CALIFORNIA INSTITUTE OF TECHNOLOGY
Division of Physics, Mathematics, and Astronomy

# Ay190 – Computational Astrophysics

Christian D. Ott, Andrew Benson, and Michael W. Eastwood
cott@tapir.caltech.edu, abenson@tapir.caltech.edu, mweastwood@astro.caltech.edu
December 21, 2014

**Notes for Winter Term 2013/14**

Document under development.

2014-05-22

Revision 3b05d41b2672eca004666eb5c52ae786aa518131

Last changed by Christian D. Ott

# Table of Contents

# Chapter I

# Introduction

These lecture notes were developed as part of the Caltech Astrophysics course Ay190 – Computational Astrophysics. This course was first taught in the winter term 2010/2011 by Christian Ott and Andrew Benson. Most of the present version of the notes were typed up by Christian Ott in the winter term 2011/2012 and work is ongoing to improve the notes and add material in the winter term 2013/2014.

The goal of this course is to briefly introduce the basics of numerical computation and then focus on various applications of numerical methods in astronomy and astrophysics.

The parts of these lecture notes dealing with the basics of numerical computation were heavily influenced by a set of lecture notes by Ian Hawke at the University of Southampton.

## I.1 Literature

Unfortunately, there is no good single reference for Computational Astrophysics. We have personally used the following books for inspiration for Ay190:

- *Numerical Methods in Astrophysics, An Introduction* by P. Bodenheimer, G. P. Laughlin, M. Rozyczka, and H. Yorke. Taylor & Francis, New York, NY, USA, 2007.
- *Numerical Mathematics and Computing*, 6th edition, by W. Cheney and D. Kincaid, Thomson Brooks/Cole, Belmont, CA, USA, 2008.
- *Finite Difference Methods for Ordinary and Partial Differential Equations*, by R. Leveque, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.
- *An Introduction to Computational Physics*, 2nd edition, by Tao Pang, Cambridge University Press, Cambridge, UK, 2006
- *Scientific Computing – An Introductory Survey*, 2nd edition, by Michael T. Heath, McGraw-Hill, New York, NY, USA, 2002.
- *Numerical Recipes*, 3rd. edition, by W. H. Press, S. Teukolsky, W. Vetterling, and B. Flannery, Cambridge University Press, Cambridge, UK, 2007.

## I.2 Acknowledgements

CDO wishes to thank Ian Hawke for sharing his class notes for inspiration.

The authors are indebted to Mislav Balokovic, Kristen Boydstun, Yi Cao, Trevor David, Sarah Gossan, Roland Haas, Stacy Kim, Io Kleiser, Ben Montet, J. Sebastian Pineda, John Pharo, Sherwood Richers, and Arya Farahi for spotting typos and factual errors.

## I.3 Future Additions

The idea of these lecture notes is that they will be continuously improved and expanded with more background material, more details, figures, and example problems. In this section we summarize concrete plans of what will be added in the near future.

- ODE: More details on stiff equations and stability.
- ODE integrators: Crank-Nicholson implicit integrator.
- PDEs/ODEs: More on the Method of Lines. Studies of the wave equation, parabolic, and elliptic PDEs.
- Application: Stellar structure with Henyey Solver.
- Multi-dimensional hydrodynamics and MHD.
- Numerical relativity and GR(M)HD.
- More details on Riemann solvers, e.g., a derivation of the HLLE solver, other Riemann solvers.
- Statistical data analysis, time series analysis, and approaches to marginalization in Bayesian analysis (MCMC, nested sampling).

# Chapter II

# General Computing Basics

In this chapter, we introduce the computational science basics needed in this class: `Python`, `git`, and `LaTeX`.

## II.1 `Python`

The primary programming language we will use in this class is `Python`. Strictly speaking, `Python` is actually a scripting language. Traditional programming refers to building applications or operating systems by compiling source code into binary machine code with a compiler. Such source code is often close to the machine level and uses low-level operations (such as explicit memory assignments, pointers, etc.). Scripting means programming at a high and flexible abstraction level using source code (i.e. scripts) that are *interpreted* rather than compiled.

`Python` is open source and free software, unlike other standard software used in physics and astronomy, e.g., IDL, Mathematica, and Matlab. Once you have gotten your feet wet in `Python` and have seen its tremendous potential and the immense amount of free and open-source code libraries for physics/astro applications available in `Python`, you will probably see no need to go back to proprietary closed-source software.

`Python` is now (as of 2013/12/24) available in version 3.4, but everything described in this section should hold for any version greater than 2.7.

There are many great resources available for learning `Python` and as references for advanced features. We particularly recommend the following:

- *Python Scripting for Computational Science* by Hans Petter Langtangen, Third Edition, Springer Verlag, is a great book that provides both an introduction to `Python` basics and an in-depth discussion of many advanced features useful for physics and astronomy.
- The **Khan Academy** provides a set of online lectures on `Python` that are extremely useful and require no background in any kind of programming. https://www.khanacademy.org/science/computer-science-subject/computer-science.
- http://python.org has a beginners guide to `Python`: https://wiki.python.org/moin/BeginnersGuide.
- http://learnpythonthehardway.org/book/ is a detailed, step-by-step tutorial for learning `Python`.

- http://www.pythonforbeginners.com/ is another website/blog providing an introduction to Python.
- http://scipy-lectures.github.io/ has an edited and curated set of lecture notes on Scientific Python.
- http://python4astronomers.github.io/ **Practical Python for Astronomers** is a series of hands-on workshops to explore the Python language and the powerful analysis tools it provides. The emphasis is on using Python to solve real-world problems that astronomers are likely to encounter in research.
- http://www.astropython.org/ **AstroPython** – Python for Astronomers.
- http://stackoverflow.com is a question & answer site for any kind of programming question. You'll find answers to most of your python questions there. The site is indexed by Google, so just typing in your question into google will do the job.

In the following, we assume that you have a working Python installation on your laptop that includes the NumPy, SciPy, and matplotlib packages. We also assume (*i*) that you are working with some kind of unix-based operating system, e.g. Linux or MacOS X and (*ii*) that you have basic familiarity with text editors and the command line (shell) in a terminal. If you are not familiar with the unix shell, you may want to consider reading this: http://cli.learncodethehardway.org/book/.

We will be working with simple script files that we will edit with a text editor (e.g., emacs or vi) and then run the script from the command line. If you are looking for a more integrated development environment (IDE) as you may be used to from Matlab, you have a number of options, which are described at https://wiki.python.org/moin/IntegratedDevelopmentEnvironments. We recommend Spyder: http://pythonhosted.org/spyder/. There is also the ipython shell (http://ipython.org) for Python, which is useful for trying out things interactively.

### II.1.1 Hello World in Python: Running your first script

Open a new file, enter the following, then save it as helloworld.py.

```
#!/usr/bin/env python
print "Hello World"  # this prints out "Hello World"
```

The first line in this script is actually a comment. Comments in python are prepended by the hash (#) symbol. Anything that comes after that symbol is ignored by the python interpreter. Now, the first line is special yet: it's an indication to the shell that the script is to be executed by a program named python. In case there are several python programs (e.g., different Python versions on your system, /usr/bin/env makes sure that the first python program encountered in the directories listed in your system PATH environment variable will be used.

You can now run the script from the shell (i.e. from your terminal command line) by either giving it as an argument to python,

```
$ python helloworld.py
Hello World
```

or by making it first executable and then running it directly:

```
$ chmod u+x helloworld.py
$ ./helloworld.py
Hello World
```

It is for the second case that the `#!/usr/bin/env python` comment is crucial, because the shell will parse this and then run the script with the first `python` it finds.

Congratulations! You have just run your first `Python` script. Granted, it's exceedingly simple, but it's a good start. In the following sections, we will introduce a number of key aspects of the `Python` langauge. Much of this will be by example and less-than-formal. You can always look up details on the web. Remember: google is your friend!

### II.1.2 Basic Code Elements of `Python`

#### II.1.2.1 Importing Modules

`Python` comes with a large assortment of packages (called *modules*) that provide functionality not included in the `Python` core. Essentially everything that goes beyond basic arithmetic and the simplest of text processing needs external modules. These must be imported at the top of the script.

Here is an example:

```
#!/usr/bin/env python

# import the sys and math modules
import sys, math

a = math.pi / 2.0
print math.sin(a)

sys.exit() # leave program here

print "Hi! :-)"
```

So we have imported the `sys` and the `math` module. `math` provides a broad range of useful functions and constants for mathematical operations[1]. And its *members*, e.g., constants like $\pi$ or the `sin` function are accessed by `math.membername`. The same is, of course, true for other modules. `sys.exit()` is a handy function that forces the script to exit execution at any point. You will find this increadibly useful later, when you will be debugging more complex code. In the above example, the second print command will never be executed.

There are actually multiple ways to import modules.

```
import * from math
```

will import everything that's in math and merge all members into the main *namespace* (the set of functions/variables direcly known to the `Python` interpreter without having to prepend a module name). This means, you can access members of `math` directly, e.g.,

```
a = pi / 2.0
print sin(a)
```

[1] See http://docs.python.org/2/library/math.html for what it provides in `Python` 2.

This way of importing modules has advantages and disadvantages. The obvious advantage is that you have less code to write. The disadvantage is that there could be *name clashes* if you load multiple modules that have members of the same name(s).

Here is a compromise:

```
import math as m
a = m.pi / 2.0
print m.sin(a)
```

In this case, you import the `math` module and give it the shorter name `m`. This saves text, but still avoids any name clashes. Alternatively, if you know you need only one or a few members of a module, you can import them explicitly:

```
from math import pi,sin
a = pi/2.0
print sin(a)
```

#### II.1.2.2 Simple Math

Here are some examples for basic mathematical operations:

```
#!/usr/bin/env python

import math as m

a = 2.0
b = 1.0
c = a+b
d = c**2     # square c
e = m.sqrt(c) # take the sqrt of c
f = d/c # divide d by c

# print results
print a,b,c
print d
print e
print f
```

Note that `Python` is a dynamically typed language. Unlike `C` or `Fortran`, `Python` does not require you to define variables and give them explicit types. `Python` will do its best to figure out what you want as it interprets your script. Of course, this means that you have to be somewhat careful about what you really want it to do.

For example, in the above, `a = 2.0` initializes the variable `a` as a floating point number (because of the decimal). `a = 2` would initialize it as an integer. Integer arithmetic in `Python` would give `1/2 = 0`, while `1.0/2.0 = 0.5`. If you use integers in an expression with other numbers that are floating point, it will convert the integer to a floating point number and your result will be a floating point number unless you force its conversion back to an integer with the `int()` function:

```
#!/usr/bin/env python
a = 2
b = 1.0
```

```
print a+b
print int(a+b)
```

The first line of the output will be `3.0`, the second line will be just `3`.

#### II.1.2.3 Conditional Statements and Indentation

Conditional statements control the program flow, executing one branch if a condition (expressed as a Boolean expression) is *true* and another branch if it is *false*. The most basic conditional statement is the `if` statement. Here is a trivial, but useful example:

```
#!/usr/bin/env python

import numpy as np

# get a random number in [0.0,1.0)
a = np.random.random()

if a < 0.5:
    print "a = %5.6g is < 0.5!" % (a)
    print "This is fun!"
else:
    print "a = %5.6g is >= 0.5!" % (a)
    print "This is okay, I guess"
```

There are a number of things to say about this. First note that we are using the `numpy` module here, which we will discuss in more detail in §II.1.3. The code calls `numpy.random.random()` to get a random floating point number between 0 and 1. The `if` statements tests if $(a < 0.5)$ is true. If so, it prints one thing, if not it prints another thing. The `print` statement used here also prints the random number. Note that the format strings work just as in `C/C++` and the variable(s) to be output is/are put in parentheses behind the string, separated by a percent sign (%). We will explore string formatting more in §II.1.2.6.

Note that the code inside the two branches of the `if-else` construct is **indented**. It is indented by 4 characters (that's the amount you get if you hit the "tab" key in `emacs`) in your text editor. Indentation defines a block of statements that are executed under the same conditions in `Python`. This is similar to the way one groups statements inside curly brackets {...} in `C/C++`.

We can construct more complicated `if` statements in `Python`. For example, connecting to the above, check out this code fragment:

```
if (a > 0.1) and (a < 0.5):
     print "a = %5.6g is > 0.1 and < 0.5!" % (a)
     print "This is fun!"
elif (a <= 0.1):
     print "a = %5.6g is <= 0.1!" % (a)
     print "This is okay!"
else:
     print "a = %5.6g is >= 0.5!" % (a)
     print "This is great!"
```

This is an `if-elif-else` construct. The keyword for an "or" comparison is just `or` and a test if two variables have equal values is accomplished by `a == b` (not recommended for floating point

numbers, because they could essentially be equal but differ in insignificant digits!). Note that the brackets around the expressions in the `if`-statement are optional, but useful to logically group the various expressions in a complex construct.

#### II.1.2.4 Loops

As in most programming languages, `for` and `while` loops are also essential elements of `Python`. Here is an exceedingly simple example of a `for` loop:

```
#!/usr/bin/env python

for i in range(100):
    print i
```

This will just print out the numbers 0 to 99. `Python` indexing, like `C/C++` indexing, starts at 0. Note the indentation again!

The `range` function is core `Python` and allows you to specify the range of `for` loops. A loop with `range(100)` will run from 0 to 99, but a loop with `range(1,101)` will run from 1 to 100.

Here is an example for a `while` loop that does exactly the same as the `for` loop in the above example:

```
#!/usr/bin/env python

i=0
while(i < 100):
    print i
    i += 1
```

If you have an array *a* of length `len(a)` and you want to loop over all its elements, you would do this with

```
for i in range(len(a)):
     ...
```

#### II.1.2.5 Functions

It is often useful to put some functionality that is needed many times into a *function* that can then be called. Here is a simplistic `Python` example:

```
#!/usr/bin/env python

def mysquare(x):
    return x**2

a = 2.0
print mysquare(a)
```

It's also possible to return more complex structures, say two variables:

```
#!/usr/bin/env python

def mysquare(x,y):
```

```
    return x**2, y**2

a = 2.0
b = 4.0

c,d = mysquare(a,b)
print c,d
```

#### II.1.2.6 Screen Output, Strings, and Formatting

We have already seen that the `print` statement is used to output strings, variables, and constants on the screen. It is often useful to first construct a character string and then output it to the screen using `print` or to a file. `Python` has extremely powerful string creation and manipulation functionality. We can only begin to describe it here and it is very worthwhile to spend some time reading the various online resources about string manipulation in `Python`.

Here are some examples of things one can do with strings in `Python`.

```
#!/usr/bin/env python

import sys,time,math

astring = "Hi"
bstring = "there"
cstring = astring+" "+bstring+"!"
print cstring

tstring = "The current date/time is %s" % (time.ctime())
print tstring

dstring = "This is Pi to 5 digits: %6.5g" %(math.pi)
print dstring

eexp = 1.0e51
estring = "The explosion energy of a supernova is %15.6E ergs." % (eexp
   )
print estring

fstring = "I like to repeat myself.\n"
print (fstring*5)[:-1]
```

And this is the output of the above:

```
Hi there!
The current date/time is Thu Dec 26 08:39:44 2013
This is Pi to 5 digits: 3.1416
The explosion energy of a supernova is    1.000000E+51 ergs.
I like to repeat myself.
I like to repeat myself.
I like to repeat myself.
I like to repeat myself.
I like to repeat myself.
```

Most of the above can be understood quite intuitively if one knows C/C++-style formatting, which goes like this:

```
%[width].[precision][type]
```

Width sets the amount of space reserved for what is to be output. Precision only makes sense for floating-point numbers and gives the number of significant digits output. Type is how a given variable is to be represented in the string. The type must obviously fit the type of the variable. Width and precision are optional arguments. Here is a list of frequently used types:

- s string
- d integer
- f float
- g general number format that rounds to the precision specified and switches to scientific notation for large numbers
- E scientific notation

There is one more thing to point out in the above example code:

```
(fstring*10)[:-1]
```

The expression in parentheses is `fstring` repeated 5 times and the `[:-1]` strips the last character, a trailing carriage return (`\n`), off the combined string.

#### II.1.2.7 Basic File I/O

**Reading Files**

Simple text files can read in the following way:

```
#!/usr/bin/env python

# open the file
infile = open("infile1.txt","r")

# read the data
indata = infile.readlines()

# close the file again
infile.close()
```

`open` returns a file object whose member function `readlines()` is called to read the the file as a list (array) of strings, each entry being precisely one line. This may not necessarily what you want. In astronomy, one often needs to read in columns of numerical values. For simple file formats (just row-column), this is best accomplished with the `numpy` function `loadtxt`, which we will discuss in the next section §II.1.3. If the file format is so complicated that `loadtxt` can't handle it, then you are back to basic `Python` and must process the strings read in. In the following, we give an example of how this can be done relatively conveniently.

Suppose you want to read in a text file that looks like this:

```
# Here is some text
0.0000000000E+00,0.0000000000E+00
1.8868424346E-02,1.8867094876E-02
3.7736848692E-02,3.7726213718E-02
```

```
5.6605273038E-02,5.6569384424E-02
7.5473697384E-02,7.5388642756E-02
9.4342121729E-02,9.4176036262E-02
1.1321054608E-01,1.1292362820E-01
1.3207897042E-01,1.3162350145E-01
```

So you have two columns of data, separated by a comma. And there is some kind of comment string on the first line. Let's pretend the first column is time and the second column is some data that vary with time. Here is how you would read and parse this file:

```
#!/usr/bin/env python

import string
import numpy as np

# open the file
infile = open("infile2.txt","r")

# read the data
indata = infile.readlines()

# close the file again
infile.close()

# get number of data lines in the file
# (-1, because first line is a comment)
n = len(indata)-1

# allocate two numpy arrays
time = np.zeros(n)
data = np.zeros(n)

# parse the data
for i in range(n):
    # use the string module's split function
    # to split each line into two strings
    splitline = string.split(indata[i+1][:-1],",")
    time[i] = float(splitline[0])
    data[i] = float(splitline[1])
```

This example already makes use of numpy arrays, which we will discuss in more detail in the next section §II.1.3. There are two key things here: (1) We use the `split` function of the `string` module to split up each line into two strings, making use of the fact that the columns are separated by commas. (2) Each of the two resulting strings is then individually converted to a floating point number and stored in an array.

### Writing Files

Writing files is also very easy:

```
#!/usr/bin/env python

import string
import numpy as np
```

```
# open the file
outfile = open("outfile.txt","w")

# how many lines do we want to output
n = 1000

# write a header
headerstring = "# This is some text\n"
outfile.write(headerstring)

for i in range(n):
    ran1 = np.random.random()
    ran2 = np.random.random()
    outstring = "%10.10E,%10.10E\n" % (ran1,ran2)
    outfile.write(outstring)

outfile.close()
```

`write` works like `print` for screen output with the difference that it does not automatically append a carriage return (also called a newline character; `\n`) to the end of the string. This is why we include `\n` in all strings to be written to the file.

### II.1.3 Scientific Computing with `NumPy` and `SciPy`

`Python` proper does not come with a set of routines for standard numerical operations and advanced computations. Also, while it provides *lists* that behave much like arrays in other languages, these lists are very high level and inefficient, making the handling of large amounts of data very slow.

`SciPy` and `NumPy` fix these deficiencies. `SciPy`, short for `Scientific Python`, provides a very large range of routines for numerical analysis and computation. The complete ever growing list of routines is available at http://docs.scipy.org/doc/scipy/reference/. Presently (as of 2013/12/16), `SciPy` provides functionality for integration, optimization, root finding, interpolation, Fourier analysis, signal processing, linear algebra, and statistics. `SciPy` also provides functionality for reading and writing binary files and to integrate `Python` scripts with native `C` code. All `SciPy` routines are efficiently implemented and practically as fast as `C/C++` code.

In your script, you can import `SciPy` like any other `Python` module:

```
import scipy
```

This gives you access to basic `SciPy` functionality. Some `SciPy` submodules must be loaded explicitly (one finds this, unfortunately, often only via trial and error). For example, to gain access to the interpolation routines, you import

```
import scipy.interpolate
```

`NumPy`, short for Numerical Python, is a `Python` module that provides multi-dimensional arrays with a high-level interface, but fast, vectorized operations. It also implements a number of basic math functions and operations and some advanced features, such as random number generators, Fourier transform etc. (these overlap partially with functionality provided by `SciPy`).

In your script, you can import `NumPy` by the simple import statement

```
import numpy as np
```

(`import numpy` by itself would of course work too, but you'll be using `numpy` so many times that it is just easier to use the `np` abbreviation.)

A great set of lecture notes on `SciPy` and `NumPy` is available here: http://scipy-lectures.github.io/. The authors of these notes provide a great example demonstrating the power of `NumPy` arrays:

```
#!/usr/bin/env python

import numpy as np
import timeit

# define a function that defines
# and fills a list, then squares each
# list entry by iterating throught the list
def listtest():
    L = range(1000)
    [i**2 for i in L]

# define a function that defines
# a numpy array by filling it and that
# then squares it
def arraytest():
    a = np.arange(1000)
    b = a**2

ntrials = 100000
# use the timeit module to execute
# and time the listtest function
time = timeit.timeit('listtest()',\
                     setup="from __main__ import listtest",\
                     number=ntrials)
print "Time for Python list: %5.3f microseconds" % (time/ntrials*1.0e6)

# use the timeit module to execute
# and time the arraytest function
time = timeit.timeit('arraytest()',\
                     setup="from __main__ import arraytest",
                     number=ntrials)
print "Time for NumPy array: %5.3f microseconds" % (time/ntrials*1.0e6)
```

This script uses the `timeit` module to time the execution of two functions:

- `listtest()` sets up a standard `Python` list with 1000 elements, filled with numbers from 0 to 999. It then iterates over it, squaring each element. The code in square brackets is equivalent to the following loop and it is just a shorter way of writing it:

  ```
  for i in range(len(L)):
      L[i] = L[i]**2
  ```

- `arraytest()` sets up a `NumPy` array with 1000 elements, filled with numbers from 0 to 999. It then uses a `NumPy` array operation to square all entries.

The call to `timeit.timeit()` is a bit convoluted and has to do with the particular way this module works. Don't worry about it now, but focus on the output that the script produces when run on a 2013 MacBook Air:

```
Time for Python list: 86.887 microseconds
Time for NumPy array: 3.154 microseconds
```

Impressive, isn't it? Are you convinced to use `NumPy` arrays?

#### II.1.3.1 Working with `NumPy` Arrays

Since `NumPy` arrays are so important for computing with `Python`, we provide a number of examples of how to create and use them in this section. We assume that your script imports `NumPy`:

```
import numpy as np
```

**(a) Creating `NumPy` Arrays**

Manually creating a 1D array works like this:

```
a = np.array([0.0,1.0,2.0,3.0])
```

The individual elements are then accessed by subscribing a with an index, starting at 0. So `a[1]` will access the second element, which is set to 1.0.

Manually creating a 2D array works like this:

```
a = np.array( [[0.0,1.0,2.0], [3.0,4.0,5.0]] )
```

There are more convenient ways to create arrays, of course.

```
a = np.arange(1000)
```

is something that we have already seen. It creates an array with 1000 integers ranging from 0 to 999. Note that `arange` is not good for floating point numbers because of the way it calculates the number of entries in the array. It can, for example, happen that the last element in the resulting array is actually larger than the specified range. It is saver to use `linspace` to set up floating point arrays. It's usage is as follows

```
fa = np.linspace(START,STOP,NPOINTS)
```

where `START` is the start value, `STOP` is the end value, and NPOINTS are the number of array entries that will be used to cover the interval `[START,STOP]`. So

```
fa = np.linspace(0.0,1000.0,10000)
```

will create an evenly spaced array `fa` with 10000 entries `fa[0] = 0.0`, `fa[9999] = 1000.0`, and separation between two consecutive entries of `fa[i+1]-fa[i] = 0.100010001`. If you want a spacing of precisely `0.1`, then you need to use

```
fa = np.linspace(0.0,1000.0,10001)
```

It is also possible to generate arrays and initialze them with zeros, ones, or random numbers:

```
a = np.zeros(1000,float)  # make a float array of 1000 zeros
b = np.zeros(1000,int)    # makes an int array of 1000 zeros
c = np.zeros( (1000,1000), float) # makes a 2D float array
```

```
d = np.ones( (1000,5), float) # makes a 2D array with 1000 rows, 5 cols
# makes a 4x4 2D array of random floats in [0,1)
e = np.random.random( [4,4] )
```

### (b) Basic NumPy Array Functions

Here are some examples of basic NumPy array functions:

```
a = np.random.random( [5,6] )
print np.ndim(a)  # returns number of dimensions (2)
print len(a) # returns size of the first dimension
print len(a[0,:]) # returns the size of the second dimension
print a.max() # returns max entry
print a.min() # returns min entry
# returns indices of max entry
print np.unravel_index(a.argmax(), a.shape)
# returns indices of min entry
print np.unravel_index(a.argmin(), a.shape)
print a.sum() # returns sum of all array elements
print a[0,:].sum() # returns sum of first row
print a[:,0].sum() # returns sum of first column
print a[0,0:2].sum() # returns sum of elements 0 and 1 in first row
```

Almost everything that can be applied to a scalar can also be applied to a NumPy array:

```
a = np.random.random(1000)
b = np.sqrt(a)
c = a*b
d = c**2
e = 10.0**d
f = np.sin(e)
```

Such full array operations are much more efficient (= faster code) than looping over the array and carrying out pointwise operations.

Note that in the first example, we have picked parts of a 2D array to evaluate some function on it. Picking a part of an array is called *slicing* and we shall discuss this process in more detail in the following.

### (c) Indexing and Slicing NumPy Arrays

Python indices begin at 0.

```
a = np.arange(10)
print a[0],a[4],a[-1]
```

returns

```
0 4 9
```

and

```
print a[::-1]
```

prints out the array in reverse order.

We can slice a NumPy array in many ways:

```
# take every other element of a and put it in b
b = a[::2]
# put the entries 1 through 7 of a into c
c = a[1:8]
# put every other entry of 1 through 7 of a into d
d = a[1:8:2]

# let's make a 2D array
e = np.random.random( [4,6] )
# assign all rows of the first column of e to f
f = e[:,0]
# assign all columns of the first row of e to g
g = e[0,:]
```

Intelligent array slicing can get one around having to use slow loops. For example, for computing a first-order forward finite difference approximation to a derivative of some discrete data $f_i(x_i)$, we want to compute

$$f'(x_i) = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \ .$$

Naively, we would implement this in the following way:

```
fprime = np.zeros(len(f)-1)
for i in range(0,len(f)-1):
    fprime[i] = (f[i+1]-f[i]) / (x[i+1]-x[i])
```

(Since we rely on the `i+1` element, we cannot compute a value for `fprime` for the last entry of `f`).

We can accomplish the same much more efficiently by slicing:

```
fprime = (f[1:] - f[:-1]) / (x[1:] - x[:-1])
```

If `f` and `x` have 10 entries (with indices 0 to 9), then `f[1:]` gives a slice from index 1 to 9 and `f[:-1]` gives a slice from 0 to 8, so precisely what we want for the derivative. The slicing example is about a factor of 100 faster than the loop example (try timing it for a large array!).

An important thing to remember is that a slicing operation in `NumPy` creates a so-called *view* of the original array and not an actual copy. This means the slice and the original array still share the same memory block and a change of an element in the slice will change this element in original array as well. To get an actual copy when slicing, one needs to append a `.copy()` to the slicing operation:

```
a = np.random.random(1000)
b = a[100:200] # slice (view) of a from index 100 to 199
c = a[100:200].copy() # actual copy of a from index 100 to 199
```

**(d) Fancy Indexing/Slicing**

`NumPy` supports the use of boolean expressions to index/slice arrays. In this case, the data are actually copied. Let's say you have an array `a` of random numbers between 0 and 1 and wanted to pick out only those smaller than or equal to 0.5:

```
a = np.random.random(1000)
b = a[a <= 0.5] #copy all elements of a that are <= 0.5 into b
```

#### II.1.3.2 Reading/Writing Files with `NumPy`

`NumPy` provides convenience functions for reading in standard ASCII row-column data, for example:

```
# This is a data file
0.0000000000E+00 0.0000000000E+00 0.0000000000E+00
1.8868424346E-02 1.8867094876E-02 8.8998748180E-05
3.7736848692E-02 3.7726213718E-02 3.5592766446E-04
5.6605273038E-02 5.6569384424E-02 8.0058480509E-04
7.5473697384E-02 7.5388642756E-02 1.4226337339E-03
9.4342121729E-02 9.4176036262E-02 2.2216037271E-03
[...]
```

Here is the `NumPy` code for reading, manipulating, and writing the data:

```
#!/usr/bin/env python

import numpy as np

# read file
data = np.loadtxt("infile3.txt",comments="#")

# do something with data, e.g., square them
data[:,1:3] = data[:,1:3]**2

# write result
np.savetxt("outfile.txt",data,fmt="%10.10E",\
               header="This is a data file")
```

### II.1.4 Plots with `matplotlib`

`matplotlib` is a powerful `Python` module for plotting/visualizing all kinds of data. The capabilities of `matplotlib` are vast and cannot possibly all be described here. We refer the reader to the `matplotlib` documenation at http://matplotlib.org/. Directly importing `matplotlib` (for some reason) does not give access to the basic plotting routines and one needs to import `matplotlib.pyplot` to get them:

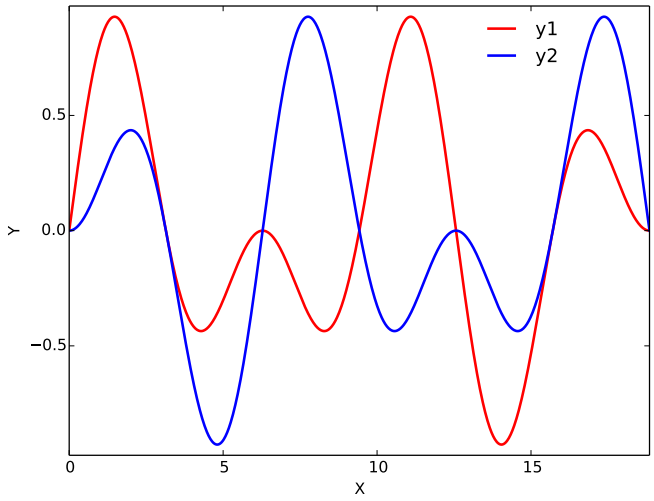
```
import matplotlib.pyplot as pl
```

Below is an example script that reads data and plots simple line graphs to the screen. The data file you need for this is available from http://www.tapir.caltech.edu/~cott/ay190/code/infile3.txt.

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as pl

# read in the data
data = np.loadtxt("infile3.txt",comments="#")

# slice them
x = data[:,0]
y1 = data[:,1]
```

```
y2 = data[:,2]

# make the plot
p1, = pl.plot(x,y1,"r",linewidth=2)
p2, = pl.plot(x,y2,"b",linewidth=2)

# set x and y ranges
pl.xlim(min(x),max(x))
pl.ylim(min(y1*1.05),max(y1*1.05))

# label the axes
pl.xlabel("X")
pl.ylabel("Y")

# legend
# loc is the location of the legend in a
# coordinate system from (0,0) to (1,1)
# frameon=False turns of the stupid box around the legend
pl.legend( (p1,p2), ("y1","y2"), loc=(0.7,0.85), frameon=False )

# comment the below line to get rid of screen display
pl.show()
# uncomment the below line to save as a pdf
#pl.savefig("simpleplot.pdf")
```

The result should look like this:



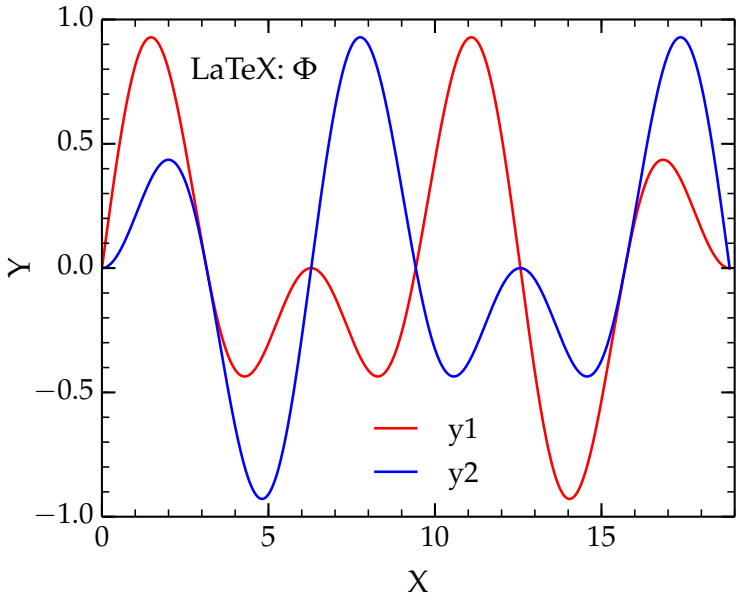

This kind of plot is great for a quick look at the data, but it is not quite the kind of plot you would find in a (good) journal article or thesis. The fonts are way too small, the tick marks are tiny, there are no minor ticks, the plot frame is too thin, and the fonts are sans serif (most journals prefer serif fonts, because of superior readability).

But, we are in luck: `matplotlib` is extremely flexible and allows one to change virtually anything one wants to change about a plot's appearance. It's even possible to use LaTeX math expressions in captions. For a more presentable plot, download http://www.tapir.caltech.edu/~cott/ay190/code/plot_defaults.py, then run

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as pl
from plot_defaults import *

# set up the figure and control white space
myfig = pl.figure(figsize=(10,8))
myfig.subplots_adjust(left=0.13)
myfig.subplots_adjust(bottom=0.14)
myfig.subplots_adjust(top=0.97)
myfig.subplots_adjust(right=0.975)

# read in the data
data = np.loadtxt("infile3.txt",comments="#")

# slice them
x = data[:,0]
y1 = data[:,1]
y2 = data[:,2]


# make the plot
p1, = pl.plot(x,y1,"r",linewidth=2.5)
p2, = pl.plot(x,y2,"b",linewidth=2.5)

# prepare x and y ranges
xmin = 0.0
xmax = 19.0
ymin = -1.0
ymax = 1.0

# set axis parameters
pl.axis([xmin,xmax,ymin,ymax])
# get axis object
ax = pl.gca()

# set locators of tick marks
xminorLocator = pl.MultipleLocator(1)
xmajorLocator = pl.MultipleLocator(5)
yminorLocator = pl.MultipleLocator(0.1)
ymajorLocator = pl.MultipleLocator(0.5)
ax.xaxis.set_major_locator(xmajorLocator)
ax.xaxis.set_minor_locator(xminorLocator)
ax.yaxis.set_minor_locator(yminorLocator)
ax.yaxis.set_major_locator(ymajorLocator)

# set the custom tick sizes we like
# these functions are defined in plot_defaults
set_ticklines(ax,2.0,0.75*2.0)
set_tick_sizes(ax, 13, 7)

# label the axes
pl.xlabel("X",labelpad=15)
pl.ylabel("Y",labelpad=-5)
```

```
# legend
# loc is the location of the legend in a
# coordinate system from (0,0) to (1,1)
# frameon=False turns of the stupid box around the legend
pl.legend( (p1,p2), ("y1","y2"),
           loc=(0.4,0.03), frameon=False )

pl.text(0.14,0.88,"LaTeX: $\\Phi$",fontsize=30,
        horizontalalignment="left",rotation="horizontal",
        transform=ax.transAxes)

# uncomment the below line to get screen display
# pl.show()
# comment the below line to save as a pdf
pl.savefig("simpleplot2.pdf")
```

The result should look like this:




Of course, there are many more things you can change and adjust. There is also functionaly to make different kinds of plots, e.g., 2D colormaps, histograms, log plots, lin-log plots, etc. Google & http://matplotlib.org are your friends!

## II.2 Version Control: `git`

A version control system (also called revision control system) is a piece of software that manages and tracks changes that occur to any file that is in a given project. In other words, you have the ability to take "snapshots" of your files during your current work, and you will be able to return to any of these snapshots *whenever* you wish. The software provides a full history of what you did, when you did it, and what files you did it to. Even better, version control software will let your collaborators have access to your project and track their changes in the same way it tracks your changes. You can revert any changes and go back to old versions or selectively pick changes that you like and discard others.

Version control is the modern way to collaborate on research projects. It also acts as backup and insurance against mistakes or hardware failures, because your files and the version control metadata are stored on a remote server as a *repository* and, in the case of `git`, each client has a copy of the full repository.

`git` is a version control system originally designed by Linus Torvalds in the mid 2000s. It is now very widely used both in software development and science. `git` is not an acronym and stands for nothing in particular. The official `git` website is `http://git-scm.com` and the Pro Git book is available at `http://git-scm.com/book`.

Most people that come into contact with `git` (and version control in general) will first find it increadibly annoying and useless, but will cherish it greatly in the long run. You will need to use `git` for Ay 190 – all homework must be pushed to your `git` repository!

### II.2.1 Setting up a `git` repository on GitHub.com

This is easy. Go to `http://github.com`, sign up, and upload your ssh key. If you don't know what an ssh key is, don't worry. Open a terminal, type

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cott/.ssh/id_rsa):
```

Hit enter.

```
Enter passphrase (empty for no passphrase):
```

ENTER a password that you can remember, hit enter again.

```
Enter same passphrase again:
```

Do what the computer tells you. Then upload the result, which resides in your home directory as `.ssh/id_rsa.pub` to GitHub.

Once you are all setup with GitHub, click on the "create new repo" icon (the one appearing on the top right of the webpage right next to your GitHub user name). Give the repo some meaningful name, e.g., `ay190`. Choose "Initialize this repository with a README," then click on "Create repository."

### II.2.2 Cloning your Repository

Presuming you have already installed the `git` software on your computer, open a terminal and enter the following commands to set up your `git` environment:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "your_email@whatever.com"
```

Obviously, you want to give your name and your e-mail address instead of the above placeholders. This is to identify your changes to the repository as changes made by you.

Before you can start working with your repository, you need to clone it first. In a terminal, enter

```
$ git clone git@github.com:[Your GitHub Username]/ay190
```

where you obviously replace `[Your GitHub Username]` with your GitHub username (you don't enter the $ sign – this is just to signify the shell prompt). When you hit enter on this command, you will be asked for the passphrase of your ssh key (which you hopefully remember!). The repository will then be cloned to your machine and you can use `cd` (change directory) to enter it:

```
$ cd ay190
```

### II.2.3 Adding Files/Changes and Pushing to the Repository

Let's say you have worked on the solution of worksheet 1. For this, you have created a directory inside your repository called `ws1` and in this repository you have a file called `ws1.py` that you want to add to the repository.

Assuming that you have a terminal open and are inside your repository, do the following to add and push your file:

```
$ git add ws1/ws1.py
$ git commit -m "added script for worksheet 1"
$ git push
```

With this, you have added the directory and the `Python` script to the repository, you have commited it locally and then pushed this local commit to the remote repository. The string after the `-m` in the `commit` line is a comment that will help you remember what the commit was about.

If you are unsure what has changed and what changes you need to add and then commit, you can use `git status` to get a list of changes. Be very careful about what files to add and commit – you don't want to commit temporary files or backup files (those ending with a $\sim$) to your repository. You can tell `git` to ignore such files by placing a file named `.gitignore` into the top-level directory of your repository. Your `.gitignore` file may look like this:

```
.DS_Store
*.[oa]
*~
.#*
*.aux
*.log
*.bbl
*.blg
*.out
```

```
*.toc
```

This will include all such files (many of which are temporary files created by LaTeX).

Let's say you have made another change to your file ws1.py. The procedure to add, commit and push it is the following (assuming you are in the top-level directory of your respository):

```
$ git add ws1/ws1.py
$ git commit -m "Changed script so that ... blah"
$ git push
```

The -m "...." part is the *commit message* that explains what your changes are about. You want to make this as informative as possible, because it will be stored as meta information with your commit in the history of the repository.

### II.2.4 Pulling in Remote Changes and Dealing with Conflicts

If someone else has pushed a change to your repository or you have pushed a change on a different computer and now want to update your local copy of the repository with the changes on the server ("remote changes"), then you need to say

```
$ git pull
```

Depending on the state of your local repository, there could be multiple results of this:

- If your local repository is up-to-date, then the output will just be

  ```
  Already up-to-date.
  ```

- If you *do not* have local changes that git thinks could be affected by the pull, the output will look similar to this:

  ```
  remote: Counting objects: 5, done.
  remote: Total 3 (delta 0), reused 3 (delta 0)
  Unpacking objects: 100% (3/3), done.
  From github.com:hypercott/ay190test
     8181af0..23ae277  master     -> origin/master
  Updating 8181af0..23ae277
  Fast-forward
   README.md | 1 +
   1 file changed, 1 insertion(+)
  ```

  All is good!

- If you *do* have local uncommitted changes that git thinks could be affected by the changes you are trying to pull in from remote, then git will say

  ```
  error: Your local changes to the following files would be overwritten
     by merge:
       [your file]
  Please, commit your changes or stash them before you can merge.
  Aborting
  ```

  So, you need to git add and git commit your file, but you can't push before trying a pull again.

- You have locally committed change that does not conflict with a change you are trying to pull in. In this case `git` is able *merge* your local changes with the remote changes automatically. You will be asked to enter a commit message for the merge commit. Here is an example:

```
Merge branch 'master' of github.com:hypercott/ay190test

# Please enter a commit message to explain why this merge is
# necessary, especially if it merges an updated upstream into a topic
# branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

  Just leave the text alone and exit your editor saving the file. `Git` will report the successful merge

```
Merge made by the 'recursive' strategy.
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

  You can avoid the extra merge commit by using `git pull --rebase`. However conflict resolution is harder. Until you feel more comfortable with git, the default behavior is safer.

- You have a locally committed change that *conflicts* with a change you are trying to pull in. This is the worst case scenario and `git` will bark at you. Here is an example:

```
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:hypercott/ay190test
   23ae277..b13d53f  master     -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

  *Good Luck!* Now you must resolve the conflict, commit your changes, then push them. Let's look at the example file in question:

```
$ cat README.md
ay190test
=========
<<<<<<< HEAD
Test345
=======
Test3
>>>>>>> b13d53f9cf6a8fe5b58e3c9c103f1dab84026161
```

  `git` has conveniently inserted information on what the conflict is. The stuff right beneath `<<<<< HEAD` is what is currently in the remote repository and the stuff just below the `=======` is what we have locally. Let's say you know your local change is correct. Then you edit the file and remove the remote stuff and all conflict stuff around it:

```
$ cat README.md
ay190test
=========
Test3
```

Next you `git add` and `git commit` the file, then you execute `git push` to update the remote with your local change. This resolves the conflict and all is good!

### II.2.5 Reverting Local Changes

If you have made a change to a file (let's say it is called `README.md`) that you don't like and you just want to go back to the version stored in the repository, then do this

```
$ git checkout README.md
```

This gets you the vanilla version stored in the repository and wipes out all your changes since the last commit of this file. So don't do this unless you are sure you really want to get rid of your changes!

### II.2.6 Checking out Old Versions

The obvious great power of version control is that you can go back in time and look at old versions of your repository.

```
$ git log
```

outputs the entire history of your repository. Here is an example:

```
commit 84c5f06ad72b2cd790aee5a332d7557d131433d8
Author: Christian D. Ott <cott@tapir.caltech.edu>
Date:   Sat Dec 28 08:57:10 2013 -0800

    test2

commit b13d53f9cf6a8fe5b58e3c9c103f1dab84026161
Author: Christian D. Ott <cott@tapir.caltech.edu>
Date:   Sat Dec 28 08:56:57 2013 -0800

    git test

commit 23ae2778680293008a92dc3aecbff61a45160baa
Author: Christian D. Ott <cott@tapir.caltech.edu>
Date:   Sat Dec 28 08:54:08 2013 -0800

    test commit

commit 8181af094b3314f83ddb1c84bca204647d22c5d9
Author: hypercott <christian.d.ott@gmail.com>
Date:   Fri Dec 27 22:08:33 2013 -0800

    Initial commit
```

Each commit is identified by a unique *hash tag*. To go back to a particular commit (i.e. version), you simply say

```
git checkout [hash tag]
```

After you are done looking at the old version,

```
git checkout master
```

brings you back to the most recent version. If you want to save some stuff from the old version to recreate some change that was later overwritten, copy the file you want to save somewhere else before going back to `master`. Then selectively (and carefully) modify the current working copy on the basis of the saved file, but be sure not to overwrite anybody else's changes that you intend to keep. Note that this is the poor-human's way of going back to previous versions. There are fancier and saver ways described, for example, here: http://stackoverflow.com/questions/4114095/revert-to-previous-git-commit. And, as always, google is your friend!

## II.3 Typesetting with LaTeX

LaTeX is an extremely powerful and flexible typesetting system that is available as open source. It is free software. These class notes are typeset in LaTeX and so is virtually every journal publication in mathematics, physics, and astronomy. It is the professional way to prepare *any* academic document in these areas. This includes applications for graduate school, fellowship and research proposals and theses of any kind, and applications for postdoc jobs and faculty jobs!

The authorative resource for learning and using LaTeX is now http://en.wikibooks.org/wiki/LaTeX. But there are many other online (and print) resources available. As always, google is your friend!

We assume that you have a working LaTeX installation on your system. In the following, we provide a very brief introduction to LaTeX by example. It is by no means rigorous nor complete, but it will familiarize you with the basics of using LaTeX in your homework assignments.

### II.3.1 A Basic LaTeX Document and How to Get a PDF

Below is a template LaTeX file that you can work with. LaTeX files are standard ASCII text files that are *compiled* into their final format, which nowadays usually is PDF. The below LaTeX example is available for download from http://www.tapir.caltech.edu/~cott/ay190/code/latex_template.tex. You will also need to download a figure that is imported: http://www.tapir.caltech.edu/~cott/ay190/code/simpleplot2.pdf.

You compile the LaTeX code to PDF in a terminal by executing the following commands

```
$ pdflatex latex_template.tex
$ pdflatex latex_template.tex
$ pdflatex latex_template.tex
```

That's right – you run `pdflatex` three times. `pdflatex` is the LaTeX compiler of choice these days. You need to run it three times so that LaTeX has a chance to pick up on internal cross references to sections and figures. It does this via auxiliary files that are created only when LaTeX is run.

Here is the LaTeX code:

```
\documentclass[11pt,letterpaper]{article}

% Load some basic packages that are useful to have
% and that should be part of any LaTeX installation.
%
% be able to include figures
\usepackage{graphicx}
% get nice colors
\usepackage{xcolor}

% change default font to Palatino (looks nicer!)
\usepackage[latin1]{inputenc}
\usepackage{mathpazo}
\usepackage[T1]{fontenc}
% load some useful math symbols/fonts
\usepackage{latexsym,amsfonts,amsmath,amssymb}

% convenience package to easily set margins
```

```
\usepackage[top=1in, bottom=1in, left=1in, right=1in]{geometry}

% control some spacings
%
% spacing after a paragraph
\setlength{\parskip}{.15cm}
% indentation at the top of a new paragraph
\setlength{\parindent}{0.0cm}


\begin{document}

\begin{center}
\Large
Ay190 -- Worksheet XX\\
Your Name\\
Date: \today
\end{center}

\section{This is a Section}

See figure~\ref{fig:simpleplot2} for an example!

{\bf This is text in bold font.}

\emph{This is text in italic font.}

{\it This also produces italic font.}

{\color{red} This is text in red!}

\subsection{This is a Subsection}

\subsubsection{This is a Subsubsection}

\begin{figure}[bth]
\centering
\includegraphics[width=0.5\textwidth]{simpleplot2.pdf}
\caption{This is a figure.}
\label{fig:simpleplot2}
\end{figure}

\end{document}
Anything that comes after \end{document} is completely
ignored by LaTeX.
```

There is a lot to say about this example. The first thing to notice is that in LaTeX comments are indicated by a % sign. There are a number of commands at the top of the file before `\begin{document}`. This part of the file is called the *preamble* and all sorts of definitions go there. The part enclosed by

```
\begin{document}
[...]
\end{document}
```

is the stuff that actually gets compiled into a PDF. Anything that comes after `\end{document}` is just ignored. Text in

```
\begin{center}
[...]
\end{center}
```

is centered on the page. The region between a `\begin{...}` – `\end{...}` combination is called an *environment*. Curly brackets ({}) also create a local environment in which one can mess with the text (make it bold, italic, or change its color etc.) without affecting the rest of the document.

The end result looks like this:

Ay190 – Worksheet XX
Your Name
Date: December 29, 2013

## 1 This is a Section

See figure 1 for an example!
**This is text in bold font.**
*This is text in italic font.*
*This also produces italic font.*
This is text in red!

### 1.1 This is a Subsection

#### 1.1.1 This is a Subsubsection

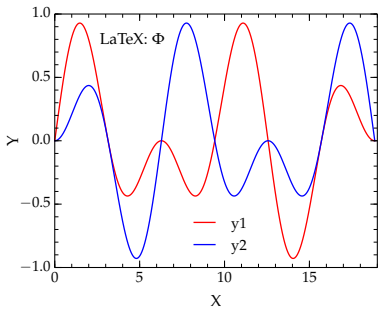



Figure 1: This is a figure.

1

LaTeX really is about learning by doing. If you want to do something that you don't know how to do: ask google! There is a virtually infinite number of ways in which you can modify your LaTeX document and there are hundreds of LaTeX packages that can help you with whatever need you may have.

### II.3.2 Math and Equations

In LaTeX, math expressions equations can be typeset in multiple ways. An extensive discussion of math in LaTeX is provided in http://en.wikibooks.org/wiki/LaTeX/Mathematics. Here we just give some examples. For equations offset from text, use

```
\begin{equation}
\int_\alpha^\beta \frac{\exp(x^2)}{x+1} dx = 5 \times 10^{15}\,\mathrm{
   g\,cm}^{-3}
\end{equation}
```

which results in

$$\int_\alpha^\beta \frac{\exp(x^2)}{x+1} dx = 5 \times 10^{15}\,\mathrm{g\,cm}^{-3} \qquad (1)$$

You can also include simple math expressions as part of the normal text. Math is then delimited by $ signs.

```
Newton came up with the idea that $\vec{F} = m \vec{a}$.
```

This results in

Newton came up with the idea that $\vec{F} = m\vec{a}$.

There is, of course, lots more that can be done. LaTeX also provides a huge amount of special symbols and styles one can use. A list of available symbols is provided by wikipedia: http://ia.wikipedia.org/wiki/Wikipedia:LaTeX_symbols. Important for astronomy is `\odot`, which gives the "sun symbol" $\odot$, which is frequently needed for $M_\odot$, $R_\odot$, $L_\odot$ etc.

Here is another examples of some LaTeX math expressions:

```
\begin{equation}
\begin{aligned}
%
\sqrt{x^2 + 4}   & = 12\\
%
\sum_{i=1}^{100} x_{i} & = N\\
%
L_{\nu_e} + L_{\bar{\nu}_e} &= L_{\nu,\mathrm{total}}\\
%
Q_i &= \frac{b_i - a_i}{6} \left[f(a_i)
+ 4 f\left(\frac{a_i+b_i}{2}\right)
+ f(b_i) \right] + \mathcal{O}([b_i-a_i]^5)
\end{aligned}
\end{equation}
```

This arguably more complicate example produces the following LaTeX output:

$$\begin{aligned}
\sqrt{x^2+4} &= 12 \\
\sum_{i=1}^{100} x_i &= N \\
L_{\nu_e} + L_{\bar{\nu}_e} &= L_{\nu,\text{total}} \\
Q_i &= \frac{b_i - a_i}{6}\left[f(a_i) + 4f\left(\frac{a_i+b_i}{2}\right) + f(b_i)\right] + \mathcal{O}([b_i-a_i]^5)
\end{aligned} \tag{2}$$

### II.3.3 Internal Cross Referencing

It is often necessary to refer to other sections, to tables, and figures within the same document. Back in the old days, sections, tables, and figures were manually labeled/given numbers and whenever one wanted to change their ordering or add a new one, one had to go back and renumber everything. This usually led to a mess. Hence, LaTeX uses automatic numbering and cross referencing.

For example, if we introduce

```
\label{sec:latexmath}
```

right at the top of the previous section in these notes, then we can make a reference to this section by

```
We are refering the reader to Section~\ref{sec:latexmath} for this.
```

This produces:

We are refering the reader to Section II.3.2 for this.

Note that the tilde (~) between `Section` and `\ref{sec:latexmath}` ensures that LaTeX will never try to break the line between `Section` and its number.

Cross referencing works for sections, tables, and figures.

### II.3.4 Basic Tables

Here is example code to produce a table listing some numerical data.

```
\begin{table}
\centering
\caption{This is a basic table listing some supernovae.}
\begin{tabular}{r|r|r|r}
% table header
Supernova & Type & Host Galaxy & Distance\\
\hline % draw horizontal line
\hline % draw another horizontal line
% table data
SN 2007gr & Ic & NGC 1058& $10.55\pm1.95$\\
SN 2008ax & IIb& NGC 4490& $9.64\pm1.38$\\
SN 2008bk & IIP& NGC 7793& $3.53\pm0.41$\\
SN 2011dh & IIb& M51     & $8.40\pm0.70$\\
```

```
\hline % draw another horizontal line
\end{tabular}
\label{tab:supernovae} % label for cross referencing
\end{table}
```

This results in

Table 1: This is a basic table listing some supernovae.

| Supernova | Type | Host Galaxy | Distance |
|---|---|---|---|
| SN 2007gr | Ic | NGC 1058 | $10.55 \pm 1.95$ |
| SN 2008ax | IIb | NGC 4490 | $9.64 \pm 1.38$ |
| SN 2008bk | IIP | NGC 7793 | $3.53 \pm 0.41$ |
| SN 2011dh | IIb | M51 | $8.40 \pm 0.70$ |

A few things to note:

- Each line in the tabular environment must be closed with `\\` – this tells LaTeX that the line ends there.
- The `{r|r|r|r}` at the top of the table sets the number of columns and their horizontal alignment: `r, c, l` for right, center, and left. The vertical bars `|` indicate where LaTeX should put vertical lines to separate columns.
- `\centering` at the top of the `table` environment makes everything horizontally centered.
- The caption may go above or below the table. Include it if you want the table number to appear.

Much more information about tables in LaTeX can be found here: http://en.wikibooks.org/wiki/LaTeX/Tables.

### II.3.5 Including Graphics/Figures

In the first LaTeX example in §II.3.1, we already included a figure. Here is the code for this again:

```
\begin{figure}[bth]
\centering
\includegraphics[width=0.5\textwidth]{simpleplot2.pdf}
\caption{This is a figure.}
\label{fig:simpleplot2}
\end{figure}
```

We already discussed the meaning/effect of `\centering` and `\label{...}`. A few things to note are:

- The `[bth]` after the start of the `figure` environment advise LaTeX where to put the figure. Figures are so-called *float* objects that LaTeX is free to distribute across the page and even put on a different page than the current one. `b` means "bottom", `t` means "top", `h` means "here." You can try to force a particular placement by using an exclamation mark (!). For example, `[t!]` will indicate to LaTeX that you really want this figure to be placed at the top of a page.

- `\includegraphics` is the command that actually includes the figure file. The file must be in PDF or PNG format. There are a variety of tags that can be placed in `[]` before the figure file name, the most important is `width`, which we have set to be half of the width of the text (`\just returns the text width in the document.`
- You can refer to the figure from the text by using cross referencing as discussed in §II.3.3.

More about figures can be found here: http://en.wikibooks.org/wiki/LaTeX/Floats,_Figures_and_Captions.

### II.3.6 BibTeX Bibliography

BibTeX is a bibliography system for LaTeX that helps you manage and organize your bibliography. This is increadibly useful, because you just have to worry about assembling a bibliography data base and making reference to individual entries (papers, books, etc.) via `\cite` commands. BibTeX and LaTeX handle everything else!

In the following, we present one example for the use of BibTeX. A much more extensive and detailed introduction to BibTeX is available at http://en.wikibooks.org/wiki/LaTeX/Bibliography_Management.

The following BibTeX database and LaTeX code can be downloaded from here:
http://www.tapir.caltech.edu/~cott/ay190/code/bibtex_example.bib
http://www.tapir.caltech.edu/~cott/ay190/code/bibtex_example.tex

The bibliography database is just a text file:

```
@ARTICLE{oconnor:13,
   author = {{O'Connor}, E. and {Ott}, C.~D.},
    title = "{The Progenitor Dependence of the Pre-explosion Neutrino
   Emission in Core-collapse Supernovae}",
  journal = {\apj},
archivePrefix = "arXiv",
   eprint = {1207.1100},
 primaryClass = "astro-ph.HE",
 keywords = {equation of state, hydrodynamics, neutrinos, stars:
   evolution, stars: neutron, supernovae: general},
     year = 2013,
   volume = 762,
      eid = {126},
    pages = {126},
      doi = {10.1088/0004-637X/762/2/126},
   adsurl = {http://adsabs.harvard.edu/abs/2013ApJ...762..126O},
}
```

One can obtain such entries for any physics/astro paper from ADS: http://adsabs.harvard.edu/abstract_service.html.

The first thing coming after `ARTICLE{` is the *bibliography key* that you will use in `\cite{key}` commands to refer to the paper in your document. We recommend that you use

```
[first author]:[2-digit year]
```

for the bibliography key. If there are multiple papers by the same first author, append letters `a`, `b`, `c`, ... after the year. We have found that this works well and keys are easy to remember this way.

Note that ADS `BibTeX` database entries use macros for most journal names (see `journal = {\apj}`). Your LaTeX file must include a definition of these macros for things to work. Here is an example:

```
\documentclass[11pt,letterpaper]{article}

% Load some basic packages that are useful to have
% and that should be part of any LaTeX installation.
%
% be able to include figures
\usepackage{graphicx}
% get nice colors
\usepackage{xcolor}

% change default font to Palatino (looks nicer!)
\usepackage[latin1]{inputenc}
\usepackage{mathpazo}
\usepackage[T1]{fontenc}
% load some useful math symbols/fonts
\usepackage{latexsym,amsfonts,amsmath,amssymb}

% convenience package to easily set margins
\usepackage[top=1in, bottom=1in, left=1in, right=1in]{geometry}

% control some spacings
%
% spacing after a paragraph
\setlength{\parskip}{.15cm}
% indentation at the top of a new paragraph
\setlength{\parindent}{0.0cm}

% some definitions for journal names
\def\aj{Astron. J.}
\def\apj{Astrophys. J.}
\def\apjl{Astrophys. J. Lett.}
\def\apjs{Astrophys. J. Supp. Ser. }
\def\aa{Astron. Astrophys. }
\def\aap{Astron. Astrophys. }
\def\araa{Ann.\ Rev. Astron. Astroph. }
\def\physrep{Phys. Rep. }
\def\mnras{Mon. Not. Roy. Astron. Soc. }
\def\prl{Phys. Rev. Lett.}
\def\prd{Phys. Rev. D.}
\def\apss{Astrophys. Space Sci.}
\def\cqg{Class. Quantum Grav.}



\begin{document}
```

```
\begin{center}
\Large
Ay190 -- Worksheet XX\\
Your Name\\
Date: \today
\end{center}

Today, we are making reference to O'Connor \& Ott's paper on the
progenitor dependence of the pre-explosion neutrino emission in
core-collapse supernovae~\cite{oconnor:13}.


\bibliographystyle{unsrt}
\bibliography{bibtex_example}

\end{document}
```

Things to note:

- The preamble contains macro definitions that replace ADS shortcuts for journal names with the actual journal names.
- `\cite{oconnor:13}` is used in the text to refer to the paper in question.
- Two important commands are placed right before the end of the document:

  ```
  \bibliographystyle{unsrt}
  \bibliography{bibtex_example}
  ```

  The first one sets the citation style. In this case, `unsrt` will build the references / bibliography section in a chronological sense, putting papers cited earlier ahead of papers cited later. This can be changed. See http://en.wikibooks.org/wiki/LaTeX/Bibliography_Management#Bibliography_styles and many journals have their own styles that come in `.bst` files.

Now, finally, here is how to compile the document:

```
$ pdflatex bibtex_example.tex
$ bibtex bibtex_example
$ pdflatex bibtex_example.tex
$ pdflatex bibtex_example.tex
```

As in the pure LaTeX case, we need to compile multiple times to get all the cross references right. BibTeX is run after the first LaTeX compile. The result is displayed on the next page.

Ay190 – Worksheet XX
Your Name
Date: January 13, 2014

Today, we are making reference to O'Connor & Ott's paper on the progenitor dependence of the pre-explosion neutrino emission in core-collapse supernovae [1].

**References**

[1] E. O'Connor and C. D. Ott. The Progenitor Dependence of the Pre-explosion Neutrino Emission in Core-collapse Supernovae. *Astrophys. J.*, 762:126, 2013.

1

## II.4 Basic Use of Makefiles

The GNU Make system works via makefiles that define wanted products, dependencies and rules for how to make the products provided the dependencies are met. This sounds abstract, but is increadibly useful.

For example, would it not be nice to compile the LaTeX/BibTeX example given in the previous section §II.3.6 simply by typing

```
$ make
```

? It sure would! So let us analyze the situation. We want to make `bibtex_example.pdf` and this depends on `bibtex_example.tex` (the LaTeX file) and on `bibtex_example.bib` (the `BibTex database`). We can write the following file (and call it Makefile):

```
bibtex_example.pdf: bibtex_example.tex bibtex_example.bib
  pdflatex bibtex_example.tex
  bibtex bibtex_example
  pdflatex bibtex_example.tex
  pdflatex bibtex_example.tex
```

So the format is the following:

```
product: dependency1 dependency2 ...
<TAB>rule1
<TAB>rule2
<TAB>rule3
<TAB>...
```

where ⟨TAB⟩ indicates hitting the tabulator button on your keyboard. Obviously, GNU Make can be used for many other applications, not just for LaTeX and BibTeX. It also has many more features. See http://www.gnu.org/software/make/manual/make.html.

# Chapter III

# Fundamentals: The Basics of Numerical Computation

## III.1 Computers & Numbers

### III.1.1 Floating Point Numbers

Computers have finite memory and compute power. Hence, real numbers must be represented as floating point (FP) numbers with finite precision. Is this a problem? Yes, because small approximation errors can in principle (and in unlucky circumstances) be amplified and grow out of bound.

$$\pm\, 0.a_1a_2a_3a_4 \ldots a_m \times 10^c \ , \tag{III.1.1}$$

where $\pm 0.a_1a_2a_3a_4 \ldots a_m$ is the mantissa, the integer $c$ is the exponent, and the integer 10 is the base of the FP number. The digits $a_i$ are the *significant digits* and the number of significant digits is the FP precision $m$.

In general, a real number $x$ can be expressed as a FP number $\overline{x}$ plus an absolute FP error (*round off* error) $e(x)$:

$$\underbrace{x}_{\in \mathbb{R}} \;=\; \underbrace{\overline{x}}_{\text{FP number}} \;+\; \underbrace{e(x)}_{\text{absolute FP error}} . \tag{III.1.2}$$

Converting a real number $x$ to a floating point number $\overline{x}$ is accomplished either by truncation or (better) rounding.

Example: $m = 4$

$$\begin{aligned} x &= \pi = 3.141592653589\ldots \ . \\ \overline{x} &= 3.141 + e(x) \ ; \ \ e(x) = \pi - \overline{x} \ . \end{aligned}$$

In computer terminology, one frequently speaks of *single precision* and *double precision* numbers. Single precision generally corresponds to 32-bit FP numbers with $m \simeq 7$ and double precision generally stands for 64-bit FP numbers with $m \simeq 14 - 16$. For reasons that we will see later, FP data types in computers actually have a mantissa of size $\geq 2m$, but their accuracy is still limited to $m$ digits.

In addition, there are limits to the maximum and minimum number that can be stored in a FP variable. This is controlled by the maximum size of the exponent. For standard base-10 numbers,

the maximum (minimum) number is of $\mathcal{O}(10^{38})$ ($\mathcal{O}(10^{-37})$) and $\mathcal{O}(10^{308})$ ($\mathcal{O}(10^{-307})$) for single and double precision, respectively.

### III.1.2 Errors

We define two kinds of errors

$$\begin{array}{rrll} \text{absolute error:} & E_a(x) & = e(x) & = x - \overline{x}\ , \\ \text{relative error:} & E_r(x) & = \frac{x-\overline{x}}{x}\ . & \end{array} \tag{III.1.3}$$

#### III.1.2.1 Errors & Floating Point Arithmetic

The basic arithmetic operations are $x + y$, $x - y$, $x \cdot y$, and $\frac{x}{y}$. We must investigate how errors propagate in these operations.

$x + y$:

$$x + y = \overline{x} + \overline{y} + e(x) + e(y)\,, \tag{III.1.4}$$

with $x = \overline{x} + e(x)$ and $y = \overline{y} + e(y)$. $e(x)$ and $e(y)$ are the absolute round-off error for $x$ and $y$, respectively.

Note that:

(a) $\overline{x + y} \neq \overline{x} + \overline{y}$
Example: $m = 3$

$$\begin{array}{rcl} x = 1.313 & \rightarrow & \overline{x} = 1.31 \\ y = 2.147 & \rightarrow & \overline{y} = 2.14 \end{array} \rightarrow \overline{x} + \overline{y} = 3.45\ ,$$

but

$$x + y = 3.460 \rightarrow \overline{x + y} = 3.46\ .$$

(b) $x + y$ could be larger (or smaller) than the largest (smallest) number that can be represented. This leads to an *overflow* or *underflow*.

(c) Addition of FP numbers is not associative. So order matters:

$$\overline{\overline{(\overline{x} + \overline{y})} + \overline{z}} \neq \overline{\overline{x} + \overline{(\overline{y} + \overline{z})}}\,. \tag{III.1.5}$$

**Exercise III.1.1 (FP addition is not associative)**
Show by contradiction that the assumption that FP is associative is false.

$x - y$:

$$x - y = \overline{x} - \overline{y} + e(x) - e(y)\,, \tag{III.1.6}$$

which can be problematic when $x$ and $y$ are nearly equal. Example for $m = 3$:

$$\begin{aligned} &x = 42.102\,,\ y = 42.043\,,\\ &x - y = 0.059\,,\\ &\overline{x} - \overline{y} = 42.1 - 42.0 = 0.1\ . \end{aligned}$$

$x \cdot y$:

$$x \cdot y = \overline{x} \cdot \overline{y} + \overline{y} \cdot e(x) + \overline{x} \cdot e(y) + \underbrace{e(x) \cdot e(y)}_{\text{usually small}} \tag{III.1.7}$$

$\overline{x} \cdot \overline{y}$ is at most $2m$ digits long. This is why computers use $2m$ digits for computations on $m$-digit numbers. The result is then rounded to $m$ digits.

$\frac{x}{y}$:

$$\frac{x}{y} = \frac{\overline{x} + e(x)}{\overline{y} + e(y)} \overset{\text{expansion}}{=} \frac{\overline{x}}{\overline{y}} + \frac{e(x)}{\overline{y}} - \frac{\overline{x} e(y)}{\overline{y}^2} + \mathcal{O}(e(x) \cdot e(y))\,. \tag{III.1.8}$$

The expansion shows that the error terms will grow immensely as $y \to 0$. Dividing by small numbers is a thing to avoid.

### III.1.3 Stability of a Calculation

A numerical calculation is unstable if small errors made at one stage of the process are magnified in subsequent steps and seriously degrade the accuracy of the overall solution.

A very simple example for $m = 3$:

$$\begin{aligned} &(1)\quad \overline{y} = \overline{\sqrt{x}}\,,\\ &(2)\quad \overline{y} = \overline{\sqrt{\overline{\overline{x} + 1} - 1}}\,. \end{aligned}$$

If we were not dealing with FP numbers, the results of (1) and (2) would be identical.

Let us now choose $\overline{x} = x = 0.02412 = 2.41 \times 10^{-2}$. Then

$$(1)\quad \overline{\sqrt{\overline{x}}} = 0.155\,,$$

but

$$\begin{aligned} (2)\quad &\overline{\overline{x} + 1} = 1.02\,,\\ &\overline{\overline{\overline{x} + 1} - 1} = 0.02\,,\\ &\overline{\sqrt{0.02}} = 0.141\ . \end{aligned}$$

The relative error between (1) and (2) is 9% and is due purely to additional unnecessary operations that amplified the FP error!

**Exercise III.1.2 (An unstable Calculation)**

Consider the following sequence and recurrence relation:

$$x_0 = 1\,,\ x_1 = \frac{1}{3}\,,\ x_{n+1} = \frac{13}{3}x_n - \frac{4}{3}x_{n-1}\,,$$

which is equivalent to

$$x_n = \left(\frac{1}{3}\right)^n\,.$$

Implement the recurrence relation for $n = 0, ..., 15$ using *single precision* FP numbers and compute absolute and relative error with respect to the closed-form expression. How big are relative and absolute error at $n = 15$?

Now do the same for $x_1 = 4$, and compare it to $x_n = 4^n$. Go to $n = 20$. Why is this calculation stable? Should absolute or relative error be used to measure accuracy and stability?

## III.2 Finite Differences

### III.2.1 Forward, Backward, and Central Finite Difference Approximations

We will often need to numerically find the derivative of a function $f(x)$,

$$f'(x) = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x}(x), \quad \text{where} \quad \Delta f = f\left(x + \frac{\Delta x}{2}\right) - f\left(x - \frac{\Delta x}{2}\right) . \tag{III.2.1}$$

Now let us assume $\Delta x/2 = h$. We can use Taylor's theorem to find an approximation for $f'$ at $x_0$, namely:

$$\begin{aligned} f(x_0 + h) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} h^n \\ &= f(x_0) + h f'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + \mathcal{O}(h^4) . \end{aligned} \tag{III.2.2}$$

By truncating at the $h^2$ term, we get

$$f(x_0 + h) = f(x_0) + h f'(x_0) + \mathcal{O}(h^2) , \tag{III.2.3}$$

and can solve for $f'(x_0)$:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h) . \tag{III.2.4}$$

This is called the *forward difference* estimate for $f'$. In other words, the slope of $f(x)$ at $x_0$ is approximated by the straight line through $(x_0, f(x_0))$ and $(x_0 + h, f(x_0 + h))$.

The expression $\mathcal{O}(h)$ on the far right of Eq. (III.2.4) indicates that the error of our estimate decreases linearly ($\propto h$) with step size $h$. Hence, we call this approximation *first order* (in $h$).

Now let us consider $f(x_0 - h)$,

$$f(x_0 - h) = f(x_0) - h f'(x_0) + \frac{h^2}{2} f''(x_0) - \frac{h^3}{6} f'''(x_0) + \mathcal{O}(h_0^4) , \tag{III.2.5}$$

which gets us the first-order *backward difference* estimate for $f'(x)$ at $x_0$:

$$f'(x_0) = \frac{f(x_0) - f(x_0 - h)}{h} + \mathcal{O}(h) . \tag{III.2.6}$$

Now, subtracting Eq. (III.2.5) from Eq. (III.2.2), we get

$$f(x_0 + h) - f(x_0 - h) = 2h f'(x_0) + \frac{h^3}{3} f'''(x_0) + \mathcal{O}(h^4) , \tag{III.2.7}$$

and, because the $h^2$ term has vanished,

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + \mathcal{O}(h^2) . \tag{III.2.8}$$

This is the *central difference* estimate for $f'(x)$ at $x_0$.

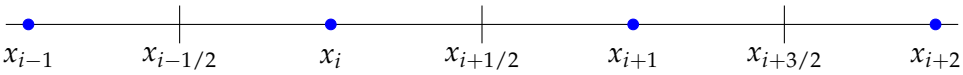



Figure III.2.1: Basic logical grid setup in 1D. Grid cell centers are marked with a filled circle. Cell interfaces are marked by vertical lines.

**Exercise III.2.1 (Finite-Difference Estiate of the Second Derivative)**
Use the methods demonstrated in this section to derive a second-order (central) expression for the second derivative of a function $f(x)$ at $x_0$.

### III.2.2 Finite Differences on evenly and unevenly spaced Grids

So far, we have simply assumed that the step size $h$ is constant between discrete nodes $x_i$ at which we know the values of function $f(x)$.

Figure III.2.1 shows a typical basic grid setup in a one-dimensional problem. A computational cell has cell center $x_i$ and is demarkated by $x_{i-1/2}$ on the left and $x_{i+1/2}$ on the right. If the grid is evenly spaced (on speaks of an *equidistant grid*) then the distance between cell centers $\Delta x = x_{i+1} - x_i$ is constant throughout the grid. In this case, we may, for example, express the centered derivative of a function $f(x)$ at $x_i$ by

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x} + \mathcal{O}(\Delta x^2)\,. \tag{III.2.9}$$

Equidistant grids are the conceptually simplest way of discretizing a problem. They are, however, in many cases by far not the most efficient way, since many problems need more resolution in some parts of their domain than in others. A great example is the modeling of stellar collapse of an iron core to a neutron star. To resolve the steep density gradients near the neutron star, a resolution of order 100 m is required within $\sim$ 30 km of the origin, while a cell size of order 10 km is sufficient at radii above $\sim$1000 km. One efficient way of dealing with these different resolution requirements is to increase the radial extent of computational cells with increasing radius.

Taking numerical derivatives on such unevenly spaced (or *non-equidistant*) grids is more complicated, but still doable. Obviously, first-order estimates are trivial, since they only involve two points. In the following, we derive a second-order estimate of $f'(x)$ for non-equidistant grids that reduces to the standard central difference estimate in the limit of equidistant grids.

We want to evaluate $f'(x)$ at $x = x_i$. We define $h_1 = x_i - x_{i-1}$ and $h_2 = x_{i+1} - x_i$. Then,

$$\begin{aligned} f(x_i + h_2) &= f(x_i) + h_2 f'(x_i) + \frac{h_2^2}{2} f''(x_i) + \mathcal{O}(h_2^3)\,, \\ f(x_i - h_1) &= f(x_i) - h_1 f'(x_i) + \frac{h_1^2}{2} f''(x_i) + \mathcal{O}(h_1^3)\,. \end{aligned} \tag{III.2.10}$$

This allows us to eliminate the $f''(x_i)$ term and to solve for $f'(x_i)$:

$$f'(x_i) = \frac{h_1}{h_2(h_1+h_2)} f(x_{i+1}) - \frac{h_1 - h_2}{h_2 h_1} f(x_i) - \frac{h_2}{h_1(h_1+h_2)} f(x_{i-1}) \,. \tag{III.2.11}$$

It is trivial to see that this reduces to the standard central difference estimate (Eq. III.2.8) if $h_1 = h_2$.

**Exercise III.2.2 (Second Derivative Estimate on non-equidistant Grids)**

Derive a second-order estimate for the second derivative of a function $f(x)$ at $x_i$ on a non-equidistant grid.

### III.2.3 Convergence

A numerical method to solve a problem with a true solution $y(x)$ is said to be convergent if the discrete solution $y(x;h)$ approaches the true solution for vanishing discretization step size $h$:

$$\lim_{h\to 0} y(x;h) = y(x) \,. \tag{III.2.12}$$

In other words, if the resolution is increased, the numerical result converges to the true result.

**Three important things**:

1. For a numerical scheme that is $n$-th order accurate, a decrease in step size by a factor $m$ leads to a decrease of the deviation from the true solution by a factor $m^n$. So, for $n = 2$, the error is reduced by a factor of 4 if the resolution is doubled, while for $n = 1$, the error goes down only by a factor of 2.

   We can express this more mathematically by defining the convergence factor

   $$Q = \frac{|y(x;h_2) - y(x)|}{|y(x;h_1) - y(x)|} = \left(\frac{h_2}{h_1}\right)^n \,. \tag{III.2.13}$$

   Here $h_1 > h_2$ are two different discretization step sizes. Since $h_2$ is smaller than $h_1$, the calculation with $h_2$ has higher resolution.

2. **Checking convergence of a numerical algorithm to known solutions is crucial for validation.** If an algorithm does not converge or converges at the wrong rate, a systematic bug in the implementation is likely!

3. **Self Convergence**. In many cases, the true solution to a problem is not known. In this case, we can use results obtained at 3 different discretization step sizes $h_1 > h_2 > h_3$.

   We define the self-convergence factor $Q_S$:

   $$Q_S = \frac{|y(x;h_3) - y(x;h_2)|}{|y(x;h_2) - y(x;h_1)|} \,, \tag{III.2.14}$$

   which, at convergence order $n$, must equal

$$Q_S = \frac{h_3^n - h_2^n}{h_2^n - h_1^n}\,. \tag{III.2.15}$$

**Exercise III.2.3 (Convergence of a Finite-Difference Estimate)**

Discretize

$$f(x) = x^3 - 5x^2 + x$$

on the interval $[-2, 6]$ and compute its first derivative with (i) forward differencing and (ii) central differencing. Demonstrate that (i) is first-order convergent and (ii) is second-order convergent by plotting the absolute error $f'(x; h_i) - f'(x)$ at resolutions $h_1$ and $h_2 = h_1/2$. At $h_2$ the absolute error should be reduced by the expected convergence factor.

## III.3 Interpolation

We are frequently confronted with the situation that we know the values of a function $f$ only at discrete locations $x_i$, but want to know its values at general points $x$.

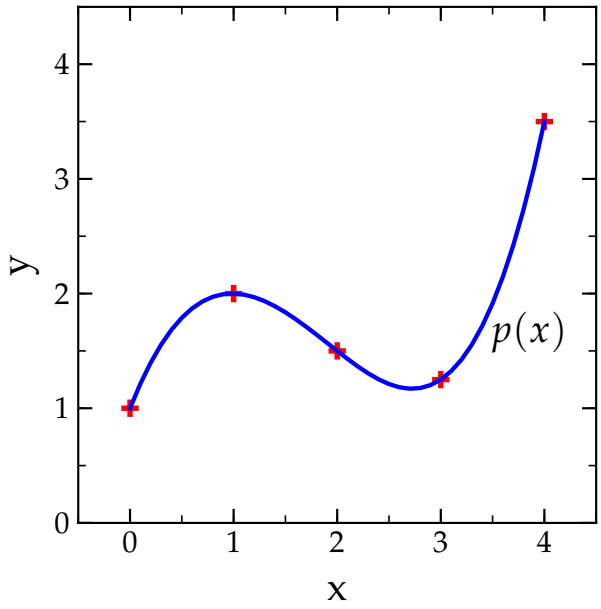



Figure III.3.1: The interpolation problem.

In order to solve this problem, we must look for an approximation $p(x)$ that uses the discrete information about $f(x)$ at $x_i$ to interpolate $f(x)$ between the $x_i$ with $p(x_i) = f(x_i)$. If $x$ is outside $[x_{\min}, x_{\max}]$, where $x_{\min} = \min\{x_i; \forall i\}$ and $x_{\max} = \max\{x_i; \forall i\}$, $p(x)$ extrapolates $f(x)$.

### III.3.1 Direct Polynomial Interpolation

Polynomial interpolation of $f(x)$ involves finding a polynomial $p(x)$ of *degree* $n$ that passes through $n+1$ points. Note that the literature will frequently talk about the *order* of a polynomial. We will stick to *degree* in order not to confuse polynomial order with order of approximation, i.e., with the exponent of the leading-order error term. For example a polynomial $p(x)$ of degree 1 is linear in $x$ and has a quadratic error term ("second order").

In general, our intergration polynomial will have the following form:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \,, \tag{III.3.1}$$

where the $a_i$ are $n+1$ real constants that can be determined by solving a set of $n+1$ linear equations:

$$\underbrace{\begin{pmatrix} 1 & x_0^1 & x_0^2 & \cdots & x_0^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n^1 & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{\text{Vandermonde Matrix}} \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ \vdots \\ \vdots \\ f(x_n) \end{pmatrix} \tag{III.3.2}$$

For large $n$ this obviously gets very complicated (we will talk later about how to solve systems of linear equations efficiently), but it is useful to consider the two simplest cases, linear ($n = 1$) and quadradtic ($n = 2$) interpolation.

#### III.3.1.1 Linear Interpolation

We obtain the linear approximation $p(x)$ for $f(x)$ in the interval $[x_i, x_{i+1}]$ by

$$p(x) = f(x_i) + \underbrace{\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}}_{\text{1st-order forward difference}} (x - x_i) + \mathcal{O}(h^2)\,, \tag{III.3.3}$$

where $h = x_{i+1} - x_i$. Linear interpolation is the most robust method for interpolation; the result can be differentiated once, but the derivative will be discontinuous at the locations $x_{i+1}$ and $x_i$.

#### III.3.1.2 Quadratic Interpolation

The quadratic approximation $p(x)$ for $f(x)$ in the interval $[x_i, x_{i+1}]$ is given by

$$\begin{aligned} p(x) = &\frac{(x - x_{i+1})(x - x_{i+2})}{(x_i - x_{i+1})(x_i - x_{i+2})} f(x_i) \\ &+ \frac{(x - x_i)(x - x_{i+2})}{(x_{i+1} - x_i)(x_{i+1} - x_{i+2})} f(x_{i+1}) \\ &+ \frac{(x - x_i)(x - x_{i+1})}{(x_{i+2} - x_i)(x_{i+2} - x_{i+1})} f(x_{i+2}) \\ &+ \mathcal{O}(h^3)\,, \end{aligned} \tag{III.3.4}$$

where $h = \max\{x_{i+2} - x_{i+1}, x_{i+1} - x_i\}$.

Note that the results will be sensitive to which three points are chosen, since there are two choices: $\{x_i, x_{i+1}, x_{i+2}\}$ or $\{x_{i-1}, x_i, x_{i+1}\}$ for interpolating $f(x)$ in $[x_i, x_{i+1}]$.

$p(x)$ is twice differentiable. Its first derivative will be continuous, but $p''(x)$ will have finite-size steps.

### III.3.2 Lagrange Interpolation

So far, we have looked at linear and quadratic polynomial interpolation. Sometimes it will be necessary to use a higher-order method. *Lagrange Interpolation* provides a means of constructing general interpolating polynomials of degree $n$ using data at $n + 1$ points.

There are alternative methods to Lagrange Interpolation. These are, for example, *Newton's Divided Differences*, *Chebychev Polynomials*, *Aitken's method*, and *Taylor Polynomials*. We will not discuss these methods, but rather refer the reader to the literature listed in §I.1.

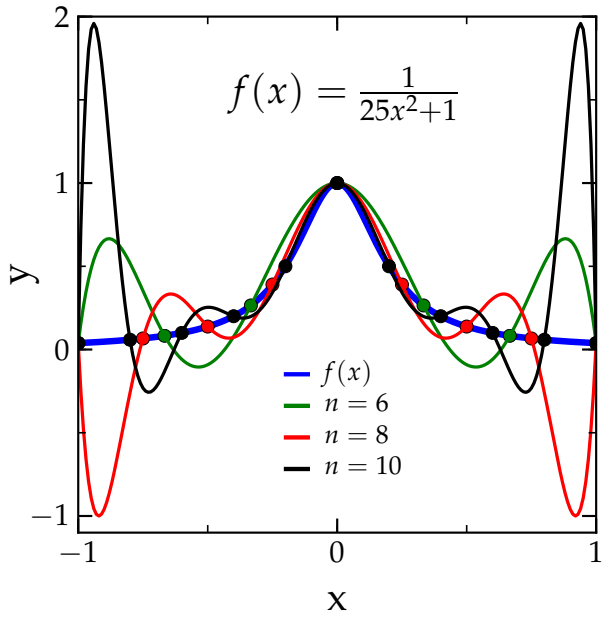



Figure III.3.2: Runge's phenomenon of non-convergence as exhibited by polynomials of degree 6, 8, and 10 for the continuous function $f(x) = (25x^2 + 1)^{-1}$.

For a moment, let us reconsider linear interpolation (Eq. [III.3.3]) and rewrite it slightly to

$$\begin{aligned} f(x) \approx p(x) &= \frac{x - x_{i+1}}{x_i - x_{i+1}} f(x_i) + \frac{x - x_i}{x_{i+1} - x_i} f(x_{i+1}) + \mathcal{O}(h^2)\,, \\ &= \sum_{j=i}^{i+1} f(x_j) L_{1j}(x) + \mathcal{O}(h^2)\,, \\ \text{where } L_{1j}(x) &= \left.\frac{x - x_k}{x_j - x_k}\right|_{k \neq j}\,. \end{aligned} \tag{III.3.5}$$

We can now easily generalize this to an $n$-th degree polynomial that passes through all $n + 1$ data points:

$$p(x) = \sum_{j=0}^{n} f(x_j) L_{nj}(x) + \mathcal{O}(h^{n+1})\,, \tag{III.3.6}$$

with

$$L_{nj}(x) = \prod_{k \neq j}^{n} \frac{x - x_k}{x_j - x_k}\,. \tag{III.3.7}$$

Note that this clearly satisfies the interpolating condition, $p(x_i) = f(x_i)$.

### III.3.3 Convergence of Interpolating Polynomials

If our data describe a continuous function $f(x)$ on an interval $[a, b]$, we would expect that, if we constuct interpolating polynomials $p_n(x)$ with increasing degree $n$, the interpolation error con-

verges to zero. Mathematically this is expressed by

$$\lim_{n\to\infty}\left[\max_{x\in[a,b]}|f(x)-p_n(x)|\right]=0\,. \tag{III.3.8}$$

**Unfortunately, it turns out that for most continuous functions, this is not the case and the error does not converge to 0**. This is called *Runge's Phenomenon*, an example of which is shown by Fig. III.3.2 for the well behaved continuous function $f(x)=\frac{1}{25x^2+1}$. While the interpolating polynomials $p_n(s)$ converge to $f(x)$ near the peak, they exhibit strong oscillations and diverge dramatically at the edges of the interval.

Runge's phenomenon teaches us that we need to be extremely careful with polynomial interpolation. A higher polynomial degree could leat to a greater error and unwanted oscillations!

High degree polynomial interpolation ($n\gtrsim 2$) should only be used if one is certain that the function to be interpolated can be approximated well globally by a single polynomial. If this is not the case, one must resort to other methods. One solution is to use *piecewise-polynomial* interpolation, which, in essence, uses many low-degree polynomials rather than a single global polynomial to approximate a function or interpolate a set of data. The simplest (and safest!) form of piecewise polynomial interpolation is piecewise linear interpolation – essentially connecting data points with straight lines.

**Exercise III.3.1 (Runge's Phenomenon)**

1. Implement a routine that generates Lagrangean interpolating polynomials of arbitrary degree $n$ based on $n+1$ data points. Then reproduce Fig. III.3.2 for $f(x)=\frac{1}{25x^2+1}$ in $[-1,1]$ with $n=6$, $n=8$, $n=10$, $n=12$.
2. Discretize the domain with $m=100$ equally spaced points and compute the error norm-2,
$$\mathrm{EN2}=\frac{1}{m}\sqrt{\sum_{i=1}^{m}\left(\frac{p(x)-f(x)}{f(x)}\right)^2},$$
for the 4 cases $n=\{6,8,10,12\}$.
3. Now introduce another discretization with $m_2$ = 50 and implement a routine that will interpolate $f(x)$ piecewise linearly between these $m_2$ data points. Now evaluate EN2 at the $m$ points used in part 2 and compare your result to the results of part 2.

### III.3.4 Hermite Interpolation (in Lagrange form)

Hermite interpolation is a special form of polynomial interpolation. It uses data points as well as derivatives of the data to construct an interpolating polynomial. This can significantly reduce unwanted oscillations, in particular if it is applied in piecewise fashion.

In the following, we will consider only the simplest case of Hermite interpolation, which interpolates a function and its first derivative. If we have $n+1$ data points, then we need to find a polynomial that satisfies

$$p(x_i) = c_{i0} \ , \ \ p'(x_i) = c_{i1} \quad \text{for } i \in [0, n] \ , \tag{III.3.9}$$

where $c_{i0} = f(x_i)$ and $c_{i1} = f'(x_i)$.

In analogy with the Lagrange interpolation formula (Eq. III.3.6), we write

$$p(x) = \sum_{i=0}^{n} c_{i0} A_i(x) + \sum_{i=0}^{n} c_{i1} B_i(x) \ , \tag{III.3.10}$$

in which $A_i(x)$ and $B_i(x)$ are polynomials with certain properties:

$$\begin{aligned} A_i(x_j) &= \delta_{ij} \ , \quad B_i(x_j) = 0 \ , \\ A_i'(x_j) &= 0 \ , \quad B_i'(x_j) = \delta_{ij} \ . \end{aligned} \tag{III.3.11}$$

Using

$$L_{nj}(x) = \prod_{\substack{j=0 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k} \ , \tag{III.3.12}$$

Without proof: $A_i$ and $B_i$ can be defined as follows for points $0 \leq i \leq n$:

$$\begin{aligned} A_i(x) &= [1 - 2(x - x_i) L_{ni}'(x_i)] L_{ni}^2(x) \ , \\ B_i(x) &= (x - x_i) L_{ni}^2(x) \ . \end{aligned} \tag{III.3.13}$$

Note that each $L_{ni}$ is of degree $n$ and therefore $A_i$ and $B_i$ are of degree $2n+1$. This is also the maximal degree of $p(x)$. A derivation of Eq. (III.3.13) can be found in Doron Levy's lecture notes on Numerical Analysis, http://www.math.umd.edu/~dlevy/books/na.pdf.

### III.3.4.1 Piecewise Cubic Hermite Interpolation

When performing piecewise cubic Hermite interpolation on a large dataset, we set $n = 1$ and for each $x$ at which we want to approximate a function $f(x)$, we find two neighboring (bracketing) points $x_i$ and $x_{i+1}$, where $f(x_i)$, $f(x_{i+1})$, $f'(x_i)$, and $f'(x_{i+1})$ are known or can be evaluated numerically.

The cubic Hermite polynomial that interpolates $f(x)$ in $[x_i, x_{i+1}]$ is then given by

$$\begin{aligned} H_3(x) =& f(x_i)\psi_0(z) + f(x_{i+1})\psi_0(1-z) \\ &+ f'(x_i)(x_{i+1} - x_i)\psi_1(z) \\ &- f'(x_{i+1})(x_{i+1} - x_i)\psi_1(1-z) \ , \end{aligned}$$

where

$$z = \frac{x - x_i}{x_{i+1} - x_i} \ , \tag{III.3.14}$$

and

$$\psi_0(z) = 2z^3 - 3z^2 + 1 \ , \tag{III.3.15}$$

$$\psi_1(z) = z^3 - 2z^2 + z \ . \tag{III.3.16}$$

Of course, the $\psi_i$ are straightforwardly related to the $A_i$ and $B_i$ of the previous section. They have been written in this form for convenience. Note that in many cases one evaluates the first derivative numerically. This may be done with a centered finite-differenc approximation inside the interval and with one-sided derivatives on the boundaries.

### III.3.5 Spline Interpolation

Splines are the ultimate method for piecewise polynomial interpolation of strongly varying data if continuity and differentiability (= smoothness) of the interpolation result is important.

Spline interpolation achieves smoothness by requiring continuity at data points not only for the function values $f(x_i)$, but also for a number of its derivatives $f^{(l)}(x_i)$. Assuming that we know the values $f(x_i)$ at points $x_i$ ($i \in [0, n]$), we can construct piecewise polynomials on each segment $[x_i, x_{i+1}]$ (there are $n$ such segments) of degree $m$,

$$p_i(x) = \sum_{k=0}^{m} c_{ik} x^k \ , \tag{III.3.17}$$

to approximate $f(x)$ for $x \in [x_i, x_{i+1}]$. Note that $m$ is the degree of the spline and there are $m+1$ $c_{ik}$ for each $i$ and there are $n$ intervals. Hence we have $n(m+1)$ coefficients $c_{ik}$ that we must determined by

(a) requiring $(n-1)(m-1)$ smoothness conditions at non-boundary points: $p_i^l(x_{i+1}) = p_{i+1}^l(x_{i+1})$ for $l = 1, \cdots, m-1$,

(b) requiring $2n$ interpolation conditions: $p_i(x_i) = f(x_i) = p_{i+1}(x_i)$,

(c) choosing the remaining $m-1$ values of some of the $p_o^l(x_o)$ and $p_{n-1}^l(x_n)$ for $l = 1, \cdots, m-1$.

Note that the simplest spline, the linear spline ($m = 1$), is equivalent to piecewise linear interpolation.

#### III.3.5.1 Cubic Natural Spline Interpolation

Note: The following discussion is based on Tao Pang's book *An Introduction to Computational Physics*.

$m = 3$, a cubic, is the most widely used spline basis function. In this case, a total of $4n$ conditions are necesssary to find the coefficients $c_{ik}$. $m - 1 = 2$ must be picked by choosing values for some of the derivatives at both boundary points. One possible choice is to set the highest derivative to zero at boths ends of the interval. This is called the *natural spline*. For the cubic spline this means

$$p_0''(x_0) = 0 \ , \ \ p_{n-1}''(x_n) = 0 \ . \tag{III.3.18}$$

To construct the cubic spline, we start with the linear interpolation of its second derivative in $[x_i, x_{i+1}]$,

$$p_i''(x) = \frac{1}{x_{i+1} - x_i} \left[ (x - x_i) p_{i+1}'' - (x - x_{i+1}) p_i'' \right] \ , \tag{III.3.19}$$

where we have set $p_i'' = p_i''(x_i) = p_{i-1}''(x_i)$ and $p_{i+1}'' = p_{i+1}''(x_{i+1}) = p_i''(x_{i+1})$.

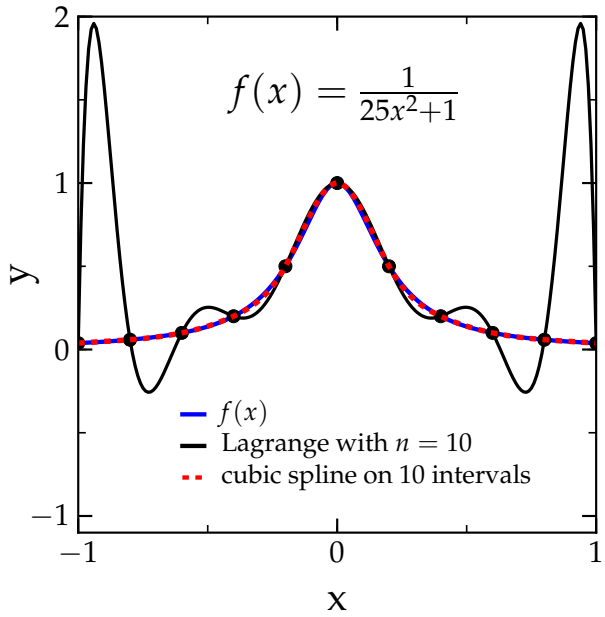



Figure III.3.3: Same as Fig. III.3.2, but this time showing only $f(x)$, the result of Lagrangean interpolation with $n = 10$, and of cubic natural spline interpolation on the same 10 intervals.

We now integrate Eq. (III.3.19) twice and identify $f_i = p_i(x_i) = f(x_i)$. We obtain

$$p_i(x) = \alpha_i(x - x_i)^3 + \beta_i(x - x_{i+1})^3 + \gamma_i(x - x_i) + \eta_i(x - x_{i+1}) \,, \tag{III.3.20}$$

where

$$\alpha_i = \frac{p''_{i+1}}{6h_i} \qquad \beta_i = \frac{-p''_i}{6h_i} \,, \tag{III.3.21}$$

$$\gamma_i = \frac{f_{i+1}}{h_i} - \frac{h_i p''_{i+1}}{6} \qquad \eta_i = \frac{h_i p''_i}{6} - \frac{f_i}{h_i} \,, \tag{III.3.22}$$

with $h_i = x_{i+1} - x_i$. Hence, all that is needed to find the spline is to find all of its second derivatives $p''_i = p''_i(x_i)$.

We may now apply the condition $p'_{i-1}(x_i) = p'_i(x_i)$ to Eq. (III.3.20) to obtain

$$h_{i-1}p''_{i-1} + 2(h_{i-1} + h_i)p''_i + h_i p''_{i+1} = 6\left(\frac{g_i}{h_i} - \frac{g_{i-1}}{h_{i-1}}\right) \,, \tag{III.3.23}$$

where $g_i = f_{i+1} - f_i$. This is a linear system with $n - 1$ unknowns $p''_i$ for $i = 1, \cdots, n-1$ and $p''_0 = p''_n = 0$ (as set by the natural spline condition).

If we set $d_i = 2(h_{i-1} + h_i)$ and $b_i = 6\left(\frac{g_i}{h_i} - \frac{g_{i-1}}{h_{i-1}}\right)$, then we can write

$$Ap'' = b \quad \text{with } A_{ij} = \begin{cases} d_i & \text{if } i = j \,, \\ h_i & \text{if } i = j - 1 \,, \\ h_{i-1} & \text{if } i = j + 1 \,, \\ 0 & \text{otherwise.} \end{cases} \tag{III.3.24}$$

Note that the coefficient matrix $A_{ij}$ is real, symmetric, and tri-diagonal. We will discuss later how to solve linear systems of equations of this type.

Figure III.3.3 shows how much better cubic spline interpolation deals with the example we used in §III.3.2 to show Runge's phenomenon.

### III.3.6 Multi-Variate Interpolation

In many situations, on ehas to deal with data that depend on more than one variable. A typical example are stellar opacities that may depend on frequency, gas density, temperature, and composition. Since they are too complicated to compute on the fly, one pre-tabulates them and then interpolates between table entries.

For deciding which approach to take for multi-variate interpolation, it is important to differentiate between two general kinds of multi-variate data:

(a) Data in a grid arrangement that are given on a regularly distributed (equidistantly or non-equidistantly) set of points (nodes).

(b) Scattered data, which are not arranged in a regular grid.

As we shall see, for case (a), we can use the so-called *Tensor Product Interpolation* to construct a multi-variate interpolation function.

For case (b), the situation is more complicated. One can try to first introduce a regular gridding of the data by linear interpolation between nearest neighbors or by piecewise constant nearest neighbor interpolation in which the value at the nearest irregular node is assigned to a node of an introduced regular grid. Alternatively, one may use a special method for multi-variate interpolation of scattered data, e.g., Shepard's method.

A good summary of methods for interpolation of scattered data can be found in Alfeld, *Scattered Data Interpolation in 3 or more Variables*, in Mathematical Methods in Computational Geometric Design, 1989.

#### III.3.6.1 Tensor Product Interpolation

For data arranged in a grid, a multi-variate (here: $n$-variate) interpolation function can be obtained by forming the tensor product

$$g(x_1, \cdots, x_n) = g_1(x_1) \circ g_2(x_2) \circ \cdots \circ g_n(x_n) \ , \tag{III.3.25}$$

using $n$ univariate interpolation functions $g_i$.

This sounds much more complicated than it really is. The above simply means that true multivariate (multi-dimensional) interpolation is mathematically equivalent to executing multiple univerative (1D) interpolations sequentially. So, if, for example, we want to interpolate a function $U(x, y, z)$, we would first interpolate it in $x$, then interpolate the result in $y$, and then that result in $z$. This, of course, requires many interpolation steps. Hence, it is generally more efficient to analytically work out and simplify the multi-variate interpolation function and then interpolate in only one step.

**Exercise III.3.2 (Bilinear Interpolation)**

Construct a bilinear interpolation polynomial that interpolates a function $f(x, y)$ for points $(x, y)$ in $[x1, x2]$ and $[y1, y2]$, respectively.

#### III.3.6.2 Shepard's Method for Scattered Data

*Shepard's Method* uses inverse distance weighting for finding an interpolated value $u$ at a given point $\vec{x}$ based on irregularly arranged data (samples) $u_i(\vec{x}_i)$ $(i = 0, \cdots, n)$:

$$u(\vec{x}) = \sum_{i=0}^{n} \frac{w_i(\vec{x})\, u_i}{\sum_{j=0}^{n} w_j(\vec{x})} \ , \quad \text{with} \quad w_i(\vec{x}) = \frac{1}{[d(\vec{x}, \vec{x}_i)]^p} \ . \tag{III.3.26}$$

Here, the $w_i(\vec{x})$ are distance-dependent weights, $d(\vec{x}, \vec{x}_i)$ is the distance function, and $p$ is the power parameter ($p > 0$). The weight decreases as distance increases from the known points. The power parameter $p$ controls the smoothness of the interpolation. For $0 < p \leq 1$, $u(\vec{x})$ is rather smooth and for $p > 1$ it quickly becomes sharp.

The *Modified Shepard's Method* uses a modified weight functions

$$w_i(\vec{x}) = \left( \frac{R - d(\vec{x}, \vec{x}_i)}{R d(\vec{x}, \vec{x}_i)} \right)^2 \ , \tag{III.3.27}$$

which emphasises points within a sphere of radius $R$ around $\vec{x}$.

## III.4 Integration

There are many situations in which we need to evalute the integral

$$Q = \int_a^b f(x)dx \ . \tag{III.4.1}$$

However, integrals can be evaluated analytically only for very few well behaved functions $f(x)$, plus, frequently, we have to deal with discrete data $f(x_i)$ that, of course, cannot be integrated analytically. Hence, we have to resort to numerical integration (quadrature).

### III.4.1 Integration based on Piecewise Polynomial Interpolation

Assume that we know (or can evaluate) $f(x)$ at a finite set of points (nodes) $\{x_j\}$ with $j = 0, \cdots, n$ in the interval $[a, b]$. Then we can replace $f(x)$ with a simpler function $p(x)$ whose analytic integral we know and which interpolates $f(x)$: $p(x_i) = f(x_i)$. Such integration formulae based on interpolation polynomials are generally called Newton-Cotes quadrature formulae.

In the following, we will assume that the full interval over which we intend to integrate can be broken down into $N$ sub-intervals $[a_i, b_i]$ that encompass $N+1$ nodes $x_i$ ($i = 0, \cdots, N$) at which we know the integrand $f(x)$. We can then express the full integral $Q$ as the sum of the sub-integrals $Q_i$:

$$Q = \sum_{i=0}^{N-1} Q_i = \sum_{i=0}^{N-1} \int_{a_i}^{b_i} f(x)dx \ . \tag{III.4.2}$$

*Open* Newton-Cotes quadruature formulae estimate an integral based on points strictly in $(a_i, b_i)$, while *closed* Newton-Cotes quadrature formulae include the values at the end points $f(a_i)$ and $f(b_i)$.

#### III.4.1.1 Midpoint Rule

The simplest approximation is to assume that the function $f(x)$ is constant on the interval $[a_i, b_i]$ and to use its central value value:

$$Q_i = \int_{a_i}^{b_i} f(x)dx = (b_i - a_i)\, f\left(\frac{a_i + b_i}{2}\right) \ . \tag{III.4.3}$$

This is also called the "rectangle rule" and is an open Newton-Cotes formula. Note that we need to be able to evaluate $f(x)$ at the midpoint. We may not be able to do this if we know $f(x)$ only at discrete locations.

The error in the midpoint quadrature can be estimated using a Taylor expansion about the midpoint $m_i = (a_i + b_i)/2$ of the interval $[a_i, b_i]$:

$$f(x) = f(m_i) + f'(m_i)(x - m_i) + \frac{f''(m_i)}{2}(x - m_i)^2 + \frac{f'''(m_i)}{6}(x - m_i)^3 + ... \tag{III.4.4}$$

Integrating this expression from $a_i$ to $b_i$, the odd-order terms drop out, and what is left is

$$Q_i = \int_{a_i}^{b_i} f(x)dx = f(m_i)(b_i - a_i) + \frac{f''(m_i)}{24}(b_i - a_i)^3 + ... \ . \tag{III.4.5}$$

So we see that the error goes to leading order with $h^3 = (b_i - a_i)^3$. The next higher order is $h^5$, but the $h^3$ term will dominate as long as $h$ is sufficiently small that $h^3 \gg h^5$ and the fourth derivative of $f$ is well behaved.

The above error pertains to the sub-interval $[a_i, b_i]$ of the full intervale $[a, b]$ over which we intend to integrate. So the total error of the quadrature will be of order $N - 1 \approx N$ *times* $h$, where, for constant interval size, $h = (a - b)/N$. So if the error is locally of order $h^3 = (a - b)^3/N^3$, it is globally scaling with $Nh^3 = (a - b)^3/N^2$, so decreases quadratically with the number of sub-intervals.

It is interesting to note that the midpoint rule uses an intergrating polynomial of degree zero (a constant), yet its leading-order error depends on the second derivative of the integrand. Hence, it will integrate constant *and* linear polynomials exactly.

#### III.4.1.2 Trapezoidal Rule

Here we approximate $f(x)$ with a linear polynomial on $[a, b]$,

$$Q_i = \int_{a_i}^{b_i} f(x)dx = \frac{1}{2}(b_i - a_i)\left[f(b_i) + f(a_i)\right] \ . \tag{III.4.6}$$

One can easily show (as we just have via Taylor expansion for the midpoint rule) that the trapezoidal rule has a local error that, like the midpoint rule, goes with $h^3$, despite the fact that in each sub-interval a linear polynomial is used to interpolate $f(x)$. As pointed out by Heath [1], one can generally show that Newton-Cotes formulae with interpolating polynomials using $n$ points will be of degree $n$ if $n$ is odd and of degree $n - 1$ if $n$ is even. This is why midpoint and trapezoidal rule have the same convergence order. Because of the different nature of the two rules, it turns out that for general well-behaved functions the midpoint rule is actually *more* accurate than the trapezoidal rule. This means that the constant in front of the error term in the series expansion is smaller in the midpoint rule than in the trapezoidal rule. This has to do with the fact that errors made left and right of the midpoint cancel partially with each other in the midpoint rule.

#### III.4.1.3 Simpson's Rule

Simpson's Rule approximates $f(x)$ on $[a_i, b_i]$ by a quadratic (a parabola; a polynomial of degree 2). Writing out the interpolating polynomial on $[a_i, b_i]$ and integrating it, one obtains

$$Q_i = \frac{b_i - a_i}{6}\left[f(a_i) + 4f\left(\frac{a_i + b_i}{2}\right) + f(b_i)\right] + \mathcal{O}([b_i - a_i]^5) \ . \tag{III.4.7}$$

Note that in the above we have assumed that we know or can evaluate $f((a_i + b_i)/2)$. If we know $f(x)$ only at discrete locations, all hope is not lost. We can always combine two intervals $[a_i, b_i]$ and $[b_i, b_{i+1}]$ to $[a_i, b_{i+1}]$ and identify $a_i = x_i$, $b_i = a_{i+1} = x_{i+1}$, and $b_{i+1} = x_{i+2}$. Of course, the error will then scale with the fifth power of the combined interval locally. Globally, the error will scale with $Nh^5 = N(a - b)^5/N^5 = (a - b)^5/N^4$.

Note that special care must be taken at the boundaries of the interpolation interval. If it is not possible to analytically extend the integrand beyond the boundary to obtain the needed integrand values, one must (a) use a lower-order integration method at the end points (but this can spoil the

global convergence rate somewhat) or (b) resort to a one-side interpolation polynomial that relies exclusively on data that is in the integration interval.

So far, we have considered only equidistant intervals of size $h = [a_i, b_i]$. If $f(x)$ is non-equidistantly sampled, we need to use a modification of Simpson's rule:

First we rewrite the interpolating quadratic as

$$f(x) = ax^2 + bx + c \qquad \text{for } x \in [x_{i-1}, x_{i+1}] \ , \tag{III.4.8}$$

where,

$$\begin{aligned} a &= \frac{h_{i-1}f(x_{i+1}) - (h_{i-1}+h_i)f(x_i) + h_i f(x_{i-1})}{h_{i-1}h_i(h_{i-1}+h_i)} \ , \\ b &= \frac{h_{i-1}^2 f(x_{i+1}) + (h_i^2 - h_{i-1}^2)f(x_i) - h_i^2 f(x_{i-1})}{h_{i-1}h_i(h_{i-1}+h_i)} \ , \\ c &= f(x_i) \ , \end{aligned} \tag{III.4.9}$$

with $h_i = x_{i+1} - x_i$ and $h_{i-1} = x_i - x_{i-1}$ and $x_i = 0$. Note that the latter is fine, since

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx$$

is independent of the origin of the coordinates.

Now, with $x_i = 0$, $-h_{i-1} = x_{i-1}$, $h_i = x_{i+1}$ we obtain

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx = \int_{-h_{i-1}}^{h_i} f(x)dx = \alpha f(x_{i+1}) + \beta f(x_i) + \gamma f(x_{i-1}) \ , \tag{III.4.10}$$

with

$$\begin{aligned} \alpha &= \frac{2h_i^2 + h_i h_{i-1} - h_{i-1}^2}{6h_i} \ , \qquad \beta = \frac{(h_i + h_{i-1})^3}{6h_i h_{i-1}} \ , \\ \gamma &= \frac{-h_i^2 + h_i h_{i-1} + 2h_{i-1}^2}{6h_{i-1}} \ , \end{aligned} \tag{III.4.11}$$

**Example:** $f(x) = x^2$, to be integrated on $[-0.5, 1]$ with $x_{i-1} = -0.5$, $x_i = 0$, $x_{i+1} = 1$.

$$\alpha = \frac{2 + 0.5 - 0.25}{6} = 0.375 \ , \ \beta = \frac{3.375}{3} = 1.125 \ , \ \gamma = 0 \ ,$$

and using Eq. III.4.10, we obtain

$$\int_{x_{i-1}}^{x_{x+1}} f(x)dx = 0.375 \ .$$

The analytic result is, of course,

$$\int_{-0.5}^{1} x^2 dx = \left[\frac{x^3}{3}\right]_{-0.5}^{1} = \frac{1}{3} + \frac{1}{3}\frac{1}{8} = 0.375.$$

Hence, the integral of our quadratic interpolating polynomial integrates the quadratic $f(x) = x^2$ exactly.

Note that as it is the case with the previously discussed methods, Simpson's rule is globally convergent to one order less than its local convergence rate.

### III.4.2 Gaussian Quadrature

*As I have received your paper about the approximate integration, I no longer can withstand to thank you for the great pleasure you have given to me.*

From a letter written by Bessel to Gauss.

In the integration methods discussed thus far, $n$ nodes were pre-specified and $n$ corresponding weights were then chosen to maximize the degree of the resulting integration method. A method with $n$ pre-specified nodes can integrate a polynomial of degree $n-1$ exactly. For example, as we have seen in the above §III.4.1.3, Simpson's rule can integrate $f(x) = x^2$ exactly.

Gaussian quadrature is an integration method in which both the nodes and the weights are optimally chosen to maximize the degree of the resulting integration rule. Since there are now $2n$ degrees of freedom, polynomials of degree $2n-1$ can be exactly integrated. For a set of $n$ nodes $x_i$ and weights $w_i$ at which a function $f$ is known:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{III.4.12}$$

#### III.4.2.1 2-point Gaussian Quadrature

As a first example of Gaussian Quadrature, we will derive a 2-point quadrature on the interval $[a,b]$. We expect

$$\int_a^b f(x)dx = w_1 f(x_1) + w_2 f(x_2) \tag{III.4.13}$$

to integrate a polynomial of degree 3 exactly. The unknowns are $w_1$, $w_2$, $x_1$, and $x_2$. We can find them by demanding what we expect, namely that the formula give exact results for integrating a general polynomial of degree 3:

$$\begin{aligned}\int_a^b f(x)dx &= \int_a^b (c_0 + c_1 x + c_2 x^2 + c_3 x^3)dx \ , \\ &= c_0(b-a) + c_1\left(\frac{b^2-a^2}{2}\right) + c_2\left(\frac{b^3-a^3}{3}\right) + c_3\left(\frac{b^4-a^4}{4}\right) \ .\end{aligned} \tag{III.4.14}$$

We also have

$$\int_a^b f(x)dx = w_1 f(x_1) + w_2 f(x_2) = w_1(c_0 + c_1 x_1 + c_2 x_1^2 + c_3 x_1^3) + w_2(c_0 + c_1 x_2 + c_2 x_2^2 + c_3 x_2^3) \ . \tag{III.4.15}$$

Equating (III.4.14) and (III.4.15) gives

$$\begin{aligned}c_0(b-a) + c_1\left(\frac{b^2-a^2}{2}\right) + c_2\left(\frac{b^3-a^3}{3}\right) + c_3\left(\frac{b^4-a^4}{4}\right) \ , \\ = w_1(c_0 + c_1 x_1 + c_2 x_1^2 + c_3 x_1^3) + w_2(c_0 + c_1 x_2 + c_2 x_2^2 + c_3 x_2^3) \ , \\ = c_0(w_1 + w_2) + c_1(w_1 x_1 + w_2 x_2) + c_2(w_1 x_1^2 + w_2 x_2^2) + c_3(w_1 x_1^3 + w_2 x_2^3) \ .\end{aligned} \tag{III.4.16}$$

Since the $c_i$ are arbitrary, we can now demand:

$$(1)\ b-a = w_1 + w_2 \qquad (3)\ \frac{b^3-a^3}{3} = w_1 x_1^2 + w_w x_2^2\ , \tag{III.4.17}$$

$$(2)\ \frac{b^2-a^2}{2} = w_1 x_1 + w_2 x_2 \qquad (4)\ \frac{b^4-a^4}{4} = w_1 x_1^3 + w_2 x_2^3\ . \tag{III.4.18}$$

From this, we obtain

$$w_1 = \frac{b-a}{2}\ , \quad w_2 = \frac{b-a}{2}\ , \tag{III.4.19}$$

$$x_1 = \left(\frac{b-a}{2}\right)\left(-\frac{1}{\sqrt{3}}\right) + \left(\frac{b+a}{2}\right)\ , \tag{III.4.20}$$

$$x_2 = \left(\frac{b-a}{2}\right)\left(\frac{1}{\sqrt{3}}\right) + \left(\frac{b+a}{2}\right)\ . \tag{III.4.21}$$

$$\tag{III.4.22}$$

So, in the special case of the interval $[-1,1]$:

$$w_1 = w_2 = 1\ , \quad x_1 = -\frac{1}{\sqrt{3}}\ , \quad x_2 = \frac{1}{\sqrt{3}}\ , \tag{III.4.23}$$

and

$$\int_{-1}^{1} f(x)dx = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)\ . \tag{III.4.24}$$

Of course, for this to work, $f(x_1)$ and $f(x_2)$ must be known.

#### III.4.2.2 Gauss-Legendre Quadrature Formulae

Finding the nodes (and weights) becomes increasingly difficult and computationally cumbersome with increasing $n$. One convenient way to find the nodes is to use orthogonal polynomials, e.g. Legendre polynomials, which leads us to Gauss-Legendre quadrature. Once the nodes are found, a linear system of equations can be solved to find the weights.

If two polynomials $f$ and $g$ are orthogonal, then their scalar product over the interval $[a,b]$,

$$\langle f|g\rangle = \int_a^b f(x)g(x)dx = 0\ .$$

Suppose we have a family of orthorgonal polynomials $p_j(x)$, where $j$ indicates the degree, and there is a unique polynomial for each $j$, then the following can be shown (though we do not do this here. See, e.g., [2] for a proof):

(a) The polynomial $p_j(x)$ has exactly $j$ distinct roots in the interval $(a,b)$ (a root is never at $a$ or $b$).

(b) The roots of $p_j(x)$ interleave the $j-1$ roots of $p_{j-1}(x)$, that is, there is exactly one root of the former in between each two adjacent roots of the latter.

(c) The sought-after nodes of the Gaussian quadrature formula of the form of Eq. (III.4.12) with $n$ nodes are precisely the $n$ roots of the polynomial $p_n(x)$ in the interval $(a,b)$.

Now this is convenient! If we know such a family of polynomials and know their roots in some interval $[a, b]$, then we can integrate polynomials of degree $2n-1$ exactly!

Legendre polynomials are such a family of orthorgonal polynomials. They are defined on the interval $[-1, 1]$ and constructed by the following recurrence relation:

$$\begin{aligned} p_{-1} &= 0 \,, \\ p_0 &= 1 \,, \\ (j+1)p_{j+1} &= (2j+1)xp_j - jp_{j-1} \,. \end{aligned} \tag{III.4.25}$$

So, for example, for $j = n = 2$ we have

$$p_2 = \frac{1}{2}(3x^2 - 1) \,,$$

which has roots at $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$ and, hence, gives the same 2-point Gaussian quadrature rule with weights $w_1 = w_2 = 1$ as derived in the previous section §III.4.2.1. Higher order Gaussian quadrature rules can now be easily generated by looking up Legendre polynomials and their roots in $[-1, 1]$. However, care must be taken to first transform any integral over the interval $[a, b]$ to an integration over the interval $[-1, 1]$.

### III.4.2.3 Change of Interval

General Gaussian quadrature formulae are usually given for special, fixed intervals. To make practical use of them, the integral in question must be transformed. For the special case $[-1, 1]$, this works in the following way:

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) dx \,. \tag{III.4.26}$$

In formal terms, this is an affine transformation that maps $[a, b] \to [-1, 1]$ via

$$t = \frac{b-a}{2}x + \frac{a+b}{2} \,, \tag{III.4.27}$$

and a change of variables

$$dt = \frac{b-a}{2}dx \,. \tag{III.4.28}$$

### III.4.2.4 General Higher-Point Gaussian Quadrature Formulae

An additional highly useful feature of Gaussian quadrature formulae is that it is possible to choose a special integration interval $[a, b]$ over which the *integral of the product of a weight function* $W(x)$ *and of a polynomial of degree* $2n-1$ *is exact*. We now have

$$\int_a^b W(x)f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i) \,, \tag{III.4.29}$$

which is exact if $f(x)$ is a polynomial of degree $2n-1$. This is useful, because the weight function can be chosen to remove integrable singularities or parts of the integrand that cannot be expresses as a polynomial. For Gauss-Legendre quadrature, the weight function is trivially $W(x) = 1$.

Table III.1: Types of Gaussian Quadrature

| $[a,b]$ | $W(x)$ | Type of Gaussian Quadrature |
|---|---|---|
| $[-1,1]$ | $1$ | Gauss-Legendre |
| $[-1,1]$ | $(1-x^2)^{-1/2}$ | Gauss-Chebychev |
| $[0,\infty)$ | $x^c e^{-x}$ | Gauss-Laguerre |
| $(-\infty,\infty)$ | $e^{-x^2}$ | Gauss-Hermite |

As an example, consider next a function $g(x)$ that contains a factor $1/\sqrt{1-x^2}$, which is obviously singular at $x=0$. We can re-write the integral over $g(x)$ as

$$\int_a^b g(x)dx = \int_a^b \frac{f(x)}{\sqrt{1-x^2}}dx \approx \sum_{i=1}^{n} w_i f(x_i) \ ,$$

where $f(x)$ is a function (ideally, a polynomial) not containing any factors of the weight function $W(x) = (1-x^2)^{-1/2}$. The orthorgonal polynomials that are needed to derive the nodes and weights for this kind of weight functions for the interval $[-1,1]$ are the *Chebychev* polynomials of the first kind.

There is a number of different Gaussian quadrature formulae that are named after the polynomials that they use to determine the nodes and weights. Table III.1 summarizes some types of Gaussian quadrature that work over different intervals and have differing weight functions. The corresponding $x_i$ and $w_i$ are tabulated and can easily be found online.

## References

[1] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, New York, NY, USA, 2 edition, 2002.

[2] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, Heidelberg, Germany, 1993.

## III.5 Root Finding

In a broad range of applications one encounters situations in which it is necessary to find the roots of a function, i.e., the values of $x$ ($x$ could be a scalar or a vector) for which $f(x) = 0$ (where $f$ could be a single equation or a system of equations). $f$ can either directly depend on $x$ (explicitly) or have and implicit dependence on $x$.

There are a variety of ways to find the roots of an equation.

### III.5.1 Newton's Method

Newton's Method is also referred to as the "Newton-Raphson Method". If we expand a function $f(x)$ about its root $x_r$, we get:

$$f(x_r) = f(x) + (x_r - x)f'(x) + \mathcal{O}((x_r - x)^2) = 0 \ . \tag{III.5.1}$$

In this, $x_r$ can be seen a trial value for the root $x_r$ at the $n$-th step of an iterative procedure. The $n+1$-th step is then

$$f(x_{n+1}) = f(x_n) + \underbrace{(x_{n+1} - x_n)}_{\delta x} f'(x_n) \approx 0 \ , \tag{III.5.2}$$

and, thus,

$$x_{n+1} = x_n + \delta x = x_n - \frac{f(x_n)}{f'(x_n)} \ . \tag{III.5.3}$$

The iteration is stopped when $f(x_{n+1}) = 0$ (which rarely happens, because we are dealing with floating point numbers) or when the fractional change between iteration $n$ and $n+1$ is smaller than some small number: $|[f(x_{n+1}) - f(x_n)]/f(x_n)| < \epsilon$. One should not expect $\epsilon$ to be smaller than floating point accuracy.

Newton's Method as quadratic convergence, provided $f(x)$ is well behaved and that one has a good initial guess for the root. It also requires the ability to evaluate the derivative $f'(x_n)$ directly. If this is not possible, one resorts to the Secant Method.

### III.5.2 Secant Method

The Secant Method is just like Newton's method, but this time, we have to evaluate the first derivative $f'(x_n)$ numerically. This is usually done with a backward difference:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \ . \tag{III.5.4}$$

Of course, this method will converge less rapidly than Newton's method, because we have introduced a first-order derivative. Also, we need now two points to start the iteration (or some other guess at $f'(x_n)$ at $n = 0$). Like Newton's method, the secant method may fail for misbehaved functions and a good initial guess at the root helps a lot in finding it.

### III.5.3 Bisection

The intermediate-value theorem states that a continuous function $f(x)$ has at least one root in the interval $[a, b]$ if $f(a)$ and $f(b)$ are of opposite sign (this seems self-evident, but mathematicians like to have theorems and proofs for such things).

The bisection method – which is really more of an algorithm than anything – exploits the intermediate-value theorem. It goes as follows:

(i) Pick initial values of $a$ and $b$ so that $f(a)$ and $f(b)$ have opposite sign.

(ii) Compute the midpoint $c = \frac{a+b}{2}$. If $f(c) = 0$ or $|[f(c) - f(a)]/f(a)| < \epsilon$ or $|[f(c) - f(b)]/f(b)| < \epsilon$, then one is done.
If not:
  (1) If $f(a)$ and $f(c)$ have opposite sign, then they bracket a root. Go to (i) with $a = a$, $b = c$.
  (2) If $f(c)$ and $f(b)$ have opposite sign, then they bracket a root. Got to (i) with $a = c$, $b = b$.

The bisection method is very effective, more robust, but generally not as fast as Newton's Method, requiring more iterations until a root is found.

### III.5.4 Multi-Variate Root Finding

It is often the case that one needs to simultaneously find the roots of multiple equations with multiple independent variables.

$\mathbf{f}(\mathbf{x})$ is a multi-variate vector function and we are looking for $\mathbf{f}(\mathbf{x}) = 0$. Analogously to the scalar case, we write the multi-variate Newton's/Secant Method,

$$\mathbf{f}(\mathbf{x}_{n+1}) \approx \mathbf{f}(\mathbf{x}_n) + \nabla \otimes \mathbf{f}(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = 0 \ , \tag{III.5.5}$$

and

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [\nabla \otimes \mathbf{f}(\mathbf{x}_n)]^{-1} \mathbf{f}(\mathbf{x}) \ . \tag{III.5.6}$$

Here,

$$\mathbf{J} \equiv \nabla \otimes \mathbf{f}(\mathbf{x}_n) \ , \tag{III.5.7}$$

is the Jacobian matrix. In index notation, it is given by

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \ . \tag{III.5.8}$$

There are also multi-variate variants of the bisection method, but they are rather complex and we will not discuss them here.

# III.6 Optimization

## III.6.1 Curve Fitting

Empirical relationships (e.g., the $M - \sigma$ relation for galaxies) are typically established by taking experimental/observational data and fitting an analytic function to them. In this section, we will introduce the most common curve fitting methods. The text in this section is heavily influenced by the discussion in Garcia's book [1].

The most common way to fit a function $Y(x, \{a_j\})$ with $M$ parameters $\{a_j\}$ to $N$ data points $(x_i, y_i)$ is the *least squares fit*. We define the difference between $Y$ and $y_i$ at point $i$ as

$$\Delta_i = Y(x_i, \{a_j\}) - y_i \ , \tag{III.6.1}$$

and our goal is to minimize the function

$$D(\{a_j\}) = \sum_{i=1}^{N} \Delta_i^2 = \sum_{i=1}^{N} \left(Y(x_i, \{a_j\}) - y_i\right)^2 \ . \tag{III.6.2}$$

The reason one uses the square of the difference is obvious: negative and positive variations would otherwise partially or fully cancel out and we would get a wrong result that looks right.

The observational data at points $(x_i, y_i)$ will have an estimated error (or confidence interval) associated with them, so at $x_i$ we have $y_i \pm \sigma_i$ (we will address how to deal with errors in the $x_i$ later). Taking the estimated errors into account, we should modify our fit criterion so as to give less weight to the points with the most error. We define the *chi-square* function

$$\chi^2(\{a_j\}) = \sum_{i=1}^{N} \left(\frac{\Delta_i}{\sigma_i}\right)^2 = \sum_{i=1}^{N} \left(\frac{Y(x_i, \{a_j\}) - y_i}{\sigma_i}\right)^2 \ , \tag{III.6.3}$$

which we now aim to minimize.

### III.6.1.1 Linear Regression

The simplest curve to which we can try to fit our data is a stright line. Fitting to

$$Y(x, \{a_1, a_2\}) = a_1 + a_2 x \ , \tag{III.6.4}$$

is known as *linear regression*. We want to determine $a_1$ and $a_2$ such that

$$\chi^2(a_1, a_2) = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}(a_1 + a_2 x_i - y_i)^2 \tag{III.6.5}$$

is minimized. We can find the minimum by differentiating Eq. (III.6.5) and setting the result to zero:

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_1} &= 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2}(a_1 + a_2 x_i - y_i) = 0 \ , \\ \frac{\partial \chi^2}{\partial a_2} &= 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2}(a_1 + a_2 x_i - y_i) x_i = 0 \ . \end{aligned} \tag{III.6.6}$$

This can be written more concisely in this way:

$$\begin{aligned} a_1 S + a_2 \Sigma x - \Sigma y = 0 \ , \\ a_1 \Sigma x + a_2 \Sigma x^2 - \Sigma xy = 0 \ , \end{aligned} \tag{III.6.7}$$

with

$$\begin{aligned} S = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \ , \quad \Sigma x = \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} \ , \quad \Sigma y = \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \ , \\ \Sigma x^2 = \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \ , \quad \Sigma xy = \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \ . \end{aligned} \tag{III.6.8}$$

The sums can be computed directly from the data, hence they are known constants. Hence, we have a linear set of equations in the two unknowns $a_1$ and $a_2$, which can be solved directly:

$$a_1 = \frac{\Sigma y \Sigma x^2 - \Sigma x \Sigma xy}{S\Sigma x^2 - (\Sigma x)^2} \ , \qquad a_2 = \frac{S\Sigma xy - \Sigma y \Sigma x}{S\Sigma x^2 - (\Sigma x)^2} \ . \tag{III.6.9}$$

Note that if all $\sigma_i$ are identical, they will cancel out of the above equations and $a_1$ and $a_2$ will be independent of them. Furthermore, if the $\sigma_i$ are unknown, then one can still use the $\chi^2$ method and just sets $\sigma_i = 1$.

Incorporating uncertainty in the $x_i$ in the $\chi^2$ fit must be handled by relating the error $\sigma_i^x$ into an additional error in the $y_i$, $\sigma_i^{\text{extra}}$. To first order, this can be done by writing

$$\sigma_{i,\text{extra}} = \left| \frac{\partial y}{\partial x} \right|_i \sigma_i^x \ , \tag{III.6.10}$$

where one needs an appropriate approximation for the slope $\partial y / \partial x$. If both $\sigma_i$ and $\sigma_{i,\text{extra}}$ contribute significantly, one simply adds their squares: $\sigma_{i,\text{total}}^2 = \sigma_i^2 + \sigma_{i,\text{extra}}^2$. If the error in $x_i$ or $y_i$ is asymmetric about $(x_i, y_i)$ one could weigh by the maximum of the left and right error, or use advanced techniques that are explicitly able to incorporate information about asymmetric error regions.

Next, we want to obtain an associated error bar, $\sigma_{a_j}^2$, for the curve fit parameter $a_j$. Using first-order error propagation, we have

$$\sigma_{a_j}^2 = \sum_{i=1}^{N} \left( \frac{\partial a_j}{\partial y_i} \right)^2 \sigma_i^2 \ , \tag{III.6.11}$$

from which we obtain with Eq. (III.6.9)

$$\sigma_{a_1} = \sqrt{\frac{\Sigma x^2}{S\Sigma x^2 - (\Sigma x)^2}} \ , \qquad \sigma_{a_2} = \sqrt{\frac{S}{S\Sigma x^2 - (\Sigma x)^2}} \ . \tag{III.6.12}$$

Should our data set not have an associated set of error bars, we may estimate $\sigma_{a_j} = \sigma_0$ from the sample variance of the data,

$$\sigma_0^2 = \frac{1}{N-2} \sum_{i=1}^{N} \left( y_i - (a_1 + a_2 x_i) \right)^2 \ . \tag{III.6.13}$$

The normalization factor $N - 2$ of the variance is due to the fact that we have already extraed two parameters, $a_1$ and $a_2$, from the data.

#### III.6.1.2 Reduction of Non-Linear Fitting Problems

Many non-linear fitting problems may be transformed to linear problems by a simple change of variables. A typical example is a power law of the form

$$Z(t,\{\alpha,\beta\}) = \alpha t^{\beta} \ . \tag{III.6.14}$$

This may be rewritten to

$$Y = \log Z \ , \quad x = \log t \quad a_1 = \log \alpha \ , \quad a_2 = \beta \ . \tag{III.6.15}$$

The same can, of course, be done with an exponential:

$$Z(t,\{\alpha,\beta\}) = \alpha e^{\beta x} \ , \tag{III.6.16}$$

and

$$Y = \ln Z \ , \quad a_1 = \ln \alpha \ , \quad a_2 = \beta \ . \tag{III.6.17}$$

### III.6.2 General Least Squares Fit

The least squares fit procedure is easy to generalize to functions of the form

$$Y(x,\{a_j\}) = a_1 Y_1(x) + \cdots + a_M Y_(x) = \sum_{j=1}^{M} a_j Y_j(x) \ . \tag{III.6.18}$$

To find the optimum parameters, we proceed as before by finding the minimum of $\chi^2$,

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_j} &= \frac{\partial}{\partial a_j} \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left( \sum_{k=1}^{M} a_k Y_k(x_i) - y_i \right)^2 = 0 \ , \\ &= \sum_{i=1}^{N} \frac{2}{\sigma_i^2} Y_j(x_i) \left( \sum_{k=1}^{M} a_k Y_k(x_i) - y_i \right) = 0 \ , \\ \Leftrightarrow \sum_{i=1}^{N} \sum_{k=1}^{M} & \frac{Y_j(x_i) Y_k(x_i)}{\sigma_i^2} a_k = \sum_{i=1}^{N} \frac{Y_j(x_i) y_i}{\sigma_i^2} \ , \end{aligned} \tag{III.6.19}$$

for each $j$ in $1, \cdots, M$. These *normal equations* of the least squares problem are more concisely written in matrix form. We define the *design matrix* $A$ via

$$A_{ij} = \frac{Y_j(x_i)}{\sigma_i} \ . \tag{III.6.20}$$

With this, Eq. (III.6.19) becomes

$$\sum_{i=1}^{N} \sum_{k=1}^{M} A_{ij} A_{ik} a_k = \sum_{i=1}^{N} A_{ij} \frac{y_i}{\sigma_i} \ . \tag{III.6.21}$$

In abstract notation, we have

$$(A^T A)\mathbf{a} = A^T \mathbf{b} \ , \tag{III.6.22}$$

which is easily solved for $\mathbf{a}$. The estimated error in the parameter $a_j$ is then given by $\sigma_{a_j} = \sqrt{C_{jj}}$, where $C = (A^T A)^{-1}$.

#### III.6.2.1 Goodness of Fit

One can easily fit every single data point exactly if the number of parameters $M$ equals the number of data points $N$. But Occam's Razor tells us: A theory with too many parameters is no good. In fact, we have seen in §III.3.2 that using all available information ($N$ points) to construct a single complex "fitting" function of order $M = N - 1$, leads to terrible oscillations.

In general, we will have "theories" (i.e., fitting functions) with $M \ll N$ and because every data point has an error, we do not expect the curve to exactly pass through the data. However, we may ask, "With the given error bars, how likely is it that the curve actually describes the data?" Of course, if we are not given any error bars, there is nothing we can say about the goodness of the fit.

To first order, we would expect that if the fit is good, then on average the difference between fit and data should be approximately equal to the error bar, so

$$|y_i - Y(x_i)| \approx \sigma_i \ . \tag{III.6.23}$$

Putting this into the definition for $\chi^2$ (Eq. III.6.3), we obtain

$$\chi^2 \approx N \tag{III.6.24}$$

Yet we know that if we use $M = N$ parameters, we can exactly fit the data and $\chi^2 = 0$, so we modify our criterion for goodness of fit to

$$\chi^2 \approx N - M \ . \tag{III.6.25}$$

Of course, this is just a crude indicator. A more rigorous analysis would use $\chi^2$-statistic to assign a probability that the data are fit by the curve.

Sticking with our crude estimate: If we find $\chi^2 \gg N - M$, then either we are not using an appropriate function $Y(x)$ for our fit or the error bars $\sigma_i$ are too small. On the other hand, if $\chi^2 \ll N - M$, then the fit is so spectacularly good that we may suspect that the error bars are actually too large.

## References

[1] *Numerical Methods for Physics*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1994.

## III.7 The Fourier Transform

The Fourier transform was originally developed by Joseph Fourier in 1822. His work studying the heat equation (a partial differential equation that describes the flow of heat in a medium) led him to look for eigenfunctions of the Laplacian. In $\mathbb{R}^3$, the Laplacian is defined by

$$\nabla^2 \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}. \tag{III.7.1}$$

Eigenfunctions $\phi$ of this operator should satisfy

$$\nabla^2 \phi = \lambda \phi, \tag{III.7.2}$$

where $\lambda$ is the eigenvalue associated with the eigenfunction $\phi$. It is obvious that $\phi$ should take the form $\phi \propto \exp(-i\boldsymbol{k}\cdot\boldsymbol{x})$, where $\boldsymbol{x} \equiv (x, y, z)$ and $\boldsymbol{k} \in \mathbb{R}^3$. Using this form for $\phi$ and Eq. III.7.2 it can be found that the eigenvalue associated with $\phi$ is $\lambda = -k^2$. We will label each eigenfunction and eigenvalue with the vector $\boldsymbol{k}$. For reasons that will become clear after the next paragraph, we will also choose the normalization such that

$$\phi_{\boldsymbol{k}} = \frac{1}{(2\pi)^{3/2}} \exp\Big(i\boldsymbol{k}\cdot\boldsymbol{x}\Big). \tag{III.7.3}$$

It can be shown that the $\{\phi_{\boldsymbol{k}}\}$ form a complete basis for functions in $\mathcal{L}^2$ (the set of functions that have finite norm with respect to the $\mathcal{L}^2$ inner product). Furthermore, the $\{\phi_{\boldsymbol{k}}\}$ are orthogonal with respect to the $\mathcal{L}^2$ inner product. This is easy to show:

$$\begin{aligned}\langle \phi_{\boldsymbol{k}}, \phi_{\boldsymbol{q}} \rangle &= \int_{\mathbb{R}^3} \overline{\phi_{\boldsymbol{k}}} \phi_{\boldsymbol{q}} \, \mathrm{d}^3x \\ &= \frac{1}{(2\pi)^3} \int_{\mathbb{R}^3} \exp\Big(-i(\boldsymbol{k}-\boldsymbol{q})\cdot\boldsymbol{x}\Big) \, \mathrm{d}^3x \\ &= \delta^3(\boldsymbol{k}-\boldsymbol{q}),\end{aligned} \tag{III.7.4}$$

where the bar indicates a complex conjugation, and $\delta$ is the Dirac delta function. The choice of normalization was made to eliminate the constant factor that would otherwise multiply $\delta$.

Now given a function $f \in \mathcal{L}^2$, it is therefore possible to write $f$ as a linear combination of these basis functions. Mathematically

$$f(\boldsymbol{x}) = \frac{1}{(2\pi)^{3/2}} \int_{\mathbb{R}^3} \hat{f}(\boldsymbol{k}) \exp\Big(i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^3x. \tag{III.7.5}$$

Then we can use the orthogonality of the basis functions to invert this expression and find

$$\hat{f}(\boldsymbol{k}) = \frac{1}{(2\pi)^{3/2}} \int_{\mathbb{R}^3} f(\boldsymbol{x}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^3x. \tag{III.7.6}$$

This new function $\hat{f}(\boldsymbol{k})$ is called the "Fourier transform of $f$", and is essentially a prescription for representing $f$ in the basis of complex exponentials. More generally in $n$ dimensions, the Fourier transform is defined by

$$\hat{f}(\boldsymbol{k}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(\boldsymbol{x}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^nx, \tag{III.7.7}$$

where $\boldsymbol{x}, \boldsymbol{k} \in \mathbb{R}^n$. The inverse Fourier transform is then simply

$$f(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} \hat{f}(\boldsymbol{k}) \exp\left(i\boldsymbol{k}\cdot\boldsymbol{x}\right) \mathrm{d}^n x. \tag{III.7.8}$$

The Fourier transform is a useful concept because it provides frequency information (when given a function of time). For example, the frequency-spectrum for emitted radiation is determined by the Fourier transform of the time-dependent emitted power. Additionally, the Fourier transform of the distribution of galaxies in a redshift-survey provides information about the typical length scale of large-scale structure. This is how baryon-acoustic oscillations (a relic imprinted in the distribution of galaxies associated with sound waves propagating in the pre-recombination universe) were discovered.

### III.7.1 Properties of the Fourier Transform

In the following table, let $f, g \in \mathcal{L}^2$ with Fourier transforms $\hat{f}, \hat{g}$ respectively. Additionally, let $a, b \in \mathbb{C}$, $c \in \mathbb{R}$, and $\boldsymbol{x}_0, \boldsymbol{k}_0 \in \mathbb{R}^n$ be numerical constants.

| Property | Function | Fourier Transform |
|---|---|---|
| Linearity | $af(\boldsymbol{x}) + bg(\boldsymbol{x})$ | $a\hat{f}(\boldsymbol{k}) + b\hat{g}(\boldsymbol{k})$ |
| Translation | $f(\boldsymbol{x} - \boldsymbol{x}_0)$ | $\hat{f}(\boldsymbol{k})e^{-i\boldsymbol{x}_0\cdot\boldsymbol{k}}$ |
| Modulation | $f(\boldsymbol{x})e^{i\boldsymbol{x}\cdot\boldsymbol{k}_0}$ | $\hat{f}(\boldsymbol{k} - \boldsymbol{k}_0)$ |
| Scaling | $f(c\boldsymbol{x})$ | $\frac{1}{\lvert c\rvert}\hat{f}(\boldsymbol{k}/c)$ |
| Conjugation | $\overline{f(\boldsymbol{x})}$ | $\overline{\hat{f}(-\boldsymbol{k})}$ |
| Convolution | $(f * g)(\boldsymbol{x})$ | $\hat{f}(\boldsymbol{k})\hat{g}(\boldsymbol{k})$ |

The final property uses $*$ to denote the convolution of the functions $f$ and $g$. These properties are straightforward applications of the definition of the Fourier transform, but let's prove the final property as an example.

Using the same choice of normalization as for the Fourier transform, the convolution operater is defined by

$$(f * g)(\boldsymbol{x}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(\boldsymbol{y}) g(\boldsymbol{x} - \boldsymbol{y}) \, \mathrm{d}^n y. \tag{III.7.9}$$

Taking the Fourier transform of this expression, we get

$$\frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n \times \mathbb{R}^n} (f * g)(\boldsymbol{x}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^n x = \frac{1}{(2\pi)^n} \iint_{\mathbb{R}^n \times \mathbb{R}^n} f(\boldsymbol{y}) g(\boldsymbol{x} - \boldsymbol{y}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^n x \, \mathrm{d}^n y.$$

Then making the change of variables $\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{z}$ gives us the result we want

$$\begin{aligned}
\frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n \times \mathbb{R}^n} (f * g)(\boldsymbol{x}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{x}\Big) \, \mathrm{d}^n x &= \left(\frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} f(\boldsymbol{y}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{y}\Big) \, \mathrm{d}^n y\right) \\
&\quad \times \left(\frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} g(\boldsymbol{z}) \exp\Big(-i\boldsymbol{k}\cdot\boldsymbol{z}\Big) \, \mathrm{d}^n z\right) \\
&= \hat{f}(\boldsymbol{k})\hat{g}(\boldsymbol{k}).
\end{aligned}$$

### III.7.2 The Discrete Fourier Transform

Imagine now that we do not have complete knowledge of the function $f$, whose Fourier transform we would like to know. Instead we know the value of $f$ sampled on a regular grid of $N$ points with grid spacing $\Delta x$ such that $x_j = j\Delta x$. Mathematically, we measure

$$g(x) = f(x)s(x), \tag{III.7.10}$$

where $s$ is defined by

$$s(x) \equiv \sum_j \delta(x - x_j). \tag{III.7.11}$$

Plugging this expression for $g$ into Equation III.7.7 tells us that the Fourier transform of $g$ is

$$\hat{g}(k) = \frac{1}{\sqrt{2\pi}} \sum_j f(x_i) \exp\Big( - ikx_j \Big). \tag{III.7.12}$$

This is the discrete Fourier transform (DFT). However, by the convolution theorem, the Fourier transform of $f(x)s(x)$ is $\hat{f}(k) * \hat{s}(k)$, where $\hat{f}$ and $\hat{s}$ are the Fourier transforms of $f$ and $s$ respectively.

Figure III.7.1 shows an example function $f$ with its Fourier transform. Notice that $\hat{f}$ is support only on a finite interval $(A, B)$. Figure III.7.2 shows the sampling function $s$ with its Fourier transform. The Fourier transform of $s$ is

$$\begin{aligned}\hat{s}(k) &= \frac{1}{\sqrt{2\pi}} \sum_j \exp\Big( - ikx_j \Big) \\ &= \frac{1}{\sqrt{2\pi}} \sum_j \exp\Big( - ijk\Delta x \Big).\end{aligned}$$

In the limit that $N \to \infty$ it can be shown that this is a Fourier series that converges to

$$\hat{s}(k) = \frac{1}{\sqrt{2\pi}} \sum_j \delta\left( k - j\frac{2\pi}{\Delta x} \right). \tag{III.7.13}$$

Finally, Figure III.7.3 shows the sampled function $g(x) = f(x)s(x)$ and its Fourier transform. $\hat{g}$ is the convolution of $\hat{f}$ with $\hat{s}$. Notice that because $\hat{f}$ is narrow enough, the convolution simply creates several copies of $\hat{f}$. However, if

$$|B - A| > \frac{1}{2}\frac{2\pi}{\Delta x} \tag{III.7.14}$$

then the copies of $\hat{f}$ will begin to overlap. Therefore, the DFT can only represent functions with

$$|B - A| < \frac{1}{2}\frac{2\pi}{\Delta x} \equiv k_{\text{Nyquist}}, \tag{III.7.15}$$

where $k_{\text{Nyquist}}$ is called the Nyquist frequency.

Now return to equation III.7.12. In principle, we can choose $k$ to be any value. However, because there are $N$ independent values of $x$, we should expect that the DFT will output the Fourier transform at $N$ independent values of $k$. In fact, if the DFT is evaluated on a uniform grid of points in $k$-space, the exact result can be recovered at all $k$ using sinc interpolation. Therefore

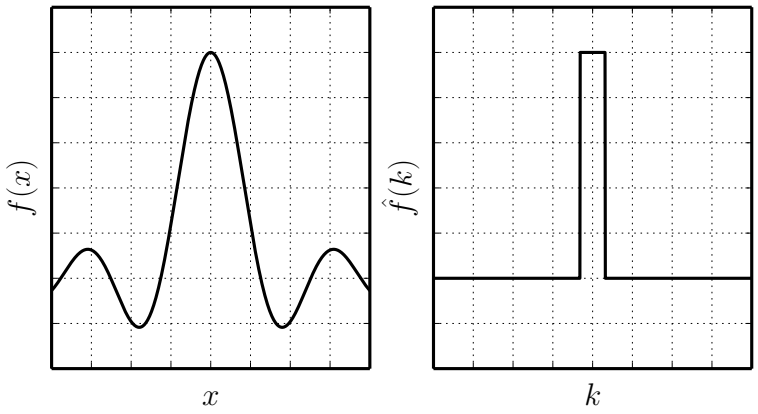



Figure III.7.1: An example function $f$ (left) with its Fourier transform (right).

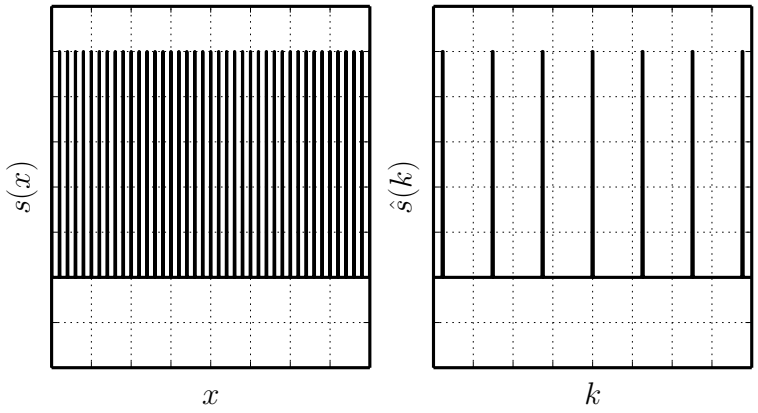



Figure III.7.2: The sampling function $s$ (left) with its Fourier transform (right).

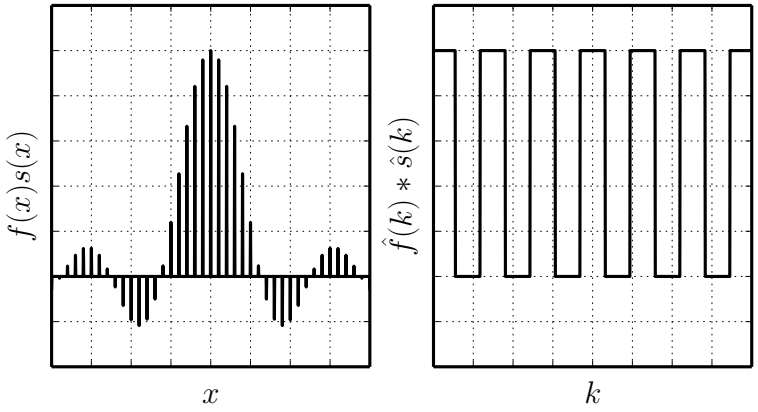



Figure III.7.3: The sampled function $g = fs$ (left) with its Fourier transform (right).

we want to evaluate the DFT on a uniform grid of $N$ points running from $-k_{\text{Nyquist}}$ to $k_{\text{Nyquist}}$. By convention, define $k_j = j\Delta k$, $\Delta k = 2\pi/N\Delta x$, and $j$ runs from $-N/2$ to $N/2-1$ if $N$ is even, or from $-(N-1)/2$ to $(N-1)/2$ if $N$ is odd.

Switching to the conventions used by most software packages, the DFT is given by

$$\boxed{\hat{f}(j\Delta k) = \sum_{j'=0}^{N-1} f\left(j'\Delta x\right) \exp\left(-2\pi i \frac{jj'}{N}\right).} \tag{III.7.16}$$

Additionally note that most software packages put $j = 0$ as the first element of their output. Use the function `numpy.fftshift` to put it back in the middle. When written in this way, the DFT can be easily written in matrix notation as

$$\begin{pmatrix} \vdots \\ \hat{f}(j\Delta k) \\ \vdots \end{pmatrix} = \begin{pmatrix} & \vdots & \\ \cdots & \exp\left(-2\pi i \frac{jj'}{N}\right) & \cdots \\ & \vdots & \end{pmatrix} \begin{pmatrix} \vdots \\ f\left(j'\Delta x\right) \\ \vdots \end{pmatrix}. \tag{III.7.17}$$

The cost of constructing this matrix and performing the matrix-vector multiplication scales as $\mathcal{O}(N^2)$. However, by using some symmetries of the DFT, it is possible to obtain a further speed-up to $\mathcal{O}(N \log_2 N)$. These algorithms are referred to as fast Fourier transform (FFT) algorithms.

### III.7.3 The Fast Fourier Transform

Suppose $N$ is a power of 2. That is, $N = 2^m$ for some positive integer $m$. Then we can write the DFT as

$$\begin{aligned}
\hat{f}(j\Delta k) &= \sum_{\text{even } j'} f\left(j'\Delta x\right) \exp\left(-2\pi i \frac{jj'}{N}\right) + \sum_{\text{odd } j'} f\left(j'\Delta x\right) \exp\left(-2\pi i \frac{jj'}{N}\right) \\
&= \sum_{n=0}^{N/2-1} f\left(2n\Delta x\right) \exp\left(-2\pi i \frac{2jn}{N}\right) + \sum_{n=0}^{N/2-1} f\left((2n+1)\Delta x\right) \exp\left(-2\pi i \frac{2jn+j}{N}\right) \\
&= \sum_{n=0}^{N/2-1} f\left(2n\Delta x\right) \exp\left(-2\pi i \frac{jn}{N/2}\right) + e^{-2\pi i j/N} \sum_{n=0}^{N/2-1} f\left((2n+1)\Delta x\right) \exp\left(-2\pi i \frac{jn}{N/2}\right).
\end{aligned}$$

The key is now to notice that we have written a single length-$N$ DFT in terms of two length-$N/2$ DFTs. This observation forms the basis for the Cooley-Tukey FFT:

$$\text{FFT}(f) = \text{FFT}(\text{even samples of } f) + \text{complex exponential} \times \text{FFT}(\text{odd samples of } f). \tag{III.7.18}$$

A length-1 DFT is trivial to compute, so this formula is applied recursively until only length-1 DFTs are computed. There are $\log_2 N$ levels of recursion because the length of the FFT is halved each time, but each level requires $\mathcal{O}(N)$ operations, so the overall complexity of this algorithm is $\mathcal{O}(N \log_2 N)$.

# III.8 Monte Carlo Methods

Monte Carlo (MC) methods allow the solution of problems using random numbers in combination with known probabilities of potential outcomes.

This section is based on Joachim Puls's (LMU Observatory, Munich) lecture notes on Computational Methods in Astrophysics, 2005. A useful book on MC methods is that of Liu [1].

## III.8.1 Review of Probability Basics

Let us review some basics of probability theory.

### III.8.1.1 Probability: The Concept

If we are performing an experiment with possible outcomes $E_k$, the probability of outcome $E_k$ is given by

$$P(E_k) = p_k \ , \tag{III.8.1}$$

where

(i) $0 \leq p_k \leq 1$,

(ii) If $E_k$ cannot be realized, then $P_k = 0$,
if $E_k$ is the only outcome, then $P_k = 1$.

(iii) If two outcomes $E_i$ and $E_j$ are mutually exclusive, then $P(E_i \text{ and } E_j) = 0$,
but $P(E_i \text{ or } E_j) = P_i + P_j$.

(iv) If there are $N$ mutually exclusive outcomes $E_i$, $i = 1, \cdots, N$, and these events are complete concerning all possible outcomes of the experiment, then

$$\sum_{i=1}^{N} P_i = 1 \ . \tag{III.8.2}$$

(v) In a two-stage experiment in which outcomes $F_i$ and $G_j$ occur after each other and are independent of eath other, $E_{ij}$ is called the combined outcome and

$$P(E_{ij}) = P_{ij} = P(G_j)P(F_i) \ . \tag{III.8.3}$$

### III.8.1.2 Random Variables

We associate each discrete outcome $E_k$ with a real number $x_k$, the *random variable*. For example, when throwing a dice, we associate the "six" outcome with $x_6 = 6$.

The **expectation value** of a random variable $x$ is

$$E(x) = \overline{x} = \sum_{i=1}^{N} x_i P_i \ , \tag{III.8.4}$$

if the events $E_k$, $k = 1, \cdots, N$ are complete and mutually exclusive.

Any arbitrary function $y(x)$ of a random variable $x$ is also a random variable and

$$E(y(x)) = \overline{y} = \sum_{i=1}^{N} y(x_i) P_i \ . \tag{III.8.5}$$

The expectation value is a linear operator, so

$$E(ay(x) + bh(x)) = aE(y(x)) + bE(h(x)) \ . \tag{III.8.6}$$

The **variance**,

$$V(x) = \overline{(x - \overline{x})^2} = \overline{x^2 - 2x\overline{x} + \overline{x}^2} = \overline{x^2} - \overline{x}^2 \ , \tag{III.8.7}$$

is the mean quadratic deviation from the mean. The **standard deviation** is the square root of the Variance,

$$\sigma(x) = (V(x))^{1/2} \ . \tag{III.8.8}$$

$V(x)$ is not a linear operator, but if $g$ and $h$ are statistically indepedent, $\overline{gh} = E(gh) = E(g)E(h) = \overline{g}\overline{h}$, then

$$V(ag + bh) = a^2 V(g) + b^2 V(h) \ , \tag{III.8.9}$$

where $a$ and $b$ are constants.

#### III.8.1.3 Continuous Random Variables

So far, we have considered only discrete events. In real life, the probability of a discrete event (i.e., a specific outcome) may be near or practically/totally zero, e.g., the occurence of a specific scattering angle (there are infinitely many!) in a scattering experiment is zero for all practical purposes. However, the probability that the result lies inside a specific interval is well defined:

$$P(x \le x' \le x + dx) = p(x)dx \ , \tag{III.8.10}$$

where $p(x)$ is a *probability distribution function* (PDF).

The probability that the outcome of the experiment lies in the interval $[a, b]$ is the calculated from the PDF:

$$P(a \le x \le b) = \int_a^b p(x')dx' \ , \tag{III.8.11}$$

with the constraints

$$\begin{aligned} &\text{(i)} \ \ p(x) \ge 0 \ \ \text{ in } \ \ -\infty < x < \infty \ , \\ &\text{(ii)} \ \ \int_{-\infty}^{\infty} p(x')dx' = 1 \ . \end{aligned} \tag{III.8.12}$$

The **cumulative PDF** is the probability that all events up to a certain threshold $x$ will be realized,

$$P(x' \le x) = F(x) = \int_{-\infty}^{x} p(x')dx' \ . \tag{III.8.13}$$

$F(x)$ increases monotonically and $F(-\infty) = 0$, $F(\infty) = 1$.

An important example of a PDF is the *PDF of a uniform distribution* in $[a,b]$, $p(x) = const. = c$ in $[a,b]$. This means that the probability that any value $x'$ in $x \le x' \le x + dx$ is realized is identical for all $x$ in $[a,b]$. In other regions $p(x) = 0$. For such a probability distribution function, we have

$$F(\infty) = 1 = \int_{-\infty}^{\infty} p(x)dx = \underbrace{\int_{-\infty}^{a} p(x)dx}_{=0} + \int_a^b p(x)dx + \underbrace{\int_b^{\infty} p(x)dx}_{=0} \ . \tag{III.8.14}$$

Hence,

$$\int_a^b p(x)dx = c(b-a) = 1 \longrightarrow p(x) = \frac{1}{b-a} \ . \tag{III.8.15}$$

**Expectation Value and Variance for Continuum Random Variables**

$$\begin{aligned} E(x) = \overline{x} &= \int_{-\infty}^{\infty} x' p(x')dx' \ , \\ E(g(x)) = \overline{g} &= \int_{-\infty}^{\infty} g(x')p(x')dx' \ , \\ V(x) = \sigma^2(x) &= \int_{-\infty}^{\infty} (x' - \overline{x})p(x')dx' \ , \\ V(g(x)) = \sigma^2(g) &= \int_{-\infty}^{\infty} (g(x') - \overline{g})p(x')dx' \ . \end{aligned} \tag{III.8.16}$$

## III.8.2 Random Numbers

A key prerequesite for MC methods is the availability of a reliable random numbers. An algorithm that generates random numbers is called a random number generator (RNG).

There are three different kinds of random numbers:

(1) *Real random numbers*. These come, e.g., from radioactive decay.

(2) *Pseudo-random numbers* are generated from a deterministic algorithm. These are the random numbers one uses in most computer applications of MC.

(3) *Sub-random numbers* (also called *quasi-random numbers*) are numbers that have an "optimal" uniform distribution, but that are not completely independent of each other.

### III.8.2.1 Generating Pseudo-Random Numbers

In many situations random number generators can be used as black boxes. It is nevertheless useful to have a general understanding of how they work.

The requirements that a good random number generator must satisfy are as follows:

- The random numbers must be uniformly distributed. In the interval $[a,b]$ any value $x$ in $[a,b]$ must be equally probable to obtain. So $p(x) = 1/(b-a)$ and

$$F(x) = \int_a^x p(x')dx' = \frac{x-a}{b-a} \ . \tag{III.8.17}$$

  On the interval $[0,1]$ we have $F(x) = x$.

- The generated random numbers must be statistically independent.
- The sequence of generated random numbers should not be periodic or at least have a very long period.
- The sequence of generated random numbers must be reproducible.
- The algorithm must be portable (i.e., work on many different architectures).
- The algorithm generating the pseudo-random numbers must be fast.

A relatively simple example of a random number generator is the so-called *Linear Congruental* RNG. Its basic algorithm goes as follows.

```
SEED = (A * SEED + c) mod M
   X = SEED/M
```

Here `M`, `A`, `c`, `SEED` are integers and `x` is a real number. For any given `SEED` value, a new `SEED` value is calculated. From this value, a second random number is created and so on. Since `SEED` is always an integer `mod` M, we have $0 \leq$ `SEED` $<$ `M`, which means that `x` will be in $[0, 1)$.

The maximum possible value of `SEED` is `M` $-1$. A new cycle in the sequence starts whenever `SEED` = `SEED`(initial). So, ideally, one would want to make `M` very large, e.g. pick it as the largest possible integer. However, this may lead to portability issues, since the largest possible integer varies between systems. An optimal algorithm would produce all possible integers in $0, \cdots, M-1$.

Here is an example: $A = 5, c = 1, M = 2^n, n = 5$, `SEED` $= 9$. With this we get

$$9, 14, 7, 4, 21, 10, ... \tag{III.8.18}$$

### III.8.3 Monte Carlo Integration

As a first application of MC methods we will discuss MC integration methods.

#### III.8.3.1 Preliminaries

We will draw $N$ random variables $x_1, \cdots, x_N$ from a given PDF $p(x)$. From these variables we compute a new random variable

$$G = \frac{1}{N} \sum_{i=1}^{N} g(x_i) \ , \tag{III.8.19}$$

where $g(x_i)$ be an arbitrary function of a random variable $x_i$ and, thus a random variable itself. The expectation value of $G$ is

$$E(G) = \overline{G} = \frac{1}{N} \sum_{i=1}^{N} E(g(x_i)) = \frac{1}{N} N \overline{g} = \overline{g} \ , \tag{III.8.20}$$

so $\overline{G} = \overline{g}$. The variance of $G$ is

$$V(G) = V\left( \frac{1}{N} \sum_{i=1}^{N} g(x_i) \right) = \frac{1}{N^2} \sum_{i=1}^{N} V(g(x_i)) = \frac{1}{N} V(g) \ . \tag{III.8.21}$$

In words: $G$, besides being a random variable, is also the arithmetic mean of the drawn random variables $g(x_i)$. The expectation value $\overline{G}$ of the arithmetic mean $G$ is nothing else than the expectation value of $g$ itself, independent of $N$. So $\overline{g} = \overline{G} \approx G$, if $N$ is sufficiently large, converging with $1/\sqrt{N}$, since $\sqrt{V}$ is reduced at this rate.

### III.8.3.2 MC Integration – The Formal Method

Given the integral $I$,

$$I = \int_a^b g(x)dx = \int_a^b \frac{g(x)}{p(x)} p(x)dx = \int_a^b h(x)p(x)dx \ , \tag{III.8.22}$$

we now demand that $p(x)$ shall be a PDF. Any integral can now be interpreted as the expectation value of $h(x)$ with respect to $p(x)$.

If $p(x)$ is a uniform PDF, then we have

$$p(x) = \begin{cases} \dfrac{1}{b-a} & a \le x \le b \\ 0 & \text{elsewhere} \end{cases} \ , \tag{III.8.23}$$

so

$$h(x) = \frac{g(x)}{p(x)} = (b-a)g(x) \ , \tag{III.8.24}$$

and

$$I = (b-a)\int_a^b g(x)p(x)dx = (b-a)\overline{g}. \tag{III.8.25}$$

Hence, we can approximate the integral by computing

$$G = \frac{1}{N}\sum_{i=1}^{N} g(x_i) \ , \tag{III.8.26}$$

where the $x_i$ are random numbers in $[a, b]$. With this, we have

$$I = (b-a)\overline{g} = (b-a)\overline{G} \overset{MC}{\approx} (b-a)G \ . \tag{III.8.27}$$

The MC integration error can be estimated from the variance

$$V(G) = \frac{1}{N}V(g) = \frac{1}{N}\left( \underbrace{\overline{g^2}}_{=J} - \underbrace{\overline{g}^2}_{=\overline{G}^2} \right) \overset{MC}{\approx} \frac{1}{N}(J - G^2) \ , \tag{III.8.28}$$

where

$$J = \frac{1}{N}\sum_{i=1}^{N} g^2(x_i) \ . \tag{III.8.29}$$

So the error $\delta I$ is proportional to $1/\sqrt{N}$.

**Example:** $g(x) = x^2$ on the interval $[0, 2]$.

$$I = \int_0^2 g(x)dx \approx \frac{2}{N}\sum_{i=1}^{N}(2x_i)^2 \ , \tag{III.8.30}$$

where the $x_i$ are randomly drawn from $[0, 1]$.

#### III.8.3.3 When to apply MC Integration

The error of the integral $I$ when evaluated via the MC method converges with $N^{-1/2}$, independent of the dimensionality of the integral. To see when it is appropriate to use MC, we compare the rate of convergence of MC with those of other methods:

| Method | $\delta I$ |
|---|---|
| Trapezoidal Rule | $\propto N^{-2/d}$ |
| Simpson's Rule | $\propto N^{-4/d}$ |

Here, $d$ is the dimensionality of the integration problem. Based on the above, MC becomes more efficient at $d \geq 5$ when compared to the trapezoidal rule and $d \geq 9$ when compared to Simpson's rule. This suggests that MC will be best to use for integration in many-body systems and in many-dimensional parameter spaces. The latter is of particular importance in Bayesian statistical analysis in which MC is broadly used to compute estimates of the marginalization integral.

Despite MC's poor convergence rate at low dimensionality it can be advantageous to apply it even there, since

(1) it's trivial to use,

(2) one does not have to worry about complex domain boundaries,

(3) it can be sued for quick, rough estimates with small $N$.

#### III.8.3.4 Variance Reduction / Importance Sampling

The error in MC integration is proportional to the variance of the integrand $g$ over the set of points chosen randomly from a domain $V$. If the integrand $g$ is constant (i.e., has zero variance), then only one sample is needed to find the correct integral. On the other hand, if $g$ varies strongly, many random samples are needed, which can get very computationally expensive.

On possible approach to overcome this problem is to replace the integrand with an integrand that has less variation. So, for example,

$$I = \int_0^1 g(x)dx = \int_0^1 \frac{g(x)}{w(x)} w(x)dx \ . \tag{III.8.31}$$

Assuming that $w(x)$ has similar behavior as $g(x)$, the ratio $g(x)/w(x)$ will be approximately constant (i.e., will have small variance). If we can now find a new integration variable $y(x)$ so that $dy/dx = w(x)$, we can rewrite the integral as

$$I' = \int_{y_o}^{y_1} \frac{g(x(y))}{w(x(y))} dy \ , \tag{III.8.32}$$

where $x(y_0) = 0$ and $x(y_1) = 1$. We can then use the standard MC integration technique to obtain $I'(N)$ with $V(I'(N)) < V(I(N))$.

### III.8.4 Monte Carlo Simulation

MC Simulations are virtual experiments that can be applied to a large variety of problems. IN general, a MC simulation will proceed as follows:

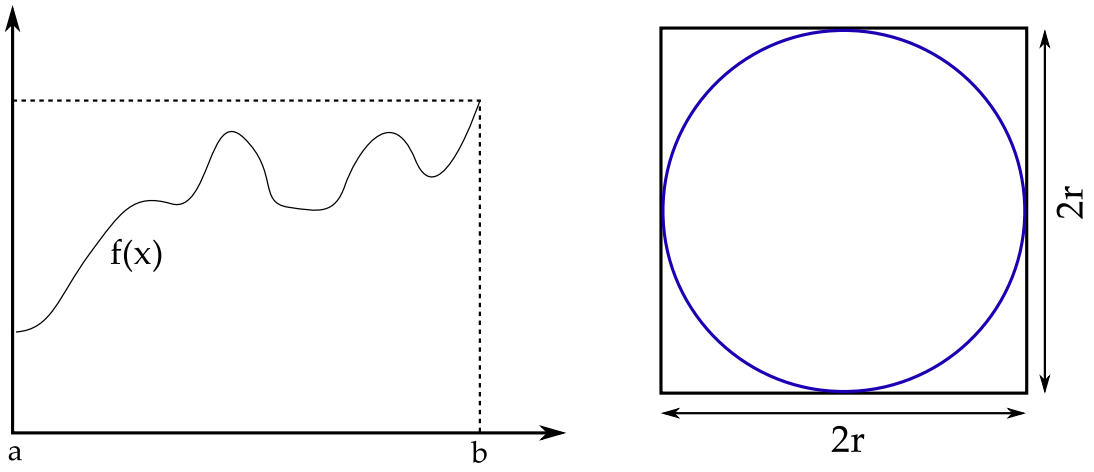



Figure III.8.1: Examples of a MC Simulations. Left: Hit-or-Miss Integration of the function $f(x)$ on the interval $[a,b]$. Right: Measurement of $\pi$.

(1) Define a domain of possible inputs.

(2) Generate inputs randomly from the domain on the basis of a specified/chosen PDF.

(3) Perform a deterministic calculation using the inputs.

(4) Aggregate the results of the individual calculations into the final result.

MC simulations are being used in many areas of physics and astrophysics to carry out simulations of physical processes and to simulate real-life experimental outcomes on the basis of theoretical knowledge of potential outcomes. The power of MC simulations is best demonstrated by example.

**Example:** The Hit or Miss Method for MC Integration

Consider the complicated function $f(x)$ on the interval $[a,b]$ in the left panel of Fig. III.8.1. We can easily find the area under the curve using the following MC simulation:

(1) Domain: $[a,b] \rightarrow [f(a),f(b)]$. The total area is broken down into $A_1 = (b-a) \times f(b)$ (the big box) and $A_2 = (b-a) \times (f(b) - f(a))$, the area above $f(a)$.

(2) PDF: uniform in $A_2$. Draw $N$ points $(x_i, y_i)$ in $A_2$.

(3) Deterministic calculation: If for $(x_i, y_i)$, $y_i \leq f(x_i)$, then increment the counter $n$ by one, $n = n+1$

(4) Result: $I = \int_a^b f(x)dx \approx A_2 \frac{n}{N} + (A_1 - A_2)$

**Example:** Measuring $\pi$

Consider the circle in the right panel of Fig. III.8.1. It's area is $A_{\rm circ} = \pi r^2$ while that of the box enclosing it is $A_{\rm box} = 4r^2$, thus $A_{\rm circ}/A_{\rm box} = \pi/4$. We can experimentally determine $\pi$ by the following MC simulation:

(1) Domain: Box

(2) PDF: uniform, draw $N$ points $(x_i, y_i)$ in the box.

(3) Deterministic calculation: If $x_i^2 + y_i^2 \leq r^2$, then increment counter $n$ by one: $n = n + 1$.

(4) Result: $\pi = 4n/N$.

## References

[1] J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer, New York, NY, USA, 2001.

# III.9 Ordinary Differential Equations (Part I)

A system of first-order ordinary differential equations (ODEs) is a relationship between an unknown (vectorial) function $\vec{y}(\vec{x})$ and its derivative $\vec{y}\,'(\vec{x})$. The general system of first-order ODEs has the form

$$\vec{y}\,'(x) = \vec{f}(\vec{x}, \vec{y}(\vec{x}))\,. \tag{III.9.1}$$

To make life easier, we will drop the vector $\vec{}$ notation in the following and just keep in mind that the symbols are vectors if more than one equation is to be solved.

A solution to the differential equation (III.9.1) is, obviously, any function $y(x)$ that satisfies it.

There are two general classes of first-order ODE problems:

(1) Initial value problems: $y(x_i)$ is given at some starting point $x_i$.

(2) Two-point boundary value problems: $y$ is known at two ends ("boundaries") of the domain and these "boundary conditions" must be satisfied simultaneously.

## III.9.1 Reduction to First-Order ODE

Any ODE can be reduced to first-order form by introducing additional variables. Here is an example:

$$y''(x) + q(x)y'(x) = r(x)\ . \tag{III.9.2}$$

We reduce this ODE to first-order form by introducing a new function $z(x)$ and solve for

$$\begin{aligned} &(1)\quad y'(x) = z(x)\,,\\ &(2)\quad z'(x) = r(x) - q(x)z(x)\ . \end{aligned}$$

## III.9.2 ODEs and Errors

For studying the kinds of errors we have to deal with when working with ODEs, let us restrict ourselves to initial value problems. All procedures to solve numerically such an ODE consist of transforming a continuous differential equation into a discrete iteration procedure that starts from the initial conditions and returns the values of the dependent variable $y(x)$ at points $x_m = x_0 + m * h$, where $h$ is the discretization step size (which we have assumed to be constant here).

Two kinds of errors can arise in this procedure:

(1) Round-off error: Due to limited FP accuracy. The global round-off is the sum of the local FP errors.

(2) **Truncation error**.

Local: The error made in one step when we replace a continuous process (e.g. a derivative) with a discrete one (e.g., a forward difference).

Global: If the local truncation error is $\mathcal{O}(h^{n+1})$, then the global truncation error must be $\mathcal{O}(h^n)$, since the number of steps used in evaluating the derivatives to reach an arbitrary point $x_f$, having started at $x_0$, is $\frac{x_f - x_0}{h}$.

### III.9.3 Euler's Method

We want to solve $y' = f(x, y)$ with $y(x_0) = y_0$. We introduce a fixed stepsize $h$ and we first obtain an estimate of $y(x)$ at $x_1 = x_0 + h$ using Taylor's theorem:

$$\begin{aligned} y(x_1) = y(x_0 + h) &= y(x_0) + y'(x_0)h + \mathcal{O}(h^2)\,, \\ &= y(x_0) + hf(x_0, y(x_0)) + \mathcal{O}(h^2)\,. \end{aligned} \tag{III.9.3}$$

By analogy, we obtain that the value $y_{n+1}$ of the function at the point $x_{n+1} = x_0 + (n+1)h$ is given by

$$y_{n+1} = y(x_{n+1}) = y_n + hf(x_n, y(x_n)) + \mathcal{O}(h^2)\,. \tag{III.9.4}$$

This is called the *forward Euler Method*. It is obviously extremely simple, but rather inaccurate and potentially unstable. Note that the error scales $\propto h^2$ locally. However, if $L$ is the length of the domain, then $h = L/N$, where $N$ is the number of points used to cover it. Since we are taking $N$ integration steps, the global error is $\propto Nh^2 = NL^2/N^2 = LL/N \propto h$. Hence, forward Euler is a first-order accurate method.

#### III.9.3.1 Stability of Forward Euler

Forward Euler is an *explicit* method. This means that $y_{n+1}$ is given explicitly in terms of known quantities $y_n$ and $f(x_n, y_n)$.

Explicit methods are simple and efficient, but the drawback is that the step size must be small for stability. Example:

$$y' = -ay\,, \qquad \text{with} \quad y(0) = 1\,,\ a > 0\,,\ y' = \frac{dy}{dt}\,. \tag{III.9.5}$$

As we know, the exact solution to this problem is $y^{\rm ex} = \exp(-at)$, which is stable & smooth with $y^{\rm ex}(0) = 1$ and $y^{\rm ex}(\infty) = 0$.

Now applying forward Euler:

$$y_{n+1} = y_n - a\,h\,y_n = (1 - ah)^2 y_{n-1} = \cdots = (1 - ah)^{n+1} y_0\,. \tag{III.9.6}$$

This implies that in order to prevent any potential amplification of errors, we must require that $|1 - ah| < 1$. In fact, we can decide between 3 cases:

(i) $0 < 1 - ah < 1$ : $(1 - ah)^{n+1}$ decays (good!).
(ii) $-1 < 1 - ah < 0$ : $(1 - ah)^{n+1}$ oscillates (not so good!).
(iii) $1 - ah < -1$ : $(1 - ah)^{n+1}$ oscillates and diverges (bad!).

With this, we arrive at an overall stability criterion of $h < 2/a$. This means that forward Euler is unstable overall, but conditionally stable if $h < 2/a$.

### III.9.4 Backward Euler

If we use

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})\,, \tag{III.9.7}$$

we get what is called the *backward Euler Method*, and *implicit* method. Implicit, because $y_{n+1}$ depends on unknown quantities, i.e. on itself!

For our toy problem of the previous section III.9.3.1, we get with backward Euler

$$\begin{aligned} y_{n+1} &= y_n + h(-a y_{n+1}) \ , \\ y_{n+1} &= \frac{1}{1+ha}\, y_n \ . \end{aligned} \tag{III.9.8}$$

Since $ha > 0$, $(1+ha)^{-1} < 1$ and the solution decays. This means that backward Euler is *unconditinally stable* for all $h$! But do not forget that the error could still be large; it will just not wreck the scheme by leading to blow up.

### III.9.5 Predictor-Corrector Method

We can improve upon the forward Euler method in many ways. One way is to try

$$y_{n+1} = y_n + h\,\frac{f(x_n, y_n) + f(x_{n+1}, y_{n+1})}{2} \ , \tag{III.9.9}$$

which will be a better estimate as it is using the "average slope" of $y$. However, we don't know $y_{n+1}$ yet.

We can get around this problem by using forward Euler to estimate $y_{n+1}$ and then use Eq. (III.9.9) for a better estimate:

$$\begin{aligned} y^{(\mathrm{P})}_{n+1} &= y_n + h f(x_n, y_n) \ , && \text{(predictor)} \\ y_{n+1} &= y_n + \frac{h}{2}\left[ f(x_n, y_n) + f(x_{n+1}, y^{(\mathrm{P})}_{n+1}) \right] \ . && \text{(corrector)} \end{aligned} \tag{III.9.10}$$

One can show that the error of the predictor-corrector method decreases locally with $h^3$, but globally with $h^2$. One says it is *second-order accurate* as opposed to the Euler method, which is first-order accurate. The predictor-corrector method is also considerably more stable than the Euler method, but we spare the reader the proof.

### III.9.6 Runge-Kutta Methods

The idea behind Runge-Kutta (RK) methos is to match the Taylor expansion of $y(x)$ at $x = x_n$ up to the highest possible and convenient order.

As an example we shall consider derivation of a second order RK method (RK2). For

$$\frac{dy}{dx} = f(x, y) \ , \tag{III.9.11}$$

we have

$$y_{n+1} = y_n + a k_1 + b k_2 \ , \tag{III.9.12}$$

with

$$\begin{aligned} k_1 &= h\, f(x_n, y_n) \ , \\ k_2 &= h\, f(x_n + \alpha h, y_n + \beta k_1) \ . \end{aligned} \tag{III.9.13}$$

We now fix the four parameters $a, b, \alpha, \beta$ so that Eq. (III.9.12) agrees as well as possible with the Taylor series expansion of $y' = f(x, y)$:

$$\begin{aligned} y_{n+1} &= y_n + hy'_n + \frac{h^2}{2}y''_n + \mathcal{O}(h^3) \ , \\ &= y_n + hf(x_n, y_n) + \frac{h^2}{2}\frac{d}{dx}f(x_n, y_n) + \mathcal{O}(h^3) \ , \\ &= y_n + hf_n + h^2\frac{1}{2}\left(\frac{\partial f_n}{\partial x} + \frac{\partial f_n}{\partial y}f_n\right) + \mathcal{O}(h^3) \ , \end{aligned} \tag{III.9.14}$$

where we have used $f_n = f(x_n, y_n)$.

On the other hand, we have, using Eq. (III.9.12),

$$y_{n+1} = y_n + ahf_n + bhf(x_n + \alpha h, y_n + \beta h f_n) \ . \tag{III.9.15}$$

Now we expand the last term of Eq. (III.9.15) in a Taylor series to first order in terms of $(x_n, y_n)$,

$$y_{n+1} = y_n + ahf_n + bh\left[f_n + \frac{\partial f}{\partial x}(x_n, y_n)\alpha h + \frac{\partial f}{\partial y}(x_n, y_n)\beta h f_n\right] \ , \tag{III.9.16}$$

and can now compare this with Eq. (III.9.12) to read off:

$$a + b = 1 \ , \qquad \alpha b = \frac{1}{2} \qquad \beta b = \frac{1}{2} \ . \tag{III.9.17}$$

So there are only 3 equations for 4 unknowns and we can assign an arbitrary value to one of the unknowns. Typical choices are:

$$\alpha = \beta = \frac{1}{2} \ , \qquad a = 0 \ , \qquad b = 1 \ . \tag{III.9.18}$$

With this, we have for RK2:

$$k_1 = hf(x_n, y_n) \ , \tag{III.9.19}$$

$$k_2 = hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \ , \tag{III.9.20}$$

$$y_{n+1} = y_n + k_2 + \mathcal{O}(h^3) \ . \tag{III.9.21}$$

Note that this method is locally $\mathcal{O}(h^3)$, but globally only $\mathcal{O}(h^2)$ (see §III.9.2). Also note that for $a = b = 1/2$ and $\alpha = \beta = 1$ we recover the predictor-corrector method!

## III.9.7 Other RK Integrators

### III.9.7.1 RK3

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1) \ , \\ k_3 &= hf(x_n + h, y_n - k_1 + 2k_2) \ , \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 4k_2 + k_3) + \mathcal{O}(h^4) \ . \end{aligned} \tag{III.9.22}$$

#### III.9.7.2 RK4

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \ , \qquad \text{(III.9.23)}\\
k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_1) \ ,\\
k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{1}{2}k_2) \ ,\\
k_4 &= hf(x_n + h, y_n + k_3) \ ,\\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + \mathcal{O}(h^5) \ . \qquad \text{(III.9.24)}
\end{aligned}
$$

#### III.9.7.3 Implementation Hint

When implementing RK integration, write a subroutine that evaluates

$$\frac{dy}{dx} = f(x, y) \ , \qquad \text{(III.9.25)}$$

for a given $x$ and $y$, since you will call it many times (with different arguments). $f(x, y)$ in the above equation is usually referred to as the *right-hand side* (RHS) of the ODE problem.

### III.9.8 Runge-Kutta Methods with Adaptive Step Size

*This section of the lecture notes has been contributed by Mark Scheel*, `scheel@tapir.caltech.edu`.

The RK methods we have discussed thus far require choosing a fixed step size $h$. How should one choose $h$? How does one know if one is making the right choice?

Better would be to choose an *error tolerance* and have $h$ be chosen automatically to satisfy this error tolerance. To do this, we need:

(1) A method for estimating error.

(2) A way to adjust the stepsize $h$, if the error is too large/small.

#### III.9.8.1 Embedded Runge-Kutta Formulae

Embedded RK formulae provide an error estimator for (almost) free.

The simplest example (not used in practice) is the following:

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \ ,\\
k_2 &= hf(x_n + h, y_n + k_1) \ ,\\
y_{n+1} &= y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2 \qquad &+\mathcal{O}(h^3) \ , \quad &\text{“2nd order global error”; predictor-corrector}\\
y^*_{n+1} &= y_n + k_1 \qquad &+\mathcal{O}(h^2) \ , \quad &\text{“1st order global error”; forward Euler} \qquad \text{(III.9.26)}
\end{aligned}
$$

So one formula gives two approximations to $y_{n+1}$: 2nd order ($y_{n+1}$) and 1st order ($y^*_{n+1}$). For updating $y$, we would of course use $y_{n+1}$, but the error can be estimated by $\delta y_{n+1} = y_{n+1} - y^*_{n+1} = \mathcal{O}(h^2)$.

### III.9.8.2 Bogaki-Shampine Embedded Runge-Kutta

Bogaki and Shampine developed the following useful 2nd/3rd order embedded RK scheme.

$$\begin{aligned} k_1 &= hf(x_n, y_n) \ , \\ k_2 &= hf(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \ , \\ k_3 &= hf(x_n + \frac{3}{4}h, y_n + \frac{3}{4}k_2) \ , \\ y_{n+1} &= y_n + \frac{2}{9}k_1 + \frac{1}{3}k_2 + \frac{4}{9}k_3 + \mathcal{O}(h^4) \\ k_4 &= hf(x_n + h, y_{n+1}) \\ y^*_{n+1} &= y_n + \frac{7}{24}k_1 + \frac{1}{4}k_2 + \frac{1}{3}k_3 + \frac{1}{8}k_4 + \mathcal{O}(h^3) \ . \end{aligned} \tag{III.9.27}$$

The error is then again

$$\delta y_{n+1} = y_{n+1} - y^*_{n+1} \ . \tag{III.9.28}$$

Note that $k_4$ of step $n$ is the same as $k_1$ of step $n+1$. So $k_1$ does not need to be recomputed on step $n+1$; simply save $k_4$ and re-use it on the next step. This trick is called FSAL, "first same as last."

There exist embedded Runge-Kutta formulae for other orders, e.g. 4th/5th, 7th/8th, etc. Formulae for given orders are, however, not unique. They are derived, for a given order, say 2nd/3rd, by (*1*) trying to match stability regions of the 2nd and 3rd order formulae and (*2* choosing coefficients so that leading-order error terms are dominant in the error estimate. This is a complicated business and we strongly recommend that you use well-studied embedded formulae rather than trying to derive new ones yourself.

### III.9.8.3 Adjusting the Step Size $h$

Now we have an error estimate $\delta y_{n+1} = y_{n+1} - y^*_{n+1}$. Our goal must now be to keep the error small: $|\delta y_{n+1}| \le \epsilon$ by adjusting $h$.

Usually, one sets

$$\epsilon = \underbrace{\epsilon_a}_{\substack{\text{absolute error} \\ \text{tolerance}}} + |y_{n+1}| \underbrace{\epsilon_r}_{\substack{\text{relative error} \\ \text{tolerance}}} \ . \tag{III.9.29}$$

Now define

$$\Delta = \frac{|\delta y_{n+1}|}{\epsilon} \ , \tag{III.9.30}$$

and we want $\Delta \approx 1$.

Note that for a $p$-th-order formula, $\Delta \sim \mathcal{O}(h^p)$. So if you took a step $h$ and got a value $\Delta$, then the step $h_{\rm desired}$ you need to get $\Delta_{\rm desired}$ is

$$h_{\rm desired} = h \left| \frac{\Delta_{\rm desired}}{\Delta} \right|^{\frac{1}{p}} \ , \tag{III.9.31}$$

and $\Delta_{\rm desired} = 1$.

The algorithm to adjust $h$ can be written as follows:

(1) Take step $h$, measure $\Delta$.

(2) If $\Delta > 1$ (error too large), then
- set $h_{\rm new} = h \left|\frac{1}{\Delta}\right|^{\frac{1}{p}} S$, where $S$ is a fudge factor ($\sim$0.9 or so).
- *reject* the old step, redo with $h_{\rm new}$.

(3) If $\Delta < 1$ (error too small), then
- set $h_{\rm new} = h \left|\frac{1}{\Delta}\right|^{\frac{1}{p}} S$.
- accept old step, take next step with $h_{\rm new}$.

#### III.9.8.4 Other Considerations for improving RK Methods

**PI Control**

In control systems language, $h_{\rm new} = h \left|\frac{1}{\Delta}\right|^{1/p} S$ is an "integral controller" for $\log h$. One can improve the stability of the controller by adding a "proportional" term. The resulting algorithm is then $h_{n+1} = S h_n \Delta_n^{-\alpha} \Delta_{n-1}^{\beta}$, with $\alpha \approx \frac{1}{p} - \frac{3}{4}\beta$ and $\beta \approx \frac{0.4}{p}$.

**Dense Output**

Some RK formulae allow free accurate interpolation to get values of $y$ at arbitrary $x$ within a single step by using the $k_i$ values that have been computed, i.e.,

$$y(x_n + \theta h) = y_n + b_1(\theta) k_1 + b_2(\theta) k_2 + \cdots \,, \tag{III.9.32}$$

where $0 \le \theta \le 1$.

With this feature, one can (for example) output at every $\Delta x = 0.2$ while still allowing fully adaptive $h$, including $h > 0.2$, if the error tolerance allows.

# III.10 Linear Systems of Equations

Linear systems of equations (LSEs) are everywhere:

- Interpolation (e.g., for the computation of the spline coefficients).
- ODEs (implicit time integration).
- Solution methods for elliptic PDEs.
- Solution methods for non-linear equations by linearization and Newton iterations.

Example applications in astrophysics are plenty: Stellar structure and evolution, Poisson solvers, radiation transport and radiation-matter coupling, nuclear reaction networks. One encounters LSEs basically anywhere where implicit solutions are necessary, because balance/equilibrium must be found.

## III.10.1 Basics

A system of linear equations can be written in matrix form:

$$A\mathbf{x} = \mathbf{b} \ . \tag{III.10.1}$$

Here, $A$ is a real $n \times n$ matrix with coefficients $a_{ij}$. $\mathbf{b}$ is a given real vector. $\mathbf{x}$ is the vector of $n$ unknowns.

In the following, we will use $\mathbf{I}$ to indicate the identity matrix and $\mathbf{O}$ to indicate the zero matrix.

A quick flash-back from Linear Algrebra:

A LSE of the form of Eq. III.10.1 has a unique solution if and only if $\det A = |A| \neq 0$ and $\mathbf{b} \neq 0$. The solution then is

$$\mathbf{x} = A^{-1}\mathbf{b} \ , \tag{III.10.2}$$

where $A^{-1}$ is the inverse of $A$ with $AA^{-1} = A^{-1}A = \mathbf{I}$. For the case of $\det A = 0$, the equations either have no solution (i.e., they form an inconsistent set of equations) or an infinite number of solutions (i.e, they are an undetermined set of equations).

## III.10.2 Matrix Inversion – The Really Hard Way of Solving an LSE

The inverse of a matrix $A$ is given by

$$A^{-1} = \frac{1}{|A|} \underbrace{\mathrm{adj}A}_{\text{adjugate}} \ . \tag{III.10.3}$$

the adjugate of $A$ is the transpose of $A$'s cofactor matrix $C$:

$$\mathrm{adj}A = C^T \ . \tag{III.10.4}$$

So the problem boils down to finding $C$ and $\det A$. How to do this, you learned in your Linear Algebra class, but here is a quick reminder:

#### III.10.2.1 Finding $\det A$ for an $n \times n$ Matrix

If the $n \times n$ matrix $A$ is triangular, that is

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & a_{22} & \vdots \\ 0 & & a_{nn} \end{pmatrix} \quad \text{upper triangular matrix,} \tag{III.10.5}$$

or

$$A = \begin{pmatrix} a_{11} & & 0 \\ \vdots & a_{22} & \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \quad \text{lower triangular matrix,} \tag{III.10.6}$$

then computing the determinant is trivial:

$$\det A = \prod_{i=1}^{n} a_{ii} \ . \tag{III.10.7}$$

If $A$ is not triangular, one resorts either to Laplace's Formula or to the Leibniz Formula.

**Laplace's Formula**:

$$\det A = \sum_{j=1}^{n} a_{ij} c_{ij} \tag{III.10.8}$$

for any row $i$. $c_{ij}$ are the coefficients of the cofactor matrix of $A$ (see §III.10.2.2).

**Leibniz Formula**:

$$\det A = \sum_{\sigma \in S_n} \operatorname{sign} \sigma \prod_{i=1}^{n} a_{i\sigma} \ , \tag{III.10.9}$$

where:

$\sigma$ : is a permuatation of the set $\{1, 2, \cdots, n\}$,
$S_n$ : is the complete set of possible permutations, namely a symmetric group on $n$ elements,

and

$$\operatorname{sign} \sigma = \begin{cases} +1 & \text{if } \sigma \text{ is an even permutation,} \\ -1 & \text{if } \sigma \text{ is an odd permutation.} \end{cases} \tag{III.10.10}$$

Here, *even* means that $\sigma$ can be obtained with by an even number of replacements and *odd* means that $\sigma$ can be obtained by an odd number of replacements.

Here is an example for $n = 3$:

| $\sigma$ | Result | # of replacements | $\operatorname{sign} \sigma$ |
|---|---|---|---|
| 1 | 123 | 0 | +1 |
| 2 | 231 | 2 | +1 |
| 3 | 312 | 2 | +1 |
| 4 | 213 | 1 | -1 |
| 5 | 132 | 1 | -1 |
| 6 | 321 | 1 | -1 |

Note that there are $n!$ possible permutations!

#### III.10.2.2 The Cofactor Matrix of an $n \times n$ Matrix

$$c_{ij} = (-1)^{i+j} m_{ij} \ , \tag{III.10.11}$$

where $m_{ij}$ is the *minor* for the coefficient $a_{ij}$ of our $n \times n$ matrix $A$.

Finding the minors $m_{ij}$ of $A$ is a multi-step process that is best implemented in a recursive algorithm:

To find the minors $m_{ij}$ of $A$,

(1) choose an entry $a_{ij}$ of $A$,

(2) create a submatrix $B$ that contains all coefficients of $A$ that do not belong to row $i$ and column $j$,

(3) obtain the determinant of $B$.
This is trivial, if $B$ is $2 \times 2$, easy if it is $3 \times 3$. If $B$ is larger than $3 \times 3$, then use

$$\det B = \sum_{j=1}^{n} b_{ij} c_{ij}^{B}$$

for any convenient $i$ of $B$. $c_{ij}^{B} = (-1)^{i+j} m_{ij}^{B}$ is determined by recursion through (1).

### III.10.3 Cramer's Rule

Cramer's Rule is a smarter way to solve LSEs of the kind

$$A\mathbf{x} = \mathbf{b} \ . \tag{III.10.12}$$

Provided $A$ is invertible (i.e., has non-zero $\det A$), the solution to Eq. III.10.12 is then

$$x_i = \frac{\det A_i}{\det A} \ , \tag{III.10.13}$$

where $A_i$ is the matrix formed from $A$ by replacing its $i$-th column by the column vector $\mathbf{b}$.

Cramer's rule is more efficient than matrix inversion. The latter scales in complexity with $n!$ (where n is the number of rows/columns of $A$), while Cramer's rule has been shown to scale with $n^3$, so is more efficient for large matrixes and has comparable efficiency to direct methods such as Gauss Elimination, which we will discuss in §III.10.4.1.

### III.10.4 Direct LSE Solvers

There are two main classes of sensible algorithms to solve LSEs. *Direct methods* consist of a finite set of transformations of the original coefficient matrix that reduce the LSE to one that is easily solved. *Indirect methods* (also called *iterative methods*) consist of algorithms that specify a series of steps that lead closer and closer to the solution without, however, ever exactly reaching it.

### III.10.4.1 Gauss Elimination

The idea behind Gauss elimination – and all other direct methods – is to bring a LSE into a form that has an easy solution. Let us consider the following:

$$A\mathbf{x} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \ . \tag{III.10.14}$$

This LSE is solved trivially by simple back-substitution:

$$x_3 = \frac{b_3}{a_{33}} \ , \qquad x_2 = \frac{1}{a_{22}}(b_2 - a_{23}x_3) \ , \quad \text{and} \quad x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3) \ . \tag{III.10.15}$$

The Gauss algorithm consists of a series of steps to bring any $n \times n$ matrix into the above upper triangular form. It goes as follows:

(1) Sort the rows of $A$ so that the diagonal coefficient $a_{ii}$ (called *the pivot*) of row $i$ (for all $i$) is non-zero. If this is not possible, the LSE cannot be solved.

(2) Replace the $j$-th equation with

$$-\frac{a_{j1}}{a_{11}} \times (\text{1-st equation}) + (j\text{-th equation}) \ , \tag{III.10.16}$$

where $j$ runs from 2 to $n$. This will zero-out column 1 for $i > 1$.

(3) Repeat the previous step, but starting with the next row down and with $j >$ (current row number). The current row be row $k$. Then we must replace rows $j$, $j > k$, with

$$-\frac{a_{jk}}{a_{kk}} \times (k\text{-th equation}) + (j\text{-th equation}) \ , \tag{III.10.17}$$

where $k < j \leq n$.

(4) Repeat (3) until all rows have been reduced and the matrix is in upper triangular form.

(5) Back-substitute to find $\mathbf{x}$.

While the Gauss Elimination algorithm is straightforward, it is not very computationally efficient, since it requires $\mathcal{O}(n^3)$ operations for $n$ equations. This can also lead to build-up of floating point errors.

### III.10.4.2 Pivoting

Gauss Elimination suffers from poor accuracy if the matrix coefficients are of very different sizes. Here is an example:

$$\begin{aligned} 10^{-5}x_1 + \quad x_2 &= 1 \\ x_1 + \quad x_2 &= 2 \ . \end{aligned} \tag{III.10.18}$$

Using Gauss Elimination, the exact solution is:

$$x_1 = \frac{10^5}{99999} \approx 1 \ , \quad x_2 = \frac{-99998}{-99999} \approx 1 \ . \tag{III.10.19}$$

If, however, we solve the problem using floating point numbers, the result may be quite different. Let's say $m = 4$,

$$\left( \begin{array}{cc|c} 0.1000 \times 10^{-4} & 0.1000 \times 10^{1} & 0.1000 \times 10^{1} \\ 0.1000 \times 10^{1} & 0.1000 \times 10^{1} & 0.2000 \times 10^{1} \end{array} \right) \ , \tag{III.10.20}$$

becomes

$$\left( \begin{array}{cc|c} 0.1000 \times 10^{-4} & 0.1000 \times 10^{1} & 0.1000 \times 10^{1} \\ 0.0000 \times 10^{0} & -0.1000 \times 10^{6} & -0.1000 \times 10^{6} \end{array} \right) \ , \tag{III.10.21}$$

and, thus,

$$x_1 = 0 \ , \quad x_2 = 1 \ , \tag{III.10.22}$$

which is an incorrect result.

The above is an example of what is called *poor scaling*. Whenever the coefficients of an LSE are of greatly varying sizes, we must expect that rounding errors may build up due to loss of significant figures.

Poor scaling can be fixed by the following simple procedure:

At each of the intermediate Gauss steps, compare the size of the pivot $a_{kk}$ with all $a_{jk}$, $k < j \leq n$, and exchange row $k$ with the row of the largest $a_{jk}$.

This simple procedure minimized the floating point error in Gaussian elimination. It is called *partial pivoting*. Here is the example from above, but this time with pivoting:

$$\left( \begin{array}{cc|c} 0.1000 \times 10^{1} & 0.1000 \times 10^{1} & 0.2000 \times 10^{1} \\ 0.1000 \times 10^{-4} & 0.1000 \times 10^{1} & 0.1000 \times 10^{1} \end{array} \right) \ , \tag{III.10.23}$$

which becomes

$$\left( \begin{array}{cc|c} 0.1000 \times 10^{1} & 0.1000 \times 10^{1} & 0.2000 \times 10^{1} \\ 0.0000 \times 10^{0} & 0.1000 \times 10^{1} & 0.1000 \times 10^{1} \end{array} \right) \ , \tag{III.10.24}$$

and

$$x_1 = 1 \ , \quad x_2 = 1 \ , \tag{III.10.25}$$

which is an almost fully correct result.

Essentially, pivoting modifies the replacement of row $j$ with row $j - \epsilon$ row $k$ so that $0 < \epsilon < 1$ is as small as possible.

*Total pivoting* exists as well and exchanges also columns so that the largest element in the matrix becomes the pivot. This is very time consuming and we do not discuss it here.

### III.10.4.3 Decomposition Methods (LU Decomposition)

The idea behind decomposition methods is to split a given LSE into smaller, considerably easier to solve parts. The simplest such method is the Lower-Upper (LU) decomposition.

Given $A\mathbf{x} = \mathbf{b}$, suppose we can write the matrix $A$ as

$$A = LU \ , \tag{III.10.26}$$

where $L$ is lower triangular and $U$ is upper triangular. The solution of the LSE than becomes straightforward:

$$A\mathbf{x} = \mathbf{b} \longrightarrow (LU)\mathbf{x} = \mathbf{b} \longrightarrow L(U\mathbf{x}) = \mathbf{b} \ . \tag{III.10.27}$$

If we now set $\mathbf{y} = U\mathbf{x}$, then we have transformed the original system into two systems

$$\begin{aligned} &(1) \quad L\mathbf{y} = \mathbf{b} \ , \\ &(2) \quad U\mathbf{x} = \mathbf{y} \ . \end{aligned} \tag{III.10.28}$$

Both these LSEs are triangular. (1) can be trivially solved by forward-substitution and (2) can be trivially solved by back-substitution. So we have now to solve two LSEs instead of one, but both are very easy to solve! The difficult part is now to find the $L$ and $U$ parts of $A$! This is done via matrix factorization.

### III.10.4.4 Factorization of a Matrix

The process of decomposing $A$ into $L$ and $U$ parts is called factorization. Any matrix $A$ that can be brought into $U$ form without swapping any rows has a $LU$ decomposition. This decomposition is not generally unique and there are multiple ways of factorizing $A$.

$A$ is an $n \times n$ matrix with $n^2$ coefficients. $L$ and $U$ are triangular and have $n(n+1)/2$ entries each for a total of $n^2 + n$ entries. Hence, $L$ and $U$ together have $n$ coefficients more than $A$ and these can be chosen to our own liking.

To derive the $LU$ factorization, we begin by writing out $A = LU$ in coefficients:

$$a_{ij} = \sum_{s=1}^{n} l_{is} u_{sj} = \sum_{s=1}^{\min(i,j)} l_{is} u_{sj} \ , \tag{III.10.29}$$

where we have used that $l_{is} = 0$ for $s > i$ and $u_{sj} = 0$ for $s > j$.

Let's start with entry $a_{ij} = a_{11}$:

$$a_{11} = l_{11} u_{11} \ . \tag{III.10.30}$$

We can now make use of the freedom of choosing $n$ coefficients of $L$ and $U$.

In *Doolittle's factorization*, one sets $l_{ii} = 1$, which makes $L$ *unit triangular*. In *Crout's factorization*, one sets $u_{ii} = 1$, which means that $U$ is unit triangular.

Following Doolittle, we set $l_{11} = 1$ and, with this, $u_{11} = a_{11}$. We can now compute all the elements of the first row of $U$ and of the first column of $L$ by setting $i = 1$ or $j = 1$,

$$\begin{aligned} u_{1j} &= a_{1j} \qquad && i = 1, j > 1 \ , \\ l_{i1} &= \frac{a_{i1}}{u_{11}} \qquad && j = 1, i > 1 \ . \end{aligned} \tag{III.10.31}$$

Consider now $a_{22}$:

$$a_{22} = l_{21} u_{12} + l_{22} u_2 2 \ . \tag{III.10.32}$$

With Doolittle, $l_{22} = 1$, thus $u_{22} = a_{22} - l_{21}u_{12}$, where $u_{12}$ and $l_{21}$ are known from the previous steps. The second row of $U$ and the second column of $L$ are now calculated by setting either $i = 2$ or $j = 2$:

$$\begin{aligned} u_{2j} &= a_{2j} - l_{21}u_{1j} \qquad & i = 2, j > 2 \ , \\ l_{i2} &= \frac{a_{i2} - l_{i1}u_{12}}{u_{22}} \qquad & j = 2, i > 2 \ . \end{aligned} \tag{III.10.33}$$

This procedure can be repeated for all the rows and columns of $U$ and $L$. In the following, we provide a compact form of the algorithm in pseudocode.

For $k = 1, 2, \cdots, n$ do

- Choose either $l_{kk}$ (Doolittle) or $u_{kk}$ (Crout) [the choice must be non-zero] and compute the other from
$$l_{kk}u_{kk} = a_{kk} - \sum_{s=1}^{k-1} l_{ks}u_{sk} \ . \tag{III.10.34}$$
- Build the $k$-th row of $U$:
For $j = k+1, \cdots, n$ do:
$$u_{kj} = \frac{1}{l_{kk}} \left( a_{kj} - \sum_{s=1}^{k-1} l_{ks}u_{sj} \right) \ . \tag{III.10.35}$$
- Build the $k$-th column of $L$:
For $i = k+1, \cdots, n$ do:
$$l_{ik} = \frac{1}{u_{kk}} \left( a_{ik} - \sum_{s=1}^{k-1} l_{is}u_{sk} \right) \ . \tag{III.10.36}$$

The LU decomposition algorithm has a complexity of $\mathcal{O}(n^3)$, but the constant in front is smaller than with pure Gaussian Elimination.

#### III.10.4.5 Tri-Diagonal Systems

Consider the following $4 \times 4$ LSE:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 \\ 0 & a_2 & b_3 & c_3 \\ 0 & 0 & a_3 & b_4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} \ . \tag{III.10.37}$$

Tri-diagonal LSEs like the above give rise to particularly simple results when using Gaussian elimination. Forward elimination at each step yeilds:

$$\begin{pmatrix} 1 & c_1/d_1 & 0 & 0 \\ 0 & 1 & c_2/d_2 & 0 \\ 0 & 0 & 1 & c_3/d_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \ , \tag{III.10.38}$$

with

$$\begin{aligned} d_1 &= b_1\,, & y_1 &= f_1/d_1\,,\\ d_2 &= b_2 - a_1c_1/d_1\,, & y_2 &= (f_2 - y_1a_1)/d_2\,,\\ d_3 &= b_3 - a_2c_2/d_2\,, & y_3 &= (f_3 - y_2a_2)/d_3\,,\\ d_4 &= b_4 - a_3c_3/d_3\,, & y_4 &= (f_4 - y_3a_3)/d_4\,. \end{aligned} \tag{III.10.39}$$

Back-substitution than gives

$$\begin{aligned} x_1 &= y_1 - x_2c_1/d_1\,,\\ x_2 &= y_2 - x_3c_2/d_2\,,\\ x_3 &= y_3 - x_4c_3/d_3\,,\\ x_4 &= y_4\,. \end{aligned} \tag{III.10.40}$$

In the general $n \times n$ case, the procedure is as follows:

- Forward Elimination:
  (1) At the first step: $d_1 = b_1$ and $y_1 = f_1/d_1$.
  (2) At the $k$-th step:
  $$d_k = b_k - a_{k-1}c_{k-1}/d_{k-1}\ ,$$
  $$y_k = (f_k - y_{k-1}a_{k-1})/d_k\ .$$
- Backward Substitution: Determine all $x_k$ by

$$x_n = y_n\ ,$$
$$x_{k-1} = y_{k-1} - x_kc_{k-1}/d_{k-1}\ .$$

The complexity of this algorithm is still $\mathcal{O}(n^3)$, but the constant in front is much smaller than that of LU decomposition for this kind of matrix.

### III.10.5 Iterative Solvers

Direct methods for the solution of LSEs generally have a computational complexity of $\mathcal{O}(n^3)$ and also suffer from round-off errors due to the many operations necessary for a direct solution. If one is satisfied with an approximate solution, one can resort to iterative methods that start from some initial "guess" $\mathbf{x}^{(0)}$ and define a sequence of approximations $\mathbf{x}^{(1)}$, ..., $\mathbf{x}^{(m)}$, which, in principle, converge to the exact solution.

A large class of iterative methods can be defined as follows:

Given

$$A\mathbf{x} = \mathbf{b}\ , \qquad \text{with } \det A \neq 0\ , \tag{III.10.41}$$

then the coefficient matrix can be split – in an infinite number of ways – into the form

$$A = N - P\ , \tag{III.10.42}$$

where $N$ and $P$ are matrices of the same order as $A$. The LSE is then written as

$$N\mathbf{x} = P\mathbf{x} + \mathbf{b}\ . \tag{III.10.43}$$

Starting from some vector $\mathbf{x}^{(0)}$, we define a sequence of vectors $\{\mathbf{x}^{(i)}\}$ by the recursion

$$N\mathbf{x}^{(i)} = P\mathbf{x}^{(i-1)} + \mathbf{b} \ , \qquad i = 1, 2, \cdots \ . \tag{III.10.44}$$

The various iteration methods available in the literature are characterized by their choices of $N$ and $P$. Right away, one notes from Eq. III.10.44 that $N$ must be chosen such that $\det N \neq 0$. It should also be chosen such that the LSE of the form $N\mathbf{y} = \mathbf{z}$ that must be solved at each iteration step is easily solved by a direct method.

It is possible to show the following (we will not show proofs):

(1) An iterative method converges if and only if the *convergence matrix* $M = N^{-1}P$ exists and has all eigenvalues $\lambda_i$ with modulus less than 1:

$$\rho(M) = \max_i |\lambda_i| < 1 \ . \tag{III.10.45}$$

(2) A weaker, but more straightforwardly useful statement is that the method converges if at least one norm of the convergence matrix is strictly less than one:

$$||M|| < 1 \ . \tag{III.10.46}$$

Note, however, that the reverse is not true. That is a method may converge, but some of the norms may be equal or larger than 1. In the above, we have used the matrix norm, which is defined analogously to the vector norm,

$$||\mathbf{x}||_p = \left[\sum_j x_j^p\right]^{1/p} \ , \qquad p > 0 \ . \tag{III.10.47}$$

The matrix norm is then

$$||A||_p = \max_{||\mathbf{x}||_p = 1} ||A\mathbf{x}||_p \ . \tag{III.10.48}$$

In the following, we will assume that all diagonal elements of $A$ are 1. This is easily achieved by dividing each row by its diagonal element. If a row happens to have a zero diagonal elment, we just have to re-order the rows.

### III.10.5.1 Jacobi Iteration

$A$ (with $a_{ii} = 1$) is split into diagonal and off-diagonal parts

$$A = \mathbf{I} - (A_L + A_U) \ , \tag{III.10.49}$$

so $N = I$ and $P = (A_L + A_U)$ is the sum of upper and lower triangular matrices with diagonal 0. The iteration scheme is then

$$\mathbf{x}^{(i+1)} = \mathbf{b} + (A_L + A_u)\mathbf{x}^{(i)} \ , \tag{III.10.50}$$

and the convergence matrix is simply $M = N^{-1}P = P$.

### III.10.5.2 Gauss-Seidel Iteration

In Jacobi's method, the old guess is used to estimate all the elements of the new guess. In Gauss-Seidel iteration, each new iterate is used as soon as it becomes available:

$$\begin{aligned} A &= \mathbf{I} - (A_L + A_U) \ , \\ \mathbf{x}^{(i+1)} &= \mathbf{b} + A_L \mathbf{x}^{(i+1)} + A_U \mathbf{x}^{(i)} \ . \end{aligned} \tag{III.10.51}$$

Hence, $N = \mathbf{I} - A_L$ and $P = A_U$. As before, $A_L$ and $A_U$ are the lower and upper diagonal parts of $A$ with diagonal elements set to zero. If we start computing the elements of $\mathbf{x}^{(i+1)}$ from the first, i.e., from $x_1^{i+1}$, then all the terms on the RHS are known by the time the are used. Specifically, we have

$$x_k^{(i+1)} = b_k + \sum_{l>j} a_{jl}^U \, x_l{}^{(i)} + \sum_{l<j} a_{jl}^L \, x_l^{(i+1)} \ , \tag{III.10.52}$$

where we denote the coefficients of $A_U$ and $A_L$ as $a_{jl}^U$ and $a_{jl}^L$, respectively.

### III.10.5.3 Successive Over-Relaxation (SOR) Method

One can interpret Gauss-Seidel iteration as a method that applies at each guess $\mathbf{x}^{(i)}$ a correction term $\mathbf{c}$. We can rewrite this scheme as

$$\begin{aligned} \mathbf{x}^{(i+1)} &= \mathbf{b} + A_L \mathbf{x}^{(i+1)} + A_U \mathbf{x}^{(i)} \ , \\ &= \mathbf{x}^{(i)} + \underbrace{\left[ \mathbf{b} + A_L \mathbf{x}^{(i+1)} + (A_U - \mathbf{I}) \mathbf{x}^{(i)} \right]}_{\mathbf{c}} \ , \\ &= \mathbf{x}^{(i)} + \mathbf{c} \ . \end{aligned} \tag{III.10.53}$$

The idea of SOR is now that the convergence of the method can be pushed by using a slightly larger correction factor $\mathbf{c}' = \omega \mathbf{c}$ with $\omega \gtrsim 1$. Since this can also accelerate divergence, $\omega$ is chosen to be close to 1.

## III.10.6 Aside on Determinants and Inverses

Now that we have studied how to solve LSEs, it is much easier to find determinants and inverses of matrices!

**Determinant**

The determinant of a well-behaved matrix is easily found via LU decomposition.

$$\begin{aligned} A &= LU \ , \\ \det A &= \det L \det U \ . \end{aligned} \tag{III.10.54}$$

Since both $L$ and $U$ triangular, we have

$$\det L = \prod_{i=1}^{n} l_{ii} \quad \text{and} \quad \det U = \prod_{i=1}^{n} u_{ii} \ . \tag{III.10.55}$$

Furthermore, in Doolite factorization $\det L = 1$ and in Crout factorization $\det U = 1$. Note that if pivoting was used, then the sign of $\det A$ depends on the number of even and odd permutations that were carried out.

**Inverse**

We can solve for $A^{-1}$,

$$AA^{-1} = \mathbf{I} \,, \tag{III.10.56}$$

by solving $n$ systems of $n$ linear equations. Let $e_i$ be the $i$-th unit vector (only its $i$-th component is non-zero), then we have to solve

$$A\mathbf{c}_i = \mathbf{e}_i \tag{III.10.57}$$

for $i = 1, \cdots, n$. $c_i$ is the $i$-th column of the inverse $A^{-1}$.

This method is faster than normal inversion the requries the evaluation of $n + 1$ determinants in Cramer's rule.

## III.10.7 The Condition Number

In your linear algebra class you learned that given a matrix $A \in \mathbb{R}^{N\times N}$, the solution to $A\boldsymbol{x} = \boldsymbol{b}$ is unique if and only if any of the following equivalent statements are true:

- $A$ is invertible or "nonsingular" ($A^{-1}$ exists)
- The rows of $A$ are linearly independent
- The columns of $A$ are linearly independent
- $A\boldsymbol{x} = \mathbf{0}$ only has the trivial solution $\boldsymbol{x} = \mathbf{0}$
- $A$ has no zero eigenvalues
- $\det A \neq 0$

This list appears to motivate the use of the determinant as a means for testing whether or not a matrix is singular. However, the determinant is unsuitable for testing whether or not a matrix is nearly singular. That is, if the determinant of a matrix $A$ is nonzero, $\det A$ can be made arbitrarily large or small by multiplying $A$ by a constant factor. The statement that "$\det A <$ a small number" does not imply that $A$ is nearly singular (and we should expect large errors when solving $A\boldsymbol{x} = \boldsymbol{b}$). The remainder of this section is heavily inspired by the discussion in Heath [1] §2.3.

Suppose $A$ is nonsingular (but perhaps nearly singular), and we make some error in representing the right-hand side $\boldsymbol{b}$ of the system of equations $A\boldsymbol{x} = \boldsymbol{b}$ such that

$$A(\boldsymbol{x} + \boldsymbol{\delta x}) = \boldsymbol{b} + \boldsymbol{\delta b}, \tag{III.10.58}$$

where $\boldsymbol{\delta b}$ is the error made in representing $\boldsymbol{b}$, and $\boldsymbol{x}$ is the error this leads to in the solution $\boldsymbol{x}$. By definition $A\boldsymbol{x} = \boldsymbol{b}$, so

$$\boldsymbol{\delta x} = A^{-1}\boldsymbol{\delta b}. \tag{III.10.59}$$

Therefore using the vector/matrix norm $\|\cdot\|$ we have $\|A\|\|\boldsymbol{x}\| \geq \|\boldsymbol{b}\|$ and $\|A^{-1}\|\|\boldsymbol{\delta b}\| \geq \|\boldsymbol{\delta x}\|$, such that

$$\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|} \leq \frac{\|A^{-1}\|}{\|A\|}\frac{\|\boldsymbol{\delta b}\|}{\|\boldsymbol{b}\|}. \tag{III.10.60}$$

The quantity $\|A^{-1}\|/\|A\|$ is called the "condition number of $A$". In particular, if the system of linear equations is specified to machine precision $\epsilon$, then we should expect

$$\frac{\|\boldsymbol{\delta x}\|}{\|\boldsymbol{x}\|} \lesssim \text{cond}(A)\cdot\epsilon. \tag{III.10.61}$$

As a rule of thumb, one can lose up to about $\log_{10}(\text{cond}(A))$ digits of precision in solving the system of equations.

## References

[1] Michael T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, New York, NY, USA, 2 edition, 2002.

# III.11 Ordinary Differential Equations: Boundary Value Problems

As mentioned already at the beginning of §III.9, a boundary value problem (BVP) consists of finding a solution of an ODE in an interval $[a, b]$ that satisfies constraints at both ends (boundary conditions).

A typical example of a BVP is

$$y'' = f(x, y, y') \ , \ \ y(a) = A \ , \ \ y(b) = B \ , \ \text{and } x \in [a, b] \ . \tag{III.11.1}$$

## III.11.1 Shooting Method

The shooting method is a very frequently used approach to BVPs. The idea behind it is to reduce a BVP to an initial value problem by making an educated guess on unknown inner boundary conditions and then iterating until a modified guessed inner boundary condition leads to the correct known outer boundary value.

In the example given in Eq. III.11.1, the value of $y'(a)$ is not known, but is needed for the problem's solution. We can make a guess $y'(a) = z$ and then we know $f(a, y(a), y'(a))$, can reduce the second-order problem to two first-order problems and just integrate the two ODEs out to $b$. Since we have chosen $z$, we have now solved $y = y(x, z)$, but our goal is to find $y$ such that $y(b, z) = B$.

In other words, we can define a new function

$$\Phi(z) = y(b, z) - B \tag{III.11.2}$$

and search for a $z$ so that $\Phi(z) = 0$. Hence, we are looking for the root of $\Phi(z)$! For this we can use what was discussed in §III.5.

The full shooting algorithm for $y'' = f(x, y, y')$ goes than as follows:

(1) Guess a starting value $z_0 = y'(a)$, set the iteration counter $i = 0$.

(2) Compute $y = y(x, z_i)$ by integrating the IVP.

(3) Compute $\Phi(z_i) = y(b, z_i) - B$. If $z_i$ does not give a sufficiently accurate solution of the full problem, increment $i$ to $i + 1$ and find a value for $z_{i+1}$ using a root finder on $\Phi(z_i) = 0$. Then go back to (2).

Note that one typically ends up with the secant method, since the derivative of $\Phi(z)$ is not known in the general case and one is stuck with having to numerically compute it. For this, at least two guesses for $z$ are needed.

## III.11.2 Finite-Difference Method

BVPs of the kind given by Eq. (III.11.1) can be solved by Taylor expanding the ODE itself to linear order (assuming there are no non-linearities in y and y'):

$$y'' = g(x) - p(x)y' - q(x)y \ , \tag{III.11.3}$$

where $g(x)$, $p(x)$, and $q(x)$ are functions of $x$ only and the sign convention is arbitrary.

We can now discretize $y'$ and $y''$ on an evenly spaced grid with step size $h$,

$$\begin{aligned} y'(x_i) &= \frac{y(x_{i+1}) - y(x_{i-1})}{2h} \ , \\ y''(x_i) &= \frac{y_{i+1} + y_{i-1} - 2y_i}{h^2} \ , \end{aligned} \tag{III.11.4}$$

where $x_i = a + ih$, $i = 0, \cdots, n+1$, and $h = (b-a)/(n+1)$.

The discrete version of Eq. (III.11.3) is then a system of $n+2$ linear algebraic equations,

$$\begin{aligned} y_0 &= A \ , \\ y_{i-1}(1 - \frac{h}{2}p_i) - y_i(2 - h^2 q_i) + y_{i+1}(1 + \frac{h}{2}p_i) &= h^2 g_i \ , \\ y_{n+1} &= B \ , \end{aligned} \tag{III.11.5}$$

where $p_i = p(x_i)$, $g_i = g(x_i)$, and $q_i = q(x_i)$. One ends up with a tri-diagonal matrix of dimension $n \times n$:

$$\begin{pmatrix} -2 + h^2 q_1 & 1 + \frac{h}{2}p_1 & 0 & & \cdots & & 0 \\ 1 - \frac{h}{2}p_2 & \ddots & \ddots & 0 & & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 & \cdots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 & \vdots \\ \vdots & \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & 0 & \ddots & \ddots & 1 + \frac{h}{2}p_{n-1} \\ \vdots & 0 & 0 & \cdots & 0 & 1 - \frac{h}{2}p_n & -2 + h^2 q_n \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} h^2 g_1 - A(1 - \frac{h}{2}p_1) \\ h^2 g_2 \\ \vdots \\ \vdots \\ \vdots \\ h^2 g_{n-1} \\ h^2 g_n - B(1 + \frac{h}{2}p_n) \end{pmatrix} \tag{III.11.6}$$

This system can be solved in a straightforward fashion using the methods discussed in §III.10.

# III.12 Partial Differential Equations

One can fill an entire term just talking about formal aspects of partial differential equations (PDEs). Here we just provide a quick and dirty introduction to PDEs in order to familiarize you with the basics and with key terminology. There are multiple specialized PDE classes offered by Caltech's applied math department (this is probably true for any applied math department in the world) and we encourage you to attend them should you be interested in learning more about the details of PDEs.

**Preliminaries and Definitions**

- A PDE is a relation between the partial derivatives of an unknown function and the independent variables.
- The order of the highest derivative sets the *order* of the PDE. If the highest derivative is a second derivative, we are dealing with a second-order PDE.
- A PDE is *linear* if it is of the first degree in the dependent variable (i.e. the unknown function) and in its partial derivatives.
- If each term of a PDE contains either the dependent variable or one of its partial derivatives, the PDE is called *homogeneous*. Otherwise it is *non-homogeneous*.

## III.12.1 Types of PDEs

There are three general types of PDEs: hyperbolic PDEs, parabolic PDEs, and elliptic PDEs. Not all PDEs fall into one of these three types, but many PDEs used in practice do. These classes of PDEs model different sorts of phenomena, display different behavior, and require different numerical techniques for their solution.

It is not always straighforward to see (to show and proof) what type of PDE a given PDE one is dealing with is.

The special case of linear second-order differential equations for an unknown function $u$ in two independent variables $x$ and $y$ of the form

$$a\partial^2_{xx}u + b\partial^2_{xy}u + c\partial^2_{yy}u + d\partial_x u + e\partial_y u + fu = g \ , \tag{III.12.1}$$

can be straightforwardly categorized based on the discriminant,

$$b^2 - 4ac \begin{cases} < 0 & \rightarrow \quad \text{elliptic}, \\ = 0 & \rightarrow \quad \text{parabolic}, \\ > 0 & \rightarrow \quad \text{hyperbolic}, \end{cases} \tag{III.12.2}$$

and the names come from analogy with conic sections in the theory of ellipses.

### III.12.1.1 Hyperbolic PDEs

Hyperbolic equations in physics and astrophysics describe *dynamical* processes and systems that generally start at some initial time $t_0 = 0$ with some initial conditions. The equations are then integrated in time.

The prototypical linear second-order hyperbolic equation is the homogeneous wave equation,

$$c^2\partial_{xx}^2 u - \partial_{tt}^2 u = 0 \ , \tag{III.12.3}$$

where $c$ is the wave speed.

Another important class of hyperbolic equations are the *first-order hyperbolic systems*. In one space dimension and assuming a linear problem, such a PDE system has the form

$$\partial_t u + A\partial_x u = 0 \ , \tag{III.12.4}$$

where $u(x,t)$ is the state vector with $s$ components and $A$ is a $s \times s$ matrix. The problem is called *hyperbolic* if $A$ has only real eigenvalues and is diagonizable, i.e., has a complet set of linearly independent eigenvectors so that one can construct a matrix

$$\Lambda = Q^{-1}AQ \ , \tag{III.12.5}$$

where $\Lambda$ is diagonal and has real numbers on its diagonal. A key example of such an equation is the linear *advection equation*:

$$\partial_t u + v\partial_x u = 0 \ , \tag{III.12.6}$$

which is first order, linear, and homogeneous.

More complicated and non-linear first-order systems can still be written in the form of Eq. (III.12.4). Consider

$$\partial_t u + \partial_x F(u) = 0 \ , \tag{III.12.7}$$

where $F(u)$ is the *flux* and may or may not be non-linear in $u$. We can re-write this PDE in *quasi-linear* form, by introducing the Jacobian

$$\bar{A} = \frac{\partial F}{\partial u} \ , \tag{III.12.8}$$

and writing

$$\partial_t u + \bar{A}\partial_x u = 0 \ . \tag{III.12.9}$$

This PDE is hyperbolic if $\bar{A}$ has real eigenvalues and is diagonizable. The *equations of hydrodynamics* are a key example of a non-linear, first-order hyperbolic PDE system.

**Initial Conditions for Hyperbolic Problems**

One must specify $u(x, t = 0)$ (at all $x$) and (order-1) time derivatives.

**Boundary Conditions for Hyperbolic Problems**

One must specify either von Neumann, Dirichlet, or Robin boundary conditions:

(1) **Dirichlet Boundary Conditions**.
Let 0 be the inner and $L$ be the outer boundary of the domain.

$$\begin{aligned} u(x = 0, t) &= \Phi_1(t) \ , \\ u(x = L, t) &= \Phi_2(t) \ . \end{aligned} \tag{III.12.10}$$

(2) **von Neumann Boundary Conditions**.
Let 0 be the inner and $L$ be the outer boundary of the domain.

$$\begin{aligned} \partial_x u(x=0,t) &= \Psi_1(t) \ , \\ \partial_x u(x=L,t) &= \Psi_2(t) \ . \end{aligned} \tag{III.12.11}$$

Note that in a multi-D problem $\partial_x$ turns into the derivative in the direction of the normal to the boundary.

(3) **Robin Boundary Conditions**.
Let 0 be the inner and $L$ be the outer boundary of the domain, $a_1, b_1, a_2, b_2$ be real numbers, not $a_i = b_i = 0$.

$$\begin{aligned} a_1 u(x=0,t) + b_1 \partial_x u(x=0,t) &= \Psi_1(t) \ , \\ a_2 u(x=L,t) + b_2 \partial_x u(x=L,t) &= \Psi_2(t) \ . \end{aligned} \tag{III.12.12}$$

Dirichlet and von Neuman boundary conditions are recovered if either both $a_i$ or both $b_i$ vanish. Note that in a multi-D problem $\partial_x$ turns into the derivative in the direction of the normal to the boundary.

#### III.12.1.2 Parabolic PDEs

Parabolic PDEs describe processes that are slowly changing, such as the slow diffusion of heat in a medium, sediments in ground water, and radiation in an opaque medium. Parabolic PDEs are second order and have generally the shape

$$\partial_t u - k \partial_{xx}^2 u = f \ . \tag{III.12.13}$$

**Initial Conditions for Parabolic Problems**

One must specify $u(x, t = 0)$ at all $x$.

**Boundary Conditions for Parabolic Problems**

Dirichlet, von Neumann or Robin boundary conditions. If the boundary conditions are independent of time, the system will evolve towards a steady state ($\partial_t u = 0$). In this case, one may set $\partial_t u = 0$ for all times and treat Eq. (III.12.13) as an elliptic equation.

#### III.12.1.3 Elliptic PDEs

Elliptic equations describe systems that are static, in steady state and/or in equilibrium. There is no time dependence. A typical elliptic equation is the Poisson equation:

$$\nabla^2 u = f \ , \tag{III.12.14}$$

which one encounters in Newtonian gravity and in electrodynamics. $\nabla^2$ is the Laplace operator, and $f$ is a given scalar function of position. Elliptic problems may be linear ($f$ does not depend on $u$ or its derivatives) or non-linear ($f$ depends on $u$ or its derivatives).

**Initial Conditions for Elliptic Problems**

Do not apply, since there is no time dependence.

**Boundary Conditions for Elliptic Problems**

Dirichlet, von Neumann or Robin boundary conditions.

### III.12.2 Numerical Methods for PDEs: A Very Rough Overview

There is no such thing as a general robust method for the solution of generic PDEs. Each type (and each sub-type) of PDE requires a different approach. Real-life PDEs may be of mixed type or may have special properties that require knowledge about the underlying physics for their successful solution.

There are three general classes of approaches to solving PDEs:

(1) Finite Difference Methods.
The differential operators are approximated using their finite-difference representation on a given grid. A sub-class of finite-difference methods, so-called finite-volume methods, can be used for PDEs arising from conservation laws (e.g., the hydrodynamics equations).

Finite difference/volume methods have polynomial convergence for smooth functions.

(2) Finite Element Methods.
The domain is divided into cells ("*elements*"). The solution is represented as a single function (e.g., a polynomial) on each cell and the PDE is transformed to an algebraic problem for the matching conditions of the simple functions at cell interfaces.

Finite element methods can have polynomial or exponential convergence for smooth functions.

(3) Spectral Methods.
The solution is represented by a linear combination of known functions (e.g. trigonometric functions or special polynomials). The PDE is transformed to a set of algebraic equations (or ODEs) for the amplitudes of the component functions. A sub-class of these methods are the collocation methods. In them, the solution is represented on a grid and the spectral decomposition of the solution in known functions is used to estimate to a high degree of accuracy the partial derivatives of the solution on the grid points.

Spectral methods have exponential convergence for smooth functions.

### III.12.3 The Linear Advection Equation, Solution Methods, and Stability

We have already seen when talking about ODEs in §III.9 that accuracy is not the only consideration when devising a numerical scheme. Stability should not be neglected and it is one of the key considerations when it comes to hyperbolic and parabolic PDEs.

Analyzing a general discretization scheme for a general PDE for stability can be a tricky and involved task. For linear PDEs, however, we can use the rather straightforward stability analysis proposed by von Neumann. *Von Neumann stability analysis* can also be applied to non-linear problems if the non-linearity is weak.

The idea is that the complete solution of a PDE can be decomposed into a linear combination of eigenmodes $e^{ikx}$ (where $k$ is a spatial wave number). Each of these eigenmodes must individually

be a solution of the PDE, so

$$u(x,t) = \gamma e^{ikx} \ , \tag{III.12.15}$$

where $\gamma = \gamma(k)$ is the complex amplitude of mode $k$.

Assuming that we can write the evolution of the discretized solution from time $n$ to time $n+1$ as a (quasi)-linear operation,

$$u_j^{(n+1)} = \mathcal{D}(\Delta t, \Delta x) u_j^{(n)} \ , \tag{III.12.16}$$

then we can apply this to the case $u^{(n)} = e^{ikx}$ (we have set $\gamma = 1$ for convenience):

$$\begin{aligned} u_j^{(n+1)} &= \mathcal{D} e^{ikx_j} \ , \\ &= \xi e^{ikx_j} \end{aligned} \tag{III.12.17}$$

where $\xi$ is a complex number and $e^{ikx_j}$ has been factored out (which is always possible). $\xi$ is called the *amplification factor*, since the evolution of a single eigenmode from time 0 to time $n+1$ leads to a product $\prod_{i=0}^{n+1} \xi = \xi^n$. Obviously, this will only be stable if $|\xi| \leq 1$.

Let us now study the linear advection equation,

$$\frac{\partial u}{\partial t} + v \frac{\partial u}{\partial x} = 0 \ . \tag{III.12.18}$$

This is the simplest example of a hyperbolic equation and its exact solution is simply given by

$$u(x,t) = u(t=0, x - vt) \ , \tag{III.12.19}$$

hence, the advection equation does (as one might expect) just translate the given data along the x-axis with constant advection velocity $v$.

### III.12.3.1 First-order in Time, Centered in Space (FTCS) Discretization

There are many ways in which we could discretize Eq. (III.12.19). We could try a first-order in time, centered (second-order) in space (FTCS) method, for example:

$$u_j^{(n+1)} = u_j^{(n)} - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^{(n)} - u_{j-1}^{(n)}\right) \ . \tag{III.12.20}$$

We may now introduce $u(x, t^n) = e^{ikx}$ into the above equation,

$$\begin{aligned} u_j^{(n+1)} &= e^{ik\Delta x j} - \frac{v\Delta t}{2\Delta x}\left(e^{ik\Delta x(j+1)} - e^{ik\Delta x(j-1)}\right) \ , \\ &= \left(1 - \frac{v\Delta t}{2\Delta x}\left(e^{ik\Delta x} - e^{-ik\Delta x}\right)\right) e^{ik\Delta x j} \ , \\ &= \underbrace{\left(1 - \frac{v\Delta t}{\Delta x} i \sin(k\Delta x)\right)}_{=\xi} e^{ik\Delta x j} \ , \end{aligned} \tag{III.12.21}$$

and

$$|\xi| = \sqrt{\xi\xi^*} = \sqrt{1 + \left(\frac{v\Delta t}{\Delta x}\sin(k\Delta x)\right)^2} > 1 \ . \tag{III.12.22}$$

Hence, the FTCS method is *unconditionally unstable* for the advection equation!

### III.12.3.2 Upwind Method

The spatially second-order FTCS method is unconditionally unstable. We could go a step back and see if a first-order in space, first-order in time method could be stable. If we discretize by expansion of the spatial derivative to first order, we get two possiblities:

$$\begin{aligned} \frac{\partial u}{\partial x} &\approx \frac{1}{\Delta x}(u_j - u_{j-1}) \qquad \text{upwind finite difference,} \\ \frac{\partial u}{\partial x} &\approx \frac{1}{\Delta x}(u_{j+1} - u_j) \qquad \text{downwind finite difference.} \end{aligned} \tag{III.12.23}$$

These are both so-called one-sided approximations, because they use data only from one side or the other of point $x_j$. Coupling one of these equations with forward Euler time integration yields

$$\begin{aligned} u_j^{(n+1)} &= u_j^{(n)} - \frac{v\Delta t}{\Delta x}\left(u_j^{(n)} - u_{j-1}^{(n)}\right) \qquad \text{upwind,} \\ u_j^{(n+1)} &= u_j^{(n)} - \frac{v\Delta t}{\Delta x}\left(u_{j+1}^{(n)} - u_j^{(n)}\right) \qquad \text{downwind .} \end{aligned} \tag{III.12.24}$$

While these one-sided approximations are (in general terms) less accurate than a centered difference, they exploit the asymmetry of the advection equation: it describes translation with speed $v$. This translation is either to the right (when $v > 0$) or to the left (when $v < 0$). So when $v > 0$, the upwind difference will rely only on information that is behind and at the location of point $x_j$. It thus "follows" the flow direction and thus is named *upwind* (or sometimes *upstream*). If $v < 0$, the *downwind* (or sometimes *downstream*) method follows the flow, as the flow is now going in the opposite direction.

Stability analysis shows that the upwind method is stable for

$$0 \le \frac{v\Delta t}{\Delta x} \le 1 \ , \tag{III.12.25}$$

while the downwind method is stable for

$$-1 \le \frac{v\Delta t}{\Delta x} \le 0 \ , \tag{III.12.26}$$

which just confirms that for $v > 0$ one should use the upwind method and for $v < 0$ the downwind method.

In the above, the demand

$$\alpha = \left|\frac{v\Delta t}{\Delta x}\right| \le 1 \tag{III.12.27}$$

plays a major role. It is a mathematical realization of the physical causality principle – the propagation of information (here via advection) in one time step $\Delta t$ must not jump ahead more than one grid interval of size $\Delta x$.

The demand that $\alpha \le 1$ is usually referred to as the *Courant-Friedrics-Lewy (CFL) condition* . In practise, the CFL condition is used to determine the allowable time step for a spatial grid size $\Delta x$ for a certain accuracy and stability,

$$\Delta t = c_{\mathrm{CFL}} \frac{\Delta x}{|v|} \ , \tag{III.12.28}$$

where $c_{\mathrm{CFL}} \le 1$ is the CFL factor.

#### III.12.3.3 Lax-Friedrich Method

The Lax-Friedrichs method, like FTCS, is first order in time, but second order in space. It is given by

$$u_j^{(n+1)} = \frac{1}{2}\left(u_{j+1}^{(n)} + u_{j-1}^{(n)}\right) - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^{(n)} - u_{j-1}^{(n)}\right) \ , \tag{III.12.29}$$

and, compared to FTCS, has been made stable for $\alpha \leq 1$ (Eq. III.12.27) by using the average of the old result at points $j+1$ and $j-1$ to compute the update at point $j$. This is equivalent to adding a dissipative term to the equations (which damps the instability of the FTCS method) and leads to rather poor accuracy.

#### III.12.3.4 Leapfrog Method

A fully second-order method is the *Leapfrog* Method (also called the midpoint method) given by

$$u_j^{(n+1)} = u^{(n-1)} - \frac{v\Delta t}{\Delta x}\left(u_{j+1}^{(n)} - u_{j-1}^{(n)}\right) \ . \tag{III.12.30}$$

One can show that it is stable for $\alpha < 1$ (Eq. III.12.27). A special feature of this method is that it is *non-dissipative*. This means that any initial condition simply translates unchanged. All modes (in a decomposition picture) travel unchanged, but they do not generally travel at the correct speed, which can lead to high-frequency oscillations that cannot damp (since there is no numerical viscosity).

#### III.12.3.5 Lax-Wendroff Method

The *Lax-Wendroff* method is an extension of the Lax-Friedrich method to second order in space and time. It is given by

$$u_j^{(n+1)} = u_j^n - \frac{v\Delta t}{2\Delta x}\left(u_{j+1}^{(n)} - u_{j-1}^{(n)}\right) + \frac{v^2(\Delta t)^2}{2(\Delta x)^2}\left(u_{j-1}^{(n)} - 2u_j^{(n)} + u_{j+1}^{(n)}\right) \ . \tag{III.12.31}$$

It has considerably better accuracy than the Lax-Friedrich method and is stable for $\alpha \leq 1$ (Eq. III.12.27).

#### III.12.3.6 Methods of Lines (MoL) Discretization

So far we have discussed methods that have discretized the advection equation (and may discretize other PDEs) in both space and time.

The *Method of Lines* is a semi-discrete approach. The PDE is discretized in space using a discretization $\mathcal{D}(u)$. Since the spatial part is now discretized, the remaining equation,

$$\frac{du}{dt} = \mathcal{D}(u) \ , \tag{III.12.32}$$

is now an ODE that can be integrated forward in time with a well-known stable ODE integrator such as Runge-Kutta. This is the method of choice if high-order accuracy in time is required. It also simplifies the implementation of complex equations significantly, since one must worry only about the spatial part of the discretization.

### III.12.4 A Linear Elliptic Equation Example: The 1D Poisson Equation

One of the simplest elliptic equations is the linear Poisson equation for the Newtonian gravitational potential $\Phi$,

$$\nabla^2\Phi = \Delta\Phi = 4\pi G\rho \ . \tag{III.12.33}$$

For simplicity, let us work with a spherically symmetric mass distribution and a 1D spherical equidistant grid. The Laplacian $\Delta$ reduces in spherical symmetry to

$$\begin{aligned}\Delta[\cdot] &= \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}[\cdot]\right) \ , \\ &= \frac{2}{r}\frac{\partial}{\partial r}[\cdot] + \frac{\partial^2}{\partial r^2}[\cdot] \ .\end{aligned} \tag{III.12.34}$$

There are two ways of solving the 1D poisson equation. One relies on the fact that in spherical symmetry $\Phi$ will depend only on the single variable $r$, hence the character of Eq. (III.12.33) changes from PDE to ODE and the equation can be straighforwardly integrated. The other method is more general, does not exploit the ODE character in 1D, and can be applied also to multi-D and non-linear problems.

#### III.12.4.1 Direct ODE Method

We will work with the second-order ODE

$$\frac{d^2}{dr^2}\Phi + \frac{2}{r}\frac{d}{dr}\Phi = 4\pi G\rho \ . \tag{III.12.35}$$

We reduce this second-order equation to two first-order equations by setting

$$\begin{aligned}\frac{d}{dr}\Phi &= z \ , \\ \frac{d}{dr}z + \frac{2}{r}z &= 4\pi G\rho \ .\end{aligned} \tag{III.12.36}$$

This can be straightforwardly integrated using the methods discussed in §III.9 using inner and outer boundary conditions,

$$\begin{aligned}\frac{d}{dr}\Phi(r=0) &= 0 \ , \\ \Phi(r=R_{\rm surface}) &= -\frac{GM(r=R_{\rm surface})}{R_{\rm surface}} \ .\end{aligned} \tag{III.12.37}$$

In practice, one sets first $\Phi(r=0)=0$, integrates out, and then corrects by adding a correction term to obtain the correct outer boundary condition set by the well-known analytic result for the gravitational potential outside a spherical mass distribution. Not that this is, in some sense, equivalent to the analytic calculation of the gravitational potential – it is determined only up to an additive constant whose choice is convention, since only derivatives of the potential have physical meaning.

### III.12.4.2 Matrix Method

In the matrix method, we directly discretize

$$\frac{d^2}{dr^2}\Phi + \frac{2}{r}\frac{d}{dr}\Phi = 4\pi G\rho \ . \tag{III.12.38}$$

With centered differences, we obtain for an interior (non-boundary) grid point $i$

$$\begin{aligned} \left.\frac{\partial}{\partial r}\Phi\right|_i &\approx \frac{1}{2\Delta r}\left(\Phi_{i+1} - \Phi_{i-1}\right) \ , \\ \left.\frac{\partial^2}{\partial r^2}\phi\right|_i &\approx \frac{1}{(\Delta r)^2}\left(\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}\right) \ . \end{aligned} \tag{III.12.39}$$

With this, we get

$$\frac{1}{r_i}\frac{1}{\Delta r}\left(\Phi_{i+1} - \Phi_{i-1}\right) + \frac{1}{(\Delta r)^2}\left(\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}\right) = 4\pi G\rho_i \ . \tag{III.12.40}$$

In the following, we will refer to the left-hand side of the above equation at point $i$ as $f_i$. From the $1/r$ term we note that we either must regularize at $r = 0$ by some kind of expansion or simply stagger the grid so that there is no point exactly at $r = 0$. The latter is easily achieved by moving the entire grid over by $0.5\Delta r$. At the inner boundary, we have $\partial\Phi/\partial r = 0$, so

$$\Phi_{-1} = \Phi_0 \tag{III.12.41}$$

and the finite difference terms in the innermost zone are adjusted to

$$\begin{aligned} \left.\frac{\partial}{\partial r}\Phi\right|_{i=0} &= \frac{1}{\Delta r}(\Phi_1 - \Phi_0) \ , \\ \left.\frac{\partial^2}{\partial r^2}\Phi\right|_{i=0} &= \frac{1}{(\Delta r)^2}(\Phi_1 - \Phi_0) \ . \end{aligned} \tag{III.12.42}$$

Let us press ahead by computing the Jacobian

$$J_{ij} = \frac{\partial f_i}{\partial \Phi_j} \ , \tag{III.12.43}$$

and solving

$$J\boldsymbol{\Phi} = \mathbf{b} \ , \tag{III.12.44}$$

for $\Phi = (\Phi_0, \cdots, \Phi_{n-1})^T$ (for a grid with $n$ points labeled 0 to $n-1$) and with $\mathbf{b} = 4\pi G(\rho_0, \cdots, \rho_{n-1})^T$. After finding the solution to (III.12.44), one corrects for the analytic outer boundary value $\Phi(r = R_{\rm surface}) = -GM/R_{\rm surface}$.

The Jacobian $J$ has tri-diagonal form and can be explicitely given as follows:

(a) $i = j = 0$:

$$J_{00} = -\frac{1}{(\Delta r)^2} - \frac{1}{r_0\Delta r} \ , \tag{III.12.45}$$

(b) $i = j$:

$$J_{ij} = \frac{-2}{(\Delta r)^2} \ , \tag{III.12.46}$$

(c) $i + 1 = j$:

$$J_{ij} = \frac{1}{(\Delta r)^2} + \frac{1}{r_i \Delta r} \ , \tag{III.12.47}$$

(d) $i - 1 = j$:

$$J_{ij} = \frac{1}{(\Delta r)^2} - \frac{1}{r_i \Delta r} \ . \tag{III.12.48}$$

# Chapter IV

# Applications in Astrophysics

# IV.1 Nuclear Reaction Networks

Nuclear reaction networks (NRNs) are a direct application of ODE theory and methods to solve (non-)linear equations. In this section, we will introduce the basics of nuclear reaction networks. A much more detailed discussion can be found, for example, in Arnett's book [1].

Thermonuclear reactions are very sensitive to temperature, since the reactants must have sufficient kinetic energy to have significant probability for quantum tunneling through the Coloumb barrier blocking the reaction. For example, the triple-$\alpha$ reaction

$$3\,^4\mathrm{He} \longrightarrow {}^{12}C + \gamma \ , \tag{IV.1.1}$$

which is the reaction in which helium burns into carbon, has an energy generation rate (which is proportional to the reaction rate) that scales with

$$\epsilon_{3\alpha} \propto \left(\frac{T_8}{2}\right)^{18.5} \tag{IV.1.2}$$

near $T_8 = 2$, where $T_8$ is the temperature measured in $10^8$ K ($T_8 = T/10^8$ K).

Since generally many different reactions are going on at the same time at generally very different rates, the equations governing the complete set of reactions are stiff systems of ODEs.

## IV.1.1 Some Preliminaries and Definitions

- Characterizing Isotopes
  An isotope is characterized by dimensionless integers:
  $Z$ # of protons (the atomic number)
  $N$ # of neutrons
  $A$ $Z + N$ = # of nucleons.

- Avogadro's Number
  $N_A = 6.02214179 \times 10^{23}\,\mathrm{mole}^{-1}$ is the number of entities ("units", nuclei etc.) in one mole of material.

- Atomic Weight
  The *atomic weight* (also called *molar mass*) of one entity with mass $m$ in grams is $W = mN_A\,\mathrm{g\,mole}^{-1}$.

- Atomic Mass Unit
  The *atomic mass unit* is given by

$$\begin{aligned} 1\,m_u = 1\,\mathrm{amu} = \frac{1}{N_A} &= 1.660538782 \times 10^{-24}\,\mathrm{g} \\ &= 931.494061\,\mathrm{MeV/c^2} \ . \end{aligned} \tag{IV.1.3}$$

  ($1\,\mathrm{MeV} = 1.6 \times 10^{-10}\,\mathrm{erg}$). For ${}^{12}C$, $W = 12\,\mathrm{g\,mole}^{-1}$. So 1 amu has then $W = 1\,\mathrm{g\,mole}^{-1}$.

- Isotope Rest Mass
  The isotope rest mass of a single isotope $k$ is

$$\begin{aligned} m_k &= Nm_n + Zm_p + Z(1-f)m_e - \Delta m \ , \\ &= Nm_n + Zm_p + Z(1-f)m_e - \frac{|B_k|}{c^2}\ \mathrm{g} \ . \end{aligned} \tag{IV.1.4}$$

Here $f$ is the ionization fraction and we have neglected the contribution from the electron binding energy, which is usually negligible. We have also neglected the small negative electron binding energy. $\Delta m$ is the *mass deficit*, which describes the amount by which the rest mass of the isotopes is reduced below the sum of the its constituent nucleon rest masses by the nuclear binding energy $B_k$. The molar mass of the isotope is $W_k = m_k N_A$.

The binding energy has a negative sign, i.e. it reduces the rest mass of the isotope in the same way gravitational binding energy reduces the observed mass of neutron stars below their baryonic mass.

- Baryon Mass Density
  For a mixture of isotopes we define the baryon mass density

$$\rho = \frac{\sum_i n_i A_i}{N_A} = \sum_i m_i n_i A_i \tag{IV.1.5}$$

  where $n_i$ is the number density of isotope $i$, the number of entities of this isotope per unit volume.

- Mass Fraction and Molar Fraction

  The *mass fraction* of isotope $i$ is its relative abundance per mass defined as

$$X_i = \frac{A_i n_i}{\rho N_A} \ . \tag{IV.1.6}$$

  And all mass fractions must add up to one: $\sum_i X_i = 1$.

  The *molar fraction* of an isotope is its relative abundance in one mole of material (so it is a number fraction).

$$Y_i = \frac{X_i}{A_i} = \frac{n_i}{\rho N_A} \ . \tag{IV.1.7}$$

  The use of the molar fraction is somewhat historic and comes from the origins of nuclear reaction networks in chemical reaction networks. It makes sense to use it, since nuclear reactions are reactions between entities and depend directly on the number of entities present. Using the molar fraction, while less intuitive than mass fraction or number density, has also the advantage that the energy generation by a nuclear reaction is very easily evaluated:

$$\frac{d\epsilon}{dt} = N_A \sum_i |B_i| \frac{dY_i}{dt} \ , \tag{IV.1.8}$$

  where $\epsilon$ is the specific internal energy per gram in erg $g^{-1}$, $|B_i|$ is the binding energy in units of erg $g^{-1}$, and the $Y_i$ are formally dimensionless.

- Average Nucleus and the Electron Fraction $Y_e$
  We define the average nucleus described by $\overline{A}$ and $\overline{Z}$ and the electron fraction (number of electrons per baryon) by

$$\begin{aligned} \overline{A} &= \frac{\sum_i n_i A_i}{\sum n_i} = \frac{1}{\sum_i Y_i} \ , \\ \overline{Z} &= \frac{\sum_i n_i Z_i}{\sum n_i} = \overline{A} \sum_i Y_i Z_i \ , \\ Y_e &= \frac{\overline{Z}}{\overline{A}} \ . \end{aligned} \tag{IV.1.9}$$

- Mass Excess and Energy Generation
  The *mass excess* of an isotope is defined as

$$\Delta M = \left( \frac{m}{m_u} - A \right) m_u c^2 \ , \tag{IV.1.10}$$

  which is just the difference of its actual mass to its mass in atomic mass units. This difference is due to differences in binding energy between the isotope in question and $^{12}$C.

  For a reaction $1 + 2 \longrightarrow 3$, the liberated energy $Q$ can be expressed in terms of the mass excess as

$$Q = \Delta M_1 + \Delta M_2 - \Delta M_3 \ , \tag{IV.1.11}$$

  which is equivalent to expressing it in terms of the binding energies of the reactants:

$$Q = |B_3| - (|B_1| + |B_2|) \ . \tag{IV.1.12}$$

### IV.1.2 A 3-Isotope Example

Let's consider a nuclear reaction network with three reactants:

$$\begin{array}{rcl} 1+2 & \longleftrightarrow & 3+\gamma \\ 2 & \longrightarrow & 1 \end{array} \tag{IV.1.13}$$

Isotopes 1 and 2 combine to make isotopes 3 and the released energy goes into a photon. Isotope 3 can be photodissociated, making isotopes 1 and 2. Also, isotope 2 is unstable and could decay to isotope 1 under the emission of an electron or positron (which has been emitted from this schematic equation). We also leave out changes in the thermodynamics due to the excess heat generated by the reaction and keep density and temperature fixed.

We can write out the change in the number fraction for isotope 1:

$$\begin{array}{rcll} \dfrac{dn_1}{dt} & = & -n_1 n_2 \langle \sigma v \rangle_{12} & \text{reaction } 1+2 \rightarrow 3 \\ & & +\lambda_{\gamma 3} n_3 & \text{reaction } 3+\gamma \rightarrow 1+2 \\ & & +\lambda_2 n_2 & \text{decay } 2 \rightarrow 1 \ . \end{array} \tag{IV.1.14}$$

The first reaction is a so-called binary reaction and its rate is proportional to the product of the number densities of the reactants. $\langle \sigma v \rangle_{12}$ is the velocity-integrated cross section (units cm$^3$ s$^{-1}$) for the reaction $1 + 2 \rightarrow 3$. This is where nuclear physics and the local thermodynamics, controlling the velocity distribution of the reactants, come in. In the following, we will use simplified notation and define

$$\lambda_{ij} = \langle \rho v \rangle_{ij} \ . \tag{IV.1.15}$$

The second reaction involves $\lambda_{\gamma 3}$, which is the rate for photodissociation of reactant 3. This depends on the temperature and on reactant 3's binding energy. $\lambda_2$ in the decay reaction is the decay rate.

The rate equations in terms of the number density for the other isotopes are

$$\begin{aligned} \frac{n_2}{dt} &= -n_1 n_2 \lambda_{12} + \lambda_{\gamma 3} n_3 - \lambda_2 n_2 \ , \\ \frac{n_3}{dt} &= +n_1 n_2 \lambda_{12} - \lambda_{\gamma 3} n_3 \ . \end{aligned} \tag{IV.1.16}$$

Next we convert to molar fractions using Eq. (IV.1.7):

$$\frac{dn_i}{dt} = \rho N_A \frac{dY_i}{dt} + N_A Y_i \frac{d\rho}{dt} \tag{IV.1.17}$$

and we set $d\rho/dt = 0$ for the purpose of this example.

We can now write out the reaction network for the change in the molar fractions $Y_i$ ($i = 1, 2, 3$):

$$\begin{aligned} f_1 &= \frac{d}{dt}Y_1 = -N_A\rho\lambda_{12}Y_1Y_2 + \lambda_{\gamma 3}Y_3 + \lambda_2 Y_2 \ , \\ f_2 &= \frac{d}{dt}Y_2 = -N_A\rho\lambda_{12}Y_1Y_2 + \lambda_{\gamma 3}Y_3 - \lambda_2 Y_2 \ , \\ f_3 &= \frac{d}{dt}Y_3 = +N_A\rho\lambda_{12}Y_1Y_2 - \lambda_{\gamma 3}Y_3 \ . \end{aligned} \tag{IV.1.18}$$

This system of ODEs is non-linear in the $Y_i$ and if the reaction rates $\lambda_{12}$, $\lambda_{\gamma 3}$, and $\lambda_2$ may be very different, it will be stiff and difficult to solve with standard explicit methods that tend to lead to instability.

Our goal is thus to write an implicit scheme. For simplicity, we will keep it first order in time. So, for evaluating $Y_i(t + \Delta t)$ with the first-order backward Euler scheme, we write

$$Y_i(t + \Delta t) = Y_i(t) + \Delta t f_i(t + \Delta t) \ . \tag{IV.1.19}$$

Furthermore, we denote

$$\Delta_i = \frac{dY_i}{dt}\Delta t \approx Y_i(t + \Delta t) - Y_i(t) \ . \tag{IV.1.20}$$

Since the $f_i$ are nonlinear in $Y_iY_j$, we need to *linearize* the system to find an approximation for $f_i(t + \Delta t)$ that we can use in our integration scheme. We expand

$$f_i(t + \Delta t) \approx f_i(t) + \frac{df_i(t)}{dt}\Delta t = f_i(t) + \sum_j \frac{df_i}{dY_j}\frac{dY_j}{dt}\Delta t = f_i(t) + \sum_j \frac{df_i}{dY_j}\Delta_j \ . \tag{IV.1.21}$$

Using (IV.1.21) in (IV.1.19) with (IV.1.20), we have

$$\Delta_i - \sum_j \frac{df_i}{dY_j}\Delta_j\Delta t = f_i(t)\Delta t \ . \tag{IV.1.22}$$

We can write out the left-hand-side using Eq. (IV.1.18),

$$\begin{aligned} (1 + N_A\rho\lambda_{12}Y_2(t)\Delta t)\Delta_1 + (N_A\rho\lambda_{12}Y_1(t) - \lambda_2)\Delta t\Delta_2 - \lambda_{\gamma 3}\Delta t\Delta_3 &= f_1(t)\Delta t \\ N_A\rho\lambda_{12}Y_2(t)\Delta t\Delta_1 + (1.0 + N_A\rho\lambda_{12}Y_1(t)\Delta t + \lambda_2\Delta t)\Delta_2 - \lambda_{\gamma 3}\Delta t\Delta_3 &= f_2(t)\Delta t \\ -N_A\rho\lambda_{12}Y_2(t)\Delta t\Delta_1 + -N_A\rho\lambda_{12}Y_1(t)\Delta t\Delta_2 + (1 + \lambda_{\gamma 3}\Delta_t)\Delta_3 &= f_3(t)\Delta t \ . \end{aligned} \tag{IV.1.23}$$

And by writing this in matrix form, we obtain

$$\begin{pmatrix} 1 + N_A\rho\lambda_{12}Y_2(t)\Delta t & N_A\rho\lambda_{12}Y_1(t)\Delta t - \lambda_2\Delta t & -\lambda_{\gamma 3}\Delta t \\ N_A\rho\lambda_{12}Y_1(t)\Delta t & 1 + N_A\rho\lambda_{12}Y_1(t)\Delta t + \lambda_2\Delta t & -\lambda_{\gamma 3}\Delta t \\ -N_A\rho\lambda_{12}Y_2(t)\Delta t & -N_A\rho\lambda_{12}Y_1(t)\Delta t & 1 + \lambda_{\gamma 3}\Delta_t \end{pmatrix} \begin{pmatrix} \Delta_1 \\ \Delta_2 \\ \Delta_3 \end{pmatrix} = \begin{pmatrix} f_1(t)\Delta t \\ f_2(t)\Delta t \\ f_3(t)\Delta t \end{pmatrix} \ , \tag{IV.1.24}$$

which can be solved for the $\Delta_i$ needed for our update,

$$Y_i(t+\Delta t) = Y_i(t) + \Delta_i \ . \tag{IV.1.25}$$

### IV.1.3 Reactant Multiplicity

A more complex situation arises when multiple entities of the same reactant are involved in a reaction. For example,

$$^{12}\mathrm{C} + {}^{12}\mathrm{C} \longrightarrow {}^{24}\mathrm{Mg} + \gamma \ . \tag{IV.1.26}$$

So for each $^{24}$Mg nucleus we make, two $^{12}$C nuclei are needed. For simplicity, we will work with number densities in the following. We can always convert to molar fractions once the reaction equations have been derived. Naively, we would now assume that the change in the number density was given by

$$\frac{dn_{^{12}\mathrm{C}}}{dt} = -2n_{^{12}\mathrm{C}}^2 \lambda_{^{12}C^{12}C} \ , \tag{IV.1.27}$$

which, however is not quite right, since reactions between each pair of nuclei are counted $2! = 2$ times ($N!$ is the number of possible permutations that would lead to an identical result when $N$ nuclei are involved). So we really have only

$$\frac{dn_{^{12}\mathrm{C}}}{dt} = -\frac{2}{2!} n_{^{12}\mathrm{C}}^2 \lambda_{^{12}C^{12}C} \ . \tag{IV.1.28}$$

So for an a bit more general binary reaction $N_i\, i + N_j\, j \rightarrow 1\, k$, we have

$$\begin{aligned}
\frac{dn_i}{dt} &= -\frac{|N_i|}{|N_i|!\,|N_j|!} n_i n_j \lambda_{ij} \ , \\
\frac{dn_j}{dt} &= -\frac{|N_j|}{|N_i|!\,|N_j|!} n_i n_j \lambda_{ij} \ , \\
\frac{dn_k}{dt} &= +\frac{1}{|N_i|!\,|N_j|!} n_i n_j \lambda_{ij} \ .
\end{aligned} \tag{IV.1.29}$$

This is straightforwardly extended to triple reactions. For example, for the triple-$\alpha$ reaction given in Eq. (IV.1.1) we have:

$$\begin{aligned}
\frac{dn_{^4\mathrm{He}}}{dt} &= -\frac{3}{3!} n_{^4\mathrm{He}}^3 \lambda_{3\alpha} \ , \\
\frac{dn_{^{12}\mathrm{C}}}{dt} &= +\frac{1}{3!} n_{^4\mathrm{He}}^3 \lambda_{3\alpha} \ .
\end{aligned} \tag{IV.1.30}$$

### IV.1.4 Open Source Resources

Fortunately, a large number of nuclear reaction networks and nuclear data and reaction rate sets are available as open source / open physics.

- Frank Timmes (a nuclear astrophysicist at Arizona State University) provides an assortment of state-of-the-art reaction networks at http://cococubed.asu.edu/code_pages/burn.shtml.
- The Modules for Explorations in Stellar Astrophysics (MESA) code comes with extensive nuclear reaction networks. http://mesa.sourceforge.net.

- The Clemson University nuclear astrophysics group provides reaction networks and online tools, e.g., a calculator for abundances in nuclear statistical equilibrium, at http://www.webnucleo.org.
- The Joint Institute for Nuclear Astrophysics (JINA), a National Science Foundation Physics Frontier Center, provides lots of resources on nuclear reactions as well as a data base of isotope masses and reaction rates. http://www.jinaweb.org.

## References

[1] D. Arnett. *Supernovae and nucleosynthesis. an investigation of the history of matter, from the Big Bang to the present*. Princeton Series in Astrophysics, Princeton, NJ: Princeton University Press, 1996.

# IV.2 N-body Methods

We will begin by exploring the gravitational N-body problems and algorithms that have been developed to solve it efficiently.

## IV.2.1 Specification of the Problem

Frequently in astrophysics we have systems of "particles" which interact only gravitationally. Examples include dark matter and purely stellar systems (in which each star is a particle). Ignoring relativistic effects (which we can in a wide variety of situations) the evolution of such systems is entirely specified by the Newtonian $1/r^2$ law of gravity, such that the acceleration of particle $i$ is given by:

$$\ddot{\mathbf{x}}_i = \sum_{j=1;j\neq i}^{N} -\frac{Gm_j}{|\mathbf{x}_{ij}|^2}\hat{\mathbf{x}}_{ij}, \tag{IV.2.1}$$

where $\mathbf{x}$ is position, $\mathbf{x}_{ij} \equiv \mathbf{x}_i - \mathbf{x}_j$ and a hat indicates a unit vector.

### IV.2.1.1 How Big Must $N$ Be?

We typically want $N$, the number of particles, to be as large as possible given available computational resources. When simulating a system of fixed total mass, increasing $N$ will allow each particle to have a smaller mass and, therefore, will allow the simulation to resolve structure on smaller scales (and more closely approach the idealized fluid that we're attempting to simulate). The number of particles used will also affect how long we can run our simulation for, before the discrete nature of our particle representation becomes a problem. Imagine an isolated, self-gravitating system of particles that is in virial equilibrium (such as a galaxy). In the limit of an infinite number of particles, the potential of this system will be unchanging in time and so each particle will conserve its total energy as it orbits through the system. However, if there are a finite number of particles, the potential will no longer remain constant in time, instead fluctuating as particles move around. This will allow for energy exchange between particles. If our simulation contains fewer particles than the real system (which it almost always will do) this energy exchange is unphysical. In a non-gravitating system this would eventually lead to equipartition and thermal equilibrium. Gravitating systems, because of their negative specific heat, have no thermal equilibrium and this process will eventually lead to the phenomenon of "core collapse" (observed in globular clusters) in which the core of the system collapses to arbitrariily high densities while a diffuse envelope of material is ejected to large radii.

To estimate how long we can run a simulation before this relaxation process becomes important, we want an order-of-magnitude estimate how long it takes for encounters with other stars to significantly change the energy of a star. Typically we are interested in collisionless systems in which the acceleration of a given particle never receives a dominant contribution from a single other particle, instead being determined by the combined effects of many particles. Consider an encounter between two stars. Assuming the collision is a small perturbation to the motion of the star we can approximate the force experienced by the star as:

$$\dot{\mathbf{v}}_\perp = \frac{Gm}{b^2+x^2}\cos\theta = \frac{Gmb}{(b^2+x^2)^{3/2}} \approx \frac{Gm}{b^2}\left[1+\left(\frac{vt}{b}\right)^2\right]^{-3/2}. \tag{IV.2.2}$$

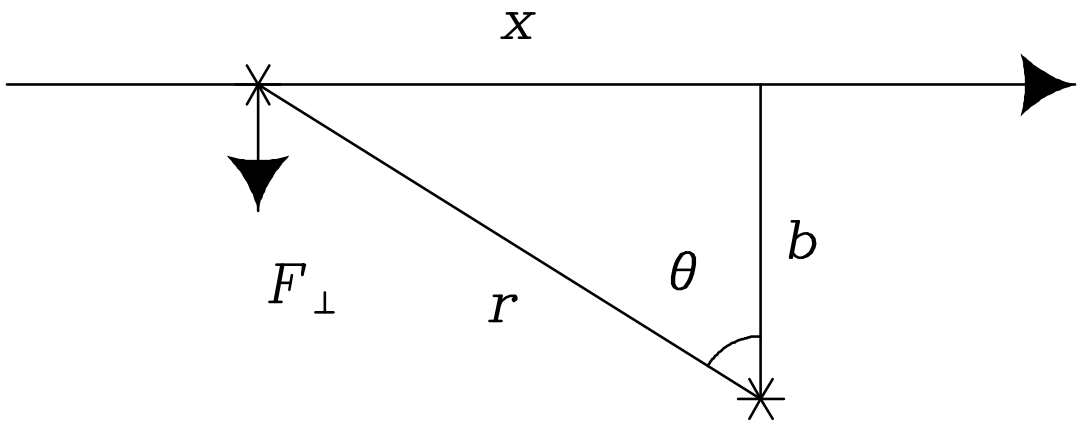



Figure IV.2.1: Geometry of a gravitational collision between two stars.

Integrating over the entire collision gives us the change in velocity of the star:

$$\mathbf{v}_\perp \approx \frac{Gm}{bv}\int_{-\infty}^{\infty}(1+s^2)^{-3/2}\mathrm{d}s = \frac{2Gm}{bv}, \tag{IV.2.3}$$

which is approximately the force at closest approach times the duration of the encounter, $b/v$. The surface density of stars in a galaxy is of order $N/\pi R^2$, so in crossing the galaxy once, the star experiences

$$\delta n = \frac{N}{\pi R^2}2\pi b\mathrm{d}b = \frac{2N}{R^2}b\mathrm{d}b, \tag{IV.2.4}$$

encounters with impact parameter between $b$ and $b+\mathrm{d}b$. The encounters cause randomly oriented changes in velocity, so $\overline{\delta\mathbf{v}_\perp}=0$, but there can be a net change in $v_\perp^2$:

$$\delta v_\perp^2 \approx \left(\frac{2Gm}{bv}\right)^2\frac{2N}{R^2}b\mathrm{d}b. \tag{IV.2.5}$$

Our perturbation approach breaks down if $\delta v_\perp \sim v_\perp$ which occurs if $b \lesssim b_{\min} = Gm/v^2$, so integrating over all impact parameters[1] from $b_{\min}$ to $R$ (the largest possible impact parameter):

$$\Delta v_\perp^2 = \int_{b_{\min}}^{R}\delta v_\perp^2 \approx 8N\left(\frac{Gm}{Rv}\right)^2\ln\Lambda, \tag{IV.2.6}$$

where $\Lambda = R/b_{\min}$ ("Coulomb logarithm"). The typical speed of a star in a self-gravitating galaxy is

$$v^2 \approx \frac{GNm}{R}. \tag{IV.2.7}$$

[1]For a more careful treatment of small $b$ encounters, take a look at the treatment of dynamical friction, §7.1 of Binney & Tremaine.

Therefore, we find

$$\frac{\Delta v_\perp^2}{v^2} = \frac{8 \ln \Lambda}{N}, \tag{IV.2.8}$$

and the number of crossings required for order unity change in velocity is:

$$n_{\rm relax} = \frac{N}{8 \ln \Lambda}. \tag{IV.2.9}$$

Since $\Lambda = R/b_{\rm min} \approx Rv^2/Gm \approx N$ we find $t_{\rm relax} \approx [0.1N/\ln N] t_{\rm cross}$.

Relaxation is important for systems up to globular cluster scales, but is entirely negligible for galaxies.

| System | $N$ | $t_{\rm relax}/t_{\rm cross}$ | $t_{\rm relax}$ |
|---|---|---|---|
| Small stellar group | 50 | 1.3 | |
| Globular cluster | $10^5$ | 870 | $10^8$yr |
| Galaxy | $10^{11}$ | $4 \times 10^8$ | $4 \times 10^7$Gyr |

It's worth taking a moment to consider what we're actually doing in an N-body calculation. Specifically, representing a fluid in 6-D phase space by a set of discrete points. We associate a mass with each of those points (usually the same mass for each point) such that we can use the points to compute the density distribution of the fluid and therefore its gravitational potential. Additionally, we move the points in phase space according to that same gravitational potential. As such, there are fundamentally two roles for the points: to represent the mass distribution at any time and to flow as would the actual fluid such that they continue to represent the mass distribution at later times. For most systems of interest (e.g. galaxies, dark matter halos) the number of particles we can use (given current computational techniques and resources) is much less than the number of particles in the real system. For example, a galaxy may contain $10^{11}$ stars, but the best current simulations of galaxies contain only $\sim 10^6$. Similarly, the best current simulations of dark matter halos contain around $10^9$ particles, but the Milky Way's halo contains maybe $10^{70}$ dark matter particles. Therefore, the relaxation times in simulations will be much shorter than in the real systems (particularly in dense regions) unless we do something to mitigate this problem.

We can break down the N-body problem into three sub-problems:

(1) Specifying initial conditions;

(2) Computing forces/accelerations on particles;

(3) Stepping particles forward in time.

### IV.2.2 Force Calculation

The N-body problem is fundamentally an $\mathcal{O}(N^2)$ problem—direct summation over all particles involves a number of calculations that increases as the square of the number of particles. Not surprisingly therefore, force computation is typically the slowest step in any N-body calculation and so numerous techniques have been devised to speed this up. We'll review these techniques, as well as the direct summation approach (since it's still useful in some cases).

#### IV.2.2.1 Direct Summation (Particle-Particle)

The direct summation approach is inherently simple—as we wrote already, given some set of $N$ particles we evaluate the force on each one using

$$\ddot{\mathbf{x}}_i = \sum_{j=1;j\neq i}^{N} -\frac{Gm_j}{|\mathbf{x}_{ij}|^2}\hat{\mathbf{x}}_{ij}. \tag{IV.2.10}$$

You can write yourself a direct summation code in a few lines. If you have a newish desktop computer with a GPU (Graphics Processing Unit) you can write a direct summation code which can handle $\gtrsim 10^4$ particles in a reasonable amount of time. Specialized hardware (GRAvity PipE or GRAPE boards) can handle several million—enough to simulate a globular cluster at one particle per star. The direct summation approach is obsolete for galactic and cosmological simulations.

In cases where we don't have one particle per star/particle two-body encounters between particles will be much more common in our simulation than in the real system. This can lead to effects (for example, binary formation) which would not occur in the real system on relatively short timescales. To circumvent this problem we recognize that each particle is better thought of as representing an extended distribution of mass. It will therefore not have a point mass potential. Instead, its potential will be "softened". We might therefore modify the force law to be

$$\ddot{\mathbf{x}}_i = \sum_{j=1;j\neq i}^{N} -\frac{Gm_j}{(|\mathbf{x}_{ij}|+\epsilon)^2}\hat{\mathbf{x}}_{ij}, \tag{IV.2.11}$$

where $\epsilon$ is a length scale (called the *softening length*) of order the "size" of the particle and which limits the maximum force between two particles. Note that this doesn't do much to change the relaxation time—the Coulomb logarithm term in our derivation of the relaxation time shows that the relaxation process gains equal contributions from particles in each logarithmic interval of radius, and so softening the force on small scales only slightly reduces the rate of relaxation.

Different functional forms have been used for the softening. For example, a common approach has been to represent particles by so-called Plummer spheres—density distributions of the form $\rho(x) \propto (1+x^2)^{-5/2}$ (where $x = r/\epsilon$) which have a potential $\Phi(x) \propto 1/\sqrt{\epsilon^2+x^2}$, such that

$$\ddot{\mathbf{x}}_i = \sum_{j=1;j\neq i}^{N} -\frac{Gm_j|\mathbf{x}_{ij}|}{(|\mathbf{x}_{ij}|^2+\epsilon^2)^{3/2}}\hat{\mathbf{x}}_{ij}. \tag{IV.2.12}$$

Another common approach is to use a cubic spline density distribution of the form:

$$\rho(x) \propto \begin{cases} 4-6x^2+3x^3 & \text{for } x<1 \\ (2-x)^3 & \text{for } 1\leq x<2 \\ 0 & \text{for } x\geq 2. \end{cases} \tag{IV.2.13}$$

This form has the advantage that the density distribution is truncated (i.e. goes to zero beyond $x=2$) and so the force law becomes precisely Newtonian at $2>2\epsilon$. In general, softening kernels of this type (known as compact kernels) are superior to non-compact kernels (e.g. Plummer).

Choosing a value for the softening length is something of an art form. It shouldn't be too large as any structure on scales smaller than the softening length will be smoothed away. However, it shouldn't be too small, or two-body collisional effects will become important again. A good discussion of choosing softening lengths is given by [3], who explores further refinements to this idea, such as compensating the reduced forces on small scales by slightly enhancing forces on larger scales and the possibility of adapative softening lengths (i.e. making $\epsilon$ a function of, for example, local density).

#### IV.2.2.2 Particle-Mesh

The fundamental limitation of the direction summation technique is that it is $\mathcal{O}(N^2)$. So, let's explore some ways to reduce the computational load. One of the first ideas was to compute forces by solving Poisson's equation rather than by direct summation. Suppose we have a density field $\rho(\mathbf{x})$. Poisson's equation tells us that the gravitational potential is related to this density field by

$$\nabla^2\Phi(\mathbf{x}) = 4\pi G\rho(\mathbf{x}). \tag{IV.2.14}$$

If we can solve this equation, then the acceleration on particle $i$ can be found from the potential using

$$\ddot{\mathbf{x}}_i = -\boldsymbol{\nabla}\Phi(\mathbf{x}_i). \tag{IV.2.15}$$

In particular, if we know the density field on a uniform grid then we can use Fourier transforms (in particular, Fast Fourier Transforms) to quickly solve these equations. If we represent the Fourier transform of some quantity $q$ on our grid as

$$q_i = \sum_{\mathbf{k}_j} \widetilde{q}_j \exp(i\mathbf{k}_j\cdot\mathbf{x}_i), \tag{IV.2.16}$$

where the sum is taken over all wavenumbers $\mathbf{k}_j$ then from Poisson's equation we have

$$\sum_{\mathbf{k}_j} k_j^2\ \widetilde{\Phi}_j \exp(i\mathbf{k}_j\cdot\mathbf{x}_i) = 4\pi G\sum_{\mathbf{k}_j}\widetilde{\rho}_j\exp(i\mathbf{k}_j\cdot\mathbf{x}_i), \tag{IV.2.17}$$

which can only hold in general if

$$-k_j^2\ \widetilde{\Phi}_j = 4\pi G\ \widetilde{\rho}_j\ . \tag{IV.2.18}$$

Thus, to compute the potential, we proceed as follows:

(1) Find the density field on a grid;

(2) Compute its Fourier transform;

(3) Multiply each Fourier component by $-4\pi G/k_j^2$;

(4) Take the inverse Fourier transform.

We can actually skip the final step since we're interested in the force on each particle which is given by

$$\widetilde{\ddot{\mathbf{x}}}_j = -i\mathbf{k}_j\ \widetilde{\Phi}\ . \tag{IV.2.19}$$

So, we simply multiply the potential by $-i\mathbf{k}_j$, take the inverse Fourier transform and are left with the particle accelerations on a grid. We can interpolate accelerations to the precise location of each particle if necessary.

The density field is, of course, constructed from the particle distribution. There are numerous ways to do this. The most common are nearest grid point (NGP), cloud-in-cell (CIC) and triangular-shaped cloud (TSC) methods which are illustrated in Figure. IV.2.2.

Particle-mesh algorithms of this sort are $\mathcal{O}(N + N_g\log N_g)$ where $N_g$ is the number of grid points and can therefore be substantially faster than direct summation techniques. Their main

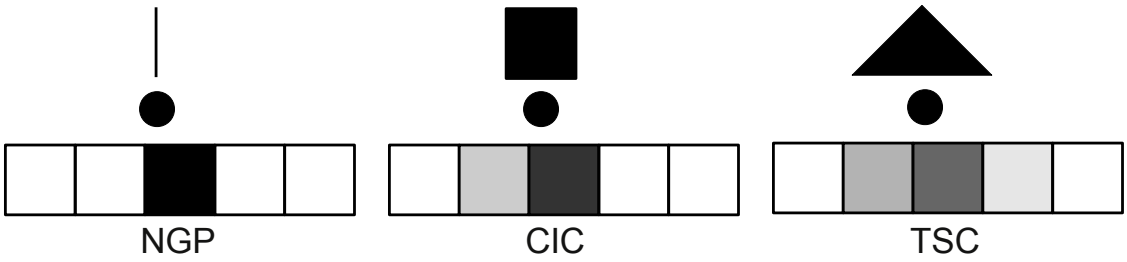



Figure IV.2.2: Methods for assigning particles to grid cells. This illustrates the 1-D case—the generalization to two additional dimensions is straightforward.

limitation is that information about the density distribution (and, therefore, the potential and acceleration fields) on scales smaller than the size of a grid cell are lost. Consequently, structures on smaller scales will be lost or fail to form. This loses one of the nice features of the N-body approach, namely that it puts the resolution where it's needed. More elaborate techniques, using nested grids of different resolution can be used to help mitigate this limitation.

#### IV.2.2.3 Particle-Particle/Particle-Mesh (P3M)

This approach aims to combine the advantages of direct summation (no loss of small scale resolution) and particle-mesh (fast) techniques. Briefly, it computes the forces by dividing contributions from particles nearby and far away. For particles that are "far away" (typically more than about 3 grid cell lengths away) the contribution to the force is computed using the particle-mesh technique. For particles that are closer by a direct summation is used. (Frequently this means doing the full particle-mesh calculation and then, for each nearby particle, subtracting the force it contributed in the particle-mesh approximation before adding its contribution using the direct summation approach). The biggest problem with this approach is that it can easily become dominated by the direct summation (particle-particle) part of the calculation and become about as slow as direct summation. This will happen in any simulation where particles become strongly clustered (and, unfortunately, gravity is attractive...).

#### IV.2.2.4 Tree Algorithms

Tree algorithms take a rather different approach. In a "top down" approach, the entire simulation volume (typically a cubic region, or, in the 2-D example shown in Fig. IV.2.3, a square) is placed into the top level cell. The next level in the tree is created by splitting this cell in half in each dimension such that the $2^{nd}$ level contains 8 cells (in 3-D; a so-called oct-tree) or 4 cells (in 2-D; a quad-tree). Further tree levels are made by repeatedly splitting cells in the next higher level in this way until each cell contains only a single particle. The resulting tree structure is illustrated in Fig. IV.2.4. In the original Barnes-Hut [1] tree algorithm, the center of mass of the particles in each cell at each level is computed. Then, to compute the force on any given particle, one simply computes the contribution due to the total mass of particles located at the center of mass of a cell. For nearby particles, we will use the finest levels of the tree to accurately determine the forces. For more distant particles we can use a coarser level of the tree and therefore compute the force due to many particles in one go. In this way, the calculation can be made faster than direct summation

while retaining good spatial resolution on small scales. The tree approach has an advantage over particle-mesh techniques in that doesn't waste time on empty regions of the simulation volume (useful if one is simulating the collision of two galaxies for example). However, there is a memory overhead associated with constructing and storing the tree structure itself.

The original Barnes-Hut tree algorithm computed forces directly by treating particles in each cell as a monopole (i.e. a point mass). More recent algorithms work with the potential instead and account for higher order moments of the particle distribution in each cell. Essentially, the potential due to particles in a cell is described by a multipole expansion—more or less like expanding a function as a Taylor series. The more terms we include, the more accurate the representation (but the slower the calculation), and the functional forms used are easily differentiable making it simple to compute the force from the potential. This method is more accurate than the simple Barnes-Hut method, but is computationally more expensive if higher order multipoles are used.

How do we determine which level of the tree we should use to compute the force on any given particle? First, keep in mind that we're always going to have a trade off between speed and accuracy with the tree algorithm. If we decide to use the tips of each branch (call them "leaves") then we have precisely one particle per cell and we would have effectively the particle-particle algorithm, which would be very accurate, but very slow (slower than the original particle-particle algorithm because we wasted a whole lot of time building the tree!). Alternatively, if we used the base of the tree (call it the "trunk") then the calculation would be extremely fast, but very inaccurate, as we'd only consider the interaction of each particle with the center of mass of the distribution. Obviously, we want something in between.

The usual approach is to adopt an "opening angle criterion". Consider the circled particle in Fig. IV.2.5. We want to compute the force on it due to all other particles. We can begin at the "trunk" of the tree and ask the following questions: Is the ratio of the size of the tree cell at this level divided by the distance from the particle to the center of mass of the cell less than some angle $\theta$? More specifically, we ask if

$$\frac{d}{r} < \theta \tag{IV.2.20}$$

where $d$ is the size of the tree cell and $r$ is the distance from the particle in question to the center of mass of this cell. We then proceed as follows:

(1) If the opening angle condition is satisfied, then compute the force from all particles in this tree cell by treating them as a single particle at the center of mass of the cell.

(2) Otherwise, walk along the tree branches to the next most refined level of the tree and recheck the opening angle condition.

In this way, as we get closer to a set of particles we will use more refined branches of the tree to compute the force from them. The parameter $\theta$ is a parameter that we can tune—making it smaller will make for a more accurate but slower calculation, while making it larger will reduce accuracy and increase speed. Typically, values of $\theta \lesssim 1$ are found to work quite well (although this depends on the details of the algorithm, such as whether or not higher order multipoles of the gravitational potential are included or not). A tree algorithm with this type of opening angle criterion is relatively easy to code as a recursive function, which simply walks along the branches of the tree accumulating forces from sets of particles as it goes and truncating the walk along each branch once the opening angle has been specified. From a coding point of view, this makes for a compact and elegant algorithm.

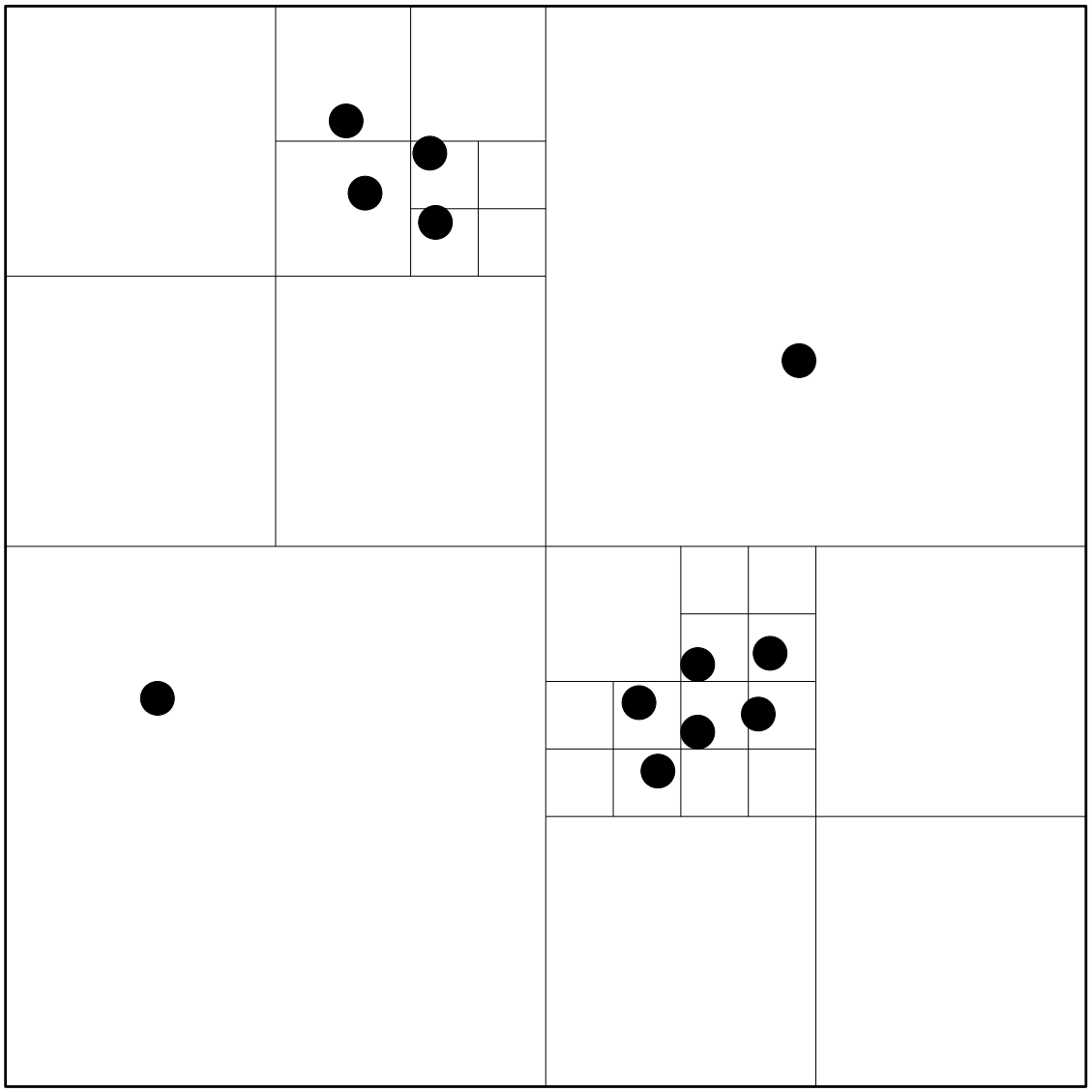


Figure IV.2.3: Example of the construction of a quad-tree.

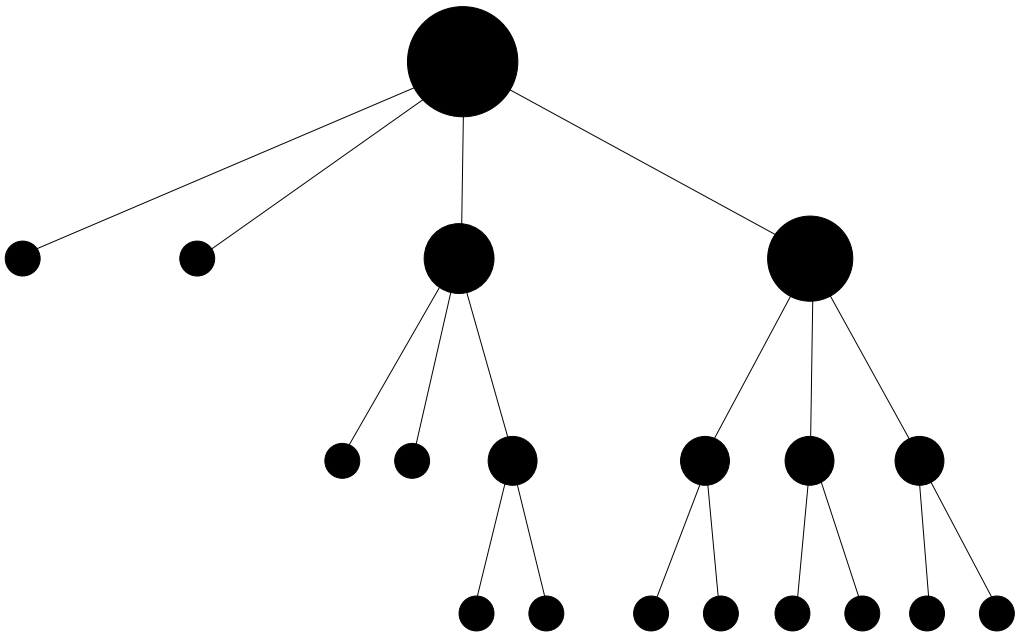


Figure IV.2.4: The tree structure resulting from Fig. IV.2.3.

Why this opening angle criterion and not some other criterion? We can think about the physics the underlies it. Consider the mass distribution in a tree cell. We can represent any density distribution as a sum over various multipoles. Therefore, we can write the gravitational potential for this mass distribution as a multipole expansion in spherical coordinates

$$\Phi(r,\theta,\phi) = \sum_{l=0}^{\infty}\sum_{m=0}^{l}\left(A_l r^l + B_l r^{-l-1}\right)\left[C_{lm}Y_{lm}^c(\theta,\phi) + S_{lm}Y_{lm}^s(\theta,\phi)\right], \tag{IV.2.21}$$

where $A_l$, $B_l$, $C_{lm}$ and $S_{lm}$ are coefficients determined by the mass distribution and the $Y_{lm}$'s are the sine and cosine real spherical harmonics. (This is actually a Laplace series, valid outside the mass distribution—it is applicable to any potential which satisfies the Laplace equation $\nabla^2\Phi = 0$.) Since we expect the gravitational potential to decline with distance from the source then $A_l = 0$ for all $l$. Then we see that the potential due to the $l^{\text{th}}$ multipole declines with distance as $r^{-l-1}$ (e.g. for a monopole distribution, $l = 0$, the usual $r^{-1}$ relation is obtained). Therefore, higher order multipole contributions to the potential will be reduced by a factor of approximately $(d/r)^l$ relative to the monopole contribution if $d$ is the characteristic size of the mass distribution. By ensuring that $d/r < \theta$ we guarantee that, for suitably small $\theta$, all terms beyond the monopole will be negligible.

### IV.2.3 Timestepping Criteria

Having computed the force acting on each of our particles, we need to evolve the particle distribution forward in time. This amounts to solving the following differential equations

$$\dot{\mathbf{x}}_i = \mathbf{v}_i \tag{IV.2.22}$$

$$\dot{\mathbf{v}}_i = \mathbf{F}_i/m_i \tag{IV.2.23}$$

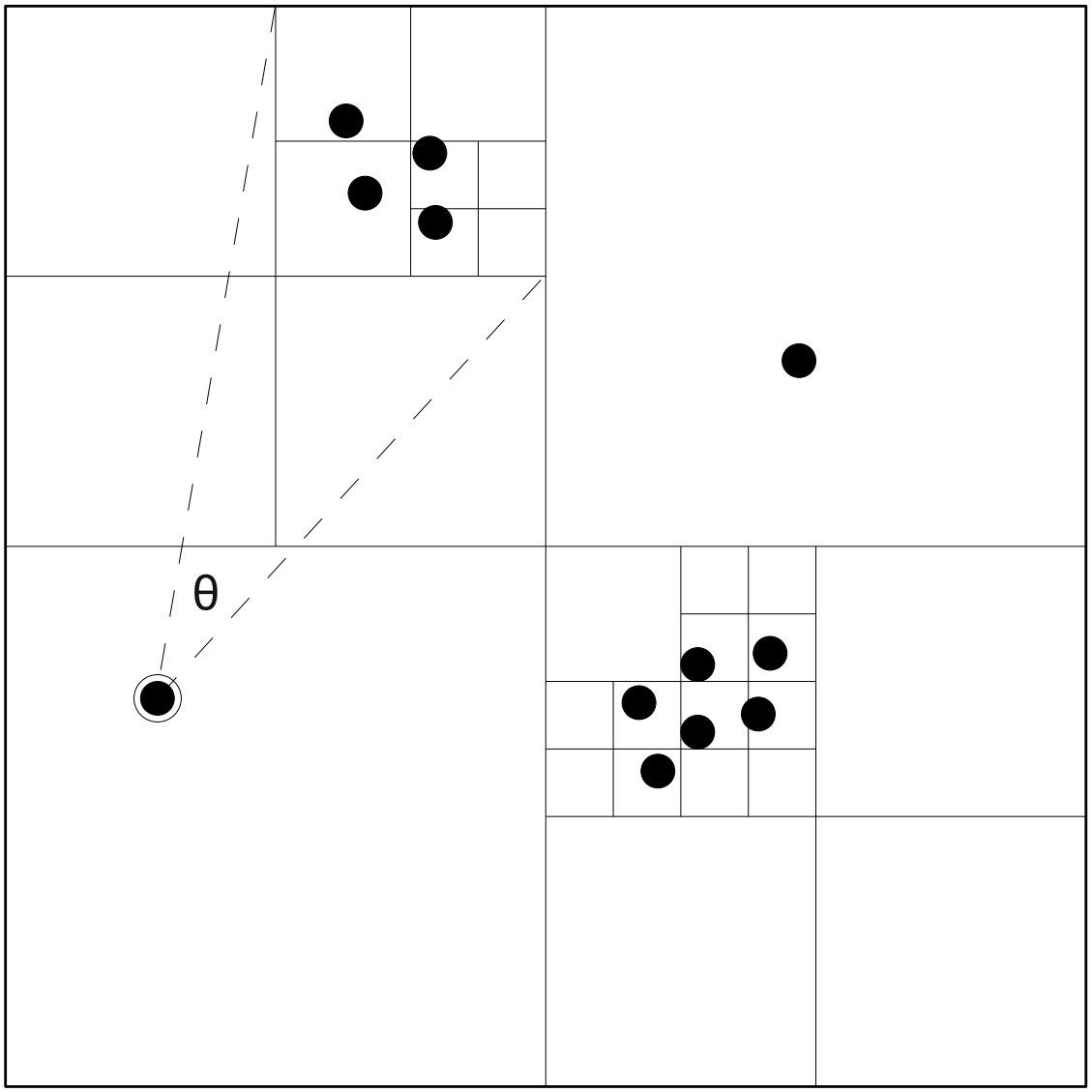



Figure IV.2.5: Opening angle criterion.

where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle $i$, $\mathbf{F}_i$ is the force acting on that particle and $m_i$ is its mass. Naively, we could choose some suitably small timestep, $\delta t$, and assume that the force is approximately constant over that period such that

$$\mathbf{x}_i \rightarrow \mathbf{x}_i + \mathbf{v}_i \delta t + \frac{1}{2}\mathbf{F}_i/m_i \delta t^2 \tag{IV.2.24}$$

$$\mathbf{v}_i \rightarrow \mathbf{v}_i + \mathbf{F}_i/m_i \delta t. \tag{IV.2.25}$$

Two problems immediately arise: First, how small should we make $\delta t$ such that our approximation will be valid. Second, since characteristic timescales in gravitating systems scale as $\rho^{-1/2}$, the densest regions of our simulation will evolve on much shorter (i.e. orders of magnitude shorter) timescales so they'll force us to take very short timesteps. The second of these suggests that we perhaps want to have different timesteps for different particles (no problem in principle, but we have to correctly synchronize particles). The first suggests that we figure out some criterion to judge how small the timestep needs to be.

To make having individual timesteps for each the particles computationally manageable it's often useful to enforce the timesteps to by binary fractions of some overall timestep, $T_0$. That is, each particle will have a timestep $T_0/2^n$ for some integer $n$. Thus, a particle with $n = 1$ will take two steps for every one step taken by a particle with $n = 0$, while a particle with $n = 2$ will take four steps over the same time. The advantage of this approach is that we can easily maintain synchronization between particles (i.e. bring them all to the same time). To ensure sufficiently accurate evolution we then enforce

$$\delta t = \frac{T_0}{2^n} < \eta \frac{1}{\sqrt{G\rho}}, \tag{IV.2.26}$$

where $\rho$ is an estimate of the locally enclosed density. For a particle orbiting at radius $r$ in a spherical potential for example, $\rho \sim M(r)/r^3$ if $M(r)$ is the mass inside radius $r$. Here, $\eta$ is a numerical parameter that we can adjust to control the size of timesteps. We want timesteps small enough that any results we extract from the calculation are converged (i.e. unaffected by timestep size) but large enough that the calculation completes in a reasonable amount of time.

For specificity, we'll consider the time integration algorithm used in the original GADGET N-body code [8]. There are many other (and better) choices, but most share similar features (see, for example, Springel 9, Zemp et al. 10). GADGET-1 updates particle positions and velocities by first predicting the position of a particle at the middle of a timestep $\delta t$:

$$\mathbf{x}^{(n+\frac{1}{2})} = \mathbf{x}^{(n)} + \mathbf{v}^{(n)} \frac{\delta t}{2}, \tag{IV.2.27}$$

where the superscripts refer to position/velocity at step $n$. This mid-step position is used to find the acceleration of the particle

$$\mathbf{a}^{(n+\frac{1}{2})} = -\nabla\Phi\big|_{\mathbf{x}^{(n+\frac{1}{2})}}. \tag{IV.2.28}$$

The particle is then advanced to the next timestep using

$$\mathbf{v}^{(n+1)} = \mathbf{v}^{(n)} + \mathbf{a}^{(n+\frac{1}{2})} \delta t \tag{IV.2.29}$$

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \frac{1}{2}\left[\mathbf{v}^{(n)} + \mathbf{v}^{(n+1)}\right] \delta t. \tag{IV.2.30}$$

Note that we use the mean velocity through the timestep when updating the position. This is a variant of the so-called "leap-frog" method. Energy (kinetic plus potential) should always be

conserved along the particle's path, but in practice it won't be precisely since we take a finite timestep. It can be shown that the error in energy for the above time integrator is

$$\Delta E = \frac{1}{4}\frac{\partial^2\Phi}{\partial x_i \partial x_j} v_i^{(n)} a_j^{(n+\frac{1}{2})} \delta t^3 + \frac{1}{24}\frac{\partial^3\Phi}{\partial x_i \partial x_j \partial x_k} v_i^{(n)} v_j^{(n)} v_k^{(n)} \delta t^3 + \mathcal{O}(\delta t^4). \tag{IV.2.31}$$

That is, this integrator is second order accurate (i.e. the first time in the above is third order in $\delta t$). In principle, the above could be used to choose $\delta t$ to keep the error in energy below some predefined level. In practice this is not very useful as determining the gradients of the potential is expensive and we don't know for sure that the higher order terms are not important. Instead, GADGET-1 uses a criterion similar to that given above using a local estimate of the density. This local estimate is found by averaging over some number of nearby particles (similar to the approach used for smoothed particle hydrodynamics as we'll discuss later). This can be problematic in low density regions (where we have to go to large distances to find neighboring particles) and in dense regions resolved by only a small number of particles. For example, consider a cosmological simulation in which a low mass dark matter halo forms from 20 particles. Suppose we are using the 32 nearest particles to estimate the local density. In this case, the particles in the halo will have their local density estimate contaminated by particles that are not in the halo. As a result, the density estimate will be too low, the resulting timestep too long and so the halo will dissipate because of accumulating errors in energy conservation.

Therefore, GADGET-1 adds a second criterion

$$\delta t < \eta_{\mathrm{a}} \frac{\sigma}{|\mathbf{a}|}, \tag{IV.2.32}$$

where $\eta_{\mathrm{a}}$ is an adjustable parameter and $\sigma$ is an estimate of the local velocity dispersion (also found by averaging over nearby particles). These prevents the velocity changing by more than a fixed fraction of the local velocity dispersion in a given timestep. Consider again the case of the low mass dark matter halo described above. If we estimate the local velocity dispersion from the 32 nearest neighbors, then we'll underestimate it due to the contamination from particles outside the halo. In this case, however, this makes our timestep shorter, and so our integration more accurate. By combining these two criteria it's possible to maintain good energy conservation through a simulation while minimizing the number of timesteps that have to be taken.

### IV.2.4 Initial Conditions

So far we've considered how to compute the forces and evolve the particle positions/velocities for some set of particles for which we already know the position and velocity of each particle. But, we have to begin somewhere. Specifically, we need to know the positions and velocities of the particles at some initial time. These are the initial conditions of the calculation. Generally speaking, the goal of creating initial conditions is to set up the particles to represent some physical system while minimizing any numerical effects due to discreteness or assumptions made. We'll consider two scenarios which are common in the world of N-body simulation: an equilibrium system of dark matter, and the early Universe.

#### IV.2.4.1 Equilibrium Dark Matter Halo

(Note that this could just as easily by an equilibrium system of stars, i.e. a galaxy.) A not uncommon N-body experiment is to see what happens when two systems, e.g. dark matter halos, merge.

To do this, we typically want to create isolated N-body representations of the halos when they are well separated and in internal equilibrium.

For cosmological halos in a cold dark matter universe, the Navarro-Frenk-White profile [6] is a good description of the run of density with radius. This profile has the form

$$\rho(r) = \frac{\rho_0}{(r/r_\mathrm{s})\left[1+(r/r_\mathrm{s})\right]^2}, \tag{IV.2.33}$$

where $\rho_0$ is a normalization and $r_\mathrm{s}$ is a characteristic radius known as the scale radius. We can select values for these constants based on cosmological calculations, but the numerical values don't matter for our purposes. We want to make an N-body representation of an NFW halo. Let's assume that the halo is perfectly spherical to keep things simple. Also, we can note that the mass of the NFW profile is logarithmically divergent as we go to larger radii. This is kind of annoying for an N-body simulation because we can't model an infinite volume. So, let's introduce a truncation such that

$$\rho(r) = \frac{\rho_0}{(r/r_\mathrm{s})\left[1+(r/r_\mathrm{s})\right]^2} \times \begin{cases} 1 & \text{if } r < r_\mathrm{out} \\ \exp\left(-\frac{r}{r_\mathrm{out}}\right) & \text{if } r > r_\mathrm{out}, \end{cases} \tag{IV.2.34}$$

where $r_\mathrm{out}$ is some outer radius. Like any good computational physicist we'll be sure to check that our adoption of a truncation radius doesn't affect out results (for example, by repeating our calculations with a larger value of $r_\mathrm{out}$).

First, we want to select a position for each particle in our N-body representation. Since the halo is spherical it makes sense to use spherical coordinates. For $\phi$ things are easy, we just pick a number at random from a distribution that's uniform between 0 and $2\pi$. For $\theta$ it's also easy—we know that the surface area on a sphere between $\theta$ and $\theta + \mathrm{d}\theta$ is proportional to $\sin\theta \mathrm{d}\theta \equiv \mathrm{d}\cos\theta$. So, if we select a number at random from a uniform distribution between $-1$ and 1 and call this $\cos\theta$ then take the inverse cosine to get $\theta$ we should have the correct distribution.

For the radial coordinate, $r$, we have to think a little more. If each of our particles has the same mass, then the chance of a particle drawn at random from the halo mass dsitribution lying between $r$ and $r + \mathrm{d}r$ should be simply proportional to the fraction of mass in that region. That is

$$\mathrm{d}P(r) = \frac{4\pi r^2 \rho(r)\mathrm{d}r}{\int_0^\infty 4\pi r^2 \rho(r)\mathrm{d}r}. \tag{IV.2.35}$$

if we integrate this, we get the cumulative probability that any randomly selected particle will be found within radius $r$:

$$P(r) = \frac{\int_0^r r'^2 \rho(r')\mathrm{d}r'}{\int_0^\infty 4\pi r'^2 \rho(r')\mathrm{d}r'}. \tag{IV.2.36}$$

By definition, $P(r)$ runs from 0 to 1. Therefore, for each particle we can simply select a random value from a uniform distribution between 0 and 1 and call this $P(r)$. We then use the above equation to infer the $r$ that corresponds to this $P(r)$. In this way we're guaranteed to build up a halo with the correct density profile. Note that the above integrals are not analytically tractable when the exponential term is included, so they usally have to be solved numerically.

After applying this procedure, we have 3-D positions for all of the particles in the N-body representation of the halo (which we can easily convert to Cartesian coordinates for input into our N-body solver). We also need velocities. For these, we make use of the fact that our halo is assumed to be in equilibrium. In collisionless gravitational dynamics, we know that any equilibrium

system must obey Jeans equation:

$$\frac{\mathrm{d}(\rho\sigma_r^2)}{\mathrm{d}r} + 2\frac{\beta(r)}{r}\rho(r)\sigma_r^2(r) = -\frac{GM(r)}{r^2}\rho(r), \tag{IV.2.37}$$

where $\sigma_r(r)$ is the radial velocity dispersion at radius $r$, $M(r)$ is the mass enclosed with radius $r$ and $\beta(r)$ is the anisotropy parameter defined as

$$\beta(r) = 1 - \frac{\sigma_\theta^2(r) + \sigma_\phi^2(r)}{2\sigma_r^2(r)}, \tag{IV.2.38}$$

with $\sigma_\phi(r)$ and $\sigma_\theta(r)$ being the velocity dispersions in the other two directions. Since we know $\rho(r)$ (and, therefore, $M(r)$) we can solve this equation for $\sigma_r(r)$ if we choose some form for $\beta(r)$. There are many choices here, but for simplicity let's assume an isotropic velocity distribution which implies $\beta(r) = 0$. The Jeans equation is then easily solved for $\sigma_r(r) = \sigma_\phi(r) = \sigma_\theta(r)$. If we further assume that the velocity distribution in each coordinate is always a Gaussian then we have a fully specified velocity distribution function at all radii in the halo. Given the radius of a particle, we can then compute the corresponding $\sigma_r$ and draw a random number from a normal distribution with standard deviation $\sigma_r$ and repeat for the other two coordinates. This will give us a set of particle velocities consistent with Jeans equation and so consistent with an equilibrium halo.

Before moving on, we should think a little more about the final assumption made above, namely that the velocity distribution is Gaussian. Jeans equation does not tell us that the distribution is Gaussian. It merely tells us what the dispersion of the velocity distribution should be. We could construct an infinite number of non-Gaussian distributions which all have the same $\sigma_r$. So, we don't know for sure that we have the correct velocity distribution. Jeans equation can be extended to higher order moments of the velocity distribution, so we could (in principle) compute all of these higher moments and use them to construct a more accurate velocity distribution. In practice this is sometimes done, but often it is not. The reason is that the Gaussian approximation is often not too bad, and computing higher order moments rapidly becomes numerically challenging. When solving numerical problems we should always keep in mind which of the many approximations we have made is the most severe. For example, is it worth solving the Jeans equation for higher order moments of the velocity distribution when we've already approximated the halo as being spherically symmetric? Real halos are not spherically symmetric, so maybe this assumption is the biggest factor limiting the accuracy of our results. Maybe the fact that we're modelling purely dark matter with no gas component is a bigger limitation... There's no point wasting time doing one part of the calculation to arbitrarily high precision if other aspects are solved in a much cruder way.

#### IV.2.4.2 Cosmological Initial Conditions

For cosmological calculations we're interested in setting up initial conditions that represent the distribution of matter in the early stages of the Universe. At early times, the Universe is almost perfectly uniform with just small perturbations in the density as a function of position (these are the source of the $10^{-5}$ level ripples seen in the cosmic microwave background). In our standard cosmological model there are two features that make the situation simpler still. First, if inflation is correct then the density perturbations should be Gaussian—that is the phases of individual Fourier modes of the density field are uncorrelated and so the density field is completely described (statistically) by a power spectrum (which measures the mean amplitude of Fourier modes of a

given wavelength). Second, in cold dark matter models the particles begin with zero random velocities—i.e. there is no velocity dispersion at a given point and so velocity is an entirely deterministic function of position, controlled entirely by the density field. A further useful feature is that, because the initial density perturbations are small they can be treated with a linear perturbation theory analysis. Providing we begin our simulation at an early enough time in the universe we can therefore use a linear analysis to set our initial conditions. We won't discuss the details of cosmological perturbation theory in this class (take Ay 127 for that). A good discussion of the details (including the unpleasant relativistic aspects that we're going to completely ignore!) can be found in [5], while a code for generating such initial conditions is given by [2].

Briefly, the standard method to construct initial conditions for such scenarios is as follows:

(1) Create a set of "pre-initial conditions" which is a random, but homogeneous set of particle positions in a periodic cube. We can construct this by simply selecting particle positions $(x, y, z)$ at random within a cubic region[2].

(2) Compute the power spectrum, $P(k)$, that gives the mean amplitude of each Fourier mode of the desired density field. For a Gaussian random field the distribution of amplitude, $\hat{\delta}$, for a given mode can be shown to be a Rayliegh distribution

$$P(\hat{\delta})\mathrm{d}\hat{\delta} = \frac{\hat{\delta}}{\sigma^2}\exp\left(-\frac{\hat{\delta}^2}{2\sigma^2}\right)\mathrm{d}\hat{\delta}, \tag{IV.2.39}$$

where $\sigma^2 = VP(k)/2)$ and $V$ is the volume of the simulation cube[3].

(3) Take the Fourier transform to convert the $\hat{\delta}$ Fourier components into an actual density field. We then want to move the particles to recreate this density field in our N-body representation. For this we can use the Zel'dovich approximation which relates the Lagrangian position (position in the intial conditions) of a particle to its Eulerian position. We won't discuss the details, but note that this is a linear perturbation theory solution to the cosmological equations governing the growth of density perturbations. The Zel'dovich approximation states that $\mathbf{x} = \mathbf{q} + \Psi(\mathbf{q})$ where $\mathbf{q}$ is the initial (Lagrangian) position of a particle and $\mathbf{x}$ the Eulerian position. We can compute the gravitational potential $\Psi(\mathbf{q})$ from our density field and so can find $\mathbf{x}$. We can also take the derivative of this equation to get the velocity $\dot{\mathbf{x}}$.

There are problems with this method that make constructing accurate cosmological methods very difficuly. For example, what about the contribution from modes with wavelengths longer than the size of our simulation box? If we compute the density field on a grid, then we will only get Fourier modes whose wavelength is an integer number of grid cell lengths—do this missing modes matter? Fortunately, these issues have been considered extensively (see, for example, Sirko 7) and accurate initial condition generating codes exist.

[2]Frequently, this is not actually what is done, because it introduces fluctuations in the density field due to Poisson statistics, i.e. fluctuation in particle number in any region. If we began evolving this "homogeneous" distribution regions that randomly happen to be denser would quickly collapse under their own gravity. An alternative is to use a distribution in which the force on each particle is zero. For example, a regular lattice satisfies this condition (although it's an unstable equilibrium). Another common approach is to use "glass" initial conditions which can be made by starting with a Poisson-random distribution, then evolving it forward in time using a repulsive gravitational force. The particles will then all try to move apart until they reach positions of zero net force.

[3]Note that since the density field must be real, the Fourier components must satisfy the Hermitian constraints: $\hat{\delta}(\mathbf{k}) = \hat{\delta}^*(-\mathbf{k})$.

### IV.2.5 Parallelization

If you can do something with one computer, you can do it bigger and better with many computers. At least, you can in principle...

N-body codes are computationally demanding and so it's no surprise that a lot of work has been put into designing them to run efficiently on parallel computers. The idea is to divide the calculations that must be performed between a large number of processors and have the each solve part of the problem. To do this well there are a few considerations:

- How do we best divide the work between the processors? This may be limited by the amount of memory available to each processor (which will limit, for example, how many particles can be stored on each processor). Most importantly though, we want each processor to have about the same amount of work to do—if we have a situation where one processor is still crunching through its tasks while all others have already finished then we're not using the processors most efficiently. This is referred to as *load balancing*.
- How can we minimize the amount of communication between processors? Processors will need to communicate their results to each other (since, for example, particles on one processor will need to know the forces they feel from particles on some other processor). Communication is a relatively slow task so we want to minimize it as much as possible.

[4] gives a description of an approach to parallelizing a tree code. We'll follow his discussion. The basic approach is one of *domain decomposition* in which the simulation volume is broken up into sub-volumes and each sub-volume is assigned to a separate processor. The task is to find a sufficiently optimal domain decomposition to meet the criteria described above.

Let us assign to each particle a cost, $C_i$, which is a measure of the computational work required for this particle. In practice, this could be the number of gravitational force evaluations to evolve it over some fixed time perdiod for example. The cost is something we can compute relatively easily for each particle as the simulation proceeds. At the start of the calculation we don't know $C_i$, so we'll set $C_i = 1$ for all particles initially. We want to choose domains such that $\sum C_i$ (where the sum is taken over all particles in the domain) is about the same for all domains. To do this, we can use a technique to the tree construction algorithm we've used before. Suppose we have $2^n$ processors, with $n$ some integer. We begin by splitting the simulation volume in two, with the division made such that $\sum C_i$ is the same (or almost the same) on each side of the division. We then proceeed to split each of these two domains into two sub-domains again balancing $\sum C_i$ in each subdomain. Repeating this process $n$ times we get $2^n$ domains, corresponding to rectangular regions of the simulation cube, each with the same total computational cost.

As the simulation proceeds, the cost for each particle will change. For example, if a particle moves into a denser region it will require more computations to evolve its position and so its cost will increase. Therefore, our initially load balanced domain decomposition will not remain load balanced. Therefore, after each timestep (or, perhaps, once the load balancing becomes sufficiently bad) we can attempt to repartition into new domains to rebalance the workload. This means moving some particles from one processor to another, and therefore requires communication. We want to minimize this communication by moving as few particles as possible. Deciding which particles to move and when depends on the details of the simulation (e.g. how clustered the particles become), the algorithm used (which affects the distribution of costs) and the hardware used (which determines the relative costs of communication vs. work load imbalance). So, this

usually involves some tuning of parameters to find the optimal way in which to move particles from one processor to another.

An additional problem arises with the need to communicate gravitational forces between domains. Once we've performed the domain decomposition it's easy for each processor to build an oct-tree for its local set of particles. We could then have each processor share its local oct-tree with every other processor. Then, each processor would have the full tree and so could proceed to compute accelerations and update positions for all of its particles. In practice this is a bad idea for two reasons. First, the memory requirements for the full tree may be more than a single processor can handle. Second, this involves a lot of communication which slows down the calculation. However, we can make use of the opening angle criterion to limit the communication and memory requirements. Consider two widely spaced domains, which we'll label 1 and 2. Since all of the particles in domain 1 are far from all of the particles in domain 2 we'll only need to use the coarsest levels of the domain 2 tree to evaluate forces for particles in domain 1 (and vice versa). Therefore, we can apply the opening angle criterion to domain 2 in a conservative way to find the finest level of the domain 2 tree that will ever be needed by any particle in domain 1. We then communicate only those levels of domain 2 to the processor working on domain 1, and none of the finer levels of the tree. This allows the *essential tree* for domain 1 to be built—it contains just enough information to compute the forces on domain 1 particles at the required level of accuracy. This minimizes the communication and memory requirements.

Modern N-body codes use more complicated algorithms, but the basic ideas are the same.

## References

[1] Josh Barnes and Piet Hut. A hierarchical O(N log n) force-calculation algorithm. *Nature*, 324: 446, December 1986. URL http://adsabs.harvard.edu/abs/1986Natur.324..446B.

[2] E. Bertschinger. ASCL: COSMICS: cosmological initial conditions and microwave anisotropy codes. http://ascl.net/cosmics.html. URL http://ascl.net/cosmics.html.

[3] Walter Dehnen. Towards optimal softening in three-dimensional n-body codes - i. minimizing the force error. *Monthly Notices of the Royal Astronomical Society*, 324:273, June 2001. URL http://adsabs.harvard.edu/abs/2001MNRAS.324..273D.

[4] John Dubinski. A parallel tree code. *New Astronomy*, 1:133–147, October 1996. URL http://adsabs.harvard.edu/abs/1996NewA....1..133D.

[5] Chung-Pei Ma and Edmund Bertschinger. Cosmological perturbation theory in the synchronous and conformal newtonian gauges. *The Astrophysical Journal*, 455:7, December 1995. URL http://adsabs.harvard.edu/abs/1995ApJ...455....7M.

[6] Julio F. Navarro, Carlos S. Frenk, and Simon D. M. White. A universal density profile from hierarchical clustering. *The Astrophysical Journal*, 490:493, December 1997. URL http://adsabs.harvard.edu/abs/1997ApJ...490..493N.

[7] Edwin Sirko. Initial conditions to cosmological N-Body simulations, or, how to run an ensemble of simulations. *The Astrophysical Journal*, 634:728–743, November 2005. URL http://adsabs.harvard.edu/abs/2005ApJ...634..728S.

[8] V. Springel, N. Yoshida, and S. D. M. White. GADGET: a code for collisionless and gas-dynamical cosmological simulations. *New Astronomy*, 6:79–117, April 2001. URL http://adsabs.harvard.edu/abs/2001NewA....6...79S.

[9] Volker Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, December 2005. URL http://adsabs.harvard.edu/abs/2005MNRAS.364.1105S.

[10] Marcel Zemp, Joachim Stadel, Ben Moore, and C. Marcella Carollo. An optimum time-stepping scheme for n-body simulations. *Monthly Notices of the Royal Astronomical Society*, 376:273–286, March 2007. URL http://adsabs.harvard.edu/abs/2007MNRAS.376..273Z.

# IV.3 Hydrodynamics I – The Basics

Strictly speaking, hydrodynamics is a gross simplification: The dynamics of a distribution of particles (e.g., atoms, atomic nuclei, elementary particles, photons) $f(\mathbf{x}, \mathbf{p}, t)d\mathbf{x}d\mathbf{p}$ in 6D phase space and time is described by the Boltzmann equation

$$\frac{\partial f}{\partial t} + \frac{\mathbf{p}}{m}\nabla_{\mathbf{x}} f + \mathbf{F}\nabla_{\mathbf{p}} f = \left(\frac{\partial f}{\partial t}\right)_{\text{coll}} . \tag{IV.3.1}$$

Here, **F** is an external force, $m$ is the mass of a particle, and the **coll**ision term on the right-hand side describes the interaction of particles with each other and with the environment: emission, absorption, scattering, and viscous effects. Details on the Boltzmann equation for classical and quantum gases (fluids, particles etc.) can be found in statistical mechanics books, e.g. in Huang's book [1].

## IV.3.1 The Approximation of Ideal Hydrodynamics

Two main assumptions are made in ideal hydrodynamcis:

- **Hydrodynamic Approximation** (local equilibrium). The mean free path $\lambda$ between particle collisions is small compared to the size of a computational region, $\lambda \ll dx$, and the time $\tau$ between collision is shorter than the dynamical timescale of the system, $\tau \ll tau_{\text{dyn}}$. This allows us to neglect individual particle collisions and treat all particles as a continuous fluid.
- **Inviscid Approximation**. The particles are interacting by hard-body collisions, but in no other way locally, i.e. there is, e.g., no viscosity. When this approximation is not appropriate, one needs to solve the equations of non-ideal hydrodynamics, the Navier-Stokes equations, which we shall not address here.

The ideal hydrodynamcis approximation works well for most astrophysical problems involving gas/fluid flow. Situations involving viscosity (e.g. in accretion flows) can usually be treated by adding approximate "viscous" terms to the inviscid equations and it is rarely necessary to solve the full Navier-Stokes equations.

## IV.3.2 The Equations of Ideal Hydrodynamics

The equations of ideal hydrodynamics are derived by taking moments of the Boltzmann euqation and represent *conservation laws* for mass, energy, momentum. They are called *Euler Equations*.

(1) **Continuity Equation** (mass conservation)

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0 . \tag{IV.3.2}$$

(2) **Momentum Equation** (momentum conservation)

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla(\rho \mathbf{v}\mathbf{v} + P) = \rho \mathbf{g} , \tag{IV.3.3}$$

where **g** is an external acceleration, e.g. gravity.

(3) **Total Energy Equation** (energy conservation)

$$\frac{\partial}{\partial t}\mathcal{E} + \nabla((\mathcal{E} + P)\mathbf{v}) = \rho\mathbf{v}\mathbf{g} \ . \tag{IV.3.4}$$

Here $\mathcal{E}$ is the total specific energy per unit volume, defined by

$$\mathcal{E} = \rho\epsilon + \frac{1}{2}\rho v^2 \ , \tag{IV.3.5}$$

where $\epsilon$ is the specific internal energy per unit mass and the $\frac{1}{2}\rho v^2$ term is the specific kinetic energy per volume.

The three equations specified in the above (2 scalar equations, 1 vector equation) are incomplete: An equation of state (EOS), connecting pressure to density, internal energy, and composition, is needed to close the system. In general, we may assume $P = P(\rho, \epsilon, \{X_i\})$, where the $X_i$ are the mass fractions of the particle species present in the fluid.

### IV.3.3 Euler and Lagrange Frames

The Euler equations given in the previous section IV.3.2 describe the evolution of a fluid at any fixed location $\mathbf{x}$ in space. This approach is called *Eulerian* and the Eulerian time derivative $\partial/\partial t$ refers to the changes occurring as a result of the flow of the fluid past the a fixed location $\mathbf{x}$.

An alternative view is called *Lagrangian* and assumes spatial coordinates that ar comoving with the fluid (they are also called *comoving coordinates*). The Lagrangian position vector $\mathbf{r}$ is defined by the instantaneous position of the fluid element. The Lagrangian time derivative $D/Dt$ refers to the changes within a fluid element as it changes its state and location.

At the location occupied by the fluid element at an instant $t$, the Lagrangian velocity equals the Eulerian velocity with which the element sweeps past $\mathbf{x}$:

$$\frac{D}{Dt}\mathbf{r} = \mathbf{v}(\mathbf{x}) \ . \tag{IV.3.6}$$

Using the definition of the Lagrangian derivative we may write for a quantity $Q$,

$$\frac{DQ}{Dt} = \lim_{\Delta t\to 0}\frac{Q(\mathbf{x} + \mathbf{v}\Delta t, t + \Delta t) - Q(\mathbf{x}, t)}{\Delta t} \ . \tag{IV.3.7}$$

Expanding this to first order yields

$$Q(\mathbf{x} + \mathbf{v}\Delta t, t + \Delta t) = Q(\mathbf{x}, t) + \frac{\partial Q}{\partial t}\Delta t + \sum_j v_j\frac{\partial Q}{\partial x_j}\Delta t \ . \tag{IV.3.8}$$

So, to first order,

$$\frac{DQ}{Dt} = \frac{\partial Q}{\partial t} + \sum_j v_j\frac{\partial Q}{\partial x_j} \tag{IV.3.9}$$

is the transformation equation between Eulerian and Lagrangian derivatives.

Consider the Eulerian continuity equation:

$$\frac{\partial\rho}{\partial t} + \sum_j \frac{\partial}{\partial x_j}(\rho v_j) = 0 \ . \tag{IV.3.10}$$

Now,

$$\frac{\partial}{\partial x_j}(\rho v_j) = v_j \frac{\partial \rho}{\partial x_j} + \rho \frac{\partial v_j}{\partial x_j} \ , \tag{IV.3.11}$$

and the Lagrangian variant is simply given by

$$\begin{aligned} \frac{D\rho}{Dt} &= \frac{\partial \rho}{\partial t} + \sum_j v_j \frac{\partial \rho}{\partial x_j} = -\rho \sum_j \frac{\partial v_j}{\partial x_j} \ , \\ \frac{D\rho}{Dt} &+ \rho \sum_j \frac{\partial v_j}{\partial x_j} = 0 \ . \end{aligned} \tag{IV.3.12}$$

The Lagrangian expressions for the momentum and energy equations can be derived in similar fashion. They are

$$\frac{Dv_i}{Dt} + \frac{1}{\rho}\frac{\partial P}{\partial x_i} = g_i \ , \tag{IV.3.13}$$

for momentum, and

$$\frac{D\mathcal{E}}{Dt} + \sum_j v_j \frac{\partial P}{\partial x_j} + (\mathcal{E} + P) \sum_j \frac{\partial v_j}{\partial x_j} = \rho \sum_j v_j g_j \ , \tag{IV.3.14}$$

for energy.

### IV.3.4 Special Properties of the Euler Equations

In the following, we shall consider the Eulerian picture and only one spatial dimension to keep the notation simple. All statements translate straightforwardly to the multi-dimensional case.

#### IV.3.4.1 System of Conservation Laws

We can write the Euler equations as a *flux-conservative system* of PDEs

$$\frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F} = \mathbf{S} \ , \tag{IV.3.15}$$

where the state vector $\mathbf{U}$ is

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho v \\ \rho\epsilon + \frac{1}{2}\rho v^2 \end{pmatrix} \ , \tag{IV.3.16}$$

and the flux vector $\mathbf{F}$ is given by

$$\mathbf{F} = \begin{pmatrix} \rho v \\ \rho v v + P \\ (\rho\epsilon + \frac{1}{2}\rho v^2 + P)v \end{pmatrix} \ . \tag{IV.3.17}$$

The source vector $\mathbf{S}$ given by

$$\mathbf{S} = \begin{pmatrix} 0 \\ \rho g \\ \rho v g \end{pmatrix} \tag{IV.3.18}$$

represents an external force. If it is zero, the system of equations (IV.3.15) is said to be conservative. Otherwise it is flux conservative.

#### IV.3.4.2 Integral Form of the Equations

Assume a domain $D$ and $d$ dimensions. In the absence of sources, we can write

$$\underbrace{\frac{d}{dt}\int_D \mathbf{U} d\mathbf{x}}_{\substack{\text{temporal change of the state vector in the domain}}} + \underbrace{\sum_{j=1}^{d}\int_{\partial D}\mathbf{F}_j \mathbf{n}_j dS}_{\substack{\text{gains \& losses throught the boundary of the domain}}} = 0 \tag{IV.3.19}$$

#### IV.3.4.3 Hyperbolicity

The Euler equations are hyperbolic. Consider

$$\frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F} = 0 \ . \tag{IV.3.20}$$

and compute the spatial derivative. With that, we can rewrite the conservation law in *quasi-linear form*,

$$\frac{\partial}{\partial t}\mathbf{U} + \bar{A}\frac{\partial}{\partial x}\mathbf{U} = 0 \ , \tag{IV.3.21}$$

where

$$\bar{A} = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} \tag{IV.3.22}$$

is the *Jacobian* matrix of the system. The qualification quasi-linear derives from the fact that $\bar{A}$ will generally depend on $\mathbf{U}$, $\mathbf{x}$, and $t$.

Following what we discussed in §III.12.1.1, our system of equations is hyperbolic if $\bar{A}$ is diagonizable and has only real eigenvalues. This is indeed the case for the Euler equations. There exists a matrix $\overline{Q}$ so that

$$\overline{\Lambda} = \overline{Q}^{-1}\bar{A}\overline{Q} \tag{IV.3.23}$$

and $\overline{\Lambda}$ is diagonal with real numbers, the eigenvalues of $\bar{A}$, on its diagonal. The eigenvalues of $\bar{A}$ are

$$\lambda_i = \{v, v + c_s, v - c_s\} \ , \tag{IV.3.24}$$

where $c_s$ is the adiabatic sound speed,

$$c_s = \sqrt{\left.\frac{dP}{d\rho}\right|_s} \ , \tag{IV.3.25}$$

where $|_s$ denotes "at constant (specific) entropy".

#### IV.3.4.4 Characteristic Form

In the special case where $A$, and therefore $Q$, is constant we write $\mathbf{U} = \overline{Q}\mathbf{U}'$, and find after substituting into (IV.3.21) and multiplying with $Q^{-1}$ from the left

$$\frac{\partial}{\partial t}\mathbf{U}' + \overline{\Lambda}\frac{\partial}{\partial x}\mathbf{U}' = 0 \ . \tag{IV.3.26}$$

If instead $A$ depends on $U$ we *define* $U'$ by requiring

$$\frac{\partial \vec{U}}{\partial U'} = Q \tag{IV.3.27}$$

which leads to the same equation for the $U'$.

Since $\overline{\Lambda}$ is diagonal, the equations for $\mathbf{U}'$ are decoupled and are called *characteristic equations*. For each $i$, we have

$$\frac{\partial}{\partial t} U_i' + \lambda_i \frac{\partial}{\partial x} U_i' = 0 \tag{IV.3.28}$$

where the $\lambda_i$ are the eigenvalues of $\bar{A}$ and are called the *characteristic speeds*.

#### IV.3.4.5 Weak Solutions

Hyperbolic conservation laws like the Euler equations admit so-called *weak solutions* that satisfy the integral form of the conservation law, but may have a finite number of discontinuities with the variables of the system obeying certain *jump conditions* at these discontinuities.

In hydrodynamics, such discontinuities are called *shocks* or *contact discontinuities*.

### IV.3.5 Shocks

In any situation, we can decompose the velocity $\mathbf{v}$ into componets that are perpendicular and tangential to a surface $\mathbf{S}$, $\mathbf{v} = \mathbf{v}_\perp + \mathbf{v}_\parallel$.

We define a *shock* (a shock wave, a shock front) as a surface across which $P$, $\rho$, $\mathbf{v}_\perp$, $T$, and the entropy $s$ change abruptly, whereas $\mathbf{v}_\parallel$ remains continuous. A typical example of a shock is the explosion of a supernova that expands into the interstellar space.

We define a *contact discontinuity* (a contact) as a surface across which (one or multiple of) $\rho$, $T$, $s$, or $\mathbf{v}_\parallel$ change, but $P$ and $\mathbf{v}_\perp$ remain continuous. An example for a contact discontinuity is the sharp interface between a cold dense region in the interstellar medium and an adjacent warm, lower-density region at the same gas pressure. The key to stability is the constant pressure across the boundary (and the absence of significant external gravitational fields that could lead to bouancy).

#### IV.3.5.1 How Shocks Develop

For simplicity, we will assume an isentropic fluid (a fluid with constant entropy). The will be clearly wrong once we have a shock, but it is an acceptable approximation for the phase before the shock appears. Afterwards, we may assume the flow left and right of the shock to be isentropic, while the entropy increases across the shock. The following discussion is based on Mihalas & Mihalas [2].

So, our equation of state (EOS) will be a polytropic one:

$$P = K\rho^\gamma \ , \tag{IV.3.29}$$

where $K$ is the polytropic constant and $\gamma$ is the adiabatic index. The adiabatic speed of sound is given by

$$c_s = \sqrt{\left.\frac{\partial P}{\partial \rho}\right|_s} = \sqrt{\gamma \frac{P}{\rho}} \,, \tag{IV.3.30}$$

where $|_s$ denotes "at constant (specific) entropy".

Let us set up a one-dimensional pulse (a smalls spatial distribution of fluid density $\rho$) launched into an infinite homogeneous medium. Since our flow is isentropic, we have to consider only the continuity and the momentum equations. The energy is always given by the first law of thermodynamics:

$$d\epsilon = -P d\left(\frac{1}{\rho}\right) \,, \quad \text{and} \quad P = K\rho^\gamma \,, \tag{IV.3.31}$$

$$\epsilon = \frac{P}{(\gamma-1)\rho} \,, \tag{IV.3.32}$$

where we have set the integration constant to zero.

Assume now that that $\rho$ and the fluid velocity $v$ are *single-valued* functions[4] of the time $t$. Then we can, using $\rho(t)$ and $v(t)$ (where we have suppressed the dependence on the spatial coordinate), find $\rho(v)$ and $v(\rho)$. We may also re-write the continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(v\rho) = 0 \tag{IV.3.33}$$

to

$$\begin{aligned} \left(\frac{d\rho}{dv}\right)\left(\frac{\partial v}{\partial t}\right) + \left[v\left(\frac{d\rho}{dv}\right) + \rho\right]\frac{\partial v}{\partial x} = 0 \,, \\ \frac{\partial v}{\partial t} + \left[v + \rho\left(\frac{dv}{d\rho}\right)\right]\frac{\partial v}{\partial x} = 0 \,. \end{aligned} \tag{IV.3.34}$$

The momentum equation may similarly be rewritten to

$$\frac{\partial v}{\partial t} + \left[v + \frac{1}{\rho}\underbrace{\left(\frac{\partial P}{\partial \rho}\right)\Big|_s}_{=c_s^2}\frac{d\rho}{dv}\right]\frac{\partial v}{\partial x} = 0 \,. \tag{IV.3.35}$$

Comparing (IV.3.34) with (IV.3.35), we find

$$\frac{dv}{d\rho} = \pm\frac{\sqrt{\left.\frac{\partial P}{\partial \rho}\right|_s}}{\rho} = \pm\frac{c_s}{\rho} \,. \tag{IV.3.36}$$

Hence, the relation between $v$ and $\rho$ (or $P$) in the wave is

$$v = \pm\int_{\rho_0}^{\rho}\frac{c_s}{\rho}d\rho = \pm\int_{P_0}^{P}\frac{dP}{\rho c_s} \,, \tag{IV.3.37}$$

[4] A single-valued function is a unique map of one argument value to one function value or multiple argument values to one function value.

where $\rho_0$ and $P_0$ are the ambient values in the unperturbed medium.

Now, using (IV.3.36) in (IV.3.34) or (IV.3.35), we obtain

$$\frac{\partial v}{\partial t} + (v \pm c_s)\frac{\partial v}{\partial x} = 0 \ . \tag{IV.3.38}$$

Similarly to before, we can again rewrite the continuity equation, this time saying $v = v(\rho)$:

$$\frac{\partial \rho}{\partial t} + \left(\rho \frac{dv}{d\rho} + v\right)\frac{\partial \rho}{\partial x} = 0 \ , \tag{IV.3.39}$$

which with (IV.3.36) implies

$$\frac{\partial \rho}{\partial t} + (v \pm c_s)\frac{\partial \rho}{\partial x} = 0 \ . \tag{IV.3.40}$$

Equations (IV.3.38) and (IV.3.40) have general solutions of the kind

$$v = F_1[x - (v \pm c_s)t] \ , \tag{IV.3.41}$$
$$\rho = F_2[x - (v \pm c_s)t] \ , \tag{IV.3.42}$$

where $F_1$ and $F_2$ are functions representing simple traveling waves. Hence, a particular value of $\rho$ or $v$ will propagate through the medium with *phase space*

$$u_p(v) = v \pm c_s(v) \ , \tag{IV.3.43}$$

where $c_s(v)$ is given by Eqs. (IV.3.36) and (IV.3.37). For the $\pm$ sign in the above equations, we now choose:

$+$ for waves traveling in the $+x$ direction,
$-$ for waves traveling in the $-x$ direction.

Since we can write $\rho = \rho(v)$ and $P = P(\rho(v))$ (and so on), all physical variables in the wave propagate in the same manner as $v$.

Now, for our perfect isentropic fluid,

$$\boxed{c_s^2 \propto \frac{P}{\rho} \propto \rho^{\gamma-1}} \ , \tag{IV.3.44}$$

from which follows

$$(\gamma - 1)\frac{d\rho}{\rho} = 2\frac{dc_s}{c_s} \ . \tag{IV.3.45}$$

Plugging this into (IV.3.37) yields

$$\boxed{v = \pm\frac{2(c_s - c_{s_0})}{\gamma - 1}} \ , \tag{IV.3.46}$$

or,

$$c_s = c_{s_0} \pm \frac{1}{2}(\gamma - 1)v \ . \tag{IV.3.47}$$

This implies that the phase velocity is given by

$$u_p(v) = \frac{1}{2}(\gamma + 1)v \pm c_{s_0} \ . \tag{IV.3.48}$$

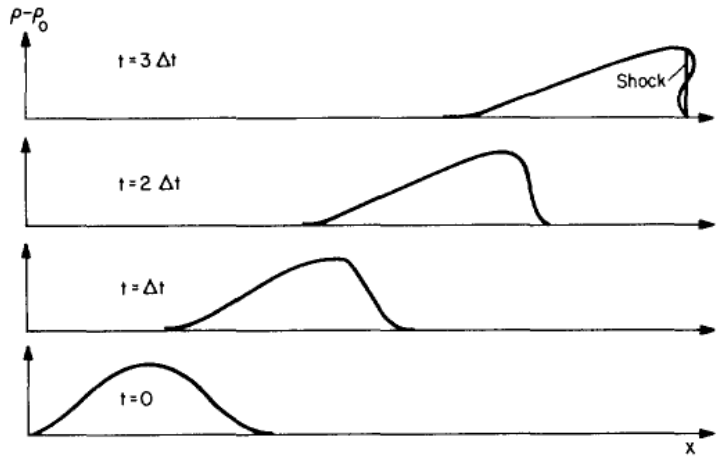



Figure IV.3.1: Non-linear steepening of a sound wave into a shock. Taken from Mihalas & Mihalas [2].

Using the polytropic gas laws (e.g., [2]) for the polytropic EOS $P = K\rho^\gamma$,

$$\begin{aligned} P &= P_0 \left(\frac{T}{T_0}\right)^{\gamma/(\gamma-1)} , \\ P &= P_0 \left(\frac{\rho}{\rho_0}\right)^{\gamma} , \\ T &= T_0 \left(\frac{\rho}{\rho_0}\right)^{\gamma-1} , \end{aligned} \tag{IV.3.49}$$

and combining them with (IV.3.46), we arrive at

$$\begin{aligned} \rho &= \rho_0 \left[1 \pm \frac{1}{2}(\gamma-1)\left(\frac{v}{c_{s_0}}\right)\right]^{2/(\gamma-1)} , \\ P &= P_0 \left[1 \pm \frac{1}{2}(\gamma-1)\left(\frac{v}{c_{s_0}}\right)\right]^{2\gamma/(\gamma-1)} , \\ T &= T_0 \left[1 \pm \frac{1}{2}(\gamma-1)\left(\frac{v}{c_{s_0}}\right)\right]^{2} . \end{aligned} \tag{IV.3.50}$$

**Interpretation**

Consider a finite-size density wave pulse with an initial half-sinosoidal shape moving to the right (Fig. IV.3.1). Eqs. (IV.3.44), (IV.3.46), and (IV.3.50) show that more dense regions of the pulse have a higher sound speed (and pressure and temperature) and move faster than less dense regions. Hence, the wave crest (highest density) travels faster than the rest of the pulse and eventually overtakes the pulse front and the wave *breaks*. At this point, $\rho(x,t)$ becomes multi-valued and the solution breaks down: We have a shock!

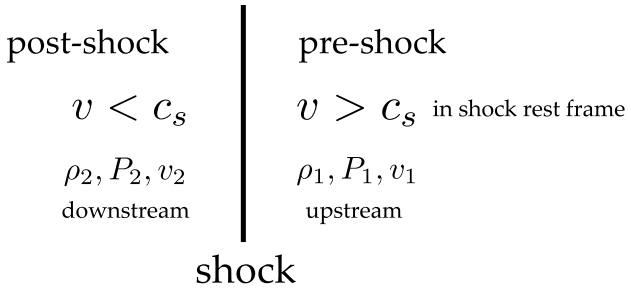



Figure IV.3.2: Schematic view of a one-dimensional shock front moving to the right in the laboratory frame. In the shock rest frame, the velocity of the gas is subsonic behind and supersonic in front of the shock.

### IV.3.6 Rankine-Hugoniot Conditions

A shock is an irreversible (non-adiabatic, entropy generating) wave that causes a transition from supersonic to subsonic flow (Fig. IV.3.2).

The shock thickness, the actual discontinuity, is of the order of the mean free path of the gas/fluid particles. This is microscopic and much smaller than the scales of gradients in the gas/fluid. For all practical purposes we may assume it to be infinitely thin.

Using the Euler equations as conservation laws, one can derive useful expressions on how key fluid quantities change across a shock. We will work in a frame in which the shock is stationary (see Fig. IV.3.2).

We must conserve mass across the shock:

$$\boxed{\text{Mass Conservation:} \quad \rho_1 v_1 = \rho_2 v_2 = \underbrace{\mathcal{F}_{\rm mass}}_{\text{Mass Flux}}} \ . \tag{IV.3.51}$$

Momentum must also be conserved. We may assume a steady solution (which is OK if we are interested in a snapshot) and set $\partial v/\partial t = 0$ and assume zero external force. Then

$$\begin{aligned} \rho v \frac{dv}{dx} + \frac{dP}{dx} &= 0 \ , \\ \frac{d}{dx}(P + \rho v^2) &= 0 \ , \end{aligned} \tag{IV.3.52}$$

where we have used $(\rho v) = const$. Integrating over the shock yields:

$$\boxed{\text{Momentum Conservation:} \quad P_1 + \rho_1 v_1^2 = P_2 + \rho_2 v_2^2 = \underbrace{\mathcal{F}_{\rm mom}}_{\text{Momentum Flux}}} \ . \tag{IV.3.53}$$

Of course, energy is also conserved and neglecting radiative/conductive losses ($\partial \mathcal{E}/\partial t = 0$), we have

$$\frac{d}{dx}\left[v\left(\frac{1}{2}\rho v^2 + \rho\epsilon + P\right)\right] = 0 \ . \tag{IV.3.54}$$

We now specialize to the case of an ideal gas/fluid with $\gamma = 5/3$ (we will go back to the general case at the end). With this

$$\epsilon = \frac{P}{(\gamma - 1)\rho} = \frac{3P}{2\rho} \ . \tag{IV.3.55}$$

Using this together with $(\rho v) = const.$, we rewrite (IV.3.54):

$$\rho v \frac{d}{dx}\left(\frac{1}{2}v^2 + \frac{5P}{2\rho}\right) = 0 \ . \tag{IV.3.56}$$

This we integrate across the shock and obtain:

$$\boxed{\text{Energy Conservation:} \quad \frac{1}{2}v_1^2 + \frac{5}{2}\frac{P_1}{\rho_1} = \frac{1}{2}v_2^2 + \frac{5}{2}\frac{P_2}{\rho_2} = \mathcal{E}} \ , \tag{IV.3.57}$$

where $\mathcal{E} = \frac{1}{2}v^2 + \frac{5}{2}\frac{P}{\rho}$ is the total specific energy.

Eqs. (IV.3.51), (IV.3.53), and IV.3.57) are called the Rankine-Hugoniot shock jump conditions.

Let us now define the Mach number

$$\mathcal{M}^2 = \frac{v^2}{c_s^2} \overset{(\gamma=5/3)}{=} \frac{3}{5}\frac{\rho v^2}{P} \ . \tag{IV.3.58}$$

Dividing (IV.3.53) by (IV.3.51) $\times v$ :

$$\frac{\mathcal{F}_{\rm mom}}{\mathcal{F}_{\rm mass} v} = \frac{P}{\rho v^2} + 1 = \frac{3}{5\mathcal{M}^2} + 1 \ , \tag{IV.3.59}$$

and

$$\mathcal{E} = \frac{1}{2}v^2 + \frac{5}{2}\frac{P}{\rho} = \frac{1}{2}v^2 + \frac{5}{2}\left(\frac{\mathcal{F}_{\rm mom} v}{\mathcal{F}_{\rm mass}} - v^2\right) \ , \tag{IV.3.60}$$

or

$$v^2 - \frac{5}{4}\frac{\mathcal{F}_{\rm mom} v}{\mathcal{F}_{\rm mass}} + \frac{1}{2}\mathcal{E} = 0 \ . \tag{IV.3.61}$$

The latter is a quadratic equation for $v$ in terms of the conserved quantities $\mathcal{F}_{\rm mass}$, $\mathcal{F}_{\rm mom}$, and $\mathcal{E}$. It has two roots, which correspond to the velocities upstream $(v_1)$ and downstream $(v_2)$ of the shock. Solving Eq. (IV.3.61), we find

$$v_1 + v_2 = \frac{5}{4}\frac{\mathcal{F}_{\rm mom}}{\mathcal{F}_{\rm mass}} \ , \tag{IV.3.62}$$

and

$$\frac{v_2}{v_1} = \frac{5}{4}\frac{\mathcal{F}_{\rm mom}}{\mathcal{F}_{\rm mass} v_1} - 1 = \frac{5}{4}\left(\frac{3}{5\mathcal{M}_1^2} + 1\right) - 1 \ , \tag{IV.3.63}$$

where we have used Eq. (IV.3.59) and where $\mathcal{M}_1$ is the upstream Mach number.

In the *strong shock limit*, $\mathcal{M}_1 \gg 1$. In this case, Eq. (IV.3.63) results in $v_2/v_1 = 1/4$ and from $\rho v = const.$, we have $\rho_2/\rho_1 = 4$.

These results can be generalized for any value of $\gamma$ (and independent of Mach number):

$$\frac{\rho_1}{\rho_2} = \frac{(\gamma+1)P_1 + (\gamma-1)P_2}{(\gamma-1)P_1 + (\gamma+1)P_2} \ . \tag{IV.3.64}$$

Using the ideal gas law, we can also write out the jump in temperature:

$$\frac{T_2}{T_1} = \frac{P_2\rho_1}{P_1\rho_2} = \frac{P_2}{P_1}\left[\frac{(\gamma+1)P_1 + (\gamma-1)P_2}{(\gamma-1)P_1 + (\gamma+1)P_2}\right] \ . \tag{IV.3.65}$$

In terms of the upstream mach number $\mathcal{M}_1$ one can derive additional useful relations:

$$\frac{\rho_2}{\rho_1} = \frac{v_1}{v_2} = \frac{(\gamma+1)\mathcal{M}_1^2}{(\gamma-1)\mathcal{M}_1^2+2}\ , \tag{IV.3.66}$$

and

$$\frac{P_2}{P_1} = \frac{2\gamma\mathcal{M}_1^2}{\gamma+1} - \frac{\gamma-1}{\gamma+1}\ . \tag{IV.3.67}$$

The downstream Mach number $\mathcal{M}_2$ in terms of $\mathcal{M}_1$ is given by

$$\mathcal{M}_2^2 = \frac{2+(\gamma-1)\mathcal{M}_1^2}{2\gamma\mathcal{M}_1^2-(\gamma-1)}\ . \tag{IV.3.68}$$

Because we have a shock, the upstream Mach number $\mathcal{M}_1$ is $> 1$. By writing out Eq. (IV.3.66) as

$$\frac{\rho_2}{\rho_1} = \frac{\gamma+1}{(\gamma-1)+2\mathcal{M}_1^{-2}}\ , \tag{IV.3.69}$$

we see that $\rho_2/\rho_1 > 1$ for $\mathcal{M}_1 > 1$ and increases with an increase in $\mathcal{M}_1$. This means that the gas behind the shock is always compressed and that a shock involving stronger compression has to move faster.

## References

[1] K. Huang. *Statistical Mechanics*. John Wiley & Sons, Hoboken, NJ, USA, 1987.

[2] D. Mihalas and B. Weibel-Mihalas. *Foundations of Radiation Hydrodynamics*. Dover Publications, Mineola, NY, USA, 1999.

## IV.4 Smoothed Particle Hydrodynamics

Original author of this section: Andrew Benson
Modified by Christian Ott in February 2014.
Note: This section will require further work in the next major revision of the notes.

So far, we have only discussed gravitational forces in the N-body technique. This is OK if we are dealing with dark matter and/or stars since these both act as collisionless particles which experience only gravitational forces. But, in general, we want to include gas as well, and so we need to include hydrodynamics and, perhaps, things like radiative energy losses etc. We will discuss, in §IV.5, the numerical solution of hydrodynamics using grid-based methods (i.e. Eulerian methods). We can certainly use those in conjunction with an N-body representation of dark matter, for example. But, since we are dealing with particles anyway, it is also worth considering if we can use a particle-based approach for hydrodynamics. One advantage of this might be that a particle based approach naturally concentrates particles in dense regions, so we automatically get better resolution in dense regions of our simulation.

The first thing to note in considering this is that hydrodynamics involves things like density and pressure gradients. These are quantities which fundamentally can't be specified for a single particle. Instead, they depend on the relation between a particle and other nearby particles. For example, the density in a region clearly depends on how many particles are present in that region. So, we are going to need to consider collections of particles to estimate these quantities. The Smoothed Particle Hydrodynamics (SPH) technique does this by smoothing over the local particle distribution in a specific way that we will examine next. In this, we will follow the notation of Springel (2010, [3]), who gives a more detailed review of the SPH technique.

### IV.4.1 Smoothing Kernels

We begin by realizing that, for any field $F(\mathbf{r})$ (which could be density, pressure etc.), we can write a smoothed version of this field as a convolution

$$F_{\rm s}(\mathbf{r}) = \int F(\mathbf{r}) W(\mathbf{r}-\mathbf{r}', h) {\rm d}^{\rm x}\mathbf{r}', \tag{IV.4.1}$$

where $W(\mathbf{r}-\mathbf{r}', h)$ is a smoothing kernel, $h$ is the characteristic length scale of the kernel (the smoothing length) and $d^x\mathbf{r}$ indicates a differential in $x$ dimensions. In the limit $h \rightarrow 0$ the kernel should approach a Dirac delta function, in which case $F_{\rm s}(\mathbf{r}) = F(\rm r)$. And important property of the kernel is its normalization to one:

$$\int W(\mathbf{r}', h) d^x\mathbf{r}' = 1 \ . \tag{IV.4.2}$$

Note that the normalization constant in front of the function form of $W$ will generally depend on the number of dimensions $x$ of the problem.

To be useful for our purposes, we will soon see that we want a kernel that is symmetric (i.e. does not have a preferred direction) and is differentialable at least twice (so we can take derivatives of the smoothed field). In the early days of SPH, people used Gaussian kernels, but most modern codes use a kernel with "finite support" (i.e. it is zero beyond some finite radius), typically a cubic spline. Writing $W(\mathbf{r}, h) = w(|\mathbf{r}|/(2h))$ (so that our kernel is automatically symmetric and scales

linearly with $h$), we can write

$$w(q) = \frac{8}{\pi} \begin{cases} 1 - 6q^2 + 6q^3 , & 0 \leq q \leq \frac{1}{2} , \\ 2(1-q)^3 , & \frac{1}{2} < q \leq 1 , \\ 0 , & q > 1 . \end{cases} \tag{IV.4.3}$$

This kernel (which belongs to a broader class of similar kernels) goes to zero beyond $r = 2h$. It is normalized for the 3D case.

In the case of a particle distribution, our field $F(\mathbf{r})$ is actually a collection of delta functions. If each particle has a mass $m_i$ and a density $\rho_i$ (we will worry about how we get this later) then they are associated with a volume $\Delta \mathbf{r}_i^x \sim m_i/\rho_i$. In this case, we can replace the convolution integral with a sum:

$$F_\mathrm{s}(\mathbf{r}) \approx \sum_j \frac{m_j}{\rho_j} F_j W(\mathbf{r} - \mathbf{r}_j, h) . \tag{IV.4.4}$$

This is then a smooth, differentiable representation of the field $F$ described by our particles. We should be clear that this is an approximation, and so has some error compared to the true field (i.e. that we would obtain with an infinite number of particles). If the particles are uniformly spaced by a distance $d$ in 1D, it can be shown that the interpolation is second-order accurate if $h = d$. For real distributions in 3D it is not so simple to prove this, but it seems reasonable that we would need $h \geq d$, which means $(4\pi/3)2^3 \approx 33$ neighbor particles should be within the non-zero part of the smoothing kernel.

So what about the density? We have already stated that we don't know the density for an individual particle, yet it appears in the above sum. Fortunately, we can use the SPH kernel to evaluate the density. Let $F_i = \rho_i$, then the smoothed density field becomes

$$\rho_\mathrm{s}(\mathbf{r}) \approx \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) , \tag{IV.4.5}$$

which we can estimate knowing only the masses of the particles. The density for each particle is then found from the smoothed density field at the position of the particle.

We are free to choose whatever $h$ we want and, in fact, to make it a function of position. This is advantageous because we can make $h$ small in regions of high density and thereby avoid smoothing away too much information in these regions. There are essentially two choices for computing a suitably position-dependent $h$. In the "scatter" approach we use a kernel $W[\mathbf{r} - \mathbf{r}_j, h(\mathbf{r})]$ (i.e. $h$ depends on the position $\mathbf{r}$) while in the "gather" approach we use $W[\mathbf{r} - \mathbf{r}_j, h(\mathbf{r}_i)]$ (i.e. $h$ is determined at the position of the particle for which we are computing a smoothed field). The gather approach has the advantage that the density of particle $i$ can be determined using only $h_i$, the smoothing length for that same particle[5]. Using the gather method we have

$$\rho_i \approx \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h_i) , \tag{IV.4.6}$$

where we have dropped the subscript "s". From this, it becomes clear why kernels with compact support are useful: once we know $h_i$ we need only find those particles within a distance $2h_i$ of particle $i$ in order to find its density. This makes the calculation $\mathcal{O}(N_\mathrm{ngb} N)$ where $N$ is total particle

[5]Technically, if we integrate over the smoothed density field when using the gather method, we do not get the correct total mass, so this method does not correctly conserve mass. This does not matter though, since we explicitly conserve mass because mass is tied to particles which are conserved by construction.

number and $N_{\rm ngb}$ is the number of neighbors within $2h_i$, which will be about the same for all particles since we choose $h_i$ based on the distance to these nearest neighbors. The $N_{\rm ngb}$ parameter is a key input to SPH simulations and it should always be checked to be sufficiently large to give converged answers, as ultimately the accuracy of SPH will be limited by the accuracy of the smoothing/interpolation that the kernel approach implies.

Note that we can apply this approach to any field. For example, the velocity field is

$$\mathbf{v}_i = \sum_j \frac{m_j}{\rho_j}\mathbf{v}_j W(\mathbf{r}_i - \mathbf{r}_j, h)\,, \tag{IV.4.7}$$

which, since this is now a smooth and continuous field, we can take the derivative of to get the local velocity divergence

$$(\nabla\cdot\mathbf{v})_i = \sum_j \frac{m_j}{\rho_j}\mathbf{v}_j \cdot \nabla_i W(\mathbf{r}_i - \mathbf{r}_j, h)\,. \tag{IV.4.8}$$

Alternatively, we can make use of the identity $\rho\nabla\cdot\mathbf{v} = \nabla(\rho\mathbf{v}) - \mathbf{v}\cdot\nabla\rho$ and estimating the smoothed versions of the two fields on the right separately. This gives

$$(\nabla\cdot\mathbf{v})_i = \frac{1}{\rho_i}\sum_j m_j(\mathbf{v}_j - \mathbf{v}_i)\cdot\nabla_i W(\mathbf{r}_i - \mathbf{r}_j, h)\,, \tag{IV.4.9}$$

which turns out to be more accurate in practice and has the useful feature that it goes precisely to zero when all velocities are equal.

#### IV.4.1.1 Variational Derivation

It is possible to derive the SPH equations from a Lagrangian. Why would we do this? It guarantees that the various conservation laws hold (since they're explicitly encoded in the Lagrangian) and also ensures that the phase space structure imposed by Hamiltonian dynamics is retained. [1] showed that the Euler equations for inviscid ideal gas flow follow from the Lagrangian

$$L = \int \rho\left(\frac{\mathbf{v}^2}{2} - u\right)\mathrm{d}V. \tag{IV.4.10}$$

This idea was first fully exploited by [4] who discretized this Lagrangian in terms of the SPH particles, yielding

$$L_{\rm SPH} = \sum_i \left(\frac{1}{2}m_i\mathbf{v}_i^2 - m_i u_i\right), \tag{IV.4.11}$$

where $\epsilon_i$ is the internal energy per unit mass of the particle. For a gas with adiabatic index $\gamma$ we can write

$$P_i = A_i\rho_i^\gamma = (\gamma-1)\rho_i\epsilon_i, \tag{IV.4.12}$$

where we will refer to $A_i$ as the "entropy". More specifically, $A_i$ is uniquely determined by the specific entropy of the particle and so is conserved in adiabatic flow. So, assuming $A_i$ to be constant we can write

$$\epsilon_i = A_i\frac{\rho^{\gamma-1}}{\gamma-1}\,. \tag{IV.4.13}$$

We then use the standard variational approach to get our equations of motion:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial\dot{\mathbf{r}}_j} - \frac{\partial L}{\partial\mathbf{r}_j} = 0\,. \tag{IV.4.14}$$

Some care is needed because the smoothing scale, $h$, varies from particle to particle (and so we can represent it too as a smooth field using our standard kernel). But, the bottom line is

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left[ f_i \frac{P_i}{\rho_i^2} \nabla_i W_{ij}(h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij}(h_j) \right], \tag{IV.4.15}$$

where

$$f_i = \left[ 1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right]^{-1}, \tag{IV.4.16}$$

and $W_{ij}(h) \equiv W(|\mathbf{r}_i - \mathbf{r}_j|, h)$. What is nice about this is that we have converted the set of partial differential equations that described inviscid flow of an ideal gas into a much simpler set of ordinary differential equations. Mass and energy conservation are handled automatically: mass because we have a fixed number of particles and energy because our flow is adiabatic so the energy is determined by the (constant) entropy.

It is also possible to derive relativistic implementations of SPH using this variational principle approach.

## IV.4.2 Other Issues

### IV.4.2.1 Artificial Viscosity

In all of the above we have explicitly assumed that the specific entropy of a particle is conserved, i.e. that the flow is adiabatic. However, even if starting from smooth initial conditions, the Euler equations can lead to shocks and contact discontinuities at which the differential form of the Euler equations breaks down and their integral form (conservation laws) must be applied instead. Analysis shows that in shocks the entropy is always increased. How can we introduce this behavior into our SPH calculation? The usual approach is to introduce an artificial viscosity term. The role of this artificial viscosity is to broaden the shocks into a resolvable layer and dissipate kinetic energy into heat, thereby increasing the entropy. If introduced as a conservative force then the conservation laws implicity in the Euler equations guarantee that the correct amount of entropy is generated, independent of the details of the viscosity used. Of course, the problem with this is that shocks are now not perfect thin—instead they will be resolved by a few smoothing lengths.

The viscous force is usually added in the form

$$\left.\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t}\right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \prod_{ij} \nabla_i \overline{W}_{ij}, \tag{IV.4.17}$$

where

$$\overline{W}_{ij} = \frac{1}{2}[W_{ij}(h_i) + W_{ij}(h_j)] \tag{IV.4.18}$$

is a symmetrized kernel. Provided that $\prod_{ij}$ is symmetric in $i$ and $j$, this viscous force is antisymmetric between pairs of particles so always conserves linear and angular momentum. It does not conserve energy (that is why we are including it) so a compensating change in the internal energy (or, equivalently, entropy) is introduced to balance this. We then have the desired conversion of kinetic to thermal energy and a corresponding entropy production.

A common form used for the viscosity factor is

$$\prod_{ij} = \begin{cases} [-\alpha c_{ij}\mu_{ij} + \beta\mu_{ij}^2]/\rho_{ij} & \text{if } \mathbf{v}_{ij}\cdot\mathbf{r}_{ij} < 0\,, \\ 0 & \text{otherwise}\,, \end{cases} \tag{IV.4.19}$$

with

$$\mu_{ij} = \frac{h_{ij}\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^2 + \varphi h_{ij}^2}\,. \tag{IV.4.20}$$

Here, $h_{ij}$ and $\rho_{ij}$ are arithmetic means of the corresponding quantities for particles $i$ and $j$, and $c_{ij}$ is the mean sound speed. The strength of the viscosity is controlled by parameters $\alpha$ and $\beta$. The goal is to make this viscosity significant in shocks but weak elsewhere (so that we do not dissipate kinetic energy in regimes that are evolving adiabatically). Typical values of $\alpha \approx 0.5$–$1.0$ and $\beta = 2\alpha$ are used. The parameter $\varphi \approx 0.01$ is introduced to avoid a singularity if two particles get very close.

#### IV.4.2.2 Energy Equation

If the flow is non-adiabatic (i.e., when a shock is present and the artificial viscosity must be invoked to account for dissipation in the shock), the internal energy $\epsilon$ is no longer simply related to pressure $P_i$ and $\rho_i$ (cf. Eq. IV.4.13). Instead, one must solve an evolution equation of the form

$$\frac{\mathrm{d}\epsilon_i}{\mathrm{d}t} = \frac{1}{2}\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \prod_{ij}\right)\mathbf{v}_{ij}\nabla_i W_{ij}(h)\,, \tag{IV.4.21}$$

where we have assumed $h_i = h_j\,\forall(i,j)$. If this is not the case, one proceeds as for the momentum equation (IV.4.15).

#### IV.4.2.3 Self-Gravity

Our SPH gas particles feel the force of gravity and, in particular, there is a self-gravity between the SPH particles. We can make use of standard N-body techniques to compute the gravitational forces, but there is also a neat way to do this using the variational approach discussed above. We simply add the gravitational self-energy of the SPH particles:

$$E_{\mathrm{pot}} = \frac{1}{2}\sum_i m_i\Phi(\mathbf{r}_i) = \frac{\mathrm{G}}{2}\sum_{ij} m_i m_j \phi(r_{ij}, h_j) \tag{IV.4.22}$$

to the SPH Lagrangian giving

$$L_{\mathrm{SPH}} = \sum_i \left(\frac{1}{2}m_i\mathbf{v}_i^2 - m_i\epsilon_i\right) - \frac{\mathrm{G}}{2}\sum_{ij} m_i m_j \phi(r_{ij}, h_j)\,. \tag{IV.4.23}$$

When we apply the variational principle, the equations of motion pick up extra terms due to gravity which look like:

$$\begin{aligned} m_i \mathbf{a}_i^{\text{grav}} &= -\frac{\partial E_{\text{pot}}}{\partial \mathbf{r}_i} \ , \\ &= -\sum_j \mathrm{G} m_i m_j \frac{\mathbf{r}_{ij}}{r_{ij}} \frac{[\phi'(r_{ij}, h_i) + \phi'(r_{ij}, h_j)]}{2} \\ &\quad -\frac{1}{2} \sum_{jk} \mathrm{G} m_j m_k \frac{\partial \phi(r_{jk}, h_j)}{\partial h} \frac{\partial h_j}{\partial \mathbf{r}_i} \ . \end{aligned} \tag{IV.4.24}$$

The first term is what you would guess: it is a symmetrized gravitational interaction. The second term shows up if the smoothing length, $h$, is allowed to be different for each particle (and so varies with position), and is required to make the interaction conservative in such cases [2]. Note that the above is very similar to the N-body case discussed in §IV.2. We have just replaced the gravitational softening length $\epsilon$ with the SPH smoothing length $h$ (see, e.g., [3]).

#### IV.4.2.4 Update of Positions

Since we are dealing with particles of $m_i$ that move with some velocity $\mathbf{v}_i$ (whose update is determined by the momentum equation IV.4.15), we also need to solve a (trivial) time-evolution equation for the positions $\mathbf{r}_i$ of the particles:

$$\frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i \ . \tag{IV.4.25}$$

#### IV.4.2.5 Time Steps

One final issue: What time steps should we use when evolving the SPH particles? The usual approach is to impose a so called Courant time step criterion, which requires:

$$\Delta t_i = C_{\text{CFL}} \frac{h_i}{c_i} \ , \tag{IV.4.26}$$

where $c_i$ is the sound speed and $C_{\text{CFL}} \sim 0.1$–$0.3$. This limits each particle to moving significantly less than one smoothing length per timestep providing the particle is moving subsonically. This does not work so well in cases such as a blast wave propagating into cold gas (for which $c_i$ will be small, allowing large timesteps which can result in cold gas particles moving through the blast wave). Improved algorithms attempt to catch such cases. This timestep is usually augmented by checking the gravitational timestep (and taking the minimum of the two) using the usual N-body techniques. In fact, in an N-body tree code the tree structure turns out to be extremely useful for one of the key SPH operations, namely finding the nearest neighbors. The cost of the operation can be reduced from $\mathcal{O}(N^2)$ for a simple search of all particles to $\mathcal{O}(N_{\text{ngb}} N \ln N)$ when using the tree.

#### IV.4.2.6 Time Evolution: Leapfrog Method

A common method for time evolution of the SPH equations is the leapfrog method (cf. §III.12.3.4), which gives us second-order accuracy in time. In the SPH case, we time-center the velocity (i.e.,

evaluate it at half-timesteps $n + 1/2$) and keep positions and the internal energy at full timesteps $n$. At the beginning of the integration, we can just set $\mathbf{v}_i^{-1/2} = \mathbf{v}_i^0$, where the superscripts denote timesteps and the subscript particle number.

We write

$$\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^{n-1/2} + \Delta t \mathbf{a}_i^n \ , \tag{IV.4.27}$$

where we evaluate the accelerations at timestep $n$ (for which we will need an approximation for $v_i^n$; see below). We then update the internal energy from timestep $n$ to $n + 1$:

$$\epsilon_i^{n+1} = \epsilon_i^n + \Delta t \frac{\mathrm{d}\epsilon_i}{\mathrm{d}t}\bigg|^{n+1/2} \ , \tag{IV.4.28}$$

where the RHS $\mathrm{d}\epsilon_i/\mathrm{d}t$ depends on the velocities at $n + 1/2$.

Next, we update the positions:

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \Delta t \mathbf{v}_i^{n+1/2} \ . \tag{IV.4.29}$$

Finally, we compute an approximat $\mathbf{v}_i^{n+1}$ for the next evaluation of the accelerations:

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{1}{2}\Delta t \mathbf{a}_i^n \ . \tag{IV.4.30}$$

## References


[1] Carl Eckart. Variation principles of hydrodynamics. *Physics of Fluids*, 3(3):421, 1960. ISSN 00319171. doi: 10.1063/1.1706053. URL http://link.aip.org/link/PFLDAS/v3/i3/p421/s1%26Agg=doi.

[2] D. J. Price and J. J. Monaghan. An energy-conserving formalism for adaptive gravitational force softening in smoothed particle hydrodynamics and N-body codes. *Mon. Not. Roy. Astron. Soc.* , 374:1347, February 2007. URL http://adsabs.harvard.edu/abs/2007MNRAS.374.1347P.

[3] Volker Springel. Smoothed particle hydrodynamics in astrophysics. *Annual Review of Astronomy and Astrophysics*, 48:391–430, September 2010. URL http://adsabs.harvard.edu/abs/2010ARA%26A..48..391S.

[4] Volker Springel and Lars Hernquist. Cosmological smoothed particle hydrodynamics simulations: the entropy equation. *Monthly Notices of the Royal Astronomical Society*, 333:649–664, July 2002. URL http://adsabs.harvard.edu/abs/2002MNRAS.333..649S.

# IV.5 Hydrodynamics III – Grid Based Hydrodynamics

In this section, we will get our feet wet in classical Newtonian hydrodynamics by tackling the Riemann problem, whose exact solution we will compute. We will then go on to grid-based hydrodynamics in its flux-conservative (= mass, momentum, energy are conserved up to source terms, e.g. external forces) formulation and work our way through the details of a building a 1D (planar) finite-volume hydrodynamics code.

## IV.5.1 Characteristics

In §IV.3.4.3, we have already talked about the hyperbolicity of the Euler equations. Here we are coming back briefly to this, because we will need the *eigenstructure* (eigenvalues and eigenvectors) of the Euler equations.

### IV.5.1.1 Characteristics: The Linear Advection Equation and its Riemann Problem

Before going to the non-linear Euler equations, let's consider briefly the linear advection equation,

$$\frac{\partial}{\partial t}u + v\frac{\partial}{\partial x}u = 0 \ . \tag{IV.5.1}$$

This is a linear conservation law and its general solution is given by $u(x,t) = u(x - vt, t = 0)$. The *characteristics* (also called *characteristic curves*) of a PDE such as Eq. (IV.5.1) are defined as curves $x = x(t)$ in the $(t, x)$ plane along which the PDE becomes an ODE (see [9] for a full discussion; the following is an abridged version of what is in [9], §2.2).

Consider $x = x(t)$ and regard $u$ as a function of $t$, that is $u(x,t) = u(x(t), t)$. $u$ changes along $x(t)$ according to

$$\frac{du}{dt} = \frac{\partial u}{\partial t} + \frac{dx}{dt}\frac{\partial u}{\partial x} \ . \tag{IV.5.2}$$

If the characteristic curve $x = x(t)$ satisfies

$$\frac{dx}{dt} = v \ , \tag{IV.5.3}$$

then (IV.5.1) together with (IV.5.2) and (IV.5.3) gives

$$\frac{du}{dt} = \frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0 \ . \tag{IV.5.4}$$

This means that $u$ is constant along *any* characteristic curve $x = x(t)$ satisfying (IV.5.2) with *characteristic speed* $v$, the slope of the characteristic curve $x = x(t)$ in the $(t, x)$ plane. It is often more convenient to look at the $(x, t)$ plane in which the slope of the characteristic $x = x(t)$ will, of course, be $1/v$ (Fig. IV.5.1).

So the linear advection equation has a *family* of characteristic curves that is a *one-parameter family* that is completely determined by the initial position $x_0$ at $t = 0$. Then the characteristic curve passes through the point $(x_0, 0)$ and is completely determined by $x(t) = x_0 + vt$ and characteristic curves with different $x_0$ are parallel to each other (a feature of linear PDEs with constant coefficients).

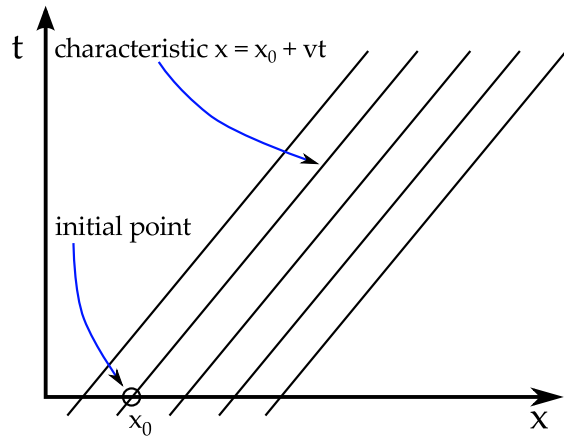



Figure IV.5.1: Characteristic curves of the linear advection equation for positive characteristic speed $v$. The initial condition at $t = 0$ fixes the position $x_0$.

Eq. (IV.5.4) shows that $u$ remains constant along characteristic curves. So if the initial condition $u_0(x) = u(x, t = 0)$ is given, we know that it will stay constant along any characteristic curve $x = x_0 + vt$ and the solution for $u(x, t)$ along this curve at all times $t$ is

$$u(x, t) = u(x_0) = u_0(x - vt) \ . \tag{IV.5.5}$$

We can formulate and solve the so-called *Riemann problem* for the linear advection equation:

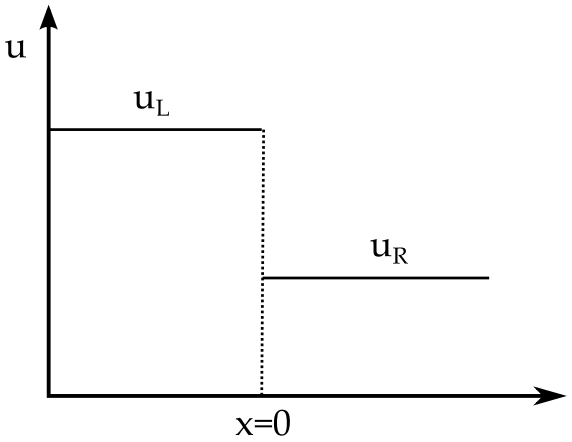



Figure IV.5.2: Illustration of the initial data for the Riemann problem of the linear advection equation.

The initial conditions for this problem are such that at $t = 0$,

$$u(x, t = 0) = \begin{cases} u_L & \text{if } x < 0 \ , \\ u_R & \text{if } x > 0 \ , \end{cases} \tag{IV.5.6}$$

Based on the just discussed solution, we expect any point on the initial profile to propagate a distance $d = vt$ in time $t$. In particular, we expect the initial discontinuity at $x = 0$ a distance $d = vt$ in time $t$. The particular characteristic curve $x = vt$ will then cleanly separate all characteristic curves to the left, on which the solution takes on the value $u_L$, from all characteristic curves to the right, on which the solution takes on the value $u_R$. This is visualized by Fig. IV.5.3.

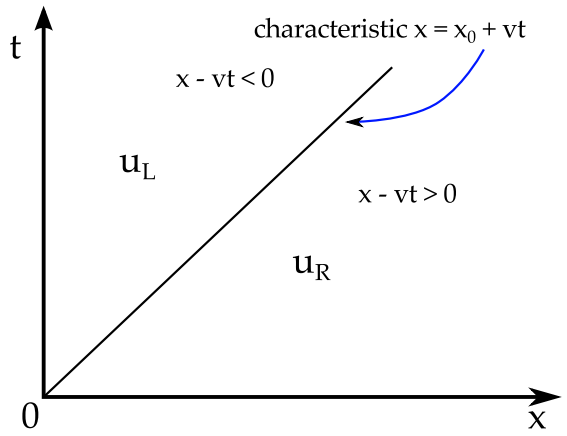



Figure IV.5.3: Characteristic view of the Riemann problem for the linear advection equation. The characteristic curve passing through the initial discontinuity at $x = 0$ separates the evolution of the left and right states.

#### IV.5.1.2 Eigenstructure of the Euler Equations

Things are a bit more complicated with the Euler Equations. This is due primarily to their non-linear nature. Here, we work out the eigenvalues and eigenvectors of the Euler equations and then apply the results to the Riemann problem in §IV.5.2.

In the following, we will always assume that our problem is 1D planar and that there are no external forces. We will work in the Eulerian frame. The Euler Equations that we have first stated in §IV.3 are then

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho v) = 0 \,, \tag{IV.5.7}$$

$$\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho v^2 + P) = 0 \,, \tag{IV.5.8}$$

$$\frac{\partial}{\partial t}\mathcal{E} + \frac{\partial}{\partial x}((\mathcal{E} + P)v) = 0 \,, \tag{IV.5.9}$$

where $\mathcal{E} = \rho\epsilon + \frac{1}{2}\rho v^2$. We can recast them into the simple form

$$\frac{\partial}{\partial t}\mathbf{U} + \frac{\partial}{\partial x}\mathbf{F} = \mathbf{S} \,, \tag{IV.5.10}$$

where the state vector $\mathbf{U}$ is

$$\mathbf{U} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} \rho \\ \rho v \\ \rho\epsilon + \frac{1}{2}\rho v^2 \end{pmatrix} \,, \tag{IV.5.11}$$

and the flux vector $\mathbf{F}$ is given by

$$\mathbf{F} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} \rho v \\ \rho v v + P \\ (\rho\epsilon + \frac{1}{2}\rho v^2 + P)v \end{pmatrix} \,. \tag{IV.5.12}$$

We can re-write this system in quasi-linear form (as we did already in §IV.3.4.3 by introducing the Jacobian

$$\bar{A} = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} = \begin{pmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \frac{\partial f_1}{\partial u_3} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \frac{\partial f_2}{\partial u_3} \\ \frac{\partial f_3}{\partial u_1} & \frac{\partial f_3}{\partial u_2} & \frac{\partial f_3}{\partial u_3} \end{pmatrix} , \tag{IV.5.13}$$

and writing

$$\frac{\partial}{\partial t}\mathbf{U} + \bar{A}\frac{\partial}{\partial x}\mathbf{U} = 0 . \tag{IV.5.14}$$

In analogy with the linear advection equation (IV.5.1) where $v$ was the characteristic speed, $\bar{A}$ contains the (three) characteristic speeds of the Euler equations as its eigenvalues. The characteristic curves along which the Euler equations reduce to ODEs (they are not constant along these curves, because they are non-linear) are given by the eigenvectors of $\bar{A}$.

The eigenvalues and eigenvectors of $\bar{A}$ are computed using standard linear algebra:

**Eigenvalues:** $\lambda_i$

$$\det(\bar{A} - \lambda_i \mathbf{I}) = 0 , \tag{IV.5.15}$$

and

**Right Eigenvectors:** $\mathbf{K}_i$

$$\bar{A}\mathbf{K}_i = \lambda_i \mathbf{K}_i . \tag{IV.5.16}$$

The Euler equations are hyperbolic, so their eigenvalues are all real and the eigenvectors are a complete set of linearly independent vectors. Computing the eigenvalues and eigenvectors of the Euler equations is a bit tedious and we will just state the end results here:

$$\lambda_1 = v - c_s , \qquad \lambda_2 = v , \qquad \lambda_3 = v + c_s , \tag{IV.5.17}$$

where $c_s$ is the adiabatic sound speed, and

$$\mathbf{K}_1 = \begin{pmatrix} 1 \\ v - c_s \\ H - v c_s \end{pmatrix} , \qquad \mathbf{K}_2 = \begin{pmatrix} 1 \\ v \\ \frac{1}{2}v^2 \end{pmatrix} , \qquad \mathbf{K}_3 = \begin{pmatrix} 1 \\ v + c_s \\ H + v c_s \end{pmatrix} , \tag{IV.5.18}$$

where $H = \frac{1}{2}v^2 + \epsilon + P/\rho$.

### IV.5.2 The Riemann Problem for the Euler Equations

The Riemann problem for the Euler equations is crucial to modern numerical methods for grid-based hydrodynamics. This is for two reasons: (1) The Riemann problem can be solved *exactly* by analytical means (involving Newton iterations to find a root) and thus can be used as a way of testing numerical implementations of the Euler equations. (2) The solution of the Riemann problem underlies the widely used *Godunov method* for numerical hydrodynamics, which we will discuss in §IV.5.3.

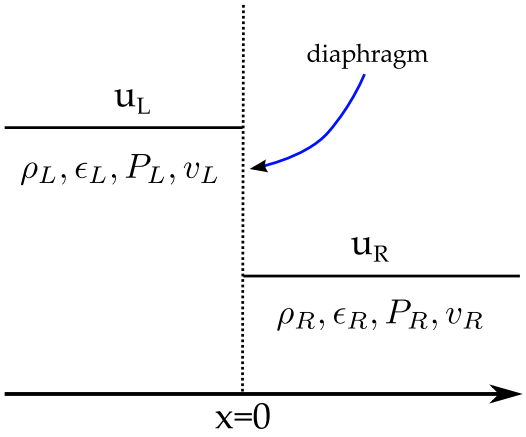



Figure IV.5.4: Initial conditions for the Riemann problem for the Euler Equations. $v_L$ and $v_R$ are initially set to zero.

Detailed discussions of the Riemann problem including its complete and *exact* solution can be found in [3, 9]. Here we will only be able to outline its most salient aspects and list results of lengthy calculations whose details can be found in [9].

The initial conditions for the Riemann problem for the Euler Equations are very similar to the one for the linear advection equation. At $t = 0$ a diaphragm separates two regions with different values of the state variables. We shall call these left (L) and right (R) states. Initially, the velocity is zero. Let's consider the case in which the L state is a hot dense gas and the right state is a cooler, lower density gas. When the diaphragm is removed, the pressure difference between L and R states will push gas from left to right and a right-moving shock develops. This is why the Riemann problem is also referred to as the *shocktube* problem or as the *Sod shocktube problem* (because it was Sod who studied a particular Riemann problem in detail). The initial conditions for the Riemann problem are shown in Fig. IV.5.4.

At $t = 0$, the diaphragm is removed and taking a snapshot at some time $t > 0$, a situation as shown in Fig. IV.5.5 presents itself. We identify five distinct regions, whose development has been experimentally verified:

(1) Unchanged left (L) fluid.

(2) *Rarefaction*, expanding left fluid, rarefaction wave propagates to the left, but fluid moves to the right.

(3) Decompressed left fluid. Connected to (4) by a contact discontinuity ($P = const.$, $v = const.$)

(4) Compressed right fluid. Connected to (5) by a shock (everything discontinuous).

(5) Unchanged right (R) fluid.

The five regions are separated by characteristic waves, i.e., the characteristics of the Euler equations discussed in §IV.5.1.2. Figure IV.5.6 shows a schematic view of the characteristics of the Euler equations in the Riemann problem. The first eigenvalue, $\lambda_1 = v - c_s$, and its eigenvector $\mathbf{K}_1$ correspond to backward traveling waves that result in the observed rarefaction. The second eigenvalue, $\lambda_2 = v$, and its eigenvector $\mathbf{K}_2$ result in a contact discontinuity ($P = const.$, $v = const.$). The third eigenvalue, $\lambda_3 = v + c_s$, and its eigenvector $\mathbf{K}_3$ result in a shock wave.

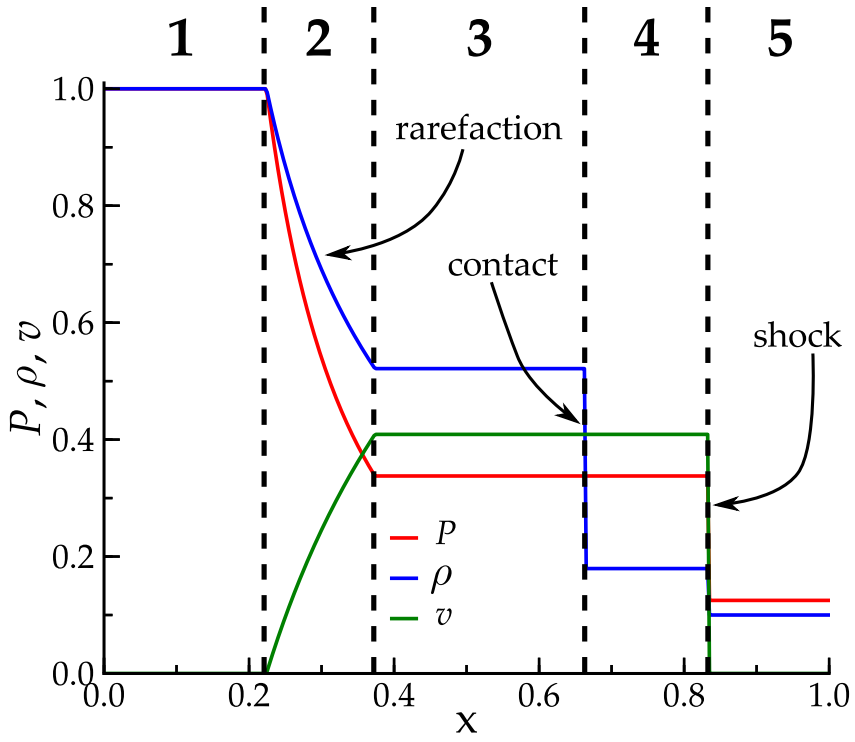



Figure IV.5.5: Exact solution of a Sod shocktube Riemann problem for $\Gamma = 5/3$ at $t = 0.4$. The initial diaphragm was placed at $x_0 = 0.5$. Regions 1 and 5 contain the untouched initial left and right states, respectively. Region 4 and 5 are connected by the shock, region 3 and 4 are connected by a contact discontinuity, region 2 has a family of rarefaction waves. Note that we have used the special-relativistic solution of Martí & Müller [6], which uses units of the speed of light to measure velocities. Note: This solution will be replaced in a future version of these notes. It is qualitatively fine, but makes quantitative comparisons difficult.

In the following, we will a bit more closely look at rarefaction, contact, and shock. The discussion that we present is heavily inspired by [2, 3, 9] and readers interested in full derivations should check in particular [9], which has the most extended and clear discussion.

#### IV.5.2.1 Rarefaction

The rarefaction wave connects the known state in region 1 with the unknown state 3 behind the contact and $\rho$, $v$, and $P$ (and thus also $\epsilon$) change across a rarefaction. Entropy is constant across a rarefaction, so we can work with the assumption of isentropic flow and use a polytropic EOS, $P = K\rho^\gamma$. In the present case, shock and contact are moving to the right, while the rarefaction wave is moving to the left. The rarefaction *head* (the part that is furthest left) is associated with the characteristic associated $\lambda^{HL} = v_1 - c_{s_1}$ and the rarefaction *tail* is associated with $\lambda^{TL} = v_3 - c_{s_3}$. These two characteristic curves enclose the *rarefaction fan*. So, since $v_1 = 0$, the rarefaction head moves to the left with velocity $-c_{s_1}$ (which is a constant) and the rarefaction tail moves to the left with velocity $v_3 - c_{s_3}$ (which is also a constant).

One can show [2, 9] that across a rarefaction, the so-called *Riemann Invariant of Flow* stays

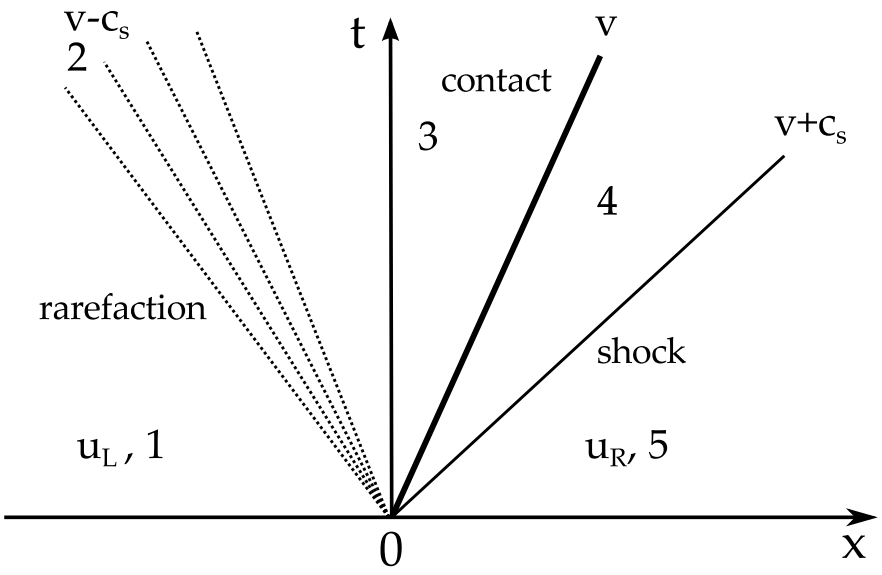



Figure IV.5.6: Characteristic wave diagram for the Riemann problem for the Euler equations. For each characteristic wave, the corresponding eigenvalue is marked. We have also marked the various regions identified in Fig. IV.5.5.

constant:

$$I_L(v, c_s) = v + \frac{2c_s}{\gamma - 1} \ . \tag{IV.5.19}$$

With this and the assumption that all state variables are single valued and can be expressed in terms of each other, one derives the state inside the rarefaction,

$$\begin{aligned} v_2(x,t) &= \frac{2}{\gamma+1}(c_{s_1} + \frac{x-x_0}{t}) \ , \\ \rho_2(x,t) &= \rho_1 \left[\frac{2}{\gamma+1} - \frac{\gamma-1}{(\gamma+1)c_{s_1}}\frac{x-x_0}{t}\right]^{\frac{2}{\gamma-1}} \ , \\ P_2(x,t) &= P_1 \left[\frac{2}{\gamma+1} - \frac{\gamma-1}{(\gamma+1)c_{s_1}}\frac{x-x_0}{t}\right]^{\frac{2\gamma}{\gamma-1}} \ . \end{aligned} \tag{IV.5.20}$$

### IV.5.2.2 Shock and Contact

We can use the Rankine-Hugoniot conditions laid out in §IV.3.6 to infer information about the shock wave connecting region 4 with the unshocked state 5. The shock travels with a laboratory-frame velocity

$$v_s = c_{s,5}\left[\frac{\gamma+1}{2\gamma}\frac{P_4}{P_5} + \frac{\gamma-1}{2\gamma}\right]^{\frac{1}{2}} \ , \tag{IV.5.21}$$

where we have used $P_4$, which we will determine in a second. First, let us write down (derivation via Rankine-Hugoniot) the velocity of the shocked fluid in region 4 behind the shock,

$$v_4 = v_s\frac{\rho_4 - \rho_5}{\rho_4} \overset{\text{(some algebra)}}{=} (P_4 - P_5)\left[\frac{1 - \frac{\gamma-1}{\gamma+1}}{\rho_5(P_4 + \frac{\gamma-1}{\gamma+1}P_5)}\right]^{\frac{1}{2}} \ . \tag{IV.5.22}$$

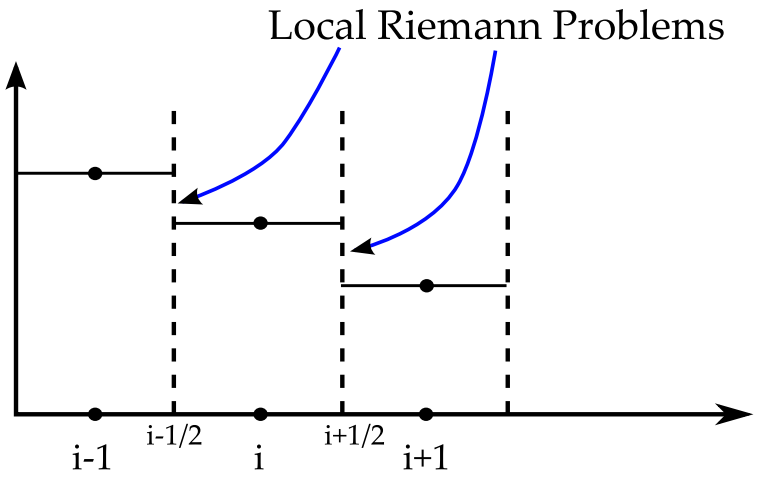



Figure IV.5.7: Schematic view of discretized data. Godunov realized that one could solve local Riemann problems to determine the flow between grid cells and in this way to compute the time update of the overall solution.

Now, accross the contact discontinuity between regions 4 and 3, both pressure and velocity remain constant. So we have $P_4 = P_3$ and $v_3 = v_4$. Furthermore, at the tail of the rarefaction, we have $P_2(\text{RF tail}) = P_3$ and $v_2(\text{RF tail}) = v_3$ (see Fig. IV.5.5). To proceed, we use a version of $v_2$ from Eq. (IV.5.20) that has been rewritten to remove the explicit dependence on position [3, 9],

$$v_2 = (P_1^{\frac{\gamma-1}{2\gamma}} - P_2^{\frac{\gamma-1}{2\gamma}}) \left[ \left( 1 - \left( \frac{\gamma-1}{\gamma+1} \right)^2 \right) \frac{P_1^{\frac{1}{\gamma}}}{\left( \frac{\gamma-1}{\gamma+1} \right)^2 \rho_1} \right]^{\frac{1}{2}} . \tag{IV.5.23}$$

Eqs. (IV.5.22) and (IV.5.23) represent two curves in the $(P, v)$ plane. They intersect at the tail of the rarefaction, where regions 2 and 3 meet. Setting the two expressions equal, one can (via Newton iteration) find $P_3$ and thus $P_4$ and thus obtains the full solution.

Finally, the contact discontinuity propagates with $v_3 = v_4$, so its position at any time t is given by

$$x_{\text{CD}} = x_0 + v_3 t \ . \tag{IV.5.24}$$

An implementation of the exact solution of the Riemann problem as described here can be found, for example, on Frank Timmes's webpage [8]. For relativistic flow, an exact solution of the Riemann problem is still possible and described in [6], which also contains an implementation of the solution.

### IV.5.3 The Godunov Method and Finite-Volume Schemes

The nature of the Euler equations leads to weak solutions, that is solutions that satisfy the integral form of the Euler equations (i.e., conserve mass, momentum, and energy), but have discontinuities in the differential representation. This makes it difficult to discretize the Euler equations, since weak solutions can arise from perfectly smooth flow and lead to oscillatory behavior that must be controlled.

Von Neumann and Richtmyer came up with the concept of *artificial viscosity* [11] to deal with shocks and oscillations in numerical simulations. We have already mentioned artificial viscosity in the context of SPH in §IV.4.2.1. This method "smears out" the shock over multiple cells, in this way avoiding numerical instability, but sacrificing resolution.

In 1959, Godunov had a groundbreaking idea. Numerical data are by their very nature piecewise constant in the interior of a computational cell, but discontinuous at cell boundaries. A schematic view of this is given by Fig. IV.5.7. Godunov realized that by the very discretization of the domain one is faced with a sequence of local Riemann problems that, in principle, can be solved exactly! Most modern hydrodynamics codes make use of schemes going back to Godunov's idea.

One class of methods for numerical hydrodynamics making use of Godunov's idea are the so-called *finite-volume* schemes. These are fully conservative schemes (in the absence of source terms, i.e. external forces acting on the fluid), which means they conserve mass, momentum, and energy by construction.

Let us consider a quantity $q$. In computational cell $i$ with center coordinate $x_i$ and interfaces $x_{i-1/2}$ and $x_{i+1/2}$, the *cell average* of q is given by

$$\bar{q}_i = \frac{1}{\Delta x_i}\int_{x_{i-1/2}}^{x_{i+1/2}} q(x)dx \ , \tag{IV.5.25}$$

where $\Delta x_i = x_{i+1/2} - x_{i+1/2}$. Now the change of $q$ be governed by

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = 0 \ . \tag{IV.5.26}$$

Then the change of $q$ from $t_1$ to $t_2$ is

$$q(x,t_2) = q(x,t_1) - \int_{t_1}^{t_2}\frac{\partial}{\partial x}f(q(x,t))dt \ . \tag{IV.5.27}$$

Analoglously, the change of the cell average is

$$\bar{q}_i(t_2) = \frac{1}{\Delta x_i}\int_{x_{i-1/2}}^{x_{i+1/2}}\left\{q(x,t_1) - \int_{t_1}^{t_2}\frac{\partial}{\partial x}f(q(x,t))dt\right\}dx \ . \tag{IV.5.28}$$

Provided the flux $f$ is well behaved, the integration order may be reversed. Also, the flow is perpendicular to the unit area of the cell and, in 1D, $\partial f/\partial x \equiv \nabla f$, so we may use Gauss's law,

$$\int_V \nabla f d^3x = \int_{\partial V} f dS \ , \tag{IV.5.29}$$

to substitute the "volume" integral of $\partial f/\partial x$ with the values of $f$ at the "surface":

$$\bar{q}_i(t_2) = \bar{q}_i(t_1) - \frac{1}{\Delta x_i}\int_{t_1}^{t_2}\underbrace{\left[f(q(x_{i+1/2},t)) - f(q(x_{i-1/2},t))\right]}_{\text{"Flux Difference"}}dt \ . \tag{IV.5.30}$$

This equation is *exact* for the cell average $\bar{q}_i$ (no approximation has been made). This is straighforwardly extended to the multi-D case by integrating over each direction separately and adding up the changes, which still yields an exact results for regular grids.

There are many ways of implementing finite-volume/Godunov schemes. One of the simplest is to follow the semi-discrete approach, discretize in space only and then use the Method of Lines (see §III.12.3.6) to treat the semi-discretized equation as an ODE and integrate it with standard integrator like Runge-Kutta.

In the semi-discrete case, the spatial order of accuracy of the scheme is set by the accuracy with which the states $q$ at the cell interfaces $x_{i+1/2}$ and $x_{i-1/2}$ are "reconstructed". The temporal order of accuracy is set by the order of accuracy in which the time integral on the RHS of Eq. IV.5.30 is handled. All the art is in computing the *flux differences* using either the exact or approximate solutions of local Riemann problems.

### IV.5.4 Anatomy of a 1D Finite-Volume Hydro Code

In this section, we will discuss the anatomy of a simple 1D finite-volume hydrodynamics code with a focus on practical aspects rather than theoretical and mathematical completeness. Due to this, we will pick up applied-math technology on the way rather than introducing it from scratch. Readers interested in a more fundamental approach are referred to Toro's book [9] or Leveque's book [4].

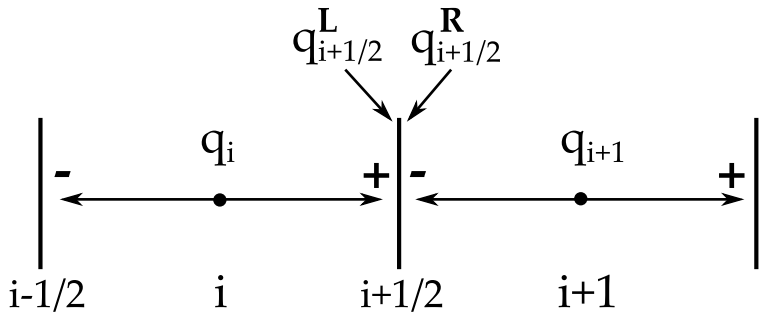



Figure IV.5.8: Schematic view of two computational cells of a finite volume hydrodynamics code. A local Riemann problem connecting states $q^L_{i+1/2}$ and $q^R_{i+1/2}$ must be solved at the interface $i+1/2$. $q^L_{i+1/2}$ and $q^R_{i+1/2}$ are obtained via interpolation of the cell centered states $q_i$ and $q_{i+1}$ (these are cell averages).

Our 1D code will implement the Euler equations (IV.5.7-IV.5.9) in flux-conservative form. The equation of state we will use is a $\gamma$-law:

$$P = (\gamma - 1)\rho\epsilon \ . \tag{IV.5.31}$$

We will set things up to solve the Riemann problem discussed in §IV.5.2.

We will follow the semi-discrete approach (see §III.12.3.6) and

(1) Determine states $q^L_{i+1/2}$ and $q^R_{i+1/2}$ at the cell interface $i+1/2$ via *reconstruction*.

(2) Solve the local Riemann problems at $i+1/2$ for all cells $i$.

(3) Compute the flux differences $f(x_{i+1/2}) - f(x_{i-1/2})$, then compute a first-order time update of $\bar{q}_i$.

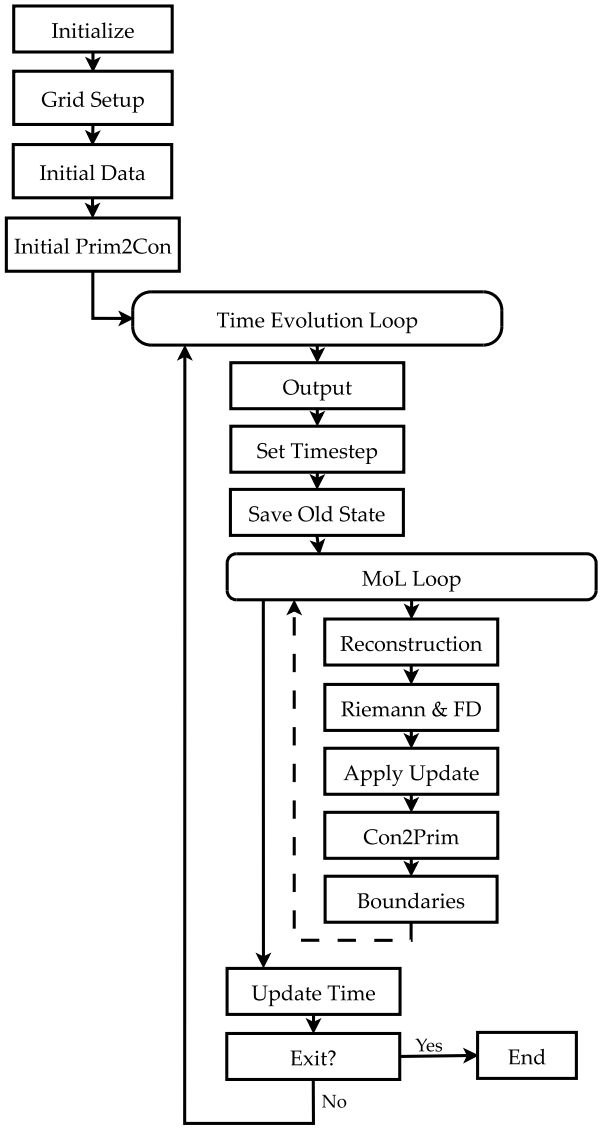
(4) Use the Method of Lines to repeat (1)-(3) multiple times to integrate in time.

### IV.5.4.1 Conserved and Primitive Variables

It is useful to make a distinction between *primitive* and *conserved* variables. We need the primitive variables for the equation of state (and other local physics), but for solving the Euler equations, we must use the conserved variables. So our code will need routines to go back and forth between conserved and primitive variables (`Con2Prim` and `Prim2Con`).

| Primitive | Conserved |
|---|---|
| $\rho$ | $\rho$ |
| $v$ | $\rho v$ |
| $\epsilon$ | $\rho\epsilon + \frac{1}{2}\rho v^2$ |
| $P$ | |

### IV.5.4.2 Program Flow Chart



#### IV.5.4.3 Grid Setup

The simplest thing to do is to use an equidistant grid covering the domain $[x_{\min}, x_{\max}]$. Set $x_{\min} = 0$ and $x_{\max} = 1$. For a possible later extension to non-equidistant grids, make sure not to hard-code in a constant $\Delta x$. Two arrays will be necessary: one containing cell center coordinates and one containing the coordinate of the inner cell interface.

Since we will need to also be able to solve a Riemann problem at the inner and outer grid cell interfaces, we need to introduce so-called *ghost cells* to provide us with boundary information. It is a safe thing to use 3 ghost cells at the inner and 3 ghost cells at the outer edge of the domain.

#### IV.5.4.4 Initial data

We will set up a shocktube-style Riemann problem. The two states are initially left and right of $x = 0.5$. For the left state, we will use $\rho_L = 1.0$, $P_L = 1.0$, $v_L = 0$ and for the right state $\rho_R = 0.1$, $P_R = 0.125$, $v_R = 0$. $\epsilon_L$ and $\epsilon_R$ are obtained via the EOS, for which we choose $\gamma = 5/3$.

#### IV.5.4.5 Equation of State

The equation of state will need to be called at various places to update the pressure $P$ and compute the speed of sound $c_s$.

We will use the $\gamma$-law,

$$P = (\gamma - 1)\rho\epsilon \ . \tag{IV.5.32}$$

For this EOS, the adiabatic speed of sound is given by

$$c_s^2 = \left.\frac{dP}{d\rho}\right|_s = \left.\frac{\partial P}{\partial \rho}\right|_\epsilon + \left.\frac{\partial P}{\partial \epsilon}\right|_\rho \frac{P}{\rho^2} \ . \tag{IV.5.33}$$

#### IV.5.4.6 Calculating the Timestep

We are going to be using explicit time integration, thus must obey the CFL limit. A general algorithm for calculating the timestep is

$$\Delta t_i = \frac{\Delta x_i}{\max(|v_i + c_{s,i}|, |v_i - c_{s,i}|)} \ , \tag{IV.5.34}$$

$$\Delta t = c_{\rm CFL} \min(\Delta t_i, i = 1, \cdots, N) \ , \tag{IV.5.35}$$

where the last min is the minimum of all $\Delta t_i$ ($i = 1, \cdots, N$, where $N$ is the number of cells). $c_{\rm CFL}$ is the Courant factor introduced in §III.12.3.2 and could be chosen to be 0.5 for stability.

### IV.5.5 Reconstruction

One reconstructs the primitive variables $\rho$, $\epsilon$, and $v$ at the cell interfaces, calls the EOS to compute the pressure $P$ at the interfaces. The conserved quantities at the interfaces are obtained by a call to `Prim2Con`.

The method chosen for reconstruction determines the spatial order of accuracy of the scheme and the precise way in which the primitive variables are interpolated to the cell interfaces can also have a strong influence on the stability of the overall scheme.

The simplest possible reconstruction method is *piecewise constant reconstruction* (PC reconstruction) in which the cell center values are just copied to the $-$ and $+$ interfaces. This leads to a stable evolution, but is only first-order accurate.

In the initial version of our hydro code, we will stick with PC reconstruction. We will talk more about reconstruction methods in §IV.5.8.

#### IV.5.5.1 Riemann Solver and Flux Differences

We need to solve the Riemann problem to calculate the physical flux at cell interfaces to allow us to compute the flux differences

$$FD_i = \frac{1}{\Delta x_i}\left[f(x_{i+1/2}) - f(x_{i-1/2})\right] \ . \tag{IV.5.36}$$

There are many exact and approximate Riemann solvers (e.g., [9]). We will use the particularly simple approximate solver of Harten, Lax, van Leer, and Einfeldt (HLLE). We will *not* derive this Riemann solver here (see [9] for the derivation), but rather will just provide a recipe for its implementation.

The HLLE solver does not use the characteristics (the eigenvectors of the Euler equations), but computes the approximate solution to the Riemann problem based only on the characteristic speeds (the eigenvalues of the Euler equations) $\lambda_i$ (Eq. IV.5.17).

First, one finds the minimum and the maximum characteristic speeds of both the left and right states:

$$s_{\max} = \max(\lambda_1^L, \lambda_2^L, \lambda_3^L, \lambda_1^R, \lambda_2^R, \lambda_3^R) \ , \tag{IV.5.37}$$

$$s_{\min} = \min(\lambda_1^L, \lambda_2^L, \lambda_3^L, \lambda_1^R, \lambda_2^R, \lambda_3^R) \ , \tag{IV.5.38}$$

The flux for conserved quantity $j$ through the interface $i+1/2$ is then given by the HLLE formula:

$$f_j(x_{i+1/2}) = \frac{s_{\max}F_j^L - s_{\min}F_j^R + s_{\min}s_{\max}(q_j^R - q_j^L)}{s_{\max} - s_{\min}} \ , \tag{IV.5.39}$$

where $q_j^R$ and $q_j^L$ are the reconstructed states (conserved variables) of equation $j$ right and left of the interface and $F_j^R$ and $F_j^L$ are the numerical fluxes left and right of the interface:

$$\begin{aligned} F_1^{R,L} &= \rho^{R,L}v^{R,L} \ , \\ F_2^{R,L} &= \rho^{R,L}v^{R,L}v^{R,L} + P^{R,L} \ , \\ F_3^{R,L} &= (\rho^{R,L}\epsilon^{R,L} + \frac{1}{2}\rho^{R,L}(v^{R,L})^2 + P^{R,L})v^{R,L} \ , \end{aligned} \tag{IV.5.40}$$

By using only the fastest and slowest characteristic speeds, the HLLE solver neglects the center characteristic speed corresponding to the contact discontinuity. This leads to somewhat smeared-out shocktubes. The HLLC (C for "contact") solver fixes this [9].

### IV.5.6 Boundary Conditions

Since we also need to compute fluxes through the inner and outer grid boundary, we need to use ghost cells whose data are set by the boundary condition(s) we choose. We will generally need more than one cell (unless we use piecewise constant reconstruction, in which case we will need only one).

For the shocktube problem, we choose outflow boundary conditions – this means that we will just copy whatever is in the last (or first, inner boundary) grid cell into the boundary cells.

Here is an example prescription for the inner boundary: If cell 0 is the innermost "real" grid cell, then copy the all data from this cell into the ghost cells with labels $-3$, $-2$, $-1$ (if negative array indices are not supported by the programming language of choice, one simply shifts all indices up to 0).

The same procedure must also be applied to the outer boundary at $x_{\max}$.

### IV.5.7 Method of Lines Integration

Since we have chosen a semi-discrete approach, we can now just use an off-the-shelf ODE integrator to discretize in time.

We could, for example, use a second-order Runge-Kutta (RK2) integrator:

$$\begin{aligned} q_i^{(1)} &= q_i^n - \frac{\Delta t}{2} FD_i(q^n) \ , \\ q_i^{n+1} &= q_i^n - \frac{\Delta t}{2} FD_i(q^{(1)}) \end{aligned} \tag{IV.5.41}$$

Here, $q_i^n$ is the state at time $n$, $q_i^{(1)}$ is the intermediate state and $q_i^{n+1}$ is the new state at time $n+1$.

### IV.5.8 More on Reconstruction

Reconstruction is the process of approximating the conserved variables (states) at the cell interfaces $i+1/2$ and $i-1/2$ for the flux calculation (see Fig. IV.5.8). The accuracy of reconstruction controls the spatial accuracy of the entire calculation. In the scheme laid out in the previous section, the primitive variables are reconstructed and the conserved variables are obtained via a call to `Prim2Con`.

So far, we have only discussed piecewise constant (PC) reconstruction:

$$\begin{aligned} \rho_+ = \rho_- &= \rho_i \ , \\ \epsilon_+ = \epsilon_- &= \epsilon_i \ , \\ v_+ = v_- &= v_i \ . \end{aligned} \tag{IV.5.42}$$

#### IV.5.8.1 The Total Variation Diminishing Property

PC reconstruction is a very rough approach, but is inherently stable, since no interpolation is used that could introduce noise to the system. The next logical step is to use linear interpolation, but for this we need to ensure that the amount of noise in the data does not increase. Ideally, we want to use a scheme that is *total variation diminishing* (TVD).

The total variation of discrete data $f_i$ at time $n$ is defined as

$$TV(f^n) = \sum_i |f_i^n - f_{i-1}^n| \ . \tag{IV.5.43}$$

A scheme that is TVD obeys

$$TV(f^{n+1}) \leq TV(f^n) \ \ \forall n \ . \tag{IV.5.44}$$

In practise, one realizes TVD for higher order schemes by dropping back to PC near discontinuities in the flow that could lead to noise/oscillations. In piecewise linear (PL) reconsturction, one speaks of *limiting* the slope.

#### IV.5.8.2 Piecewise Linear Reconstruction

In PL reconstruction, we interpolate linearly to the $+$ and $-$ interfaces of grid cell $i$, using information from cell $i$, $i-1$, and $i+1$. This seems trivial, though it is a true art to pick the right *slope limiter* that drops the reconstruction scheme down to PC near a discontinuity. This is a topic on which many computational physics papers have and continue to be written.

The simplest slope limiter one can imagine is the so-called *minmod* limiter. It is defined by the following lines of pseudocode:

```
minmod(a,b):
  if (a*b < 0):
      minmod = 0
  elif (|a| < |b|):
      minmod = a
  else:
      minmod = b

  return minmod
```

In the above, $a$ and $b$ are two approximations to the slope of a quantity $u$ (i.e. $u_i - u_{i-1}$ and $u_{i+1} - u_i$). If the slope changes sign ($ab < 0$), then we are most clearly at an extremum and should drop back to PC (minmod = 0). Otherwise the smaller slope is used.

With this, we can write down minmod-limited PL reconstruction for cell $i$ and quantity $u$:

$$\begin{aligned} du_{\rm up} &= \frac{u_i - u_{i-1}}{x_i - x_{i-1}} \\ du_{\rm down} &= \frac{u_{i+1} - u_i}{x_{i+1} - x_i} \\ \Delta &= \texttt{minmod}(du_{\rm up}, du_{\rm down}) \\ u_+ &= u_i + \Delta(x_{i+1/2} - x_i) \\ u_- &= u_i - \Delta(x_i - x_{i-1/2}) \end{aligned} \tag{IV.5.45}$$

minmod-limited PL reconstruction gives already much better results than PC reconstruction. However, one can always do better! For example, van Leer came up [10] with the so-called *monotonized central* (MC) limiter that is only slightly more complicated than minmod, but gives significantly more accurate results. In pseudocode, the computation of $\Delta$ in Eq. (IV.5.45 is replaced with:

```
if(du_up * du_down < 0):
    Delta = 0
else
    Delta = signum( min( 2*abs(du_up),
                         2*abs(du_down),
                         0.5*( abs(du_up) + abs(du_down) ) ),
                    du_up + du_down )
```

where

$$\text{signum}(x,y) = \begin{cases} |x| & \text{, if } y \geq 0 \text{ ,} \\ -|x| & \text{, if } y < 0 \text{ .} \end{cases} \tag{IV.5.46}$$

There exist, of course, yet higher order methods. The piecewise parabolic method (PPM) [1], which provides fourth-order accurate reconstruction in smooth parts of the flow. Other alternatives are the essentially non-oscillatory method (ENO) [7] and the weighted ENO (WENO) method [5].

### IV.5.9 The Euler Equations in Spherical Symmetry

Many astrophysical systems and phenomena are approximately spherical (stars, globular clusters, stellar winds, planets, supernovae) and can, in a lowest-order approach, be modeled with the greatly simplifying assumption of spherical symmetry.

In spherical symmetry, the Newtonian Euler equations read:

$$\frac{\partial}{\partial t}\rho + \frac{1}{r^2}\frac{\partial}{\partial r}(r^2\rho v) = 0 \text{ ,} \tag{IV.5.47}$$

$$\frac{\partial}{\partial t}(\rho v) + \frac{1}{r^2}\frac{\partial}{\partial r}(r^2\rho v v) + \frac{\partial}{\partial r}P = -\rho\frac{\partial}{\partial r}\Phi \text{ ,} \tag{IV.5.48}$$

$$\frac{\partial}{\partial t}(\rho\epsilon + \frac{1}{2}\rho v^2) + \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2(\rho\epsilon + \frac{1}{2}\rho v^2 + P)v\right) = -v\rho\frac{\partial}{\partial r}\Phi \text{ .} \tag{IV.5.49}$$

Here we have included the Newtonian gravitational force contributions on the RHS. In 1D, of course, $\partial\Phi/\partial r = GM(r)/r^2$. Note also that, in the momentum equation, the term $\partial P/\partial r$ outside the flux term arises from the fact that the term involving the velocity is a divergence, whereas the term involving the gradient is just a pressure. This can lead to non-conservation of momentum and ideally one wants to always include any derivatives of thermodynamic quantities as part of the Riemann problem. For this reason, it is best to rewrite the momentum equation to

$$\frac{\partial}{\partial t}(\rho v) + \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2(\rho v v + P)\right) = -\rho\frac{\partial}{\partial r}\Phi + \frac{2}{r}P \text{ .} \tag{IV.5.50}$$

When implementing the spherically-symmetric Euler equations in a finite-volume hydrodynamics code, one must keep in mind that the solution of the Riemann problem is purely local, so locally flat geometry is a good assumption. Geometry terms come only in in the computation of the flux differences. The formula for this reads

$$FD_i = \frac{1}{\Delta r^*}\frac{1}{r_i^2}\left[r_{i+1/2}^2 f_{i+1/2} - r_{i-1/2}^2 f_{i-1/2}\right] \text{ ,} \tag{IV.5.51}$$

where there are multiple ways of writing $\Delta r^*$, namely:

$$\Delta r^* = r_{i+1/2} - r_{i-1/2} \ , \tag{IV.5.52}$$

or

$$\Delta r^* = \frac{r_{i+1/2}^3 - r_{i-1/2}^3}{3r_i^2} \ . \tag{IV.5.53}$$

Note that our choice of cell centering of all variables saves us from potential problems near $r = 0$, since a cell interface will be located there and we will never have to devide by the cell interface coordinate.

## References


[1] P. Colella and P. R. Woodward. The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. *J. Comp. Phys.*, 54:174, 1984.

[2] R. Courant and K. O. Friedrichs. *Supersonic Flow and Shock Waves*. Springer Verlag, New York, NY, USA, 1976.

[3] J. F. Hawley, L. L. Smarr, and J. R. Wilson. A numerical study of nonspherical black hole accretion. I Equations and test problems. *Astrophys. J.*, 277:296, February 1984. doi: 10.1086/161696.

[4] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, Cambridge UK, 2002.

[5] X.-D. Liu, S. Osher, and T. Chan. Weighted essentially non-oscillatory schemes. *J. Comp. Phys.*, 115:200, 1994.

[6] J. M. Martí and E. Müller. Numerical Hydrodynamics in Special Relativity. *Liv. Rev. Rel.*, 6:7, December 2003.

[7] C. W. Shu. High Order ENO and WENO Schemes for Computational Fluid Dynamics. In T. J. Barth and H. Deconinck, editors, *High-Order Methods for Computational Physics*. Springer, 1999.

[8] F. Timmes. An exact solution to the riemann problem (source code). URL http://cococubed.asu.edu/code_pages/exact_riemann.shtml.

[9] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, Berlin, Germany, 1999.

[10] B. J. van Leer. *J. Comp. Phys.*, 23:276, 1977.

[11] J. von Neumann and R. D. Richtmyer. *J. Appl. Phys.*, 21:232, 1950.

# IV.6 Radiation Transport

## IV.6.1 The Boltzmann Equation

We have already briefly talked about the Boltzmann equation in the context of hydrodynamics. Here we will go into somewhat more detail. The presentation in this section is based largely on Bodenheimer et al. [1].

We consider an ensemble of identical particles in the 6D phase space spanned by particle position $\mathbf{x}$ and particle momentum $\mathbf{p}$. The particles are assumed to have no internal degrees of freedom. The distribution function $f(\mathbf{x}, \mathbf{p}, t)$ gives the number of particles in the phase space volume $[\mathbf{x}, \mathbf{x} + d\mathbf{x}] \times [\mathbf{p}, \mathbf{p} + d\mathbf{p}]$,

$$dN = f(\mathbf{x}, \mathbf{p}, t) d\mathbf{x} d\mathbf{p} \ . \tag{IV.6.1}$$

Unless special conditions are met, the distribution function is not known. Examples of known distribution functions are:

| | |
|---|---|
| Maxwell-Boltzmann: | classical gases in equilibrium |
| Planck: | photons in equilibrium |
| Fermi-Dirac: | fermions in equilibrium (neutrinos, electrons, ...) |

Let $\mathbf{F}$ be a force field. Then, in time $dt$, the momentum of any particle will change to

$$\mathbf{p} + m \cdot \mathbf{F} dt \ , \tag{IV.6.2}$$

and its position will change to

$$\mathbf{x} + \frac{\mathbf{p}}{m} dt \ . \tag{IV.6.3}$$

Since the number of particles in a comoving volume of phase space does not change (Liouville's Theorem), we may write

$$f(\mathbf{x} + \frac{\mathbf{p}}{m} dt, \mathbf{p} + m \cdot \mathbf{F} dt, t + dt) - f(\mathbf{x}, \mathbf{p}, t) = 0 \ . \tag{IV.6.4}$$

Particle creation/distruction and collisions (i.e. change of momentum) can be taken into account by a collision term $[\Delta f]_{\text{coll}}$. The left-hand-side of Eq. (IV.6.4) is the change of the distribution during an interval $dt$. Hence, we may expand Eq. (IV.6.4) to

$$\frac{\partial f}{\partial t} + u_1 \frac{\partial f}{\partial x_1} + u_2 \frac{\partial f}{\partial x_2} + u_3 \frac{\partial f}{\partial x_3} + m_1 F_1 \frac{\partial f}{\partial p_1} + m_2 F_2 \frac{\partial f}{\partial p_2} + m_3 F_3 \frac{\partial f}{\partial p_3} = \left[ \frac{\partial f}{\partial t} \right]_{\text{coll}} \ , \tag{IV.6.5}$$

using $u_i = p_i / m$. More concisely, we have

$$\frac{\partial f}{\partial t} + \frac{\mathbf{p}}{m} \nabla_{\mathbf{x}} f + m \mathbf{F} \nabla_{\mathbf{p}} f = \left[ \frac{\partial f}{\partial t} \right]_{\text{coll}} \ , \tag{IV.6.6}$$

which is the usual form of the Boltzmann equation.

Processes that contribute to the collision term are creation (*emission*), destruction (*absorption*), and non-destructive collision (*scattering*).

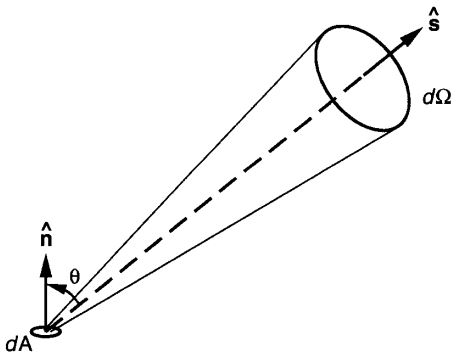



Figure IV.6.1: Illustration of the definition of the specific intensity $I$. Radiation is propagating in the direction $\mathbf{s}$, inclined at an angle $\theta$ from the normal $\mathbf{n}$ to the area $dA$, into a narrow cone of solid angle $d\Omega$ about the direction $\mathbf{s}$. This figure is taken from [1]

### IV.6.2 The Radiation Transport Equation

In the following, we will introduce the basics of radiation transport (also called *radiative transfer*). All equations will be generally applicable to any kind of radiation, be it photons, neutrinos, or other particles (e.g., neutrons in a nuclear reactor).

$dE_\nu$ is the energy (of photons, for example) that passes through an area $dA$ (with unit normal $\mathbf{n}$) at the point $\mathbf{x}$, coming from a direction $\mathbf{s}$, within the solid angle $d\Omega$, in the energy interval $[\epsilon, \epsilon + d\epsilon]$ (or frequency interval $[\nu, \nu + d\nu]$), in time $dt$. The radiation *specific intensity* is then defined via

$$dE_\nu = I_\nu(\mathbf{x}, \mathbf{s}, \nu, t)\, \mathbf{n}\cdot\mathbf{s}\, dA\, d\Omega\, d\nu dt \ , \tag{IV.6.7}$$

or

$$dE_\nu = I_\nu(\mathbf{x}, \mathbf{s}, \epsilon_\nu, t)\, \mathbf{n}\cdot\mathbf{s}\, dA\, d\Omega\, d\epsilon_\nu dt \ . \tag{IV.6.8}$$

Note that these two definitions of the specific intensity $I_\nu$ must differ only by a constant factor.

In the above, $\mathbf{n}\cdot\mathbf{s}$ can also be written as $dA\cos\theta$, where $\theta$ is the angle between the direction of propagation and the normal to the surface $dA$ (see Fig. IV.6.1).

If we introduce a spherical coordinate system also for the position $\mathbf{x}$ by describing it with $\{r, \theta', \varphi'\}$, we can write out the dimensionality of the radiation transport problem in typical scenarios:

| Spacial Dimensionaly | Symmetry | $I = I(...)$ | Problem Dimensionalty |
|---|---|---|---|
| 1 | spherical symmetry | $r, \theta, \nu, t$ | $3+1$ |
| 2 | axisymmetry | $r, \theta', \theta, \varphi, \nu, t$ | $5+1$ |
| 3 | none | $r, \theta', \varphi', \theta, \varphi, \nu, t$ | $6+1$ |

Angular variables with a prime ($'$) in the above stand for position space, while the un-primed stand for momentum space and $\nu$ symbolizes the energy/frequency dependence of the radiation field.

Now going back to the distribution function and the Boltzmann equation: We know that the number of particles (e.g. photons) in $[\mathbf{x}, \mathbf{x} + d\mathbf{x}] \times [\mathbf{p}, \mathbf{p} + d\mathbf{p}]$ is $f(\mathbf{x}, \mathbf{p}, t) d\mathbf{x} d\mathbf{p}$. Assuming that $\epsilon_\nu = h\nu$, as appropriate for photons,

$$dE_\nu = h\nu f(\mathbf{x}, \mathbf{p}, t) d\mathbf{x} d\mathbf{p} \ . \tag{IV.6.9}$$

Now, we can furthermore write

$$\begin{aligned} d\mathbf{x} &= cdt\,(\mathbf{n}\cdot\mathbf{s})dA\ , \\ d\mathbf{p} &= p^2\,dp\,d\Omega\ , \end{aligned} \tag{IV.6.10}$$

and for massless particles, $E = pc$, or, specifically for photons, $p = h\nu/c$ and $dp = h/cd\nu$. Substituting this into (IV.6.9), we obtain

$$dE_\nu = \left(\frac{h\nu}{c}\right)^2 \left(\frac{h^2\nu}{c}\right)\, d\nu\, d\Omega\, c\, dt(\mathbf{n}\cdot\mathbf{s})\, dA\, f(r,\theta',\varphi',\theta,\varphi,\nu,t)\ , \tag{IV.6.11}$$

and we identify

$$I_\nu = \frac{h^4\nu^3}{c^2} f\ . \tag{IV.6.12}$$

$I_\nu$ has the dimension energy per (solid angle, area, time, frequency). For particle radiation, we may be interested in expressing $I_\nu$ as energy per (solid angle, area, time, particle energy). In this case,

$$I_\nu = \frac{\epsilon_\nu^3}{h^3c^2} f\ . \tag{IV.6.13}$$

The Boltzmann transport equation can now be rewritten in in terms of the specific intensity. With that we arrive at the radiation transport equation (for massless particles in Newtonian gravity, Euclidian space):

$$\frac{1}{c}\frac{\partial}{\partial t} I_\nu + (\mathbf{s}\nabla) I_\nu = \frac{1}{c}\frac{\partial}{\partial t} I_\nu \bigg|_{\text{coll}}\ , \tag{IV.6.14}$$

where the collision term encompasses emission, absorption, and scattering.

## IV.6.3 Optically Thin and Thick Limits of the Transport Problem

### IV.6.3.1 Thick Limit

In the optically-thick limit, radiation and matter are in equilibrium at high optical depth ($\tau \gg 1$). In this scenario, emission equals absorption. Examples for such situations may be be local thermodynamic equilibrium inside a star for photons or, for neutrinos, weak ($\beta$) equilibrium inside a hot newborn neutron star.

In equilibrium, Kirchhoff's Law states:

$$\eta_\nu = \sigma_\nu^a B_\nu(T) = \sigma_\nu^a I_\nu^{\text{eq}}\ . \tag{IV.6.15}$$

In LTE conditions, the radiation flux will be small and will be physically driven by temperature gradients. Considering the radiation momentum equation (assuming isotropic scattering):

$$\frac{1}{c}\frac{\partial H_\nu^i}{\partial t} + \nabla K_\nu + (\sigma_\nu^a + \sigma_\nu^s) H_\nu^i = 0 \tag{IV.6.16}$$

### IV.6.3.2 Thin Limit

In the optically-thin limit, $\tau \ll 1$ and there is essentially no radiation-matter interaction. In this limit, $I_\nu = const.$ and the radiation is *streaming freely*. In this case all radiation is going into the same direction and $F_\nu = cE_\nu$.

### IV.6.4 Flux-Limited Diffusion

One common approach to radiation transport is to simply solve the 0-th moment equation for $J_\nu$,

$$\frac{1}{c}\frac{\partial}{\partial t}J_\nu + \frac{1}{4\pi}\nabla \mathbf{F}_\nu = \eta_\nu - \sigma_\nu^a J_\nu \ , \tag{IV.6.17}$$

and to use the diffusion approximation for the flux term:

$$\mathbf{F}_\nu \approx -\frac{4\pi}{3\sigma_\nu^t}\nabla J_\nu \ . \tag{IV.6.18}$$

What is the problem with this? Well, it only works in the optically thick limit. In the thin limit, $J_\nu \to |\mathbf{F}_\nu|/4\pi$, but $\sigma_\nu^t \to 0$ and $\mathbf{F}_\nu$ blows up. It must be *limited* to reprodcue the free-streaming limit.

The *flux limiter* $\Lambda$ must be defined in a way that it reproduces the diffusion limit at high optical depth and the free streaming limit at low optical depth. If we treat it as a number to be multplied with Eq. (IV.6.18), we know that for $\tau \gg 1$, $\Lambda = 1$. For finding its value in the free streaming limit, we recall that in this case, the radiation field is pure flux, so

$$\mathbf{F}_\nu = -4\pi J_\nu \frac{\nabla J_\nu}{|\nabla J_\nu|} \ , \tag{IV.6.19}$$

where the last factor sets the direction of the flux. Hence,

$$\Lambda(\tau \ll 1) = 3\sigma_\nu^t \frac{\nabla J_\nu}{|\nabla J_\nu|} \ . \tag{IV.6.20}$$

The art is now to interpolate $\Lambda(\tau)$ between $\tau \gg 1$ and $\tau \ll 1$. An example way of doing this was proposed by Bruenn [2]:

$$\begin{aligned} \mathbf{F}_\nu &= -4\pi\Lambda\nabla J_\nu \\ \Lambda(J_\nu, \sigma_\nu^t) &= \frac{D}{1 + D\frac{|\nabla J_\nu|}{J_\nu}} = \frac{1}{\frac{1}{D} + \frac{|\nabla J_\nu|}{J_\nu}} \ , \end{aligned} \tag{IV.6.21}$$

where $D = 1/3\sigma_\nu^t$. It is easy to verify that this flux limiter produces the required limits.

## References

[1] P. Bodenheimer, G. P. Laughlin, M. Rozyczka, and H. W. Yorke. *Numerical Methods in Astrophysics, An Introducction*. CRC Press, Taylor & Francis Group, Boca Raton, FL, USA, 2007.

[2] S. W. Bruenn. Stellar core collapse - Numerical model and infall epoch. *Astrophys. J. Suppl. Ser.*, 58:771, August 1985.