also exists for matrices with complex entries. The standard inner product in \mathbb{R}^m is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y}$, but the (more general) standard inner product in \mathbb{C}^m is $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^H \mathbf{y}$. The H stands for the Hermitian conjugate, the conjugate of the transpose. Making a few small adjustments in the implementations of Algorithms 7.1 and 7.2 accounts for using the complex inner product.

1. Replace any transpose operations with the conjugate of the transpose.

```
>>> A = np.reshape(np.arange(4) + 1j*np.arange(4), (2,2))
>>> print(A)
[[ 0.+0.j  1.+1.j]
 [ 2.+2.j  3.+3.j]]

>>> print(A.T)                                     # Regular transpose.
[[ 0.+0.j  2.+2.j]
 [ 1.+1.j  3.+3.j]]

>>> print(A.conj().T)                            # Hermitian conjugate.
[[ 0.-0.j  2.-2.j]
 [ 1.-1.j  3.-3.j]]
```

2. Conjugate the first entry of vector or matrix multiplication before multiplying with `np.dot()`.

```
>>> x = np.arange(2) + 1j*np.arange(2)
>>> print(x)
[ 0.+0.j  1.+1.j]

>>> np.dot(x, x)                                # Standard real inner product.
2j

>>> np.dot(x.conj(), y)                         # Standard complex inner product.
(2 + 0j)
```

3. In the complex plane, there are infinitely many reflections that map a vector \mathbf{x} into the span of \mathbf{e}_k , not just the two displayed in Figure 7.2. Using $\text{sign}(z_k)$ to choose one is still a valid method, but it requires updating the `sign()` function so that it can handle complex numbers.

```
sign = lambda x: x/np.abs(x) if x!=0 else 1
```

QR with Pivoting

The LU decomposition can be improved by employing Gaussian elimination with partial pivoting, where the rows of A are strategically permuted at each iteration. The QR factorization can be similarly improved by permuting the columns of A at each iteration. The result is the factorization $AP = QR$, where P is a permutation matrix that encodes the column swaps. To compute the pivoted QR decomposition with `scipy.linalg.qr()`, set the keyword `pivoting` to `True`.

```
# Get the decomposition AP = QR for a random matrix A.
>>> A = np.random.random((8,10))
>>> Q,R,P = la.qr(A, pivoting=True)

# P is returned as a 1-D array that encodes column ordering,
# so A can be reconstructed with fancy indexing.
>>> np.allclose(Q @ R, A[:,P])
True
```

QR via Givens

The Householder algorithm uses reflections to triangularize A . However, A can also be made upper triangular using rotations. To illustrate the idea, recall that the matrix for a counterclockwise rotation of θ radians is given by

$$R_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}.$$

This transformation is orthonormal. Given $\mathbf{x} = [a, b]^\top$, if θ is the angle between \mathbf{x} and \mathbf{e}_1 , then $R_{-\theta}$ maps \mathbf{x} to the span of \mathbf{e}_1 .

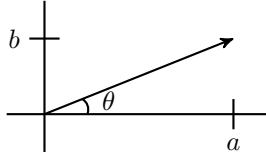


Figure 7.3: Rotating clockwise by θ sends the vector $[a, b]^\top$ to the span of \mathbf{e}_1 .

In terms of a and b , $\cos \theta = \frac{a}{\sqrt{a^2+b^2}}$ and $\sin \theta = \frac{b}{\sqrt{a^2+b^2}}$. Therefore,

$$R_{-\theta} \mathbf{x} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \frac{a}{\sqrt{a^2+b^2}} & \frac{b}{\sqrt{a^2+b^2}} \\ -\frac{b}{\sqrt{a^2+b^2}} & \frac{a}{\sqrt{a^2+b^2}} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sqrt{a^2+b^2} \\ 0 \end{bmatrix}.$$

The matrix R_θ above is an example of a 2×2 Givens rotation matrix. In general, the Givens matrix $G(i, j, \theta)$ represents the orthonormal transformation that rotates the 2-dimensional span of \mathbf{e}_i and \mathbf{e}_j by θ radians. The matrix representation of this transformation is a generalization of R_θ .

$$G(i, j, \theta) = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & I & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix}$$

Here I represents the identity matrix, $c = \cos \theta$, and $s = \sin \theta$. The c 's appear on the i th and j th diagonal entries.

Givens Triangularization

As demonstrated, θ can be chosen such that $G(i, j, \theta)$ rotates a vector so that its j th-component is 0. Such a transformation will only affect the i th and j th entries of any vector it acts on (and thus the i th and j th rows of any matrix it acts on).

To compute the QR decomposition of A , iterate through the subdiagonal entries of A in the order depicted by Figure 7.4. Zero out the ij th entry with a rotation in the plane spanned by \mathbf{e}_{i-1} and \mathbf{e}_i , represented by the Givens matrix $G(i-1, i, \theta)$.

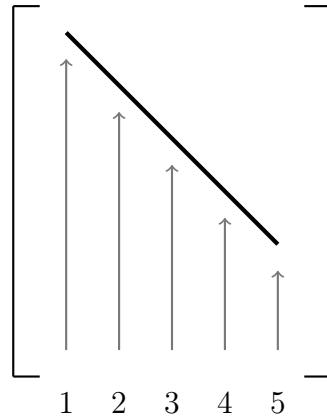


Figure 7.4: The order in which to zero out subdiagonal entries in the Givens triangularization algorithm. The heavy black line is the main diagonal of the matrix. Entries should be zeroed out from bottom to top in each column, beginning with the leftmost column.

On a 2×3 matrix, the process can be visualized as follows.

$$\left[\begin{array}{cc} * & * \\ * & * \\ * & * \end{array} \right] \xrightarrow{G(2,3,\theta_1)} \left[\begin{array}{cc} * & * \\ * & * \\ 0 & * \end{array} \right] \xrightarrow{G(1,2,\theta_2)} \left[\begin{array}{cc} * & * \\ 0 & * \\ 0 & * \end{array} \right] \xrightarrow{G(2,3,\theta_3)} \left[\begin{array}{cc} * & * \\ 0 & * \\ 0 & 0 \end{array} \right]$$

At each stage, the boxed entries are those modified by the previous transformation. The final transformation $G(2,3,\theta_3)$ operates on the bottom two rows, but since the first two entries are zero, they are unaffected.

Assuming that at the ij th stage of the algorithm a_{ij} is nonzero, Algorithm 7.4 computes the Givens triangularization of a matrix. Notice that the algorithm does not actually form the entire matrices $G(i, j, \theta)$; instead, it modifies only those entries of the matrix that are affected by the transformation.

Algorithm 7.4

```

1: procedure Givens Triangularization( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $R \leftarrow \text{copy}(A)$ 
4:    $Q \leftarrow I_m$ 
5:   for  $j = 0 \dots n - 1$  do
6:     for  $i = m - 1 \dots j + 1$  do
7:        $a, b \leftarrow R_{i-1,j}, R_{i,j}$ 
8:        $G \leftarrow [[a, b], [-b, a]] / \sqrt{a^2 + b^2}$ 
9:        $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$ 
10:       $Q_{i-1:i+1,:} \leftarrow GQ_{i-1:i+1,:}$ 
11:   return  $Q^T, R$ 

```

QR of a Hessenberg Matrix via Givens

The Givens algorithm is particularly efficient for computing the QR decomposition of a matrix that is already in upper Hessenberg form, since only the first subdiagonal needs to be zeroed out. Algorithm 7.5 details this process.

Algorithm 7.5

```

1: procedure Givens Triangularization of Hessenberg( $H$ )
2:    $m, n \leftarrow \text{shape}(H)$ 
3:    $R \leftarrow \text{copy}(H)$ 
4:    $Q \leftarrow I_m$ 
5:   for  $j = 0 \dots \min\{n - 1, m - 1\}$  do
6:      $i = j + 1$ 
7:      $a, b \leftarrow R_{i-1,j}, R_{i,j}$ 
8:      $G \leftarrow [[a, b], [-b, a]] / \sqrt{a^2 + b^2}$ 
9:      $R_{i-1:i+1,j:} \leftarrow GR_{i-1:i+1,j:}$ 
10:     $Q_{i-1:i+1,:i+1} \leftarrow GQ_{i-1:i+1,:i+1}$ 
11:   return  $Q^T, R$ 

```

Note

When A is symmetric, its upper Hessenberg form is a tridiagonal matrix, meaning its only nonzero entries are on the main diagonal, the first subdiagonal, and the first superdiagonal. This is because the Q_k 's zero out everything below the first subdiagonal of A and the Q_k^T 's zero out everything to the right of the first superdiagonal. Tridiagonal matrices make computations fast, so computing the Hessenberg form of a symmetric matrix is very useful.

8

Least Squares and Computing Eigenvalues

Lab Objective: Because of its numerical stability and convenient structure, the QR decomposition is the basis of many important and practical algorithms. In this lab we introduce linear least squares problems, tools in Python for computing least squares solutions, and two fundamental algorithms for computing eigenvalue. The QR decomposition makes solving several of these problems quick and numerically stable.

Least Squares

A linear system $A\mathbf{x} = \mathbf{b}$ is overdetermined if it has more equations than unknowns. In this situation, there is no true solution, and \mathbf{x} can only be approximated.

The least squares solution of $A\mathbf{x} = \mathbf{b}$, denoted $\hat{\mathbf{x}}$, is the “closest” vector to a solution, meaning it minimizes the quantity $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$. In other words, $\hat{\mathbf{x}}$ is the vector such that $A\hat{\mathbf{x}}$ is the projection of \mathbf{b} onto the range of A , and can be calculated by solving the normal equations,¹

$$A^\top A\hat{\mathbf{x}} = A^\top \mathbf{b}.$$

If A is full rank (which it usually is in applications) its QR decomposition provides an efficient way to solve the normal equations. Let $A = \widehat{Q}\widehat{R}$ be the reduced QR decomposition of A , so \widehat{Q} is $m \times n$ with orthonormal columns and \widehat{R} is $n \times n$, invertible, and upper triangular. Since $\widehat{Q}^\top \widehat{Q} = I$, and since \widehat{R}^\top is invertible, the normal equations can be reduced as follows (we omit the hats on \widehat{Q} and \widehat{R} for clarity).

$$\begin{aligned} A^\top A\hat{\mathbf{x}} &= A^\top \mathbf{b} \\ (QR)^\top QR\hat{\mathbf{x}} &= (QR)^\top \mathbf{b} \\ R^\top Q^\top QR\hat{\mathbf{x}} &= R^\top Q^\top \mathbf{b} \\ R^\top R\hat{\mathbf{x}} &= R^\top Q^\top \mathbf{b} \\ R\hat{\mathbf{x}} &= Q^\top \mathbf{b} \end{aligned} \tag{8.1}$$

Thus $\hat{\mathbf{x}}$ is the least squares solution to $A\mathbf{x} = \mathbf{b}$ if and only if $R\hat{\mathbf{x}} = Q^\top \mathbf{b}$. Since R is upper triangular, this equation can be solved quickly with back substitution.

¹See Volume 1 for a formal derivation of the normal equations.

Problem 1. Write a function that accepts an $m \times n$ matrix A of rank n and a vector \mathbf{b} of length m . Use the reduced QR decomposition of A and (8.1) to solve the normal equations corresponding to $A\mathbf{x} = \mathbf{b}$.

You may use either SciPy's reduced QR routine (`la.qr()` with `mode="economic"`) or one of your own reduced QR routines. In addition, you may use `la.solve_triangular()`, SciPy's optimized routine for solving triangular systems.

Fitting a Line

The least squares solution can be used to find the best fit curve of a chosen type to a set of points. Consider the problem of finding the line $y = ax + b$ that best fits a set of m points $\{(x_k, y_k)\}_{k=1}^m$. Ideally, we seek a and b such that $y_k = ax_k + b$ for all k . These equations can be simultaneously represented by the linear system

$$A\mathbf{x} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \quad (8.2)$$

Note that A has full column rank as long as not all of the x_k values are the same.

Because this system has two unknowns, it is guaranteed to have a solution if it has two or fewer equations. However, if there are more than two data points, the system is overdetermined if any set of three points is not collinear. We therefore seek a least squares solution, which in this case means finding the slope \hat{a} and y -intercept \hat{b} such that the line $y = \hat{a}x + \hat{b}$ best fits the data.

Figure 8.1 is a typical example of this idea where $\hat{a} \approx \frac{1}{2}$ and $\hat{b} \approx -3$.

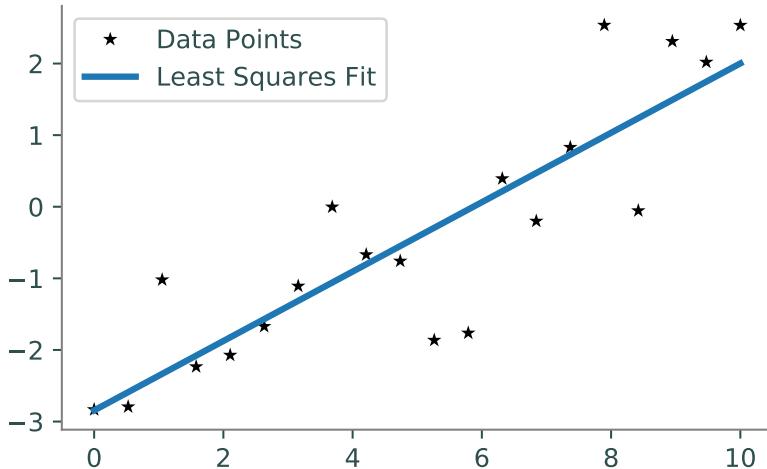


Figure 8.1: A linear least squares fit.

Problem 2. The file `housing.npy` contains the purchase-only housing price index, a measure of how housing prices are changing, for the United States from 2000 to 2010.^a Each row in the array is a separate measurement; the columns are the year and the price index, in that order. To avoid large numerical computations, the year measurements start at 0 instead of 2000.

Find the least squares line that relates the year to the housing price index (i.e., let year be the x -axis and index the y -axis).

1. Construct the matrix A and the vector \mathbf{b} described by (8.2).
(Hint: `np.vstack()`, `np.column_stack()`, and/or `np.ones()` may be helpful.)
2. Use your function from Problem 1 to find the least squares solution.
3. Plot the data points as a scatter plot.
4. Plot the least squares line with the scatter plot.

^aSee <http://www.fhfa.gov/DataTools/Downloads/Pages/House-Price-Index.aspx>.

Note

The least squares problem of fitting a line to a set of points is often called linear regression, and the resulting line is called the linear regression line. SciPy's specialized tool for linear regression is `scipy.stats.linregress()`. This function takes in an array of x -coordinates and a corresponding array of y -coordinates, and returns the slope and intercept of the regression line, along with a few other statistical measurements.

For example, the following code produces Figure 8.1.

```
>>> import numpy as np
>>> from scipy.stats import linregress

# Generate some random data close to the line y = .5x - 3.
>>> x = np.linspace(0, 10, 20)
>>> y = .5*x - 3 + np.random.randn(20)

# Use linregress() to calculate m and b, as well as the correlation
# coefficient, p-value, and standard error. See the documentation for
# details on each of these extra return values.
>>> a, b, rvalue, pvalue, stderr = linregress(x, y)

>>> plt.plot(x, y, 'k*', label="Data Points")
>>> plt.plot(x, a*x + b, label="Least Squares Fit")
>>> plt.legend(loc="upper left")
>>> plt.show()
```

Fitting a Polynomial

Least squares can also be used to fit a set of data to the best fit polynomial of a specified degree. Let $\{(x_k, y_k)\}_{k=1}^m$ be the set of m data points in question. The general form for a polynomial of degree n is

$$p_n(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_2 x^2 + c_1 x + c_0 = \sum_{i=0}^n c_i x^i.$$

Note that the polynomial is uniquely determined by its $n+1$ coefficients $\{c_i\}_{i=0}^n$. Ideally, then, we seek the set of coefficients $\{c_i\}_{i=0}^n$ such that

$$y_k = c_n x_k^n + c_{n-1} x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all values of k . These m linear equations yield the linear system

$$Ax = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = b. \quad (8.3)$$

If $m > n+1$ this system is overdetermined, requiring a least squares solution.

Working with Polynomials in NumPy

The $m \times (n+1)$ matrix A of (8.3) is called a Vandermonde matrix.² NumPy's `np.vander()` is a convenient tool for quickly constructing a Vandermonde matrix, given the values $\{x_k\}_{k=1}^m$ and the number of desired columns.

```
>>> print(np.vander([2, 3, 5], 2))
[[2 1]                                     # [[2**1, 2**0]
 [3 1]                                     # [3**1, 3**0]
 [5 1]]                                    # [5**1, 5**0]]

>>> print(np.vander([2, 3, 5, 4], 3))
[[ 4  2  1]                                # [[2**2, 2**1, 2**0]
 [ 9  3  1]                                # [3**2, 3**1, 3**0]
 [25  5  1]                                # [5**2, 5**1, 5**0]
 [16  4  1]]                               # [4**2, 4**1, 4**0]
```

NumPy also has powerful tools for working efficiently with polynomials. The class `np.poly1d` represents a 1-dimensional polynomial. Instances of this class are callable like a function.³ The constructor accepts the polynomial's coefficients, from largest degree to smallest.

Table 8.1 lists some attributes and methods of the `np.poly1d` class.

²Vandermonde matrices have many special properties and are useful for many applications, including polynomial interpolation and discrete Fourier analysis.

³Class instances can be made callable by implementing the `__call__()` magic method.

Attribute	Description
<code>coeffs</code>	The $n + 1$ coefficients, from greatest degree to least.
<code>order</code>	The polynomial degree (n).
<code>roots</code>	The n roots of the polynomial.
Method	Returns
<code>deriv()</code>	The coefficients of the polynomial after being differentiated.
<code>integ()</code>	The coefficients of the polynomial after being integrated (with $c_0 = 0$).

Table 8.1: Attributes and methods of the `np.poly1d` class.

```
# Create a callable object for the polynomial f(x) = (x-1)(x-2) = x^2 - 3x + 2.
>>> f = np.poly1d([1, -3, 2])
>>> print(f)
      2
1 x - 3 x + 2

# Evaluate f(x) for several values of x in a single function call.
>>> f([1, 2, 3, 4])
array([0, 0, 2, 6])
```

Problem 3. The data in `housing.npy` is nonlinear, and might be better fit by a polynomial than a line.

Write a function that uses (8.3) to calculate the polynomials of degree 3, 6, 9, and 12 that best fit the data. Plot the original data points and each least squares polynomial together in individual subplots.

(Hint: define a separate, refined domain with `np.linspace()` and use this domain to smoothly plot the polynomials.)

Instead of using Problem 1 to solve the normal equations, you may use SciPy's least squares routine, `scipy.linalg.lstsq()`.

```
>>> from scipy import linalg as la

# Define A and b appropriately.

# Solve the normal equations using SciPy's least squares routine.
# The least squares solution is the first of four return values.
>>> x = la.lstsq(A, b)[0]
```

Compare your results to `np.polyfit()`. This function receives an array of x values, an array of y values, and an integer for the polynomial degree, and returns the coefficients of the best fit polynomial of that degree.

Achtung!

Having more parameters in a least squares model is not always better. For a set of m points, the best fit polynomial of degree $m - 1$ interpolates the data set, meaning that $p(x_k) = y_k$ exactly for each k . In this case there are enough unknowns that the system is no longer overdetermined. However, such polynomials are highly subject to numerical errors and are unlikely to accurately represent true patterns in the data.

Choosing to have too many unknowns in a fitting problem is (fittingly) called overfitting, and is an important issue to avoid in any statistical model.

Fitting a Circle

Suppose the set of m points $\{(x_k, y_k)\}_{k=1}^m$ are arranged in a nearly circular pattern. The general equation of a circle with radius r and center (c_1, c_2) is

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \quad (8.4)$$

The circle is uniquely determined by r , c_1 , and c_2 , so these are the parameters that should be solved for in a least squares formulation of the problem. However, (8.4) is not linear in any of these variables.

$$\begin{aligned} (x - c_1)^2 + (y - c_2)^2 &= r^2 \\ x^2 - 2c_1x + c_1^2 + y^2 - 2c_2y + c_2^2 &= r^2 \\ x^2 + y^2 &= 2c_1x + 2c_2y + r^2 - c_1^2 - c_2^2 \end{aligned} \quad (8.5)$$

The quadratic terms x^2 and y^2 are acceptable because the points $\{(x_k, y_k)\}_{k=1}^m$ are given. To eliminate the nonlinear terms in the unknown parameters r , c_1 , and c_2 , define a new variable $c_3 = r^2 - c_1^2 - c_2^2$. Then for each point (x_k, y_k) , (8.5) becomes

$$2c_1x_k + 2c_2y_k + c_3 = x_k^2 + y_k^2.$$

These m equations are linear in c_1 , c_2 , and c_3 , and can be written as the linear system

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_m & 2y_m & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_m^2 + y_m^2 \end{bmatrix}. \quad (8.6)$$

After solving for the least squares solution, r can be recovered with the relation $r = \sqrt{c_1^2 + c_2^2 + c_3}$. Finally, plotting a circle is best done with polar coordinates. Using the same variables as before, the circle can be represented in polar coordinates by setting

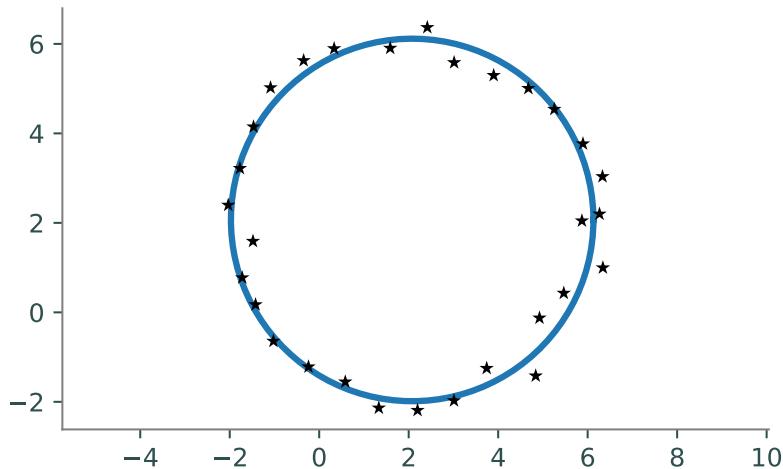
$$x = r \cos(\theta) + c_1, \quad y = r \sin(\theta) + c_2, \quad \theta \in [0, 2\pi]. \quad (8.7)$$

To plot the circle, solve the least squares system for c_1 , c_2 , and r , define an array for θ , then use (8.7) to calculate the coordinates of the points the circle.

```
# Load some data and construct the matrix A and the vector b.
>>> xk, yk = np.load("circle.npy").T
>>> A = np.column_stack((2*xk, 2*yk, np.ones_like(xk)))
>>> b = xk**2 + yk**2

# Calculate the least squares solution and solve for the radius.
>>> c1, c2, c3 = la.lstsq(A, b)[0]
>>> r = np.sqrt(c1**2 + c2**2 + c3)

# Plot the circle using polar coordinates.
>>> theta = np.linspace(0, 2*np.pi, 200)
>>> x = r*np.cos(theta) + c1
>>> y = r*np.sin(theta) + c2
>>> plt.plot(x, y) # Plot the circle.
>>> plt.plot(xk, yk, 'k*') # Plot the data points.
>>> plt.axis("equal")
```



Problem 4. The general equation for an ellipse is

$$ax^2 + bx + cxy + dy + ey^2 = 1.$$

Write a function that calculates the parameters for the ellipse that best fits the data in the file `ellipse.npy`. Plot the original data points and the ellipse together, using the following function to plot the ellipse.

```
def plot_ellipse(a, b, c, d, e):
    """Plot an ellipse of the form ax^2 + bx + cxy + dy + ey^2 = 1."""
    theta = np.linspace(0, 2*np.pi, 200)
    cos_t, sin_t = np.cos(theta), np.sin(theta)
```

```

A = a*(cos_t**2) + c*cos_t*sin_t + e*(sin_t**2)
B = b*cos_t + d*sin_t
r = (-B + np.sqrt(B**2 + 4*A)) / (2*A)
plt.plot(r*cos_t, r*sin_t, lw=2)
plt.gca().set_aspect("equal", "datalim")

```

Computing Eigenvalues

The eigenvalues of an $n \times n$ matrix A are the roots of its characteristic polynomial $\det(A - \lambda I)$. Thus, finding the eigenvalues of A amounts to computing the roots of a polynomial of degree n . However, for $n \geq 5$, it is provably impossible to find an algebraic closed-form solution to this problem.⁴ In addition, numerically computing the roots of a polynomial is a famously ill-conditioned problem, meaning that small changes in the coefficients of the polynomial (brought about by small changes in the entries of A) may yield wildly different results. Instead, eigenvalues must be computed with iterative methods.

The Power Method

The dominant eigenvalue of the $n \times n$ matrix A is the unique eigenvalue of greatest magnitude, if such an eigenvalue exists. The power method iteratively computes the dominant eigenvalue of A and its corresponding eigenvector.

Begin by choosing a vector \mathbf{x}_0 such that $\|\mathbf{x}_0\|_2 = 1$, and define

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{\|A\mathbf{x}_k\|_2}.$$

If A has a dominant eigenvalue λ , and if the projection of \mathbf{x}_0 onto the subspace spanned by the eigenvectors corresponding to λ is nonzero, then the sequence of vectors $(\mathbf{x}_k)_{k=0}^{\infty}$ converges to an eigenvector \mathbf{x} of A corresponding to λ .

Since \mathbf{x} is an eigenvector of A , $A\mathbf{x} = \lambda\mathbf{x}$. Left multiplying by \mathbf{x}^T on each side results in $\mathbf{x}^T A \mathbf{x} = \lambda \mathbf{x}^T \mathbf{x}$, and hence $\lambda = \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$. This ratio is called the Rayleigh quotient. However, since each \mathbf{x}_k is normalized, $\mathbf{x}^T \mathbf{x} = \|\mathbf{x}\|_2^2 = 1$, so $\lambda = \mathbf{x}^T A \mathbf{x}$.

The entire algorithm is summarized below.

Algorithm 8.1

```

1: procedure PowerMethod( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                       $\triangleright A$  is square so  $m = n$ .
3:    $\mathbf{x}_0 \leftarrow \text{random}(n)$                                  $\triangleright$  A random vector of length  $n$ 
4:    $\mathbf{x}_0 \leftarrow \mathbf{x}_0 / \|\mathbf{x}_0\|_2$                           $\triangleright$  Normalize  $\mathbf{x}_0$ 
5:   for  $k = 0, 1, \dots, N - 1$  do
6:      $\mathbf{x}_{k+1} \leftarrow A\mathbf{x}_k$ 
7:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1} / \|\mathbf{x}_{k+1}\|_2$ 
8:   return  $\mathbf{x}_N^T A \mathbf{x}_N, \mathbf{x}_N$ 

```

⁴This result, called Abel's impossibility theorem, was first proven by Niels Heinrik Abel in 1824.

The power method is limited by a few assumptions. First, not all square matrices A have a dominant eigenvalue. However, the Perron-Frobenius theorem guarantees that if all entries of A are positive, then A has a dominant eigenvalue. Second, there is no way to choose an \mathbf{x}_0 that is guaranteed to have a nonzero projection onto the span of the eigenvectors corresponding to λ , though a random \mathbf{x}_0 will almost surely satisfy this condition. Even with these assumptions, a rigorous proof that the power method converges is most convenient with tools from spectral calculus, and as such will not be pursued here.

Problem 5. Write a function that accepts an $n \times n$ matrix A , a maximum number of iterations N , and a stopping tolerance tol . Use Algorithm 8.1 to compute the dominant eigenvalue of A and a corresponding eigenvector. Continue the loop in step 5 until either $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$ is less than the tolerance tol , or until iterating the maximum number of times N .

Test your function on square matrices with all positive entries, verifying that $A\mathbf{x} = \lambda\mathbf{x}$. Use SciPy's eigenvalue solver, `scipy.linalg.eig()`, to compute all of the eigenvalues and corresponding eigenvectors of A and check that λ is the dominant eigenvalue of A .

```
# Construct a random matrix with positive entries.
>>> A = np.random.random((10,10))

# Compute the eigenvalues and eigenvectors of A via SciPy.
>>> eigs, vecs = la.eig(A)

# Get the dominant eigenvalue and eigenvector of A.
# The eigenvector of the kth eigenvalue is the kth column of 'vecs'.
>>> loc = np.argmax(eigs)
>>> lamb, x = eigs[loc], vecs[:,loc]

# Verify that Ax = lambda x.
>>> np.allclose(A @ x, lamb * x)
True
```

The QR Algorithm

An obvious shortcoming of the power method is that it only computes one eigenvalue and eigenvector. The QR algorithm, on the other hand, attempts to find all eigenvalues of A .

Let $A_0 = A$, and for arbitrary k let $Q_k R_k = A_k$ be the QR decomposition of A_k . Since A is square, so are Q_k and R_k , so they can be recombined in reverse order:

$$A_{k+1} = R_k Q_k.$$

This recursive definition establishes an important relation between the A_k :

$$Q_k^{-1} A_k Q_k = Q_k^{-1} (Q_k R_k) Q_k = (Q_k^{-1} Q_k) (R_k Q_k) = R_k.$$

Thus, A_k is orthonormally similar to A_{k+1} , and similar matrices have the same eigenvalues. The series of matrices $(A_k)_{k=0}^{\infty}$ converges to the block matrix

$$S = \begin{bmatrix} S_1 & * & \cdots & * \\ \mathbf{0} & S_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & * \\ \mathbf{0} & \cdots & \mathbf{0} & S_m \end{bmatrix}. \quad \text{For example, } S = \begin{bmatrix} s_1 & * & * & \cdots & * \\ 0 & s_{2,1} & s_{2,2} & \cdots & * \\ & s_{2,3} & s_{2,4} & \cdots & * \\ & & \ddots & \vdots & \\ & & & & s_m \end{bmatrix}.$$

Each S_i is either a 1×1 or 2×2 matrix.⁵ In the example above on the right, since the first subdiagonal entry is zero, S_1 is the 1×1 matrix with a single entry, s_1 . But as $s_{2,3}$ is not zero, S_2 is 2×2 .

Since S is block upper triangular, its eigenvalues are the eigenvalues of its diagonal S_i blocks. Then because A is similar to each A_k , those eigenvalues of S are the eigenvalues of A .

When A has real entries but complex eigenvalues, 2×2 S_i blocks appear in S . Finding eigenvalues of a 2×2 matrix is equivalent to finding the roots of a 2nd degree polynomial,

$$\det(S_i - \lambda I) = \begin{vmatrix} a - \lambda & b \\ c & d - \lambda \end{vmatrix} = (a - \lambda)(d - \lambda) - bc = \lambda^2 - (a + d)\lambda + (ad - bc), \quad (8.8)$$

which has a closed form solution via the quadratic equation. This also demonstrates that complex eigenvalues come in conjugate pairs.

Hessenberg Preconditioning

A matrix in upper Hessenberg form is one that has all entries below the first subdiagonal equal to zero. This is similar to an upper triangular matrix, except that the entries directly below the diagonal are also allowed to be nonzero. The QR algorithm works more accurately and efficiently on matrices that are in upper Hessenberg form, as upper Hessenberg matrices are already close to triangular. Furthermore, if $H = QR$ is the QR decomposition of upper Hessenberg H then RQ is also upper Hessenberg, so the almost-triangular form is preserved at each iteration. Putting a matrix in upper Hessenberg form before applying the QR algorithm is called Hessenberg preconditioning.

With preconditioning in mind, the entire QR algorithm is as follows.

⁵If all of the S_i are 1×1 matrices, then the upper triangular S is called the Schur form of A . If some of the S_i are 2×2 matrices, then S is called the real Schur form of A .

Algorithm 8.2

```

1: procedure QR_Algorithm( $A, N$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $S \leftarrow \text{hessenberg}(A)$                                  $\triangleright$  Put  $A$  in upper Hessenberg form.
4:   for  $k = 0, 1, \dots, N - 1$  do
5:      $Q, R \leftarrow S$                                           $\triangleright$  Get the QR decomposition of  $A_k$ .
6:      $S \leftarrow RQ$                                           $\triangleright$  Recombine  $R_k$  and  $Q_k$  into  $A_{k+1}$ .
7:     eigs  $\leftarrow []$                                       $\triangleright$  Initialize an empty list of eigenvalues.
8:      $i \leftarrow 0$ 
9:     while  $i < n$  do
10:       if  $S_i$  is  $1 \times 1$  then
11:         Append the only entry  $s_i$  of  $S_i$  to eigs
12:       else if  $S_i$  is  $2 \times 2$  then
13:         Calculate the eigenvalues of  $S_i$ 
14:         Append the eigenvalues of  $S_i$  to eigs
15:          $i \leftarrow i + 1$ 
16:        $i \leftarrow i + 1$                                           $\triangleright$  Move to the next  $S_i$ .
17:   return eigs

```

Problem 6. Write a function that accepts an $n \times n$ matrix A , a number of iterations N , and a tolerance tol . Use Algorithm 8.2 to implement the QR algorithm with Hessenberg preconditioning, returning the eigenvalues of A .

Consider the following implementation details.

- Use `scipy.linalg.hessenberg()` or your own Hessenburg algorithm to reduce A to upper Hessenberg form in step 3.
- The loop in step 4 should run for N total iterations.
- Use `scipy.linalg.qr()` or one of your own QR factorization routines to compute the QR decomposition of S in step 5. Note that since S is in upper Hessenberg form, Givens rotations are the most efficient way to produce Q and R .
- Assume that S_i is 1×1 in step 10 if one of two following criteria hold:
 - S_i is the last diagonal entry of S .
 - The absolute value of element below the i th main diagonal entry of S (the lower left element of the 2×2 block) is less than tol .
- If S_i is 2×2 , use the quadratic formula and (8.8) to compute its eigenvalues. Use the function `cmath.sqrt()` to correctly compute the square root of a negative number.

Test your function on small random symmetric matrices, comparing your results to SciPy's `scipy.linalg.eig()`. While the QR algorithm works on arbitrary matrices, it has better convergence properties for symmetric matrices, which makes them better for testing. To construct a random symmetric matrix, note that $A + A^T$ is always symmetric.

Note

Algorithm 8.2 is theoretically sound, but can still be greatly improved. Most modern computer packages instead use the implicit QR algorithm, an improved version of the QR algorithm, to compute eigenvalues.

For large matrices, there are other iterative methods besides the power method and the QR algorithm for efficiently computing eigenvalues. They include the Arnoldi iteration, the Jacobi method, the Rayleigh quotient method, and others.

Additional Material

Variations on the Linear Least Squares Problem

If W is an $n \times n$ is symmetric positive-definite matrix, then the function $\|\cdot\|_{W^2} : \mathbb{R}^n \rightarrow \mathbb{R}$ given by

$$\|\mathbf{x}\|_{W^2} = \|W\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top W^\top W \mathbf{x}}$$

defines a norm and is called a weighted 2-norm. Given the overdetermined system $A\mathbf{x} = \mathbf{b}$, the problem of choosing $\hat{\mathbf{x}}$ to minimize $\|A\hat{\mathbf{x}} - \mathbf{b}\|_{W^2}$ is called a weighted least squares (WLS) problem. This problem has a slightly different set of normal equations,

$$A^\top W^\top W A \hat{\mathbf{x}} = A^\top W^\top W \mathbf{b}.$$

However, letting $C = WA$ and $\mathbf{z} = W\mathbf{b}$, this equation reduces to the usual normal equations,

$$C^\top C \hat{\mathbf{x}} = C^\top \mathbf{z},$$

so a WLS problem can be solved in the same way as an ordinary least squares (OLS) problem.

Weighted least squares is useful when some points in a data set are more important than others. Typically W is chosen to be a diagonal matrix, and each positive diagonal entry $W_{i,i}$ indicate how much weight should be given to the i th data point. For example, Figure 8.2a shows OLS and WLS fits of an exponential curve $y = ae^{kx}$ to data that gets more sparse as x increases, where the matrix W is chosen to give more weight to the data with larger x values.

Alternatively, the least squares problem can be formulated with other common vector norms, but such problems cannot be solved via the normal equations. For example, minimizing $\|A\mathbf{x} - \mathbf{b}\|_1$ or $\|A\mathbf{x} - \mathbf{b}\|_\infty$ is usually done by solving an equivalent linear program, a type of constrained optimization problem. These norms may be better suited to a particular application than the regular 2-norm. Figure 8.2b illustrates how different norms give slightly different results in the context of Problem 4.

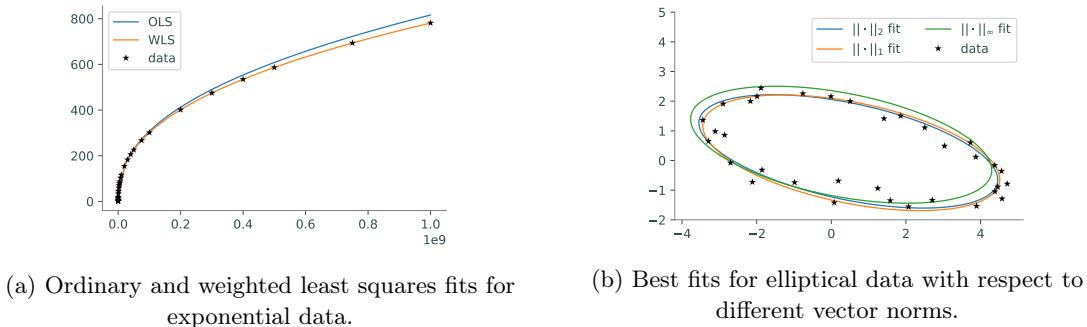


Figure 8.2: Variations on the ordinary least squares problem.

The Inverse Power Method

The major drawback of the power method is that it only computes a single eigenvector-eigenvalue pair, and it is always the eigenvalue of largest magnitude. The inverse power method, sometimes simply called the inverse iteration, is a way of computing an eigenvalue that is closest in magnitude to an initial guess. The key observation is that if λ is an eigenvalue of A , then $1/(\lambda - \mu)$ is an eigenvalue of $(A - \mu I)^{-1}$, so applying the power method to $(A - \mu I)^{-1}$ yields the eigenvalue of A that is closest in magnitude to μ .

The inverse power method is more expensive than the regular power method because at each iteration, instead of a matrix-vector multiplication (step 6 of Algorithm 8.1), a system of the form $(A - \mu I)\mathbf{x} = \mathbf{b}$ must be solved. To speed this step up, start by taking the LU or QR factorization of $A - \mu I$ before the loop, then use the factorization and back substitution to solve the system quickly within the loop. For instance, if $QR = A - \mu I$, then since $Q^{-1} = Q^T$,

$$\mathbf{b} = (A - \mu I)\mathbf{x} = QR\mathbf{x} \Leftrightarrow R\mathbf{x} = Q^T\mathbf{b},$$

which is a triangular system. This version of the algorithm is described below.

Algorithm 8.3

```

1: procedure InversePowerMethod( $A, \mu$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $\mathbf{x}_0 \leftarrow \text{random}(n)$ 
4:    $\mathbf{x}_0 \leftarrow \mathbf{x}_0 / \|\mathbf{x}_0\|$ 
5:    $Q, R \leftarrow A - \mu I$                                  $\triangleright$  Factor  $A - \mu I$  with la.qr().
6:   for  $k = 0, 1, 2, \dots, N - 1$  do
7:     Solve  $R\mathbf{x}_{k+1} = Q^T\mathbf{x}_k$                        $\triangleright$  Use la.solve_triangular().
8:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1} / \|\mathbf{x}_{k+1}\|$ 
9:   return  $\mathbf{x}_N^T A \mathbf{x}_N, \mathbf{x}_N$ 

```

It is worth noting that the QR algorithm can be improved with a similar technique: instead of computing the QR factorization of A_k , factor the shifted matrix $A_k - \mu_k I$, where μ_k is a guess for an eigenvalue of A , and unshift the recombined factorization accordingly. That is, compute

$$\begin{aligned} Q_k R_k &= A_k - \mu_k I, \\ A_{k+1} &= R_k Q_k + \mu_k I. \end{aligned}$$

This technique yields the single-shift QR algorithm. Another variant, the practical QR algorithm, uses intelligent shifts and recursively operates on smaller blocks of A_{k+1} where possible. See [QSS10, TB97] for further discussion.

9

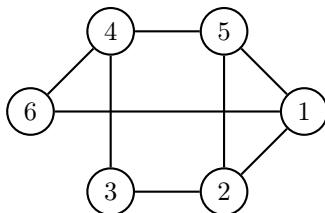
Image Segmentation

Lab Objective: Graph theory has a variety of applications. A graph (or network) can be represented in many ways on a computer. In this lab we study a common matrix representation for graphs and show how certain properties of the matrix representation correspond to inherent properties of the original graph. We also introduce tools for working with images in Python, and conclude with an application of using graphs and linear algebra to segment images.

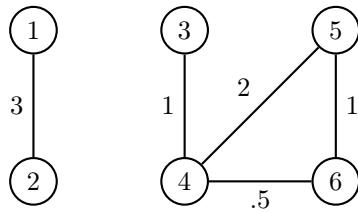
Graphs as Matrices

A graph is a mathematical structure that represents relationships between objects. Graphs are defined by $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges, each of which connects one node to another. A graph can be classified in several ways.

- The edges of an undirected graph are bidirectional: if an edge goes from node A to node B , then that same edge also goes from B to A . For example, the graphs G_1 and G_2 in Figure 9.1 are both undirected. In a directed graph, edges only go one way, usually indicated by an arrow pointing from one node to another. In this lab, we focus on undirected graphs.
- The edges of a weighted graph have a weight assigned to them, such as G_2 . A weighted graph could represent a collection of cities with roads connecting them: each vertex would represent a city, and the edges would represent roads between the cities. The length of each road could be the weight of the corresponding edge. An unweighted graph like G_1 does not have weights assigned to its edges, but any unweighted graph can be thought of as a weighted graph by assigning a weight of 1 to every edge.



(a) G_1 , an unweighted undirected graph.



(b) G_2 , a weighted undirected graph.

Figure 9.1

Adjacency, Degree, and Laplacian Matrices

For computation and analysis, graphs are commonly represented by a few special matrices. For these definitions, let G be a graph with N nodes and let w_{ij} be the weight of the edge connecting node i to node j (if such an edge exists).

1. The adjacency matrix of G is the $N \times N$ matrix A with entries

$$a_{ij} = \begin{cases} w_{ij} & \text{if an edge connects node } i \text{ and node } j \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrices A_1 of G_1 and A_2 of G_2 are

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 0 & 3 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 2 & .5 \\ 0 & 0 & 0 & 2 & 0 & 1 \\ 0 & 0 & 0 & .5 & 1 & 0 \end{bmatrix}.$$

Notice that these adjacency matrices are symmetric. This is always the case for undirected graphs since the edges are bidirectional.

2. The degree matrix of G is the $N \times N$ diagonal matrix D whose i th diagonal entry is

$$d_{ii} = \sum_{j=1}^N w_{ij}. \quad (9.1)$$

The degree matrices D_1 of G_1 and D_2 of G_2 are

$$D_1 = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}, \quad D_2 = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.5 \end{bmatrix}.$$

The i th diagonal entry of D is called the degree of node i , the sum of the weights of the edges leaving node i .

3. The Laplacian matrix of G is the $N \times N$ matrix L defined as

$$L = D - A, \quad (9.2)$$

where D is the degree matrix of G and A is the adjacency matrix of G . For G_1 and G_2 , the Laplacian matrices L_1 and L_2 are

$$L_1 = \begin{bmatrix} 3 & -1 & 0 & 0 & -1 & -1 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ -1 & 0 & 0 & -1 & 0 & 2 \end{bmatrix}, \quad L_2 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix}.$$

Problem 1. Write a function that accepts the adjacency matrix A of a graph G . Use (9.1) and (9.2) to compute the Laplacian matrix L of G .

(Hint: The diagonal entries of D can be computed in one line by summing A over an axis.)

Test your function on the graphs G_1 and G_2 from Figure 9.1 and validate your results with `scipy.sparse.csgraph.laplacian()`.

Connectivity

A connected graph is a graph where every vertex is connected to every other vertex by at least one path. For example, G_1 is connected, whereas G_2 is not because there is no path from node 1 (or node 2) to node 3 (or nodes 4, 5, or 6). The naïve brute-force algorithm for determining if a graph is connected is to check that there is a path from each edge to every other edge. While this may work for very small graphs, most interesting graphs have thousands of vertices, and for such graphs this approach is prohibitively expensive. Luckily, an interesting result from algebraic graph theory relates the connectivity of a graph to its Laplacian matrix.

If L is the Laplacian matrix of a graph, then the definition of D and the construction $L = D - A$ guarantees that the rows (and columns) of L must each sum to 0. Therefore L cannot have full rank, so $\lambda = 0$ must be an eigenvalue of L . Furthermore, if L represents a graph that is **not** connected, more than one of the eigenvalues of L must be zero. To see this, let $J \subset \{1, 2, \dots, N\}$ such that the vertices $\{v_j\}_{j \in J}$ form a connected component of the graph, meaning that there is a path between each pair of vertices in the set. Next, let \mathbf{x} be the vector with entries

$$x_k = \begin{cases} 1, & k \in J \\ 0, & k \notin J. \end{cases}$$

Then \mathbf{x} is an eigenvector of L corresponding to the eigenvalue $\lambda = 0$.

For example, the example graph G_2 has two connected components.

1. $J_1 = \{1, 2\}$ so that $\mathbf{x}_1 = [1, 1, 0, 0, 0, 0]^T$. Then

$$L_2 \mathbf{x}_1 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}.$$

2. $J_2 = \{3, 4, 5, 6\}$ and hence $\mathbf{x}_2 = [0, 0, 1, 1, 1, 1]^T$. Then

$$L_2 \mathbf{x}_2 = \begin{bmatrix} 3 & -3 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3.5 & -2 & -.5 \\ 0 & 0 & 0 & -2 & 3 & -1 \\ 0 & 0 & 0 & -.5 & -1 & 1.5 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{0}.$$

In fact, it can be shown that the number of zero eigenvalues of the Laplacian exactly equals the number of connected components. This makes calculating how many connected components are in a graph only as hard as calculating the eigenvalues of its Laplacian.

A Laplacian matrix L is always a positive semi-definite matrix when all weights in the graph are positive, meaning that its eigenvalues are each nonnegative. The second smallest eigenvalue of L is called the algebraic connectivity of the graph. It is clearly 0 for non-connected graphs, but for a connected graph, the algebraic connectivity provides useful information about its sparsity or “connectedness.” A higher algebraic connectivity indicates that the graph is more strongly connected.

Problem 2. Write a function that accepts the adjacency matrix A of a graph G and a small tolerance value `tol`. Compute the number of connected components in G and its algebraic connectivity. Consider all eigenvalues that are less than the given `tol` to be zero.

Use `scipy.linalg.eig()` or `scipy.linalg.eigvals()` to compute the eigenvalues of the Laplacian matrix. These functions return complex eigenvalues (with negligible imaginary parts); use `np.real()` to extract the real parts.

Images as Matrices

Computer images are stored as arrays of integers that indicate pixel values. Most $m \times n$ grayscale (black and white) images are stored in Python as a $m \times n$ NumPy arrays, while most $m \times n$ color images are stored as 3-dimensional $m \times n \times 3$ arrays. Color image arrays can be thought of as a stack of three $m \times n$ arrays, one each for red, green, and blue values. The datatype for an image array is `np.uint8`, unsigned 8-bit integers that range from 0 to 255. A 0 indicates a black pixel while a 255 indicates a white pixel.

Use `imageio.imread()` to read an image from a file and `imageio.imwrite()` to save an image. Matplotlib’s `plt.imshow()` displays an image array, but it displays arrays of floats between 0 and 1 more cleanly than arrays of 8-bit integers. Therefore it is customary to scale the array by dividing each entry by 255 before processing or showing the image. In this case, a 0 still indicates a black pixel, but now a 1 indicates pure white.

```
>>> from imageio import imread
>>> from matplotlib import pyplot as plt

>>> image = imread("dream.png")      # Read a (very) small image.
>>> print(image.shape)            # Since the array is 3-dimensional,
(48, 48, 3)                      # this is a color image.

# The image is read in as integers from 0 to 255.
>>> print(image.min(), image.max(), image.dtype)
0 254 uint8

# Scale the image to floats between 0 and 1 for Matplotlib.
>>> scaled = image / 255.
>>> print(scaled.min(), scaled.max(), scaled.dtype)
0.0 0.996078431373 float64

# Display the scaled image.
>>> plt.imshow(scaled)
>>> plt.axis("off")
```

A color image can be converted to grayscale by averaging the RGB values of each pixel, resulting in a 2-D array called the brightness of the image. To properly display a grayscale image, specify the keyword argument `cmap="gray"` in `plt.imshow()`.

```
# Average the RGB values of a colored image to obtain a grayscale image.
>>> brightness = scaled.mean(axis=2)          # Average over the last axis.
>>> print(brightness.shape)                 # Note that the array is now 2-D.
(48, 48)

# Display the image in gray.
>>> plt.imshow(brightness, cmap="gray")
>>> plt.axis("off")
```

Finally, it is often important in applications to flatten an image matrix into a large 1-D array. Use `np.ravel()` to convert a $m \times n$ array into a 1-D array with mn entries.

```
>>> import numpy as np
>>> A = np.random.randint(0, 10, (3,4))
>>> print(A)
[[4 4 7 7]
 [8 1 2 0]
 [7 0 0 9]]

# Unravel the 2-D array (by rows) into a 1-D array.
>>> np.ravel(A)
array([4, 4, 7, 7, 8, 1, 2, 0, 7, 0, 0, 9])

# Unravel a grayscale image into a 1-D array and check its size.
>>> M,N = brightness.shape
>>> flat_brightness = np.ravel(brightness)
>>> M*N == flat_brightness.size
True
>>> print(flat_brightness.shape)
(2304,)
```

Problem 3.

Define a class called `ImageSegmenter`.

1. Write the constructor so that it accepts the name of an image file. Read the image, scale it so that it contains floats between 0 and 1, then store it as an attribute. If the image is in color, compute its brightness matrix by averaging the RGB values at each pixel (if it is a grayscale image, the image array itself is the brightness matrix). Flatten the brightness matrix into a 1-D array and store it as an attribute.
2. Write a method called `show_original()` that displays the original image. If the original image is grayscale, remember to use `cmap="gray"` as part of `plt.imshow()`.

Achtung!

Matplotlib's `plt.imread()` also reads image files. However, this function automatically scales PNG image entries to floats between 0 and 1, but it still reads non-PNG image entries as 8-bit integers. To avoid this inconsistent behavior, always use `imageio.imread()` to read images and divide by 255 when scaling is desired.

Graph-based Image Segmentation

Image segmentation is the process of finding natural boundaries in an image and partitioning the image along those boundaries (see Figure 9.2). Though humans can easily pick out portions of an image that “belong together,” it takes quite a bit of work to teach a computer to recognize boundaries and sections in an image. However, segmenting an image often makes it easier to analyze, so image segmentation is ongoing area of research in computer vision and image processing.



Figure 9.2: The image `dream.png` and its segments.

There are many ways to approach image segmentation. The following algorithm, developed by Jianbo Shi and Jitendra Malik in 2000 [SM00], converts the image to a graph and “cuts” it into two connected components.

Constructing the Image Graph

Let G be a graph whose vertices are the mn pixels of an $m \times n$ image (either grayscale or color). Each vertex i has a brightness $B(i)$, the grayscale or average RGB value of the pixel, as well as a coordinate location $X(i)$, the indices of the pixel in the original image array.

Define w_{ij} , the weight of the edge between pixels i and j , by

$$w_{ij} = \begin{cases} \exp\left(-\frac{|B(i)-B(j)|}{\sigma_B^2} - \frac{\|X(i)-X(j)\|}{\sigma_X^2}\right) & \text{if } \|X(i) - X(j)\| < r \\ 0 & \text{otherwise,} \end{cases} \quad (9.3)$$

where r , σ_B^2 and σ_X^2 are constants for tuning the algorithm. In this context, $\|\cdot\|$ is the standard euclidean norm, meaning that $\|X(i) - X(j)\|$ is the physical distance between vertices i and j , measured in pixels.

With this definition for w_{ij} , pixels that are farther apart than the radius r are not connected at all in G . Pixels within r of each other are more strongly connected if they are similar in brightness and close together (the value in the exponential is negative but close to zero). On the other hand, highly contrasting pixels where $|B(i) - B(j)|$ is large have weaker connections (the value in the exponential is highly negative).

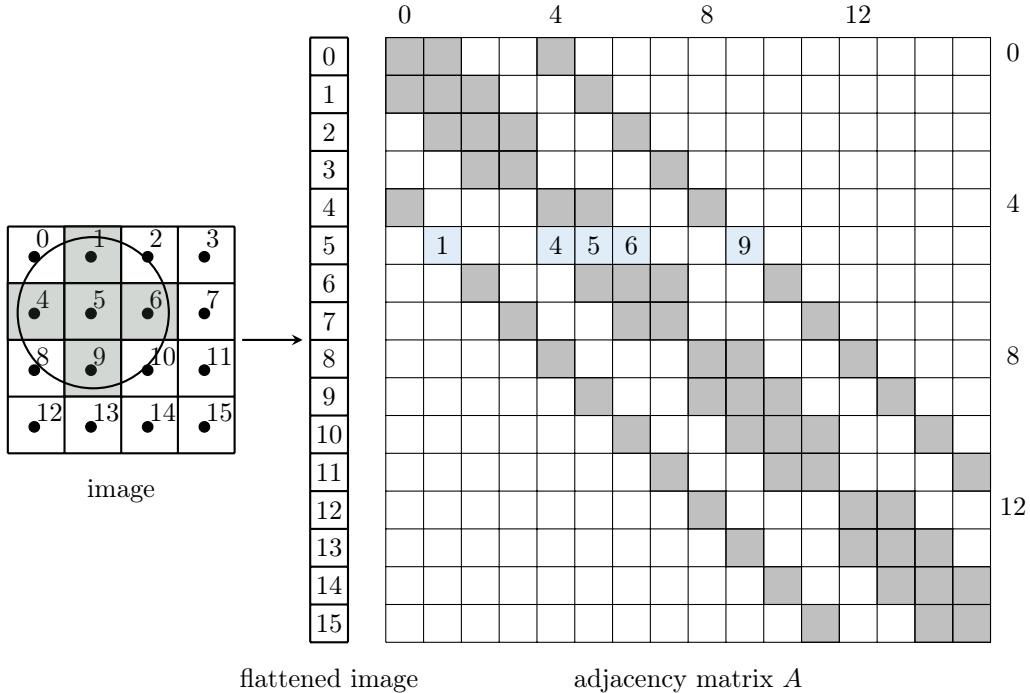


Figure 9.3: The grid on the left represents a 4×4 ($m \times n$) image with 16 pixels. On the right is the corresponding 16×16 ($mn \times mn$) adjacency matrix with all nonzero entries shaded. For example, in row 5, entries 1, 4, 5, 6, and 9 are nonzero because those pixels are within radius $r = 1.2$ of pixel 5.

Since there are mn total pixels, the adjacency matrix A of G with entries w_{ij} is $mn \times mn$. With a relatively small radius r , A is relatively sparse, and should therefore be constructed and stored as a sparse matrix. The degree matrix D is diagonal, so it can be stored as a regular 1-dimensional NumPy array. The procedure for constructing these matrices can be summarized in just a few steps.

1. Initialize A as a sparse $mn \times mn$ matrix and D as a vector with mn entries.
2. For each vertex i ($i = 0, 1, \dots, mn - 1$),
 - (a) Find the set of all vertices J_i such that $\|X(i) - X(j)\| < r$ for each $j \in J_i$. For example, in Figure 9.3 $i = 5$ and $J_i = \{1, 4, 5, 6, 9\}$.
 - (b) Calculate the weights w_{ij} for each $j \in J_i$ according to (9.3) and store them in A .
 - (c) Set the i th element of D to be the sum of the weights, $d_i = \sum_{j \in J_i} w_{ij}$.

The most difficult part to implement efficiently is step 2a, computing the neighborhood J_i of the current pixel i . However, the computation only requires knowing the current index i , the radius r , and the height and width m and n of the original image. The following function takes advantage of this fact and returns (as NumPy arrays) both J_i and the distances $\|X(i) - X(j)\|$ for each $j \in J_i$.

```

def get_neighbors(index, radius, height, width):
    """Calculate the flattened indices of the pixels that are within the given
    distance of a central pixel, and their distances from the central pixel.

    Parameters:
        index (int): The index of a central pixel in a flattened image array
                      with original shape (radius, height).
        radius (float): Radius of the neighborhood around the central pixel.
        height (int): The height of the original image in pixels.
        width (int): The width of the original image in pixels.

    Returns:
        (1-D ndarray): the indices of the pixels that are within the specified
                      radius of the central pixel, with respect to the flattened image.
        (1-D ndarray): the euclidean distances from the neighborhood pixels to
                      the central pixel.
    """
    # Calculate the original 2-D coordinates of the central pixel.
    row, col = index // width, index % width

    # Get a grid of possible candidates that are close to the central pixel.
    r = int(radius)
    x = np.arange(max(col - r, 0), min(col + r + 1, width))
    y = np.arange(max(row - r, 0), min(row + r + 1, height))
    X, Y = np.meshgrid(x, y)

    # Determine which candidates are within the given radius of the pixel.
    R = np.sqrt(((X - col)**2 + (Y - row)**2))
    mask = R < radius
    return (X[mask] + Y[mask]*width).astype(np.int), R[mask]

```

To see how this works, consider Figure 9.3 where the original image is 4×4 and the goal is to compute the neighborhood of the pixel $i = 5$.

```

# Compute the neighbors and corresponding distances from the figure.
>>> neighbors_1, distances_1 = get_neighbors(5, 1.2, 4, 4)
>>> print(neighbors_1, distances_1, sep='\n')
[1 4 5 6 9]
[ 1.  1.  0.  1.  1.]

# Increasing the radius from 1.2 to 1.5 results in more neighbors.
>>> neighbors_2, distances_2 = get_neighbors(5, 1.5, 4, 4)
>>> print(neighbors_2, distances_2, sep='\n')
[ 0  1  2  4  5  6  8  9 10]
[ 1.41421356  1.           1.41421356  1.           0.           1.
  1.41421356  1.           1.41421356]

```

Problem 4. Write a method for the `ImageSegmenter` class that accepts floats r defaulting to 5, σ_B^2 defaulting to .02, and σ_X^2 defaulting to 3. Compute the adjacency matrix A and the degree matrix D according to the weights specified in (9.3).

Initialize A as a `scipy.sparse.lil_matrix`, which is optimized for incremental construction. Fill in the nonzero elements of A one row at a time. Use `get_neighbors()` at each step to help compute the weights.

(Hint: Try to compute and store an entire row of weights at a time. What does the command `A[5, np.array([1, 4, 5, 6, 9])] = weights` do?)

Finally, convert A to a `scipy.sparse.csc_matrix`, which is faster for computations. Then return A and D .

Use `blue_heart.png` to test A and D , saved as `HeartMatrixA.npz` and `HeartMatrixD.npy` datafiles.

Segmenting the Graph

With an image represented as a graph G , the goal is to now split G into two distinct connected components by removing edges from the existing graph. This is called cutting G , and the set of edges that are removed is called the cut. The cut with the least weight will best segment the image.

Let D be the degree matrix and L be the Laplacian matrix of G . Shi and Malik [SM00] proved that the eigenvector corresponding to the second smallest¹ eigenvalue of $D^{-1/2}LD^{-1/2}$ can be used to minimize the cut: the indices of its positive entries are the indices of the pixels in the flattened image which belong to one segment, and the indices of its negative entries are the indices of the pixels which belong to the other segment. In this context $D^{-1/2}$ refers to element-wise exponentiation, so the (i, j) th entry of $D^{-1/2}$ is $1/\sqrt{d_{ij}}$.

Because A is $mn \times mn$, the desired eigenvector has mn entries. Reshaping the eigenvector to be $m \times n$ allows it to align with the original image. Use the reshaped eigenvector to create a boolean mask that indexes one of the segments. That is, construct a $m \times n$ array where the entries belonging to one segment are `True` and the other entries are `False`.

```
>>> x = np.arange(-5,5).reshape((5,2)).T
>>> print(x)
[[ -5 -3 -1  1  3]
 [-4 -2  0  2  4]]

# Construct a boolean mask of x describing which entries of x are positive.
>>> mask = x > 0
>>> print(mask)
[[False False False  True  True]
 [False False False  True  True]]

# Use the mask to zero out all of the nonpositive entries of x.
>>> x * mask
array([[ 0,  0,  0,  1,  3],
       [ 0,  0,  0,  2,  4]])
```

¹Both D and L are symmetric matrices, so all eigenvalues of $D^{-1/2}LD^{-1/2}$ are real, and therefore “the second smallest one” is well-defined.

Problem 5. Write a method for the `ImageSegmenter` class that accepts an adjacency matrix A as a `scipy.sparse.csc_matrix` and a degree matrix D as a 1-D NumPy array. Construct an $m \times n$ boolean mask describing the segments of the image.

1. Compute the Laplacian L with `scipy.sparse.csgraph.laplacian()` or by converting D to a sparse diagonal matrix and computing $L = D - A$ (do not use your function from Problem 1 unless it works correctly and efficiently for sparse matrices).
2. Construct $D^{-1/2}$ as a sparse diagonal matrix using D and `scipy.sparse.diags()`, then compute $D^{-1/2}LD^{-1/2}$. Use `@` or the `dot()` method of the sparse matrix for the matrix multiplication, **not** `np.dot()`.
3. Use `scipy.sparse.linalg.eigsh()` to compute the eigenvector corresponding to the second-smallest eigenvalue of $D^{-1/2}LD^{-1/2}$. Set the keyword arguments `which="SM"` and `k=2` to compute only the two smallest eigenvalues and their eigenvectors.
4. Reshape the eigenvector as a $m \times n$ matrix and use this matrix to construct the desired boolean mask. Return the mask.

Multiplying the boolean mask component-wise by the original image array produces the positive segment, a copy of the original image where the entries that aren't in the segment are set to 0. Computing the negative segment requires inverting the boolean mask, then multiplying the inverted mask with the original image array. Finally, if the original image is a $m \times n \times 3$ color image, the mask must be stacked into a $m \times n \times 3$ array to facilitate entry-wise multiplication.

```
>>> mask = np.arange(-5,5).reshape((5,2)).T > 0
>>> print(mask)
[[False False False  True  True]
 [False False False  True  True]]

# The mask can be negated with the tilde operator ~.
>>> print(~mask)
[[ True  True  True False False]
 [ True  True  True False False]]

# Stack a mask into a 3-D array with np.dstack().
>>> print(mask.shape, np.dstack((mask, mask, mask)).shape)
(2, 5) (2, 5, 3)
```

Problem 6. Write a method for the `ImageSegmenter` class that accepts floats r , σ_B^2 , and σ_X^2 , with the same defaults as in Problem 4. Call your methods from Problems 4 and 5 to obtain the segmentation mask. Plot the original image, the positive segment, and the negative segment side-by-side in subplots. Your method should work for grayscale or color images.

Use `dream.png` as a test file and compare your results to Figure 9.2.

10

The SVD and Image Compression

Lab Objective: The Singular Value Decomposition (SVD) is an incredibly useful matrix factorization that is widely used in both theoretical and applied mathematics. The SVD is structured in a way that makes it easy to construct low-rank approximations of matrices, and it is therefore the basis of several data compression algorithms. In this lab we learn to compute the SVD and use it to implement a simple image compression routine.

The SVD of a matrix A is a factorization $A = U\Sigma V^H$ where U and V have orthonormal columns and Σ is diagonal. The diagonal entries of Σ are called the singular values of A and are the square roots of the eigenvalues of $A^H A$. Since $A^H A$ is always positive semidefinite, its eigenvalues are all real and nonnegative, so the singular values are also real and nonnegative. The singular values σ_i are usually sorted in decreasing order so that $\Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$. The columns \mathbf{u}_i of U , the columns \mathbf{v}_i of V , and the singular values of A satisfy $A\mathbf{v}_i = \sigma_i \mathbf{u}_i$.

Every $m \times n$ matrix A of rank r has an SVD with exactly r nonzero singular values. Like the QR decomposition, the SVD has two main forms.

- **Full SVD:** Denoted $A = U\Sigma V^H$. U is $m \times m$, V is $n \times n$, and Σ is $m \times n$. The first r columns of U span $\mathcal{R}(A)$, and the remaining $n - r$ columns span $\mathcal{N}(A^H)$. Likewise, the first r columns of V span $\mathcal{R}(A^H)$, and the last $m - r$ columns span $\mathcal{N}(A)$.
- **Compact (Reduced) SVD:** Denoted $A = U_1 \Sigma_1 V_1^H$. U_1 is $m \times r$ (the first r columns of U), V_1 is $n \times r$ (the first r columns of V), and Σ_1 is $r \times r$ (the first $r \times r$ block of Σ). This smaller version of the SVD has all of the information needed to construct A and nothing more. The zero singular values and the corresponding columns of U and V are neglected.

$$\begin{array}{ccc}
 U_1 \ (m \times r) & \Sigma_1 \ (r \times r) & V_1^H \ (r \times n) \\
 \left[\begin{array}{ccccccccc}
 \boxed{\mathbf{u}_1 & \cdots & \mathbf{u}_r} & \mathbf{u}_{r+1} & \cdots & \mathbf{u}_m
 \end{array} \right] & \left[\begin{array}{ccccc}
 \sigma_1 & & & & \\
 & \ddots & & & \\
 & & \sigma_r & & \\
 & & & 0 & \\
 & & & & \ddots & \\
 & & & & & 0
 \end{array} \right] & \left[\begin{array}{ccccccccc}
 \boxed{\mathbf{v}_1^H} & & & & & & & & \\
 \vdots & & & & & & & & \\
 \boxed{\mathbf{v}_r^H} & & & & & & & & \\
 \mathbf{v}_{r+1}^H & & & & & & & & \\
 \vdots & & & & & & & & \\
 \boxed{\mathbf{v}_n^H} & & & & & & & &
 \end{array} \right] \\
 U \ (m \times m) & \Sigma \ (m \times n) & V^H \ (n \times n)
 \end{array}$$

Finally, the SVD yields an outer product expansion of A in terms of the singular values and the columns of U and V ,

$$A = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^H. \quad (10.1)$$

Note that only terms from the compact SVD are needed for this expansion.

Computing the Compact SVD

It is difficult to compute the SVD from scratch because it is an eigenvalue-based decomposition. However, given an eigenvalue solver such as `scipy.linalg.eig()`, the algorithm becomes much simpler. First, obtain the eigenvalues and eigenvectors of $A^H A$, and use these to compute Σ . Since $A^H A$ is normal, it has an orthonormal eigenbasis, so set the columns of V to be the eigenvectors of $A^H A$. Then, since $A \mathbf{v}_i = \sigma_i \mathbf{u}_i$, construct U by setting its columns to be $\mathbf{u}_i = \frac{1}{\sigma_i} A \mathbf{v}_i$.

The key is to sort the singular values and the corresponding eigenvectors in the same manner. In addition, it is computationally inefficient to keep track of the entire matrix Σ since it is a matrix of mostly zeros, so we need only store the singular values as a vector σ . The entire procedure for computing the compact SVD is given below.

Algorithm 10.1

```

1: procedure compact_SVD( $A$ )
2:    $\lambda, V \leftarrow \text{eig}(A^H A)$                                  $\triangleright$  Calculate the eigenvalues and eigenvectors of  $A^H A$ .
3:    $\sigma \leftarrow \sqrt{\lambda}$                                       $\triangleright$  Calculate the singular values of  $A$ .
4:    $\sigma \leftarrow \text{sort}(\sigma)$                                   $\triangleright$  Sort the singular values from greatest to least.
5:    $V \leftarrow \text{sort}(V)$                                       $\triangleright$  Sort the eigenvectors the same way as in the previous step.
6:    $r \leftarrow \text{count}(\sigma \neq 0)$                              $\triangleright$  Count the number of nonzero singular values (the rank of  $A$ ).
7:    $\sigma_1 \leftarrow \sigma_{:r}$                                         $\triangleright$  Keep only the positive singular values.
8:    $V_1 \leftarrow V_{:,:r}$                                           $\triangleright$  Keep only the corresponding eigenvectors.
9:    $U_1 \leftarrow A V_1 / \sigma_1$                                  $\triangleright$  Construct  $U$  with array broadcasting.
10:  return  $U_1, \sigma_1, V_1^H$ 

```

Problem 1. Write a function that accepts a matrix A and a small error tolerance `tol`. Use Algorithm 10.1 to compute the compact SVD of A . In step 6, compute r by counting the number of singular values that are greater than `tol`.

Consider the following tips for implementing the algorithm.

- The Hermitian A^H can be computed with `A.conj().T`.
- In step 4, the way that σ is sorted needs to be stored so that the columns of V can be sorted the same way. Consider using `np.argsort()` and fancy indexing to do this, but remember that by default it sorts from least to greatest (not greatest to least).
- Step 9 can be done by looping over the columns of V , but it can be done more easily and efficiently with array broadcasting.

Test your function by calculating the compact SVD for random matrices. Verify that U and V are orthonormal, that $U \Sigma V^H = A$, and that the number of nonzero singular values is the rank of A . You may also want to compare your results to SciPy's SVD algorithm.

```

>>> import numpy as np
>>> from scipy import linalg as la

# Generate a random matrix and get its compact SVD via SciPy.
>>> A = np.random.random((10,5))
>>> U,s,Vh = la.svd(A, full_matrices=False)
>>> print(U.shape, s.shape, Vh.shape)
(10, 5) (5,) (5, 5)

# Verify that U is orthonormal, U Sigma Vh = A, and the rank is correct.
>>> np.allclose(U.T @ U, np.identity(5))
True
>>> np.allclose(U @ np.diag(s) @ Vh, A)
True
>>> np.linalg.matrix_rank(A) == len(s)
True

```

Visualizing the SVD

An $m \times n$ matrix A defines a linear transformation that sends points from \mathbb{R}^n to \mathbb{R}^m . The SVD decomposes a matrix into two rotations and a scaling, so that any linear transformation can be easily described geometrically. Specifically, V^H represents a rotation, Σ a rescaling along the principal axes, and U another rotation.

Problem 2. Write a function that accepts a 2×2 matrix A . Generate a 2×200 matrix S representing a set of 200 points on the unit circle, with x -coordinates on the top row and y -coordinates on the bottom row (recall the equation for the unit circle in polar coordinates: $x = \cos(\theta)$, $y = \sin(\theta)$, $\theta \in [0, 2\pi]$). Also define the matrix

$$E = [\mathbf{e}_1 \mid \mathbf{0} \mid \mathbf{e}_2] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

so that plotting the first row of S against the second row of S displays the unit circle, and plotting the first row of E against its second row displays the standard basis vectors in \mathbb{R}^2 .

Compute the full SVD $A = U\Sigma V^H$ using `scipy.linalg.svd()`. Plot four subplots to demonstrate each step of the transformation, plotting S and E , $V^H S$ and $V^H E$, $\Sigma V^H S$ and $\Sigma V^H E$, then $U\Sigma V^H S$ and $U\Sigma V^H E$.

For the matrix

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix},$$

your function should produce Figure 10.1.

(Hint: Use `plt.axis("equal")` to fix the aspect ratio so that the circles don't appear elliptical.)

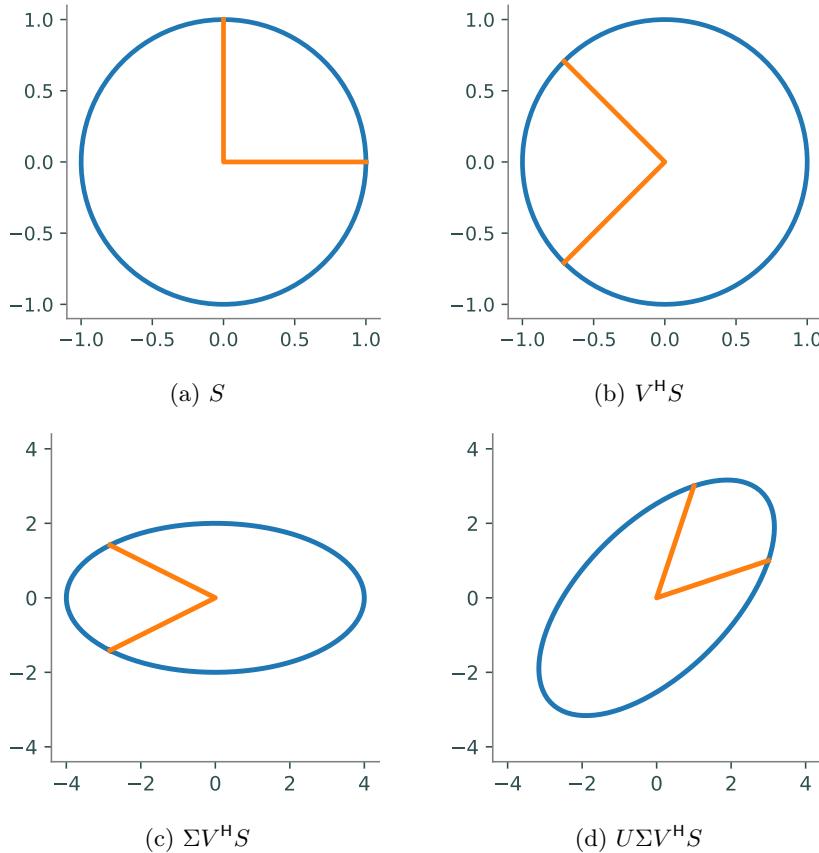


Figure 10.1: Each step in transforming the unit circle and two unit vectors using the matrix A .

Using the SVD for Data Compression

Low-Rank Matrix Approximations

If A is a $m \times n$ matrix of rank $r < \min\{m, n\}$, then the compact SVD offers a way to store A with less memory. Instead of storing all mn values of A , storing the matrices U_1 , Σ_1 and V_1 only requires saving a total of $mr + r + nr$ values. For example, if A is 100×200 and has rank 20, then A has 20,000 values, but its compact SVD only has total 6,020 entries, a significant decrease.

The truncated SVD is an approximation to the compact SVD that allows even greater efficiency at the cost of a little accuracy. Instead of keeping all of the nonzero singular values, the truncated SVD only keeps the first $s < r$ singular values, plus the corresponding columns of U and V . In this case, (10.1) becomes

$$A_s = \sum_{i=1}^s \sigma_i \mathbf{u}_i \mathbf{v}_i^H.$$

More precisely, the truncated SVD of A is $A_s = \widehat{U} \widehat{\Sigma} \widehat{V}^H$, where \widehat{U} is $m \times s$, \widehat{V} is $n \times s$, and $\widehat{\Sigma}$ is $s \times s$. The resulting matrix A_s has rank s and is only an approximation to A , since $r - s$ nonzero singular values are neglected.

$$\begin{bmatrix}
 \widehat{U} & (m \times s) \\
 \left[\begin{array}{ccccccccc}
 \mathbf{u}_1 & \cdots & \mathbf{u}_s & \mathbf{u}_{s+1} & \cdots & \mathbf{u}_r
 \end{array} \right] & U_1 & (m \times r)
 \end{bmatrix} \quad
 \begin{bmatrix}
 \widehat{\Sigma} & (s \times s) \\
 \left[\begin{array}{ccccccccc}
 \sigma_1 & & & & & & & & \\
 & \ddots & & & & & & & \\
 & & \sigma_s & & & & & & \\
 & & & & \sigma_{s+1} & & & & \\
 & & & & & \ddots & & & \\
 & & & & & & \sigma_r & &
 \end{array} \right] & \Sigma_1 & (r \times r)
 \end{bmatrix} \quad
 \begin{bmatrix}
 \widehat{V}^H & (s \times n) \\
 \left[\begin{array}{ccccccccc}
 \mathbf{v}_1^H & & & & & & & & \\
 & \vdots & & & & & & & \\
 & & \mathbf{v}_s^H & & & & & & \\
 & & & & \mathbf{v}_{s+1}^H & & & & \\
 & & & & & \ddots & & & \\
 & & & & & & \mathbf{v}_r^H & &
 \end{array} \right] & V_1^H & (r \times n)
 \end{bmatrix}$$

The beauty of the SVD is that it makes it easy to select the information that is most important. Larger singular values correspond to columns of U and V that contain more information, so dropping the smallest singular values retains as much information as possible. In fact, given a matrix A , its rank- s truncated SVD approximation A_s is the best rank s approximation of A with respect to both the induced 2-norm and the Frobenius norm. This result is called the Schmidt, Mirsky, Eckhart-Young theorem, a very significant concept that appears in signal processing, statistics, machine learning, semantic indexing (search engines), and control theory.

Problem 3. Write a function that accepts a matrix A and a positive integer s .

1. Use your function from Problem 1 or `scipy.linalg.svd()` to compute the compact SVD of A , then form the truncated SVD by stripping off the appropriate columns and entries from U_1 , Σ_1 , and V_1 . Return the best rank s approximation A_s of A (with respect to the induced 2-norm and Frobenius norm).
2. Also return the number of entries required to store the truncated form $\widehat{U}\widehat{\Sigma}\widehat{V}^H$ (where $\widehat{\Sigma}$ is stored as a one-dimensional array, not the full diagonal matrix). The number of entries stored in NumPy array can be accessed by its `size` attribute.

```
>>> A = np.random.random((20, 20))
>>> A.size
400
```

3. If s is greater than the number of nonzero singular values of A (meaning $s > \text{rank}(A)$), raise a `ValueError`.

Use `np.linalg.matrix_rank()` to verify the rank of your approximation.

Error of Low-Rank Approximations

Another result of the Schmidt, Mirsky, Eckhart-Young theorem is that the exact 2-norm error of the best rank- s approximation A_s for the matrix A is the $(s + 1)$ th singular value of A :

$$\|A - A_s\|_2 = \sigma_{s+1}. \quad (10.2)$$

This offers a way to approximate A within a desired error tolerance ε : choose s such that σ_{s+1} is the largest singular value that is less than ε , then compute A_s . This A_s throws away as much information as possible without violating the property $\|A - A_s\|_2 < \varepsilon$.

Problem 4. Write a function that accepts a matrix A and an error tolerance ε .

1. Compute the compact SVD of A , then use (10.2) to compute the lowest rank approximation A_s of A with 2-norm error less than ε . Avoid calculating the SVD more than once. (Hint: `np.argmax()`, `np.where()`, and/or fancy indexing may be useful.)
2. As in the previous problem, also return the number of entries needed to store the resulting approximation A_s via the truncated SVD.
3. If ε is less than or equal to the smallest singular value of A , raise a `ValueError`; in this case, A cannot be approximated within the tolerance by a matrix of lesser rank.

This function should be close to identical to the function from Problem 3, but with the extra step of identifying the appropriate s . Construct test cases to validate that $\|A - A_s\|_2 < \varepsilon$.

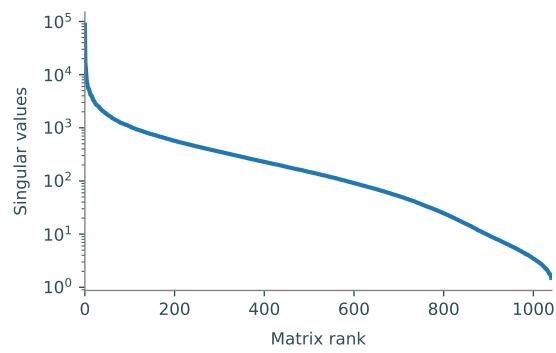
Image Compression

Images are stored on a computer as matrices of pixel values. Sending an image over the internet or a text message can be expensive, but computing and sending a low-rank SVD approximation of the image can considerably reduce the amount of data sent while retaining a high level of image detail. Successive levels of detail can be sent after the initial low-rank approximation by sending additional singular values and the corresponding columns of V and U .

Examining the singular values of an image gives us an idea of how low-rank the approximation can be. Figure 10.2 shows the image in `hubble_gray.jpg` and a log plot of its singular values. The plot in 10.2b is typical for a photograph—the singular values start out large but drop off rapidly. In this rank 1041 image, 913 of the singular values are 100 or more times smaller than the largest singular value. By discarding these relatively small singular values, we can retain all but the finest image details, while storing only a rank 128 image. This is a **huge** reduction in data size.



(a) NGC 3603 (Hubble Space Telescope).



(b) Singular values on a log scale.

Figure 10.2

Figure 10.3 shows several low-rank approximations of the image in Figure 10.2a. Even at a low rank the image is recognizable. By rank 120, the approximation differs very little from the original.

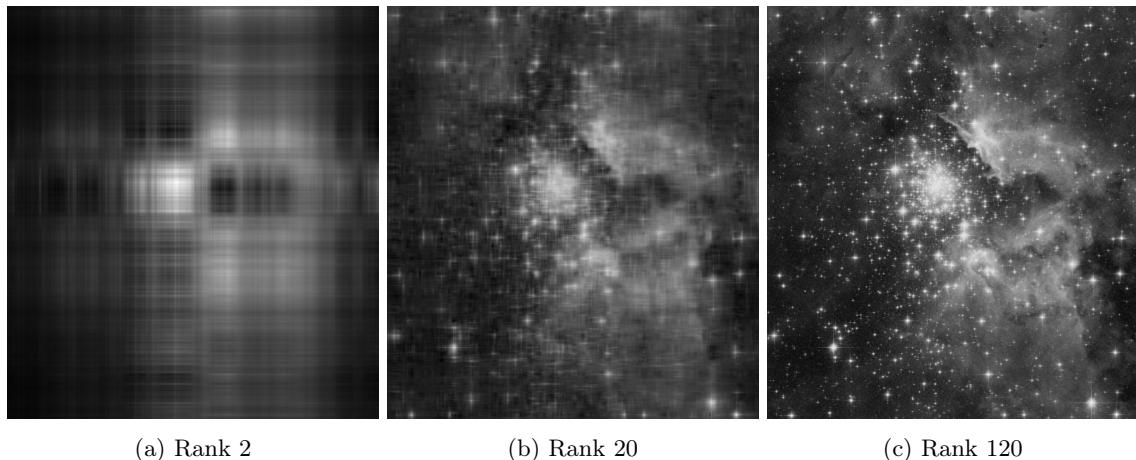


Figure 10.3

Grayscale images are stored on a computer as 2-dimensional arrays, while color images are stored as 3-dimensional arrays—one layer each for red, green, and blue arrays. To read and display images, use `imageio.imread()` and `plt.imshow()`. Images are read in as integer arrays with entries between 0 and 255 (`dtype=np.uint8`), but `plt.imshow()` works better if the image is an array of floats in the interval [0, 1]. Scale the image properly by dividing the array by 255.

```
>>> from imageio import imread
>>> from matplotlib import pyplot as plt

# Send the RGB values to the interval (0,1).
>>> image_gray = imread("hubble_gray.jpg") / 255.
>>> image_gray.shape           # Grayscale images are 2-d arrays.
(1158, 1041)
>>> image_color = imread("hubble.jpg") / 255.
>>> image_color.shape         # Color images are 3-d arrays.
(1158, 1041, 3)

# The final axis has 3 layers for red, green, and blue values.
>>> red_layer = image_color[:, :, 0]
>>> red_layer.shape
(1158, 1041)

# Display a gray image.
>>> plt.imshow(red_layer, cmap="gray")
>>> plt.axis("off")          # Turn off axis ticks and labels.
>>> plt.show()

# Display a color image.
>>> plt.imshow(image_color)   # cmap=None by default.
>>> plt.axis("off")
>>> plt.show()
```

Problem 5. Write a function that accepts the name of an image file and an integer s . Use your function from Problem 3, to compute the best rank- s approximation of the image. Plot the original image and the approximation in separate subplots. In the figure title, report the difference in number of entries required to store the original image and the approximation (use `plt.suptitle()`).

Your function should be able to handle both grayscale and color images. Read the image in and check its dimensions to see if it is color or not. Grayscale images can be approximated directly since they are represented by 2-dimensional arrays. For color images, let R , G , and B be the matrices for the red, green, and blue layers of the image, respectively. Calculate the low-rank approximations R_s , G_s , and B_s separately, then put them together in a new 3-dimensional array of the same shape as the original image.

(Hint: `np.dstack()` may be useful for putting the color layers back together.)

Finally, it is possible for the low-rank approximations to have values slightly outside the valid range of RGB values. Set any values outside of the interval $[0, 1]$ to the closer of the two boundary values.

(Hint: fancy indexing and/or `np.clip()` may be useful here.)

To check, compressing `hubble_gray.jpg` with a rank 20 approximation should appear similar to Figure 10.3b and save 1,161,478 matrix entries.

Additional Material

More on Computing the SVD

For an $m \times n$ matrix A of rank $r < \min\{m, n\}$, the compact SVD of A neglects last $m - r$ columns of U and the last $n - r$ columns of V . The remaining columns of each matrix can be calculated by using Gram-Schmidt orthonormalization. If $m < r < n$ or $n < r < m$, only one of U_1 and V_1 will need to be filled in to construct the full U or V . Computing these extra columns is one way to obtain a basis for $\mathcal{N}(A^H)$ or $\mathcal{N}(A)$.

Algorithm 10.1 begins with the assumption that we have a way to compute the eigenvalues and eigenvectors of $A^H A$. Computing eigenvalues is a notoriously difficult problem, and computing the SVD from scratch without an eigenvalue solver is much more difficult than the routine described by Algorithm 10.1. The procedure involves two phases:

1. Factor A into $A = U_a B V_a^H$ where B is bidiagonal (only nonzero on the diagonal and the first superdiagonal) and U_a and V_a are orthonormal. This is usually done via Golub-Kahan Bidiagonalization, which uses Householder reflections, or Lawson-Hanson-Chan bidiagonalization, which relies on the QR decomposition.
2. Factor B into $B = U_b \Sigma V_b^H$ by the QR algorithm or a divide-and-conquer algorithm. Then the SVD of A is given by $A = (U_a U_b) \Sigma (V_a V_b)^H$.

For more details, see Lecture 31 of [TB97] or Section 5.4 of Applied Numerical Linear Algebra by James W. Demmel.

Animating Images with Matplotlib

Matplotlib can be used to animate images that change over time. For instance, we can show how the low-rank approximations of an image change as the rank s increases, showing how the image is recovered as more ranks are added. Try using the following code to create such an animation.

```
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation

def animate_images(images):
    """Animate a sequence of images. The input is a list where each
    entry is an array that will be one frame of the animation.
    """
    fig = plt.figure()
    plt.axis("off")
    im = plt.imshow(images[0], animated=True)

    def update(index):
        plt.title("Rank {} Approximation".format(index))
        im.set_array(images[index])
        return im,           # Note the comma!

    a = FuncAnimation(fig, update, frames=len(images), blit=True)
    plt.show()
```

See https://matplotlib.org/examples/animation/dynamic_image.html for another example.

11

Facial Recognition

Lab Objective: Facial recognition algorithms attempt to match a person's portrait to a database of many portraits. Facial recognition is becoming increasingly important in security, law enforcement, artificial intelligence, and other areas. Though humans can easily match pictures to people, computers are beginning to surpass humans at facial recognition. In this lab, we implement a basic facial recognition system that relies on eigenvectors and the SVD to efficiently determine the difference between faces.

Preparing an Image Database

The `faces94` face image dataset¹ contains several photographs of 153 people, organized into folders by person. To perform facial recognition on this dataset, select one image per person and convert these images into a database. For this particular facial recognition algorithm, the entire database can be stored in just a few NumPy arrays.

Digital images are stored on computers as arrays of pixels. Therefore, an $m \times n$ image can be stored in memory as an $m \times n$ matrix or, equivalently, as an mn -vector by concatenating the rows of the matrix. Then a collection of k images can be stored as a single $mn \times k$ matrix F , where each column of F represents a single image. That is, if

$$F = \left[\begin{array}{c|c|c|c} & & & \\ \mathbf{f}_1 & \mathbf{f}_2 & \cdots & \mathbf{f}_k \\ & & & \end{array} \right],$$

then each \mathbf{f}_i is a mn -vector representing a single image.

The following function obtains one image for each person in the `faces94` dataset and converts the collection of images into an $mn \times k$ matrix F described above.

```
import os
import numpy as np
from imageio import imread

def get_faces(path="."):
    # Traverse the directory and get one image per subdirectory.
```

¹See <http://cswww.essex.ac.uk/mv/allfaces/faces94.html>.

```

faces = []
for (dirpath, dirnames, filenames) in os.walk(path):
    for fname in filenames:
        if fname[-3:]=="jpg":      # Only get jpg images.
            # Load the image, convert it to grayscale,
            # and flatten it into a vector.
            faces.append(np.ravel(imread(dirpath+"/"+fname, as_gray=True)))
            break
# Put all the face vectors column-wise into a matrix.
return np.transpose(faces)

```

Problem 1. Write a function that accepts an image as a flattened mn -vector, along with its original dimensions m and n . Use `np.reshape()` to convert the flattened image into its original $m \times n$ shape and display the result with `plt.imshow()`.

(Hint: use `cmap="gray"` in `plt.imshow()` to display images in grayscale.)

Unzip the `faces94.zip` archive and use `get_faces()` to construct F . Each `faces94` image is 200×180 , and there are 153 people in the dataset, so F should be 36000×153 . Use your function to display one of the images stored in F .

The Eigenfaces Method

With the image database F , we could construct a simple facial recognition system with the following strategy. Let \mathbf{g} be an mn -vector representing an unknown face that is not part of the database F . Then the \mathbf{f}_i that minimizes $\|\mathbf{g} - \mathbf{f}_i\|_2$ is the matching face. Unfortunately, computing $\|\mathbf{g} - \mathbf{f}_i\|_2$ for each i is very computationally expensive, especially if the images are high-resolution and/or the database contains a large number of images. The eigenfaces method is a way to reduce the computational cost of finding the closest matching face by focusing on only the most important features of each face. Because the method ignores less significant facial features, it is also usually more accurate than the naïve method.

The first step of the algorithm is to shift the images by the mean face. Shifting a set of data by the mean exaggerates the distinguishing features of each entry. In the context of facial recognition, shifting by the mean accentuates the unique features of each face. For the images vectors stored in F , the mean face $\boldsymbol{\mu}$ is defined to be the element-wise average of the \mathbf{f}_i :

$$\boldsymbol{\mu} = \frac{1}{k} \sum_{i=1}^k \mathbf{f}_i.$$

Hence, the i th mean-shifted face vector $\bar{\mathbf{f}}_i$ is given by

$$\bar{\mathbf{f}}_i = \mathbf{f}_i - \boldsymbol{\mu}.$$

Next, define \bar{F} as the $mn \times k$ matrix whose columns are given by the mean-shifted face vectors,

$$\bar{F} = \left[\begin{array}{c|c|c|c} \bar{\mathbf{f}}_1 & \bar{\mathbf{f}}_2 & \cdots & \bar{\mathbf{f}}_k \end{array} \right].$$



(a) The mean face. (b) An original face. (c) A mean-shifted face.

Figure 11.1

Problem 2. Write a class called `FacialRec` whose constructor accepts a path to a directory of images. In the constructor, use `get_faces()` to construct F , then compute the mean face μ and the shifted faces \bar{F} . Store each array as an attribute.

(Hint: Both μ and \bar{F} can be computed in a single line of code by using NumPy functions and/or array broadcasting.)

Use your function from Problem 1 to visualize the mean face, and compare it to Figure 11.1a. Also display an original face and its corresponding mean-shifted face. Compare your results with Figures 11.1b and 11.1c.

To increase computational efficiency and minimize storage, the face vectors can be represented with fewer values by projecting \bar{F} onto a lower-dimensional subspace. Let s be a natural number such that $s < r$, where r is the rank of \bar{F} . By projecting \bar{F} onto an s -dimensional subspace, each face can be stored with only s values.

Specifically, let $U\Sigma V^H$ be the compact SVD of \bar{F} with rank r , which can also be represented by

$$\bar{F} = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^H.$$

The first r columns of U form a basis for the range of \bar{F} . Recall that the Schmidt, Mirsky, Eckart-Young Theorem states that the matrix

$$\bar{F}_s = \sum_{i=1}^s \sigma_i \mathbf{u}_i \mathbf{v}_i^H$$

is the best rank- s approximation of \bar{F} for each $s < r$. This means that $\|\bar{F} - \bar{F}_s\|$ is minimized against all other $\|\bar{F} - B\|$ where B has rank s . As a consequence of this theorem, the first s columns of U form a basis that provides the “best” s -dimensional subspace for approximating \bar{F} .

The s basis vectors $\mathbf{u}_1, \dots, \mathbf{u}_s$ are commonly called the eigenfaces because they are eigenvectors of $\bar{F}\bar{F}^T$ and because they resemble face images. Each original face image can be efficiently represented in terms of these eigenfaces. See Figure 11.2 for visualizations of some of the eigenfaces for the `facesd94` data set.



Figure 11.2: The first, 50th, and 100th eigenfaces.

In general, the lower eigenfaces provide a more general information of a face and higher-ordered eigenfaces provide the details necessary to distinguish particular faces [MMH04]. These eigenfaces will be used to construct the face images in the dataset. The more eigenfaces used, the more detailed the resulting image will be.

Next, let U_s be the matrix with the first s eigenfaces as columns. Since the eigenfaces $\{\mathbf{u}_i\}_{i=1}^s$ form an orthonormal set, U_s is an orthonormal matrix (independent of s) and hence $U_s^\top U_s = I$. The matrix $P_s = U_s U_s^\top$ projects vectors in \mathbb{R}^{mn} to the subspace spanned by the orthonormal basis $\{\mathbf{u}_i\}_{i=1}^s$, and the change of basis matrix U_s^\top puts the projection in terms of the basis of eigenfaces. Thus the projection $\hat{\mathbf{f}}_i$ of $\bar{\mathbf{f}}_i$ in terms of the basis of eigenfaces is given by

$$\hat{\mathbf{f}}_i = U_s^\top P_s \bar{\mathbf{f}}_i = U_s^\top U_s U_s^\top \bar{\mathbf{f}}_i = U_s^\top \bar{\mathbf{f}}_i. \quad (11.1)$$

Note carefully that though the shifted image $\bar{\mathbf{f}}_i$ has mn entries, the projection $\hat{\mathbf{f}}_i$ has only s entries since U_s is $mn \times s$. Likewise, the matrix \hat{F} that has the projections $\hat{\mathbf{f}}_i$ as columns is $s \times k$, and

$$\hat{F} = U_s^\top F. \quad (11.2)$$

Problem 3. In the constructor of `FacialRec`, calculate the compact SVD of \bar{F} and save the matrix U as an attribute. Compare the computed eigenfaces (the columns of U) to Figure 11.2.

Also write a method that accepts a vector of length mn or an $mn \times \ell$ matrix, as well as an integer $s < mn$. Construct U_s by taking the first s columns of U , then use (11.1) or (11.2) to calculate the projection of the input vector or matrix onto the span of the first s eigenfaces. (Hint: this method should be implemented with a single line of code.)

Reducing the mean-shifted face image $\bar{\mathbf{f}}_i$ to the lower-dimensional projection $\hat{\mathbf{f}}_i$ drastically reduces the computational cost of the facial recognition algorithm, but this efficiency gain comes at a price. A projection image only approximates the corresponding original image, but as long as s isn't too small, the approximation is usually good enough for the algorithm to work well. Before completing the facial recognition system, we reconstruct some of these projections to visualize the amount of information lost.

From (11.1), since U_s^\top projects $\bar{\mathbf{f}}_i$ and performs a change of basis to get $\hat{\mathbf{f}}_i$, its transpose U_s puts $\hat{\mathbf{f}}_i$ back into the original basis with as little error as possible. That is,

$$U_s \hat{\mathbf{f}}_i \approx \bar{\mathbf{f}}_i = \mathbf{f}_i - \boldsymbol{\mu},$$

so that we have the approximation

$$\tilde{\mathbf{f}}_i = U_s \hat{\mathbf{f}}_i + \boldsymbol{\mu} \approx \mathbf{f}_i. \quad (11.3)$$

This $\tilde{\mathbf{f}}_i$ is called the reconstruction of \mathbf{f}_i .



(a) A reconstruction with $s = 5$. (b) A reconstruction with $s = 19$. (c) A reconstruction with $s = 75$.

Figure 11.3: An image rebuilt with various numbers of eigenfaces. The image is already recognizable when it is reconstructed with only 19 eigenfaces, less than an eighth of the 153 eigenfaces corresponding to nonzero eigenvalues or $\bar{F}\bar{F}^T$. Note the similarities between this method and regular image compression via the truncated SVD.

Problem 4. Instantiate a `FacialRec` object that draws from the `faces94` dataset. Select one of the shifted images $\bar{\mathbf{f}}_i$. For at least 4 values of s , use your method from Problem 3 to compute the corresponding s -projection $\hat{\mathbf{f}}_i$, then use (11.3) to compute the reconstruction $\tilde{\mathbf{f}}_i$. Display the various reconstructions and the original image. Compare your results to Figure 11.3

Matching Faces

Let \mathbf{g} be a vector representing an unknown face that is not part of the database. We determine which image in the database is most like \mathbf{g} by comparing $\hat{\mathbf{g}}$ to each of the $\hat{\mathbf{f}}_i$. First, shift \mathbf{g} by the mean to obtain $\bar{\mathbf{g}}$, then project $\bar{\mathbf{g}}$ using a given number of eigenfaces:

$$\hat{\mathbf{g}} = U_s^T \bar{\mathbf{g}} = U_s^T (\mathbf{g} - \boldsymbol{\mu}) \quad (11.4)$$

Next, determine which $\hat{\mathbf{f}}_i$ is closest to $\hat{\mathbf{g}}$. Since the columns of U_s are an orthonormal basis, the computation in this basis yields the same result as computing in the standard Euclidean basis would. Then setting

$$j = \underset{i}{\operatorname{argmin}} \|\hat{\mathbf{f}}_i - \hat{\mathbf{g}}\|_2, \quad (11.5)$$

we have that the j th face image \mathbf{f}_j is the best match for \mathbf{g} . Again, since $\hat{\mathbf{f}}_i$ and $\hat{\mathbf{g}}$ only have s entries, the computation in (11.5) is much cheaper than comparing the raw \mathbf{f}_i to \mathbf{g} .

Problem 5. Write a method for the `FacialRec` class that accepts an image vector \mathbf{g} and an integer s . Use your method from Problem 3 to compute \hat{F} and $\hat{\mathbf{g}}$ for the given s , then use (11.5) to determine the best matching face in the database. Return the index of the matching face. (Hint: `scipy.linalg.norm()` and `np.argmin()` may be useful.)

Note

This facial recognition system works by solving a nearest neighbor search, since the goal is to find the \mathbf{f}_i that is “nearest” to the input image \mathbf{g} . Nearest neighbor searches can be performed more efficiently with the use of a k -d tree, a binary search tree for storing vectors. The system could also be called a k -neighbors classifier with $k = 1$.

Problem 6. Write a method for the `FacialRec` class that accepts an flat image vector \mathbf{g} , an integer s , and the original dimensions of \mathbf{g} . Use your method from Problem 5 to find the index j of the best matching face, then display the original face \mathbf{g} alongside the best match \mathbf{f}_j .

The following generator yields random faces from `faces94` that can be used as test cases.

```
def sample_faces(num_faces, path=".//faces94"):
    # Get the list of possible images.
    files = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":      # Only get jpg images.
                files.append(dirpath+"/"+fname)

    # Get a subset of the image names and yield the images one at a time.
    test_files = np.random.choice(files, num_faces, replace=False)
    for fname in test_files:
        yield np.ravel(imread(fname, as_gray=True))
```

The `yield` keyword is like a `return` statement, but the next time the generator is called, it will resume immediately after the last `yield` statement.^a

Use `sample_faces()` to get at least 5 random faces from `faces94`, and match each random face to the database with $s = 38$. Iterate through the random faces with the following syntax.

```
for test_image in sample_faces(5):
    # 'test_image' is a now flattened face vector.
```

^aSee the Python Essentials lab on Profiling for more on generators.

Although there are other approaches to facial recognition that utilize more complex techniques, the method of eigenfaces remains a wonderfully simple and effective solution.

Additional Material

Improvements on the Facial Recognition System with Eigenfaces

The `FacialRec` class does its job well, but it could be improved in several ways. Here are a few ideas.

- The most computationally intensive part of the algorithm is computing \hat{F} . Instead of recomputing \hat{F} every time the method from Problem 5 is called, store \hat{F} and s as attributes the first time the method is called. In subsequent calls, only recompute \hat{F} if the user specifies a different value for s .
- Load a `scipy.spatial.KDTree` object with \hat{F} and use its `query()` method to compute (11.5). Building a kd-tree is expensive, so be sure to only build a new tree when necessary (i.e., the user specifies a new value for s).
- Include an error tolerance ε in the method for Problem 5. If $\|\mathbf{f}_j - \mathbf{g}\| > \varepsilon$, print a message or raise an exception to indicate that there is no suitable match for \mathbf{g} in the database. In this case, add \mathbf{g} to the database for future reference.
- Generalize the system by turning it into a k -neighbors classifier. In the constructor, add several faces per person to the database (this requires modifying `get_faces()`). Assign each individual a unique ID so that the system knows which faces correspond to the same person. Modify the method from Problem 5 so that it also accepts an integer k , then use `scipy.spatial.KDTree` to find the k nearest images to \mathbf{g} . Choose the ID that belongs to the most nearest neighbors, then return an index that corresponds to an individual with that ID.

In other words, choose the k faces \mathbf{f}_i that give the smallest values of $\|\mathbf{f}_i - \hat{\mathbf{g}}\|_2$. These faces then get to vote on which person \mathbf{g} belongs to.

- Improve the user interface of the class by modifying the method from Problem 6 so that it accepts a file name to read from instead of an array. A few lines of code from `get_faces()` or `sample_faces()` might be helpful for this.

Other Methods for Facial Recognition

The method of facial recognition presented here is more formally called principal component analysis (PCA) using eigenfaces. Several other machine learning and optimization techniques, such as linear discriminant analysis (LDA), elastic matching, dynamic link matching, and hidden Markov models (HMMs) have also been applied to the facial recognition problem. Other techniques focus on getting better information about the faces in the first place, the most prevalent being 3-dimensional recognition and thermal imaging. See https://en.wikipedia.org/wiki/Facial_recognition_system for a good survey of different approaches to the facial recognition problem.

12

Data Visualization

Lab Objective: This lab demonstrates how to communicate information through clean, concise, and honest data visualization. We recommend completing the exercises in a Jupyter Notebook.

The Importance of Visualizations

Visualizations of data can reveal insights that are not immediately obvious from simple statistics. The data set in the following exercise is known as Anscombe's quartet. It is famous for demonstrating the importance of data visualization.

Problem 1. The file `anscombe.npy` contains the quartet of data points shown in the table below. For each section of the quartet,

- Plot the data as a scatter plot on the box $[0, 20] \times [0, 13]$.
- Use `scipy.stats.linregress()` to calculate the slope and intercept of the least squares regression line for the data and its correlation coefficient (the first three return values).
- Plot the least squares regression line over the scatter plot on the domain $x \in [0, 20]$.
- Report (print) the mean and variance in x and y , the slope and intercept of the regression line, and the correlation coefficient. Compare these statistics to those of the other sections.
- Describe how the section is similar to the others and how it is different.

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Improving Specific Types of Visualizations

Effective data visualizations show specific comparisons and relationships in the data. Before designing a visualization, decide what to look for or what needs to be communicated. Then choose the visual scheme that makes sense for the data. The following sections demonstrate how to improve commonly used plots to communicate information visually.

Line Plots

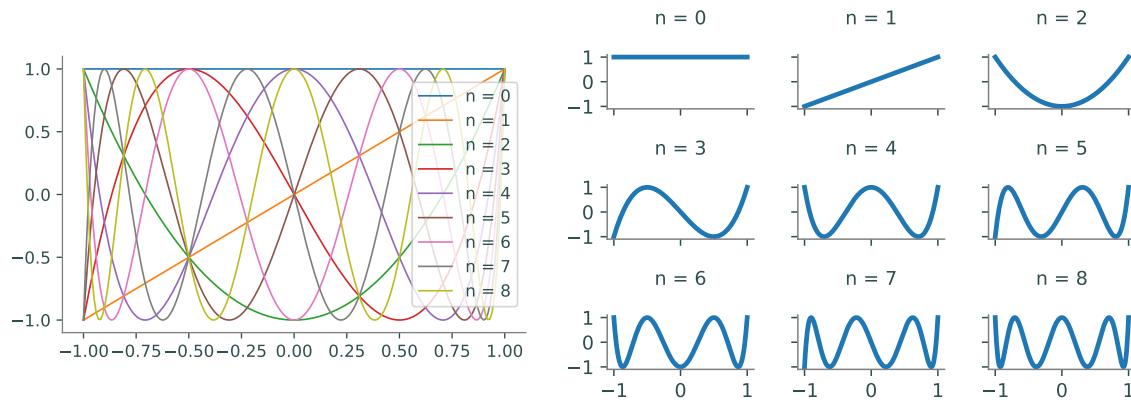


Figure 12.1: Line plots can be used to visualize and compare mathematical functions. For example, this figure shows the first nine Chebyshev polynomials in one plot (left) and small multiples (right). Using small multiples makes comparison easy and shows how each polynomial changes as n increases.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

# Plot the first 9 Chebyshev polynomials in the same plot.
>>> T = np.polynomial.Chebyshev.basis
>>> x = np.linspace(-1, 1, 200)
>>> for n in range(9):
...     plt.plot(x, T(n)(x), label="n = "+str(n))
...
>>> plt.axis([-1.1, 1.1, -1.1, 1.1])           # Set the window limits.
>>> plt.legend(loc="right")
>>> plt.show()
```

A line plot connects ordered (x, y) points with straight lines, and is best for visualizing one or two ordered arrays, such as functional outputs over an ordered domain or a sequence of values over time. Sometimes, plotting multiple lines on the same plot helps the viewer compare two different data sets. However, plotting several lines on top of each other makes the visualization difficult to read, even with a legend. For example, Figure 12.1 shows the first nine Chebyshev polynomials, a family of orthogonal polynomials that satisfies the recursive relation

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{n+1} = 2xT_n(x) - T_{n-1}(x).$$

The plot on the right makes comparison easier by using small multiples. Instead of using a legend, the figure makes a separate subplot with a title for each polynomial. Adjusting the figure size and the line thickness also makes the information easier to read.

Note

Matplotlib titles and annotations can be formatted with L^AT_EX, a system for creating technical documents.^a To do so, use an `r` before the string quotation mark and surround the text with dollar signs. For example, add the following line of code to the loop from the previous example.

```
...     plt.title(r"$T_{\{n\}}(x)".format(n))
```

The `format()` method inserts the input `n` at the curly braces. The title of the sixth subplot, instead of being “`n = 5`,” will then be “ $T_5(x)$.”

^aSee <http://www.latex-project.org/> for more information.

Problem 2. The $n + 1$ Bernstein basis polynomials of degree n are defined as follows:

$$b_{v,n}(x) = \binom{n}{v} x^v (1-x)^{n-v}, \quad v = 0, 1, \dots, n$$

Plot the first 10 Bernstein basis polynomials ($n = 0, 1, 2, 3$) as small multiples on the domain $[0, 1] \times [0, 1]$. Label the subplots for clarity, adjust tick marks and labels for simplicity, and set the window limits of each plot to be the same. Consider arranging the subplots so that the rows correspond with n and the columns with v .

Hint: The constant $\binom{n}{v} = \frac{n!}{v!(n-v)!}$ is called the binomial coefficient and can be efficiently computed with `scipy.special.comb()`.

Bar Charts

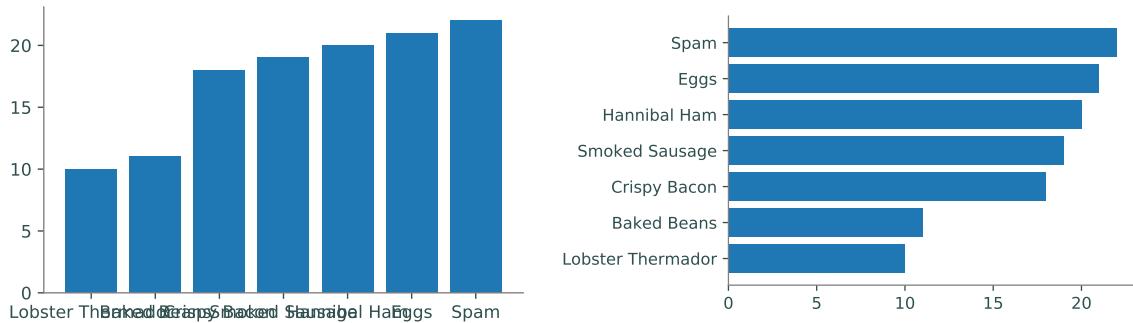


Figure 12.2: Bar charts are used to compare quantities between categorical variables. The labels on the vertical bar chart (left) are more difficult to read than the labels on the horizontal bar chart (right). Although the labels can be rotated, horizontal text is much easier to read than vertical text.

```

>>> labels = ["Lobster Thermador", "Baked Beans", "Crispy Bacon",
...             "Smoked Sausage", "Hannibal Ham", "Eggs", "Spam"]
>>> values = [10, 11, 18, 19, 20, 21, 22]
>>> positions = np.arange(len(labels))

>>> plt.bar(positions, values, align="center") # Vertical bar chart.
>>> plt.xticks(positions, labels)
>>> plt.show()

>>> plt.barr(positions, values, align="center") # Horizontal bar char (better).
>>> plt.yticks(positions, labels)
>>> plt.tight_layout()
>>> plt.show()

```

A bar chart plots categorical data in a sequence of bars. They are best for small, discrete, one-dimensional data sets. In Matplotlib, `plt.bar()` creates a vertical bar chart or `plt.barr()` creates a horizontal bar chart. These functions receive the locations of each bar followed by the height of each bar (as lists or arrays). In most situations, horizontal bar charts are preferable to vertical bar charts because horizontal labels are easier to read than vertical labels. Data in a bar chart should also be sorted in a logical way, such as alphabetically, by size, or by importance.

Histograms

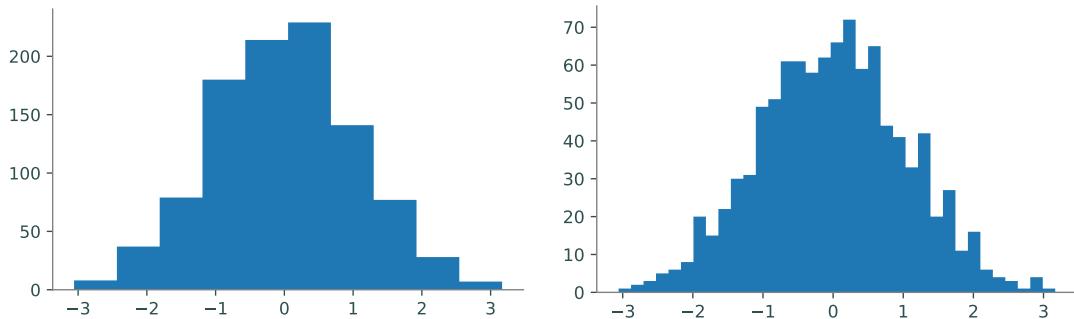


Figure 12.3: Histograms are used to show the distribution of one-dimensional data. Experimenting with different values for the bin size is important when plotting a histogram. Using only 10 bins (left) doesn't give a good sense for how the randomly generated data is distributed. However, using 35 bins (right) reveals the shape of a normal distribution.

```

>>> data = np.random.normal(size=10000)
>>> fig, ax = plt.subplots(1, 2)
>>> ax[0].hist(data, bins=10)
>>> ax[1].hist(data, bins=35)
>>> plt.show()

```

A histogram partitions an interval into a number of bins and counts the number of values that fall into each bin. Histograms are ideal for visualizing how unordered data in a single array is distributed over an interval. For example, if data are drawn from a probability distribution, a histogram approximates the distribution's probability density function. Use `plt.hist()` to create a histogram. The arguments `bins` and `range` specify the number of bins to draw and over what domain. A histogram with too few or too many bins will not give a clear view of the distribution.

Scatter Plots

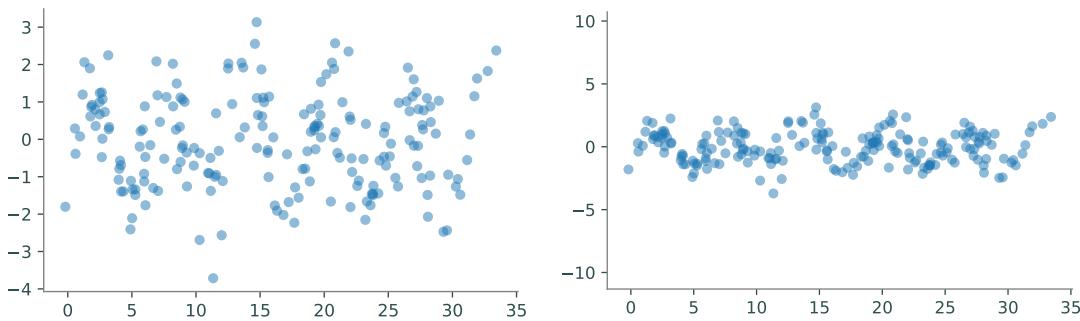


Figure 12.4: Scatter plots show correlations between variables by plotting markers at coordinate points. The figure above displays randomly perturbed data that is visualized using two scatter plots with `alpha=.5` and `edgecolor='none'`. The default (left) makes it harder to see correlation and pattern whereas making the axes equal better reveals the oscillatory behavior in the perturbed sine wave.

```
>>> np.random.seed(0)
>>> x = np.linspace(0,10*np.pi,200) + np.random.normal(size=200)
>>> y = np.sin(x) + np.random.normal(size=200)

>>> plt.scatter(x, y, alpha=.5, edgecolor='none')
>>> plt.show()

>>> plt.scatter(x, y, alpha=.5, edgecolor='none')
>>> plt.axis('equal')
>>> plt.show()
```

A scatter plot draws (x, y) points without connecting them. Scatter plots are best for displaying data sets without a natural order, or where each point is a distinct, individual instance. They are frequently used to show correlation between variables in a data set. Use `plt.scatter()` to create a scatter plot.¹

¹Scatter plots can also be drawn with `plt.plot()` by specifying a point marker such as `'.'`, `'|'`, `'o'`, or `'+'`. The keywords `markersize` and `color` can be used to change the marker size and marker color, respectively.

Similar data points in a scatter plot may overlap, as in Figure 12.4. Specifying an alpha value reveals overlapping data by making the markers transparent (see Figure 12.5 for an example). The keyword `alpha` accepts values between 0 (completely transparent) and 1 (completely opaque). When plotting lots of overlapping points, the outlines on the markers can make the visualization look cluttered. Setting the `edgecolor` keyword to zero removes the outline and improves the visualization.

Problem 3. The file `MLB.npy` contains measurements from over 1,000 recent Major League Baseball players, compiled by UCLA.^a Each row in the array represents a player; the columns are the player's height (in inches), weight (in pounds), and age (in years), in that order.

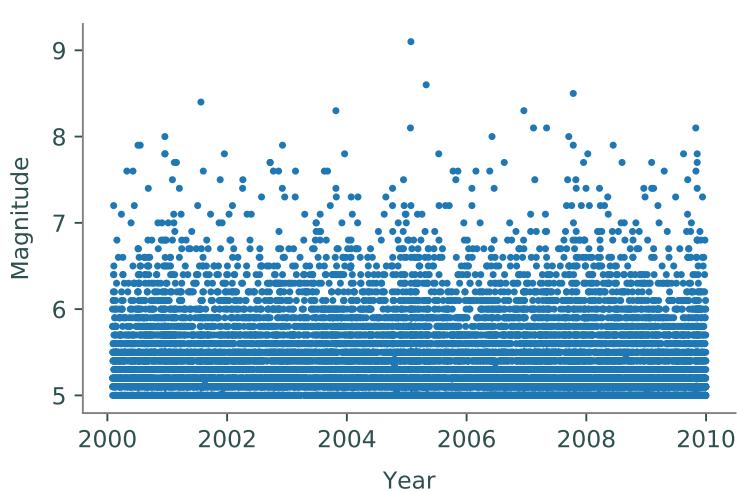
Create several visualizations to show the correlations between height, weight, and age in the MLB data set. Use at least one scatter plot. Adjust the marker size, plot a regression line, change the window limits, and use small multiples where appropriate.

^aSee http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_MLB_HeightsWeights.

Problem 4. The file `earthquakes.npy` contains data from over 17,000 earthquakes between 2000 and 2010 that were at least a 5 on the Richter scale.^a Each row in the array represents an earthquake; the columns are the earthquake's date (as a fraction of the year), magnitude (on the Richter scale), longitude, and latitude, in that order.

Because each earthquake is a distinct event, a good way to start visualizing this data might be a scatter plot of the years versus the magnitudes of each earthquake.

```
>>> year, magnitude, longitude, latitude = np.load("earthquakes.npy").T
>>> plt.plot(year, magnitude, '.')
>>> plt.xlabel("Year")
>>> plt.ylabel("Magnitude")
```



Unfortunately, this plot communicates very little information because the data is so cluttered. Describe the data with at least two better visualizations. Include line plots, scatter plots, and histograms as appropriate. Your plots should answer the following questions:

1. How many earthquakes happened every year?
2. How often do stronger earthquakes happen compared to weaker ones?
3. Where do earthquakes happen? Where do the strongest earthquakes happen?
(Hint: Use `plt.axis("equal")` or `ax.set_aspect("equal")` to fix the aspect ratio, which may improve comparisons between longitude and latitude.)

^aSee <http://earthquake.usgs.gov/earthquakes/search/>.

Hexbins

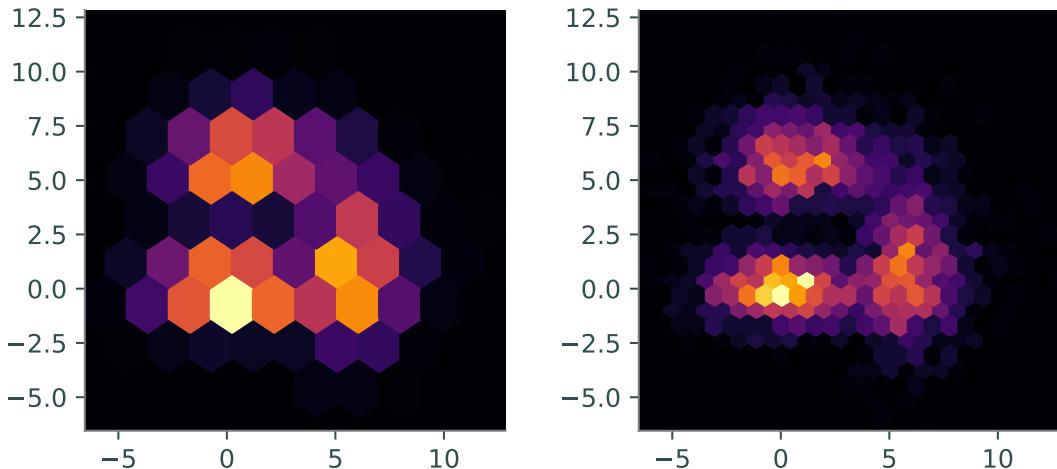


Figure 12.5: Hexbins can be used instead of using a three-dimensional histogram to show the distribution of two-dimensional data. Choosing the right `gridsize` will give a better picture of the distribution. The figure above shows random data plotted as hexbins with a `gridsize` of 10 (left) and 25 (right). Hexbins use color to show height via a colormap and both histograms above use the '`inferno`' colormap.

```
# Add random draws from various distributions in two dimensions.
>>> a = np.random.exponential(size=1000) + np.random.normal(size=1000) + 5
>>> b = np.random.exponential(size=1000) + 2*np.random.normal(size=1000)
>>> x = np.hstack((a, b, 2*np.random.normal(size=1000)))
>>> y = np.hstack((b, a, np.random.normal(size=1000)))

# Plot the samples with hexbins of gridsize 10 and 25.
>>> fig, axes = plt.subplots(1, 2)
>>> window = [x.min(), x.max(), y.min(), y.max()]
>>> for ax, size in zip(axes, [10, 25]):
```

```

...     ax.hexbin(x, y, gridsize=size, cmap='inferno')
...     ax.axis(window)
...     ax.set_aspect("equal")
...
>>> plt.show()

```

A hexbin is a way of representing the frequency of occurrences in a two-dimensional plane. Similar to a histogram, which sorts one-dimensional data into bins, a hexbin sorts two-dimensional data into hexagonal bins arranged in a grid and uses color instead of height to show frequency. Creating an effective hexbin relies on choosing an appropriate `gridsize` and colormap. The colormap is a function that assigns data points to an ordering of colors. Use `plt.hexbin()` to create a hexbin and use the `cmap` keyword to specify the colormap.

Heat Maps and Contour Plots

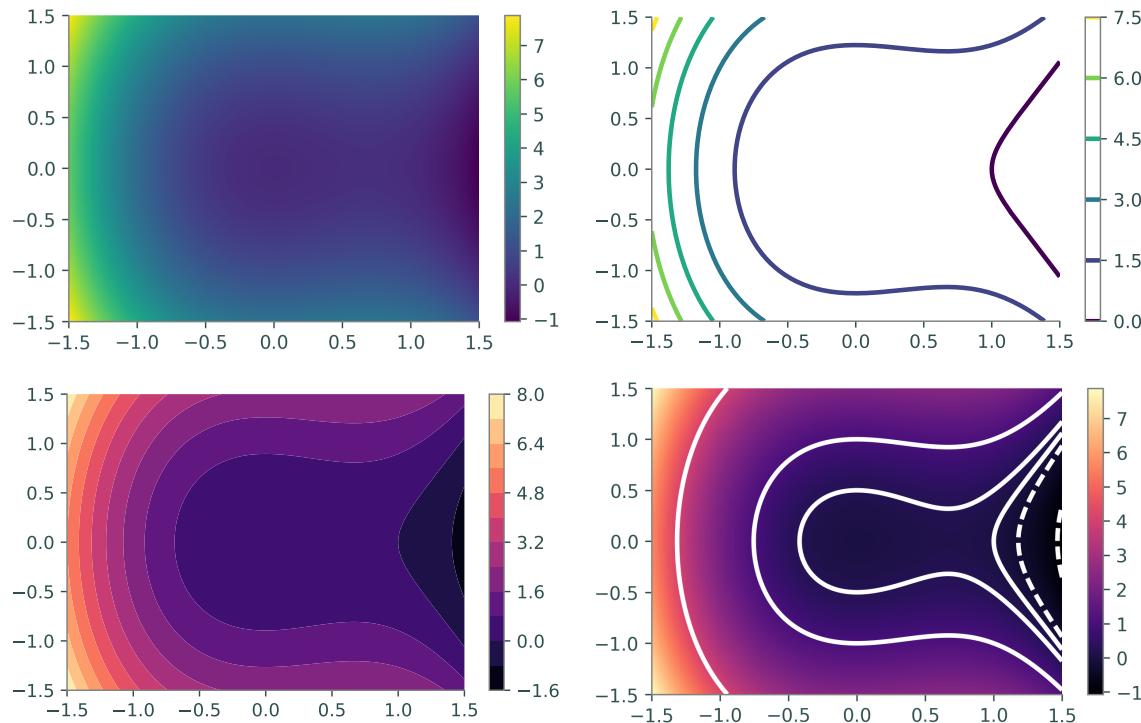


Figure 12.6: Heat maps visualize three-dimensional functions or surfaces by using color to represent the value in one dimension. With continuous data, it can be hard to identify regions of interest. Contour plots solve this problem by visualizing the level curves of the surface. Top left: heat map. Top right: contour plot. Bottom left: heat map. Bottom right: contours plotted on a heat map.

```

# Construct a 2-D domain with np.meshgrid() and calculate f on the domain.
>>> x = np.linspace(-1.5, 1.5, 200)
>>> X, Y = np.meshgrid(x, x)
>>> Z = Y**2 - X**3 + X**2

```

```
# Plot f using a heat map, a contour map, and a filled contour map.
>>> fig, ax = plt.subplots(2,2)
>>> ax[0,0].pcolormesh(X, Y, Z, cmap="viridis")      # Heat map.
>>> ax[0,1].contour(X, Y, Z, 6, cmap="viridis")      # Contour map.
>>> ax[1,0].contourf(X, Y, Z, 12, cmap="magma")    # Filled contour map.

# Plot specific level curves and a heat map with a colorbar.
>>> ax[1,1].contour(X, Y, Z, [-1, -.25, 0, .25, 1, 4], colors="white")
>>> cax = ax[1,1].pcolormesh(X, Y, Z, cmap="magma")
>>> fig.colorbar(cax, ax=ax[1,1])

>>> plt.show()
```

Let $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ be a scalar-valued function on a 2-dimensional domain. A heat map of f assigns a color to each (x, y) point in the domain based on the value of $f(x, y)$, while a contour plot is a drawing of the level curves of f . The level curve corresponding to the constant c is the set $\{(x, y) \mid c = f(x, y)\}$. A filled contour plot colors in the sections between the level curves and is a discretized version of a heat map. The values of c corresponding to the level curves are automatically chosen to be evenly spaced over the range of values of f on the domain. However, it is sometimes better to strategically specify the curves by providing a list of c constants.

Consider the function $f(x, y) = y^2 - x^3 + x^2$ on the domain $[-\frac{3}{2}, \frac{3}{2}] \times [-\frac{3}{2}, \frac{3}{2}]$. A heat map of f reveals that it has a large basin around the origin. Since $f(0, 0) = 0$, choosing several level curves close to 0 more closely describes the topography of the basin. The fourth subplot in 12.6 uses the curves with $c = -1, -\frac{1}{4}, 0, \frac{1}{4}, 1$, and 4.

When plotting hexbins, heat maps, and contour plots, be sure to choose a colormap that best represents the data. Avoid using spectral or rainbow colormaps like "`jet`" because they are not perceptually uniform, meaning that the rate of change in color is not constant. Because of this, data points may appear to be closer together or farther apart than they actually are. This creates visual false positives or false negatives in the visualization and can affect the interpretation of the data. As a default, we recommend using the sequential colormaps "`viridis`" or "`inferno`" because they are designed to be perceptually uniform and colorblind friendly. For the complete list of Matplotlib color maps, see http://matplotlib.org/examples/color/colormaps_reference.html.

Problem 5. The Rosenbrock function is defined as

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2.$$

The minimum value of f is 0, which occurs at the point $(1, 1)$ at the bottom of a steep, banana-shaped valley of the function.

Use a heat map and a contour plot to visualize the Rosenbrock function. Also plot the minimizer $(1, 1)$. Use a different sequential colormap for each visualization.

Best Practices

Good scientific visualizations make comparison easy and clear. The eye is very good at detecting variation in one dimension and poor in two or more dimensions. For example, consider Figure 12.7. Despite the difficulty, most people can probably guess which slice of a pie chart is the largest or smallest. However, it's almost impossible to confidently answer the question by how much? The bar charts may not be as aesthetically pleasing but they make it much easier to precisely compare the data. Avoid using pie charts as well as other visualizations that make accurate comparison difficult, such as radar charts, bubble charts, and stacked bar charts.

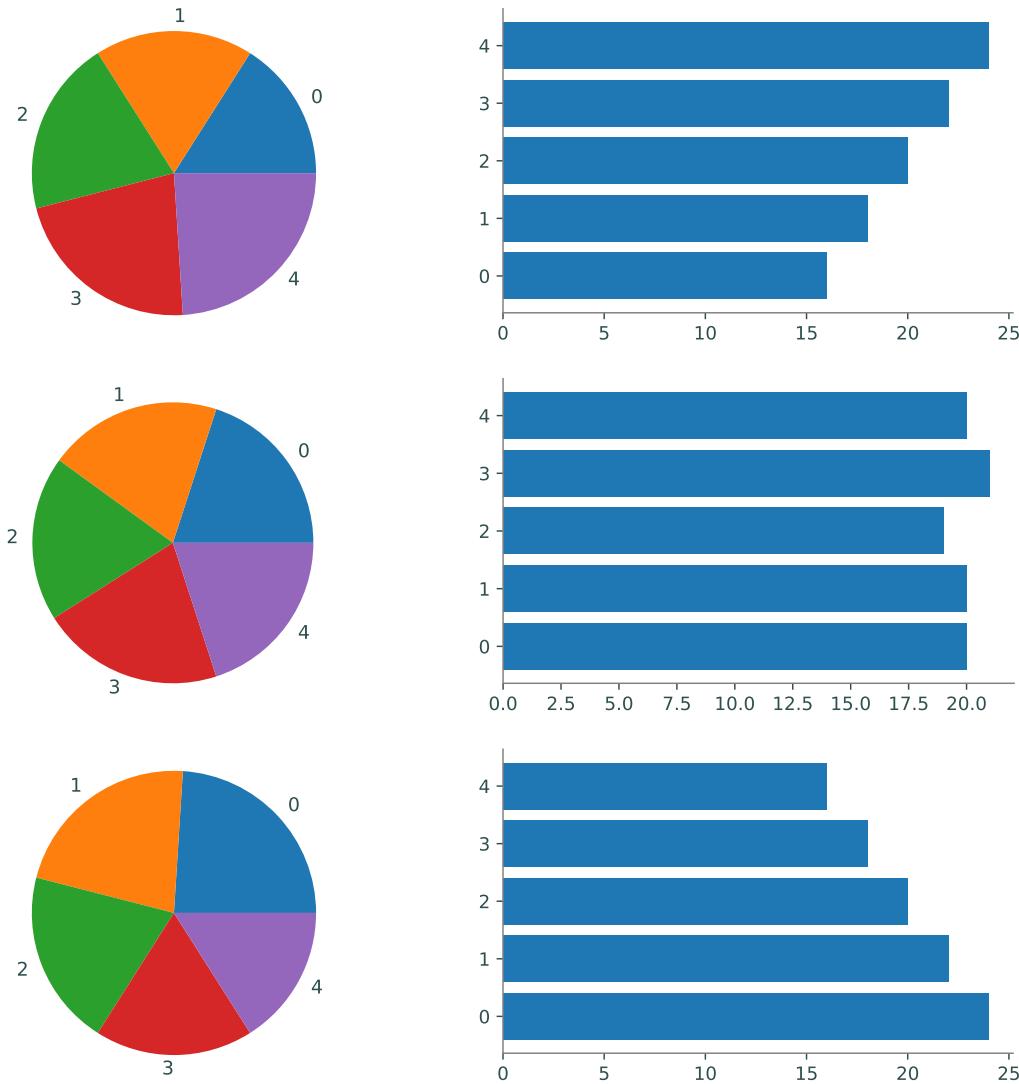


Figure 12.7: The pie charts on the left may be more colorful but it's extremely difficult to quantify the difference between each slice. Instead, the horizontal bar charts on the right make it very easy to see the difference between each variable.

No visualization perfectly represents data, but some are better than others. Finding the best visualization for a data set is an iterative process. Experiment with different visualizations by adjusting their parameters: color, scale, size, shape, position, and length. It may be necessary to use a data transformation or visualize various subsets of the data. As you iterate, keep in mind the saying attributed to George Box: “All models are wrong, but some are useful.” Do whatever is needed to make the visualization useful and effective.

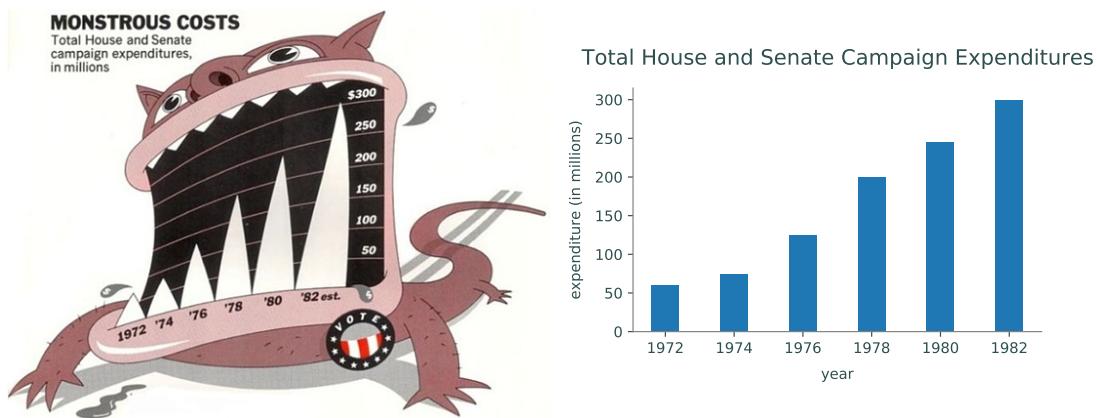


Figure 12.8: Chartjunk refers to anything that does not communicate data. In the image on the left, the cartoon monster distorts the bar chart and manipulates the feelings of the viewer to think negatively about the results. The image on the right shows the same data without chartjunk, making it simple and very easy to interpret the data objectively.

Good visualizations are as simple as possible and no simpler. Edward Tufte coined the term chartjunk to mean anything (pictures, icons, colors, and text) that does not represent data or is distracting. Though chartjunk might appear to make data graphics more memorable than plain visualizations, **it is more important to be clear and precise in order to prevent misinterpretation**. The physicist Richard Feynman said, “For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.” Remove chartjunk and anything that prevents the viewer from objectively interpreting the data.

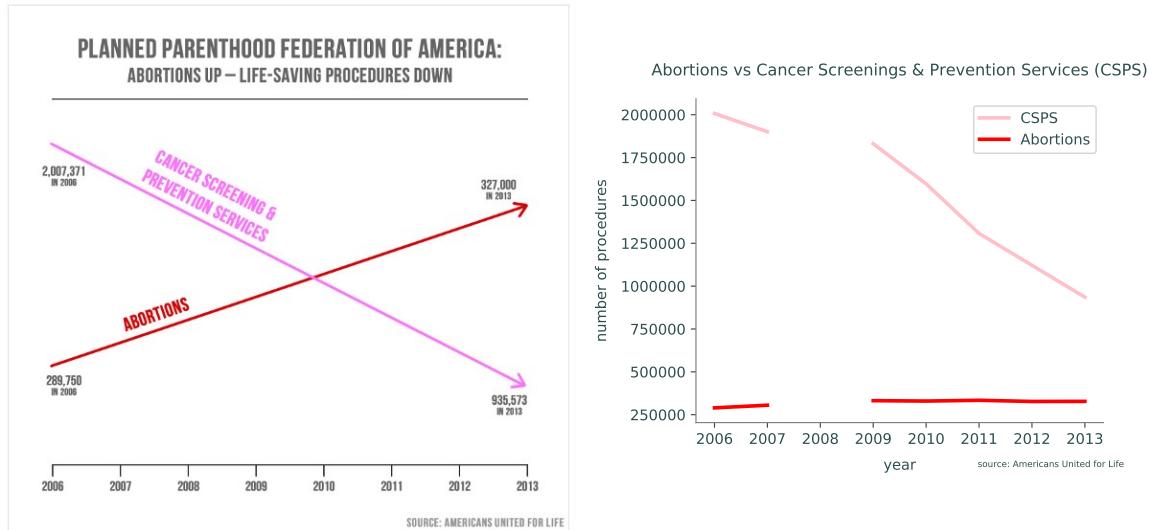


Figure 12.9: The chart on the left is an example of a dishonest graphic shown at a United States congressional hearing in 2015. The chart on the right shows a more accurate representation of the data by showing the y-axis and revealing the missing data from 2008. Source: PolitiFact.

Visualizations should be honest. Figure 12.9 shows how visualizations can be dishonest. The misleading graphic on the left was used as evidence in a United States congressional hearing in 2015. With the *y*-axis completely removed, it is easy to miss that each line is shown on a different *y*-axis even though they are measured in the same units. Furthermore, the chart fails to indicate that data is missing from the year 2008. The graphic on the right shows a more accurate representation of the data.²

Never use data visualizations to deceive or manipulate. Always present information on who created it, where the data came from, how it was collected, whether it was cleaned or transformed, and whether there are conflicts of interest or possible biases present. Use specific titles and axis labels, and include units of measure. Choose an appropriate window size and use a legend or other annotations where appropriate.

Problem 6. The file `countries.npy` contains information from 20 different countries. Each row in the array represents a different country; the columns are the 2015 population (in millions of people), the 2015 GDP (in billions of US dollars), the average male height (in centimeters), and the average female height (in centimeters), in that order.^a

The countries corresponding are listed below in order.

```
countries = ["Austria", "Bolivia", "Brazil", "China",
             "Finland", "Germany", "Hungary", "India",
             "Japan", "North Korea", "Montenegro", "Norway",
             "Peru", "South Korea", "Sri Lanka", "Switzerland",
             "Turkey", "United Kingdom", "United States", "Vietnam"]
```

^aFor more information about this graphic, visit <http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/>.

Visualize this data set with at least four plots, using at least one scatter plot, one histogram, and one bar chart. List the major insights that your visualizations reveal.
(Hint: consider using `np.argsort()` and fancy indexing to sort the data for the bar chart.)

^aSee [https://en.wikipedia.org/wiki/List_of_countries_by_GDP_\(nominal\)](https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)),
https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population, and
<http://www.averageheight.co/>.

For more about data visualization, we recommend the following books and websites.

- How to Lie with Statistics by Darrell Huff (1954).
- The Visual Display of Quantitative Information by Edward Tufte (2nd edition).
- Visual Explanations by Edward Tufte.
- Envisioning Information by Edward Tufte.
- Beautiful Evidence by Edward Tufte.
- The Functional Art by Alberto Cairo.
- Visualization Analysis and Design by Tamara Munzner.
- Designing New Default Colormaps: <https://bids.github.io/colormap/>.

13

Convolution and Filtering

Lab Objective: The Fourier transform reveals information in the frequency domain about signals and images that might not be apparent in the usual time (sound) or spatial (image) domain. In this lab, we use the discrete Fourier transform to efficiently convolve sound signals and filter out some types of unwanted noise from both sounds and images. This lab is a continuation of the Discrete Fourier Transform lab and should be completed in the same Jupyter Notebook.

Convolution

Mixing two sounds signals—a common procedure in signal processing and analysis—is usually done through a discrete convolution. Given two periodic sound sample vectors \mathbf{f} and \mathbf{g} of length n , the discrete convolution of \mathbf{f} and \mathbf{g} is a vector of length n where the k th component is given by

$$(\mathbf{f} * \mathbf{g})_k = \sum_{j=0}^{n-1} f_{k-j} g_j, \quad k = 0, 1, 2, \dots, n-1. \quad (13.1)$$

Since audio needs to be sampled frequently to create smooth playback, a recording of a song can contain tens of millions of samples; even a one-minute signal has 2,646,000 samples if it is recorded at the standard rate of 44,100 samples per second (44,100 Hz). The naïve method of using the sum in (13.1) n times is $O(n^2)$, which is often too computationally expensive for convolutions of this size.

Fortunately, the discrete Fourier transform (DFT) can be used compute convolutions efficiently. The finite convolution theorem states that the Fourier transform of a convolution is the element-wise product of Fourier transforms:

$$F_n(\mathbf{f} * \mathbf{g}) = n(F_n \mathbf{f}) \odot (F_n \mathbf{g}). \quad (13.2)$$

In other words, convolution in the time domain is equivalent to component-wise multiplication in the frequency domain. Here F_n is the DFT on \mathbb{R}^n , $*$ is discrete convolution, and \odot is component-wise multiplication. Thus, the convolution of \mathbf{f} and \mathbf{g} can be computed by

$$\mathbf{f} * \mathbf{g} = n F_n^{-1}((F_n \mathbf{f}) \odot (F_n \mathbf{g})), \quad (13.3)$$

where F_n^{-1} is the inverse discrete Fourier transform (IDFT). The fast Fourier transform (FFT) puts the cost of (13.3) at $O(n \log n)$, a huge improvement over the naïve method.

Note

Although individual samples are real numbers, results of the IDFT may have small complex components due to rounding errors. These complex components can be safely discarded by taking only the real part of the output of the IDFT.

```
>>> import numpy
>>> from scipy.fftpack import fft, ifft # Fast DFT and IDFT functions.

>>> f = np.random.random(2048)
>>> f_dft_idft = ifft(fft(f)).real           # Keep only the real part.
>>> np.allclose(f, f_dft_idft)               # Check that IDFT(DFT(f)) = f.
True
```

Achtung!

SciPy uses a different convention to define the DFT and IDFT than this and the previous lab, resulting in a slightly different form of the convolution theorem. Writing SciPy's DFT as \hat{F}_n and its IDFT as \hat{F}_n^{-1} , we have $\hat{F}_n = nF_n$, so (13.3) becomes

$$\mathbf{f} * \mathbf{g} = \hat{F}_n^{-1}((\hat{F}_n \mathbf{f}) \odot (\hat{F}_n \mathbf{g})), \quad (13.4)$$

without a factor of n . Use (13.4), not (13.3), when using `fft()` and `ifft()` from `scipy.fftpack`.

Circular Convolution

The definition (13.1) and the identity (13.3) require \mathbf{f} and \mathbf{g} to be periodic vectors. However, the convolution $\mathbf{f} * \mathbf{g}$ can always be computed by simply treating each vector as periodic. The convolution of two raw sample vectors is therefore called the periodic or circular convolution. This strategy mixes sounds from the end of each signal with sounds at the beginning of each signal.

Problem 1.

Implement the `__mul__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A * B` creates a new `SoundWave` object whose samples are the circular convolution of the samples from `A` and `B`. If the samples from `A` and `B` are not the same length, append zeros to the shorter array to make them the same length before convolving. Use `scipy.fftpack` and (13.4) to compute the convolution, and raise a `ValueError` if the sample rates from `A` and `B` are not equal.

A circular convolution creates an interesting effect on a signal when convolved with a segment of white noise: the sound loops seamlessly from the end back to the beginning. To see this, generate two seconds of white noise (at the same sample rate as `tada.wav`) with the following code.

```
>>> rate = 22050      # Create 2 seconds of white noise at a given rate.
>>> white_noise = np.random.randint(-32767, 32767, rate*4, dtype=np.int16)
```

Next, convolve `tada.wav` with the white noise. Finally, use the `>>` operator to append the convolution result to itself. This final signal sounds the same from beginning to end, even though it is the concatenation of two signals.

Linear Convolution

Although circular convolutions can give interesting results, most common sound mixtures do not combine sounds at the beginning of one signal with sounds at the end of another. Whereas circular convolution assumes that the samples represent a full period of a periodic function, linear convolution aims to combine non-periodic discrete signals in a way that prevents the beginnings and endings from interacting. Given two samples with lengths n and m , the simplest way to achieve this is to pad both samples with zeros so that they each have length $n + m - 1$, compute the convolution of these larger arrays, and take the first $n + m - 1$ entries of that convolution.

Problem 2.

Implement the `__pow__()` magic method for the `SoundWave` class so that if `A` and `B` are `SoundWave` instances, `A ** B` creates a new `SoundWave` object whose samples are the linear convolution of the samples from `A` and `B`. Raise a `ValueError` if the sample rates from `A` and `B` are not equal.

Because `scipy.signal.fftconvolve()` performs best when the length of the inputs is a power of 2, start by computing the smallest 2^a such that $2^a \geq n + m - 1$, where $a \in \mathbb{N}$ and n and m are the number of samples from `A` and `B`, respectively. Append zeros to each sample so that they each have 2^a entries, then compute the convolution of these padded samples using (13.4). Use only the first $n + m - 1$ entries of this convolution as the samples of the returned `SoundWave` object.

To test your method, read `CGC.wav` and `GCG.wav`. Time (separately) the convolution of these signals with `SoundWave.__pow__()` and with `scipy.signal.fftconvolve()`. Compare the results by listening to the original and convolved signals.

Problem 3. Clapping in a large room with an echo produces a sound that resonates in the room for up to several seconds. This echoing sound is referred to as the impulse response of the room, and is a way of approximating the acoustics of a room. When the sound of a single instrument in a carpeted room is convolved with the impulse response from a concert hall, the new signal sounds as if the instrument is being played in the concert hall.

The file `chopin.wav` contains a short clip of a piano being played in a room with little or no echo, and `balloon.wav` is a recording of a balloon being popped in a room with a substantial echo (the impulse). Use your method from Problem 2 or `scipy.signal.fftconvolve()` to compute the linear convolution of `chopin.wav` and `balloon.wav`.

Filtering Frequencies with the DFT

The DFT also provides a way to clean a signal by altering some of its frequencies. Consider `noisy1.wav`, a noisy recording of a short voice clip. The time-domain plot of the signal only shows that the signal has a lot of static. On the other hand, the signal's DFT suggests that the static may be the result of some concentrated noise between about 1250–2600 Hz. Removing these frequencies could result in a much cleaner signal.

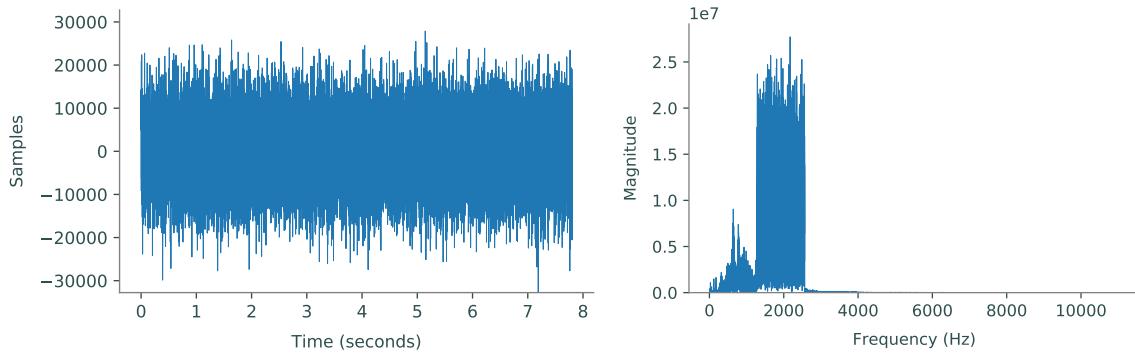


Figure 13.1: The time-domain plot (left) and DFT (right) of `noisy1.wav`.

To implement this idea, recall that the k th entry of the DFT array $\mathbf{c} = F_n \mathbf{f}$ corresponds to the frequency $v = kr/n$ in Hertz, where r is the sample rate and n is the number of samples. Hence, the DFT entry c_k corresponding to a given frequency v in Hertz has index $k = vn/r$, rounded to an integer if needed. In addition, since the DFT is symmetric, c_{n-k} also corresponds to this frequency. This suggests a strategy for filtering out an unwanted interval of frequencies $[v_{\text{low}}, v_{\text{high}}]$ from a signal:

1. Compute the integer indices k_{low} and k_{high} corresponding to v_{low} and v_{high} , respectively.
2. Set the entries of the signal's DFT from k_{low} to k_{high} and from $n - k_{\text{high}}$ to $n - k_{\text{low}}$ to zero, effectively removing those frequencies from the signal.
3. Take the IDFT of the modified DFT to obtain the cleaned signal.

Using this strategy to filter `noisy1.wav` results in a much cleaner signal. However, any “good” frequencies in the affected range are also removed, which may decrease the overall sound quality. The goal, then, is to remove only as many frequencies as necessary.

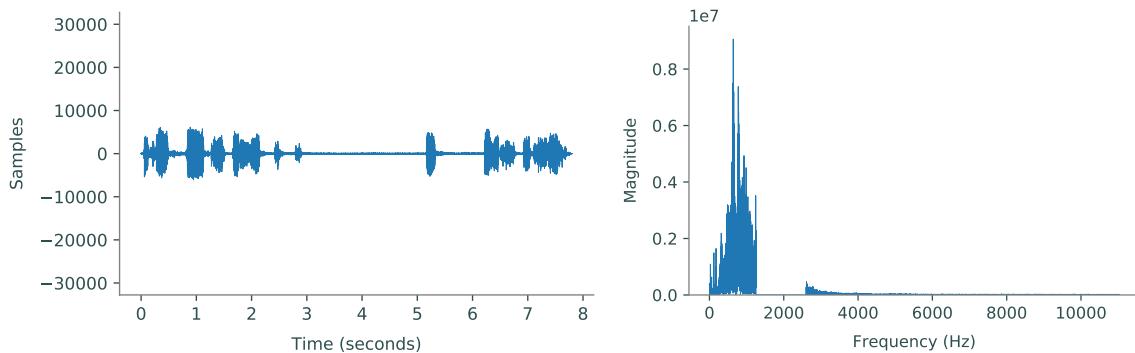


Figure 13.2: The time-domain plot (left) and DFT (right) of `noisy1.wav` after being cleaned.

Problem 4. Add a method to the `SoundWave` class that accepts two frequencies v_{low} and v_{high} in Hertz. Compute the DFT of the stored samples and zero out the frequencies in the range $[v_{\text{low}}, v_{\text{high}}]$ (remember to account for the symmetry DFT). Take the IDFT of the altered array and store it as the sample array.

Test your method by cleaning `noisy1.wav`, then clean `noisy2.wav`, which also has some artificial noise that obscures the intended sound.

(Hint: plot the DFT of `noisy2.wav` to determine which frequencies to eliminate.)

A digital audio signal made of a single sample vector with is called monoaural or mono. When several sample vectors with the same sample rate and number of samples are combined into a matrix, the overall signal is called stereophonic or stereo. This allows multiple speakers to each play one channel—one of the original sample vectors—simultaneously. “Stereo” usually means there are two channels, but there may be any number of channels (5.1 surround sound, for instance, has five).

Most stereo sounds are read as $n \times m$ matrices, where n is the number of samples and m is the number of channels (i.e., each column is a channel). However, some functions, including Jupyter’s embedding tool `IPython.display.Audio()`, receive stereo signals as $m \times n$ matrices (each row is a channel). Be aware that both conventions are common.

Problem 5. During the 2010 World Cup in South Africa, large plastic horns called vuvuzelas were blown excessively throughout the games. Broadcasting organizations faced difficulties with their programs due to the incessant noise level. Eventually, audio filtering techniques were used to cancel out the sound of the vuvuzela, which has a frequency of around 200–500 Hz.

The file `vuvuzela.wav`^a is a stereo sound with two channels. Use your function from Problem 4 to clean the sound clip by filtering out the vuvuzela frequencies in each channel. Recombine the two cleaned samples.

^aSee https://www.youtube.com/watch?v=g_0NoBKWCt8.

The Two-dimensional Discrete Fourier Transform

The DFT can be easily extended to any number of dimensions. Computationally, the problem reduces to performing the usual one-dimensional DFT iteratively along each of the dimensions. For example, to compute the two-dimensional DFT of an $m \times n$ matrix, calculate the usual DFT of each of the n columns, then take the DFT of each of the m rows of the resulting matrix. Calculating the two-dimensional IDFT is done in a similar fashion, but in reverse order: first calculate the IDFT of the rows, then the IDFT of the resulting columns.

```
>>> from scipy.fftpack import fft2, ifft2  
  
>>> A = np.random.random((10,10))  
>>> A_dft = fft2(A)                      # Calculate the 2d DFT of A.  
>>> A_dft_ifft = ifft2(A_dft).real      # Calculate the 2d IDFT.  
>>> np.allclose(A, A_dft_ifft)  
True
```

Just as the one-dimensional DFT can be used to remove noise in sounds, its two-dimensional counterpart can be used to remove “noise” in images. The procedure is similar to the filtering technique in Problems 4 and 5: take the two-dimensional DFT of the image matrix, modify certain entries of the DFT matrix to remove unwanted frequencies, then take the IDFT to get a cleaner version of the original image. This strategy makes the fairly strong assumption that the noise in the image is periodic and corresponds to certain frequencies. While this may seem like an unlikely scenario, it does actually occur in many digital images—for an example, try taking a picture of a computer screen with a digital camera.

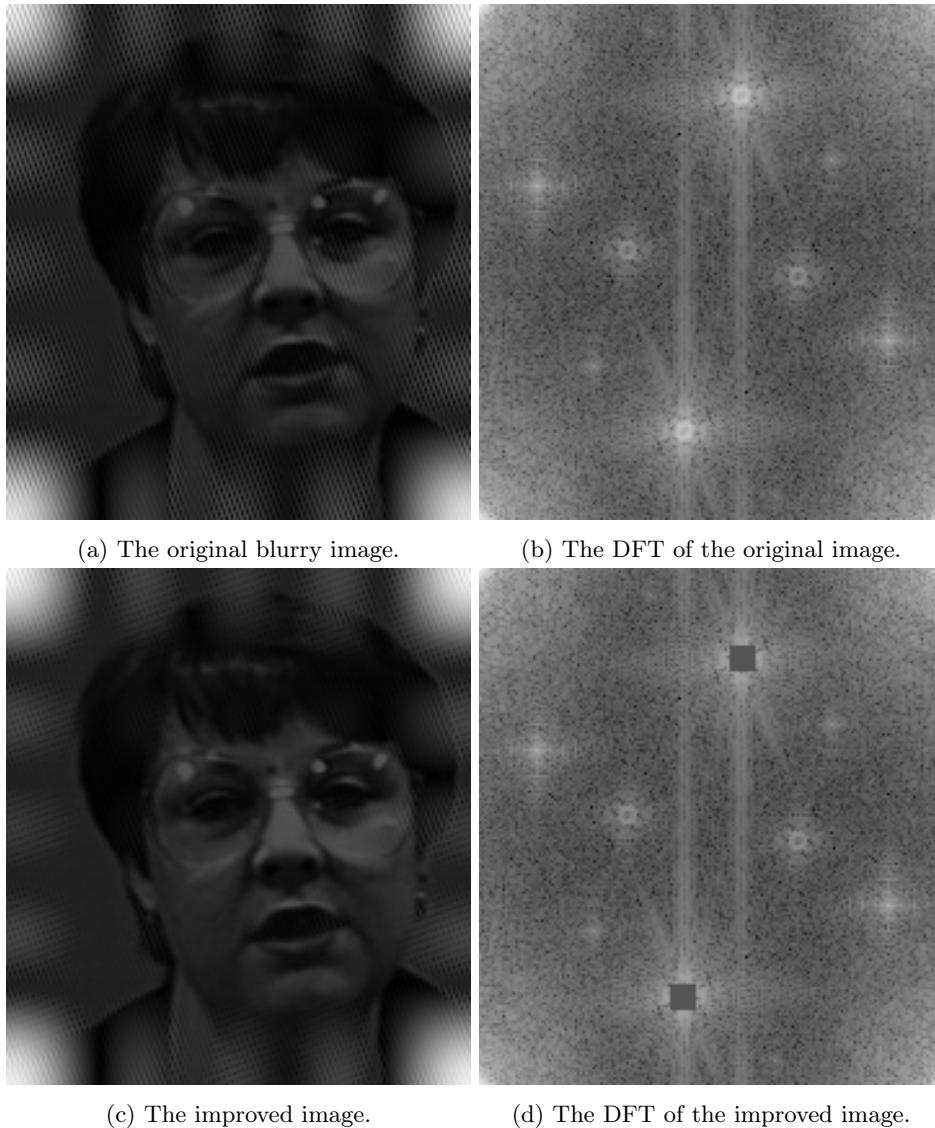


Figure 13.3: To remove noise from an image, take the DFT of the image and replace the abnormalities with values more consistent with the rest of the DFT. Notice that the new image is less noisy, but only slightly. This is because only some of the abnormalities in the DFT were changed; in order to further decrease the noise, we would need to further alter the DFT.

To begin cleaning an image with the DFT, take the two-dimensional DFT of the image matrix. Identify spikes—abnormally high frequency values that may be causing the noise—in the image DFT by plotting the log of the magnitudes of the Fourier coefficients. With `cmap="gray"`, spikes show up as bright spots. See Figures 13.3a–13.3b.

```
# Read the image.
>>> import imageio
>>> image = imageio.imread("noisy_face.png")

# Plot the log magnitude of the image's DFT.
```

```
>>> im_dft = fft2(image)
>>> plt.imshow(np.log(np.abs(im_dft)), cmap="gray")
>>> plt.show()
```

Instead of setting spike frequencies to zero (as was the case for sounds), replace them with values that are similar to those around them. There are many ways to do this, but one convention is to simply “patch” each spike by setting portions of the DFT matrix to some set value, such as the mean of the DFT array. See Figure 13.3d.

Once the spikes have been covered, take the IDFT of the modified DFT to get a (hopefully cleaner) image. Notice that Figure 13.3c still has noise present, but it is a slight improvement over the original. However, it often suffices to remove some of the noise, even if it is not possible to remove it all with this method.

Problem 6. The file `license_plate.png` contains a noisy image of a license plate. The bottom right corner of the plate has a sticker with information about the month and year that the vehicle registration was renewed. However, in its current state, the year is not clearly legible.

Use the two-dimensional DFT to clean up the image enough so that the year in the bottom right corner is legible. This may require a little trial and error.

14

Introduction to SymPy

Lab Objective: Most implementations of numerical algorithms focus on crunching, relating, or visualizing numerical data. However, it is sometimes convenient or necessary to represent parts of an algorithm symbolically. The SymPy module provides a way to do symbolic mathematics in Python, including algebra, differentiation, integration, and more. In this lab we introduce SymPy syntax and emphasize how to use symbolic algebra for numerical computing.

Symbolic Variables and Expressions

Most variables in Python refer to a number, string, or data structure. Doing computations on such variables results in more numbers, strings, or data structures. A symbolic variable is a variable that represents a mathematical symbol, such as x or θ , not a number or another kind of data. Operating on symbolic variables results in an expression, representative of an actual mathematical expression. For example, if a symbolic variable Y refers to a mathematical variable y , the multiplication $3*Y$ refers to the expression $3y$. This is all done without assigning an actual numerical value to Y .

SymPy [MSP⁺¹⁷] is Python's library for doing symbolic algebra and calculus. It is typically imported with `import sympy as sy`, and symbolic variables are usually defined using `sy.symbols()`.

```
>>> import sympy as sy
>>> x0 = sy.symbols('x0')                                # Define a single variable.

# Define multiple symbolic variables simultaneously.
>>> x2, x3 = sy.symbols('x2, x3')                      # Separate symbols by commas,
>>> m, a = sy.symbols('mass acceleration')             # by spaces,
>>> x, y, z = sy.symbols('x:z')                         # or by colons.
>>> x4, x5, x6 = sy.symbols('x4:7')

# Combine symbolic variables to form expressions.
>>> expr = x**2 + x*y + 3*x*y + 4*y**3
>>> force = m * a
>>> print(expr, force, sep='\n')
x**2 + 4*x*y + 4*y**3
acceleration*mass
```

SymPy has its own version for each of the standard mathematical functions like $\sin(x)$, $\log(x)$, and \sqrt{x} , and includes predefined variables for special numbers such as π . The naming conventions for most functions match NumPy, but some of the built-in constants are named slightly differently.

Functions	$\sin(x)$ sy.sin()	$\arcsin(x)$ sy.asin()	$\sinh(x)$ sy.sinh()	e^x sy.exp()	$\log(x)$ sy.log()	\sqrt{x} sy.sqrt()
Constants	π sy.pi	e sy.E	$i = \sqrt{-1}$ sy.I	∞ sy.oo		

Other trigonometric functions like $\cos(x)$ follow the same naming conventions. For a complete list of SymPy functions, see <http://docs.sympy.org/latest/modules/functions/index.html>.

Achtung!

Always use SymPy functions and constants when creating expressions instead of using NumPy's functions and constants. Later we will show how to make NumPy and SymPy cooperate.

```
>>> import numpy as np

>>> x = sy.symbols('x')
>>> np.exp(x)                               # Try to use NumPy to represent e**x.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Symbol' object has no attribute 'exp'

>>> sy.exp(x)                             # Use SymPy's version instead.
exp(x)
```

Note

SymPy defines its own numeric types for integers, floats, and rational numbers. For example, the `sy.Rational` class is similar to the standard library's `fractions.Fraction` class, and should be used to represent fractions in SymPy expressions.

```
>>> x = sy.symbols('x')
>>> (2/3) * sy.sin(x)                      # 2/3 returns a float, not a rational.
0.6666666666666667*sin(x)

>>> sy.Rational(2, 3) * sy.sin(x)    # Keep 2/3 symbolic.
2*sin(x)/3
```

Always be aware of which numeric types are being used in an expression. Using rationals and integers where possible is important for simplifying expressions.

Problem 1. Write a function that returns the expression $\frac{2}{5}e^{x^2-y} \cosh(x+y) + \frac{3}{7}\log(xy+1)$ symbolically. Make sure that the fractions remain symbolic.

Sums and Products

Expressions that can be written as a sum or a product can be constructed with `sy.summation()` or `sy.product()`, respectively. Each of these functions accepts an expression that represents one term of the sum or product, then a tuple indicating the indexing variable and which values it should take on. For example, the following code constructs the sum and product given below.

$$\sum_{i=1}^4 x + iy \quad \prod_{i=0}^5 x + iy$$

```
>>> x, y, i = sy.symbols('x y i')

>>> sy.summation(x + i*y, (i, 1, 4))      # Sum over i=1,2,3,4.
4*x + 10*y

>>> sy.product(x + i*y, (i, 0, 5))        # Multiply over i=0,1,2,3,4,5.
x*(x + y)*(x + 2*y)*(x + 3*y)*(x + 4*y)*(x + 5*y)
```

Simplifying Expressions

The expressions for the summation and product in the previous example are automatically simplified. More complicated expressions can be simplified with one or more of the following functions.

Function	Description
<code>sy.cancel()</code>	Cancel common factors in the numerator and denominator.
<code>sy.expand()</code>	Expand a factored expression.
<code>sy.factor()</code>	Factor an expanded expression.
<code>sy.radsimp()</code>	Rationalize the denominator of an expression.
<code>sy.simplify()</code>	Simplify an expression.
<code>sy.trigsimp()</code>	Simplify only the trigonometric parts of the expression.

```
>>> x = sy.symbols('x')
>>> expr = (x**2 + 2*x + 1) / ((x+1)*((sy.sin(x)/sy.cos(x))**2 + 1))
>>> print(expr)
(x**2 + 2*x + 1)/((x + 1)*(sin(x)**2/cos(x)**2 + 1))

>>> sy.simplify(expr)
(x + 1)*cos(x)**2
```

The generic `sy.simplify()` tries to simplify an expression in any possible way. This is often computationally expensive; using more specific simplifiers when possible reduces the cost.

```

>>> expr = sy.product(x + i*y, (i, 0, 3))
>>> print(expr)
x*(x + y)*(x + 2*y)*(x + 3*y)

>>> expr_long = sy.expand(expr)           # Expand the product terms.
>>> print(expr_long)
x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3

>>> expr_long /= (x + 3*y)
>>> print(expr_long)
(x**4 + 6*x**3*y + 11*x**2*y**2 + 6*x*y**3)/(x + 3*y)

>>> expr_short = sy.cancel(expr_long)    # Cancel out the denominator.
>>> print(expr_short)
x**3 + 3*x**2*y + 2*x*y**2

>>> sy.factor(expr_short)              # Factor the result.
x*(x + y)*(x + 2*y)

# Simplify the trigonometric parts of an expression.
>>> sy.trigsimp(2*sy.sin(x)*sy.cos(x))
sin(2*x)

```

See <http://docs.sympy.org/latest/tutorial/simplification.html> for more examples.

Achtung!

1. Simplifications return new expressions; they do not modify existing expressions in place.
2. The == operator compares two expressions for exact structural equality, not algebraic equivalence. Simplify or expand expressions before comparing them with ==.
3. Expressions containing floats may not simplify as expected. Always use integers and SymPy rationals in expressions when appropriate.

```

>>> expr = 2*sy.sin(x)*sy.cos(x)
>>> sy.trigsimp(expr)
sin(2*x)
>> print(expr)
2*sin(x)*cos(x)                  # The original expression is unchanged.

>>> 2*sy.sin(x)*sy.cos(x) == sy.sin(2*x)
False                            # The two expression structures differ.

>>> sy.factor(x**2.0 - 1)
x**2.0 - 1                      # Factorization fails due to the 2.0.

```

Problem 2. Write a function that symbolically computes and simplifies the following expression.

$$\prod_{i=1}^5 \sum_{j=i}^5 j(\sin(x) + \cos(x))$$

Evaluating Expressions

Every SymPy expression has a `subs()` method that substitutes one variable for another. The result is usually still a symbolic expression, even if a numerical value is used in the substitution. The `evalf()` method actually evaluates the expression numerically after all symbolic variables have been assigned a value. Both of these methods can accept a dictionary to reassign multiple symbols simultaneously.

```
>>> x,y = sy.symbols('x y')
>>> expr = sy.expand((x + y)**3)
>>> print(expr)
x**3 + 3*x**2*y + 3*x*y**2 + y**3

# Replace the symbolic variable y with the expression 2x.
>>> expr.subs(y, 2*x)
27*x**3

# Replace x with pi and y with 1.
>>> new_expr = expr.subs({x:sy.pi, y:1})
>>> print(new_expr)
1 + 3*pi + 3*pi**2 + pi**3
>>> new_expr.evalf()                      # Numerically evaluate the expression.
71.0398678443373

# Evaluate the expression by providing values for each variable.
>>> expr.evalf(subs={x:1, y:2})
27.0000000000000
```

These operations are good for evaluating an expression at a single point, but it is typically more useful to turn the expression into a reusable numerical function. To this end, `sy.lambdify()` takes in a symbolic variable (or list of variables) and an expression, then returns a callable function that corresponds to the expression.

```
# Turn the expression sin(x)^2 into a function with x as the variable.
>>> f = sy.lambdify(x, sy.sin(x)**2)
>>> print(f(0), f(np.pi/2), f(np.pi), sep='   ')
0.0  1.0  1.4997597826618576e-32

# Lambdify a function of several variables.
>>> f = sy.lambdify((x,y), sy.sin(x)**2 + sy.cos(y)**2)
>>> print(f(0,1), f(1,0), f(np.pi, np.pi), sep='   ')
0.2919265817264289  1.708073418273571  1.0
```

By default, `sy.lambdify()` uses the `math` module to convert an expression to a function. For example, `sy.sin()` is converted to `math.sin()`. By providing "`numpy`" as an additional argument, `sy.lambdify()` replaces symbolic functions with their NumPy equivalents instead, so `sy.sin()` is converted to `np.sin()`. This allows the resulting function to act element-wise on NumPy arrays, not just on single data points.

```
>>> f = sy.lambdify(x, 2*sy.sin(2*x), "numpy")
>>> f(np.linspace(0, 2*np.pi, 9))  # Evaluate f() at many points.
array([ 0.0000000e+00,  2.0000000e+00,  2.44929360e-16,
       -2.0000000e+00,  -4.89858720e-16,  2.0000000e+00,
       7.34788079e-16,  -2.0000000e+00,  -9.79717439e-16])
```

Note

It is almost always computationally cheaper to lambdify a function than to use substitutions. According to the SymPy documentation, using `sy.lambdify()` to do numerical evaluations “takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `subs()` method.”

```
In [1]: import sympy as sy
In [2]: import numpy as np

# Define a symbol, an expression, and points to plug into the expression.
In [3]: x = sy.symbols('x')
In [4]: expr = sy.tanh(x)
In [5]: points = np.random.random(10000)

# Time using evalf() on each of the random points.
In [6]: %time _ = [expr.subs(x, pt).evalf() for pt in points]
CPU times: user 5.29 s, sys: 40.3 ms, total: 5.33 s
Wall time: 5.36 s

# Lambdify the expression and time using the resulting function.
In [7]: f = sy.lambdify(x, expr)
In [8]: %time _ = [f(pt) for pt in points]
CPU times: user 5.39 ms, sys: 648 microseconds, total: 6.04 ms
Wall time: 7.75 ms      # About 1000 times faster than evalf().

# Lambdify the expression with NumPy and repeat the experiment.
In [9]: f = sy.lambdify(x, expr, "numpy")
In [10]: %time _ = f(points)
CPU times: user 381 microseconds, sys: 63 microseconds, total: 444 microseconds
Wall time: 282 microseconds  # About 10 times faster than regular lambdify.
```

Problem 3. The Maclaurin series up to order N for e^x is defined as

$$e^x \approx \sum_{n=0}^N \frac{x^n}{n!}. \quad (14.1)$$

Write a function that accepts an integer N . Define an expression for (14.1), then substitute in $-y^2$ for x to get a truncated Maclaurin series of e^{-y^2} . Lambdify the resulting expression and plot the series on the domain $y \in [-2, 2]$. Plot e^{-y^2} over the same domain for comparison.
(Hint: use `sy.factorial()` to compute the factorial.)

Call your function with increasing values of N to check that the series converges correctly.

Solving Symbolic Equations

A SymPy expression by itself is not an equation. However, `sy.solve()` equates an expression with zero and solves for a specified variable. In this way, SymPy can be used to solve equations.

```
>>> x,y = sy.symbols('x y')

# Solve x^2 - 2x + 1 = 0 for x.
>>> sy.solve(x**2 - 2*x + 1, x)
[1]                                         # The result is a list of solutions.

# Solve x^2 - 1 = 0 for x.
>>> sy.solve(x**2 - 1, x)
[-1, 1]                                     # This equation has two solutions.

# Solutions can also be expressions involving other variables.
>>> sy.solve(x/(y-x) + (x-y)/y, x)
[y*(-sqrt(5) + 3)/2, y*(sqrt(5) + 3)/2]
```

Problem 4. The following equation represents a rose curve in cartesian coordinates:

$$0 = 1 - \frac{(x^2 + y^2)^{7/2} + 18x^5y - 60x^3y^3 + 18xy^5}{(x^2 + y^2)^3}. \quad (14.2)$$

The curve is not the image of a single function (such a function would fail the vertical line test), so the best way to plot it is to convert (14.2) to a pair of parametric equations that depend on the angle parameter θ .

Construct an expression for the nonzero side of (14.2) and convert it to polar coordinates with the substitutions $x = r \cos(\theta)$ and $y = r \sin(\theta)$. Simplify the result, then solve it for r . There are two solutions due to the presence of an r^2 term; pick one and lambdify it to get a function $r(\theta)$. Use this function to plot $x(\theta) = r(\theta) \cos(\theta)$ against $y(\theta) = r(\theta) \sin(\theta)$ for $\theta \in [0, 2\pi]$.

(Hint: use `sy.Rational()` for the fractional exponent.)

Linear Algebra

Sympy can also solve systems of equations. A system of linear equations $Ax = \mathbf{b}$ is solved in a slightly different way than in NumPy and SciPy: instead of defining the matrix A and the vector \mathbf{b} separately, define the augmented matrix $M = [A | \mathbf{b}]$ and call `sy.solve_linear_system()` on M .

SymPy matrices are defined with `sy.Matrix()`, with the same syntax as 2-dimensional NumPy arrays. For example, the following code solves the system given below.

$$\begin{array}{l} x + y + z = 5 \\ 2x + 4y + 3z = 2 \\ 5x + 10y + 2z = 4 \end{array}$$

```
>>> x, y, z = sy.symbols('x y z')

# Define the augmented matrix M = [A|b].
>>> M = sy.Matrix([ [1, 1, 1, 5],
                  [2, 4, 3, 2],
                  [5, 10, 2, 4] ])

# Solve the system, providing symbolic variables to solve for.
>>> sy.solve_linear_system(M, x, y, z)
{x: 98/11, y: -45/11, z: 2/11}
```

SymPy matrices support the standard matrix operations of addition `+`, subtraction `-`, and multiplication `*`. Additionally, SymPy matrices are equipped with many useful methods, some of which are listed below. See <http://docs.sympy.org/latest/modules/matrices/matrices.html> for more methods and examples.

Method	Returns
<code>det()</code>	The determinant.
<code>eigenvals()</code>	The eigenvalues and their multiplicities.
<code>eigenvects()</code>	The eigenvectors and their corresponding eigenvalues.
<code>inv()</code>	The matrix inverse.
<code>is_nilpotent()</code>	<code>True</code> if the matrix is nilpotent.
<code>norm()</code>	The Frobenius, ∞ , 1, or 2 norm.
<code>nullspace()</code>	The nullspace as a list of vectors.
<code>rref()</code>	The reduced row-echelon form.
<code>singular_values()</code>	The singular values.

Achtung!

The `*` operator performs matrix multiplication on SymPy matrices. To perform element-wise multiplication, use the `multiply_elementwise()` method instead.

Problem 5. Find the eigenvalues of the following matrix by solving for λ in the characteristic equation $\det(A - \lambda I) = 0$.

$$A = \begin{bmatrix} x-y & x & 0 \\ x & x-y & x \\ 0 & x & x-y \end{bmatrix}$$

Also compute the eigenvectors by solving the linear system $A - \lambda I = \mathbf{0}$ for each eigenvalue λ . Return a dictionary mapping the eigenvalues to their eigenvectors.

(Hint: the `nullspace()` method may be useful.)

Check that $Av = \lambda v$ for each eigenvalue-eigenvector pair (λ, v) . Compare your results to the `eigenvals()` and `eigenvects()` methods for SymPy matrices.

Calculus

SymPy is also equipped to perform standard calculus operations, including derivatives, integrals, and taking limits. Like other elements of SymPy, calculus operations can be temporally expensive, but they give exact solutions whenever solutions exist.

Differentiation

The command `sy.Derivative()` creates a closed form, unevaluated derivative of an expression. This is like putting $\frac{d}{dx}$ in front of an expression without actually calculating the derivative symbolically. The resulting expression has a `doit()` method that can be used to evaluate the actual derivative. Equivalently, `sy.diff()` immediately takes the derivative of an expression.

Both `sy.Derivative()` and `sy.diff()` accept a single expression, then the variable or variables that the derivative is being taken with respect to.

```
>>> x, y = sy.symbols('x y')
>>> f = sy.sin(y)*sy.cos(x)**2

# Make an expression for the derivative of f with respect to x.
>>> df = sy.Derivative(f, x)
>>> print(df)
Derivative(sin(y)*cos(x)**2, x)

>>> df.doit()                               # Perform the actual differentiation.
-2*sin(x)*sin(y)*cos(x)

# Alternatively, calculate the derivative of f in a single step.
>>> sy.diff(f, x)
-2*sin(x)*sin(y)*cos(x)

# Calculate the derivative with respect to x, then y, then x again.
>>> sy.diff(f, x, y, x)
2*(sin(x)**2 - cos(x)**2)*cos(y)      # Note this expression could be simplified.
```

Problem 6. Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a smooth function. A critical point of f is a number $x_0 \in \mathbb{R}$ satisfying $f'(x_0) = 0$. The second derivative test states that a critical point x_0 is a local minimum of f if $f''(x_0) > 0$, or a local maximum of f if $f''(x_0) < 0$ (if $f''(x_0) = 0$, the test is inconclusive).

Now consider the polynomial

$$p(x) = 2x^6 - 51x^4 + 48x^3 + 312x^2 - 576x - 100.$$

Use SymPy to find all critical points of p and classify each as a local minimum or a local maximum. Plot $p(x)$ over $x \in [-5, 5]$ and mark each of the minima in one color and the maxima in another color. Return the collections of the x -values corresponding to the local minima and local maxima as two separate sets.

The Jacobian matrix of a multivariable function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ at a point $\mathbf{x}_0 \in \mathbb{R}^n$ is the $m \times n$ matrix J whose entries are given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x}_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ is defined by

$$J = \left[\begin{array}{c|c|c} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{array} \right] = \left[\begin{array}{ccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{array} \right], \quad \text{where} \quad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

To calculate the Jacobian matrix of a multivariate function with SymPy, define that function as a symbolic matrix (`sy.Matrix()`) and use its `jacobian()` method. The method requires a list of variables that prescribes the ordering of the differentiation.

```
# Create a matrix of symbolic variables.
>>> r, t = sy.symbols('r theta')
>>> f = sy.Matrix([r*sy.cos(t), r*sy.sin(t)])

# Find the Jacobian matrix of f with respect to r and theta.
>>> J = f.jacobian([r,t])
>>> J
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta), r*cos(theta)]])

# Evaluate the Jacobian matrix at the point (1, pi/2).
>>> J.subs({r:1, t:sy.pi/2})
Matrix([
[0, -1],
[1, 0]])

# Calculate the (symbolic) determinant of the Jacobian matrix.
>>> sy.simplify(J.det())
r
```

Integration

The function `sy.Integral()` creates an unevaluated integral expression. This is like putting an integral sign in front of an expression without actually evaluating the integral symbolically or numerically. The resulting expression has a `doit()` method that can be used to evaluate the actual integral. Equivalently, `sy.integrate()` immediately integrates an expression.

Both `sy.Derivative()` and `sy.diff()` accept a single expression, then a tuple or tuples containing the variable of integration and, optionally, the bounds of integration.

```
# Calculate the indefinite integral of sec(x).
>>> sy.integrate(sy.sec(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2

# Integrate cos(x)^2 from 0 to pi/2.
>>> sy.integrate(sy.cos(x)**2, (x,0,sy.pi/2))
pi/4

# Compute the integral of (y^2)(x^2) dx dy with x from 0 to 2, y from -1 to 1.
>>> sy.integrate(y**2 * x**2, (x,0,2), (y,-1,1))
16/9
```

Problem 7. Let $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ be a smooth function. The volume integral of f over the sphere S of radius r can be written in spherical coordinates as

$$\iiint_S f(x, y, z) dV = \int_0^\pi \int_0^{2\pi} \int_0^r f(h_1(\rho, \theta, \phi), h_2(\rho, \theta, \phi), h_3(\rho, \theta, \phi)) |\det(J)| d\rho d\theta d\phi,$$

where J is the Jacobian of the function $h : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ given by

$$h(\rho, \theta, \phi) = \begin{bmatrix} h_1(\rho, \theta, \phi) \\ h_2(\rho, \theta, \phi) \\ h_3(\rho, \theta, \phi) \end{bmatrix} = \begin{bmatrix} \rho \sin(\phi) \cos(\theta) \\ \rho \sin(\phi) \sin(\theta) \\ \rho \cos(\phi) \end{bmatrix}.$$

Calculate the volume integral of $f(x, y, z) = (x^2 + y^2 + z^2)^2$ over the sphere of radius r . Lambdify the resulting expression (with r as the independent variable) and plot the integral value for $r \in [0, 3]$. In addition, return the value of the integral when $r = 2$.

(Hint: simplify the integrand before computing the integral. In this case, $|\det(J)| = -\det(J)$.)

To check your answer, when $r = 3$, the value of the integral is $\frac{8748}{7}\pi$.

Achtung!

SymPy isn't perfect. It solves some integrals incorrectly, simplifies some expressions poorly, and is significantly slower than numerical computations. However, it is generally very useful for simplifying parts of an algorithm, getting exact answers, and handling tedious algebra quickly.

Additional Material

Pretty Printing

SymPy expressions, especially complicated ones, can be hard to read. Calling `sy.init_printing()` changes the way that certain expressions are displayed to be more readable; in a Jupyter Notebook, the rendering is done with L^AT_EX, as displayed below. Furthermore, the function `sy.latex()` converts an expression into actual L^AT_EX code for use in other settings.

```
In [1]: import sympy as sy
sy.init_printing()

In [2]: x, y, z, theta = sy.symbols('x y z \\\theta')
expr = sy.sin(theta) * sy.exp(y) * sy.log(z) * (x + y*theta)**4
I = sy.Integral(expr, (x,0,2), (y,-1,1), (z,1,sy.pi))
dI = sy.Derivative(I, theta)

dI
```

Out[2]: $\frac{d}{d\theta} \int_1^{\pi} \int_{-1}^1 \int_0^2 (\theta y + x)^4 e^y \log(z) \sin(\theta) dx dy dz$

Limits

Limits can be expressed, similar to derivatives or integrals, with `sy.Limit()`. Alternatively, `sy.limit()` (lowercase) evaluates a limit directly.

```
# Define the limit of a^(1/x) as x approaches infinity.
>>> a, x = sy.symbols('a x')
>>> sy.Limit(a**(1/x), x, sy.oo)
Limit(a**(1/x), x, oo, dir='-')

# Use the doit() method or sy.limit() to evaluate a limit.
>>> sy.limit((1+x)**(1/x), x, 0)
E

# Evaluate a limit as x approaches 0 from the negative direction.
>>> sy.limit(1/x, x, 0, '-')
-oo
```

Use limits instead of the `subs()` method when the value to be substituted is ∞ or is a singularity.

```
>>> expr = x / 2**x
>>> expr.subs(x, sy.oo)
nan
>>> sy.limit(expr, x, sy.oo)
0
```

Refer to <http://docs.sympy.org/latest/tutorial/calculus.html> for SymPy's official documentation on calculus operations.

Numerical Integration

Many integrals cannot be solved analytically. As an alternative to the `doit()` method, the `as_sum()` method approximates the integral with a summation. This method accepts the number of terms to use and a string indicating which approximation rule to use ("`left`", "`right`", "`midpoint`", or "`trapezoid`").

```
>>> x = sy.symbols('x')

# Try integrating e^(x^2) from 0 to pi.
>>> I = sy.Integral(sy.exp(x**2), (x,0,sy.pi))
>>> I.doit()
sqrt(pi)*erfi(pi)/2                                # The result is not very helpful.

# Instead, approximate the integral with a sum.
>>> I.as_sum(10, 'left').evalf()
1162.85031639195
```

See <http://docs.sympy.org/latest/modules/integrals/integrals.html> for more documentation on integration with SymPy.

Differential Equations

SymPy can be used to solve both ordinary and partial differential equations. The documentation for working with PDE functions is at <http://docs.sympy.org/dev/modules/solvers/pde.html>

The general form of a first-order differential equation is $\frac{dx}{dt} = f(x(t), t)$. To represent the unknown function $x(t)$, use `sy.Function()`. Just as `sy.solve()` is used to solve an expression for a given **variable**, `sy.dsolve()` solves an ODE for a particular **function**. When there are multiple solutions, `sy.dsolve()` returns a list; when arbitrary constants are involved they are given as `C1`, `C2`, and so on. Use `sy.checkodesol()` to check that a function is a solution to a differential equation.

```
>>> t = sy.symbols('t')
>>> x = sy.Function('x')

# Solve the equation x''(t) - 2x'(t) + x(t) = sin(t).
>>> ode = x(t).diff(t, 2) - 2*x(t).diff(t) + x(t) - sy.sin(t)
>>> sy.dsolve(ode, x(t))
Eq(x(t), (C1 + C2*t)*exp(t) + cos(t)/2) # C1 and C2 are arbitrary constants.
```

Since there are many types of ODEs, `sy.dsolve()` may also take a hint indicating what solving strategy to use. See `sy.ode.allhints` for a list of possible hints, or use `sy.classify_ode()` to see the list of hints that may apply to a particular equation.

15 Differentiation

Lab Objective: Derivatives are central in many applications. Depending on the application and on the available information, the derivative may be calculated symbolically, numerically, or with differentiation software. In this lab we explore these three ways to take a derivative, discuss what settings they are each appropriate for, and demonstrate their strengths and weaknesses.

Symbolic Differentiation

The derivative of a known mathematical function can be calculated symbolically with SymPy. This method is the most precise way to take a derivative, but it is computationally expensive and requires knowing the closed form formula of the function. Use `sy.diff()` to take a symbolic derivative.

```
>>> import sympy as sy

>>> x = sy.symbols('x')
>>> sy.diff(x**3 + x, x)      # Differentiate x^3 + x with respect to x.
3*x**2 + 1
```

Problem 1. Write a function that defines $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$ and takes its symbolic derivative with respect to x using SymPy. Lambdify the resulting function so that it can accept NumPy arrays and return the resulting function handle.

To check your function, plot f and its derivative f' over the domain $[-\pi, \pi]$. It may be helpful to move the bottom spine to 0 so you can see where the derivative crosses the x -axis.

```
>>> from matplotlib import pyplot as plt

>>> ax = plt.gca()
>>> ax.spines["bottom"].set_position("zero")
```

Numerical Differentiation

One definition for the derivative of a function $f : \mathbb{R} \rightarrow \mathbb{R}$ at a point x_0 is

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

Since this definition relies on h approaching 0, choosing a small, fixed value for h approximates $f'(x_0)$:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}. \quad (15.1)$$

This approximation is called the first order forward difference quotient. Using the points x_0 and $x_0 - h$ in place of $x_0 + h$ and x_0 , respectively, results in the first order backward difference quotient,

$$f'(x_0) \approx \frac{f(x_0) - f(x_0 - h)}{h}. \quad (15.2)$$

Forward difference quotients use values of f at x_0 and points greater than x_0 , while backward difference quotients use the values of f at x_0 and points less than x_0 . A centered difference quotient uses points on either side of x_0 , and typically results in a better approximation than the one-sided quotients. Combining (15.1) and (15.2) yields the second order centered difference quotient,

$$f'(x_0) = \frac{1}{2}f'(\bar{x}) + \frac{1}{2}f'(\tilde{x}) \approx \frac{f(x_0 + h) - f(x_0)}{2h} + \frac{f(x_0) - f(x_0 - h)}{2h} = \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

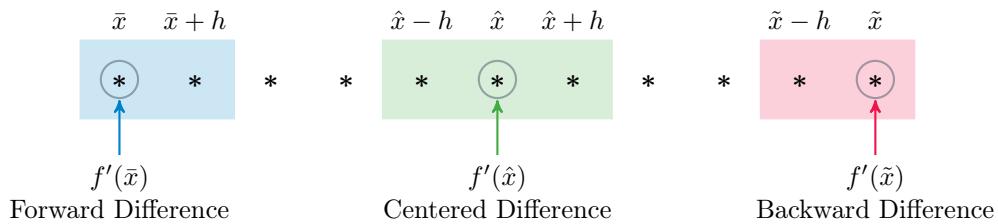


Figure 15.1

Note

The finite difference quotients in this section all approximate the first derivative of a function. The terms first order and second order refers to how quickly the approximation converges on the actual value of $f'(x_0)$ as h approaches 0, not to how many derivatives are being taken.

There are finite difference quotients for approximating higher order derivatives, such as f'' or f''' . For example, the centered difference quotient

$$f''(x_0) \approx \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2}$$

approximates the second derivative. This particular quotient is important for finite difference methods that approximate numerical solutions to some partial differential equations.

While we do not derive them here, there are other finite difference quotients that use more points to approximate the derivative, some of which are listed below. Using more points generally results in better convergence properties.

Type	Order	Formula
Forward	1	$\frac{f(x_0+h) - f(x_0)}{h}$
	2	$\frac{-3f(x_0) + 4f(x_0+h) - f(x_0+2h)}{2h}$
Backward	1	$\frac{f(x_0) - f(x_0-h)}{h}$
	2	$\frac{3f(x_0) - 4f(x_0-h) + f(x_0-2h)}{2h}$
Centered	2	$\frac{f(x_0+h) - f(x_0-h)}{2h}$
	4	$\frac{f(x_0-2h) - 8f(x_0-h) + 8f(x_0+h) - f(x_0+2h)}{12h}$

Table 15.1: Common finite difference quotients for approximating $f'(x_0)$.

Problem 2. Write a function for each of the finite difference quotients listed in Table 15.1. Each function should accept a function handle f , an array of points \mathbf{x} , and a float h ; each should return an array of the difference quotients evaluated at each point in \mathbf{x} .

To test your functions, approximate the derivative of $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$ at each point of a domain over $[-\pi, \pi]$. Plot the results and compare them to the results of Problem 1.

Convergence of Finite Difference Quotients

Finite difference quotients are typically derived using Taylor's formula. This method also shows how the accuracy of the approximation increases as $h \rightarrow 0$:

$$f(x_0 + h) = f(x_0) + f'(x_0)h + R_2(h) \implies \frac{f(x_0 + h) - f(x_0)}{h} - f'(x_0) = \frac{R_2(h)}{h}, \quad (15.3)$$

where $R_2(h) = h^2 \int_0^1 (1-t)f''(x_0 + th) dt$. Thus the absolute error of the first order forward difference quotient is

$$\left| \frac{R_2(h)}{h} \right| = |h| \left| \int_0^1 (1-t)f''(x_0 + th) dt \right| \leq |h| \int_0^1 |1-t||f''(x_0 + th)| dt.$$

If f'' is continuous, then for any $\delta > 0$, setting $M = \sup_{x \in (x_0 - \delta, x_0 + \delta)} f''(x)$ guarantees that

$$\left| \frac{R_2(h)}{h} \right| \leq |h| \int_0^1 M dt = M|h| \in O(h).$$

whenever $|h| < \delta$. That is, the error decreases at the same rate as h . If h gets twice as small, the error does as well. This is what is meant by a first order approximation. In a second order approximation, the absolute error is $O(h^2)$, meaning that if h gets twice as small, the error gets four times smaller.

Note

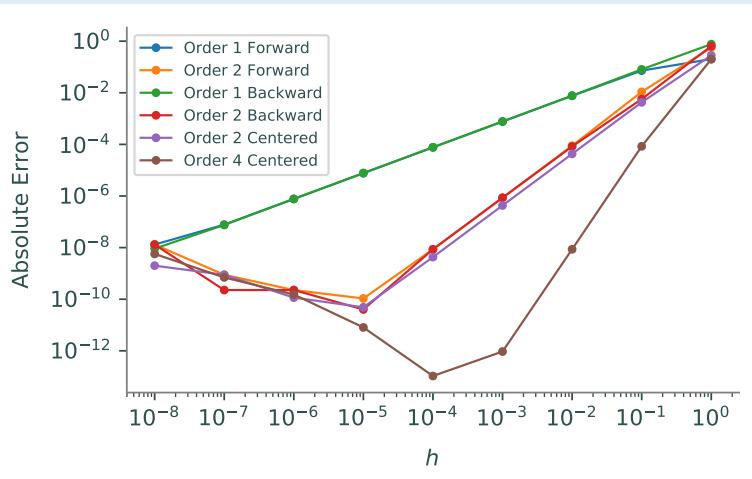
The notation $O(f(n))$ is commonly used to describe the temporal or spatial complexity of an algorithm. In that context, a $O(n^2)$ algorithm is much worse than a $O(n)$ algorithm. However, when referring to error, a $O(h^2)$ algorithm is **better** than a $O(h)$ algorithm because it means that the accuracy improves faster as h decreases.

Problem 3. Write a function that accepts a point x_0 at which to compute the derivative of $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$. Use your function from Problem 1 to compute the exact value of $f'(x_0)$. Then use each your functions from Problem 2 to get an approximate derivative $\tilde{f}'(x_0)$ for $h = 10^{-8}, 10^{-7}, \dots, 10^{-1}, 1$. Track the absolute error $|f'(x_0) - \tilde{f}'(x_0)|$ for each trial, then plot the absolute error against h on a log-log scale (use `plt.loglog()`).

Instead of using `np.linspace()` to create an array of h values, use `np.logspace()`. This function generates logarithmically spaced values between two powers of 10.

```
>>> import numpy as np
>>> np.logspace(-3, 0, 4)           # Get 4 values from 1e-3 to 1e0.
array([ 0.001,  0.01 ,  0.1   ,  1.    ])
```

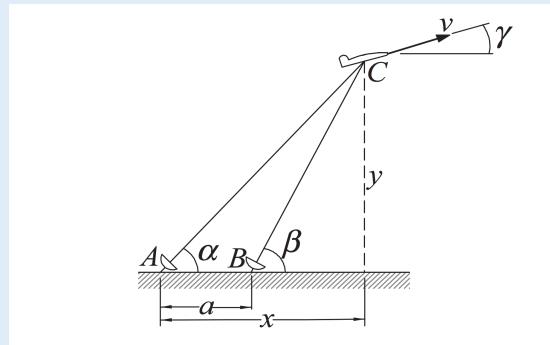
For $x_0 = 1$, your plot should resemble the following figure.



Achtung!

Mathematically, choosing smaller h values results in tighter approximations of $f'(x_0)$. However, Problem 3 shows that when h gets too small, the error stops decreasing. This numerical error is due to the denominator in each finite difference quotient becoming very small. The optimal value of h is usually one that is small, but not too small.

Problem 4. The radar stations A and B , separated by the distance $a = 500$ m, track a plane C by recording the angles α and β at one-second intervals. Your goal, back at air traffic control, is to determine the speed of the plane.^a



Let the position of the plane at time t be given by $(x(t), y(t))$. The speed at time t is the magnitude of the velocity vector, $\|\frac{d}{dt}(x(t), y(t))\| = \sqrt{x'(t)^2 + y'(t)^2}$. The closed forms of the functions $x(t)$ and $y(t)$ are unknown (and may not exist at all), but we can still use numerical methods to estimate $x'(t)$ and $y'(t)$. For example, at $t = 3$, the second order centered difference quotient for $x'(t)$ is

$$x'(3) \approx \frac{x(3+h) - x(3-h)}{2h} = \frac{1}{2}(x(4) - x(2)).$$

In this case $h = 1$ since data comes in from the radar stations at 1 second intervals.

Successive readings for α and β at integer times $t = 7, 8, \dots, 14$ are stored in the file `plane.npy`. Each row in the array represents a different reading; the columns are the observation time t , the angle α (in degrees), and the angle β (also in degrees), in that order. The Cartesian coordinates of the plane can be calculated from the angles α and β as follows.

$$x(\alpha, \beta) = a \frac{\tan(\beta)}{\tan(\beta) - \tan(\alpha)} \quad y(\alpha, \beta) = a \frac{\tan(\beta) \tan(\alpha)}{\tan(\beta) - \tan(\alpha)} \quad (15.4)$$

Load the data, convert α and β to radians, then compute the coordinates $x(t)$ and $y(t)$ at each given t using 15.4. Approximate $x'(t)$ and $y'(t)$ using a first order forward difference quotient for $t = 7$, a first order backward difference quotient for $t = 14$, and a second order centered difference quotient for $t = 8, 9, \dots, 13$ (see Figure 15.1). Return the values of the speed $\sqrt{x'(t)^2 + y'(t)^2}$ at each t .

(Hint: `np.deg2rad()` will be helpful.)

^aThis problem is adapted from an exercise in [Kiu13].

Numerical Differentiation in Higher Dimensions

Finite difference quotients can also be used to approximate derivatives in higher dimensions. The Jacobian matrix of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ at a point $\mathbf{x}_0 \in \mathbb{R}^n$ is the $m \times n$ matrix J whose entries are given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(\mathbf{x}_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ is defined by

$$J = \left[\begin{array}{c|c|c} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \hline \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{array} \right] = \left[\begin{array}{ccc} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{array} \right], \quad \text{where} \quad f(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

The difference quotients in this case resemble directional derivatives. The first order forward difference quotient for approximating a partial derivative is

$$\frac{\partial f}{\partial x_j}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_j) - f(\mathbf{x}_0)}{h},$$

where \mathbf{e}_j is the j th standard basis vector. The second order centered difference approximation is

$$\frac{\partial f}{\partial x_j}(\mathbf{x}_0) \approx \frac{f(\mathbf{x}_0 + h\mathbf{e}_j) - f(\mathbf{x}_0 - h\mathbf{e}_j)}{2h}. \quad (15.5)$$

Problem 5. Write a function that accepts a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, a point $\mathbf{x}_0 \in \mathbb{R}^n$, and a float h . Approximate the Jacobian matrix of f at \mathbf{x} using the second order centered difference quotient in (15.5).

(Hint: the standard basis vector \mathbf{e}_j is the j th column of the $n \times n$ identity matrix I .)

To test your function, define a simple function like $f(x, y) = [x^2, x^3 - y]^T$ where the Jacobian is easy to find analytically, then check the results of your function against SymPy or your own scratch work.

Differentiation Software

Many machine learning algorithms and structures, especially neural networks, rely on the gradient of a cost or objective function. To facilitate their research, several organizations have recently developed Python packages for numerical differentiation. For example, the Harvard Intelligent Probabilistic Systems Group (HIPS) started developing `autograd` in 2014 (<https://github.com/HIPS/autograd>) and Google created JAX (<https://github.com/google/jax>) as a successor to `autograd`. Popular deep learning libraries also contain automatic differentiation libraries. These tools use an algorithm known as automatic differentiation that is incredibly robust: they can differentiate functions with NumPy routines, `if` statements, `while` loops, and even recursion.

We conclude with a brief introduction to JAX. JAX is not included in Anaconda. It can be installed as follows on Mac and Linux:

```
pip install "jax[cpu]"
```

Installation directly via `pip` is not currently supported on Windows, however. Some unofficial builds for Windows are available at <https://github.com/clouduan/jax-windows-builder>. JAX also has additional installation options that allow it to do computations on a GPU using the CUDA library. See <https://github.com/google/jax#installation> for additional options for these cases.

JAX's `grad()` accepts a scalar-valued function and returns its gradient as a function that accepts the same parameters as the original. To support most of the NumPy features, JAX comes with its own thinly-wrapped version of Numpy, `jax.numpy`. Import this version of NumPy as `jnp` to avoid confusion.

```
>>> from jax import numpy as jnp          # Use JAX's version of NumPy.
>>> from jax import grad

>>> g = lambda x: jnp.exp(jnp.sin(jnp.cos(x)))
>>> dg = grad(g)                      # dg() is a callable function.
>>> dg(1.)                           # Use floats as input, not ints.
DeviceArray(-1.2069776, dtype=float32, weak_type=True)
```

Functions that `grad()` produces do not support array broadcasting, meaning they do not accept arrays as input. The easiest way to create a function is to use `jnp.vectorize()` on the derivative.

```
>>> pts = jnp.array([1, 2, 3], dtype=float)
>>> dg = jnp.vectorize(grad(g))      # Calculate g'(x) with array support.
>>> dg(pts)                        # Evaluate g'(x) at each of the points.
DeviceArray([-1.2069776, -0.5551414, -0.03356146], dtype=float32)
```

SymPy would have no trouble differentiating $g(x)$ in these examples. However, JAX can also differentiate Python functions that look nothing like traditional mathematical functions. For example, the following code computes the Taylor series of e^x with a loop.

```
>>> from sympy import factorial

>>> def taylor_exp(x, tol=.0001):
...     """Compute the Taylor series of e^x with terms greater than tol."""
...     result, i, term = 0, 0, 1
```

```

...     while jnp.abs(term) > tol:
...         term = x**i / int(factorial(i))
...         result, i = result + term, i + 1
...     return result
...
>>> d_exp = grad(taylor_exp)
>>> d_exp(2., .1), d_exp(2., .0001)
(DeviceArray(7.266667, dtype=float32, weak_type=True),
 DeviceArray(7.3889947, dtype=float32, weak_type=True))

```

Problem 6. The Chebyshev Polynomials satisfy the recursive relation

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$$

Write a function that accepts an array x and an integer n and recursively computes $T_n(x)$. Use JAX and your first function to create a function for $T'_n(x)$. Use this last function to plot each $T'_n(x)$ over the domain $[-1, 1]$ for $n = 0, 1, 2, 3, 4$.

(Hint: Use `jnp.ones_like(x)` to handle the case when $n = 0$.)

Problem 7. Let $f(x) = (\sin(x) + 1)^{\sin(\cos(x))}$ as in Problems 1 and 3. Write a function that accepts an integer N and performs the following experiment N times.

1. Choose a random value x_0 .
2. Use your function from Problem 1 to calculate the “exact” value of $f'(x_0)$. Time how long the entire process takes, including calling your function (each iteration).
3. Time how long it takes to get an approximation $\tilde{f}'(x_0)$ of $f'(x_0)$ using the fourth-order centered difference quotient from Problem 3. Record the absolute error $|f'(x_0) - \tilde{f}'(x_0)|$ of the approximation.
4. Time how long it takes to get an approximation $\bar{f}'(x_0)$ of $f'(x_0)$ using JAX (calling `grad()` every time). Record the absolute error $|f'(x_0) - \bar{f}'(x_0)|$ of the approximation.

Plot the computation times versus the absolute errors on a log-log plot with different colors for SymPy, the difference quotient, and JAX. For SymPy, assume an absolute error of `1e-18` (since only positive values can be shown on a log plot).

For $N = 200$, your plot should resemble the following figure. Note that SymPy has the least error but longer computation time, and that the difference quotient takes the least amount of time but has the most error. JAX, on the other hand, does not appear to be as well-suited to this particular problem. However, for more complicated functions and functions of multiple variables, it tends to be a “happy medium” between the two, with faster runtime than SymPy.

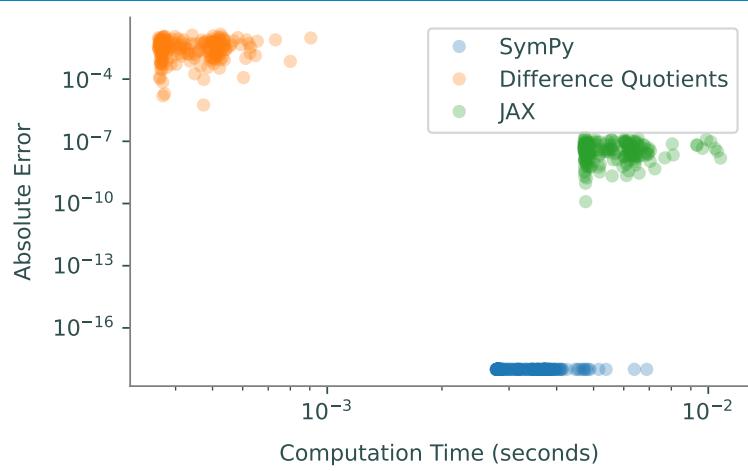


Figure 15.2: Solution with $N = 200$.

Additional Material

More JAX

For scalar-valued functions with multiple inputs, the parameter `argnums` specifies the variable that the derivative is computed with respect to. Providing a list for `argnums` gives several outputs.

```
>>> f = lambda x,y: 3*x*y + 2*y - x

# Take the derivative of f with respect to the first variable, x.
>>> dfdx = grad(f, argnums=0)           # Should be dfdx(x,y) = 3y - 1,
>>> dfdx(5., 1.)                      # so dfdx(5,1) = 3 - 1 = 2.
DeviceArray(2., dtype=float32, weak_type=True)

# Take the gradient with respect to the second variable, y.
>>> dfdy = grad(f, argnums=1)           # Should be dfdy(x,y) = 3x + 2,
>>> dfdy(5., 1.)                      # so dfdy(5,1) = 15 + 2 = 17.
DeviceArray(17., dtype=float32, weak_type=True)

# Get the full gradient.
>>> grad_f = grad(f, argnums=[0,1])
>>> jnp.array(grad_f(5., 1.))
DeviceArray([ 2., 17.], dtype=float32)
```

Finally, JAX's `jacobian()` can differentiate vector-valued functions.

```
>>> from jax import jacobian

>>> f = lambda x: jnp.array([x[0]**2, x[0]+x[1]])
>>> f_jac = jacobian(f)
>>> f_jac(jnp.array([1., 1.]))
DeviceArray([[2., 0.],
            [1., 1.]], dtype=float32)
```

16 Newton's Method

Lab Objective: Newton's method, the classical method for finding the zeros of a function, is one of the most important algorithms of all time. In this lab we implement Newton's method in arbitrary dimensions and use it to solve a few interesting problems. We also explore in some detail the convergence (or lack of convergence) of the method under various circumstances.

Iterative Methods

An iterative method is an algorithm that must be applied repeatedly to obtain a result. The general idea behind any iterative method is to make an initial guess at the solution to a problem, apply a few easy computations to better approximate the solution, use that approximation as the new initial guess, and repeat until done. More precisely, let F be some function used to approximate the solution to a problem. Starting with an initial guess of x_0 , compute

$$x_{k+1} = F(x_k) \tag{16.1}$$

for successive values of k to generate a sequence $(x_k)_{k=0}^{\infty}$ that hopefully converges to the true solution. If the terms of the sequence are vectors, they are denoted by \mathbf{x}_k .

In the best case, the iteration converges to the true solution x , written $\lim_{k \rightarrow \infty} x_k = x$ or $x_k \rightarrow x$. In the worst case, the iteration continues forever without approaching the solution. In practice, iterative methods require carefully chosen stopping criteria to guarantee that the algorithm terminates at some point. The general approach is to continue iterating until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. That is, choose a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$, and update the approximation using (16.1) until either

$$|x_k - x_{k-1}| < \varepsilon \quad \text{or} \quad k > N. \tag{16.2}$$

The choices for ε and N are significant: a “large” ε (such as 10^{-6}) produces a less accurate result than a “small” ε (such 10^{-16}), but demands less computations; a small N (10) also potentially lowers accuracy, but detects and halts nonconvergent iterations sooner than a large N (10,000). In code, ε and N are often named `tol` and `maxiter`, respectively (or similar).

While there are many ways to structure the code for an iterative method, probably the cleanest way is to combine a `for` loop with a `break` statement. As a very simple example, let $F(x) = \frac{x}{2}$. This method converges to $x = 0$ independent of starting point.

```
>>> F = lambda x: x / 2
>>> x0, tol, maxiter = 10, 1e-9, 8
>>> for k in range(maxiter):           # Iterate at most N times.
...     print(x0, end=' ')
...     x1 = F(x0)                      # Compute the next iteration.
...     if abs(x1 - x0) < tol:          # Check for convergence.
...         break                         # Upon convergence, stop iterating.
...     x0 = x1                          # Otherwise, continue iterating.
...
10  5.0  2.5  1.25  0.625  0.3125  0.15625  0.078125
```

In this example, the algorithm terminates after $N = 8$ iterations (the maximum number of allowed iterations) because the tolerance condition $|x_k - x_{k-1}| < 10^{-9}$ is not met fast enough. If N had been larger (say 40), the iteration would have quit early due to the tolerance condition.

Newton's Method in One Dimension

Newton's method is an iterative method for finding the zeros of a function. That is, if $f : \mathbb{R} \rightarrow \mathbb{R}$, the method attempts to find a \bar{x} such that $f(\bar{x}) = 0$. Beginning with an initial guess x_0 , calculate successive approximations for \bar{x} with the recursive sequence

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \quad (16.3)$$

The sequence converges to the zero \bar{x} of f if three conditions hold:

1. f and f' exist and are continuous,
2. $f'(\bar{x}) \neq 0$, and
3. x_0 is “sufficiently close” to \bar{x} .

In applications, the first two conditions usually hold. If \bar{x} and x_0 are not “sufficiently close,” Newton’s method may converge very slowly, or it may not converge at all. However, when all three conditions hold, Newton’s method converges quadratically, meaning that the maximum error is squared at every iteration. This is very quick convergence, making Newton’s method as powerful as it is simple.

Problem 1. Write a function that accepts a function f , an initial guess x_0 , the derivative f' , a stopping tolerance defaulting to 10^{-5} , and a maximum number of iterations defaulting to 15. Use Newton’s method as described in (16.3) to compute a zero \bar{x} of f . Terminate the algorithm when $|x_k - x_{k-1}|$ is less than the stopping tolerance or after iterating the maximum number of allowed times. Return the last computed approximation to \bar{x} , a boolean value indicating whether or not the algorithm converged, and the number of iterations completed.

Test your function against functions like $f(x) = e^x - 2$ (see Figure 16.1) or $f(x) = x^4 - 3$. Check that the computed zero \bar{x} satisfies $f(\bar{x}) \approx 0$. Also consider comparing your function to `scipy.optimize.newton()`, which accepts similar arguments.

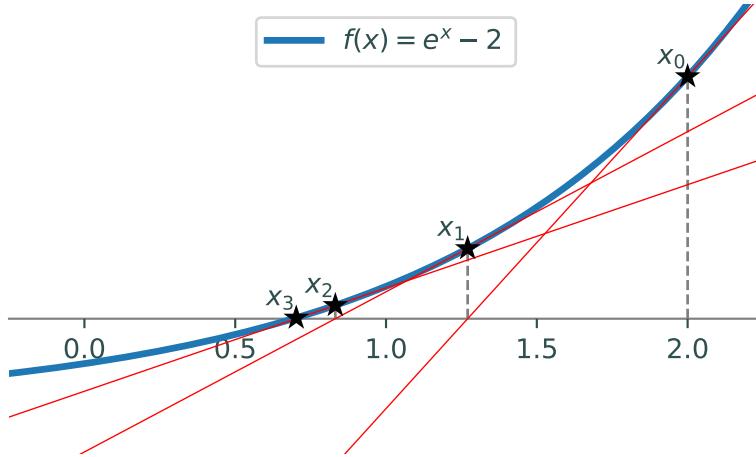


Figure 16.1: Newton’s method approximates the zero of a function (blue) by choosing as the next approximation the x -intercept of the tangent line (red) that goes through the point $(x_k, f(x_k))$. In this example, $f(x) = e^x - 2$, which has a zero at $\bar{x} = \log(2)$. Setting $x_0 = 2$ and using (16.3) to iterate, we have $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{e^2 - 2}{e^2} \approx 1.2707$. Similarly, $x_2 \approx 0.8320$, $x_3 \approx .7024$, and $x_4 \approx 0.6932$. After only a few iterations, the zero $\log(2) \approx 0.6931$ is already computed to several digits of accuracy.

Note

Newton’s method can be used to find zeros of functions that are hard to solve for analytically. For example, the function $f(x) = \frac{\sin(x)}{x} - x$ is not continuous on any interval containing 0, but it can be made continuous by defining $f(0) = 1$. Newton’s method can then be used to compute the zeros of this function.

Problem 2. Suppose that an amount of P_1 dollars is put into an account at the beginning of years $1, 2, \dots, N_1$ and that the account accumulates interest at a fractional rate r (so $r = .05$ corresponds to 5% interest). In addition, at the beginning of years $N_1 + 1, N_1 + 2, \dots, N_1 + N_2$, an amount of P_2 dollars is withdrawn from the account and that the account balance is exactly zero after the withdrawal at year $N_1 + N_2$. Then the variables satisfy

$$P_1[(1+r)^{N_1} - 1] = P_2[1 - (1+r)^{-N_2}].$$

Write a function that, given N_1 , N_2 , P_1 , and P_2 , uses Newton’s method to determine r . For the initial guess, use $r_0 = 0.1$.

(Hint: Construct $f(r)$ such that when $f(r) = 0$, the equation is satisfied. Also compute $f'(r)$.)

To test your function, if $N_1 = 30$, $N_2 = 20$, $P_1 = 2000$, and $P_2 = 8000$, then $r \approx 0.03878$. (From Atkinson, page 118).

Backtracking

Newton's method may not converge for a variety of reasons. One potential problem occurs when the step from x_k to x_{k+1} is so large that the zero is stepped over completely. Backtracking is a strategy that combats the problem of overstepping by moving only a fraction of the full step from x_k to x_{k+1} . This suggests a slight modification to (16.3),

$$x_{k+1} = x_k - \alpha \frac{f(x_k)}{f'(x_k)}, \quad \alpha \in (0, 1]. \quad (16.4)$$

Note that setting $\alpha = 1$ results in the exact same method defined in (16.3), but for $\alpha \in (0, 1)$, only a fraction of the step is taken at each iteration.

Problem 3. Modify your function from Problem 1 so that it accepts a parameter α that defaults to 1. Incorporate (16.4) to allow for backtracking.

To test your modified function, consider $f(x) = x^{1/3}$. The command `x**(1/3.)` fails when `x` is negative, so the function can be defined with NumPy as follows.

```
import numpy as np
f = lambda x: np.sign(x) * np.power(np.abs(x), 1./3)
```

With $x_0 = .01$ and $\alpha = 1$, the iteration should **not** converge. However, setting $\alpha = .4$, the iteration should converge to a zero that is close to 0.

The backtracking constant α is significant, as it can result in faster convergence or convergence to a different zero (see Figure 16.2). However, it is not immediately obvious how to choose an optimal value for α .

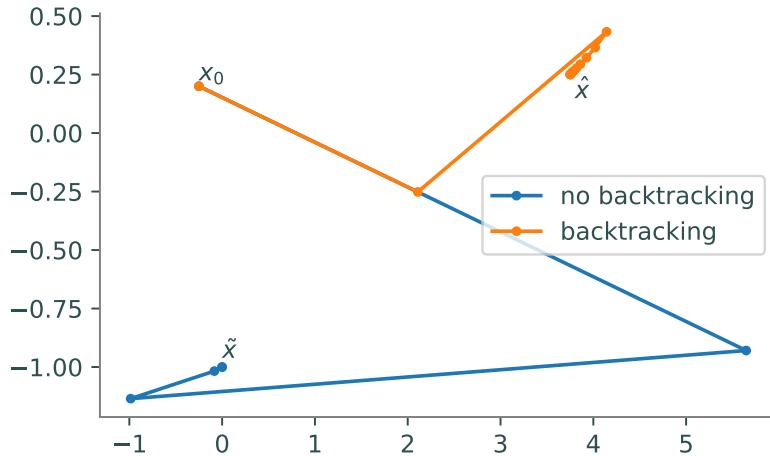


Figure 16.2: Starting at the same initial value but using different backtracking constants can result in convergence to two different solutions. The blue line converges to $\tilde{x} = (0, -1)$ with $\alpha = 1$ in 5 iterations of Newton's method while the orange line converges to $\hat{x} = (3.75, .25)$ with $\alpha = 0.4$ in 15 iterations. Note that the points in this example are 2-dimensional, which is discussed in the next section.

Problem 4. Write a function that accepts the same arguments as your function from Problem 3 except for α . Use Newton's method to find a zero of f using various values of α in the interval $(0, 1]$. Plot the values of α against the number of iterations performed by Newton's method. Return a value for α that results in the lowest number of iterations.

A good test case for this problem is the function $f(x) = x^{1/3}$ discussed in Problem 3. In this case, your plot should show that the optimal value for α is actually closer to .3 than to .4.

Newton's Method in Higher Dimensions

Newton's method can be generalized to work on functions with a multivariate domain and range. Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be given by $f(\mathbf{x}) = [f_1(\mathbf{x}) \ f_2(\mathbf{x}) \ \dots \ f_k(\mathbf{x})]^\top$, with $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$ for each i . The derivative $Df : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$ is the $n \times n$ Jacobian matrix of f .

$$Df = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_k} \end{bmatrix}$$

In this setting, Newton's method seeks a vector $\bar{\mathbf{x}}$ such that $f(\bar{\mathbf{x}}) = \mathbf{0}$, the vector of n zeros. With backtracking incorporated, (16.4) becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha Df(\mathbf{x}_k)^{-1} f(\mathbf{x}_k). \quad (16.5)$$

Note that if $n = 1$, (16.5) is exactly (16.4) because in that case, $Df(x)^{-1} = 1/f'(x)$.

This vector version of Newton's method terminates when the maximum number of iterations is reached or the difference between successive approximations is less than a predetermined tolerance ε with respect to a vector norm, that is, $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| < \varepsilon$.

Problem 5. Modify your function from Problems 1 and 3 so that it can compute a zero of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ for any $n \in \mathbb{N}$. Take the following tips into consideration.

- If $n > 1$, f should be a function that accepts a 1-D NumPy array with n entries and returns another NumPy array with n entries. Similarly, Df should be a function that accepts a 1-D array with n entries and returns a $n \times n$ array. In other words, f and Df are callable functions, but $f(\mathbf{x})$ is a vector and $Df(\mathbf{x})$ is a matrix.
- `np.isscalar()` may be useful for determining whether or not $n > 1$.
- Instead of computing $Df(\mathbf{x}_k)^{-1}$ directly at each step, solve the system $Df(\mathbf{x}_k)\mathbf{y}_k = f(\mathbf{x}_k)$ and set $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{y}_k$. In other words, use `la.solve()` instead of `la.inv()`.
- The stopping criterion now requires using a norm function instead of `abs()`.

After your modifications, carefully verify that your function still works in the case that $n = 1$, and that your functions from Problems 2 and 4 also still work correctly. In addition, your function from Problem 4 should also work for any $n \in \mathbb{N}$.

Problem 6. Bioremediation involves the use of bacteria to consume toxic wastes. At a steady state, the bacterial density x and the nutrient concentration y satisfy the system of nonlinear equations

$$\begin{aligned}\gamma xy - x(1 + y) &= 0 \\ -xy + (\delta - y)(1 + y) &= 0,\end{aligned}$$

where γ and δ are parameters that depend on various physical features of the system.^a

For this problem, assume the typical values $\gamma = 5$ and $\delta = 1$, for which the system has solutions at $(x, y) = (0, 1), (0, -1)$, and $(3.75, .25)$. Write a function that finds an initial point $\mathbf{x}_0 = (x_0, y_0)$ such that Newton's method converges to either $(0, 1)$ or $(0, -1)$ with $\alpha = 1$, and to $(3.75, .25)$ with $\alpha = 0.55$. As soon as a valid \mathbf{x}_0 is found, return it (stop searching).

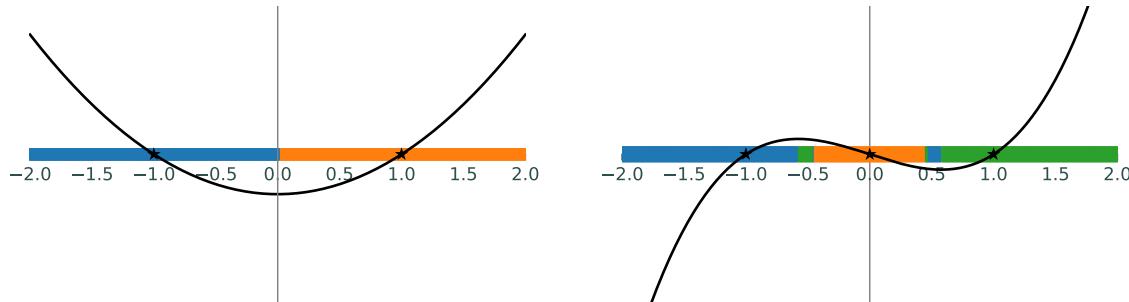
(Hint: search within the rectangle $[-\frac{1}{4}, 0] \times [0, \frac{1}{4}]$.)

^aThis problem is adapted from exercise 5.19 of [Hea02] and the notes of Homer Walker).

Basins of Attraction

When a function f has many zeros, the zero that Newton's method converges to depends on the initial guess x_0 . For example, the function $f(x) = x^2 - 1$ has zeros at -1 and 1 . If $x_0 < 0$, then Newton's method converges to -1 ; if $x_0 > 0$ then it converges to 1 (see Figure 16.3a). The regions $(-\infty, 0)$ and $(0, \infty)$ are called the basins of attraction of f . Starting in one basin of attraction leads to finding one zero, while starting in another basin yields a different zero.

When f is a polynomial of degree greater than 2, the basins of attraction are much more interesting. For example, the basis of attraction for $f(x) = x^3 - x$ are shown in Figure 16.3b. The basin for the zero at the origin is connected, but the other two basins are disconnected and share a kind of symmetry.



(a) Basins of attraction for $f(x) = x^2 - 1$.

(b) Basins of attraction for $f(x) = x^3 - x$.

Figure 16.3: Basins of attraction with $\alpha = 1$. Since choosing a different value for α can change which zero Newton's method converges to, the basins of attraction may change for other values of α .

It can be shown that Newton's method converges in any Banach space with only slightly stronger hypotheses than those discussed previously. In particular, Newton's method can be performed over the complex plane \mathbb{C} to find imaginary zeros of functions. Plotting the basins of attraction over \mathbb{C} yields some interesting results.

The zeros of $f(x) = x^3 - 1$ are 1, and $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$. To plot the basins of attraction for $f(x) = x^3 - 1$ on the square complex domain $X = \{a+bi \mid a \in [-\frac{3}{2}, \frac{3}{2}], b \in [-\frac{3}{2}, \frac{3}{2}]\}$, create an initial grid of complex points in this domain using `np.meshgrid()`.

```
>>> x_real = np.linspace(-1.5, 1.5, 500)      # Real parts.
>>> x_imag = np.linspace(-1.5, 1.5, 500)      # Imaginary parts.
>>> X_real, X_imag = np.meshgrid(x_real, x_imag)
>>> X_0 = X_real + 1j*X_imag                  # Combine real and imaginary parts.
```

The grid X_0 is a 500×500 array of complex values to use as initial points for Newton's method. Array broadcasting makes it easy to compute an iteration of Newton's method at every grid point.

```
>>> f = lambda x: x**3 - 1
>>> Df = lambda x: 3*x**2
>>> X_1 = X_0 - f(X_0)/Df(X_0)
```

After enough iterations, the (i, j) th element of the grid X_k corresponds to the zero of f that results from using the (i, j) th element of X_0 as the initial point. For example, with $f(x) = x^3 - 1$, each entry of X_k should be close to 1, $-\frac{1}{2} + \frac{\sqrt{3}}{2}i$, or $-\frac{1}{2} - \frac{\sqrt{3}}{2}i$. Each entry of X_k can then be assigned a value indicating which zero it corresponds to. Some results of this process are displayed below.

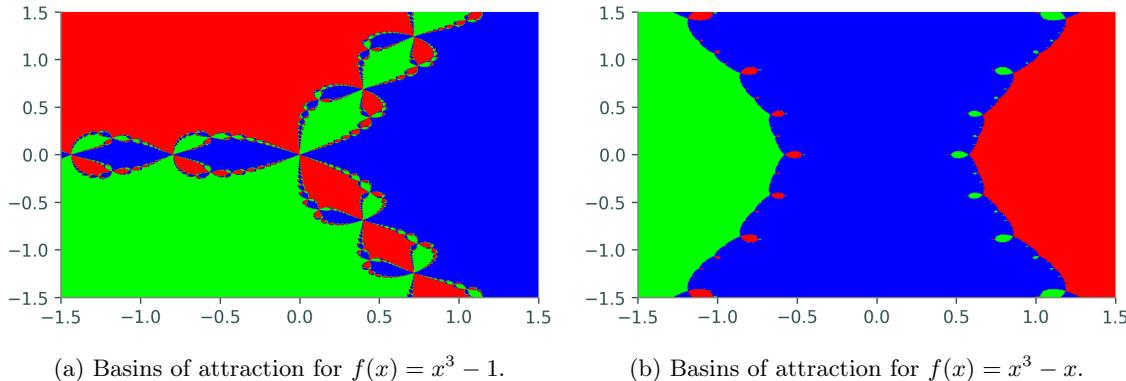


Figure 16.4

Note

Notice that in some portions of Figure 16.4a, whenever red and blue try to come together, a patch of green appears in between. This behavior repeats on an infinitely small scale, producing a fractal. Because it arises from Newton's method, this kind of fractal is called a Newton fractal.

Newton fractals show that the long-term behavior of Newton's method is **extremely** sensitive to the initial guess x_0 . Changing x_0 by a small amount can change the output of Newton's method in a seemingly random way. This phenomenon is called chaos in mathematics.

Problem 7. Write a function that accepts a function $f : \mathbb{C} \rightarrow \mathbb{C}$, its derivative $f' : \mathbb{C} \rightarrow \mathbb{C}$, an array `zeros` of the zeros of f , bounds $[r_{\min}, r_{\max}, i_{\min}, i_{\max}]$ for the domain of the plot, an integer `res` that determines the resolution of the plot, and number of iterations `iters` to run the iteration. Compute and plot the basins of attraction of f in the complex plane over the specified domain in the following steps.

1. Construct a `res`×`res` grid X_0 over the domain $\{a + bi \mid a \in [r_{\min}, r_{\max}], b \in [i_{\min}, i_{\max}]\}$.
2. Run Newton's method (without backtracking) on X_0 `iters` times, obtaining the `res`×`res` array x_k . To avoid the additional computation of checking for convergence at each step, do not use your function from Problem 5.
3. X_k cannot be directly visualized directly because its values are complex. Solve this issue by creating another `res`×`res` array Y . To compute the (i, j) th entry $Y_{i,j}$, determine which zero of f is closest to the (i, j) th entry of X_k . Set $Y_{i,j}$ to the index of this zero in the array `zeros`. If there are R distinct zeros, each $Y_{i,j}$ should be one of $0, 1, \dots, R - 1$. (Hint: `np.argmin()` may be useful.)
4. Use `plt.pcolormesh()` to visualize the basins. Recall that this function accepts three array arguments: the x -coordinates (in this case, the real components of the initial grid), the y -coordinates (the imaginary components of the grid), and an array indicating color values (Y). Set `cmap="brg"` to get the same color scheme as in Figure 16.4.

Test your function using $f(x) = x^3 - 1$ and $f(x) = x^3 - x$. The resulting plots should resemble Figures 16.4a and 16.4b, respectively (perhaps with the colors permuted).

17

Conditioning and Stability

Lab Objective: The condition number of a function measures how sensitive that function is to changes in the input. On the other hand, the stability of an algorithm measures how accurately that algorithm computes the value of a function from exact input. Both of these concepts are important for answering the crucial question, “is my computer telling the truth?” In this lab we examine the conditioning of common linear algebra problems, including computing polynomial roots and matrix eigenvalues. We also present an example to demonstrate how two different algorithms for the same problem may not have the same level of stability.

Note: There may be some variation in the solutions to problems in this lab between the different updates of NumPy, SciPy, and SymPy. Consider updating these packages if you are currently using older versions.

Conditioning

The absolute condition number of a function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ at a point $\mathbf{x} \in \mathbb{R}^m$ is defined by

$$\hat{\kappa}(\mathbf{x}) = \lim_{\delta \rightarrow 0^+} \sup_{\|\mathbf{h}\| < \delta} \frac{\|f(\mathbf{x} + \mathbf{h}) - f(\mathbf{x})\|}{\|\mathbf{h}\|}. \quad (17.1)$$

In other words, the absolute condition number of f is the limit of the change in output over the change of input. Similarly, the relative condition number of f is the limit of the relative change in output over the relative change in input,

$$\kappa(\mathbf{x}) = \lim_{\delta \rightarrow 0^+} \sup_{\|\mathbf{h}\| < \delta} \left(\frac{\|f(\mathbf{x} + \mathbf{h}) - f(\mathbf{x})\|}{\|f(\mathbf{x})\|} \right) \Bigg/ \frac{\|\mathbf{x}\|}{\|\mathbf{x}\|} \hat{\kappa}(\mathbf{x}). \quad (17.2)$$

A function with a large condition number is called ill-conditioned. Small changes to the input of an ill-conditioned function may produce large changes in output. It is important to know if a function is ill-conditioned because floating point representation almost always introduces some input error, and therefore the outputs of ill-conditioned functions cannot be trusted.

The condition number of a matrix A , $\kappa(A) = \|A\| \|A^{-1}\|$, is an upper bound on the condition number for many of the common problems associated with the matrix, such as solving the system $Ax = b$. If A is square but not invertible, then $\kappa(A) = \infty$ by convention. To compute $\kappa(A)$, we often use the matrix 2-norm, which is the largest singular value σ_{\max} of A . Recall that if σ is a singular value of A , $\frac{1}{\sigma}$ is a singular value of A^{-1} . Thus, we have that

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}, \quad (17.3)$$

which is also a valid equation for non-square matrices.

Achtung!

Ill-conditioned matrices can wreak havoc in even simple applications. For example, the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1.0000000001 \end{bmatrix}$$

is extremely ill-conditioned, with $\kappa(A) \approx 4 \times 10^{10}$. Solving the systems $Ax = b_1$ and $Ax = b_2$ can result in wildly different answers, even when b_1 and b_2 are extremely close.

```
>>> import numpy as np
>>> from scipy import linalg as la

>>> A = np.array([[1, 1], [1, 1+1e-10]])
>>> np.linalg.cond(A)
39999991794.058899

# Set up and solve a simple system of equations.
>>> b1 = np.array([2, 2])
>>> x1 = la.solve(A, b1)
>>> print(x1)
[ 2.  0.]

# Solve a system with a very slightly different vector b.
>>> b2 = np.array([2, 2+1e-5])
>>> la.norm(b1 - b2)
>>> x2 = la.solve(A, b2)
>>> print(x2)
[-99997.99172662  99999.99172662] # This solution is hugely different!
```

If you find yourself working with matrices that have large condition numbers, check your math carefully or try to reformulate the problem entirely.

Note

An orthonormal matrix U has orthonormal columns and satisfies $U^T U = I$ and $\|U\|_2 = 1$. If U is square, then $U^{-1} = U^T$ and U^T is also orthonormal. Therefore $\kappa(U) = \|U\|_2 \|U^{-1}\|_2 = 1$. Even if U is not square, all of its singular values are equal to 1, and again $\kappa(U) = \sigma_{\max}/\sigma_{\min} = 1$.

The condition number of a matrix cannot be less than 1 since $\sigma_{\max} \geq \sigma_{\min}$ by definition. Thus orthonormal matrices are, in a sense, the best kind of matrices for computations. This is one of the main reasons why numerical algorithms based on the QR decomposition or the SVD are so important.

Problem 1. Write a function that accepts a matrix A and computes its condition number using (17.3). Use `scipy.linalg.svd()`, or `scipy.linalg.svdvals()` to compute the singular values of A . Avoid computing A^{-1} . If the smallest singular value is 0, return ∞ (`np.inf`).

Validate your function by comparing it to `np.linalg.cond()`. Check that orthonormal matrices have a condition number of 1 (use `scipy.linalg.qr()` to generate an orthonormal matrix) and that singular matrices have a condition number of ∞ according to your function.

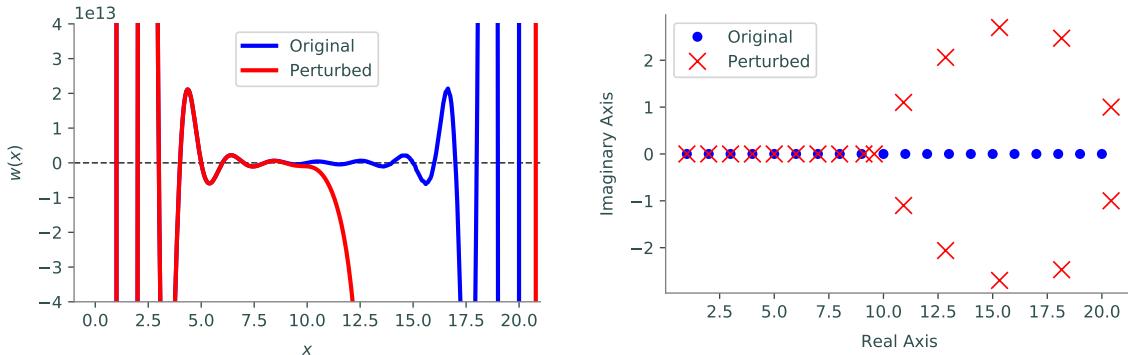
The Wilkinson Polynomial

Let $f : \mathbb{C}^{n+1} \rightarrow \mathbb{C}^n$ be the function that maps a collection of $n + 1$ coefficients $(c_n, c_{n-1}, \dots, c_0)$ to the n roots of the polynomial $c_n x^n + c_{n-1} x^{n-1} + \dots + c_2 x^2 + c_1 x + c_0$. Finding polynomial roots is an extremely ill-conditioned problem in general, so the condition number of f is likely very large. To see this, consider the Wilkinson polynomial, made famous by James H. Wilkinson in 1963:

$$w(x) = \prod_{r=1}^{20} (x - r) = x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + \dots$$

Let $\tilde{w}(x)$ be $w(x)$ where the coefficient on x^{19} is very slightly perturbed from -210 to -209.9999999 . The following code computes and compares the roots of $\tilde{w}(x)$ and $w(x)$ using NumPy and SymPy.

Figure 17.1a plots $w(x)$ and $\tilde{w}(x)$ together, and Figure 17.1b and compares their roots in the complex plane.



(a) The original and perturbed Wilkinson polynomials. They match for about half of the domain, then differ drastically.

(b) Roots of the original and perturbed Wilkinson polynomials. About half of the perturbed roots are complex.

Figure 17.1

Figure 17.1 clearly indicates that a very small change in just a single coefficient drastically changes the nature of the polynomial and its roots. To quantify the difference, estimate the condition numbers (this example uses the ∞ norm to compute $\hat{\kappa}$ and κ).

```
>>> import sympy as sy
>>> from matplotlib import pyplot as plt

# The roots of w are 1, 2, ..., 20.
>>> w_roots = np.arange(1, 21)

# Get the exact Wilkinson polynomial coefficients using SymPy.
>>> x, i = sy.symbols('x i')
>>> w = sy.poly_from_expr(sy.product(x-i, (i, 1, 20)))[0]
>>> w_coeffs = np.array(w.all_coeffs())
>>> print(w_coeffs[:6])
[1 -210 20615 -1256850 53327946 -1672280820]

# Perturb one of the coefficients very slightly.
>>> h = np.zeros(21)
>>> h[1]=1e-7
>>> new_coeffs = w_coeffs + h
>>> print(new_coeffs[:6])
[1 -209.999999900000 20615 -1256850 53327946 -1672280820]

# Use NumPy to compute the roots of the perturbed polynomial.
>>> new_roots = np.roots(np.poly1d(new_coeffs))

# Sort the roots to ensure that they are in the same order.
>>> w_roots = np.sort(w_roots)
>>> new_roots = np.sort(new_roots)

# Estimate the absolute condition number in the infinity norm.
```

```
>>> k = la.norm(new_roots - w_roots, np.inf) / la.norm(h, np.inf)
>>> print(k)
27841936.8061

# Estimate the relative condition number in the infinity norm.
>>> k * la.norm(w_coeffs, np.inf) / la.norm(w_roots, np.inf)
1.92161703373792e+25           # This is huge!!
```

There are some caveats to this example.

1. Computing the quotients in (17.1) and (17.2) for a fixed perturbation \mathbf{h} only approximates the condition number. The true condition number is the limit of such quotients. We hope that when $\|\mathbf{h}\|$ is small, a random quotient is at least the same order of magnitude as the limit, but there is no way to be sure.
2. This example assumes that NumPy's root-finding algorithm, `np.roots()`, is stable, so that the difference between `w_roots` and `new_roots` is due to the difference in coefficients, and not to problems with `np.roots()`. We will return to this issue in the next section.

Even with these caveats, it is apparent that root finding is a difficult problem to solve correctly. Always check your math carefully when dealing with polynomial roots.

Problem 2. Write a function that carries out the following experiment 100 times.

1. Randomly perturb the true coefficients of the Wilkinson polynomial by replacing each coefficient c_i with $c_i * r_i$, where r_i is drawn from a normal distribution centered at 1 with standard deviation 10^{-10} (use `np.random.normal()`).
2. Plot the perturbed roots as small points in the complex plane. That is, plot the real part of the coefficients on the x -axis and the imaginary part on the y -axis. Plot on the same figure in each experiment.
(Hint: use a pixel marker, `marker='.'`, to avoid overcrowding the figure.)
3. Compute the absolute and relative condition numbers with the ∞ norm.

Plot the roots of the unperturbed Wilkinson polynomial with the perturbed roots. Your final plot should resemble Figure 17.2. Finally, return the average computed absolute and relative condition numbers.

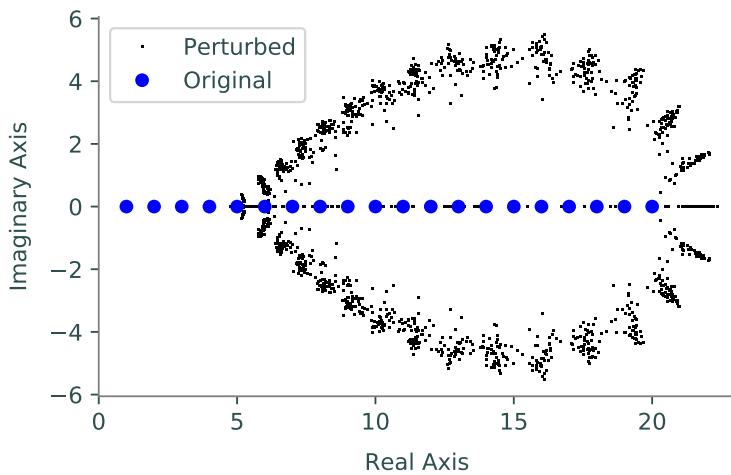


Figure 17.2: This figure replicates Figure 12.1 on p. 93 of [TB97].

Calculating Eigenvalues

Let $f : M_n(\mathbb{C}) \rightarrow \mathbb{C}^n$ be the function that maps an $n \times n$ matrix with complex entries to its n eigenvalues. This problem is well-conditioned for symmetric matrices, but it can be extremely ill-conditioned for non-symmetric matrices. Let A be an $n \times n$ matrix and let λ be the vector of the n eigenvalues of A . If $\tilde{A} = A + H$ is a perturbation of A and $\tilde{\lambda}$ are its eigenvalues, then the condition numbers of f can be estimated by

$$\hat{\kappa}(A) = \frac{\|\lambda - \tilde{\lambda}\|}{\|H\|}, \quad \kappa(A) = \frac{\|A\|}{\|\lambda\|} \hat{\kappa}(A). \quad (17.4)$$

Problem 3. Write a function that accepts a matrix A and estimates the condition number of the eigenvalue problem using (17.4). For the perturbation H , construct a matrix with complex entries where the real and imaginary parts are drawn from normal distributions centered at 0 with standard deviation $\sigma = 10^{-10}$.

```
reals = np.random.normal(0, 1e-10, A.shape)
imags = np.random.normal(0, 1e-10, A.shape)
H = reals + 1j*imags
```

Use `scipy.linalg.eig()` or `scipy.linalg.eigvals()` to compute the eigenvalues of A and $A + H$, and use the 2-norm for both the vector and matrix norms. Return the absolute and relative condition numbers.

The order `scipy.linalg.eig()` and `scipy.linalg.eigvals()` return eigenvalues is completely arbitrary, which means that the eigenvalues of A and $A+H$ may be returned in different orders. Without correcting for this, the computed value of $\|\lambda - \tilde{\lambda}\|$ can be very large even if the two matrices have very similar eigenvalues. Additionally, `np.sort()` does not help, as many matrices have sets of eigenvalues for which the sort order of this function is itself ill-conditioned. So, before comparing the two lists of eigenvalues, use the following function to reorder both to be as close as possible to each other:

```
def reorder_eigvals(orig_eigvals, pert_eigvals):
    """Reorder the perturbed eigenvalues to be as close to the original ←
        eigenvalues as possible.

    Parameters:
        orig_eigvals ((n,) ndarray) - The eigenvalues of the unperturbed ←
            matrix A
        pert_eigvals ((n,) ndarray) - The eigenvalues of the perturbed ←
            matrix A+H

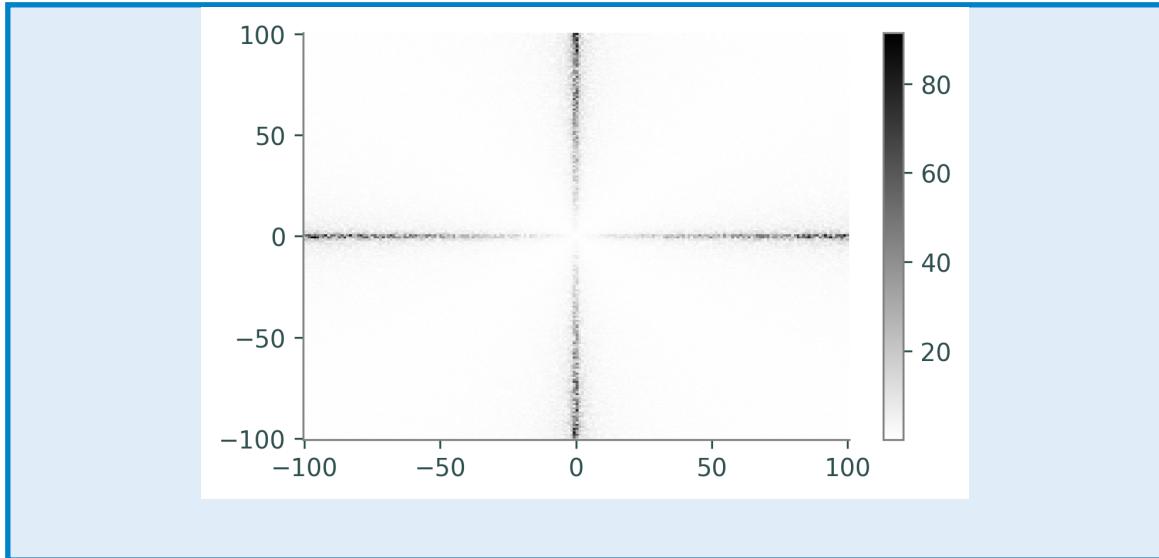
    Returns:
        ((n,) ndarray) - the reordered eigenvalues of the perturbed matrix
    """

    n = len(pert_eigvals)
    sort_order = np.zeros(n).astype(int)
    dists = np.abs(orig_eigvals - pert_eigvals.reshape(-1,1))
    for _ in range(n):
        index = np.unravel_index(np.argmin(dists), dists.shape)
        sort_order[index[0]] = index[1]
        dists[index[0],:] = np.inf
        dists[:,index[1]] = np.inf
    return pert_eigvals[sort_order]
```

Problem 4. Write a function that accepts bounds $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$ and an integer `res`. Use your function from Problem 3 to compute the relative condition number of the eigenvalue problem for the 2×2 matrix

$$\begin{bmatrix} 1 & x \\ y & 1 \end{bmatrix}$$

at every point of an evenly spaced `res` \times `res` grid over the domain $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Plot these estimated relative condition numbers using `plt.pcolormesh()` and the colormap `cmap='gray_r'`. With `res=200`, your plot should look similar to the following figure.



Problem 4 shows that the conditioning of the eigenvalue problem depends heavily on the matrix, and that it is difficult to know a priori how bad the problem will be. Luckily, most real-world problems requiring eigenvalues are symmetric. In their book on Numerical Linear Algebra, L. Trefethen and D. Bau III summed up the issue of conditioning and eigenvalues when they stated, “if the answer is highly sensitive to perturbations, you have probably asked the wrong question.”

Stability

The stability of an algorithm is measured by the error in its output. Let $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ be a problem to be solved, as in the previous section, and let \tilde{f} be an actual algorithm for solving the problem. The forward error of f at \mathbf{x} is $\|f(\mathbf{x}) - \tilde{f}(\mathbf{x})\|$, and the relative forward error of f at \mathbf{x} is

$$\frac{\|f(\mathbf{x}) - \tilde{f}(\mathbf{x})\|}{\|f(\mathbf{x})\|}.$$

An algorithm is called stable if its relative forward error is small.¹

As an example, consider again NumPy’s root-finding algorithm that we used to investigate the Wilkinson polynomial. The exact roots of $w(x)$ are clearly $1, 2, \dots, 20$. Had we not known this, we could have tried computing the roots from the coefficients using `np.roots()` (without perturbing the coefficients at all).

```
# w_coeffs holds the coefficients and w_roots holds the true roots.
>>> computed_roots = np.sort(np.roots(np.poly1d(w_coeffs)))
>>> print(computed_roots[:6])      # The computed roots are close to integers.
[ 1.                2.                3.                3.99999999  5.00000076  5.99998749]

# Compute the forward error.
>>> forward_error = la.norm(w_roots - computed_roots)
>>> print(forward_error)
0.020612653126379665
```

¹See the Additional Material section for alternative (and more rigorous) definitions of algorithmic stability.

```
# Compute the relative forward error.
>>> forward_error / la.norm(w_roots)
0.00038476268486104599 # The error is nice and small.
```

This analysis suggests that `np.roots()` is a stable algorithm, so large condition numbers of Problem 2 really are due to the poor conditioning of the problem, not the way in which the problem was solved.

Note

Conditioning is a property of a **problem** to be solved, such as finding the roots of a polynomial or calculating eigenvalues. Stability is a property of an **algorithm** to solve a problem, such as `np.roots()` or `scipy.linalg.eig()`. If a problem is ill-conditioned, any algorithm used to solve that problem may result in suspicious solutions, even if that algorithm is stable.

Least Squares

The ordinary least squares (OLS) problem is to find the \mathbf{x} that minimizes $\|A\mathbf{x} - \mathbf{b}\|_2$ for fixed A and \mathbf{b} . It can be shown that an equivalent problem is finding the solution of $A^H A \mathbf{x} = A^H \mathbf{b}$, called the normal equations. A common application of least squares is polynomial approximation. Given a set of m data points $\{(x_k, y_k)\}_{k=1}^m$, the goal is to find the set of coefficients $\{c_i\}_{i=0}^n$ such that

$$y_k \approx c_n x_k^n + c_{n-1} x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all k , with the smallest possible error. These m linear equations yield the linear system

$$A\mathbf{x} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \quad (17.5)$$

Problem 5. Write a function that accepts an integer n . Solve for the coefficients of the polynomial of degree n that best fits the data found in `stability_data.npy`. Use two approaches to get the least squares solution:

1. Use `la.inv()` to solve the normal equations: $\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$. Although this approach seems intuitive, it is actually highly unstable and can return an answer with a very large forward error.
2. Use `la.qr()` with `mode='economic'` and `la.solve_triangular()` to solve the system $R\mathbf{x} = Q^T \mathbf{b}$, which is equivalent to solving the normal equations. This algorithm has the advantage of being stable.

Load the data and set up the system (17.5) with the following code.

```
xk, yk = np.load("stability_data.npy").T
A = np.vander(xk, n+1)
```

Plot the resulting polynomials together with the raw data points. Return the forward error $\|Ax - b\|_2$ of both approximations.

(Hint: The function `np.polyval()` will be helpful for plotting the resulting polynomials.)

Test your function using various values of n , taking special note of what happens for values of n near 14.

Catastrophic Cancellation

When a computer takes the difference of two very similar numbers, the result is often stored with a small number of significant digits and the tiniest bit of information is lost. However, these small errors can propagate into large errors later down the line. This phenomenon is called catastrophic cancellation, and is a common cause for numerical instability.

Catastrophic cancellation is a potential problem whenever floats or large integers that are very close to one another are subtracted. This problem can be avoided by either rewriting the program to not use subtraction, or by increasing the number of significant digits that the computer tracks.

For example, consider the simple problem of computing $\sqrt{a} - \sqrt{b}$. The computation can be done directly with subtraction, or by performing the equivalent division

$$\sqrt{a} - \sqrt{b} = (\sqrt{a} - \sqrt{b}) \frac{\sqrt{a} + \sqrt{b}}{\sqrt{a} + \sqrt{b}} = \frac{a - b}{\sqrt{a} + \sqrt{b}}.$$

```
>>> from math import sqrt          # np.sqrt() fails for very large numbers.

>>> a = 10**20 + 1
>>> b = 10**20
>>> sqrt(a) - sqrt(b)           # Do the subtraction directly.
0.0                            # a != b, so information has been lost.

>>> (a - b) / (sqrt(a) + sqrt(b))  # Use the alternative formulation.
5e-11                           # Much better!
```

In this example, a and b are distinct enough that the computer can still tell that $a - b = 1$, but \sqrt{a} and \sqrt{b} are so close to each other that $\sqrt{a} - \sqrt{b}$ is computed as 0.

Problem 6. Let $I(n) = \int_0^1 x^n e^{x-1} dx$. It can be shown that for a positive integer n ,

$$I(n) = (-1)^n (!n - \frac{n!}{e}), \quad (17.6)$$

where $!n = n! \sum_{k=0}^n \frac{(-1)^k}{k!}$ is the subfactorial of n . Write a function to do the following.

1. Use SymPy's `sy.integrate()` to evaluate the integral form of $I(n)$ for $n = 5, 10, \dots, 50$. Convert the symbolic results of each integration to a float. Since this is done symbolically, these values can be accepted as the true values of $I(n)$. For this problem, use `sy.exp()` in the integrand.
(Hint: be careful that the values of n in the integrand are of type `int`.)
2. Use (17.6) to compute $I(n)$ for the same values of n . Use `sy.subfactorial()` to compute $!n$ and `sy.factorial()` to compute $n!$. The function used for e in this equation changes the returned error value. For this problem, use `np.e` instead of `sy.exp()`.
(Hint: be careful to only pass Python integers to these functions.)
3. Plot the relative forward error of the results computed in step 2 at each of the given values of n . When computing the relative forward error use absolute values instead of `la.norm()`. Use a log scale on the y -axis.

The examples presented in this lab are just a few of the ways that a mathematical problem can turn into a computational train wreck. Always use stable algorithms when possible, and remember to check if problems are well conditioned or not.

Additional Material

Other Notions of Stability

The definition of stability can be made more rigorous in the following way. Let f be a problem to solve and \tilde{f} an algorithm to solve it. If for every \mathbf{x} in the domain there exists a $\tilde{\mathbf{x}}$ such that

$$\frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \quad \text{and} \quad \frac{\|\tilde{f}(\mathbf{x}) - f(\tilde{\mathbf{x}})\|}{\|f(\tilde{\mathbf{x}})\|}$$

are small (close to $\epsilon_{\text{machine}} \approx 10^{-16}$), then \tilde{f} is called stable. In other words, “A stable algorithm gives nearly the right answer to nearly the right question” (Trefethen, Bao, 104). Note carefully that the quantity on the right is slightly different from the plain forward error introduced earlier.

Stability is desirable, but plain stability isn’t the best possible condition. For example, if for every input \mathbf{x} there exists a $\tilde{\mathbf{x}}$ such that $\|\tilde{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$ is small and $\tilde{f}(\mathbf{x}) = f(\tilde{\mathbf{x}})$ exactly, then \tilde{f} is called backward stable. Thus “A backward stable algorithm gives exactly the right answer to nearly the right question” (Trefethen, Bao, 104). Backward stable algorithms are generally more trustworthy than stable algorithms, but they are also less common.

Stability of Linear System Solvers

The algorithms presented so far in this manual have different levels of stability. The LU decomposition (with pivoting) is usually very good, but there are some pathological examples of matrices that can cause it to break down. Even so, `scipy.linalg.solve()` uses the LU decomposition. The QR decomposition (also with pivoting) is generally considered to be a better option than the LU decomposition and is more stable. However, solving a linear system using the SVD is even more stable than using the QR decomposition. For this reason, `scipy.linalg.lstsq()` uses the SVD.

18 Monte Carlo Integration

Lab Objective: Many important integrals cannot be evaluated symbolically because the integrand has no antiderivative. Traditional numerical integration techniques like Newton-Cotes formulas and Gaussian quadrature usually work well for one-dimensional integrals, but rapidly become inefficient in higher dimensions. Monte Carlo integration is an integration strategy that has relatively slow convergence, but that does extremely well in high-dimensional settings compared to other techniques. In this lab we implement Monte Carlo integration and apply it to a classic problem in statistics.

Volume Estimation

Since the area of a circle of radius r is $A = \pi r^2$, one way to numerically estimate π is to compute the area of the unit circle. Empirically, we can estimate the area by randomly choosing points in a domain that encompasses the unit circle. The percentage of points that land within the unit circle approximates the percentage of the area of the domain that the unit circle occupies. Multiplying this percentage by the total area of the sample domain gives an estimate for the area of the circle.

Since the unit circle has radius $r = 1$, consider the square domain $\Omega = [-1, 1] \times [-1, 1]$. The following code samples 2000 uniformly distributed random points in Ω , determines what percentage of those points are within the unit circle, then multiplies that percentage by 4 (the area of Ω) to get an estimate for π .

```
>>> import numpy as np
>>> from scipy import linalg as la

# Get 2000 random points in the 2-D domain [-1,1]x[-1,1].
>>> points = np.random.uniform(-1, 1, (2,2000))

# Determine how many points are within the circle.
>>> lengths = la.norm(points, axis=0)
>>> num_within = np.count_nonzero(lengths < 1)

# Estimate the circle's area.
>>> 4 * (num_within / 2000)
3.198
```

The estimate $\pi \approx 3.198$ isn't perfect, but it only differs from the true value of π by about 0.0564. On average, increasing the number of sample points decreases the estimate error.

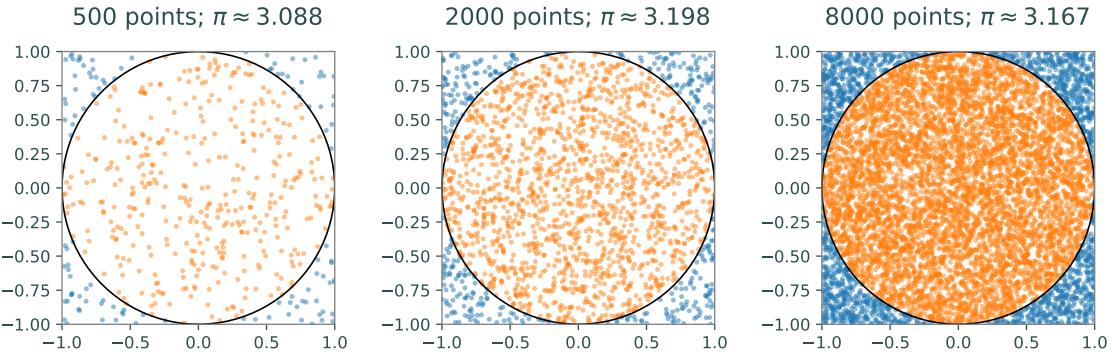


Figure 18.1: Estimating the area of the unit circle using random points.

Problem 1. The n -dimensional open unit ball is the set $U_n = \{\mathbf{x} \in \mathbb{R}^n \mid \|\mathbf{x}\|_2 < 1\}$. Write a function that accepts an integer n and a keyword argument N defaulting to 10^4 . Estimate the volume of U_n by drawing N points over the n -dimensional domain $[-1, 1] \times [-1, 1] \times \cdots \times [-1, 1]$. (Hint: the volume of $[-1, 1] \times [-1, 1] \times \cdots \times [-1, 1]$ is 2^n .)

When $n = 2$, this is the same experiment outlined above so your function should return an approximation of π . The volume of the U_3 is $\frac{4}{3}\pi \approx 4.18879$, and the volume of U_4 is $\frac{\pi^2}{2} \approx 4.9348$. Try increasing the number of sample points N to see if your estimates improve.

Integral Estimation

The strategy for estimating π can be formulated as an integral problem. Define $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ by

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \|\mathbf{x}\|_2 < 1 \text{ (\mathbf{x} is within the unit circle)} \\ 0 & \text{otherwise,} \end{cases}$$

and let $\Omega = [-1, 1] \times [-1, 1]$ as before. Then

$$\int_{-1}^1 \int_{-1}^1 f(x, y) dx dy = \int_{\Omega} f(\mathbf{x}) dV = \pi.$$

To estimate the integral we chose N random points $\{\mathbf{x}_i\}_{i=1}^N$ in Ω . Since f indicates whether or not a point lies within the unit circle, the total number of random points that lie in the circle is the sum of the $f(\mathbf{x}_i)$. Then the average of these values, multiplied by the volume $V(\Omega)$, is the desired estimate:

$$\int_{\Omega} f(\mathbf{x}) dV \approx V(\Omega) \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i). \quad (18.1)$$

This remarkably simple equation can be used to estimate the integral of any integrable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ over any domain $\Omega \subset \mathbb{R}^n$ and is called the general formula for Monte Carlo integration.

The intuition behind (18.1) is that $\frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i)$ approximates the average value of f on Ω , and multiplying the approximate average value by the volume of Ω yields the approximate integral of f over Ω . This is a little easier to see in one dimension: for a single-variable function $f : \mathbb{R} \rightarrow \mathbb{R}$, the Average Value Theorem states that the average value of f over an interval $[a, b]$ is given by

$$f_{avg} = \frac{1}{b-a} \int_a^b f(x) dx.$$

Then using the approximation $f_{avg} \approx \frac{1}{N} \sum_{i=1}^N f(x_i)$, the previous equation becomes

$$\int_a^b f(x) dx = (b-a)f_{avg} \approx V(\Omega) \frac{1}{N} \sum_{i=1}^N f(x_i), \quad (18.2)$$

which is (18.1) in one dimension. In this setting $\Omega = [a, b]$ and hence $V(\Omega) = b - a$.

Problem 2. Write a function that accepts a function $f : \mathbb{R} \rightarrow \mathbb{R}$, bounds of integration a and b , and an integer N defaulting to 10^4 . Use `np.random.uniform()` to sample N points over the interval $[a, b]$, then use (18.2) to estimate the integral

$$\int_a^b f(x) dx.$$

Test your function on the following integrals, or on other integrals that you can check by hand.

$$\begin{aligned} \int_{-4}^2 x^2 dx &= 24 & \int_{-2\pi}^{2\pi} \sin(x) dx &= 0 & \int_1^{10} \frac{1}{x} dx &= \log(10) \approx 2.30259 \\ \int_1^5 |\sin(10x) \cos(10x) + \sqrt{x} \sin(3x)| dx &\approx 4.502 \end{aligned}$$

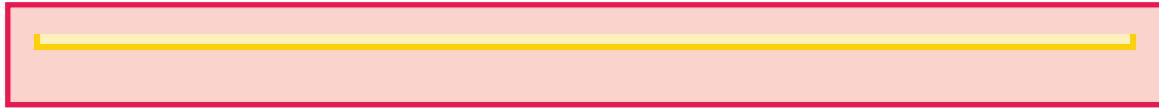
Achtung!

Be careful not to use Monte Carlo integration to estimate integrals that do not converge. For example, since $1/x$ approaches ∞ as x approaches 0 from the right, the integral

$$\int_0^1 \frac{1}{x} dx$$

does not converge. Even so, attempts at Monte Carlo integration still return a finite value. Use various numbers of sample points to see whether or not the integral estimate is converging.

```
>>> for N in [5000, 7500, 10000]:
...     print(np.mean(1. / np.random.uniform(0, 1, N)), end='\t')
...
11.8451683722    25.5814419888    7.64364735049    # No convergence.
```



Integration in Higher Dimensions

The implementation of (18.1) for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ with $n > 1$ introduces a few tricky details, but the overall procedure is the same for the case when $n = 1$. We consider only the case where $\Omega \subset \mathbb{R}^n$ is an n -dimensional box $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$.

1. If $n = 1$ then Ω is a line, so $V(\Omega) = b_1 - a_1$. If $n = 2$ then Ω is a rectangle, and hence $V(\Omega) = (b_1 - a_1)(b_2 - a_2)$, the product of the side lengths. The volume of a higher-dimensional box Ω is also the product of the side lengths,

$$V(\Omega) = \prod_{i=1}^n (b_i - a_i) \quad (18.3)$$

2. It is easy to sample uniformly over an interval $[a, b]$ with `np.random.uniform()`, or even over the n -dimensional cube $[a, b] \times [a, b] \times \cdots \times [a, b]$ (such as in Problem 1). However, if $a_i \neq a_j$ or $b_i \neq b_j$ for any $i \neq j$, the samples need to be constructed in a slightly different way.

The interval $[0, 1]$ can be transformed to the interval $[a, b]$ by scaling it so that it is the same length as $[a, b]$, then shifting it to the appropriate location.

$$[0, 1] \xrightarrow{\text{scale by } b-a} [0, b-a] \xrightarrow{\text{shift by } a} [a, b]$$

This suggests a strategy for sampling over $[a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$: sample uniformly from the n -dimensional box $[0, 1] \times [0, 1] \times \cdots \times [0, 1]$, multiply the i th component of each sample by $b_i - a_i$, then add a_i to that component.

$$[0, 1] \times \cdots \times [0, 1] \xrightarrow{\text{scale}} [0, b_1 - a_1] \times \cdots \times [0, b_n - a_n] \xrightarrow{\text{shift}} [a_1, b_1] \times \cdots \times [a_n, b_n] \quad (18.4)$$

Problem 3. Write a function that accepts a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, a list of lower bounds $[a_1, a_2, \dots, a_n]$, a list of upper bounds $[b_1, b_2, \dots, b_n]$, and an integer N defaulting to 10^4 . Use (18.1), (18.3), and (18.4) with N sample points to estimate the integral

$$\int_{\Omega} f(\mathbf{x}) dV,$$

where $\Omega = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_n, b_n]$. Return your answer as a float.

(Hint: use a list comprehension to calculate all of the $f(\mathbf{x}_i)$ quickly.)

«««< HEAD Test your function on the following integrals (as well as the one-dimensional examples in problem 2). ===== Test your function on the following integrals and the single dimensional examples from Problem 2 (hint: make sure bounds are inputted as lists, even in the single dimensional case). »»»> e6a1ccbacbc0b83d15381a8db4cb2b6efd50ecbd

$$\int_0^1 \int_0^1 x^2 + y^2 \, dx \, dy = \frac{2}{3} \quad \int_{-2}^1 \int_1^3 3x - 4y + y^2 \, dx \, dy = 54$$

$$\int_{-4}^4 \int_{-3}^3 \int_{-2}^2 \int_{-1}^1 x + y - wz^2 \, dx \, dy \, dz \, dw = 0$$

Note carefully how the order of integration defines the domain; in the last example, the x - y - z - w domain is $[-1, 1] \times [-2, 2] \times [-3, 3] \times [-4, 4]$, so the lower and upper bounds passed to your function should be $[-1, -2, -3, -4]$ and $[1, 2, 3, 4]$, respectively.

Convergence

Monte Carlo integration has some obvious pros and cons. On the one hand, it is difficult to get highly precise estimates. In fact, the error of the Monte Carlo method is proportional to $1/\sqrt{N}$, where N is the number of points used in the estimation. This means that dividing the error by 10 requires using 100 times more sample points.

On the other hand, the convergence rate is independent of the number of dimensions of the problem. That is, the error converges at the same rate whether integrating a 2-dimensional function or a 20-dimensional function. This gives Monte Carlo integration a huge advantage over other methods, and makes it especially useful for estimating integrals in high dimensions where other methods become computationally infeasible.

Problem 4. The probability density function of the joint distribution of n independent normal random variables, each with mean 0 and variance 1, is the function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ defined by

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}} e^{-\frac{\mathbf{x}^T \mathbf{x}}{2}}.$$

Though this is a critical distribution in statistics, f does not have a symbolic antiderivative.

Integrate f several times to study the convergence properties of Monte Carlo integration.

- Let $n = 4$ and $\Omega = [-\frac{3}{2}, \frac{3}{4}] \times [0, 1] \times [0, \frac{1}{2}] \times [0, 1] \subset \mathbb{R}^4$. Define f and Ω so that you can integrate f over Ω using your function from Problem 3.
- Use `scipy.stats.mvn.mvnun()` to get the “exact” value of $F = \int_{\Omega} f(\mathbf{x}) \, dV$. As an example, the following code computes the integral over $[-1, 1] \times [-1, 3] \times [-2, 1] \subset \mathbb{R}^3$.

```
>>> from scipy import stats

# Define the bounds of integration.
>>> mins = np.array([-1, -1, -2])
>>> maxs = np.array([1, 3, 1])
```

```
# The distribution has mean 0 and covariance I (the nxn identity).
>>> means, cov = np.zeros(3), np.eye(3)

# Compute the integral with SciPy.
>>> stats.mvn.mvnu(mins, maxs, means, cov)[0]
0.4694277116055261
```

3. Use `np.logspace()` to get 20 **integer** values of N that are roughly logarithmically spaced from 10^1 to 10^5 . For each value of N , use your function from Problem 3 to compute an estimate $\tilde{F}(N)$ of the integral with N samples. Compute the relative error $\frac{|F - \tilde{F}(N)|}{|F|}$ for each value of N .
4. Plot the relative error against the sample size N on a log-log scale. Also plot the line $1/\sqrt{N}$ for comparison. Your results should be similar to Figure 18.2.

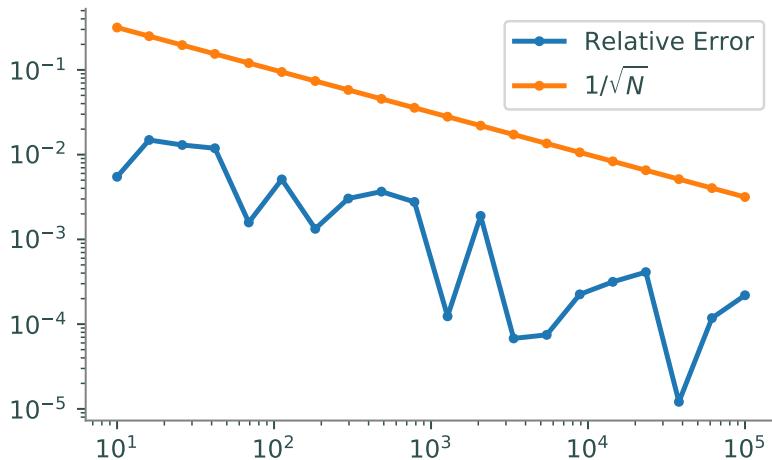


Figure 18.2: Monte Carlo integration converges at the same rate as $1/\sqrt{N}$ where N is the number of samples used in the estimate. However, the convergence is independent of dimension, which is why this strategy is so commonly used for high-dimensional integration.

19

Visualizing Complex-valued Functions

Lab Objective: Functions that map from the complex plane into the complex plane are difficult to fully visualize because the domain and range are both 2-dimensional. However, such functions can be visualized at the expense of partial information. In this lab we present methods for analyzing complex-valued functions visually, including locating their zeros and poles in the complex plane. We recommend completing the exercises in a Jupyter Notebook.

Representations of Complex Numbers

A complex number $z = x + iy$ can be written in polar coordinates as $z = re^{i\theta}$ where

- $r = |z| = \sqrt{x^2 + y^2}$ is the magnitude of z , and
- $\theta = \arg(z) = \arctan(y/x)$ is the argument of z , the angle in radians between z and 0.

Conversely, Euler's formula is the relation $re^{i\theta} = r \cos(\theta) + ir \sin(\theta)$. Then setting $re^{i\theta} = x + iy$ and equating real and imaginary parts yields the equations $x = r \cos(\theta)$ and $y = r \sin(\theta)$.

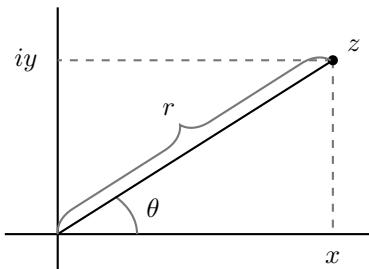


Figure 19.1: The complex number z can be represented in Cartesian coordinates as $z = x + iy$ and in polar coordinates as $z = re^{i\theta}$, when θ is in radians.

NumPy makes it easy to work with complex numbers and convert between coordinate systems. The function `np.angle()` returns the argument θ of a complex number (between $-\pi$ and π) and `np.abs()` (or `np.absolute()`) returns the magnitude r . These functions also operate element-wise on NumPy arrays.

```

>>> import numpy as np

>>> z = 2 - 2*j                      # 1j is the imaginary unit i = sqrt(-1).
>>> r, theta = np.abs(z), np.angle(z)
>>> print(r, theta)                  # The angle is between -pi and pi.
2.82842712475 -0.785398163397

# Check that z = r * e^(i*theta)
>>> np.isclose(z, r*np.exp(1j*theta))
True

# These function also work on entire arrays.
>>> np.abs(np.arange(5) + 2j*np.arange(5))
array([ 0.          ,  2.23606798,  4.47213595,  6.70820393,  8.94427191])

```

Complex Functions

A function $f : \mathbb{C} \rightarrow \mathbb{C}$ is called a complex-valued function. Visualizing f is difficult because \mathbb{C} has 2 real dimensions, so the graph of f should be 4-dimensional. However, since it is possible to visualize 3-dimensional objects, f can be visualized by ignoring one dimension. There are two main strategies for doing this: assign a color to each point $z \in \mathbb{C}$ corresponding to either the argument θ of $f(z)$, or to the magnitude r of $f(z)$. The graph that uses the argument is called a complex color wheel graph. Figure 19.2 displays the identity function $f(z) = z$ using these two methods.

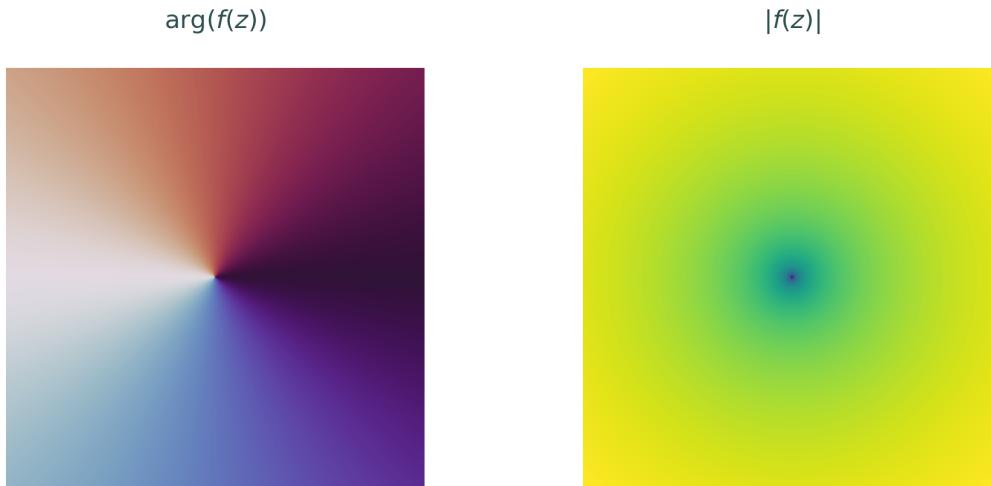


Figure 19.2: The identity function $f : \mathbb{C} \rightarrow \mathbb{C}$ defined by $f(z) = z$. On the left, the color at each point z represents the angle $\theta = \arg(f(z))$. As θ goes from $-\pi$ to π , the colors cycle smoothly counterclockwise from white to blue to red and back to white (this colormap is called "[twilight](#)"). On the right, the color represents the magnitude $r = |f(z)|$. The further a point is from the origin, the greater its magnitude (the colormap is the default, "[viridis](#)").

The plots in Figure 19.2 use Cartesian coordinates in the domain and polar coordinates in the codomain. The procedure for plotting in this way is fairly simple. Begin by creating a grid of complex numbers: create the real and imaginary parts separately, then use `np.meshgrid()` to turn them into a single array of complex numbers. Pass this array to the function f , compute the angle and argument of the resulting array, and plot them using `plt.pcolormesh()`. The following code sets up the complex domain grid.

```
>>> x = np.linspace(-1, 1, 400)      # Real domain.
>>> y = np.linspace(-1, 1, 400)      # Imaginary domain.
>>> X, Y = np.meshgrid(x, y)        # Make grid matrices.
>>> Z = X + 1j*Y                  # Combine the grids into a complex array.
```

Visualizing the argument and the magnitude separately provides different perspectives of the function f . The angle plot is generally more useful for visualizing function behavior, though the magnitude plot often makes it easy to spot important points such as zeros and poles.

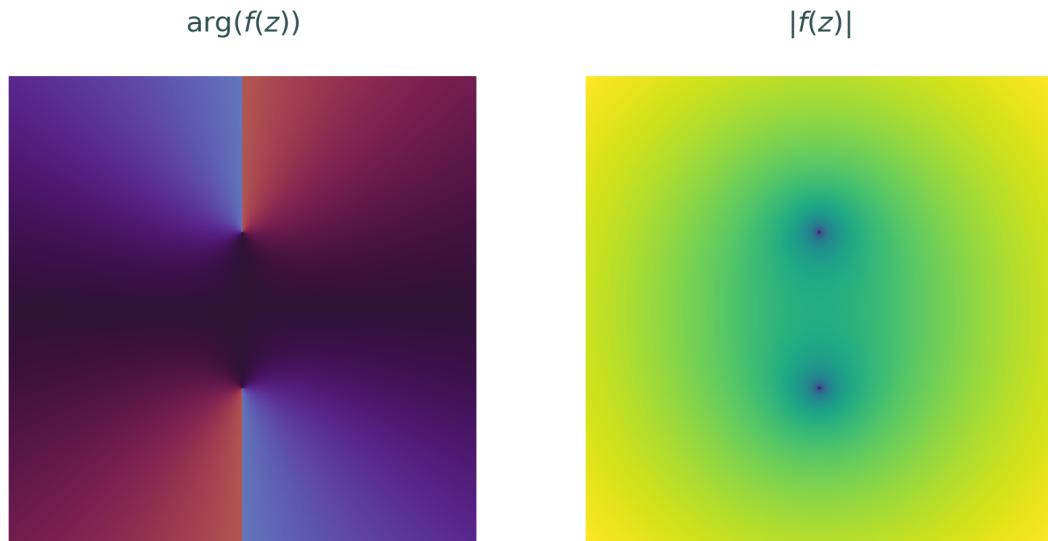


Figure 19.3: Plots of $f(z) = \sqrt{z^2 + 1}$ on $\{x+iy \mid x, y \in [-3, 3]\}$. Notice how a discontinuity is clearly visible in the angle plot on the left, but disappears from the magnitude plot on the right.

Problem 1. Write a function that accepts a function $f : \mathbb{C} \rightarrow \mathbb{C}$, bounds $[r_{\min}, r_{\max}, i_{\min}, i_{\max}]$ for the domain, an integer `res` that determines the resolution of the plot, and a string to set the figure title. Plot $\arg(f(z))$ and $|f(z)|$ on an equally-spaced `res`×`res` grid over the domain $\{x+iy \mid x \in [r_{\min}, r_{\max}], y \in [i_{\min}, i_{\max}]\}$ in separate subplots.

1. For $\arg(f(z))$, set the `plt.pcolormesh()` keyword arguments `vmin` and `vmax` to $-\pi$ and π , respectively. This forces the color spectrum to work well with `np.angle()`. Use the colormap "`twilight`", which starts and ends white, so that the color is the same for $-\pi$ and π .

2. For $|f(z)|$, set `norm=matplotlib.colors.LogNorm()` in `plt.pcolormesh()` so that the color scale is logarithmic. Use a sequential colormap like "`viridis`" or "`magma`".
3. Set the aspect ratio to "`equal`" in each plot. Give each subplot a title, and set the overall figure title with the given input string.

Use your function to visualize $f(z) = z$ on $\{x + iy \mid x, y \in [-1, 1]\}$ and $f(z) = \sqrt{z^2 + 1}$ on $\{x + iy \mid x, y \in [-3, 3]\}$. Compare the resulting plots to Figures 19.2 and 19.3, respectively.

Interpreting Complex Plots

Plots of a complex function can be used to quickly identify important points in the function's domain.

Zeros

A complex number z_0 is called a zero of the complex-valued function f if $f(z_0) = 0$. The mutliplicity or order of z_0 is the largest integer n such that f can be written as $f(z) = (z - z_0)^n g(z)$ where $g(z_0) \neq 0$. In other words, f has a zero of order n at z_0 if the Taylor series of f centered at z_0 can be written as

$$f(z) = \sum_{k=n}^{\infty} a_k (z - z_0)^k, \quad a_n \neq 0.$$

Angle and magnitude plots make it easy to locate a function's zeros and to determine their multiplicities.

Problem 2. Use your function from Problem 1 to plot the following functions on the domain $\{x + iy \mid x, y \in [-1, 1]\}$.

- $f(z) = z^n$ for $n = 2, 3, 4$.
- $f(z) = z^3 - iz^4 - 3z^6$. Compare the resulting plots to Figure 19.4.

Use a Markdown cell to write a sentence or two about how the zeros of a function and their multiplicity appear in angle and magnitude plots.

Problem 2 shows that in an angle plot of $f(z) = z^n$, the colors cycle n times counterclockwise around 0. This is explained by looking at z^n in polar coordinates,

$$z^n = (re^{i\theta})^n = r^n e^{i(n\theta)}.$$

Multiplying θ by a number greater than 1 compresses the graph along the “ θ -axis” by a factor of n . In other words, the output angle repeats itself n times in one cycle of θ . This is similar to taking a scalar-valued function $f : \mathbb{R} \rightarrow \mathbb{R}$ and replacing $f(x)$ with $f(nx)$.

Problem 2 also shows that the plot of $f(z) = z^3 - iz^4 - 3z^6$ looks very similar to the plot of $f(z) = z^3$ near the origin. This is because when z is close to the origin, z^4 and z^6 are much smaller in magnitude than z^3 , and so the behavior of z^3 dominates the function. In terms of the Taylor series centered at $z_0 = 0$, the quantity $|z - z_0|^{n+k}$ is much smaller than $|z - z_0|^n$ for z close to z_0 , and so the function behaves similar to $a_n(z - z_0)^n$.

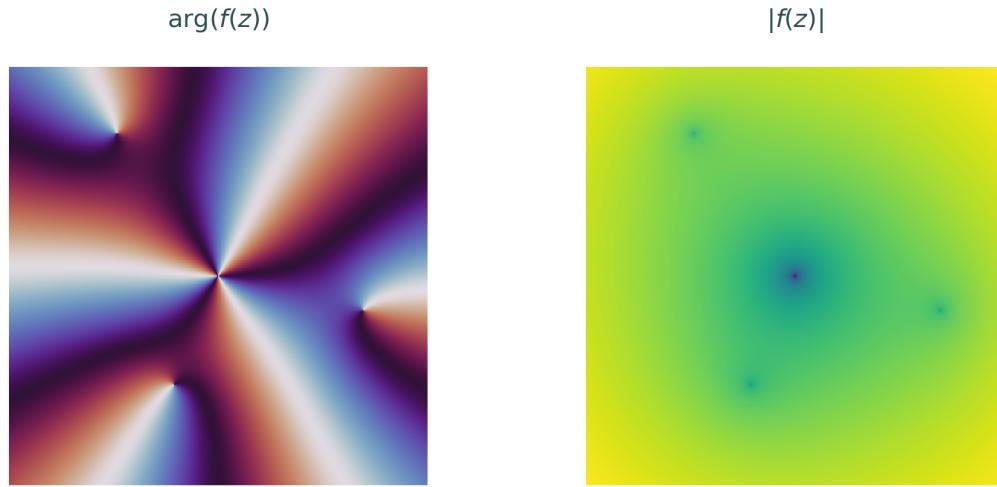


Figure 19.4: The angle plot of $f(z) = z^3 - iz^4 - 3z^6$ on $\{x + iy \mid x, y \in [-1, 1]\}$. The angle plot shows that $f(z)$ has a zero of order 3 at the origin and 3 distinct zeros of order 1 scattered around the origin. The magnitude plot makes it easier to pinpoint the location of the zeros.

Poles

A complex number z_0 is called a pole of the complex-valued function f if f can be written as $f(z) = g(z)/(z - z_0)$ where $g(z_0) \neq 0$. From this definition it is easy to see that $\lim_{z \rightarrow z_0} |f(z)| = \infty$, but knowing that $\lim_{z \rightarrow z_1} |f(z)| = \infty$ is not enough information to conclude that z_1 is a pole of f .

The order of z_0 is the largest integer n such that f can be written as $f(z) = g(z)/(z - z_0)^n$ with $g(z_0) \neq 0$. In other words, f has a pole of order n at z_0 if its Laurent series on a punctured neighborhood of z_0 can be written as

$$f(z) = \sum_{k=-n}^{\infty} a_k (z - z_0)^k \quad , a_{-n} \neq 0.$$

Problem 3. Plot the following functions on domains that show all of its zeros and/or poles.

- $f(z) = z^{-n}$ for $n = 1, 2, 3$.
- $f(z) = z^2 + iz^{-1} + z^{-3}$.

Use a Markdown cell to write a sentence or two about how the poles of a function appear in angle and magnitude plots. How can you tell the multiplicity of the poles from the plot?

Problem 3 shows that in angle plot of z^{-1} , the colors cycle n times clockwise around 0, as opposed to the counter-clockwise rotations seen around roots. Again, this can be explained by looking at the polar representation,

$$z^{-n} = (re^{i\theta})^{-n} = r^{-n}e^{i(-n\theta)}.$$

The minus sign on the θ reverses the direction of the colors, and the n makes them cycle n times.

From Problem 3 it is also clear that $f(z) = z^2 + iz^{-1} + z^{-3}$ behaves similarly to z^{-3} for z near the pole at $z_0 = 0$. Since $|z - z_0|^{-n+k}$ is much smaller than $|z - z_0|^{-n}$ when $|z - z_0|$ is small, near z_0 the function behaves like $a_{-n}(z - z_0)^{-n}$. This is why the order of a pole can be estimated by counting the number of times the colors circle a point in the clockwise direction.

Counting Zeros and Poles

The Fundamental Theorem of Algebra states that a polynomial f with highest degree n has exactly n zeros, counting multiplicity. For example, $f(z) = z^2 + 1$ has two zeros, and $f(z) = (z - i)^3$ has three zeros, all at $z_0 = i$ (that is, $z_0 = i$ is a zero with multiplicity 3).

The number of poles of function can also be apparent if it can be written as a quotient of polynomials. For example, $f(z) = z/(z+i)(z-i)^2$ has one zeros and three poles, counting multiplicity.

Problem 4. Plot the following functions and count the number and order of their zeros and poles. Adjust the bounds of each plot until you have found all zeros and poles.

- $f(z) = -4z^5 + 2z^4 - 2z^3 - 4z^2 + 4z - 4$
- $f(z) = z^7 + 6z^6 - 131z^5 - 419z^4 + 4906z^3 - 131z^2 - 420z + 4900$
- $f(z) = \frac{16z^4+32z^3+32z^2+16z+4}{16z^4-16z^3+5z^2}$

It is usually fairly easy to see how many zeros or poles a polynomial or quotient of polynomials has. However, it can be much more difficult to know how many zeros or poles a different function may or may not have without visualizing it.

Problem 5. Plot the following functions on the domain $\{x + iy \mid x, y \in [-8, 8]\}$. Explain carefully in a Markdown cell what each graph reveals about the function and why the function behaves that way.

- $f(z) = e^z$
- $f(z) = \tan(z)$

(Hint: use the polar coordinate representation to mathematically examine the magnitude and angle of each function.)

Essential Poles

A complex-valued function f has an essential pole at z_0 if its Laurent series in a punctured neighborhood of z_0 requires infinitely many terms with negative exponents. For example,

$$e^{1/z} = \sum_{k=0}^{\infty} \frac{1}{n!z^n} = 1 + \frac{1}{z} + \frac{1}{2}\frac{1}{z^2} + \frac{1}{6}\frac{1}{z^3} + \dots$$

An essential pole can be thought of as a pole of order ∞ . Therefore, in an angle plot the colors cycle infinitely many times around an essential pole.

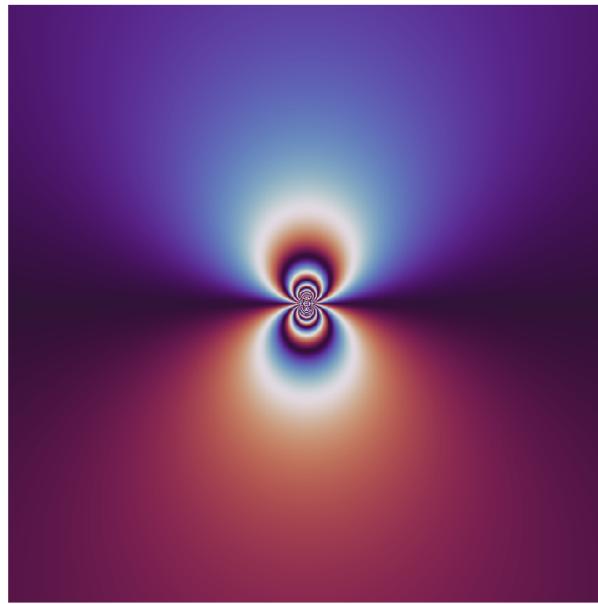


Figure 19.5: Angle plot of $f(z) = e^{1/z}$ on the domain $\{x + iy \mid x, y \in [-1, 1]\}$. The colors circle clockwise around the origin because it is a pole, not a zero. Because the pole is essential, the colors repeat infinitely many times.

Achtung!

Often, color plots like the ones presented in this lab can be deceptive because of a bad choice of domain. Be careful to validate your observations mathematically.

Problem 6. For each of the following functions, plot the function on $\{x + iy \mid x, y \in [-1, 1]\}$ and describe what this view of the plot seems to imply about the function. Then plot the function on a domain that allows you to see the true nature of the roots and poles and describe how it is different from what the original plot implied. Use Markdown cells to write your answers.

- $f(z) = 100z^2 + z$
- $f(z) = \sin\left(\frac{1}{100z}\right)$.

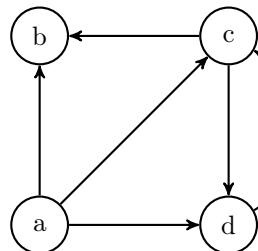
(Hint: zoom way in.)

20 The PageRank Algorithm

Lab Objective: Many real-world systems—the internet, transportation grids, social media, and so on—can be represented as graphs (networks). The PageRank algorithm is one way of ranking the nodes in a graph by importance. Though it is a relatively simple algorithm, the idea gave birth to the Google search engine in 1998 and has shaped much of the information age since then. In this lab we implement the PageRank algorithm with a few different approaches, then use it to rank the nodes of a few different networks.

The PageRank Model

The internet is a collection of webpages, each of which may have a hyperlink to any other page. One possible model for a set of n webpages is a directed graph, where each node represents a page and node j points to node i if page j links to page i . The corresponding adjacency matrix A satisfies $A_{ij} = 1$ if node j links to node i and $A_{ij} = 0$ otherwise.



$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \left[\begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix} \right] \end{matrix}$$

Figure 20.1: A directed unweighted graph with four nodes, together with its adjacency matrix. Note that the column for node b is all zeros, indicating that b is a sink—a node that doesn’t point to any other node.

If n users start on random pages in the network and click on a link every 5 minutes, which page in the network will have the most views after an hour? Which will have the fewest? The goal of the PageRank algorithm is to solve this problem in general, therefore determining how “important” each webpage is.

Before diving into the mathematics, there is a potential problem with the model. What happens if a webpage doesn't have any outgoing links, like node b in Figure 20.1? Eventually, all of the users will end up on page b and be stuck there forever. To obtain a more realistic model, modify each sink in the graph by adding edges from the sink to every node in the graph. This means users on a page with no links can start over by selecting a random webpage.

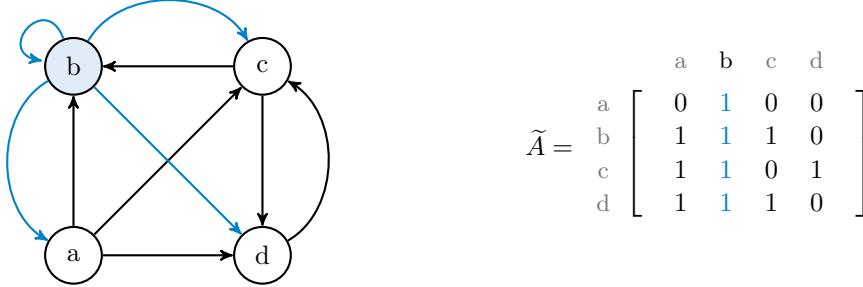


Figure 20.2: Here the graph in Figure 20.1 has been modified to guarantee that node b is no longer a sink (the added links are blue). We denote the modified adjacency matrix by \tilde{A} .

Now let $p_k(t)$ be the likelihood that a particular internet user is surfing webpage k at time t . Suppose at time $t+1$, the user clicks on a link to page i . Then $p_i(t+1)$ can be computed by counting the number of links pointing to page i , weighted by the total number of outgoing links for each node.

As an example, consider the graph in Figure 20.2. To get to page a at time $t+1$, the user had to be on page b at time t . Since there are four outgoing links from page b, assuming links are chosen with equal likelihood,

$$p_a(t+1) = \frac{1}{4}p_b(t).$$

Similarly, to get to page b at time $t+1$, the user had to have been on page a, b, or c at time t . Since a has 3 outgoing edges, b has 4 outgoing edges, and c has 2 outgoing edges,

$$p_b(t+1) = \frac{1}{3}p_a(t) + \frac{1}{4}p_b(t) + \frac{1}{2}p_c(t).$$

The previous equations can be written in a way that hints at a more general linear form:

$$\begin{aligned} p_a(t+1) &= 0p_a(t) + \frac{1}{4}p_b(t) + 0p_c(t) + 0p_d(t), \\ p_b(t+1) &= \frac{1}{3}p_a(t) + \frac{1}{4}p_b(t) + \frac{1}{2}p_c(t) + 0p_d(t). \end{aligned}$$

The coefficients of the terms on the right hand side are precisely the entries of the i th row of the modified adjacency matrix \tilde{A} , divided by the j th column sum. In general, $p_i(t+1)$ satisfies

$$p_i(t+1) = \sum_{j=1}^n \tilde{A}_{ij} \frac{p_j(t)}{\sum_{k=1}^n \tilde{A}_{kj}}. \quad (20.1)$$

Note that the column sum $\sum_{k=1}^n \tilde{A}_{kj}$ in the denominator can never be zero since, after the fix in Figure 20.2, none of the nodes in the graph are sinks.

Accounting for Boredom

The model in (20.1) assumes that the user can only click on links from their current page. It is more realistic to assume that the user sometimes gets bored and randomly picks a new starting page. Let $0 \leq \varepsilon \leq 1$, called the damping factor, be the probability that a user stays interested at step t . Then the probability that the user gets bored at any time (and then chooses a new random page) is $1 - \varepsilon$, and (20.1) becomes

$$p_i(t+1) = \underbrace{\varepsilon \sum_{j=1}^n \left(\tilde{A}_{ij} \frac{p_j(t)}{\sum_{k=1}^n \tilde{A}_{kj}} \right)}_{\text{User stayed interested and clicked a link on the current page}} + \underbrace{\frac{1-\varepsilon}{n}}_{\text{User got bored and chose a random page}}. \quad (20.2)$$

Note that (20.2) can be rewritten as the matrix equation

$$\mathbf{p}(t+1) = \varepsilon \hat{A} \mathbf{p}(t) + \frac{1-\varepsilon}{n} \mathbf{1}, \quad (20.3)$$

where $\mathbf{p}(t) = [p_1(t), p_2(t), \dots, p_n(t)]^\top$, $\mathbf{1}$ is a vector of n ones, and \hat{A} is the $n \times n$ matrix with entries

$$\hat{A}_{ij} = \frac{\tilde{A}_{ij}}{\sum_{k=1}^n \tilde{A}_{kj}}. \quad (20.4)$$

In other words, \hat{A} is \tilde{A} normalized so that the columns each sum to 1. For the graph in Figure 20.2, the matrix \hat{A} is given by

$$\hat{A} = \begin{matrix} & \text{a} & \text{b} & \text{c} & \text{d} \\ \text{a} & 0 & \textcolor{blue}{1/4} & 0 & 0 \\ \text{b} & 1/3 & \textcolor{blue}{1/4} & 1/2 & 0 \\ \text{c} & 1/3 & \textcolor{blue}{1/4} & 0 & 1 \\ \text{d} & 1/3 & \textcolor{blue}{1/4} & 1/2 & 0 \end{matrix}. \quad (20.5)$$

Problem 1. Write a class for representing directed graphs via their adjacency matrices. The constructor should accept an $n \times n$ adjacency matrix A and a list of node labels (such as `[a, b, c, d]`) defaulting to `None`. Modify A as in Figure 20.2 so that there are no sinks in the corresponding graph, then calculate the \hat{A} from (20.4). Save \hat{A} and the list of labels as attributes. Use $[0, 1, \dots, n-1]$ as the labels if none are provided. Finally, raise a `ValueError` if the number of labels is not equal to the number of nodes in the graph.
(Hint: use array broadcasting to compute \hat{A} efficiently.)

For the graph in Figure 20.1, check that your \hat{A} matches (20.5).

Computing the Rankings

In the model (20.2), define the rank of node i as the limit

$$p_i = \lim_{t \rightarrow \infty} p_i(t).$$

There are several ways to solve for $\mathbf{p} = \lim_{t \rightarrow \infty} \mathbf{p}(t)$.

Linear System

If \mathbf{p} exists, then taking the limit as $t \rightarrow \infty$ to both sides of (20.3) gives the following.

$$\begin{aligned} \lim_{t \rightarrow \infty} \mathbf{p}(t+1) &= \lim_{t \rightarrow \infty} \left[\varepsilon \hat{A} \mathbf{p}(t) + \frac{1-\varepsilon}{n} \mathbf{1} \right] \\ \mathbf{p} &= \varepsilon \hat{A} \mathbf{p} + \frac{1-\varepsilon}{n} \mathbf{1} \\ (I - \varepsilon \hat{A}) \mathbf{p} &= \frac{1-\varepsilon}{n} \mathbf{1} \end{aligned} \quad (20.6)$$

This linear system is easy to solve as long as the number of nodes in the graph isn't too large.

Eigenvalue Problem

Let E be an $n \times n$ matrix of ones. Then $E\mathbf{p}(t) = \mathbf{1}$ since $\sum_{i=1}^n p_i(t) = 1$. Substituting into (20.3),

$$\mathbf{p}(t+1) = \varepsilon \hat{A} \mathbf{p}(t) + \frac{1-\varepsilon}{n} E \mathbf{p}(t) = \left(\varepsilon \hat{A} + \frac{1-\varepsilon}{n} E \right) \mathbf{p}(t) = B \mathbf{p}(t), \quad (20.7)$$

where $B = \varepsilon \hat{A} + \frac{1-\varepsilon}{n} E$. Now taking the limit at $t \rightarrow \infty$ of both sides of (20.7),

$$B\mathbf{p} = \mathbf{p}.$$

That is, \mathbf{p} is an eigenvector of B corresponding to the eigenvalue $\lambda = 1$. In fact, since the columns of B sum to 1, and because the entries of B are strictly positive (because the entries of E are all positive), Perron's theorem guarantees that $\lambda = 1$ is the unique eigenvalue of B of largest magnitude, and that the corresponding eigenvector \mathbf{p} is unique up to scaling. Furthermore, \mathbf{p} can be scaled so that each of its entries are positive, meaning $\mathbf{p}/\|\mathbf{p}\|_1$ is the desired PageRank vector.

Note

A Markov chain is a weighted directed graph where each node represents a state of a discrete system. The weight of the edge from node j to node i is the probability of transitioning from state j to state i , and the adjacency matrix of a Markov chain is called a transition matrix.

Since B from (20.7) contains nonnegative entries and its columns all sum to 1, it can be viewed as the transition matrix of a Markov chain. In that context, the limit vector \mathbf{p} is called the steady state of the Markov chain.

Iterative Method

Solving (20.6) or (20.7) is feasible for small networks, but they are not efficient strategies for very large systems. The remaining option is to use an iterative technique. Starting with an initial guess $\mathbf{p}(0)$, use (20.3) to compute $\mathbf{p}(1), \mathbf{p}(2), \dots$ until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|$ is sufficiently small. From (20.7), we can see that this is just the power method¹ for finding the eigenvector corresponding to the dominant eigenvalue of B .

¹See the Least Squares and Computing Eigenvalues lab for details on the power method.

Problem 2. Add the following methods to your class from Problem 1. Each should accept a damping factor ε (defaulting to 0.85), compute the PageRank vector \mathbf{p} , and return a dictionary mapping label i to its PageRank value p_i .

1. `linsolve()`: solve for \mathbf{p} in (20.6).
2. `eigensolve()`: solve for \mathbf{p} using (20.7). Normalize the resulting eigenvector so its entries sum to 1.
3. `itersolve()`: in addition to ε , accept an integer `maxiter` and a float `tol`. Iterate on (20.3) until $\|\mathbf{p}(t) - \mathbf{p}(t-1)\|_1 < \text{tol}$ or $t > \text{maxiter}$. Use $\mathbf{p}(0) = [\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}]^\top$ as the initial vector (any positive vector that sums to 1 will do, but this assumes equal starting probabilities).

Check that each method yields the same results. For the graph in Figure 20.1 with $\varepsilon = 0.85$, you should get the following dictionary mapping labels to PageRank values.

```
{'a': 0.095758635, 'b': 0.274158285, 'c': 0.355924792, 'd': 0.274158285}
```

Problem 3. Write a function that accepts a dictionary mapping labels to PageRank values, like the outputs in Problem 2. Return a list of labels sorted **from highest to lowest** rank. (Hint: if d is a dictionary, use `list(d.keys())` and `list(d.values())` to get the list of keys and values in the dictionary, respectively.)

For the graph in Figure 20.1 with $\varepsilon = 0.85$, this is the list [c, b, d, a] (or [c, d, b, a], since b and d have the same PageRank value).

Problem 4. The file `web_stanford.txt` contains information on Stanford University webpages^a and the hyperlinks between them, gathered in 2002.^b Each line of the file is formatted as `a/b/c/d/e/f...`, meaning the webpage with ID `a` has hyperlinks to webpages with IDs `b`, `c`, `d`, and so on.

Write a function that accepts a damping factor ε defaulting to 0.85. Read the data and get a list of the n unique page IDs in the file (the labels). Construct the $n \times n$ adjacency matrix of the graph where node j points to node i if webpage j has a hyperlink to webpage i . Use your class from Problem 1 and its `itersolve()` method from Problem 2 to compute the PageRank values of the webpages, then rank them with your function from Problem 3. In the case where two webpages have the same rank, resolve ties by first listing the webpage whose ID comes first alphabetically. Note that even though the IDs are numbers, we can sort them alphabetically because they are defined as strings. (Hint: Sorting the list of unique webpage IDs by string before ranking them will place the site IDs in the desired order; there is no need to convert the IDs to integers.) Return the ranked list of webpage IDs.

(Hint: After constructing the list of webpage IDs, make a dictionary that maps a webpage ID to its index in the list. For Figure 20.1, this would be `{'a': 0, 'b': 1, 'c': 2, 'd': 3}`. The values are the row/column indices in the adjacency matrix for each label.)

With $\varepsilon = 0.85$, the top three ranked webpage IDs are 98595, 32791, and 28392.

^a<http://www.stanford.edu/>

^bSee <http://snap.stanford.edu/data/web-Stanford.html> for the original (larger) dataset.

PageRank on Weighted Graphs

Nothing in the formulation of the PageRank model (20.3) requires that the edges of the graph are unweighted. If A_{ij} is the weight of the edge from node j to node i (weight 0 meaning there is no edge from j to i), then the columns of \widehat{A} still sum to 1. Thus $B = \varepsilon\widehat{A} + \frac{1-\varepsilon}{n}E$ is still positive definite, so we can expect a unique PageRank vector \mathbf{p} to exist.

Adding weights to the edges can improve the fidelity of the model and produce a slightly more realistic PageRank ordering. On a given webpage, for example, if hyperlinks to page a are clicked on more frequently hyperlinks to page b , the edge from node a should be given more weight than the edge to node b .

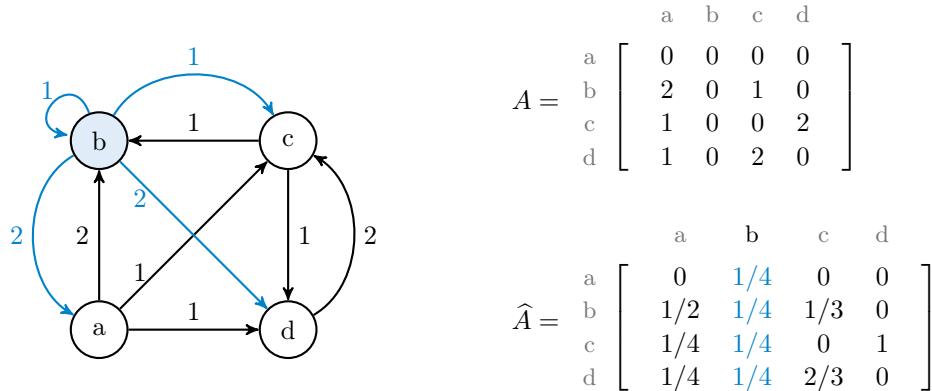


Figure 20.3: A directed weighted graph with four nodes, together with its adjacency matrix and the corresponding PageRank transition matrix. Edges that are added to fix sinks have weight 1, so the computation of \widetilde{A} and \widehat{A} are exactly the same as in Figure 20.2 and (20.4), respectively.

Problem 5. The files `ncaa2010.csv`, `ncaa2011.csv`, ..., `ncaa2017.csv` each contain data for men's college basketball for a given school year.^a Each line (except the very first line, which is a header) represents a different basketball game, formatted `winning_team,losing_team`.

Write a function that accepts a filename and a damping factor ε defaulting to 0.85. Read the specified file (skipping the first line) and get a list of the n unique teams in the file. Construct the $n \times n$ adjacency matrix of the graph where node j points to node i with weight w if team j was defeated by team i in w games. That is, **edges point from losers to winners**. For instance, the graph in Figure 20.3 would indicate that team c lost to team b once and to team d twice, team b was undefeated, and team a never won a game. Use your class from Problem 1 and its `itersolve()` method from Problem 2 to compute the PageRank values of the teams, then rank them with your function from Problem 3. Return the ranked list of team names.

Using `ncaa2010.csv` with $\varepsilon = 0.85$, the top three ranked teams (of the 607 total teams) should be UConn, Kentucky, and Louisville, in that order. That season, UConn won the championship, Kentucky was a semifinalist, and Louisville lost in the first tournament round (a surprising upset).

^a`ncaa2010.csv` has data for the 2010–2011 season, `ncaa2011.csv` for the 2011–2012 season, and so on.

Note

In Problem 5, the damping factor ε acts as an “upset” factor: a larger ε puts more emphasis on win history; a smaller ε allows more randomness in the system, giving underdog teams a higher probability of defeating a team with a better record.

It is also worth noting that the sink-fixing procedure is still reasonable for this model because it gives every other team **equal** likelihood of beating an undefeated team. That is, the additional edges don’t provide an extra advantage to any one team.

PageRank with NetworkX

NetworkX, usually imported as `nx`, is a third-party package for working with networks. It represents graphs internally with dictionaries, thus taking full advantage of the sparsity in a graph. The base class for directed graphs is called `nx.DiGraph`. Nodes and edges are usually added or removed incrementally with the following methods.

Method	Description
<code>add_node()</code>	Add a single node.
<code>add_nodes_from()</code>	Add a list of nodes.
<code>add_edge()</code>	Add an edge between two nodes, adding the nodes if needed.
<code>add_edges_from()</code>	Add multiple edges (and corresponding nodes as needed).
<code>remove_edge()</code>	Remove a single edge (no nodes are removed).
<code>remove_edges_from()</code>	Remove multiple edges (no nodes are removed).
<code>remove_node()</code>	Remove a single node and all adjacent edges.
<code>remove_nodes_from()</code>	Remove multiple nodes and all adjacent edges.

Table 20.1: Methods of the `nx.DiGraph` class for inserting or removing nodes and edges.

For example, the weighted graph in Figure 20.3 can be constructed with the following code.

```
>>> import networkx as nx

# Initialize an empty directed graph.
>>> DG = nx.DiGraph()

# Add the directed edges (nodes are added automatically).
>>> DG.add_edge('a', 'b', weight=2)      # a --> b (adds nodes a and b)
>>> DG.add_edge('a', 'c', weight=1)      # a --> c (adds node c)
>>> DG.add_edge('a', 'd', weight=1)      # a --> d (adds node d)
```

```
>>> DG.add_edge('c', 'b', weight=1)      # c --> b
>>> DG.add_edge('c', 'd', weight=2)      # c --> d
>>> DG.add_edge('d', 'c', weight=2)      # d --> c
```

Once constructed, an `nx.DiGraph` object can be queried for information about the nodes and edges. It also supports dictionary-like indexing to access node and edge attributes, such as the weight of an edge.

Method	Description
<code>has_node(A)</code>	Return <code>True</code> if A is a node in the graph.
<code>has_edge(A,B)</code>	Return <code>True</code> if there is an edge from A to B.
<code>edges()</code>	Iterate through the edges.
<code>nodes()</code>	Iterate through the nodes.
<code>number_of_nodes()</code>	Return the number of nodes.
<code>number_of_edges()</code>	Return the number of edges.

Table 20.2: Methods of the `nx.DiGraph` class for accessing nodes and edges.

```
# Check the nodes and edges.
>>> DG.has_node('a')
True
>>> DG.has_edge('b', 'a')
False
>>> list(DG.nodes())
['a', 'b', 'c', 'd']
>>> list(DG.edges())
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('c', 'b'), ('c', 'd'), ('d', 'c')]

# Change the weight of the edge (a, b) to 3.
>>> DG['a']['b']["weight"] += 1
>>> DG['a']['b']["weight"]
3
```

NetworkX efficiently implements several graph algorithms. The function `nx.pagerank()` computes the PageRank values of each node iteratively with sparse matrix operations. This function returns a dictionary mapping nodes to PageRank values, like the methods in Problem 2.

```
# Calculate the PageRank values of the graph.
>>> nx.pagerank(DG, alpha=0.85)      # alpha is the damping factor (epsilon).
{'a': 0.08767781186947843,
 'b': 0.23613138394239835,
 'c': 0.3661321209576019,
 'd': 0.31005868323052127}
```

Achtung!

NetworkX also has a class, `nx.Graph`, for undirected graphs. The edges in an undirected graph are bidirectional, so the corresponding adjacency matrix is symmetric.

The PageRank algorithm is not very useful for undirected graphs. In fact, the PageRank value for node is close to its *degree*—the number of edges it connects to—divided by the total number of edges. In Problem 5, that would mean the team who simply played the most games would be ranked the highest. Always use `nx.DiGraph`, not `nx.Graph`, for PageRank and other algorithms that rely on directed edges.

Problem 6. The file `top250movies.txt` contains data from the 250 top-rated movies according to IMDb.^a Each line in the file lists a movie title and its cast as `title/actor1/actor2/...`, with the actors listed mostly in billing order (stars first), though some casts are listed alphabetically or in order of appearance.

Create a `nx.DiGraph` object with a node for each actor in the file. The weight from actor *a* to actor *b* should be the number of times that actor *a* and *b* were in a movie together but actor *b* was listed first. That is, **edges point to higher-billed actors** (see Figure 20.4). Compute the PageRank values of the actors and use your function from Problem 3 to rank them. Return the list of ranked actors.

(Hint: Consider using `itertools.combinations()` while constructing the graph. Also, use `encoding="utf-8"` as an argument to `open()` to read the file, since several actors and actresses have nonstandard characters in their names such as ø and æ.)

With $\varepsilon = 0.7$, the top three actors should be Leonardo DiCaprio, Robert De Niro, and Tom Hanks, in that order.

^ahttps://www.imdb.com/search/title?groups=top_250&sort=user_rating

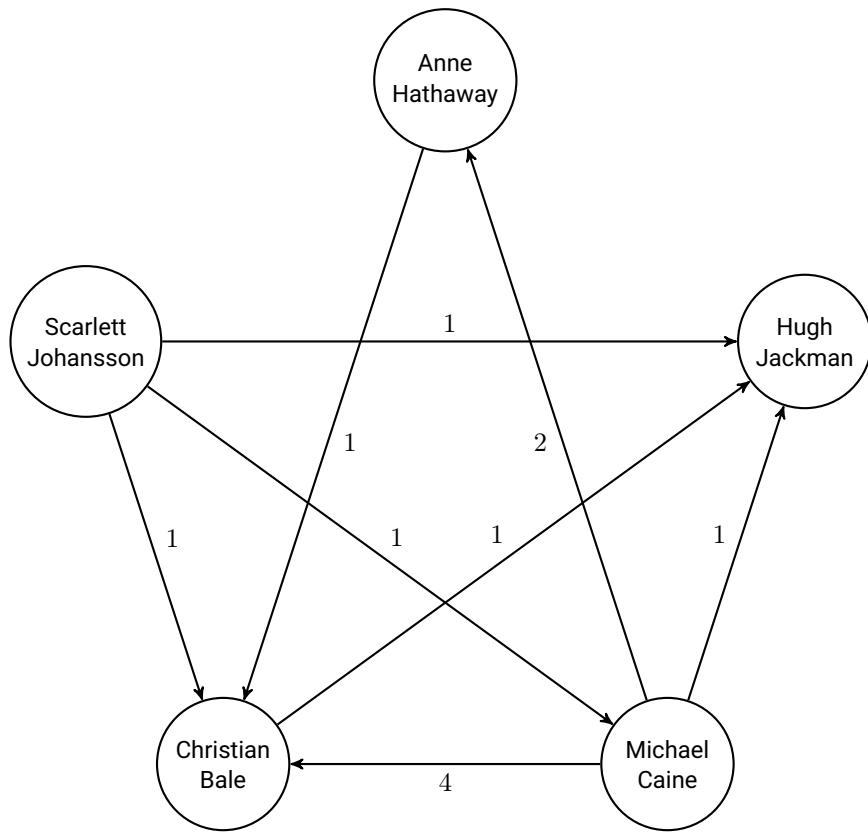


Figure 20.4: A portion of the graph from Problem 6. Michael Caine was in four movies with Christian Bale where Christian Bale was listed first in the cast.

Additional Material

Sparsity

On very large networks, the PageRank algorithm becomes computationally difficult because of the size of the adjacency matrix A . Fortunately, most adjacency matrices are highly sparse, meaning the number of edges is much lower than the number of entries in the matrix. Consider adding functionality to your class from Problem 1 so that it stores \hat{A} as a sparse matrix and performs sparse linear algebra operations in the methods from Problem 2 (use `scipy.sparse.linalg`).

PageRank as a Predictive Model

The data files in Problem 5 include tournament games for their respective seasons, so the resulting rankings naturally align with the outcome of the championship. However, it is also useful to use PageRank as a predictive model: given data for all regular season games, can the outcomes of the tournament games be predicted? Over 40 million Americans fill out 60–100 million March Madness brackets each year and bet over \$9 billion on the tournament, so being able to predict the outcomes of the games is a big deal. See <http://games.espn.com/tournament-challenge-bracket> for more details.

Given regular season data, PageRank can be used to predict tournament results as in Problem 5. There are some pitfalls though; for example, how should ε be chosen? Using $\varepsilon = .5$ with `ncaa2010.csv` minus tournament data (all but the last 63 games in the file) puts UConn—the actual winner that year—in seventh place, while $\varepsilon = .9$ puts UConn in fourth. Both values for ε also rank BYU as number one, but BYU lost in the Sweet Sixteen that year. In practice, Google uses .85 as the damping factor, but there is no rigorous reasoning behind that particular choice.

Other Centrality Measures

In network theory, the centrality of a node refers to its importance. Since there are lots of ways to measure importance, there are several different centrality measures.

- Degree centrality uses the degree of a node, meaning the number of edges adjacent to it (independent of edge direction), for ranking. An academic paper that has been cited many times has a high degree and is considered more important than a paper that has only been cited once.
- Eigenvector centrality is an extension of degree centrality. Instead of each neighbor contributing equally to the centrality, nodes that are important are given a higher weight. Thus a node connected to lots of unimportant nodes can have the same measure as a node connected to a few, important nodes. Eigenvector centrality is measured by the eigenvector associated with the largest eigenvalue of the adjacency matrix of the network.
- Katz centrality is a modification to eigenvalue centrality for directed networks. Outgoing nodes contribute centrality to their neighbors, so an important node makes its neighbors more important.
- PageRank adapts Katz centrality by averaging out the centrality that a node can pass to its neighbors. For example, if Google—a website that should have high centrality—points to a million websites, then it shouldn’t pass on that high centrality to all of million of its neighbors, so each neighbor gets one millionth of Google’s centrality.

For more information on these centralities, as well as other ways to measure node importance, see [New10].

21 Profiling

Lab Objective: Efficiency is essential to algorithmic programming. Profiling is the process of measuring the complexity and efficiency of a program, allowing the programmer to see what parts of the code need to be optimized. In this lab we present common techniques for speeding up Python code, including the built-in profiler and the Numba module.

Magic Commands in IPython

IPython has tools for quickly timing and profiling code. These “magic commands” start with one or two % characters—one for testing a single line of code, and two for testing a block of code.

- `%time`: Execute some code and print out its execution time.
- `%timeit`: Execute some code several times and print out the average execution time.
- `%prun`: Run a statement through the Python code profiler,¹ printing the number of function calls and the time each takes. We will demonstrate this tool a little later.

```
# Time the construction of a list using list comprehension.  
In [1]: %time x = [i**2 for i in range(int(1e5))]  
CPU times: user 36.3 ms, sys: 3.28 ms, total: 39.6 ms  
Wall time: 40.9 ms  
  
# Time the same list construction, but with a regular for loop.  
In [2]: %%time # Use a double %% to time a block of code.  
....: x = []  
....: for i in range(int(1e5)):  
....:     x.append(i**2)  
....:  
CPU times: user 50 ms, sys: 2.79 ms, total: 52.8 ms  
Wall time: 55.2 ms # The list comprehension is faster!
```

¹`%prun` is a shortcut for `cProfile.run()`; see <https://docs.python.org/3/library/profile.html> for details.

Choosing Faster Algorithms

The best way to speed up a program is to use an efficient algorithm. A bad algorithm, even when implemented well, is never an adequate substitute for a good algorithm.

Problem 1. This problem comes from <https://projecteuler.net> (problems 18 and 67).

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

```

3
7 4
2 4 6
8 5 9 3

```

That is, $3 + 7 + 4 + 9 = 23$.

The following function finds the maximum path sum of the triangle in `triangle.txt` by recursively computing the sum of every possible path—the “brute force” approach.

```

def max_path(filename="triangle.txt"):
    """Find the maximum vertical path in a triangle of values."""
    with open(filename, 'r') as infile:
        data = [[int(n) for n in line.split()]
                for line in infile.readlines()]
    def path_sum(r, c, total):
        """Recursively compute the max sum of the path starting in row r
        and column c, given the current total.
        """
        total += data[r][c]
        if r == len(data) - 1:      # Base case.
            return total
        else:                      # Recursive case.
            return max(path_sum(r+1, c, total), # Next row, same column.
                       path_sum(r+1, c+1, total)) # Next row, next column.

    return path_sum(0, 0, 0)       # Start the recursion from the top.

```

The data in `triangle.txt` contains 15 rows and hence 16384 paths, so it is possible to solve this problem by trying every route. However, for a triangle with 100 rows, there are 2^{99} paths to check, which would take billions of years to compute even for a program that could check one trillion routes per second. No amount of improvement to `max_path()` can make it run in an acceptable amount of time on such a triangle—we need a different algorithm.

Write a function that accepts a filename containing a triangle of integers. Compute the largest path sum with the following strategy: starting from the next to last row of the triangle, replace each entry with the sum of the current entry and the greater of the two “child entries.” Continue this replacement up through the entire triangle. The top entry in the triangle will be the maximum path sum. In other words, work from the bottom instead of from the top.

3	3	3	23
7 4	7 4	20 19	20 19
2 4 6	10 13 15	10 13 15	10 13 15
8 5 9 3	8 5 9 3	8 5 9 3	8 5 9 3

Use your function to find the maximum path sum of the 100-row triangle stored in `triangle_large.txt`. Make sure that your new function still gets the correct answer for the smaller `triangle.txt`. Finally, use `%time` or `%timeit` to time both functions on `triangle.txt`. Your new function should be about 100 times faster than the original.

The Profiler

The profiling command `%prun` lists the functions that are called during the execution of a piece of code, along with the following information.

Heading	Description
<code>primitive calls</code>	The number of calls that were not caused by recursion.
<code>ncalls</code>	The number of calls to the function. If recursion occurs, the output is <code><total number of calls>/<number of primitive calls></code> .
<code>tottime</code>	The amount of time spent in the function, not including calls to other functions.
<code>percall</code>	The amount of time spent in each call of the function.
<code>cumtime</code>	The amount of time spent in the function, including calls to other functions.

```
# Profile the original function from Problem 1.
In[3]: %prun max_path("triangle.txt")
```

```
81947 function calls (49181 primitive calls) in 0.036 seconds
Ordered by: internal time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
32767/1    0.025    0.000    0.034    0.034  profiling.py:18(path_sum)
 16383    0.005    0.000    0.005    0.000  {built-in method builtins.max}
 32767    0.003    0.000    0.003    0.000  {built-in method builtins.len}
   1    0.002    0.002    0.002    0.002  {method 'readlines' of '_io._IOBase' objects}
   1    0.000    0.000    0.000    0.000  {built-in method io.open}
   1    0.000    0.000    0.036    0.036  profiling.py:12(max_path)
   1    0.000    0.000    0.000    0.000  profiling.py:15(<listcomp>)
   1    0.000    0.000    0.036    0.036  {built-in method builtins.exec}
   2    0.000    0.000    0.000    0.000  codecs.py:318(decode)
   1    0.000    0.000    0.036    0.036  <string>:1(<module>)
  15    0.000    0.000    0.000    0.000  {method 'split' of 'str' objects}
   1    0.000    0.000    0.000    0.000  _bootlocale.py:23(getpreferredencoding)
   2    0.000    0.000    0.000    0.000  {built-in method _codecs.utf_8_decode}
   1    0.000    0.000    0.000    0.000  {built-in method _locale.nl_langinfo}
   1    0.000    0.000    0.000    0.000  codecs.py:259(__init__)
   1    0.000    0.000    0.000    0.000  codecs.py:308(__init__)
   1    0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler' objects}
```

Optimizing Python Code

A poor implementation of a good algorithm is better than a good implementation of a bad algorithm, but clumsy implementation can still cripple a program's efficiency. The following are a few important practices for speeding up a Python program. Remember, however, that such improvements are futile if the algorithm is poorly suited for the problem.

Avoid Repetition

A clean program does no more work than is necessary. The `ncalls` column of the profiler output is especially useful for identifying parts of a program that might be repetitive. For example, the profile of `max_path()` indicates that `len()` was called 32,767 times—exactly as many times as `path_sum()`. This is an easy fix: save `len(data)` as a variable somewhere outside of `path_sum()`.

```
In [4]: def max_path_clean(filename="triangle.txt"):
....    with open(filename, 'r') as infile:
....        data = [[int(n) for n in line.split()]
....                  for line in infile.readlines()]
....    N = len(data)      # Calculate len(data) outside of path_sum().
....    def path_sum(r, c, total):
....        total += data[r][c]
....        if r == N - 1: # Use N instead of len(data).
....            return total
....        else:
....            return max(path_sum(r+1, c, total),
....                       path_sum(r+1, c+1, total))
....    return path_sum(0, 0, 0)
....:
In [5]: %prun max_path_clean("triangle.txt")
```

```
49181 function calls (16415 primitive calls) in 0.026 seconds
Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
32767/1    0.020    0.000    0.025    0.025 <ipython-input-5-9e8c48bb1aba>:6(path_sum)
 16383    0.005    0.000    0.005    0.000 {built-in method builtins.max}
   1    0.002    0.002    0.002    0.002 {method 'readlines' of '_io._IOBase' objects}
   1    0.000    0.000    0.000    0.000 {built-in method io.open}
   1    0.000    0.000    0.026    0.026 <ipython-input-5-9e8c48bb1aba>:1(max_path_clean)
   1    0.000    0.000    0.000    0.000 <ipython-input-5-9e8c48bb1aba>:3(<listcomp>)
   1    0.000    0.000    0.027    0.027 {built-in method builtins.exec}
  15    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
   1    0.000    0.000    0.027    0.027 <string>:1(<module>)
   2    0.000    0.000    0.000    0.000 codecs.py:318(decode)
   1    0.000    0.000    0.000    0.000 _bootlocale.py:23(getpreferredencoding)
   2    0.000    0.000    0.000    0.000 {built-in method _codecs.utf_8_decode}
   1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
   1    0.000    0.000    0.000    0.000 codecs.py:308(__init__)
   1    0.000    0.000    0.000    0.000 codecs.py:259(__init__)
   1    0.000    0.000    0.000    0.000 {built-in method builtins.len}
   1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Note that the total number of primitive function calls decreased from 49,181 to 16,415. Using `%timeit` also shows that the run time decreased by about 15%. Moving code outside of a loop or an often-used function usually results in a similar speedup.

Another important way of reducing repetition is carefully controlling loop conditions to avoid unnecessary iterations. Consider the problem of identifying Pythagorean triples, sets of three distinct integers $a < b < c$ such that $a^2 + b^2 = c^2$. The following function identifies all such triples where each term is less than a parameter N by checking all possible triples.

```
>>> def pythagorean_triples_slow(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):           # Try values of a from 1 to N-1.
...         for b in range(1, N):       # Try values of b from 1 to N-1.
...             for c in range(1, N):   # Try values of c from 1 to N-1.
...                 if a**2 + b**2 == c**2 and a < b < c:
...                     triples.append((a,b,c))
...     return triples
...
```

Since $a < b < c$ by definition, any computations where $b \leq a$ or $c \leq b$ are unnecessary. Additionally, once a and b are chosen, c can be no greater than $\sqrt{a^2 + b^2}$. The following function changes the loop conditions to avoid these cases and takes care to only compute $a^2 + b^2$ once for each unique pairing (a, b) .

```
>>> from math import sqrt
>>> def pythagorean_triples_fast(N):
...     """Compute all pythagorean triples with entries less than N."""
...     triples = []
...     for a in range(1, N):           # Try values of a from 1 to N-1.
...         for b in range(a+1, N):      # Try values of b from a+1 to N-1.
...             _sum = a**2 + b**2
...             for c in range(b+1, min(int(sqrt(_sum))+1, N)):
...                 if _sum == c**2:
...                     triples.append((a,b,c))
...     return triples
...
```

These improvements have a drastic impact on run time, even though the main approach—checking by brute force—is the same.

```
In [6]: %time triples = pythagorean_triples_slow(500)
CPU times: user 1min 51s, sys: 389 ms, total: 1min 51s
Wall time: 1min 52s      # 112 seconds.

In [7]: %time triples = pythagorean_triples_fast(500)
CPU times: user 1.56 s, sys: 5.38 ms, total: 1.57 s
Wall time: 1.57 s        # 98.6% faster!
```

Problem 2. The following function computes the first N prime numbers.

```
def primes(N):
    """Compute the first N primes."""
    primes_list = []
    current = 2
    while len(primes_list) < N:
        isprime = True
        for i in range(2, current):
            if current % i == 0:
                isprime = False
        if isprime:
            primes_list.append(current)
        current += 1
    return primes_list
```

This function takes about 6 minutes to find the first 10,000 primes on a fast computer.

Without significantly modifying the approach, rewrite `primes()` so that it can compute the first 10,000 primes in under 0.1 seconds. Use the following facts to reduce unnecessary iterations.

- A number is not prime if it has one or more divisors other than 1 and itself.
(Hint: recall the `break` statement.)
- If $p \nmid n$, then $ap \nmid n$ for any integer a . Also, if $p \mid n$ and $0 < p < n$, then $p \leq \sqrt{n}$.
- Except for 2, primes are always odd.

Your new function should be helpful for solving problem 7 on <https://projecteuler.net>.

Avoid Loops

NumPy routines and built-in functions are often useful for eliminating loops altogether. Consider the simple problem of summing the rows of a matrix, implemented in three ways.

```
>>> def row_sum_awful(A):
...     """Sum the rows of A by iterating through rows and columns."""
...     m,n = A.shape
...     row_totals = np.empty(m)           # Allocate space for the output.
...     for i in range(m):               # For each row...
...         total = 0
...         for j in range(n):           # ...iterate through the columns.
...             total += A[i,j]
...         row_totals[i] = total       # Record the total.
...     return row_totals
...
>>> def row_sum_bad(A):
...     """Sum the rows of A by iterating through rows."""
```

```

...     return np.array([sum(A[i,:]) for i in range(A.shape[0])])
...
>>> def row_sum_fast(A):
...     """Sum the rows of A with NumPy."""
...     return np.sum(A, axis=1)      # Or A.sum(axis=1).
...

```

None of the functions are fundamentally different, but their run times differ dramatically.

```

In [8]: import numpy as np
In [9]: A = np.random.random((10000, 10000))

In [10]: %time rows = row_sum_awful(A)
CPU times: user 22.7 s, sys: 137 ms, total: 22.8 s
Wall time: 23.2 s          # SLOW!

In [11]: %time rows = row_sum_bad(A)
CPU times: user 8.85 s, sys: 15.6 ms, total: 8.87 s
Wall time: 8.89 s          # Slow!

In [12]: %time rows = row_sum_fast(A)
CPU times: user 61.2 ms, sys: 1.3 ms, total: 62.5 ms
Wall time: 64 ms           # Fast!

```

In this experiment, `row_sum_fast()` runs several hundred times faster than `row_sum_awful()`. This is primarily because looping is expensive in Python, but NumPy handles loops in C, which is much quicker. Other NumPy functions like `np.sum()` with an `axis` argument can often be used to eliminate loops in a similar way.

Problem 3. Let A be an $m \times n$ matrix with columns $\mathbf{a}_0, \dots, \mathbf{a}_{n-1}$, and let \mathbf{x} be a vector of length m . The nearest neighbor problem^a is to determine which of the columns of A is “closest” to \mathbf{x} with respect to some norm. That is, we compute

$$\operatorname{argmin}_j \|\mathbf{a}_j - \mathbf{x}\|.$$

The following function solves this problem naïvely for the usual Euclidean norm.

```

def nearest_column(A, x):
    """Find the index of the column of A that is closest to x."""
    distances = []
    for j in range(A.shape[1]):
        distances.append(np.linalg.norm(A[:,j] - x))
    return np.argmin(distances)

```

Write a new version of this function without any loops or list comprehensions, using array broadcasting and the `axis` keyword in `np.linalg.norm()` to eliminate the existing loop. Try to implement the entire function in a single line.

(Hint: See the NumPy Visual Guide in the Appendix for a refresher on array broadcasting.)

Profile the old and new versions with `%prun` and compare the output. Finally, use `%time` or `%timeit` to verify that your new version runs faster than the original.

^aThe nearest neighbor problem is a common problem in many fields of artificial intelligence. The problem can be solved more efficiently with a *k*-d tree, a specialized data structure for storing high-dimensional data.

Use Data Structures Correctly

Every data structure has strengths and weaknesses, and choosing the wrong data structure can be costly. Here we consider three ways to avoid problems and use sets, dictionaries, and lists correctly.

- **Membership testing.** The question “is <value> a member of <container>” is common in numerical algorithms. Sets and dictionaries are implemented in a way that makes this a trivial problem, but lists are not. In other words, the `in` operator is near instantaneous with sets and dictionaries, but not with lists.

```
In [13]: a_list = list(range(int(1e7)))

In [14]: a_set = set(a_list)

In [15]: %timeit 12.5 in a_list
413 ms +- 48.2 ms per loop (mean+-std.dev. of 7 runs, 1 loop each)

In [16]: %timeit 12.5 in a_set
170 ns +- 3.8 ns per loop (mean+-std.dev. of 7 runs, 10000000 loops each)
```

Looking up dictionary values is also almost immediate. Use dictionaries for storing calculations to be reused, such as mappings between letters and numbers or common function outputs.

- **Construction with comprehension.** Lists, sets, and dictionaries can all be constructed with comprehension syntax. This is slightly faster than building the collection in a loop, and the code is highly readable.

```
# Map the integers to their squares.

In [17]: %%time
....: a_dict = {}
....: for i in range(1000000):
....:     a_dict[i] = i**2
....:
CPU times: user 432 ms, sys: 54.4 ms, total: 486 ms
Wall time: 491 ms

In [18]: %time a_dict = {i:i**2 for i in range(1000000)}
CPU times: user 377 ms, sys: 58.9 ms, total: 436 ms
Wall time: 440 ms
```

- **Intelligent iteration.** Unlike looking up dictionary values, indexing into lists takes time. Instead of looping over the indices of a list, loop over the entries themselves. When indices and entries are both needed, use `enumerate()` to get the index and the item simultaneously.

```
In [19]: a_list = list(range(1000000))

In [20]: %%time          # Loop over the indices of the list.
...: for i in range(len(a_list)):
...:     item = a_list[i]
...:
CPU times: user 103 ms, sys: 1.78 ms, total: 105 ms
Wall time: 107 ms

In [21]: %%time          # Loop over the items in the list.
...: for item in a_list:
...:     _ = item
...:
CPU times: user 61.2 ms, sys: 1.31 ms, total: 62.5 ms
Wall time: 62.5 ms      # Almost twice as fast as indexing!
```

Problem 4.

This is problem 22 from <https://projecteuler.net>.

Using the rule $A \mapsto 1, B \mapsto 2, \dots, Z \mapsto 26$, the alphabetical value of a name is the sum of the digits that correspond to the letters in the name. For example, the alphabetic value of “COLIN” is $3 + 15 + 12 + 9 + 14 = 53$.

The following function reads the file `names.txt`, containing over five-thousand first names, and sorts them in alphabetical order. The name score of each name in the resulting list is the alphabetic value of the name multiplied by the name’s position in the list, starting at 1. “COLIN” is the 938th name alphabetically, so its name score is $938 \times 53 = 49714$. The function returns the total of all the name scores in the file.

```
def name_scores(filename="names.txt"):
    """Find the total of the name scores in the given file."""
    with open(filename, 'r') as infile:
        names = sorted(infile.read().replace('"', '').split(','))
    total = 0
    for i in range(len(names)):
        name_value = 0
        for j in range(len(names[i])):
            alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            for k in range(len(alphabet)):
                if names[i][j] == alphabet[k]:
                    letter_value = k + 1
                    name_value += letter_value
        total += (names.index(names[i]) + 1) * name_value
    return total
```

Rewrite this function—removing repetition, eliminating loops, and using data structures correctly—so that it runs in less than 10 milliseconds on average.

Use Generators

A generator is an iterator that yields multiple values, one at a time, as opposed to returning a single value. For example, `range()` is a generator. Using generators appropriately can reduce both the run time and the spatial complexity of a routine. Consider the following function, which constructs a list containing the entries of the sequence $\{x_n\}_{n=1}^N$ where $x_n = x_{n-1} + n$ with $x_1 = 1$.

```
>>> def sequence_function(N):
...     """Return the first N entries of the sequence x_n = x_{n-1} + n."""
...     sequence = []
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         sequence.append(x)
...     return sequence
...
>>> sequence_function(10)
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

A potential problem with this function is that all of the values in the list are computed before anything is returned. This can be a big issue if the parameter N is large. A generator, on the other hand, yields one value at a time, indicated by the keyword `yield` (instead of `return`). When the generator is asked for the next entry, the code resumes right where it left off.

```
>>> def sequence_generator(N):
...     """Yield the first N entries of the sequence x_n = x_{n-1} + n."""
...     x = 0
...     for n in range(1, N+1):
...         x += n
...         yield x      # "return" a single value.
...
# Get the entries of the generator one at a time with next().
>>> generated = sequence_generator(10)
>>> next(generated)
1
>>> next(generated)
3
>>> next(generated)
6

# Put each of the generated items in a list, as in sequence_function().
>>> list(sequence_generator(10))    # Or [i for i in sequence_generator(10)].
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]

# Use the generator in a for loop, like range().
for i in sequence_generator(10):
    print(i)
```

```
>>> for entry in sequence_generator(10):
...     print(entry, end=' ')
...
1 3 6 10 15 21 28 36 45 55
```

Many generators, like `range()` and `sequence_generator()`, only yield a finite number of values. However, generators can also continue yielding indefinitely. For example, the following generator yields the terms of $\{x_n\}_{n=1}^{\infty}$ forever. In this case, using `enumerate()` with the generator is helpful for tracking the index n as well as the entry x_n .

```
>>> def sequence_generator_forever():
...     """Yield the sequence  $x_n = x_{n-1} + n$  forever."""
...     x = 0
...     n = 1
...     while True:
...         x += n
...         n += 1
...         yield x      # "return" a single value.
...
#
# Sum the entries of the sequence until the sum exceeds 1000.
>>> total = 0
>>> for i, x in enumerate(sequence_generator_forever()):
...     total += x
...     if total > 1000:
...         print(i)      # Print the index where the total exceeds.
...         break         # Break out of the for loop to stop iterating.
...
17
#
# Check that 18 terms are required (since i starts at 0 but n starts at 1).
>>> print(sum(sequence_generator(17)), sum(sequence_generator(18)))
969 1140
```

Problem 5. This is problem 25 from <https://projecteuler.net>.

The Fibonacci sequence is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, where $F_1 = F_2 = 1$. The 12th term, $F_{12} = 144$, is the first term to contain three digits.

Write a generator that yields the terms of the Fibonacci sequence indefinitely. Next, write a function that accepts an integer N . Use your generator to find the first term in the Fibonacci sequence that contains N digits. Return the index of this term.

(Hint: a generator can have more than one `yield` statement.)

Problem 6. The function in Problem 2 could be turned into a prime number generator that yields primes indefinitely, but it is not the only strategy for yielding primes. The Sieve of Eratosthenes^a is a faster technique for finding all of the primes below a certain number.

1. Given a cap N , start with all of the integers from 2 to N .
2. Remove all integers that are divisible by the first entry in the list.
3. Yield the first entry in the list and remove it from the list.
4. Return to step 2 until the list is empty.

Write a generator that accepts an integer N and that yields all primes (in order, one at a time) that are less than N using the Sieve of Eratosthenes. Your generator should be able to find all primes less than 100,000 in under 5 seconds.

Your generator and your fast function from Problem 2 may be helpful in solving problems 10, 35, 37, 41, 49, and 50 (for starters) of <https://projecteuler.net>.

^aSee https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

Numba

Python code is simpler and more readable than many languages, but Python is also generally much slower than compiled languages like C. The `numba` module bridges the gap by using just-in-time (JIT) compilation to optimize code, meaning that the code is actually compiled right before execution.

```
>>> from numba import jit

>>> @jit                  # Decorate a function with @jit to use Numba.
... def row_sum_numba(A):
...     """Sum the rows of A by iterating through rows and columns,
...     optimized by Numba.
...
...     m,n = A.shape
...     row_totals = np.empty(m)
...     for i in range(m):
...         total = 0
...         for j in range(n):
...             total += A[i,j]
...         row_totals[i] = total
...     return row_totals
```

Python is a dynamically typed language, meaning variables are not defined explicitly with a datatype (`x = 6` as opposed to `int x = 6`). This particular aspect of Python makes it flexible, easy to use, and slow. Numba speeds up Python code primarily by assigning datatypes to all the variables. Rather than requiring explicit definitions for datatypes, Numba attempts to infer the correct datatypes based on the datatypes of the input. In `row_sum_numba()`, if `A` is an array of integers, Numba will infer that `total` should also be an integer. On the other hand, if `A` is an array of floats, Numba will infer that `total` should be a double (a similar datatype to float in C).

Once all datatypes have been inferred and assigned, the original Python code is translated to machine code. Numba caches this compiled version of code for later use. The first function call takes the time to compile and then execute the code, but subsequent calls use the already-compiled code.

```
In [22]: A = np.random.random((10000, 10000))

# The first function call takes a little extra time to compile first.
In [23]: %time rows = row_sum_numba(A)
CPU times: user 408 ms, sys: 11.5 ms, total: 420 ms
Wall time: 425 ms

# Subsequent calls are consistently faster than the first call.
In [24]: %timeit row_sum_numba(A)
138 ms +- 1.96 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Note that the only difference between `row_sum_numba()` and `row_sum_awful()` from a few pages ago is the `@jit` decorator, and yet the Numba version is about 99% faster than the original!

The inference engine within Numba does a good job, but it's not always perfect. Adding the keyword argument `nopython=True` to the `@jit` decorator raises an error if Numba is unable to convert each variable to explicit datatypes. The `inspect_types()` method can also be used to check if Numba is using the desired types.

```
# Run the function once first so that it compiles.
>>> rows = row_sum_numba(np.random.random((10,10)))
>>> row_sum_numba.inspect_types()
# The output is very long and detailed.
```

Alternatively, datatypes can be specified explicitly in the `@jit` decorator as a dictionary via the `locals` keyword argument. Each of the desired datatypes must also be imported from Numba.

```
>>> from numba import int64, double

>>> @jit(nopython=True, locals=dict(A=double[:, :], m=int64, n=int64,
...                                     row_totals=double[:, total=double]))
...     def row_sum_numba(A):
...         # 'A' is a 2-D array of doubles.
...         m,n = A.shape
...         # 'm' and 'n' are both integers.
...         row_totals = np.empty(m)
...         # 'row_totals' is a 1-D array of doubles.
...         for i in range(m):
...             total = 0
...             # 'total' is a double.
...             for j in range(n):
...                 total += A[i,j]
...             row_totals[i] = total
...     return row_totals
...
```

While it sometimes results in a speed boost, there is a caveat to specifying the datatypes: `row_sum_numba()` no longer accepts arrays that contain anything other than floats. When datatypes are not specified, Numba compiles a new version of the function each time the function is called with a different kind of input. Each compiled version is saved, so the function can still be used flexibly.

Problem 7. The following function calculates the n th power of an $m \times m$ matrix A .

```
def matrix_power(A, n):
    """Compute A^n, the n-th power of the matrix A."""
    product = A.copy()
    temporary_array = np.empty_like(A[0])
    m = A.shape[0]
    for power in range(1, n):
        for i in range(m):
            for j in range(m):
                total = 0
                for k in range(m):
                    total += product[i,k] * A[k,j]
                temporary_array[j] = total
            product[i] = temporary_array
    return product
```

1. Write a Numba-enhanced version of `matrix_power()` called `matrix_power_numba()`.
2. Write a function that accepts an integer n . Run `matrix_power_numba()` once with a small random input so it compiles. Then, for $m = 2^2, 2^3, \dots, 2^7$,
 - (a) Generate a random $m \times m$ matrix A with `np.random.random()`.
 - (b) Time (separately) `matrix_power()`, `matrix_power_numba()`, and NumPy's `np.linalg.matrix_power()` on A with the specified value of n .
(If you are unfamiliar with timing code inside of a function, see the Additional Material section on timing code.)

Plot the times against the size m on a log-log plot with a base 2 scale (use `plt.loglog()`).

With $n = 10$, the plot should show that the Numba and NumPy versions far outperform the pure Python implementation, with NumPy eventually becoming faster than Numba.

Achtung!

Optimizing code is an important skill, but it is also important to know when to refrain from optimization. The best approach to coding is to write unit tests, implement a solution that works, test and time that solution, **then** (and only then) optimize the solution with profiling techniques. As always, the most important part of the process is choosing the correct algorithm to solve the problem. Don't waste time optimizing a poor algorithm.

Additional Material

Other Timing Techniques

Though `%time` and `%timeit` are convenient and work well, some problems require more control for measuring execution time. The usual way of timing a code snippet by hand is via the `time` module (which `%time` uses). The function `time.time()` returns the number of seconds since the Epoch²; to time code, measure the number of seconds before the code runs, the number of seconds after the code runs, and take the difference.

```
>>> import time

>>> start = time.time()           # Record the current time.
>>> for i in range(int(1e8)):    # Execute some code.
...     pass
... end = time.time()            # Record the time again.
... print(end - start)          # Take the difference.
...
4.20402193069458 # (seconds)
```

The `timeit` module (which `%timeit` uses) has tools for running code snippets several times. The code is passed in as a string, as well as any setup code to be run before starting the clock.

```
>>> import timeit

>>> timeit.timeit("for i in range(N): pass", setup="N = int(1e6)", number=200)
4.884839255013503      # Total time in seconds to run the code 200 times.
>>> _ / 200
0.024424196275067516  # Average time in seconds.
```

The primary advantages of these techniques are the ability automate timing code and being able save the results. For more documentation, see <https://docs.python.org/3.6/library/time.html> and <https://docs.python.org/3.6/library/timeit.html>.

Customizing the Profiler

The output from `%prun` is generally long, but it can be customized with the following options.

Option	Description
<code>-l <limit></code>	Include a limited number of lines in the output.
<code>-s <key></code>	Sort the output by call count, cumulative time, function name, etc.
<code>-T <filename></code>	Save profile results to a file (results are still printed).

For example, `%prun -l 3 -s ncalls -T path_profile.txt max_path()` generates a profile of `max_path()` that lists the 3 functions with the most calls, then write the results to `path_profile.txt`. See <http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-prun> for more details.

²See [https://en.wikipedia.org/wiki/Epoch_\(reference_date\)#Computing](https://en.wikipedia.org/wiki/Epoch_(reference_date)#Computing).

22 SQL 1: Introduction

Lab Objective: Being able to store and manipulate large data sets quickly is a fundamental part of data science. The SQL language is the classic database management system for working with tabular data. In this lab we introduce the basics of SQL, including creating, reading, updating, and deleting SQL tables, all via Python's standard SQL interaction modules.

Relational Databases

A relational database is a collection of tables called relations. A single row in a table, called a tuple, corresponds to an individual instance of data. The columns, called attributes or features, are data values of a particular category. The collection of column headings is called the schema of the table, which describes the kind of information stored in each entry of the tuples.

For example, suppose a database contains demographic information for M individuals. If a table had the schema `(Name, Gender, Age)`, then each row of the table would be a 3-tuple corresponding to a single individual, such as `(Jane Doe, F, 20)` or `(Samuel Clemens, M, 74.4)`. The table would therefore be $M \times 3$ in shape. Note that including a person's age in a database means that the data would quickly be outdated since people get older every year. A better choice would be to use birth year. Another table with the schema `(Name, Income)` would be $M \times 2$ if it included all M individuals.

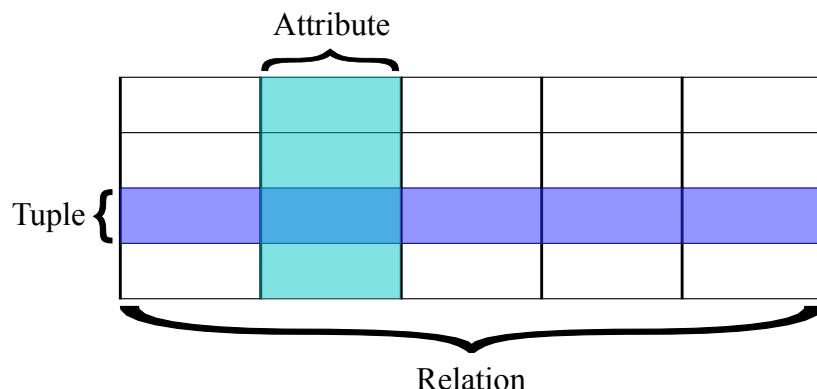


Figure 22.1: See https://en.wikipedia.org/wiki/Relational_database.

SQLite

The most common database management systems (DBMS) for relational databases are based on Structured Query Language, commonly called SQL (pronounced¹ “sequel”). Though SQL is a language in and of itself, most programming languages have tools for executing SQL routines. In Python, the most common variant of SQL is SQLite, implemented as the `sqlite3` module in the standard library.

A SQL database is stored in an external file, usually marked with the file extension `db` or `mdf`. These files should **not** be opened in Python with `open()` like text files; instead, any interactions with the database—creating, reading, updating, or deleting data—should occur as follows.

1. Create a connection to the database with `sqlite3.connect()`. This creates a database file if one does not already exist.
2. Get a cursor, an object that manages the actual traversal of the database, with the connection’s `cursor()` method.
3. Alter or read data with the cursor’s `execute()` method, which accepts an actual SQL command as a string.
4. Save any changes with the cursor’s `commit()` method, or revert changes with `rollback()`.
5. Close the connection.

```
>>> import sqlite3 as sql

# Establish a connection to a database file or create one if it doesn't exist.
>>> conn = sql.connect("my_database.db")
>>> try:
...     cur = conn.cursor()                      # Get a cursor object.
...     cur.execute("SELECT * FROM MyTable")      # Execute a SQL command.
... except sql.Error:
...     conn.rollback()                          # If there is an error,
...     raise                                # revert the changes
... else:                                    # and raise the error.
...     conn.commit()                           # If there are no errors,
... finally:                                 # save the changes.
...     conn.close()                            # Close the connection.
```

Achtung!

¹See <https://english.stackexchange.com/questions/7231/how-is-sql-pronounced> for a brief history of the somewhat controversial pronunciation of SQL.

Some changes, such as creating and deleting tables, are automatically committed to the database as part of the cursor's `execute()` method. Be **extremely cautious** when deleting tables, as the action is immediate and permanent. Most changes, however, do not take effect in the database file until the connection's `commit()` method is called. Be careful not to close the connection before committing desired changes, or those changes will not be recorded.

The `with` statement can be used with `open()` so that file streams are automatically closed, even in the event of an error. Likewise, combining the `with` statement with `sql.connect()` automatically rolls back changes if there is an error and commits them otherwise. However, the actual database connection is **not** closed automatically. With this strategy, the previous code block can be reduced to the following.

```
>>> try:
...     with sql.connect("my_database.db") as conn:
...         cur = conn.cursor()                      # Get the cursor.
...         cur.execute("SELECT * FROM MyTable")      # Execute a SQL command.
...     finally:
...         conn.close()                           # Commit or revert, then
                                                # close the connection.
```

Managing Database Tables

SQLite uses five native data types (relatively few compared to other SQL systems) that correspond neatly to native Python data types.

Python Type	SQLite Type
<code>None</code>	<code>NULL</code>
<code>int</code>	<code>INTEGER</code>
<code>float</code>	<code>REAL</code>
<code>str</code>	<code>TEXT</code>
<code>bytes</code>	<code>BLOB</code>

The `CREATE TABLE` command, together with a table name and a schema, adds a new table to a database. The schema is a comma-separated list where each entry specifies the column name, the column data type,² and other optional parameters. For example, the following code adds a table called `MyTable` with the schema (`Name`, `ID`, `Age`) with appropriate data types.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("CREATE TABLE MyTable (Name TEXT, ID INTEGER, Age REAL)")
...
>>> conn.close()
```

²Though SQLite does not force the data in a single column to be of the same type, most other SQL systems enforce uniform column types, so it is good practice to specify data types in the schema.

The `DROP TABLE` command deletes a table. However, using `CREATE TABLE` to try to create a table that already exists or using `DROP TABLE` to remove a nonexistent table raises an error. Use `DROP TABLE IF EXISTS` to remove a table without raising an error if the table doesn't exist. See Table 22.1 for more table management commands.

Operation	SQLite Command
Create a new table	<code>CREATE TABLE <table> (<schema>);</code>
Delete a table	<code>DROP TABLE <table>;</code>
Delete a table if it exists	<code>DROP TABLE IF EXISTS <table>;</code>
Add a new column to a table	<code>ALTER TABLE <table> ADD <column> <dtype></code>
Remove an existing column	<code>ALTER TABLE <table> DROP COLUMN <column>;</code>
Rename an existing column	<code>ALTER TABLE <table> ALTER COLUMN <column> <dtype>;</code>

Table 22.1: SQLite commands for managing tables and columns.

Note

SQL commands like `CREATE TABLE` are often written in all caps to distinguish them from other parts of the query, like the table name. This is only a matter of style: SQLite, along with most other versions of SQL, is case insensitive. In Python's SQLite interface, the trailing semicolon is also unnecessary. However, most other database systems require it, so it's good practice to include the semicolon in Python.

Problem 1. Write a function that accepts the name of a database file. Connect to the database (and create it if it doesn't exist). Drop the tables `MajorInfo`, `CourseInfo`, `StudentInfo`, and `StudentGrades` from the database if they exist. Next, add the following tables to the database with the specified column names and types.

- `MajorInfo`: `MajorID` (integers) and `MajorName` (strings).
- `CourseInfo`: `CourseID` (integers) and `CourseName` (strings).
- `StudentInfo`: `StudentID` (integers), `StudentName` (strings), and `MajorID` (integers).
- `StudentGrades`: `StudentID` (integers), `CourseID` (integers), and `Grade` (strings).

Remember to commit and close the database. You should be able to execute your function more than once with the same input without raising an error.

To check the database, use the following commands to get the column names of a specified table. Assume here that the database file is called `students.db`.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM StudentInfo;")
...     print([d[0] for d in cur.description])
... 
```

```
[ 'StudentID', 'StudentName', 'MajorID' ]
```

Inserting, Removing, and Altering Data

Tuples are added to SQLite database tables with the `INSERT INTO` command.

```
# Add the tuple (Samuel Clemens, 1910421, 74.4) to MyTable in my_database.db.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

With this syntax, SQLite assumes that values match sequentially with the schema of the table. The schema of the table can also be written explicitly for clarity.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable(Name, ID, Age) "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

Achtung!

Never use Python's string operations to construct a SQL query from variables. Doing so makes the program susceptible to a SQL injection attack.^a Instead, use parameter substitution to construct dynamic commands: use a ? character within the command, then provide the sequence of values as a second argument to `execute()`.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     values = ('Samuel Clemens', 1910421, 74.4)
...     # Don't piece the command together with string operations!
...     # cur.execute("INSERT INTO MyTable VALUES " + str(values)) # BAD!
...     # Instead, use parameter substitution.
...     cur.execute("INSERT INTO MyTable VALUES(?, ?, ?);", values) # Good.
```

^aSee <https://xkcd.com/327/> for an example.

To insert several rows at a time to the same table, use the cursor object's `executemany()` method and parameter substitution with a list of tuples. This is typically much faster than using `execute()` repeatedly.

```
# Insert (Samuel Clemens, 1910421, 74.4) and (Jane Doe, 123, 20) to MyTable.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
```

```
...     rows = [('John Smith', 456, 40.5), ('Jane Doe', 123, 20)]
...     cur.executemany("INSERT INTO MyTable VALUES(?, ?, ?);", rows)
```

Problem 2. Expand your function from Problem 1 so that it populates the tables with the data given in Tables 22.2a–22.2d.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Meredes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

(d) StudentGrades

Table 22.2: Student database.

The `StudentInfo` and `StudentGrades` tables are also recorded in `student_info.csv` and `student_grades.csv`, respectively, with `NULL` values represented as `-1` (we'll leave them as `-1` for now). A CSV (comma-separated values) file can be read like a normal text file or with the `csv` module.

```
>>> import csv
>>> with open("student_info.csv", 'r') as infile:
...     rows = list(csv.reader(infile))
```

To validate your database, use the following command to retrieve the rows from a table.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     for row in cur.execute("SELECT * FROM MajorInfo"):
...         print(row)
(1, 'Math')
(2, 'Science')
(3, 'Writing')
```

```
(4, 'Art')
```

Problem 3. The data file `us_earthquakes.csv`^a contains data from about 3,500 earthquakes in the United States since the 1769. Each row records the year, month, day, hour, minute, second, latitude, longitude, and magnitude of a single earthquake (in that order). Note that latitude, longitude, and magnitude are floats, while the remaining columns are integers.

Write a function that accepts the name of a database file. Drop the table `USEarthquakes` if it already exists, then create a new `USEarthquakes` table with schema (`Year`, `Month`, `Day`, `Hour`, `Minute`, `Second`, `Latitude`, `Longitude`, `Magnitude`). Populate the table with the data from `us_earthquakes.csv`. Remember to commit the changes and close the connection.
(Hint: using `executemany()` is much faster than using `execute()` in a loop.)

^aRetrieved from <https://datarepository.wolframcloud.com/resources/Sample-Data-US-Earthquakes>.

The WHERE Clause

Deleting or altering existing data in a database requires some searching for the desired row or rows. The `WHERE` clause is a predicate that filters the rows based on a boolean condition. The operators `==`, `!=`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, and `NOT` all work as expected to create search conditions.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     # Delete any rows where the Age column has a value less than 30.
...     cur.execute("DELETE FROM MyTable WHERE Age < 30;")
...     # Change the Name of "Samuel Clemens" to "Mark Twain".
...     cur.execute("UPDATE MyTable SET Name='Mark Twain' WHERE ID==1910421;")
```

If the `WHERE` clause were omitted from either of the previous commands, every record in `MyTable` would be affected. **Always** use a very specific `WHERE` clause when removing or updating data.

Operation	SQLite Command
Add a new row to a table	<code>INSERT INTO table VALUES(<values>);</code>
Remove rows from a table	<code>DELETE FROM <table> WHERE <condition>;</code>
Change values in existing rows	<code>UPDATE <table> SET <column1>=<value1>, ... WHERE <condition>;</code>

Table 22.3: SQLite commands for inserting, removing, and updating rows.

Note

SQLite treats `=` and `==` as equivalent operators. For clarity, in this lab we will always use `==` when comparing two values, such as in a `WHERE` clause. The only time we use `=` is in `SET` statements. Be aware that some flavors of SQL (such as MySQL) have no concept of `==`, and typing it in a query will return an error.

Problem 4. Modify your function from Problems 1 and 2 so that in the `StudentInfo` table, values of `-1` in the `MajorID` column are replaced with `NULL` values.

Also modify your function from Problem 3 in the following ways.

1. Remove rows from `USEarthquakes` that have a value of `0` for the `Magnitude`.
2. Replace `0` values in the `Day`, `Hour`, `Minute`, and `Second` columns with `NULL` values.

Reading and Analyzing Data

Constructing and managing databases is fundamental, but most time in SQL is spent analyzing existing data. A query is a SQL command that reads all or part of a database without actually modifying the data. Queries start with the `SELECT` command, followed by column and table names and additional (optional) conditions. The results of a query, called the result set, are accessed through the cursor object. After calling `execute()` with a SQL query, use `fetchall()` or another cursor method from Table 22.4 to get the list of matching tuples.

Method	Description
<code>execute()</code>	Execute a single SQL command
<code>executemany()</code>	Execute a single SQL command over different values
<code>executescript()</code>	Execute a SQL script (multiple SQL commands)
<code>fetchone()</code>	Return a single tuple from the result set
<code>fetchmany(n)</code>	Return the next n rows from the result set as a list of tuples
<code>fetchall()</code>	Return the entire result set as a list of tuples

Table 22.4: Methods of database cursor objects.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get tuples of the form (StudentID, StudentName) from the StudentInfo table.
>>> cur.execute("SELECT StudentID, StudentName FROM StudentInfo;")
>>> cur.fetchone()           # List the first match (a tuple).
(401767594, 'Michelle Fernandez')

>>> cur.fetchmany(3)        # List the next three matches (a list of tuples).
[(678665086, 'Gilbert Chapman'),
 (553725811, 'Roberta Cook'),
 (886308195, 'Rene Cross')]
```

```
>>> cur.fetchall()          # List the remaining matches.
[(103066521, 'Cameron Kim'),
 (821568627, 'Mercedes Hall'),
 (206208438, 'Kristopher Tran'),
 (341324754, 'Cassandra Holland'),
 (262019426, 'Alfonso Phelps'),
 (622665098, 'Sammy Burke')]

# Use * in place of column names to get all of the columns.
>>> cur.execute("SELECT * FROM MajorInfo;").fetchall()
[(1, 'Math'), (2, 'Science'), (3, 'Writing'), (4, 'Art')]

>>> conn.close()
```

The `WHERE` predicate can also refine a `SELECT` command. If the condition depends on a column in a different table from the data that is being selected, create a table alias with the `AS` command to specify columns in the form `table.column`.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the names of all math majors.
>>> cur.execute("SELECT SI.StudentName "
...             "FROM StudentInfo AS SI, MajorInfo AS MI "
...             "WHERE SI.MajorID == MI.MajorID AND MI.MajorName == 'Math'")
# The result set is a list of 1-tuples; extract the entry from each tuple.
>>> [t[0] for t in cur.fetchall()]
['Cassandra Holland', 'Michelle Fernandez']

# Get the names and grades of everyone in English class.
>>> cur.execute("SELECT SI.StudentName, SG.Grade "
...             "FROM StudentInfo AS SI, StudentGrades AS SG "
...             "WHERE SI.StudentID == SG.StudentID AND CourseID == 2;")
>>> cur.fetchall()
[('Roberta Cook', 'C'),
 ('Cameron Kim', 'C'),
 ('Mercedes Hall', 'A+'),
 ('Kristopher Tran', 'A'),
 ('Cassandra Holland', 'D-'),
 ('Alfonso Phelps', 'B'),
 ('Sammy Burke', 'A-')]

>>> conn.close()
```

Problem 5. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problems 1 and 2, query the database for all tuples of the form (StudentName, CourseName) where that student has an “A” or “A+” grade in that course. Return the list of tuples.

Aggregate Functions

A result set can be analyzed in Python using tools like NumPy, but SQL itself provides a few tools for computing a few very basic statistics: `AVG()`, `MIN()`, `MAX()`, `SUM()`, and `COUNT()` are aggregate functions that compress the columns of a result set into the desired quantity.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the number of students and the lowest ID number in StudentInfo.
>>> cur.execute("SELECT COUNT(StudentName), MIN(StudentID) FROM StudentInfo;")
>>> cur.fetchall()
[(10, 103066521)]
```

Problem 6. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problem 3, query the `USEarthquakes` table for the following information.

- The magnitudes of the earthquakes during the 19th century (1800–1899).
- The magnitudes of the earthquakes during the 20th century (1900–1999).
- The average magnitude of all earthquakes in the database.

Create a single figure with two subplots: a histogram of the magnitudes of the earthquakes in the 19th century, and a histogram of the magnitudes of the earthquakes in the 20th century. Show the figure, then return the average magnitude of all of the earthquakes in the database. Be sure to return an actual number, not a list or a tuple.

(Hint: use `np.ravel()` to convert a result set of 1-tuples to a 1-D array.)

Note

Problem 6 raises an interesting question: are the number of earthquakes in the United States increasing with time, and if so, how drastically? A closer look shows that only 3 earthquakes were recorded (in this data set) from 1700–1799, 208 from 1800–1899, and a whopping 3049 from 1900–1999. Is the increase in earthquakes due to there actually being more earthquakes, or to the improvement of earthquake detection technology? The best answer without conducting additional research is “probably both.” Be careful to question the nature of your data—how it was gathered, what it may be lacking, what biases or lurking variables might be present—before jumping to strong conclusions.

See the following for more info on the `sqlite3` and SQL in general.

- <https://docs.python.org/3/library/sqlite3.html>
- <https://www.w3schools.com/sql/>
- https://en.wikipedia.org/wiki/SQL_injection

Additional Material

Shortcuts for WHERE Conditions

Complicated **WHERE** conditions can be simplified with the following commands.

- **IN**: check for equality to one of several values quickly, similar to Python's `in` operator. In other words, the following SQL commands are equivalent.

```
SELECT * FROM StudentInfo WHERE MajorID == 1 OR MajorID == 2;  
SELECT * FROM StudentInfo WHERE MajorID IN (1,2);
```

- **BETWEEN**: check two (inclusive) inequalities quickly. The following are equivalent.

```
SELECT * FROM MyTable WHERE AGE >= 20 AND AGE <= 60;  
SELECT * FROM MyTable WHERE AGE BETWEEN 20 AND 60;
```

23 SQL 2 (The Sequel)

Lab Objective: Since SQL databases contain multiple tables, retrieving information about the data can be complicated. In this lab we discuss joins, grouping, and other advanced SQL query concepts to facilitate rapid data retrieval.

We will use the following database as an example throughout this lab, found in `students.db`.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B-
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C-
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D-
341324754	1	A-
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A-

(d) StudentGrades

Table 23.1: Student database

Joining Tables

A join combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
SELECT <alias.column, ...>
    FROM <table> AS <alias> JOIN <table> AS <alias>, ...
    ON <alias.column> == <alias.column>, ...
    WHERE <condition>;
```

The `ON` clause tells the query how to join tables together. Typically if there are N tables being joined together, there should be $N - 1$ conditions in the `ON` clause.

Inner Joins

An inner join creates a temporary table with the rows that have exact matches on the attribute(s) specified in the `ON` clause. Inner joins **intersect** two or more tables, as in Figure 23.1a.

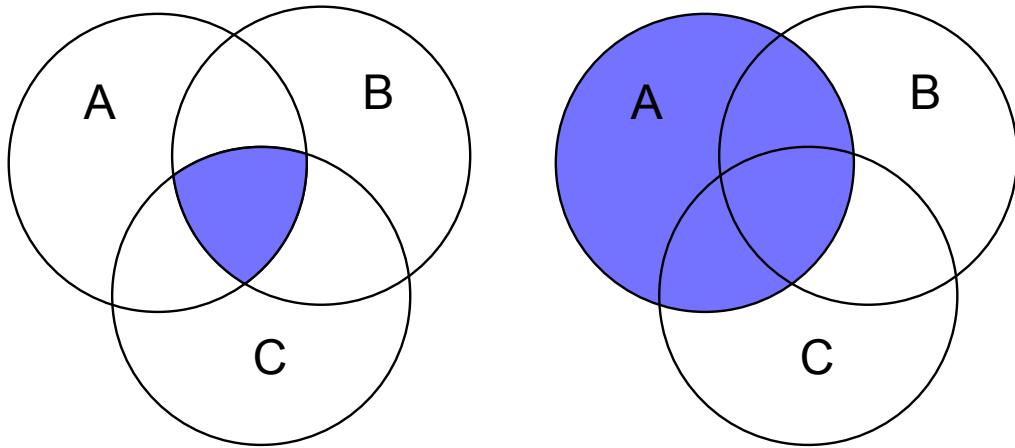


Figure 23.1

For example, Table 23.1c (`StudentInfo`) and Table 23.1a (`MajorInfo`) both have a `MajorID` column, so the tables can be joined by pairing rows that have the same `MajorID`. Such a join temporarily creates the following table.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
622665098	Sammy Burke	2	2	Science

Table 23.2: An inner join of `StudentInfo` and `MajorInfo` on `MajorID`.

Notice that this table is missing the rows where `MajorID` was `NULL` in the `StudentInfo` table. This is because there was no match for `NULL` in the `MajorID` column of the `MajorInfo` table, so the inner join throws those rows away.

Because joins deal with multiple tables at once, it is important to assign table aliases with the **AS** command. Join statements can also be supplemented with **WHERE** clauses like regular queries.

```
>>> import sqlite3 as sql
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

>>> cur.execute("SELECT * "
...                 "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...                 "ON SI.MajorID == MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]

# Select the names and ID numbers of the math majors.
>>> cur.execute("SELECT SI.StudentName, SI.StudentID "
...                 "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...                 "ON SI.MajorID == MI.MajorID "
...                 "WHERE MI.MajorName == 'Math';").fetchall()
[('Cassandra Holland', 341324754), ('Michelle Fernandez', 401767594)]
```

Problem 1. Write a function that accepts the name of a database file. Assuming the database to be in the format of Tables 23.1a–23.1d, query the database for the list of the names of students who have a B grade in any course (not a B- or a B+). Be sure to return a list of strings, not a list of tuples of strings.

Outer Joins

A left outer join, sometimes called a left join, creates a temporary table with **all** of the rows from the first (left-most) table, and all the “matched” rows on the given attribute(s) from the other relations. Rows from the left table that don’t match up with the columns from the other tables are supplemented with **NULL** values to fill extra columns. Compare the following table and code to Table 23.2.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
678665086	Gilbert Chapman	NULL	NULL	NULL
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
821568627	Mercedes Hall	NULL	NULL	NULL
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
262019426	Alfonso Phelps	NULL	NULL	NULL
622665098	Sammy Burke	2	2	Science

Table 23.3: A left outer join of `StudentInfo` and `MajorInfo` on `MajorID`.

```
>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...             "ON SI.MajorID == MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (678665086, 'Gilbert Chapman', None, None, None),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (821568627, 'Mercedes Hall', None, None, None),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (262019426, 'Alfonso Phelps', None, None, None),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]
```

Some flavors of SQL also support the `RIGHT OUTER JOIN` command, but `sqlite3` does not recognize the command since T1 `RIGHT OUTER JOIN` T2 is equivalent to T2 `LEFT OUTER JOIN` T1.

Joining Multiple Tables

Complicated queries often join several different relations. If the same kind of join is being used, the relations and conditional statements can be put in list form. For example, the following code selects courses that Kristopher Tran has taken, and the grades that he got in those courses, by joining three tables together. Note that 2 conditions are required in the `ON` clause in this case.

```
>>> cur.execute("SELECT CI.CourseName, SG.Grade "
...             "FROM StudentInfo AS SI" # Join 3 tables.
...             "INNER JOIN CourseInfo AS CI, StudentGrades SG "
...             "ON SI.StudentID==SG.StudentID AND CI.CourseID==SG.CourseID "
...             "WHERE SI.StudentName == 'Kristopher Tran;").fetchall()
[('Calculus', 'C+'), ('English', 'A')]
```

To use different kinds of joins in a single query, append one join statement after another. The join closest to the beginning of the statement is executed first, creating a temporary table, and the next join attempts to operate on that table. The following example performs an additional join on Table 23.3 to find the name and major of every student who got a C in a class.

```
# Do an inner join on the results of the left outer join.
>>> cur.execute("SELECT SI.StudentName, MI.MajorName "
...                 "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...                 "ON SI.MajorID == MI.MajorID "
...                 "INNER JOIN StudentGrades AS SG "
...                 "ON SI.StudentID == SG.StudentID "
...                 "WHERE SG.Grade == 'C';").fetchall()
[('Michelle Fernandez', 'Math'),
 ('Roberta Cook', 'Science'),
 ('Cameron Kim', 'Art'),
 ('Alfonso Phelps', None)]
```

In this last example, note carefully that Alfonso Phelps would have been excluded from the result set if an inner join was performed first instead of an outer join (since he lacks a major).

Problem 2. Write a function that accepts the name of a database file. Query the database for all tuples of the form `(Name, MajorName, Grade)` where `Name` is a student's name and `Grade` is their grade in Calculus. Only include results for students that are actually taking Calculus, but be careful not to exclude students who haven't declared a major.

Grouping Data

Many data sets can be naturally sorted into groups. The `GROUP BY` command gathers rows from a table and groups them by a certain attribute. The groups are then combined by one of the aggregate functions `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`. Each of these functions accepts the name of the column to be operated on. Note that the first four of these require the column to hold numerical data. Since our database has no such data, we will delay examples of using the functions other than `COUNT()` until later.

The following code groups the rows in Table 23.1d by `studentID` and counts the number of entries in each group.

```
>>> cur.execute("SELECT StudentID, COUNT(*) "    # * means "all of the rows".
...                 "FROM StudentGrades "
...                 "GROUP BY StudentID;").fetchall()
[(103066521, 3),
 (206208438, 2),
 (262019426, 2),
 (341324754, 2),
 (401767594, 2),
 (553725811, 1),
 (622665098, 2),
 (678665086, 3),
 (821568627, 3),
 (886308195, 1)]
```

`GROUP BY` can also be used in conjunction with joins. The join creates a temporary table like Tables 23.2 or 23.3, the results of which can then be grouped.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) "
...                 "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...                 "ON SG.StudentID == SI.StudentID "
...                 "GROUP BY SG.StudentID;").fetchall()
[('Cameron Kim', 3),
 ('Kristopher Tran', 2),
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Michelle Fernandez', 2),
 ('Roberta Cook', 1),
 ('Sammy Burke', 2),
 ('Gilbert Chapman', 3),
 ('Mercedes Hall', 3),
 ('Rene Cross', 1)]
```

Just like the `WHERE` clause chooses rows in a relation, the `HAVING` clause chooses groups from the result of a `GROUP BY` based on some criteria related to the groupings. For this particular command, it is often useful (but not always necessary) to create an alias for the columns of the result set with the `AS` operator. For instance, the result set of the previous example can be filtered down to only contain students who are taking 3 courses.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) as num_courses "    # Alias.
...                 "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...                 "ON SG.StudentID == SI.StudentID "
...                 "GROUP BY SG.StudentID "
...                 "HAVING num_courses == 3;").fetchall()    # Refer to alias later←
.
[('Cameron Kim', 3), ('Gilbert Chapman', 3), ('Mercedes Hall', 3)]

# Alternatively, get just the student names.
>>> cur.execute("SELECT SI.StudentName "                                # No alias.
...                 "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...                 "ON SG.StudentID == SI.StudentID "
...                 "GROUP BY SG.StudentID "
...                 "HAVING COUNT(*) == 3;").fetchall()
[('Cameron Kim',), ('Gilbert Chapman',), ('Mercedes Hall',)]
```

Other Miscellaneous Commands

Ordering Result Sets

The `ORDER BY` command sorts a result set by one or more attributes. Sorting can be done in ascending or descending order with `ASC` or `DESC`, respectively. This is always the very last statement in a query.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) AS num_courses "    # Alias.
```

```

...
    "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...
    "ON SG.StudentID == SI.StudentID "
...
    "GROUP BY SG.StudentID "
...
    "ORDER BY num_courses DESC, SI.StudentName ASC;").fetchall()
[('Cameron Kim', 3),                      # The results are now ordered by the
 ('Gilbert Chapman', 3),                   # number of courses each student is in,
 ('Mercedes Hall', 3),                    # then alphabetically by student name.
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Kristopher Tran', 2),
 ('Michelle Fernandez', 2),
 ('Sammy Burke', 2),
 ('Rene Cross', 1),
 ('Roberta Cook', 1)]

```

Problem 3. Write a function that accepts a database file. Query the given database for tuples of the form `(MajorName, N)` where `N` is the number of students in the specified major. Sort the results in descending order by the count `N`, and then in alphabetic order by `MajorName`. Include `Null` majors.

Searching Text with Wildcards

The `LIKE` operator within a `WHERE` clause matches patterns in a `TEXT` column. The special characters `%` and `_` and called wildcards that match any number of characters or a single character, respectively. For instance, `%Z_` matches any string of characters ending in a Z then another character, and `%i%` matches any string containing the letter i.

```

>>> results = cur.execute("SELECT StudentName FROM StudentInfo "
...                         "WHERE StudentName LIKE '%i%'").fetchall()
>>> [r[0] for r in results]
['Michelle Fernandez', 'Gilbert Chapman', 'Cameron Kim', 'Kristopher Tran']

```

Case Expressions

A case expression maps the values in a column using boolean logic. There are two forms of a case expression: simple and searched. A simple case expression matches and replaces specified attributes.

```

# Replace the values MajorID with new custom values.
>>> cur.execute("SELECT StudentName, CASE MajorID "
...                 "WHEN 1 THEN 'Mathematics' "
...                 "WHEN 2 THEN 'Soft Science' "
...                 "WHEN 3 THEN 'Writing and Editing' "
...                 "WHEN 4 THEN 'Fine Arts' "
...                 "ELSE 'Undeclared' END "
...                 "FROM StudentInfo "

```

```

...           "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Fine Arts'),
 ('Cassandra Holland', 'Mathematics'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Soft Science'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Mathematics'),
 ('Rene Cross', 'Writing and Editing'),
 ('Roberta Cook', 'Soft Science'),
 ('Sammy Burke', 'Soft Science')]

```

A searched case expression involves using a boolean expression at each step, instead of listing all of the possible values for an attribute.

```

# Change NULL values in MajorID to 'Undeclared' and non-NULL to 'Declared'.
>>> cur.execute("SELECT StudentName, CASE "
...                 "WHEN MajorID IS NULL THEN 'Undeclared' "
...                 "ELSE 'Declared' END "
...                 "FROM StudentInfo "
...                 "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Declared'),
 ('Cassandra Holland', 'Declared'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Declared'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Declared'),
 ('Rene Cross', 'Declared'),
 ('Roberta Cook', 'Declared'),
 ('Sammy Burke', 'Declared')]

```

Chaining Queries

The result set of any SQL query is really just another table with data from the original database. Separate queries can be made from result sets by enclosing the entire query in parentheses. For these sorts of operations, it is very important to carefully label the columns resulting from a subquery.

```

# Count how many declared and undeclared majors there are
# The subquery changes NULL values in MajorID to 'Undeclared' and
#   non-NULL to 'Declared'.
>>> cur.execute("SELECT majorstatus, COUNT(*) AS majorcount "
...                 "FROM ( "                                # Begin subquery.
...                   "SELECT StudentName, CASE "
...                     "WHEN MajorID IS NULL THEN 'Undeclared' "
...                     "ELSE 'Declared' END AS majorstatus "
...                   "FROM StudentInfo) "                  # End subquery.
...                 "GROUP BY majorstatus "

```

```

...
    "ORDER BY majorcount DESC;").fetchall()
[('Declared', 7), ('Undeclared', 3)]

```

Subqueries can also be joined with other tables, as in the following example. Note also that a subquery can be used to create numerical data out of non-numerical data, which can then be passed into any of the aggregate functions.

```

# Find the proportion of classes each student has an A+, A, or A- in.
# The inner query creates a column 'gradeisa' which is 1 if the student's grade
#     is A+, A, or A-, and 0 otherwise.
>>> cur.execute("SELECT SI.StudentName, AVG(SG.gradeisa) "
...             "FROM (
...                 SELECT StudentID, CASE Grade "
...                     "WHEN 'A+' THEN 1 "
...                     "WHEN 'A' THEN 1 "
...                     "WHEN 'A-' THEN 1 "
...                     "ELSE 0 END AS gradeisa "
...                 FROM StudentGrades) AS SG "
...             "INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID;").fetchall()
[('Cameron Kim', 0.3333333333333333),
 ('Kristopher Tran', 0.5),
 ('Alfonso Phelps', 0.0),
 ('Cassandra Holland', 0.5),
 ('Michelle Fernandez', 0.0),
 ('Roberta Cook', 0.0),
 ('Sammy Burke', 1.0),
 ('Gilbert Chapman', 0.6666666666666666),
 ('Mercedes Hall', 0.3333333333333333),
 ('Rene Cross', 1.0)]

```

Problem 4. Write a function that accepts the name of a database file. Query the database for tuples of the form (StudentName, N, GPA) where N is the number of courses that the specified student is enrolled in and GPA is their grade point average based on the following point system:

$$\begin{array}{llll}
 \text{A+}, \text{A} = 4.0 & \text{B} = 3.0 & \text{C} = 2.0 & \text{D} = 1.0 \\
 \text{A-} = 3.7 & \text{B-} = 2.7 & \text{C-} = 1.7 & \text{D-} = 0.7 \\
 \text{B+} = 3.4 & \text{C+} = 2.4 & \text{D+} = 1.4
 \end{array}$$

Order the results from greatest GPA to least.

Problem 5. The file `mystery_database.db` contains 4 tables called `table_1`, `table_2`, `table_3`, and `table_4` which contain information on over 5000 subjects. Hidden within these subjects is an obvious outlier. Use what you've learned about SQL to identify the outlier in this database. Return the outlier's name, ID number, eye color, and height as a list.

Hint: you may find that joining the tables is more difficult than it's worth; instead, try finding one clue at a time. Most of these subjects lived a long time ago in a galaxy far, far away... so a good place to start might be to find a subject who doesn't meet that criteria. Also, recall that the following commands can be used to get the column names of a specified table.

```
>>> with sql.connect("database.db") as conn:  
...     cur = conn.cursor()  
...     cur.execute("SELECT * FROM specified_table;")  
...     print([d[0] for d in cur.description])  
...  
['column_1', 'column_2', 'column_3']
```

24 Iterative Solvers

Lab Objective: Many real-world problems of the form $A\mathbf{x} = \mathbf{b}$ have tens of thousands of parameters. Solving such systems with Gaussian elimination or matrix factorizations could require trillions of floating point operations (FLOPs), which is of course infeasible. Solutions of large systems must therefore be approximated iteratively. In this lab we implement three popular iterative methods for solving large systems: Jacobi, Gauss-Seidel, and Successive Over-Relaxation.

Iterative methods are often useful to solve large systems of equations. In this lab, let $\mathbf{x}^{(k)}$ denote the k th iteration of the iterative method for solving the problem $A\mathbf{x} = \mathbf{b}$ for \mathbf{x} . Furthermore, let x_i be the i th component of \mathbf{x} so that $x_i^{(k)}$ is the i th component of \mathbf{x} in the k th iteration. Like other iterative methods, there are two stopping parameters: a very small $\varepsilon > 0$ and an integer $N \in \mathbb{N}$. Iterations continue until either

$$\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\| < \varepsilon \quad \text{or} \quad k > N. \quad (24.1)$$

The Jacobi Method

The Jacobi Method is a simple but powerful method used for solving certain kinds of large linear systems. The main idea is simple: solve for each variable in terms of the others, then use the previous values to update each approximation. As a (very small) example, consider the 3×3

$$\begin{aligned} 2x_1 - x_3 &= 3, \\ -x_1 + 3x_2 + 2x_3 &= 3, \\ +x_2 + 3x_3 &= -1. \end{aligned}$$

Solving the first equation for x_1 , the second for x_2 , and the third for x_3 yields

$$\begin{aligned} x_1 &= \frac{1}{2}(3 + x_3), \\ x_2 &= \frac{1}{3}(3 + x_1 - 2x_3), \\ x_3 &= \frac{1}{3}(-1 - x_2). \end{aligned}$$

Now begin with an initial guess $\mathbf{x}^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}]^T = [0, 0, 0]^T$. To compute the first approximation $\mathbf{x}^{(1)}$, use the entries of $\mathbf{x}^{(0)}$ as the variables on the right side of the previous equations:

$$\begin{aligned} x_1^{(1)} &= \frac{1}{2}(3 + x_3^{(0)}) = \frac{1}{2}(3 + 0) = \frac{3}{2}, \\ x_2^{(1)} &= \frac{1}{3}(3 + x_1^{(0)} - 2x_3^{(0)}) = \frac{1}{3}(3 + 0 - 0) = 1, \\ x_3^{(1)} &= \frac{1}{3}(-1 - x_2^{(0)}) = \frac{1}{3}(-1 - 0) = -\frac{1}{3}. \end{aligned}$$

Thus $\mathbf{x}^{(1)} = [\frac{3}{2}, 1, -\frac{1}{3}]^T$. Computing $\mathbf{x}^{(2)}$ is similar:

$$\begin{aligned} x_1^{(2)} &= \frac{1}{2}(3 + x_3^{(1)}) = \frac{1}{2}(3 - \frac{1}{3}) = \frac{4}{3}, \\ x_2^{(2)} &= \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(1)}) = \frac{1}{3}(3 + \frac{3}{2} + \frac{2}{3}) = \frac{31}{18}, \\ x_3^{(2)} &= \frac{1}{3}(-1 - x_2^{(1)}) = \frac{1}{3}(-1 - 1) = -\frac{2}{3}. \end{aligned}$$

The process is repeated until at least one of the two stopping criteria in (24.1) is met. For this particular problem, convergence to 8 decimal places ($\varepsilon = 10^{-8}$) is reached in 29 iterations.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$\mathbf{x}^{(0)}$	0	0	0
$\mathbf{x}^{(1)}$	1.5	1	-0.33333333
$\mathbf{x}^{(2)}$	1.33333333	1.72222222	-0.66666667
$\mathbf{x}^{(3)}$	1.16666667	1.88888889	-0.90740741
$\mathbf{x}^{(4)}$	1.04629630	1.99382716	-0.96296296
\vdots	\vdots	\vdots	\vdots
$\mathbf{x}^{(28)}$	0.99999999	2.00000001	-0.99999999
$\mathbf{x}^{(29)}$	1	2	-1

Matrix Representation

The iterative steps performed above can be expressed in matrix form. First, decompose A into its diagonal entries, its entries below the diagonal, and its entries above the diagonal, as $A = D + L + U$.

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \dots & a_{n,n-1} & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

$D \qquad \qquad \qquad L \qquad \qquad \qquad U$

With this decomposition, \mathbf{x} can be expressed in the following way.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D + L + U)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= -(L + U)\mathbf{x} + \mathbf{b} \\ \mathbf{x} &= D^{-1}(-(L + U)\mathbf{x} + \mathbf{b}) \end{aligned}$$

Now using $\mathbf{x}^{(k)}$ as the variables on the right side of the equation to produce $\mathbf{x}^{(k+1)}$ on the left, and noting that $L + U = A - D$, we have the following.

$$\begin{aligned} \mathbf{x}^{(k+1)} &= D^{-1}(-(A - D)\mathbf{x}^{(k)} + \mathbf{b}) \\ &= D^{-1}(D\mathbf{x}^{(k)} - A\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)}) \end{aligned} \tag{24.2}$$

There is a potential problem with (24.2): calculating a matrix inverse is the cardinal sin of numerical linear algebra, yet the equation contains D^{-1} . However, since D is a diagonal matrix, D^{-1} is also diagonal, and is easy to compute.

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{bmatrix}$$

Because of this, the Jacobi method requires that A have nonzero diagonal entries.

The diagonal D can be represented by the 1-dimensional array \mathbf{d} of the diagonal entries. Then the matrix multiplication $D\mathbf{x}$ is equivalent to the component-wise vector multiplication $\mathbf{d} * \mathbf{x} = \mathbf{x} * \mathbf{d}$. Likewise, the matrix multiplication $D^{-1}\mathbf{x}$ is equivalent to the component-wise “vector division” \mathbf{x}/\mathbf{d} .

Problem 1. Write a function that accepts a matrix A , a vector \mathbf{b} , a convergence tolerance tol defaulting to 10^{-8} , and a maximum number of iterations maxiter defaulting to 100. Implement the Jacobi method using (24.2), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Run the iteration until $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty < \text{tol}$, and only iterate at most maxiter times. Avoid using `la.inv()` to calculate D^{-1} , but use `la.norm()` to calculate the vector ∞ -norm.

Your function should be robust enough to accept systems of any size. To test your function, generate a random \mathbf{b} with `np.random.random()` and use the following function to generate an $n \times n$ matrix A for which the Jacobi method is guaranteed to converge. Run the iteration, then check that $A\mathbf{x}^{(k)}$ and \mathbf{b} are close using `np.allclose()`.

```
def diag_dom(n, num_entries=None, as_sparse=False):
    """Generate a strictly diagonally dominant (n, n) matrix.

    Parameters:
        n (int): The dimension of the system.
        num_entries (int): The number of nonzero values.
            Defaults to n^(3/2)-n.
        as_sparse: If True, an equivalent sparse CSR matrix is returned.

    Returns:
        A ((n,n) ndarray): A (n, n) strictly diagonally dominant matrix.
    """
    if num_entries is None:
        num_entries = int(n**1.5) - n
    A = sparse.dok_matrix((n,n))
    rows = np.random.choice(n, size=num_entries)
    cols = np.random.choice(n, size=num_entries)
    data = np.random.randint(-4, 4, size=num_entries)
    for i in range(num_entries):
        A[rows[i], cols[i]] = data[i]
    B = A.tocsr()           # convert to row format for the next step
    for i in range(n):
        A[i,i] = abs(B[i]).sum() + 1
    return A.tocsr() if as_sparse else A.toarray()
```

Also test your function on random $n \times n$ matrices. If the iteration is non-convergent, the successive approximations will have increasingly large entries.

Convergence

Most iterative methods only converge under certain conditions. For the Jacobi method, convergence mostly depends on the nature of the matrix A . If the entries a_{ij} of A satisfy the property

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all } i = 1, 2, \dots, n,$$

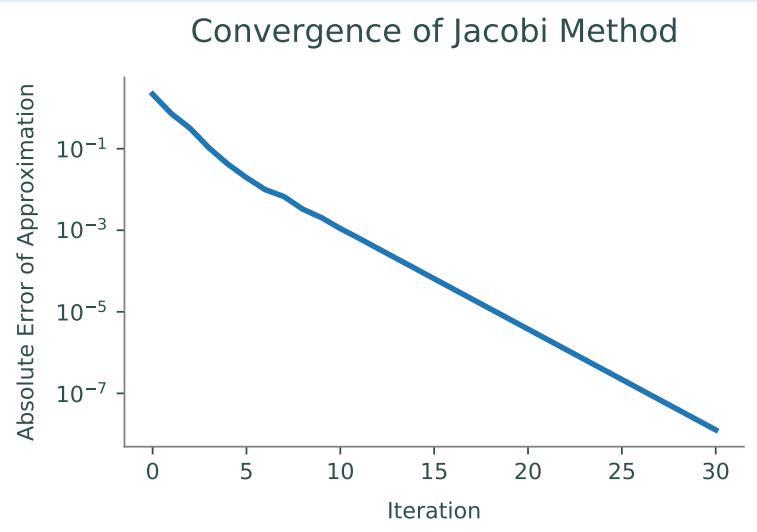
then A is called strictly diagonally dominant (`diag_dom()` in Problem 1 generates a strictly diagonally dominant $n \times n$ matrix). If this is the case,¹ then the Jacobi method always converges, regardless of the initial guess \mathbf{x}_0 . This is a very different convergence result than many other iterative methods such as Newton's method where convergence is highly sensitive to the initial guess.

There are a few ways to determine whether or not an iterative method is converging. For example, since the approximation $\mathbf{x}^{(k)}$ should satisfy $A\mathbf{x}^{(k)} \approx \mathbf{b}$, the normed difference $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ should be small. This value is called the absolute error of the approximation. If the iterative method converges, the absolute error should decrease to ε .

Problem 2. Modify your Jacobi method function in the following ways.

1. Add a keyword argument called `plot`, defaulting to `False`.
2. Keep track of the absolute error $\|A\mathbf{x}^{(k)} - \mathbf{b}\|_\infty$ of the approximation at each iteration.
3. If `plot` is `True`, produce a lin-log plot (use `plt.semilogy()`) of the error against iteration count. Remember to still return the approximate solution \mathbf{x} .

If the iteration converges, your plot should resemble the following figure.



¹Although this seems like a strong requirement, most real-world linear systems can be represented by strictly diagonally dominant matrices.

The Gauss-Seidel Method

The Gauss-Seidel method is essentially a slight modification of the Jacobi method. The main difference is that in Gauss-Seidel, new information is used immediately. As an example, consider again the system from the previous section,

$$\begin{array}{rcl} 2x_1 & - & x_3 = 3, \\ -x_1 + 3x_2 + 2x_3 = 3, \\ + x_2 + 3x_3 = -1. \end{array}$$

As with the Jacobi method, solve for x_1 in the first equation, x_2 in the second equation, and x_3 in the third equation:

$$\begin{aligned} x_1 &= \frac{1}{2}(3 + x_3), \\ x_2 &= \frac{1}{3}(3 + x_1 - 2x_3), \\ x_3 &= \frac{1}{3}(-1 - x_2). \end{aligned}$$

Using $\mathbf{x}^{(0)}$ to compute $x_1^{(1)}$ in the first equation as before,

$$x_1^{(1)} = \frac{1}{2}(3 + x_3^{(0)}) = \frac{1}{2}(3 + 0) = \frac{3}{2}.$$

Now, however, use the updated value of $x_1^{(1)}$ in the calculation of $x_2^{(1)}$:

$$x_2^{(1)} = \frac{1}{3}(3 + x_1^{(1)} - 2x_3^{(0)}) = \frac{1}{3}\left(3 + \frac{3}{2} - 0\right) = \frac{3}{2}.$$

Likewise, use the updated values of $x_1^{(1)}$ and $x_2^{(1)}$ to calculate $x_3^{(1)}$:

$$x_3^{(1)} = \frac{1}{3}(-1 - x_2^{(1)}) = \frac{1}{3}\left(-1 - \frac{3}{2}\right) = -\frac{5}{6}.$$

This process of using calculated information immediately is called forward substitution, and causes the algorithm to (generally) converge much faster.

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
$x^{(0)}$	0	0	0
$x^{(1)}$	1.5	1.5	-0.83333333
$x^{(2)}$	1.08333333	1.91666667	-0.97222222
$x^{(3)}$	1.01388889	1.98611111	-0.99537037
$x^{(4)}$	1.00231481	1.99768519	-0.99922840
\vdots	\vdots	\vdots	\vdots
$x^{(11)}$	1.00000001	1.99999999	-1
$x^{(12)}$	1	2	-1

Notice that Gauss-Seidel converges in less than half as many iterations as Jacobi does for this system.

Implementation

Because Gauss-Seidel updates only one element of the solution vector at a time, the iteration cannot be summarized by a single matrix equation. Instead, the process is most generally described by the equation

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k)} - \sum_{j > i} a_{ij} x_j^{(k)} \right). \quad (24.3)$$

Let \mathbf{a}_i be the i th **row** of A . The two sums closely resemble the regular vector product of \mathbf{a}_i and $\mathbf{x}^{(k)}$ without the i^{th} term $a_{ii}x_i^{(k)}$. This suggests the simplification

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}} \left(b_i - \mathbf{a}_i^\top \mathbf{x}^{(k)} + a_{ii}x_i^{(k)} \right) \\ &= x_i^{(k)} + \frac{1}{a_{ii}} \left(b_i - \mathbf{a}_i^\top \mathbf{x}^{(k)} \right). \end{aligned} \quad (24.4)$$

One sweep through all the entries of \mathbf{x} completes one iteration.

Problem 3. Write a function that accepts a matrix A , a vector \mathbf{b} , a convergence tolerance `tol` defaulting to 10^{-8} , a maximum number of iterations `maxiter` defaulting to 100, and a keyword argument `plot` that defaults to `False`. Implement the Gauss-Seidel method using (24.4), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$.

Use the same stopping criterion as in Problem 1. Also keep track of the absolute errors of the iteration, as in Problem 2. If `plot` is `True`, plot the error against iteration count. Use `diag_dom()` to generate test cases.

Achtung!

Since the Gauss-Seidel algorithm operates on the approximation vector in place (modifying it one entry at a time), the previous approximation $\mathbf{x}^{(k-1)}$ must be stored at the beginning of the k th iteration in order to calculate $\|\mathbf{x}^{(k-1)} - \mathbf{x}^{(k)}\|_\infty$. Additionally, since NumPy arrays are mutable, the past iteration must be stored as a `copy`.

```
>>> x0 = np.random.random(5)          # Generate a random vector.
>>> x1 = x0                        # Attempt to make a copy.
>>> x1[3] = 1000                   # Modify the "copy" in place.
>>> np.allclose(x0, x1)            # But x0 was also changed!
True

# Instead, make a copy of x0 when creating x1.
>>> x0 = np.copy(x1)               # Make a copy.
>>> x1[3] = -1000
>>> np.allclose(x0, x1)
False
```

Convergence

Whether or not the Gauss-Seidel method converges depends on the nature of A . If all of the eigenvalues of A are positive, A is called positive definite. If A is positive definite or if it is strictly diagonally dominant, then the Gauss-Seidel method converges regardless of the initial guess $\mathbf{x}^{(0)}$.

Solving Sparse Systems Iteratively

Iterative solvers are best suited for solving very large sparse systems. However, using the Gauss-Seidel method on sparse matrices requires translating code from NumPy to `scipy.sparse`. The algorithm is the same, but there are some functions that are named differently between these two packages.²

Problem 4. Write a new function that accepts a `sparse` matrix A , a vector \mathbf{b} , a convergence tolerance `tol`, and a maximum number of iterations `maxiter` (plotting the convergence is not required for this problem). Implement the Gauss-Seidel method using (24.4), returning the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual default stopping criterion.

The Gauss-Seidel method requires extracting the rows A_i from the matrix A and computing $A_i^T \mathbf{x}$. There are many ways to do this that cause some fairly serious runtime issues, so we provide the code for this specific portion of the algorithm.

```
# Get the indices of where the i-th row of A starts and ends if the
# nonzero entries of A were flattened.
rowstart = A.indptr[i]
rowend = A.indptr[i+1]

# Multiply only the nonzero elements of the i-th row of A with the
# corresponding elements of x.
Aix = A.data[rowstart:rowend] @ x[A.indices[rowstart:rowend]]
```

To test your function, remember to call `diag_dom()` using `as_sparse=True`

```
>>> A = diag_dom(50000, as_sparse=True)
>>> b = np.random.random(50000)
```

Successive Over-Relaxation

There are many systems that meet the requirements for convergence with the Gauss-Seidel method, but for which convergence is still relatively slow. A slightly altered version of the Gauss-Seidel method, called Successive Over-Relaxation (SOR), can result in faster convergence. This is achieved by introducing a relaxation factor $\omega \geq 1$ and modifying (24.3) as

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij}x_j^{(k)} - \sum_{j > i} a_{ij}x_j^{(k)} \right).$$

Simplifying the equation, we have

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \mathbf{a}_i^T \mathbf{x}^{(k)} \right). \quad (24.5)$$

Note that when $\omega = 1$, SOR reduces to Gauss-Seidel. The relaxation factor ω weights the new iteration between the current best approximation and the next approximation in a way that can sometimes dramatically improve convergence.

²See the lab on Linear Systems for a review of `scipy.sparse` matrices and syntax.

Problem 5. Write a function that accepts a sparse matrix A , a vector \mathbf{b} , a relaxation factor ω , a convergence tolerance tol , and a maximum number of iterations maxiter . Implement SOR using (24.5), compute the approximate solution to the equation $A\mathbf{x} = \mathbf{b}$. Use the usual stopping criterion. Return the approximate solution \mathbf{x} as well as a boolean indicating whether the function converged and the number of iterations computed.

(Hint: this requires changing only one line of code from the sparse Gauss-Seidel function.)

A Finite Difference Method

Laplace's equation is an important partial differential equation that arises often in both pure and applied mathematics. In two dimensions, the equation has the following form.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad (24.6)$$

Laplace's equation can be used to model heat flow. Consider a square metal plate where the top and bottom borders are fixed at 0° Celsius and the left and right sides are fixed at 100° Celsius. Given these boundary conditions, we want to describe how heat diffuses through the rest of the plate. The solution to Laplace's equation describes the plate when it is in a steady state, meaning that the heat at a given part of the plate no longer changes with time.

It is possible to solve (24.6) analytically. However, the problem can also be solved numerically using a finite difference method. To begin, we impose a discrete, square grid on the plate with uniform spacing. Denote the points on the grid by (x_i, y_j) and the value of u at these points (the heat) as $u(x_i, y_j) = U_{i,j}$. Using the centered difference quotient for second derivatives to approximate the partial derivatives,

$$\begin{aligned} 0 &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ &\approx \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2} + \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h^2} \\ &= \frac{1}{h^2} (-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}), \end{aligned} \quad (24.7)$$

where $h = x_{i+1} - x_i = y_{j+1} - y_j$ is the distance between the grid points in either direction. This problem can be formulated as a linear system. Suppose the grid has exactly $(n+2) \times (n+2)$ entries. Then the interior of the grid (where $u(x, y)$ is unknown) is $n \times n$, and can be flattened into an $n^2 \times 1$ vector \mathbf{u} . The entire first row goes first, then the second row, proceeding to the n th row.

$$\mathbf{u} = [U_{1,1} \quad U_{1,2} \quad \cdots \quad U_{1,n} \quad U_{2,1} \quad U_{2,2} \quad \cdots \quad U_{2,n} \quad \cdots \quad U_{n,n}]^\top$$

From (24.7), for an interior point $U_{i,j}$, we have

$$-4U_{i,j} + U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} = 0. \quad (24.8)$$

If any of the neighbors to $U_{i,j}$ is a boundary point on the grid, its value is already determined by the boundary conditions. For example, the neighbor $U_{3,0}$ of the gridpoint for $U_{3,1}$ is fixed at $U_{3,0} = 100$. In this case, (24.8) becomes

$$-4U_{3,1} + U_{2,1} + U_{3,2} + U_{4,1} = -100.$$

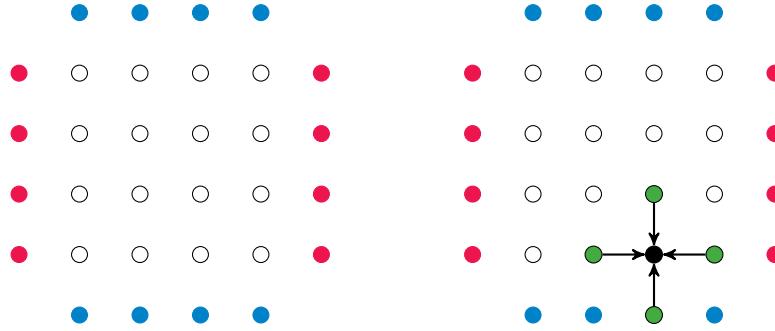


Figure 24.1: On the left, an example of a 6×6 grid ($n = 4$) where the red dots are hot boundary zones and the blue dots are cold boundary zones. On the right, the green dots are the neighbors of the interior black dot that are used to approximate the heat at the black dot.

The constants on the right side of (24.8) become the $n^2 \times 1$ vector \mathbf{b} . All nonzero entries of \mathbf{b} correspond to interior points that touch the left or right boundaries.

As an example, writing (24.8) for the 16 interior points of the grid in Figure 24.1 results in the following 16×16 system $A\mathbf{u} = \mathbf{b}$. Note the block structure (empty blocks are all zeros).

$$\left[\begin{array}{cccc|cccc|c|c|c|c|c|c|c|c} -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & & & & & \\ 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & & \\ 0 & 1 & -4 & 0 & 0 & 0 & 1 & 0 & & & & & & & & \\ 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & & & & & \\ \hline 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & \\ \hline & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\ \hline & & & & & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\ & & & & & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\ & & & & & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\ & & & & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 \end{array} \right] = \left[\begin{array}{c} U_{1,1} \\ U_{1,2} \\ U_{1,3} \\ U_{1,4} \\ \hline U_{2,1} \\ U_{2,2} \\ U_{2,3} \\ U_{2,4} \\ \hline U_{3,1} \\ U_{3,2} \\ U_{3,3} \\ U_{3,4} \\ \hline U_{4,1} \\ U_{4,2} \\ U_{4,3} \\ U_{4,4} \end{array} \right] = \left[\begin{array}{c} -100 \\ 0 \\ 0 \\ -100 \\ \hline -100 \\ 0 \\ 0 \\ -100 \\ \hline -100 \\ 0 \\ 0 \\ -100 \\ \hline -100 \\ 0 \\ 0 \\ -100 \\ \hline -100 \end{array} \right]$$

More concisely, for any positive integer n , the matrix A can be written as

$$A = \left[\begin{array}{ccccc} B & I & & & \\ I & B & I & & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{array} \right], \quad \text{where } B = \left[\begin{array}{ccccc} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{array} \right] \text{ is } n \times n.$$

Problem 6. Write a function that accepts an integer n , a relaxation factor ω , a convergence tolerance `tol` that defaults to 10^{-8} , a maximum number of iterations `maxiter` that defaults to 100, and a bool `plot` that defaults to `False`. Generate and solve the corresponding system $\mathbf{A}\mathbf{u} = \mathbf{b}$ using Problem 5. Also return a boolean indicating whether the function converged and the number of iterations computed.

(Hint: see Problem 5 of the Linear Systems lab for the construction of A . Also, `np.tile()` may be useful for constructing \mathbf{b} .)

If `plot=True`, visualize the solution \mathbf{u} with a heatmap using `plt.pcolormesh()` (the colormap "`coolwarm`" is a good choice in this case). This shows the distribution of heat over the hot plate after it has reached its steady state. Note that the \mathbf{u} must be reshaped as an $n \times n$ array to properly visualize the result.

Problem 7. To demonstrate how convergence is affected by the value of the relaxation factor ω in SOR, run your function from Problem 6 with $\omega = 1, 1.05, 1.1, \dots, 1.9, 1.95$ and $n = 20$. Plot the number of computed iterations as a function of ω . Return the value of ω that results in the least number of iterations.

Note that the matrix A from Problem 6 is not strictly diagonally dominant. However, A is positive definite, so the algorithm will converge. Unfortunately, convergence for these kinds of systems usually requires more iterations than for strictly diagonally dominant systems. Therefore, set `tol=1e-2` and `maxiter=1000`.

Recall that $\omega = 1$ corresponds to the Gauss-Seidel method. Choosing a more optimal relaxation factor saves a large number of iterations. This could translate to saving days or weeks of computation time while solving extremely large linear systems on a supercomputer.

Part II
Appendices

A

Getting Started

The labs in this curriculum aim to introduce computational and mathematical concepts, walk through implementations of those concepts in Python, and use industrial-grade code to solve interesting, relevant problems. Lab assignments are usually about 5–10 pages long and include code examples (yellow boxes), important notes (green boxes), warnings about common errors (red boxes), and about 3–7 exercises (blue boxes). Get started by downloading the lab manual(s) for your course from <http://foundations-of-applied-mathematics.github.io/>.

Submitting Assignments

Labs

Every lab has a corresponding specifications file with some code to get you started and to make your submission compatible with automated test drivers. Like the lab manuals, these materials are hosted at <http://foundations-of-applied-mathematics.github.io/>.

Download the `.zip` file for your course, unzip the folder, and move it somewhere where it won't get lost. This folder has some setup scripts and a collection of folders, one per lab, each of which contains the specifications file(s) for that lab. See [Student-Materials/wiki/Lab-Index](#) for the complete list of labs, their specifications and data files, and the manual that each lab belongs to.

Achtung!

Do **not** move or rename the lab folders or the enclosed specifications files; if you do, the test drivers will not be able to find your assignment. Make sure your folder and file names match [Student-Materials/wiki/Lab-Index](#).

To submit a lab, modify the provided specifications file and use the file-sharing program specified by your instructor (discussed in the next section). The instructor will drop feedback files in the lab folder after grading the assignment. For example, the Introduction to Python lab has the specifications file `PythonIntro/python_intro.py`. To complete that assignment, modify `PythonIntro/python_intro.py` and submit it via your instructor's file-sharing system. After grading, the instructor will create a file called `PythonIntro/PythonIntro_feedback.txt` with your score and some feedback.

Homework

Non-lab coding homework should be placed in the `_Homework/` folder and submitted like a lab assignment. Be careful to name your assignment correctly so the instructor (and test driver) can find it. The instructor may drop specifications files and/or feedback files in this folder as well.

Setup

Achtung!

We strongly recommend using a Unix-based operating system (Mac or Linux) for the labs. Unix has a true bash terminal, works well with git and python, and is the preferred platform for computational and data scientists. It is possible to do this curriculum with Windows, but expect some road bumps along the way.

There are two ways to submit code to the instructor: with git (<http://git-scm.com/>), or with a file-syncing service like Google Drive. Your instructor will indicate which system to use.

Setup With Git

Git is a program that manages updates between an online code repository and the copies of the repository, called clones, stored locally on computers. If git is not already installed on your computer, download it at <http://git-scm.com/downloads>. If you have never used git, you might want to read a few of the following resources.

- Official git tutorial: <https://git-scm.com/docs/gittutorial>
- Bitbucket git tutorials: <https://www.atlassian.com/git/tutorials>
- GitHub git cheat sheet: services.github.com/.../github-git-cheat-sheet.pdf
- GitLab git tutorial: <https://docs.gitlab.com/ce/gitlab-basics/start-using-git.html>
- Codecademy git lesson: <https://www.codecademy.com/learn/learn-git>
- Training video series by GitHub: <https://www.youtube.com/playlist?list=PLg7.../>

There are many websites for hosting online git repositories. Your instructor will indicate which web service to use, but we only include instructions here for setup with Bitbucket.

1. Sign up. Create a Bitbucket account at <https://bitbucket.org>. If you use an academic email address (ending in `.edu`, etc.), you will get free unlimited public and private repositories.
2. Make a new repository. On the Bitbucket page, click the `+` button from the menu on the left and, under **CREATE**, select **Repository**. Provide a name for the repository, mark the repository as **private**, and make sure the repository type is **Git**. For **Include a README?**, select **No** (if you accidentally include a README, delete the repository and start over). Under **Advanced settings**, enter a short description for your repository, select **No forks** under forking, and select **Python** as the language. Finally, click the blue **Create repository** button. Take note of the URL of the webpage that is created; it should be something like <https://bitbucket.org/<name>/<repo>>.

3. Give the instructor access to your repository. On your newly created Bitbucket repository page (<https://bitbucket.org/<name>/<repo>> or similar), go to **Settings** in the menu to the left and select **User and group access**, the second option from the top. Enter your instructor's Bitbucket username under **Users** and click **Add**. Select the blue **Write** button so your instructor can read from and write feedback to your repository.
4. Create an SSH key. This step needs to be done only once on each computer that you want to be able to use to access your repository. If you have multiple repositories on the same computer, you do not need to repeat this step for each one. To create an SSH key, in a shell application (Terminal on Linux or Mac, or Git Bash (<https://gitforwindows.org/>) on Windows), enter the following command:

```
$ ssh-keygen
```

Press the Enter or Return key to accept the default file location. It will then prompt to enter a passphrase; this acts as a password to use the SSH key. If you do not want a passphrase, leave it blank and press Enter again. The key will then be created. The file for the key will be placed in in the `/home/<username>/ .ssh` directory on Linux; in `/Users/<username>/ .ssh` on macOS; and in `/c/users/<username>/ .ssh` on Windows.

Now that the key is created, you need to add it to your Bitbucket account. From Bitbucket, choose **Personal settings** and then **SSH keys**. Click **Add key** and enter a label (what it is doesn't matter). Now, using the file explorer, navigate to the SSH key you created, and open the public key file. The file will be called something like `id_rsa.pub`; do NOT use `id_rsa` (without the `.pub` extension). Copy the contents of this file, paste it into the Key field on Bitbucket, and press Save.

For more options and some troubleshooting information, refer to <https://support.atlassian.com/bitbucket-cloud/docs/set-up-an-ssh-key/>.

5. Connect your folder to the new repository. In a shell application (Terminal on Linux or Mac, or Git Bash (<https://gitforwindows.org/>) on Windows), enter the following commands.

```
# Navigate to your folder.
$ cd /path/to/folder # cd means 'change directory'.


# Make sure you are in the right place.
$ pwd # pwd means 'print working directory'.
/path/to/folder
$ ls *.md # ls means 'list files'.
README.md # This means README.md is in the working directory.


# Connect this folder to the online repository.
$ git init
$ git remote add origin git@bitbucket.org:<name>/<repo>.git


# Record your credentials.
$ git config --local user.name "your name"
$ git config --local user.email "your email"


# Add the contents of this folder to git and update the repository.
```

```
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

For example, if your Bitbucket username is `greek314`, the repository is called `acmev1`, and the folder is called `Student-Materials/` and is on the desktop, enter the following commands.

```
# Navigate to the folder.
$ cd ~/Desktop/Student-Materials

# Make sure this is the right place.
$ pwd
/Users/Archimedes/Desktop/Student-Materials
$ ls *.md
README.md

# Connect this folder to the online repository.
$ git init
$ git remote add origin git@bitbucket.org:greek314/acmev1.git

# Record credentials.
$ git config --local user.name "archimedes"
$ git config --local user.email "greek314@example.com"

# Add the contents of this folder to git and update the repository.
$ git add --all
$ git commit -m "initial commit"
$ git push origin master
```

At this point you should be able to see the files on your repository page from a web browser. If you enter the repository URL incorrectly in the `git remote add origin` step, you can reset it with the following line:

```
$ git remote set-url origin git@bitbucket.org:<name>/<repo>.git
```

Note

You may get the an error like the following when you run `git push`:

```
remote: Bitbucket Cloud recently stopped supporting account passwords←
      for Git authentication.
...
fatal: Authentication failed for 'https://bitbucket.org/<name>/<repo←
>.git/'
```

If this error occurs, your repository URL is in the wrong format; most likely, you used the `https` version instead of what is shown above. You can use the `git remote set-url origin` command to fix this issue as well.

6. Download data files. Many labs have accompanying data files. To download these files, navigate to your clone and run the `download_data.sh` bash script, which downloads the files and places them in the correct lab folder for you. You can also find individual data files through [Student-Materials/wiki/Lab-Index](#).

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash download_data.sh
```

7. Install Python package dependencies. The labs require several third-party Python packages that don't come bundled with Anaconda. Run the following command to install the necessary packages.

```
# Navigate to your folder and run the script.
$ cd /path/to/folder
$ bash install_dependencies.sh
```

8. (Optional) Clone your repository. If you want your repository on another computer after completing steps 1–5, use the following commands.

```
# Navigate to where you want to put the folder.
$ cd ~/Desktop/or/something/

# Clone the folder from the online repository.
$ git clone git@bitbucket.org:<name>/<repo>.git <foldername>

# Record your credentials in the new folder.
$ cd <foldername>
$ git config --local user.name "your name"
$ git config --local user.email "your email"

# Download data files to the new folder.
$ bash download_data.sh
```

Setup Without Git

Even if you aren't using git to submit files, you must install it (<http://git-scm.com/downloads>) in order to get the data files for each lab. Share your folder with your instructor according to their directions, and follow steps 6 and 7 of the previous section to download the data files and install package dependencies.

Using Git

Git manages the history of a file system through commits, or checkpoints. Use `git status` to see the files that have been changed since the last commit. These changes are then moved to the staging area, a list of files to save during the next commit, with `git add <filename(s)>`. Save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

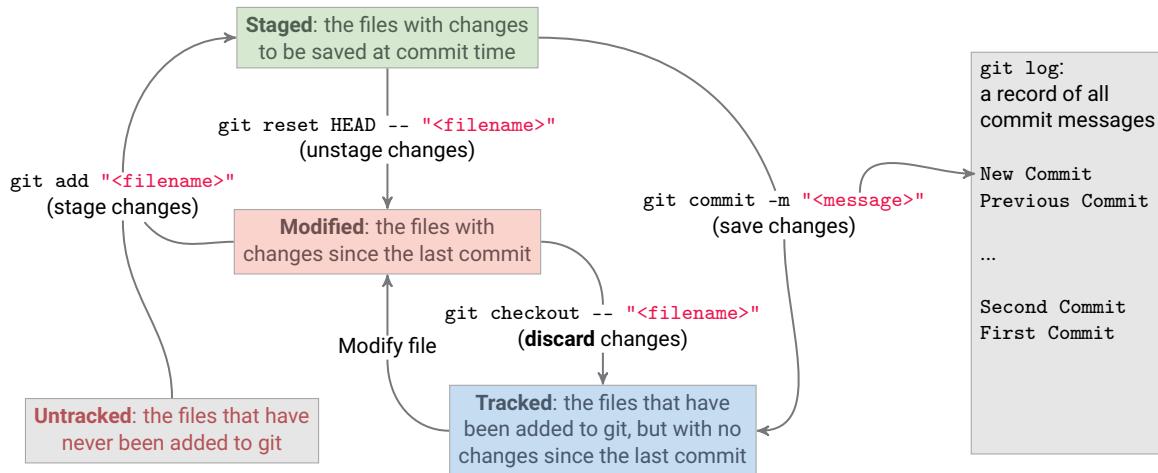


Figure A.1: Git commands to stage, unstage, save, or discard changes. Commit messages are recorded in the log.

All of these commands are done within a clone of the repository, stored somewhere on a computer. This repository must be manually synchronized with the online repository via two other git commands: `git pull origin master`, to pull updates from the web to the computer; and `git push origin master`, to push updates from the computer to the web.

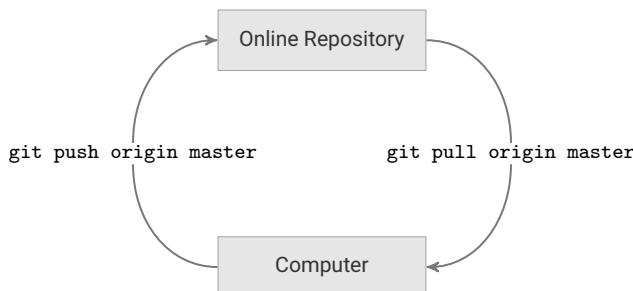


Figure A.2: Exchanging git commits between the repository and a local clone.

Command	Explanation
git status	Display the staging area and untracked changes.
git pull origin master	Pull changes from the online repository.
git push origin master	Push changes to the online repository.
git add <filename(s)>	Add a file or files to the staging area.
git add -u	Add all modified, tracked files to the staging area.
git commit -m "<message>"	Save the changes in the staging area with a given message.
git checkout -- <filename>	Revert changes to an unstaged file since the last commit.
git reset HEAD -- <filename>	Remove a file from the staging area.
git diff <filename>	See the changes to an unstaged file since the last commit.
git diff --cached <filename>	See the changes to a staged file since the last commit.
git config --local <option>	Record your credentials (<code>user.name</code> , <code>user.email</code> , etc.).

Table A.1: Common git commands.

Note

When pulling updates with `git pull origin master`, your terminal may sometimes display the following message.

```
Merge branch 'master' of git@bitbucket.org:<name>/<repo> into master

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.

~
```

This means that someone else (the instructor) has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have. This screen, displayed in vim ([https://en.wikipedia.org/wiki/Vim_\(text_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))), is asking you to enter a message (or use the default message) to create a merge commit that will reconcile both changes. To close this screen and create the merge commit, type :wq and press `enter`.

Example Work Sessions

```
$ cd ~/Desktop/Student-Materials/
$ git pull origin master                                # Pull updates.
### Make changes to a file.
$ git add -u                                            # Track changes.
$ git commit -m "Made some changes."                   # Commit changes.
$ git push origin master                               # Push updates.
```

```
# Pull any updates from the online repository (such as TA feedback).
$ cd ~/Desktop/Student-Materials/
$ git pull origin master
From bitbucket.org:username/repo
 * branch           master      -> FETCH_HEAD
Already up-to-date.

### Work on the labs. For example, modify PythonIntro/python_intro.py.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    PythonIntro/python_intro.py

# Track the changes with git.
$ git add PythonIntro/python_intro.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   PythonIntro/python_intro.py

# Commit the changes to the repository with an informative message.
$ git commit -m "Made some changes"
[master fed9b34] Made some changes
  1 file changed, 10 insertion(+) 1 deletion(-)

# Push the changes to the online repository.
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 327 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@bitbucket.org:username/repo.git
  5742a1b..fed9b34  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

B

Installing and Managing Python

Lab Objective: One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.

Installing Python via Anaconda

A Python distribution is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the Anaconda distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to <https://www.anaconda.com/download/>.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see <https://docs.anaconda.com/anaconda/install/>. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

Achtung!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A Python package manager is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see <https://xkcd.com/349/>).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See <https://docs.anaconda.com/anaconda/packages/pkg-docs> for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
<code>conda install <package-name></code>	Install the specified package.
<code>conda update <package-name></code>	Update the specified package.
<code>conda update conda</code>	Update <code>conda</code> itself.
<code>conda update anaconda</code>	Update all packages included in Anaconda.
<code>conda --help</code>	Display the documentation for <code>conda</code> .

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda          # Make sure that conda is up to date.
$ conda install matplotlib    # Attempt to install matplotlib.
$ conda update matplotlib     # Attempt to update matplotlib.
```

See <https://conda.io/docs/user-guide/tasks/manage-pkgs.html> for more examples.

Note

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

Achtung!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
<code>pip install package-name</code>	Install the specified package.
<code>pip install --upgrade package-name</code>	Update the specified package.
<code>pip freeze</code>	Display the version number on all installed packages.
<code>pip --help</code>	Display the documentation for <code>pip</code> .

See https://pip.pypa.io/en/stable/user_guide/ for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: <https://atom.io/>
- Sublime Text: <https://www.sublimetext.com/>
- Notepad++ (Windows): <https://notepad-plus-plus.org/>
- Geany: <https://www.geany.org/>
- Vim: <https://www.vim.org/>
- Emacs: <https://www.gnu.org/software/emacs/>

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                               # List the files in the current directory.
hello_world.py
$ cat hello_world.py               # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py            # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
```

```
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.  
  
In [1]: %run hello_world.py  
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

Note

While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with Powershell, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:

- Windows subsystem for linux: docs.microsoft.com/en-us/windows/wsl/.
- Git bash: <https://gitforwindows.org/>.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See <https://github.com/jupyter/jupyter/wiki/> for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Jupyter Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and L^AT_EX, and can embed images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An integrated development environment (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: <http://jupyterlab.readthedocs.io/en/stable/>
- PyCharm: <https://www.jetbrains.com/pycharm/>

- Spyder: <http://code.google.com/p/spyderlib/>
- Eclipse with PyDev: <http://www.eclipse.org/>, <https://www.pydev.org/>

See <https://realpython.com/python-ides-code-editors-guide/> for a good overview of these (and other) workflow tools.

C

NumPy Visual Guide

Lab Objective: NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n -dimensional arrays.

Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2, 1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax $[a:b]$ can be read as “the a th entry up to (but not including) the b th entry.” Similarly, $[a:]$ means “the a th entry to the end” and $[:b]$ means “everything up to (but not including) the b th entry.”

$$A[1] = A[1, :] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[:, 2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[1:, :2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[1:-1, 1:-1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

$$B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\text{np.hstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\text{np.vstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = [\times \quad \times \quad \times \quad \times]$$

$$y = [* \quad * \quad * \quad *]$$

$$\text{np.hstack}((x, y, x)) = [\times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times]$$

$$\text{np.vstack}((x, y, x)) = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix}$$

$$\text{np.column_stack}((x, y, x)) = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> for more in-depth examples and broadcasting rules.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad \mathbf{x} = [10 \quad 20 \quad 30]$$

$$\mathbf{A} + \mathbf{x} = \begin{array}{c} \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ + \\ [10 & 20 & 30] \end{bmatrix} \\ = \end{array} \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\mathbf{A} + \mathbf{x}.reshape((1, -1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$A.sum(axis=0) = \left[\begin{array}{c|c|c|c} 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{array} \right] = [4 \quad 8 \quad 12 \quad 16]$$

$$A.sum(axis=1) = \left[\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \hline \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \hline \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \end{array} \right] = [10 \quad 10 \quad 10 \quad 10]$$

Bibliography

- [ADH⁺01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, et al. Numerical python, 2001.
- [Hea02] Michael T Heath. Scientific computing. McGraw-Hill New York, 2002.
- [Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. Computing In Science & Engineering, 9(3):90–95, 2007.
- [JOP⁺] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 11/05/18].
- [Kiu13] Jaan Kiusalaas. Numerical methods in engineering with Python 3. Cambridge university press, 2013.
- [MMH04] Neil Muller, Lourenco Magaia, and B. M. Herbst. Singular value decomposition, eigenfaces, and 3D reconstructions. SIAM Rev., 46(3):518–545, 2004.
- [MSP⁺17] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondrej Certík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. PeerJ Computer Science, 3:e103, 2017.
- [New10] Mark Newman. Networks: an introduction. Oxford university press, 2010.
- [Oli06] Travis E Oliphant. A guide to NumPy, volume 1. Trelgol Publishing USA, 2006.
- [Oli07] Travis E Oliphant. Python for scientific computing. Computing in Science & Engineering, 9(3), 2007.
- [QSS10] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. Numerical mathematics, volume 37. Springer Science & Business Media, 2010.
- [SM00] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(8):888–905, 2000.
- [TB97] Lloyd N. Trefethen and David Bau, III. Numerical linear algebra. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.
- [VD10] Guido VanRossum and Fred L Drake. The python language reference. Python software foundation Amsterdam, Netherlands, 2010.