COMPUTING FOR SCIENCE AND ENGINEERING
Revised through end of Chapter 6

Robert D. Skeel

Contents

Preface

The purpose of the course is to expose students to computational concepts, tools, and skills useful for research in computational science and engineering, beyond what is learned in a first programming course (and basic mathematics courses).

Ideally computational science and engineering (CS&E) is practiced by using convenient input devices to define (a model of) physical reality and the questions that are being asked. Results are displayed in a way that most effectively conveys the information—usually graphically.

Even in the best cases, application-oriented software and environments require computational skills from the user. The user will have to write scripts for the application and for job control. He(/she) will have to choose algorithm parameters and make judgments about the accuracy of the results.

In less than best cases, the computational scientist will have to do some software development, even if "only" to modify the code. In the worst case, he(/she) may have to develop algorithms.

Prerequisites for this course are

- mathematical knowledge and maturity of an MS student in the physical sciences. (It is not assumed that the student is in the physical sciences, only that he(/she) has this level of mathematics.) In particular, some familiarity with matrix algebra is expected.
- programming experience in C, C++, Java, or Fortran or extensive scripting experience; also, commensurate computer skills.

There is no suitably comprehensive textbook; hence these notes with their links to web resources. A good reference is Software Carpentry. Some useful books:

- Python in a Nutshell, 2nd Edition, by Alex Martelli, O'Reilly, July 2006,
- The C Programming Language, 2nd Edition, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1988, ISBN 0-13-110362-8,

The World Wide Web is an essential resource for up-to-date information. To make searching more efficient, learn the advanced features of a search engine. For example, it is often useful is to narrow the search to a limited domain.

The course outline is

Part I: scripting to get things done

1. introduction to CS&E, Unix, software installation
2. files/text processing and mathematical computing in Python, including Numpy, graphics, floating-point arithmetic, computer algebra, object-oriented programming

   Part II: programming for performance

3. C programming, including data structures, interfacing to Python
4. parallel programming, including computer organization, MPI, OpenMP

   Part III: survey of other topics

5. regular expressions, developing software, GUIs, version control, automated builds

   Appendix: numerical computing

6. functions: polynomials, piecewise polynomials, trigonometric polynomials
7. matrices: operations, factorizations, parallelism

The appendix is a source of applications.

Chapter 1

Introduction

1.1 Role of Computing in Science and Engineering

Many say that computation has joined experiment and theory to become the third pillar of science; an alternative view is that computation is a tool of theory (and experiment).

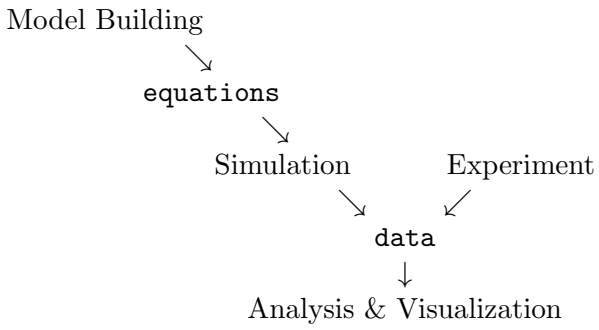
Three stages of scientific computing might be identified:

model building , which produces equations

simulation , which produces data, and

analysis

Experiment also produces data (for analysis):



Mathematics is central to science: scientific claims must be falsifiable, and only precise—i.e., mathematical—statements are falsifiable. Mathematics is central to computation: only precisely formulated instructions can be executed.

1.2 Using Unix

References: Software Carpentry, Basic Unix Shell, More Unix Shell; http://en.wikipedia.org/wiki/Shell_(computing).

The operating system handles the logistics of computing. Application programs interact with the operating system through function calls as specified by an *application program interface* (API).

There are three main flavors of Unix: Solaris, BSD, and Linux. Max OS X is partly based on BSD. Quoting from *Unix Power Tools* p. 1,

> UNIX . . . wasn't designed as a commercial operating system meant to run application programs, but as a hacker's toolset, by and for programmers.

(The word "hacker" is used here in its original benign sense to mean a computer enthusiast.) Unix tools are freely available for Windows in a collection called Cygwin.

POSIX is a standard for Unix operating systems. Solaris and OS X are POSIX compliant; most versions of Linux are not.

Using *virtualization software*, two operating systems can be run simultaneously.

Shells Users interact with the operating system through a program called a *shell*, either a *command line interface* or a *graphical user interface* (GUI). Examples of the latter include Windows Explorer and OS X Finder. The most popular command line shell is bash, which is the default for Linux and OS X. Also popular is csh/tsch. The bash prompt is "`$`", and the csh prompt is "`%`". To change your login shell, run the `chsh` command. The command line shell for Windows has been `cmd.exe`; beginning with Windows 7, it is PowerShell 2.0, which provides commands not so different from bash.

Command line interfaces are, for most purposes, not only faster than GUIs, but their use helps to prevent repetitive strain injury: "Avoid using the mouse and trackball whenever possible. Use keystrokes instead."

Navigating the file system A file is the basic unit of storage on a file system, e.g., a disk drive. These are hierarchically grouped into directories (or folders). Especially useful commands are

`pwd`
`ls` or `ls -l` or `ls -a` or `ls -al`
`cd` ⟨*name*⟩
`cd ..`
`cd /  # the root`
`cd ~  # home directory`
`cd ~myfriend`

An *absolute path* starts at the root; a *relative path* starts at the current working directory, denoted by a single dot. Navigating becomes more complicated for networked file systems, e.g., NFS, that span several machines. Be aware that some files and directories are merely links to (or aliases of) actual files and directories. The command `ls -F` distinguishes among files, directories, links, executables, etc.

Globbing The command

```
ls *.py
```

lists all files whose last three characters are ".py". The wildcard character "*" matches any string including an empty string. The character "?" matches any single character. Such a pattern is called a *glob*, and the expansion of such a pattern is called *globbing*. Section 6.4 introduces a more powerful language for patterns called regular expressions.

Command syntax A command begins with the name of the command followed by a list of zero or more arguments separated by spaces. Most commands have optional "keyword" arguments known as *flags*. The keyword (usually a single letter) is preceded by a hyphen and followed by the argument value, if any. Sometimes the position of the flag matters. Some generally useful flags for running programs are

- help, e.g., `python -h` , `grep --help` ,
- version, e.g., `python -V` , `gcc -v` , `mpirun -version` , `ipython -Version` ,
- verbose, e.g., `gcc` -v .

The last of these is useful for determining location of files and directories used by the program. One can use a limited set of emacs operations to edit the command line before hitting return: arrow keys, esc-B, esc-F, control-A, control-E, control-R, control-T, esc-D, esc-delete, control-D, delete, control-K, control-U, esc-L, esc-C, esc-U. Command completion uses the tab.

Modifying the file hierarchy The simplest way to create a short file is to enter

```
cat > myfile
```

and then sequentially enter the lines of the file terminated by control-D. To modify existing text in file, you need to use a text editor, which is discussed below. *Renaming* a file is done via

```
mv file1 file2
```

removing a file via

```
rm file1
```

and *copying* a file via

```
cp file1 file2
```

To make a new directory

```
mkdir tmp
```

to remove an empty directory

```
rmdir tmp
```

and to ("recursively") remove a directory and all its contents

```
rm -r tmp
```

Searching The command

```
find $HOME -name "*python*"
```

prints the path of each file in your home directory and its subdirectories whose name contains the string "python". To find those files in a larger set of files that contain occurrences of a string of

characters (and their locations within the files), use, e.g.,

```
grep "site-packages" *.py
```

The location of a *limited* set of existing commands and programs can be determined using the `whereis` command; e.g.,

```
$ whereis python
```

may reveal the location as

```
/usr/bin/python
```

The command/program that is chosen on the basis of your `PATH` value can be determined using the `which` command; e.g.,

```
$ which python
```

may reveal the location as

```
/Library/Frameworks/Python.framework/Versions/2.7/bin/python
```

To call the default python, you can use `$ /usr/bin/python`

Variables and scope Variables can be defined to have strings as their values, e.g.,

```
x=one+2
export y="three, 4"
```

If we then run a new shell, only those variables that are exported will be known. And they will be known *only* in the shell in which they are defined and in descendants of that shell. Try

```
bash
echo $x
echo $y
x=five,6
```

If we exit the new shell,

```
exit
```

the first value of `x` is restored.

Environment variables Their values can be seen by entering

```
printenv
```

An important variable is `PATH`, whose value is a colon-separated list of directory paths, which are searched for command names.

Job control The `jobs` command shows those processes initiated by the user from the command line. To refer to a process by its number in the jobs listing, precede that number by `%`, e.g., `kill %2` terminates the second job listed. To interrupt a running program, enter control-Z. To resume execution, enter `fg` (foreground). To abort a computation, control-C often works. To run a job in background, follow the command with `&`.

File compression A cross-platform way for compressing files and directories is to use

`zip` ⟨*name*⟩ ⟨*list*⟩

for compressing

`unzip` ⟨*name*⟩`.zip`
for uncompressing.

I/O redirection and pipes Commands that normally print their output to a display device can have it redirected to a disk file using the > operator. Appending is done using >>. Error output is redirected using `2>`. Similarly, a command expecting input from the keyboard can be redirected to a file using the `<` operator. Connecting the output of one command to the input of another can be done using a *pipe*:

```
ls | grep .c
```

1.2.1 Text editors

To modify a file, you need to use a text editor like

`vim myfile` or `emacs myfile`

Here is a complete list of editors useful for Python source code.

In emacs, you use the arrow keys to navigate and you do insertions simply by typing them in. To save updates, enter control-X, control-S, and to quit, enter control-X, control-C. In writing programming language source code, it is convenient to use the tab key for indenting. However, these should be converted to spaces by entering

`ESC-x untabify`

at end of file. You can customize emacs by putting Lisp commands into the `.emacs` file in your home directory, e.g., change the number of spaces associated with the tab key for C code

```
(setq-default c-basic-offset 3)
```

You can also make emacs aware of the syntax of the code in a file, if the file name has the appropriate extension. To do such things, search the web for a solution.

1.2.2 Integrated development environments

Reference: http://en.wikipedia.org/wiki/Integrated_development_environment.

An integrated development environment (IDE) includes language-aware editors, compilers/interpreters, build systems, debuggers, etc.

IDLE is an example of a Python-specific IDE.

Microsoft Visual Studio and Xcode (for OS X) each support several programming languages.

The best examples of multi-language cross-platform IDEs are *Eclipse* and *Netbeans*, though the latter does not currently support Python.

1.2.3 Shell scripts

Creating commands You may create a command from a shell script. The simplest way is to put commands into some file `myscript` and enter "`source myscript`." Using the `source` command is unnecessary if the first line is

```
#!/bin/bash
```

and the file is made executable by entering the command

```
chmod 700 myscript
```

at the shell prompt, in which is it enough to enter `./myscript`. If the current directory "`.`" is in `$PATH`, you need enter only `myscript`. The first digit specifies your permissions and the other two digits the permissions of other users. Each digit is a weighted sum of 4, 2, and 1 where

```
4 read   2 write   1 execute
```

To make the file executable (and readable) by others, use `755` instead.

Providing arguments As an example, consider the secure copy command. It is used to copy between two machines, e.g.,

```
scp -Cpr me@myothercomputer.purdue.edu:file1 file2
```

If `file1` is a relative path, it is understood to be relative to ∼`me`. To have a shorter command with usage

```
here2scholar file1 file2
```

you can create the shell script `here2scholar`

```
#!/bin/bash
# e.g., here2scholar calc.py Scripts
scp -Cpr $1 me@scholar.rcac.purdue.edu:$2
```

as an executable file in the `bin` subdirectory of your home directory of your desktop computer. Note that `$1` and `$2` represent arguments 1 and 2. Here is a script `scholar2here` that does the opposite

```
#!/bin/bash
# e.g., here2scholar Scripts/calc.py .
scp -Cpr me@scholar.rcac.purdue.edu:$1 $2
```

The `.profile` and `.bashrc` files These file contains commands that are executed when you open a new shell. If it is a "login shell", the `.profile` file is executed; otherwise the `.bashrc` file is. (With OS X each new `Terminal` window is a login shell.) In csh the `.login` and `.cshrc` files serve the same purpose. These files contain commands like

```
export PATH=~/bin:${PATH}:/usr/local/bin
export EDITOR=emacs
alias mv="mv -i"
```

It is likely that you want your `.profile` file to do everything that your `.bashrc` does (but not vice versa), which can be accomplished by putting the line "`source .bashrc`" at the end of the `.profile` file. The `source` command executes a file of shell commands. Into the `.bashrc` file, put commands, such as `alias`, whose effect is suppressed upon opening a new shell. Most other startup commands go into the `.profile` file. A change to the `.bashrc` or `.profile` file takes effect only upon opening a new shell or login shell, respectively (unless you execute a `source` command).

1.2.4 Unix tools

As an example, the `awk` command

```
awk '{if ($1=="ENERGY:") print $2 " " $11}' < all.data > select.data
```

reads from `all.data` and for each line having "`ENERGY:`" as its first item it prints the second and eleventh item of that line to `select.data`.

Supplementary note

To be able to `ssh` without a password, try this.

An alternative way to compress files is to combine a packing process with a compression process:

1. To pack a set of files and directories into a single file, use
   `tar -cf -` ⟨*list*⟩ `> name.tar`
   and to unpack
   `tar -xf name.tar`

2. To compress a file, use
   `gzip name.tar`
   and to uncompress
   `gunzip name.tar.gz`

Review questions

1. What is the Unix shell command for changing to the parent of the current directory?

2. Suppose

```
$ which python
/usr/bin/python
$ ls -l /usr/bin/python
lrwxr-xr-x  1 root  wheel  72 Nov 16  2007 /usr/bin/python@ -> ../System/Li
brary/Frameworks/Python.framework/Versions/2.5/bin/python
```

   What would be the absolute path for the `python` binary?

3. What is the effect of the command
   mv file1 file2 if `file2` does not exist? if `file2` does exist?

4. Give a Unix shell command for removing all files having extension `.pyc`.

5. Assume that the result of entering the Unix command "`ls -aF`" is
   `./ ../ .py 1a.py a12.py b.py c2.py/ dx.py`
   Which of these items would be listed if "`ls [a-z]*.py`" were entered?
   `a12.py b.py c2.py dx.py`
   If "`ls [a-z]?.py`" were entered?
   `c2.py dx.py`

6. Give Unix shell commands for creating a directory `MyPy` and copying each file with extension `.py` into that directory.

7. Give the Unix shell command for printing the value of a variable `myvar`.

8. The default name for the executable produced by the C compiler is `a.out`. If the command `a.out` does not work, what should one try next?

9. What is the name of the environment variable that lists directories which are searched to find a command name?

10. What is the effect of `chmod 644 myfile`? of `chmod 600 myfile`?

11. How should you set the permissions to make a file executable by any user but readable and writeable only by the owner?

12. Let `mycmd` be a file in the current working directory containing commands. If I enter `mycmd` at the bash (Unix) command line, what two requirements must be met for the commands to be executed?

13. In order for a command `mycmd` in the current working directory to execute in Unix, what two requirements must be met?

14. In Unix, how do you typically kill a job running in foreground? suspend a job running in foreground?

15. $*$ Give a Unix command for creating a `tar` file consisting of all files in the current directory with extensions `.tex`, `.eps`, and `.bbl`.

16. What does the `source` command do?

1.3 Installation of Software

Installation of software is done in three ways:

1. from a binary. This is typically performed by point-and-click and generally seems to require that you have superuser/administrator privileges.

2. from source at the command line. If you have superuser privileges, you can preface the install command with `sudo`. If not, you can install in `$HOME/local` using an option like `--prefix=$HOME/local` or `--home=$HOME/local`. In such case, make sure that `$HOME/local/bin` is in your path.

3. from source using a *package manager*. Ubuntu has one. Several other flavors of Linux use RPM (Red Hat Package Manager).

The installation process includes

> download, unpack, uninstall*, configure, build, install, test.

If there is an old version having the same name, it might be necessary to uninstall it (remove or rename it). Much of what is downloaded can be discarded after you are satisfied with the installation.

1.3.1 Unix, C compiler, and OpenMP

Some of this software might already be installed.

Needed is a C compiler that supports OpenMP and variable dimension arrays, e.g., version 4.2 or later from `www.gnu.org`. Test with the program `ohello.c`, adapted from Wikipedia:

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello World from thread %d\n", id);
        #pragma omp barrier
        if ( id == 0 ) {
            int nthreads = omp_get_num_threads();
            printf("There are %d threads\n",nthreads);
        }
    }
    return 0;
}
```

as follows:

```
$ export OMP_NUM_THREADS=4
$ gcc -fopenmp  ohello.c -lgomp
$ ./a.out
```

Platform specific notes follow:

Linux

For Ubuntu you might start with `http://www.ubuntu.com/desktop`.

OS X

There are instructions here. The first steps are to download Xcode and find the updated command line tools. During the downloading and installation, you get little feedback.

See `http://hpc.sourceforge.net/` for many useful binaries, in particular, gcc 4.9.

Note that the compiler that comes with Xcode is an older version of clang/LLVM, which does not support OpenMP. The current version, clang 3.8.0, does support OpenMP, and you are free to install this. (Using clang 3.8.0 with MPI is may be difficult, but MPI is used here only with Python.)

Windows

For Windows users, a probably better choice than Cygwin is to install Ubuntu. Perhaps this is best done by visiting http://www.ubuntu.com/desktop.

1.3.2 The Anaconda Python Distribution and MPI

Consider installing the comprehensive collection of tools and libraries included in the Anaconda 4.1.1 package. In particular, it includes Python 3.5.2, NumPy 1.1.11.1, scipy 0.17.1, matplotlib 1.5.1, IPython 4.2.0, and sympy 1.0. Go to https://www.continuum.io/downloads. Installation of Matplotlib can be tested by running IPython and entering

```
[1] import matplotlib.pyplot as plt
[2] plt.ion()
[3] plt.plot([1, 2, 3])
```

It seems that the Anaconda distribution for Linux includes MPI and `mpi4py`. For other platforms, see below.

Test your installation of `mpi4py` by entering

```
$ mpiexec -n 4 python mpihello.py
```

at the operating system command line, where `mpihello.py` is the script

```
#!/usr/bin/env python
from mpi4py import MPI
import sys
hw_msg = "Hello, World! I am process %d of %d on %s.\n"
myrank = MPI.COMM_WORLD.Get_rank()
nprocs = MPI.COMM_WORLD.Get_size()
procnm = MPI.Get_processor_name()
sys.stdout.write(hw_msg % (myrank, nprocs, procnm))
```

Also, test with the C program `hello.c`

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int npes; MPI_Comm_size(MPI_COMM_WORLD, &npes);
    int myrank; MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Hi there from %d of %d\n", myrank, npes);
    MPI_Finalize();
}
```

as follows:

```
$ mpicc hello.c
$ mpiexec -n 4 ./a.out
```

If you have a second installation of MPI in `/usr/local/bin`, it will be hidden by that of Anaconda. You can, if needed, undo this by appending

```
export PATH=/usr/local/bin/:$PATH
```

to your `.profile` file. If your MPI installation is OpenMPI, you can determine the version by entering

```
$ ompi_info --version
```

Linux

If you want your own separate installation, the `.rpm` file at http://www.open-mpi.org/software/ompi/v2.0/ might work for you.

OS X

To install `mpi4py`, you might try entering

```
$ conda install mpi4py
```

though this currently does not seem to work. Assuming this fails, try this:

```
$ conda remove mpi4py
$ conda install pip
$ pip install mpi4py
```

and/or this:

```
$ conda remove mpi4py
$ conda install --channel mpi4py openmpi=1.10.2 mpi4py
```

A certain other problem may crop up, which can be eliminated by entering

```
$ sudo ln -s ~/ananconda /opt/anaconda1anaconda2anconda3
```

Finally, there are instructions for an Open MPI install on Mac OS X. *However*, if a directory `/usr/local/lib/openmpi` already exists, remove or rename it before doing the installation.

Windows

For Windows, first try installing MS-MPI, and go to the Web for further advice.

1.3.3 ∗ The Enthought Python Distribution

As an alternative, you might consider installing the comprehensive collection of tools and libraries from the Enthought Canopy Express distribution. Go to http://www.enthought.com/products/getepd.php.

1.3.4 ∗ Separate Installation of Python and Packages

Some of this material is outdated.

This is another alternative.

For a package system for Ubuntu Linux, try http://packages.ubuntu.com/. There are also package systems for other Unix distributions, including Debian, Fedora, SUSE, fink, FreeBSD, and OpenBSD.

The latest version of Python are 2.7.8 and 3.4.1. Major changes were made between versions 2.5.x and 3.x. However, some packages may not have upgraded yet, so there are Versions 2.6.x and 2.7, which contain features present in both of these. If you want to install Python, proceed carefully and search the web for advice.

Download Python 2.7, from http://www.python.org/download/. Using `which python`, determine whether 2.7 is the default or whether `python` references a pre-existing installation, in which case `python2.7` might work. For installation and use of some software, you need the location of the actual python binary:

```
python -c "import sys;print sys.prefix"
```

Installation of packages is facilitated by EasyInstall. For upgrading an existing installation, use the `--upgrade` flag.

Obtain NumPy 1.8.1 from http://sourceforge.net/projects/numpy/files/NumPy/. Afterwards, execute "`import numpy; numpy.test()`". You may wish also to install SciPy, available from http://sourceforge.net/projects/scipy/files/.

For a plotting program, use Matplotlib. Read http://matplotlib.sourceforge.net/users/installing.html and install from http://sourceforge.net/projects/matplotlib . If you do not have administrator permissions for your machine, use
`python setup.py install --home $HOME/local` .

Quite useful for interactive use and code development is IPython. You can install it and other packages such as `mpi4py` using

```
sudo easy_install ipython[kernel,security,test]
```

After installation open a new Terminal window and type

```
ipython
```

Installation instructions for IPython are at http://ipython.org/install.html The latest version is 2.1.

Of possible interest is Spyder, "the Scientific PYthon Development EnviRonment."

1.3.5 ∗ MPI for Python

To install `mpi4py`, see http://mpi4py.scipy.org/docs/usrman/install.html. A simple way to install it is

```
pip install mpi4py
```

1.4 Use of Rosen Center Linux Cluster

1.4.1 Remote access and file transfer

You may choose to use the Rosen Center Red Hat Linux cluster `scholar`. Here is documentation for Scholar. In particular, there is information on remote access and file transfer. In particular to login in with support for X11 windows, use either

```
ssh -Y myusername@scholar.rcac.purdue.edu
```

or

```
ssh -X myusername@scholar.rcac.purdue.edu
```

Also, the same web site gives links to file transfer GUIs for Windows. Alternatively, from a Unix shell you can use the `scp` command described in Section 1.2.3.

1.4.2 Python

To use Python on the Scholar cluster, first enter

```
module load anaconda/4.4.1-py35
```

Then use `python3.5` for execution or `ipython3` for interaction. There are numerous other Pythons on Scholar, but they are missing some packages.

Chapter 2

Python

Python is a scripting language suitable for a multitude of tasks. The advantages of typical scripting languages over programming languages like C and Fortran are that (i) there is no need to declare variables, (ii) there is no need to compile first, and (iii) they provide powerful operations and a rich set of functions.

Python is a full-featured programming language which together with extensions offers the features of both Matlab and Perl with only modest reduction in convenience. It is a scripting language suitable for a variety of high-level tasks. An example is given in Langtangen(2006), Section 2.4, of using Python to conduct a battery of numerical experiments and present the results on web pages. It is also suitable for rapid protyping of algorithms later to be translated into a more efficient programming language. And it is useful for gluing together functions written in several programming languages.

Python was created by Guido van Rossum, Benevolent Dictator For Life, at CWI in Amsterdam in 1991. It is free and has an unrestrictive license.

Python includes a standard library. In addition, numerous extensions have been developed, most important of which is NumPy, which permits efficient array operations, essential for scientific computing.

Review question

1. What is the current name of the package necessary for efficient scientific computing with Python?

2.1 Interactive Computing

References: Software Carpentry: Programming with Python and Python Tutorial. The tutorial has links to the documentation, and starting from the tutorial is an excellent way to access the documentation. The SciPy lecture notes Getting started with Python for science discuss not only Python but also IPython, NumPy, and Matplotlib.

Python can be used interactively in various ways:

best by typing "ipython" from the command line.

better by launching IDLE, which is an integrated development environment GUI. Either double-click its icon or enter "`idle -n &`" (or equivalent) from the command line. The "`-n`" flag is needed if using Matplotlib.

good by typing "python" from the command line.

Interaction with Python at the most basic level is illustrated below:

```
$ python
>>> import math
>>> math.sin(3.141592)
6.53589793076e-07
```

2.1.1 Importing a module

The Python `import` command can be used in various ways:
with `import math` you can use `math.sin`, `math.cos`. etc.
with `from math import sin` you can use `sin` but `math.cos`, etc. are unavailable,
with `from math import *` you can use `sin`, `cos`, etc.
The last of these should be used sparingly. It may be convenient to put such commands into an interactive startup file.

To continue a statement to the next line put a backslash at the end of the line.

2.1.2 Documentation

There is online Python 3.4.3 Documentation. At the operating system command line, type `pydoc`; at the Python command line, `help`.

2.1.3 Lists

An example of a `list` is

```
names = ['dick', 'jane', 'spot']
```

An interactive example is

```
>>> ns = list(range(10)); ns
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(ns)
10
>>> ns[3: ]
[3, 4, 5, 6, 7, 8, 9]
>>> ns[:3]
[0, 1, 2]
```

Lists are to be distinguished from Numpy arrays:

```
>>> from numpy import *
```

```
>>> array(range(1, 11))
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

2.1.4 Strings

```
str.split('12 34')
str.join(' ', ['12', '34'])
str.replace('102 -100 = 0102', '102', '___', 1)
```

Note that in strings "\" is an escape character, which changes the meaning of what follows, e.g. `\a` is bell, `\b` is backspace, `\n` is newline, `\r` is carriage return, and `\t` is tab. Use `\\` to denote a backslash. Associated with each character is a nonnegative integer value, which can be obtained with the `ord` function. To get a character from a number, use the function `chr`. ASCII characters (printable and control) are in range 0–127.

Unicode is a standardization of most of the world's character sets, in which each character is assigned a nonnegative integer called a *code point*. These are typically written in hexadecimal, e.g., the Greek letter Π has the code point `U+03A0`. The integers 0 through 127 are assigned to the ASCII characters. The standard does *not* specify how these characters should be encoded. The most popular encoding is *UTF-8*, which encodes a character using anywhere from 1 to 4 bytes: ASCII takes 1 byte; characters in most Western alphabets take 2 bytes. Most programming languages can handle Unicode. Python 3 strings consist of unicode characters. As an example, `print('\u03a0')` prints Π.

2.1.5 Data

Boolean

There are two values of type `bool`: `True` and `False`.

None

This is a special value available to the programmer. It is not the same as being undefined, e.g.,

```
x = None; x
```

returns nothing, whereas

```
del x; x
```

is an error. (The `del` command makes a variable undefined.) A variable can be tested for equality to `None`.

Numbers

Python supports the following types of numbers: `int`, `float`, and `complex`. Values of type `int` can be expressed in hexadecimal by prepending `'0x'` to the hexadecimal digits, e.g. `'0xFF'` is 255. The `float` type of Python is really a double precision number. It is recommended that *conversions between different types of numbers be explicit*, because this assists in documenting number types, which is important due to the approximate nature of `float`ing-point arithmetic.

Container objects

Lists can be created element by element using brackets or from the `range` function. Also useful are the constructions:

```
zeros = [0]*10
map(float, mylist)
```

and operations that mutate lists:

```
del mylist[i]
list.reverse(mylist)
list.sort(mylist)
```

Other useful list operations are

```
i = list.index(mylist, 'junk')
'junk' in mylist
```

A tuple is like a list except that it is immutable and constructed with parentheses, which can be omitted when there is no ambiguity. Here is an example where the tuple is on the left-hand side:

```
a, b, c = range(3)
```

A 1-tuple is written thus: "`12,`".

Strings that cross lines should be enclosed in triple quotes:

```
'''This string
crosses a line'''
```

Useful constructions include

```
'string'[1:2]
str.upper(mystring)
```

Strings are immutable.

Lists, tuples, and strings are special types of container objects known as *sequences*, because they are indexed by consecutive integers beginning with zero. One can select parts of a sequence `x`, e.g., odd-numbered items can be selected with `x[1::2]`. Such slice designations can be explicitly created,

```
odd_ones = slice(1, None, 2)
```

and used as in `x[odd_ones]`. The arguments of the `slice` function are called the *start*, *stop*, and *stride* values.

A dictionary is a container object in which entries are accessed by a *key* rather than a nonnegative integer index. For example if

```
sqrt = {1:1, 4:2, 9:3}  # a dictionary
```

one uses `sqrt[1]`, `sqrt[4]`, and `sqrt[9]` to get values.

Python provides array capabilities via the `array` module. We do not use this; instead, we use the `ndarray` type provide by NumPy.

2.1.6 Operators

Here is a list of operators and their precedence.

A boolean expression is not always evaluated completely. It is evaluated only up to the point needed for determining the result, e.g.,

```
sharpstart  =  len(l) > 0 and l[0] == '#'
```

does not evalute the pare if the first part is `False`, thereby avoiding an `IndexError`. This feature (of most programming languages) is called *short circuit evaluation*.

Review questions

1. What is the value of `sys.argv`, in the case where
   ```
   merge.py 1 '2 3'
   ```
   has been entered.
2. Write a Python code segment that constructs a list of (double-precision) `float`s with integer values from 0 to `n`.
3. For each of the types `bool`, `int`, `float`, `str`, `tuple`, `list`, `NoneType`, which value yields false in an `if` clause?
4. Give examples of two (builtin) types of objects that are mutable.
5. Illustrate how the `*` operator can be used to construct a list.
6. Illustrate how to access the last 4 characters of a string `mystring`.
7. Create a list of a thousand No's.

2.2 Developing a Script

An example from Langtangen(2006): file `hw.py` containing

```
#!/usr/bin/env python
import sys, math
r =  float(sys.argv[1])
s = math.sin(r)
print("Hello, World! sin(" + str(r) + ")=" + str(s))
```

can be executed by entering

```
./hw.py 1.4
```

Scripts can be developed using your favorite editor and running them either at the command line—or inside IPython:

```
$ ipython
In [1]: run hw.py 1.4
Hello, World! sin(1.4)=0.985449729988
In [2]: edit hw.py
```

To edit without running the code afterwards use instead `edit -x`. The "`run`" and "`edit`" commands are IPython extensions to Python. Also very useful is "`history`." Here are tips for using IPython effectively.

If you wish Python to execute commands at the beginning of every script and every interactive session, put these commands into a file called `sitecustomize.py` into the `site-packages` directory.

To find the location of this directory,

```
>>> import site
>>> site.getusersitepackages()
```

2.2.1 Control structures

References:

Indentation is part of the syntax of Python:

```
>>> for name in names:
...     print(name)
...
dick
jane
spot
```

Creating a list with embedded for loop:

```
>>> [float(n) for n in range (1, 11)]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]
```

This construct is referred to as a *list comprehension*. To create a loop that uses pairs of elements from two lists `n` and `x` of the same length, it is convenient to use

```
for ni, xi in zip(n, x):
```

If `n` were simply the indices of the elements of x, `range(len(x))`, use

```
for ni, xi in enumerate(x):
```

This also works for dictionaries.

For more general loops, use `while` ...: .... To execute a loop "N plus one half" times, use the construct

```
while True:
    ...
    if ...: break
    ...
```

It is good form to have exactly one normal exit from a loop.

Selection can be done with `if` ...: ...`elif` ...: ...`else`: .... The bodies of each part of a control structure are indented. Use of function definition and conditional:

```
>>> def heaviside(x):
...     if x > 0.:
...         return 1.
...     elif x < 0.:
...         return 0.
...     else:
...         return 0.5
...
>>> heaviside(0.)
0.5
```

For each data type there is typically only one value that tests as false, e.g., `False`, `0`, `0.`, `None`, `''`,

`[]`.

2.2.2 Objects

An *object* is a unit of data residing in memory. Objects come in different types, e.g., `int`, `float`, `str`, `list`, `tuple`, `dict`, `bool`, `long`, `NoneType`. Each type of object has a *constructor* for creating it, the name of the constructor being the same as the name of the type, e.g., the use of `list(range(10))` to construct a `list`. Associated with a type or *class* of objects are *methods* for operating on them, e.g., `list.reverse` and `list.sort`. The first argument of such a method must be an object from the designated class.

Object-oriented style The following pairs are equivalent:

```
str.split('12 34')              '12 34'.split()
str.join(' ', ['12', '34'])     ' '.join(['12', '34'])
list.append(names, name)        names.append(name)
```

Since the type of an object is known, the name of the type can be replaced by the object.

2.2.3 Reading and writing

Reading from a file and writing to a file:

```
ifile = open('data', 'r')
ofile = open('results', 'w')
for line in ifile:
   items = str.split(line)
   ofile.write(items[2] + '\n')
ifile.close()
ofile.close()
```

The `open` command constructs a file object, which is a representation of the file in computer memory. The `for` statement can be used to iterate over a variety of objects, including files as well as `lists`. There is also a file method `writelines` for writing the items of a list (without adding line terminators). Lines in Windows files end with "`\r\n`". Go here for conversion utilities.

File objects File objects come in different types, e.g., a file object constructed by an `open` in `'r'` or `'w'` mode is of type `io.TextIOWrapper`. The following pairs are equivalent:

```
io.TextIOWrapper.write(ofile, line)       ofile.write(line)
io.TextIOWrapper.close(ifile)             ifile.close()
```

except that the short form does not require `import`ing the `io` module.

String formatting The string formatting (or interpolation) operator is important for output.

Input Following are functions for reading: `file.readline`, `file.readlines`, `file.read`. If `ifile` is a readable file object, the conversion `list(ifile)` is equivalent to `ifile.readlines()`.

Input from the terminal is from file object `sys.stdin`. For more information see "7.2 Reading and Writing Files"

Example of a filter With this much knowledge of Python, we can write a script equivalent to the awk script of Section 1.2.4:

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    items = line.split()
    if items != [] and items[0] == 'ENERGY:':
        print(items[1], items[-4])
```

A program such as this which reads from `stdin` and writes to `stdout` is called a *filter*.

Review questions

1. Define a function `comma2gether` which takes as an argument a list of one or more strings and returns a single string that joins together the strings in the list using a comma and a space. There are several ways of doing this, some very simple.

2. If file `echo.py` has contents

   ```
   import sys
   for item in sys.argv:
       print(item)
   ```

   what is printed by the following?

   ```
   $ python echo.py 1  2.0 three
   ```

3. Give a Python expression that yields a string representation of a `float`ing-point value `x` to 6 decimal places with no exponent. (The format code is "`f`" and goes at the end of the format specification.)

4. Give a Python expression that converts a `float` to a `str`ing using up to 1000 digits and a possible exponent. (The format code is "`g`".)

5. Find an equivalent statement without using the `print` function. Assume `sys` has already been `import`ed.

   ```
   print('x =', x)
   ```

6. Which file object does `print` write to? and what extra things does `print` do to the output that `write` does not do?

7. What is the type of the value returned by each of
   `file.read`
   `file.readline`
   `file.readlines`
   Which of these remove newline characters?

8. What is the alternative objected-oriented syntax for `file.close(myfile)` ?

2.3 Variables and Functions

2.3.1 Variables and data

Note that *variables do not have types; rather it is their values that have types.*

Determining an object's type

To determine the type of a variable's value, use the `isinstance` function.

Attributes

Associated with virtually all objects are `attributes`, which are specified by identifiers. The value of an attribute is obtained as follows:

⟨*expression*⟩`.`⟨*identifier*⟩

Because the dot operator has high precedence, the expression may have to be enclosed in parentheses. The specific attributes associated with an object depend on its type. Modules are objects and they may have data attributes, e.g.,

`sys.argv` `math.pi`

and function attributes

`sys.exit` `math.sin`

Types are objects and they can have method attributes, e.g.,

`str.find`

Other objects like strings and files have method attributes, e.g.,

`'abc'.find` `myfile.close`

These examples are far from exhaustive.

The list of attributes of an object, say `str`, is available as `dir(str)`.

2.3.2 Assignment and copy

A Python variable is a *reference* to an object. To be more specific, a Python variable contains the address of an object in memory. What happens in the following

`x = 2; y = x; x = 3.`

is that `x` first contains the address of `2`, then this address is copied into `y`, and finally the address of `3.` is put into `x`. The actual object consists of both a reference to its type and its data.[1]

[1] It also includes a count of the number of references to it for garbage collection purposes.

A `list a = [2., 3., 5.]` is itself a sequence of references, each one of which might refer to a different type of object:

```
a
|
[ . , . , . ]
  |   |   |
'two' 3   5.
```

The assignment `b = a` would make `b` a reference to the same list as `a`. On the other hand

```
b = a[:]
```

would make `b` refer to a copy of the list but not of the objects that are referenced by `a[0]`, `a[1]`, `a[2]`. If these objects are mutable and copies of them are also desired, `import copy` and set

```
b = copy.deepcopy(a)
```

If `a` is a dictionary, use `b = dict.copy(a)` to make a shallow copy.

Each object `x` has a unique ID `id(x)`, typically its address in memory. Whether or not variables `x` and `y` refer to the same object can be checked with

```
x is y
```

To save storage space, the compiler sometimes stores two *immutable* objects having the same value in the same memory. Therefore, the identity operator `is` *should not be used* to compare two immutable objects: instead, use the equality operator`==`.

2.3.3 Functions

Call by reference

Consider the code

```
def zero (x, y):
   x = 0.
   for i in range(len(y)): y[i] = 0.
a = 1.
b = [2., 3., 5.]
zero(a, b)
```

When the function `zero` is called, the references in `a` and `b` are copied to local variables `x` and `y`, respectively. The function changes `x` to refer to `0.`, *but there is no change* to `a`. The function *does not* change `y`. It changes just the references constituting the list, and this same list is referenced by `b` as well as `y`. Upon exit, the local variables become undefined. The intended effect is successful for `b` but not `a`. The modification of `b` is an example of a *side effect*. As discussed in Section 2.4.3, side effects are to be avoided.

All Python functions return a value, the default is `None`.

Argument lists

Python permits *keyword arguments*, which are optional labeled arguments. e.g., the matplotlib function call

```
plot(x, y, linewidth=1.0)
```
It also permits a variable number of arguments as in `min(a, b, c)`.

Passing function names

A couple of examples are
```
myplot(f, -1., 1.)
myplot(lambda x: x*x, 0., 5.)
```
The `lambda` operator constructs a function without naming it. It is limited to functions that can be defined with a single expression. The equivalent mathematical notation is $x \mapsto x^2$.

Top-down design

A script organized as follows

```
def main():
  ...
# define other functions
...
...
main()
```

better conveys the intent of the computation.

Scope

As discussed in the preceding paragraphs, the parameters of a function have only local scope. In the following code
```
def func(x):
   a = 1.
   return x + a + b
a = 2.; b = 3.
func(5.)
```
the variable `a` is a local variable because it is defined in its first occurrence, whereas `b` is a global variable because it is used without first being defined. (An uninitialized variable will be assumed to come from the enclosing static context but not otherwise from its calling function.) The presumption that `a` is local can be overridden by declaring it to be `global` before using it.

The list of defined variable names in the current scope is available as `dir()`.

Review questions

1. Give alternative syntax for `list.append([2, 3], 5)`. What is the return value?
2. Give an example of a data attribute? a function attribute? Your examples must be either for builtin modules or objects or those from the standard library.
3. What is the result of the following code?

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = a[:]
print(b is a, b[0] is a[0], b[0][0] is a[0][0])
```

If a were a dictionary, what should be used instead of `b = a[:]`?

4. For `a = [1]*3; b = [2]*3`, what is the result for each of the following (considered separately): (i) `a = b`, (ii) `a[:] = b`, (iii) `a = b[:]`, (iv) `a[:] = b[:]`.

5. What function from what module should we use to make a completely separate copy of an object?

6. Describe what is printed:

```
def zero(a, b):
   a = 0.
   for i in range(len(b)): b[i] = 0.
x = 1.; y = [2., 3., 5.]
zero(x, y)
print(x, y)
print(a, b)
```

7. Describe what is printed:

```
def func(x):
   a = 1.
   return x + a + b
a = 2.; b = 3.
print(func(5.))
```

8. What is the name of the mechanism by which Python passes an argument? What does this mean?

9. In what two situations would would a variable `x` in a function be considered to be global?

10. Construct a Python expression for the function $x \mapsto x^2 + 1$. without giving it a name.

2.4 Developing a Module

Reference: Software Carpentry: Defensive Programming

For example, if file `hwm.py` contains

```
def hw(r):
   import math
   s = math.sin(r)
   print("Hello, World! sin(" + str(r) + ")=" + str(s))
```

the following would result:

```
$ python
>>> import hwm; hwm.hw(3.141592)
Hello, World! sin(3.141592)=6.53589793076e-07
```

A convenient way to develop a module is to make it a script file with the function (and variable) definitions followed by tests inside a block headed by

```
if __name__ == '__main__':
```

These are executed if the module is run as a standalone script but not if it is imported. The individual testing of functions that constitute a program is known as *unit testing*.

If instead, you are testing the module interactively, you can use the IPython `edit` command. Exiting the editor causes the contents of the file to be executed, including definitions.

The use of `import` is *not recommended* for a module that you are modifying. The `import` command loads a copy of the module into memory *only if it is not already loaded*, and and creates a reference to it. A `reload` is needed to overwrite the existing copy. Loaded modules are listed in the `sys.modules` dictionary. (An additional limitation of `import` is noted in Section 3.4.2.)

2.4.1 Exception handling

Python provides special constructs for catching errors. Here is an example:

```
try:
    i = string.index('x')
    print(string[i+2:])
except ValueError:
    pass
except Exception as e:
    print(e, 'string =', string)
```

If the `index` method fails to find an `'x'`, it raises a `ValueError`. Ordinarily this would halt execution and produced an error message. The `except` clause squelches both actions. For any other exception, the error message and supporting information is assigned to the variable `e.args`, and this is printed as well as the value of the offending `string`. For further information, see also the Python Tutorial: 8. Errors and Exceptions.

Input checking is important. The Python `raise` statement is useful for this.

2.4.2 Verification

For checking the correctness of the program, the `assert` statement is useful If the object of the assertion is False, an `AssertionError` is `raise`d. If Python is called with the optimization option `-O`, assertions will be omitted from the code. An example follows:

```
def grid_index(xg, x):
    """Return the index of the cell in xg containing x.

    Return the first i>0 such that xg[i-1] <= x <= xg[i] where
    it is ASSUMED that the values in xg are nondecreasing floats"""
    if not isinstance(x, float):
```

```
        raise ValueError(str(x) + " is not a float")
    if x < xg[0] or x > xg[-1]:
        raise ValueError(str(x) + " is out of range")
    i = 1
    while x > xg[i]: i = i + 1
    assert xg[i-1] <= x <= xg[i], "bug!"
    return i
```

2.4.3 Side effects

If a function call changes state—modifies one of its arguments, modifies a global variable, or does output—it is said to have a side effect. The main effect is the value returned by the function. Generally, side effects are to be avoided. An important exception is that a *method* may modify its very first argument (the object to which it is "bound"), in which case, the function name should use an imperative verb. Section 2.6 discusses how the user can define his(/her) own types and associated methods. Additionally, the use of global variables, even if not modified, is to be avoided in general.

2.4.4 Debugging

If running from IPython, an aborted execution can be analyzed by entering

```
debug
```

This can be made to happen automatically by first turning

```
pdb on
```

From IPython, a script can be run executed by the debugger using the command `run -d`. Respond to the initial debugger prompt by entering `c` (as instructed). Help is available by entering `help`. You can look at a listing with line numbers using `list` and "`list 1`". You can set a *breakpoint* at, say line 4, with "`break 4`" and cancel this with "`disable 4`". To run the program, enter `continue`. If it pauses at a breakpoint, you may `print` the values of Python expressions. You can then `step` through the program or `continue` it. Exit the debugger using `quit`.

Tip. To understand unexpected behavior, create the simplest possible program that exhibits the error.

2.4.5 Coding standards

Python Style Guide

2.4.6 Documentation

A Python *docstring* is a string at the beginning of a module or function, which serves as documentation available at runtime via the `help` function or the attribute `__doc__`.

There exist tools that automatically produce documentation from documented source code, e.g., `doxygen` generates documentation as web pages or text for programming languages such as Python and C.

Review questions

1. For the Python syntax
   ```
   try:
       ⟨part1⟩
   except ⟨part2⟩, e:
       ⟨part3⟩
   ```
   explain the following:
   (a) What is ⟨*part2*⟩?
   (b) What is ⟨*part3*⟩ for?
   (c) What is the value of variable `e`?
2. Where is a Python docstring placed? What role does a docstring serve?

2.5 Operating System Commands

Operating system commands can be executed using generic Python commands:

```
import os
os.remove('junk')
os.listdir(os.getcwd())
import glob
glob.glob('*') + glob.glob('.*')
```

The documentation on files and directories is a good place to look for other `os` functions.

Alternatively, operating system commands can be executed using shell commands:

```
import subprocess
subprocess.call('ls', shell=True)
subprocess.call('gnuplot sine.gplt', shell=True)
```

You can use `sys.platform` to determine the operating system.

For making a temporary file name, you can use

```
name = tempfile.mktemp(dir=os.getcwd())
```

Returned is a path as a string giving a path that can be used to create a file. Between the time the name is created and the file is created, there is no absolute guarantee that the name might not be used by some other process executing concurrently.

Review questions

1. What module contains functions for navigating the file hierarchy and otherwise interacting with the operating system?
2. What is `os.path.join('tmp', 'work')` for Unix? for Windows?
3. Assuming that `glob` has been `import`ed, what does `glob.glob('*.bib')` return?

2.6 Object-oriented Programming

Programming languages are often classified as being either procedural, object-oriented, or functional. The object-oriented approach is well suited for "reactive" computing such as GUIs and transaction processing. A functional programming style is a better fit for "transformational" computing more typical of science.

A basic idea of object-oriented programming is that of defining new types of objects in ways in which the details of the implementation are hidden. As an example, consider an adaptation of the `SparseVector` class in version 3 of Software Carpentry. First, here is an illustration of usage:

```
from mySpVec import SparseVector
x = SparseVector(60)
x[18] = 3.
x[45] = 7.
print(x[33])
y = SparseVector(60)
y[8] = 11.
z = x + y
w = x.dot(y)
```

Before explaining the implementation, it is helpful to discuss the Python feature of *special methods*. Practically every operator and many builtin functions have special functional forms, which are useful for defining new types of objects. This illustrated by the following examples, in which `a` and `b` are `floats`, `x` is a `list`, and `i` is an `int`:

`a + b`	`float.__add__(a, b)`	`a.__add__(b)`
`str(a)`	`float.__str__(a)`	`a.__str__()`
`x[i] = a`	`list.__setitem__(x, i, a)`	`x.__setitem__(i, a)`
`x[i]`	`list.__getitem__(x, i)`	`x.__getitem__(i)`

The constructor, as used in `x = SparseVector(6)`, might be defined as follows:

```
class SparseVector(object):
    def __init__(self, len):
        self.data = {};
        self.len = len
```

The call to `SparseVector` creates an empty object and then initializes it using the optional special method `__init__`, whose first argument is the object just created. The use of the variable name `self` for the first argument of a method is a convention, not a requirement.

Also, part of the definition of a class are various other methods. The assignment to an item of an object, illustrated by `x[2] = 3.`, might be defined as follows:

```
    def __setitem__(self, index, value):
        if not isinstance(index, int):
            raise KeyError, 'non-integer index to sparse vector'
        self.data[index] = value
```

The indexing operation, illustrated by `print(x[4])`, might be defined as follows:

```
    def __getitem__(self, index):
        if index in self.data:
            return self.data[index]
        return 0.
```

The addition operation, illustrated by `z = x + y`, might be defined as follows:

```
    def __add__(self, other):
        result = SparseVector(self.len)
        result.data.update(self.data)
        for k in other.data:
            result[k] += other[k]
        return result
```

The dot product, illustrated by `w = x.dot(y)`, might be defined as follows:

```
    def dot(self, other):
        result = 0.
        for k in self.data:
            result += self[k] * other[k]
        return result
```

Absent from the `SparseVector` class in Wilson(2009) is the following special method

```
        def __str__(self):
            strself = [str(self[i]) for i in range(self.len)]
            return '[' + ', '.join(strself) + ']'
```

for defining the `print`ed form of the object.

Note that Python attributes can be added on the fly to a user-defined object within a class definition *and outside of it*.

Object-oriented programming embraces three principles:

1. encapulation,
2. inheritance, and
3. polymorphism.

The motivation for these properties is software reuse.

Encapsulation means hiding the implementation details, allowing access to the data of an object only through its methods. Abstract data types like the stack are good examples of this if implemented using an opaque pointer like the C implementation of a stack in Chapter 4. Such an implementation provides methods like `push`, `pop`, and `isempty` without any hint of whether a linked list or an array stores the contents of the stack. Encapsulation means that certain data attributes of an object ought not to be directly accessed by the user. Such attributes can be marked as *private* by beginning their names with a *single underscore*.

Inheritance permits the definition of subclasses, which inherit the methods of the base class without having to redefine them. For example, a matrix class might have a square matrix subclass

with the subclass inheriting the definition of a matrix product and implementing an inverse method just for square matrices.

Polymorphism allows functions whose argument types are not known until execution time, at which time an appropriate function is called. This property is automatically part of Python but not of C (in any straightforward way). As an example, suppose that the variable `seq` has a value either of type `str` or `list`, and a count is desired of the number of occurrences of the one-character string `'a'`. Without polymorphism, this would be done by testing the type:

```
if isinstance(seq, str): count = str.count(seq, 'a')
elif isinstance(seq, list): count = list.count(seq, 'a')
```

This is unnecessary in Python because one can use `type(seq).count(seq, 'a')` or simply `seq.count('a')`.

You can emulate the `struct` mechanism of C by defining the following minimal Python `class`:

```
class struct_(object): pass
```

You can then create a `struct_` object and, as indicated previously, add attributes at will. Structs are good for bundling together related variables so they can be passed to a function as a single argument in a parameter list. Use of this mechanism is *not* object-oriented programming.

To illustrate another possibility, consider the following:

```
class Polynomial(object):
    def __init__(self, coeff=[]):
        self.coeff = coeff[:]
    def __call__(self, x):
        px = 0.  # use Horner's rule
        for coeff in reversed(self.coeff):
            px = px*x + coeff
        return px
```

The following illustrates its use:

```
p = Polynomial(coeffs=[0., 1., -1./2., 1./3.])
print(p(1.1))
```

The call to `p(1.1)` is effectively transformed to `Polynomial.__call__(p, 1.2)` after determining that `p` is a `Polynomial` object. If wanted, the `Polynomial` can be extended by defining additional methods. For example, it might be useful to define a method `deriv(self, x)` that returns the derivative of `self` at `x`.

Review questions

1. If a method `dot(self, other)` is defined for some class `Vector`, how is this method actually applied to two objects `u` and `v` where `u` is an instance of the `Vector` class? Give your answer as a Python expression giving both the common short form and the full long form.

2. Define the method `__setitem__(self, index, value)` for the `SparseVector` class (without any error checking).

3. Define the method `__getitem__(self, index)` for the `SparseVector` class.

4. Define the method `dot` for the `SparseVector` class.

5. Explain why the definition of `__setitem__` for the `SparseVector` class is not a circular definition.

6. What are the three principles of object-oriented programming?

7. What does encapsulation mean?

8. What is the advantage of making a class a subclass of some other class (other than `object`)?

9. Given the implementation of the `Polynomial` class, how would you construct a polynomial `p` for $1 + x + x^2/2 + x^3/6$? How would you evaluate `p` at $x = 1$?

10. Define a `Polynomial` class having (i) a constructor
    ```
    p = Polynomial(c)
    ```
    which constructs the polynomial $c_0 + c_1 x + \cdots + c_n x^n$ whose coefficients are given as the elements of a list `c` and (ii) a `__call__` method such that `p(x)` evaluates a polynomial `p` at `x`. The construction of `p` should be such that it is not mutated if the list `c` is later modified.

Chapter 3

Python Extensions

Python has the functionality to make it an excellent scripting language for numerical computation—if used with Numpy.

Computing time in seconds can be calculated by `import`ing the `time` module and comparing the value of `time.time()` before and after the computation.

3.1 NumPy

For information about NumPy see `http://docs.scipy.org/doc/`, in particular, Tentative NumPy Tutorial and NumPy for MATLAB Users. If NumPy has been installed, it can be `import`ed under the name `numpy`.

3.1.1 Array constructors

An array provides an efficient way to store a set of values of the same type. Python has an `array` type, but with limited functionality; NumPy has an `ndarray` type, with greater functionality. A NumPy array can be constructed by applying its `array` function to an appropriate list. There are also more direct constructions. For example,

```
z = zeros((3, 3), 'f4')
```

creates a 3 by 3 array of single-precision zeros (4-byte floating-point values). For type `int` use `'i4'` and for double precision omit the second argument. Another example is

```
a = arange(0., 5., 0.5)
```

which creates an array with values from `0.` up to but not including `5.` with increments of `0.5`. To obtain the intended result, use only decimal values that are machine numbers; otherwise, there may be an extra element at the end of the array due to roundoff error, e.g., `arange(0., 1., 1./49.)`. An alternative construction that avoids such problems is

```
a = linspace(0., 48./49., 49)
```

Arrays have a number of attributes, e.g, `dtype` and `flags`. Some of these such as `shape` can be

changed, e.g.,

```
a.shape = (2, 5)
```

Multidimensional arrays are not structured in the same way as nested lists. An array object contains but a single reference to a block of memory containing its data, as well as addition indexing information for specifying the locations of its elements. The location of any element is calculated using indexing information stored in the array object. To copy an array `a`, it suffices to use `ndarray.copy(a)` or `a.copy()`; there is no need for a "deepcopy."

reduction operations A reduction operation applied to an array creates a new array of reduced rank. Useful reduction operations include `numpy.sum`, `numpy.prod`, `numpy.max`, `numpy.min`, `numpy.argmax`, and `numpy.argmin`. You can specify the index axes along with the reduction is performed.

3.1.2 Indexing and slicing

An array is indexed as `a[1, 3]` (though `a[1][3]` also works). One can select parts of an array using slices, e.g., odd-numbered rows and columns can be selected with `a[1::2, 1::2]` or with `a[odd_ones, odd_ones]` where `odd_ones = slice(1, None, 2)`.

Omitted indices are assumed to be trailing indices and their values are assumed to be the slice ":", e.g., if `a` is an array of shape `(3, 3)`, `a[1]` is equivalent to `a[1,:]`.

The identity operator `is` seems to behave in an anomalous manner for arrays:

```
>>> from numpy import array
>>> a = array([2, 3, 5])
>>> a is a
True
>>> a[0] is a[0]
False
>>> id(a[0]) == id(a[0])
True
>>> a.data is a.data
False
>>> id(a.data) == id(a.data)
True
```

3.1.3 Views

It is important to note that if an array section is assigned to variable, e.g., `row = a[i, :]` or `row = a[i]` where `a` is 2-dimensional, then `row` is assigned to a new object that references the same data as the object assigned to `a`. By the same principle, `b = a[:]` causes `b` to reference the same data as `a`.

To see how this is implemented consider the following example:

```
import numpy as np
a = np.arange(1., 17., dtype='float32').reshape(4, 4)
```

The array `a` is allocated continguous memory for 16 single precision values:

```
 1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16.
```

It is evident that an `ndarray` having more than one dimension is *not* a nested container. The statement

```
b = a[1:4:2,1:4:2]
```

creates an array of shape `(2,2)` using a subset of these same memory locations:

```
 6.  8. 14. 16.
```

To index elements of `b`, it is enough to know the location of its first element and the row and column strides. The column stride is the increment that must be added to a memory address to move from one column to the next. Since each value occupies 4 bytes, the column stride is $2 \cdot 4 = 8$. Similarly, the row stride is $8 \cdot 4 = 32$. In this way, `a` and `b` are different views of the same data. Strides are given by the `stride` attribute of an `ndarray`. To transpose `b`, it is enough to swap the row and column strides!

3.1.4 Array operations

There is a set of operations known as *ufuncs* (Universal Functions) which can be applied element-wise to arrays; these include arithmetic operations and elementary functions. For example, if `a` and `b` are arrays of the same dimension

```
a = 2.*a + b
```

assigns to `a` a new array defined in the obvious way. More generally, you can perform a binary operation with two arrays if they have the same trailing dimensions, e.g., if `a.shape == (4, 5)` and `b.shape == (5,)`, it is permitted to write `a/b`.

For efficiency, it may be preferable to mutate an existing array rather than construct a new one. For example

```
a[:] = 2.*a + b
```

overwrites array `a` with the result.

Using array-level operations rather than looping through elements of an array greatly improves efficiency, a technique known in this context as *vectorization*.

The object `ufunc` is a class: it has objects like `add`, `subtract`, ..., `power`. For binary ufuncs, there are special methods such as `ufunc.reduce` and `ufunc.outer` that are of considerable utility. A `ufunc.outer` method creates an array of greater rank from two arrays by applying the operator to pairs of operands from the given two arrays.

3.1.5 Fancy indexing

A relational ufunc creates an array of type `bool`. Applying `where` to such an array creates a tuple of `int`eger arrays containing array indices that correspond to `True` values. This tuple can be used to access those particular elements of an array. For example,

```
a[where(a < 0.)] = 0.
```

sets negative values to zero in the array `a`.

3.1.6 Input/output functions

text data Recommended for I/O are the NumPy I/O functions `savetxt` and `loadtxt`. These functions take as their first argument either the name of a file, in which case opening and closing are done automatically, or a file object. The use of a file object is recommended. The `loadtxt` function ignores empty lines.

binary data Up to now, we have been using only text I/O. For files that need not be human-readable, it is less wasteful and simpler to store values in binary rather than converting them into text. Binary data in is units of bytes. Although all numerical data types store their values in binary, `bytes` objects keep the data as a contiguous block of bytes. The binary data can be extracted from an `ndarray` with the `bytes` constructor, e.g., `bytes(a)`. The result is a Python `sequence` object whose elements are single bytes. To convert a string to bytes, use its `encode` method, e.g., `s.encode()`.

endianness For binary data, one might need to contend with the issue of *endianness*: little-endian format orders the bytes from least significant to most significant, and big-endian format does the opposite. These terms arise from the disagreement between Lilliput and Blefescu.

```
>>> bytes(np.ones(3, dtype='int32'))
b'\x01\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00'
```

Note the reversal of the byte order. Little endian format is characteristic of Intel hardware.

binary I/O (with sequential access) Building on a previous example,

```
a = np.arange(1., 17., dtype='float32').reshape(4, 4)
ofile = open('a.dat', 'wb')
ofile.write(bytes(a))
ofile.close()
```

Only the bytes are written. To read it back in again, we need to know the data type and the shape:

```
ifile = open('a.dat', 'rb')
a2 = np.frombuffer(file.read(), dtype='float32').reshape(4,4)
ifile.close()
```

memoryview The `bytes` constructor makes a copy, which can be inefficient. Python provides an object of type `memoryview` that provides a `bytes` view of data without making a copy. In the example above, one could use instead

```
ofile.write(memoryview(a))
```

random access I/O NumPy provides a very convenient mechanism for storing an array on disk. Here is an example:

```
a = np.arange(1., 17., dtype='float32').reshape(4, 4)
a.tofile('a.dat')
a2 = np.fromfile('a.dat').reshape(a.shape)
```

These operations should used together as a pair. Unlike sequential access I/O, random access I/O operations are applicable only to disk files and not to `stdin` or `stdout`.

3.1.7 Matrix operations

NumPy matrix operations include `dot`, `outer`, `identity`, and `diagonal`. The `dot` product is defined for any pair of arrays for which the last dimension of the first array is equal to the second last dimension of the second array, e.g., `C = dot(A, B)` would mean

$$c_{ijk} = \sum_l a_{il} b_{jlk}$$

if A has two indices and B has three. The outer product of two column vectors, uv^T, can be computed by `outer(u, v)`.

Supplementary note

The use of an operator such as `*=` to assign a value to an array has the effect of overwriting the elements of the array, e.g., `a *= 2.` is equivalent to `a[:] = 2.*a`.

Review questions

1. How does one construct a one-dimensional array of ten double precision zeros?
2. What is the shape attribute of
   `a = array([[1., 0., 1., 0.], [0., 1., 0., 1]])` ?
   How can this be converted to a one-dimensional array without rewriting the array?
3. Given that `a` is a two-dimensional array, how can one construct a two-dimensional array consisting of those elements of `a` whose row index is even and column index is odd.
4. What is the difference between `a = 2.*b + c` and `a[:] = 2.*b + c` ?
5. How can the operation `a = 2.*a` be accomplished without creating a new array to store `2.*a` ?
6. Vectorize the innermost two loops of the following algorithm by replacing it by a loop-less assignment statement using appropriate `numpy` function(s). The outer loop must not be changed.

```
c = numpy.zeros((m, p))
for k in range(n):
    for i in range(m):
        for j in range(p):
            c[i, j] += a[i, k]*b[k, j]
```

7. Let `a` be a `numpy.array`. Write a "vectorized" Python function that returns an array of those indices `i` for which `a[i]` is nonzero.

8. (3 points) Let `x = numpy.array([3., -2., 0., 4., -1.])`. What is the value of `x >= 0.`? of `x[x >= 0.]`? of `numpy.where(x >= 0.)`? of `x[numpy.where(x >= 0.)]`?

3.2 NumPy Modules

3.2.1 Pseudo-random numbers

Functions for generating *pseudorandom numbers* are in the `random` module of the NumPy package. Here are a several of them:

```
random.rand(d1, d2, ..., dk)    # uniform [0, 1[
random.randn(d1, d2, ..., dk)   # normal(0, 1)
random.randint(a, b, shape)        # equal probability for [a:b]
```

Here `d1`, `d2`, ..., `dk` are array dimensions. To have a program that gives reproducible results, use the `random.seed` function before calling a random number generator. This is particularly useful for *regression testing* (checking that changes to the code do not introduce bugs).

3.2.2 Linear algebra functions

The module `numpy.linalg` includes a variety of such functions. See Appendix B.

Review questions

1. In a Markov chain Monte Carlo method, the Metropolis criterion is to accept a new configuration `xnew` with probability $\min\{\texttt{ratio}, 1\}$; otherwise to use again the old configuration `x`. Write a Python function
   ```
   def next(x, xnew, ratio):
   ```
   that returns either `x` or `xnew` depending on values of `ratio` and `mumpy.random.rand()`.

2. Explain how to generate a random point uniformly distributed on the rectangle $a \le x \le b$, $c \le y \le d$ given a random number generator for a uniform distribution on the interval $[0, 1]$.

3. (9 points) Complete a Python function `die_roll()` so that it returns the roll of a die (a cube with faces labeled from 1 to 6). You must use only the library function `numpy.random.rand()`. Note that `int` gives the integer part of a `float`.

3.3 Matplotlib

Matplotlib is first in the list of plotting packages on the "Topical Software" page of `scipy.org`. It provides Matlab-style plotting functions. A package is a collection of defined objects including functions, modules, and subpackages. It exists in the file system as a directory. For interactive use, the subpackage `pyplot` is suggested. An example of its use follows:

```
import matplotlib.pyplot as plt
import numpy as np
plt.ion()
data = np.loadtxt('temp_vs_t')
t = data[:, 0]
temp = data[:, 1]
plt.plot(t, temp, 'o')
plt.close()
```

Here the file `temp_vs_t` contains a `t` value and a `temp` value on each line. Use `plt.ion` at the beginning with IPython or an interactive script; use `plt.show` at the end if there is no interaction. More than one plot can be viewed at a time using the `plt.figure` function. For a filled contour plot use `plt.contourf(x, y, z)` with gridded values arranged as follows:

```
y[2]  z[2,0]--z[2,1]--z[2,2]--z[2,3]
         |       |       |       |
y[1]  z[1,0]--z[1,1]--z[1,2]--z[1,3]
         |       |       |       |
y[0]  z[0,0]--z[0,1]--z[0,2]--z[0,3]

        x[0]    x[1]    x[2]    x[3]
```

You can explicitly set the range using

```
plt.axis([xmin, xmax, ymin, ymax])
```

otherwise, it is determined by the data. For setting the x- and y-limits individually use `xlim` and `ylim`. For plotting a curve, as opposed to a function, use

```
plt.axes().set_aspect('equal')
```

to avoid distortion of shapes. Saving an electronic of the current figure can be done by commands like `plt.savefig('myfig.eps')`, `plt.savefig('myfig.pdf')`, and `plt.savefig('myfig.png')` provided the file extension is a supported format. See the *Beginner's Guide* for more information.

Other popular plotting programs for which there are Python interfaces include gnuplot and xmgrace.

The `tri` module Given scattered data in 2D or higher, it is often required (e.g., for piecewise linear interpolation) to determine a triangulation of the convex hull of the nodes consisting of triangles that are as fat as possible. The appropriate triangulation is most often the Delaunay triangulation. Here are two alternative characterizations of a Delaunay triangulation for a set of points in the plane:

empty circle property The circle that circumscribes each triangle contains no nodes in its interior.

dual of Voronoi diagram The Voronoi diagram associates with each node the set of points that are closer to that node than to the others. The triangulation is obtained by connecting each pair of nodes whose Voronoi cells touch.

Computing a Delaunay triangulation is an example of the problem that arise in *computational geometry*. There are efficient algorithms for computing a Delaunay triangulation, but they are not

simple. The module `matplotlib.tri` has a function

```
center, ends, nodes, neighbors = delaunay(x, y)
```

which, given the coordinates of a set of points, returns four arrays such that

1. `center[k]` are the coordinates of the circumcenter of triangle `k` (the center of the circle that circumscribes the triangle),
2. `ends[j]` are the two points `i1, i2` that define edge `j`,
3. `nodes[k]` are the three points that define triangle `k`, and
4. `neighbors[k]` are the three or fewer triangles that share an edge with triangle `k`. The value `-1` represents the absence of a neighbor. The list of neighbors is ordered so that `neighbors[k][n]` is opposite node `nodes[k][n]`.

where `x[i], y[i]` are the coordinates of point `i`. The concepts of a Delaunay triangulation and a Voronoi diagram generalize to dimensions higher than two.

Scientific visualization is an important part of CS&E. It is used both for exploring data and for presenting data. Here are tutorials from Boston U. A good package is the Visualization Toolkit (VTK), for which there is a Python interface called Mayavi2, included in the Anaconda Python distribution.

Supplementary note

In your file system there is a directory `matplotlib`, which can be located using the function `matplotlib.get_configdir()`. In it, you can create a file `matplotlibrc` containing directives.

3.4 Other Extensions

3.4.1 Computer algebra

There is a free, easy-to-install Python library called SymPy, which is part of the Enthought Python distribution. SymPy includes the arbitrary precision package mpmath mentioned in Section 6.3.4. To install unpacked downloaded source, change to the directory `sympy-0.6.5` and

```
$ sudo python setup.py install
```

To test it, change to the directory `examples` and

```
$ python basic.py
```

Assuming you have a directory `~/bin`, change back to the parent directory and

```
$ mv bin/* ~/bin
```

Then to run Python with sympy preloaded enter

```
$ isympy
```

The `sympy-0.6.5` directory can be discarded unless you want to save `examples`.

There is a comprehensive mathematical system based on Python called SAGE, which includes many existing open-source packages including the pioneering Maxima computer algebra system.

3.4.2 SciPy

SciPy is a collection of Python packages for scientific computing. The excellent Fortran package LAPACK is included as `scipy.linalg.lapack.flapack`. These routines are built on optimized Basic Linear Algebra Subroutines available as `scipy.linalg.blas.fblas`. An example is `scipy.lib.blas.fblas.saxpy`, which is a single precision implementation of $ax + y$ where a is a scalar and x, y are vectors. There are other packages with names `interpolate`, `integrate`, and `sparse`.

Note. Most `SciPy` packages require an "explicit import" meaning, e.g., `import scipy.linalg` instead of just `import scipy` . Simply importing `scipy` loads mostly just `numpy` functions. To learn about such hidden packages, enter `help(scipy)` .

Chapter 4

C Programming

Reference: *The C Programming Language*, 2nd Edition by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1988. For library functions, see http://www.cplusplus.com/reference/clibrary/.

The greatest performance is obtained with programming languages that use variables of declared type and compile into machine language executables.

Here we use C, because it is the "lowest common denominator" of popular high performance languages like Fortran and C++. It was originally developed for writing systems software by the original AT&T, and because of the shortcomings of Fortran 77, its use spread to writing application software. C was standardized internationally in 1990. The standard was revised in 1999 to include variable-dimensioned multidimensional arrays and complex numbers.

Free C compilers are available from the Free Software Foundation. It supports OpenMP. The latest version for C/C++ is GCC 4.6.1. The C compiler `gcc` implements most of C99.

Here is a very simple C program:

```
#include <stdio.h>
int main(int argc, char **argv){
    printf("Hello, world!\n");
}
```

If there are no arguments, one may use `main(void)`. Note the (complicated) type declarations of the parameters `argc` and `argv`.

If the program is run from a bash command line, its return value it obtained from `$?`.

For tutorials or references, consider http://en.wikipedia.org/wiki/C_(programming_language), or http://www.iso-9899.info/wiki/Web_resources.

4.1 Declarations and Control Structures

A C variable labels a fixed memory location large enough to hold the type of value(s) for which it is declared.

```
char a, b, c;
int n;
n = 10;
double s;
s = 2.718281828;
s = s + 1.;
```

A variable may be initialized when it is defined.

The most important types of executable statements are the following: an expression statement, a compound statement, a while loop, a for loop, and an if else construct. To execute a loop "N plus one half" time, use the construct

```
while (true) {
   <statements>
   if (<condition>) break;
   <more statements>}
```

Syntactically, the `if (...) ... else if (...) else ...` pattern is a nested construction, but, as far as indentation is concerned, it is better to treat it like a Python `if elif else` construct:

```
if (<condition1>)
   <statement1>
else if (<condition2>)
   <statement2>
else
   <statement3>
```

The `if (...) if (...) ... else ...` construction leads to what is called a *dangling else*. In C, the ambiguity is resolved by associating the `else` with the nearest `if`.

When calling a function, the value of the argument (in the calling program) is *copied* to the memory location associated with the parameter (in the called program). Assigning a new value to the parameter has no effect on the argument—nor should it, since there is no requirement that an argument should be a valid target for assignment. (The argument might be a constant or an expression.) For example,

```
#include <stdio.h>
#include <stdbool.h>
int reciprocate(double x) {
   if (x != 0.)
      {x = 1./x; return true;}
   else
      return false;}
int main(void){
   double x = 3.;
   if (reciprocate(x))
      printf("%f\n", x);
   else
      printf("undefined\n");}
```

prints the value `3.0000000`. To achieve the desired effect, we introduce pointers.

Review questions

1. Write a "Hello, world!" program in C.
2. What header file must be included to use the C function `printf`? `atof` or `atoi` or `abs`? `pow`?
3. What are the possible terminating chars of a statement?

4.2 Pointers, Arrays, and `malloc`

4.2.1 Pointers

Here is continuation of the example from the preceding section:

```
double *ps;
ps = &s;  // address "operator"
printf("%f\n", *ps);  // deferencing operator
```

Note that declarations are inverted in C—*declaration reflects use*. Instead of declaring `ps` to be a pointer to a double, we declare its de-referenced value, `*ps`, to be a `double`.

To change the value of a argument in a function call, one should pass the address of the argument:

```
#include <stdio.h>
#include <stdbool.h>
int reciprocate(double *x) {
   if (*x != 0.)
      {*x = 1./*x; return true;}
   else
      return false;}
int main(void){
   double x = 3.;
   if (reciprocate(&x))
      printf("%f\n", x);
   else
      printf("undefined\n");}
```

4.2.2 One-dimensional arrays

An array in C is a contiguous block of memory. For example,

```
double x[10];
```

defines `x` to be a block of memory just large enough to store 10 `double`s. In most contexts, an array *decays* to a pointer to the location of its first element. For this example `x` decays to a value of type `double` $*$. Hence, its first element is `*x`. The remaining elements are `*(x+1)`,..., `*(x+9)`, where the increments are multiples of `sizeof(double)`, which is the number of bytes required to store a `double`.

The pattern `*(pointer + offset)` occurs very often, so C provides the more concise syntax `pointer[offset]`. For the given example, this provides the alternative syntax

```
// x[0], x[1], ..., x[9]
```

An array differs from a pointer in at least a couple of respects:

1. The array itself is associated with a constant memory location and cannot be assigned a value; only its elements can be assigned values.

2. The `sizeof` operator gives, in bytes, the size of the actual array, not that of its address.

The definition `double x[10];` allocates memory for `x[0]`, `x[1]`, ..., `x[9]` but not for the address `x` itself, since it is constant. On the other hand, the definition `double *y;` allocates memory for the pointer `y` but none for an object to which it might point, i.e., no memory is allocated for `y[0]`. The address `x` can be assigned to a variable (or passed as an argument to a parameter) of type `double *`, i.e., a pointer to a `double`.

C99 permits an array of variable dimension to be defined:

```
double x[n];
```

An array of fixed dimension may be initialized when it is defined:

```
double z[5] = {0., 1., 2., 3., 4.};
```

Unfortunately, the array notation is allowed only in initializations.

A word of caution: *Whether or not a calculated address is within bounds is not checked at runtime.* It is up to the programmer to keep track of upper bounds.

A parameter of a function that is defined to be an array is actually a pointer: storage is allocated only for an address, not a sequence of a values. So a parameter definition `double x[n]` or `double x[]` is in reality of type `double *x`. If the form `double x[n]` is used, the value of `n` is ignored. It follows then that the argument corresponding to an array parameter must be an address, e.g., if the calling function defines `double y[q]` and wants to "pass" it to a function having the parameter `double x[n]`, the argument corresponding to the parameter `double x[n]` could be `&y[0]`—or, equivalently, `y`. It is also permitted to pass some other array element, e.g., `&y[1]`, as the argument. As another example, the definition `char **argv` is the same as `char *argv[]`.

As stated previously C declarations are inverted. To unpack them, read them from the inside out. For example,

```
int (*b)[7];  -->  int( ( *b )[7] );
                   -->  ((b is a pointer to) an array of 7) ints
```

and

```
int *b[7];  -->  int(  *( b[7] ) );
                 -->  ((b is an array of 7) pointers to ) an int
```

4.2.3 Multidimensional arrays

A multidimensional array is more complicated.

A C multidimensional array

```
double a[m][n];
```

is implemented as a linear array of consecutive storage locations with multidimensional indexing. In most contexts, a 2-dimensional array *decays* to a pointer to a 1-dimensional array. For the example above, the value of `a` is again an address, and it can be assigned to a variable (or passed as an argument to a parameter) of type `double (*)[n]`, i.e., a pointer to `n` doubles. (Indexing has higher precedence than deferencing.)

For a 2 D arrays, such as `a`, the number of rows `m` is needed only for the initial allocation of memory.

If a function parameter `b` is defined as `double b[p][q]` or `double[][q]`, it is in reality of type `double (*)[q]` and `p` is ignored. The value of `q` is required for index calculations. Again, the argument corresponding to an array parameter must be an address, e.g., if the calling function defines `double a[m][n]`, the argument corresponding to the parameter `double b[p][q]` could be `&a[0][0]`—or, equivalently, `a`. It is also permitted to pass some other array element, e.g., `&a[1][1]`, as the argument.

For `a` defined above, we can create a "view" of the lower right `m-1` by `n-1` block as follows:

```
double (*d)[n] = (double (*)[n])&a[1][1];
```

The logic of this is a follows: `&a[1][1]` is a pointer to a `double`, and `n` is appropriate because it is the stride (in units of `double`s) between rows of `d`.

De-referencing a 2 D array merely changes its type to a 1 D array (and this in most contexts decays to a pointer). For the example above, `a[1]` is an array of type `double[n]` with address `&a[1][0]` (and in most contexts decays to a value of type `double` $*$.

A pointer to an array is not permitted as a return type.

Arrays with more than 2 dimensions can be defined, in which case more than one stride value must be provided to enable the compiler to do address calculations.

Note that `double (*a)[n]` is not the same as `double **a`: in the former case `a[1]` is the address `a + n*sizeof(double)`; in the latter case it is the address `a + sizeof(double *)`.

4.2.4 malloc

```
x = (double *)malloc(n*sizeof(double));
// elements are *x, *(x+1), ..., *(x+9)
free(x);  // prevents memory leak
```

The call to `malloc` returns a pointer of type `void *` to a block of memory just large enough to store 10 `double`s. This *generic pointer* is then *cast* to one of type `double *`, which instructs the compiler how to do address calculations. Thus a pointer can serve as an array if memory is allocated. To use functions `malloc` and `free`, it is necessary to put `#include stdlib.h` at the beginning.

To create a 2 D array dynamically, use

```
double (*a)[n] = (double (*)[n])malloc(m*n*sizeof(double));
```

This allocates `m*n` doubles from the heap and 1 pointer from the stack. The value `[n]` is the stride needed for double index calculations.

The memory available to a program consists of several segments: read-only, unprotected static, the heap, and the stack. Read-only storage is for code and constants. Unprotected static storage is for variables defined outside of functions. The stack is for (most) variables defined inside a function and grows and shrinks as functions are entered and exited. An array points to memory allocated from the stack. The heap is from where `malloc` gets memory and to where `free` releases it. (Over time it becomes fragmented.)

Memory allocated from the stack for a variable, including an array, is *de-allocated* upon exit from the function (or block) in which the variable is defined. A pointer to such a variable should never be returned by a function.

Review questions

1. In words, what does
   ```
   char (*a)[20]
   ```
   declare the variable `a` to be?

2. How would you declare `y` to be a pointer to an array of `double`s?

3. Answer the following questions in the form of C code: (a) Show how to allocate memory *from the heap* for `n` double precision values. Show both the declaration of the variable and the allocation of the memory. (b) What should be done after this memory is no longer needed?

4. Using the function `sizeof`, give a C expression for the amount of storage required for `x` defined by
   ```
   double *x[4]
   ```

5. What are the two ways of indexing contiguous memory locations for a pointer `px` to that memory?

6. Does the C definition
   ```
   double *x[4]
   ```
   mean that `x` is a pointer to an array of `double`s or an array of pointers to `double`s?

7. Write a C function `void setToZero(...)` which can set the value of a `float` variable to `0.F`. Then show how this function can be used to set the value of a `float` variable `x` to `0.F`.

8. Suppose that

   ```
   double x[20];
   double *y = (double *)malloc(20*sizeof(double));
   ```

   Which of the following statement would then be either illegal or lead to unpredictable behavior?

   (a) `x = y;`

   (b) `y = x;`

(c) `*(x+1) = 1.;`

(d) `y[1] = 1.;`

(e) `return x;`

(f) `return y;`

9. Consider the two definitions
```
int (*b)[7];
```
and
```
int *b[7];  // equivalently int *(b[7]);
```
Answer the following questions, assuming a 64-bit computer:

(a) For each definition, give a numerical value for the number of bytes allocated (on the stack) in memory.

(b) For each definition, what is the number of bytes represented by `1` in the expression `b + 1`?

(c) For each definition, what is the type of `*b`?

4.3 Functions and Programs

4.3.1 Definitions vs. declarations

For top-down design, it is desirable to define a function after its use. However, in C every name must be declared before use. This can be achieved by declaring a function before use by means of a *prototype* and defining the function later:

```
void setToZero(double y);
int main(void){
    double x = 1.;
    setToZero(x)
    printf("%f\n", x);
}
void setToZero(double y) {
    y = 0.;
}
```

The *declaration* of a name is to be distinguished from its *definition*. A declaration gives information that the compiler needs. In a function declaration, the body of its definition is replace by a semicolon. Defining a name not only declares it but also generates code for the object associated with the name. A name can be declared several times but must be defined exactly once.

A variable can be defined and initialized outside any function definition.

4.3.2 Function pointers

It can be useful to "pass" a function as an argument to another function, sometimes referred to as a *callback*. As with an array, C does not permit functions to be passed, but does allow a pointer

to a function to be passed. And as with an array, a function decays to a pointer to the location of the function in most contexts.

The parameter in the calling function that corresponds to the function name must be declared to be a pointer to a function. The type of the function pointer must exactly match that of the corresponding argument: the return types must match, the number of arguments must match, and the types of each argument must match. The declaration of a function pointer `fp` would be identical to that of an actual function `f` to which it might point except that `(*fp)` would replace `f`. For example, defining a function `f`

```
double f(double x){return x}
```

generates code for the function (in read-only storage); whereas, defining a function pointer variable `g`,

```
double (* g)(double x);
```

allocates storage (on the stack) for a pointer.

Function pointers can be useful outside of parameter list. For example, one might have several functions `f1`, `f2`, and `f3`, all having the same type (*signature*), and one might want to select one of these at execution time. This can be done by defining a function `fp` of the same type, making assignment, e.g.,

```
if (<condition1>) f = f1; else if ...
```

and then calling `f`.

(Unlike most other pointers, dereferencing a function pointer has no effect.)

4.3.3 Scope

Larger C programs are normally written as several `.c` source code files. This facilitates the reuse of code for different `main` functions.

For a variable or function name that has been declared, there are three levels of visibility possible:

1. *local/block scope.* The names defined in the parameter list of a function and in its body are visible only inside the function. (Functions may not be nested.) Similarly for the inside of a compound statement and for the union of the inside of a `for` header and its body.

2. *global/program scope.* Names of functions and of variables defined outside functions are visible across all files that constitute the program. To declare a variable that is defined in some other file, put the keyword `extern` at the beginning of the declaration.

3. *file scope.* In the case of a helper function or variable, it is desirable that it not be visible outside of the file in which it is defined. This can be achieved by inserting the key word `static` in front of its definition.

As stated earlier, if some function `void setToZero(double *x)` is defined in file `two.c`

```
void setToZero(double *x) {
    ...
}
```

and you wish to call it from a function in file `one.c`, you must declare it in file `one.c` before use by means of a prototype. Typically, this is done by creating a header file `two.h`

```
void setToZero(double *y);
```

with declarations of all external variables that are defined in `two.c` and inserting a preprocessor directive near the beginning of `one.c`:

```
#include "two.h"
int main(int argc, char **argv) {
    ...
}
```

Review questions

1. How does the syntax of a function definition differ from that of a function declaration? Contrast the number and placement of declarations and definitions of a function.
2. If some function `void setToZero(double *y)` is defined in file `two.c` and you wish to call it from a function in file `one.c`, what must you do to make this possible, and how is this typically done?
3. What key word must be included in the definition of a function to prevent from being visible outside of the file in which it is being defined?

4.4 Strings and Input/Output

4.4.1 Strings

A *string* in C is *a pointer to* the first character of a sequence of `char`s terminated with a null character. For example, the string `"ace"` is equivalent to `{'a', 'c', 'e', '\0'}`. Fortunately, string literals may be used anywhere in a C program where a value of type `*char` is accepted. Functions to convert from a string to another type are in the `stdlib` library, e.g., `atoi` converts a string to an `int`.

Note that the string `"abc"` is a pointer to an object that you should not change, e.g.,

```
char *s = "abc";
s[0] = 'e';
```

causes a bus error. On the other the initialization `char t[4] = "abc";` places the characters `'a'`, `'b'`, `'c'`, `'\0'` into variables `t[0]`, `t[1]`, `t[2]`, `t[3]` (whose values you are free to change).

4.4.2 I/O functions

To read a line from `stdin` and extract values according to a format specifications use `scanf`, e.g.,

```
scanf("%d", &i)
```

Note that the address of `int` variable `i` is passed. To extract values in this way from a string, use `sscanf`. For reading into doubles, use format codes `le`, `lf`, or `lg`.

For file I/O you need to use functions `fopen`, `fscanf`, `fprintf`, and `fclose`.

To do binary I/O in C, use `fread` and `fwrite`.

Supplementary note

Use the `stdio.h` function `getchar` to read from `stdio.h` one `char` at a time, Use `EOF` to test for end of file. Its value is defined by a preprocessor directive in `stdio.h` to be some non-printable `char`, typically -1.

Review questions

1. What is the difference between `'x'` and `"x"`?
2. Complete the following C implementation of a `strlen` function similar to that from the `string.c` file of the C standard library:

```
int strlen(char s[]) {
   ...
}
```

4.5 Structures

New data types can be defined in C using `struct` to define their format. For example, the following defines a `struct Vector` type and defines 3 variables to be of this type:

```
struct Vector {
    double x; double y; double z;
};
struct Vector r1, r2;
```

The three *members* of a variable such as `r1` are referenced using the *member operator*, e.g.,

```
r1.x = 2.;
r1.y = r1.x + 3.;
```

The type designation "`struct Vector`" is somewhat clumsy. This is typically avoided by using `typedef` to equate "`struct Vector`" with, say, "`Vector`". (One can use the same identifier for both the structure "tag" and the new type.) The type definition can be combined with the structure definition as follows:

```
typedef struct Vector {
    double x; double y; double z;
} Vector;
```

The `typedef` construct has other uses as well. For example, conversion between single and double precision is facilitated by defining floating-point variables to be, say, `Real`, and `typedef`ining `Real` to be either `float` or `double`.

Another example of the use of a defined type is the "`time` module" of the C standard library, whose use is illustrated below:

```
#include <time.h>
clock_t start = clock();
...
clock_t end = clock();
double time = (double)(end - start)/CLOCKS_PER_SEC;
```

4.5.1 Abstract data types

An abstract data type (ADT) is one whose internal structure is unspecified; access is by means of a specified set of operations. A *dictionary* is an excellent example of an ADT that is built into Python. The `set` type would be another example. Yet other examples are *stack*, *queue*, and *priority queue*. The functionality of the stack and queue is available in the Python `list`. The `heapq` module provides efficient priority queue operations for a `list`.

Operations on a Python `list` undoubted incur significant CPU time. If the limited functionality of a stack suffices, implementation of a stack in C might be worthwhile. Following are prototypes for a function `Stack_new` that constructs an empty stack of integers and a function `Stack_delete` that `free`s memory allocated for a stack. (For some applications, there is a need for more than one stack.)

```
typedef struct Stack Stack;
Stack *Stack_new(void);
void Stack_delete(Stack *stack);
```

The definition of `struct Stack` without its members is called an *incomplete type*. This is permitted as long as you define variables only to be *pointers* to an incomplete type and not to be of an incomplete type themselves. There are only two basic operations to consider, with the following prototypes:

```
void Stack_push(Stack *stack, int a);
int Stack_pop(Stack *stack);
```

The `push` operation pushes the value of `a` onto the top of the stack. The `pop` operation removes the value at the top of the stack and returns it. These prototypes would be declared in a header file `stack.h`.

Note. A pointer to an incomplete type is an opaque pointer. An opaque pointer is an example of a *handle*, which acts as a reference to an object and whose specific value is meaningful only to the application that created it.

As an application of a stack, we consider the evaluation of arithmetic expressions involving only integers and the subtraction operator. It is standard to represent expressions involving binary operators using *infix* notation. To avoid ambiguity, parentheses must often be used, e.g., to distinguish $(4-2)-1$ from $4-(2-1)$. The *Polish* logician Łukasiewicz observed that the use of *prefix* notation, as used for functions, obviates the need for parentheses. The same is true for *postfix* notation,

$$(4-2)-1 \;\rightarrow\; (4\ 2\ -)\ 1\ - \;\rightarrow\; 4\ 2\ -\ 1\ -,$$

$$4-(2-1) \;\rightarrow\; 4\ (2\ 1\ -)\ - \;\rightarrow\; 4\ 2\ 1\ -\ -,$$

which happens to be particularly convenient for computation. Converting infix to postfix is the first of two steps in computer evaluation or compilation of an expression. The algorithm for doing so employs a stack. Here we consider the second step: evaluating a postfix expression. We illustrate by the following step-by-step example in which the stack is on the left from bottom to top and the

unprocessed part of the postfix expression is on the right:

[	9 8 7 − 6 − −
[9	8 7 − 6 − −
[9 8	7 − 6 − −
[9 8 7	− 6 − −
[9 1	6 − −
[9 1 6	−−
[9 –5	−
[14	

This is implemented by C code in `calc.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "stack.h"
int main(int argc, char *argv[argc+1]) {
   Stack *stack = Stack_new();
   for (int i = 1; i < argc; i++){
      char *s = argv[i];
      if (isdigit(s[0]) || isdigit(s[1]))
          Stack_push(stack, atoi(s));
      else if (s[0] == '-') {
         int b = Stack_pop(stack),
             a = Stack_pop(stack);
         Stack_push(stack, a - b);}}
   printf("result is %d\n", Stack_pop(stack));
   Stack_delete(stack);}
```

The value of `EOF` is defined by a preprocessor directive in `stdio.h` to be some non-printable `char`, typically -1.

A stack can also be used to effect the reordering needed for converting an infix string to the equivalent postfix string. For this purpose it is convenient to add a stack method

```
bool Stack_isempty(Stack *stack);
```

preceded by

```
#include stdbool.h
```

as well as

```
int Stack_top(Stack *stack);
```

The function `Stack_top` returns the top of the stack without popping it. Assume the infix string consists of operands, binary operators, and parentheses. There are two stages to the algorithm. The first is to process each *token* c of the infix string in turn and perform some action:

1. If c is an operand (a single digit in our example), append it to the postfix string.
2. If c is an opening parenthesis "(", push it onto the stack.

3. If c is an operator and the top of the stack is an operator of higher precedence, pop the stack and append that operator to the postfix string. Repeat this until there is no longer an operator of higher precedence on top of the stack. Then push c onto the stack. If the two operator types are nominally of equal precedence and operations are performed from left to right, the operator on top of the stack is deemed to have higher precedence.

4. If c is an closing parenthesis ")", successively pop operators off the top of the stack until an opening parenthesis is encountered. Pop that off and discard both parentheses.

The second stage is to pop off any operands still on the stack and append them to the postfix string. Following is a step-by-step example for $7 - (4 + 2 - 3)$ in which the infix expression is on the left, the stack is in the middle, and the postfix expression is on the right:

$7 - (4 + 2 - 3)$	[	
$- (4 + 2 - 3)$	[	7
$(4 + 2 - 3)$	[−	7
$4 + 2 - 3)$	[− (	7
$+ 2 - 3)$	[− (	7 4
$2 - 3)$	[− (+	7 4
$- 3)$	[− (+	7 4 2
$3)$	[− (−	7 4 2+
$)$	[− (−	7 4 2 + 3
	[−	7 4 2 + 3−
	[	7 4 2 + 3 − −

4.5.2 Data structures

There are basically two ways to implement a stack: with a one-dimensional array and with a *linked list*.

Following is a partial implementation for file `stack.c` using an array:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct Stack {
    int value[80];
    int n;
} Stack;
Stack *Stack_new(void) {
    Stack *stack = (Stack *)malloc(sizeof(Stack));
    (*stack).n = 0;
    return stack;
}
```

Constructions of the form `(*stack).n` occur so frequently that C provides the abbreviated form `stack->n` This notation is used in the following:

```
void Stack_push(Stack *stack, int a) {
    if (stack->n == 80) {
        fprintf(stderr, "Stack full.\n");
        exit(1);
    }
    stack->value[stack->n] = a;
    stack->n += 1;
}
```

The array implementation is more efficient but less elegant than the use of a linked list. The problem with using an array is that its size must be specified in advance. If the stack gets too large, it will have to be copied to a larger array.

Following is an implementation using a linked-list :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct Node {
    int value;
    struct Node *next;  // an incomplete type--at this point
} Node;
typedef struct Stack {
    Node *top;
} Stack;
Stack *Stack_new(void) {
    Stack *stack = (Stack *)malloc(sizeof(Stack));
    stack->top = NULL;
    return stack;
}
void Stack_push(Stack *stack, int a) {
    Node *node = (Node *)malloc(sizeof(Node));
    if ( node == NULL ) {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
    }
    node->value = a;
    node->next = stack->top;
    stack->top = node;
}
int Stack_pop(Stack *stack) {
    Node *node = stack->top;
    if ( node == NULL ) {
        fprintf(stderr, "Cannot pop empty stack.\n");
        exit(1);
    }
    int a = node->value;
    stack->top = node->next;
    free(node);
    return a;
```

```
}
```

More generally, one can define a pointer `mystruct`,

```
struct Mystruct *mystruct;
```

without defining `struct Mystruct`. Of course, no operations (dereferencing, indexing, incrementing) can be performed on `mystruct` until `struct Mystruct` is defined.

As an illustration of the algorithm, the insertion of

`node` [·]⟶[7 | ·]

onto the stack

`stack`→`top` [·]⟶[8 | ·]⟶[9 | ·]⟶ `NULL`

yields

`stack`→`top` [·]⟶[7 | ·]⟶[8 | ·]⟶[9 | ·]⟶ `NULL`

In addition to the array and linked list, data structures include various forms of trees and the hash table, the last of which is an efficient implementation of a dictionary.

Review questions

1. The type designation "`struct Node`" is somewhat clumsy. How is this typically avoided?
2. Give a plausible definition for `clock_t` in file `time.h`.
3. Show how to use a stack to evaluate the postfix expression `9 4 3 * 6 - +` , in particular, show how the stack evolves after each item is processed.
4. Show how to use two stacks to *evaluate* the infix expression `9 - ((4 * 3) - 6)` , in particular, show how the stacks evolve after each item is processed. The algorithm for evaluation is similar to that for conversion to postfix.
5. Using Python, define a `Stack` class having a constructor
   ```
   stack = Stack()
   ```
   which constructs an empty stack as an empty list and methods
   ```
   stack.push(a)
   b = stack.pop()
   ```
   There is no need to check for an empty stack for the `pop` method.
6. What is the difficulty of implementing a stack using an array?
7. Define a C structure having two members: the first being an `int` and the second being a pointer to the type of structure being define.
8. Give an array and a linked list implementation of the stack destructor, given the definitions of their constructors and the associated structs.
9. The reference "`(*topNode).value`" is usually abbreviated as what?
10. Give a linked list implementation of a stack for functions `push` and `pop` .

4.6 Compiling and Debugging

Notable C compilers include the gnu C compiler and those from Intel and PGI.

There are several stages in producing a running program from a file of source code:

1. Each file of source code is (preprocessed and) compiled into a file of object code.
2. Files of object code and static libraries are *linked* into a file of executable, or binary, code.
3. The executable file is *loaded* into memory to create a running program, or a *process*.

The source code, object code, and executable code are all disk files. The final process is a memory image.

The preprocessing, compilation, and linking stages are included in the command

```
gcc calc.c stack.c
```

which can be abbreviated as `gcc *.c` if all `.c` files in the current directory are to be compiled. This creates an executable, `a.out`. To give it a better name, use

```
gcc -o calc *.c
```

and invoke it as `./calc` or `calc`. The first two stages of producing an executable lead to an intermediate form called object code, whose files have `.o` extensions. To stop after object code has been produced, use

```
gcc -c *.c
```

and to produce an executable from object code, use

```
gcc -o calc *.o
```

C99 features can be enabled by using the flag `-std=c99`, which can be made the default with the Unix shell command

```
alias gcc gcc -std=c99
```

For a verbose response enter

```
gcc -v *.c
```

Libraries Libraries of functions can be linked into the executable, e.g.,

```
gcc *.c -lm
```

may be needed to link in the C `math` library, if this is not done automatically. (It is important to put the `-l` flag after the list of source files.) Other free numerical libraries written in C include the GNU Scientific Library (GSL), distributed under the GNU General Public License, and CLAPACK, which is an f2c translation of LAPACK.

Assertions The standard C library provides an `assert()` macro in the header file `assert.h`. To suppress the assertions, insert `#def NDEBUG` before `#include <assert.h>` or compile with `-D NDEBUG`.

Debugging For debugging, compile using

```
gcc -ggdb3 -o calc *.c
```

and invoke the debugger with

```
gdb calc
```

Help is available by entering `help`. You can look at a listing with line numbers using `list` and "`list -`". You can set a *breakpoint* at, say line 7, with "`break 7`" and cancel this with "`disable 7`". To run the program enter `run`. If it pauses at a breakpoint, you may `print` the value of a variable. Indeed, you may print any C expression, even one with side effects, and even a function of return type `void`! You can then `step` through the program or `continue` it. You may also `print` values if the program terminates abnormally. The command `backtrace` exhibits the memory stack, showing one *frame* for each active function call. To access the values of variables local to other functions, change the frame using, say, `frame 1`. Exit the debugger using `quit`. Here is a tutorial.

4.6.1 Creating dynamically loaded libraries

Some libraries are copied and become part of the executable. Such libraries are known as *static libraries* and are stored in files with extension `.a` in the case of Unix and OS X.

With dynamically-loaded libraries the process of loading a copy of a library into memory is delayed until execution time, thereby saving disk space. More importantly, it saves having to recompile programs that use the library whenever the library is updated. The extension of a dynamically-loaded library is `.so` for Unix ("shared object"), `.dylib` for OS X, and `.dll` for Windows ("dynamical-link library").

As an example, let `my.c` contain

```
double dot(int len, double* vec1, double* vec2) {
    double dotprod = 0.;
    for (int i=0; i < len; i +=1)
        dotprod += vec1[i]*vec2[i];
    return dotprod;
}
```

A dynamically-loaded library is created as follows:

```
$ gcc -std=c11 -c my.c
$ gcc -shared my.o -o my.so
```

Some Linux systems such as Ubuntu may require an `-fPIC` flag in the compile step. For OS X, use instead

```
$ gcc -std=c11 -c my.c
$ gcc -dynamiclib my.o -o my.so
```

4.6.2 Dynamic loading

The use of dynamic loading is operating-system dependent. For Unix-like operating systems, the Posix extension of the C standard library provides "dynamic linking functions" in a `dlfcn` library. If, for example, the dynamically-loaded libary `my.so` has already been created, here is how it could be loaded and used:

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main (void) {
    void *dl = dlopen("./my.so", RTLD_LAZY);
    if (dl == NULL) {printf("%s\n", dlerror()); exit(1);}
    typedef double dot_t(int len, double *vec1, double *vec2);
    dot_t *dot = (dot_t *)dlsym(dl, "dot");
    double u[3] = {0., 1., 2.};
    double v[3] = {1., -2., 2.};
    printf("%f\n", dot(3, u, v));
    dlclose(dl);}
```

Defining `dot` in this way makes is a pointer to any function of the same type as the coded function `"dot"`

Some Linux systems may require an `-ldl` flag for the linking step. For information on how to load a dynamically-loaded library on Windows, go here.

4.6.3 Improving performance

Performance profiling Performance profiling tools enable one to instrument C code so that executing the code produces histograms of where code is spending most of its time. Counts of function calls can be determined by using `gprof`, e.g.,

```
gcc -std=c99  testmy.c my.c -o testmy -g -pg
testmy
gprof testmy > testmy.gprof
```

The output file presents its results in two parts: the "call graph" profile gives the number of calls that each function makes; the "flat" profile gives information only for the top levels. More for testing purposes is another tool `gcov`, which reveals which lines of code are actually executed and gives a count of the number of times each line is executed, e.g.,

```
gcc -std=c99 -fprofile-arcs -ftest-coverage testmy.c my.c -o testmy
my
gcov my  # processes the file my.c
```

An output file `my.c.gcov` is created that contains the line counts.

Compiler optimizing options Compilers have available several levels of optimization. For `gcc` the default optimization option `-O0` is recommended for development and debugging and the option `-O2` is recommended for deployment.

Review questions

1. What are the four stages of producing an executable from a C program?
2. If running a program in a debugger, how can you cause the program to stop each time it reaches a certain point in the program?

3. What are the two advantages of a dynamically-loaded library over a static library?
4. What type of C library is linked with the main program to become part of the executable? What type is not part of the executable?
5. What is the Unix meaning of the extension `.a`? `.c`? `.o`?, `.so`?

4.7 Calling Compiled Code from Python

There are numerous ways to do this:

C API to Python and NumPy This is a library of C functions and variables that can be used to create wrapper functions that together with the targeted C code can be compiled into fast binary Python modules. See Extending Python with C or C++ for more information.

ctypes module and attribute The `ctypes` module from the Python standard library and the `ctypes` attribute of NumPy arrays can be used to create a Python wrapper for an existing dynamically-loaded library written in C.

SWIG This automates the process of writing wrappers in C for C functions. SWIG is easy to use if the argument list is to be limited to builtin Python types but can be cumbersome if efficient conversion to NumPy arrays is desired. The difficulty is due to the need to match C array parameters to predefined patterns in the `numpy.i` file, which is a SWIG interface file for NumPy.

Cython This facilitates the writing of C extensions for Python.

weave This allows the inclusion of C code in Python programs.

f2py This is for interfacing to Fortran.

See `http://www.scipy.org/Topical_Software` for links to some of these. Presented here is the use of `ctypes`. Unlike the use of the C API or SWIG, it permits the interface to be written in Python.

4.7.1 The Python `ctypes` module

The `ctypes` module of the Python standard library provides definitions of fundamental data types that can be passed to C programs. For example, assuming we

```
import ctypes as C
```

these types would have names like `C.c_int` and `C.c_double`.

They can be used to construct objects, e.g.,

```
x = C.c_double(2.71828)
```

that can be passed (to Python interfaces to) C programs.

Fundamental types can be composed to get new types, e.g.,

```
xp = C.POINTER(C.c_double)(x)
```

For an opaque pointer `void *`, use `C.c_void_p`

Array types can be created by "multiplying" a `ctype` by a positive integer, e.g.,

```
y = (C.c_double*n)(*yseq)
```

Here `yseq` is an object that can be indexed. The asterisk is a Python operator for expanding the elements of a sequence into the arguments of a function, i.e.,

```
function(*yseq)  -> function(yseq[0], yseq[1], ..., yseq[-1])
```

The `ctypes` module has a utility subpackage to assist in locating a dynamically-loaded library, e.g.,

```
import ctypes.util  # an explicit import is necessary
C.util.find_library('m')
```

locates the C math library. For loading a library there are constructors, e.g.,

```
myDL = C.CDLL('./my.so')
```

which makes `my` a module-like object (a CDLL object to be precise). The following may work for Windows:

```
import os
myDL = C.cdll.LoadLibrary(os.getcwd() + os.sep +'my.so')
```

Similar to a Python module, `myDL` has function-like objects as attributes, and these have the same names as the C functions in the library, e.g., `myDL.dot`. Such a function-like object has itself attributes `argtypes` and `restype`. The `restype` must be assigned a `ctype` corresponding to the return type of the C function. This enables its automatic conversion to an appropriate Python object. For a C function whose return type is `void`, use `None`. The `argtypes` attribute enables the automatic conversion of the arguments, to a limited extent. Here is an example:

```
from ctypes import CDLL, c_int, c_double, POINTER
_dot = CDLL('my.so').dot
_dot.argtypes = [c_int, POINTER(c_double), POINTER(c_double)]
_dot.restype = c_double
def dot(vec1, vec2):  # vec1, vec2 are lists
    n = len(vec1)
    vec1 = (c_double*n)(*vec1)
    vec2 = (c_double*n)(*vec2)
    return _dot(n, vec1, vec2)
```

The conversion of `lists` to the correct `ctype` would not happen automatically. Also, `vec1` and `vec2` could be any type of Python sequence object containing Python `floatss`.

Warning. If you use the extension `.so` for the name of a file, do not make its stem the same as a `.py` file in the same directory, e.g., do not have both a `funcs.py` and a `funcs.so`. One convention is to use `funcs`.py and `_funcs.so`.

4.7.2 The NumPy `ctypeslib` module

Reference: http://www.scipy.org/Cookbook/Ctypes

The construction of `ctype` arrays from indexed data undermines the efficiency gained from calling a function coded in C. Fortunately, NumPy provides a couple of ways to avoid this. The simpler way is to place the `numpy.ctypeslib` object `ndpointer()` in the `argtypes` `list`.

An alternative to the `ctypes.CDLL` constructor is the NumPy function `numpy.ctypeslib.load_library`.

Following is an example of a Python wrapper for the C file `my.c` using the foregoing `numpy.ctypeslib` methods:

```
from ctypes import c_int, c_double
from numpy.ctypeslib import load_library, ndpointer
_dot = load_library('my.so', '.').dot
_dot.argtypes = [c_int, ndpointer(), ndpointer()]
_dot.restype = c_double
def dot(vec1, vec2):  # vec1, vec2 are arrays
    return _dot(len(vec1), vec1, vec2)
```

A weakness of the example above is that it assumes that the NumPy array elements are stored in contiguous memory locations. To take care of the possibility that this is not the case, apply the function `numpy.ascontiguousarray` to the arrays `vec1` and `vec2` before passing them as arguments. (The `strides` attribute of an `ndarray` provides information about the distance between locations of successive elements of an array.)

Supplementary notes

For a `ctypes` object such as `x = C.c_double(2.71828)`, the attribute `x.value` returns the Python object. An alternative to `xp = C.POINTER(C.c_double)(x)` is

```
xp = C.POINTER(C.c_double)(); xp.contents = x
```

You can change the value of `x` using

```
xp[0] = 3.14159
```

Analogously, a C array can be converted array back to a Python value or `list` by indexing it with an `int` or a `slice`.

NumPy arrays (type `ndarrays`) have a method `ctypes.data_as` for conversion to a Ctype, e.g.,

```
px = x.ctypes.data_as(ptrd)
```

constructs a pointer to the data in `x`.

Review questions

1. How does one convert 3.141592 to an object of type `ctypes.c_double`?
2. ∗ To which C operator does the `ctypes` attribute `contents` correspond?

3. Using the `c_double` constructor and the `POINTER` function from the Python `ctypes` module, show how to construct each of the following C types:
   pointer to an array of 3 doubles,
   array of 3 pointers to doubles.

4. In the Python expression
   ```
   (ctypes.c_double*n)(*x)
   ```
   what is the meaning of the first asterisk?
   the second asterisk?

5. $*$ In the Python assignment statement
   ```
   xx = x.ctypes.data_as(ctypes.POINTER(ctypes.c_double))
   ```
   what kind of object is `x`?
   `xx`?

Chapter 5

Parallel Computing

5.1 Computer System Basics

5.1.1 Computer organization

References: Grama,Gupta,Karypis&Kumar(2003), Chapter 2; http://en.wikipedia.org/wiki/Central_processing_unit.

At the most basic level, a computer consists of a CPU, memory (RAM, usually DRAM), a hard disk, and numerous ports to external devices. Memory is divided into *bytes* addressed by consecutive integers starting with zero. On a 32-bit computer, addresses are representable with 32 bits, which limits memory to 4 Gbytes. Both data and computer instructions are stored in memory. On modern computers one byte is 8 bits. Typical data types consist of 1, 2, 4, or 8 bytes.

Memory hierarchy The execution of instructions requires that instructions and data be fetched by the CPU from memory. The CPU stores data in a small number of registers each consisting a few bytes, e.g., 4 bytes. Instructions are executed very quickly, and it is the *communication* of data that is the limiting factor in performance. There are two parameters that measure communication performance: *latency* and *bandwidth*. Latency is the delay between the request for data and its arrival; bandwidth is the rate at which data arrives if there is a request or requests for a large amount of data. Memory latency is equal to many clock cycles. For this reason modern processors also have much smaller but much faster memory called *cache*, which stores a duplicate of those blocks of memory that are being accessed. Those parts of memory that are in the cache change automatically in response to memory accesses by the CPU. This greatly improves performance (i) if the same memory location is accessed numerous times in a short interval—*temporal locality* or (ii) if a set of contiguous memory locations is accessed in a short interval—*spatial locality*. Accessing a memory location not in cache is called a *cache miss*. Typically there are three levels of caches: from a very small and very fast L1 cache to a much larger and slower L3 cache. When memory is not large enough, the hard disk can be used as an extension of memory, but this can greatly degrade performance.

Parallel computers A shared memory computer has a single addressable memory space and multiple processors. The term *multicore* is used when the processors are on the same chip. It is typical for these cores to share an L3 cache. It is common to have several such chips share memory. For greater parallelism a number of multiprocessors are connected together to form a cluster, each multiprocessor being a *node* of the cluster. In this case we have a *distributed memory* and it is necessary to do *message passing* to share information. Reference: http://en.wikipedia.org/wiki/Parallel_computing.

To learn hardware characteristics, enter

```
cat /proc/cpuinfo
```

if running Linux and

```
system_profiler SPHardwareDataType
```

if running Mac OS X.

5.1.2 Processes

Numerous *processes* run concurrently on a computer, only some of them belonging to a user, each process having an independent allocation of memory. *Concurrent* means that the order of execution among processes is immaterial, so if multiple processors are available, different processes can be executed on different processors. A process consists of a multiplicity of *threads*, which all share the same memory. The process begins as a single thread, which can then spawn additional threads. In this way a process can execute in parallel on multiple cores. To monitor the activity of multiple cores on a Unix machine, enter the `top` command and then enter 1. For MacOS X, use instead the Activity Monitor utility.

Processes can also spawn child processes, and communication can be effected through files. The parent and child processes are executed concurrently, meaning that the each process gets slices of CPU time. Using disk files to communicate between separate processes is inefficient. More direct communication can be accomplished using pipes or MPI (often implemented on top of *sockets*) or yet other mechanisms. For efficient communication between processes, If the child program is a filter, the parent program can communicate through an *unnamed pipe* using `stdin` and `stdout`. Be aware that reading from a pipe cannot begin until the writer closes the pipe. Unix also provides *named pipe*s, but their capacity is quite limited.

The assignment of threads and processes to cores can be managed by the user by calling system-dependent C functions. This topic is called thread or process *affinity*.

5.1.3 Python as glue

Python can incorporate standalone programs by spawning child processes and communicating through files. The Python function `subprocess.Popen` spawns a child process and returns immediately with a handle to the process (a `Popen` object). This handle has a `wait` method that can be used to wait for the child process to terminate. Use `wait()` judiciously to avoid unnecessary idle time. The `Popen` object also has a `communicate` method for creating a pipe to and/or from the child process, in which case the parent process waits until the child process terminates. If communicating with `stdin` and/or `stdout` on Windows, use `Popen` with the keyword parameter

`Universal_newlines=True`. Be aware that each invocation of `subprocess.Popen` (with keyword argument `shell=True`) opens a new shell, which means that `export`ing an environment variable in this shell has no effect on other shells that are opened this way. For further information see http://docs.python.org/library/subprocess.html.

Following is a example of two processes communicating using `stdin` and `stdout`. Here is the child program:

```
#!/usr/bin/env python
import sys
from math import *
for line in sys.stdin:
    print(eval(line[:-1]))
```

and here is the parent program:

```
#!/usr/bin/env python
from subprocess import Popen, PIPE
datalist = \
    ["sqrt(2.)\n sqrt(3.)\n sqrt(5.)\n sqrt(6.)\n", \
     "log(2.)\n log(3.)\n log(6.)\n"]
for idata in datalist:
    proc = Popen("./child.py", \
                 stdin=PIPE, stdout=PIPE, stderr=PIPE)
    idata = idata.encode()
    odata, edata = proc.communicate(input=idata)  # waits
    odata = odata.decode()
    edata = edata.decode()
    print("output is\n", odata, "errors are\n", edata)
```

Review questions

1. What term do we use for the smallest addressable unit of memory?
2. A multicore processor is a special kind of multiprocessor. What is special about it?
3. A multicore computer has several processors on the same chip. Typically, what additional hardware difference distinguishes it from a regular shared-memory multiprocessor?
4. What is the principal difference between doing a computation using multiple processes and using multiple threads?
5. By what device, other than a regular disk file, can two processes on the same processor communicate significant data?

5.2 Distributed Memory Programming

References: Grama,Gupta,Karypis&Kumar(2003), Sections 6.3.0–6.3.3, 6.5, and 6.6.3; http://en.wikipedia.org/wiki/Message_Passing_Interface; a detailed tutorial.

5.2.1 MPI

MPI (Message Passing Interface) is a library of functions. It is implemented for C90, C++, Fortran 90, and Fortran 2008. Recall from Section 1.3.1, the following example of compiling and executing a C program that uses MPI,

```
mpicc hello.c
mpiexec -n 2 ./a.out
```

where the `n` flag specifies the number of processes. The current version of MPI is MPI-3.1. The principal implementations are MPICH and Open MPI (used by Mac OS X).

The package `MPI for Python` (`mpi4py`) supports the use of MPI-3 with Python. This package is used by iPython for some of its parallel computing functionality.

Each MPI process executes the same program but with different data. There is no synchronization unless explicity requested. Needless synchronization creates idleness Insufficient synchronization creates a *race condition* and leads to indeterminate (and hence possibly incorrect) answers.

MPI employs the notion of a *communicator*, which is a set of communicating processes. The builtin `mpi4py` communicator `MPI.COMM_WORLD` comprises all processors.

Because each instance of the program is run as a separate process, one needs special MPI functions if more than one process is to do I/O to the same file.

5.2.2 Point-to-point operations

Included here are the send and receive operations.

Following is a toy example (but having the character of a real problem), that uses nonblocking sends and receives to perform an iteration

$$x_i^{\text{new}} = f_i + \frac{1}{2}(x_{i-1} + x_{i+1}), \quad i = 1, 2 \ldots, 80,$$

where x_0 and x_{81} are given. The vectors of length 80 are divided equally among 8 processors. The program uses communication methods `Isend` and `Irecv`, which denote "initiate send" and "initiate receive," respectively.

```
from mpi4py import MPI
import numpy as np
cw = MPI.COMM_WORLD
r = cw.Get_rank()   # r is one of 0, 1, ..., 7
x = np.empty(12); f = np.empty(12)
... initialize x and f to be elements 10r, 10r+1, ..., 10r+11 of global x, f
x0, x1, x10, x11 = np.empty((4,1))  # message containers
while not converged:
    # assert(all values are the result of iteration k - 1)
    x1[:] = f[1] + 0.5*(x[0] + x[2])
    if r > 0: sendL = cw.Isend(x1, dest=r-1)
    if r < 7: recvR = cw.Irecv(x11, source=r+1)
    x10[:] = f[10] + 0.5*(x[9] + x[11])
    if r < 7: sendR = cw.Isend(x10, dest=r+1)
```

```
    if r > 0: recvL = cw.Irecv(x0, source=r-1)
    x[2:10] = f[2:10] + 0.5*(x[1:9] + x[3:11])
    x[1] = x1
    x[10] = x10
    if r > 0: recvL.Wait(); x[0] = x0
    if r < 7: recvR.Wait(); x[11] = x11
    if r > 0: sendL.Wait()
    if r < 7: sendR.Wait()
    converged = ...
```

There is *no waiting* after an `Isend` or `Irecv`. Moreover, the `Wait` for a send request lasts only until the data is sent; it does not extend until the data is received. The `Wait` for `sendL` and `sendR` is needed to ensure that `x1` and `x10` are not assigned new values until their current values have actually been sent.

MPI also has functions for blocked sends and receives. For example, `cw.Irecv(...).Wait()` could be replaced by `cw.Recv(...)` and `cw.Isend(...).Wait()` by `cw.Send(...)`. The careless use of blocked sends and receives can lead to *deadlocks*. Also, their use does not allow computation to be performed while waiting for communication to start, thus leading to increased *idle time*.

An alternative to `MPI.Request.Wait` is `MPI.Request.Test`, which is useful for polling the state of several requests until one of them tests `True`.

Most `mpi4py.MPI` functions do not require elements of an array to be stored in contiguous memory locations.

To ensure the proper sequencing of I/O, it is best either to do all I/O from a single MPI process or to use `MPI` file I/O functions.

Using C Given here is the C equivalent for selected parts of the Python code:

```
#include <mpi.h>
...
   MPI_Init(&argc, &argv);
   ...
   int r; MPI_Comm_rank(MPI_COMM_WORLD, &r);
   ...
   MPI_Request sendL, recvR, ...;
   if (r>0)
      MPI_Isend(&x[1], 1, MPI_DOUBLE, r-1, 0, MPI_COMM_WORLD, &sendL);
   ...
   MPI_Status status;
   if (r<7) MPI_Wait(&recvR, &status);
   ...
   MPI_Finalize();
...
```

The call to `MPI_Init` must occur before the variables `argc` and `argv` are used, because their values are held by the `mpiexec` command. The call to `MPI_Init` initializes `argc` and `argv`. After a return from a call to `MPI_Wait`, a request object variable can be reused. MPI functions in C and Fortran require elements of an array to be stored in contiguous memory locations.

5.2.3 Collective operations

MPI also has collective operations involving all processors (within a communicator)—they are all blocking.

The broadcast function `MPI.Comm.Bcast(cw, x, root=0)` sends data from one processor to the others. The setting `root=0` indicates the default value.

To perform a summation, or similar operation, with each processor contributing a term, use

```
MPI.Comm.Reduce(cw, x, y, op=MPI.SUM, root=0)
```

The first array is for the term to be sent and the second for the result to be received. There are three other predefined reduction operations (`PROD`, `MAX`, `MIN`), which can be passed as arguments to `MPI.Comm.Reduce`. However, `MPI.MAXLOC` is not implemented for `Reduce`. To have the result returned to each processor, use `MPI.Comm.Allreduce`.

To gather values from all processors to a designated processor, use `MPI.Comm.Gather`:

```
MPI.Comm.Gather(cw, a, ag, root=0)
```

where each processor sends an equal-sized message `a` to be placed into the array `ag` of the receiver in an order based on the rank of the sender. To have these values sent to all processors, use `MPI.Comm.Allgather`.

Other useful functions are `Barrier`, `Scatter`, and `Alltoall`.

Using C The following example for `Allreduce`,

```
MPI_Allreduce(MPI_IN_PLACE, &sum, 1, MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);
```

sums up the values of each processor's `sum` and overwrites each such value with the total sum. In the following example for `Allgather`,

```
MPI_Allgather(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
              x, n, MPI_DOUBLE, MPI_COMM_WORLD);
```

`x` is a double precision array with `n` columns and as many rows as there are processors. The array values `x[r][0]` through `x[r][n-1]` are the contributions of processor `r`.

5.2.4 Grid computing and Condor

Grid computing is like computing on a cluster with loosely coupled single-processor nodes of varying performance; it is suitable for *embarassingly parallel* jobs where there is very little communication between tasks. Globus is the standard way of submitting jobs on grids. An alternative is Condor, which was developed to exploit otherwise unused cycles on computers with diverse ownership.

Review questions

1. MPI is not a language extension nor is it compiler directives? What is it?
2. ∗ What command is used to compile an MPI job in C?
3. What command is used to execute an MPI job?
4. What does the following code do?

```
r = MPI.COMM_WORLD.Get_rank();
```

5. After the call
   `recvR = MPI.COMM_WORLD.Irecv(x, source=r+1)`
   what must we do before using the value of `x`?

6. After the call
   `sendR = MPI.COMM_WORLD.Isend(x, dest=r-1)`
   what must we do, if anything, before changing the value of `x`?

7. Under what circumstances can we reuse request objects?

8. Why is unblocked communication potentially more efficient than blocked communication?

9. What is the alternative to `MPI.Request.Wait`?

10. What is the equivalent of

    ```
    MPI.Comm.Send(comm, x, dest=s)
    ```

    using unblocked communication?

11. Give an example of a predefined MPI reduction operation.

12. Consider `p` processors with the `i`th processor computing the `i`th row of an array of dimension `p` by `n`. Assume that each array stores its row in an array `u` of dimension `n`. Complete the following `mpi4py` operation so that its execution on each processor sends the contents of each `u` array to the appropriate row of an array `ug` of dimension `p` by `n` residing on processor 0. Assume all arrays are `C_CONTIGUOUS`.

    ```
    MPI.COMM_WORLD.____________(______, ______)
    ```

5.3 Computing on Scholar and PBS Job Submission

Here is information on how to compile and run MPI programs.

Log on to Scholar with your Purdue Career Account login name and password using an SSH2 client:

```
ssh myusername@scholar.rcac.purdue.edu
```

To compile a C program that uses MPI, enter

```
module load devel
mpicc hello.c
```

You can use `module avail` to see what other software is available and `module list` to see what has been loaded.

Jobs may be submitted by preparing a PBS (Portable Batch System) job script, and submitting that script to the PBS job management system via the `qsub` command:

```
qsub -q scholar -W group_list=scholar-queue \
     -l nodes=NN:ppn=PP,walltime=HH:MM:SS scriptfile_name
```

where `NN` is the number of nodes you want the job to use, `PP` is the number of processors per node you want to use, and `HH:MM:SS` is the wallclock time you want the job to be allowed to run. The `scholar` PBS queue has access to 8 nodes with 2 processor chips per node and 8 cores per processor chip. The queue has a default wallclock time limit of 30 minutes and a maximum of 8 hours. Following is an example of a script file:

```
#!/bin/bash -l
module load devel
cd $PBS_O_WORKDIR
mpiexec -n 8 a.out
```

Assuming the script is named `hello.sh`, here is a possible `qsub` command:

```
qsub -q scholar -W group_list=scholar-queue \
     -l nodes=4:ppn=2,walltime=0:01:00 hello.sh
```

For email notification of job beginning and ending put `-mbe` after `qsub`. To run interactively, enter

```
qsub -q scholar -W group_list=scholar-queue -l nodes=4:ppn=2 -I
```

which opens a new shell into which you might enter

```
module load devel
mpiexec -n 8 ./a.out
exit
```

assuming `a.out` resides in your home directory. Additional useful commands:

```
qstat -u LOGINNAME
qstat -a scholar
qdel JOBNUMBER
qstat -Q
```

Here is an example of mpi4py run on the front end.

```
#!/bin/bash -l
# hello.sh
module load devel
module load python/3.4.1
mpiexec -n 4 python3 hello.py
```

∗ **Review questions**

1. What is the Unix command used to submit a job under PBS?
2. What is the argument (not the flags/options) in the `qsub` command?
3. What is the `q` flag for in the `qsub` command?
4. What is the `l` flag for in the `qsub` command?
5. What is the Unix command used to check status of jobs under PBS?
6. What is the `u` flag for in the `qstat` command?
7. What is the Unix command used to abort a job under PBS?

5.4 Shared Memory Programming

5.4.1 POSIX Threads

POSIX threads, also referred to as Pthreads, is implemented as a C library on virtually all platforms. It permits the concurrent executions of multiple lightweight processes known as *threads*. The original purpose of threads was to facilitate the scheduling of other computation, while waiting for input from disk or some other device. The most significant difference between a process and a thread is that threads of the same process share memory whereas different processes do not. Python has a `thread` module, but these are not capable of parallel execution (unlike Java threads). The assignment of Pthreads to processors is done by the operating system, and several threads may be assigned to the same processor or core. For Linux, Windows, and OS X, there are functions that can be used to control or influence assignment of threads to cores, e.g., one may wish a set of threads to share the same L3 cache.

The program begins execution as a single thread. By performing a `fork` operation, it can spawn, say, 8 other threads and assign tasks to them. It will at some point wait for the other threads to complete their tasks by means of a `join` operation.

If different threads want to write into the same variable, there can be a problem, for example, consider the following situation:

```
void *add_term(void *data) {  // required prototype for a pthread
   // data is assumed to point to a struct data
   double *sum_p = &((struct data *)data)->sum
   double term = <some lengthy calculation>;
   *sum_p += term;
   // load into register the 8-byte value in  *sum_p
   // add to register the 8-byte value in  term
   // store register value back into  *sum_p
   return NULL;
}
```

The problem is that with two or more threads performing the update `*sum_p += term`, a second update may be started before the first is complete. To limit the execution of a *critical section* of code to a single thread, Pthreads provides mutual exclusion objects of type `pthread_mutex_t` called *mutex-locks*. Also provided are functions `pthread_mutex_init`, `pthread_mutex_lock`, and `pthread_mutex_unlock`. The mutex-lock is initialized once before it is used. If a thread calls `pthreads_mutex_lock` for a mutex-lock which is already locked, it will wait until that mutex-lock is unlocked.

For another example, consider the accumulation of a histogram

```
...
count[k] += 1;
...
```

Here, one can use a different mutex lock for each value of `k`. Actually, *it is shared variables* not sections of code *that need the protection.* However, a mutex lock is the mechanism that is provided, and the programmer must ensure that it is applied everywhere in the code where any element of `count` is being updated.

Mutexes can also be used to implement a barrier for synchronizing threads.

∗ **A simple example** This is similar to the example given for MPI:

```
#include <pthread.h>
void *set_y_ax(void *kptr);
void *set_x_y(void *kptr);
double x[802], y[802];
int main(int argc, char *argv[]) {
   x[801] = x[0] = 0.;
   int start[8];
   pthread_t thread[8];
   for (int i=0; i<8; i+=1) {
      start[i] = 1 + 100*i;
      pthread_create(&thread[i], NULL, set_y_ax, &start[i]);
   }
   for (int i=0; i<8; i+=1) {
      pthread_join(&thread[i], NULL);
   ...do the same for set_x_y...
}
void *set_y_ax(void*kptr) {
   int k = *(int *)kptr;
   ...initialize x to be elements 100k+1, 100p+2, ..., 100p+100 of vector
   for (int i=k; i<k+100; i+=1)
      y[i] = - x[i-1] + 2.*x[i] - x[i+1];
   return NULL;
}
void *set_x_y(void *kptr) { ...}
```

5.4.2 Cilk Plus

Cilk Plus, which is trademarked by Intel, is an extension of C and C++. It can spawn threads to exploit multicore parallelism. It is implemented in GCC 5.

5.4.3 OpenMP

OpenMP (Open Multi-Processing) is a set of compiler directives, library routines, and environment variables for C/C++ and Fortran that supports shared memory parallel programming. The compiler directives insert calls to Pthreads functions. The current version is 4.0. Version 3.1 is implemented in GCC 4.7. Here is a simple tutorial and here is a very detailed tutorial. Here is the official website and here is the wikipedia entry.

To compile and execute a C program that uses OpenMP, e.g., `ohello.c` from Section 1.3.1, enter

```
export OMP_NUM_THREADS=4
gcc -std=c99 -fopenmp ohello.c -o ohello
./ohello
```

(Mac OS X requires also the compiler flag `-lgomp`.) To compile a DLL, use

```
gcc -std=c99 -fopenmp ohello.c -c
gcc -shared -lgomp ohello.o -o ohello.so
```

(Recall that Linux may require `-fPIC` flag in the compile step. Mac OS X requires the use of `ld` and its associated flags instead of `gcc -shared`.) There is no need for PBS unless you want to run on more than one node. Here is more information on how to compile and run OpenMP programs specific to the Rosen Center.

The `parallel` directive applies to the statement that follows. Frequently that statement is a compound statement (a sequence of statements enclosed in braces), also known as a *block*. Variables defined prior to the scope of the directive are shared variables; those defined within that scope are private variables. Upon entering the scope of the parallel directive, the master thread forks other threads. Upon leaving that scope, the other threads exit and the master thread does a join for each of them. To synchronize the threads, use the `barrier` directive.

To assign tasks to specific threads, use functions `omp_get_thread_num` and `omp_get_num_threads`, illustrated by `ohello.c` from Section 1.3.1. Be aware that the number of threads equals 1 in the serial part of the code.

The `critical` directive specifies a block of code that is to be executed by only one thread at a time. In very special cases, e.g.,

```
pragma omp atomic
x += ⟨expression not containing x⟩
```

the `atomic` directive can be used instead to alert the compiler of the possibility of using special hardware instructions for better performance.

Within the scope of a `parallel` directive, the `for` directive can be used to partition the execution of a `for` loop among the threads. A simple example is

```
#pragma omp parallel
#pragma omp for
for (int i=1; i<799; i+=1) y[i] = - x[i-1] + 2.*x[i] - x[i+1];
```

which can be compressed as

```
#pragma omp parallel for
for (int i=1; i<799; i+=1) y[i] = - x[i-1] + 2.*x[i] - x[i+1];
```

If the loop is a reduction loop, use something like

```
#pragma omp for reduction (+:sum)
```

Other operators supported include `*`, `min`, and `max`.

Note. The C library function `clock` included with `time.h` measures the sum of times on all threads. Use instead the the OpenMP library function

```
double omp_get_wtime( )
```

to get the wall clock time in seconds. The precision of the clock is available from `omp_get_wtick`.

Function calls The scope of an OpenMP compiler directive applies only to code that physically follows the directive. In particular, it does not extend to functions that are called from that code. However, every invocation of a function creates a fresh set of local variables (in the stack part of memory) except for local variables that have been declared to be `static`. Such variables reside in the static part of memory and they retain their values between function calls. All invocations of a function *share* such variables. If any of them are read/write, the function is not *thread-safe*. (Nor would it be safe to make a function recursive if it employs read-write static variables.) The function `rand` in the standard C library is not thread-safe, because it maintains state using a variable in static storage. Using OpenMP, a call to `rand` should be in a critical section. This is not an issue with MPI, because every process has its own memory, and hence its own copy of `rand`.

5.4.4 Coprocessors, including graphics processing units

The graphics processing units in computers employ large-scale data parallelism making them much faster than CPUs for certain operations. GPU chips trade cache memory for more processing units and use hardware multithreading to hide memory access latency. Further economies are obtained by grouping the processors into units that share a single stream of instructions—called single-instruction multiple-data (SIMD) processors. There is currently interest in doing general-purpose computing on graphics processing units. Algorithms must contain sufficient data parallelism to permit acceleration by GPUs, say, 10 000 threads of computationally intensive work. Those algorithms that do so can exhibit performance of one to two orders of magnitude faster than that available to current CPUs.

As an example, the Nvidia GTX 470 (Fermi series) GPU has 448 processors. These are grouped into a set of 14 *streaming multiprocessors* (SM), all having access to a single *global memory*. Global memory access is high bandwidth but high latency. Each *streaming multiprocessor* is an 32-processor SIMD unit with its own *shared memory* with access speed comparable to L1 cache. Each SM executes its instructions independently from the other SMs. The host CPU reads and write to the GPU *global memory* and oversees execution of *kernels* by the GPU.

Another example is AMD's ATI Radeon HD 5770 (Evergreen series). It has 10 SIMD units, each with 80 processors for a total of 800.

For programming GPUs, there is a standard, OpenCL (Open Computing Language), which is a subset of C99 with extensions and a library of functions for heterogeneous parallel computing across GPUs and CPUs. Existing implementations include Nvidia GPUs, AMD GPUs, Apple computers, and IBM POWER6 and Cell/B.E. Linux systems. Additionally, Nvidia has a proprietary product called CUDA, and Microsoft has one called DirectCompute.

As an example, consider CUDA. The CPU uses the GPU as a coprocessor to execute code by means of a special kind of function call. While the GPU is executing, the CPU can do computation and I/O. A subprogram for the GPU is called a *kernel* and is written in an extension of a subset of C called *C for CUDA*. (There is also a Fortran CUDA compiler from PGI.) A *kernel* is executed by a *grid* of *thread blocks*, so-called because these *thread blocks* are indexed as elements of a one- or two-dimensional array. A *thread block* consists of 2–16 warps of threads to execute, each warp having 32 threads. Each thread block has the same number of threads, each executing the same code. (By means of conditionals, different thread blocks can execute different sections of the code.) A thread block is assigned by the system to to a streaming multiprocessor for execution. Several may be assigned to the same streaming multiprocessor. The scheduling of thread blocks and their assignment to SMs is beyond programmer control, so they should be designed to run independently of each other. All thread blocks have access to *global memory*. All the threads within a block have access to that block's partition of shared memory. For good performance, the code should be written so that all threads in the same block take the same branch in the case of a conditional. Additionally, performance is sensitive to the manner in which memory is accessed. There is an extensive runtime library that includes, e.g., an FFT. Section 3.2 of the CUDA Programming Guide Version 3.0 has an example of matrix multiplication on the GPU.

"NVIDIA and Udacity have teamed up to offer Intro to Parallel Programming course, a free online course. You can complete the course at your own pace, using your own computer or Udacity's powerful GPU-accelerated systems in the cloud."

5.4.5 OpenACC

OpenACC is an alternative to OpenMP designed for accelerator devices such as GPUs (graphics processing units). A preliminary implementation is available in GCC 5.

5.4.6 Python multiprocessing module

The multiprocessing module enables the use of multiple processors by using (sub)processes instead of threads.

Review questions

1. What does it mean for 2 threads to execute concurrently?
2. Why might it be advantageous to have many more threads than processors?
3. What mechanism does Pthreads provide for giving one thread exclusive right to execute a critical section of code?

4. What happens if a thread calls `pthreads_mutex_lock` for a mutex-lock which is already locked?

5. $*$ How do we specify the task to be done using `pthread_create`? In particular, what do we provide as arguments that specify the task to be done?

6. How are OpenMP commands put into a C program?

Chapter 6

Survey of Other Topics

6.1 Software Development

6.1.1 Version control

A version control system is good for (i) doing collaborative work and (ii) undoing changes to a set of files. There are two types: client-server and distributed. A client-server version control system keeps a master copy of the files together with a record of revisions in a central *repository*, with each person maintaining their own working copy. Periodically, the working copy is updated to get changes from the repository. Modifications to the working copy are shared by *committing* them to the repository. A popular client-server system is Subversion (`svn`). In a distributed version control system, each participant not only has a working copy but has also a local repository that keeps track of changes. In practice, one of these local repositories might be treated as a central repository Notable examples of distributed systems are Mercurial and Git.

To do version control over the Internet, it is common to employ a hosting site such as `bitbucket.org` (Mercurial and Git), `github.com` (Git), `code.google.com` (Mercurial, Git, and Subversion), or `sourceforge.net` (Mercurial, Git, and Subversion).

Consider a *centralized workflow* scenario, where several collaborators work from a remote repository that has been desigated as a central repository. Following are steps that might be take for simple use of Git: Assuming the remote central repository has already been created, do the following to create a local repository:

```
$ git clone <URL>
```

This makes a local project directory and creates a local repository in hidden subdirectory `.git`. To make modifications to the set of files, you might do the following:

1. `$ cd <directory created by git clone>`
2. Modify the working copy as desired. Any new files you create are considered to be *untracked* until you `add` them.

3. `$ git commit -am "<message>"`
   Commits changes to your local repository. There are now at least two different *branches* in the local repository: `origin/master` and `master`, the latter reflecting the changes just made. The name `origin` is the default alias for the your most recent copy of the remote repository.

4. `$ git push origin master`
   Pushes your changes to the remote repository. If that repository has changed since your `clone` or most recent `fetch`, you get a message telling you to do a `git pull`, which is equivalent to `git fetch` followed by `git merge`.

5. If needed, do
   `$ git fetch origin`
   Fetches the latest version from the cloned repository, and updates the branch `origin/master` in the local repository.
   `$ git merge origin/master`
   This updates your working copy and commits the changes to your local repository. If you are warned about conflicts, you need to edit files to remove the conflicts (conflicts are delimited by `<<<<<<<` and `>>>>>>>`) and then conclude with another `commit` and `push`.

To make modifications again some time later:

1. `$ cd <directory created by git clone>`
2. `$ git fetch origin`
3. `$ git merge origin/master`
4. Modify the working copy as desired, proceeding as in steps 2–5 above.

Omitted here is the Git concept of *staging*, which has been bypassed using the `-a` option of `commit`. Here is a quick reference for Git.

6.1.2 Automated builds

Creating executable programs—and documents—is often a process with many steps. If changes are made to only a few of several source files, the updating will be faster if only the affected steps of the process are rerun.

In the example of the postfix calculator,

- the executable, call it `calc`, depends on `calc.o` and `stack.o`,
- `calc.o` depends on `calc.c` and `stack.h`, and
- `stack.o` depends on `stack.c`.

Efficient updating can be achieved by using an automated build system, which makes use of a configuration file that lists dependencies among files and actions that must be taken to regenerate

a file. Timestamps of each pair of files related by a dependency are compared to see which actions must be repeated.

By far, the most popular automated build utility is `make`, which, unfortunately, has unusual syntax. Following is an example of a `make` file:

```
calc: calc.o stack.o
        gcc -std=c99 -o calc calc.o stack.o
calc.o: calc.c stack.h
        gcc -std=c99 -c calc.c
stack.o: stack.c
        gcc -std=c99 -c stack.c
clean:
        rm *.o
```

The (default) name of the file is `Makefile`. Each dependency begins with the name of a target, which is followed by a list of files that it depends on. This is followed by a command line that begins with a tab character and specifies the action needed to regenerate the target. In this example, `clean` is a phony target because it is not the name of a file to be created. The command `make` is to be used with the desired target as its argument. If the command `make` is used without an argument, the first target is regenerated and those that it depends on.

Additional features, namely macros and automatic variables, are illustrated in the example below:

```
CC = gcc -std=c99
calc: calc.o stack.o
        ${CC} $^ -o $@
calc.o: calc.c stack.h
        ${CC} $< -c
stack.o: stack.c
        ${CC} $< -c
clean:
        rm *.o
```

Here `CC` is a macro, which is expanded using the evaluation operator `${ }`. Without the braces, the evaluation operator is applied only to the character that immediately follows. The automatic variables are the caret, representing the dependencies in the list above, the at-sign, representing the target, and the less-than-sign, representing the first dependency.

SCons is a build tool that allows one to specify dependencies and actions as a Python script. For the example of the `calc` program, one would create a file named `Sconstruct` with contents:

```
Program(['calc.c', 'stack.c'])
```

The command `scons` does the build; `scons -c` removes all targets including `calc`. Here is another example:

```
DefaultEnvironment(CC='gcc', CCFLAGS='-std=c11 -O2')
Program(['testdll.c', 'qr.c'], LIBS='gomp')
SharedLibrary('qr', '_qr.c')
```

Review questions

1. What kind of software tool is good for doing collaborative work and undoing changes?
2. What is the name of the process whereby a user of a version control system incorporates their changes into the (local) repository?
3. What is the name of the operation whereby a user of Git gets changes from a remote repository into his/her local repository? puts changes from his/her repository into a remote local repository?
4. Assuming a scenario in which a group of collaborators are working from a remote central repository, what would be the typical order in which the following Git operations are performed? `commit -a`, edit file(s), `fetch`, `push`.
5. If `git push` fails, what operations *must* be done before doing another `push`?
6. If `git merge` produces warnings about conflicts, what operations *must* be done before doing a `push`?
7. In a `make` file, what character is used exclusively to begin a command line? What character separates the dependencies from the target(s)?
8. What is the (default) name for the file executed by the `make` command?
9. If the command `make` is used without an argument, which target is created?
10. What is a phony target?
11. Convert the following `make` file into a `bash` script:

```
CC = gcc -std=c99 -c
testdll.dat: testdll.py twotests.py qr.py testdll _qr.so
<tab>   python $< > $@
testdll: testdll.o qr.o
<tab>   gcc -lgomp -ldl $^ -o $@
_qr.so: _qr.o
<tab>   ld -shared $^ -o $@
testdll.o: testdll.c
<tab>   ${CC} $<
qr.o.: qr.c
<tab>   ${CC} $<
_qr.o: _qr.c
<tab>   ${CC} -fPIC -O2 $<
```

12. Convert the following `bash` script into a `make` file supplemented by a phony target `clean` associated with the `rm` command:

```
gcc -std=c99 -fopenmp -fPIC factor.c -c
gcc -shared -lgomp factor.o -o _factor.so
rm factor.o
export OMP_NUM_THREADS=4
python solve.py > solve.dat
```

13. For the example of automatic variables and macros, replace every variable beginning with `$` with its value.

14. Classify each of the following tools as one of (a) version control system, (b) automated build utility, or (c) integrated development environment: Eclipse, SCons, make, subversion, Xcode & Visual Studio, IDLE, Mercurial, Git.

6.2 Algorithms

6.2.1 Loop invariants

To construct correct loops and to understand loops that others have written, it is helpful to identify the *loop invariant*. Informally, a loop invariant is the "organizing principle" for the loop, the key idea behind the loop.

As an example, consider the computation of

$$k = \text{argmin}_{0 \le i < n} a[i] \quad \text{or, equivalently} \quad k \text{ is such that } a[k] = \min_{0 \le i < n} a[i].$$

This might be computed by

```
k = 0
for i in range(1, n):
    if a[i] < a[k]: k = i
```

To identify the loop invariant, rewrite as a `while` loop:

```
k = 0
i = 1
while i < n:
    if a[i] < a[k]: k = i
    i = i + 1
```

Formally, a loop invariant is a condition (a logical expression)

1. that holds just before entering the loop and at the end of each iteration of the (equivalent `while`) loop, and

2. that, together with the termination condition for the loop, expresses the goal of the loop.

Here the *apparent* loop invariant is

```
#  a[k] = min{a[j] : 0 <= j < i}
```

and the termination condition is

```
#  i >= n
```

These do not quite suffice, so as an afterthought, we choose

```
#  a[k] = min{a[j] : 0 <= j < i}  and  i <= n
```

to be the loop invariant. One could actually test the loop invariant and the code by expressing the loop invariant as an assert statement:

```
assert  a[k] = min(a[:i])  and  i <= n
```

(The use of an `assert` statement to express a loop invariant would often require writing extra functions, however.)

Here is another example, for a binary search:

```
# Suppose a[0] < a[1] < ... < a[-1] and a[0] <= x < a[-1].
# Find i such that a[i] <= x < a[i+1].
i = 0; j = len(a) - 1
assert  a[i] <= x < a[j]  and  j > i
while j > i + 1:
    k = (i + j)//2
    if x < a[k]:  j = k
    else:  i = k
    assert  a[i] <= x < a[j]  and  j > i
```

After execution of the algorithm, the loop invariant

```
a[i] <= x < a[j]  and  j >i
```

together with the termination condition for the loop

```
j <= i + 1
```

ensure the success of the algorithm.

6.2.2 Computational complexity

Computational complexity is a technical term referring to the cost in time (CPU time) and space (memory) of executing an algorithm. Of special interest is the dependence of the cost on the size N of the input. For example, the standard algorithm for solving n linear equations in n unknowns requires $n^2 + n$ input values and has a running time proportional to n^3.

6.2.3 Parallel algorithms

The *parallel speedup* of a computation for p processors is the ratio of the computing time for a serial computation to the computing time for a parallel computation. The *parallel efficiency* is the speedup divided by the number of processors p.

Failure to achieve ideal speedup can be attributed to three factors (i) idle time due to load imbalance and serial execution of parts of the program, (ii) communication overhead, and (iii)

redundant computation due to use of a parallelizable algorithm. In particular, the parallel speedup of an algorithm as p increases is limited to $1/f$ where f is the fraction of the computation that is done serially, an observation known as *Amdahl's Law*. This fraction f typically decreases as the problem size n grows.

On the hand, *superlinear* speedup is possible for programs that use a lot of data and access it in such a way that there are frequent cache misses. With data distributed among several processors, the size of the cache becomes less of an issue.

Designing a parallel program involves the following two issues:

1. how to allocate data among nodes (and in memory) to minimize communication.
2. how to allocate computation among processors to balance the load.

Efficient allocation of computation depends on data allocation. Tools for automated load balancing are available.

If the parallel algorithm is such that the same instructions are being performed in parallel but on different data, it is said to *data parallel*; otherwise, it is simply *task parallel*.

loops Loops can be parallelized if there are *no dependencies* of an iteration of the loop on any previous iteration.

divide and conquer Simple examples of this principle are *reduction operations*, such as summing a set of values and finding the argmin of an array of real numbers. More substantial examples of note are the *fast Fourier transform* (FFT) and *quicksort*.

Review questions

1. Convert

   ```
   k = 0
   for i in range(1, n):
       if a[i] < a[k]: k = i
   ```

   to a while loop.

2. Show how by changing the termination condition for the `while` loop in the binary search algorithm, the loop invariant can be simplified.

3. Fill in the blanks for the following binary search algorithm:

   ```
   # Suppose a[0] < a[1] < ... < a[-1] and a[0] <= x < a[-1].
   # Find i such that a[i] <= x < a[i+1].
   i = 0; j = len(a) - 1
   assert  a[i] <= x < a[j]  and  j > i
   while j > i + 1:
       k = ____________________________
   ```

```
if x < a[k]:  __________
else:  __________
assert  a[i] <= x < a[j]  and  j > i
```

4. Use list comprehension and `sum` to construct an `assert`ion that expresses the loop invariant for Horner's rule, as given below:

```
n = len(c)
k = n - 1
px = c[k]
while k != 0:
    k -= 1
    px = c[k] + px * x
```

6.3 Floating-point Computation

This is a topic for which there are numerous widespread misconceptions. The actual situation is surprisingly simple and satisfying, thanks to the efforts of Turing Prize winner William Kahan. There are three main ideas here:

- The computer represents a finite set of values known as *machine numbers*.
- *Rounding* maps other values to the nearest machine number.
- Floating-point arithmetic rounds the exact result of an operation.

Machine numbers have precise values, which happen to have terminating decimal expansions. Testing for exact equality is acceptable!—if you know what you are doing. Addition/subtraction/multiplication with machine numbers having small integer values is exact; indeed, Matlab depends on this.

You can count on floating-point numbers.

6.3.1 Floating-point numbers

For binary floating-point arithmetic, the machine numbers are a finite subset of those values whose fraction part has a denominator which is a power of two. Hence, for example, $0.1 = 1/(2 \cdot 5)$ cannot be a machine number, whereas $0.125 = 1/2^3$ normally is. Two sets of machine numbers are specified by the IEEE Standard for Binary Floating-Point Arithmetic:

- Single precision with precision 24 and exponent range $[-126 : 127]$ inclusive.
- Double precision with precision 53 and exponent range $[-1022 : 1023]$ inclusive.

Specifically, a single-precision machine number x can be expressed

$$x = \pm \left(b_0 + \frac{b_1}{2} + \cdots + \frac{b_{23}}{2^{23}} \right) \times 2^e, \quad b_i \in \{0, 1\}, \quad -126 \le e \le 127.$$

This can be expressed as

$$\begin{aligned} x &= \pm \left(b_0 \cdot 2^{23} + \cdots + b_{22} \cdot 2 + b_{23} \right) \times 2^{e-23} \\ &= \langle \text{modest integer} \rangle \times 2^{\langle \text{small integer} \rangle} \end{aligned} \tag{6.1}$$

Additionally, there are representations for $\pm\infty$, *NaN* (not-a-number), and ± 0. Each NaN that is generated is a different object. The sign of zero is normally undetected. The Python `float` is double precision. Use `numpy.float32( )` to get a single precision value.

In Python, the double precision versions of $\pm\infty$, *NaN* (not-a-number), and ± 0 are represented by `float('inf')`, `float('-inf')`, `float('nan')`, 0.0, and -0.0, respectively. The `math` module and the `numpy` extension provide functions `isinf` and `isnan`.

To see the precise value of a machine number as a ratio of two integers, use the method `float.as_integer_ratio` . Alternatively, converting it to a string using format code '`%.767g`' shows all digits without trailing zeros.

denormalized numbers The representation of single-precision floating-point numbers is normalized so that either $b_0 = 1$ or $e = -126$. A number for which $b_0 = 0$ and $e = -126$ is said to be *denormalized*. Since its leading bit or bits (binary digits) are zero, it has fewer than 24 signficant bits. The inclusion of denormalized numbers eliminates the possibility of exponent underflow for addition/substraction.

6.3.2 Rounding

For any real value x, we define its rounded value $\mathrm{fl}(x)$ to be the machine number closest to x. In case of a tie choose the machine number whose last bit (24th in single precision) is zero. For example,

$$1, 1 + 2^{-23}, 1 + 2^{-22}$$

are consecutive machine numbers. The value $1 + 2^{-24}$ is midway between the first two and would be rounded down to 1 because its 24th significant bit is zero. The value $1 + 3 \cdot 2^{-24}$ is midway between the last two and would be rounded up to $1 + 2^{-22}$ because its 24th significant bit is zero. If x is closer to $\pm 2^{128}$ than it is to any machine number, define its rounded value to be $\pm\infty$, respectively (in the case of single precision).

Consecutive floating-point numbers can be obtained using the `numpy` function `nextafter` or the C `math` library function of the same name.

exponent overflow Exponent overflow is said to occur if x is finite but $\mathrm{fl}(x)$ is infinite.

exponent underflow There is no standard definition in that computer hardware designers have some leeway in deciding when to raise the underflow flag. It would be good to define underflow as a loss of accuracy due to the lower limit on the exponent range; that is, underflow occurs if the result would have been different with no lower limit on the exponent range. For example, $2^{-127} + 2^{-150}$ is representable with 24 bits with an exponent of -127 but must be rounded to 2^{-127} to be represented with an exponent of -126. By this definition, addition/subtraction do not generate underflows.

directed rounding The IEEE standard also specifies directed rounding modes: *round toward* $+\infty$, e.g, $\mathrm{fl}_\uparrow(1 + 2^{-24}) = 1 + 2^{-23}$, and *round toward* $-\infty$, e.g, $\mathrm{fl}_\downarrow(1 + 2^{-24}) = 1$.

6.3.3 Floating-point operations

Let $\circ$ be one of the arithmetic operations $+$, $-$, $\times$, or $/$. The corresponding floating-point operation $\hat{\circ}$ is defined by

$$a \,\hat{\circ}\, b = \mathrm{fl}(a \circ b), \quad \text{for machine numbers } a, b.$$

It is that simple. Similarly for the square root and decimal-to-binary conversion. Note that floating-point operations are defined only for machine numbers, contrary to what is implied by many textbooks.

The hardware can optionally trap *floating-point exceptions* such as exponent overflow, exponent underflow, and division by zero. The default is generally not to do so. In the case of Python, the default installation does not allow detection of floating-point exception; however, some of these are caught by the interpreter and raised as `Error`s. For NumPy `ufunc` operations (see Sec. 3.1.4), the handling of floating-point exceptions can be controlled using the function `numpy.seterr`.

For elementary functions it is impractical to maintain such high standards. It is enough that they round to the nearest or next-to-nearest machine number.

Also, the definition implies that for multiple operations, there is a rounding after each operation, e.g.,

$$a\hat{\times}b\hat{+}c = \mathrm{fl}(ab)\hat{+}c = \mathrm{fl}(\mathrm{fl}(ab) + c).$$

6.3.4 Other types of arithmetic

The Python standard library includes a `decimal` module that provides arbitrary precision decimal arithmetic and a `fraction` module that enables exact computation using rational arithmetic.

Rigorous computation with floating-point arithmetic is possible using *interval arithmetic*, in which values are represented by floating-point lower and upper bounds. This approach makes use of directed rounding. The package mpmath provides interval arithmetic.

Review questions

1. In Equation (6.1) what is the range for ⟨modest integer⟩ and for ⟨small integer⟩? Determine whether 1 000 000 000 is a machine number.

2. The following program does not terminate:

```
x = 1.
while x != 0.:
    x = x - 0.1
    print(x)
```

Explain *precisely* why this happens.

3. For a binary floating-point number system, describe in words the distribution of machine numbers along the real line. How are they spaced?

4. Which double-precision numbers are denormalized?

5. What are the two consecutive double-precision values that bracket $1 + 5 \cdot 2^{-53}$? To which value is $1 + 5 \cdot 2^{-53}$ rounded?

6. What, if anything, can we say about the accuracy of floating-point subtraction of two nearly equal machine numbers? *Hint:* What can we say about the exact result of subtracting two nearly equal machine numbers?

7. Give examples of floating-point arithmetic operations that will produce each of the exceptional values `Inf` and `NaN` (assuming no interception by the Python interpreter).

8. What is the *precise* value of `b` calculated by the following code?

```
a = 2.**(-512) +  2.**(-564)  # this is exact
b = a*a
```

Show your work step by step, and simplify your answer.

9. If `x` is a positive double-precision value and `x + 1. == x`, what *all* can we deduce about `x`? In other words, specify precisely the set of possible values of `x`.

6.4 Regular Expressions

Python Library Reference, 4.2.

A *regular expression* (regex) is a construct for specifying string patterns. A string *pattern* is a set of strings. (The patterns specified by regular expressions may also containing *anchors*, which restrict where the string can embedded within a longer string.) Regular expressions can be used in text editors like VIM/vi and emacs, in programs like grep, and in languages like awk, perl, and Tcl.

There are three levels of regular expressions: basic, extended, and Perl. Python and `grep -P` use Perl regexes; emacs regexes are somewhat different from these. (An emacs forward regex search begins with `esc` control-S.)

Here are some basic rules:

- Most characters represent themselves. To represent a special character precede it with a backslash.

- A dot represents any character.
- Quantifiers, `* + ?` follow the pattern they quantify.
- Parentheses are for grouping.
- Alternatives are separated by a vertical line. Because the *alternation* operator has low precedence, the expression must often be enclosed in parentheses.
- An enumerated set of single characters is enclosed in square brackets. In this context an initial caret means "not."
- At the beginning of a regular expression, a caret means "beginning of string" and at the end a dollar sign means "end of string." A restriction of this type is called an *anchor*.
- An abbreviation for `[0-9]` is `\d` .

Within a Python string, a backslash `\` is represented by `\\`, e.g., `'\\d+(\\.\\d*)?'`. Alternatively, use a raw string `r'\d+(\.\d*)?'`, which is recommended for regular expressions.

As an example, one might use

```
integer = r'0|[1-9]\d*'
```

to locate a properly formated integer in a string. The Python code

```
import re
match = re.search(integer, '102 -100 = 0102')
if match: print(match.group(), match.start(), match.end())
```

prints `102 0 3`. To find all nonoverlapping substrings that match, use

```
re.findall(integer, '102 -100 = 0102')
```

,

which returns `['102', '100', '0', '102']`. Note that quantifiers return the longest possible matches and that matches do not overlap. The last two matches, however, may not be what is intended. A better pattern may be

```
integer = r'(^|[^\d])(0|[1-9]\d*)($|[^\d])'
```

This requires that the integer be either at the beginning of the string or be preceded by other than a digit, and it requires that it either be at the end of the string or be followed by other than a digit. However, the substring we want should exclude a character that precedes or follows the integer. This is denoted by `match.group(2)` in this example; the 2 refers to the sub-regex between the 2nd left parenthesis and its matching right parenthesis. Text matching the entire regex itself is `match.group()`.

As a second example, consider those strings that represent a `float` value in Python. To be more specific, consider those strings `s` for which (i) `type(eval(s))` is `float` and (ii) `float(s)` does not raise a `ValueError`. A suitable regex will encompass both "F" format and "E" format. An F-formatted `float` will have exactly one decimal point and at least one digit:

```
real_f = r'[+-]?(\d*\.\d+|\d+\.)'
```

The order of two alternatives separated by "`|`" is important, because if a match is found for the first, the second one is not tried even if it matched more characters. An E-formatted `float` need not have a decimal point:

`real_e = r'[+-]?(\d*\.\d+|\d+\.?)[Ee][+-]?\d+'`
Combining the two:
`real = real_e + '|' + real_f` There is an exception to the rule that only the longest matches are found: for the alternation operator, the alternatives are checked left to right and the search ends when a match is found.

Review questions

1. What is the result of

```
import re
re.findall(r'(0|-?[1-9]\d*)', '102 -100 = 0102')
```

2. Write a regular expression for a string of zeros and ones which begins with a one. (Zero need not occur in the string.) The regular expression must match the entire string, not just a substring.

3. Write a regular expression for a dollar amount. The string must begin with "$" and have a decimal point preceded by one or more digits and followed by two digits. The first digit may not be a zero unless the amount is less than one dollar.

4. Write a regex that matches one or more trailing white space characters at the end of a string. A white space character (space, newline, return, tab, form) is denoted by "\s".

5. Define a regular expression that matches any *complete* string containing exactly two occurrences of a (lower case) "x". The following strings should produce a match:
   'xx'
   'axxa'
   'Xaxx'
   'xyx#'
   The following strings should not produce a match:
   'axa'
   'xaxx'

6.5 GUI Programming

A graphical user interface (GUI) is a hierarchy of *widgets* with the window as its root. Examples of widgets are button, label, text entry, menu, and slider. Some widgets are invisible. The GUI executes a loop driven by the processing of events. Each type of significant event has its own *event handler*.

The standard Python library includes a module `Tkinter` for building a GUI. An alternative to TKinter worth considering is wxPython.

6.5.1 Tkinter

The standard Python library module `Tkinter` is based on the `Tk` toolkit, which needs to be separately installed if not already present. Building a GUI with `Tkinter` is facilitated by the use of megawidgets, e.g., the Pmw toolkit for Python.

The following example is condensed from a Tkinter example in Langtangen(2006). It is typical to write the code as a class definition, but for clarity of exposition that aspect is omitted here. In Tkinter, the layout of the window is determined by the *geometry manager*, which is invoked here by calls to the `pack` method.

```
from Tkinter import *
root = Tk()

t36 = 'times 36'
text = Label(root, text='The sine of', font=t36)
text.pack(side='left')

r = StringVar()
entry = Entry(root, width=6, textvariable=r, font=t36)
entry.pack(side='left')

s = StringVar()
from math import sin
def onButton(): s.set(str( sin(float(r.get())) ))
button = Button(root, text=' = ', command=onButton, font=t36)
button.pack(side='left')

result = Label(root, textvariable=s, width=18, font=t36)
result.pack(side='left')

root.mainloop()
```

In this example, the only event is a mouseclick on the button.

6.5.2 Web interfaces

Dynamic web pages can be created in two ways:

- using Javascript or Java applets to be run on the client, and
- using a *CGI script* on the web server.

Javascript has a syntax similar to Java, but unlike Java, Javascript is a true scripting language in which values are typed rather than variables. CGI (Common Gateway Interface) is a protocol for responding to requests from web browsers. The CGI script communicates using `stdin`, `stdout`, and environment variables. It is typically written in PHP, Python, or Perl.

Review questions

1. What name is given to the parts of a GUI? And how are they structured?
2. What name is given to that part of Tkinter that determines the layout of the window?
3. What scripting language is run on the client to create a dynamic web page?
4. What is the standard way to interface a program on a web server to outside requests?

6.6 Programming Languages

6.6.1 Other scripting languages

Tcl/Tk Tcl is a scripting language and Tk is a toolkit for building GUIs. It is free, stable, and lightweight. The Python standard library contains is an interface to Tk called Tkinter.

Perl Perl is a popular scripting language with syntax from C and Unix shell scripting.

Lua Lua is a light-weight scripting language, which is said to interface easily to C.

6.6.2 Computer algebra systems

Mathematica is an engine for *evaluating expressions* using builtin and user-defined *rules*. The expression is the only data type, an example being

```
Plus[Times[a, b], c]
```

although the more familiar syntax

```
a b + x
```

is the default for communicating with the user. Expressions are entered for the purpose of evaluation, e.g.,

```
a b + c   →   a b + c
  1 2 + 3   →   5
```

The user can make new rules, e.g.,

```
a = 3
```

after which

```
a b + c   →   3 b + c
```

Although it supports a variety of programming paradigms, *Mathematica* is like Lisp fundamentally a *functional programming language*.

Maple is a programming language providing computer algebra, numerics, and graphics.

6.6.3 Matlab and Octave

Matlab is a dynamic programming language sold by The MathWorks that facilitates matrix operations, graphics, and building of user interfaces. The default data type is a 2-dimensional array of doubles. Also, Matlab has a Symbolic Math Toolbox.

Octave is a free software mostly compatible with Matlab that provides many of its functions for numerical computations and that supports plotting using gnuplot. Another Matlab clone is SciLab.

Octave m-files can be run from the operating systme command line but Matlab M-files cannot. Hence, Matlab cannot be used for writing scripts.

6.6.4 R

R is a popular programming language for statistical computing and graphics. It can be used for general matrix calculations.

6.6.5 Julia

Julia is a *high-performance* dynamic programming language released in 2012 for numerical computing with Matlab-like syntax released in 2012.

6.6.6 Fortran, C, and C++

C++ is a popular extension of C90 that facilitates object-oriented programming. It was standardized in 1998 with a revision in 2011. C++ is a complicated language with a lot of baggage from the early development of C. An alternative to C++ known as Objective C was adopted by Apple for software development. Java (from Sun Microsystems, now Oracle) and C# (from Microsoft) are languages based on C syntax that are targeted for the Internet.

Fortran, developed in 1956 by IBM and standardized in 1966, was the first real programming language. Fortran 77 was the second standard, and it was superseded by Fortran 90, followed by a minor revision in the form of Fortran 95. There was a major revision in the form of Fortran 2003, which makes Fortran object-oriented and interoperable with C, and there is a minor revision in the form of Fortran 2008.

Free compilers for major languages are available from the Free Software Foundation. It supports OpenMP for C, C++, and Fortran compilers. The latest version for C/C++ is GCC 4.6.1. The C++ compiler `g++` implements a few features of C++11. The `gfortran` compiler implements Fortran 95 and some features of Fortran 2008.

Here is a list of freely available Fortran, C, and C++ software for the solution of linear algebra problems.

Review questions

1. Which computer algebra programming language uses brackets rather than parentheses for function calls?

2. Which interpreted language is particularly popular for engineering calculations and simulations? statistical calculations?

6.7 Data

The `cPickle` module of the Python standard library can be used to store a Python object onto disk for later use.

6.7.1 XML

XML is a (verbose) language with syntax like HTML for representing data in a standardized hierarchical fashion so that it can be readily processed by computer. It is customizable to specific applications.

6.7.2 Relational databases

Data bases refer to the organization of data stored permanently, e.g., on hard disks. (Data structures on the other hand refers to the temporary representation of data in memory.)

It is popular to use relational data bases and the Structured Query Language SQL for accessing them. A well known free database management system (DBMS) is MySQL.

Review questions

1. What is the HTML-like language for describing data?
2. What is the popular standardized language for using relational data bases?

Appendix A

Functions

A.1 Polynomial approximation

Just as real numbers must be approximated on the computer by a finite set of machine numbers, so must functions also be approximated. The most common approximations use piecewise polynomials, discussed in the section that follows. As a first step, polynomials are introduced in this section. In addition to their simplicity, polynomials are special in the following sense: if $p(x)$ is a polynomial of degree n, so is $p(ax + b)$, assuming $a \neq 0$, i.e., the set of polynomials of a given degree has no intrinsic scale or origin.

A.1.1 Polynomial representation

The canonical representation of a polynomial of degree n is in terms of its coefficients:

$$c_n x^n + \cdots + c_1 x + c_0.$$

For numerical computation, these coefficients are machine numbers, and this can significantly limit the precision of such a representation. Specifically, changing the last bit, b_{23} or b_{52}, of a coefficient c_i can have a dramatic effect on the value of the polynomial and the polynomial with a halfway-between value for c_i cannot be approximated accurately. This incapacity for accurate approximation means that the canonical representation has low *precision*. (On the other hand, the factored form of a polynomial has very high precision; however, it is not practical in general.)

A practical high-precision representation is the Newton form, used, e.g., in *Mathematica*.

Another high-precision representation is the *Lagrange form*. In the Lagrange form, a polynomial of degree at most n is represented as a set of pairs (x_0, y_0), (x_1, y_1), $\ldots$, (x_n, y_n) where each y_i is the value of the polynomial at x_i. The nodes (also called abscissas) x_i are required to be distinct but not necessarily ordered. ("Distinct" is mathematics jargon meaning "all different.") Note that this representation requires $2n + 2$ values rather than $n + 1$ to be stored. However, redundancy in programming is often a virtue, either for reasons of efficiency or accuracy—in this case it is

accuracy. Because the Lagrange form generalizes easily to multiple dimensions, that is what we study here.

A.1.2 Existence and uniqueness

The Lagrange form is of general usefulness only if it uniquely specifies a polynomial. That it does is an important fact:

Proposition. Given pairs (x_0, y_0), (x_1, y_1), $\ldots$, (x_n, y_n), there exists a unique polynomial $p(x)$ of degree $\leq n$ such $p(x_i) = y_i$, $i = 0, 1, \ldots, n$ if and only if the nodes x_i are distinct.

If the nodes are not distinct, the data is either redundant or inconsistent.

A.1.3 Lagrange form

The most basic operation to be performed on a polynomial $p(x)$ is to evaluate it at a given value x. To illustrate this for the Lagrange form, consider the example

x_i	1	2	4
$y_i = p(x_i)$	1	1/2	1/4

The idea is first to construct nodal basis functions $l_0(x)$, $l_0(x)$, $l_0(x)$:

$$l_i(x_j) = \begin{cases} 1, & j = i, \\ 0, & j \neq i. \end{cases}$$

The conditions on each polynomial specify all its roots together with its leading coefficient:

$$l_0(x) = \frac{(x-2)(x-4)}{(1-2)(1-4)}, \quad l_1(x) = \frac{(x-1)(x-4)}{(2-1)(2-4)}, \quad l_2(x) = \frac{(x-1)(x-2)}{(4-1)(4-2)}.$$

The desired interpolant is a linear combination of the nodal basis functions:

$$p(x) = 1 \cdot l_0(x) + \frac{1}{2} \cdot l_1(x) + \frac{1}{4} \cdot l_2(x).$$

In general, for $n + 1$ pairs of values,

$$p(x) = \sum_{i=0}^{n} y_i \underbrace{\prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}}_{l_i(x)}.$$

If evaluation is expected at several values x, most of the computation can be done in a single preprocessing step requiring $\mathcal{O}(n^2)$ operations, with each evaluation requiring only $\mathcal{O}(n)$ operations. The preprocessing step is to compute quantities

$$d_i = y_i \prod_{j=0, j\neq i}^{n} \frac{1}{x_i - x_j}.$$

Evaluation then amounts to

$$p(x) = \left(\sum_{i=0}^{n} \frac{d_i}{x - x_i} \right) \prod_{j=0}^{n} (x - x_j),$$

which costs only $\mathcal{O}(n)$ operations. To prevent division by zero, cases where x equals a node must be handled separately. Also, there is the danger of exponent underflow or overflow when n is extremely large.

A polynomial *class* based on this approach might be defined as follows:

```
class Polynomial:
    def __init__(self, xa, ya):
        self._xa = xa[:]
        self._da = ya[:]
        for i in range(len(xa)):
            ... compute denom of self._da ...
        self._da /= denom
    def __call__(self, x):
        ... compute summ and prod using self._xa, self._da, and x ...
        return summ*prod
```

(*Note.* Recall that the syntax `xa[:]` causes `xa` to be copied. This would not be the case if `xa` were a Numpy array.)

A.1.4 Interpolation

If a polynomial approximation $p(x)$ of degree $\leq n$ is constructed to match exactly the values of a given function $f(x)$ at the nodes

$$p(x) = f(x), \quad x = x_0, x_1, \ldots, x_n,$$

this is called *interpolation*. There will be error at values of x other than the nodes. For high degree interpolation at equally spaced nodes, the error is quite large near the nodes at the two ends—*Runge's phenomenon*. This can be avoided by instead using Chebyshev nodes.

A.1.5 Bivariate linear polynomials

A linear polynomial in 2 variables x and y has the form

$$p(x, y) = ax + by + c.$$

It can be represented by its values at 3 points

$$p(x_i, y_i) = z_i$$

as long as the 3 points (x_i, y_i) are not collinear. (Equivalently, the three points (x_i, y_i, z_i) define a plane $z = ax + by + c$ in 3D.) Again arises the question of how to evaluate $p(x, y)$ at an arbitrary

point (x, y) given the data (x_i, y_i, z_i). Rather than derive a formula good only for the 2D case, we use matrix notation to derive a less obvious formula having an obvious generalization to higher dimensions. Begin by writing

$$\vec{r} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{and} \quad \vec{r}_i = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}, \quad i = 1, 2, 3.$$

(These 3D representations of points in 2D are known as homogeneous coordinates.) The unknown coefficients a, b, c satisfy

$$ax_i + by_i + c = z_i, \quad i = 1, 2, 3,$$

and this can be written

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{r}_3 \end{bmatrix} = \begin{bmatrix} z_1 & z_2 & z_3 \end{bmatrix}.$$

Therefore,

$$p(x, y) = \begin{bmatrix} a & b & c \end{bmatrix} \vec{r} = \begin{bmatrix} z_1 & z_2 & z_3 \end{bmatrix} \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{r}_3 \end{bmatrix}^{-1} \vec{r},$$

and the coefficients of the z_i are obtained using Cramer's rule.

The formula gives

$$l_1(x, y) = \frac{\det(\vec{r}, \vec{r}_2, \vec{r}_3)}{\det(\vec{r}_1, \vec{r}_2, \vec{r}_3)}.$$

for the coefficient of z_1. (The denominator can be shown to be nonzero if the three points in the x–y plane are not collinear.) We verify that this is, indeed, the 1st nodal basis function: The linearity of $l_1(x, y)$ follows from the fact that the determinant is a multilinear function (linear in each of its arguments):

$$\det(x\hat{\imath} + y\hat{\jmath} + \hat{k}, \vec{r}_2, \vec{r}_3) = x \det(\hat{\imath}, \vec{r}_2, \vec{r}_3) + y \det(\hat{\jmath}, \vec{r}_2, \vec{r}_3) + \det(\hat{k}, \vec{r}_2, \vec{r}_3).$$

Also, it is readily verified that

$$l_1(x_j, y_j) = \begin{cases} 1, & j = 1, \\ 0, & j \neq 1. \end{cases}$$

Indeed, $l_1(x, y)$ vanishes on the entire line that passes through (x_2, y_2) and (x_3, y_3). The other two nodal basis functions $l_2(x, y)$ and $l_3(x, y)$ are similarly defined, and

$$p(x, y) = z_1 l_1(x, y) + z_2 l_2(x, y) + z_3 l_3(x, y).$$

The values $(l_1(x, y), l_2(x, y), l_3(x, y))$ uniquely determine the point (x, y) in the plane and are known as areal coordinates.

Extension to higher degree bivariate polynomials is fairly easy using nodal basis functions constructed from the linear nodal basis functions.

Review questions

1. Consider the problem of determining a polynomial $p(x)$of degree $\leq n$ such that $p(x_i) = y_i$, $i = 1, 2, \ldots, m$ where the data x_i, y_i are given. Under what conditions on n and the data, can we be certain that $p(x)$ exists and is unique? What can happen if these conditions are violated?

2. Give the Lagrange form for the polynomial $p(x)$ of lowest degree such that $p(x_i) = y_i$, $i = 0, 1, \ldots, m$ where the data x_i, y_i are given.

3. Give the Lagrange form for the interpolating polynomial for a function f for which $f(-2) = -27, f(0) = -1, f(1) = 0$.

4. What is the form of a linear polynomial in two variables x and y? Data at how many points x_k, y_k are needed to uniquely determine a linear interpolant? What condition must be satisfied by this set of points to ensure the existence of a unique linear interpolant?

5. The equation $z = ax + by + c$ defines a plane in 3 dimensions. Give the value of a point through which this plane passes and a nonzero vector normal to this plane.

6. What is the form of a trivariate linear polynomial? Give a formula for the value of such a polynomial $p(x, y, z)$ in terms of data (x_i, y_i, z_i, u_i) where $u_i = p(x_i, y_i, z_i)$.

A.2 Piecewise Polynomial Approximation

A general approach to function approximation uses piecewise polynomials.

A.2.1 Piecewise polynomials

It important to recognize that

a piecewise polynomial is a function but not usually a polynomial.

More specifically, a piecewise polynomial is a function defined on some interval for which there exists a partition of the interval into subintervals such that on each subinterval the function equals some polynomial. Generally, the subintervals are chosen so that a piecewise polynomial coincides with a different polynomial on each of them. The boundaries between these subintervals are known as *knots* or *breakpoints*. The *degree* of a piecewise polynomial is the highest degree of any piece. If neighboring pieces share the same value at each knot, the piecewise polynomial will be *continuous*. If, in addition, the degree is one, the function is a C^0 piecewise linear polynomial.

Each piece of the polynomial can be represented in Lagrange form. The kth piece is a linear combination of nodal values with coefficients $l_{k,i}(x)$, $i = 0, 1, \ldots, n$ known as *shape functions*. They are not basis functions because they are to be used only on the kth subinterval. If a continuous piecewise polynomial is required to be continuous, this can be ensured by including the subinterval endpoints among the nodes for each piece.

A.2.2 Multivariate piecewise linear polynomials

The generalization of a set of adjoining subinterals to 2D is a set of adjoining triangles. The linear polynomial in each triangle is represented by its nodal values at the 3 vertices of the triangle. This ensures continuity of the piecewise linear polynomial (along triangle edges). Each triangle k has 3 shape functions $l_{k,i}(x, y)$, $l = 1, 2, 3$, which together with the 3 nodal values define the value of the piecewise linear polynomial inside triangle k.

To evaluate a piecewise polynomial at a given point (x, y), it is necessary to determine the triangle that contains it. The point (x, y) is in triangle k if and only if

$$l_{k,1}(x, y) \geq 0 \text{ and } l_{k,2}(x, y) \geq 0 \text{ and } l_{k,3}(x, y) \geq 0.$$

Doing a search to find the right triangle would be time consuming. This can be avoided with a preprocessing step in which a rectangular grid on the region is overlaid on the triangulation and a list is constructed for each gridcell of the triangles that overlap it. For evaluation, a simple calculation places a point in a gridcell and only a limited number of triangles need to be checked. Due to roundoff error, a point that is on or very near an edge separating a pair of triangles may fail the containment test for *both* triangles. To avoid this, do the test with each linear polynomial expressed as $(Ax + By + C)/d$, where the numerator and denominator correspond to the determinants in the numerator and denominator of Cramer's rule. In particular, do not perform the calculation as $ax + by + c$ with $a = A/d$, $b = B/d$, and $c = C/d$. It can be shown that the consistent use of recommended formula—even with floating-point arithmetic—avoids the problem of spurious gaps between triangles.

Piecewise linear polynomials are often used to interpolate scattered data in 2D or higher. An additional step is necessary, namely, to determine an appropriate triangulation of the convex hull of the nodes. The appropriate triangulation is most often the *Delaunay triangulation*, introduced in Section 3.3.

Review questions

1. Let $g(x)$, $a \leq x \leq b$, be a piecewise polynomial with knots $x_1 < x_2 < \cdots < x_{m-1}$. Let $x_0 = a$ and $x_m = b$, and let $p_k(x)$ be polynomials such that $g(x) = p_k(x)$ for $x_{k-1} < x < x_k$. Precisely what is the condition for $g(x)$ to be continuous?

2. Sketch a shape function for a piece of a piecewise linear polynomial in one variable.

3. Characterize the Delaunay triangulation of a set of points in the plane.

4. Define the empty circle property for a triangularization of a set of points in the plane.

5. Define the Voronoi diagram for a set of points in the plane.

6. Given the Voronoi diagram for a set of points in the plane, describe how to construct the Delaunay triangulation for that same set of points.

A.3 Trigonometric Approximation

A function is said to periodic of period P if $f(x+P) \equiv f(x)$. For convenience take $P = 2\pi$ so that

$$f(x + 2\pi) \equiv f(x).$$

For approximating such functions it is natural to use a linear combination of functions that have period 2π: 1, $\sin x$, $\cos x$, $\sin 2x$, $\cos 2x$, ...

More specifically, assume that N values for $f(x)$ are given at equally spaced increments $h = 2\pi/N$ of x, for convenience,

$$x = 0, h, 2h, \ldots, (N-1)h,$$

and the goal is to interpolate this data. Let

$$M = \lceil N/2 \rceil$$

so that $N = 2M$ if N is even and $N = 2M - 1$ if N is odd. We seek an approximation of the form

$$f(x) \approx u(x) = b_0 + \sum_{m=1}^{M-1} (a_m \sin mx + b_m \cos mx) + \begin{cases} 0, & N = 2M-1, \\ b_M \cos Mx, & N = 2M, \end{cases} \quad (A.1)$$

such $u(kh) = f(kh)$, $k = 0, 1, \ldots, N-1$. The calculation of $u(x)$ is formulated in Section A.3.1 using complex exponentials:

$$u(x) = \sum_{m=1-M}^{M-1} c_m e^{imx} + \begin{cases} 0, & N = 2M-1, \\ c_M \cos Mx, & N = 2M. \end{cases}$$

Power spectrum The trigonometric polynomial $u(x)$ in (A.1) can be expressed as a linear combination of sine waves

$$u(x) = \sum_{m=0}^{M} A_m \sin(mx + \phi_m)$$

where the amplitudes

$$\begin{aligned} A_0 &= |b_0| = |c_0|, \\ A_m &= (a_m^2 + b_m^2)^{1/2} = 2|c_m|, \quad m = 1, 2, \ldots, M-1, \\ A_M &= |b_M| = |c_M| \quad \text{if } N = 2M. \end{aligned}$$

The sequence

$$A_0, A_1, \ldots, A_{M-1}, A_M?$$

is known as the *power spectrum*. (Phase angles ϕ_m can be calculated as `math.atan2(`b_m`,`a_m`)`).

Least squares approximation The linear least squares approximation by a trigonometric polynomial of degree $q < M$ is given by

$$f(x) \approx \sum_{m=-q}^{q} c_m \mathrm{e}^{\mathrm{i}mx}.$$

This is a special property of these sine and cosine basis functions.

A.3.1 Fast Fourier Transform

The fast Fourier transform (FFT) is a fast method for multiplication by a very special kind of matrix, which converts real space data to frequency space data.

In particular, consider the case where one is given N values of some function $u(x)$ at equally spaced values of x, which for convenience we take to be

$$x = 0, h, 2h, \ldots, (N-1)h,$$

where $h = 2\pi/N$. It is often useful to represent the data as a trigonometric polynomial Let

$$M = \lceil N/2 \rceil$$

so that $N = 2M$ if N is even and $N = 2M - 1$ if N is odd. We seek a representation of the form

$$u(x) = b_0 + \sum_{m=1}^{M-1} (a_m \sin mx + b_m \cos mx) + \begin{cases} 0, & N = 2M-1, \\ b_M \cos Mx, & N = 2M, \end{cases} . \tag{A.2}$$

The reason for excluding $\sin Mx$ if $N = 2M$ is that it vanishes at the nodes $x = 2\pi k/N$, $k = 0, 1, \ldots, N-1$.

It can be shown that the coefficients a_m, b_m are determined from the data by multiplication by a certain matrix. The algebra simplifies significantly if sines and cosines are expressed in terms of complex exponentials. In Python this means using the function `cmath.exp` or `numpy.exp` (rather than `math.exp`) with an argument like `1j*float(m)*x` to represent $\mathrm{i}mx$. Using complex exponentials, $u(x)$ has the form

$$u(x) = \sum_{m=1-M}^{M-1} c_m \mathrm{e}^{\mathrm{i}mx} + \begin{cases} 0, & N = 2M-1, \\ c_M \cos Mx, & N = 2M, \end{cases}$$

where

$$\begin{aligned} c_0 &= b_0, \\ c_m &= \frac{1}{2}(b_m - \mathrm{i}a_m), \quad m = 1, 2, \ldots, M-1, \\ c_{-m} &= \overline{c_m}, \quad m = 1, 2, \ldots, M-1, \\ c_M &= b_M, \quad \text{if } N = 2M. \end{aligned}$$

Note that, by construction, the sum $u(x)$ must be real, even though individual terms might be imaginary. ("Imaginary" means not real; use the term "pure imaginary" if the real part is zero.)

Note. Real and imaginary parts of a complex number can be extracted using attributes `real` and `imag`.

To compute the coefficients of $u(x)$ from

$$u(0), u(h), \ldots, u((N-1)h),$$

use `numpy.fft.fft`. It returns an N-dimensional array with the values

$$[Nc_0, Nc_1, \ldots, Nc_{M-1}, Nc_M?, Nc_{1-M}, \ldots, Nc_{-1}]$$

where Nc_M is present only if $N = 2M$.

Motivation for the FFT is given in Appendix A.3.

Review questions

1. For approximating functions of period 2π, why does it suffice to consider only $\sin x$ and $\cos x$ and not $\sin(x + \phi)$ for a whole host of phase angles ϕ?
2. Consider interpolating a function $f(x)$ of period 2π at $2M$ equally spaced points 0, h, ..., $2\pi = h$, where $h = \pi/M$, by a trigonometric polynomial of degree $2M$. Why is it necessary to omit one term of the trigonometric polynomial? Which term is omitted and why?
3. What condition on the coefficients c_m of $u(x)$ is satisfied that ensures that $u(x)$ is real?
4. Relate the coefficients a_m and b_m to the coefficients c_m.
5. Obtain the amplitudes A_m of the power spectrum from the coefficients c_m.
6. Consider a function $f(x)$ of period 2π with values given at $2M$ equally spacedpoints 0, h, ..., $2\pi = h$, where $h = \pi/M$, and let $u(x)$ be its interpolant by a trigonometric polynomial of degree $2M$. What is its linear least squares approximation by a trigonometric polynomial of degree $2q$ where $q < M$?

Appendix B

Matrices

B.1 Matrix Computations

Many calculations, numerical and otherwise, can be expressed a matrix operations. Hence, the value of learning matrix algebra.

Recall the definition of a *matrix product* in which a table of inner products is computed using each row of the first matrix "dotted with" each column of the second matrix. The *outer product* is a special case of this. Recall the *identity matrix* I. Recall that a matrix A is *singular* if $\det A = 0$ and *nonsingular* (aka regular) otherwise. A nonsingular matrix A has an inverse A^{-1} such that

$$A^{-1}A = AA^{-1} = I.$$

Recall the concept of *linear independence* of a set of vectors x_1, x_2, ..., x_k. Also, there is the concept of the *rank* of a matrix. (Recall that a matrix's row rank equals it column rank.)

A system of linear equations $Ax = b$ to be solved will arise in many applications, e.g., interpolation. Recall the definition and properties of the *determinant* of a matrix. If A is nonsingular, there is a unique solution $x = A^{-1}b$; otherwise there is either no solution or infinitely many solutions. Whether a solution exists depends on whether b can be expressed as a linear combination of the columns of A.

B.1.1 Partitioned matrices

A partitioning $m = m_1 + m_2 + \cdots + m_p$ of the rows and a partitioning $n = n_1 + n_2 + + \cdots + n_q$ of the columns of an m by n matrix defines a partitioning of the matrix into blocks A_{ij}, $i = 1, 2, \ldots, p$, $j = 1, 2, \ldots, q$ where submatrix A_{ij} is m_i by n_j.

The "*partitioned matrix multiplication principle*" is the observation that the multiplication of two matrices A and B can be performed block by block if the two matrices have a *conformable partitioning*, meaning that the rows of B have the same partitioning as the columns of A. A conformable partitioning of A and B induces a partitioning of the product $C = AB$ in which C

inherits the row partitioning of A and the column partitioning of B. It can be shown that the (i, j)th block of C is given in terms of the blocks of A and B by the formula

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

where the summation is over the parts of the intermediate partitioning. This is the essence of the partitioned matrix multiplication principle.

As an example, the matrix–vector product Ax can be expressed as

$$\left[\begin{array}{c|c|c|c} a_1 & a_2 & \cdots & a_n \end{array}\right] \left[\begin{array}{c} x_1 \\ \hline x_2 \\ \hline \vdots \\ \hline x_n \end{array}\right] = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = x_1 a_1 + x_2 a_2 + \cdots + x_n a_n$$

where a_j is the jth column of A and x_j the jth element of x.

As another example, several systems $Ax_1 = b_1$, $Ax_2 = b_2$, ..., $Ax_m = b_m$ can be expressed as a single system $AX = B$ where $B = [b_1, b_2, \ldots, b_m]$ and $X = [x_1, x_2, \ldots, x_m]$.

B.1.2 Structured matrices

A matrix A is said to be *symmetric* if $A^{\mathsf{T}} = A$ where $^{\mathsf{T}}$ denotes the transpose. The symmetry is with respect to the main diagonal. The common occurrence of symmetric matrices is due to the fact that Hessians are symmetric.

A matrix is said to be *banded* if all of the nonzero elements lie in a band close to the main diagonal—close enough that efficiency is improved by storing only that part of the matrix ranging from the lowest subdiagonal to uppermost superdiagonal that define the band of nonzeros. A matrix is said to be *sparse* if the number of nonzero elements is few—so few that efficiency is improved by storing only the nonzeros together with their indices. If they are stored row by row, only column indices need to be recorded.

B.1.3 Efficient matrix operations

Most algorithms for linear algebra spend the bulk of the time executing a small set of basic operations such as

$$\text{set } y = ax + y$$

where a is a scalar and x and y are vectors. For this reason a set of optimized Basic Linear Algebra Subprograms (BLAS) have developed. In particular the operation above is performed by a call to `saxpy(n, a, x, incx, y, incy)`. The first letter "s" refers to single precision. Double precision is "d", complex is "c", and double precision complex is "z". There are three levels of BLAS:

1. Level 1: vector operations,
2. Level 2: matrix–vector operations,

3. Level 3: matrix–matrix operations.

Because performance is so dependent on the characteristics of the particular machine, there has been developed a program, Automatically Tuned Linear Algebra Software (ATLAS), that does automatic tuning at installation time for the BLAS and a few routines from LAPACK.

Review questions

1. The (i, j)th element of AB is the dot product of which two vectors?
2. Define the row rank of a matrix. the column rank.
3. What does the acronym BLAS stand for?
4. What is a "saxpy"?

B.2 Triangular Factorizations

A matrix factorization is generally a preprocessing step in the solution of a linear algebra problem.

The primary application of a triangular factorization is to compute a quantity such as $A^{-1}b$. However, there is actually *no need to compute a matrix inverse*. Just as $5/3$ is computed directly by division instead of as $3^{-1} \cdot 5$, so one should "predivide" by a matrix, as in $A \backslash b$. This is performed most efficiently by means of a triangular factorization, for which the total cost of computing $A^{-1}b$ is only $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops as opposed to $2n^3 + \mathcal{O}(n^2)$ flops for the use of matrix inversion or $n^3 + \mathcal{O}(n^2)$ flops for the elegant Gauss-Jordan algorithm. A *flop* is one floating-point operation, and $\mathcal{O}(n^2)$ represents some unspecified function that is bounded in magnitude by some unspecified number times n^2.

A triangular factorization might have the form $A = LU$ where U is upper triangular meaning that $u_{ij} = 0$ for $j < i$ and L is *unit* lower triangular meaning that $l_{ii} = 1$ and $l_{ij} = 0$ for $j > i$. For example, a 3 by 3 matrix would have a factorization

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

Clearly, the number of unknowns equals the number of scalar equations in $A = LU$, and if these are ordered appropriately, they can be solved explicitly. The matrix A need not be square. An LU factorization does not always exist, e.g., if

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

A special case in which existence is assured is when A is square and *strictly diagonally dominant*, meaning that

$$|a_{ii}| > |a_{i1}| + \cdots + |a_{i,i-1}| + |a_{i,i+1}| + \cdots + |a_{in}| \quad \text{for } i = 1, 2, \ldots, n.$$

Later we generalize the LU factorization so that it exists for any nonsingular matrix.

To apply a factorization to the calculation of $A^{-1}b$, express this as

$$A^{-1}b = (LU)^{-1}b = U^{-1}L^{-1}B = U\backslash(L\backslash b)$$

where inversion of the triangular factors is to be avoided by instead performing matrix predivision. The operation $b \mapsto U\backslash(L\backslash b)$ is known as a *backsolve*.

B.2.1 Division by a triangular matrix

There are two operations to be performed: $y = L\backslash b$ and $x = U\backslash y$.

Consider $x = U\backslash y$. This is equivalent to solving $Ux = y$, which when written in detail is

$$u_{ii}x_i + u_{i,i+1}x_{i+1} + \cdots + u_{in}x_n = y_i, \quad \text{for } i = 1, 2, \ldots, n.$$

These can be easily solved by *back substitution*

$$\begin{aligned} x_n &= y_n/u_{nn}, \\ x_i &= (y_i - u_{i,i+1}x_{i+1} - \cdots - u_{in}x_n)/u_{ii}, \quad \text{for } i = n-1, n-2, \ldots, 1. \end{aligned}$$

Python code follows:
Back substitution.

```
for i in range(n-1, -1, -1):   # n-1, n-2, ..., 0
    temp = y[i]
    for j in range(i+1, n):   # i+1, i+2, ..., n-1
        temp = temp - u[i, j]*x[j]
    x[i] = temp/u[i, i]
```

Improved efficiency is possible through vectorization. In particular, replace the `for j` loop by

```
if i < n-1: temp = temp - dot(u[i, i+1:n], x[i+1:n])
```

An alternative algorithm known as *back elimination* references the elements of the matrix U column by column instead of row by row.

Likewise, the calculation of $y = L\backslash b$ can be performed using either forward substitution or forward elimination.

Each division by a triangular matrix costs $n^2 + \mathcal{O}(n)$ flops, so the total cost of a backsolve is only $2n^2 + \mathcal{O}(n)$ flops.

If needed, the inverse A^{-1} can be calculated by observing that

$$A^{-1} = A^{-1}I = A^{-1}\left[\begin{array}{c|c|c|c} e_1 & e_2 & \cdots & e_n \end{array}\right] = \left[\begin{array}{c|c|c|c} A^{-1}e_1 & A^{-1}e_2 & \cdots & A^{-1}e_n \end{array}\right]$$

where e_i is a column vector with 1 in the ith position and zeros elsewhere. By exploiting the sparsity of the vectors e_i, the cost of the n backsolves can be kept to $\frac{4}{3}n^3 + \mathcal{O}(n^2)$ instead of $2n^3 + \mathcal{O}(n^2)$.

B.2.2 LU factorization

An LU factorization can be calculated by the forward elimination stage of Gaussian elimination:

$$\begin{matrix} \\ m_{21} \\ m_{31} \end{matrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \longrightarrow \begin{matrix} \\ \\ m_{32} \end{matrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}.$$

It can be shown that the resulting reduced matrix is the second factor U in an LU factorization of the original matrix A and the first factor is given by

$$L = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{bmatrix}.$$

In practice, the multipliers overwrite the eliminated elements of A and the nonzero elements of U overwrite the other elements of A.

Theorem B.1 *The multipliers and the reduced matrix computed by Gaussian elimination give an LU factorization of the original matrix.*

Forward elimination stage of Gaussian elimination.

```
for k in range(n-1):
    for i in range(k+1,n):
        m = a[i,k]/a[k,k]
        for j in range(k+1,n):
            a[i,j] = a[i,j] - m*a[k,j]
        a[i, k] = m
```

Note the beautiful symmetry in the innermost loop if we substitute in the value of `m`:

```
a[i,j] = a[i,j] - (a[i,k]/a[k,k])*a[k,j]
```

As stated earlier, the cost of this algorithm is $\frac{2}{3}n^3 + \mathcal{O}(n^2)$ flops. The *two* innermost loops can be vectorized using the `outer` function. Also, the nesting of the three loops can be changed. The above loop nesting is known as kij; the other possibilities are kji, ikj, jki, ijk, and jik. (The naming is based on the assumption that the indexing in the innermost loop is as shown above.)

Gaussian elimination can be considered as a recursive algorithm in that each step of the outside `k` loop reduces the problem of triangularizing an n by n matrix to that of triangularizing an $n-1$ by $n-1$ matrix. This perspective leads to a concise index-less description using partitioned matrices. With a $(1, n-1)$ by $(1, n-1)$ partitioning of the matrices in $LU = A$, we have

$$\begin{bmatrix} 1 & 0^\mathsf{T} \\ l & \hat{L} \end{bmatrix} \begin{bmatrix} \omega & u^\mathsf{T} \\ 0 & \hat{U} \end{bmatrix} = \begin{bmatrix} \alpha & b^\mathsf{T} \\ a & \hat{A} \end{bmatrix}.$$

Equating the 4 blocks gives

$$\begin{aligned} \omega &= \alpha, \\ u^\mathsf{T} &= b^\mathsf{T}, \\ l &= a/\omega, \\ \hat{L}\hat{U} &= \hat{A} - lu^\mathsf{T}. \end{aligned}$$

Gaussian elimination is, in general, numerically unstable. To fix this, one needs to incorporate row interchanges—*partial pivoting*. At the beginning of each `k` loop, elements in rows `k` through `n-1` of column `k` of the partial LU decomposition are compared to find the element `a[ell, k]` having the largest element (in magnitude). Then rows `k` and `ell` are interchanged, including multipliers.

Forward elimination stage of Gaussian elimination with partial pivoting.

```
p = list(range(n))
for k in range(n-1):
    ell = k
    for i in range(k+1,n):
        if abs(a[i,k]) > abs(a[ell,k]): ell = i
    p[k], p[ell] = p[ell], p[k]
    for j in range(n):
        a[k,j], a[ell,j] = a[ell,j], a[k,j]
    for i in range(k+1,n):
        m = a[i,k]/a[k,k]
        for j in range(k+1,n):
            a[i,j] = a[i,j] - m*a[k,j]
        a[i, k] = m
```

Theorem B.2 *The partial pivoting algorithm computes a factorization $PA = LU$ where P is a permutation matrix.*

For the algorithm above, P is a matrix with row `i` having a one in column `p[i]` and zeros in the other columns—if numbering is from `0` to `n-1`.

B.2.3 Symmetric matrices

A symmetric matrix A is *positive definite* if $x^\mathsf{T} Ax > 0$ whenever $x \neq 0$. For a symmetric matrix, Gaussian elimination without pivoting generates positive pivots (diagonal elements of U matrix) if and only if the matrix is positive definite.

Gaussian elimination without pivoting is stable for a symmetric positive definite matrix. A closely related algorithm is Cholesky factorization

$$A = GG^\mathsf{T}$$

where G is lower triangular with positive diagonal elements. The cost of a Cholesky factorization is $\frac{1}{3}n^3 + \mathcal{O}(n^2)$ flops, which is half the cost of an LU factorization.

Special efficient algorithms also exist for symmetric indefinite matrices.

B.2.4 NumPy and LAPACK functions

The module `numpy.linalg` has functions

```
solve(a, b)   # return a^-1 b
cholesky(a)   # return lower triangular g such that a = g g^T
```

The use of functions from the module `numpy.linalg.lapack_lite` requires a knowledge of storage order for array elements. In `'C'` storage order the elements of each row of a 2-dimensional

array `a` are stored in contiguous storage locations. The address increment for consecutive elements in a row of an array `a` is given by `a.strides[1]` and for `'C'` storage order this is equal to the number of bytes required to store one element, e.g., 8 for a double-precision value. In `'Fortran'` storage order the elements of each column are stored in contiguous storage locations. The address increment for consecutive elements in a column is given by `a.strides[0]`. The NumPy `copy` method has three variants: `a.copy('C')`, or simply `a.copy()`, makes a copy that uses `'C'` storage order; `a.copy('Fortran')` makes a copy that uses `'Fortran'` storage order; and `a.copy('Any')`, or simply `copy(a)`, makes a copy that uses the same storage order as `a`.

Because NumPy stores stride values, it can handle different storage orders. The NumPy function `transpose` performs the transpose on a 2-dimensional array by making a copy of the original except for the array elements themselves and interchanging the elements of the `strides` attribute. In particular, if `a` is stored in `'C'` order, `transpose(a)` will be in `'Fortran'` order. On the other hand, `fastCopyAndTranspose(a)` does a transpose in which the copy is stored in `'C'` order.

Another matter of importance concerns whether the data in a Numpy are in contiguous storage locations. For example, with

```
a = arange(36.); a.shape = 6,6
b = a[1:4:2,1:4:2]
```

there are gaps of size 1, 9, 1 between the 4 elements of `b`.

The module `numpy.linalg.lapack_lite` has functions
`dgesv(n, nrhs, a, lda, ipiv, b, ldb, info)`,
`dgetrf(m, n, a, lda, ipiv, info)`, and
`dpotrf(uplo, n, a, lda, info)`,
for computing $A^{-1}B$, factorization $PA = LU$, and factorization $A = GG^{\mathsf{T}}$, respectively. Because these functions are wrappers for the original Fortran implementations, their usage involves several complications. In the case of `dgetrf`, there are 4 things to note:

1. The array `a` should contain the transpose of the matrix A to be factored. On return `a` will be the transpose of the LU factorization of PA where P specified by `ipiv`.
2. The array `a` should be contiguous and stored in `'C'` order.
3. The array `ipiv` should have its `dtype` set to `'intc'` rather than `'int'`.
4. Because `info` is an "output" variable in the Fortran, it is set to zero, and instead the return value of `dgetrf` is a `dict`ionary with an entry for `'info'`.

Here is an example of usage:

```
m, n = a.shape
at = fastCopyAndTranspose(a)
ipiv = zeros(m, 'intc')
info = dgetrf(m, n, at, m, ipiv, 0)['info']
lu = fastCopyAndTranspose(at)
p = list(range(m))
for k in range(m-1):
    ell = ipiv[k] - 1
    p[k], p[ell] = p[ell], p[k]
```

Review questions

1. Give a high level explanation of how an LU factorization can be employed to solve a system $Ax = b$ of n equations.
2. Give in complete detail the algorithm for back substitution applied to a system $Ux = y$ where U is upper triangular.
3. Vectorize the back substitution algorithm.
4. Give in complete detail the algorithm for forward elimination applied to a system $Ly = b$ where L is unit lower triangular.
5. What are the two parts of a *backsolve* algorithm?
6. Give in complete detail the algorithm for Gaussian elimination for computing an LU factorization LU of a matrix A.
7. Contrast the number of operations required for an LU factorization with those needed to do a backsolve.
8. Vectorize the forward elimination stage of Gaussian elimination by replacing the 2 innermost loops by vector and matrix operations.
9. In the kij version of Gaussian elimination with partial pivoting what happens at the beginning of each k loop? Be specific about which array elements are involved.
10. In what sense does Gaussian elimination with partial pivoting compute a factorization? State your answer algebraically.
11. Define what it means for a symmetric matrix A to be positive definite.
12. If a matrix A is symmetric, how might we test in practice whether or not it is positive definite?
13. What is a Cholesky factorization of a symmetric positive definite matrix?

B.3 Orthogonal Factorizations

The least squares problem is to find x which minimizes

$$\|b - Ax\|_2$$

where A is an m by n matrix with $n \leq m$ and the 2-norm of a vector v is defined as $\|v\|_2 = (v^\mathsf{T} v)^{1/2}$. A solution always exists; it is unique if A is of full rank, in which case it can be shown to satisfy the *normal equations*

$$A^\mathsf{T} Ax = A^\mathsf{T} b,$$

whence

$$x = A^\dagger b \quad \text{where } A^\dagger = (A^\mathsf{T} A)^{-1} A^\mathsf{T}.$$

The matrix $A^\dagger$ is known as the Moore-Penrose pseudoinverse.

Sometimes only Ax is wanted; this is always unique.

The most obvious algorithm for computing

$$f(A,b) = (A^\mathsf{T}A)^{-1}A^\mathsf{T}b$$

is based on the decomposition

$$f = f_2 \circ f_1 \quad \text{where} \quad f_1(A,b) = (A^\mathsf{T}A, A^\mathsf{T}b) \quad \text{and} \quad f_2(M,c) = M^{-1}c.$$

However, f_2 is much more sensitive than is f, so this approach is numerically unstable.

B.3.1 Full and reduced QR factorizations

Two vectors u and v are said to be *orthogonal* if $u^\mathsf{T}v = 0$. A matrix Q is an *orthogonal matrix* if it is square and $Q^\mathsf{T}Q = I$.

A full QR factorization of an m by n matrix A where $n \leq m$ is a factorization $A = QR$ where Q is orthogonal and R is an m by n upper (or right) triangular. A reduced QR factorization is a factorization $A = Q_1R_1$ where Q_1 is an m by n matrix having orthonormal columns and R_1 is upper triangular. Given a full factorization QR, one constructs a reduced factorization by taking the first n columns of Q for Q_1 and the first n rows of R for R_1:

$$A = QR = \left[\begin{array}{c|c} Q_1 & Q_2 \end{array}\right] \left[\frac{R_1}{0}\right] = Q_1R_1.$$

That Q_1 has orthonormal columns uses the fact that

$$\left[\begin{array}{cc} Q_1 & Q_2 \end{array}\right]^\mathsf{T} \left[\begin{array}{cc} Q_1 & Q_2 \end{array}\right] = \left[\begin{array}{cc} I_n & 0 \\ 0 & I_{m-n} \end{array}\right]$$

where I_k denotes the k by k identity matrix.

To apply a reduced QR factorization to the calculation of $A^\dagger b$ when A is of full rank n, express this as

$$A^\dagger b = (Q_1R_1)^\dagger b = \cdots = R_1\backslash(Q_1^\mathsf{T}b).$$

(Note that $(AB)^\mathsf{T} = B^\mathsf{T}A^\mathsf{T}$, that $(AB)^{-1} = B^{-1}A^{-1}$, and $(A^\mathsf{T})^{-1} = (A^{-1})^\mathsf{T} = A^{-\mathsf{T}}$.)

The calculation using the full QR factorization is as follows:

$$A^\dagger b = R_1\backslash(\left[\begin{array}{cc} I_n & 0 \end{array}\right] Q^\mathsf{T}b).$$

The full QR factorization may be calculated using Householder reflections or Givens rotations. In both cases the matrix Q is represented in factored form. The reduced QR factorization is typically calculated using the modified Gram-Schmidt algorithm.

B.3.2 Singular value decomposition

The rank-deficient least squares problem, $\text{rank}(A) < n \leq m$, has infinitely many solutions x. For example, if

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 3 & 4 \end{bmatrix},$$

$A \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^{\mathsf{T}} = 0$ and any multiple of $\begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^{\mathsf{T}}$ can be added to x without changing the value of the residual $b - Ax$.

Of the all the solutions of a rank-deficient least squares problem, we choose the one having the smallest 2-norm. It is unique.

The SVD of a tall matrix A is $A = U\Sigma V^{\mathsf{T}}$ where U and V are orthogonal matrices and Σ is a diagonal tall matrix with diagonal elements $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. The Moore-Penrose inverse is defined to be $A^\dagger = V\Sigma^\dagger U^{\mathsf{T}}$ where $\Sigma^\dagger$ is Σ^{T} with each nonzero element replaced by its reciprocal. The solution of the least squares problem is $x = A^\dagger b$. Caution: the Moore-Penrose inverse does not have the properties that an inverse has; in particular $(AB)^\dagger$ is not necessarily equal to $B^\dagger A^\dagger$.

Determining rank is ill-posed; instead, one computes the nearness of an m by n matrix to a matrix of rank r for $r = n-1, n-2, \ldots$ and makes a judgment of the appropriate value of r. We use the Frobenius norm to measure nearness:

$$\|B - A\|_{\text{F}} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |b_{ij} - a_{ij}|^2 \right)^{1/2}.$$

It can be shown that the nearest rank r matrix B is obtained from the SVD $A = U\Sigma V^{\mathsf{T}}$ by setting the smallest $n - r$ singular values, σ_{r+1}, σ_{r+2}, $\ldots$, σ_n to zero. If the columns of U are denoted by u_i and those of V by v_i, then B can be expressed as a linear combination of outer products:

$$B = \sigma_1 u_1 v_1^{\mathsf{T}} + \sigma_2 u_2 v_2^{\mathsf{T}} + \sigma_r u_r v_r^{\mathsf{T}}.$$

For small $r \ll n$ this is a compressed representation of $B \approx A$.

B.3.3 NumPy and LAPACK functions

The module `numpy.linalg` has functions

```
qr(a, mode='full')   # returns Q, R
lstsq(a, b, rcond=-1)   # returns a^dagger b, resids, rank, s
svd(a, full_matrices=1, compute_uv=1)
    # returns singular value decomposition  U, Sigma, V^H
```

The module `numpy.linalg.lapack_lite` has functions

```
dgeqrf(m, n, a, lda, tau, work, lwork, info),
dorgqr(m, n, k, a, lda, tau, work, lwork, info)
dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info),
```

`dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)`,
for computing a QR factorization, the Q_1 part of a QR factorization. $A^\dagger B$, and a singular value decomposition, respectively

Review questions

1. Using matrix and norm notation, state the linear least squares problem.
2. Give necessary and sufficient conditions for the existence of a solution to the linear least squares problem $Ax \approx_2 b$.
3. Give necessary and sufficient condition for the uniqueness of a solution to the linear least squares problem of minimizing $\|b - Ax\|_2$ where A has more rows than columns.
4. Express $f(A, b) = (A^\mathsf{T} A)^{-1} A^\mathsf{T} b$ as a decomposition $f = f_2 \circ f_1$ where f_2 denotes the solution of a system of linear equations given a square coefficient matrix M and a right-hand side vector c.
5. What is an orthogonal matrix?
6. Prove that the product of two orthogonal matrices is orthogonal.
7. If Q is an orthogonal matrix, what is the possible range of values for q_{ij}?
8. Show that if a matrix Q is square and $Q^\mathsf{T} Q = I$, then it follows that $QQ^\mathsf{T} = I$ also.
9. Under what condition is $I - \tau v v^\mathsf{T}$ an orthogonal matrix?
10. Define a full QR factorization of an m by n matrix A where $n \leq m$.
11. Define a reduced QR factorization of an m by n matrix A where $n \leq m$.
12. Given a full QR factorization of an m by n matrix A, $n \leq m$, describe how to construct a reduced QR factorization.
13. Complete the details for the proof that $A^\dagger = \cdots = R_1^{-1} Q_1^\mathsf{T}$.
14. What is the effect of multiplication of a vector v of dimension $m \geq n$ by an n by m matrix $\begin{bmatrix} I_n & 0 \end{bmatrix}$.
15. For which type of QR factorization is the orthogonal matrix typically not explicitly computed? And how is it represented?
16. If the linear least squares problem has more than one solution x, what additional condition is typically imposed to ensure uniqueness?
17. Let A be m by n matrix, $n \leq m$. Give the form of a singular value decomposition. Be specific about the ordering of the singular values.

18. Let A be m by n matrix, $n \leq m$, and let $U\Sigma V^{\mathsf{T}}$ be a singular value decomposition of A with singular values σ_1, σ_2, ..., σ_n. Give a formula for the Moore-Penrose inverse $A^{\dagger}$. Be complete.

19. What is the Moore-Penrose inverse of a 1 by 1 matrix?

20. Let $U\Sigma V^{\mathsf{T}}$ be a singular value decomposition of an m by n rank-deficient matrix A of rank r. Then we can write

$$\Sigma = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{bmatrix}, \quad U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}, \quad V = \begin{bmatrix} V_1 & V_2 \end{bmatrix},$$

where Σ_1 is r by r, U_1 is m by r, and V_1 is n by r. Use this partitioning to construct a "reduced" SVD of A and of its Moore-Penrose inverse $A^{\dagger}$.

B.4 Spectral Factorizations

Any function $f(x)$ that can be expressed as a finite or infinite sum of positive and negative powers of x can be defined for a matrix in the obvious way—though it may not be well defined for all matrices. The solution of a linear system of (algebraic) equations with coefficient matrix A requires the product of $f(A) = A^{-1}$ times a vector; the solution of a system of linear (ordinary) differential equations with constant coefficient matrix A requires the product of $f(A) = \exp(tA)$ times a vector where t is the independent variable.

For $f(A)$ to be well defined requires that f be to be well defined at all *eigenvalues* of A. The eigenvalues of a square matrix A are the roots, λ_1, λ_2, ..., λ_n, of its *characteristic polynomial*

$$p(\lambda) = \det(\lambda I - A).$$

Note that eigenvalues are either real or complex conjugate pairs of imaginary numbers. (Imaginary means not real.)

If $A = XBX^{-1}$, it can be shown that

$$f(XBX^{-1}) = Xf(B)X^{-1},$$

which simplifies the calculation if B is a "simpler" matrix than A. The transformation $B = X^{-1}AX$ is known as a *similarity transformation* and B is said to be *similar* to A. It can be shown that similar matrices have the same eigenvalues.

In almost all cases, a nonsingular matrix X can be chosen so that

$$X^{-1}AX = \Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n),$$

in which case we say that A is *diagonalizable*. Otherwise, the matrix is said to be *defective*, for example,

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

is defective. The problem of computing a function of a diagonal matrix reduces to the scalar case:

$$f(\mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)) = \mathrm{diag}(f(\lambda_1), f(\lambda_2), \ldots, f(\lambda_n)).$$

If A is diagonalizable, we can write

$$AX = X\Lambda,$$

and, if X is partitioned into columns x_1, x_2, ..., x_n, this becomes

$$\left[\begin{array}{c|c|c|c} Ax_1 & Ax_2 & \cdots & Ax_n \end{array}\right] = \left[\begin{array}{c|c|c|c} x_1\lambda_1 & x_2\lambda_2 & \cdots & x_n\lambda_n \end{array}\right].$$

The vector x_i is called the *eigenvector* corresponding to the eigenvalue λ_i.

B.4.1 General matrices

Note that the eigenvalues of a triangular matrix are simply its diagonal entries, so to get eigenvalues it suffices to find a similar matrix that is triangular. It can be shown that every matrix A is similar to some upper triangular matrix R if complex numbers are used. Moreover, the transformation matrix can be chosen to be a *unitary matrix* U, which is a matrix satisfying

$$U^{\mathsf{H}}U = I$$

where $U^{\mathsf{H}} = \bar{U}^{\mathsf{T}}$ is the complex conjugate transpose of U. For example,

$$\begin{bmatrix} 1+2\mathrm{i} & 4\mathrm{i} \\ 7 & 1-3\mathrm{i} \end{bmatrix}^{\mathsf{H}} = \begin{bmatrix} 1-2\mathrm{i} & 7 \\ -4\mathrm{i} & 1+3\mathrm{i} \end{bmatrix}.$$

A unitary matrix has $U^{-1} = U^{\mathsf{H}}$, and, hence, is a generalization of an orthogonal matrix to the complex case. Such a factorization

$$A = URU^{\mathsf{H}}$$

is known as the *Schur form*. The diagonal elements of R are the eigenvalues of A.

It is generally desirable to avoid imaginary numbers.

Theorem B.3 (real Schur form) *Every real (square) matrix A can be expressed as*

$$A = QRQ^{\mathsf{T}}$$

where Q is orthogonal and R is block upper triangular with diagonal blocks that are either 1 by 1 or 2 by 2 with imaginary eigenvalues.

An example is

$$R = \left[\begin{array}{cc|c} 1 & -3 & \mathsf{x} \\ 1 & 1 & \mathsf{x} \\ \hline 0 & 0 & -2 \end{array}\right].$$

Theorem B.4 *The eigenvalues of a block triangular matrix are the eigenvalues of its diagonal blocks.*

The algorithm for finding eigenvalues is necessarily iterative, but the iterations converge very rapidly. The total cost is $\approx 10n^3$ flops to get the eigenvalues and $\approx 25n^3$ for both the eigenvalues and eigenvectors (reference: page 188 of Scientific Computing: An Introductory Survey).

B.4.2 Symmetric matrices

It can be shown that an orthogonal similarity transformation preserves symmetry; hence the upper triangular matrix R in the real Schur form of a symmetric matrix A is diagonal:

$$A = Q\Lambda Q^\mathsf{T}$$

where Q is orthogonal and Λ is a real diagonal matrix. It follows that a symmetric matrix has a complete orthogonal set of eigenvectors.

The total cost is only $\approx \frac{4}{3}n^3$ flops to get the eigenvalues and $\approx 9n^3$ for both the eigenvalues and eigenvectors (same reference as above).

The singular values and singular vectors can be obtained form eigenvalues and eigenvectors of $A^\mathsf{T}A$ and AA^T,

$$A^\mathsf{T}A = V\mathrm{diag}(\sigma_1^2, \ldots, \sigma_n^2)V^\mathsf{T}, \quad AA^\mathsf{T} = U\mathrm{diag}(\sigma_1^2, \ldots, \sigma_n^2, 0, \ldots, 0)U^\mathsf{T},$$

but it is more efficient and accurate to use specialized algorithms.

B.4.3 NumPy and LAPACK functions

The module `numpy.linalg` has functions

```
eig(a)    # returns eigenvalues and vectors Lambda, X
eigvals(a)    # returns eigenvalues Lambda
eigh(a, UPLO='L')
    # returns eigenvalues and vectors Lambda, X of symmetric matrix
eigvalsh(a, UPLO='L')    # returns eigenvalues Lambda of symmetric matrix
```

The module `numpy.linalg.lapack_lite` has functions

`dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)` and `dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)`

for computing the eigenvalues/eigenvectors of a general and a symmetric matrix, respectively.

Review questions

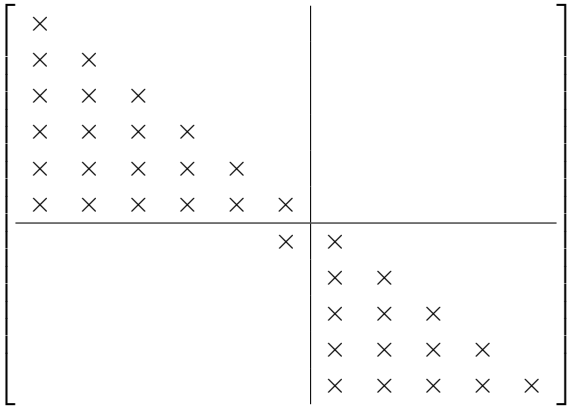
1. Why can we be sure that a 3 by 3 real matrix has at least one real eigenvalue?
2. What does it mean for a matrix A to be similar to B? What is a diagonalizable matrix? Give the form of a spectral decomposition of a diagonalizable matrix, identifying eigenvalues and eigenvectors.
3. What is a defective matrix?
4. Define the Euclidean norm of a complex vector $x = [x_1\ x_2\ \cdots\ x_n]^\mathsf{T}$?

B.5 Parallel Algorithms

B.5.1 Divide and conquer

An example of this principle is *nested dissection*. Here we present a very simple example of it.

Consider a system of linear equations having as a coefficient matrix a lower triangular matrix that is nearly decoupled:



The lower left hand block is a rank one matrix, which is the crucial property that makes divide and conquer attractive. So consider, more generally, the systems of equations

$$\begin{bmatrix} L_1 & 0 \\ u_2 v_1^{\mathsf{T}} & L_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

where L_1 is an n_1 by n_1 lower triangular matrix, u_2 is a column vector of height n_2, v_1^{T} is a row vector of width n_1, and L_2 is an n_2 by n_2 lower triangular matrix. Solving this algebraically yields the solution

$$\begin{aligned} x_1 &= L_1 \backslash y_1 \\ x_2 &= L_2 \backslash y_2 - (L_2 \backslash u_2)(v_1^{\mathsf{T}} x_1) \end{aligned}$$

The major part of the computation, $L_1 \backslash y_1$, $L_2 \backslash y_2$, $L_2 \backslash u_2$, can be done in parallel on 3 processors yielding an overall speedup of perhaps 2. If so, the efficiency is 2/3.

B.5.2 Relaxation methods and reordering

Gaussian elimination can be impractical for very large sparse systems of equations even if sparsity is fully exploited. The only practical approach is an iterative method in which a succession of better approximations are obtained.

In practice, iterative methods are applied to matrices that are sparse (or matrices for which there exists a fast method to perform a matrix–vector product). Typically, the coefficient matrix A has only $\mathcal{O}(n)$ nonzero elements.

Relaxation methods

Begin by writing the ith equation so that the ith unknown is solved for in terms of the others:

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j - \sum_{j=i+1}^{n} a_{ij} x_j \right), \quad i = 1, 2, \ldots, n.$$

For *Jacobi relaxation*

$$x_i^{\text{new}} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{\text{old}} - \sum_{j=i+1}^{n} a_{ij} x_j^{\text{old}} \right), \quad i = 1, 2, \ldots, n.$$

For *Gauss-Seidel relaxation*

$$x_i^{\text{new}} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{\text{new}} - \sum_{j=i+1}^{n} a_{ij} x_j^{\text{old}} \right), \quad i = 1, 2, \ldots, n,$$

which typically converges twice as fast since it always uses the most recent estimates for the values of the other unknowns. Jacobi is guaranteed to converge for a strictly diagonally dominant matrix and Gauss-Seidel is guaranteed to converge for a symmetric positive definite matrix. However, both methods are very slow. Generalizations such as *block relaxation* and *successive over-relaxation (SOR)* are faster. And some other iterative methods are even faster. Still, Gauss-Seidel is quite useful in two contexts: (i) as a preconditioner for algebraic iterative methods such as conjugate gradient, and (ii) as a smoother for multigrid methods.

A representative example of a symmetric positive definite matrix is a discrete Laplacian on an $M+1$ by $N+1$ rectangular grid. It is convenient to use double indices to label the unknowns and the corresponding equations, even though mathematically the unknowns are considered to constitute a solution vector. The unknowns are u_{ij}, $0 < i < M$, $0 < j < N$ and the (i, j)th equation is

$$u_{ij} = \frac{1}{4}(u_{i-1,j} + u_{i,j-1} + u_{i,j+1} + u_{i+1,j}).$$

Values of u_{ij} for $i = 0$, $i = M$, $j = 0$, and $j = N$ are given. Hence, there are $n = (M-1)(N-1)$ unknowns and equations. If we assume lexicographic ordering, u_{11}, u_{12}, $\ldots$, $u_{M-1,N-1}$, Gauss-Seidel is

$$u_{ij}^{\text{new}} = \frac{1}{4}(u_{i-1,j}^{\text{new}} + u_{i,j-1}^{\text{new}} + u_{i,j+1}^{\text{old}} + u_{i+1,j}^{\text{old}}).$$

This algorithm has limited parallelism. The values u_{ij}, $i+j =$ constant, can be updated in parallel, but this still requires $M + N - 3$ parallel relaxations to do one sweep.

Red/black ordering

It is useful to associate a graph with a symmetric matrix, especially one which is positive definite. The graph has nodes labeled 1, 2, $\ldots$, n, and there is an edge connecting two distinct nodes i and j if $a_{ij} \neq 0$. Two nodes are said to be *adjacent* if they share an edge.

For a discrete Laplacian on a rectangular grid, the graph of its matrix is just the rectangular grid with the boundary nodes removed. For this graph it is possible to color the nodes red and black so that no two adjacent nodes have the same color. The values of the red nodes can be relaxed knowing only values for the black nodes and vice versa. Hence, if the nodes are ordered such that all the red nodes precede all the black nodes, then one sweep can be accomplished with only two parallel relaxations.

Review questions

1. For the system of linear equations

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

   write down the equations showing how x_i^{new} is obtained from x_i^{old} for one sweep of

   (a) Jacobi relaxation,

   (b) Gauss-Seidel relaxation.

2. The problem of a discrete Laplacian on a rectangular grid can be written as $Ax = b$. What are A and b for $M = 3$, $N = 2$ assuming lexicographic ordering?

3. Define the graph of a symmetric matrix.

4. Consider a system of linear equations in which the unknowns and equations are labeled with a double indices (i, j). Assume that equation (i, j) depends only on unknowns labeled (i, j), $(i \pm 1, j)$, and $(i, j \pm 1)$. Give a formula for partitioning indices between red and black.