MATH96023/MATH97032/MATH97140 - Computational Linear Algebra

Edition 2023.0

Colin J. Cotter

Nov 29, 2024

CONTENTS

CHAPTER

ONE

GETTING READY FOR COMPUTATIONAL EXERCISES

In the course notes you will encounter computational exercises for you to complete. The object of these exercises is to build up understanding about how computational linear algebra algorithms actually work. Along the way you will have the opportunity to pick up valuable scientific computing skills in coding, software engineering and rigorous testing. They involve completing unfinished "skeleton" code, which we will then use in the courseworks.

In this preliminary section, we will explain how to get set up to do the computational exercises. If you have taken Principles of Programming (a second year optional course on our undergraduate Mathematics programme) then you will have already been shown how to do most of this. If not, we will provide some links to some material on that course that it is well worth reading.

You can use your own laptop running Windows, Linux, or macOS.

There is a lot of information below, but here is a summary checklist to check that you have everything ready to do your work.

1. Install Python, Git and a text editor using the instructions below.
2. Create a working folder and put a virtual environment in it (venv).
3. Clone your course repository from Github Classroom into the working folder.
4. Activate the venv.
5. Install `numpy` to the venv, and `pytest`.
6. Install the course module to the venv.
7. Remember to activate the venv every time you work on the course module.
8. Make sure that you understand how to run code in the course repository (including your own code you have added).
9. Make sure that you understand how to commit your changes to your Git repository and how to push them to Github.
10. Make sure that you know how to run the tests.

To follow these steps read the sections below.

1.1 Getting the software that you need

The core requirements are Python (version >=3.7), Git, and a Python-aware text editor.

In order to write the code required for the implementation exercise, you'll need to use a Python-aware text editor. There are many such editors available and you can use any you like. If you haven't used Python and/or Git before, it is a good idea to use Visual Studio Code (VSCode) which is a Python-aware text editor, since VSCode also provides a Terminal and an interface to Git.

Up to date information on how to install Python, Git and VSCode on a Windows, Linux or Mac machine is available at the Installing the necessary software section of the Principles of Programming website.

Task 1 *Install Python, Git and VSCode (or your preferred text editor for coding) on your computer that you will use for this course.*

Hint

If you are a Mac user, you'll need to avoid using the preinstalled Python on your system, as it is a very cut down version for interacting with the MacOS. You should install a fully featured Python (using Anaconda or Homebrew, as described in the link above).

Hint

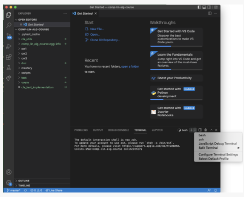
If you want to use Imperial's computer lab machines, they have the software you need installed, in some cases via the *Software Hub*. To get started, double click the *Software Hub* icon on the desktop or visit the Software Hub page.

1.2 The Terminal

A lot of the routine activity involved in this module revolves around executing commands on the Bash terminal (sometimes referred to as the "command line"). For example you use the terminal to work with the revision control system. If you're not familiar with the Linux terminal, then you can read this brief guide to the terminal. That guide focusses on the Bash shell, which is the one we will use.

Hint

In VSCode you can get a terminal by selecting New Terminal from the Terminal menu. This should open a Terminal window on your VS Code screen. To the top right of this window is a pulldown menu to select the interpreter, which needs to be Bash. The image to the right shows this pulldown menu.



Hint

In VSCode, to ensure you are using the correct Python interpreter,

1. Go to the View menu and select the Command Palette.
2. Start typing *Python: Select Interpreter*, and click on it when it comes up.
3. Select the correct Python interpreter from the pulldown menu (on Mac, the one you installed from Homebrew or Anaconda, on Windows, the one that you installed from the Microsoft Store).

1.3 Python virtual environment

The next step is to set up a Python virtual environment using the Terminal. This is described in Section 1.2 of the Principles of Programming website.

Task 2 *Create a new virtual environment for your Computational Linear Algebra work and activate it, following the instructions of Section 1.3 of PoP linked above.*

Two differences are as follows.

1. *You should name the working folder something more relevant to this course!*
2. *You should name the venv:*

clavenv

instead of:

PoP_venv

Hint

It is recommended you keep this name to avoid spuriously committing venv files to the git repository. If you decide to give your venv a different name, please add that name to the .gitignore file in your git repository. If you don't know what this means, it is probably best to use the name "clavenv".

Task 3 *Following the instructions in Section 1.3 of the Principles of Programming website, install the `numpy` and `pytest` packages to your venv (don't forget to activate it first).*

1.4 GitHub and git

Revision control is a mechanism for recording and managing different versions of changing software. This enables changes to be tracked and helps in the process of debugging code, and in managing conflicts when more than one person is working on the same project. Revision control can be combined with online hosting to provide secure backups and to enable you to work on code from different locations.

In this module, you'll use revision control to access the skeleton files. You'll also use the same revision control system to record the edits you make over time and to submit your work for feedback and, eventually, marking.

We will be using the revision control system git, which has cornered the market in this area now. We'll be combining git with the online hosting service GitHub.

There is a brief introduction to Git in the Just Enough Git to Get By section of the Principles of Programming website.

Task 4 *Read through (or review, if you read it before) Sections 2.1, 2.2, and 2.3 of Just Enough Git to Get By.*

Task 5 *Configure your Git installation by following the instructions in Section 2.4 of Just Enough Git to Get By.*

Hint

These instructions involve typing into the Terminal. VSCode provides other ways to configure but it is much easier to get help if you are typing into the Terminal. Make sure that you have selected the Bash interpreter for your Terminal.

Task 6 *Create and/or setup your Github account following the instructions in Section 2.5 of Just Enough Git to Get By.*

Task 7 *If you have not done it before, complete the simple exercise in Section 2.6 of Just Enough Git to Get By. This exercise uses the Git Training Assignment which is linked on Blackboard, which you should clone into your working folder.*

Hint

Above all else, never use:

```
git add -A
```

or:

git add *

to add all the files in the repository. This is bad practice and makes a mess for the markers, making them grumpy. When you commit changes to files in your repository for this course, just use:

```
git add
```

to add the files you changed to the list of files to be updated in the commit history.

Similarly, VSCode has a graphic interface for Git. It is preferred to use the Terminal in this course, as it is easier to get help. If you do decide to use the graphic interface, just ignore any files that are marked as not added. Do not try to click them to remove the marks.

⚠ **Warning**

Never clone a repository inside the folder of another folder.

1.5 Setting up your repository

We're using a tool called GitHub classroom to automate the creation of your copies of the repository. Follow the link on Blackboard marked "Course Repository" to create your personal repository for the course. Then, clone it to your working folder on your computer following the instructions in the previous section.

⚠ **Warning**

When you follow the link, you will be asked to select your "school's identifier" from a list. This will be your user ID that you use to log into Outlook (e.g., sbc21). If you don't find your ID on the list, *do not* click "Skip to the next step". And, *do not* click someone else's ID! Instead, contact the course leader and ask to have your user ID added. We need to do this so that we can grade your work.

Hint

To change folder in the terminal, type *cd <path>* where *<path>* is the path to the folder you want to change to. Paths can be "absolute" e.g. */home/users/jbloggs/comp-lin-alg/* or "relative" e.g. if you are currently in */home/users/jbloggs* then you can use *comp-lin-alg*. Typing *pwd* shows the current path, and typing *ls* shows the contents of the current folder. Typing *cd ..* changes to the enclosing folder, and typing *cd -* changes back to the previous folder. For more information see the "brief guide to the terminal" linked above.

Hint

In VSCode, you will be asked if you want to make this venv the default for your project. Select "yes" as this will help to ensure that it is activated.

Hint

Every time you want to work on the implementation exercises and courseworks, you need to activate the venv.

⚠ **Warning**

Your classroom repository will contain a branch called "feedback". Do not commit to, or remove, this branch. This is how we will provide feedback on your code.

1.5.1 Installing the course package to the venv

In this course we will be working on skeleton code stored as a Python package in the repository. This means that we will be able to import everything as a module using *from cla_utils import ** without needing to be in a particular folder. This is what makes the tests work, for example.

Task 8 *Install the course package to your venv. To do this:*

1. *Activate the clavenv as above.*
2. *Change folder to the repository that you just checked out (this should contain folders called* doc*,* cla_utils*,* test*, etc.).*
3. *Type* python -m pip install -e .

Task 9 *Read this useful information on Modules and Packages that will be useful later.*

1.6 How to do the computational exercises

For the computational exercises, quite a lot of the coding infrastructure you will need is provided already. Your task is to write the crucial mathematical operations at key points, as described on this website.

The code on which you will build is in the `cla_utils` folder of your repository. The code has embedded documentation which is used to build the *cla_utils package* web documentation.

As you do the exercises, **commit your code** to your repository. This will build up your computational exercise solution sets. You should commit code early and often - small commits are easier to understand and debug than large ones. Push your commits to your remote repository on Github.

Hint

In Git, we use the Terminal to commit changes and push them to the remote repository on Github Classroom. A repository is a record of the history of the code as you are working. To add a file to the list of files whose changes will be committed to the repository, type *git add <filename> -m <log message>*, where *<log message>* is a short description of the changes you made. To commit those changes, type *git commit*. They will now be saved locally. To push these changes to the “remote” repository on Github Classroom, type *git push* (you may be asked to set the name of the remote, just paste the suggested command into the Terminal). To pull changes from the remote repository on Github Classroom, type *git pull*. For further features and better explanation, please take a look at the Github Tutorial linked above.

Warning

Never use *git add **, since this will add unwanted files to the repository which shouldn’t be there. You should never add machine specific files such as your venv, or *.pyc* files which are temporary machine specific files generated by the Python interpreter. This really slows down the marking process and makes the markers grumpy. You should only add the *.py* files that you are working on.

Warning

Do not commit to the feedback branch. This branch is just there so that we can provide feedback on your changes to the main branch, and if you commit there, it will mess up our marking system.

1.7 Running your work

If you want to execute your code written in *cla_utils*, this can be imported into IPython (in the terminal, or using a Jupyter notebook), or in a script.

To use IPython, type *ipython* in the Terminal (when the venv is activated). You may need to install it first using *python -m pip install ipython* (you must start the venv first). Then you can import `cla_utils` interactively using *from cla_utils import **. To exit IPython type Ctrl-D.

Task 10 *Briefly read this Information about IPython.*

If you also import *numpy* then you can create example *numpy* arrays and pass them to `cla_utils` functions to try them out. You can also do this in a script, e.g.:

```
from cla_utils import *
from numpy import *
A = numpy.array([[1.0,2.0,0.,0.,1.0+1.0j],
                [0.0,1.0,3.,0.,0.],
                [0.0,0.0,1.,0.,0.],
                [0.0,0.0,0.,1.,0.],
                [0.0,0.0,0.,0.,1.]])
xr = numpy.array([1.,2.,1.,0.5,0.3])
xi = numpy.array([1.1,0.2,0.,1.5,-0.7])
ABiC(A, xr, xi)
```

After saving your text to a script with a filename ending in *.py*, e.g. *run_ABiC.py*, you can execute the script in the Terminal by typing *python run_ABiC.py* (remember to change to the folder where the file is located). Scripts are better because you can run the whole thing again more easily if you make a mistake, and you can save them.

Task 11 *Briefly read this information about Python scripts.*

> ⚠ **Warning**
>
> Don't clutter up your repository by adding these experimental scripts with *git add*. If you want to store them it is best to use another separate git repository for that.

1.8 Testing your work

As you complete the exercises, there will often be test scripts which check the code you have just written. These are located in the `test` folder and employ the pytest testing framework. You run the tests with:

```
python -m pytest test_script.py
```

from the bash Terminal, replacing `test_script.py` with the appropriate test file name (remember to activate the venv first). The `-x` option to `pytest` will cause the test to stop at the first failure it finds, which is often the best place to start fixing a problem. For those familiar with debuggers, the `--pdb` option will drop you into the Python debugger at the first error.

You can also run all the tests by running `pytest` on the tests folder. This works particularly well with the -x option, resulting in the tests being run in course order and stopping at the first failing test:

```
python -m pytest -x
```

You should make sure that your code passes tests before moving on to the next exercise.

1.9 Coding style and commenting

Computer code is not just functional, it also conveys information to the reader. It is important to write clear, intelligible code. **The readability and clarity of your code will count for marks**.

The Python community has agreed standards for coding, which are documented in PEP8. There are programs and editor modes which can help you with this. The skeleton implementation follows PEP8 quite closely. You are encouraged, especially if you are a more experienced programmer, to follow PEP8 in your implementation. Full code marks will require PEP8 compliance.

1.10 Skeleton code documentation

There is web documentation for the complete *cla_utils package*. There is also an alphabetical index and a search page.

CHAPTER

TWO

LINEAR ALGEBRA PRELIMINARIES

In this preliminary section, we revise a few key linear algebra concepts that will be used in the rest of the course, emphasising the column space of matrices. We will quote some standard results that should be found in an undergraduate linear algebra course.

Hint

Before you attempt any exercises, you need to make sure that you have everything you need set up on your computer. See the checklist in *the previous section*.

2.1 Matrices, vectors and matrix-vector multiplication

Supplementary video

https://player.vimeo.com/video/450145459

We will consider the multiplication of a vector

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad x_i \in \mathbb{C},\ i = 1, 2, \ldots, n, \text{ i.e. } x \in \mathbb{C}^n,$$

by a matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},$$

i.e. $A \in \mathbb{C}^{m\times n}$. A has m rows and n columns so that the product

$$b = Ax$$

produces $b \in \mathbb{C}^m$, defined by

$$b_i = \sum_{j=1}^{n} a_{ij} x_j,\ i = 1, 2, \ldots, m. \tag{2.1}$$

In this course it is important to consider the general case where $m \neq n$, which has many applications in data analysis, curve fitting etc. We will usually state generalities in this course for vectors over the field $\mathbb{C}$, noting where things specialise to $\mathbb{R}$.

Supplementary video

https://player.vimeo.com/video/450156255

We can quickly check that the map $x \to Ax$ given by matrix multiplication is a linear map from $\mathbb{C}^n \to \mathbb{C}^m$, since it is straightforward to check from the definition that

$$A(\alpha x + y) = \alpha Ax + Ay,$$

for all $x, y \in \mathbb{C}^n$ and $\alpha \in \mathbb{C}$. (Exercise: show this for yourself.)

Supplementary video

https://player.vimeo.com/video/450157385

It is very useful to interpret matrix-vector multiplication as a linear combination of the columns of A with coefficients taken from the entries of x. If we write A in terms of the columns,

$$A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \end{pmatrix},$$

where

$$a_i \in \mathbb{C}^m, \, i = 1, 2, \ldots, n,$$

then

$$b = \sum_{j=1}^{n} x_j a_j,$$

i.e. a linear combination of the columns of A as described above.

Supplementary video

https://player.vimeo.com/video/450161699

We can extend this idea to matrix-matrix multiplication. Taking $A \in \mathbb{C}^{m\times l}$, $C \in \mathbb{C}^{l\times n}$, $B \in \mathbb{C}^{m\times n}$, with $B = AC$, then the components of B are given by

$$b_{ij} = \sum_{k=1}^{l} a_{ik} c_{kj}, \quad 1 \leq i \leq m, \, 1 \leq j \leq n.$$

Writing $b_j \in \mathbb{C}^m$ as the jth column of B, for $1 \leq j \leq n$, and c_j as the jth column of C, we see that

$$b_j = Ac_j.$$

This means that the jth column of B is the matrix-vector product of A with the jth column of C. This kind of "column thinking" is very useful in understanding computational linear algebra algorithms.

Supplementary video

https://player.vimeo.com/video/450162431

An important example is the outer product of two vectors, $u \in \mathbb{C}^m$ and $v \in \mathbb{C}^n$. Here it is useful to see these vectors as matrices with one column, i.e. $u \in \mathbb{C}^{m\times 1}$ and $v \in \mathbb{C}^{n\times 1}$. The outer product is $uv^T \in \mathbb{C}^{m\times n}$. The columns of v^T are just single numbers (i.e. vectors of length 1), so viewing this as a matrix multiplication we see

$$uv^T = \begin{pmatrix} uv_1 & uv_2 & \ldots & uv_n \end{pmatrix},$$

which means that all the columns of uv^T are multiples of u. We will see in the next section that this matrix has rank 1. In the complex number case, the transpose ^{T} is replaced by the adjoint * which is the complex conjugate of the transpose. There will be more about this later.

2.1.1 Your first programming exercises

In this course, there will be programming exercises, the first one of which is coming up right now. The aim of these programming exercises is to gain understanding of the mathematical algorithms by expressing them as code. The `numpy` Python package has a module called `numpy.linalg` that contains many of these algorithms. Hence for this course we will not use this module, just use the functions and classes available when you import `numpy` itself. There is one exception, which is that `numpy.linalg.norm()` is quite useful, but also covers a lot of different cases which are not very edifying to replicate. Hence, we have included `numpy.linalg.norm()` in the *`cla_utils`* package as `cla_utils.norm()`, should you wish to use it.

Exercise 12 *The* `cla_utils.exercises1.basic_matvec()` *function has been left unimplemented. To finish the function, add code so that it computes the matrix-vector product $b = Ax$ from inputs A and x. In this first implementation, you should simply implement (2.1) with a double nested for loop (one for the sum over j, and one for the i elements of b). Run this script to test your code (and all the exercises from this exercise set):*

```
py.test test/test_exercises1.py
```

from the Bash Terminal. Make sure you commit your modifications and push them to your course repository.

Hint

Don’t forget to activate the virtual environment before running the tests to make sure that you have access to all the necessary packages

Hint

The fast array features of Python are provided by Numpy for which there is a helpful tutorial. There is also a handy guide for Matlab users. In that context, the code provided in this course will always use Numpy arrays, and never Numpy matrices.

Exercise 13 *The* `cla_utils.exercises1.column_matvec()` *function has been left unimplemented. To finish the function, add code so that it computes the matrix-vector product $b = Ax$ from inputs A and x. This second implementation should use the column-space formulation of matrix-vector multiplication, i.e., b is a weighted sum of the columns of A with coefficients given by the entries in x. This should be implemented with a single for loop over the entries of x. The test script* `test_exercises1.py` *will also test this function.*

Hint

It will be useful to use the Python “slice” notation, for example:

```
A[:, 3]
```

will return the 4th (since Python numbers from zero) column of A. For more information, see the Numpy documentation on slicing.

Exercise 14 *The* `cla_utils.exercises1.time_matvecs()` *function computes the execution time for these two implementations for some example matrices and compares them with the built-in Numpy matrix-vector product. Run this function and examine the output. You should observe that the basic implementation is much slower than the built-in implementation. This is because built-in Numpy operations use compiled C code that is wrapped in Python, which avoids the overheads of run-time interpretation of the Python code and manipulation of Python objects. Numpy is really useful for computational linear algebra programming because it preserves the readability and flexibility of Python (writing code that looks much more like maths, access to object-oriented programming models) whilst giving near-C speed if used appropriately. You can read more about the advantages of using Numpy here. You should also observe that the column implementation is somewhere between the speed of the basic implementation and the built-in implementation. This is because (if you did it correctly), each iteration of the for loop involves adding an entire array (a scaling of one of the columns of A) to another array (where b is being calculated). This will also use compiled C code through Numpy, removing some (but not all) of the Python overheads in the basic implementation.*

In this course, we will present algorithms in the notes that generally do not express the way that Numpy should be used to implement them. In these exercises you should consider the best way to make use of Numpy built-in operations (which will often make the code more maths-like and readable, as well as potentially faster).

2.2 Range, nullspace and rank

Supplementary video

https://player.vimeo.com/video/450162984

In this section we'll quickly rattle through some definitions and results.

Definition 15 (Range) *The range of A, range(A), is the set of vectors that can be expressed as Ax for some x.*

The next theorem follows as a result of the column space interpretation of matrix-vector multiplication.

Theorem 16 *range(A) is the vector space spanned by the columns of A.*

Definition 17 (Nullspace) *The nullspace null(A) of A (or kernel) is the set of vectors x satisfying $Ax = 0$, i.e.*

$$\textit{null}(A) = \{x \in \mathbb{C}^n : Ax = 0\}.$$

Supplementary video

https://player.vimeo.com/video/450166119

Definition 18 (Rank) *The column rank rank(A) of A is the dimension of the column space of A. The row rank rank(A) of A is the dimension of the row space of A. It can be shown that the column rank and row rank of a matrix are equal, so we shall just refer to the rank.*

If

$$A = \begin{pmatrix} a_1 & a_2 & \ldots & a_n \end{pmatrix},$$

the column space of A is span$(a_1, a_2, \ldots, a_n)$.

Definition 19 *An $m \times n$ matrix A is full rank if it has maximum possible rank i.e. rank equal to $\min(m, n)$.*

If $m \geq n$ then A must have n linearly independent columns to be full rank. The next theorem is then a consequence of the column space interpretation of matrix-vector multiplication.

Theorem 20 *An $m \times n$ matrix A is full rank if and only if it maps no two distinct vectors to the same vector.*

Definition 21 *A matrix A is called nonsingular, or invertible, if it is a square matrix ($m = n$) of full rank.*

Exercise 22 *The* `cla_utils.exercises1.rank2()` *function has been left unimplemented. To finish the function, add code so that it computes the rank-2 matrix $A = u_1v_1^* + u_2v_2^*$ from $u_1, u_2 \in \mathbb{C}^m$ and $v_1, v_2 \in \mathbb{C}^n$. As you can see, the function needs to implement this rank-2 matrix by first forming two matrices B and C from the inputs, and then forming A as the product of B and C. The test script* `test_exercises1.py` *in the* `test` *directory will also test this function.*

To measure the rank of A, we can use the built-in rank function:

```
r = numpy.linalg.matrix_rank(A)
```

and we should find that the rank is equal to 2. Can you explain why this should be the case (use the column space interpretation of matrix-matrix multiplication)?

2.3 Invertibility and inverses

Supplementary video

https://player.vimeo.com/video/450171203

This means that an invertible matrix has columns that form a basis for $\mathbb{C}^m$. Given the canonical basis vectors defined by

$$e_j = \begin{pmatrix} 0 \\ \dots \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{pmatrix},$$

i.e. e_j has all entries zero except for the jth entry which is 1, we can write

$$e_j = \sum_{k=1}^{m} z_{jk} a_k, \quad 1 \leq j \leq m.$$

In other words,

$$\begin{aligned} I &= \begin{pmatrix} e_1 & e_2 & \dots & e_m \end{pmatrix} \\ &= ZA. \end{aligned}$$

We call Z a (left) inverse of A. It can be shown that Z is the

unique left inverse of A, and that Z is also the unique right inverse of A, satisfying $I = AZ$. We write $Z = A^{-1}$.

The first four parts of the next theorem are a consequence of what we have so far, and we shall quote the fifth and sixth (see a linear algebra course).

Theorem 23 *Let $A \in \mathbb{C}^{m\times m}$. Then the following are equivalent.*

1. *A has an inverse.*
2. *rank$(A) = m$.*
3. *range$(A) = \mathbb{C}^m$.*
4. *null$(A) = \{0\}$.*
5. *0 is not an eigenvalue of A.*

6. *The determinant* $\det(A) \neq 0$.

Supplementary video

https://player.vimeo.com/video/450172407

Finding the inverse of a matrix can be seen as a change of basis. Considering the equation $Ax = b$, we have $x = A^{-1}b$ for invertible A. We have seen already that b can be written as

$$b = \sum_{j=1}^{m} x_j a_j.$$

Since the columns of A span $\mathbb{C}^m$, the entries of x thus provide the unique expansion of b in the columns of A which form a basis. Hence, whilst the entries of b give basis coefficients for b in the canonical basis $(e_1, e_2, \ldots, e_m)$, the entries of x give basis coefficients for b in the basis given by the columns of A.

Exercise 24 *For matrices of the form, $A = I + uv^*$, where I is the $m \times m$ identity matrix, and $u, v \in \mathbb{C}^m$, show that whenever A is invertible, the inverse is of the form $A^{-1} = I + \alpha uv^*$ where $\alpha \in \mathbb{C}$, and calculate the form of α.*

The `cla_utils.exercises1.rank1pert_inv()` *function has been left unimplemented. To finish the function, add code so that it computes A^{-1} using your formula (and not any built-in matrix inversion routines). The test script* `test_exercises1.py` *in the* `test` *directory will also test this function.*

Add a function to `cla_utils.exercises1` *that measures the time to compute the inverse of A for an input matrix of size 400, and compare with the time to compute the inverse of A using the built-in inverse:*

```
numpy.linalg.inv(A)
```

What do you observe? Why do you think this is? We will examine the cost of general purpose matrix inversion algorithms later.

2.4 Adjoints and Hermitian matrices

Supplementary video

https://player.vimeo.com/video/450173092

Definition 25 (Adjoint) *The adjoint (or Hermitian conjugate) of $A \in \mathbb{C}^{m\times n}$ is a matrix $A^* \in \mathbb{C}^{n\times m}$ (sometimes written $A^\dagger$ or A'), with*

$$a_{ij}^* = \bar{a_{ji}},$$

where the bar denotes the complex conjugate of a complex number. If $A^* = A$ then we say that A is Hermitian.

For real matrices, $A^* = A^T$. If $A = A^T$, then we say that the matrix is symmetric.

The following identity is very important when dealing with adjoints.

Theorem 26 *For matrices A, B with compatible dimensions (so that they can be multiplied),*

$$(AB)^* = B^* A^*.$$

Exercise 27 *(This is an advanced exercise if the other exercises are complete. If you are behind on the exercises please skip this one.)*

Consider a matrix $A = B + iC$ where $B, C \in \mathbb{R}^{m\times m}$ and A is Hermitian. Show that $B = B^T$ and $C = -C^T$. To save memory, instead of storing values of A ($m \times m$ complex numbers to store), consider equivalently storing a real-valued $m \times m$ array $\hat{A}$ with $\hat{A}_{ij} = B_{ij}$ for $i \geq j$ and $\hat{A}_{ij} = C_{ij}$ for $i < j$.

The `cla_utils.exercises1.ABiC()` *function has been left unimplemented. It should implement matrix vector multiplication $z = Ax$, returning the real and imaginary parts of z, given the real and imaginary parts of x as* *inputs, and given the real array $\hat{A}$ as above. You should implement the multiplication using real arithmetic only, with just one loop over the entries of x, using the column space interpretation of matrix-vector multiplication. The test script* `test_exercises1.py` *in the* `test` *directory will also test this function.*

Hint

You can use the Python "slice" notation, to assign into a slice of an array, for example:

```
x[3:5] = y[3:5]
```

will copy the 4th and 5th entries of y (Python numbers from zero, and the upper limit of the slice is the first index value not to use. For more information, see the Numpy documentation on slicing.

2.5 Inner products and orthogonality

Supplementary video

https://player.vimeo.com/video/450172520

The inner product is a critical tool in computational linear algebra.

Definition 28 (Inner product) *Let $x, y \in \mathbb{C}^m$. Then the inner product of x and y is*

$$x^*y = \sum_{i=1}^{m} \bar{x}_i y_i.$$

We will frequently use the natural norm derived from the inner product to define size of vectors.

Definition 29 (2-Norm) *Let $x \in \mathbb{C}^m$. Then the 2-norm of x is*

$$\|x\| = \sqrt{\sum_{i=1}^{m} |x_i|^2} = \sqrt{x^*x}.$$

Orthogonality will emerge as an early key concept in this course.

Definition 30 (Orthogonal vectors) *Let $x, y \in \mathbb{C}^m$. The two vectors are orthogonal if $x^*y = 0$.*

Similarly, let X, Y be two sets of vectors. The two sets are orthogonal if

$$x^*y = 0, \quad \forall x \in X,\, y \in Y.$$

A set S of vectors is itself orthogonal if

$$x^*y = 0, \quad \forall x, y \in S.$$

We say that S is orthonormal if we also have $\|x\| = 1$ for all $x \in S$.

2.6 Orthogonal components of a vector

Supplementary video

https://player.vimeo.com/video/450184086

Let $S = \{q_1, q_2, \ldots, q_n\}$ be an orthonormal set of vectors in $\mathbb{C}^m$, and take another arbitrary vector $v \in \mathbb{C}^m$. Now take

$$r = v - (q_1^*v)q_1 - (q_2^*v)q_2 - \ldots - (q_n^*v)q_n.$$

Then, we can check that r is orthogonal to S, by calculating for each $1 \leq i \leq n$,

$$\begin{aligned} q_i^* r = q_i^* v - (q_1^* v)(q_i^* q_1) - \ldots - (q_n^* v)(q_i^* q_n) \\ = q_i^* v - q_i^* v = 0, \end{aligned}$$

since $q_i^* q_j = 0$ if $i \neq j$, and 1 if $i = j$. Thus,

$$v = r + \sum_{i=1}^{n} (q_i^* v) q_i = r + \sum_{i=1}^{n} \underbrace{(q_i q_i^*)}_{\text{rank-1 matrix}} v.$$

If S is a basis for $\mathbb{C}^m$, then $n = m$ and $r = 0$, and we have

$$v = \sum_{i=1}^{m} (q_i q_i^*) v.$$

Exercise 31 *The* `cla_utils.exercises2.orthog_cpts()` *function has been left unimplemented. It should implement the above computation, returning r and the coefficients of the component of v in each orthonormal direction. The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

2.7 Unitary matrices

Supplementary video

https://player.vimeo.com/video/450184373

Definition 32 (Unitary matrices) *A matrix $Q \in \mathbb{C}^{m \times m}$ is unitary if $Q^* = Q^{-1}$.*

For real matrices, a matrix Q is orthogonal if $Q^T = Q^{-1}$.

Theorem 33 *The columns of a unitary matrix Q are orthonormal.*

Proof 34 *We have $I = Q^* Q$. Then using the column space interpretation of matrix-matrix multiplication,*

$$e_j = Q^* q_j,$$

where q_j is the jth column of Q. Taking row i of e_j, we have

$$\delta_{ij} = q_i^* q_j, \text{ where } \delta_{ij} = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{otherwise} \end{cases} .$$

Extending a theme from earlier, we can interpret $Q^* = Q^{-1}$ as representing a change of orthogonal basis. If $Qx = b$, then $x = Q^* b$ contains the coefficients of b expanded in the basis given by the orthonormal columns of Q.

Exercise 35 *The* `cla_utils.exercises2.solveQ()` *function has been left unimplemented. Given a square unitary matrix Q and a vector b it should solve $Qx = b$ using information above (it is not expected to work when Q is not unitary or square). The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

Add a function to `cla_utils.exercises2` *that measures the time to solve $Qx = b$ using* `solveQ` *for an input matrix of sizes 100, 200, 400, and compare with the times to solve the equation using the general purpose solve (which uses LU factorisation, which we will discuss later):*

```
x = numpy.linalg.solve(Q, b)
```

What did you expect and was it observed?

A quick way to get an orthogonal matrix is to take a general matrix A and find the QR factorisation, which we will cover in the next section.

Q, R = numpy.linalg.qr(A)

returns two matrices, of which Q is orthogonal.

2.8 Vector norms

Supplementary video

https://player.vimeo.com/video/450184674

Various vector norms are useful to measure the size of a vector. In computational linear algebra we need them for quantifying errors etc.

Definition 36 (Norms) *A norm is a function $\|\cdot\|:\mathbb{C}^m\to\mathbb{R}$, such that*

1. $\|x\|\geq 0$*, and* $\|x\|=0 \implies x=0$.
2. $\|x+y\|\leq\|x\|+\|y\|$ *(triangle inequality).*
3. $\|\alpha x\|=|\alpha|\|x\|$ *for all* $x\in\mathbb{C}^m$ *and* $\alpha\in\mathbb{C}$.

We have already seen the 2-norm, or Euclidean norm, which is part of a larger class of norms called p-norms, with

$$\|x\|_p=\left(\sum_{i=1}^m |x_i|^p\right)^{1/p},$$

for real $p>0$. We will also consider weighted norms

$$\|x\|_{W,p}=\|Wx\|_p,$$

where W is a matrix.

2.9 Projectors and projections

Supplementary video

https://player.vimeo.com/video/450185110

Definition 37 (Projector) *A projector P is a square matrix that satisfies $P^2=P$.*

If $v\in\text{range}(P)$, then there exists x such that $Px=v$. Then,

$$Pv=P(Px)=P^2x=Px=v,$$

and hence multiplying by P does not change v.

Now suppose that $Pv\neq v$ (so that $v\notin\text{range}(P)$). Then,

$$P(Pv-v)=P^2v-Pv=Pv-Pv=0,$$

which means that $Pv-v$ is the nullspace of P. We have

$$Pv-v=-(I-P)v.$$

Definition 38 (Complementary projector) *Let P be a projector. Then we call $I-P$ the complementary projector.*

To see that $I-P$ is also a projector, we just calculate,

$$(I-P)^2 = I^2 - 2P + P^2 = I - 2P + P = I - P.$$

If $Pu = 0$, then $(I-P)u = u$.

In other words, the nullspace of P is contained in the range of $I-P$.

On the other hand, if v is in the range of $I-P$, then there exists some w such that

$$v = (I-P)w = w - Pw.$$

We have

$$Pv = P(w - Pw) = Pw - P^2 w = Pw - Pw = 0.$$

Hence, the range of $I-P$ is contained in the nullspace of P. Combining these two results we see that the range of $I-P$ is equal to the nullspace of P. Since P is the complementary projector to $I-P$, we can repeat the same argument to show that the range of P is equal to the nullspace of $I-P$.

We see that a projector P separates $\mathbb{C}^m$ into two subspaces, the nullspace of P and the range of P. In fact the converse is also true: given two subspaces S_1 and S_2 of $\mathbb{C}^m$ with $S_1 \cap S_2 = \{0\}$, then there exists a projector P whose range is S_1 and whose nullspace is S_2.

Supplementary video

https://player.vimeo.com/video/450185494

Now we introduce orthogonality into the concept of projectors.

Definition 39 (Orthogonal projector) *P is an orthogonal projector if*

$$(Pv)^*(Pv - v) = 0, \, \forall v \in \mathbb{C}^m.$$

In this case, P separates the space into two orthogonal subspaces.

2.10 Constructing orthogonal projectors from sets of orthonormal vectors

Let $\{q_1, \ldots, q_n\}$ be an orthonormal set of vectors in $\mathbb{C}^m$. We write

$$\hat{Q} = \begin{pmatrix} q_1 & q_2 & \ldots & q_n \end{pmatrix}.$$

Previously we showed that for any $v \in \mathbb{C}^m$, we have

$$v = \underbrace{r}_{\text{Orthogonal to column space of } \hat{Q}} + \underbrace{\sum_{i=1}^n (q_i q_i^*) v}_{\text{in the column space of } \hat{Q}}.$$

Hence, the map

$$v \mapsto Pv = \underbrace{\sum_{i=1}^n (q_i q_i^*)}_{=P} v,$$

is an orthogonal projector. In fact, P has very simple form.

Theorem 40 *The orthogonal projector P takes the form*

$$P = \hat{Q}\hat{Q}^*.$$

Proof 41 *From the change of basis interpretation of multiplication by $\hat{Q}^*$, the entries in $\hat{Q}^* v$ gives coefficients of the projection of v onto the column space of $\hat{Q}$ when expanded using the columns as a basis. Then, multiplication by $\hat{Q}$ gives the projection of v expanded again in the canonical basis. Hence, multiplication by $\hat{Q}\hat{Q}^*$ gives exactly the same result as multiplication by the formula for P above.*

This means that $\hat{Q}\hat{Q}^*$ is an orthogonal projection onto the range of $\hat{Q}$. The complementary projector is $P_\perp = I - \hat{Q}\hat{Q}^*$ is an orthogonal projection onto the nullspace of $\hat{Q}$.

An important special case is when $\hat{Q}$ has just one column, and then

$$P = q_1 q_1^*, \, P_\perp = I - q_1 q_1^*.$$

We notice that $P^* = (\hat{Q}\hat{Q}^*) = \hat{Q}\hat{Q}^* = P$. In fact the following is true.

Theorem 42 *$P = P^*$ if and only if $\hat{Q}$ is orthogonal.*

Exercise 43 *The* `cla_utils.exercises2.orthog_proj()` *function has been left unimplemented. Given an orthonormal set $q_1, q_2, \ldots, q_n$, it should provide the orthogonal projector P. The test script* `test_exercises2.py` *in the* `test` *directory will also test this function.*

CHAPTER

THREE

QR FACTORISATION

A common theme in computational linear algebra is transformations of matrices and algorithms to implement them. A transformation is only useful if it can be computed efficiently and sufficiently free of pollution from truncation errors (either due to finishing an iterative algorithm early, or due to round-off errors). A particularly powerful and insightful transformation is the QR factorisation. In this section we will introduce the QR factorisation and some good and bad algorithms to compute it.

3.1 What is the QR factorisation?

Supplementary video

https://player.vimeo.com/video/450191857

We start with another definition.

Definition 44 (Upper triangular matrix) *An $m \times n$ upper triangular matrix R has coefficients satisfying $r_{ij} = 0$ when $i > j$.*

It is called upper triangular because the nonzero rows form a triangle on and above the main diagonal of R.

Now we can describe the QR factorisation.

Definition 45 (QR factorisation) *A QR factorisation of an $m \times n$ matrix A consists of an $m \times m$ unitary matrix Q and an $m \times n$ upper triangular matrix R such that $A = QR$.*

The QR factorisation is a key tool in analysis of datasets, and polynomial fitting. It is also at the core of one of the most widely used algorithms for finding eigenvalues of matrices. We shall discuss all of this later during this course.

When $m > n$, R must have all zero rows after the n th row. Hence, it makes sense to only work with the top $n \times n$ block of R consisting of the first n rows, which we call $\hat{R}$. Similarly, in the matrix-matrix product QR, all columns of Q beyond the n th column get multiplied by those zero rows in R, so it makes sense to only work with the first n columns of Q, which we call $\hat{Q}$. We then have the reduced QR factorisation, $\hat{Q}\hat{R}$.

Exercise 46 *The `cla_utils.exercises2.orthog_space()` function has been left unimplemented. Given a set of vectors $v_1, v_2, \ldots, v_n$ that span the subspace $U \subset \mathbb{C}^m$, the function should find an orthonormal basis for the orthogonal complement $U^\perp$ given by*

$$U^\perp = \{x \in \mathbb{C}^m : x^* v = 0, \, \forall v \in U\}.$$

It is expected that it will only compute this up to a tolerance. You should make use of the built in QR factorisation routine `numpy.linalg.qr()`. The test script `test_exercises2.py` in the `test` directory will test this function.

In the rest of this section we will examine some algorithms for computing the QR factorisation, before discussing the application to least squares problems. We will start with a bad algorithm, before moving on to some better ones.

3.2 QR factorisation by classical Gram-Schmidt algorithm

Supplementary video

https://player.vimeo.com/video/450192200

The classical Gram-Schmidt algorithm for QR factorisation is motivated by the column space interpretation of the matrix-matrix multiplication $A = QR$, namely that the jth column a_j of A is a linear combination of the orthonormal columns of Q, with the coefficients given by the jth column r_j of R.

The first column of R only has a non-zero entry in the first row, so the first column of Q must be proportional to A, but normalised (i.e. rescaled to have length 1). The scaling factor is this first row of the first column of R. The second column of R has only non-zero entries in the first two rows, so the second column of A must be writeable as a linear combination of the first two columns of Q. Hence, the second column of Q must by the second column of A with the first column of Q projected out, and then normalised. The first row of the second column of R is then the coefficient for this projection, and the second row is the normalisation scaling factor. The third row of Q is then the third row of A with the first two columns of Q projected out, and so on.

Hence, finding a QR factorisation is equivalent to finding an orthonormal spanning set for the columns of A, where the span of the first j elements of the spanning set and of the first j columns of A is the same, for $j = 1, \ldots, n$.

Hence we have to find R coefficients such that

$$\begin{aligned} q_1 &= \frac{a_1}{r_{11}}, \\ q_2 &= \frac{a_2 - r_{12}q_1}{r_{22}} \\ &\vdots \\ q_n &= \frac{a_n - \sum_{i=1}^{n-1} r_{in}q_i}{r_{nn}}, \end{aligned}$$

with $(q_1, q_2, \ldots, q_n)$ an orthonormal set. The non-diagonal entries of R are found by inner products, i.e.,

$$r_{ij} = q_i^* a_j, \, i < j,$$

and the diagonal entries are chosen so that $\|q_i\| = 1$, for $i = 1, 2, \ldots, n$, i.e.

$$|r_{jj}| = \left\| a_j - \sum_{i=1}^{j-1} r_{ij} q_i \right\|.$$

Note that this absolute value does leave a degree of nonuniqueness in the definition of R. It is standard to choose the diagonal entries to be real and non-negative.

We now present the classical Gram-Schmidt algorithm as pseudo-code.

```
• FOR $j = 1$ TO $n$
    – $v_j \leftarrow a_j$
    – FOR $i = 1$ TO $j - 1$
        * $r_{ij} \leftarrow q_i^* a_j$
    – END FOR
    – FOR $i = 1$ TO $j - 1$
        * $v_j \leftarrow v_j - r_{ij} q_i$
    – END FOR
    – $r_{jj} \leftarrow \|v_j\|_2$
    – $q_j \leftarrow v_j / r_{jj}$
• END FOR
```

(Remember that Python doesn't have END FOR statements, but instead uses indentation to terminate code blocks. We'll write END statements for code blocks in pseudo-code in these notes.)

Exercise 47 (‡) *The* `cla_utils.exercises2.GS_classical()` *function has been left unimplemented. It should implement the classical Gram-Schmidt algorithm above, using Numpy slice notation so that only one Python for loop is used. The function should work "in place" by changing the values in A, without introducing additional intermediate arrays (you will need to create a new array to store R). The test script* `test_exercises2.py` *in the* `test` *directory will test this function.*

Hint

The (‡) symbol in an exercise indicates that the code for that exercise is in scope for use in the coursework. When the code is used in the coursework, we will grade the code quality, for appropriate use of Numpy slice operations, efficient use of array memory, loop minimisation, and avoiding computation inside loops that could be done beforehand.

3.3 Projector interpretation of Gram-Schmidt

Supplementary video

https://player.vimeo.com/video/450192723

At each step of the Gram-Schmidt algorithm, a projector is applied to a column of A. We have

$$\begin{aligned} q_1 &= \frac{P_1 a_1}{\|P_1 a_1\|}, \\ q_2 &= \frac{P_2 a_2}{\|P_2 a_2\|}, \\ &\vdots \\ q_n &= \frac{P_n a_n}{\|P_n a_n\|}, \end{aligned}$$

where P_j are orthogonal projectors that project out the first $j-1$ columns $(q_1, \ldots, q_{j-1})$ (P_1 is the identity as this set is empty when $j = 1$). The orthogonal projector onto the first $j-1$ columns is $\hat{Q}_{j-1}\hat{Q}^*_{j-1}$, where

$$\hat{Q}_{j-1} = \begin{pmatrix} q_1 & q_2 & \ldots & q_{j-1} \end{pmatrix}.$$

Hence, P_j is the complementary projector, $P_j = I - \hat{Q}_{j-1}\hat{Q}^*_{j-1}$.

3.4 Modified Gram-Schmidt

Supplementary video

https://player.vimeo.com/video/450193303

There is a big problem with the classical Gram-Schmidt algorithm. It is unstable, which means that when it is implemented in inexact arithmetic on a computer, round-off error unacceptably pollutes the entries of Q and R, and the algorithm is not useable in practice. What happens is that the columns of Q are not quite orthogonal, and this loss of orthogonality spoils everything. We will discuss stability later in the course, but right now we will just discuss the fix for the classical Gram-Schmidt algorithm, which is based upon the projector interpretation which we just discussed.

To reorganise Gram-Schmidt to avoid instability, we decompose P_j into a sequence of $j-1$ projectors of rank $m-1$, that each project out one column of Q, i.e.

$$P_j = P_{\perp q_{j-1}} \ldots P_{\perp q_2} P_{\perp q_1},$$

where

$$P_{\perp q_j} = I - q_j q_j^*.$$

Then,

$$v_j = P_j a_j = P_{\perp q_{j-1}} \ldots P_{\perp q_2} P_{\perp q_1} a_j.$$

Here we notice that we must apply $P_{\perp q_1}$ to all but one columns of A, and $P_{\perp q_2}$ to all but two columns of A, $P_{\perp q_3}$ to all but three columns of A, and so on.

By doing this, we gradually transform A to a unitary matrix, as follows.

$$\begin{aligned} A &= \begin{pmatrix} a_1 & a_2 & a_3 & \ldots & a_n \end{pmatrix} \\ & \begin{pmatrix} q_1 & v_2^1 & v_3^1 & \ldots & v_n^1 \end{pmatrix} \\ \rightarrow & \begin{pmatrix} q_1 & q_2 & v_3^2 & \ldots & v_n^2 \end{pmatrix} \\ \ldots \rightarrow & \begin{pmatrix} q_1 & q_2 & q_3 & \ldots & q_n \end{pmatrix}. \end{aligned}$$

Then it is just a matter of keeping a record of the coefficients of the projections and normalisation scaling factors and storing them in R.

This process is mathematically equivalent to the classical Gram-Schmidt algorithm, but the arithmetic operations happen in a different order, in a way that turns out to reduce accumulation of round-off errors.

We now present this modified Gram-Schmidt algorithm as pseudo-code.

```
• FOR $i = 1$ TO $n$
    – $v_i \leftarrow a_i$
• END FOR
• FOR $i = 1$ TO $n$
    – $r_{ii} \leftarrow \|v_i\|_2$
    – $q_i = v_i / r_{ii}$
    – FOR $j = i + 1$ TO $n$
        * $r_{ij} \leftarrow q_i^* v_j$
        * $v_j \leftarrow v_j - r_{ij} q_i$
    – END FOR
• END FOR
```

This algorithm can be applied "in place", overwriting the entries in A with the v s and eventually the q s.

Exercise 48 (‡) *The* `cla_utils.exercises2.GS_modified()` *function has been left unimplemented. It should implement the modified Gram-Schmidt algorithm above, using Numpy slice notation where possible. What is the minimal number of Python for loops possible?*

The function should work "in place" by changing the values in A, without introducing additional intermediate arrays (you will need to create a new array to store R). The test script `test_exercises2.py` *in the* `test` *directory will test this function.*

Exercise 49 *Investigate the mutual orthogonality of the Q matrices that are produced by your classical and modified Gram-Schmidt implementations. Is there a way to test mutual orthogonality without writing a loop? Round-off typically causes problems for matrices with large condition numbers and large off-diagonal values. You could also try the opposite of what was done in* `test_GS_classical`*: instead of ensuring that all of the entries in the diagonal matrix D are $\mathcal{O}(1)$, try making some of the values small and some large. See if you can find a matrix that illustrates the differences in orthogonality between the two algorithms.*

3.5 Modified Gram-Schmidt as triangular orthogonalisation

Supplementary video

https://player.vimeo.com/video/450193575

This iterative transformation process can be written as right-multiplication by an upper triangular matrix. For example, at the first iteration,

$$\underbrace{\begin{pmatrix} v_1^0 & v_2^0 & \ldots & v_n^0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \frac{1}{r_{11}} & -\frac{r_{12}}{r_{11}} & \ldots & \ldots & -\frac{r_{1n}}{r_{11}} \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ldots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}}_{R_1} = \underbrace{\begin{pmatrix} q_1 & v_2^1 & \ldots & v_n^1 \end{pmatrix}}_{A_1}.$$

To understand this equation, we can use the column space interpretation of matrix-matrix multiplication. The columns of A_1 are linear combinations of the columns of A with coefficients given by the columns of R_1. Hence, q_1 only depends on v_1^0, scaled to have length 1, and v_i^1 is a linear combination of (v_1^0, v_i^0) such that v_i^1 is orthogonal to q_1, for $1 < i \leq n$.

Similarly, the second iteration may be written as

$$\underbrace{\begin{pmatrix} v_1^1 & v_2^1 & \ldots & v_n^1 \end{pmatrix}}_{A_1} \underbrace{\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & \frac{1}{r_{22}} & -\frac{r_{23}}{r_{22}} & \ldots & -\frac{r_{2n}}{r_{22}} \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ldots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}}_{R_2} = \underbrace{\begin{pmatrix} q_1 & q_2 & v_3^2 \ldots & v_n^2 \end{pmatrix}}_{A_2}.$$

It should become clear that each transformation from A_i to A_{i+1} takes place by right multiplication by an upper triangular matrix R_{i+1}, which is an identity matrix plus entries in row i. By combining these transformations together, we obtain

$$A \underbrace{R_1 R_2 \ldots R_n}_{\hat{R}^{-1}} = \hat{Q}.$$

Since upper triangular matrices form a group, the product of the R_i matrices is upper triangular. Further, all the R_i matrices have non-zero determinant, so the product is invertible, and we can write this as $\hat{R}^{-1}$. Right multiplication by $\hat{R}$ produces the usual reduced QR factorisation. We say that modified Gram-Schmidt implements triangular orthogonalisation: the transformation of A to an orthogonal matrix by right multiplication of upper triangular matrices.

This is a powerful way to view the modified Gram-Schmidt process from the point of view of understanding and analysis, but of course we do not form the matrices R_i explicitly (we just follow the pseudo-code given above).

Exercise 50 *In a break from the format so far, the* `cla_utils.exercises2.GS_modified_R()` *function has been implemented. It implements the modified Gram-Schmidt algorithm in the form describe above using upper triangular matrices. This is not a good way to implement the algorithm, because of the inversion of R at the end, and the repeated multiplication by zeros in multiplying entries of the R_k matrices, which is a waste. However it is important as a conceptual tool for understanding the modified Gram-Schmidt algorithm as a triangular orthogonalisation process, and so it is good to see this in a code implementation. Study this function to check that you understand what is happening.*

However, the `cla_utils.exercises2.GS_modified_get_R()` *function has not been implemented. This function computes the R_k matrices at each step of the process. Complete this code. The test script* `test_exercises2.py` *in the* `test` *directory will also test this function.*

3.6 Householder triangulation

Supplementary video

https://player.vimeo.com/video/450199222

This view of the modified Gram-Schmidt process as triangular orthogonalisation gives an idea to build an alternative algorithm. Instead of right multiplying by upper triangular matrices to transform A to $\hat{Q}$, we can consider left multiplying by unitary matrices to transform A to R,

$$\underbrace{Q_n \ldots Q_2 Q_1}_{=Q^*} A = R.$$

Multiplying unitary matrices produces unitary matrices, so we obtain $A = QR$ as a full factorisation of A.

Supplementary video

https://player.vimeo.com/video/450199366

To do this, we need to work on the columns of A, from left to right, transforming them so that each column has zeros below the diagonal. These unitary transformations need to be designed so that they don't spoil the structure created in previous columns. The easiest way to ensure this is construct a unitary matrix Q_k with an identity matrix as the $(k-1) \times (k-1)$ submatrix,

$$Q_k = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F \end{pmatrix}.$$

This means that multiplication by Q_k won't change the first $k-1$ rows, leaving the previous work to remove zeros below the diagonal undisturbed. For Q_k to be unitary and to transform all below diagonal entries in column k to zero, we need the $(n-k+1) \times (n-k+1)$ submatrix F to also be unitary, since

$$Q_k^* = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F^* \end{pmatrix}, \, Q_k^{-1} = \begin{pmatrix} I_{k-1} & 0 \\ 0 & F^{-1} \end{pmatrix}.$$

We write the k th column v_k^k of A_k as

$$v_k^k = \begin{pmatrix} \hat{v}_k^k \\ x \end{pmatrix},$$

where $\hat{v}_k^k$ contains the first $k-1$ entries of v_k^k. The column gets transformed according to

$$Q_k v_k^k = \begin{pmatrix} \hat{v}_k^k \\ Fx \end{pmatrix}.$$

and our goal is that Fx is zero, except for the first entry (which becomes the diagonal entry of $Q_k v_k^k$). Since F is unitary, we must have $\|Fx\| = \|x\|$. For now we shall specialise to real matrices, so we choose to have

$$Fx = \pm \|x\| e_1,$$

where we shall consider the sign later. Complex matrices have a more general formula for Householder transformations which we shall not discuss here.

We can achieve this by using a Householder reflector for F, which is a unitary transformation that does precisely what we need. Geometrically, the idea is that we consider a line joining x and $Fx = \pm\|x\|e_1$, which points in the direction $v = \pm\|x\|e_1 - x$. We can transform x to Fx by a reflection in the hyperplane H that is orthogonal to v. Since reflections are norm preserving, F must be unitary. Applying the projector P given by

$$Px = \left(I - \frac{vv^*}{v^*v}\right)x,$$

does half the job, producing a vector in H. To do a reflection we need to go twice as far,

$$Fx = \left(I - 2\frac{vv^*}{v^*v}\right)x.$$

We can check that this does what we want,

$$\begin{aligned}
Fx &= \left(I - 2\frac{vv^*}{v^*v}\right)x, \\
&= x - 2\frac{(\pm\|x\|e_1 - x)}{\|\pm\|x\|e_1 - x\|^2}(\pm\|x\|e_1 - x)^*x, \\
&= x - 2\frac{(\pm\|x\|e_1 - x)}{\|\pm\|x\|e_1 - x\|^2}\|x\|(\pm x_1 - \|x\|), \\
&= x + (\pm\|x\|e_1 - x) = \pm\|x\|e_1,
\end{aligned}$$

as required, having checked that (assuming x is real)

$$\|\pm\|x\|e_1 - x\|^2 = \|x\|^2 \mp 2\|x\|x_1 + \|x\|^2 = -2\|x\|(\pm x_1 - \|x\|).$$

We can also check that F is unitary. First we check that F is Hermitian,

$$\begin{aligned}
\left(I - 2\frac{vv^*}{v^*v}\right)^* &= I - 2\frac{(vv^*)^*}{v^*v}, \\
&= I - 2\frac{(v^*)^*v^*}{v^*v}, \\
&= I - 2\frac{vv^*}{v^*v} = F.
\end{aligned}$$

Now we use this to show that F is unitary,

$$\begin{aligned}
F^*F &= \left(I - 2\frac{vv^*}{v^*v}\right)\left(I - 2\frac{vv^*}{v^*v}\right) \\
&= I - 4\frac{vv^*}{v^*v} + 4\frac{vv^*}{v^*v}\frac{vv^*}{v^*v} = I,
\end{aligned}$$

so $F^* = F^{-1}$. In summary, we have constructed a unitary matrix Q_k that transforms the entries below the diagonal of the kth column of A_k to zero, and leaves the previous $k-1$ columns alone.

Supplementary video

https://player.vimeo.com/video/450200163

Earlier, we mentioned that there is a choice of sign in v. This choice gives us the opportunity to improve the numerical stability of the algorithm. In the case of real matrices, to avoid unnecessary numerical round off, we choose the sign that makes v furthest from x, i.e.

$$v = \mathrm{sign}(x_1)\|x\|e_1 + x.$$

(Exercise, show that this choice of sign achieves this.) It is critical that we use a definition of sign that always returns a number that has magnitude 1, so we conventionally choose $\mathrm{sign}(0) = 1$.

Hint

Note that the `numpy.sign` function has $\mathrm{sign}(0) = 0$, so you need to take care of this case separately in your Python implementation.

For complex valued matrices, the Householder reflection uses $x_1/|x_1|$ (except for $x_1 = 0$ where we use 1 as above).

We are now in a position to describe the algorithm in pseudo-code. Here it is described an "in-place" algorithm, where the successive transformations to the columns of A are implemented as replacements of the values in A. This means that we can allocate memory on the computer for A which is eventually replaced with the values for R. To present the algorithm, we will use the "slice" notation to describe submatrices of A, with $A_{k:l,r:s}$ being the submatrix of A consisting of the rows from k to l and columns from r to s.

```
• FOR $k = 1$ TO $n$
    – $x = A_{k:m,k}$
    – $v_k \leftarrow \mathrm{sign}(x_1)\|x\|_2 e_1 + x$
    – $v_k \leftarrow v_k/\|v_k\|$
    – $A_{k:m,k:n} \leftarrow A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})$.
• END FOR
```

Exercise 51 (‡) *The* `cla_utils.exercises3.householder()` *function has been left unimplemented. It should implement the algorithm above, using only one loop over k. It should return the resulting R matrix. The test script* `test_exercises3.py` *in the* `test` *directory will test this function.*

Hint

The "slice" notation using colons ":" is more an more important when we are doing operations on submatrices. For example,

Hint

Don't forget that Python numbers from zero, which will be important when implementing the submatrices using Numpy slice notation.

Hint

The Python functions "inner", "outer", and "dot" are very useful for succinctly expressing many linear algebra operations. "inner(a,b)" multiplies each component of a multidimensional array a with the corresponding component of another array b (with the same "shape") and sums over all indices. "dot(a, b)" only sums over the last index of a and the first index of b. For some algorithms it may be necessary to swap the order of a and b and make use of transposes to sum over the correct index.

Supplementary video

https://player.vimeo.com/video/450201578

Note that we have not explicitly formed the matrix Q or the product matrices Q_i. In some applications, such as solving least squares problems, we don't explicitly need Q, just the matrix-vector product Q^*b with some vector b. To compute this product, we can just apply the same operations to b that are applied to the columns of A. This can be expressed in the following pseudo-code, working "in place" in the storage of b.

```
• FOR $k = 1$ TO $n$
    – $b_{k:m} \leftarrow b_{k:m} - 2v_k(v_k^* b_{k:m})$
• END FOR
```

We call this procedure "implicit multiplication".

Supplementary video

https://player.vimeo.com/video/450202242

In this section we will frequently encounter systems of the form

$$\hat{R}x = y.$$

This is an upper triangular system that can be solved efficiently using back-substitution.

Written in components, this equation is

$$\begin{aligned} R_{11}x_1 + R_{12}x_2 + \ldots + R_{1(m-1)}x_{m-1} + R_{1m}x_m &= y_1, \\ 0x_1 + R_{22}x_2 + \ldots + R_{2(m-1)}x_{m-1} + R_{2m}x_m &= y_2, \\ &\vdots \\ 0x_1 + 0x_2 + \ldots + R_{(m-1)(m-1)}x_{m-1} + R_{(m-1)m}x_m &= y_{m-1}, \\ 0x_1 + 0x_2 + \ldots + 0x_{m-1} + R_{mm}x_m &= y_m. \end{aligned}$$

The last equation yields x_m directly by dividing by R_{mm}, then we can use this value to directly compute x_{m-1}. This is repeated for all of the entries of x from m down to 1. This procedure is called back substitution, which we summarise in the following pseudo-code.

```
• $x_m \leftarrow y_m/R_{mm}$
• FOR $i = m - 1$ TO 1 (BACKWARDS)
    – $x_i \leftarrow (y_i - \sum_{k=i+1}^{m} R_{ik}x_k)/R_{ii}$
```

Exercise 52 (‡) *The function* `cla_utils.exercises3.solve_U()` *has been left unimplemented. It should use backward substitution to solve upper triangular systems. The interfaces are set so that multiple right hand sides can be provided and solved at the same time. The functions should only use one loop over the rows of U, to efficiently solve the multiple problems. The test script* `test_exercises3.py` *in the* `test` *directory will test these functions.*

Exercise 53 (‡) *Show that the implicit multiplication procedure is equivalent to computing an extended array*

$$\hat{A} = \begin{pmatrix} a_1 & a_2 & \ldots & a_n & b \end{pmatrix}$$

and performing Householder on the first n rows. Transform the equation $Ax = b$ into $Rx = \hat{b}$ where $QR = A$, and find the form of $\hat{b}$, explaining how to get $\hat{b}$ from Householder applied to $\hat{A}$ above. Solving systems with upper triangular matrices is much cheaper than solving general matrix systems as we shall discuss later.

Now, say that we want to solve multiple equations

$$Ax_i = b_i, i = 1, 2, \ldots, k,$$

which have the same matrix A but different right hand sides $b = b_i$, $i = 1, 2, \ldots, k$. Extend this idea above to the case $k > 1$, by describing an extended $\hat{A}$ containing all the b_i vectors.

The `cla_utils.exercises3.householder_solve()` *function has been left unimplemented. It takes in a set of right hand side vectors $b_1, b_2, \ldots, b_k$ and returns a set of solutions $x_1, x_2, \ldots, x_k$. In the course of solving, it should construct an extended array $\hat{A}$, and then pass it to* `cla_utils.exercises3.householder()`*. You will also need* `cla_utils.exercises3.solve_U()`*.*

If we really need Q, we can get it by matrix-vector products with each element of the canonical basis $(e_1, e_2, \ldots, e_n)$. This means that first we need to compute a matrix-vector product Qx with a vector x. One way to do this is to apply the Householder reflections in reverse, since

$$Q = (Q_n \ldots Q_2 Q_1)^* = Q_1 Q_2 \ldots Q_n,$$

having made use of the fact that the Householder reflections are Hermitian. This can be expressed in the following pseudo-code.

```
• FOR $k = n$ TO 1 (DOWNWARDS)
    – $x_{k:m} \leftarrow x_{k:m} - 2v_k(v_k^* x_{k:m})$
• END FOR
```

Note that this requires to record all of the history of the v vectors, whilst the Q^* application algorithm above can be interlaced with the steps of the Householder algorithm, using the v values as they are needed and throwing them away. Then we can compute Q via

$$Q = \begin{pmatrix} Qe_1 & Qe_2 & \ldots & Qe_n \end{pmatrix},$$

with each column using the Q application algorithm described above.

Exercise 54 (‡) *Show that the implicit multiplication procedure applied to the columns of I produces Q^*, from which we can easily obtain Q, explaining how. Show how to implement this by applying Householder to an augmented matrix $\hat{A}$ of some appropriate form.*

The `cla_utils.exercises3.householder_qr()` *function has been left unimplemented. It takes in the $m \times n$ array A and returns Q and R. It should use the method of this exercise to compute them by forming an appropriate $\hat{A}$, calling* `cla_utils.exercises3.householder()` *and then extracting appropriate subarrays using slice notation. The test script* `test_exercises3.py` *in the* `test` *directory will also test this function.*

3.7 Application: Least squares problems

Supplementary video

https://player.vimeo.com/video/450202726

Least square problems are relevant in data fitting problems, optimisation and control, and are also a crucial ingredient of modern massively parallel linear system solver algorithms such as GMRES, which we shall encounter later in the course. They are a way of solving "long thin" matrix vector problems $Ax = b$ where we want to obtain $x \in \mathbb{C}^n$ from $b \in \mathbb{C}^m$ with A an $m \times n$ matrix. Often the problem does not have a solution as it is overdetermined for $m > n$. Instead we just seek x that minimises the 2-norm of the residual $r = b - Ax$, i.e. x is the minimiser of

$$min_x \|Ax - b\|^2.$$

This residual will not be zero in general, when b is not in the range of A. The nearest point in the range of A to b is Pb, where P is the orthogonal projector onto the range of A. From `theorem {number}`, we know that $P = \hat{Q}\hat{Q}^*$, where $\hat{Q}$ from the reduced QR factorisation has the same column space as A (but with orthogonal columns).

Then, we just have to solve

$$Ax = Pb,$$

which is now solveable since Pb is in the column space of A (and hence can be written as a linear combination of the columns of A i.e. as a matrix-vector product Ax for some unknown x).

Now we have the reduced QR factorisation of A, and we can write

$$\hat{Q}\hat{R}x = \hat{Q}\hat{Q}^*b.$$

Left multiplication by $\hat{Q}^*$ then gives

$$\hat{R}x = \hat{Q}^*b.$$

This is an upper triangular system that can be solved efficiently using back-substitution.

Exercise 55 (‡) *The* `cla_utils.exercises3.householder_ls()` *function has been left unimplemented. It takes in the* $m \times n$ *array* A *and a right-hand side vector* b *and solves the least squares problem minimising* $\|Ax - b\|$ *over* x. *It should do this by forming an appropriate augmented matrix* $\hat{A}$, *calling* `cla_utils.exercises3.householder()` *and extracting appropriate subarrays using slice notation, before using* `cla_utils.exercises3.solve_U()` *to solve the resulting upper triangular system, before returning the solution* x. *The test script* `test_exercises3.py` *in the* `test` *directory will also test this function.*

Hint

You will need to do extract the appropriate submatrix to obtain the square (and invertible) reduced matrix $\hat{R}$.

CHAPTER

FOUR

ANALYSING ALGORITHMS

In the previous section we saw three algorithms to compute the QR factorisation of a matrix. They have a beautiful mathematical structure based on orthogonal projectors. But are they useful? To answer this we need to know:

1. Is one faster than others?
2. Is one more sensitive than others to small perturbations due to early truncation of the algorithm or due to round-off errors?

In this course we will characterise answers to the first question by operation count (acknowledging that this is an incomplete evaluation of speed), and answers to the second question by analysing stability.

In this section we will discuss both of these questions by introducing some general concepts but also looking at the examples of the QR algorithms that we have seen so far.

4.1 Operation count

Supplementary video

https://player.vimeo.com/video/450203625

Operation count is one aspect of evaluating how long algorithms take. Here we just note that this is not the only aspect, since transferring data between different levels of memory on chips can be a serious (and often dominant) consideration, even more so when we consider algorithms that make use of large numbers of processors running in parallel. However, operation count is what we shall focus on here.

In this course, a floating point operation (FLOP) will be any arithmetic unary or binary operation acting on single numbers (such as $+$, $-$, $\times$, $\div$, $\sqrt{}$). Of course, in reality, these different operations have different relative costs, and codes can be made more efficient by blending multiplications and additions (fused multiply-adds) for example. Here we shall simply apologise to computer scientists in the class, and proceed with this interpretation, since we are just making relative comparisons between schemes. We shall also concentrate on asymptotic results in the limit of large n and/or m.

4.2 Operation count for modified Gram-Schmidt

We shall discuss operation counts through the example of the modified Gram-Schmidt algorithm. We shall find that the operation count is $\sim 2mn^2$ to compute the QR factorisation, where the $\sim$ symbol means

$$\lim_{m,n\to\infty} \frac{N_{\text{FLOPS}}}{2mn^2} = 1.$$

To get this result, we return to the pseudocode for the modified Gram-Schmidt algorithm, and concentrate on the operations that are happening inside the inner j loop. Inside that loop there are two operations,

1. $r_{ij} \leftarrow q_i^* v_i$. This is the inner product of two vectors in $\mathbb{R}^m$, which requires m multiplications and $m-1$ additions, so we count $2m-1$ FLOPS per inner iteration.

2. $v_j \leftarrow v_j - r_{ij}q_i$. This requires m multiplications and m subtractions, so we count $2m$ FLOPS per inner iteration.

At each iteration we require a combined operation count of $\sim 4m$ FLOPS. There are n outer iterations over i, and $n-i-1$ inner iterations over j, which we can estimate by approximating the sum as an integral,

$$N_{\text{FLOPS}} \sim \sum_{i=1}^{n}\sum_{j=i+1}^{n} 4m \sim 4m\int_0^n\int_x^n dx'\,dx = 4m\frac{n^2}{2} = 2mn^2,$$

as suggested above.

4.3 Operation count for Householder

In the Householder algorithm, the computation is dominated by the transformation

$$A_{k:m,k:n} \leftarrow A_{k:m,k:n} - \underbrace{2v_k\underbrace{(v_k^*A_{k:m,k:n})}_{1}}_{2},$$

which must be done for each k iteration. To evaluate the part marked 1 requires $n-k$ inner products of vectors in $\mathbb{C}^{m-k}$, at a total cost of $\sim 2(n-k)(m-k)$ (we already examined inner products in the previous example). To evaluate the part marked 2 then requires the outer product of two vectors in $\mathbb{C}^{m-k}$ and $\mathbb{C}^{n-k}$ respectively, at a total cost of $(m-k)(n-k)$ FLOPs. Finally two $(m-k)\times(n-k)$ matrices are substracted, at cost $(m-k)(n-k)$. Putting all this together gives $\sim 4(n-k)(m-k)$ FLOPs per k iteration.

Now we have to sum this over k, so the total operation count is

$$\begin{aligned} 4\sum_{k=1}^{n}(n-k)(m-k) &= 4\sum_{k=1}^{n}(nm - k(n+m) + k^2) \\ &\sim 4n^2m - 4(n+m)\frac{n^2}{2} + 4\frac{n^3}{3} = 2mn^2 - \frac{2n^3}{3}. \end{aligned}$$

Exercise 56 *Compute FLOP counts for the following operations.*

1. $\alpha = x^*y$ *for* $x, y \in \mathbb{C}^m$.
2. $y = y + ax$ *for* $x, y \in \mathbb{C}^m$, $a \in \mathbb{C}$.
3. $y = y + Ax$ *for* $x \in \mathbb{C}^n$, $y \in \mathbb{C}^m$, $A \in \mathbb{C}^{m\times n}$.
4. $C = C + AB$ *for* $A \in \mathbb{C}^{m\times r}$, $B \in \mathbb{C}^{r\times n}$, $C \in \mathbb{C}^{m\times n}$.

Exercise 57 *Suppose $D = ABC$ where $A \in \mathbb{C}^{m\times n}$, $B \in \mathbb{C}^{n\times p}$, $C \in \mathbb{C}^{p\times q}$. This can either be computed as $D = (AB)C$ (multiply A and B first, then C), or $D = A(BC)$ (multiply B and C first, then A). Compute the FLOP count for both approaches. For which values of m, n, p, q would the first approach be more efficient?*

Exercise 58 *Suppose $W \in \mathbb{C}^{n\times n}$ is defined by*

$$w_{ij} = \sum_{q=1}^{n}\sum_{p=1}^{n} x_{ip}y_{pq}z_{qj},$$

where $X, Y, Z \in \mathbb{C}^{n\times n}$. What is the FLOP count for computing the entries of W?

The equivalent formula

$$w_{ij} = \sum_{p=1}^{n} x_{ip}\left(\sum_{q=1}^{n} y_{pq}z_{qj}\right),$$

computes the bracket contents first for all p, j, before doing the sum over p. What is the FLOP count for this alternative method of computing the entries of W?

Using what you have learned, propose an $\mathcal{O}(n^3)$ procedure for computing $A \in \mathbb{C}^{n\times n}$ with entries

$$a_{ij} = \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{m=1}^{n} E_{ki}F_{ki}G_{lk}H_{lm}F_{lm}G_{mj}.$$

Exercise 59 *Let $L_1, L_2 \in \mathbb{C}^{m\times m}$ be lower triangular matrices. If we apply the usual formula for multiplying matrices, we will waste computation time by multiplying numbers by zero and then adding the result to other numbers. Describe a more efficient algorithm as pseudo-code and compute the FLOP count, comparing with the FLOP count for the standard algorithm.*

4.4 Matrix norms for discussing stability

Supplementary video

https://player.vimeo.com/video/450204495

In the rest of this section we will discuss another important aspect of analysing computational linear algebra algorithms, stability. To do this we need to introduce some norms for matrices in addition to the norms for vectors that we discussed in Section 1.

If we ignore their multiplication properties, matrices in $\mathbb{C}^{m\times n}$ can be added and scalar multiplied, hence we can view them as a vector space, in which we can define norms, just as we did for vectors.

One type of norm arises from simply treating the matrix entries as entries of a vector and evaluating the 2-norm.

Definition 60 (Frobenius norm) *The Frobenius norm is the matrix version of the 2-norm, defined as*

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}^2}.$$

(Exercise: show that $\|AB\|_F \leq \|A\|_F\|B\|_F$.)

Another type of norm measures the maximum amount of stretching the matrix can cause when multiplying a vector.

Definition 61 (Induced matrix norm) *Given an $m\times n$ matrix A and any chosen vector norms $\|\cdot\|_{(n)}$ and $\|\cdot\|_{(m)}$ on $\mathbb{C}^n$ and $\mathbb{C}^m$, respectively, the induced norm on A is*

$$\|A\|_{(m,n)} = \sup_{x\in\mathbb{C}^n, x\neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}}.$$

Directly from the definition we can show

$$\frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} \leq \sup_{x\in\mathbb{C}^n, x\neq 0} \frac{\|Ax\|_{(m)}}{\|x\|_{(n)}} = \|A\|_{(m,n)},$$

and hence $\|Ax\| \leq \|A\|\|x\|$ whenever we use an induced matrix norm.

Exercise 62 *We can reformulate the induced definition as a constrained optimisation problem*

$$\|A\|_{(m,n)} = \sqrt{\sup_{x\in\mathbb{C}^n, \|x\|^2=1} \|Ax\|_{(m)}^2}.$$

Introduce a Lagrange multiplier $\lambda \in \mathbb{C}$ to enforce the constraint $\|x\|^2 = 1$. Consider the case above where the norms on $\mathbb{C}^m$ and $\mathbb{C}^n$ are both 2-norms. Show that λ must be an eigenvalue of some matrix (which you should compute). Hence, given those eigenvalues, provide an expression for the operator norm of A.

4.5 Norm inequalities

Often it is difficult to find exact values for norms, so we compute upper bounds using inequalities instead. Here are a few useful inequalities.

Definition 63 (Hölder inequality) *Let $x, y \in \mathbb{C}^m$, and $p, q \in \mathbb{R}+$ such that $\frac{1}{p} + \frac{1}{q} = 1$. Then*

$$|x^* y| \leq \|x\|_p \|y\|_q.$$

In the case $p = q = 2$ this becomes the Cauchy-Schwartz inequality.

Definition 64 (Cauchy-Schwartz inequality) *Let $x, y \in \mathbb{C}^m$. Then*

$$|x^* y| \leq \|x\|_2 \|y\|_2.$$

For example, we can use this to bound the operator norm of the outer product $A = uv^*$ of two vectors.

$$\|Ax\|_2 = \|uv^* x\|_2 = \|u(v^* x)\|_2 = |v^* x| \|u\|_2 \leq \|u\|_2 \|v\|_2 \|x\|_2,$$

so $\|A\|_2 \leq \|u\|_2 \|v\|_2$.

We can also compute bounds for $\|AB\|_2$.

Theorem 65 *Let $A \in \mathbb{C}^{l \times m}$, $B \in \mathbb{C}^{m \times n}$. Then*

$$\|AB|_{(l,n)} \leq \|A\|_{(l,m)} \|B\|_{(m,n)}.$$

Proof 66

$$\|ABx\|_{(l)} \leq \|A\|_{(l,m)} \|Bx\|_{(m)} \leq \|A\|_{(l,m)} \|B\|_{(m,n)} \|x\|_{(n)},$$

so

$$\|AB\|_{(l,n)} = \sup_{x \neq 0} \frac{\|ABx\|_{(l)}}{\|x\|_{(n)}} \leq \|A\|_{(l,m)} \|B\|_{(m,n)},$$

as required.

4.6 Condition number

Supplementary video

https://player.vimeo.com/video/450205296

The key tool to understanding numerical stability of computational linear algebra algorithms is the condition number. The condition number is a very general concept that measures the behaviour of a mathematical problem under perturbations. Here we think of a mathematical problem as a function $f : X \to Y$, where X and Y are normed vector spaces (further generalisations are possible). It is often the case that f has different properties under perturbation for different values of $x \in X$.

Definition 67 (Well conditioned and ill conditioned.) *We say that a problem is well conditioned (at x) if small changes in x lead to small changes in $f(x)$. We say that a problem is ill conditioned if small changes in x lead to large changes in $f(x)$.*

These changes are measured by the condition number.

Definition 68 (Absolute condition number.) *Let δx be a perturbation so that $x \mapsto x + \delta x$. The corresponding change in $f(x)$ is $\delta f(x)$,*

$$\delta f(x) = f(x + \delta x) - f(x).$$

The absolute condition number of f at x is

$$\hat{\kappa} = \sup_{\delta x \neq 0} \frac{\|\delta f\|}{\|\delta x\|},$$

i.e. the maximum that f can change relative to the size of the perturbation δx.

It is easier to consider linearised perturbations, defining a Jacobian matrix $J(x)$ such that

$$J(x)\delta x = \lim_{\epsilon \to 0} \frac{f(x + \epsilon \delta x) - f(x)}{\epsilon}$$

and then the linear absolute condition number is

$$\hat{\kappa} = \sup_{\delta x \neq 0} \frac{\|J(x)\delta x\|}{\|\delta x\|} = \|J(x)\|,$$

which is the operator norm of $J(x)$.

This definition could be improved by measuring this change relative to the size of f itself.

Definition 69 (Relative condition number.) *The relative condition number of a problem f measures the changes δx and δf relative to the sizes of x and f.*

$$\kappa = \sup_{\delta \neq 0} \frac{\|\delta f\|/\|f\|}{\|\delta x\|/\|x\|}.$$

The linear relative condition number is

$$\kappa = \frac{\|J\|/\|f\|}{1/\|x\|} = \frac{\|J\|\|x\|}{\|f\|}.$$

Since we use floating point numbers on computers, it makes more sense to consider relative condition numbers in computational linear algebra, and from here on we will always use them whenever we mention condition numbers. If κ is small ($1 - 100$, say) then we say that a problem is well conditioned. If κ is large ($> 10^6$, say), then we say that a problem is ill conditioned.

Supplementary video

https://player.vimeo.com/video/450211558

As a first example, consider the problem of finding the square root, $f : x \mapsto \sqrt{x}$, a one dimensional problem. In this case, $J = x^{-1/2}/2$. The (linear) condition number is

$$\kappa = \frac{|x^{-1/2}/2||x|}{|x^{1/2}|} = 1/2.$$

Hence, the problem is well-conditioned.

As a second example, consider the problem of finding the roots of a polynomial, given its coefficients. Specifically, we consider the polynomial $x^2 - 2x + 1 = (x - 1)^2$, which has two roots equal to 1. Here we consider the change in roots relative to the coefficient of x^0 (which is 1). Making a small perturbation to the polynomial, $x^2 - 2x + 0.9999 = (x - 0.99)(x - 1.01)$, so a relative change of 10^{-4} gives a relative change of 10^{-2} in the roots. Using the general formula

$$r = 1 \pm \sqrt{1 - c} = 1 \pm \sqrt{\delta c} \implies \delta r = \pm\sqrt{\delta c},$$

where r returns the two roots with perturbations δr and c is the coefficient of x^0 with perturbatino δc. is the perturbation to the coefficient of x^0 (so 1 becomes $1 + \delta c$). The (nonlinear) condition number is then the sup over $\delta c \neq 0$ of

$$\frac{|\delta r|/|r|}{|\delta c|/|c|} = \frac{|\delta r|}{|\delta c|} = \frac{|\delta c|^{1/2}}{|\delta c|} = |\delta c|^{-1/2} \to \infty \text{ as } \delta c \to 0,$$

so the condition number is unbounded and the problem is catastrophically ill conditioned. For an even more vivid example, see the conditioning of the roots of the Wilkinson polynomial.

4.7 Conditioning of linear algebra computations

Supplementary video

https://player.vimeo.com/video/450211706

We now look at the condition number of problems from linear algebra. The first problem we examine is the problem of matrix-vector multiplication, i.e. for a fixed matrix $A \in \mathbb{C}^{m\times n}$, the problem is to find Ax given x. The problem is linear, with $J = A$, so the condition number is

$$\kappa = \frac{\|A\|\|x\|}{\|Ax\|}.$$

When A is non singular, we can write $x = A^{-1}Ax$, and

$$\|x\| = \|A^{-1}Ax\| \leq \|A^{-1}\|\|Ax\|,$$

so

$$\kappa \leq \frac{\|A\|\|A^{-1}\|\|Ax\|}{\|Ax\|} = \|A\|\|A^{-1}\|.$$

We call this upper bound the condition number $\kappa(A)$ of the matrix A.

Supplementary video

https://player.vimeo.com/video/450212408

The next problem we consider is the condition number of solving $Ax = b$, with b fixed but considering perturbations to A. So, we have $f : A \mapsto x$. The condition number of this problem measures how small changes δA to A translate to changes δx to x. The perturbed problem is

$$(A + \delta A)(x + \delta x) = b,$$

which simplifies (using $Ax = b$) to

$$\delta A(x + \delta x) + A\delta x = 0,$$

which is independent of b. If we are considering the linear condition number, we can drop the nonlinear term, and we get

$$\delta A x + A\delta x = 0, \implies \delta x = -A^{-1}\delta A x,$$

from which we may compute the bound

$$\|\delta x\| \leq \|A^{-1}\|\|\delta A\|\|x\|.$$

Then, we can compute the condition number

$$\kappa = \sup_{\|\delta A\|\neq 0} \frac{\|\delta x\|/\|x\|}{\|\delta A\|/\|A\|}. \leq \sup_{\|\delta A\|\neq 0} \frac{\|A^{-1}\|\|\delta A\|\|x\|/\|x\|}{\|\delta A\|/\|A\|}. = \|A^{-1}\|\|A\| = \kappa(A),$$

having used the bound for δx. Hence the bound on the condition number for this problem is the condition number of A.

4.8 Floating point numbers and arithmetic

Supplementary video

https://player.vimeo.com/video/450212648

Floating point number systems on computers use a discrete and finite representation of the real numbers. One of the first things we can deduce from this fact is that there exists a largest and a smallest positive number. In "double precision", the standard floating point number format for scientific computing these days, the largest number is $N_{\max} \approx 1.79 \times 10^{308}$, and the smallest number is $N_{\min} \approx 2.23 \times 10^{-308}$. The second thing that we can deduce is that there must be gaps between adjacent numbers in the number system. In the double precision format, the interval $[1, 2]$ is subdivided as $(1, 1+2^{-52}, 1+2\times 2^{-52}, 1+3\times 2^{-52}, \ldots, 2)$. The next interval $[2, 4]$ is subdivided as $(2, 2 + 2^{-51}, 2 + 2 \times 2^{-51}, \ldots, 4)$. In general, the interval $[2^j, 2^{j+1}]$ is subdivided by multiplying the set subdividing $[1, 2]$ by 2^j. In this representation, the gaps between numbers scale with the number size. We call this set of numbers the (double precision) floating point numbers $\mathbb{F} \subset \mathbb{R}$.

A key aspect of a floating point number system is "machine epsilon" (ε), which measures the largest relative distance between two numbers. Considering the description above, we see that ε is the the distance between 1 and the adjacent number, i.e.

$$\varepsilon = 2^{-53} \approx 1.11 \times 10^{-16}.$$

ε defines the accuracy with which arbitrary real numbers (within the range of the maximum magnitude above) can be approximated in $\mathbb{F}$.

$$\forall x \in \mathbb{R}, \exists x' \in \mathbb{F} \text{ such that } |x - x'| \leq \varepsilon|x|.$$

Supplementary video

https://player.vimeo.com/video/450213018

Definition 70 (Floating point rounding function) *We define $f_L : \mathbb{R} \to \mathbb{F}$ as the function that rounds $x \in \mathbb{R}$ to the nearest floating point number.*

The following axiom is just a formal presentation of the properties of floating point numbers that we discussed below.

Definition 71 (Floating point axiom I)

$$\forall x \in \mathbb{R},\ \exists \epsilon' \textit{ with } |\epsilon'| \leq \varepsilon,$$
$$\textit{such that } f_L(x) = x(1+\epsilon').$$

The arithmetic operations $+, -, \times, \div$ on $\mathbb{R}$ have analogous operations $\oplus, \ominus, \otimes$, etc. In general, binary operators $\odot$ (as a general symbol representing the floating point version of a real arithmetic operator $\cdot$ which could be any of the above) are constructed such that

$$x \odot y = f_L(x \cdot y),$$

for $x, y \in \mathbb{F}$, with $\cdot$ being one of $+, -, \times, \div$.

Definition 72 (Floating point axiom II)

$$\forall x, y \in \mathbb{F}, \exists \epsilon' \textit{ with } |\epsilon'| \leq \varepsilon, \textit{ such that}$$
$$x \odot y = (x \cdot y)(1+\epsilon').$$

Exercise 73 *The formula for the roots of a quadratic equation* $x^2 - 2px - q = 0$ *is well-known,*

$$x = p \pm \sqrt{p^2+q}.$$

Show that the smallest root (with the minus sign above) also satisfies

$$x = -\frac{q}{p+\sqrt{p^2+q}}.$$

In the case $p = 12345678$ *and* $q = 1$, *compare the result of these two methods for computing the smallest root when using double floating point arithmetic (the default floating point numbers in Python/NumPy). Which is more accurate? Why is this?*

4.9 Stability

Supplementary video

https://player.vimeo.com/video/450213263

Stability describes the perturbation behaviour of a numerical algorithm when used to solve a problem on a computer. Now we have two problems $f : X \to Y$ (the original problem implemented in the real numbers), and $\tilde{f} : X \to Y$ (the modified problem where floating point numbers are used at each step).

Given a problem f (such as computing the QR factorisation), we are given:

1. A floating point system $\mathbb{F}$,
2. An algorithm for computing f,
3. A floating point implementation $\tilde{f}$ for f.

Then the chosen $x \in X$ is rounded to $x' = f_L(x)$, and supplied to the floating point implementation of the algorithm to obtain $\tilde{f}(x) \in Y$.

Now we want to compare $f(x)$ with $\tilde{f}(x)$. We can measure the absolute error

$$\|\tilde{f}(x) - f(x)\|,$$

or the relative error (taking into account the size of f),

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|}.$$

An aspiration (but an unrealistic one) would be to aim for an algorithm to accurate to machine precision, i.e.

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \mathcal{O}(\varepsilon),$$

by which we mean that $\exists C > 0$ such that

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} \leq C\varepsilon,$$

for sufficiently small ε (assuming, albeit unrealistically, that we have a sequence of computers with smaller and smaller ε). We shall see below that we have to lower our aspirations depending on the condition number of A.

Definition 74 (Stability) *An algorithm $\tilde{f}$ for f is stable if for each $x \in X$, there exists $\tilde{x}$ with*

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = \mathcal{O}(\varepsilon),$$

$$and$$

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon).$$

We say that a stable algorithm gives nearly the right answer to nearly the right question.

Supplementary video

https://player.vimeo.com/video/450213664

Supplementary video

https://player.vimeo.com/video/454094432

Definition 75 (Backward stability) *An algorithm $\tilde{f}$ for f is backward stable if for each $x \in X$, $\exists\tilde{x}$ such that*

$$\tilde{f}(x) = f(\tilde{x}), \text{ with } \frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\varepsilon).$$

A backward stable algorithm gives exactly the right answer to nearly the right question. The following result shows what accuracy we can expect from a backward stable algorithm, which involves the condition number of f.

Theorem 76 (Accuracy of a backward stable algorithm) *Suppose that a backward stable algorithm is applied to solve problem $f : X \to Y$ with condition number κ using a floating point number system satisfying the floating point axioms I and II. Then the relative error satisfies*

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = \mathcal{O}(\kappa(x)\epsilon).$$

Proof 77 *Since $\tilde{f}$ is backward stable, we have $\tilde{x}$ with $\tilde{f}(x) = f(\tilde{x})$ and $\|\tilde{x} - x\|/\|x\| = \mathcal{O}(\varepsilon)$ as above. Then,*

$$\begin{aligned}\frac{\|\tilde{f}(x)-f(x)\|}{\|f(x)\|} &= \frac{\|f(\tilde{x})-f(x)\|}{\|f(x)\|}, \\ &= \underbrace{\frac{\|f(\tilde{x})-f(x)\|}{\|f(x)\|}\frac{\|x\|}{\|\tilde{x}-x\|}}_{=\kappa}\underbrace{\frac{\|\tilde{x}-x\|}{\|x\|}}_{=\mathcal{O}(\epsilon)},\end{aligned}$$

as required.

This type of calculation is known as backward error analysis, originally introduced by Jim Wilkinson to analyse the accuracy of eigenvalue calculations using the PILOT ACE, one of the early computers build at the National Physical Laboratory in the late 1940s and early 1950s. In backward error analysis we investigate the accuracy via conditioning and stability. This is usually much easier than forward analysis, where one would simply try to keep a running tally of errors committed during each step of the algorithm.

4.10 Backward stability of the Householder algorithm

Supplementary video

https://player.vimeo.com/video/450214127

We now consider the example of the problem of finding the QR factorisation of a matrix A, implemented in floating point arithmetic using the Householder method. The input is A, and the exact output is Q, R, whilst the floating point algorithm output is $\tilde{Q}$, $\tilde{R}$. Here, we consider $\tilde{Q}$ as the exact unitary matrix produced by composing Householder rotations made by the floating point vectors $\tilde{v}_k$ that approximate the v_k vectors in the exact arithmetic Householder algorithm.

For this problem, backwards stability means that there exists a perturbed input $A+\delta A$, with $\|\delta A\|/\|A\| = \mathcal{O}(\varepsilon)$, such that $\tilde{Q}$, $\tilde{R}$ are exact solutions to the problem, i.e. $\tilde{Q}\tilde{R} = A+\delta A$. This means that there is very small backward error,

$$\frac{\|A-\tilde{Q}\tilde{R}\|}{\|A\|} = \mathcal{O}(\varepsilon).$$

It turns out that the Householder method is backwards stable.

Theorem 78 *Let the QR factorisation be computed for A using a floating point implementation of the Householder algorithm. This factorisation is backwards stable, i.e. the result $\tilde{Q}\tilde{R}$ satisfy*

$$\tilde{Q}\tilde{R} = A+\delta A, \quad \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\varepsilon).$$

Proof 79 *See the textbook by Trefethen and Bau, Lecture 16.*

Exercise 80 *Use your Householder implementation to generate random Q_1 and R_1 matrices of dimension m. It is very important that the two matrices Q_1 and R_1 are uncorrelated. In particular, computing them as the QR factorisation of the same matrix would spoil the experiment). So what you need to do is to randomly generate two independent $m\times m$ matrices A_1 and A_2. Then Q_1 should come from the QR factorisation of A_1, and R_2 should come from the QR factorisation of A_2.*

Print out the value of $\|Q_2-Q_1\|$, $\|R_2-R_1\|$, $\|A-Q_2R_2\|$. Explain what you see using what you know about the stability of the Householder algorithm.

4.11 Backward stability for solving a linear system using QR

Supplementary video

https://player.vimeo.com/video/450214601

The QR factorisation provides a method for solving systems of equations $Ax = b$ for x given b, where A is an invertible matrix. Substituting $A = QR$ and then left-multiplying by Q^* gives

$$Rx = Q^* b = y.$$

The solution of this equation is $x = R^{-1}y$, but if there is one message to take home from this course, it is that you should *never* form the inverse of a matrix. It is especially disasterous to use Kramer's rule, which the m dimensional extension of the formula for the inverse of 2×2 matrices that you learned at school. Kramer's rule has an operation count scaling like $\mathcal{O}(m!)$ and is numerically unstable. Hence it is so disasterous that we won't even show the formula for Kramer's rule here.

There are some better algorithms for finding the inverse of a matrix if you really need it, but in almost every situation it is better to *solve* a matrix system rather than forming the inverse of the matrix and multiplying it. As we have already seen, it is particularly easy to solve an equation formed from an upper triangular matrix. We recall the backward substitution algorithm below.

```
• $x_m \leftarrow y_m/R_{mm}$
• FOR $i = m - 1$ TO 1 (BACKWARDS)
    – $x_i \leftarrow (y_i - \sum_{k=i+1}^m R_{ik}x_k)/R_{ii}$
```

In each iteration, there are $m - i - 1$ multiplications and subtractions plus a division, so the total operation count is $\sim m^2$ FLOPs.

In comparison, the least bad way to form the inverse Z of R is to write $RZ = I$. Then, the k-th column of this equation is

$$Rz_k = e_k,$$

where z_k is the kth column of Z. Solving for each column independently using back substitution leads to an operation count of $\sim m^3$ FLOPs, much slower than applying back substitution directly to b. Hopefully this should convince you to always seek an alternative to forming the inverse of a matrix.

There are then three steps to solving $Ax = b$ using QR factorisation.

1. Find the QR factorisation of A (here we shall use the Householder algorithm).
2. Set $y = Q^* b$ (using the implicit multiplication algorithm).
3. Solve $Rx = y$ (using back substitution).

So our f here is the solution of $Ax = b$ given b and A, and our $\tilde{f}$ is the composition of the three algorithms above. Now we ask: "Is this composition of algorithms stable?"

We already know that the Householder algorithm is stable, and a floating point implementation produces $\tilde{Q}, \tilde{R}$ such that $\tilde{Q}\tilde{R} = A + \delta A$ with $\|\delta A\|/\|A\| = \mathcal{O}(\varepsilon)$. It turns out that the implicit multiplication algorithm is also backwards stable, for similar reasons (as it is applying the same Householder reflections). This means that given $\tilde{Q}$ (we have already perturbed Q when forming it using Householder) and b, the floating point implementation gives $\tilde{y}$ which is not exactly equal to $\tilde{Q}^* b$, but instead satisfies

$$\tilde{y} = (\tilde{Q} + \delta Q)^* b \implies (\tilde{Q} + \delta Q)\tilde{y} = b,$$

for some perturbation δQ with $\|\delta Q\| = \mathcal{O}(\varepsilon)$ (note that $\|Q\| = 1$ because it is unitary). Note that here, we are treating b as fixed and considering the backwards stability under perturbations to $\tilde{Q}$.

Finally, it can be shown (see Lecture 17 of Trefethen and Bau for a proof) that the backward substitution algorithm is backward stable. This means that given $\tilde{y}$ and $\tilde{R}$, the floating point implementation of backward substitution produces $\tilde{x}$ such that

$$(\tilde{R} + \delta\tilde{R})\tilde{x} = \tilde{y},$$

for some upper triangular perturbation such that $\|\delta\tilde{R}\|/\|\tilde{R}\| = \mathcal{O}(\varepsilon)$.

Using the individual backward stability of these three algorithms, we show the following result.

Theorem 81 *The QR algorithm to solve $Ax = b$ is backward stable, producing a solution $\tilde{x}$ such that*

$$(A + \Delta A)\tilde{x} = b,$$

for some $\|\Delta A\|/\|A\| = \mathcal{O}(\varepsilon)$.

Proof 82 *From backward stability for the calculation of Q^*b, we have*

$$\begin{aligned} b &= (\tilde{Q} + \delta Q)\tilde{y}, \\ &= (\tilde{Q} + \delta Q)(\tilde{R} + \delta R)\tilde{x}, \end{aligned}$$

having substituted the backward stability formula for back substitution in the second line. Multiplying out the brackets and using backward stability for the Householder method gives

$$\begin{aligned} b &= (\tilde{Q}\tilde{R} + (\delta Q)\tilde{R} + \tilde{Q}\delta R + (\delta Q)\delta R)\tilde{x}, \\ &= (A + \underbrace{\delta A + (\delta Q)\tilde{R} + \tilde{Q}\delta R + (\delta Q)\delta R}_{=\Delta A}\tilde{x}). \end{aligned}$$

This defines ΔA and it remains to estimate each of these terms. We immediately have $\|\delta A\| = \mathcal{O}(\varepsilon)$ from backward stability of the Householder method.

Next we estimate the second term. Using $A + \delta A = \tilde{Q}\tilde{R}$, we have

$$\tilde{R} = \tilde{Q}^*(A + \delta A),$$

we have

$$\frac{\|\tilde{R}\|}{\|A\|} \leq \|\tilde{Q}^*\|\frac{\|A + \delta A\|}{\|A\|} = \mathcal{O}(1), \text{ as } \varepsilon \to 0.$$

Then we have

$$\frac{\|(\delta Q)\tilde{R}\|}{\|A\|} \leq \|\delta Q\|\frac{\|\tilde{R}\|}{\|A\|} = \mathcal{O}(\varepsilon).$$

To estimate the third term, we have

$$\frac{\|\tilde{Q}\delta R\|}{\|A\|} \leq \frac{\|\delta R\|}{\|A\|}\underbrace{\|\tilde{Q}\|}_{=1} = \underbrace{\frac{\|\delta R\|}{\|\tilde{R}\|}}_{\mathcal{O}(\varepsilon)}\underbrace{\frac{\|\tilde{R}\|}{\|A\|}}_{\mathcal{O}(1)} = \mathcal{O}(\varepsilon).$$

Finally, the fourth term has size

$$\frac{\|\delta Q \delta R\|}{\|A\|} \leq \underbrace{\|\delta Q\|}_{\mathcal{O}(\varepsilon)} \underbrace{\frac{\|\delta R\|}{\|\tilde{R}\|}}_{\mathcal{O}(\varepsilon)} \underbrace{\frac{\|\tilde{R}\|}{\|A\|}}_{\mathcal{O}(1)} = \mathcal{O}(\epsilon^2),$$

hence $\|\Delta A\|/\|A\| = \mathcal{O}(\varepsilon)$.

Supplementary video

https://player.vimeo.com/video/450215261

Corollary 83 *When solving* $Ax = b$ *using the QR factorisation procedure above, the floating point implementation produces an approximate solution* $\tilde{x}$ *with*

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\kappa(A)\varepsilon).$$

Proof 84 *From Theorem 76, using the backward stability that we just derived, we know that*

$$\frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\kappa\varepsilon),$$

where κ *is the condition number of the problem of solving* $Ax = b$*, which we have shown is bounded from above by* $\kappa(A)$*.*

CHAPTER

FIVE

FINDING EIGENVALUES OF MATRICES

Supplementary video

https://player.vimeo.com/video/454117340

We start with some preliminary terminology. A vector $x \in \mathbb{C}^m$ is an *eigenvector* of a square matrix $A \in \mathbb{C}^{m\times m}$ with *eigenvalue* λ if $Ax = \lambda x$. An eigenspace is the subspace $E_\lambda \subset \mathbb{C}^m$ containing all eigenvectors of A with eigenvalue λ.

There are a few reasons why we are interested in computing eigenvectors and eigenvalues of a matrix A.

1. Eigenvalues and eigenvectors encode information about A.
2. Eigenvalues play an important role in stability calculations in physics and engineering.
3. We can use eigenvectors to underpin the solution of linear systems involving A.
4. …

5.1 How to find eigenvalues?

The method that we first encounter in our mathematical education is to find solutions of $(A - \lambda I)x = 0$, which implies that $\det(A-\lambda I) = 0$. This gives a degree m polynomial to solve for λ, called the *characteristic polynomial*. Unfortunately, there is no general solution for polynomials of degree 5 or greater (from Galois theory). Further, the problem of finding roots of polynomials is numerically unstable. All of this means that we should avoid using polynomials finding eigenvalues. Instead, we should try to apply transformations to the matrix A to a form that means that the eigenvalues can be directly extracted.

Example 85 *Consider the $m \times m$ diagonal matrix*

$$A = \begin{pmatrix} 1 & 0 & \ldots & \ldots & 0 \\ 0 & 2 & \ldots & \ldots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ldots & \ddots & \vdots \\ 0 & 0 & \ldots & \ldots & m \end{pmatrix}.$$

The characteristic polynomial of A is

$$(\lambda - 1)(\lambda - 2)\ldots(\lambda - m),$$

and the eigenvalues are clearly $1, 2, 3, \ldots, m$. This is called the Wilkinson Polynomial. Numpy has some tools for manipulating polynomials which are useful here. When an $m \times m$ array is passed in to `numpy.poly()`, it returns an array of coefficients of the polynomial. For $m = 20$, obtain this array and then perturb the coefficients $a_k \rightarrow \tilde{a}_k = a_k(1 + 10^{-10} r_k)$ where r_k are randomly sampled normally distributed numbers with mean 0 and variance 1. `numpy.roots()` will compute the roots of this perturbed polynomial. Plot these roots as points in the *complex plane. Repeat this 100 times, superposing the root plots on the same graph. What do you observe? What does it tell you about this problem and what should we conclude about the wisdom of finding eigenvalues using characteristic polynomials?*

Supplementary video

https://player.vimeo.com/video/454118485

The eigenvalue decomposition of a matrix A finds a nonsingular matrix X and a diagonal matrix Λ such that

$$A = X\Lambda X^{-1}.$$

The diagonal entries of Λ are the eigenvalues of A. Hence, if we could find the eigenvalue decomposition of A, we could just read off the eigenvalues of A; the eigenvalue decomposition is "eigenvalue revealing". Unfortunately, it is not always easy or even possible to transform to an eigenvalue decomposition. Hence we shall look into some other eigenvalue revealing decompositions of A.

We quote the following result that explains when an eigenvalue decomposition can be found.

Theorem 86 *An $m \times m$ matrix A has an eigenvalue decomposition if and only if it is non-defective, meaning that the geometric multiplicity of each eigenvalue (the dimension of the eigenspace for that eigenvalue) is equal to the algebraic multiplicity (the number of times that the eigenvalue is repeated as a root in the characteristic polynomial* $\det(I\lambda - A) = 0$.

If the algebraic multiplicity is greater than the geometric multiplicity for any eigenvalue of A, then the matrix is defective, the eigenvectors do not span $\mathbb{C}^m$, and an eigenvalue decomposition is not possible.

This all motivates the search for other eigenvalue revealing decompositions of A.

Supplementary video

https://player.vimeo.com/video/454122744

Definition 87 (Similarity transformations) *For $X \in \mathbb{C}^{m\times m}$ a nonsingular matrix, the map $A \mapsto X^{-1}AX$ is called a similarity transformation of A. Two matrices A and B are* similar *if $B = X^{-1}AX$.*

The eigenvalue decomposition shows that (when it exists), A is similar to Λ. The following result shows that it may be useful to examine other similarity transformations.

Theorem 88 *Two similar matrices A and B have the same characteristic polynomial, eigenvalues, and geometric multiplicities.*

Proof 89 *See a linear algebra textbook.*

The goal is to find a similarity transformation such that A is transformed to a matrix B that has some simpler structure where the eigenvalues can be easily computed (with the diagonal matrix of the eigenvalue decomposition being one example).

One such transformation comes from the Schur factorisation.

Supplementary video

https://player.vimeo.com/video/454122918

Definition 90 (Schur factorisation) *A Schur factorisation of a square matrix A takes the form $A = QTQ^*$, where Q is unitary (and hence $Q^* = Q^{-1}$) and T is upper triangular.*

It turns out that, unlike the situation for the eigenvalue decomposition, the following is true.

Theorem 91 *Every square matrix has a Schur factorisation.*

This is useful, because the characteristic polynomial of an upper triangular matrix is just $\prod_{i=1}^{m}(\lambda - T_{ii})$, i.e. the eigenvalues of T are the diagonal entries $(T_{11}, T_{22}, \ldots, T_{mm})$. So, if we can compute the Schur factorisation of A, we can just read the eigenvalues from the diagonal matrices of A.

There is a special case of the Schur factorisation, called the unitary diagonalisation

Definition 92 (Unitary diagonalisation) *A unitary diagonalisation of a square matrix A takes the form $A = Q\Lambda Q^*$, where Q is unitary (and hence $Q^* = Q^{-1}$) and Λ is diagonal.*

A unitary diagonalisation is a Schur factorisation *and* an eigenvalue decomposition.

Theorem 93 *A Hermitian matrix is unitarily diagonalisable, with real Λ.*

Hence, if we have a Hermitian matrix, we can follow a Schur factorisation strategy (such as we shall develop in this section), and obtain an eigenvalue decomposition as a bonus.

5.2 Transformations to Schur factorisation

Supplementary video

https://player.vimeo.com/video/454123177

Just as for the QR factorisations, we will compute the Schur factorisation successively, with multiplication by a sequence of unitary matrices $Q_1, Q_2, \ldots$. There are two differences for the Schur factorisation. First, the matrices must be multiplied not just on the left but also on the right with the inverse, i.e.

$$A \mapsto \underbrace{Q_1^* A Q_1}_{A_1} \mapsto \underbrace{Q_2^* Q_1^* A Q_1 Q_2}_{A_2}, \ldots$$

At each stage, we have a similarity transformation,

$$A = \underbrace{Q_1 Q_2 \ldots Q_k}_{=Q} A_k \underbrace{Q_k^* \ldots Q_2^* Q_1^*}_{=Q^*},$$

i.e. A is similar to A_k. Second, the successive sequence is infinite, i.e. we will develop an iterative method that converges in the limit $k \to \infty$. We should terminate the iterative method when A_k is sufficiently close to being upper triangular (which can be measured by checking some norm on the lower triangular part of A and stopping when it is below a tolerance).

We should not be surprised by the news that the method needs to be iterative, since if the successive sequence were finite, we would have derived an explicit formula for computing the eigenvalues of the characteristic polynomial of A which is explicit in general.

In fact, there are two stages to this process. The first stage, which is finite (takes $m-1$ steps) is to use similarity transformations to upper Hessenberg form ($H_{ij} = 0$ for $i > j+1$). If A is Hermitian, then H will be tridiagonal. This stage is not essential but it makes the second, iterative, stage much faster.

5.3 Similarity transformation to upper Hessenberg form

Supplementary video

https://player.vimeo.com/video/454123306

We already know how to use a unitary matrix to set all entries to zero below the diagonal in the first column of A by left multiplication $Q_1^* A$, because this is the Householder algorithm. The problem is that we then have to right multiply by Q_1 to make it a similarity transformation, and this puts non-zero entries back in the column again. The easiest way to see this is to write $Q_1^* A Q_1 = (Q_1^*(Q_1^* A)^*)^*$. $(Q_1^* A)^*$ has zeros in the first row to the right of the first entry. Then, $Q_1^*(Q_1^* A)$ creates linear combinations of the first column with the other columns, filling the zeros in with non-zero values again. Then finally taking the adjoint doesn't help with these non-zero values. Again, we shouldn't be surprised that this is impossible, because if it was, then we would be able to build a finite procedure for computing eigenvalues of the characteristic polynomial, which is impossible in general.

Exercise 94 *The* `cla_utils.exercises8.Q1AQ1s()` *function has been left uncompleted. It should apply the Householder transformation Q_1 to the input A (without forming Q_1 of course) that transforms the first column* *of A to have zeros below the diagonal, and then apply a transformation equivalent to right multiplication by Q_1^* (again without forming Q_1). The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

Experiment with the output of this function. What happens to the first column?

A slight modification of this idea (and the reason that we can transform to upper Hessenberg form) is to use a Householder rotation Q_1^* to set all entries to zero below the *second* entry in the first column. This matrix leaves the first row unchanged, and hence right multiplication by Q_1 leaves the first column unchanged. We can create zeros using Q_1^* and Q_1 will not destroy them. This procedure is then repeated with multiplication by Q_2^*, which leaves the first two rows unchanged and puts zeros below the third entry in the second column, which are not spoiled by right multiplication by Q_2. Hence, we can transform A to a similar upper Hessenberg matrix H in $m-2$ iterations.

Supplementary video

https://player.vimeo.com/video/454123643

This reduction to Hessenberg form can be expressed in the following pseudo-code.

```
• FOR $k = 1$ TO $m - 2$
    – $x \leftarrow A_{k+1:m,k}$
    – $v_k \leftarrow \text{sign}(x_1)\|x\|_2 e_1 + x$
    – $v_k \leftarrow v_k/\|v\|_2$
    – $A_{k+1:m,k:m} \leftarrow A_{k+1:m,k:m} - 2v_k(v_k^* A_{k+1:m,k:m})$
    – $A_{1:m,k+1:m} \leftarrow A_{1:m,k+1:m} - 2(A_{1:m,k+1:m} v_k)v_k^*$
• END FOR
```

Note the similarities and differences with the Householder algorithm for computing the QR factorisation.

Exercise 95 (‡) *The* `cla_utils.exercises8.hessenberg()` *function has been left unimplemented. It should implement the algorithm above, using only one loop over k. It should work "in-place", changing the input matrix. At the left multiplication, your implementation should exploit the fact that zeros do not need to be recomputed where there are already expected to be zeros. The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

To calculate the operation count, we see that the algorithm is dominated by the two updates to A, the first of which applies a Householder reflection to rows from the left, and the second applies the same reflections to columns from the right.

The left multiplication applies a Householder reflection to the last $m - k$ rows, requiring 4 FLOPs per entry. However, these rows are zero in the first $k - 1$ columns, so we can skip these and just work on the last $m - k + 1$ entries of each of these rows.

Then, the total operation count for the left multiplication is

$$4 \times \sum_{k=1}^{m-1} (m-k)(m-k+1) \sim \frac{4}{3}m^3.$$

The right multiplication does the same operations but now there are no zeros to take advantage of, so all m entries in the each of the last $m - k$ columns need to be manipulated. With 4 FLOPs per entry, this becomes

$$4 \times \sum_{k=1}^{m-1} m(m-k) \sim \frac{10}{3} m^3 FLOPs.$$

Supplementary video

https://player.vimeo.com/video/454123926

In the Hermitian case, the Hessenberg matrix becomes tridiagonal, and these extra zeros can be exploited, leading to an operation count $\sim 4m^3/3$.

It can be shown that this transformation to a Hessenberg matrix is backwards stable, i.e. in a floating point implementation, it gives $\tilde{Q}, \tilde{H}$ such that

$$\tilde{Q}\tilde{H}\tilde{Q}^* = A + \delta A, \ \frac{\|\delta A\|}{\|A\|} = \mathcal{O}(\varepsilon),$$

for some δA.

Exercise 96 *The* `cla_utils.exercises8.hessenbergQ()` *function has been left unimplemented. It should implement the Hessenberg algorithm again (you can just copy paste the code from the previous exercise) but it should also return the matrix Q such that $QHQ^* = A$. You need to work out how to alter the algorithm to construct this. The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

Exercise 97 *The* `cla_utils.exercises8.ev()` *function has been left unimplemented. It should return the eigenvectors of A by first reducing to Hessenberg form, using the functions you have already created, and then calling* `cla_utils.exercises8.hessenberg_ev()`*, which computes the eigenvectors of upper Hessenberg matrices (do not edit this function!). The test script* `test_exercises8.py` *in the* `test` *directory will test this function.*

Supplementary video

https://player.vimeo.com/video/454124279

In the next few sections we develop the iterative part of the transformation to the upper triangular matrix T. This algorithm works for a broad class of matrices, but the explanation is much easier for the case of real symmetric matrices, which have real eigenvalues and orthogonals eigenvectors (which we shall normalise to $\|q_i\| = 1$, $i = 1, 2, \ldots, m$). The idea is that we will have already transformed to Hessenberg form, which will be tridiagonal in this case. Before describing the iterative transformation, we will discuss a few key tools in explaining how it works.

5.4 Rayleigh quotient

The first tool that we shall consider is the Rayleigh quotient. If $A \in \mathbb{C}^{m\times m}$ is a real symmetric matrix, then the Rayleigh quotient of a vector $x \in \mathbb{C}^m$ is defined as

$$r(x) = \frac{x^T A x}{x^T x}.$$

If x is an eigenvector of A, then

$$r(x) = \frac{x^T \lambda x}{x^T x} = \lambda,$$

i.e. the Rayleigh quotient gives the corresponding eigenvalue.

Supplementary video

https://player.vimeo.com/video/454124455

If x is not exactly an eigenvector of A, but is just close to one, we might hope that $r(x)$ is close to being an eigenvalue. To investigate this we will consider the Taylor series expansion of $r(x)$ about an eigenvector q_J of A. We have

$$\nabla r(x) = \frac{2}{x^T x}\left(Ax - r(x)x\right),$$

which is zero when $x = q_J$, because then $r(q_J) = \lambda_J$: eigenvectors of A are stationary points of $r(x)$! Hence, the Taylor series has vanishing first order term,

$$r(x) = r(q_J) + (x - q_J)^T \underbrace{\nabla r(q_J)}_{=0} + \mathcal{O}(\|x - q_J\|^2),$$

i.e.

$$r(x) - r(q_J) = \mathcal{O}(\|x - q_J\|^2), \quad \text{as } x \to q_J.$$

The Rayleigh quotient gives a quadratically accurate estimate to the eigenvalues of A.

Exercise 98 *Add a function to* `cla_utils.exercises8` *that investigates this property by:*

1. *Forming a Hermitian matrix A,*
2. *Finding an eigenvector v of A with eigenvalue λ (you can use* `numpy.linalg.eig()` *for this),*
3. *Choosing a perturbation vector r, and perturbation parameter $\epsilon > 0$,*
4. *Comparing the Rayleigh quotient of $v + \epsilon r$ with λ,*
5. *Plotting (on a log-log graph, use* `matplotlib.pyplot.loglog()`*) the error in estimating the eigenvalue as a function of ϵ.*

The best way to do this is to plot the computed data values as points, and then superpose a line plot of $a\epsilon^k$ for appropriate value of k and a chosen so that the line appears not to far away from the points on the same scale. This means that we can check by eye if the error is scaling with ϵ at the expected rate.

5.5 Power iteration

Supplementary video

https://player.vimeo.com/video/454124701

Power iteration is a simple method for finding the largest eigenvalue of A (in magnitude). It is based on the following idea. We expand a vector v in eigenvectors of A,

$$v = a_1 q_1 + a_2 q_2 + \ldots a_m q_m,$$

where we have ordered the eigenvalues so that $|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_m$.

Then,

$$Av = a_1 \lambda_1 q_1 + a_2 \lambda_2 q_2 + \ldots a_m \lambda_m q_m,$$

and hence, repeated applications of A gives

$$\begin{aligned} A^k v &= \underbrace{AA\ldots A}_{k \text{ times}} v \\ &= a_1 \lambda_1^k q_1 + a_2 \lambda_2^k q_2 + \ldots a_m \lambda_m^k q_m. \end{aligned}$$

If $|\lambda_1| > |\lambda_2|$, then provided that $a_1 = q_1^T v \neq 0$, the first term $a_1\lambda_1^k q_1$ rapidly becomes larger than all of the others, and so $A^k v \approx a_1 \lambda_1^k q_1$, and we can normalise to get $q_1 \approx A^k v/\|A^k v\|$. To keep the magnitude of the estimate from getting too large or small (depending on the size of λ_1 relative to 1), we can alternately apply A and normalise, which gives the power iteration. Along the way, we can use the Rayleigh quotient to see how our approximation of the eigenvalue is going.

```
• Set $v_0$ to some initial vector (hoping that $\|q_1^T v_0\| > 0$).
• FOR $k = 1, 2, \ldots$
    – $w \leftarrow Av^{k-1}$,
    – $v^k \leftarrow w/\|w\|$,
    – $\lambda^{(k)} \leftarrow (v^k)^T A v^k$.
```

Here we have used the fact that $\|v^k\| = 1$, so there is no need to divide by it in the Rayleigh quotient. We terminate the power iteration when we decide that the changes in λ indicate that the error is small. This is guided by the following result.

Theorem 99 *If $|\lambda_1| > |\lambda_2|$ and $\|q_1^T v_0\| > 0$, then after k iterations of power iteration, we have*

$$\|v^k - \pm q_1\| = \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right), \quad |\lambda^{(k)} - \lambda_1| = \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right),$$

as $k \to \infty$. At each step $\pm$ we mean that the result holds for either $+$ or $-$.

Proof 100 *We have already shown the first equation using the Taylor series, and the second equation comes by combining the Taylor series error with the Rayleigh quotient error.*

The ± feature is a bit annoying, and relates to the fact that the normalisation does not select v^k to have the direction as q_1.

Exercise 101 *The* `cla_utils.exercises9.pow_it()` *function has been left unimplemented. It should apply power iteration to a given matrix and initial vector, according to the docstring. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 102 *The functions* `cla_utils.exercises9.A3()` *and* `cla_utils.exercises9.B3()` *each return a 3x3 matrix, A_3 and B_3 respectively. Apply* `cla_utils.exercises9.pow_it()` *to each of these functions. What differences in behaviour do you observe? What is it about A_3 and B_3 that causes this?*

5.6 Inverse iteration

Supplementary video

https://player.vimeo.com/video/454124799

Inverse iteration is a modification of power iteration so that we can find eigenvalues other than λ_1. To do this, we use the fact that eigenvectors q_j of A are also eigenvectors of $(A - \mu I)^{-1}$ for any $\mu \in \mathbb{R}$ not an eigenvalue of A (otherwise $A - \mu I$ is singular). To show this, we write

$$(A - \mu I)q_j = (\lambda_j - \mu)q_j \implies (A - \mu I)^{-1} q_j = \frac{1}{\lambda_j - \mu} q_j.$$

Thus q_j is an eigenvalue of $(A-\mu I)^{-1}$ with eigenvalue $1/(\lambda_j - \mu)$. We can then apply power iteration to $(A-\mu I)^{-1}$ (which requires a matrix solve per iteration), which converges to an eigenvector q for which $1/|\lambda - \mu|$ is smallest, where λ is the corresponding eigenvalue. In other words, we will find the eigenvector of A whose eigenvalue is closest to μ.

This algorithm is called inverse iteration, which we express in pseudo-code below.

```
• $v^0 \leftarrow$ some initial vector with $\|v^0\| = 1$.
```

```
- FOR $k = 1, 2, \ldots$
    - SOLVE $(A - \mu I)w = v^{k-1}$ for $w$
    - $v^k \leftarrow w/\|w\|$
    - $\lambda^{(k)} \leftarrow (v^k)^T A v^k$
```

We can then directly extend `Theorem {number}` to the inverse iteration algorithm. We conclude that the convergence rate is not improved relative to power iteration, but now we can "dial in" to different eigenvalues by choosing μ.

Exercise 103 *The* `cla_utils.exercises9.inverse_it()` *function has been left unimplemented. It should apply inverse iteration to a given matrix and initial vector, according to the docstring. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 104 *Using the A_3 and B_3 matrices, explore the inverse iteration using different values of μ. What do you observe?*

5.7 Rayleigh quotient iteration

Supplementary video

https://player.vimeo.com/video/454303115

Since we can use the Rayleigh quotient to find an approximation of an eigenvalue, and we can use an approximation of an eigenvalue to find the nearest eigenvalue using inverse iteration, we can combine them together. The idea is to start with a vector, compute the Rayleigh quotient, use the Rayleigh quotient for μ, then do one step of inverse iteration to give an updated vector which should now be closer to an eigenvector. Then we iterate this whole process. This is called the Rayleigh quotient iteration, which we express in pseudo-code below.

```
- $v^0$ some initial vector with $\|v^0\| = 1$.
- $\lambda^{(0)} \leftarrow (v^0)^T A v^0$
- FOR $k = 1, 2, \ldots$
    - SOLVE $(A - \lambda^{(k-1)} I)w = v^{k-1}$ for $w$
    - $v^k \leftarrow w/\|w\|$
    - $\lambda^{(k)} \leftarrow (v^k)^T A v^k$
```

This dramatically improves the convergence since if $\|v^k - q_J\| = \mathcal{O}(\delta)$ for some small δ, then the Rayleigh quotient gives $|\lambda^{(k)} - q_J| = \mathcal{O}(\delta^2)$. Then, inverse iteration gives an estimate

$$\|v^{k+1} - \pm q_J\| = \mathcal{O}(|\lambda^{(k)} - \lambda_J| \|v^k - q_J\|) = \mathcal{O}(\delta^3).$$

Thus we have cubic convergence, which is super fast!

Exercise 105 *The* `cla_utils.exercises9.rq_it()` *function has been left unimplemented. It should apply inverse iteration to a given matrix and initial vector, according to the docstring. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 106 *The interfaces to* `cla_utils.exercises9.inverse_it()` *and* `cla_utils.exercises9.rq_it()` *have been designed to optionally provide the iterated values of the eigenvector and eigenvalue. For a given initial condition (and choice of μ in the case of inverse iteration), compare the convergence speeds of the eigenvectors and eigenvalues, using some example matrices of different sizes (don't forget to make them Hermitian).*

5.8 The pure QR algorithm

Supplementary video

https://player.vimeo.com/video/454124953

We now describe the QR algorithm, which will turn out to be an iterative algorithm that converges to the diagonal matrix (upper triangular matrix for the general nonsymmetric case) that A is similar to. Why this works is not at all obvious at first, and we shall explain this later. For now, here is the algorithm written as pseudo-code.

```
• $A^{(0)} \leftarrow A$
• FOR $k = 1, 2, \ldots$
    – FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)}R^{(k)} = A^{(k-1)}$ (USING QR FACTORISATION)
    – $A^{(k)} = R^{(k)}Q^{(k)}$
```

Exercise 107 (‡) *The* `cla_utils.exercises9.pure_QR()` *function has been left unimplemented. It should implement the pure QR algorithm as above, using your previous code for finding the QR factorisation using Householder transformations. You should think about avoiding unecessary allocation of new numpy arrays inside the loop. The method of testing for convergence has been left as well. Have a think about how to do this and document your implementation. The test script* `test_exercises9.py` *in the* `test` *directory will test this function.*

Exercise 108 *Investigate the behaviour of the pure QR algorithm applied to the functions provided by* `cla_utils.exercises9.get_A100()`, `cla_utils.exercises9.get_B100()`, `cla_utils.exercises9.get_C100()`, *and* `cla_utils.exercises9.get_D100()`. *You can use* `matplotlib.pyplot.pcolor()` *to visualise the entries, or compute norms of the components of the matrices below the diagonal, for example. What do you observe? How does this relate to the structure of the four matrices?*

Hint

Some of this examples will require a complex valued QR factorisation. This just requires a minor modification of your Householder QR code. If you have complex entries, then you should replace $\mathrm{sign}(x_1)$ with $-e^{i\arg(x_1)}$ which is the numerically stable choice in the complex case.

The algorithm simply finds the QR factorisation of A, swaps Q and R, and repeats. We call this algorithm the "pure" QR algorithm, since it can be accelerated with some modifications that comprise the "practical" QR algorithm that is used in practice.

We can at least see that this is computing similarity transformations since

$$A^{(k)} = R^{(k)}Q^{(k)} = (Q^{(k)})^*Q^{(k)}R^{(k)}Q^{(k)} = (Q^{(k)})^*A^{(k-1)}Q^{(k)},$$

so that $A^{(k)}$ is similar to $A^{(k-1)}$ and hence to $A^{(k-2)}$ and all the way back to A. But why does $A^{(k)}$ converge to a diagonal matrix? To see this, we have to show that the QR algorithm is equivalent to another algorithm called simultaneous iteration.

5.9 Simultaneous iteration

Supplementary video

https://player.vimeo.com/video/454125180

One problem with power iteration is that it only finds one eigenvector/eigenvalue pair at a time. Simultaneous iteration is a solution to this. The starting idea is simple: instead of working on just one vector v, we pick a set of linearly independent vectors $v_1^0, v_2^0, \ldots, v_n^0$ and repeatedly apply A to each of these vectors. After a large number applications and normalisations in the manner of the power iteration, we end up with a linear independent set $v_1^k, v_2^k, \ldots, v_n^k$, $n \leq m$. All of the vectors in this set will be very close to q_1, the eigenvector with largest magnitude of corresponding eigenvalue. We can choose v_1^k as our approximation of q_1, and project this approximation of q_1 from the rest of the vectors $v_2^k, v_3^k, \ldots v_n^k$. All the remaining vectors will be close to q_2, the eigenvector with the next largest magnitude of corresponding eigenvalue. Similarly we can choose the first one of the remaining projected vectors as an approximation of q_2 and project it again from the rest.

We can translate this idea to matrices by defining $V^{(0)}$ to be the matrix with columns given by the set of initial v s. Then after k applications of A, we have $V^{(k)} = A^k V^{(0)}$. By the column space interpretation of matrix-matrix multiplication, each column of $V^{(k)}$ is A^k multiplied by the corresponding column of $V^{(0)}$. To make the normalisation and projection process above, we could just apply the Gram-Schmidt algorithm, sequentially forming an orthonormal spanning set for the columns of $V^{(k)}$ working from left to right. However, we know that an equivalent way to do this is to form the (reduced) QR factorisation of $V^{(k)}$, $\hat{Q}^{(k)}\hat{R}^{(k)} = V^{(k)}$; the columns of $\hat{Q}^{(k)}$ give the same orthonormal spanning set. Hence, the columns of $\hat{Q}^{(k)}$ will converge to eigenvectors of A, provided that:

1. The first n eigenvalues of A are distinct in absolute value: $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$. If we want to find all of the eigenvalues $n = m$, then all the absolute values of the eigenvalues must be distinct.
2. The v vectors can be expressed as a linear sum of the first n eigenvectors $q_1, \ldots, q_n$ in a non-degenerate way. This turns out to be equivalent (we won't show it here) to the condition that $\hat{Q}^T V^{(0)}$ has an LU factorisation (where $\hat{Q}$ is the matrix whose columns are the first n eigenvectors of A).

One problem with this idea is that it is not numerically stable. The columns of $V^{(k)}$ rapidly become a very ill-conditioned basis for the spanning space of the original independent set, and the values of eigenvectors will be quickly engulfed in rounding errors. There is a simple solution to this though, which is to orthogonalise after each application of A. This is the simultaneous iteration algorithm, which we express in the following pseudo-code.

```
• TAKE A UNITARY MATRIX $\hat{Q}^{(0)}$
• FOR $k = 1, 2, \ldots$
    – $Z \leftarrow A\hat{Q}^{(k-1)}$
    – FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)}R^{(k)} = Z$ (USING QR FACTORISATION)
```

This is mathematically equivalent to the process we described above, and so it converges under the same two conditions listed above.

We can already see that this looks rather close to the QR algorithm. The following section confirms that they are in fact equivalent.

5.10 The pure QR algorithm and simultaneous iteration are equivalent

Supplementary video

https://player.vimeo.com/video/454125393

To be precise, we will show that the pure QR algorithm is equivalent to simultaneous iteration when the initial independent set is the canonical basis I, i.e. $Q^{(0)} = I$. From the above, we see that that algorithm converges provided that Q^T has an LU decomposition, where Q is the limiting unitary matrix that simultaneous iteration is converging to. To show that the two algorithms are equivalent, we append them with some auxiliary variables, which are not needed for the algorithms but are needed for the comparison.

To simultaneous iteration we append a running similarity transformation of A, and a running product of all of the R matrices.

```
• ${Q'}^{(0)} \leftarrow I$
• FOR $k = 1, 2, \ldots$
    – $Z \leftarrow A{Q'}^{(k-1)}$
    – FIND ${Q'}^{(k)}, R^{(k)}$ such that ${Q'}^{(k)}R^{(k)} = Z$ (USING QR FACTORISATION)
    – $A^{(k)} = ({Q'}^{(k)})^T A{Q'}^{(k)}$
    – ${R'}^{(k)} = R^{(k)}R^{(k-1)}\ldots R^{(1)}$
```

To the pure QR factorisation we append a running product of the Q^k matrices, and a running product of all of the R matrices (again).

```
• $A^{(0)} \leftarrow A$
• FOR $k = 1, 2, \ldots$
    – FIND $Q^{(k)}, R^{(k)}$ such that $Q^{(k)}R^{(k)} = A^{(k-1)}$ (USING QR FACTORISATION)
    – $A^{(k)} = R^{(k)}Q^{(k)}$
    – $Q'^{(k)} = Q^{(1)}Q^{(2)} \ldots Q^{(k)}$
    – $R'^{(k)} = R^{(k)}R^{(k-1)} \ldots R^{(1)}$
```

Theorem 109 (pure QR and simultaneous iteration with I are equivalent) *The two processes above generate identical sequences of matrices $R'^{(k)}$, $Q'^{(k)}$ and $A^{(k)}$, which are related by $A^k = Q'^{(k)}R'^{(k)}$ (the k-th power of A, not $A^{(k)}$!), and $A^{(k)} = (Q'^{(k)})^T AQ'^{(k)}$.*

Proof 110 *We prove by induction. At $k = 0$, $A_k = R'^{(k)} = Q'^{(k)} = I$. Now we assume that the inductive hypothesis is true for k, and aim to deduce that it is true for $k + 1$.*

For simultaneous iteration, we immediately have the simularity formula for $A^{(k)}$ by definition, and we just need to verify the QR factorisation of A^k. From the inductive hypothesis,

$$A^k = AA^{k-1} = AQ'^{(k-1)}R'^{(k-1)} = ZR'^{(k-1)} = Q'^{(k)} \underbrace{R^{(k)}R'^{(k-1)}}_{=R'^{(k)}} = Q'^{(k)}R'^{(k)},$$

as required (using the definition of Z and then the definition of $R'^{(k)}$).

For the QR algorithm, we again use the inductive hypothesis on the QR factorization of A^k followed by the inductive hypothesis on the similarity transform to get

$$A^k = AA^{k-1} = AQ'^{(k-1)}R'^{(k-1)} = Q'^{(k-1)}A^{(k-1)}R'^{(k-1)} = Q'^{(k-1)}Q^{(k)}R^{(k)}R'^{(k-1)} = Q'^{(k)}R'^{(k)},$$

where we used the algorithm definitions in the third equality and then the definitions of $Q'^{(k)}$ and $R'^{(k)}$. To verify the similarity transform at iteration k we use the algorithm definitions to write

$$A^{(k)} = R^{(k)}Q^{(k)} = (Q^{(k)})^T Q^{(k)}R^{(k)}Q^{(k)} = (Q'^{(k)})^T A(Q')^{(k)},$$

as required.

This theorem tells us that the QR algorithm will converge under the conditions that simultaneous iteration converges. It also tells us that the QR algorithm finds an orthonormal basis (the columns of $Q'^{(k)}$) from the columns of each power of A^k; this is how it relates to power iteration.

5.11 Connections between power iteration, inverse iteration, and QR algorithm

There are some subtle connections between these algorithms that we can exploit to accelerate the convergence of the QR algorithm. After k iterations we have

$$A^k = Q'^{(k)}R'^{(k)},$$

from the above theorem. (Remember that $Q'^{(k)}$ and $R'^{(k)}$, are different from $Q^{(k)}$ and $R^{(k)}$.) In particular, the first column of $R'^{(k)}$ is $e_1 r_{11}^{(k)}$ (because $R'^{(k)}$ is an upper triangular matrix), so the first column of A^k is

$$A^k e_1 = r_{11}^{(k)} Q'^{(k)} e_1.$$

In other words, the first column of ${Q'}^{(k)}$ is the result of k iterations of power iteration starting at e_1. (We already knew this from the previous theorem, but here we are introducing ways to look at different components of ${Q'}^{(k)}$ and ${R'}^{(k)}$). This means that it will converge to the eigenvector of A with the eigenvalue of largest magnitude, and the convergence rate will depend on the gap between the magnitude of that eigenvalue and the magnitude of the next largest.

To look at connections with inverse iteration, we use the inverse formula,

$$A^{-k} = \left({R'}^{(k)}\right)^{-1} \left({Q'}^{(k)}\right)^{*} := \tilde{R}^{(k)} \left({Q'}^{(k)}\right)^{*},$$

where $\tilde{R}^{(k)} = \left({R'}^{(k)}\right)^{-1}$ is also upper triangular since invertible upper triangular matrices form a group.

Then,

$$(A^{-k})^T = {Q'}^{(k)}(\tilde{R}^{(k)})^T.$$

The last column of $(\tilde{R}^{(k)})^T$ is $\tilde{r}_{nn}^{(k)} e_n$, so the last column of $(A^{-k})^T$ is

$$(A^{-k})^T e_n = {Q'}^{(k)} \tilde{r}_{nn} e_n,$$

i.e. the last column of ${Q'}^{(k)}$ corresponds to the k th step of inverse iteration applied to A^T. When A is symmetric, this is the same as inverse iteration applied to e_n.

5.12 The practical QR algorithm

Supplementary video

https://player.vimeo.com/video/454125822

The practical QR algorithm for real symmetric matrices has a number of extra elements that make it fast. First, recall that we start by transforming to tridiagonal (symmetric Hessenberg) form. This cuts down the numerical cost of the steps of the QR algorithm.

For the second element, the Rayleigh quotient algorithm idea is incorporated by applying shifts $A^{(k)} - \mu^{(k)} I$, where $\mu^{(k)}$ is some eigenvalue estimate for the smallest eigenvalues. Relying on the fact that if A is symmetric, then we know that the last diagonal in $R^{(k)}$ will be converging to the smallest eigenvalue of A via inverse iteration. Then, applying QR to the shifted matrix, the last diagonal will converge to the eigenvalue of A closest to $\mu^{(k)}$. This will happen quickly if $\mu^{(k)}$ is an accurate estimate of an eigenvalue of A.

For the third element, when an eigenvalue is found (i.e. an eigenvalue appears accurately on the diagonal of $A^{(k)}$) the off-diagonal components are very small, and the matrix decouples into a block diagonal matrix where the QR algorithm can be independently applied to the blocks (which is cheaper than doing them all together). This final idea is called deflation.

A sketch of the practical QR algorithm is as follows.

```
• $A^{(0)} \leftarrow$ TRIDIAGONAL MATRIX
• FOR $k = 1, 2, \ldots$
    – PICK A SHIFT $\mu^{(k)}$ (discussed below)
    – $Q^{(k)} R^{(k)} = A^{(k-1)} - \mu^{(k)} I$ (from QR factorisation)
    – $A^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I$
    – IF $A^{(k)}_{j,j+1} \approx 0$ FOR SOME $j$
        * $A_{j,j+1} \leftarrow 0$
        * $A_{j+1,j} \leftarrow 0$
        * continue by applying the practical QR algorithm to the diagonal blocks $A_1$ and $A_2$ of
```

$$A_k = \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix}$$

One possible way to select the shift $\mu^{(k)}$ is to calculate a Rayleigh quotient with A using the last column $q_m^{(k)}$ of ${Q'}^{(k)}$, which then gives cubic convergence for this eigenvector and eigenvalue. In fact, this is just A^k_{mm},

$$A^{(k)}_{mm} = e_m^T A^{(k)} e_m = e_m^T ({Q'}^{(k)})^T A {Q'}^{(k)} e_m = (q_m^{(k)})^T A q_m = \mu^{(k)}.$$

This is very cheap, we just read off the bottom right-hand corner from $A^{(k)}$! This is called the Rayleigh quotient shift.

It turns out that the Rayleigh quotient shift is not guaranteed to work in all cases, so there is an alternative approach called the Wilkinson shift, but we won't discuss that here.

CHAPTER

SIX

ITERATIVE KRYLOV METHODS FOR $Ax = b$

Supplementary video

https://player.vimeo.com/video/454126320

In the previous section we saw how iterative methods are necessary (but can also be fast) for eigenvalue problems $Ax = \lambda x$. Iterative methods can also be useful for solving linear systems $Ax = b$, generating a sequence of vectors x^k that converge to the solution. We shall examine Krylov subspace methods, where each iteration mainly involves a matrix-vector multiplication. For dense matrices, matrix-vector multiplication costs $\mathcal{O}(m^2)$, but often (e.g. numerical solution of PDEs, graph problems, etc.) A is sparse (i.e. has zeros almost everywhere) and the matrix-vector multiplication costs $\mathcal{O}(m)$. The goal is then to find a method where only a few iterations are necessary before the error is very small, so that the solver has total cost $\mathcal{O}(mN)$ where N is the total number of iterations, hopefully small.

Since we only need the result of a matrix-vector multiplication, it is even possible to solve a linear system without storing A explicitly. Instead one can just provide a subroutine that implements matrix-vector multiplication in some way; this is called a "matrix-free" implementation.

6.1 Krylov subspace methods

Supplementary video

https://player.vimeo.com/video/454126582

In this section we will introduce Krylov subspace methods for solving $Ax = b$ (we will not specialise to real or symmetric matrices here). The idea is to approximate the solution using the basis

$$(b, Ab, A^2b, A^3b, \ldots, A^kb)$$

whose span is called a Krylov subspace. After each iteration the Krylov subspace grows by one dimension. As we have already seen from studying power iteration, the later elements in this sequence will get very parallel (they will all be approximating the eigenvector with largest magnitude of eigenvalue). Hence, we once again need to resort to orthogonalising the basis. We could simply take the QR factorisation of this basis, but the Arnoldi iteration coming up next also provides a neat way to solve the equation when projected onto the Krylov subspace.

6.2 Arnoldi iteration

The key to Krylov subspace methods turns out to be the transformation of A to an upper Hessenberg matrix by orthogonal similarity transforms, so that $A = QHQ^*$. We have already looked at using Householder transformations to do this in the previous section. The Householder technique is not so suitable for large dimensional problems, so instead we look at a way of proceeding column by column, just like the Gram-Schmidt method does for finding QR factorisations.

We do this by rewriting $AQ = QH$. The idea is that at iteration n we only look at the first n columns of Q, which we call $\hat{Q}_n$. When m is large, this is a significant saving: $mn \ll m^2$. To execute the iteration, it turns out that we should look at the $(n + 1) \times n$ upper left-hand section of H, i.e.

$$\tilde{H}_n = \begin{pmatrix} h_{11} & \ldots & h_{1n} \\ h_{21} & \ddots & \vdots \\ 0 & \ddots & h_{nn} \\ 0 & 0 & h_{n+1,n} \end{pmatrix}$$

Then, $A\hat{Q}_n = \hat{Q}_{n+1}\tilde{H}_n$.

Supplementary video

https://player.vimeo.com/video/454127181

Using the column space interpretation of matrix-matrix multiplication, we see that the n-th column is

$$Aq_n = h_{1n}q_1 + h_{2n}q_2 + \ldots + h_{n,n}q_n + h_{n+1,n}q_{n+1}.$$

This formula shows us how to construct the non-zero entries of the nth column of H; this defines the Arnoldi algorithm which we provide as pseudo-code below.

```
• $q_1 \leftarrow b/\|b\|$
• FOR $n = 1, 2, \ldots$
    – $v \leftarrow Aq_n$
    – FOR $j = 1$ TO $n$
        * $h_{jn} \leftarrow q_j^* v$
        * $v \leftarrow v - h_{jn}q_j$
    – END FOR
    – $h_{n+1,n} \leftarrow \|v\|$
    – $q_{n+1} \leftarrow v/\|v\|$
• END FOR
```

Exercise 111 (‡) *The* `cla_utils.exercises10.arnoldi()` *function has been left unimplemented. It should implement the Arnoldi algorithm using Numpy array operations where possible, and return the Q and H matrices after the requested number of iterations is complete. What is the minimal number of Python for loops possible?*

The test script `test_exercises10.py` *in the* `test` *directory will test this function.*

If we were to form the (reduced) QR factorisation of the $m \times n$ Krylov matrix

$$K_n = \begin{pmatrix} b & Ab & \ldots & A^{n1}b \end{pmatrix}$$

then we would get $Q = \hat{Q}_n$. Importantly, in the Arnoldi iteration, we never form K_n or R_n explicitly, since these are very ill-conditioned and not useful numerically.

Supplementary video

https://player.vimeo.com/video/454136990

But what is the use of the $\tilde{H}_n$ matrix? Applying $\hat{Q}_n^*$ to $A\hat{Q}_n = \hat{Q}_{n+1}\tilde{H}_n$ gives

$$\begin{aligned}\hat{Q}_n^* A \hat{Q}_n &= \hat{Q}_n^* \hat{Q}_{n+1} \tilde{H}_n, \\ &= \begin{pmatrix} 1 & 0 & \ldots & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \ldots & \ldots & 1 & 0 \end{pmatrix} \tilde{H}_n = H_n,\end{aligned}$$

where H_n is the $n \times n$ top left-hand corner of H.

Supplementary video

https://player.vimeo.com/video/454171516

The intrepretation of this is that H_n is the orthogonal projection of A onto the Krylov subspace $\mathrm{span}(K_n)$. To see this, take any vector v, and project Av onto the the Krylov subspace $\mathrm{span}(K_n)$.

$$PAv = \hat{Q}_n \hat{Q}_n^* v.$$

Then, changing basis to the orthogonal basis gives

$$\hat{Q}_n^*(\hat{Q}_n \hat{Q}_n^* A)\hat{Q}_n = \hat{Q}_n^* A \hat{Q}_n = H_n.$$

6.3 GMRES

Supplementary video

https://player.vimeo.com/video/454171559

The Generalised Minimum Residual method (GMRES), due to Saad (1986), exploits these properties of the Arnoldi iteration. The idea is that we build up the orthogonal basis for the Krylov subspaces one by one, and at each iteration we solve the projection of $Ax = b$ onto the Krylov basis as a least squares problem, until the residual $\|Ax - b\|$ is below some desired tolerance.

To avoid the numerical instabilities that would come from using the basis $(b, Ab, A^2 b, \ldots)$, we use the Arnoldi iteration to build an orthonormal basis, and seek approximate solutions of the form $x_n = \hat{Q}_n y$ for $y \in \mathbb{C}^n$. We then seek the value of y that minimises the residual

$$\mathcal{R}_n = \|A\hat{Q}_n y - b\|.$$

This explains the Minimum Residual part of the name. We also see from this definition that the residual cannot increase with iterations, because it only increases the subspace where we seek a solution.

This problem can be simplified further by using $A\hat{Q}_n = \hat{Q}_{n+1}\tilde{H}_n$, so

$$\mathcal{R}_n = \|\hat{Q}_{n+1}\tilde{H}_n y - b\|.$$

Remembering that $b = \|b\|q_1$, we see that the entire residual is in the column space of $\hat{Q}_{n+1}$, and hence we can invert on the column space using $\hat{Q}_{n+1}^*$ which does not change the norm of the residual due to the orthonormality.

$$\mathcal{R}_n = \|\tilde{H}_n y - \hat{Q}_{n+1}^* b\| = \|\tilde{H}_n y - e_1 \|b\|\|.$$

Finding y to minimise $\mathcal{R}_n$ requires the solution of a least squares problem, which can be computed via QR factorisation as we saw much earlier in the course.

Supplementary video

https://player.vimeo.com/video/454171921

We are now in position to present the GMRES algorithm as pseudo-code.

```
• $q_1 \leftarrow b/\|b\|$
• FOR $n = 1, 2, \ldots$
    – APPLY STEP $n$ OF ARNOLDI
    – FIND $y$ TO MINIMIZE $\|\tilde{H}_n y - \|b\| e_1\|$
    – $x_n \leftarrow \hat{Q}_n y$
    – CHECK IF $\mathcal{R}_n <$ TOL
• END FOR
```

Exercise 112 (‡) *The* `cla_utils.exercises10.GMRES()` *function has been left unimplemented. It should implement the basic GMRES algorithm above, using one loop over the iteration count. You can paste code from your* `cla_utils.exercises10.arnoldi()` *implementation, and you should use your least squares code to solve the least squares problem. The test script* `test_exercises10.py` *in the* `test` *directory will test this function.*

Exercise 113 *The least squares problem in GMRES requires the QR factorisation of H_k. It is wasteful to rebuild this from scratch given that we just computed the QR factorisation of H_{k-1}. Modify your code so that it recycles the QR factorisation, applying just one extra Householder rotation per GMRES iteration. Don't forget to check that it still passes the test.*

Hint

Don't get confused by the two Q matrices involved in GMRES! There is the Q matrix for the Arnoldi iteration, and the Q matrix for the least squares problem. These are not the same.

6.4 Convergence of GMRES

Supplementary video

https://player.vimeo.com/video/454198706

The algorithm presented as pseudocode is the way to implement GMRES efficiently. However, we can make an alternative formulation of GMRES using matrix polynomials.

We know that $x_n \in \operatorname{span}(K_n)$, so we can write

$$x_n = c_0 b + c_1 Ab + c_2 A^2 b + \ldots + c_{n-1} A^{n-1} b = p'(A) b,$$

where

$$p'(z) = c_0 + c_1 z + c_2 z^2 + \ldots + \ldots c_{n-1} z^{n-1} \implies p'(A) = c_0 I + c_1 A + c_2 A^2 + \ldots + c_{n-1} A^{n-1}.$$

Here we have introduced the idea of a matrix polynomial, where the kth power of z is replaced by the kth power of A.

The residual becomes

$$r_n = b - Ax_n = b - Ap'(A)b = (I - Ap'(A))b = p(A)b,$$

where $p(z) = 1 - zp'(z)$. Thus, the residual is a matrix polynomial p of A applied to b, where $p \in \mathcal{P}_n$, and

$$\mathcal{P}_n = \{\text{degree } \leq n \text{ polynomials with } p(0) = 1\}.$$

Hence, we can recast iteration n of GMRES as a polynomial optimisation problem: find $p_n \in \mathcal{P}_n$ such that $\|p_n(A)b\|$ is minimised. We have

$$\|r_n\| = \|p_n(A)b\| \leq \|p_n(A)\|\|b\| \implies \frac{\|r_n\|}{\|b\|} \leq \|p_n(A)\|.$$

Assuming that A is diagonalisable, $A = V\Lambda V^{-1}$, then $A^s = V\Lambda^s V^{-1}$, and

$$\|p_n(A)\| = \|Vp_n(\Lambda^s)V^{-1}\| \leq \underbrace{\|V\|\|V^{-1}\|}_{=\kappa(V)} \|p_n\|_{\Lambda(A)},$$

where $\Lambda(A)$ is the set of eigenvalues of A, and

$$\|p\|_{\Lambda(A)} = \sup_{x\in\Lambda(A)} \|p(x)\|.$$

Hence we see that GMRES will converge quickly if V is well-conditioned, and $p(x)$ is small for all $x \in \Lambda(A)$. This latter condition is not trivial due to the $p(0) = 1$ requirement. One way it can happen is if A has all eigenvalues clustered in a small number of groups, away from \$0\$. Then we can find a low degree polynomial that passes through 1 at $x = 0$, and 0 near each of the clusters. Then GMRES will essentially converge in a small number of iterations (equal to the degree of the polynomial). There are problems if the eigenvalues are scattered over a wide region of the complex plane: we need a very high degree polynomial to make $p(x)$ small at all the eigenvalues and hence we need a very large number of iterations. Similarly there are problems if eigenvalues are very close to zero.

Exercise 114 *Investigate the convergence of the matrices provided by the functions* `cla_utils.exercises10.get_AA100()`, `cla_utils.exercises10.get_BB100()`, *and* `cla_utils.exercises10.get_CC100()`, *by looking at the residual after each iteration (which should be provided by* `cla_utils.exercises10.GMRES()` *as specified in the docstring). What do you observe? What is it about the three matrices that causes this different behaviour?*

6.5 Preconditioned GMRES

Supplementary video

https://player.vimeo.com/video/454218547

This final topic has been a strong focus of computational linear algebra over the last 30 years. Typically, the matrices that we want to solve do not have eigenvalues clustered in a small number of groups, and so GMRES is slow. The solution (and the challenge) is to find a matrix $\hat{A}$ such that $\hat{A}x = y$ is cheap to solve (diagonal, or triangular, or something else) and such that $\hat{A}^{-1}A$ *does* have eigenvalues clustered in a small number of groups (e.g. $\hat{A}$ is a good approximation of A, so that $\hat{A}^{-1}A \approx I$ which has eigenvalues all equal to 1). We call $\hat{A}$ the preconditioning matrix, and the idea is to apply GMRES to the (left) preconditioned system

$$\hat{A}^{-1}Ax = \hat{A}^{-1}b.$$

GMRES on this preconditioned system is equivalent to the following algorithm, called preconditioned GMRES.

```
• SOLVE $\hat{A}\tilde{b} = b$.
• $q_1 \leftarrow \tilde{b}/\|\tilde{b}\|$
• FOR $n = 1, 2, \ldots$
    – SOLVE $\hat{A}v = Aq_n$
    – FOR $j = 1$ TO $n$
        ∗ $h_{jn} = q_j^* v$
        ∗ $v = v - h_{jn}q_j$
    – END FOR
    – $h_{n+1,n} \leftarrow \|v\|$
    – $q_{n+1}$gets v/|v|`
    – FIND $y$ TO MINIMIZE $\|\tilde{H}_n y - \|\tilde{b}\|e_1\|$
    – $x_n \leftarrow \hat{Q}_n y$
    – CHECK IF $\mathcal{R}_n <$ TOL
• END FOR
```

Exercise 115 *Show that this algorithm is equivalent to GMRES applied to the preconditioned system.*

The art and science of finding preconditioning matrices $\hat{A}$ (or matrix-free procedures for solving $\hat{A}x = y$) for specific problems arising in data science, engineering, physics, biology etc. can involve ideas from linear algebra, functional analysis, asymptotics, physics, etc., and represents a major activity in scientific computing today.

6.6 Knowing when to stop

We should stop an iterative method when the error is sufficiently small. But, we don't have access the the exact solution, so we can't compute the error. Two things we can look at are:

- The residual $r^k = Ax^k - b$, or
- The pseudo-residual $s^k = x^{k+1} - x^k$, which both tend to zero as $x^k \to x^*$ provided that A is invertible.

How do their sizes relate to the size of $e^k = x^k - x^*$?

$$\begin{aligned} e^k =& x^* - x^k \\ =& A^{-1}(Ax^* - Ax^k) \\ =& A^{-1}(b - Ax^k) \\ =& A^{-1}r^k, \end{aligned}$$

so $\|e^k\| \leq \|A^{-1}\|\|r^k\|$.

The relative error $\|e^k\|/\|x^*\|$ satisfies

$$\frac{\|e^k\|}{\|x^*\|} = \frac{\|A^{-1}r^k\|}{\|x\|} \leq \|A^{-1}\|\frac{\|r^k\|}{\|x\|} \leq \|A^{-1}\|\|A\|\frac{\|r^k\|}{\|b\|},$$

so the relative error is bounded from above by the condition number of $\|A\|$ multiplied by the relative residual $\|r^k\|/\|b\|$. If the condition number is large, it is possible to have a small residual but still have a large condition number.

Similar results hold for the pseudoresidual, but are more complicated to show for the case of GMRES.

CHAPTER

SEVEN

PRECONDITIONING KRYLOV METHODS

In this section we will discuss some preconditioners and how to analyse them. The most important question is how quickly the preconditioned GMRES algorithm converges for a given matrix and preconditioner. We will focus on the link between stationary iterative methods and preconditioners.

7.1 Stationary iterative methods

As we have already discussed, given a matrix equation $Ax = b$, iterative methods provide a way of obtaining a (hopefully) better approximate solution x^{k+1} from a previous approximate x^k. Stationary iterative methods are defined from splittings as follows.

Definition 116 (Stationary iterative methods) *A stationary iterative method is constructed from matrices M and N with $A = M + N$. Then the iterative method is defined by*

$$Mx^{k+1} = -Nx^k + b.$$

The word "stationary" refers to the fact that exactly the same thing is done at each iteration. This contrasts with Krylov methods such as GMRES, where the sequence of operations depends on the previous iterations (e.g. a different size least square system is solve in each GMRES iteration).

In this section we will introduce/recall some classic stationary methods.

Definition 117 (Richardson iteration) *For a chosen parameter $\omega > 0$, take $M = I/\omega$. This defines the iterative method given by*

$$x^{k+1} = x^k + \omega\left(b - Ax^k\right).$$

Richardson, L.F. (1910). *The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam*. Philos. Trans. Roy. Soc. London Ser. A 210: 307-357.

This approach is convenient for parallel computing, because each entry in x^{k+1} can be updated independently, once Ax^k has been evaluated.

Definition 118 (Jacobi's method) *Split $A = L + D + U$ with L strictly lower triangular, D diagonal and U strictly upper triangular, i.e.*

$$L_{ij} = 0,\, j \geq i, \quad D_{ij} = 0,\, i \neq j, \quad U_{ij},\, i \geq j.$$

Then, Jacobi's method is

$$Dx^{k+1} = b - (L + U)x^k.$$

D is very cheap to invert because it is diagonal; entries in x^{k+1} can be updated independently once $(L + U)x^k$ has been evaluated.

Jacobi, C.G.J. (1845). *Ueber eine neue Aufloesungsart der bei der Methode der kleinsten Quadrate vorkommenden linearen Gleichungen*, Astronomische Nachrichten, 22, 297-306.

Definition 119 (Gauss-Seidel Method) *Split $A = L + D + U$ with L strictly lower triangular, D diagonal and U strictly upper triangular. The Gauss-Seidel method (forward or backwards) is*

$$(L + D)x^{k+1} = b - Ux^k, \quad or, \quad (U + D)x^{k+1} = b - Lx^k.$$

Each Gauss-Seidel iteration requires the solution of a triangular system by forward/backward substitution.

Exercise 120 *Show that forward Gauss-Seidel is a modification Jacobi's method but using new values as soon as possible.*

Definition 121 (Scaled Gauss-Seidel method) *We introduce a scaling/relaxation parameter $\omega > 0$ and take $M = D/\omega + L$, so that*

$$\left(\frac{1}{\omega}D + L\right)x^{k+1} = b + \left(\left(\frac{1}{\omega} - 1\right)D - U\right)x^k.$$

For $\omega = 1$, we recover Gauss-Seidel. For $1 < \omega < 2$, we often obtain faster convergence. This is called Successive Over-Relaxation (SOR). The optimal value of ω is known for some problems. This was state of the art for numerical solution of PDEs in the 50s and 60s.

- Richardson and Jacobi are *simultaneous displacement methods*: updates can be done simultaneously (e.g. on a GPU). Changing variables by a permutation does not alter the algorithm.
- Gauss-Seidel and SOR are *successive displacement methods*: we can only overwrite the old vector with the new one element by element. Successive displacement methods usually converge faster, and changing variables by a permutation does alter the algorithm.

7.2 Using splitting methods as preconditioners

A (non-symmetric) preconditioner $\hat{A}$ can be built from a splitting method by applying one iteration with initial guess $x^0 = 0$.

A preconditioner is used to compute v by solving

$$\hat{A}v = r,$$

such that $Av \approx r$, and the above equation is easy to solve. We can use a stationary method by setting $v = x^1$ and $x^0 = 0$,

to get

$$Mv := Mx^1 = -N \underbrace{x^0}_{=0} + r = r,$$

i.e. we are choosing $\hat{A} = M$. Later we shall see how to relate convergence properties of splitting methods to the convergence of preconditioned CG using $\hat{A} = M$.

7.3 Symmetric iterative methods

Consider a symmetric matrix $A = A^T$ If we can build iterative methods from the splitting $A = M + N$, then we can also build iterative methods from the splitting $A = A^T = M^T + N^T$. We can then combine them together.

Definition 122 (Symmetric iterative method) *Given a splitting $A = M + N$, a symmetric method performs one stationary iteration using $M + N$, followed by one stationary iteration using $M^T + N^T$, i.e.*

$$Mx^{k+\frac{1}{2}} = -Nx^k + b, \quad M^T x^{k+1} = -N^T x^{k+\frac{1}{2}} + b.$$

Example 123 (Symmetric Successive Over-Relaxation (SSOR).) *For a symmetric matrix $A = L + D + U$, $L = U^T$. The symmetric version of SOR is then*

$$(L + \frac{1}{\omega}D)x^{k+\frac{1}{2}} = \left(\left(\frac{1}{\omega} - 1\right)D - U\right)x^k + b,$$
$$(U + \frac{1}{\omega}D)x^{k+1} = \left(\left(\frac{1}{\omega} - 1\right)D - L\right)x^{k+\frac{1}{2}} + b.$$

Some Krylov methods, notably the Conjugate Gradient method, require the preconditioner $\hat{A}$ to be symmetric. We can build symmetric preconditioners from symmetric splitting methods. Write the symmetric iteration as a single step with $x^0 = 0$.

$$\begin{aligned} M^T x^1 &= (M - A)^T x^{\frac{1}{2}} + b, \\ &= (M - A)^T M^{-1} b + b, \\ &= (M^T + M - A) M^{-1} b, \end{aligned}$$

so that

$$x^1 = M^{-T}(M^T + M - A)M^{-1}b,$$

i.e. $\hat{A}^{-1} = M^{-T}(M^T + M - A)M^{-1}$.

Example 124 (Symmetric Gauss-Seidel preconditioner) $\hat{A}^{-1} = (L + D)^{-T} D (L + D)^{-1}$.

7.4 Convergence criteria for stationary methods

In this section we will look at the convergence of stationary methods. This is relevant because it relates directly to the convergence properties of the corresponding preconditioned Krylov method when the stationary method is used as a preconditioner.

For a splitting $A = M + N$, recall that the iterative method is

$$Mx^{k+1} = -Nx^k + b.$$

On the other hand, the solution x^* of $Ax = b$ satisfies

$$Mx^* = -Nx^* + b.$$

Subtracting these two equations gives

$$Me^{k+1} = -Ne^k, \quad e^k = x^* - x^k,$$

so

$$e^{k+1} = Ce^k \implies e^k = C^k e^0, \quad C := -M^{-1}N = -M^{-1}(A - M) = I - M^{-1}A.$$

C is called the *iteration matrix*.

For a symmetric iterative method,

$$Mx^{k+\frac{1}{2}} = -Nx^k + b, \quad M^T x^{k+1} = -N^T x^{k+\frac{1}{2}} + b,$$

we subtract $Ax^* = b$ from both equations to get

$$Me^{k+\frac{1}{2}} = -Ne^k, \quad M^T e^{k+1} = -N^T e^{k+\frac{1}{2}}.$$

Then eliminating $e^{k+1/2}$ gives

$$M^T e^{k+1} = N^T M^{-1} N e^k,$$

i.e. the iteration matrix is

$$C = M^{-T} N^T M^{-1} N$$

Exercise 125 *Show that*

$$C = I - (M_s)^{-1} A,$$

where

$$M_s = M(M + M^T - A)^{-1} M^T.$$

From the above exercise, note the relationship between M_s and $\hat{A}$ for symmetric methods.

Definition 126 (Convergence of stationary methods) *An iterative method based on the splitting $A = M + N$ with iteration matrix $C = -M^{-1}N$ is called {convergent} if*

$$y^k = C^k y^0 \to 0$$

for any initial vector y^0.

Exercise 127 *Show that this implies that $e^k = x^* - x^k \to 0$ i.e. $x^k \to x^*$ as $k \to \infty$.*

Theorem 128 (A first convergence criterion) If $\|C\| < 1$, using the operator norm for some chosen vector norm, *then the iterative method converges.*

Proof 129

$$\begin{aligned} \|y^k\| =& \|C^k y^0\| \\ \le& \|C^k\| \|y^0\| \\ \le& \ (\|C\|)^k \ \|y^0\| \to 0 \quad \textit{as}\, k \to \infty. \end{aligned}$$

This is only a sufficient condition. There may be matrices C with $\|C\| > 1$, but the method is still convergent.

To obtain a necessary condition, we need to use the spectral radius.

Definition 130 *The spectral radius $\rho(C)$ of a matrix C is the maximum of the absolute values of all the eigenvalues λ_i of C:*

$$\rho(C) = \max_{1 \le i \le n} |\lambda_i|.$$

Theorem 131 *An iterative method converges $\iff$ $\rho(C) < 1$.*

Proof 132 *[Proof that $\rho(C) \ge 1 \implies$ non-convergence]*

If $\rho(C) \ge 1$, then C has an eigenvector v with $\|v\|_2 = 1$ and eigenvalue λ with $|\lambda| > 1$. Then

$$\|C^k v\|_2 = \|\lambda^k v\|_2 = |\lambda|^k \|v\|_2 \ge 1,$$

which does not converge to zero.

[Proof that $\rho(C) < 1 \implies$ convergence]

Assume a linearly independent eigenvalue expansion (not necessary for the proof but it simplifies things a lot) $z = \sum_{i=1}^n \alpha_i v_i$. Then,

$$C^k z = \sum_{i=1}^n \alpha_i C^k v_i = \sum_{i=1}^n \alpha_i \lambda^k v_i \to 0.$$

- For symmetric matrices B, $\rho(B) = \|B\|_2$, so

the two convergence theorems are related.

- If $\|C\| = c < 1$, then

  $$\|e^{k+1}\| = \|Ce^k\| \le \|C\| \|e^k\| = c\|e^k\|.$$

  This guarantees that the error will be reduced by a factor of at least c in each iteration. If we only have $\rho(C) < 1$, not $\|C\| < 1$ then the error may not converge monotonically.

Example 133 (Range of SOR parameter) *We can use this to analyse the SOR parameter ω.*

$$\left(\frac{1}{\omega} D + L\right) x^{k+1} = b + \left(\left(\frac{1}{\omega} - 1\right) D - U\right) x^k$$

What values of ω? For SOR,iteration matrix C is

$$C = \left(\frac{1}{\omega} D + L\right)^{-1} \left(\frac{1-\omega}{\omega} D - U\right) = (D + \omega L)^{-1}((1-\omega)D - \omega U).$$

so

$$\begin{aligned}\det(C) &= \det\left((D+\omega L)^{-1}((1-\omega)D-\omega U)\right) \\ &= \det\left((D+\omega L)^{-1}\right)\det\left((1-\omega)D-\omega U\right) \\ &= \det\left(D^{-1}\right)\det(D)\det\left((I-\omega I)-\omega D^{-1}U\right) \\ &= \det\left((1-\omega)I\right) = (1-\omega)^n.\end{aligned}$$

The determinant is the product of the eigenvalues, hence $\rho(C) < 1$ requires $|1-\omega| < 1$.

7.5 Splitting methods as preconditioners

Recall that preconditioned GMRES converges well if the eigenvalues of $\hat{A}^{-1}A$ are clustered together.

Theorem 134 *Let A be a matrix with splitting $M+N$, such that $\rho(C) < c < 1$. Then, the eigenvalues of the left preconditioned matrix $\hat{A}^{-1}A$ with $\hat{A} = M$ are located in a disk of radius c around 1 in the complex plane.*

Proof 135

$$C = -M^{-1}N = M^{-1}(M-A) = I - M^{-1}A.$$

Then,

$$1 > c > \rho(C) = \rho(I - M^{-1}A),$$

and the result follows since I and $M^{-1}A$ have a simultaneous eigendecomposition.

We deduce that good convergence of the GMRES algorithm occurs when c is small.

For symmetric splittings, we have already observed that the iteration matrix is

$$C = I - (M_s)^{-1}A,$$

where

$$M_s = M(M + M^T - A)^{-1}M^T.$$

For symmetric splittings we can say a little more about the preconditioner.

Theorem 136 *Let A be a matrix with splitting $M+N$, such that the symmetric splitting has iteration matrix*

$$\rho(C) = c < 1,$$

and assume further that M_s is positive definite.

Then, the eigenvalues of the symmetric preconditioned matrix $\hat{A}^{-1}A$ are contained in the interval $[1-c, 1+c]$.

Proof 137 *We have*

$$\begin{aligned}C &= I - (M_s)^{-1}A, \\ &= I - \hat{A}^{-1}A,\end{aligned}$$

so $\rho(I - \hat{A}^{-1}A) = \rho(C) = c$. Further, M_s is symmetric and positive definite, so there exists a unique symmetric positive definite matrix square root S such that $SS = M_s$. Then,

$$M_s^{-1}A = SSA = S(SAS)S^{-1}.$$

Thus, $M_s^{-1}A$ is similar to (and therefore has the same eigenvalues as) SAS, which is symmetric, and therefore has real eigenvalues, and the result follows.

7.6 Convergence analysis for Richardson

First we examine Richardson iteration. In the unscaled case,

$$x^{k+1} = x^k - \left(Ax^k - b\right), \quad M = I,\, N = A - I, \implies C = I - A.$$

Let e be an eigenvector of A with eigenvalue λ, so $Ae = \lambda e$. Then $(I - A)e = e - \lambda e = (1 - \lambda)e$. So, e is an eigenvector of $I - A$ with eigenvalue $1 - \lambda$. Richardson's method will converge if $\rho(C) < 1$ i.e. $|1 - \lambda| < 1$ for all eigenvalues λ of A.

This is restrictive, which motivates the scaled Richardson iteration,

$$x^{k+1} = x^k - \omega\left(Ax^k - b\right), \quad M = \frac{I}{\omega},\, N = A - \frac{I}{\omega}, \implies C = I - \omega A.$$

If A has eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ then the iterative matrix C has eigenvalues $1 - \omega\lambda_1, 1 - \omega\lambda_2, \ldots, 1 - \omega\lambda_n$. This requires $|1 - \omega\lambda_i| < 1$, $i = 1, \ldots, n$, for convergence.

If, further, A is symmetric positive definite, then all eigenvalues are real and positive. Then, all of the eigenvalues of C lie between $1 - \omega\lambda_{\min}$ and $1 - \omega\lambda_{\max}$. We can minimise $\rho(C)$ by choosing $\omega = 2/(\lambda_{\min} + \lambda_{\max})$. The resulting iteration matrix has spectral radius

$$\rho(C) = 1 - 2\frac{\lambda_{\min}}{\lambda_{\min} + \lambda_{\max}} = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\min} + \lambda_{\max}}.$$

7.7 Convergence analysis for symmetric matrices

For a symmetric positive definite matrix A, recall the Rayleigh Quotient formula,

$$\lambda_{\max} = \max_{x\neq 0} \frac{x^T A x}{x^T x} \equiv \|A\|_2^2, \quad \lambda_{\min} = \min_{x\neq 0} \frac{x^T A x}{x^T x},$$

implying that

$$\lambda_{\min}\|y\|_2^2 \leq y^T A y \leq \lambda_{\max}\|y\|_2^2$$

for any non-zero vector y.

Definition 138 (A-weighted norm) *For symmetric positive definite A, we can define the weighted vector norm*

$$\|x\|_A = \sqrt{x^T A x},$$

and the corresponding matrix (operator) norm

$$\|B\|_A = \|A^{1/2} B A^{-1/2}\|_2.$$

These norms are useful for studying convergence of iterative methods for $Ax = b$ in the symmetric positive definite case.

Theorem 139 *For a splitting $A = M + N$, if the (symmetric) matrix $M + M^T - A$ is positive definite then*

$$\|I - M^{-1}A\|_A < 1.$$

Proof 140 *If $y = (I - M^{-1}A)x$, $w = M^{-1}Ax$, then*

$$\begin{aligned}
\|y\|_A^2 &= (x - w)^T A (x - w) = x^T A x - 2w^T M w + w^T A w \\
&= x^T A x - w^T(M + M^T)w + w^T A w \\
&= x^T A x - w^T(M + M^T - A)w \\
&\leq \|x\|_A^2 - \mu_{\min}\|w\|_2^2,
\end{aligned}$$

where $\mu_{\min}$ is the (positive) minimum eigenvalue of $M^T + M - A$.

Further,

$$\begin{aligned}\|w\|_2^2 &= x^T A \left(M^{-1}\right)^T M^{-1}Ax \\ &= \left(A^{1/2}x\right)^T A^{1/2}\left(M^{-1}\right)^T M^{-1}A^{1/2}\left(A^{1/2}x\right) \\ &\geq \hat{\mu}_{\min}\|A^{1/2}x\|_2^2 = \hat{\mu}_{\min}\|x\|_A^2,\end{aligned}$$

where $\hat{\mu}_{\min}$ is the minimum eigenvalue of $A^{1/2}\left(M^{-1}\right)^T M^{-1}A^{1/2}$ i.e. the square of the minimum eigenvalue of $M^{-1}A^{1/2}$, which is invertible so $\hat{\mu}_{\min} > 0$. If $y = (I - M^{-1}A)x$, $w = M^{-1}Ax$, then

$$\|y\|_A^2 \leq (1 - \mu_{\min}\hat{\mu}_{\min})\,\|x\|_A^2 < \|x\|_A^2.$$

This enables us to show the following useful result for symmetric positive definite matrices.

Theorem 141 *Let A be a symmetric positive definite matrix with splitting $A = M + N$, if M is positive definite, then*

$$\rho(I - M^{-1}A) = \|I - M^{-1}A\|_A = \|I - M^{-1}A\|_M.$$

Proof 142

$$\begin{aligned}I - A^{1/2}M^{-1}A^{1/2} &= A^{1/2}(I - M^{-1}A)A^{-1/2}, \\ I - M^{-1/2}AM^{-1/2} &= M^{1/2}(I - M^{-1}A)M^{-1/2},\end{aligned}$$

so $I - M^{-1}A$, $I - A^{1/2}M^{-1}A^{1/2}$, and $I - M^{-1/2}AM^{-1/2}$ all have the same eigenvalues, since they are similar matrices. Hence,

$$\begin{aligned}\rho(I - M^{-1}A) &= \rho(I - A^{1/2}M^{-1}A^{1/2}) \\ &= \|I - A^{1/2}M^{-1}A^{1/2}\|_2 \\ &= \|I - M^{-1}A\|_A,\end{aligned}$$

and similarly for $I - M^{-1/2}AM^{-1/2}$.

The consequence of this is that if $M + M^T - A$ is symmetric positive definite then there is a guaranteed reduction in the A-norm of the error in each iteration. If M is also symmetric positive definite the there is guaranteed reduction in the M-norm of the error in each iteration.

Now we apply this to the convergence of Jacobi iteration. In this case $M = D$, so $M^T + M - A = 2D - A$ which may not be positive definite. We generalise to scaled Jacobi iteration with $M = D/\omega$.

Proposition 143 *Let A be a symmetric positive definite matrix. Let λ be the (real) maximum eigenvalue of $D^{-1/2}AD^{-1/2}$. If $\omega < 2/\lambda$ then scaled Jacobi iteration converges.*

Proof 144 *For scaled Jacobi iteration with $M = D/\omega$, we have $M^T + M - A = 2D/\omega - A$. To check positive definiteness we need to show that*

$$x^T\left(\frac{2}{\omega}D - A\right)x > 0,$$

for all $x \neq 0$.

To show this, we write $x = D^{-1/2}y$, so that

$$\begin{aligned}x^T\left(\frac{2}{\omega}D - A\right)x &= y^T\left(\frac{2}{\omega}I - D^{-1/2}AD^{-1/2}\right)y \\ &\geq \mu\|y\|^2 > 0,\end{aligned}$$

provided that the minimum eigenvalue μ of $F = \frac{2}{\omega}I - D^{-1/2}AD^{-1/2}$ is positive (it is real since F is symmetric). We have $\mu = 2/\omega - \lambda$ Hence, $2D/\omega - A$ is positive definite (so scaled Jacobi converges) if $2/\omega - \lambda > 0$ i.e. $\omega < 2/\lambda$.

Proposition 145 *Let A be a symmetric positive definite matrix. Then Gauss-Seidel iteration always converges.*

Proof 146 *For Gauss-Seidel,*

$$M^T + M - A = (D+L)^T + D + L - A = D + U + D + L - A = D,$$

which is symmetric-positive definite, so Gauss-Seidel always converges.

Proposition 147 *Let A be a symmetric positive definite matrix. Then SOR converges provided that $0 < \omega 2$.*

Proof 148 *For SOR,*

$$\begin{aligned} M^T + M - A &= \left(\frac{1}{\omega}D + L\right)^T + \frac{1}{\omega}D + L - A \\ &= \frac{2}{\omega}D + U + L - (L + D + U) = \left(\frac{2}{\omega} - 1\right)D, \end{aligned}$$

which is symmetric positive definite provided that $0 < \omega < 2$.

7.8 An example matrix

We consider stationary methods for an example arising from the finite difference discretisation of the two point boundary value problem

$$-\frac{d^2u}{dx^2} = f, \quad u(0) = u(1) = 0.$$

Here, f is assumed known and we have to find u. We approximate this problem by writing $u_k = u(k/(n+1))$ for $k = 0, 1, 2, \ldots, n+1$. From the boundary conditions we have $u_0 = u_{n+1} = 0$, meaning we just have to find u_k with $1 \leq k \leq n$, that solve the finite difference approximation

$$-u_{k-1} + 2u_k - u_{k+1} = f_k, \quad 1 \leq k \leq n,$$

where $f_k = f(k/n)/n^2$, $1 \leq k \leq n$. Taking into account the boundary conditions $u_0 = u_{n+1} = 0$, we can write this as a matrix system $Ax = b$ with

$$A = \begin{pmatrix} 2 & -1 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ 0 & -1 & 2 & \cdots & \vdots \\ \vdots & & & & \vdots \\ \vdots & 0 & -1 & 2 & -1 \\ 0 & 0 & \cdots & -1 & 2 \end{pmatrix}, \quad x = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, \quad b = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}.$$

However, it is possible to evaluate Ax and to implement our classic stationary iterative methods without ever forming A. This is critically important for efficient implementations (especially when extending to 2D and 3D problems).

We introduce this example matrix because it is possible to compute spectral radii for all of the matrices arising in the analysis of classic stationary methods. In the next example we consider Jacobi.

Example 149 (Jacobi iteration for the example matrix) *In this case, $D = 2I$. Thus in fact, scaled Jacobi and scaled Richardson are equivalent. We have to find the maximum eigenvalue of $K = D^{-1}A$. We can compute this by knowing that the eigenvectors v of K are all of the form*

$$v = \left(\begin{pmatrix} \sin(l\pi/(n+1)) \\ \sin(2l\pi/(n+1)) \\ \sin(nl\pi/(n+1)) \end{pmatrix}\right),$$

with one eigenvector for each value of $0 < l < n+1$. This can be proved by considering symmetries of the matrix, but here we just assume this form and establish that we have eigenvectors after substituting into the definition of *an eigenvector $Av = \lambda v$. This is a general approach that can be tried for any matrices arising in the analysis of convergence of classic stationary methods for this example matrix.*

$$(D^{-1}Au)_k = -u_{k-1}/2 + u_k - u_{k+1}/2 = \lambda u_k$$

which becomes

$$\lambda \sin(kl\pi/(n+1)) = -\sin((l-1)k\pi/(n+1))/2 + \sin(kl\pi/(n+1)) - \sin((l+1)k\pi/(n+1))/2,$$

and you can use trigonometric formulae, or write

$$\begin{aligned}\lambda \sin(kl\pi/(n+1) &= -\sin((l-1)k\pi/(n+1))/2 + \sin(lk\pi/(n+1)) - \sin((l+1)k\pi/(n+1))/2\\ &= \sin(kl\pi/(n+1)) - \Im\left(\exp(ik(l-1)\pi/(n+1)) + \exp(ik(l+1)\pi/(n+1))\right)/2\\ &= \sin(kl\pi/(n+1)) - \Im\left((\exp(-ik\pi/(n+1)) + \exp(ik\pi/(n+1)))\exp(ikl\pi/(n+1))\right)/2\\ &= \sin(kl\pi/(n+1)) - \Im\left(\sin(k\pi/(n+1))\exp(ikl\pi/(n+1))\right)\\ &= \sin(kl\pi/(n+1))(1-\sin(k\pi/(n+1)))\end{aligned}$$

and we conclude that $\lambda = 1-\sin(k\pi/(n+1))$ are the eigenvalues with $0 < k < n+1$. The maximum eigenvalue corresponds to $k=1$ and $k=n$, with $\lambda = 1-\sin(\pi/(n+1))$.

The condition $\omega < 2/\lambda$ thus requires that

$$\omega < \frac{2}{1-\sin(\pi/(n+1))}.$$

Exercise 150 *Find the value of ω for scaled Jacobi such that the convergence rate is maximised, i.e. so that $\rho(C)$ is minimised. What happens to this rate as $n \to \infty$?*

7.9 Chebyshev acceleration (nonexaminable in 2023/24)

Say we have computed iterates $x^0, x^1, \ldots, x^k$ using

$$Mx^{k+1} = -Nx^k + b.$$

If the method is convergent, then these iterates are homing in on the solution. Can we use extrapolation through these iterates to obtain a better guess for the solution?

$$\text{Find } c_{jk},\ j = 1,\ldots,k, \text{ with } y^k = \sum_{j=0}^{k} c_{jk}x^j,$$

with y^k the best possible approximation to x^*.

The usual iterative method has $c_{kk} = 1$, and $c_{jk} = 0$ for $j < k$. If $x^i = x^*$, $i = 0,1,\ldots,k$ then

$$y^k = \sum_{j=0}^{k} c_{jk}x^* = x^* \sum_{j=0}^{k} c_{jk},$$

so we need $\sum_{j=0}^{k} c_{jk} = 1$. Subject to this constraint, we

seek to minimise $y^k - x^* = \sum_{j=0}^{k} c_{jk}(x^j - x^*)$.

We can interpret this in terms of matrix polynomials by writing

$$\begin{aligned}x^* - y^k &= \sum_{j=0}^{k} c_{jk}(x^* - x^j),\\ &= \sum_{j=0}^{k} c_{jk}\left(-M^{-1}N\right)^j e^0,\\ &= p_k\left(-M^{-1}N\right)e^0,\end{aligned}$$

where

$$p_k(X) = c_{0k} + c_{1k}X + c_{2k}X^2 + \ldots + c_{kk}X^k,$$

with $p_k(1) = 1$ (from our condition $\sum_{j=0}^{k} c_{jk} = 1$).

We want to try to minimise $y^k - x^*$ by choosing c_{0k}, c_{1k}, …, c_{kk} so that the eigenvalues of p_k are as small as possible. If λ is an eigenvalue of $C = -M^{-1}N$, then $p_k(\lambda)$ is an eigenvalue of $p_k(C)$. It is not practical to know all the eigenvalues of a large matrix, so we will develop methods that work if we know that all eigenvalues of C are real, and satisfy $-1 < \alpha < \lambda < \beta < 1$, for some constants α and β (we know that $|\lambda| < 1$ otherwise the basic method is not convergent.

If all eigenvalues of C are real, and satisfy $-1 < \alpha < \lambda < \beta < 1$, then we try to make $\rho_{\max} = \max_{\alpha \le t \le \beta} |p_k(t)|$ as small as possible. Then, if λ is an eigenvalue of C, then the corresponding eigenvalue of $p_k(C)$ will satisfy $|\lambda_{p_k}| = |p_k(\lambda)| \le \rho_{\max}$. We have reduced the problem to trying to find polynomials $p(t)$ that have the smallest absolute value in a given range, subject to $p(1) = 1$. The solution to this problem is known: Chebyshev polynomials.

Definition 151 (The Chebyshev polynomial of degree k, $T_k(t)$ is defined by the recurrence)

$$T_0(t) = 1,\, T_1(t) = t,\, T_k(t) = 2tT_{k-1}(t) - T_{k-2}(t).$$

For example: $T_2(t) = 2tT_1(t) - T_0(t) = 2t^2 - 1$.

If we search for the k-th degree polynomial $p_k(t)$ that minimises

$$\max_{-1 \le t \le 1} |p_k(t)|$$

subject to the constraint that the coefficient of t^k is 2^{k-1} then we get the k-th order Chebyshev polynomial $T_k(t)$. The maximum value is 1.

This is not quite what we want, so we change variables, to get

$$T_k\left(\frac{2t - \beta - \alpha}{\beta - \alpha}\right) \quad \text{minimises} \quad \max_{\alpha \le t \le \beta} |p_k(t)|$$

subject to the constraint that the coefficient of t^k is $2^{2k-1}/(\beta - \alpha)$. The maximum value is 1.

Then we scale the polynomial to reach the condition $p_k(0) = 1$.

$$p_k = \frac{T_k\left(\frac{2t-\beta-\alpha}{\beta-\alpha}\right)}{T_k\left(\frac{2-\beta-\alpha}{\beta-\alpha}\right)} \quad \text{minimises} \quad \max_{\alpha \le t \le \beta} |p_k(t)|$$

subject to the constraint that $p_k(0) = 1$. The maximum value is

$$\frac{1}{T_k\left(\frac{2-\beta-\alpha}{\beta-\alpha}\right)}.$$

Say we have computed iterates $x^0, x^1, \ldots, x^k$ using

$$Mx^{k+1} = -Nx^k + b.$$

Write

$$p_k = \frac{T_k\left(\frac{2t-\beta-\alpha}{\beta-\alpha}\right)}{T_k\left(\frac{2-\beta-\alpha}{\beta-\alpha}\right)}$$

in the form

$$p_k(t) = c_{0k} + c_{1k}t + c_{2k}t^2 + \ldots + c_{kk}t^k,$$

then

$$y^k = \sum_{j=0}^{k} c_{jk} x^k.$$

There appears to be a practical problem: we need to store x^0, x^1, ..., x^k in order to calculate y^k. However, we can get a formula for y^k in terms of y^{k-1} and y^{k-2} by using

$$T_k(t) = 2tT_{k-1}(t) - T_{k-2}(t).$$

We get

$$p_k(t) = 2\frac{2t-\beta-\alpha}{\beta-\alpha}\frac{T_{k-1}(s)}{T_k(s)}p_{k-1}(t) - \frac{T_{k-2}(s)}{T_k(s)}p_{k-2}(t),$$

where $s = \frac{2-\beta-\alpha}{\beta-\alpha}$.

After some manipulations we obtain

$$y^k = \omega_k\left(y^{k-1} - y^{k-2} + \gamma z^{k-1}\right) + y^{k-2},$$

where

$$\gamma = \frac{2}{2-\beta-\alpha}, \quad Mz^{k-1} = b - Ay^{k-1}.$$

with starting formulas

$$\begin{aligned} y^0 &= x^0 \\ y^1 &= x^0 + \gamma M^{-1}(b - Ax^0). \end{aligned}$$

Also,

$$\omega_k = \frac{1}{1 - \omega_{k-1}/(4s^2)}, \, \omega_1 = 2.$$

(See Golub and Van Loan for details).

Chebyshev can dramatically accelerate preconditioners provided that the preconditioned operator is positive definite and upper and lower bounds on the eigenvalues are known.

The automated documentation for the skeleton code is below.

CHAPTER

EIGHT

CLA_UTILS PACKAGE

8.1 Submodules

8.2 cla_utils.exercises1 module

cla_utils.exercises1.**ABiC**(*Ahat*, *xr*, *xi*)

Return the real and imaginary parts of z = A*x, where A = B + iC with

Parameters
Ahat – an mxm-dimensional numpy array with Ahat[i,j] = B[i,j] for i>=j and Ahat[i,j] = C[i,j] for i<j.

Return zr
m-dimensional numpy arrays containing the real part of z.

Return zi
m-dimensional numpy arrays containing the imaginary part of z.

cla_utils.exercises1.**basic_matvec**(*A*, *x*)

Elementary matrix-vector multiplication.

Parameters

- **A** – an mxn-dimensional numpy array
- **x** – an n-dimensional numpy array

returns an m-dimensional numpy array which is the product of A with x

This should be implemented using a double loop over the entries of A

Return b
m-dimensional numpy array

cla_utils.exercises1.**column_matvec**(*A*, *x*)

Matrix-vector multiplication using the representation of the product Ax as linear combinations of the columns of A, using the entries in x as coefficients.

Parameters

- **A** – an mxn-dimensional numpy array
- **x** – an n-dimensional numpy array

Return b
an m-dimensional numpy array which is the product of A with x

This should be implemented using a single loop over the entries of x

cla_utils.exercises1.**rank1pert_inv**(*u*, *v*)

Return the inverse of the matrix A = I + uv^*, where I is the mxm dimensional identity matrix, with

Parameters

- **u** – m-dimensional numpy array
- **v** – m-dimensional numpy array

`cla_utils.exercises1.rank2`(*u1*, *u2*, *v1*, *v2*)

Return the rank2 matrix A = u1*v1^* + u2*v2^*.

Parameters

- **u1** – m-dimensional numpy array
- **u2** – m-dimensional numpy array
- **v1** – n-dimensional numpy array
- **v2** – n-dimensional numpy array

`cla_utils.exercises1.time_matvecs()`

Get some timings for matvecs.

`cla_utils.exercises1.timeable_basic_matvec()`

Doing a matvec example with the basic_matvec that we can pass to timeit.

`cla_utils.exercises1.timeable_column_matvec()`

Doing a matvec example with the column_matvec that we can pass to timeit.

`cla_utils.exercises1.timeable_numpy_matvec()`

Doing a matvec example with the builtin numpy matvec so that we can pass to timeit.

8.3 cla_utils.exercises10 module

`cla_utils.exercises10.GMRES`(*A*, *b*, *maxit*, *tol*, *return_residual_norms=False*, *return_residuals=False*)

For a matrix A, solve Ax=b using the basic GMRES algorithm.

Parameters

- **A** – an mxm numpy array
- **b** – m dimensional numpy array
- **maxit** – integer, the maximum number of iterations
- **tol** – floating point number, the tolerance for termination
- **return_residual_norms** – logical
- **return_residuals** – logical

Return x
an m dimensional numpy array, the solution

Return nits
if converged, the number of iterations required, otherwise equal to -1

Return rnorms
nits dimensional numpy array containing the norms of the residuals at each iteration

Return r
mxnits dimensional numpy array, column k contains residual at iteration k

`cla_utils.exercises10.arnoldi`(*A*, *b*, *k*)

For a matrix A, apply k iterations of the Arnoldi algorithm, using b as the first basis vector.

Parameters

- **A** – an mxm numpy array
- **b** – m dimensional numpy array, the starting vector

- **k** – integer, the number of iterations

Return Q
an mx(k+1) dimensional numpy array containing the orthonormal basis

Return H
a (k+1)xk dimensional numpy array containing the upper Hessenberg matrix

`cla_utils.exercises10.`**`get_AA100`**`()`

Get the AA100 matrix.

Return A
a 100x100 numpy array used in exercises 10.

`cla_utils.exercises10.`**`get_BB100`**`()`

Get the BB100 matrix.

Return B
a 100x100 numpy array used in exercises 10.

`cla_utils.exercises10.`**`get_CC100`**`()`

Get the CC100 matrix.

Return C
a 100x100 numpy array used in exercises 10.

8.4 cla_utils.exercises2 module

`cla_utils.exercises2.`**`GS_classical`**`(`*A*`)`

Given an mxn matrix A, compute the QR factorisation by classical Gram-Schmidt algorithm, transforming A to Q in place and returning R.

Parameters
A – mxn numpy array

Return R
nxn numpy array

`cla_utils.exercises2.`**`GS_modified`**`(`*A*`)`

Given an mxn matrix A, compute the QR factorisation by modified Gram-Schmidt algorithm, transforming A to Q in place and returning R.

Parameters
A – mxn numpy array

Return R
nxn numpy array

`cla_utils.exercises2.`**`GS_modified_R`**`(`*A*`)`

Implement the modified Gram Schmidt algorithm using the lower triangular formulation with Rs provided from GS_modified_get_R.

Parameters
A – mxn numpy array

Return Q
mxn numpy array

Return R
nxn numpy array

`cla_utils.exercises2.`**`GS_modified_get_R`**`(`*A*`, `*k*`)`

Given an mxn matrix A, with columns of A[:, 0:k] assumed orthonormal, return upper triangular nxn matrix R such that Ahat = A*R has the properties that 1) Ahat[:, 0:k] = A[:, 0:k], 2) A[:, k] is normalised and orthogonal to the columns of A[:, 0:k].

Parameters

- **A** – mxn numpy array
- **k** – integer indicating the column that R should orthogonalise

Return R
nxn numpy array

`cla_utils.exercises2.`**`orthog_cpts`**`(`*v*, *Q*`)`

Given a vector v and an orthonormal set of vectors q_1,…q_n, compute v = r + u_1q_1 + u_2q_2 + … + u_nq_n for scalar coefficients u_1, u_2, …, u_n and residual vector r

Parameters

- **v** – an m-dimensional numpy array
- **Q** – an mxn-dimensional numpy array whose columns are the orthonormal vectors

Return r
an m-dimensional numpy array containing the residual

Return u
an n-dimensional numpy array containing the coefficients

`cla_utils.exercises2.`**`orthog_proj`**`(`*Q*`)`

Given a vector v and an orthonormal set of vectors q_1,…q_n, compute the orthogonal projector P that projects vectors onto the subspace spanned by those vectors.

Parameters
Q – an mxn-dimensional numpy array whose columns are the orthonormal vectors

Return P
an mxm-dimensional numpy array containing the projector

`cla_utils.exercises2.`**`orthog_space`**`(`*V*`)`

Given set of vectors u_1,u_2,…, u_n, compute the orthogonal complement to the subspace U spanned by the vectors.

Parameters
V – an mxn-dimensional numpy array whose columns are the vectors u_1,u_2,…,u_n.

Return Q
an mxl-dimensional numpy array whose columns are an orthonormal basis for the subspace orthogonal to U, for appropriate l.

`cla_utils.exercises2.`**`solve_Q`**`(`*Q*, *b*`)`

Given a unitary mxm matrix Q and a vector b, solve Qx=b for x.

Parameters

- **Q** – an mxm dimensional numpy array containing the unitary matrix
- **b** – the m dimensional array for the RHS

Return x
m dimensional array containing the solution.

8.5 cla_utils.exercises3 module

`cla_utils.exercises3.`**`householder`**`(`*A*`)`

Given a real mxn matrix A, find the reduction to upper triangular matrix R using Householder transformations. The reduction should be done “in-place”, so that A is transformed to R.

Parameters
A – an mxn-dimensional numpy array

cla_utils.exercises3.**householder_ls**(*A*, *b*)

Given a real mxn matrix A and an m dimensional vector b, find the least squares solution to Ax = b.

Parameters

- **A** – an mxn-dimensional numpy array
- **b** – an m-dimensional numpy array

Return x

an n-dimensional numpy array

cla_utils.exercises3.**householder_qr**(*A*)

Given a real mxn matrix A, use the Householder transformation to find the full QR factorisation of A.

Parameters

A – an mxn-dimensional numpy array

Return Q

an mxm-dimensional numpy array

Return R

an mxn-dimensional numpy array

cla_utils.exercises3.**householder_solve**(*A*, *b*)

Given a real mxm matrix A, use the Householder transformation to solve Ax_i=b_i, i=1,2,…,k.

Parameters

- **A** – an mxm-dimensional numpy array
- **b** – an mxk-dimensional numpy array whose columns are the right-hand side vectors b_1,b_2,…,b_k.

Return x

an mxk-dimensional numpy array whose columns are the right-hand side vectors x_1,x_2,…,x_k.

cla_utils.exercises3.**solve_U**(*U*, *b*)

Solve systems Ux_i=b_i for x_i with U upper triangular, i=1,2,…,k

Parameters

- **U** – an mxm-dimensional numpy array, assumed upper triangular
- **b** – an mxk-dimensional numpy array, with ith column containing b_i

Return x

an mxk-dimensional numpy array, with ith column containing the solution x_i

8.6 cla_utils.exercises8 module

cla_utils.exercises8.**Q1AQ1s**(*A*)

For a matrix A, find the unitary matrix Q1 such that the first column of Q1*A has zeros below the diagonal. Then return A1 = Q1*A*Q1^*.

Parameters

A – an mxm numpy array

Return A1

an mxm numpy array

cla_utils.exercises8.**ev**(*A*)

Given a matrix A, return the eigenvectors of A. This should be done by using your functions to reduce to upper Hessenberg form, before calling hessenberg_ev (which you should not edit!).

Parameters
`A` – an mxm numpy array

Return V
an mxm numpy array whose columns are the eigenvectors of A

`cla_utils.exercises8.`**`hessenberg`**`(A)`

For a matrix A, transform to Hessenberg form H by Householder similarity transformations, in place.

Parameters
`A` – an mxm numpy array

`cla_utils.exercises8.`**`hessenbergQ`**`(A)`

For a matrix A, transform to Hessenberg form H by Householder similarity transformations, in place, and return the matrix Q for which QHQ^* = A.

Parameters
`A` – an mxm numpy array

Return Q
an mxm numpy array

`cla_utils.exercises8.`**`hessenberg_ev`**`(H)`

Given a Hessenberg matrix, return the eigenvectors.

Parameters
`H` – an mxm numpy array

Return V
an mxm numpy array whose columns are the eigenvectors of H

Do not change this function.

8.7 cla_utils.exercises9 module

`cla_utils.exercises9.`**`get_A100`**`()`

Return A100 matrix for investigating QR factoration.

Return A
The 100x100 numpy array

`cla_utils.exercises9.`**`get_A3`**`()`

Return A3 matrix for investigating power iteration.

Return A3
a 3x3 numpy array.

`cla_utils.exercises9.`**`get_B100`**`()`

Return B100 matrix for investigating QR factoration.

Return A
The 100x100 numpy array

`cla_utils.exercises9.`**`get_B3`**`()`

Return B3 matrix for investigating power iteration.

Return B3
a 3x3 numpy array.

`cla_utils.exercises9.`**`get_C100`**`()`

Return C100 matrix for investigating QR factoration.

Return A
The 100x100 numpy array

`cla_utils.exercises9.get_D100()`

Return D100 matrix for investigating QR factoration.

Return A
The 100x100 numpy array

`cla_utils.exercises9.inverse_it(A, x0, mu, tol, maxit, store_iterations=False)`

For a Hermitian matrix A, apply the inverse iteration algorithm with initial guess x0, using the same termination criteria as for pow_it.

Parameters

- **A** – an mxm numpy array
- **mu** – a floating point number, the shift parameter
- **x0** – the starting vector for the power iteration
- **tol** – a positive float, the tolerance
- **maxit** – integer, max number of iterations
- **store_iterations** – if True, then return the entire sequence of inverse iterates, instead of just the final iteration. Default is False.

Return x
an m dimensional numpy array containing the final iterate, or if store_iterations, an mxmaxit dimensional numpy array containing all the iterates.

Return l
a floating point number containing the final eigenvalue estimate, or if store_iterations, a maxit dimensional numpy array containing all the iterates.

`cla_utils.exercises9.pow_it(A, x0, tol, maxit, store_iterations=False)`

For a matrix A, apply the power iteration algorithm with initial guess x0, until either

||r|| < tol where

r = Ax - lambda*x,

or the number of iterations exceeds maxit.

Parameters

- **A** – an mxm numpy array
- **x0** – the starting vector for the power iteration
- **tol** – a positive float, the tolerance
- **maxit** – integer, max number of iterations
- **store_iterations** – if True, then return the entire sequence of power iterates, instead of just the final iteration. Default is False.

Return x
an m dimensional numpy array containing the final iterate, or if store_iterations, an mxmaxit dimensional numpy array containing all the iterates.

Return lambda0
the final eigenvalue.

`cla_utils.exercises9.pure_QR(A, maxit, tol)`

For matrix A, apply the QR algorithm and return the result.

Parameters

- **A** – an mxm numpy array
- **maxit** – the maximum number of iterations

- **tol** – termination tolerance

Return Ak
the result

`cla_utils.exercises9.`**`rq_it`**(*A*, *x0*, *tol*, *maxit*, *store_iterations=False*)

For a Hermitian matrix A, apply the Rayleigh quotient algorithm with initial guess x0, using the same termination criteria as for pow_it.

Parameters

- **A** – an mxm numpy array
- **x0** – the starting vector for the power iteration
- **tol** – a positive float, the tolerance
- **maxit** – integer, max number of iterations
- **store_iterations** – if True, then return the entire sequence of inverse iterates, instead of just the final iteration. Default is False.

Return x
an m dimensional numpy array containing the final iterate, or if store_iterations, an mxmaxit dimensional numpy array containing all the iterates.

Return l
a floating point number containing the final eigenvalue estimate, or if store_iterations, an m dimensional numpy array containing all the iterates.

8.8 Module contents

PYTHON MODULE INDEX

INDEX