Heidelberg Institute for Theoretical Studies
Schloss-Wolfsbrunnenweg 35
69118 Heidelberg

Zentrum für Astronomie, Universität Heidelberg
Astronomisches Recheninstitut
Mönchhofstr. 12-14
69120 Heidelberg

WS2015/16

Lecture Notes for MVComp1

Fundamentals of Simulation Methods

Prof. Dr. Volker Springel

February 3, 2016

Contents

Page

1 Issues of Floating Point Math **7**
- 1.1 Integer arithmetic . . . 7
  - 1.1.1 Two's complement for negative numbers: . . . 8
  - 1.1.2 Common integer types in C . . . 8
  - 1.1.3 Things to watch out for in integer arithmetic . . . 8
- 1.2 Floating point arithmetic . . . 10
  - 1.2.1 The set of representable numbers . . . 12
  - 1.2.2 Double and higher precision . . . 15

2 Integration of ordinary differential equations **17**
- 2.1 Explicit Euler method . . . 18
- 2.2 Implicit Euler method . . . 18
- 2.3 Implicit midpoint rule . . . 19
- 2.4 Runge-Kutta methods . . . 20
- 2.5 Adaptive step sizes . . . 21
- 2.6 The leapfrog . . . 22
- 2.7 Symplectic integrators . . . 23

3 Collisionless particle systems **27**
- 3.1 N-particle ensembles . . . 27
- 3.2 Uncorrelated (collisionless) systems . . . 28
- 3.3 When is a system collisionless? . . . 30
- 3.4 N-body models of collisionless systems . . . 32

4 Tree algorithms **33**
- 4.1 Multipole expansion . . . 34
- 4.2 Hierarchical grouping . . . 35
- 4.3 Tree walk . . . 37

5 The particle-mesh technique **39**
- 5.1 Mass/charge assignment . . . 39
  - 5.1.1 Nearest grid point (NGP) assignment . . . 40
  - 5.1.2 Clouds-in-cell (CIC) assignment . . . 41
  - 5.1.3 Triangular shaped clouds (TSC) assignment . . . 42
- 5.2 Solving for the gravitational potential . . . 43
- 5.3 Calculation of the forces . . . 43

5.4 Interpolating from the mesh to the particles 44

6 Force calculation with Fourier transform techniques 47
6.1 Convolution problems 47
6.2 The discrete Fourier transform (DFT) 49
6.3 Storage conventions for the DFT 51
6.4 Non-periodic problems with 'zero padding' 53

7 Iterative solvers and the multigrid technique 55
7.1 The Poisson equation as a linear system of equations 55
7.2 Jacobi iteration 56
7.3 Gauss-Seidel iteration 57
7.3.1 Red black ordering 58
7.4 The multigrid technique 58
7.4.1 Prolongation and restriction operations 59
7.4.2 The multigrid V-cycle 60
7.4.3 The full multigrid method 62

8 Molecular dynamics simulations 65
8.1 Simple interaction potentials 65
8.2 Statistical mechanics aspects 67
8.2.1 Temperature adjustment 67
8.3 Practical aspects 69
8.3.1 Initial conditions and boundary conditions 69
8.3.2 Finite range interactions 69
8.3.3 Time integration 71

9 Basic gas dynamics 73
9.1 Euler and Navier-Stokes equations 73
9.1.1 Euler equations 73
9.1.2 Navier-Stokes equations 74
9.1.3 Scaling properties of viscous flows 75
9.2 Shocks 76
9.3 Fluid instabilities 77
9.3.1 Stability of a shear flow 77
9.4 Turbulence 80
9.4.1 Kolmogorov's theory of incompressible turbulence 80
9.4.2 Energy spectrum of Kolmogorov turbulence 82

10 Eulerian hydrodynamics 85
10.1 Types of PDEs 85
10.2 Solution schemes for PDEs 89
10.3 Simple advection 90
10.4 Riemann problem 95

10.5 Finite volume discretization . . . 98
10.6 Godunov's method and Riemann solvers . . . 100
10.7 Extensions to multiple dimensions . . . 101
10.7.1 Dimensional splitting . . . 102
10.7.2 Unsplit schemes . . . 103
10.8 Extensions for high-order accuracy . . . 104

11 Smoothed particle hydrodynamics 107
11.1 Kernel Interpolants . . . 107
11.2 Variational Derivation of SPH . . . 110
11.3 Artificial Viscosity . . . 113
11.4 Advantages and disadvantages of SPH . . . 116

12 Finite element methods 117
12.1 Classic finite element methods for linear PDEs . . . 117
12.2 Discontinuous Galerkin methods . . . 121
12.2.1 Solution representation . . . 122
12.2.2 Initial conditions . . . 124
12.2.3 Gauss-Legendre quadrature . . . 124
12.2.4 Evolution equation for the weights . . . 126
12.2.5 Efficiency of DG schemes . . . 127

13 Monte Carlo Techniques 129
13.1 Monte Carlo Integration . . . 129
13.2 Error in Monte Carlo integration . . . 130
13.3 Importance Sampling . . . 133
13.4 Random number generation . . . 134
13.4.1 Pseudo-random numbers . . . 135
13.5 Using random numbers . . . 137
13.5.1 Exact inversion . . . 137
13.5.2 Rejection method . . . 139
13.5.3 Sampling with a stochastic process . . . 140
13.5.4 The Metropolis-Hastings algorithm . . . 142
13.6 Monte Carlo simulations of lattice models . . . 144
13.7 Monte Carlo Markov Chains in parameter estimation . . . 146

14 Parallelization techniques 149
14.1 Hardware overview . . . 149
14.1.1 Serial computer . . . 149
14.1.2 Multi-core nodes . . . 150
14.1.3 Multi-socket compute nodes . . . 150
14.1.4 Compute clusters . . . 151
14.1.5 Device computing . . . 152
14.1.6 Vector cores . . . 152

14.1.7 Hyperthreading 153
14.2 Ahmdahl's law 153
14.3 Shared memory parallelization with OpenMP 154
14.3.1 OpenMP's "fork-join-model" 155
14.3.2 Loop-level parallelism with OpenMP 155
14.3.3 Race conditions 157
14.4 Distributed memory parallelization with MPI 159
14.4.1 General structure of an MPI program 160
14.4.2 A simple point to point message 161
14.4.3 Collective communications 161

References **161**

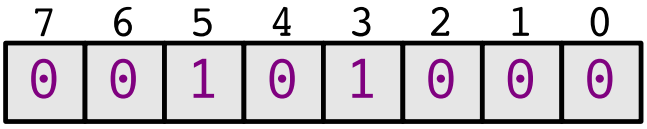
1 Issues of Floating Point Math

Numerical calculations on standard computers can sometimes lead to unexpected results due to complications such as overflow or round-off errors. It is important to be aware of the different things that can go wrong, and to adopt suitable precautions where possible. In this introductory section we recapitulate how standard computer languages handle arithmetic operations in order to get a more precise understanding of these issues.

1.1 Integer arithmetic

- Integers are whole numbers that computers represent in binary.
- The representable range depends on the available storage size of the chosen variable type.
- Negative numbers are represented as so-called two's complement.

The basic unit is the byte with 8 bits. We can number the bits from 0 (the 'least significant bit', or LSB) to 7 (the most significant bit, MSB). Example:



This number has the value

$$2^5 + 2^3 = 32 + 8 = 40. \tag{1.1}$$

The C-types for single bytes are:

type	range of values
`unsigned char`	$\{0, \ldots, 255\}$
`(signed) char`	$\{-128, -127, \ldots, 0, 1, \ldots, 127\}$

1.1.1 Two's complement for negative numbers:

- The MSB bit flags negative numbers.
- To change the sign of an integer number, one needs two carry out the following steps:
  1. invert all bits
  2. add +1 to the number

Example: Find the binary representation of -9.
start with +9: `00001001`
invert all bits: `11110110`
add +1: `11110111` that's it!
With the two's complement, no special treatment is necessary when adding negative numbers to positive numbers.

1.1.2 Common integer types in C

C-name	bits	range		
`char`	8	-128	...	$+127$
`short`	16	-32768	...	$+32767$
`int`	32	-2147463648	...	$+2147463647$
`long long`	64	-9.2×10^{18}	...	$+9.2 \times 10^{18}$
	n	-2^{n-1}	...	$+2^{n-1}-1$

One can also use unsigned versions of these types (by adding `unsigned` in front), but it is recommended to do this only in exceptional cases because of the danger of nasty surprises in case a negative number is attempted to be stored in such a type.

1.1.3 Things to watch out for in integer arithmetic

The largest danger is **overflow**. Here is an example:

```
char a, b, c;
a = 100;
b = 5;
c = a * b;
```

When one tries to print out the variable c, one gets: -12 !
Why? `a * b = 500` has binary representation

```
....000111110100
```

This will now be truncated to just the 8 least significant bits, because the result is stored in a variable of type char. So one gets:

```
11110100
```

Because the most significant bit is set, this will be interpreted as a negative number. We can apply the two's complement to flip the sign and obtain the absolute value of the number. After the two's complement, we have

```
00001100
```

This corresponds to the value +12.

So when using integer variables, always be aware of their finite range. In the C-language, the type `int` is often sufficient as universal counting and indexing variable. But increasingly often, its range of about 2 billion may not be enough any more for large simulations. In this case, the use of 64-bit integer variables should be considered. (But using this for everything will cost performance and typically result in a waste of storage space.)

Another issue where some caution is in order is **integer division**. In most languages, this is defined to always truncate all dezimal places, i.e.

```
 8 /  3 =  2

-8 /  3 = -2

 8 / -3 = -2
```

Also watch out for the definition of the **modulus** operation in your language of choice. In C, the syntax for `8 modulus 3` is `8 % 3`, and the result is defined as `n modulus m = n - (n/m)*m`. Hence:

```
 8 %  3 =  2

-8 %  3 = -2

 8 % -3 =  2
```

Finally, another thing to watch out for are **implicit type conversions**. Suppose we have

```
char a = 85;
char b = 5;
int  c = a * b;
```

When we check the value of c we get the correct result 425, and not -87 that we would get due to overflow if the type of c would also be `char`. This happens because C does an automatic, implicit promotion of a and b to `signed int`, and only then carries out the arithmetic operation. But now contrast this with:

```
       int  a = 2000000000;
       int  b = 3;
long long  c = a * b;
```

When we now check the value of c, we get `1705032704`, which is equal to $6 \times 10^9 - 2^{32}$. Apparently, here our operation has overflown and was not rescued by implicit type conversion, because C will at most use `int` for this. To get the correct result, we have to force a type conversion ourselves, for example in the following form:

```
long long  c = a * ((long long)b);
```

This now yields the correct result of `6000000000`.

1.2 Floating point arithmetic

In general, floating point representations have a

- base β
- precision p (the 'number of digits')
- exponent e

For example, if $\beta = 10$ and $p = 4$, the representations of the two numbers 0.1 and 523 are

number	representation ($\beta = 10$, $p = 4$)
0.1	1.000×10^{-1}
523	5.230×10^{2}

Nowadays, floating point calculations on computers are most commonly done according to the IEEE-754 standard, using a binary base $\beta = 2$. In particular, single precision numbers use $\beta = 2$ and $p = 23(+1)$. This gives the representations

number	representation ($\beta = 2$)
0.1	$1.10011001100110011001101 \times 2^{-4}$
523	$1.00000101100000000000000 \times 2^{9}$

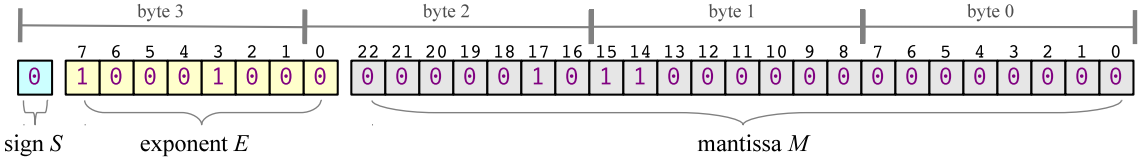
The first encoding has the value $(2^0 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + \ldots) \times 2^{-4} \simeq 0.0996$, which differs from 0.1. In fact, 0.1 cannot be represented exactly in the binary format.

Some further notes:

- The representation is not unique – by shifting the exponent, one can arrive at other representations. This is addressed by *normalization*. Every number is stored as convention in the standard such that the leading digit before the dot is 1.

- If this is the case, then the "1" doesn't have to be stored anymore (since it's always there anyway), and one gains a bit of significance if this is exploited. Hence p is really 24 for single precision IEEE-754 numbers. However, representing zero requires a special solution in this case.
- Some numbers cannot be represented exactly – such as the 0.1 from the example.
- The IEEE standard defines a smallest and largest exponent. For single precision numbers this is $e_{\min} = -126$ and $e_{\max} = +127$. This restricts the range of representable numbers, i.e. there is a minimum and a maximum value that is possible, and values beyond this will cause an over- or underflow.

The storage scheme for single-precision IEEE-754 numbers is as follows:



The exponent is stored here in the 8 bits reserved for it not as a two-complement, but rather in a *biased* way, defined through

$$E = e + 127, \tag{1.2}$$

where E is stored as an unsigned byte. Because the range for e is restricted for regular floating point numbers, we have for them $1 \leq E \leq 254$. The values $E = 0$ and $E = 255$ fulfil a special purpose, see below.

The bits are set for the example of encoding the number 523 as a single precision floating point number. In binary representation, 523 can be written in normalized form as

$$1.000001011 \times 2^9 \tag{1.3}$$

The exponent is hence $e = 9$, such that $E = 136$. Because one of the exponent bits is moved to byte 2 in the encoding, byte 3 will then have the value 68. Byte 2 has the value 2, byte 1 the value 192, and byte 0 the value 0.

On little Endian computers (Intel, AMD chips), these bytes will be stored in memory from right to left, with the least significant byte first, i.e. in the sequence 0, 192, 2, 68. On big Endian machines (e.g. PowerPC architecture), the sequence is reversed.

In general, the value f of a IEEE floating point number is

$$f = (-1)^s \cdot \left(1 + \frac{M}{2^p}\right) \times 2^{E-b}, \tag{1.4}$$

where here s is the sign bit, M the integer number representing the mantissa, and E is the exponent. For single precision, we have $p = 23$ and the bias is $b = 127$.

The exponent values $E = 0$ and $E = 255$ are used to represent special values:

1. $E = 0$, $M = 0$: This is zero. Actually, one can have ± 0, depending on the sign bit.

2. $E = 255$, $M = 0$: By convention, this represents $\pm\infty$, depending on the sign bit.

3. $E = 255$, $M \neq 0$: This signals "not a number" (`NaN`), which can result as part of an invalid mathematical operation (division by 0, or square root of a negative number).

4. $E = 0$, $M \neq 0$: These are so-called *denormalized numbers*, because here no leading 1 in front of the dot can be assumed. The value of these numbers is given by

$$f = (-1)^s \cdot \frac{M}{2^p} \times 2^{-b+1} \tag{1.5}$$

Machine precision

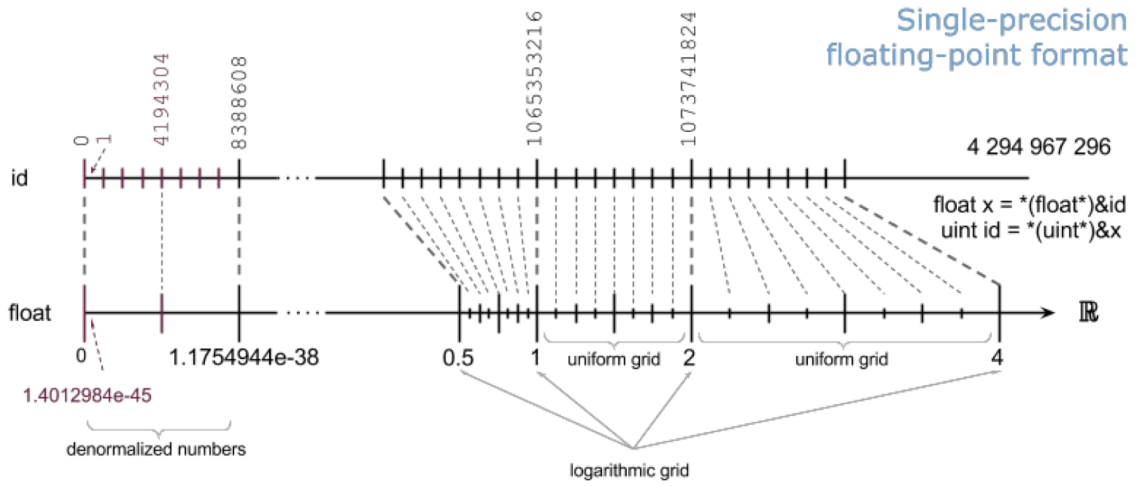
The smallest increment in the mantissa,

$$\epsilon_m = \frac{1}{2^p}. \tag{1.6}$$

is called machine precision. This can be coarsely interpreted as the smallest relative spacing between two floating point numbers that can still be distinguished by the representation scheme. For single precision this is $\epsilon_m \sim 1.19 \times 10^{-7}$.

1.2.1 The set of representable numbers

The set of floating point numbers is finite and subdivides the reals in a special kind of logarithmic grid. (figure below by Denis Yurin)



The largest representable single precision number is

$$f_{\max} = \left(1 + \frac{2^p - 1}{2^p}\right) \times 2^{127} \simeq 3.403 \times 10^{38}, \tag{1.7}$$

while the smallest positive normalized number is

$$f_{\min} = (1 + 0) \times 2^{-126} \simeq 1.175 \times 10^{-38}. \tag{1.8}$$

But actually, thanks to denormalized numbers, the smallest possible single precision value is

$$f_{\text{smallest}} = \frac{1}{2^p} \times 2^{-126} \simeq 1.4 \times 10^{-45}. \tag{1.9}$$

Due to the introduction of denormalized numbers, IEEE-754 guarantees that for $x \neq y$ one always has $x - y \neq 0$. This is meant to protect, for example, against the common bug-prone idiom:

```
if  x != y  then z = 1.0 / (x-y)
```

Notes on floating point arithmetic

- Any arithmetic operation's result is mapped to one of the numbers in the *finite* set of representable floating point numbers. This is called *rounding*. In total, there are about 4 billion different single precision numbers.
- The IEEE standard requires that the result of addition, subtraction, multiplication, and division is rounded exactly. This means that the result is as if the calculation was done exactly, and is then rounded to the *nearest* representable number.
- Any operation involving NaN will again yield NaN.

Some consequences and pitfalls

- It is possible that the result of $a + b$ is identical to a for $b \neq 0$. (In fact, this will typically happen when $|b| < \epsilon_m \cdot |a|$.)
- The law of associativity is not guaranteed (due to rounding errors). This means that $(a + b) + c$ does not necessarily evaluate to the same number as $a + (b + c)$.
- Even though $x/2.0$ and $0.5 * x$ are always the same, this is not true for $x/10.0$ and $0.1 * x$. This is because 0.1 is not exactly representable for base $\beta = 2$. Also, beware that optimizing compilers may change $x/10.0$ to $0.1 * x$ (since multiplication is much faster than division), but this can then change the result of the calculation and the semantics of the program.

- When numbers nearly cancel, a loss of precision (reduced number of significant digits) results. For example, consider $x = 10^8$, $y = 10^5$ and $z = -1 - 10^5$. Then:

```
x * y + x * z =  -1.0066 x 10^8

x * (y + z)   =  -1.0 x 10^8
```

  Here the second result is correct, but the first one is 0.6% off.

Closer look at loss of precision

Let us more formally define the number of significant digits. Assume $x^\star$ is our floating point approximation to a number x, and likewise $y^\star$ for y. We would then call

$$\frac{x^\star - x}{x} \tag{1.10}$$

the relative error of the representation. One says that $x^\star$ approximates x to r significant digits if

$$|x^\star - x| < \frac{1}{2} 10^{s-r+1}, \tag{1.11}$$

where s is the largest integer such that $10^s < |x|$ (i.e. the absolute error is at most 0.5 in the r-th significant digit of x). For example, $x^\star = 22/7 = 3.1428\ldots$ approximates π to three significant digits.

Cancellation of large numbers typically leads to a loss of significant digits and an explosion of the relative error. Example:

$$x^\star = 0.76545421 \tag{1.12}$$

$$y^\star = 0.76544200 \tag{1.13}$$

Both numbers are stored with 7 significant digits. But the difference

$$z^\star = x^\star - y^\star = 0.12210000 \times 10^{-4} \tag{1.14}$$

has only 3 significant digits in its approximation of z. Consequently, the relative error has become larger by a factor of 1000 or so!

The lesson is, if cancellation can be foreseen *avoid it if possible.* For example, the evaluation of

$$f(x) = 1 - \cos(x) \tag{1.15}$$

involves a cancellation for x near zero. Calculating instead

$$f(x) = \frac{\sin^2(x)}{1 + \cos(x)} \tag{1.16}$$

will yield a more accurate (but potentially more costly) result in this case.

1.2.2 Double and higher precision

Standard compliant double precision numbers use $p = 52(+1)$ bits for the mantissa, and 11 bits for the exponent which has the allowed range $\epsilon_{\max} = +1023$ and $\epsilon_{\min} = -1022$. (Btw: $|\epsilon_{\min}| < |\epsilon_{\max}|$ is adopted such that the inverse of the smallest representable doesn't overflow.) The storage footprint of a double is hence 64 bits, or 8 bytes. The largest and smallest positive representable numbers are

$$f_{\max} \simeq 1.8 \times 10^{308}, \tag{1.17}$$

$$f_{\min} \simeq 2.2 \times 10^{-308}. \tag{1.18}$$

And the machine precision is

$$\epsilon_m \simeq 2^{-52} = 2.2 \times 10^{-16}. \tag{1.19}$$

- Double precision has all the same principle pitfalls as single precision – but "much less".
- Recommendation: Use always double precision unless memory constraints force the use of single precision for some reason.
- But: Don't believe the use of double precision protects against inaccurate floating point results in all situations!

Quad-double precision

- This is a 128-bit floating point format in the IEEE standard based on a 16 byte presentation.
- It offers twice as many significant digits as plain double precision ($\sim$ 34 dezimal places) and an extended exponent range.
- Unfortunately, this is typically not supported in hardware by current processors, but it can be emulated in software by some compilers, in which case every floating point operation is between 2-10 times slower than a corresponding double precision operation.
- Note that `long double` will be accepted by C-compilers, but in many cases this will either be simply identical to 64-bit double precision values, or it will yield a 96-bit format. Sometimes, the software emulation of 128-bit accuracy can however be enabled through compiler flags.

Arbitrary precision

There are good floating point libraries that can be used for calculations in (nearly) arbitrary, user-defined precision. One very capable package is GMP, the 'GNU big number library' (`https://gmplib.org`).

2 Integration of ordinary differential equations

We discuss in the following some basic methods for the integration of *ordinary differential equations* (ODEs). These are relations between an unknown scalar or vector-valued function $\mathbf{y}(t)$ and its derivatives with respect to the dependent variable (t in this case – the following discussion associates this with 'time', but this could of course be also any other variable). Such equations hence formally take the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(\mathbf{y}, t), \tag{2.1}$$

and we seek the solution $\mathbf{y}(t)$, subject to boundary conditions.

Many simple dynamical problems can be written in this form, including ones that involve second or higher derivatives. This is done through a procedure called **reduction to 1st order**. One does this by adding the higher derivatives, or combinations of them, as further rows to the vector $\mathbf{y}$.

For example, consider a simple pendulum with the equation of motion

$$\ddot{q} = -\frac{g}{l}\sin(q), \tag{2.2}$$

where q is the angle with respect to the vertical. Now define $p \equiv q$, yielding a state vector

$$\mathbf{y} \equiv \begin{pmatrix} q \\ p \end{pmatrix}, \tag{2.3}$$

and a first order ODE of the form:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(\mathbf{y}) = \begin{pmatrix} p \\ -\frac{g}{l}\sin(q) \end{pmatrix}. \tag{2.4}$$

A numerical approximation to the solution of an ODE is a set of values $\{y_0, y_1, y_2, \ldots\}$ at discrete times $\{t_0, t_1, t_2, \ldots\}$, obtained for certain boundary conditions. The most common boundary condition for ODEs is the **initial value problem** (IVP), where the state of $\mathbf{y}$ is known at the beginning of the integration interval. It is however also possible to have mixed boundary conditions where $\mathbf{y}$ is partially known at both ends of the integration interval.

There are many different methods for obtaining a discrete solution of an ODE system (e.g. Press et al., 1992). We shall here discuss some of the most basic ones, restricting ourselves to the IVP, for simplicity.

2.1 Explicit Euler method

This solution method, sometimes also called "forward Euler", uses the iteration

$$y_{n+1} = y_n + f(y_n)\Delta t, \tag{2.5}$$

where y can also be a vector. Δt is the integration step.

- This approach is the simplest of all.
- The method is called *explicit* because y_{n+1} is computed with a right-hand-side that only depends on things that are already known.
- The stability of the method can be a sensitive function of the stepsize, and will in general only be reached for a sufficiently small step size.
- It is recommended to refrain from using this scheme in practice, since there are other methods that offer higher accuracy at the same or lower computational cost. The reason is that the Euler method is only *first order accurate*. To see this, note that the truncation error in a single step is of order $\mathcal{O}_s(\Delta t^2)$, which follows simply from a Taylor expansion. To simulate over time T, we need however $N_s = T/\Delta t$ steps, producing a total error that scales as $N_s\mathcal{O}_s(\Delta t^2) = \mathcal{O}_T(\Delta t)$.

We remark in passing that for a method to reach a global error that scales as $\mathcal{O}_T(\Delta t^n)$ (which is then called an "n^{th}-order accurate scheme), a local truncation error one order higher is required, i.e. $\mathcal{O}_s(\Delta t^{n+1})$.

2.2 Implicit Euler method

In a so-called "backwards Euler" scheme, one uses

$$y_{n+1} = y_n + f(y_{n+1})\Delta t, \tag{2.6}$$

which seemingly represents only a tiny change compared to the explicit scheme.

- This approach has excellent stability properties, and for some problems, is in fact essentially always stable even for extremely large timestep. Note however that the accuracy will usually become very bad when using such large steps.
- This stability property makes implicit Euler sometimes useful for *stiff equations* where the derivatives (suddenly) can become very large.
- The implicit equation for y_{n+1} that needs to be solved here corresponds in many practical applications to a non-linear equation that can be complicated to solve for y_{n+1}. Often, the root of the equation has to be found numerically, for example through an iterative technique.

Stability of the Euler method in an example

Let's look at a simple problem to investigate the stability of forward and backwards Euler. Suppose we have the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\alpha\, y, \tag{2.7}$$

with $\alpha > 0$ and $y(0) = y_0$. Here we of course know the analytic solution, given by $y(t) = y_0 \exp(-\alpha\, t)$.

What does *explicit* Euler give for this problem? We can work this out as

$$y_{n+1} = y_n + \dot{y}_n \Delta t = y_n - \alpha\, y_n \Delta t = y_n(1 - \alpha\, \Delta t). \tag{2.8}$$

Hence every step gives us a factor $(1 - \alpha\, \Delta t)$, and after n steps we have

$$y_n = (1 - \alpha\, \Delta t)^n\, y_0. \tag{2.9}$$

We see that for $0 < 1 - \alpha\, \Delta t < 1$ (i.e. equivalently for $\Delta t < 1/\alpha$) the y_n monotonically decline. This is acceptable. For $-1 < 1 - \alpha\, \Delta t < 0$ (i.e. equivalently for $1/\alpha < \Delta t < 2/\alpha$), the series oscillates but still declines overall, which is already somewhat problematic. But for $1 - \alpha\, \Delta t < -1$ (i.e. for $\Delta t > 2/\alpha$), the solution blows up and oscillates, which is as bad as it gets. So here the integration clearly becomes unstable for a too large timestep.

Now let's look at the *implicit* Euler instead. Here we have

$$y_{n+1} = y_n + \dot{y}_{n+1} \Delta t = y_n - \alpha\, y_{n+1} \Delta t, \tag{2.10}$$

yielding

$$y_{n+1} = \frac{y_n}{1 + \alpha\, \Delta t}, \tag{2.11}$$

which declines and is unconditionally stable for arbitrarily large timestep.

2.3 Implicit midpoint rule

If we use

$$y_{n+1} = y_n + f\left(\frac{y_n + y_{n+1}}{2}\right) \Delta t \tag{2.12}$$

we obtain the implicit midpoint rule. This is *second order accurate*, but still implicit, so difficult to use in practice. Interestingly, it is also time-symmetric, i.e. one can formally integrate backwards and recover exactly the same steps (modulo floating point round-off errors) as in a forward integration.

2.4 Runge-Kutta methods

The Runge-Kutta schemes form a whole class of versatile integration methods (e.g. Atkinson, 1978; Stoer & Bulirsch, 2002). Let's derive one of the simplest Runge-Kutta schemes.

1. We start from the exact solution,
$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f\left(y(t)\right) \mathrm{d}t. \tag{2.13}$$
2. Next, we approximate the integral with the (implicit) trapezoidal rule:
$$y_{n+1} = y_n + \frac{f(y_n) + f(y_{n+1})}{2} \Delta t. \tag{2.14}$$
3. Runge proposed in 1895 to predict the unknown y_{n+1} on the right hand side by an Euler step, yielding a *2nd order accurate Runge-Kutta scheme*, sometimes also called predictor-corrector scheme:
$$\begin{aligned} k_1 &= f(y_n, t_n), & (2.15)\\ k_2 &= f(y_n + k_1 \Delta t, t_{n+1}), & (2.16)\\ y_{n+1} &= y_n + \frac{k_1 + k_2}{2} \Delta t. & (2.17) \end{aligned}$$
Here the step done with the derivate of equation (2.15) is called the 'predictor' and the one done with equation (2.16) is the corrector step.

Higher order Runge-Kutta schemes

A variety of further Runge-Kutta schemes of different order can be defined. Perhaps the most commonly used is the classical 4^{th}-order Runge-Kutta scheme:
$$\begin{aligned} k_1 &= f(y_n, t_n) & (2.18)\\ k_2 &= f\left(y_n + k_1 \frac{\Delta t}{2}, t_n + \frac{\Delta t}{2}\right) & (2.19)\\ k_3 &= f\left(y_n + k_2 \frac{\Delta t}{2}, t_n + \frac{\Delta t}{2}\right) & (2.20)\\ k_4 &= f\left(y_n + k_3 \Delta t, t_n + \Delta t\right). & (2.21) \end{aligned}$$
These four function evaluations per step are then combined in a weighted fashion to carry out the actual update step:
$$y_{n+1} = y_n + \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right) \Delta t + \mathcal{O}(\Delta t^5). \tag{2.22}$$
We note that the use of higher order schemes also entails more function evaluations per step, i.e. the individual steps become more complicated and expensive. Because of this, higher order schemes are not always better; they usually are up to some point, but sometimes even a simple second-order accurate scheme can be the best choice for certain problems.

2.5 Adaptive step sizes

One issue left open in our discussion thus far is the choice of the optimum integration step. How can we get an optimum compromise between accuracy, efficiency and stability?

To control the accuracy, we need a way to estimate the local integration error, and a scheme for adjusting the step size such that a prescribed desired maximum error is guaranteed. One useful idea for controlling this lies in solving a step of size Δt twice, once with Δt (case A), and once by doing two steps of size $\frac{\Delta t}{2}$ (case B). The difference between the results then yields an estimate of the truncation error.

For example, suppose we use a p-th order scheme (e.g. Runge-Kutta forth order), then we have for the two end states:

$$y_A - y(t_0 + \Delta t) = \alpha \cdot (\Delta t)^{p+1} + \mathcal{O}(\Delta t^{p+2}), \tag{2.23}$$

$$y_B - y(t_0 + \Delta t) = 2 \times \left[\alpha \cdot (\Delta t/2)^{p+1}\right] + \mathcal{O}(\Delta t^{p+2}). \tag{2.24}$$

This yields an error estimate of

$$\epsilon = |y_A - y_B| = \alpha \cdot (\Delta t)^{p+1} \cdot (1 - 2^{-p}). \tag{2.25}$$

Assuming we are given a local error bound ϵ_0 for each step, we can now construct an algorithm where this is respected in every integration step:

1. Take a step of size Δt (yielding y_A), and two of stepsize $\frac{\Delta t}{2}$ (yielding y_B). We can then get an error estimate as $y = |y_A - y_B|$.

2. If $\epsilon > \epsilon_0$, discard the step, halve the timestep, $\Delta t' = \frac{\Delta t}{2}$, and try again.

3. If $\epsilon \ll \epsilon_0$, keep y_B and double the step for the next step, $\Delta t' = 2\Delta t$.

4. Else if $\epsilon < \epsilon_0$, keep y_B and retain Δt for the next step.

When does it make sense to double the step in point 3? This should only be done if the estimated new error is still smaller than the error bound, i.e. for

$$\epsilon \cdot 2^{p+1} < \epsilon_0. \tag{2.26}$$

But there is another complication. So far, our error estimate has been entirely local, disregarding the fact that we might need to do many integration steps. If we want to guarantee a certain error globally, even when all the errors add up with the same sign, we have to lower ϵ_0 with smaller step size Δt, because we then need to do more steps!

We can for example account for this by setting

$$\epsilon_0 = \frac{\Delta t}{T} \epsilon_0^{\text{global}}, \tag{2.27}$$

where T is the total integrated time, and $\epsilon_0^{\text{global}}$ is our prescribed error bound for the whole integration. This means we loose effectively one power of Δt again, because of

$$\epsilon \simeq \alpha \cdot (1 - 2^{-p}) \cdot \Delta t^{p+1} < \frac{\Delta t}{T} \epsilon_0^{\text{global}}. \tag{2.28}$$

Instead of step doubling one can also use a continuous step adjustment. The idea is to scale the timestep such that the estimated error matches the desired error ϵ_0:

$$(\Delta t)^{\text{desired}} = (\Delta t) \cdot \left(\frac{\epsilon_0}{\epsilon}\right)^{1/(p+1)}, \tag{2.29}$$

where ϵ is the error if a step of size Δt is taken, and ϵ_0 is the error level that one wants. $(\Delta t)^{\text{desired}}$ is then an estimate of the timestep that would deliver this error level.

Use of this in practice requires that we obtain for each step also an estimate of the error ϵ that is made. Then we can use this expression to determine the timestep for the next step. Such a 'built-in' error estimate can be delivered by so-called embedded Runge-Kutta schemes, which compute it more cheaply than done with the step-doubling technique.

A typical algorithm for continuous adaptive step size control would then for example work as follows:

1. Advance the system for a step Δt and estimate the error ϵ of the step at the same time (we here assume a p-th order scheme with $\mathcal{O}(\epsilon) = \Delta t^{p+1}$).

2. Calculate the new step size as

$$(\Delta t)^{\text{new}} = \beta \cdot (\Delta t) \cdot \left(\frac{\epsilon_0}{\epsilon}\right)^{1/(p+1)} \tag{2.30}$$

   where $\beta \sim 0.9$ is an empirical "safety factor".

3. If $\epsilon < \epsilon_0$ accept the step taken in (1), otherwise discard it and try again with new step size.

The well-known Runge-Kutta-Fehlberg integration method is of this type.

2.6 The leapfrog

Suppose we have a second order differential equation of the type

$$\ddot{x} = f(x). \tag{2.31}$$

This could of course be brought into standard form, $\dot{\mathbf{y}} = \tilde{\mathbf{f}}(\mathbf{y})$, by defining something like $\mathbf{y} = (x, \dot{x})$ and $\tilde{\mathbf{f}} = (\dot{x}, f(x))$, followed by applying a Runge-Kutta scheme as introduced above.

However, there is also another approach in this case, which turns out to be particularly simple and interesting. Let's define $v \equiv \dot{x}$. Then the so-called Leapfrog integration scheme is the mapping $(x_n, v_n) \to (x_{n+1}, v_{n+1})$ defined as:

$$v_{n+\frac{1}{2}} = v_n + f(x_n)\frac{\Delta t}{2}, \tag{2.32}$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}}\,\Delta t, \tag{2.33}$$

$$v_{n+1} = v_{n+\frac{1}{2}} + f(x_{n+1})\frac{\Delta t}{2}. \tag{2.34}$$

- This scheme is 2nd-order accurate (proof through Taylor expansion).
- It requires only 1 evaluation of the right hand side per step (note that $f(x_{n+1})$ can be reused in the next step.
- The scheme can be written in a number of alternative ways, for example by combining the two half-steps of two subsequent steps. One then gets

  $$x_{n+1} = x_n + v_{n+\frac{1}{2}}\,\Delta t, \tag{2.35}$$

  $$v_{n+\frac{3}{2}} = v_{n+\frac{1}{2}} + f(x_{n+1})\,\Delta t. \tag{2.36}$$

  One here sees the time-centered nature of the formulation very clearly, and the interleaved advances of position and velocity give it the name leapfrog.

The performance of the leapfrog on certain problems is found to be surprisingly good, better than that of other schemes such as Runge-Kutta which have formally the same or even a better error order. This is illustrated in Figure 2.1 for the Kepler problem, i.e. the integration of the motion of a small point mass in the gravitational field of a large mass.

We see that the long-term evolution is entirely different. Unlike the RK schemes, the leapfrog does not build up a large energy error. So why is the leapfrog behaving here so much better than other 2nd order or even 4th order schemes?

2.7 Symplectic integrators

The reason for these beneficial properties lies in the fact that the leapfrog is a so-called symplectic method. These are structure-preserving integration methods (e.g. Saha & Tremaine, 1992; Hairer et al., 2002) that observe important special properties of Hamiltonian systems: Such systems have first conserved integrals (such as the energy), they also exhibit phase-space conservation as described by the Liouville theorem, and more generally, they preserve Poincare's integral invariants.

Symplectic transformations

- A linear map $F : \mathbb{R}^{2d} \to \mathbb{R}^{2d}$ is called symplectic if $\omega(F\xi, F\eta) = \omega(\xi, \eta)$ for all vectors $\xi, \eta \in \mathbb{R}^{2d}$, where w gives the area of the parallelgram spanned by the the two vectors.

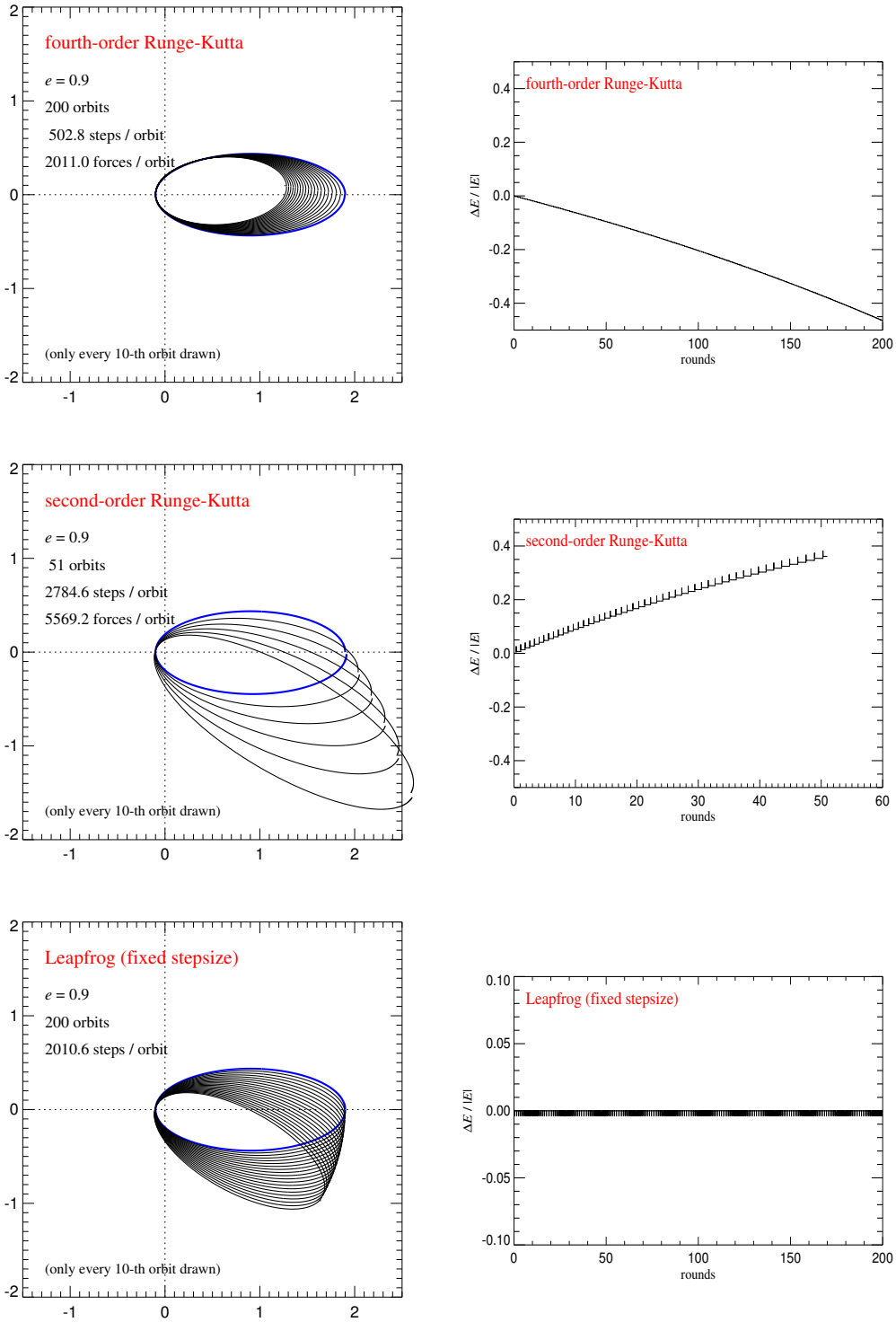


Figure 2.1: Kepler problem integrated with different integration schemes (Springel, 2005). The panels on top are for a 4th-order Runge Kutta scheme, the middle for a 2nd order Runge-Kutta, and the bottom for a 2nd-order leapfrog. The leapfrog does not show a secular drift of the total energy, and is hence much more suitable for long-term integration of this Hamiltonian system.

- A differentiable map $g : U \to \mathbb{R}^{2d}$ is called symplectic if its Jacobian matrix is everywhere symplectic, i.e. $\omega(g'\xi, g'\eta) = \omega(\xi, \eta)$.
- **Poincare's theorem** states that the time evolution generated by a Hamiltonian in phase-space is a symplectic transformation.

The above suggests that there is a close connection between exact solutions of Hamiltonians and symplectic transformations. Also, two consecutive symplectic transformations are again symplectic.

Separable Hamiltonians

Dynamical problems that are described by Hamiltonians of the form

$$H(p, q) = \frac{p^2}{2m} + U(q) \tag{2.37}$$

are quite common. These systems have separable Hamiltonians that can be written as

$$H(p, q) = H_{\text{kin}}(p) + H_{\text{pot}}(q). \tag{2.38}$$

Now we will allude to the general idea of *operator splitting* (Strang, 1968). Let's try to solve the two parts of the Hamiltonian individually:

1. For the part $H = H_{\text{kin}} = \frac{p^2}{2m}$, the equations of motion are

$$\dot{q} = \frac{\partial H}{\partial p} = \frac{p}{m}, \tag{2.39}$$

$$\dot{p} = -\frac{\partial H}{\partial q} = 0. \tag{2.40}$$

   These equations are straightforwardly solved and give

$$q_{n+1} = q_n + p_n \Delta t, \tag{2.41}$$
$$p_{n+1} = p_n. \tag{2.42}$$

   Note that this solution is exact for the given Hamiltonian, for arbitrarily long time intervals Δt. Given that it is a solution of a Hamiltonian, the solution constitutes a symplectic mapping.

2. The potential part, $H = H_{\text{pot}} = U(q)$, leads to the equations

$$\dot{q} = \frac{\partial H}{\partial p} = 0 \tag{2.43}$$

$$\dot{p} = -\frac{\partial H}{\partial q} = -\frac{\partial U}{\partial q}. \tag{2.44}$$

This is solved by

$$q_{n+1} = q_n, \tag{2.45}$$

$$p_{n+1} = p_n - \frac{\partial U}{\partial q}\Delta t. \tag{2.46}$$

Again, this is an exact solution independent of the size of Δt, and therefore a symplectic transformation.

Let's now introduce an operator $\varphi_{\Delta t}(H)$ that describes the mapping of phase-space under a Hamiltonian H that is evolved over a time interval Δt, then it is easy to see that the leapfrog is given by

$$\varphi_{\Delta t}(H) = \varphi_{\frac{\Delta t}{2}}(H_{\mathrm{pot}}) \circ \varphi_{\Delta t}(H_{\mathrm{kin}}) \circ \varphi_{\frac{\Delta t}{2}}(H_{\mathrm{pot}}) \tag{2.47}$$

for a separable Hamiltonian $H = H_{\mathrm{kin}} + H_{\mathrm{pot}}$.

- Since each individual step of the leapfrog is symplectic, the concatenation is also symplectic.
- In fact, the leapfrog generates the exact solution to a modified Hamiltonian H_{leap}, where $H_{\mathrm{leap}} = H + H_{\mathrm{err}}$. The difference lies in the 'error Hamiltonian' H_{err}, which is given by

  $$H_{\mathrm{err}} \propto \frac{\Delta t^2}{12}\left\{\{H_{\mathrm{kin}}, H_{\mathrm{pot}}\}, H_{\mathrm{kin}} + \frac{1}{2}H_{\mathrm{pot}}\right\} + \mathcal{O}(\Delta t^3), \tag{2.48}$$

  where the curly brackets are Poisson brackets (Goldstein, 1950). This can be demonstrated by expanding

  $$\mathrm{e}^{(H+H_{\mathrm{err}})\Delta t} = \mathrm{e}^{H_{\mathrm{pot}}\frac{\Delta t}{2}}\ \mathrm{e}^{H_{\mathrm{kin}}\Delta t}\ \mathrm{e}^{H_{\mathrm{pot}}\frac{\Delta t}{2}} \tag{2.49}$$

  with the help of the Baker-Campbell-Hausdorff formula (Campbell, 1897; Saha & Tremaine, 1992).
- This property explains the superior long-term stability of the integration of conservative systems with the leapfrog. Because it respects phase-space conservation, secular trends are largely absent, and the long-term energy error stays bounded and reasonably small.

3 Collisionless particle systems

According to the ΛCDM paradigm, the matter density of our Universe is dominated by *dark matter*, which is thought to be composed of a yet unidentified, non-baryonic elementary particle (e.g. Bertone et al., 2005). A full description of the dark mass in a galaxy would hence be based on following the trajectories of each dark matter particle – resulting in a gigantic N-body model. This is clearly impossible due to the large number of particles involved. Similarly, describing all the trajectories of electrons in a plasma is infeasible. In this chapter we discuss why we can nevertheless describe these systems as discrete N-body systems, but composed of far fewer particles than there are in reality.

3.1 N-particle ensembles

The state of an N-particle ensemble at time t can be specified by the *exact* particle distribution function (Hockney & Eastwood, 1988), in the form

$$F(\mathbf{x}, \mathbf{v}, t) = \sum_{i=1}^{N} \delta(\mathbf{x} - \mathbf{x}_i(t)) \cdot \delta(\mathbf{v} - \mathbf{v}_i(t)). \tag{3.1}$$

This gives effectively the number of particles at phase-space point $(\mathbf{x}, \mathbf{v})$ at time t. Let now

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N, \mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_N)\, \mathrm{d}\mathbf{x}_1\, \mathrm{d}\mathbf{x}_2 \cdots \mathrm{d}\mathbf{x}_N\, \mathrm{d}\mathbf{v}_1\, \mathrm{d}\mathbf{v}_2 \cdots \mathrm{d}\mathbf{v}_N \tag{3.2}$$

be the probability that the system is in the given state at time t. Then a reduced statistical description is obtained by *ensemble averaging*:

$$f_1(\mathbf{x}, \mathbf{v}, t) = \langle F(\mathbf{x}, \mathbf{v}, t) \rangle = \int F \cdot p \cdot \mathrm{d}\mathbf{x}_1\, \mathrm{d}\mathbf{x}_2 \cdots \mathrm{d}\mathbf{x}_N\, \mathrm{d}\mathbf{v}_1\, \mathrm{d}\mathbf{v}_2 \cdots \mathrm{d}\mathbf{v}_N. \tag{3.3}$$

We can integrate out one of the Dirac delta-functions in F to obtain

$$f_1(\mathbf{x}, \mathbf{v}, t) = N \int p(\mathbf{x}, \mathbf{x}_2, \ldots, \mathbf{x}_N, \mathbf{v}, \mathbf{v}_2, \ldots \mathbf{v}_N)\, \mathrm{d}\mathbf{x}_2 \cdots \mathrm{d}\mathbf{x}_N\, \mathrm{d}\mathbf{v}_2 \cdots \mathrm{d}\mathbf{v}_N. \tag{3.4}$$

Note that we can permute the arguments in p where $\mathbf{x}$ and $\mathbf{v}$ appear. $f_1(\mathbf{x}, \mathbf{v}, t)\, \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{v}$ now gives the *mean number* of particles in a phase-space volume $\mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{v}$ around $(\mathbf{x}, \mathbf{v})$.

Similarly, the ensemble-averaged two-particle distribution ("the mean product of the numbers of particles at $(\mathbf{x}, \mathbf{v})$ and $(\mathbf{x}', \mathbf{v}')$") is given by

$$f_2(\mathbf{x}, \mathbf{v}, \mathbf{x}', \mathbf{v}', t) = \langle F(\mathbf{x}, \mathbf{v}, t) F(\mathbf{x}', \mathbf{v}', t) \rangle \tag{3.5}$$
$$= N(N-1) \int p(\mathbf{x}, \mathbf{x}', \mathbf{x}_3, \ldots, \mathbf{x}_N, \mathbf{v}, \mathbf{v}', \mathbf{v}_3, \ldots \mathbf{v}_N) \, \mathrm{d}\mathbf{x}_3 \cdots \mathrm{d}\mathbf{x}_N \, \mathrm{d}\mathbf{v}_3 \cdots \mathrm{d}\mathbf{v}_N.$$

Likewise one may define $f_3, f_4, \ldots$ and so on. This yields the so-called BBGKY (Bogoliubov-Born-Green-Kirkwood-Yvon) chain (e.g. Kirkwood, 1946), see also Hockney & Eastwood (1988) for a detailed discussion.

3.2 Uncorrelated (collisionless) systems

The simplest closure for the BBGKY hierarchy is to assume that particles are *uncorrelated*, i.e. that we have

$$f_2(\mathbf{x}, \mathbf{v}, \mathbf{x}', \mathbf{v}', t) = f_1(\mathbf{x}, \mathbf{v}, t) \, f_1(\mathbf{x}', \mathbf{v}', t). \tag{3.6}$$

Physically, this means that a particle at $(\mathbf{x}, \mathbf{v})$ is completely unaffected by one at $(\mathbf{x}', \mathbf{v}')$. Systems in which this is approximately the case include

- electrons in a plasma
- stars in a galaxy
- dark matter particles in the universe

We will later consider in more detail under which conditions a system is collisionless.

Let's now go back to the probability density $p(\mathbf{w})$ which depends on the N-particle phase-space state $\mathbf{w} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N, \mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_N)$. The conservation of probability in phase-space means that it fulfills a continuity equation

$$\frac{\partial p}{\partial t} + \nabla_{\mathbf{w}} \cdot (p \, \dot{\mathbf{w}}) = 0. \tag{3.7}$$

We can cast this into

$$\frac{\partial p}{\partial t} + \sum_i \left(p \frac{\partial \dot{\mathbf{x}}_i}{\partial \mathbf{x}_i} + \frac{\partial p}{\partial \mathbf{x}_i} \dot{\mathbf{x}}_i + p \frac{\partial \dot{\mathbf{v}}_i}{\partial \mathbf{v}_i} + \frac{\partial p}{\partial \mathbf{v}_i} \dot{\mathbf{v}}_i \right) = 0. \tag{3.8}$$

Now we recall Hamiltonian dynamics with the equations of motion $\dot{\mathbf{x}} = \frac{\partial H}{\partial \mathbf{p}}$ and $\dot{\mathbf{p}} = -\frac{\partial H}{\partial \mathbf{x}}$ (Goldstein, 1950). We can differentiate them to get $\frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{x}} = \frac{\partial^2 H}{\partial \mathbf{x} \partial \mathbf{p}}$, and $\frac{\partial \dot{\mathbf{p}}}{\partial \mathbf{p}} = -\frac{\partial^2 H}{\partial \mathbf{x} \partial \mathbf{p}}$. Hence it follows $\frac{\partial \dot{\mathbf{x}}}{\partial \mathbf{x}} = -\frac{\partial \dot{\mathbf{v}}}{\partial \mathbf{v}}$. Using this we get

$$\frac{\partial p}{\partial t} + \sum_i \left(\mathbf{v}_i \frac{\partial p}{\partial \mathbf{x}_i} + \mathbf{a}_i \frac{\partial p}{\partial \mathbf{v}_i} \right) = 0, \tag{3.9}$$

where $\mathbf{a}_i = \dot{\mathbf{v}}_i = \mathbf{F}_i/m_i$ is the particle acceleration. This is *Liouville's theorem*.

Now, in the collisionless/uncorrelated limit, this directly carries over to the one-point distribution function $f = f_1$, yielding the *Vlasov equation*, also known as collisionless Boltzmann equation:

$$\frac{\partial f}{\partial t} + \mathbf{v}\frac{\partial f}{\partial \mathbf{x}} + \mathbf{a}\frac{\partial f}{\partial \mathbf{v}} = 0. \tag{3.10}$$

The close relation to Liouville's equation means that also here the phase space-density stays constant along characteristics (i.e. along orbits of individual particles) of the system.

What about the accelerations?

In the limit of a collisionless system, the accelerations cannot be due to a single other particle. However, collective effects, for example from the gravitational or electric field produced by the whole system are still allowed.

For example, the source field of self-gravity can be described as

$$\rho(\mathbf{x}, t) = m \int f(\mathbf{x}, \mathbf{v}, t)\, \mathrm{d}\mathbf{v}. \tag{3.11}$$

This then produces a gravitational field through Poisson's equation,

$$\nabla^2 \Phi = 4\pi G \rho, \tag{3.12}$$

which yields the accelerations as

$$\mathbf{a} = -\frac{\partial \Phi}{\partial \mathbf{x}}. \tag{3.13}$$

One can also combine these equations to yield the Poisson-Vlasov system, given by

$$\frac{\partial f}{\partial t} + \mathbf{v}\frac{\partial f}{\partial \mathbf{x}} - \frac{\partial \Phi}{\partial \mathbf{x}}\frac{\partial f}{\partial \mathbf{v}} = 0, \tag{3.14}$$

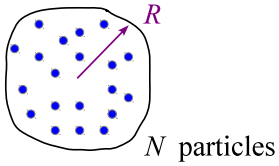
$$\nabla^2 \Phi = 4\pi G m \int f(\mathbf{x}, \mathbf{v}, t)\, \mathrm{d}\mathbf{v}. \tag{3.15}$$

This holds in an analogous way also for a plasma where the mass density is replaced by a charge density.

It is interesting to note that in this description the particles have basically completely vanished and have been replaced with a continuum fluid description. Later, for the purpose of solving the equations, we will have to reintroduce particles as a means of discretizing the equations (but these are then not the real physical particles any more, rather they are fiducial macro particles that sample the phase-space in a Monte-Carlo fashion).

3.3 When is a system collisionless?

Consider a system of size R containing N particles.



The time for one crossing of a particle through the system is of order

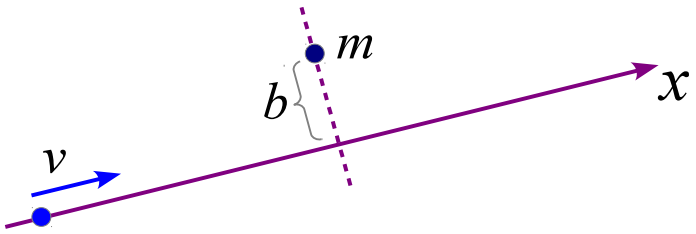
$$t_{\rm cross} = \frac{R}{v}, \tag{3.16}$$

where v is the typical particle velocity (Binney & Tremaine, 1987, 2008). For a self-gravitating system of that size we expect

$$v^2 \simeq \frac{GNm}{R} = \frac{GM}{R}, \tag{3.17}$$

where $M = N\,m$ is the total mass.

We now want to estimate the rate at which a particle experiences weak deflections by other particles, which is the process that violates perfect collisionless behavior and which induces relaxation. We calculate the deflection in the impulse approximation where the particle's orbit is taken as a straight path.



To get the deflection, we compute the transverse momentum acquired by the particle as it flies by the perturber (assumed to be stationary for simplicity):

$$\Delta p = m\Delta v = \int F_\perp {\rm d}t = \int \frac{Gm^2}{x^2+b^2}\frac{b}{\sqrt{x^2+b^2}}\frac{{\rm d}x}{v} = \frac{2Gm^2}{bv}. \tag{3.18}$$

How many encounters do we expect in one crossing? For impact parameters between $[b, b+{\rm d}b]$ we have

$${\rm d}n = N\frac{2\pi b\,{\rm d}b}{\pi R^2} \tag{3.19}$$

targets. The velocity perturbations from each encounter have random orientations, so they add up in quadrature. Per crossing we hence have for the quadratic velocity perturbation:

$$(\Delta v)^2 = \int \left(\frac{2Gm}{bv}\right)^2 {\rm d}n = 8N\left(\frac{Gm}{Rv}\right)^2 \ln\Lambda, \tag{3.20}$$

where

$$\ln \Lambda = \ln \frac{b_{\max}}{b_{\min}} \tag{3.21}$$

is the so-called Coulomb logarithm. We can now define the relaxation time as

$$t_{\rm relax} \equiv \frac{v^2}{(\Delta v)^2 / t_{\rm cross}}, \tag{3.22}$$

i.e. after this time the individual perturbations have reached $\sim 100\%$ of the typical squared velocity, and one certainly not neglect the interactions any more. With our result for $(\Delta v)^2$, and using equation (3.17) this now becomes

$$t_{\rm relax} = \frac{N}{8 \ln \Lambda}\, t_{\rm cross}. \tag{3.23}$$

But we still have to clarify what we can sensibly use for $b_{\min}$ and $b_{\max}$ in the Coulomb logarithm. For $b_{\max}$, we can set the size of the system, i.e. $b_{\max} \simeq R$. For $b_{\min}$, we can use as a lower limit the b where very strong deflections ensue, which is given by

$$\frac{2Gm}{b_{\min} v} \simeq v, \tag{3.24}$$

i.e where the transverse velocity perturbation becomes as large as the velocity itself. This then yields $b_{\min} = 2R/N$. We hence get for the Coulomb logarithm $\ln \Lambda \simeq \ln(N/2)$. But a factor of 2 in a logarithm might as well be neglected in this coarse estimate, so that we expect $\ln \Lambda \sim \ln N$. We hence arrive at the final result (Chandrasekhar, 1943):

$$t_{\rm relax} = \frac{N}{8 \ln N}\, t_{\rm cross}. \tag{3.25}$$

A system can be viewed as collisionless if $t_{\rm relax} \gg t_{\rm age}$, where $t_{\rm age}$ is the time of interest. We note that $t_{\rm cross}$ depends only on the size and mass of the system, but *not* on the particle number N or the individual masses of the N-body particles. We therefore clearly see that the primary requirement to obtain a collisionless system is to use a sufficiently large N.

Examples

- globular star clusters have $N \sim 10^5$, $t_{\rm cross} \sim \frac{3\,{\rm pc}}{6\,{\rm km/sec}} \simeq 0.5\,{\rm Myr}$. This implies that such systems are strongly affected by collisions over the age of the Universe, $t_{\rm age} = \frac{1}{H_0} \sim 10\,{\rm Gyr}$.
- stars in a typical galaxy: Here we have $N \sim 10^{11}$ and $t_{\rm cross} \sim \frac{1}{100\,H_0}$. This means that these large stellar systems are collisionless over the age of the Universe to extremely good approximation.
- dark matter in a galaxy: Here we have $N \sim 10^{77}$ if the dark matter is composed of a $\sim 100\,{\rm GeV}$ weakly interacting massive particle (WIMP). In addition, the crossing time is longer than for the stars, $t_{\rm cross} \sim \frac{1}{10\,H_0}$, due to the larger size of the 'halo' relative to the embedded stellar system. Clearly, the dark matter represents the mother of all collisionless systems.

3.4 N-body models of collisionless systems

We now reintroduce particles in order to discretize the collisionless fluid described by the Poisson-Vlasov system. We use however *far fewer* particles than in real physical systems, and we correspondingly give them a higher mass (and/or charge). These are hence fiducial macro-particles. Their equations of motions in the case of gravity are written as:

$$\ddot{\mathbf{x}} = -\nabla_i \Phi(\mathbf{x}_i), \tag{3.26}$$

$$\Phi(\mathbf{x}) = -G \sum_{j=1}^{N} \frac{m_j}{[(\mathbf{x} - \mathbf{x}_j)^2 + \epsilon^2]^{1/2}}. \tag{3.27}$$

A few comments are in order here:

- Provided we can ensure $t_{\rm relax} \gg t_{\rm sim}$ despite the smaller N than in the real physical system, the numerical model keeps behaving as a collisionless system over the simulated time-span $t_{\rm sim}$, and the collective gravitational potential is sufficiently smooth.

- The mass of the macro-particles used to discretize the collision system does not enter in the equations of motion. Provided there are enough particles to describe the gravitational potential well, the orbits of the macro-particles will be just as valid as the orbits of the real physical particles.

- The N-body model gives only one (quite noisy) realization of the one-point function. It does not give the ensemble average directly (this would require multiple simulations).

- The equations of motion contain a **softening length** ϵ. The purpose of the force softening is to avoid large angle scatterings and the numerical expense that would be needed to integrate the orbits with sufficient accuracy in singular potentials. Also, we would like to prevent the possibility of the formation of bound particle pairs – they would obviously be highly correlated and hence strongly violate collisionless behaviour. We don't get bound pairs if

  $$\left\langle v^2 \right\rangle \gg \frac{Gm}{\epsilon}, \tag{3.28}$$

  which can be used as a simple condition on reasonable softening settings. The adoption of a softening length also implies the introduction of a smallest resolved length-scale. The specific softening choice one makes ultimately represents a compromise between spatial resolution, discreteness noise in the orbits and the gravitational potential, computational cost, and the relaxation effects that may negatively influence results.

4 Tree algorithms

Once we have discretized a collisionless fluid in terms of an N-body system, two questions come up:

1. How do we integrate the equations of motion in time?

2. How do we compute the right hand side of the equations of motion, i.e. the gravitational forces?

For the first point, we can simply use one of our ODE integration schemes, preferably a symplectic one since we are dealing with a Hamiltonian system. The second point seems also straightforward at first, as the accelerations (forces) can be readily calculated through *direct summation*:

$$\ddot{\mathbf{x}}_i = -G \sum_{j=1}^{N} \frac{m_j}{[(\mathbf{x}_i - \mathbf{x}_j)^2 + \epsilon^2]^{3/2}} (\mathbf{x}_i - \mathbf{x}_j). \tag{4.1}$$

This calculation is *exact*, but for each of the N equations we have to calculate a sum with N partial forces, yielding a computational cost of order $\mathcal{O}(N^2)$. This quickly becomes prohibitive for large N, and causes a conflict with our urgent need to have a large N!

Perhaps a simple example is in order to show how bad the N^2 scaling really is in practice. Suppose you can do $N = 10^6$ in a month of computer time, which is close to the maximum that one may want to do in practice. A particle number of $N = 10^{10}$ would then already take of order 10 million years.

We hence need faster, approximative force calculation schemes. We shall discuss three different possibilities:

- Hierarchical multipole methods ("tree-algorithms")

- Fourier-transform based methods ("particle-mesh algorithms")

- Iterative solvers for Poisson's equation ("relaxation methods", multigrid-methods)

Various combinations of these approaches may also be used, and sometimes they are also applied together with direct summation on small scales. The latter may also be accelerated with special-purpose hardware (e.g. the GRAPE board), or with graphics processing units (GPUs) that are used as fast number-crunshers.

4.1 Multipole expansion

The central idea is here to use the multipole expansion of a distant group of particles to describe its gravity (Barnes & Hut, 1986), instead of summing up the forces from all individual particles.

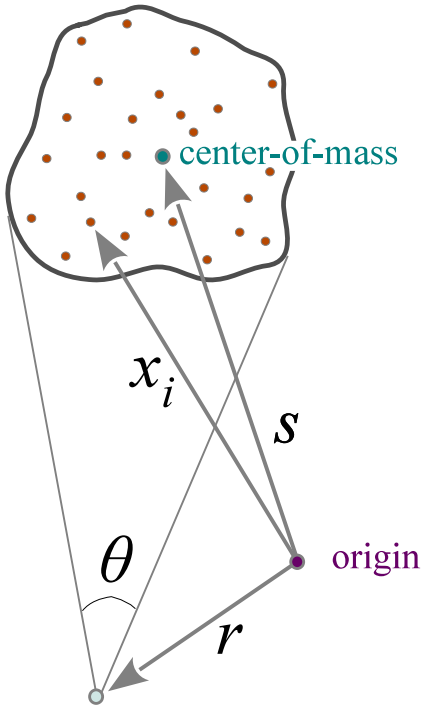


Figure 4.1: Multipole expansion for a group of distant particles. Provided the reference point $\mathbf{r}$ is sufficiently far away, the particles are seen under a small opening angle θ, and the field created by the particle group can be approximated by the monopole term at its center of mass, augmented with higher order multipole corrections if desired.

The potential of the group is given by

$$\Phi(\mathbf{r}) = -G \sum_i \frac{m_i}{|\mathbf{r} - \mathbf{x}_i|}, \tag{4.2}$$

which we can re-write as

$$\Phi(\mathbf{r}) = -G \sum_i \frac{m_i}{|\mathbf{r} - \mathbf{s} + \mathbf{s} - \mathbf{x}_i|}. \tag{4.3}$$

Now we expand the denominator assuming $|\mathbf{x}_i - \mathbf{s}| \ll |\mathbf{r} - \mathbf{s}|$, which will be the case provided the *opening angle* θ under which the group is seen is sufficiently small, see the sketch of Figure 4.1. We can then use the Taylor expansion

$$\frac{1}{|\mathbf{y} + \mathbf{s} - \mathbf{x}_i|} = \frac{1}{|\mathbf{y}|} - \frac{\mathbf{y} \cdot (\mathbf{s} - \mathbf{x}_i)}{|\mathbf{y}|^3} + \frac{1}{2} \frac{\mathbf{y}^T \left[3(\mathbf{s} - \mathbf{x}_i)(\mathbf{s} - \mathbf{x}_i)^T - (\mathbf{s} - \mathbf{x}_i)^2 \right] \mathbf{y}}{|\mathbf{y}|^5} + \dots, \tag{4.4}$$

where we introduced $\mathbf{y} \equiv \mathbf{r} - \mathbf{s}$ as a short-cut. The first term on the right hand side gives rise to the monopole moment, the second to the dipole moment, and the third to the quadrupole moment. If desired, one can continue the expansion to ever higher order terms.

These multipole moments then become properties of the group of particles:

$$\text{monopole:} \quad M = \sum_i m_i \tag{4.5}$$

$$\text{quadrupole:} \quad Q_{ij} = \sum_k m_k \left[3(\mathbf{s} - \mathbf{x}_k)_i (\mathbf{s} - \mathbf{x}_k)_j - \delta_{ij} (\mathbf{s} - \mathbf{x}_k)^2\right] \tag{4.6}$$

The dipole vanishes, because we have done the expansion relative to the center-of-mass, defined as

$$\mathbf{s} = \frac{1}{M} \sum_i m_i \mathbf{x}_i. \tag{4.7}$$

If we restrict ourselves to terms of up to quadrupole order, we hence arrive at the expansion

$$\Phi(\mathbf{r}) = -G \left(\frac{M}{|\mathbf{y}|} + \frac{1}{2} \frac{\mathbf{y}^T \mathbf{Q} \mathbf{y}}{|\mathbf{y}|^5} \right); \quad \mathbf{y} = \mathbf{r} - \mathbf{s}, \tag{4.8}$$

from which also the force can be readily obtained through differentiation. Recall that we expect the expansion to be accurate if

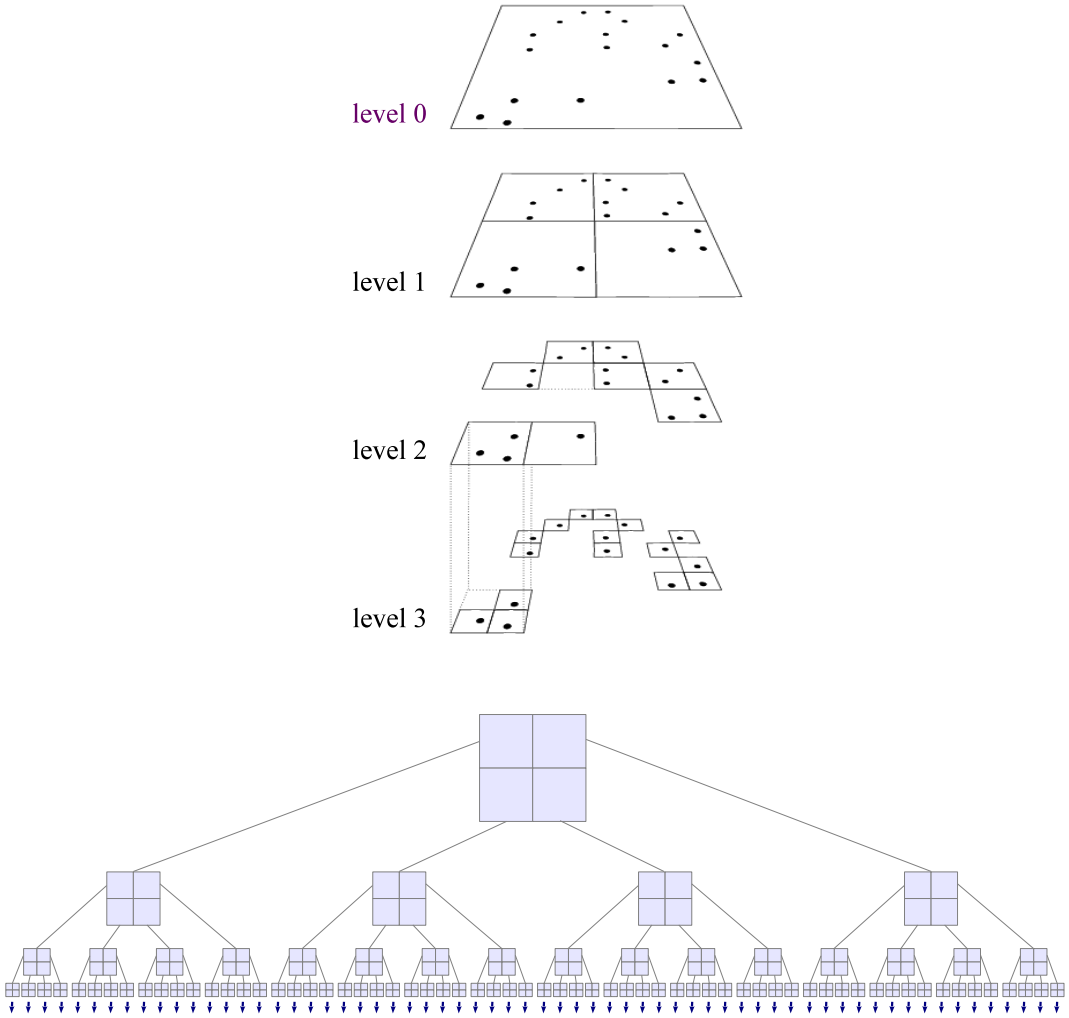
$$\theta \simeq \frac{\langle |\mathbf{x}_i - \mathbf{s}| \rangle}{|\mathbf{y}|} \simeq \frac{l}{y} \ll 1, \tag{4.9}$$

where l is the radius of the group.

4.2 Hierarchical grouping

Tree algorithms are based on a hierarchical grouping of the particles, and for each group, one then pre-computes the multipole moments for later use in approximations of the force due to distant groups. Usually, the hierarchy of groups is organized with the help of a tree-like data structure, hence the name "tree algorithms".

There are different strategies for defining the groups. In the popular Barnes & Hut (1986) oct-tree, one starts out with a cube that contains all the particles. This cube is then subdivided into 8 sub-cubes of half the size in each spatial dimension. One continues with this refinement recursively until each subnode contains only a single particle. Empty nodes (sub-cubes) need not be stored. Here is schematic sketch how this can look like in two dimensions (where one has a 'quad-tree'):



The sketch at the top shows the topological organization of the tree.

- We note that the oct-tree is not the only possible grouping strategy. Sometimes also so-called kd-trees (Stadel, 2001), or other binary trees are used where subdivisions are done along alternating spatial axes.

- An important property of such hierarchical, tree-based groupings is that they are geometrically fully flexible and adjust to any clustering state the particles may have. They are hence automatically adaptive.

- Also, there is no significant slow-down when severe clustering starts.

- The simplest way to construct the hierarchical grouping is to sequentially insert particles into the tree, and then to compute the multipole moments recursively.

4.3 Tree walk

The force calculation with the tree then proceeds by *walking the tree*. Starting at the root node, one checks for every node whether the opening angle under which it is seen is smaller than a prescribed tolerance angle θ_c (see also Salmon & Warren, 1994). If this is the case, the multipole expansion of the node can be accepted, and the corresponding partial force is evaluated and added to an accumulation of the total force. The tree walk along this branch of the tree can then be stopped. Otherwise, one must open the tree node and consider all its sub-nodes in turn.

The resulting force is then approximate in nature by construction, but the overall size of the error can be conveniently controlled by the tolerance opening angle θ_c. If one makes this smaller, more nodes will have to be opened. This will make the residual force errors smaller, but at the price of a higher computational cost. In the limit of $\theta_c \to 0$ one gets back to the expensive direct summation force.

An interesting variant of this approach to walk the tree is obtained by not only expanding the potential on the source side into a multipole expansion, but also around the target coordinate. This can yield a substantial additional acceleration and results in so-called fast multipole methods (FFM). The FALCON code of Dehnen (2000, 2002) employs this approach. A further advantage of the FFM formulation is that force anti-symmetry is manifest, so that momentum conservation to machine precision can be achieved. Unfortunately, the speed advantages of FFM compared to ordinary tree codes are significantly alleviated once individual time-step schemes are considered. Also, FFM is more difficult to parallelize efficiently on distributed memory machines.

Cost of the tree-based force computations

How do we expect the total cost of the tree algorithm to scale with particle number N? For simplicity, let's consider a sphere of size R containing N particles that are approximately homogeneously distributed. The mean particle spacing of these particles will then be

$$d = \left[\frac{(4\pi/3)R^3}{N}\right]^{1/3}. \tag{4.10}$$

We now want to estimate the number of nodes that we need for calculating the force on a central particle in the middle of the sphere. We can identify the computational cost with the number of interaction terms that are needed. Since the used nodes must tessellate the sphere, their number can be estimated as

$$N_{\rm nodes} = \int_d^R \frac{4\pi r^2\,{\rm d}r}{l^3(r)}, \tag{4.11}$$

where $l(r)$ is the expected node size at distance r, and d is the characteristic distance of the nearest particle. Since we expect the nodes to be close to their maximum

allowed size, we can set $l \simeq \theta_c r$. We then obtain

$$N_{\rm nodes} = \frac{4\pi}{\theta_c^3} \ln \frac{R}{d} \propto \frac{\ln N}{\theta_c^3}. \tag{4.12}$$

The total computational cost for a calculation of the forces for all particles is therefore expected to scale as $\mathcal{O}(N \ln N)$. This is a very significant improvement compared with the N^2-scaling of direct summation.

We may also try to estimate the expected typical force errors. If we keep only monopoles, the error in the force per unit mass from one node should roughly be of the order of the truncation error, i.e. about

$$\Delta F_{\rm node} \sim \frac{G M_{\rm node}}{r^2} \theta^2. \tag{4.13}$$

The errors from multipole nodes will add up in quadrature, hence

$$(\Delta F_{\rm tot})^2 \sim N_{\rm node} (\Delta F_{\rm node})^2 = N_{\rm node} \left(\frac{G M_{\rm node}}{r^2} \theta^2 \right)^2 \propto \frac{\theta^4}{N_{\rm node}} \propto \theta^7. \tag{4.14}$$

The force error for a scheme with monopoles only therefore scales as $(\Delta F_{\rm tot}) \propto \theta^{3.5}$, roughly inversely as the invested computational cost. A much more detailed analysis of the performance characteristics of tree codes can be found, for example, in Hernquist (1987).

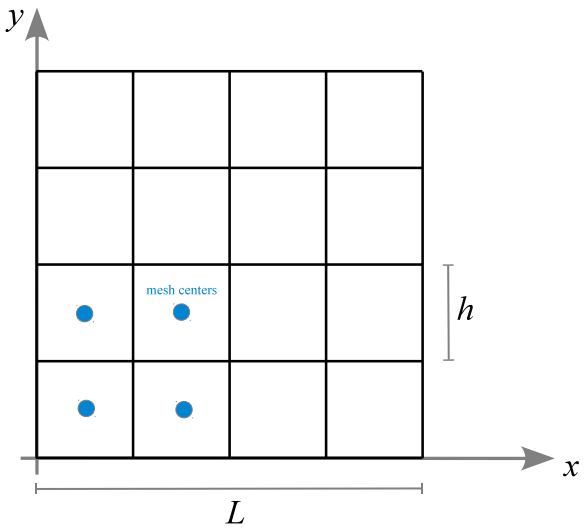
5 The particle-mesh technique

An important approach to accelerate the force calculation for an N-body system lies in the use of an auxiliary mesh. Conceptually, this so-called particle-mesh (PM) technique (White et al., 1983; Klypin & Shandarin, 1983) that was originally pioneered in plasma physics (Hockney & Eastwood, 1988) involves four steps:

1. Construction of a density field ρ on a suitable mesh.
2. Computation of the potential on the mesh by solving the Poisson equation.
3. Calculation of the force field from the potential.
4. Calculation of the forces at the original particle positions.

We shall now discuss these four steps in turn.

5.1 Mass/charge assignment



We want to put N particles with mass m_i and coordinates $\mathbf{x}_i$ $(i = 1, 2, \ldots, N)$ onto a mesh with uniform spacing $h = L/N_\mathrm{g}$. For simplicity, we will assume a cubical calculational domain with extension L and a number of N_g grid cells per dimension. Let $\{\mathbf{x_p}\}$ denote the set of discrete cell-centers, with $\mathbf{p} = (p_x, p_y, p_z)$

being a suitable integer index ($0 \leq p_{x,y,z} < N_g$). Note that one may equally well identify the $\{\mathbf{x_p}\}$ with the lower left corner of a mesh cell, if this is more practical.

We associate a shape function $S(\mathbf{x})$ with each particle, normalized according to

$$\int S(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1. \tag{5.1}$$

To each mesh-cell, we then assign the fraction $W_{\mathbf{p}}(\mathbf{x}_i)$ of particle i's mass that falls into the cell indexed by $\mathbf{p}$. This is given by the overlap of the mesh cell with the shape function, namely:

$$W_{\mathbf{p}}(\mathbf{x}_i) = \int_{\mathbf{x_p}-\frac{\mathbf{h}}{2}}^{\mathbf{x_p}+\frac{\mathbf{h}}{2}} S(\mathbf{x}_i - \mathbf{x})\,\mathrm{d}\mathbf{x} \tag{5.2}$$

The integration extends here over the cubical cell $\mathbf{p}$. By introducing the top-hat function

$$\Pi(\mathbf{x}) = \begin{cases} 1 & \text{for } |\mathbf{x}| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

we can extend the integration boundaries to all space and write instead:

$$W_{\mathbf{p}}(\mathbf{x}_i) = \int \Pi\left(\frac{\mathbf{x} - \mathbf{x_p}}{h}\right) S(\mathbf{x}_i - \mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{5.4}$$

Note that this also shows that the assignment function W is a convolution of Π with S. The full density in grid cell $\mathbf{p}$ is then given by

$$\rho_{\mathbf{p}} = \frac{1}{h^3}\sum_{i=1}^{N} m_i W_{\mathbf{p}}(\mathbf{x}_i). \tag{5.5}$$

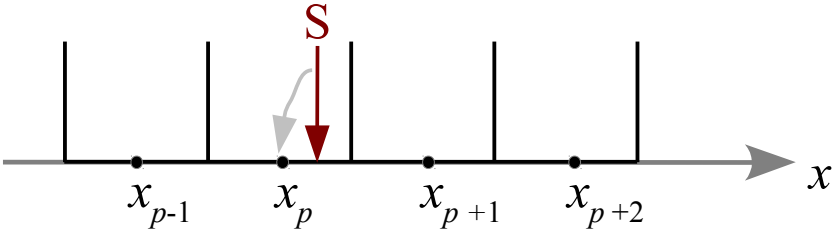
These general formula evidently depend on the specific choice one makes for the shape function $S(\mathbf{x})$. We discuss below a few of the most commonly employed low-order assignment schemes.

5.1.1 Nearest grid point (NGP) assignment

The simplest possible choice for S is a Dirac δ-function. One then gets:

$$W_{\mathbf{p}}(\mathbf{x}_i) = \int \Pi\left(\frac{\mathbf{x} - \mathbf{x_p}}{h}\right) \delta(\mathbf{x}_i - \mathbf{x})\,\mathrm{d}\mathbf{x} = \Pi\left(\frac{\mathbf{x}_i - \mathbf{x_p}}{h}\right). \tag{5.6}$$

In other words, this means that $W_{\mathbf{p}}$ is either 1 (if the coordinate of particle i lies inside the cell), or otherwise it is zero. Consequently, the mass of particle i is fully assigned to exactly one cell – the nearest grid point. The sketch below illustrates this further.



5.1.2 Clouds-in-cell (CIC) assignment

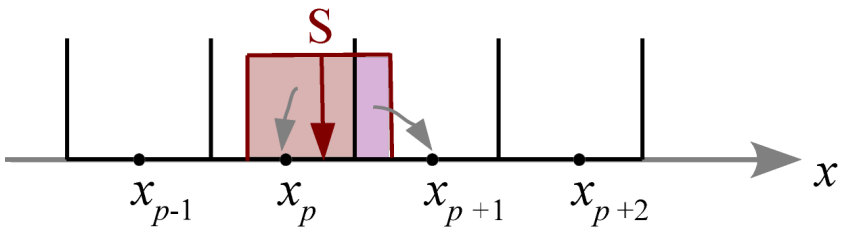
Here one adopts as shape function

$$S(\mathbf{x}) = \frac{1}{h^3}\Pi\left(\frac{\mathbf{x}}{h}\right), \tag{5.7}$$

which is the same cubical 'cloud' shape as that of individual mesh cells. The assignment function is

$$W_{\mathbf{p}}(\mathbf{x}_i) = \int \Pi\left(\frac{\mathbf{x}-\mathbf{x_p}}{h}\right)\frac{1}{h^3}\Pi\left(\frac{\mathbf{x}_i-\mathbf{x}}{h}\right)\mathrm{d}\mathbf{x}, \tag{5.8}$$

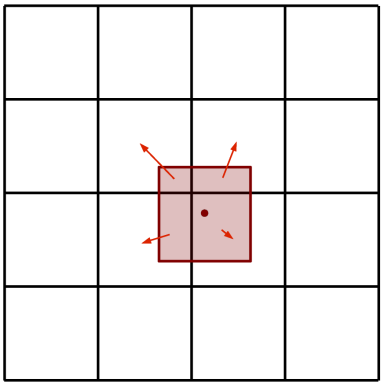
which only has a non-zero (and then constant) integrand if the cubes centred on $\mathbf{x}_i$ and $\mathbf{x_p}$ overlap. How can this overlap be calculated? The 1D sketch below can help to make this clear.



Recall that for one of the dimensions we have $x_p = (p_x{+}1/2)h$ for $p \in \{0, 1, 2, \ldots, N-1\}$. for a given particle coordinate x_i we may first calculate a 'floating point index' by inverting this relation, yielding $p_f = x_i/h - 1/2$. The index of the left cell of the two cells with some overlap is then given by $p = \lfloor p_f \rfloor$, where the brackets denote the integer floor, i.e. the largest integer not larger than p_f. We may then further define $p^\star \equiv p_f - p$, which is a number between 0 and 1. From the sketch, we see that the length of the overlap of the particle's cloud with the cell p is $h - hp^\star$, hence the assignment function at cell p takes on the value $W_p = 1 - m^\star$ for this location of the particle, whereas the assignment function for the neighboring cell $p+1$ will take on the value $W_{p+1} = m^\star$.

These considerations readily generalize to 2D and 3D. For example, in 2D, we first assign to the y_i-coordinate of point i a 'floating point index' $q_f = y_i/h - 1/2$. We can then use this to compute a cell index as the integer floor $q = \lfloor q_f \rfloor$, and a fractional contribution $q^\star = q_f - q$. We then obtain the following weights for the assignment of a particle's mass to the four cells its 'cloud' touches in 2D (as sketched):

$$W_{p,q} = (1-p^\star)(1-q^\star) \tag{5.9}$$
$$W_{p+1,q} = p^\star(1-q^\star) \tag{5.10}$$
$$W_{p,q+1} = (1-p^\star)q^\star \tag{5.11}$$
$$W_{p+1,q+1} = p^\star q^\star \tag{5.12}$$



In the corresponding 3D case, each particle contributes to the weight functions of 8 cells, or in other words, it is spread over 8 cells.

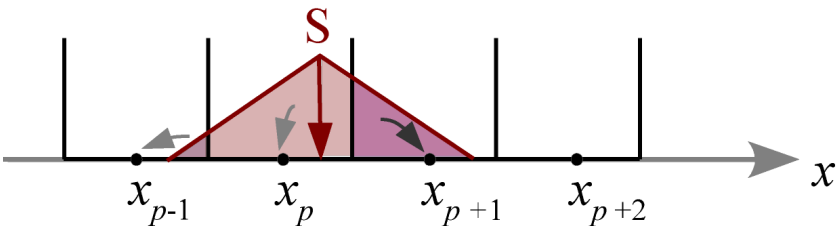
5.1.3 Triangular shaped clouds (TSC) assignment

One can construct a systematic sequence of ever higher-order shape functions by adding more convolutions with the top-hat kernel. For example, the next higher order (in 3D) is given by

$$W_{\mathbf{p}}(\mathbf{x}_i) = \int \Pi\left(\frac{\mathbf{x}-\mathbf{x}_\mathbf{p}}{h}\right) \frac{1}{h^3}\Pi\left(\frac{\mathbf{x}_i-\mathbf{x}-\mathbf{x}'}{h}\right) \frac{1}{h^3}\Pi\left(\frac{\mathbf{x}'}{h}\right) \mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{x}' \quad (5.13)$$

$$= \frac{1}{h^6}\int \Pi\left(\frac{\mathbf{x}-\mathbf{x}_\mathbf{p}}{h}\right) \Pi\left(\frac{\mathbf{x}_i-\mathbf{x}}{h}\right) \Pi\left(\frac{\mathbf{x}'-\mathbf{x}}{h}\right) \mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{x}'. \quad (5.14)$$

This still has a simple geometric interpretation. If one pictures the kernel shape as a triangle with total base length $2h$, then the fraction assigned to a certain cell is given by the area of overlap of this triangle with the cell of interest. The triangle will now in general touch 3 cells per dimension, making an evaluation correspondingly more expensive. In 3D, 27 cells are touched for every particle.



What's then the advantage of using TSC over CIC, if any? Or should one stick with NGP? The assignment schemes differ in the smoothness and differentiability of the reconstructed density field. In particular, for NGP, the assigned density and hence the resulting force jump discontinuously when a particle crosses a cell boundary. The resulting force law will then at best be piece-wise constant.

In contrast, the CIC scheme produces a force that is piece-wise linear and continuous, but has a first derivative that jumps. Here the information where a particle is inside a certain cell is not completely lost, unlike in NGP.

Table 5.1: Commonly used shape functions.

Name	Cloud shape $S(x)$	# of cells used	assignment function shape
NGP	$\delta(x)$	1^d	Π
CIC	$\frac{1}{h^d}\Pi\left(\frac{\mathbf{x}}{h}\right)$	2^d	$\Pi \star \Pi$
TSC	$\frac{1}{h^d}\Pi\left(\frac{\mathbf{x}}{h}\right) \star \frac{1}{h^d}\Pi\left(\frac{\mathbf{x}}{h}\right)$	3^d	$\Pi \star \Pi \star \Pi$

Finally, TSC is yet smoother, and also the first derivative of the force is continuous. Which of these schemes is the preferred choice is ultimately problem-dependent. In most cases, CIC and TSC are quite good options, providing sufficient accuracy with still reasonably small (and hence computationally cheap) assignment kernels. The latter get invariably bigger and bigger for higher-order assignment schemes, which not only is computationally ever more costly but also invokes additional communication overhead in certain parallelization schemes.

5.2 Solving for the gravitational potential

Once the density field is obtained, one would like to solve Poisson's equation

$$\nabla^2\Phi = 4\pi G\,\rho \tag{5.15}$$

and obtain the gravitational potential discretized on the same mesh. There are primarily two methods that are in widespread use for this:

First, there are Fourier-transform based methods which exploit the fact that the potential can be viewed as a convolution of a Green's function with the density field. In Fourier-space, one can then exploit the convolution theorem and cast the computationally expensive convolution into a cheap algebraic multiplication. Due to the importance of this approach, we will discuss it extensively in the next chapter.

Second, there are so-called iterative solvers for Poisson's equation which yield a solution directly in real-space. Simple versions of such iterative solvers use Jacobi or Gauss-Seidel iteration, more complicated ones employ a sophisticated multi-grid approach to speed up convergence. We shall discuss these methods in chapter 7.

5.3 Calculation of the forces

Let's assume for the moment that we already obtained the gravitational potential Φ on the mesh, with one of the methods mentioned above. We would then like to get the acceleration field from

$$\mathbf{a} = -\nabla\Phi. \tag{5.16}$$

One can achieve this by calculating a numerical derivative of the potential by *finite differencing*. For example, the simplest estimate of the force in the x-direction would

be

$$a_x(i,j,k) = -\frac{\Phi(i+1,j,k) - \Phi(i-1,j,k)}{2h}, \tag{5.17}$$

where $\mathbf{p} = (i,j,k)$ is a cell index. The truncation error of this expression is $\mathcal{O}(h^2)$, hence the estimate of the derivative is second-order accurate.

Alternatively, one can use *larger stencils* to obtain a more accurate finite difference approximation of the derivative, at greater computational cost. For example, the 4-point expression

$$a_x(i,j,k) = -\frac{1}{2h}\left\{\frac{4}{3}\left[\Phi(i+1,j,k) - \Phi(i-1,j,k)\right] - \frac{1}{6}\left[\Phi(i+2,j,k) - \Phi(i-2,j,k)\right]\right\} \tag{5.18}$$

can be used, which has a truncation error of $\mathcal{O}(h^4)$ (which can be simply proven through Taylor expansion).

For the y- and z-dimensions, corresponding formulae where j or k are varied and the other cell coordinates are held fixed can be used. Whether a second- or fourth-order discretization formula is used depends again on the question which compromise between accuracy and speed one considers best for a given problem. In many collisionless systems, the residual truncation error of a second-order finite difference approximation of the force will be negligible compared to other errors inherent in the simulation scheme, hence the second-order formula would be sufficient in this case.

5.4 Interpolating from the mesh to the particles

Once we have the force field on a mesh, we are not yet fully done. We actually desire the forces at the particle coordinates of the N-body system, not at the coordinates of the mesh cells of our auxiliary computational grid. We are hence left with the problem of interpolating the forces from the mesh to the particle coordinates.

Recall that we defined the density field in terms of mass assignment functions, of the form

$$\rho_{\mathbf{p}} = \frac{1}{h^3}\sum_i W_{\mathbf{p}}(\mathbf{x}_i) = \frac{1}{h^3}\sum_i W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}}). \tag{5.19}$$

Here we introduced in the last expression an alternative notation for the weight assignment function.

Assume that we have computed the acceleration field on the grid, $\{\mathbf{a}_{\mathbf{p}}\}$. It turns out to be very important to *use the same* assignment kernel as used in the density construction also for the force interpolation, i.e. the force at coordinate $\mathbf{x}$ for a mass m needs to be computed as

$$\mathbf{F}(\mathbf{x}) = m\sum_{\mathbf{p}} \mathbf{a}_{\mathbf{p}} W(\mathbf{x} - \mathbf{x}_{\mathbf{p}}), \tag{5.20}$$

where W denotes the assignment function used for computing the density field on the mesh. This requirement results from the desire to have a vanishing *self-force*, as

well as pairwise antisymmetric forces between every particle pair. The self-force is the force that a particle would feel if just it alone would be present in the system. If numerically this force would be calculated with a non-zero value, the particle would accelerate all by itself, violating momentum conservation. Likewise, for two particles, we would like to have that the forces they mutually exert on each other are equal in magnitude and opposite in direction to each other, such that momentum conservation is manifest.

We now prove that using the same kernels for the mass assignment and force interpolation protects against these numerical artefacts. We start by noting that the acceleration field at a mesh point $\mathbf{p}$ is a linear response to the mass at another mesh point $\mathbf{p}'$ (this can be trivially seen when Fourier techniques are used to solve the Poisson equation). We can hence express the field as

$$\mathbf{a}_{\mathbf{p}} = \sum_{\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}')\, h^3 \rho_{\mathbf{p}'} \tag{5.21}$$

with a Green's function $\mathbf{d}(\mathbf{p}, \mathbf{p}')$. This vector-valued Green's function for the force is antisymmetric, i.e. it changes sign when the two points in the argument are swapped. Note that $h^3\rho_{\mathbf{p}'}$ is simply the mass contained in mesh cell $\mathbf{p}'$.

We can now calculate the self-force resulting from the density assignment and interpolation steps:

$$\begin{aligned}
\mathbf{F}_{\text{self}}(\mathbf{x}_i) &= m_i \sum_{\mathbf{p}} W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}}) \mathbf{a}_{\mathbf{p}} && (5.22)\\
&= m_i \sum_{\mathbf{p}} W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}}) \sum_{\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') h^3 \rho_{\mathbf{p}'} && (5.23)\\
&= m_i \sum_{\mathbf{p}} W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}}) \sum_{\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') m_i W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}'}) && (5.24)\\
&= m_i^2 \sum_{\mathbf{p},\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}}) W(\mathbf{x}_i - \mathbf{x}_{\mathbf{p}'}) && (5.25)\\
&= 0. && (5.26)
\end{aligned}$$

Here we have started out with the interpolation from the mesh-based acceleration field, and then inserted the expansion of the latter as as convolution over the density field of the mesh. Finally, we put in the density contribution created by the particle i at a mesh cell $\mathbf{p}'$. We then see that the double sum vanishes because of the antisymmetry of $\mathbf{d}$ and the symmetry of the kernel product under exchange of $\mathbf{p}$ and $\mathbf{p}'$. Note that this however only works because the kernels used for force interpolation and density assignment are actually equal – it would have not worked out if they would be different, which brings us back to the point emphasized above.

Now let's turn to the force antisymmetry. The force exerted on a particle 1 of

mass m_1 at location $\mathbf{x}_1$ due to a particle 2 of mass m_2 at location $\mathbf{x}_2$ is given by

$$\begin{aligned}
\mathbf{F}_{12} &= m_1 \mathbf{a}(\mathbf{x}_1) = m_1 \sum_{\mathbf{p}} W(\mathbf{x}_1 - \mathbf{x}_\mathbf{p}) \mathbf{a}_\mathbf{p} && (5.27)\\
&= m_1 \sum_{\mathbf{p}} W(\mathbf{x}_1 - \mathbf{x}_\mathbf{p}) \sum_{\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') h^3 \rho_{\mathbf{p}'} && (5.28)\\
&= m_1 \sum_{\mathbf{p}} W(\mathbf{x}_1 - \mathbf{x}_\mathbf{p}) \sum_{\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') m_2 W(\mathbf{x}_2 - \mathbf{x}_{\mathbf{p}'}) && (5.29)\\
&= m_1 m_2 \sum_{\mathbf{p},\mathbf{p}'} \mathbf{d}(\mathbf{p}, \mathbf{p}') W(\mathbf{x}_1 - \mathbf{x}_\mathbf{p}) W(\mathbf{x}_2 - \mathbf{x}_{\mathbf{p}'}) && (5.30)
\end{aligned}$$

Likewise we obtain for the force experienced by particle 2 due to particle 1:

$$\mathbf{F}_{21} = m_1 m_2 \sum_{\mathbf{p}',\mathbf{p}} \mathbf{d}(\mathbf{p}, \mathbf{p}') W(\mathbf{x}_2 - \mathbf{x}_\mathbf{p}) W(\mathbf{x}_1 - \mathbf{x}_{\mathbf{p}'}) \tag{5.31}$$

We may swap the summation indices through relabeling and exploit the antisymmetry of $\mathbf{d}$, obtaining:

$$\mathbf{F}_{21} = -m_1 m_2 \sum_{\mathbf{p}',\mathbf{p}} \mathbf{d}(\mathbf{p}, \mathbf{p}') W(\mathbf{x}_1 - \mathbf{x}_\mathbf{p}) W(\mathbf{x}_2 - \mathbf{x}_{\mathbf{p}'}) \tag{5.32}$$

Hence we have $\mathbf{F}_{12} + \mathbf{F}_{21} = 0$, independent of where the points are located on the mesh.

6 Force calculation with Fourier transform techniques

Fourier transforms provide a powerful tool for solving certain partial differential equations. In this subsection we shall consider the particularly important example of using them to solve Poisson's equation, but we note that the basic technique can be used in similar form also for other systems of equations.

6.1 Convolution problems

Suppose we want to solve Poisson's equation,

$$\nabla^2 \Phi = 4\pi G \rho, \tag{6.1}$$

for a given density distribution ρ. Actually, we can readily write down a solution for a non-periodic space, since we know the Newtonian potential of a point mass, and the equation is linear. The potential is simply a linear superposition of contributions from individual mass elements, which in the continuum can be written as the integration:

$$\Phi(\mathbf{x}) = -\int G \, \frac{\rho(\mathbf{x}') \, \mathrm{d}\mathbf{x}'}{|\mathbf{x} - \mathbf{x}'|}. \tag{6.2}$$

This is recognized to be a convolution integral of the form

$$\Phi(\mathbf{x}) = \int g(\mathbf{x} - \mathbf{x}') \, \rho(\mathbf{x}') \, \mathrm{d}\mathbf{x}', \tag{6.3}$$

where

$$g(\mathbf{x}) = -\frac{G}{|\mathbf{x}|} \tag{6.4}$$

is the *Green's function* of Newtonian gravity. The convolution may also be formally written as:

$$\Phi = g \star \rho. \tag{6.5}$$

We now may recall the *convolution theorem*, which says that the Fourier transform of the convolution of two functions is equal to the product of the individual Fourier transforms of the two functions, i.e.

$$\mathcal{F}(f \star g) = \mathcal{F}(f) \cdot \mathcal{F}(g), \tag{6.6}$$

where $\mathcal{F}$ denotes the Fourier transform and f and g are the two functions. A convolution in real space can hence be transformed to a much simpler, point-by-point multiplication in Fourier space.

There are many problems where this can be exploited to arrive at efficient calculational schemes, for example in solving Poisson's equation for a given density field. Here the central idea is to compute the potential through:

$$\Phi = \mathcal{F}^{-1}\left[\mathcal{F}(g) \cdot \mathcal{F}(\rho)\right], \tag{6.7}$$

i.e. in Fourier space, with $\hat{\Phi}(\mathbf{k}) \equiv \mathcal{F}(\Phi)$, we have the simple equation

$$\hat{\Phi}(\mathbf{k}) = \hat{g}(\mathbf{k}) \cdot \hat{\rho}(\mathbf{k}). \tag{6.8}$$

But how do we solve this in practice?

Let's first assume that we have *periodic boundary conditions* with a box of size L in each dimension. The continuous $\rho(\mathbf{x})$ can in this case be written as a Fourier series of the form

$$\rho(\mathbf{x}) = \sum_{\mathbf{k}} \rho_{\mathbf{k}}\, \mathrm{e}^{i\mathbf{k}\mathbf{x}}, \tag{6.9}$$

where the sum over the $\mathbf{k}$-vectors extends over a discrete spectrum of wave vectors, with

$$\mathbf{k} \in \frac{2\pi}{L} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix}, \tag{6.10}$$

where n_1, n_2, n_3 are all positive and negative integer numbers. The allowed modes in $\mathbf{k}$ hence form an infinitely extended Cartesian grid with spacing $2\pi/L$ – because of the periodicity condition, only these waves 'fit' into the box. (For a real field such as ρ, there is also a reality constraint of the form $\rho_{\mathbf{k}} = \rho^{\star}_{-\mathbf{k}}$, hence here the modes are not all independent.) The Fourier coefficients can be calculated as

$$\rho_{\mathbf{k}} = \frac{1}{L^3} \int_V \rho(\mathbf{x})\, \mathrm{e}^{-i\mathbf{k}\mathbf{x}} \mathrm{d}\mathbf{x}, \tag{6.11}$$

where the integration is over one instance of the periodic box.

More generally, the periodic Fourier series features the following orthogonality and closure relationships:

$$\frac{1}{L^3} \int \mathrm{d}\mathbf{x}\, \mathrm{e}^{i(\mathbf{k}-\mathbf{k}')} = \delta_{\mathbf{k},\mathbf{k}'} \tag{6.12}$$

$$\frac{1}{L^3} \sum_{\mathbf{k}} \mathrm{e}^{i\mathbf{k}\mathbf{x}} = \delta(\mathbf{x}), \tag{6.13}$$

where the first relation gives a Kronecker delta, the second a Dirac δ-function.

Let's now look at the Poisson equation again and replace the potential and the density field with their corresponding Fourier series:

$$\nabla^2 \left(\sum_{\mathbf{k}} \Phi_{\mathbf{k}} \, \mathrm{e}^{i\mathbf{kx}} \right) = 4\pi G \left(\sum_{\mathbf{k}} \rho_{\mathbf{k}} \, \mathrm{e}^{i\mathbf{kx}} \right) \tag{6.14}$$

We see that we can easily carry out the spatial derivate on the left hand side, yielding:

$$\sum_{\mathbf{k}} \left(-\mathbf{k}^2 \Phi_{\mathbf{k}} \right) \, \mathrm{e}^{i\mathbf{kx}} = 4\pi G \sum_{\mathbf{k}} \rho_{\mathbf{k}} \, \mathrm{e}^{i\mathbf{kx}} \tag{6.15}$$

The equality must hold for each of the Fourier modes separately, hence we infer

$$\Phi_{\mathbf{k}} = -\frac{4\pi G}{\mathbf{k}^2} \rho_{\mathbf{k}}. \tag{6.16}$$

Comparing with equation (6.8), this means we have identified the Green's function of the Poisson equation in a periodic space as

$$g_{\mathbf{k}} = -\frac{4\pi G}{\mathbf{k}^2}. \tag{6.17}$$

6.2 The discrete Fourier transform (DFT)

The above considerations were still for a continuos density field. On a computer, we will usually only have a discretized version of the field $\rho(\mathbf{x})$, defined at a set of points. Assuming we have N equally spaced points per dimension, the $\mathbf{x}$ positions may only take on the discrete positions

$$\mathbf{x}_{\mathbf{p}} = \frac{L}{N} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \quad \text{where } p_1, p_2, p_3 \in \{0, 1, \ldots, N-1\}. \tag{6.18}$$

With the replacement $\mathrm{d}^3\mathbf{x} \to (L/N)^3$, we can cast the Fourier integral (6.11) into a discrete sum:

$$\rho_{\mathbf{k}} = \frac{1}{N^3} \sum_{\mathbf{p}} \rho_{\mathbf{p}} \, \mathrm{e}^{-i\mathbf{kx_p}}. \tag{6.19}$$

Because of the periodicity and the finite number of density values that is summed over, it turns out that this also restricts the number of $\mathbf{k}$ values that give different answers – shifting $\mathbf{k}$ in any of the dimensions by N times the fundamental mode $2\pi/L$ gives again the same result. We may then for example select as primary set of $\mathbf{k}$-modes the values

$$\mathbf{k}_{\mathbf{l}} = \frac{2\pi}{L} \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} \quad \text{where } l_1, l_2, l_3 \in \{0, 1, \ldots, N-1\}, \tag{6.20}$$

and the construction of ρ through the Fourier series becomes a finite sum over these N^3 modes. We have now arrived at the *discrete Fourier transform* (DFT), which can equally well be written as:

$$\hat{\rho}_{\mathbf{l}} = \frac{1}{N^3} \sum_{\mathbf{p}} \rho_{\mathbf{p}} \, \mathrm{e}^{-i \frac{2\pi}{N} \mathbf{l}\mathbf{p}} \tag{6.21}$$

$$\rho_{\mathbf{p}} = \sum_{\mathbf{l}} \hat{\rho}_{\mathbf{l}} \, \mathrm{e}^{i \frac{2\pi}{N} \mathbf{l}\mathbf{p}} \tag{6.22}$$

Here are some notes about different aspects of the Fourier pair defined by these relations:

- The two transformations are an invertible linear mapping of a set of N^3 (or N in 1D) complex values $\rho_{\mathbf{p}}$ to N^3 complex values $\hat{\rho}_{\mathbf{l}}$, and vice versa.

- To label the frequency values, $\mathbf{k} = (2\pi/L) \cdot \mathbf{l}$, one often conventionally uses the set $l \in \{-N/2, \ldots, -1, 0, 1, \ldots, \frac{N}{2} - 1\}$ instead of $l \in \{0, 1, \ldots, N-1\}$, which is always possible because shifting l by multiples of N does not change anything because this yields only a 2π phase factor. With this shift, the occurrence of both negative and positive frequencies is made more explicit, and they are arranged quasi-symmetrically in a box in $\mathbf{k}$-space centered on $\mathbf{k} = (0, 0, 0)$. The box extends out to
$$k_{\max} = \frac{N}{2} \frac{2\pi}{L}, \tag{6.23}$$
which is the so-called Nyquist frequency (e.g. Diniz et al., 2002). Adding waves beyond the Nyquist frequency in a reconstruction of ρ on a given grid would add redundant information that could not be unambiguously recovered again from the discretized density field. (Instead, the power in these waves would be erroneously mapped to lower frequencies – this is called *aliasing*, see also the so-called *sampling theorem*.)

- Parseval's theorem relates the quadratic norm of the transform pair, namely
$$\sum_{\mathbf{p}} |\rho_{\mathbf{p}}|^2 = N^3 \sum_{\mathbf{l}} |\hat{\rho}_{\mathbf{l}}|^2. \tag{6.24}$$

- The $1/N^3$ normalization factor could equally well be placed in front of the Fourier series instead of the Fourier transform, or one may split it symmetrically and introduce a factor $1/\sqrt{N^3}$ in front of both. This is just a matter of convention, and both alternative conventions are sometimes used.

- In fact, many computer libraries for the DFT will omit the factor N completely and leave it up to the user to introduce it where needed. Commonly, the DFT

library functions define as forward transform of a set of N complex numbers x_j, with $j \in \{0, \ldots, N-1\}$, the set of N complex numbers:

$$y_k = \sum_{j=0}^{N-1} x_j \, \mathrm{e}^{-i\frac{2\pi}{N} j \cdot k}. \tag{6.25}$$

The backwards transform is then defined as

$$y_k = \sum_{j=0}^{N-1} x_j \, \mathrm{e}^{i\frac{2\pi}{N} j \cdot k}. \tag{6.26}$$

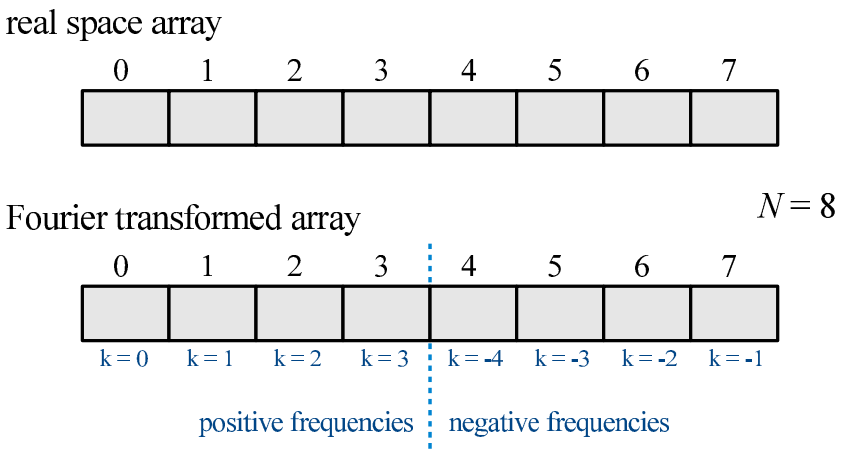
This form of writing the Fourier transform is now nicely symmetric, with the *only difference* between forward and backward transforms being a sign in the exponential function. However, in this case we have that $\mathcal{F}^{-1}(\mathcal{F}(\mathbf{x})) = N\mathbf{x}$, i.e. to get back to the original input vector $\mathbf{x}$ one must eventually divide by N. Note that the multi-dimensional transforms are simply Cartesian products of one-dimensional transforms, i.e. those are obtained as straightforward generalizations of the one-dimensional definition.

- Computing the DFT of N numbers has in principal a computational cost of order $\mathcal{O}(N^2)$. This is because for each of the N numbers one has to calculate N terms and sum them up. Fortunately, in 1965, the *Fast Fourier Transform* (FFT) algorithm (Cooley & Tukey, 1965) has been (re)discovered (Gauss had already known it in principle). This method for calculating the DFT breaks down the problem recursively into smaller and smaller blocks. It turn out that this divide and conquer strategy can reduce the computational cost to $\mathcal{O}(N \log N)$, which is a very significant difference. The result of the FFT algorithm is mathematically identical to the DFT. But actually, in practice the FFT is even better than a direct computation of the DFT, because as an aside the FFT also reduces the round-off error that would otherwise be incurred. It is ultimately only because of the existence of the FFT algorithm that Fourier methods are so widely used in numerical calculations and applicable to even very large problem sizes.

6.3 Storage conventions for the DFT

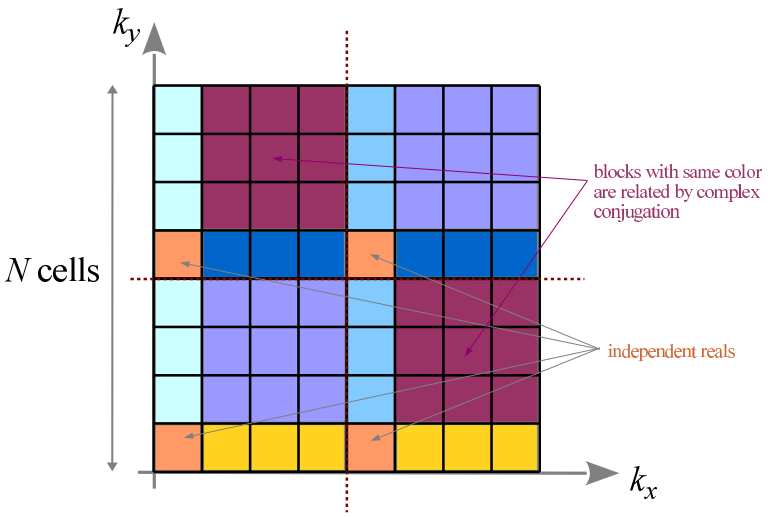
Most numerical libraries for computing the FFT store both the original field and its Fourier transform as simple arrays indexed by $k \in \{0, \ldots, N-1\}$. The negative frequencies will then be stored in the upper half of the array, consistent with what one obtains by subtracting N from the linear index.

An example in 1D for $N = 8$ may help to make this clear:



Correspondingly, in 2D, the grid of real-space values is mapped to a grid of k-space values of the same dimensions. Again, negative frequencies seem to be stored 'backwards', with the smallest negative frequency having the largest linear index, and the most negative frequency appearing as first value past the middle of the mesh. But this is again fully consistent with the translational invariance in k-space with respect to shifts of the indices by multiples of N.

Finally, when we have a real real-space field (such as the physical density), the discrete Fourier transform fulfills a reality constraint of the form $\hat{\rho}_{\mathbf{k}} = \hat{\rho}^{\star}_{-\mathbf{k}}$. This implies a set of relations between the complex values that make up the Fourier transform of ρ, reducing the number of values that can be chosen arbitrarily. How is this manifested in the discrete case? Consider the following sketch, in which regions of like color are related to each other by the reality constraint. Note that $k_x = N/2$ indices are aliased to themselves under complex conjugation, i.e. negating this gives $k_x = -N/2$, but since N can be added, this mode really maps again to $k_x = N/2$. Nevertheless, for the yellow regions there are always different partner cells when one considers the corresponding $-\mathbf{k}$ cell. Only for the red cells, this is not the case; those are mapped to themselves and are hence real due to the reality constraint.



If we now count how many independent numbers we have in the Fourier transformed grid of a 2D real field, we find

$$2\left(\frac{N}{2}-1\right)^2 \times 2 \;+\; 4\left(\frac{N}{2}-1\right) \times 2 \;+\; 4 \times 1 \tag{6.27}$$

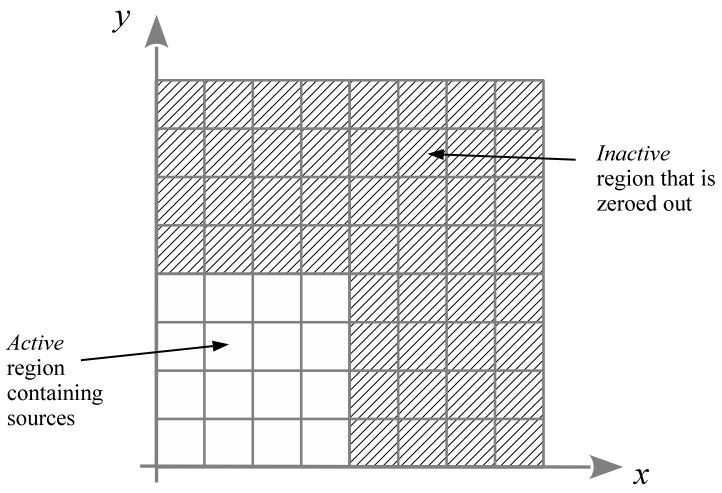
The first term accounts for the two square-shaped regions that have different mirrored regions. Those contain $(\frac{N}{2}-1)^2$ complex numbers, each with two independent real and imaginary values. Then there are 4 different sections of rows and columns that are related to each other by mirroring in k-space. Those contain $\frac{N}{2}-1$ complex numbers each. Finally, there are 4 independent cells that are real and hence account for one independent value each. Reassuringly, the sum of equation (6.27) works out to N^2, which is the result we expect: the number of independent values in Fourier space must be exactly equal to the N^2 real values we started out with, otherwise we would not expect a strictly reversible transformation.

6.4 Non-periodic problems with 'zero padding'

Can we use the FFT/DFT techniques discussed above also to calculate non-periodic force fields? At first, this may seem impossible since the DFT is intrinsically periodic. However, through the so-called zero-padding trick one can circumvent this limitation.

Let's discuss the procedure based on a 2D example (it works also in 1D/3D, of course):

1. We need to arrange our mesh such that the source distribution lives only in one quarter of the mesh, the rest of the density field needs to be zeroed out. Schematically we hence have the following situation:



2. We now set up our desired real-space Green's function, i.e. the response of a mass/charge at the origin. The Green's function for the whole mesh is set-up as $g_{N-i,j} = g_{i,N-j} = g_{N-i,N-j} = g_{i,j}$ where $0 \leq i, j \leq N/2$. This is equivalent to defining g everywhere on the mesh and using as relevant distance the distance to the *nearest periodic image* of the origin. Note that by replicating g with the condition of periodicity, the periodically extended mesh then effectively yields a Green's function that is nicely symmetric around the origin.

3. We now want to carry out the real-space convolution

$$\phi = g \star \rho \tag{6.28}$$

   by using the definition of the discrete, periodic convolution

$$\Phi_{\mathbf{p}} = \sum_{\mathbf{n}} g_{\mathbf{p}-\mathbf{n}}\, \rho_{\mathbf{n}}, \tag{6.29}$$

   where both g and ρ are treated as periodic fields for which adding multiples of N to the indices does not change anything. We see that this sum indeed yields the correct result for the non-periodic potential in the quarter of the mesh that contains our source distribution. This is because the Green's function 'sees' only one copy of the source distribution in this sector; the zero-padded region is big enough to prevent any cross-talk from the (existing) periodic images of the source distribution. This is different in the other three quadrants of the mesh. Here we obtain incorrect potential values that are basically useless and need to be discarded.

4. Given that equation (6.29) yields the correct result in the region of the mesh covered by the sources, we may now just as well use periodic FFTs in the usual way to carry out this convolution quickly! The only downside of this procedure is that it features an enlarged cost in terms of CPU and memory usage. Because we have to effectively double the mesh-size compared to the corresponding periodic problem, the cost goes up by a factor of 4 in 2D, and by a factor of 8 in 3D.

5. We note that James (1977) proposed an ingenious trick that allows a more efficient treatment of isolated source distributions. Through suitably determined correction masses on the boundaries, the memory and CPU cost can be reduced compared to the zero-padding approach described above.

7 Iterative solvers and the multigrid technique

7.1 The Poisson equation as a linear system of equations

Let's return to the problem of solving the Poisson equation,

$$\nabla^2 \Phi = 4\pi G \rho, \tag{7.1}$$

and consider first the one-dimensional problem, i.e.

$$\frac{\partial^2 \Phi}{\partial x^2} = 4\pi G \rho(x). \tag{7.2}$$

The spatial derivative on the left hand-side can be approximated as

$$\left(\frac{\partial^2 \Phi}{\partial x^2}\right)_i \simeq \frac{\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}}{h^2}, \tag{7.3}$$

where we have assumed that Φ is discretized with N points on a regular mesh with spacing h, and i is the cell index. This means that we have the equations

$$\frac{\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}}{h^2} = 4\pi G \rho_i. \tag{7.4}$$

There are N of these equations, for the N unknowns Φ_i, with $i \in \{0, 1, \ldots, N-1\}$. This means we should in principle be able to solve this algebraically! In other words, the system of equations can be rewritten as a standard linear set of equations, in the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{7.5}$$

with a vector of unknowns, $\mathbf{x} = \mathbf{\Phi}$, and a right hand side $\mathbf{b} = 4\pi G\, h^2 \rho$. In the 1D case, the matrix $\mathbf{A}$ (assuming periodic boundary conditions) is explicitly given as

$$\mathbf{A} = \begin{pmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \cdots & & & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{pmatrix} \tag{7.6}$$

Solving equation (7.5) directly constitutes a matrix inversion that can in principle be carried out by LU-decomposition or Gauss elimination with pivoting (e.g. Press et al., 1992). However, the computational cost of these procedures is of order $\mathcal{O}(N^3)$, meaning that it becomes extremely costly, and sooner than later infeasible, already for problems of small to moderate size.

But, if we are satisfied with an approximate solution, then we can turn to iterative solvers that are much faster.

7.2 Jacobi iteration

Suppose we decompose the matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U}), \tag{7.7}$$

where $\mathbf{D}$ is the diagonal part, $\mathbf{L}$ is the (negative) lower diagonal part and $\mathbf{U}$ is the upper diagonal part. Then we have

$$[\mathbf{D} - (\mathbf{L} + \mathbf{U})]\,\mathbf{x} = \mathbf{b}, \tag{7.8}$$

and from this

$$\mathbf{x} = \mathbf{D}^{-1}\mathbf{b} + \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}. \tag{7.9}$$

We can use this to define an iterative sequence of vectors $\mathbf{x}^n$:

$$\mathbf{x}^{(n+1)} = \mathbf{D}^{-1}\mathbf{b} + \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(n)}. \tag{7.10}$$

This is called Jacobi iteration (e.g. Saad, 2003). Note that $\mathbf{D}^{-1}$ is trivially obtained because $\mathbf{D}$ is diagonal. i.e. here $(\mathbf{D}^{-1})_{ii} = 1/\mathbf{A}_{ii}$.

The scheme converges if and only if the so-called convergence matrix

$$\mathbf{M} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \tag{7.11}$$

has only eigenvalues that are less than 1, or in other words, that the spectral radius $\rho_s(\mathbf{M})$ fullfils

$$\rho_s(\mathbf{M}) \equiv \max_i |\lambda_i| < 1. \tag{7.12}$$

We can easily derive this condition by considering the error vector of the iteration. At step n it is defined as

$$\mathbf{e}^{(n)} \equiv \mathbf{x}_{\text{exact}} - \mathbf{x}^{(n)}, \tag{7.13}$$

where $\mathbf{x}_{\text{exact}}$ is the exact solution. We can use this to write the error at step $n+1$ of the iteration as

$$\mathbf{e}^{(n+1)} = \mathbf{x}_{\text{exact}} - \mathbf{x}^{(n+1)} = \mathbf{x}_{\text{exact}} - \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{x}^{(n)} = \mathbf{M}\mathbf{x}_{\text{exact}} - \mathbf{M}\mathbf{x}^{(n)} = \mathbf{M}\mathbf{e}^{(n)} \tag{7.14}$$

Hence we find

$$\mathbf{e}^{(n)} = \mathbf{M}^n\mathbf{e}^{(0)}. \tag{7.15}$$

This implies $|\mathbf{e}^{(n)}| \leq [\rho_s(\mathbf{M})]^n |\mathbf{e}^{(0)}|$, and hence convergence if the spectral radius is smaller than 1.

For completeness, we state the Jacobi iteration rule for the Poisson equation in 3D when a simple 2-point stencil is used in each dimension for estimating the corresponding derivatives:

$$\Phi_{i,j,k}^{(n+1)} = \frac{1}{6}\left(\Phi_{i+1,j,k}^{(n)} + \Phi_{i-1,j,k}^{(n)} + \Phi_{i,j+1,k}^{(n)} + \Phi_{i,j-1,k}^{(n)} + \Phi_{i,j,k+1}^{(n)} + \Phi_{i,j,k-1}^{(n)} - 4\pi G h^2 \rho_{i,j,k}\right) \tag{7.16}$$

7.3 Gauss-Seidel iteration

The central idea of Gauss-Seidel iteration is to use the updated values, as soon as they become available, for computing further updated values. We can formalize this as follows. Adopting the same decomposition of $\mathbf{A}$ as before, we can write

$$(\mathbf{D} - \mathbf{L})\mathbf{x} = \mathbf{U}\mathbf{x} + \mathbf{b}, \tag{7.17}$$

from which we obtain

$$\mathbf{x} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{x} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}, \tag{7.18}$$

suggesting the iteration rule

$$\mathbf{x}^{(n+1)} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{x}^{(n)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}. \tag{7.19}$$

This seems at first problematic, because we can't easily compute $(\mathbf{D} - \mathbf{L})^{-1}$. But we can modify the last equation as follows:

$$\mathbf{D}\mathbf{x}^{(n+1)} = \mathbf{U}\mathbf{x}^{(n)} + \mathbf{L}\mathbf{x}^{(n+1)} + \mathbf{b}. \tag{7.20}$$

From which we get the alternative form:

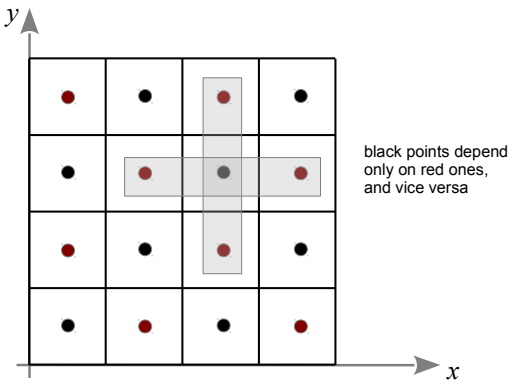
$$\mathbf{x}^{(n+1)} = \mathbf{D}^{-1}\mathbf{U}\mathbf{x}^{(n)} + \mathbf{D}^{-1}\mathbf{L}\mathbf{x}^{(n+1)} + \mathbf{D}^{-1}\mathbf{b}. \tag{7.21}$$

Again, this may seem of little help because it looks like $\mathbf{x}^{(n+1)}$ would only be implicitly given. However, if we start computing the new elements in the first row $i = 1$ of this matrix equation, we see that no values of $\mathbf{x}^{(n+1)}$ are actually needed, because $\mathbf{L}$ has only elements below the diagonal. For the same reason, if we then proceed with the second row $i = 2$, then with $i = 3$, etc., only elements of $\mathbf{x}^{(n+1)}$ from rows above the current one are needed. So we can calculate things in this order without problem and make use of the already updated values. It turns out that this speeds up the convergence quite a bit, with one Gauss-Seidel step often being close to two Jacobi steps.

7.3.1 Red black ordering

A problematic point about Gauss-Seidel is that the equations have to be solved in a specific sequential order, meaning that this part cannot be parallelized. Also, the result will in general depend on which element is selected to be the first. To overcome this problem, one can sometimes use so-called red-black ordering, which effectively is a compromise between Jacobi and Gauss-Seidel.

Certain update rules, such as that for the Poisson equation, allow a decomposition of the cells into disjoint sets whose update rules depend only on cells from other sets. For example, for the Poisson equation, this is the case for a chess-board like pattern of 'red' and 'black' cells.



One can then first update all the black points (which rely only on the red points), followed by an update of all the red points (which rely only on the black ones). In the second of these two half-steps, one can then use the updated values from the first half-step, making it intuitively clear why such a scheme can almost double the convergence rate relative to Jacobi.

7.4 The multigrid technique

Iterative solvers like Jacobi or Gauss-Seidel often converge quite slowly, in fact, the convergence seems to "stall" after a few steps and proceeds only anemically. One also sees that high-frequency errors in the solution are damped out quickly by the iterations, but long-wavelength errors die out much more slowly. Intuitively this is not unexpected: In every iteration, only neighboring points communicate, so the information travels only by one cell (or more generally, one stencil length) per iteration. And for convergence, it has to propagate back and forth over the whole domain a few times.

Idea: By going to a coarser mesh, we may be able to compute an improved initial guess which may help to speed up the convergence on the fine grid (Brandt, 1977). Note that on the coarser mesh, the relaxation will be computationally cheaper (since there are only 1/8 as many points in 3D, or 1/4 in 2D), and the convergence rate should be faster, too, because the perturbation is there less smooth and effectively on a smaller scale relative to the coarser grid.

So schematically, we for example might imagine an iteration scheme where we first iterate the problem $\mathbf{Ax} = \mathbf{b}$ on a mesh with cells $4h$, i.e. for times coarser than the fine mesh. Once we have a solution there, we continue to iterate it on a mesh coarsened with cell size $2h$, and only finally we iterate to solution on the fine mesh h.

A couple of questions immediately come up when we want to work out the details of this basic idea:

1. How to get from a coarse solution to a guess on a finer grid?
2. How to solve $\mathbf{Ax} = \mathbf{b}$ on the coarser mesh?
3. What if there is still an error left with long wavelength on the fine grid?

We clearly need mappings from a finer grid to a coarser one, and vice versa! This is the most important issue to solve.

7.4.1 Prolongation and restriction operations

Coarse-to-fine: This transition is an interpolation step, or in the language of multigrid methods (Briggs et al., 2000), it is called *prolongation*. Let $\mathbf{x}^{(h)}$ be a vector defined on a mesh $\Omega^{(h)}$ with N cells and spacing h, covering our computational domain. Similarly, let $\mathbf{x}^{(2h)}$ be a vector living on a coarser mesh $\Omega^{(2h)}$ with twice the spacing and half as many points per dimension. We now define a linear interpolation operator $\mathbf{I}_{2h}^{h}$ that maps points from the coarser to the fine mesh, as follows:

$$\mathbf{I}_{2h}^{h}\mathbf{x}^{(2h)} = \mathbf{x}^{(h)}. \tag{7.22}$$

A simple example in 1D would be the following:

$$\mathbf{I}_{2h}^{h}: \quad \begin{aligned} x_{2i}^{(h)} &= x_i^{(2h)} \\ x_{2i+1}^{(h)} &= \tfrac{1}{2}(x_i^{(2h)} + x_{i+1}^{(2h)}) \end{aligned} \qquad \text{for } 0 \leq i < \tfrac{N}{2} \tag{7.23}$$

Here, every second point is simply injected from the coarse to the fine mesh, and the intermediate points are linearly interpolated from the neighboring points, which here is a simple arithmetic average.

Fine-to-coarse: The converse mapping represents a smoothing operation, or a *restriction* in multigrid-language. We can define the restriction operator as

$$\mathbf{I}_{h}^{2h}\mathbf{x}^{(h)} = \mathbf{x}^{(2h)}, \tag{7.24}$$

which hence takes a vector defined on the fine grid $\Omega^{(h)}$ to one that lives on the coarse grid $\Omega^{(2h)}$. Again, lets give a simple example in 1D:

$$\mathbf{I}_h^{2h}: \quad x_i^{(2h)} = \frac{x_{2i-1}^{(h)} + 2x_{2i}^{(h)} + x_{2i+1}^{(h)}}{4} \quad \text{for } 0 \leq i < \tfrac{N}{2} \tag{7.25}$$

This evidently is a smoothing operation with a simple 3-point stencil.

One usually chooses these two operators such that the transpose of one is proportional to the other, i.e. they are related as follows:

$$\mathbf{I}_h^{2h} = c\,[\mathbf{I}_{2h}^h]^{\mathrm{T}} \tag{7.26}$$

were c is a real number.

In a shorter notation, the above prolongation operator can be written as

$$\text{1D-prolongation, } \mathbf{I}_{2h}^h: \quad \left]\ \frac{1}{2}\ \ 1\ \ \frac{1}{2}\ \right[\tag{7.27}$$

which means that every coarse point is added with these weights to three points of the fine grid. The fine-grid points accessed with weight 1/2 will get contributions from two coarse grid points. Similarly, the restriction operator can be written with the short-hand notation

$$\text{1D-restriction, } \mathbf{I}_h^{2h}: \quad \left[\ \frac{1}{4}\ \ \frac{1}{2}\ \ \frac{1}{4}\ \right] \tag{7.28}$$

This expresses that every coarse grid point is a weighted sum of three fine grid points.

For reference, we also state the corresponding low-order prolongation and restriction operators in 2D:

$$\text{2D-prolongation, } \mathbf{I}_{2h}^h: \quad \left]\ \begin{matrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{matrix}\ \right[\tag{7.29}$$

$$\text{2D-restriction, } \mathbf{I}_h^{2h}: \quad \left[\ \begin{matrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{matrix}\ \right] \tag{7.30}$$

7.4.2 The multigrid V-cycle

The error vector plays an important role in the multigrid approach. It is defined as

$$\mathbf{e} \equiv \mathbf{x}_{\text{exact}} - \tilde{\mathbf{x}}, \tag{7.31}$$

where $\mathbf{x}_{\text{exact}}$ is the exact solution, and $\tilde{\mathbf{x}}$ the (current) approximate solution.

Another important concept is the *residual*, defined as

$$\mathbf{r} \equiv \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}. \tag{7.32}$$

Note that error and residual are solutions of the linear system, i.e. we have

$$\mathbf{A}\mathbf{e} = \mathbf{r}. \tag{7.33}$$

Coarse-grid correction scheme: We now define a function,

$$\tilde{\mathbf{x}}'^{(h)} = \mathrm{CG}(\tilde{\mathbf{x}}^{(h)}, \mathbf{b}^{(h)}), \tag{7.34}$$

that is supposed to return an improved solution for the problem $\mathbf{A}^{(h)}\mathbf{x}^{(h)} = \mathbf{b}^{(h)}$ on grid level h, based on some starting guess $\tilde{\mathbf{x}}^{(h)}$ and a right hand side $\mathbf{b}^{(h)}$. This so-called *coarse grid correction* proceeds along the following steps:

1. Carry out a relaxation step on h (for example by using one Gauss-Seidel or one Jacobi iteration).

2. Compute the residual: $\mathbf{r}^{(h)} = \mathbf{b}^{(h)} - \mathbf{A}^{(h)}\tilde{\mathbf{x}}^{(h)}$.

3. Restrict the residual to a coarser mesh: $\mathbf{r}^{(2h)} = \mathbf{I}_h^{2h}\,\mathbf{r}^{(h)}$.

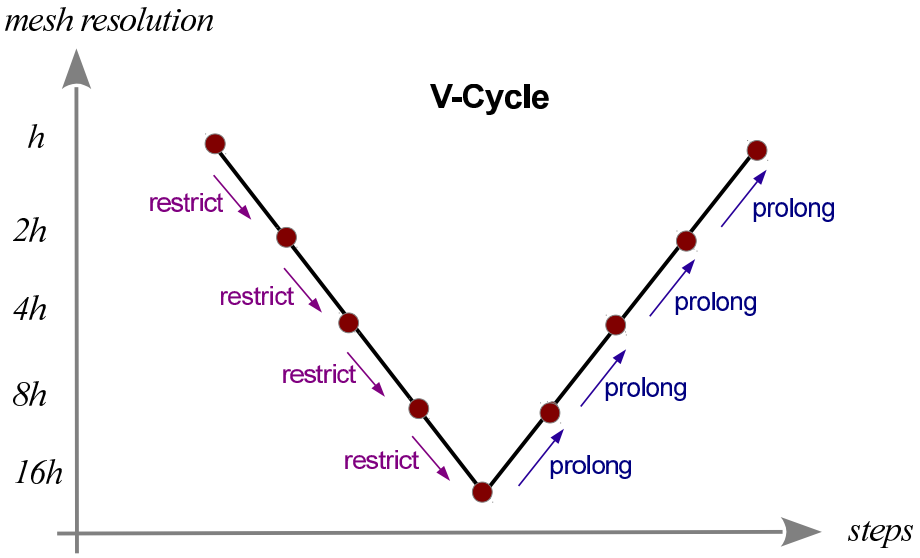
4. Solve $\mathbf{A}^{(2h)}\mathbf{e}^{(2h)} = \mathbf{r}^{(2h)}$ on the coarsened mesh, with $\tilde{\mathbf{e}}^{(2h)} = 0$ as initial guess.

5. Prolong the obtained error $\mathbf{e}^{(2h)}$ to the finer mesh, $\mathbf{e}^{(h)} = \mathbf{I}_{2h}^{h}\,\mathbf{e}^{(2h)}$, and use it to correct the current solution on the fine grid: $\tilde{\mathbf{x}}'^{(h)} = \tilde{\mathbf{x}}^{(h)} + \mathbf{e}^{(h)}$.

6. Carry out a further relaxation step on the fine mesh h.

How do we carry out step 4 in this scheme? We can use recursion! Because what we have to do in step 4 is exactly the function CG(.,.) is defined to do. We however then also need a stopping condition for the recursion, which is simply a prescription that tells us under which conditions we should skip steps 2 to 5 in the above scheme. We can do this by simply saying that further coarsening of the problem should stop once we have reached a minimum number of cells N. At this point we either just do the relaxation steps, or we solve the remaining problem exactly.

V-Cycle: When the coarse grid correction scheme is recursively called, we arrive at the following schematic diagram for how the iteration progresses, which is called a V-cycle:



One finds that the V-cycle rather dramatically speeds up the convergence rate of simple iterative solvers for linear systems of equations. It is easy to show that the computational cost of one V-cycle is of order $\mathcal{O}(N_{\mathrm{grid}})$, where N_{grid} is the number of grid cells on the fine mesh. A convergence to truncation error (i.e. machine precision) requires several V-cycles and involves a computational cost of order $\mathcal{O}(N_{\mathrm{grid}} \log N_{\mathrm{grid}})$. For the Poisson equation, this is the same cost scaling as one gets with FFT-based methods. In practice, good implementations of the two schemes should roughly be equally fast. In cosmology, a multigrid solver is for example used by the MLAPM (Knebe et al., 2001) and RAMSES codes (Teyssier, 2002). An interesting advantage of multigrid is that it requires less data communication when parallelized on distributed memory machines.

One problem we haven't addressed yet is how one finds the operator $\mathbf{A}^{(2h)}$ required on the coarse mesh. The two most commonly used options for this are:

- Direct coarse grid approximation: Here one simply uses the same discrete equations on the coarse grid as on the fine grid, just scaled by the grid resolution h as needed. In this case, the stencil of the matrix does not change.

- Galerkin coarse grid approximation: Here one defines the coarse operator as

$$\mathbf{A}^{(2h)} = \mathbf{I}_h^{2h}\, \mathbf{A}^{(h)}\, \mathbf{I}_{2h}^{h}, \tag{7.35}$$

  which is formally the most consistent way of defining $\mathbf{A}^{(2h)}$, and in this sense optimal. However, computing the matrix in this way can be a bit cumbersome, and it may involve a growing size of the stencil, which then leads to an enlarged computational cost.

7.4.3 The full multigrid method

The V-cycle scheme discussed thus far still relies on an initial guess for the solution, and if this guess is bad, one has to do more V-cycles to reach satisfactory

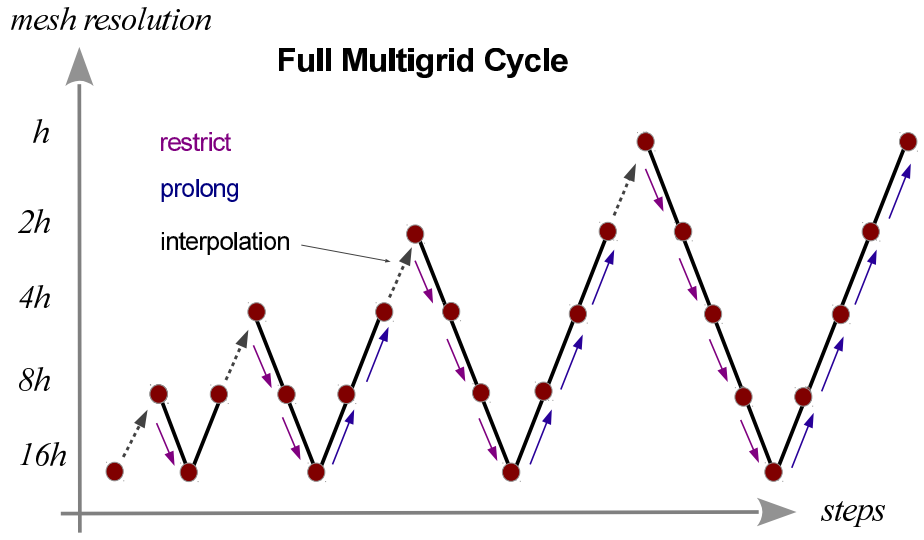
convergence.

This raises the question on how one may get a good guess. If one is dealing with the task of recurrently having to solve the same problem over and over again, with only small changes from solution to solution, as will often be the case in simulation problems, one may be able to simply use the solution from the previous timestep as a guess. In all other cases, one can allude to the following idea: Let's get a good guess by solving the problem on a coarser grid first, and then interpolate the coarse solution to the fine grid as a starting guess.

But at the coarser grid, one is then again confronted with the task to solve the problem without a starting guess. Well, we can then simply recursively apply the idea again, and delegate the finding of a good guess to a yet coarser grid, etc. This then yields the **Full Multigrid Cycle:**

1. Initialize the right hand side on all grid levels, $\mathbf{b}^{(h)}$, $\mathbf{b}^{(2h)}$, $\mathbf{b}^{(4h)}$, ..., $\mathbf{b}^{(H)}$, down to some coarsest level H.
2. Solve the problem (exactly) on the coarsest level H.
3. Given a solution on level i with spacing $2h$, map it to the next level $i+1$ with spacing h and obtain the initial guess $\tilde{\mathbf{x}}^{(h)} = I_{2h}^{h}\,\mathbf{x}^{(2h)}$.
4. Use this starting guess to solve the problem on the level $i+1$ with one V-cycle.
5. Repeat Step 3 until the finest level is reached.

We arrive at the scheme depicted in the sketch.



The computational cost of such a full multigrid cycle is still of order the number of mesh cells, as before.

8 Molecular dynamics simulations

The aim of molecular dynamics (MD) simulations is to model a system in microscopic detail, over a physical length of time relevant for certain properties of interest. The systems that are studied may consist of, for example, individual atoms, molecules, macromolecules, proteins in a solvent, etc.

An important point is that one studies the molecules through *classical equations of motions*, based on an approximate representation of the inter-molecule and/or intra-molecular forces. The corresponding force laws may be empirically derived or in some cases can be motivated by quantum mechanical calculations.

Molecular dynamics simulations are intimately connected to statistical mechanics, and in fact, for interfering macroscopic properties, concepts of statistical mechanics need to be used. This in particular means that the precise *microstate* (given for example in terms of the positions and velocities of all atoms) of an MD simulation is unimportant, instead we are interested in *ensemble averages* of *macroscopic variables*, such as temperature, pressure, diffusion coefficient, etc.

In principle, carrying out a proper ensemble average would mean to average over many different simulations of the system. This is usually impossible. One can however resort to the *ergodic hypothesis* which postulates that the ensemble average is equal to the time average of a specific system. We can hence hope to accurately measure the macroscopic thermodynamic properties of a system by looking at a single realization for a long enough time and average our measurements over this time span.

8.1 Simple interaction potentials

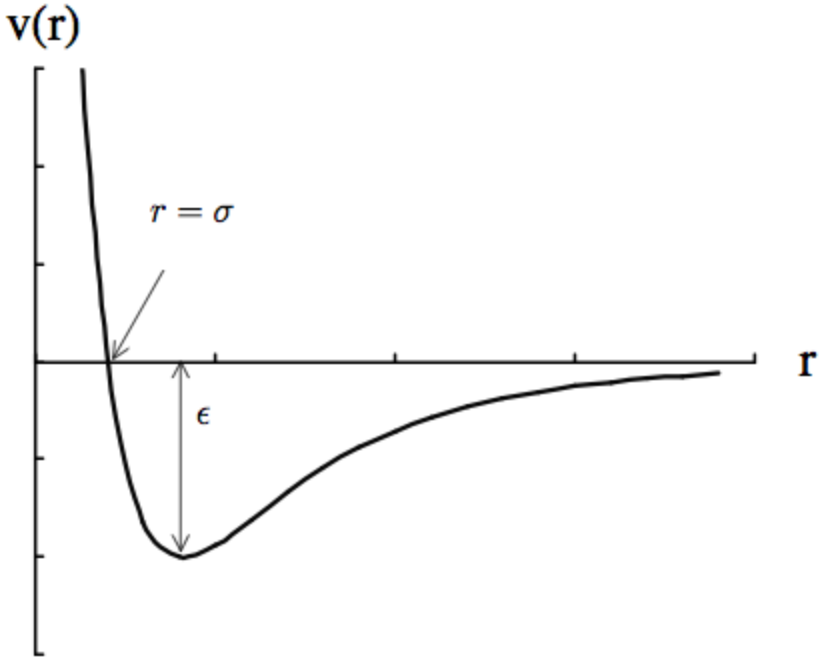
Interatomic interactions are usually quite weak compared to chemical bonds. A good first approximation is often to assume that they are central and pair-wise additive. Then the total potential of the system can be written as

$$V(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \frac{1}{2} \sum_{i \neq j}^{N} v(|\mathbf{r}_i - \mathbf{r}_j|) \tag{8.1}$$

where $v(r)$ is the pair-wise interaction potential. We hence arrive at a conservative Hamiltonian system,

$$H = \sum_{i=1}^{N} \frac{\mathbf{p}_i^2}{2m_i} + V(\mathbf{r}_1, \ldots, \mathbf{r}_N), \tag{8.2}$$

for which we can readily derive equations of motion in the standard way. Once the equations of motion are written down, we can integrate them as an ordinary N-body system, using, for example, the Leapfrog or Verlet integration schemes. While these integration schemes are of low-order, recall that they are symplectic, hence they have particularly good stability properties for long-term integrations of conservative systems.



A potential often used to approximately describe the interaction of molecules or atoms is the 12-6 Lennard-Jones potential:

$$v(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]. \tag{8.3}$$

Here ϵ characterizes the interaction strength, and σ the range. There is a minimum of the potential at $r_0 = 2^{1/6}\sigma \simeq 1.12\,\sigma$, with $v(r_0) = -\epsilon$. For distances below r_0, the force is (strongly) repulsive, for larger distances it is attractive and approaches zero quite quickly.

For example, for argon suitable parameters to describe the potential are $\epsilon/k_{\mathrm{B}} = 120\,\mathrm{K}$ and $\sigma = 3.4 \times 10^{-8}\,\mathrm{cm}$. This implies a characteristic timescale $\tau \sim 2 \times 10^{-12}\,\mathrm{s}$. Typical simulations in molecular dynamics will do perhaps $10^4 - 10^5$ steps or so, hence they cover a timespan of the order of nanoseconds. Quite short indeed!

We note that sometimes also long-range electrostatic forces are important in MD simulations, which are then described by solutions of the Poisson equation. For obtaining the forces in this case, the same techniques as in the gravitational dynamics can be applied.

8.2 Statistical mechanics aspects

Usually, MD simulations are first evolved to reach a certain dynamical equilibrium state in which any memory of the initial conditions has been completely erased. One is then often interested in time averages $\overline{A}$ of some macroscopic quantity $A(\Gamma)$, which itself can be calculated in terms of the microstate Γ of the system. Formally, we are interested in the quantity

$$\overline{A} = \lim_{\tau\to\infty} \frac{1}{\tau} \int_{t_0}^{t_0+\tau} A(\Gamma(t))\,\mathrm{d}t. \tag{8.4}$$

In a typical MD simulation, this average is simply carried out as an average over a finite number of timesteps:

$$\overline{A} \simeq \frac{1}{N_{\text{steps}}} \sum_{k=1}^{N_{\text{steps}}} A(\Gamma_k) \tag{8.5}$$

8.2.1 Temperature adjustment

If we define the instantaneous kinetic temperature $\mathcal{T}$ of our N-body MD-system as

$$\frac{3}{2} k_{\mathrm{B}} \mathcal{T} = \left\langle \frac{m\mathbf{v}_i^2}{2} \right\rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{m\mathbf{v}_i^2}{2}, \tag{8.6}$$

then $\mathcal{T}$ averaged over many microstates produced by a MD simulation corresponds to the temperature of the system, i.e.

$$T = \frac{1}{N_{\text{steps}}} \sum_{k} \mathcal{T}(\Gamma_k) \tag{8.7}$$

How do we control the temperature of our MD system? This is of central importance as the temperature is often a key thermodynamic control variable of a MD system. If the current (instantaneous) temperature $\mathcal{T}$ is very different from the desired target temperature T, we can rescale all velocities according to

$$\mathbf{v}_i' = \sqrt{\frac{T}{\mathcal{T}}} \mathbf{v}_i. \tag{8.8}$$

Note that this will not automatically lead to a thermodynamic equilibrium distribution of the velocities, where for each Cartesian component we expect a Gaussian, i.e.

$$p(v_x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{v_x^2}{2\sigma^2}\right) \tag{8.9}$$

with $\sigma^2 = k_{\mathrm{B}}T/m$. To speed up the approach of a new thermodynamic equilibrium and hence a Gaussian, one may also sample from a Gaussian with the desired temperature, especially when initializing the particles in the beginning.

Note that specifying the number density N/V combined with the temperature also specifies the pressure $P = (N/V)k_{\mathrm{B}}T$. This already fully determines the thermodynamic state of a pure liquid. Of course, to get as close as possible to the thermodynamic limit, one needs to try to make N as large as possible, limited only by the computational cost.

Also, one should aim to run sufficiently long to allow proper statistical sampling of all possible particle trajectories, which is needed to justify the use of the ergodic theorem. Depending on what one wants to measure, the required time interval can be quite different. For example, measuring a diffusion constant will require more simulation time than just measuring the average temperature.

In many molecular dynamics studies one follows the classical equations of motion for system of given particle number N, volume V and total energy E, which corresponds to the *micro-canonical ensemble*. Sometimes this is also called the 'NVE-Ensemble', after the quantities that are kept constant. The averages one computes are hence micro-canonical ensemble averages. Here, every microstate (which all have the same energy, by definition) is accessible with equal probability.

However, experimentally much more accessible are systems in which the temperature is constant, not the energy. These 'NVT-Ensembles' represent the *canonical ensemble*. In the canonical ensemble, each microstate of energy E has probability

$$p = \frac{1}{Z}\,\mathrm{e}^{-\frac{E}{k_{\mathrm{B}}T}}, \tag{8.10}$$

of occurring, and the expectation value of a macroscopic variable A is given by

$$\langle A\rangle = \frac{1}{Z}\int \mathrm{d}\Gamma\, A(\Gamma)\exp\left(-\frac{E(\Gamma)}{k_{\mathrm{B}}T}\right), \tag{8.11}$$

which is the classical Boltzmann-Gibbs average. The partition function

$$Z = \int \mathrm{d}\Gamma\, \exp\left(-\frac{E}{k_{\mathrm{B}}T}\right) \tag{8.12}$$

acts essentially as a normalization factor.

In order to realize a constant temperature in a MD simulation, one can for example rescale the kinetic temperature every timestep to the target temperature, yielding *isokinetic simulations*. This leaves however some residual distortions in the distribution functions, which do not exactly reflect thermodynamic equilibrium when this procedure is applied. This can be mitigated by the use of the Nose-Hoover 'thermostat'. Here one adds a thermostat variable as an additional degree of freedom, and uses it to drive the velocities to the desired temperature through a suitable friction/antifriction term in the equations of motion. For example, we may use:

$$m_i\dot{\mathbf{v}}_i = \mathbf{F}_i - 2\alpha\mathbf{v}_i \tag{8.13}$$

$$\frac{\mathrm{d}\alpha}{\mathrm{d}t} = \frac{1}{\tau_E}\left(\sum_i \frac{1}{2}m_i\mathbf{v}_i^2 - \frac{3}{2}Nk_{\mathrm{B}}T\right) \tag{8.14}$$

where τ_E is a parameter that controls the timescale over which the temperature is regulated and α is the thermostat variable.

Molecular dynamics simulations can also be used to approximate other types of ensembles. Of particular importance are the *grand canonical ensemble* (μVT-Ensemble) in which the chemical potential μ instead of the particle number is held constant. Also important is the NPT-Ensemble, in which the volume of the system is adjusted (through rescaling the coordinates) such that the pressure of the system is kept constant.

8.3 Practical aspects

8.3.1 Initial conditions and boundary conditions

One usually adopts periodic boundary conditions, because this largely eliminates surface effects in a simple way. Note that because simulated MD systems only contain particle numbers that are very much smaller than those in any macroscopic sample, surface effects would easily spoil any attempt to approximate the continuum limit if they cannot be efficiently suppressed.

The initial conditions do not require particular care as their details should anyway be quickly forgotten in a proper MD simulation as it transitions to equilibrium. One usually sets up the initial particle positions on a simple regular grid. To speed up the settling to thermodynamic equilibrium, it is helpful to draw random initial velocities from an appropriate Maxwellian distribution.

8.3.2 Finite range interactions

Formally, calculating all the pairwise forces for N molecules is a $\mathcal{O}(N^2)$ problem, which would rather seriously limit the system sizes that can be studied. However, at $r = 3\sigma$, the Lennard-Jones potential has dropped already to about $v(r) \simeq -0.005\epsilon$, and this low binding energy indicates the weak forces that act at distances this large and beyond. It is hence clear that forces at large distances become negligible, and the full N^2-interactions do not all have to be calculated. One therefore commonly introduces a cut-off radius into the potential, e.g. in the form of a hard cut-off:

$$v_c(r) = \begin{cases} v(r) & \text{for } r \leq r_c, \\ 0 & \text{otherwise.} \end{cases} \tag{8.15}$$

As an alternative to a hard cut-off at r_c one may also modify the potential such that it smoothly drop to zero, for example in the form

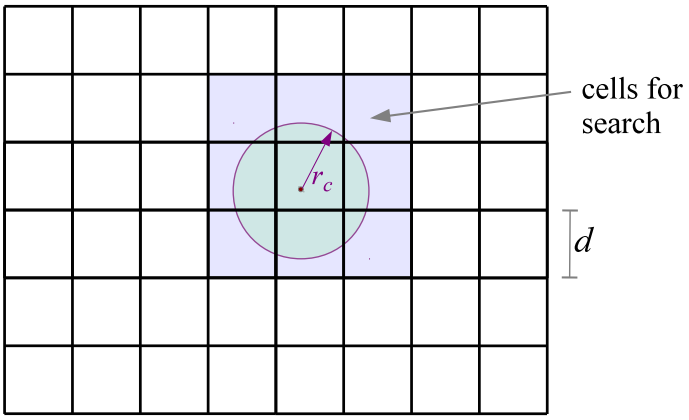
$$v_c(r) = \begin{cases} v(r) - v(r_c) - v'(r_c)(r - r_c) & \text{for } r \leq r_c, \\ 0 & \text{otherwise.} \end{cases} \tag{8.16}$$

Note that the latter form modifies the potential everywhere, which can introduce a small bias in the results.

But independently of how exactly it is carried out, the most important feature of the cut-off is that it reduces the force calculation problem from an unwieldy $\mathcal{O}(N^2)$ to one of order $\mathcal{O}(N)$, provided one has a fast (ideally zeroth-order) method of finding the neighbors that are actually participating in the interactions. Two methods can be used for this:

Search grids

The simplest approach is to use a Cartesian search grid with cell size $d \geq r_c$ overlaid over the system. In a first step, one bins all the particles onto this grid, using for example link-lists as a book-keeping device to organize lists of particles contained in every single cell.



Then, in the calculation of the forces for a given particle, one only considers the particles in the same search grid cell as the target particle, as well as the 26 surrounding cells (in three dimensions). Since we have $d \geq r_c$, this search will already guarantee that we definitely find all interacting neighbors with distance up to r_c, independent of where our target point lies with respect to the target cell.

We note that a search grid with $d = r_c$ still leads to many in principal superfluous look-ups of potential interaction neighbors. This is because one effectively checks all particles in a volume of $(3d)^3 = (3r_c)^3$, while contributing to the interactions is only the spherical volume $(4\pi/3)r_c^3$ around the target particle. This sphere covers hence only 15.5% of the region that is actually searched. Assuming a roughly uniform distribution of the particles, one then needs to carry out ~ 6 times more distance computations than really used later for the interactions. By using a somewhat finer search grid, one can reduce this overhead somewhat and increase the efficiency of the neighbor search further.

Range search with a tree

A generalization of the search grid idea is to use a *search tree*. Here one considers a hierarchical grid in which the cell size from level to level differs by a factor of 2.

This is simply the oct-tree we considered in the gravity calculation with a 'tree-code', except that we do not need the multipole moments here.

The interaction neighbors out to a distance r_c can then simply be found by performing a special tree walk: A tree node is opened when it has a geometric overlap with the search region (which is a sphere or box of size r_c around the target position), otherwise the walk across the branch is terminated. An advantage of this method is that it works well also for variable r_c; in fact, the search region is allowed to be widely different from particle to particle, but the method always works with nearly constant efficiency. In most MD simulations, the particle density is fairly constant, and r_c has a global value, too, hence search trees are not needed. However, in smoothed particle hydrodynamics (SPH), this is different. Here one needs, just like in MD, interaction neighbors out to certain distance, but these distances may vary widely in space and time.

8.3.3 Time integration

A very popular scheme for MD simulations is the so-called Verlet time integrator. This is given by

$$\mathbf{r}_i^{(n+1)} = 2\,\mathbf{r}_i^{(n)} - \mathbf{r}_i^{(n-1)} + \mathbf{a}_i^{(n)}(\Delta t)^2. \tag{8.17}$$

Note that this can in principle work without storing the velocities explicitly, but one needs two copies of the old positions. If one also wants to know the velocities, these are computed in this scheme from

$$\mathbf{v}_i^{(n)} = \frac{1}{2\Delta t}\left(\mathbf{r}_i^{(n+1)} - \mathbf{r}_i^{(n-1)}\right). \tag{8.18}$$

The Verlet scheme may look a bit strange at first, but actually it is really *identical* to the ordinary Leapfrog. The latter can be written as

$$\mathbf{v}_i^{(n+1/2)} = \mathbf{v}_i^{(n)} + \mathbf{a}_i^{(n)}\frac{\Delta t}{2}, \tag{8.19}$$

$$\mathbf{r}_i^{(n+1)} = \mathbf{r}_i^{(n)} + \mathbf{v}_i^{(n+1/2)}\Delta t, \tag{8.20}$$

$$\mathbf{v}_i^{(n+1)} = \mathbf{v}_i^{(n+1/2)} + \mathbf{a}_i^{(n+1)}\frac{\Delta t}{2}. \tag{8.21}$$

If we now plug in equation (8.19) into equation (8.20), consider the same equation a second time for index n instead of $n+1$, and subtract the two from each other, we obtain the Verlet scheme (8.17).

9 Basic gas dynamics

Gravity is the dominant driver behind cosmic structure formation (e.g. Mo et al., 2010), but at small scales hydrodynamics in the baryonic components becomes very important, too. In this section we very briefly review the basic equations and some prominent phenomena related to gas dynamics in order to make the discussion of the numerical fluid solvers used in galaxy evolution more accessible. For a detailed introduction to hydrodynamics, the reader is referred to the standard textbooks on this subject (e.g. Landau & Lifshitz, 1959; Shu, 1992).

9.1 Euler and Navier-Stokes equations

The gas flows in astrophysics are often of extremely low density, making internal friction in the gas extremely small. In the limit of assuming internal friction to be completely absent, we arrive at the so-called ideal gas dynamics as described by the Euler equations. Most calculations in cosmology and galaxy formation are carried out under this assumption. However, in certain regimes, viscosity may still become important (for example in the very hot plasma of rich galaxy clusters), hence we shall also briefly discuss the hydrodynamical equations in the presence of physical viscosity, the Navier-Stokes equations, which in a sense describe *real* fluids as opposed to ideal ones. Phenomena such as fluid instabilities or turbulence are also best understood if one does not neglect viscosity completely.

9.1.1 Euler equations

If internal friction in a gas flow can be neglected, the dynamics of the fluid is governed by the Euler equations:

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0, \tag{9.1}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla(\rho \mathbf{v}\mathbf{v}^T + P) = 0, \tag{9.2}$$

$$\frac{\partial}{\partial t}(\rho e) + \nabla[(\rho e + P)\mathbf{v}] = 0, \tag{9.3}$$

where $e = u + \mathbf{v}^2/2$ is the total energy per unit mass, and u is the thermal energy per unit mass. Each of these equations is a continuity law, one for the mass, one for the momentum, and one for the total energy. The equations hence form a set of hyperbolic conservation laws. In the form given above, they are not yet complete,

however. One still needs a further expression that gives the pressure in terms of the other thermodynamic variables. For an ideal gas, the pressure law is

$$P = (\gamma - 1)\rho u, \tag{9.4}$$

where $\gamma = c_p/c_v$ is the ratio of specific heats. For a monoatomic gas, we have $\gamma = 5/3$.

9.1.2 Navier-Stokes equations

Real fluids have internal stresses, due to *viscosity*. The effect of viscosity is to dissipate relative motions of the fluid into heat. The Navier-Stokes equations are then given by

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{v}) = 0, \tag{9.5}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{v}) + \nabla(\rho \mathbf{v}\mathbf{v}^T + P) = \nabla \Pi, \tag{9.6}$$

$$\frac{\partial}{\partial t}(\rho e) + \nabla[(\rho e + P)\mathbf{v}] = \nabla(\Pi \mathbf{v}). \tag{9.7}$$

Here Π is the so-called viscous stress tensor, which is a material property. For $\Pi = 0$, the Euler equations are recovered. To first order, the viscous stress tensor must be a linear function of the velocity derivatives (Landau & Lifshitz, 1959). The most general tensor of rank-2 of this type can be written as

$$\Pi = \eta \left[\nabla \mathbf{v} + (\nabla \mathbf{v})^T - \frac{2}{3}(\nabla \cdot \mathbf{v})\mathbf{1} \right] + \xi(\nabla \cdot \mathbf{v})\mathbf{1}, \tag{9.8}$$

where $\mathbf{1}$ is the unit matrix. Here η scales the traceless part of the tensor and describes the shear viscosity. ξ gives the strength of the diagonal part, and is the so-called bulk viscosity. Note that η and ξ can in principle be functions of local fluid properties, such as ρ, T, etc.

Incompressible fluids

In the following we shall assume constant viscosity coefficients. Also, we specialize to incompressible fluids with $\nabla \cdot \mathbf{v} = 0$, which is a particularly important case in practice. Let's see how the Navier-Stokes equations simplify in this case. Obviously, ξ is then unimportant and we only need to deal with shear viscosity. Now, let us consider one of the components of the viscous shear force described by equation (9.6):

$$\begin{aligned} \frac{1}{\eta}(\nabla \Pi)_x &= \frac{\partial}{\partial x}\left(2\frac{\partial v_x}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}\right) + \frac{\partial}{\partial z}\left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}\right) \\ &= \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) v_x = \nabla^2 v_x, \end{aligned} \tag{9.9}$$

where we made use of the $\nabla \cdot \mathbf{v} = 0$ constraint. If we furthermore introduce the *kinematic viscosity* ν as

$$\nu \equiv \frac{\eta}{\rho}, \tag{9.10}$$

we can write the equivalent of equation (9.6) in the compact form

$$\frac{\mathrm{D}\,\mathbf{v}}{\mathrm{D}\,t} = -\frac{\nabla P}{\rho} + \nu \nabla^2 \mathbf{v}, \tag{9.11}$$

where the derivative on the left-hand side is the Lagrangian derivative,

$$\frac{\mathrm{D}}{\mathrm{D}\,t} = \frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla. \tag{9.12}$$

We hence see that the motion of individual fluid elements responds to pressure gradients and to viscous forces. The form (9.11) of the equation is also often simply referred to as the Navier-Stokes equation.

9.1.3 Scaling properties of viscous flows

Consider the Navier-Stokes equations for some flow problem that is characterized by some characteristic length L_0, velocity V_0, and density scale ρ_0. We can then define dimensionless fluid variables of the form

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{V_0}, \qquad \hat{\mathbf{x}} = \frac{\mathbf{x}}{L_0}, \qquad \hat{P} = \frac{P}{\rho_0 V_0^2}. \tag{9.13}$$

Similarly, we define a dimensionless time, a dimensionless density, and a dimensionless Nabla operator:

$$\hat{t} = \frac{t}{L_0/V_0}, \qquad \hat{\rho} = \frac{\rho}{\rho_0}, \qquad \hat{\nabla} = L_0 \nabla. \tag{9.14}$$

Inserting these definitions into the Navier-Stokes equation (9.11), we obtain the dimensionless equation

$$\frac{D\hat{\mathbf{v}}}{D\hat{t}} = -\frac{\hat{\nabla}\hat{P}}{\hat{\rho}} + \frac{\nu}{L_0 V_0}\,\hat{\nabla}^2 \hat{\mathbf{v}}. \tag{9.15}$$

Interestingly, this equation involves one number,

$$\mathrm{Re} \equiv \frac{L_0 V_0}{\nu}, \tag{9.16}$$

which characterizes the flow and determines the structure of the possible solutions of the equation. This is the so-called Reynolds number. Problems which have similar Reynolds number are expected to exhibit very similar fluid behavior. One then has *Reynolds-number similarity*. In contrast, the Euler equations ($\mathrm{Re} \to \infty$) exhibit always scale similarity because they are invariant under scale transformations.

One intuitive interpretation one can give the Reynolds number is that it measures the importance of inertia relative to viscous forces. Hence:

$$\mathrm{Re} \approx \frac{\text{inertial forces}}{\text{viscous forces}} \approx \frac{D\mathbf{v}/Dt}{\nu\nabla^2\mathbf{v}} \approx \frac{V_0/(L_0/V_0)}{\nu V_0/L_0^2} = \frac{L_0 V_0}{\nu}. \tag{9.17}$$

If we have $\mathrm{Re} \sim 1$, we are completely dominated by viscosity. On the other hand, for $\mathrm{Re} \to \infty$ viscosity becomes unimportant and we approach an ideal gas.

9.2 Shocks

An important feature of hydrodynamical flows is that they can develop shock waves in which the density, velocity, temperature and specific entropy jump by finite amounts (e.g. Toro, 1997). In the case of the Euler equations, such shocks are true mathematical discontinuities. Interestingly, shocks can occur even from perfectly smooth initial conditions, which is a typical feature of hyperbolic partial differential equations. In fact, acoustic waves with sufficiently large amplitude will suffer from wave-steeping (because the slightly hotter wave crests travel faster than the colder troughs), leading eventually to shocks. Of larger practical importance in astrophysics are however the shocks that occur when flows collide supersonically; here kinetic energy is irreversibly transferred into thermal energy, a process that also manifests itself with an increase in entropy.

In the limit of vanishing viscosity (i.e. for the Euler equations), the differential form of the fluid equations breaks down at the discontinuity of a shock, but the integral form (the *weak formulation*) remains valid. In other words this means that the flux of mass, momentum and energy must remain continuous at a shock front. Assuming that the shock connects two piecewise constant states, this leads to the Rankine-Hugoniot jump conditions (Rankine, 1870). If we select a frame of reference where the shock is stationary ($v_s = 0$) and denote the pre-shock state with (v_1, P_1, ρ_1), and the post-shock state as (v_2, P_2, ρ_2) (hence $v_1, v_2 > 0$), we have

$$\rho_1 v_2 = \rho_2 v_2, \tag{9.18}$$

$$\rho_1 v_1^2 + P_1 = \rho_2 v_2^2 + P_2, \tag{9.19}$$

$$(\rho_1 e_1 + P_1) v_1 = (\rho_2 e_2 + P_2) v_2. \tag{9.20}$$

For an ideal gas, the presence of a shock requires that the pre-shock gas streams supersonically into the discontinuity, i.e. $v_1 > c_1$, where $c_1^2 = \gamma P_1/\rho_1$ is the sound speed in the pre-shock phase. The Mach number

$$\mathcal{M} = \frac{v_1}{c_1} \tag{9.21}$$

measures the strength of the shock ($\mathcal{M} > 1$). The shock itself decelerates the fluid and compresses it, so that we have $v_2 < v_1$ and $\rho_2 > \rho_1$. It also heats it up, so

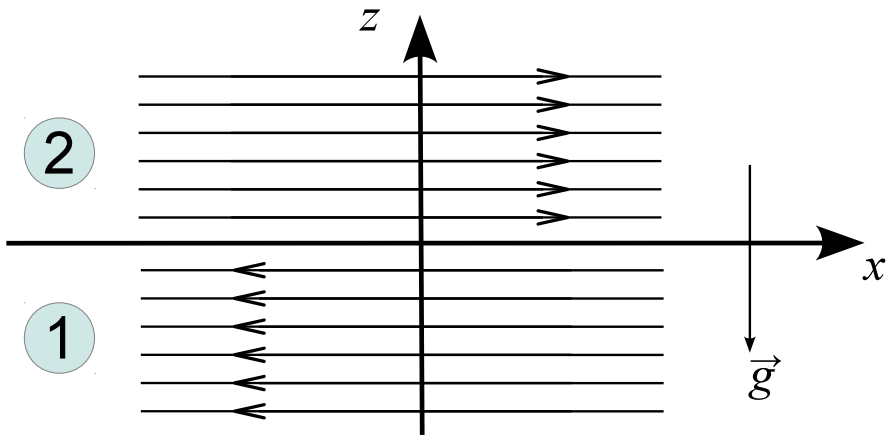


Figure 9.1: Geometry of a generic shear flow.

that $T_2 > T_1$, and makes the postshock flow subsonic, with $v_2/c_2 < 1$. Manipulating equations (9.18) to (9.20), we can express the relative jumps in the thermodynamic quantities (density, temperature, entropy, etc.) through the Mach number alone, for example:

$$\frac{\rho_2}{\rho_1} = \frac{(\gamma+1)\mathcal{M}^2}{(\gamma-1)\mathcal{M}^2+2}. \tag{9.22}$$

9.3 Fluid instabilities

In many situations, gaseous flows can be subject to fluid instabilities in which small perturbations can rapidly grow, thereby tapping a source of free energy. An important example of this are Kelvin-Helmholtz and Rayleigh-Taylor instabilities, which we briefly discuss in this section.

9.3.1 Stability of a shear flow

We consider a flow in the x-direction, which in the lower half-space $z < 0$ has velocity U_1 and density ρ_1, whereas in the upper half-space the gas streams with U_2 and has density ρ_2. In addition there can be a homogeneous gravitational field $\mathbf{g}$ pointing into the negative z-direction, as sketched in Figure 9.1.

The stability of the flow can be analysed through perturbation theory. To this end, one can for example treat the flow as an incompressible potential flow, and carry out an Eigenmode analysis in Fourier space. With the help of Bernoulli's theorem one can then derive an equation for a function $\xi(x,t) = z$ that describes the z-location of the interface between the two phases of the fluid. Details of this calculation can for example be found in Pringle & King (2007). For a single perturbative Fourier mode

$$\xi \;=\; \hat{\xi}\exp[i(kx-\omega t)], \tag{9.23}$$

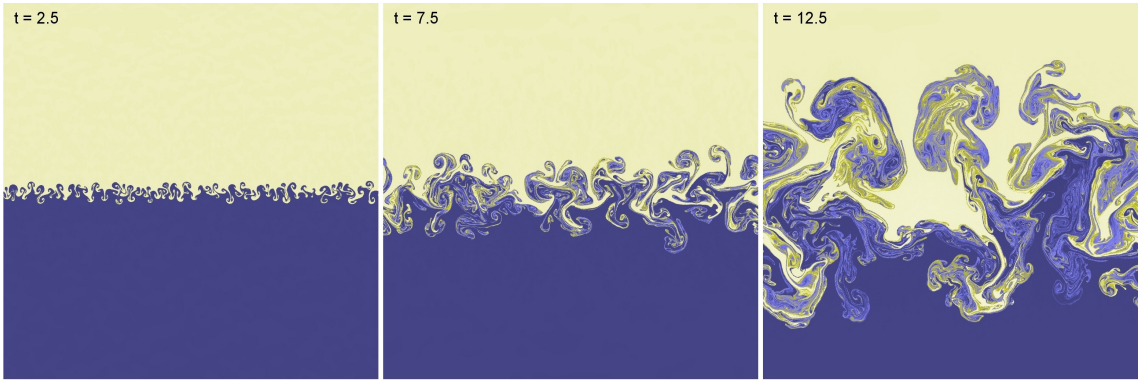


Figure 9.2: A growing Rayleigh-Taylor instability in which a lighter fluid (blue) is covered by a heaver fluid (yellow).

one then finds that non-trivial solutions with $\hat{\xi} \neq 0$ are possible for

$$\omega^2(\rho_1 + \rho 2) - 2\omega k(\rho_1 U_1 + \rho_2 U_2) + k^2(\rho_1 U_1^2 + \rho_2 U_2^2) + (\rho_2 - \rho 1)kg = 0, \qquad (9.24)$$

which is the *dispersion relation*. Unstable, exponentially growing mode solutions appear if there are solutions for ω with positive imaginary part. Below, we examine the dispersion relation for a few special cases.

Rayleigh-Taylor instability

Let us consider the case of a fluid at rest, $U_1 = U_2 = 0$. The dispersion relation simplifies to

$$\omega^2 = \frac{(\rho_1 - \rho_2)kg}{\rho_1 + \rho_2}. \qquad (9.25)$$

We see that for $\rho_2 > \rho_1$, i.e. the denser fluid lies on top, unstable solutions with $\omega^2 < 0$ exist. This is the so-called Rayleigh-Taylor instability. It is in essence buoyancy driven and leads to the rise of lighter material underneath heavier fluid in a stratified atmosphere, as illustrated in the simulation shown in Figure 9.2. The free energy that is tapped here is the potential energy in the gravitational field. Also notice that for an ideal gas, arbitrarily small wavelengths are unstable, and those modes will grow fastest. If on the other hand we have $\rho_1 > \rho_2$, then the interface is stable and will only oscillate when perturbed.

Kelvin-Helmholtz instability

If we set the gravitational field to zero, $g = 0$, we have the situation of a pure shear flow. In this case, the solutions of the dispersion relation are given by

$$\omega_{1/2} = \frac{k(\rho_1 U_1 + \rho_2 U_2)}{\rho_1 + \rho_2} \pm ik\frac{\sqrt{\rho_1 \rho_2}}{\rho_1 + \rho_2}|U_1 - U_2|. \qquad (9.26)$$

Interestingly, in an ideal gas there is an imaginary growing mode component for every $|U_1 - U_2| > 0$! This means that a small wave-like perturbation at an interface will

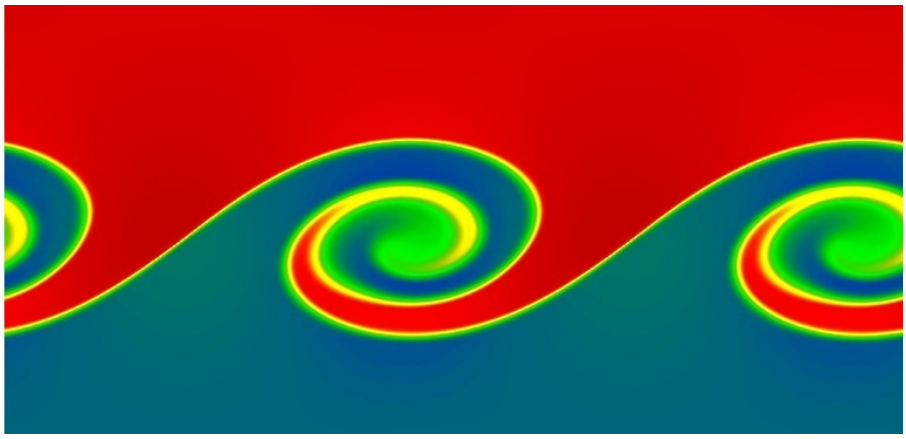


Figure 9.3: Characteristic Kelvin-Helmholtz billows arising in a shear flow.

grow rapidly into large waves that take the form of characteristic Kelvin-Helmholtz "billows". In the non-linear regime reached during the subsequent evolution of this instability the waves are rolled up, leading to the creation of vortex like structures, as seen in Figure 9.3. As the instability grows fastest for small scales (high k), the billows tend to get larger and larger with time.

Because the Kelvin-Helmholtz instability basically means that any sharp velocity gradient in a shear flow is unstable in a freely streaming fluid, this instability is particularly important for the creation of fluid turbulence. Under certain conditions, some modes can however be stabilized against the instability. This happens for example if we consider shearing with $U_1 \neq U_2$ in a gravitational field $g > 0$. Then the dispersion relation has the solutions

$$\omega = \frac{k(\rho_1 U_1 + \rho_2 U_2)}{\rho_1 + \rho_2} \pm \frac{\sqrt{-k^2 \rho_1 \rho_2 (U_1 - U_2)^2 - (\rho_1 + \rho_2)(\rho_2 - \rho_1) k g}}{\rho_1 + \rho_2}. \tag{9.27}$$

Stability is possible if two conditions are met. First, we need $\rho_1 > \rho_2$, i.e. the lighter fluid needs to be on top (otherwise we would have in any case a Rayleigh-Taylor instability). Second, the condition

$$(U_1 - U_2)^2 < \frac{(\rho_1 + \rho_2)(\rho_1 - \rho_2) g}{k \rho_1 \rho_2} \tag{9.28}$$

must be fulfilled. Compared to the ordinary Kelvin-Holmholtz instability without a gravitational field, we hence see that sufficiently small wavelengths are stabilized below a threshold wavelength. The larger the shear becomes, the further this threshold moves to small scales.

The Rayleigh-Taylor and Kelvin-Helmholtz instabilities are by no means the only fluid instabilities that can occur in an ideal gas (Pringle & King, 2007). For example, there is also the Richtmyer-Meshov instability, which can occur when an interface is suddenly accelerated, for example due to the passage of a shock wave. In self-gravitating gases, there is the Jeans instability, which occurs when the internal gas

pressure is not strong enough to prevent a positive density perturbation from growing and collapsing under its own gravitational attraction. This type of instability is particularly important in cosmic structure growth and star formation. If the gas dynamics is coupled to external sources of heat (e.g. through a radiation field), a number of further instabilities are possible. For example, a thermal instability (Field, 1965) can occur when a radiative cooling function has a negative dependence on temperature. If the temperature drops somewhere a bit more through cooling than elsewhere, the cooling rate of this cooler patch will increase such that it is cooling even faster. In this way, cool clouds can drop out of the background gas.

9.4 Turbulence

Fluid flow which is unsteady, irregular, seemingly random, and chaotic is called *turbulent* (Pope, 2000). Familiar examples of such situations include the smoke from a chimney, a waterfall, or the wind field behind a fast car or airplane. The characteristic feature of turbulence is that the fluid velocity varies significantly and irregularly both in position and time. As a result, turbulence is a statistical phenomenon and is best described with statistical techniques.

If the turbulent motions are subsonic, the flow can often be approximately treated as being incompressible, even for an equation of state that is not particularly stiff. Then only solenoidal motions that are divergence free can occur, or in other words, only shear flows are present. We have already seen that such flows are subject to fluid instabilities such as the Kelvin-Helmholtz instability, which can easily produce swirling motions on many different scales. Such vortex-like motions, also called *eddies*, are the conceptual building blocks of Kolmogorov's theory of incompressible turbulence (Kolmogorov, 1941), which yields a surprisingly accurate description of the basic phenomenology of turbulence, even though many aspects of turbulence are still not fully understood.

9.4.1 Kolmogorov's theory of incompressible turbulence

We consider a fully turbulent flow with characteristic velocity U_0 and length scale L_0. We assume that a quasi-stationary state for the turbulence is achieved by some kind of driving process on large scales, which in a time-averaged way injects an energy ϵ per unit mass. We shall also assume that the Reynolds number Re is large. We further imagine that the turbulent flow can be considered to be composed of eddies of different size l, with characteristic velocity $u(l)$, and associated timescale $\tau(l) = l/u(l)$.

For the largest eddies, $l \sim L_0$ and $u(l) \sim U_0$, hence viscosity is unimportant for them. But large eddies are unstable und break up, transferring their energy to somewhat smaller eddies. This continues to yet smaller scales, until

$$\mathrm{Re}(l) = \frac{l u(l)}{\nu} \tag{9.29}$$

becomes of the order of unity, where ν is the kinematic viscosity. For these eddies, viscosity will be very important so that their kinetic energy is dissipated away. We will see that this transfer of energy to smaller scales gives rise to the *energy cascade* of turbulence. But several important questions are still unanswered:

1. What is the actual size of the smallest eddies that dissipate the energy?

2. How do the velocities $u(l)$ of the eddies vary with l when the eddies become smaller?

Kolmogorov's hypotheses

Kolmogorov conjectured a number of hypotheses that can answer these questions. In particular, he proposed:

- For high Reynolds number, the small-scale turbulent motions ($l \ll L_0$) become statistically isotropic. Any memory of large-scale boundary conditions and the original creation of the turbulence on large scales is lost.

- For high Reynolds number, the statistics of small-scale turbulent motions has a universal form and is only determined by ν and the energy injection rate per unit mass, ϵ.

From ν and ϵ, one can construct characteristic Kolmogorov length, velocity and timescales. Of particular importance is the *Kolmogorov length*:

$$\eta \equiv \left(\frac{\nu^3}{\epsilon}\right)^{1/4}. \tag{9.30}$$

Velocity and timescales are given by

$$u_\eta = (\epsilon\nu)^{1/4}, \qquad \tau_\eta = \left(\frac{\nu}{\epsilon}\right)^{1/2}. \tag{9.31}$$

We then see that the Reynolds number at the Kolmogorov scale is

$$\mathrm{Re}(\eta) = \frac{\eta u_\eta}{\nu} = 1, \tag{9.32}$$

showing that they describe the dissipation range. Kolmogorov has furthermore made a second similarity hypothesis, as follows:

- For high Reynolds number, there is a range of scales $L_0 \gg l \gg \eta$ over which the statistics of the motions on scale l take a universal form, and this form is *only* determined by ϵ, *independent* of ν.

In other words, this also means that viscous effects are unimportant over this range of scales, which is called the *inertial range*. Given an eddy size l in the inertial range, one can construct its characteristic velocity and timescale just from l and ϵ:

$$u(l) = (\epsilon l)^{1/3}, \qquad \tau(l) = \left(\frac{l^2}{\epsilon}\right)^{1/3}. \tag{9.33}$$

One further consequence of the existence of the inertial range is that here the energy transfer rate

$$T(l) \sim \frac{u^2(l)}{\tau(l)} \tag{9.34}$$

of eddies to smaller scales is expected to be scale-invariant. Indeed, putting in the expected characteristic scale dependence we get $T(l) \sim \epsilon$, i.e. $T(l)$ is equal to the energy injection rate. This also implies that we have

$$\epsilon \sim \frac{U_0^3}{L_0}. \tag{9.35}$$

With this result we can also work out what we expect for the ratio between the characteristic quantities of the largest and smallest scales:

$$\frac{\eta}{L_0} \sim \left(\frac{\nu^3}{\epsilon L_0^4}\right)^{1/4} = \left(\frac{\nu^3}{U_0^3 L_0^3}\right)^{1/4} = \mathrm{Re}^{-\frac{3}{4}}, \tag{9.36}$$

$$\frac{u_\eta}{U_0} \sim \left(\frac{\epsilon\nu}{U_0^4}\right)^{1/4} = \left(\frac{U_0^3\nu}{L_0 U_0^4}\right)^{1/4} = \mathrm{Re}^{-\frac{1}{4}}, \tag{9.37}$$

$$\frac{\tau_\eta}{\tau} \sim \left(\frac{\nu U_0^2}{\epsilon L_0^2}\right)^{1/2} = \left(\frac{\nu U_0^2 L_0}{U_0^3 L_0^2}\right)^{1/2} = \mathrm{Re}^{-\frac{1}{2}}. \tag{9.38}$$

This shows that the Reynolds number directly sets the dynamic range of the inertial range.

9.4.2 Energy spectrum of Kolmogorov turbulence

Eddy motions on a length-scale l correspond to wavenumber $k = 2\pi/l$. The kinetic energy ΔE contained between two wave numbers k_1 and k_2 can be described by

$$\Delta E = \int_{k_1}^{k_2} E(k)\,\mathrm{d}k, \tag{9.39}$$

where $E(k)$ is the so-called energy spectrum. For the inertial range in Kolmogorov's theory, we know that $E(k)$ is a universal function that only depends on ϵ and k. Hence $E(k)$ must be of the form

$$E(k) = C\,\epsilon^a\,k^b, \tag{9.40}$$

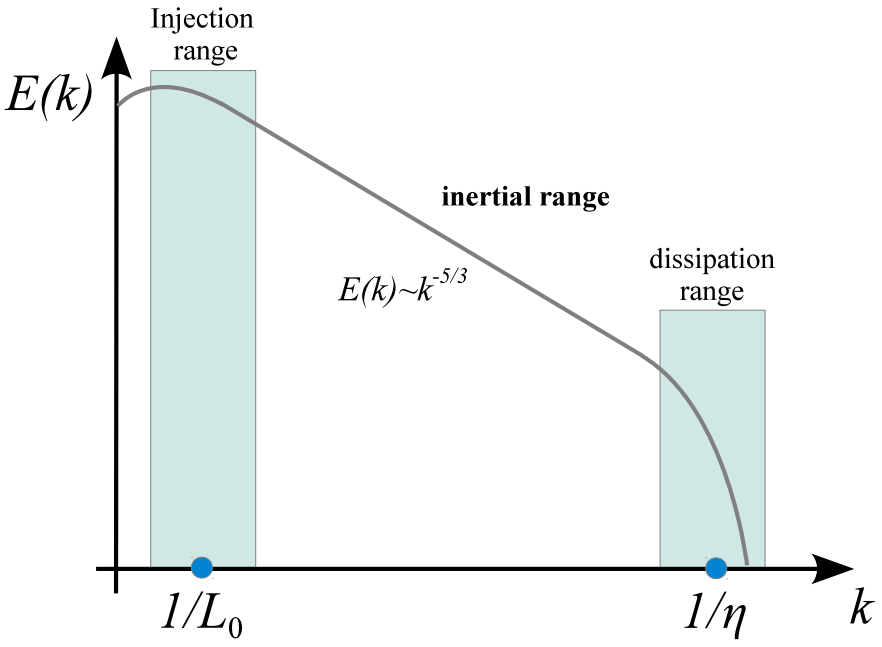


Figure 9.4: Schematic energy spectrum of Kolmogorov turbulence.

where C is a dimensionless constant. Through dimensional analysis it is easy to see that one must have $a = 2/3$ and $b = -5/3$. We hence obtain the famous $-5/3$ slope of the Kolmogorov energy power spectrum:

$$E(k) = C\,\epsilon^{2/3}\,k^{-5/3}. \tag{9.41}$$

The constant C is universal in Kolmogorov's theory, but cannot be computed from first principles. Experiment and numerical simulations give $C \simeq 1.5$ (Pope, 2000).

Actually, if we recall Kolmogorov's first similarity hypothesis, it makes the stronger claim that the statistics for all small scale motion is universal. This means that also the dissipation part of the turbulence must have a universal form. To include this in the description of the spectrum, we can for example write

$$E(k) = C\,\epsilon^{2/3}\,k^{-5/3} f_\eta(k\eta), \tag{9.42}$$

where $f_\eta(k\eta)$ us a universal function with $f_\eta(x) = 1$ for $x \ll 1$, and with $f_\eta(x) \to 0$ for $x \to \infty$. This function has to be determined experimentally or numerically. A good fit to different results is given by

$$f_\eta(x) = \exp\left(-\beta[(x^4 + c^4)^{1/4} - c]\right), \tag{9.43}$$

with $\beta_0 \sim 5.2$ and $c \sim 0.4$ (Pope, 2000).

10 Eulerian hydrodynamics

Many physical theories are expressed as partial differential equations (PDEs), including some of the most fundamental laws of nature, such as fluid dynamics (Euler and Navier Stokes equations), electromagnetism (Maxwell's equations) or general relativity/gravity (Einstein's field equations). Broadly speaking, partial differential equations (PDE) are equations describing relations between partial derivatives of a dependent variable with respect to several independent variables. For example,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{10.1}$$

is a PDE for the function $u = u(x, t)$. The independent variables are here x and t, the dependent variable is u (i.e. the function value). Unlike for ordinary differential equations (ODEs), there is no simple unified theory for PDEs. Rather, there are different types of PDEs which exhibit special features.

Given that the fundamental laws of classical physics are expressed in terms of partial differential equations, it is clear that such equations are of fundamental importance in all of physics. The solutions of PDEs can be extremely complex, encoding many physical phenomena. Simulation techniques are crucial to uncover and understand parts of the solution spaces of PDEs. But depending on the PDEs, numerically solving them can be very tricky, and is quite generally more difficult than solving systems of ordinary differential equations.

10.1 Types of PDEs

It is useful to distinguish different properties of PDEs in order to help classifying them. Some of the most important characteristics include:

- **Order of the PDE:** This is simply the highest derivative appearing in the PDE. A "second-order PDE" will have up to second partial derivatives.
- **Linearity:** A PDE (or system of PDEs) is linear if all terms are linear in the unknown function (the dependent variable) and its partial derivatives. Linear PDEs are (unsurprisingly) much simpler to solve than non-linear ones.
- **Homogeneity:** If all its terms contain the independent variable or its derivative, the PDE is said to be homogenous. Otherwise it contains "source-terms" and is inhomogenous.

In addition, PDEs can be classified according to some less obvious features which are however reflecting the nature of the physics that the PDEs describe. For example, the hydrodynamic fluid equations are local conservation laws, hence they are expressed as so-called *hyperbolic* conservation laws. What this means will be discussed later in more detail.

The most important linear and homogenous PDEs

In order to introduce the different characters of PDEs, let us look at the **three most impost important linear and homogenous PDEs**. These arguably are:

1. Laplace equation:
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \tag{10.2}$$

2. Heat conduction equation
$$\frac{\partial u}{\partial t} = \lambda^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{10.3}$$

3. Wave equation
$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{10.4}$$

These equations look quite similar, but their type and character is fundamentally different. In fact, one needs different techniques for solving them. One thing they have in common however is that, thanks to their linearity, if u_1 and u_2 are two solutions, then all linear combinations $c_1 u_1 + c_2 u_2$ are also solutions, which is the superposition principle.

Often, PDEs are characterized in terms of properties that are called

- elliptic
- parabolic
- hyperbolic

We now discuss how this general characterization of the PDE-type is defined. For linear 2nd-order PDEs of the form (a, b, c not all zero)

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + f\,u = g, \tag{10.5}$$

the classification is based on the discriminant

$$D = b^2 - 4ac. \tag{10.6}$$

The following cases are distinguished:

$$D = \begin{cases} < 0 & \text{elliptic,} \\ 0 & \text{parabolic,} \\ > 0 & \text{hyperbolic.} \end{cases} \tag{10.7}$$

This is done in analogy to conic sections, a connection that can be made explicit by making the replacements $\partial^2 u/\partial x^2 \to X^2$, $\partial^2 u/\partial x \partial y \to XY$, etc.

PDEs that fall in the same class of these types often exhibit similar mathematical and physical properties, and also require similar solution strategies.

Examples for types of 2nd-order PDEs:

Let's go back to our 'three most important PDEs' from above and consider only two independent variables for now.

- For the wave equation we have:

$$\frac{\partial^2 u}{\partial t^2} - c_s^2 \frac{\partial^2 u}{\partial x^2} = 0. \tag{10.8}$$

  So here we can identify $a = 1$, $b = 0$ and $c = -c_s^2$. Hence $D = 4c_s^2 > 0$. This equation is therefore *hyperbolic*.

- The heat conduction equation takes the form:

$$\frac{\partial u}{\partial t} - \lambda^2 \frac{\partial^2 u}{\partial x^2} = 0. \tag{10.9}$$

  Here we have $a = 0$, $b = 0$ and $c = -\lambda^2$, hence $D = 0$. This is a *parabolic* equation.

- Finally, the Laplace equation is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{10.10}$$

  yielding $a = 1$, $b = 0$, and $c = 1$. Therefore $D = -4 < 0$, and the equation is *elliptic*.

General linear PDEs of 2nd-order with more unknowns than two can be written as

$$\sum_{i,j=1}^{n} a_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_i b_i \frac{\partial u}{\partial x_i} + cu + d = 0. \tag{10.11}$$

Here the eigenvalues of the coefficient matrix a_{ij} can be used to determine the type, according to the following scheme:

- elliptic: all eigenvalues positive, or all negative

- parabolic: one zero eigenvalue, the others all positive or all negative
- hyperbolic: one negative eigenvalue the rest all positive, or one positive the rest all negative

Other cases (multiple eigenvalues of different sign) are sometimes called ultra-hyperbolic. Note that in 2D, this definition should be the same as our previous one. We can readily check that. To make contact with the previous notation, the coefficient matrix is then

$$a_{ij} = \begin{pmatrix} a & b/2 \\ b/2 & a \end{pmatrix}. \tag{10.12}$$

The characteristic equation for the eigenvalues then has solutions

$$\lambda_{1/2} = \frac{(a+c) \pm \sqrt{(a+c)^2 - 4ac + b^2}}{2}. \tag{10.13}$$

This means, for example, that we have two real eigenvalues with opposite signs if $D = b^2 - 4ac > 0$, which corresponds to the hyperbolic case, consistent with our previous simpler definition.

Linear systems of first-order PDEs

Linear systems of first-order homogeneous PDEs can be written as

$$\frac{\partial u_i}{\partial t} + \sum_j A_{ij} \cdot \frac{\partial u_i}{\partial x_j} = 0, \tag{10.14}$$

where $\mathbf{A} = (A_{ij})$ is a coefficient matrix, and one independent variable has been singled out as 'time' t. We may also use a more compact notation of the form

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{A} \cdot \frac{\partial \mathbf{u}}{\partial \mathbf{x}} = 0 \tag{10.15}$$

for this. Here, if $\mathbf{A}$ has only *real eigenvalues* and is diagonalizable, then the PDE system is called *hyperbolic* as well.

This definition can also be extended to non-linear PDEs of the form

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial \mathbf{x}} \left(\mathbf{F}(\mathbf{u}) \right) = 0, \tag{10.16}$$

where $\mathbf{F}(\mathbf{u})$ is some function of $\mathbf{u}$. Such PDEs are called conservation laws (compare with the continuity equation), and $\mathbf{F}$ is the flux. One can also write this equation in quasi-linear form by invoking the chain rule:

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{F}}{\partial \mathbf{u}} \cdot \frac{\partial \mathbf{u}}{\partial \mathbf{x}} = 0, \tag{10.17}$$

or more compactly as

$$\frac{\partial \mathbf{u}}{\partial t} + \overline{\mathbf{A}} \cdot \frac{\partial \mathbf{u}}{\partial \mathbf{x}} = 0, \tag{10.18}$$

where $\overline{\mathbf{A}} = \frac{\partial \mathbf{F}}{\partial \mathbf{u}}$ is the Jacobian of the flux. Now, if $\overline{\mathbf{A}}$ is diagonalizable and has real eigenvalues, we still call the (non-linear) set of equations a hyperbolic PDE system.

Character and boundary conditions of different PDE types

The different classes of PDEs show qualitatively different behavior in their solutions. Broadly speaking, the following observations can be made:

- **Hyperbolic PDEs in physics** typically describe dynamical processes of systems that start with some known initial conditions at time t_0. Solutions can develop steep regions or real discontinuities with time, even if they start with perfectly smooth initial states. To specify the initial conditions completely, one needs $u(x, t_0)$ and $\frac{\partial u}{\partial x}(x, t_0)$ and higher derivatives if present, as well as boundary conditions.

- **Parabolic PDEs** are often of second-order and describe slowly changing processes (for example diffusion). Solutions become here *smoother* with time. For describing the problem, one needs the initial state $u(x, t_0)$ as well as boundary conditions.

- **Elliptic PDEs** often describe static problems without time dependence, or equilibrium states of some kind. An important example is the Poisson equation for the gravitational or electrostatic field. The solutions of elliptic problems tend to be as smooth as allowed by the boundary conditions and source terms.

For all types of PDEs, boundary conditions are very important and need to be specified to determine the solution uniquely. If the value of the sought function is specified with a fixed value on the boundaries, one calls this *Dirichlet* boundary conditions. If instead the value of the derivative of u is prescribed on the boundary (usually in the normal direction with respect to the boundary), one has so-called *von Neumann* boundary conditions.

10.2 Solution schemes for PDEs

Unfortunately, for partial differential equations one cannot give a general solution method that works equally well for all types of problems. Rather, each type requires different approaches, and certain PDEs encountered in practice may even be best addressed with special custom techniques built by combining different elements from standard techniques. Important classes of solution schemes include the following:

- **Finite difference methods:** Here the differential operators are approximated through finite difference approximations, usually on a regular (cartesian) mesh, or some other kind of structured mesh (for example a polar grid). An example we already previously discussed is Poisson's equation treated with iterative (multigrid) methods.

- **Finite volume methods:** These may be seen as a subclass of finite difference methods. They are particularly useful for hyperbolic conservation laws. We

shall discuss examples for this approach in applications to fluid dynamics later in this section.

- **Spectral methods:** Here the solution is represented by a linear combination of functions, allowing the PDE to be transformed to algebraic equations or ordinary differential equations. Often this is done by applying Fourier techniques. For example, solving the Poisson equation with FFTs, as we discussed earlier, is a spectral method.

- **Method of lines:** This is a semi-discrete approach where all derivatives except for one are approximated with finite differences. The remaining derivative is then the only one left, so that the remaining problem forms a set of ordinary differential equations (ODEs). Very often, this approach is used in time-dependent problems. One here discretizes space in terms of a set of N points x_i, and for each of these points one obtains an ODE that describes the time evolution of the function at this point. The PDE is transformed in this way into a set of N coupled ODEs. For example, consider the heat diffusion equation in one dimension,

$$\frac{\partial u}{\partial t} + \lambda \frac{\partial^2 u}{\partial x^2} = 0. \tag{10.19}$$

  If we discretize this into a set of points that are spaced h apart, we obtain N equations

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} + \lambda \frac{u_{i+1} + u_{i-1} - 2u_i}{h^2} = 0. \tag{10.20}$$

  These differential equations can now be integrated in time as an ODE system. Note however that this is not necessarily stable. Some problems may require upwinding, i.e. asymmetric forms for the finite difference estimates to recover stability.

- **Finite element methods:** Here the domain is subdivided into "cells" (elements) of fairly arbitrary shape. The solution is then represented in terms of simple, usually polynomial functions on the element, and then the PDE is transformed to an algebraic problem for the coefficients in front of these simple functions. This is hence similar in spirit to spectral methods, except that the expansion is done in terms of highly localized functions on an element by element basis, and is truncated already at low order.

In practice, many different variants of these basic methods exist, and sometimes also combinations of them are used.

10.3 Simple advection

First-order equations of hyperbolic type are particularly useful for introducing the numerical difficulties that then also need to be addressed for more complicated nonlinear conservation laws (e.g. Toro, 1997; LeVeque, 2002; Stone et al., 2008). The

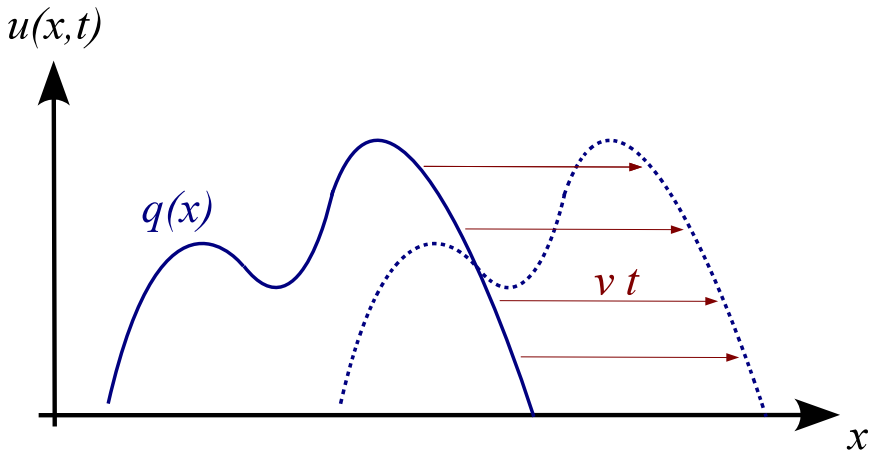


Figure 10.1: Simple advection with constant velocity to the right.

simplest equation of this type is the *advection equation* in one dimension. This is given by

$$\frac{\partial u}{\partial t} + v \cdot \frac{\partial u}{\partial x} = 0, \tag{10.21}$$

where $u = u(x,t)$ is a function of x and t, and v is a constant parameter. This equation is hyperbolic because the so-called coefficient matrix[1] is real and trivially diagonalizable.

If we are given any function $q(x)$, then

$$u(x,t) = q(x - vt) \tag{10.23}$$

is a solution of the PDE, as one can easily check. We can interpret $u(x, t = 0) = q(x)$ as initial condition, and the solution at a later time is then an exact copy of q, simply translated by $v\,t$ along the x-direction, as shown in Fig. 10.1.

Points that start at a certain coordinate x_0 are advected to a new location $x_{\mathrm{ch}}(t) = vt + x_0$. These so-called *characteristics* (see Fig. 10.2), which can be viewed as mediating the propagation of information in the system, are straight lines, all oriented in the downstream direction. Note that "downstream" refers to the direction in which the flow goes, whereas "upstream" is from where the flow comes.

Let's now assume we want to solve the advection problem numerically. (Strictly speaking this is of course superfluous as we have an analytic solution in this case, but we want to see how well a numerical technique would perform here.) We can approach this with a straightforward discretization of u on a special mesh, using for

[1] A linear system of first-order PDEs can be written in the generic form

$$\frac{\partial u_i}{\partial t} + \sum_j A_{ij} \frac{\partial u_i}{\partial x_j} = 0, \tag{10.22}$$

where A_{ij} is the coefficient matrix.

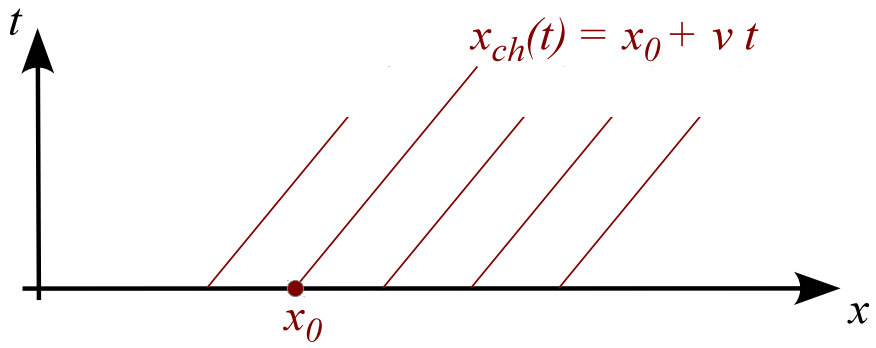


Figure 10.2: A set of flow characteristics for advection to the right with constant velocity v.

example the method of lines. This gives us:

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} + v\frac{u_{i+1} - u_{i-1}}{2h} = 0. \tag{10.24}$$

If we go one step further and also discretize the time derivative with a simple Euler scheme, we get

$$u_i^{(n+1)} = u_i^{(n)} - v\frac{u_{i+1}^{(n)} - u_{i-1}^{(n)}}{2h}\Delta t. \tag{10.25}$$

This is a complete update formula which can be readily applied to a given initial state on the grid. The big surprise is that this turns out to be quite violently unstable! For example, if one applies this to the advection of a step function, one invariably obtains strong oscillatory errors in the downstream region of the step, quickly rendering the numerical solution into complete garbage. What is the reason for this fundamental failure?

- First note that all characteristics (signals) propagate downstream in this problem, or in other words, information strictly travels in the flow direction in this problem.
- But, the information to update u_i is derived both from the upstream (u_{i-1}) and the downstream (u_{i+1}) side.
- According to how the information flows, u_i should not really depend on the downstream side at all, which in some sense is causally disconnected. So let's try to get rid off this dependence by going to a one-sided approximation for the spatial derivative, of the form:

  $$\frac{\mathrm{d}u_i}{\mathrm{d}t} + v\frac{u_i - u_{i-1}}{h} = 0. \tag{10.26}$$

  This is called *upwind differencing*. Interestingly, now the stability problems are completely gone!

- But there are still some caveats to observe: First of all, the discretization now depends on the sign of v. For negative v, one instead has to use

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} + v\frac{u_{i+1} - u_i}{h} = 0. \tag{10.27}$$

  The other is that the solution is not advected in a perfectly faithful way, instead it is quite significantly smoothed out, through a process one calls *numerical diffusion*.

We can actually understand where this strong diffusion in the 1st-order upwind scheme comes from. To this end, let's rewrite the upwind finite difference approximation of the spatial derivative as

$$\frac{u_i - u_{i-1}}{h} = \frac{u_{i+1} - u_{i-1}}{2h} - \frac{u_{i+1} - 2u_i + u_{i-1}}{2h}. \tag{10.28}$$

Hence our stable upwind scheme can also be written as

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} + v\frac{u_{i+1} - u_{i-1}}{2h} = \frac{vh}{2}\,\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}. \tag{10.29}$$

But recall from equation (7.3) that

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i \simeq \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \tag{10.30}$$

so if we define a diffusion constant $D = (vh)/2$, we are effectively solving the following problem,

$$\frac{\partial u}{\partial t} + v \cdot \frac{\partial u}{\partial x} = D\frac{\partial^2 u}{\partial x^2}, \tag{10.31}$$

and not the original advection problem. The diffusion term on the right hand side is here a byproduct of the numerical algorithm that we have used. We needed to add this numerical diffusion in order to obtain stability of the integration.

Note however that for better grid resolution, $h \to 0$, the diffusion becomes smaller, so in this limit one obtains an ever better solution. Also note that the diffusivity becomes larger for larger velocity v, so the faster one needs to advect, the stronger the numerical diffusion effects become.

Besides the upwinding requirement, integrating a hyperbolic conservation law with an explicit method in time also requires the use of a sufficiently small integration timestep, not only to get sufficiently good accuracy, but also for reasons of *stability*. In essence, there is a maximum timestep that may be used before the integration brakes down. How large can we make this timestep? Again, we can think about this in terms of information travel. If the timestep exceeds $\Delta t_{\max} = h/v$, then the updating of u_i would have to include information from u_{i-2}, but if we don't do this, the updating will likely become unstable.

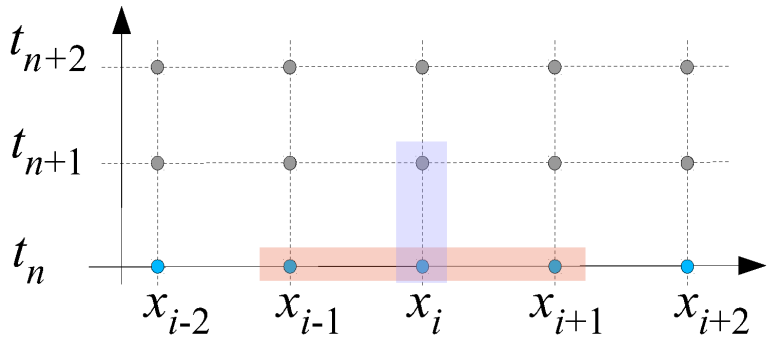


Figure 10.3: A discretization scheme for the continuity equation in one spatial dimension. The red and blue boxes mark the stencils that are applied for calculating the spatial and time derivatives.

This leads to the so-called *Courant-Friedrichs-Levy* (CFL) timestep condition (Courant et al., 1928), which for this problem takes the form

$$\Delta t \leq \frac{h}{v}. \tag{10.32}$$

This is a necessary but not sufficient condition for any explicit finite different approach of the hyperbolic advection equation. For other hyperbolic conservation laws, similar CFL-conditions apply.

Hyperbolic conservation laws

We now consider a hyperbolic conservation law, such as the continuity equation for the mass density of a fluid:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0. \tag{10.33}$$

We see that this is effectively the advection equation, but with a spatially variable velocity $\mathbf{v} = \mathbf{v}(\mathbf{x})$. Here $\mathbf{F} = \rho \mathbf{v}$ is the mass flux.

Let's study the problem in one spatial dimension, and consider a discretization both of the x- and t-axis. This corresponds to

$$\frac{\rho_i^{(n+1)} - \rho_i^{(n)}}{\Delta t} + \frac{F_{i+1}^{(n)} - F_{i-1}^{(n)}}{2\Delta x} = 0, \tag{10.34}$$

leading to the update rule

$$\rho_i^{(n+1)} = \rho_i^{(n)} + \frac{\Delta t}{2\Delta x}\left(F_{i-1}^{(n)} - F_{i+1}^{(n)}\right). \tag{10.35}$$

This is again found to be highly unstable, for the same reasons as in the plain advection problem: we have not observed in 'which direction the wind blows', or in other words, we have ignored in which direction the local characteristics point. For example, if the mass flux is to the right, we know that the characteristics point also

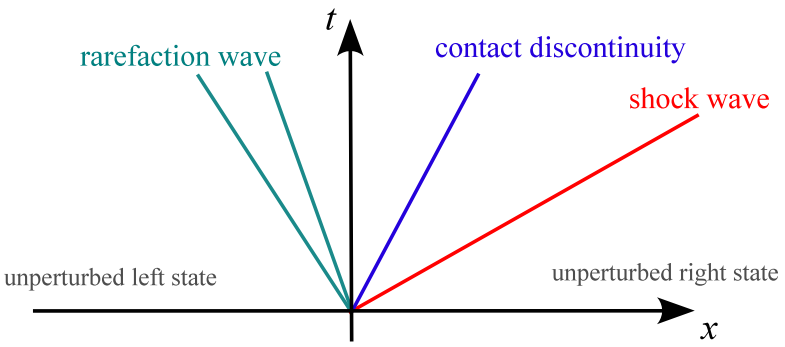


Figure 10.4: Wave structure of the solution of the Riemann problem. The central contact wave separates the original fluid phases. On the left and the right, there is either a shock or a rarefaction wave.

to the right. The upwind direction is therefore towards negative x, and by using only this information in making our spatial derivative one-sided, we should be able to resurrect stability.

Now, for the mass continuity equation identifying the local characteristics is quite easy, and in fact, their direction can simply be inferred from the sign of the mass flux. However, in more general situations for systems of non-linear PDEs, this is far less obvious. Here we need to use a so-called Riemann solvers to give us information about the local solution and the local characteristics (Toro, 1997). This then also implicitly identifies the proper upwinding that is needed for stability.

10.4 Riemann problem

The Riemann problem is an initial value problem for a hyperbolic system, consisting of two piece-wise constant states (two half-spaces) that meet at a plane at $t = 0$. The task is then to solve for the subsequent evolution at $t > 0$.

An important special case is the Riemann problem for the Euler equations (i.e. for ideal gas dynamics). Here the left and right states of the interface, can, for example, be uniquely specified by giving the three "primitive" variables density, pressure and velocity, viz.

$$U_L = \begin{pmatrix} \rho_L \\ P_L \\ \mathbf{v}_L \end{pmatrix}, \quad U_R = \begin{pmatrix} \rho_R \\ P_R \\ \mathbf{v}_R \end{pmatrix}. \tag{10.36}$$

Alternatively one can also specify density, momentum density, and energy density. For an ideal gas, this initial value problem can be solved analytically (Toro, 1997), modulo an implicit equation which requires numerical root-finding, i.e. the solution cannot be written down explicitly. The solution always contains characteristics for three self-similar waves, as shown schematically in Fig. 10.4. Some notes on this:

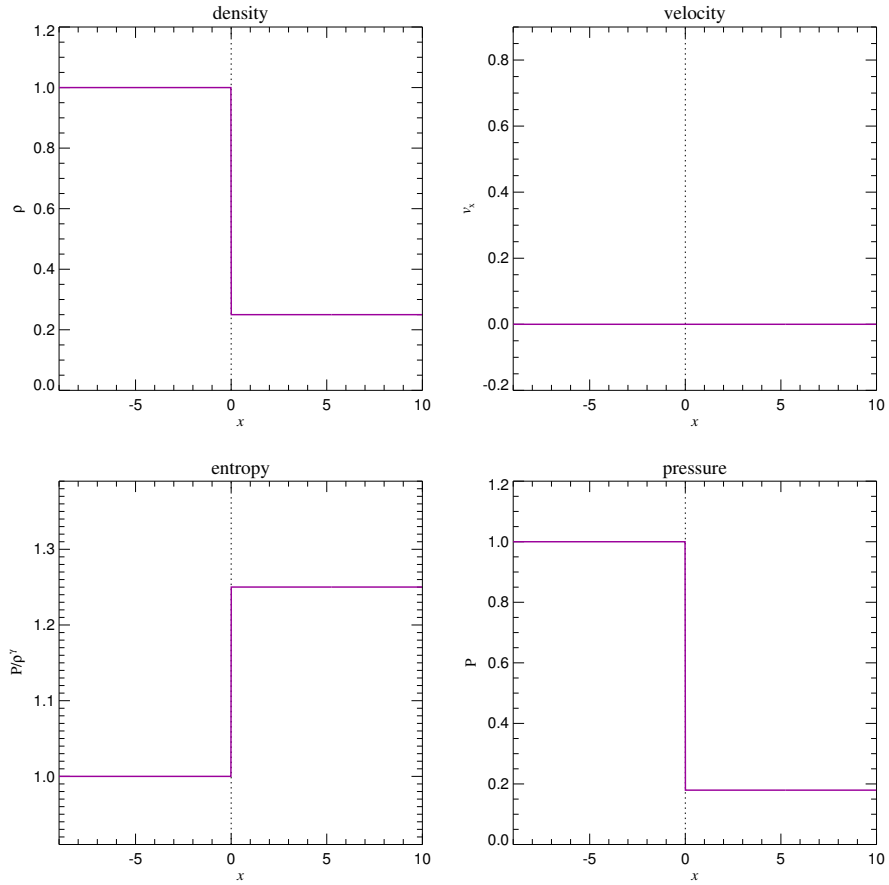


Figure 10.5: Initial state of an example Riemann problem, composed of two phases in different states that are brought into contact at $x = 0$ at time $t = 0$. (Since $v_x = 0$, the initial conditions are actually an example of the special case of a Sod shock-tube problem.)

- The middle wave is always present and is a contact wave that marks the boundary between the original fluid phases from the left and right sides.

- The contact wave is sandwiched between a shock or a rarefaction wave on either side (it is possible to have shocks on both sides, or rarefactions on both sides, or one of each). The rarefaction wave is not a single characteristic but rather a rarefaction fan with a beginning and an end.

- These waves propagate with constant speed. If the solution is known at some time $t > 0$, it can also be obtained at any other time through a suitable scaling transformation. An important corollary is that at $x = 0$, the fluid quantities $(\rho^\star, P^\star, \mathbf{v}^\star)$ are *constant in time* for $t > 0$.

- For $\mathbf{v}_L = \mathbf{v}_R = 0$, the Riemann problem simplifies and becomes the 'Sod shock tube' problem.

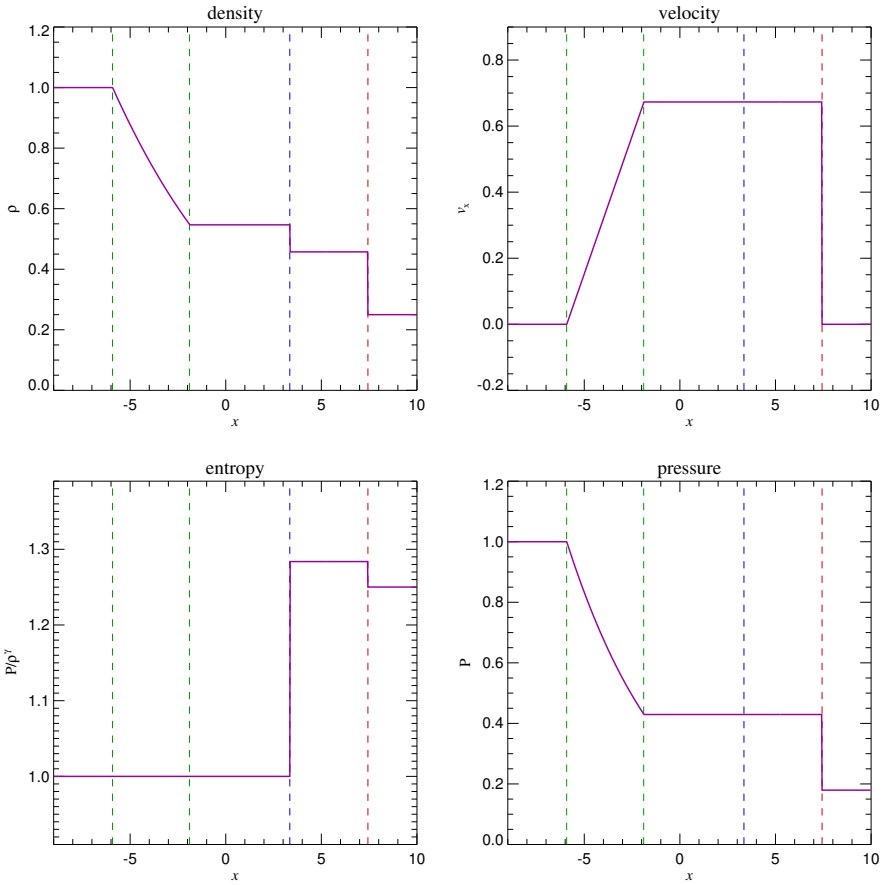


Figure 10.6: Evolved state at $t = 5.0$ of the initial fluid state displayed in Fig. 10.5. The blue dashed line marks the position of the contact wave, the green dashed lines give the location of the rarefaction fan, and the red dashed line marks the shock.

Let's consider an example how this wave structure looks in a real Riemann problem. We consider, for definiteness, a Riemann problem with $\rho_L = 1.0$, $P_L = 1.0$, $v_L = 0$, and $\rho_R = 0.25$, $P_R = 0.1795$, $v_R = 0$ (which is of Sod-shock type). The adiabatic exponent is taken to be $\gamma = 1.4$. We hence deal at $t = 0.0$ with the initial state displayed in Figure 10.5. After time $t = 5.0$, the wave structure formed by a rarefaction to the left (location marked in green), a contact in the middle (blue) and a shock to the right (red) can be nicely seen in Figure 10.6.

Some general properties of the waves appearing in the Riemann problem can be summarized as follows:

- *Shock:* This is a sudden compression of the fluid, associated with an irreversible conversion of kinetic energy to heat, i.e. here entropy is produced. The density, normal velocity component, pressure, and entropy all change discontinuously at a shock.

- *Contact discontinuity:* This traces the original separating plane between the two fluid phases that have been brought into contact. Pressure as well as the normal velocity are constant across a contact, but density, entropy and temperature can jump.
- *Rarefaction wave:* This occurs when the gas (suddenly) expands. The rarefaction wave smoothly connects two states over a finite spatial region; there are no discontinuities in any of the fluid variables.

10.5 Finite volume discretization

Let's now take a look how Riemann solvers can be used in the finite volume discretization approach to the PDEs of fluid dynamics. Recall that we can write our hyperbolic conservation laws as

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F} = 0. \tag{10.37}$$

Here $\mathbf{U}$ is a state vector and $\mathbf{F}$ is the flux vector. For example, the Euler equations of section 9.1.1 can be written in the form

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ \rho e \end{pmatrix}, \qquad \mathbf{F} = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v}^T + P \\ (\rho e + P)\mathbf{v} \end{pmatrix}, \tag{10.38}$$

with the specific energy $e = u + \mathbf{v}^2/2$ and u being the thermal energy per unit mass. The ideal gas equation gives the pressure as $P = (\gamma - 1)\rho u$ and provides a closure for the system.

In a finite volume scheme, we describe the system through the averaged state over a set of finite cells. These cell averages are defined as

$$\mathbf{U}_i = \frac{1}{V_i} \int_{\text{cell } i} \mathbf{U}(\mathbf{x})\, \mathrm{d}V. \tag{10.39}$$

Let's now see how we could devise an update scheme for these cell-averaged quantities.

1. We start by integrating the conservation law over a cell, and over a finite interval in time:
$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \mathrm{d}x \int_{t_n}^{t_{n+1}} \mathrm{d}t \left(\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} \right) = 0. \tag{10.40}$$
2. This gives
$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \mathrm{d}x \left[\mathbf{U}(x, t_{n+1}) - \mathbf{U}(x, t_n) \right] + \int_{t_n}^{t_{n+1}} \mathrm{d}t \left[\mathbf{F}(x_{i+\frac{1}{2}}, t) - \mathbf{F}(x_{i-\frac{1}{2}}, t) \right] = 0. \tag{10.41}$$

In the first term, we recognize the definition of the cell average:

$$\mathbf{U}_i^{(n)} \equiv \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \mathbf{U}(x, t_n) \mathrm{d}x. \tag{10.42}$$

Hence we have

$$\Delta x \left[\mathbf{U}_i^{(n+1)} - \mathbf{U}_i^{(n)} \right] + \int_{t_n}^{t_{n+1}} \mathrm{d}t \left[\mathbf{F}(x_{i+\frac{1}{2}}, t) - \mathbf{F}(x_{i-\frac{1}{2}}, t) \right] = 0. \tag{10.43}$$

3. Now, $\mathbf{F}(x_{i+\frac{1}{2}}, t)$ for $t > t_n$ is given by the solution of the Riemann problem with left state $\mathbf{U}_i^{(n)}$ and right state $\mathbf{U}_{i+1}^{(n)}$. At the interface, this solution is *independent* of time. We can hence write

$$\mathbf{F}(x_{i+\frac{1}{2}}, t) = \mathbf{F}^\star_{i+\frac{1}{2}}, \tag{10.44}$$

where $\mathbf{F}^\star_{i+\frac{1}{2}} = \mathbf{F}_{\text{Riemann}}(\mathbf{U}_i^{(n)}, \mathbf{U}_{i+1}^{(n)})$ is a short-hand notation for the corresponding Riemann solution sampled at the interface. Hence we now get

$$\Delta x \left[\mathbf{U}_i^{(n+1)} - \mathbf{U}_i^{(n)} \right] + \Delta t \left[\mathbf{F}^\star_{i+\frac{1}{2}} - \mathbf{F}^\star_{i-\frac{1}{2}} \right] = 0. \tag{10.45}$$

Or alternative, as an explicit update formula:

$$\mathbf{U}_i^{(n+1)} = \mathbf{U}_i^{(n)} + \frac{\Delta t}{\Delta x} \left[\mathbf{F}^\star_{i-\frac{1}{2}} - \mathbf{F}^\star_{i+\frac{1}{2}} \right]. \tag{10.46}$$

The first term in the square bracket gives the flux that flows from left into the cell, the second term is the flux out of the cell on its right side. The idea to use the Riemann solution in the updating step is due to Godunov, that's why such schemes are often called *Godunov schemes*.

It is worthwhile to note that we haven't really made any approximation in the above (yet). In particular, if we calculate $\mathbf{F}_{\text{Riemann}}$ analytically (and hence exactly), then the above seems to account for the correct fluxes for arbitrarily long times. So does this mean that we get a perfectly accurate result even for very large timesteps? This certainly sounds too good to be true, so there must be a catch somewhere.

Indeed, there is. First of all, we have assumed that the Riemann problems are independent of each other and each describe infinite half-spaces. This is not true once we consider finite volume cells, but it is still ok for a while as long t_{n+1} is close enough to t_n such that the waves emanating in one interface have not yet arrived at the next interface left or right. This then leads to a CFL-timestep criterion, were $\Delta t \leq \Delta x / c_{\max}$ and $c_{\max}$ is the maximum wavespeed.

Another point is more subtle and comes into play when we consider more than one timestep. We assumed that the $\mathbf{U}_i^{(n)}$ describe piece-wise constant states which can then be fed to the Riemann solver to give us the flux. However, even when this is true initially, we have just seen that after one timestep it will not be true anymore. By ignoring this in the subsequent timestep (which is done by performing an averaging step that washes out the cell substructure that developed as part of the evolution during the previous timestep) we make some error.

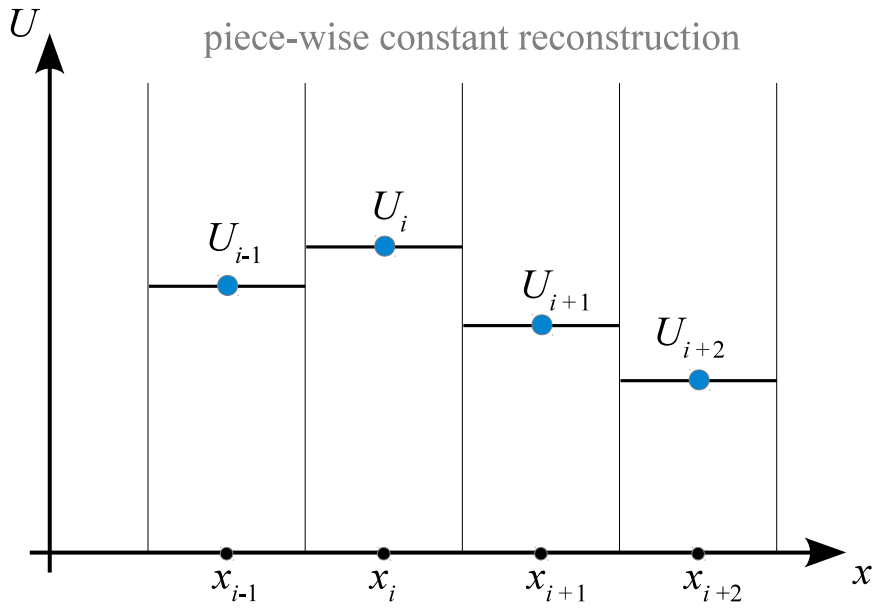


Figure 10.7: Piece-wise constant states of a fluid forming the simplest possible reconstruction of its state based on a set of discrete values U_i known at spatial positions x_i.

10.6 Godunov's method and Riemann solvers

It is useful to introduce another interpretation of common finite-volume discretizations of fluid dynamics, so-called Reconstruct-Evolve-Average (REA) schemes. We also use this here for a short summary of Godunov's important method, and the way Riemann solvers come into play in it.

An REA update scheme of a hydrodynamical system discretized on a mesh can be viewed as a sequence of three steps:

1. *Reconstruct:* Using the cell-averaged quantities (as shown in Fig. 10.7), this defines the run of these quantities everywhere in the cell. In the sketch, a piecewise constant reconstruction is assumed, which is the simplest procedure one can use and leads to 1st order accuracy.

2. *Evolve:* The reconstructed state is then evolved forward in time by Δt. In Godunov's approach, this is done by treating each cell interface as a piecewise constant initial value problem which is solved with the Riemann solver exactly or approximately. This solution is formally valid as long as the waves emanating from opposite sides of a cell do not yet start to interact. In practice, one therefore needs to limit the timestep Δt such that this does not happen.

3. *Average:* The wave structure resulting from the evolution over timestep Δt is spatially averaged in a conservative fashion to compute new states $\mathbf{U}^{n+1}$ for each cell. Fortunately, the averaging step does not need to be done explicitly;

instead it can simply be carried out by accounting for the fluxes that enter or leave the control volume of the cell. Then the whole cycle repeats again.

What is needed for the *evolve* step is a prescription to either exactly or approximately solve the Riemann problem for a piece-wise linear left and right state that are brought into contact at time $t = t_n$. Formally, this can be written as

$$\mathbf{F}^\star = \mathbf{F}_{\text{Riemann}}(\mathbf{U}_L, \mathbf{U}_R). \tag{10.47}$$

In practice, a variety of approximate Riemann solvers $\mathbf{F}_{\text{Riemann}}$ are commonly used in the literature (Rusanov, 1961; Harten et al., 1983; Toro, 1997; Miyoshi & Kusano, 2005). For the ideal gas and for isothermal gas, it is also possible to solve the Riemann problem exactly, but not in closed form (i.e. the solution involves an iterative root finding of a non-linear equation).

There are now two main issues left:

- How can this be extended to multiple spatial dimensions?
- How can it be extended such that a higher order integration accuracy both in space and time is reached?

We'll discuss these issues next.

10.7 Extensions to multiple dimensions

So far, we have considered *one-dimensional* hyperbolic conservation laws of the form

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) = 0, \tag{10.48}$$

where ∂_t is a short-hand notation for $\partial_t = \frac{\partial}{\partial t}$, and similarly $\partial_x = \frac{\partial}{\partial x}$. For example, for isothermal gas with soundspeed c_s, the state vector $\mathbf{U}$ and flux vector $\mathbf{F}(\mathbf{U})$ are given as

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} \rho u \\ \rho u^2 + \rho c_s^2 \end{pmatrix}, \tag{10.49}$$

where u is the velocity in the x-direction.

In three dimensions, the PDEs describing a fluid become considerably more involved. For example, the Euler equations for an ideal gas are given in explicit form as

$$\partial_t \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{pmatrix} + \partial_x \begin{pmatrix} \rho u \\ \rho u^2 + P \\ \rho uv \\ \rho uw \\ \rho u(\rho e + P) \end{pmatrix} + \partial_y \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + P \\ \rho vw \\ \rho v(\rho e + P) \end{pmatrix} + \partial_z \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + P \\ \rho w(\rho e + P) \end{pmatrix} = 0, \tag{10.50}$$

where $e = e_{\rm therm} + (u^2 + v^2 + w^2)/2$ is the total specific energy per unit mass, $e_{\rm therm}$ is the thermal energy per unit mass, and $P = (\gamma - 1)\rho\, e_{\rm therm}$ is the pressure. These equations are often written in the following notation:

$$\partial_t \mathbf{U} + \partial_x \mathbf{F} + \partial_y \mathbf{G} + \partial_z \mathbf{H} = 0. \tag{10.51}$$

Here the functions $\mathbf{F}(\mathbf{U})$, $\mathbf{G}(\mathbf{U})$ and $\mathbf{H}(\mathbf{U})$ give the flux vectors in the x-, y- and z-direction, respectively.

10.7.1 Dimensional splitting

Let us now consider the three dimensionally split problems derived from equation (10.51):

$$\partial_t \mathbf{U} + \partial_x \mathbf{F} = 0, \tag{10.52}$$

$$\partial_t \mathbf{U} + \partial_y \mathbf{G} = 0, \tag{10.53}$$

$$\partial_t \mathbf{U} + \partial_z \mathbf{H} = 0. \tag{10.54}$$

Note that the vectors appearing here have still the same dimensionality as in the full equations. They are *augmented* one-dimensional problems, i.e. the transverse variables still appear but spatial differentiation happens only in one direction. Because of this, these additional transverse variables do not make the 1D problem more difficult compared to the 'pure' 1D problem considered earlier, but the fluxes appearing in them still need to be included.

Now let us assume that he have a method to solve/advance each of these one-dimensional problems. We can for example express this formally through time-evolution operators $\mathcal{X}(\Delta t)$, $\mathcal{Y}(\Delta t)$, and $\mathcal{Z}(\Delta t)$, which advance the solution by a timestep Δt. Then the full time advance of the system can for example be approximated by

$$\mathbf{U}^{n+1} \simeq \mathcal{Z}(\Delta t)\mathcal{Y}(\Delta t)\mathcal{X}(\Delta t)\mathbf{U}^n. \tag{10.55}$$

This is one possible dimensionally split update scheme. In fact, this is the exact solution if equations (10.52)-(10.53) represent the linear advection problem, but for more general non-linear equations it only provides a first order approximation. However, higher-order dimensionally split update schemes can also be easily constructed. For example, in two-dimensions,

$$\mathbf{U}^{n+1} = \frac{1}{2}[\mathcal{X}(\Delta t)\mathcal{Y}(\Delta t) + \mathcal{Y}(\Delta t)\mathcal{X}(\Delta t)]\mathbf{U}^n \tag{10.56}$$

and

$$\mathbf{U}^{n+1} = \mathcal{X}(\Delta t/2)\mathcal{Y}(\Delta t)\mathcal{X}(\Delta t/2)\mathbf{U}^n \tag{10.57}$$

are second-order accurate. Similarly, for three dimensions the scheme

$$\mathbf{U}^{n+1} = \mathcal{X}(\Delta t/2)\mathcal{Y}(\Delta t/2)\mathcal{Z}(\Delta t)\mathcal{Y}(\Delta t/2)\mathcal{X}(\Delta t/2)\mathbf{U}^n \tag{10.58}$$

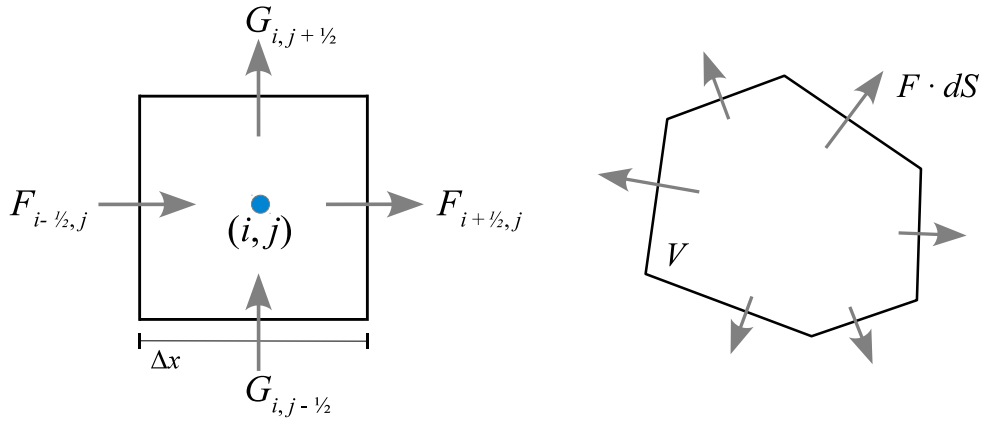


Figure 10.8: Sketch of unsplit finite-volume update schemes. On the left, the case of a structured Cartesian grid is shown, the case on the right is for an unstructured grid.

is second-order accurate. As a general rule of thumb, the time evolution operators have to be applied alternatingly in reverse order to reach second-order accuracy. We see that the dimensionless splitting reduces the problem effectively to a sequence of one-dimensional solution operations which are applied to multi-dimensional domains. Note that each one-dimensional operator leads to an update of $\mathbf{U}$, and is a complete step for the corresponding augmented one-dimensional problem. Gradients, etc., that are needed for the next step then have to be recomputed before the next time-evolution operator is applied. In practical applications of mesh codes, these one-dimensional solves are often called *sweeps*.

10.7.2 Unsplit schemes

In an unsplit approach, all flux updates of a cell are applied simultaneously to a cell, not sequentially. This is for example illustrated in 2D in the situations depicted in Figure 10.8. The unsplit update of cell i, j in the Cartesian case is then given by

$$U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{\Delta t}{\Delta x}\left(\mathbf{F}_{i-\frac{1}{2},j} - \mathbf{F}_{i+\frac{1}{2},j}\right) + \frac{\Delta t}{\Delta y}\left(\mathbf{G}_{i,j-\frac{1}{2}} - \mathbf{G}_{i,j+\frac{1}{2}}\right). \tag{10.59}$$

Unsplit approaches can also be used for irregular shaped cells like those appearing in unstructured meshes (see Fig. 10.8). For example, integrating over a cell of volume V and denoting with $\mathbf{U}$ the cell average, we can write the cell update with the divergence theorem as

$$\mathbf{U}^{n+1} = \mathbf{U}^{n} - \frac{\Delta t}{V}\int \mathbf{F} \cdot \mathbf{dS}, \tag{10.60}$$

where the integration is over the whole cell surface, with outwards pointing face area vectors $\mathbf{dS}$.

10.8 Extensions for high-order accuracy

We should first clarify what we mean with higher order schemes. Loosely speaking, this refers to the convergence rate of a scheme in smooth regions of a flow. For example, if we know the analytic solution $\rho(x)$ for some problem, and then obtain a numerical result ρ_i at a set of N points at locations x_i, we can ask what the typical error of the solution is. One possibility to quantify this would be through a L1 error norm, for example in the form

$$L1 = \frac{1}{N} \sum_i |\rho_i - \rho(x_i)|, \tag{10.61}$$

which can be interpreted as the average error per cell. If we now measure this error quantitatively for different resolutions of the applied discretization, we would like to find that L1 decreases with increasing N. In such a case our numerical scheme is converging, and provided we use sufficient numerical resources, we have a chance to get below any desired absolute error level. But the *rate of convergence* can be very different between different numerical schemes when applied to the same problem. If a method shows a $L1 \propto N^{-1}$ scaling, it is said to be first-order accurate; a doubling of the number of cells will then cut the error in half. A second-order method has $L1 \propto N^{-2}$, meaning that a doubling of the number of cells can actually reduce the error by a factor of 4. This much better convergence rate is of course highly desirable. It is also possible to construct schemes with still higher convergence rates, but they tend to quickly become much more complex and computationally involved, so that one eventually reaches a point of diminishing return, depending on the specific type of problem. But the extra effort one needs to make to go from first to second-order is often very small, sometimes trivially small, so that one basically should always strive to try at least this.

A first step in constructing a 2nd order extension of Godunov's method is to replace the piece-wise constant with a piece-wise linear reconstruction. This requires that one first estimates gradients for each cell (usually by a simple finite difference formula). These are then slope-limited if needed such that the linear extrapolations of the cell states to the cell interfaces do not introduce new extrema. This slope-limiting procedure is quite important; it needs to be done to avoid that real fluid discontinuities introduce large spurious oscillations into the fluid.

Given slope limited gradients, for example $\nabla\rho$ for the density, one can then estimate the left and right states adjacent to an interface $x_{i+\frac{1}{2}}$ by spatial extrapolation from the centers of the cells left and right from the interface:

$$\rho^L_{i+\frac{1}{2}} = \rho_i + (\nabla\rho)_i \frac{\Delta x}{2}, \tag{10.62}$$

$$\rho^R_{i+\frac{1}{2}} = \rho_{i+1} - (\nabla\rho)_{i+1} \frac{\Delta x}{2}. \tag{10.63}$$

The next step would in principle be to use these states in the Riemann solver. In doing this we will ignore the fact that our reconstruction has now a gradient over

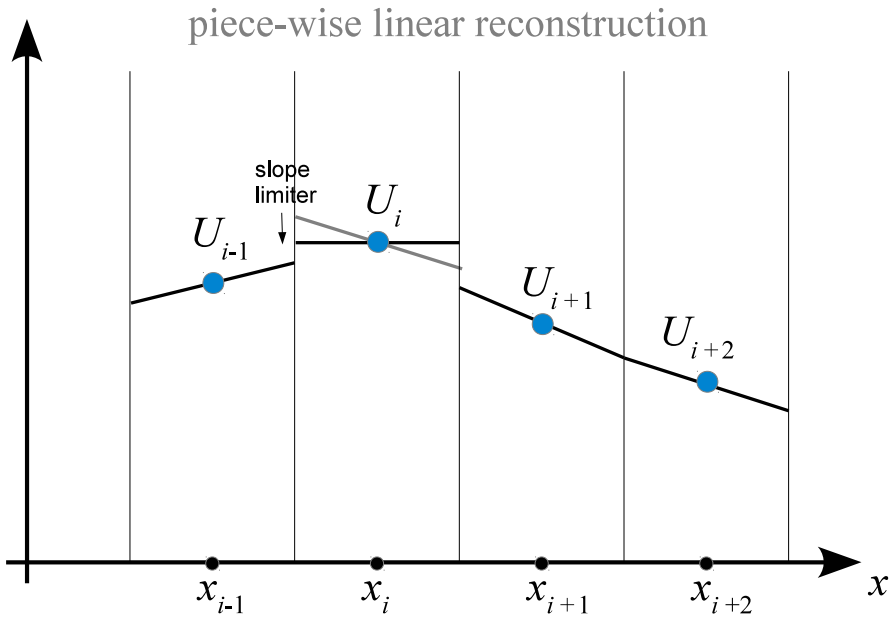


Figure 10.9: Piece-wise linear reconstruction scheme applied to a fluid state represented through a regular mesh.

the cell; instead we still pretend that the fluid state can be taken as piece-wise constant left and right of the interface as far as the Riemann solver is concerned. However, it turns out that the spatial extrapolation needs to be augmented with a temporal extrapolation one half timestep into the future, such that the flux estimate is now effectively done in the middle of the timestep. This is necessary both to reach second-order accuracy in time and also for stability reasons. Hence we really need to use

$$\rho^L_{i+\frac{1}{2}} = \rho_i + (\nabla\rho)_i \frac{\Delta x}{2} + \left(\frac{\partial\rho}{\partial t}\right)_i \frac{\Delta t}{2}, \tag{10.64}$$

$$\rho^R_{i+\frac{1}{2}} = \rho_{i+1} - (\nabla\rho)_{i+1} \frac{\Delta x}{2} + \left(\frac{\partial\rho}{\partial t}\right)_{i+1} \frac{\Delta t}{2}, \tag{10.65}$$

for extrapolating to the interfaces. More generally, this has to be done for the whole state vector of the system, i.e.

$$\mathbf{U}^L_{i+\frac{1}{2}} = \mathbf{U}_i + (\partial_x\mathbf{U})_i \frac{\Delta x}{2} + (\partial_t\mathbf{U})_i \frac{\Delta t}{2}, \tag{10.66}$$

$$\mathbf{U}^R_{i+\frac{1}{2}} = \mathbf{U}_{i+1} - (\partial_x\mathbf{U})_{i+1} \frac{\Delta x}{2} + (\partial_t\mathbf{U})_{i+1} \frac{\Delta t}{2}. \tag{10.67}$$

Note that here the quantity $(\partial_x\mathbf{U})_i$ is a (slope-limited) *estimate* of the gradient in cell i, based on finite-differences plus a slope limiting procedure. Similarly, we somehow need to estimate the time derivative encoded in $(\partial_t\mathbf{U})_i$. How can this be

done? One way to do this is to exploit the Jacobian matrix of the Euler equations. We can write the Euler equations as

$$\partial_t \mathbf{U} = -\partial_x \mathbf{F}(\mathbf{U}) = -\frac{\partial \mathbf{F}}{\partial \mathbf{U}} \partial_x \mathbf{U} = -\mathbf{A}(\mathbf{U}) \partial_x \mathbf{U}, \tag{10.68}$$

where $\mathbf{A}(\mathbf{U})$ is the Jacobian matrix. Using this, we can simply estimate the required time-derivative based on the spatial derivatives:

$$(\partial_t \mathbf{U})_i = -\mathbf{A}(\mathbf{U}_i) (\partial_x \mathbf{U})_i. \tag{10.69}$$

Hence the extrapolation can be done as

$$\mathbf{U}^L_{i+\frac{1}{2}} = \mathbf{U}_i + \left[\frac{\Delta x}{2} - \frac{\Delta t}{2}\mathbf{A}(\mathbf{U}_i)\right] (\partial_x \mathbf{U})_i, \tag{10.70}$$

$$\mathbf{U}^R_{i+\frac{1}{2}} = \mathbf{U}_{i+1} + \left[-\frac{\Delta x}{2} - \frac{\Delta t}{2}\mathbf{A}(\mathbf{U}_{i+1})\right] (\partial_x \mathbf{U})_{i+1}. \tag{10.71}$$

This procedure defines the so-called MUSCL-Hancock scheme (van Leer, 1984; Toro, 1997; van Leer, 2006), which is a 2nd-order accurate extension of Godunov's method.

Higher-order extensions such as the piece-wise parabolic method (PPM) start out with a higher order polynomial reconstruction. In the case of PPM, parabolic shapes are assumed in each cell instead of piece-wise linear states. The reconstruction is still guaranteed to be conservative, i.e. the integral underneath the reconstruction recovers the total values of the conserved variables individually in each cell. So-called ENO and WENO schemes (e.g. Balsara et al., 2009) use yet higher-order polynomials to reconstruct the state in a conservative fashion. Here many more cells in the environment need to be considered (i.e. the so-called *stencil* of these methods is much larger) to robustly determine the coefficients of the reconstruction. This can for example involve a least-square fitting procedure (Ollivier-Gooch, 1997).

11 Smoothed particle hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a technique for approximating the continuum dynamics of fluids through the use of particles, which may also be viewed as interpolation points (SPH; Lucy, 1977; Gingold & Monaghan, 1977; Monaghan, 1992; Springel, 2010). The principal idea of SPH is to treat hydrodynamics in a completely mesh-free fashion, in terms of a set of sampling particles. Hydrodynamical equations of motion are then derived for these particles, yielding a quite simple and intuitive formulation of gas dynamics. Moreover, it turns out that the particle representation of SPH has excellent conservation properties. Energy, linear momentum, angular momentum, mass, and entropy (if no artificial viscosity operates) are all simultaneously conserved. In addition, there are no advection errors in SPH, and the scheme is fully Galilean invariant, unlike alternative mesh-based Eulerian techniques. Due to its Lagrangian character, the local resolution of SPH follows the mass flow automatically, a property that is convenient in representing the large density contrasts often encountered in astrophysical problems.

11.1 Kernel Interpolants

At the heart of smoothed particle hydrodynamics lie so-called kernel interpolants. In particular, we use a kernel summation interpolant for estimating the density, which then determines the rest of the basic SPH equations through the variational formalism.

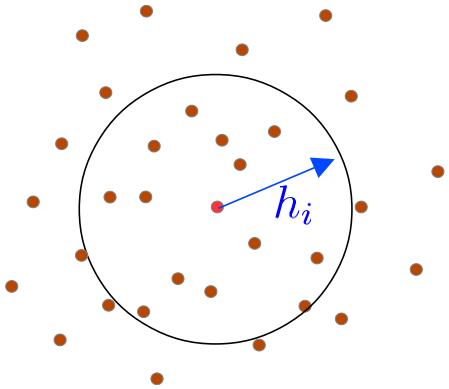
For any field $F(\mathbf{r})$, we may define a smoothed interpolated version, $F_s(\mathbf{r})$, through a convolution with a kernel $W(\mathbf{r}, h)$:

$$F_s(\mathbf{r}) = \int F(\mathbf{r}')\, W(\mathbf{r} - \mathbf{r}', h)\, \mathrm{d}\mathbf{r}'. \tag{11.1}$$

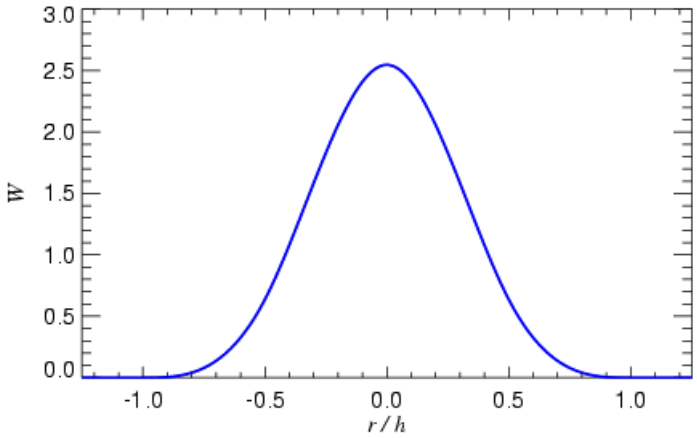
Here h describes the characteristic width of the kernel, which is normalized to unity and approximates a Dirac δ-function in the limit $h \to 0$. We further require that the kernel is symmetric and sufficiently smooth to make it at least twice differentiable. One possibility for W is a Gaussian. However, most current SPH implementations are based on kernels with a finite support. Usually a cubic spline is adopted with $W(r, h) = w(\frac{r}{2h})$, and

$$w_{3\mathrm{D}}(q) = \frac{8}{\pi} \begin{cases} 1 - 6q^2 + 6q^3, & 0 \le q \le \frac{1}{2}, \\ 2\,(1-q)^3\,, & \frac{1}{2} < q \le 1, \\ 0, & q > 1, \end{cases} \tag{11.2}$$

in three-dimensional normalization. Through Taylor expansion, it is easy to see that the kernel interpolant is at least second-order accurate due to the symmetry of the kernel.



B-Spline Kernel:



Suppose now we know the field at a set of points $\mathbf{r}_i$, i.e. $F_i = F(\mathbf{r}_i)$. The points have an associated mass m_i and density ρ_i, such that $\Delta\mathbf{r}_i \sim m_i/\rho_i$ is their associated finite volume element. Provided the points sufficiently densely sample the kernel volume, we can approximate the integral in Eqn. (11.1) with the sum

$$F_s(\mathbf{r}) \simeq \sum_j \frac{m_j}{\rho_j} F_j W(\mathbf{r} - \mathbf{r}_j, h). \tag{11.3}$$

This is effectively a Monte-Carlo integration, except that thanks to the comparatively regular distribution of points encountered in practice, the accuracy is better than for a random distribution of the sampling points. In particular, for points in one dimension with equal spacing d, one can show that for $h = d$ the sum of Eqn. (11.3) provides a second order accurate approximation to the real underlying function. Unfortunately, for the irregular yet somewhat ordered particle configurations encountered in real applications, a formal error analysis is not straightforward.

It is clear however, that at the very least one should have $h \geq d$, which translates to a minimum of ~ 33 neighbours in 3D.

Importantly, we see that the estimate for $F_s(\mathbf{r})$ is defined everywhere (not only at the underlying points), and is differentiable thanks to the differentiability of the kernel, albeit with a considerably higher interpolation error for the derivative. Moreover, if we set $F(\mathbf{r}) = \rho(\mathbf{r})$, we obtain

$$\rho_s(\mathbf{r}) \simeq \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h), \tag{11.4}$$

yielding a density estimate just based on the particle coordinates and their masses. In general, the smoothing length can be made variable in space, $h = h(\mathbf{r}, t)$, to account for variations in the sampling density. This adaptivity is one of the key advantages of SPH and is essentially always used in practice. There are two options to introduce the variability of h into Eqn. (11.4). One is by adopting $W(\mathbf{r} - \mathbf{r}_j, h(\mathbf{r}))$ as kernel, which corresponds to the so-called 'scatter' approach. It has the advantage that the volume integral of the smoothed field recovers the total mass, $\int \rho_s(\mathbf{r})\, \mathrm{d}\mathbf{r} = \sum_i m_i$. On the other hand, the so-called 'gather' approach, where we use $W(\mathbf{r} - \mathbf{r}_j, h(\mathbf{r}_i))$ as kernel in Eqn. (11.4), requires only knowledge of the smoothing length h_i for estimating the density of particle i, which leads to computationally convenient expressions when the variation of the smoothing length is consistently included in the SPH equations of motion. Since the density is only needed at the coordinates of the particles and the total mass is conserved anyway (since it is tied to the particles), it is not important that the volume integral of the gather form of $\rho_s(\mathbf{r})$ exactly equals the total mass.

In the following we drop the subscript s for indicating the smoothed field, and adopt as SPH estimate of the density of particle i the expression

$$\rho_i = \sum_{j=1}^{N} m_j\, W(\mathbf{r}_i - \mathbf{r}_j, h_i). \tag{11.5}$$

It is clear now why kernels with a finite support are preferred. They allow the summation to be restricted to the N_{ngb} neighbors that lie within the spherical region of radius $2h$ around the target point $\mathbf{r}_i$, corresponding to a computational cost of order $\mathcal{O}(N_{\mathrm{ngb}}\, N)$ for the full density estimate. Normally this number N_{ngb} of neighbors within the support of the kernel is approximately (or exactly) kept constant by choosing the h_i appropriately. N_{ngb} hence represents an important parameter of the SPH method and needs to be made large enough to provide sufficient sampling of the kernel volumes. Kernels like the Gaussian on the other hand would require a summation over all particles N for every target particle, resulting in a $\mathcal{O}(N^2)$ scaling of the computational cost.

If SPH was really a Monte-Carlo method, the accuracy expected from the interpolation errors of the density estimate would be rather problematic. But the errors are much smaller because the particles do not sample the fluid in a Poissonian fashion.

Instead, their distances tend to equilibrate due to the pressure forces, which makes the interpolation errors much smaller (Price, 2012). Yet, they remain a significant source of error in SPH and are ultimately the primary origin of the noise inherent in SPH results (Bauer & Springel, 2012).

Even though we have based most of the above discussion on the density, the general kernel interpolation technique can also be applied to other fields, and to the construction of differential operators. For example, we may write down a smoothed velocity field and take its derivative to estimate the local velocity divergence, yielding:

$$(\nabla \cdot \mathbf{v})_i = \sum_j \frac{m_j}{\rho_j} \mathbf{v}_j \cdot \nabla_i W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{11.6}$$

However, an alternative estimate can be obtained by considering the identity $\rho \nabla \cdot \mathbf{v} = \nabla \cdot (\rho \mathbf{v}) - \mathbf{v} \cdot \nabla \rho$, and computing kernel estimates for the two terms on the right hand side independently. Their difference then yields

$$(\nabla \cdot \mathbf{v})_i = \frac{1}{\rho_i} \sum_j m_j (\mathbf{v}_j - \mathbf{v}_i) \cdot \nabla_i W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{11.7}$$

This pair-wise formulation turns out to be more accurate in practice. In particular, it has the advantage of always providing a vanishing velocity divergence if all particle velocities are equal.

11.2 Variational Derivation of SPH

The Euler equations for inviscid gas dynamics in Lagrangian (comoving) form are given by

$$\frac{\mathrm{d}\rho}{\mathrm{d}t} + \rho \nabla \cdot \mathbf{v} = 0, \tag{11.8}$$

$$\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} + \frac{\nabla P}{\rho} = 0, \tag{11.9}$$

$$\frac{\mathrm{d}u}{\mathrm{d}t} + \frac{P}{\rho} \nabla \cdot \mathbf{v} = 0, \tag{11.10}$$

where $\mathrm{d}/\mathrm{d}t = \partial/\partial t + \mathbf{v} \cdot \nabla$ is the convective derivative. This system of partial differential equations expresses conservation of mass, momentum and energy. Eckart (1960) has shown that the Euler equations for an inviscid ideal gas follow from the Lagrangian

$$L = \int \rho \left(\frac{\mathbf{v}^2}{2} - u \right) \mathrm{d}V. \tag{11.11}$$

This opens up an interesting route for obtaining discretized equations of motion for gas dynamics. Instead of working with the continuum equations directly and trying to heuristically work out a set of accurate difference formulas, one can discretize the Lagrangian and then derive SPH equations of motion by applying the variational

principals of classical mechanics (Springel & Hernquist, 2002). Using a Lagrangian also immediately guarantees certain conservation laws and retains the geometric structure imposed by Hamiltonian dynamics on phase space.

We start by discretizing the Lagrangian in terms of fluid particles of mass m_i, yielding

$$L_{\rm SPH} = \sum_i \left(\frac{1}{2} m_i \mathbf{v}_i^2 - m_i u_i \right), \tag{11.12}$$

where it has been assumed that the thermal energy per unit mass of a particle can be expressed through an entropic function A_i of the particle, which simply labels its specific thermodynamic entropy. The pressure of the particles is

$$P_i = A_i \rho_i^\gamma = (\gamma - 1)\rho_i u_i, \tag{11.13}$$

where γ is the adiabatic index. Note that for isentropic flow (i.e. in the absence of shocks, and without mixing or thermal conduction) we expect the A_i to be constant. We hence define u_i, the thermal energy per unit mass, in terms of the density estimate as

$$u_i(\rho_i) = A_i \frac{\rho_i^{\gamma-1}}{\gamma - 1}. \tag{11.14}$$

This raises the question of how the smoothing lengths h_i needed for estimating ρ_i should be determined. As we discussed above, we would like to ensure adaptive kernel sizes, meaning that the number of points in the kernel should be approximately constant. In much of the older SPH literature, the number of neighbours was allowed to vary within some (small) range around a target number. Sometimes the smoothing length itself was evolved with a differential equation in time, exploiting the continuity relation and the expectation that ρh^3 should be approximately constant. In case the number of neighbours outside the kernel happened to fall outside the allowed range, h was suitably readjusted, at the price of some errors in energy conservation.

A better method is to require that the mass in the kernel volume should be constant, viz.

$$\rho_i h_i^3 = const \tag{11.15}$$

for three dimensions. Since $\rho_i = \rho_i(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N, h_i)$ is only a function of the particle coordinates and of h_i, this equation implicitly defines the function $h_i = h_i(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N)$ in terms of the particle coordinates.

We can then proceed to derive the equations of motion from

$$\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial L}{\partial \dot{\mathbf{r}}_i} - \frac{\partial L}{\partial \mathbf{r}_i} = 0. \tag{11.16}$$

This first gives

$$m_i \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \frac{P_j}{\rho_j^2} \frac{\partial \rho_j}{\partial \mathbf{r}_i}, \tag{11.17}$$

where the derivative $\partial\rho_j/\partial\mathbf{r}_i$ stands for the total variation of the density with respect to the coordinate $\mathbf{r}_i$, including any variation of h_j this may entail. We can hence write

$$\frac{\partial\rho_j}{\partial\mathbf{r}_i} = \nabla_i\rho_j + \frac{\partial\rho_j}{\partial h_j}\frac{\partial h_j}{\partial\mathbf{r}_i}, \tag{11.18}$$

where the smoothing length is kept constant in the first derivative on the right hand side (in our notation, the Nabla operator $\nabla_i = \partial/\partial\mathbf{r}_i$ means differentiation with respect to $\mathbf{r}_i$ holding the smoothing lengths constant). On the other hand, differentiation of $\rho_j h_j^3 = const$ with respect to $\mathbf{r}_i$ yields

$$\frac{\partial\rho_j}{\partial h_j}\frac{\partial h_j}{\partial\mathbf{r}_i}\left[1 + \frac{3\,\rho_j}{h_j}\left(\frac{\partial\rho_j}{\partial h_j}\right)^{-1}\right] = -\nabla_i\rho_j. \tag{11.19}$$

Combining equations (11.18) and (11.19) we then find

$$\frac{\partial\rho_j}{\partial\mathbf{r}_i} = \left(1 + \frac{h_j}{3\rho_j}\frac{\partial\rho_j}{\partial h_j}\right)^{-1}\nabla_i\rho_j. \tag{11.20}$$

Using

$$\nabla_i\rho_j = m_i\nabla_i W_{ij}(h_j) + \delta_{ij}\sum_{k=1}^{N} m_k\nabla_i W_{ki}(h_i)\,, \tag{11.21}$$

we finally obtain the equations of motion

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j\left[f_i\frac{P_i}{\rho_i^2}\nabla_i W_{ij}(h_i) + f_j\frac{P_j}{\rho_j^2}\nabla_i W_{ij}(h_j)\right], \tag{11.22}$$

where the f_i are defined by

$$f_i = \left[1 + \frac{h_i}{3\rho_i}\frac{\partial\rho_i}{\partial h_i}\right]^{-1}, \tag{11.23}$$

and the abbreviation $W_{ij}(h) = W(|\mathbf{r}_i - \mathbf{r}_j|, h)$ has been used. Note that the correction factors f_i can be easily calculated alongside the density estimate, all that is required is an additional summation to get $\partial\rho_i/\partial\mathbf{r}_i$ for each particle. This quantity is in fact also useful to get the correct smoothing radii by iteratively solving $\rho_i h_i^3 = const$ with a Newton-Raphson iteration.

The equations of motion (11.22) for inviscid hydrodynamics are remarkably simple. In essence, we have transformed a complicated system of partial differential equations into a much simpler set of ordinary differential equations. Furthermore, we only have to solve the momentum equation explicitly. The mass conservation equation as well as the total energy equation (and hence the thermal energy equation) are already taken care of, because the particle masses and their specific entropies stay constant for reversible gas dynamics. However, later we will introduce

an artificial viscosity that is needed to allow a treatment of shocks. This will introduce additional terms in the equation of motion and requires the time integration of one thermodynamic quantity per particle, which can either be chosen as entropy or thermal energy. Indeed, the above formulation can also be equivalently expressed in terms of thermal energy instead of entropy. This follows by taking the time derivative of Eqn. (11.14), which first yields

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = \frac{P_i}{\rho_i^2} \sum_j \mathbf{v}_j \cdot \frac{\partial \rho_i}{\partial \mathbf{r}_j}. \tag{11.24}$$

Using equations (11.20) and (11.21) then gives the evolution of the thermal energy as

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = f_i \frac{P_i}{\rho_i^2} \sum_j m_j (\mathbf{v}_i - \mathbf{v}_j) \cdot \nabla W_{ij}(h_i), \tag{11.25}$$

which needs to be integrated along the equation of motion if one wants to use the thermal energy as independent thermodynamic variable. There is no difference however to using the entropy; the two are completely equivalent in the variational formulation.

Note that the above formulation readily fulfills the conservation laws of energy, momentum and angular momentum. This can be shown based on the discretized form of the equations, but it is also manifest due to the symmetries of the Lagrangian that was used as a starting point. The absence of an explicit time dependence gives the energy conservation, the translational invariance implies momentum conservation, and the rotational invariance gives angular momentum conservation.

11.3 Artificial Viscosity

Even when starting from perfectly smooth initial conditions, the gas dynamics described by the Euler equations may readily produce true discontinuities in the form of shock waves and contact discontinuities. At such fronts the differential form of the Euler equations breaks down, and their integral form (equivalent to the conservation laws) needs to be used. At a shock front, this yields the Rankine-Hugoniot jump conditions that relate the upstream and downstream states of the fluid. These relations show that the specific entropy of the gas always increases at a shock front, implying that in the shock layer itself the gas dynamics can no longer be described as inviscid. In turn, this also implies that the discretized SPH equations we derived above can not correctly describe a shock, simply because they keep the entropy strictly constant.

One thus must allow for a modification of the dynamics at shocks and somehow introduce the necessary dissipation. This is usually accomplished in SPH by an artificial viscosity. Its purpose is to dissipate kinetic energy into heat and to produce entropy in the process. The usual approach is to parameterize the artificial viscosity in terms of a friction force that damps the relative motion of particles. Through

the viscosity, the shock is broadened into a resolvable layer, something that makes a description of the dynamics everywhere in terms of the differential form possible. It may seem a daunting task though to somehow tune the strength of the artificial viscosity such that just the right amount of entropy is generated in a shock. Fortunately, this is however relatively unproblematic. Provided the viscosity is introduced into the dynamics in a conservative fashion, the conservation laws themselves ensure that the right amount of dissipation occurs at a shock front.

What is more problematic is to devise the viscosity such that it is only active when there is really a shock present. If it also operates outside of shocks, even if only at a weak level, the dynamics may begin to deviate from that of an ideal gas.

The viscous force is most often added to the equation of motion as

$$\left.\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t}\right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij}\,, \tag{11.26}$$

where

$$\overline{W}_{ij} = \frac{1}{2}\left[W_{ij}(h_i) + W_{ij}(h_j)\right] \tag{11.27}$$

denotes a symmetrized kernel, which some researchers prefer to define as $\overline{W}_{ij} = W_{ij}([h_i + h_j]/2)$. Provided the viscosity factor Π_{ij} is symmetric in i and j, the viscous force between any pair of interacting particles will be antisymmetric and along the line joining the particles. Hence linear momentum and angular momentum are still preserved. In order to conserve total energy, we need to compensate the work done against the viscous force in the thermal reservoir, described either in terms of entropy,

$$\left.\frac{\mathrm{d}A_i}{\mathrm{d}t}\right|_{\mathrm{visc}} = \frac{1}{2}\frac{\gamma-1}{\rho_i^{\gamma-1}} \sum_{j=1}^{N} m_j \Pi_{ij} \mathbf{v}_{ij} \cdot \nabla_i \overline{W}_{ij}\,, \tag{11.28}$$

or in terms of thermal energy per unit mass,

$$\left.\frac{\mathrm{d}u_i}{\mathrm{d}t}\right|_{\mathrm{visc}} = \frac{1}{2} \sum_{j=1}^{N} m_j \Pi_{ij} \mathbf{v}_{ij} \cdot \nabla_i \overline{W}_{ij}\,, \tag{11.29}$$

where $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$. There is substantial freedom in the detailed parametrization of the viscosity Π_{ij}. The most commonly used formulation of the viscosity is

$$\Pi_{ij} = \begin{cases} \left[-\alpha c_{ij}\mu_{ij} + \beta\mu_{ij}^2\right]/\rho_{ij} & \text{if } \mathbf{v}_{ij}\cdot\mathbf{r}_{ij} < 0 \\ 0 & \text{otherwise,} \end{cases} \tag{11.30}$$

with

$$\mu_{ij} = \frac{h_{ij}\,\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^2 + \epsilon h_{ij}^2}. \tag{11.31}$$

Here h_{ij} and ρ_{ij} denote arithmetic means of the corresponding quantities for the two particles i and j, with c_{ij} giving the mean sound speed, whereas $\mathbf{r}_{ij} \equiv \mathbf{r}_i - \mathbf{r}_j$.

The strength of the viscosity is regulated by the parameters α and β, with typical values in the range $\alpha \simeq 0.5 - 1.0$ and the frequent choice of $\beta = 2\,\alpha$. The parameter $\epsilon \simeq 0.01$ is introduced to protect against singularities if two particles happen to get very close.

In this form, the artificial viscosity is basically a combination of a bulk and a von Neumann-Richtmyer viscosity. Historically, the quadratic term in μ_{ij} has been added to the original Monaghan-Gingold form to prevent particle penetration in high Mach number shocks. Note that the viscosity only acts for particles that rapidly approach each other, hence the entropy production is always positive definite. Also, the viscosity vanishes for solid-body rotation, but not for pure shear flows. To cure this problem in shear flows, Balsara (1995) suggested adding a correction factor to the viscosity, reducing its strength when the shear is strong. This can be achieved by multiplying Π_{ij} with a prefactor $(f_i^{\rm AV} + f_j^{\rm AV})/2$, where the factors

$$f_i^{\rm AV} = \frac{|\nabla \cdot \mathbf{v}|_i}{|\nabla \cdot \mathbf{v}|_i + |\nabla \times \mathbf{v}|_i} \tag{11.32}$$

are meant to measure the rate of local compression in relation to the strength of the local shear (estimated with formulas such as Eqn. 11.7).

In some studies, alternative forms of viscosity have been tested. For example, Monaghan (1997) proposed a modified form of the viscosity which can be written as

$$\Pi_{ij} = -\frac{\alpha}{2}\frac{v_{ij}^{\rm sig} w_{ij}}{\rho_{ij}}, \tag{11.33}$$

where $v_{ij}^{\rm sig} = [c_i + c_j - 3w_{ij}]$ is an estimate of the signal velocity between two particles i and j, and $w_{ij} = \mathbf{v}_{ij} \cdot \mathbf{r}_{ij}/|\mathbf{r}_{ij}|$ is the relative velocity projected onto the separation vector.

In attempting to reduce the numerical viscosity of SPH in regions away from shocks, several studies have recently advanced the idea of keeping the functional form of the artificial viscosity, but making the viscosity strength parameter α variable in time. Adopting $\beta = 2\alpha$, one may evolve the parameter α individually for each particle with an equation such as

$$\frac{{\rm d}\alpha_i}{{\rm d}t} = -\frac{\alpha_i - \alpha_{\rm max}}{\tau_i} + S_i, \tag{11.34}$$

where S_i is some source function meant to ramp up the viscosity rapidly if a shock is detected, while the first term lets the viscosity exponentially decay again to a prescribed minimum value $\alpha_{\rm min}$ on a timescale τ_i. So far, simple source functions like $S_i = \max[-(\nabla \cdot \mathbf{v})_i, 0]$ and timescales $\tau_i \simeq h_i/c_i$ have been explored and the viscosity α_i has often also been prevented from becoming higher than some prescribed maximum value $\alpha_{\rm max}$. It is clear that the success of such a variable α-scheme depends critically on an appropriate source function. The form above can still not distinguish purely adiabatic compression from that in a shock, so is not completely free of creating unwanted viscosity.

11.4 Advantages and disadvantages of SPH

Smoothed particle hydrodynamics is a remarkably versatile and simple approach for numerical fluid dynamics. The ease with which it can provide a large dynamic range in spatial resolution and density, as well as an automatically adaptive resolution, are unmatched in Eulerian methods. At the same time, SPH has excellent conservation properties, not only for energy and linear momentum, but also for angular momentum. The latter is not automatically guaranteed in Eulerian codes, even though it is usually fulfilled at an acceptable level for well-resolved flows. When coupled to self-gravity, SPH conserves the total energy exactly, which is again not manifestly true in most mesh-based approaches to hydrodynamics. Finally, SPH is Galilean-invariant and free of any errors from advection alone, which is another advantage compared to Eulerian mesh-based approaches.

Thanks to its completely mesh-free nature, SPH can easily deal with complicated geometric settings and large regions of space that are completely devoid of particles. Implementations of SPH in a numerical code tend to be comparatively simple and transparent. At the same time, the scheme is characterized by remarkable robustness. For example, negative densities or negative temperatures, sometimes a problem in mesh-based codes, can not occur in SPH by construction. Although shock waves are broadened in SPH, the properties of the post-shock flow are correct.

The main disadvantage of SPH is its limited accuracy in multi-dimensional flows. One source of noise originates in the approximation of local kernel interpolants through discrete sums over a small set of nearest neighbours. While in 1D the consequences of this noise tend to be reasonably benign, particle motion in multiple dimensions has a much higher degree of freedom. Here the mutually repulsive forces of pressurized neighbouring particle pairs do not easily cancel in all dimensions simultaneously, especially not given the errors of the discretized kernel interpolants. As a result, some 'jitter' in the particle motions readily develops, giving rise to velocity noise up to a few percent of the local sound speed. This noise seriously messes up the accuracy that can be reached with the technique, especially for subsonic flow, and also leads to a slow convergence rate.

12 Finite element methods

Finite element methods (FEM) are a general class of methods for solving partial differential equations. They are very popular in engineering disciplines (here it is also sometimes called FEA, finite element analysis), and in some more advanced forms also in applied mathematics. The particular strengths of FFM lie in its ability to easily work with flexible geometries, to cope with geometrically odd boundary conditions, and to conveniently allow a spatially variable resolution.

12.1 Classic finite element methods for linear PDEs

Conceptually simplest are FEM methods applied to linear partial differential equations. They involve the following characteristics:

- The central idea of FEMs is to divide the solution into smaller regions that we call *elements*. The element contains a certain number of points that we call *nodes*. Different shapes for the boundaries are possible:
  - simple segments (in 1D)
  - triangles, rectangles, etc. (in 2D)
  - tetrahedra, octahedra, cubes, etc. (in 3D)

- A set of basis functions is chosen to describe the solution on each element. These basis functions need to be linearly independent but not necessarily orthogonal to each other. Most commonly employed is a polynomial basis. For instance, for an element we may set

$$\phi(x) = a_0 + a_1 x, \tag{12.1}$$

  which would be a linear element, or we use

$$\phi(x) = a_0 + a_1 x + a_2 x^2 \tag{12.2}$$

  for a quadratic element.

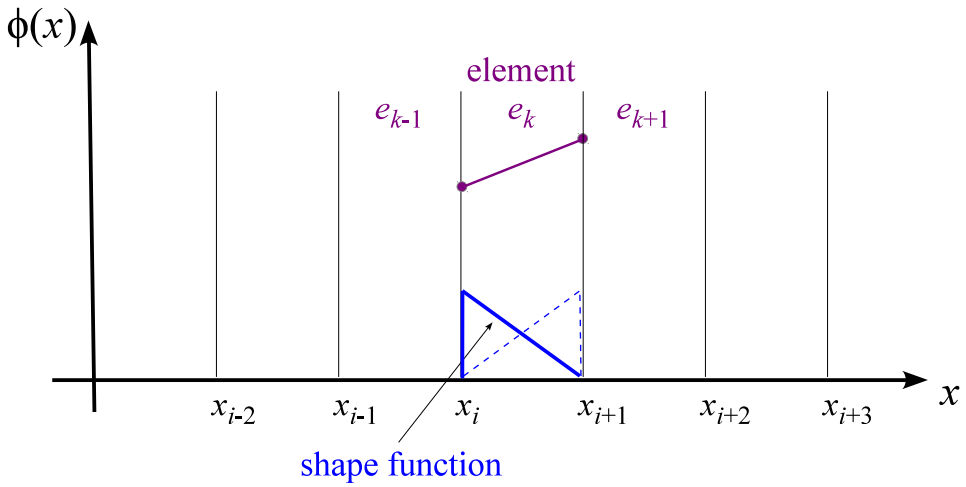
- If we have n coefficients in the element expansion, then we also need n nodes to determine them, and hence to fully specify the "reconstruction" inside the element.

- If $\phi_1, \ldots, \phi_n$ are the values at the nodes, one can also write the element as

$$\phi(x) = \phi_1 N_1(x) + \phi_2 N_2(x) + \ldots + \phi_n N_n(x), \tag{12.3}$$

  where $N_n(x)$ are the so-called shape functions. They live only on the elements themselves and are zero outside of them.

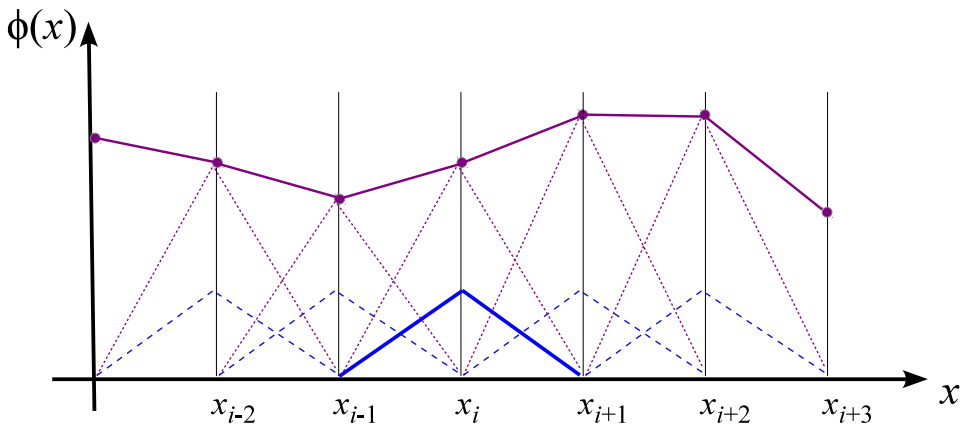
In 1D, linear shape functions are often used:



Then:

$$\phi^{(k)}(x) = \phi_i S_i^{(k)}(x) + \phi_{i+1} S_{i+1}^{(k)}(x). \tag{12.4}$$

But since left and right of each interface we have triangle halves scaled by the ϕ's, we can also require continuity of the global reconstruction (which is not in general necessary), effectively obtaining a basis of triangules that are two elements wide. This then yields a piece-wise linear approximation supported by scaled triangular shape functions:



The question however is how the expansion coefficients should be determined. As we will see, this can be done by transforming the PDE into a set of algebraic equations for the expansion coefficients ϕ_i. Suppose we have a linear PDE written in the form

$$L\hat{\phi} + \hat{s} = 0, \tag{12.5}$$

where L is some linear differential operator, $\hat{\phi}(x)$ is the solution, and $\hat{s}(x)$ is the real source function. If we plug in our finite element approximations $\phi(x)$ and $s(x)$, the residual becomes

$$R^{(k)}(x; \phi_1, \ldots, \phi_n) = L\phi + s, \tag{12.6}$$

which is now not necessarily zero any more because ϕ and s are approximations to the real solution and source function, respectively.

We now would like to choose the expansion coefficients such that the residual is in some sense minimal. There are several possibilities for this:

- One could ask for
$$\int R(x; \phi_i)\, \mathrm{d}x = 0, \tag{12.7}$$
which is known as "Ritz method".

- More generally, one can use a so-called *weighted residual method*. Here one introduces a set of fairly arbitrary weight functions $w_i(x)$ with $i = 1, 2, \ldots, n$, and then demands that
$$\int_{\text{domain}} R(x; \phi_i)\, \mathrm{d}x = 0, \tag{12.8}$$
for each of the weights. Some common choices for these weight functions include:
  - **Collocation method:** Here we ask the residuum to vanish at n points $\{x_i\}$ inside the domain. This is equivalent to setting $w_i(x) = \delta(x - x_i)$.
  - **Least-square method:** We set $w_i = \frac{\partial R}{\partial \phi_i}$, which is equivalent to minimizing $\int R^2\, \mathrm{d}x$.
  - **Galerkin method:** We set $w_i(x) = N_i(x)$, i.e. we choose the basis functions themselves as weights:
  $$\int_{\text{domain}} R(x; \phi_i)\, N_i(x)\, \mathrm{d}x = 0 \tag{12.9}$$
  Note that this corresponds to N equations that are linear in the N unknowns ϕ_i. Solving for the coefficients now becomes equivalent to solving an ordinary linear system of equations,
  $$\mathbf{A}\phi = \mathbf{b}. \tag{12.10}$$

The matrix $\mathbf{A}$ is however extremely sparse because the expansion elements are highly localized. For this system, the problem can be solved efficiently with linear algebra methods for sparse matrices. The Galerkin approach is the one usually adopted in FEM.

- Once the algebraic equations approximating the PDE in the FEM approach are found, the coefficients can be computed with a sparse matrix solver.
- Through the element expansion, one then in fact obtains an approximation to the solution at every point.

The scheme generalizes straightforwardly to 2D and 3D, as well as to higher dimensions. Also, the polynomial degree used for the approximations can be increased, if desired. One of the primary advantages of the FEM is the flexibility of the employed shapes. One can easily use irregular triangulations! This is for example exploited in structural engineering to model the shapes of aircraft wings, houses, bridges, etc. Clearly, the calculation of the mesh element overlap becomes very tedious for such irregular geometries. In practice, there are however good software packages that can do that efficiently even for complex geometries.

Finally, let us again repeat the major limitation of FEM, namely that it is only good for linear PDEs. One can sometimes linearize a non-linear PDE (for example by treating parts of the solution as constant used for guessing), in which case FEM can still be applied.

Example

Consider the 1D Poisson equation:

$$\phi''(x) = 4\pi G\rho(x) \tag{12.11}$$

with von Neumann boundary conditions, $\phi'(x_L) = \phi'(x_R) = 0$. We discretize this with N points spread Δx apart, using linear elements. Then the shape functions are triangles with

$$S_i(x) = \begin{cases} \frac{x-x_{i-1}}{\Delta x} & \text{for } \; x \in [x_{i-1}, x_i], \\ \frac{x_{i+1}-x}{\Delta x} & \text{for } \; x \in [x_i, x_{i+1}], \\ 0 & \text{otherwise.} \end{cases} \tag{12.12}$$

We now demand

$$\int_{x_L}^{x_R} \left[\phi'' - 4\pi G\rho(x)\right] \, S_i(x) = 0. \tag{12.13}$$

We can do an integration by parts in the first term:

$$\int_{x_L}^{x_R} \phi' S_i' \, \mathrm{d}x = -\int_{x_L}^{x_R} 4\pi G\rho S_i(x) \, \mathrm{d}x = b_i. \tag{12.14}$$

Finally, we insert our expansion, $\phi(x) = \sum \phi_i S_i(x)$, and obtain

$$\int_{x_L}^{x_R} \sum_j \phi_j S_j'(x) S_i'(x)\,\mathrm{d}x = b_i. \tag{12.15}$$

Let's then define

$$A_{ij} \equiv \int S_i'(x) S_j'(x)\,\mathrm{d}x = \begin{cases} \frac{2}{\Delta x} & \text{for } i = j, \\ \frac{2}{\Delta x} & \text{for } i = j \pm 1, \\ 0 & \text{otherwise.} \end{cases} \tag{12.16}$$

And hence the linear set of equations we want to solve becomes

$$\sum_j A_{ij}\phi_j = b_i, \tag{12.17}$$

which is here actually equivalent to what one obtains from a simple finite difference approximation that we considered earlier in the lecture.

12.2 Discontinuous Galerkin methods

A very powerful FEM approach is to solve a PDE system in a so-called weak formulation in which one gives up on requiring continuity of the solution across elements. In this form, one obtaines discontinuous Galerkin (DG) schemes that can be applied, for example, to the non-linear hyperbolic PDE system of the Euler equations. A particular advantage of DG is a systematic path to derive high-order methods, which is difficult to obtain in classic finite volume methods, where already 3rd order begins to become unwieldy. Also, DG approaches have been shown to be more accurate than finite volume schemes at equivalent computational cost, suggesting that they are also more efficient.

Here we shall use the Euler equations to introduce such a state-of-the-art DG formulation (based on Schaal et al., 2015). We recall that the Euler equations are a system of hyperbolic partial differential equations and can be written in compact form as

$$\frac{\partial \mathbf{u}}{\partial t} + \sum_{\alpha=1}^{3} \frac{\partial \mathbf{f}_\alpha}{\partial x_\alpha} = 0, \tag{12.18}$$

with the state vector

$$\mathbf{u} = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ \rho e \end{pmatrix} = \begin{pmatrix} \rho \\ \rho\mathbf{v} \\ \rho u + \frac{1}{2}\rho\mathbf{v}^2 \end{pmatrix}, \tag{12.19}$$

and the flux vectors

$$\mathbf{f}_1 = \begin{pmatrix} \rho v_1 \\ \rho v_1^2 + p \\ \rho v_1 v_2 \\ \rho v_1 v_3 \\ (\rho e + p) v_1 \end{pmatrix} \quad \mathbf{f}_2 = \begin{pmatrix} \rho v_2 \\ \rho v_1 v_2 \\ \rho v_2^2 + p \\ \rho v_2 v_3 \\ (\rho e + p) v_2 \end{pmatrix} \quad \mathbf{f}_3 = \begin{pmatrix} \rho v_3 \\ \rho v_1 v_3 \\ \rho v_2 v_3 \\ \rho v_3^2 + p \\ (\rho e + p) v_3 \end{pmatrix}. \tag{12.20}$$

The unknown quantities are density ρ, velocity $\mathbf{v}$, pressure p, and total energy per unit mass e. The latter can be expressed in terms of the internal energy per unit mass u and the kinetic energy of the fluid, $e = u + \frac{1}{2}\mathbf{v}^2$. For an ideal gas, the system is closed with the equation of state

$$p = \rho u(\gamma - 1), \tag{12.21}$$

where γ denotes the adiabatic index.

12.2.1 Solution representation

Lets assume that we partition the domain into elements consisting of non-overlapping cubical cells, which may correspond to a Cartesian mesh of constant resolution, or to an adaptively refined nested grid. Moreover, we follow the approach of a classical modal DG scheme, where the solution in the interior of cell K is given by a linear combination of $N(k)$ orthogonal and normalized basis functions ϕ_l^K:

$$\mathbf{u}^K(\mathbf{x}, t) = \sum_{l=1}^{N(k)} \mathbf{w}_l^K(t) \phi_l^K(\mathbf{x}). \tag{12.22}$$

In this way, the dependence on time and space of the solution is split into time-dependent weights, and basis functions which are constant in time. Consequently, the state of a cell is completely characterized by the $N(k)$ weight vectors $\mathbf{w}_j^K(t)$.

The above equation can be solved for the weights by multiplying with the corresponding basis function ϕ_j^K and integrating over the cell volume. Using the orthogonality and normalization of the basis functions yields

$$\mathbf{w}_j^K = \frac{1}{|K|} \int_K \mathbf{u}^K \phi_j^K \, \mathrm{d}V, \quad j = 1, \ldots, N(k), \tag{12.23}$$

where $|K|$ is the volume of the cell. The first basis function is chosen to be $\phi_1 = 1$ and hence the weight $\mathbf{w}_1^K$ is the cell average of the state vector $\mathbf{u}^K$. The higher order moments of the state vector are described by weights $\mathbf{w}_j^K$ with $j \geq 2$.

The basis functions can be defined on a cube in terms of scaled variables ξ,

$$\phi_l(\xi) : [-1, 1]^3 \to \mathbb{R}. \tag{12.24}$$

The transformation between coordinates ξ in the cell frame of reference and coordinates $\mathbf{x}$ in the laboratory frame of reference is

$$\xi = \frac{2}{\Delta x^K}(\mathbf{x} - \mathbf{x}^K), \tag{12.25}$$

where Δx^K and $\mathbf{x}^K$ are the edge length and cell centre of cell K, respectively. One potential basis is obtained by constructing a set of three-dimensional polynomial basis functions with a maximum degree of k as products of one-dimensional scaled Legendre polynomials $\tilde{P}$:

$$\{\phi_l(\xi)\}_{l=1}^{N(k)} = \left\{ \tilde{P}_u(\xi_1)\tilde{P}_v(\xi_2)\tilde{P}_w(\xi_3) | u, v, w \in \mathbb{N}_0 \wedge u + v + w \leq k \right\}. \tag{12.26}$$

The first few Legendre polynomials are

$$\begin{aligned} P_0(\xi) &= 1 & P_1(\xi) &= \xi \\ P_2(\xi) &= \frac{1}{2}(3\xi^2 - 1) & P_3(\xi) &= \frac{1}{2}(5\xi^3 - 3\xi) \\ P_4(\xi) &= \frac{1}{8}(35\xi^4 - 30\xi^2 + 3) & P_5(\xi) &= \frac{1}{8}(63\xi^5 - 70\xi^3 + 15\xi). \end{aligned} \tag{12.27}$$

and are obtained as solutions of Legendre's differential equation:

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\left[(1-\xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}P_n(\xi)\right] + n(n+1)P_n(\xi) = 0, \quad n \in \mathbb{N}_0. \tag{12.28}$$

They are pairwise orthogonal to each other. Moreover, we define scaled polynomials as

$$\tilde{P}(\xi)_n = \sqrt{2n+1}P(\xi)_n, \tag{12.29}$$

such that

$$\int_{-1}^{1} \tilde{P}_i(\xi)\tilde{P}_j(\xi)\,\mathrm{d}\xi = \begin{cases} 0 & \text{if } i \neq j \\ 2 & \text{if } i = j. \end{cases} \tag{12.30}$$

The number of basis functions for polynomials with a maximum degree of k is

$$N(k) = \sum_{u=0}^{k}\sum_{v=0}^{k-u}\sum_{w=0}^{k-u-v} 1 = \frac{1}{6}(k+1)(k+2)(k+3). \tag{12.31}$$

We note that when polynomials with a maximum degree of k are used, a scheme with spatial order $p = k + 1$ is obtained. For example, piece-wise linear basis functions ($k = 1$) lead to a scheme which is of second order in space.

12.2.2 Initial conditions

Given initial conditions $\mathbf{u}(\mathbf{x}, t = 0) = \mathbf{u}(\mathbf{x}, 0)$, we have to provide an initial state for the DG scheme which is consistent with the solution representation. To this end, the initial conditions can be expressed by means of the polynomial basis on cell K, which will then be

$$\mathbf{u}^K(\mathbf{x}, 0) = \sum_{l=1}^{N(k)} \mathbf{w}_l^K(0)\phi_l^K(\mathbf{x}). \tag{12.32}$$

If the initial conditions at hand are polynomials with degree $\leq k$, this representation preserves the exact initial conditions, otherwise equation (12.32) is an approximation to the given initial conditions. Otherwise, optimal initial weights can be obtained by performing an L^2-projection,

$$\min_{\left\{w_{l,i}^K(0)\right\}_l} \int_K \left(u_i^K(\mathbf{x}, 0) - u_i(\mathbf{x}, 0)\right)^2 \mathrm{d}V, \tag{12.33}$$

where $i = 1, \ldots, 5$ enumerates the conserved variables. The projection above leads to the integral

$$\mathbf{w}_j^K(0) = \frac{1}{|K|} \int_K \mathbf{u}(\mathbf{x}, 0)\phi_j^K(\mathbf{x}) \,\mathrm{d}V, \quad j = 1, \ldots, N(k), \tag{12.34}$$

which can be transformed to the reference frame of the cell, viz.

$$\mathbf{w}_j^K(0) = \frac{1}{8} \int_{[-1,1]^3} \mathbf{u}(\xi, 0)\phi_j(\xi) \,\mathrm{d}\xi, \quad j = 1, \ldots, N(k). \tag{12.35}$$

This integral can be computed numerically by means of tensor product Gauss-Legendre quadrature (hereafter called Gaussian quadrature) with $(k+1)^3$ nodes:

$$\mathbf{w}_j^K(0) \approx \frac{1}{8} \sum_{q=1}^{(k+1)^3} \mathbf{u}(\xi_q^{3\mathrm{D}}, 0)\phi_j(\xi_q^{3\mathrm{D}})\omega_q^{3\mathrm{D}}, \quad j = 1, \ldots, N(k). \tag{12.36}$$

Here, $\xi_q^{3\mathrm{D}}$ is the position of the quadrature node q in the cell frame of reference, and $\omega_q^{3\mathrm{D}}$ denotes the corresponding quadrature weight.

12.2.3 Gauss-Legendre quadrature

In the technique of Gaussian quadrature, the numerical integration of a function $f : [-1, +1] \to \mathbb{R}$ is approximated with a Gaussian quadrature rule involving n points and a weighted sum,

$$\int_{-1}^{+1} f(\xi) \,\mathrm{d}\xi \approx \sum_{q=1}^{n} f(\xi_q^{1\mathrm{D}})\omega_q^{1\mathrm{D}}. \tag{12.37}$$

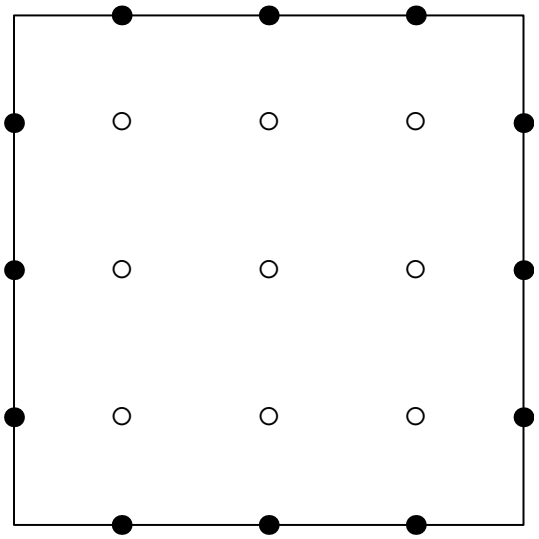


Figure 12.1: In the DG scheme, a surface and a volume integral has to be computed numerically for every cell, see equation (12.41). These integrals can be done by means of Gauss-Legendre quadrature. This example shows the nodes for a third order DG method (with second order polynomials) when used in a two-dimensional configuration. The black nodes indicate the positions where the surface integral is evaluated, which involves a numerical flux calculation with a Riemann solver. The white nodes are used for numerically estimating the volume integral.

Here, $\xi_q^{\mathrm{1D}} \in (-1,+1)$ are the Gaussian quadrature nodes and ω_q^{1D} are the corresponding weights. To integrate a 2D function $f : [-1,+1]^2 \to \mathbb{R}$ the tensor product of the n Gauss points can be used, viz.

$$\begin{aligned}&\int_{-1}^{+1}\int_{-1}^{+1} f(\xi_1,\xi_2)\,\mathrm{d}\xi_1\mathrm{d}\xi_2\\ &\approx \sum_{q=1}^{n}\sum_{r=1}^{n} f(\xi_{1,q}^{\mathrm{1D}},\xi_{2,r}^{\mathrm{1D}})\omega_q^{\mathrm{1D}}\omega_r^{\mathrm{1D}} = \sum_{q=1}^{n^2} f(\xi_q^{\mathrm{2D}})\omega_q^{\mathrm{2D}}.\end{aligned} \tag{12.38}$$

The n-point Gaussian quadrature rule is exact for polynomials of degree up to $2n-1$ when the one-dimensional nodes are given as the roots of the Legendre polynomial $P_n(\xi)$. These roots need to be found numerically, e.g. by means of the Newton-Raphson method, and can then be tabulated. The corresponding weights are then calculated as

$$\omega_q = \frac{2}{(1-\xi_q^2)P_n'(\xi_q)^2}, \quad q = 1,\ldots,n. \tag{12.39}$$

12.2.4 Evolution equation for the weights

In order to derive the DG scheme on a cell K, the Euler equations for a polynomial state vector $\mathbf{u}^K$ are multiplied by the basis function ϕ_j^K and integrated over the cell volume,

$$\int_K \left[\frac{\partial \mathbf{u}^K}{\partial t} + \sum_{\alpha=1}^{3} \frac{\partial \mathbf{f}_\alpha}{\partial x_\alpha} \right] \phi_j^K \, \mathrm{d}V = 0, \tag{12.40}$$

yielding a weak formulation of the PDE. Integration by parts of the flux divergence term and a subsequent application of Gauss' theorem leads to

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_K \mathbf{u}^K \phi_j^K \, \mathrm{d}V - \sum_{\alpha=1}^{3} \int_K \mathbf{f}_\alpha \frac{\partial \phi_j^K}{\partial x_\alpha} \, \mathrm{d}V + \sum_{\alpha=1}^{3} \int_{\partial K} \mathbf{f}_\alpha n_\alpha \phi_j^K \, \mathrm{d}S = 0, \tag{12.41}$$

where $\hat{n} = (n_1, n_2, n_3)^T$ denotes the outward pointing unit normal vector of the surface ∂K. In the following, we discuss each of the terms separately.

According to equation (12.23) the first term is simply the time variation of the weights,

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_K \mathbf{u}^K \phi_j^K \, \mathrm{d}V = |K| \frac{\mathrm{d}\mathbf{w}_j^K}{\mathrm{d}t}. \tag{12.42}$$

The second and third terms are discretized by transforming the integrals to the cell frame and applying Gaussian quadrature (Fig. 12.1). With this procedure the second term becomes

$$\begin{aligned} &\sum_{\alpha=1}^{3} \int_K \mathbf{f}_\alpha \left(\mathbf{u}^K(\mathbf{x}, t) \right) \frac{\partial \phi_j^K(\mathbf{x})}{\partial x_\alpha} \, \mathrm{d}V \\ &= \frac{(\Delta x^K)^2}{4} \sum_{\alpha=1}^{3} \int_{[-1,1]^3} \mathbf{f}_\alpha \left(\mathbf{u}^K(\xi, t) \right) \frac{\partial \phi_j(\xi)}{\partial \xi_\alpha} \, \mathrm{d}\xi \\ &\approx \frac{(\Delta x^K)^2}{4} \sum_{\alpha=1}^{3} \sum_{q=1}^{(k+1)^3} \mathbf{f}_\alpha \left(\mathbf{u}^K(\xi_q^{3\mathrm{D}}, t) \right) \frac{\partial \phi_j(\xi)}{\partial \xi_\alpha} \bigg|_{\xi_q^{3\mathrm{D}}} \omega_q^{3\mathrm{D}}. \end{aligned} \tag{12.43}$$

Note that the transformation of the derivative $\partial / \partial x_\alpha$ gives a factor of 2, see equation (12.25). The volume integral is computed by Gaussian quadrature with $k+1$ nodes per dimension. These nodes allow the exact integration of polynomials up to degree $2k+1$.

While the flux functions $\mathbf{f}_\alpha$ in the above expression can be evaluated analytically, this is not the case for the fluxes in the last term of the evolution equation (12.41). This is because the solution is *discontinuous* across cell interfaces. We hence have to introduce a numerical flux function $\overline{\mathbf{f}} \left(\mathbf{u}^{K-}, \mathbf{u}^{K+}, \hat{n} \right)$, which in general depends on both states left and right of the interface and on the normal vector. With this numerical flux, the third term in equation (12.41) takes the form

$$\begin{aligned} &\sum_{\alpha=1}^{3} \int_{\partial K} \mathbf{f}_\alpha n_\alpha(\mathbf{x}) \phi_j^K(\mathbf{x}) \, \mathrm{d}S \\ &= \frac{(\Delta x^K)^2}{4} \int_{\partial [-1,1]^3} \overline{\mathbf{f}} \left(\mathbf{u}^{K-}(\xi, t), \mathbf{u}^{K+}(\xi, t), \hat{n}(\xi) \right) \phi_j(\xi) \, \mathrm{d}S_\xi \\ &\approx \frac{(\Delta x^K)^2}{4} \sum_{A \in \partial [-1,1]^3} \sum_{q=1}^{(k+1)^2} \overline{\mathbf{f}} \left(\mathbf{u}^{K-}(\xi_{q,A}^{2\mathrm{D}}, t), \mathbf{u}^{K+}(\xi_{q,A}^{2\mathrm{D}}, t), \hat{n} \right) \phi_j(\xi_{q,A}^{2\mathrm{D}}) \omega_q^{2\mathrm{D}} \end{aligned} \tag{12.44}$$

Here for each interface of the normalized cell a two-dimensional Gaussian quadrature rule with $(k+1)^2$ nodes is applied. The numerical flux across each node can be calculated with a one-dimensional Riemann solver, as in ordinary Godunov schemes.

We have now discussed each term of the basic equation (12.41) and arrived at a spatial discretization of the Euler equations of the form

$$\frac{\mathrm{d}\mathbf{w}_j^K}{\mathrm{d}t} + \mathbf{R}_K = 0, \quad j = 1, \ldots, N(k), \tag{12.45}$$

which represents a system of coupled ordinary differential equations. What is left to do is to intergrate them forward in time with a suitable integration scheme. Normally Runge-Kutta schemes are used for this, which can also be systematically extended to higher order so that the order of the time integration matches that of the spatial discretization. Additionally, problems involving strong shock waves usually require so-called limiting schemes that damp oscillatory parts of the oscillations should they be excited by true discontinuities in the physical solution. The limiting may involve reducing or setting to zero the higher order expansion coefficients in elements directly adjacent to a detected discontinuity.

12.2.5 Efficiency of DG schemes

In the above, the order p of the scheme can be chosen relatively freely, unlike in common finite volume approaches. If one has the goal to compute a more accurate numerical solution for a given problem, one has in principle two options. One either chooses a finer grid with a smaller spacing h (so called h-refinement) or one increases the order of the scheme (so called p-refinement). Note that in both cases, the number of degrees of freedom (which are the total number of weight coefficients that one evolves in time) goes up. An interesting question is then whether it is better to make the grid finer, or to go to higher order, where "better" typically means lower computational cost, i.e. less CPU time.

A general answer to this question cannot be given as it is problem-dependent. However, one can systematically study the situation for a well defined problem. One such target problem is a stationary, isentropic vortex flow, the so-called Yee vortex. Here the analytic solution is known (which is identical to the initial conditions due to the stationarity), so that one can compute an error norm for a numerical solution by calculating the integral

$$\mathrm{L1} = \frac{1}{V}\int_V |\rho'(x,y) - \rho^0(x,y)|\,\mathrm{d}V, . \tag{12.46}$$

with the density solution polynomial $\rho'(x,y)$ and the analytic expression $\rho^0(x,y)$ of the initial conditions.

In Figure 12.2 results are showns where this problem was simulated in 2D for some time for 2nd, 3rd, and 4th order DG, and compared to solutions obtained with a 2nd order finite volume approach. The error norm is then shown as a function of

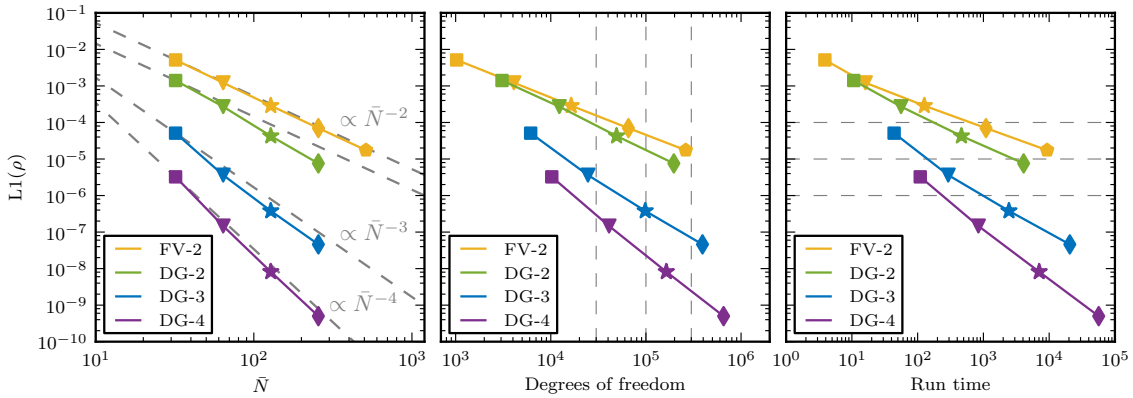


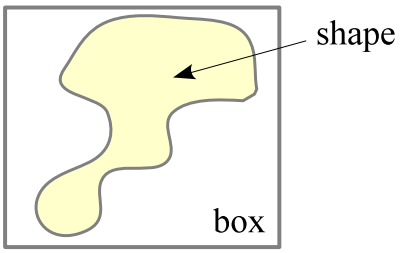
Figure 12.2: Performance of DG for the isentropic vortex flow (Schaal et al., 2015). *Left panel:* L1 error norm as a function of linear resolution for a two-dimensional isentropic vortex test. Each data point corresponds to a simulation, different colours indicate the different methods. The convergence rate is as expected (dashed line) or slightly better for all schemes. *Middle panel:* The same simulation errors as a function of degrees of freedom (DOF), which is an indicator for the memory requirements. *Right panel:* L1 error norm versus the measured run time of the simulations. The second order FV implementation (FV-2) and the second order DG (DG-2) realization are approximately equally efficient in this test, i.e. a given precision can be obtained with a similar computational cost. In comparison, the higher order methods can easily be faster by more than an order of magnitude for this smooth problem. This illustrates the fact that an increase of order (p-refinement) of the numerical scheme can be remarkably more efficient than a simple increase of grid resolution (h-refinement).

the number of elements used per dimension, demonstrating the different convergence order of the schemes. Also shown are is the error as a function of the number of degrees used, and finally as a function of the CPU time. It is clearly seen that in this case the 4th order DG scheme wins, i.e. it provides for a given investment of CPU time the highest accuracy.

13 Monte Carlo Techniques

13.1 Monte Carlo Integration

So-called Monte Carlo integrations lie at the heart of many stochastic simulation methods. The basic idea can be intuitively understood with the "dartboard method" of integrating the area of an irregular domain.



This works as follows:

- Choose points randomly (i.e. uniformly) within the box.
- We know that the probability that a point hits inside the area is proportional to the ratio of the areas:

$$p(\text{dart hits inside area}) = \frac{A_{\text{shape}}}{A_{\text{box}}} \tag{13.1}$$

- We can now approximate this probability, and hence the area ratio, through the experimental result:

$$\frac{A_{\text{shape}}}{A_{\text{box}}} \simeq \frac{\#\text{hits in shape}}{\#\text{hits in box}} \tag{13.2}$$

  This is expected to become arbitrarily accurate as the number of trials goes to infinity.

Standard Monte Carlo integration

Let us now formalize this technique. We consider an integral in d-dimensions,

$$I = \int_V f(\mathbf{x})\, \mathrm{d}^d\mathbf{x}, \tag{13.3}$$

where V is a d-dimensional hypercube with (for simplicity) dimensions $[0,1]^d$. To compute this as a Monte Carlo integral, we do the following:

1. Generate N random vectors $0 \leq \mathbf{x}_i \leq 1$ with flat distribution (i.e. each component of the vector is drawn independently from a uniform distribution).

2. We compute
$$I_N = \frac{V}{N} \sum_i^N f(\mathbf{x}_i). \tag{13.4}$$
For $N \to \infty$, we then get $I_n \to I$.

3. The error of the result scales as $1/\sqrt{N}$, *independent* of the number of dimensions of the integral.

Especially the last point is quite remarkable – we'll later have to look at this in more detail. Before we do this, it is instructive to compare with the steps taken in standard integration techniques. In them, we divide each dimension in n regularly spaced points. The total number of points is hence $N = n^d$. Depending on the integration rule selected, the error will then scale as some power of $1/n$. For example, for the midpoint and trapezoidal rules, it will simply be $\propto 1/n^2$, and for Simpson's rule $\propto 1/n^4$.

If d is small, it is clear that the Monte Carlo integration has much larger errors than standard methods when the same N is used. However, the higher d becomes, the better Monte Carlos looks because then the standard method can spend comparatively fewer regular sampling points per dimension.

One can then for example ask: At what point is Monte Carlo as good as Simpson? Well, Simpson's error should scale as
$$\frac{1}{n^4} = \frac{1}{N^{4/d}}, \tag{13.5}$$
which starts to decline with N more weakly than Monte Carlo integration when $d \geq 8$. We hence clearly see that high dimensional problems are the regime where Monte Carlo integration becomes particularly interesting.

In fact, in some lattice simulations one has dimensions in the range $d = 10^6 - 10^{10}$. Here the only viable approach is to use Monte Carlo integration. In practice, standard methods fail already at much more moderate numbers of dimensions. For example, even with $d = 10$, putting down just a grid with $n = 10$ cells per dimension already yields a number of $N = 10^{10}$ grid points.

13.2 Error in Monte Carlo integration

Let $y_i = V f(\mathbf{x}_i)$ be the value of the i-th function evaluation of our Monte Carlo integration. After N samples, we can thus write the approximation to the desired integral as
$$I_N = \frac{y_1 + y_2 + \ldots + y_N}{N} \tag{13.6}$$

The error of this quantity is *defined* as the width of its probability distribution $P_N(I_N)$, i.e.

$$\sigma_N^2 \equiv \left\langle I_N^2 \right\rangle - \langle I_N \rangle^2 . \tag{13.7}$$

Let's try to calculate this in order to get an idea of the size of this error. First, let's introduce the probability distribution of the y_i, denoted with $p(y)$. For it we have

$$\int p(y)\,\mathrm{d}y = 1, \tag{13.8}$$

$$\langle y \rangle = \int y\, p(y)\,\mathrm{d}y, \tag{13.9}$$

$$\left\langle y^2 \right\rangle = \int y^2\, p(y)\,\mathrm{d}y, \tag{13.10}$$

$$\sigma^2 = \left\langle y^2 \right\rangle - \langle y \rangle^2 . \tag{13.11}$$

We can then write

$$P_N(I_N) = \int \delta\left(I_N - \sum_i \frac{y_i}{N}\right) p(y_1)\, p(y_2) \cdots p(y_N)\,\mathrm{d}y_1\,\mathrm{d}y_1 \cdots \mathrm{d}y_N, \tag{13.12}$$

where the Dirac delta-function enforces that the average of the y_i is equal to I_N. We now take the Fourier transform of $p(y)$:

$$\hat{p}(k) = \int p(y)\,\mathrm{e}^{ik(y-\langle y \rangle)}\,\mathrm{d}y \tag{13.13}$$

Similarly, let's consider the Fourier transform of $P_N(I_N)$:

$$\begin{aligned}
\hat{P}_N(k) &= \int P_N(I_N)\,\mathrm{e}^{ik(I_N - \langle I_N \rangle)}\,\mathrm{d}I_N && (13.14)\\
&= \int p(y_1) \cdots p(y_N)\,\mathrm{e}^{i\frac{k}{N}(y_1 - \langle y_1 \rangle + y_2 - \langle y_2 \rangle + \ldots + y_N - \langle y_N \rangle)}\mathrm{d}y_1 \cdots \mathrm{d}y_N && (13.15)\\
&= \left[\hat{p}\left(\frac{k}{N}\right)\right]^N . && (13.16)
\end{aligned}$$

Here we made use of $\langle I_N \rangle = \langle y \rangle$. Now we expand $\hat{p}\left(\frac{k}{N}\right)$ in powers of k/N, in the limit of large N. We get

$$\begin{aligned}
\hat{p}\left(\frac{k}{N}\right) &= \int p(p)\,\mathrm{e}^{i\frac{k}{N}(y - \langle y \rangle)}\,\mathrm{d}y && (13.17)\\
&= \int p(y)\left[1 + \frac{ik}{N}(y - \langle y \rangle) - \frac{k^2}{2N^2}(y - \langle y \rangle)^2 + \ldots\right]\mathrm{d}y && (13.18)\\
&= 1 - \frac{k^2\sigma^2}{2N^2} + \ldots && (13.19)
\end{aligned}$$

Thus we find

$$\hat{P}_N(k) = \left[\hat{p}\left(\frac{k}{N}\right)\right]^N = \left(1 - \frac{k^2\sigma^2}{2N^2}\right)^N \simeq 1 - \frac{k^2\sigma^2}{2N} \simeq \mathrm{e}^{-\frac{k^2\sigma^2}{2N}}, \tag{13.20}$$

because N is very large. With this result in hand, we can now use it to calculate $P_N(I_N)$ through an inverse Fourier transform:

$$\begin{aligned} P_N(I_N) &= \frac{1}{2\pi}\int \mathrm{e}^{-ik(I_N - \langle I_N\rangle)}\,\hat{P}_N(k)\,\mathrm{d}k & (13.21)\\ &= \frac{1}{2\pi}\int \mathrm{e}^{-\frac{k^2\sigma^2}{2N} - ik(I_N - \langle I_N\rangle)}\,\mathrm{d}k & (13.22)\\ &= \frac{1}{2\pi}\mathrm{e}^{-\frac{1}{2}\frac{N}{\sigma^2}(I_N - \langle y\rangle)^2}\int \mathrm{e}^{-\frac{\sigma^2}{2N}\left(k + i(I_N - \langle y\rangle)\frac{N}{\sigma^2}\right)^2}\,\mathrm{d}k & (13.23) \end{aligned}$$

We now can do this integral by shifting k and recalling

$$\int \exp(-\alpha x^2)\,\mathrm{d}x = \sqrt{\frac{\pi}{\alpha}}, \tag{13.24}$$

obtaining

$$P_N(I_N) = \frac{\sqrt{N}}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2}\frac{N}{\sigma^2}(I_N - \langle y\rangle)^2\right). \tag{13.25}$$

This is a Gaussian with dispersion $\sigma_N = \sigma/\sqrt{N}$, *independent* of the shape of $p(y)$. What we just have derived is the *central limit theorem*! Independent of the detailed shape of $p(y)$, if we average enough of these distributions we will get a Gaussian.

Summary:

For standard Monte Carlo integration with N samples, the error is

$$\sigma_N = V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}} \tag{13.26}$$

where $\langle f\rangle$ and $\langle f^2\rangle$ are exact moments of the function we integrate, i.e.

$$\langle f\rangle \equiv \frac{1}{V}\int f(x)\mathrm{d}x = \frac{1}{V}\int y\,p(y)\mathrm{d}y. \tag{13.27}$$

In practice, we can estimate these moments from the Monte-Carlo samples themselves, i.e. we can estimate

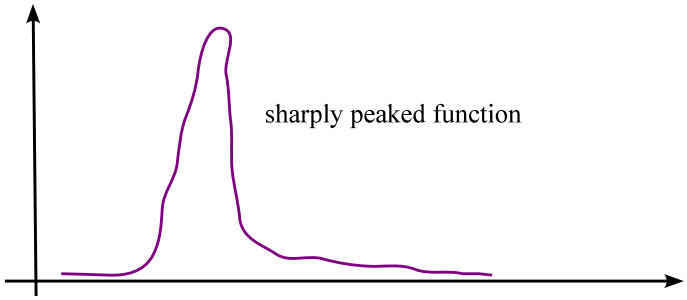
$$\langle f\rangle \simeq \frac{1}{N}\sum_i f(x_i), \tag{13.28}$$

$$\left\langle f^2\right\rangle \simeq \frac{1}{N}\sum_i f^2(x_i), \tag{13.29}$$

and then use these moments to estimate the error.

13.3 Importance Sampling

One common problem in Monte Carlo integration is that often the integrand is very small on a dominant fraction of the integration volume. For example, if the integrand is sharply peaked, only points sampled close to the peak will give a significant contribution.



The idea of **importance sampling** is to choose the random points somehow preferentially around the peak, and putting less points where the integrand is small. This should be more efficient and help to reduce the error for a given number of points.

So let us assume we want to integrate

$$I = \int_V f(x)\,\mathrm{d}x, \tag{13.30}$$

and suppose we choose a distribution $p(x)$ which is close to the function $f(x)$, but which is simple enough so that it is possible to generate x-values from this distribution. We can then write:

$$I = \int_V p(x)\frac{f(x)}{p(x)}\,\mathrm{d}x. \tag{13.31}$$

Thus, if we sample points around a point x with probability $\mathrm{d}p = p(x)\,\mathrm{d}x$ (which is exactly the definition of sampling from the distribution $p(x)$), we simply obtain:

$$I = \lim_{N\to\infty}\frac{1}{N}\sum_i \frac{f(x_i)}{p(x_i)}. \tag{13.32}$$

Because f/p is flatter than f if the shape of p is similar to that of f, the variance of f/p will be smaller than the variance of f, i.e. we obtain a smaller error for given N. The ideal choice is $p(x) \propto f(x)$. This is often not possible in practice, but in fact possible in lattice Monte Carlo simulations, as we will see.

Example for importance sampling

Let's consider a simple 1D integration to demonstrate the concept of importance sampling. The integral we want to compute is:

$$I = \int_0^1 \left(x^{-1/3} + \frac{x}{10}\right)\,\mathrm{d}x. \tag{13.33}$$

This can be solved analytically and has the value $I = 31/20 \simeq 1.55$, so we don't really need Monte Carlo integration here, but for the sake of demonstrating the method we apply it anyway.

Doing this integral with standard Monte Carlo integration gives the error

$$\sigma_N = \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \simeq \frac{0.85}{\sqrt{N}}. \tag{13.34}$$

Let's now try to do it with importance sampling, using the sampling probability

$$p(x) = \frac{2}{3} x^{-1/3} \tag{13.35}$$

over the interval $0 < x \leq 1$, based on the realization that $p(x)$ captures part of the shape of $f(x)$, while being simple enough to allow a creation of properly sampled points by direct inversion (see below). The new function to integrate is then $g = f/p$, and the width of the corresponding Monte Carlo error distribution function becomes

$$\sigma_N = \sqrt{\frac{\langle g^2 \rangle - \langle g \rangle^2}{N}} \simeq \frac{0.045}{\sqrt{N}}. \tag{13.36}$$

This is nearly 20 times better than obtained with plain sampling, hence a substantial gain in efficiency has been reached.

13.4 Random number generation

Good random number obviously play a central role in Monte Carlo techniques.

- Usually they are produced by *deterministic algorithms* leading to *pseudorandom numbers*.
- Such pseudorandom numbers need to be distinguished from "truly random" numbers generated by some physical process (like rolling the dice, radioactive decay, quantum transitions, etc.). Some modern CPUs include a hardware random number generator, based for example on coupled non-linear oscillators and additional sources of entropy. However, these generators are normally not used for Monte Carlo techniques, because
  - the sequence is not repeatable, making debugging difficult and preventing exact reproducibility
  - the generators are often slow
  - the quality of the distribution may not be perfect
  - the quality of the distribution may degrade with time, or correlate in subtle ways with environmental factors such as operating temperature, etc.

- There is value in having good random numbers. In 1950, the RAND corporation published a book entitled "1 million random digits", whose primary virtue is to contain no discernable information at all. This classic is available online (`http://www.rand.org/publications/classics/randomdigits`).

13.4.1 Pseudo-random numbers

Here we usually create an integer sequence that is then converted to a floating point number in the interval $[0, 1[$. The essential desirable properties of a good random number generator are:

- Repeatability: For the same seed, we want to obtain the same sequence of random numbers.
- Randomness: Good random numbers should
  - be uniformly and homogeneously distributed in the interval $[0, 1[$.
  - be independent of each other, i.e. show no correlations whatsoever (this is difficult and not true exactly for pseudo-random number generators)
- Speed: In modern applications, we may need billions of random numbers.
- Portability: We want the same results on different computer architectures.
- Long period: After a finite number of pseudo-random numbers, the sequence repeats. This period should be as large as possible.
- Insensitivity to seed: Neither the period nor the quality of the randomness should depend on the value of the seed, i.e. on where the sequence is started.

Linear congruential generators

The simplest pseudo-random number generators work with an integer mapping of the form

$$X_{\mathrm{i}+1} = (aX_i + b) \mod m, \tag{13.37}$$

where a, b, and m are integers. The numbers X_i lie then in the range $[0, m-1]$ and can be mapped to floating point numbers in the unit interval by dividing with m. It is clear that such a generator can have a period of at most m. Examples for such *linear congruential generators* include:

- **ANSI-C:**

$$a = 1103515245 \quad b = 12345 \quad m = 2^{31} \tag{13.38}$$

  The period here is quite small, just $m = 2^{31} \sim 2 \times 10^9$, which is quickly reached in modern computers. This is not good enough for serious Monte Carlo applications.

- **RAND generator in Matlab:**

$$a = 16807 \quad b = 0 \quad m = 2^{31} - 1 \tag{13.39}$$

  Again, this has an uncomfortably short period.

- **UNIX drand48():**

$$a = 25214903917 \quad b = 11 \quad m = 2^{48} \tag{13.40}$$

  This is starting to be somewhat usable, with a period of $2^{48} \simeq 2.8 \times 10^{14}$. Note however that the low order bits have shorter cycling times and show less randomness than they should, which is a common problem for all simple linear congruential random number generators.

- **NAG-generator:**

$$a = 13^{13} \quad b = 0 \quad m = 2^{59} \tag{13.41}$$

  This has a very long period, but the low order bits are still not very good.

To get better random numbers, one needs to go to more complicated schemes than a simple integer mapping. One approach is to combine two or several linear congruential mappings. This is done for example in **ran2** of Numerical Recipes. This uses

$$X_{i+1} = (40014 X_i) \mod 2147483563 \tag{13.42}$$
$$Y_{i+1} = (40692 Y_i) \mod 2147483399 \tag{13.43}$$
$$Z_{i+1} = (X_i + Y_i) \mod 2147483563 \tag{13.44}$$

The Z_i are then mapped to floating point numbers. Here the period is extended to $\sim 10^{18}$.

Lagged Fibonacci generators

A refinement of this approach consists of using several integers to define the internal state of the generator. One then uses a prescription of the form

$$X_i = (X_{i-p} \odot X_{i-q}) \mod m \tag{13.45}$$

to create new integers, where p and q are the 'lags' (offsets to other past numbers), and $\odot$ is some arithmetic operation, for example addition, multiplication, etc., or also bitwise logical operations such as XOR. For large lags, the quality of these generators becomes very good.

For example, **RANLUX** is of this type, reaching a period 10^{171}. Another modern generator using this principle is the **Mersenne Twister**, also known as MT19937. This uses 624 internal 32-bit integers to describe its state, and abundantly employs XOR as well as other bit-shuffling and swapping operations. It's period is huge, a staggering $2^{19937} - 1$. This should be a pretty good choice for Monte Carlo! The Mersenne Twister is for example available in the GSL-library.[1]

[1] `http://www.gnu.org/software/gsl`

13.5 Using random numbers

Random numbers are usually generated in the standard interval $[0, 1[$ with a uniform distribution. If that's what you need – fine. But often we need random numbers drawn from some other probability distribution function (e.g. a Gaussian). How is this done?

13.5.1 Exact inversion

Recall, the PDF satisfies $\int p(x)\,\mathrm{d}x = 1$ and $p(x) \geq 0$ for all x. Such probability distributions can be transformed to other distributions by observing conservation of probability:

$$p_1(x)\,\mathrm{d}x = p_2(y)\,\mathrm{d}y \tag{13.46}$$

where $y = y(x)$. This leads to the transformation rule

$$p_2(y) = p_1(x)\left|\frac{\mathrm{d}y}{\mathrm{d}x}\right|, \tag{13.47}$$

where here a modulus has been added to neutralize a possible sign change due to the mapping. This can now be used as follows: Suppose we know $p_1(x)$ (usually the distribution returned by our random number generator, in which case $p_1(x) = 1$) and we have a desired distribution $p_2(y)$, then we need to find the mapping $y = y(x)$ that transforms one into the other. We can obtain this by integrating the differential equation

$$\int_{-\infty}^{x} p_1(x')\,\mathrm{d}x' = \int_{-\infty}^{y} p_2(y')\,\mathrm{d}y'. \tag{13.48}$$

This is simply saying that $P_1(x) = P_2(y)$, where $P_1(x)$ and $P_2(y)$ are the cumulative probability distribution functions of p_1 and p_2, respectively. Hence, we need to calculate

$$y = P_2^{-1}\left[P_1(x)\right]. \tag{13.49}$$

In case $p_1(x)$ is an ordinary random number generator, this can also be written as

$$x = \int_{-\infty}^{y} p_2(y')\,\mathrm{d}y'. \tag{13.50}$$

Unfortunately, the inversion cannot always be carried out algebraically, but if this is possible, this is the method of choice.

Example

Suppose you want to have random numbers from the distribution

$$p(y) = \frac{1}{4}y^3 \quad \text{for } y \in [0, 2]. \tag{13.51}$$

The cumulative distribution is here

$$P(y) = \int_0^y \frac{y'^3}{4}\,\mathrm{d}y' = \frac{y^4}{16}. \tag{13.52}$$

Hence, we can draw random numbers uniformly from $x \in [0, 1[$ and convert them to

$$y = (16x)^{1/4}, \tag{13.53}$$

which then sample our desired distribution function.

Important special cases

The Gaussian distribution

$$p(y) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) \tag{13.54}$$

is often needed. The cumulative distribution is the error function, which can not be easily inverted without resorting to iterative (and hence comparatively expensive) methods.

There is however a simple trick, known as the Box-Muller method, that can circumvent this issue. Suppose we consider generating a 2D Gaussian distribution

$$p(x, y) = \frac{1}{2\pi} \exp\left(-\frac{x^2 + y^2}{2}\right), \tag{13.55}$$

which is simply the product of two 1D-distributions. We can transform this to polar coordinates in the (x, y)-plane:

$$p(x, y)\,\mathrm{d}x\,\mathrm{d}y = \frac{1}{2\pi} \exp\left(-\frac{r^2}{2}\right) r\,\mathrm{d}r\,\mathrm{d}\phi. \tag{13.56}$$

Hence ϕ is uniformly distributed in $[0, 2\pi[$, i.e.

$$\phi = 2\pi \cdot X_1 \tag{13.57}$$

for some standard random number X_1 from the unit interval. For the radial coordinate we have on the other hand:

$$X_2 = \int_0^r r' \exp\left(-\frac{r'^2}{2}\right) \mathrm{d}r' \tag{13.58}$$

This can be integrated and inverted to yield

$$r = \sqrt{-2 \ln X_2}, \tag{13.59}$$

where X_2 is again a random number independently drawn from $[0, 1[$. Finally, we can calculate

$$x = r \cos\phi, \tag{13.60}$$

$$y = r \sin\phi, \tag{13.61}$$

which now yields two perfectly fine Gaussian distributed numbers x and y, which may both be used. This procedure hence always converts two random numbers from $[0, 1[$ to two independent Gaussian distributed numbers.

13.5.2 Rejection method

Assume that $p(x)$ is the desired random number distribution, and $f(x)$ is the distribution that we can create. If we have

$$p(x) \leq C \cdot f(x) \tag{13.62}$$

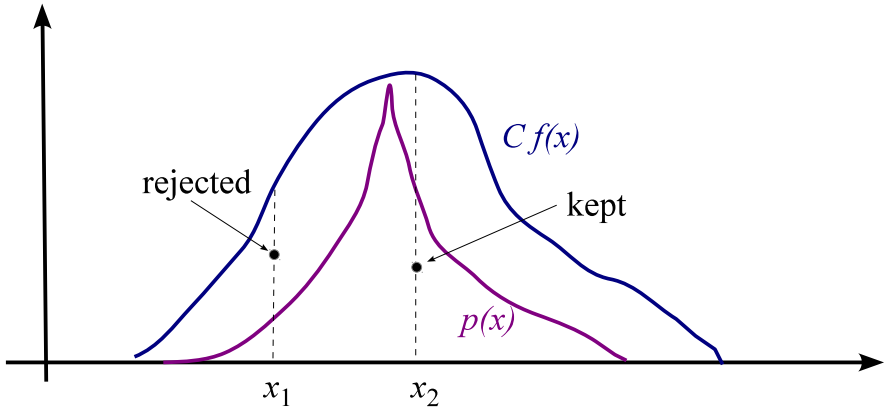
with some known constant C, then we can generate random numbers that sample $p(x)$ with the rejection method. This method works as follows:

1. Generate an x from $f(x)$.
2. Generate a y from a uniform distribution with the bounds $0 \leq y < C \cdot f(x)$.
3. If $y \leq p(x)$ return x as a sample value.
4. Otherwise, i.e. for $y > p(x)$ reject the trial value for x and repeat at step 1.

The probability $\mathrm{d}q$ to get a certain x within $\mathrm{d}x$ with this procedure is:

$$\mathrm{d}q = f(x)\mathrm{d}x \cdot \frac{p(x)}{Cf(x)} = \frac{1}{C}\, p(x)\, \mathrm{d}x \propto p(x)\, \mathrm{d}x. \tag{13.63}$$

Hence this will reproduce the desired probability distribution.



There are a number of advantaged of this approach, in particular, it works in any dimension, and $p(x)$ does not necessarily have to be normalized. The main disadvantage can lie in a low efficiency if the rejection rate is high. The latter is given by the complement to the acceptance rate, which is given by the area under $p(x)$ relative to the area under $Cf(x)$.

Example

Let's assume we want to distribute points uniformly on a sphere. The standard way is to use direct inversion. In 3D, this is still readily possible by the use of spherical polar coordinates. We have for the surface element

$$\sin\theta \,\mathrm{d}\theta \,\mathrm{d}\phi = \mathrm{d}\cos\theta \;\mathrm{d}\phi, \tag{13.64}$$

hence the distributions of $\mathrm{d}\cos\theta$ and $\mathrm{d}\phi$ are uniform over their range. Hence we can set

$$\cos\theta = 2\,u_1 - 1, \tag{13.65}$$

$$\phi = 2\pi u_2, \tag{13.66}$$

where u_1 and u_2 are standard uniform numbers. We can then calculate the coordinates as

$$x = \sin\theta\,\cos\phi, \tag{13.67}$$

$$x = \sin\theta\,\sin\phi, \tag{13.68}$$

$$z = \cos\theta. \tag{13.69}$$

This is fine, but cumbersome to generalize to hyperspheres in higher dimensions. A much simpler approach is to use rejection sampling. Suppose we draw three random numbers u_1, u_2, and u_3, which we then map to $[-1, 1]$ through $\tilde{u}_i = 2u_i - 1$. Now we calculate $r^2 = \tilde{u}_1^2 + \tilde{u}_2^2 + \tilde{u}_3^2$, and use the rejection method: We only keep the point if $r^2 \leq 1$, which effectively uniformly samples the inside of a sphere. If we then stretch the kept points as

$$x = \frac{\tilde{u}_1}{r}, \quad x = \frac{\tilde{u}_2}{r}, \quad z = \frac{\tilde{u}_3}{r}, \tag{13.70}$$

they are uniformly distributed on the unit sphere. This method works for any number of dimensions.

13.5.3 Sampling with a stochastic process

There are situations when neither direct inversion nor the rejection method can be readily used to sample from a given distribution function $p(x)$. In this case we can construct a sample of $p(x)$ through a stochastic process that has $p(x)$ as its equilibrium distribution.

We will accomplish this with a so-called *Markov process*, which generates a Markov chain. A Markov chain is a discrete sequence of states,

$$x_1 \xrightarrow{f} x_2 \xrightarrow{f} x_3 \xrightarrow{f} \ldots \xrightarrow{f} x_n, \tag{13.71}$$

where f is a Monte Carlo update operator. The characterizing property of a Markov process is that the transition probability from one state to the next state in the chain,

$$W_f(x \to x') = W_f(x'|x), \tag{13.72}$$

depends *only* on the current state, i.e. information about the history is not used at all. Note that f can here mediate a small update or an arbitrarily large one.

The transition probability has the natural properties

$$\int W_f(x \to x')\,\mathrm{d}x' = 1, \tag{13.73}$$

and $W_f(x \to x') \geq 0$.

We can also apply the transition probability to whole probability distributions, getting the new probability distribution after one transition:

$$p(x) \xrightarrow{f} p'(x') = \int p(x) W_f(x \to x')\,\mathrm{d}x. \tag{13.74}$$

We will now demand two properties of the update step that will turn the Markov process into a very powerful tool:

1. f must preserve $p_{\mathrm{eq}}(x)$ as an equilibrium distribution of the stochastic process, or in other words $p_{\mathrm{eq}}(x)$ must be a fix point of f. This requires

$$p_{\mathrm{eq}}(x') = \int p_{\mathrm{eq}}(x) W_f(x \to x')\,\mathrm{d}x. \tag{13.75}$$

2. Starting from any state x, repeated applications of f must be able to get arbitrarily close to any other state x'. This is called the ergodic property.

Two important results follow from these properties:

- Any ensemble of states approaches the equilibrium distribution if f is applied sufficiently often.
- The collection of states in a single Markov chain under the action of f approaches $p(x)$ as the number of steps goes to infinity.

Let us proof the first of these results. To this end, let $p(x)$ be the PDF of the initial ensemble, and $p_{\mathrm{eq}}(x)$ the equilibrium distribution. After applying f once, we obtain $p'(x')$. We now want to show that p' is closer to p_{eq} than p. To this end, we consider the norm

$$\begin{aligned}
||p' - p_{\mathrm{eq}}|| &\equiv \int |p'(x') - p_{\mathrm{eq}}(x')|\,\mathrm{d}x' && (13.76)\\
&= \int \mathrm{d}x' \left| \int \mathrm{d}x\, W_f(x \to x')(p(x) - p_{\mathrm{eq}}(x)) \right| && (13.77)\\
&\leq \int \mathrm{d}x' \int \mathrm{d}x\, W_f(x \to x')\, |p(x) - p_{\mathrm{eq}}(x)| && (13.78)\\
&= \int \mathrm{d}x\, |p(x) - p_{\mathrm{eq}}(x)| && (13.79)\\
&= ||p - p_{\mathrm{eq}}||. && (13.80)
\end{aligned}$$

For the third line, we have basically used the triangle inequality, $|a+b| \leq |a| + |b|$. Thus, the difference between p and p_{eq} shrinks if f is applied. But, perhaps $||p - p_{\text{eq}}||$ gets stuck at some finite value and doesn't really go to zero. This would mean that there must be another fix-point $\tilde{p}_{\text{eq}}$ with $||\tilde{p}_{\text{eq}} - p_{\text{eq}}|| > 0$, and $||\tilde{p}'_{\text{eq}} - p'_{\text{eq}}|| = ||\tilde{p}_{\text{eq}} - p_{\text{eq}}||$.

However, the ergodicity property of the mapping f implies that there are some x' for which

$$\left| \int \mathrm{d}x \, W_f(x \to x')(\tilde{p}_{\text{eq}}(x) - p_{\text{eq}}(x)) \right| < \int \mathrm{d}x \, W_f(x \to x') \, |\tilde{p}_{\text{eq}}(x) - p_{\text{eq}}(x)| \,. \quad (13.81)$$

This is because if A is the set for which $\tilde{p}_{\text{eq}}(x) - p_{\text{eq}}(x) \leq 0$, and B the set for which $\tilde{p}_{\text{eq}}(x) - p_{\text{eq}}(x) > 0$, then there must be some x' from B and some x from A for which we have a non-zero $W_f(x \to x') > 0$, otherwise the two sets would be isolated from each other, violating the ergodic assumption. On the other hand, for the norm of $||\tilde{p}'_{\text{eq}} - p'_{\text{eq}}||$ we get

$$\begin{aligned} ||\tilde{p}'_{\text{eq}} - p'_{\text{eq}}|| &= \int \mathrm{d}x' \, |\tilde{p}'(x')_{\text{eq}} - p'_{\text{eq}}(x')| = \int \mathrm{d}x' \left| \int \mathrm{d}x \, W_f(x \to x')(\tilde{p}(x)_{\text{eq}} - p_{\text{eq}}(x)) \right| \\ &< \int \mathrm{d}x' \int \mathrm{d}x \, W_f(x \to x') \, |\tilde{p}(x)_{\text{eq}} - p_{\text{eq}}(x)| \quad (13.82) \\ &= \int \mathrm{d}x \, |\tilde{p}(x)_{\text{eq}} - p_{\text{eq}}(x)| = ||\tilde{p}_{\text{eq}} - p_{\text{eq}}||, \quad (13.83) \end{aligned}$$

where for establishing the $<$-sign we used equation (13.81). The conclusion reached here, $||\tilde{p}'_{\text{eq}} - p'_{\text{eq}}|| < ||\tilde{p}_{\text{eq}} - p_{\text{eq}}||$, contradicts the existence of two equilibrium distributions.

Detailed balance

Almost all of the commonly used update steps follow the **detailed balance condition**, i.e.:

$$p_{\text{eq}}(x) \cdot W_f(x \to x') = p_{\text{eq}}(x') \cdot W_f(x' \to x). \quad (13.84)$$

Here it is obvious and easy to show that $p_{\text{eq}}(x)$ is a fix point under f, while for other choices of f this may still be the case but could be difficult to prove.

So detailed balance and ergodicity are already sufficient conditions to obtain a Markov chain that samples $p_{\text{eq}}(x)$. But we still need to find a concrete realization of W_f.

13.5.4 The Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm provides a simple and generic way for constructing a suitable transition operation. It works as follows:

1. When the currents state is x, propose a new state x' with a proposal probability $q(x \to x')$.

2. Calculate the Hasting's ratio

$$r = \min\left(1, \frac{p(x')\,q(x' \to x)}{p(x)\,q(x \to x')}\right), \tag{13.85}$$

where the min-operation is used to restrict the value of r to the range $[0, 1]$.

3. Accept the proposed move with probability r (i.e. draw a random number $u \in [0, 1[$, and if its smaller than r, accept), in which case x' is made the next element in the Markov chain. Otherwise, the proposed state is *rejected*, and the old state is added (again) as an element in the Markov chain. This is sometimes called the Metropolis rejection step.

Does the Metropolis algorithm fulfill detailed balance? We can check this by working out the transition probability $W(x \to x')$, which is the product of the proposal probability of the new state and its acceptance probability:

$$W(x \to x') = q(x \to x') \cdot \frac{p(x')\,q(x' \to x)}{p(x)\,q(x \to x')} = \frac{p(x')}{p(x)}\,q(x' \to x). \tag{13.86}$$

Here we have assumed without loss of generality that the Hastings ratio is less than 1. In this case, the inverse transition is then simply given as

$$W(x' \to x) = q(x' \to x) \cdot 1 \tag{13.87}$$

Combining equations (13.86) and (13.87) we then verify the condition of detailed balance.

The proposal probability $q(x \to x')$ is fairly arbitrary – it only must be ergodic, i.e. all states must be reachable through successive applications of q, then the Monte Carlo Markov Chain (MCMC) created by the algorithm will eventually produce a fair sample of the target distribution function $p(x)$. This is a quite remarkable property.

Metropolis update

This is the special case in which the stochastic proposal operator is symmetric, i.e.

$$q(x \to x') = q(x' \to x). \tag{13.88}$$

Then the acceptance probability simply becomes

$$r = \min\left(1, \frac{p(x')}{p(x)}\right). \tag{13.89}$$

A proposed move to a state of higher probability is hence always accepted. (But one sometimes also moves to a proposed state of lower probability.)

The simplest form of such a symmetric update would be something like

$$q(x \to x') : \quad x' = x + e, \tag{13.90}$$

where e is distributed symmetrically around zero and is independent of x. For example, e could be drawn from a normal or uniform distribution of some prescribed width.

Example

Let's come back to our starting point, the generation of a sample from a distribution $p(x)$ with a stochastic process. For simplicity, we want to try this out on a simple Gaussian distribution, $p(x) \propto \exp(-x^2/2)$, using the Metropolis algorithm. As a proposal function, we could for example choose $q(x'|x): \quad x' = x + (2u - 1)/10$, where u is drawn uniformly from $[0, 1[$. Then we could proceed like this:

1. Start with some x with $p(x) > 0$.
2. Draw u and calculate the proposal x'.
3. Now compute $r = \min[1, \exp(-x'^2/2)/\exp(-x^2/2)]$.
4. Draw another random number u' from $[0, 1[$ and take x' as new point in the chain if $u' \leq r$, otherwise take x as new point.
5. Repeat at step 2 until you believe you have enough points in the chain.

Note that there can be lots of repeated entries in the chain if the rejection rate is high. Also, you may not use the chain as a randomly sampled sequence from the underlying distribution. Subsequent entries in the change will be highly correlated with each other! Nevertheless, the collection of all the points in a single chain represent a proper sample from the distribution in the limit of an infinitely long chain, thanks to the ergodic property.

13.6 Monte Carlo simulations of lattice models

An important application of MCMC techniques lies in the thermodynamics of physical systems, for example solids described on a lattice. In this case, we may have a field $\phi_{\mathbf{x}}$ at each lattice site $\mathbf{x}$, whose dynamics is described by some Hamiltonian $H(\phi)$.

Assuming the canonical ensemble, the partition function is then given by

$$Z = \int \exp\left[-\frac{H(\phi)}{kT}\right] [\mathrm{d}\phi], \tag{13.91}$$

where $[\mathrm{d}\phi] = \mathrm{d}\phi_1\, \mathrm{d}\phi_2\, \mathrm{d}\phi_3 \cdots \mathrm{d}\phi_N$ is a short-hand notation for the differential volume element in the extremely high-dimensional space of all possible field configurations.

In practice, very often the task to compute the thermal average of some quantity A arises. This is given by

$$\langle A \rangle = \frac{1}{Z} \int A(\phi) \exp\left[-\frac{H(\phi)}{kT}\right] [\mathrm{d}\phi]. \tag{13.92}$$

Unfortunately, because of the high dimensionality, these integrals cannot be carried out with standard techniques. We hence would like to employ Monte Carlo integration combined with importance sampling in which the phase-space points of the

system are chosen according to $p(\phi) \propto \exp\left[-\frac{H(\phi)}{kT}\right]$. For obtaining such a sample, we need a stochastic process, because neither direct inversion nor the rejection method are feasible.

For producing the required Markov chain, one can for example use the Metropolis-Hastings algorithm. If we symmetrically propose new states, then the acceptance probability will become

$$r = \min\left[1, \exp\left(-\frac{H(\phi') - H(\phi)}{kT}\right)\right], \tag{13.93}$$

still leaving many ways for how the proposals are generated.

Another possibility is to employ the so-called *heat bath*, or Gibbs sampler. This directly sets

$$W_f(\phi \to \phi') = C \exp\left(-\frac{H(\phi')}{kT}\right), \tag{13.94}$$

which even doesn't depend on the state ϕ at all. In practice, it is however not always trivial to invert this and actually use it.

In either case, once a sufficiently long Markov chain with sampled states has been constructed, we can use it to straightforwardly calculate alls sorts of thermal averages by replacing the integrals with averages of $A(\phi)$ at the sampled points.

Example: Ising Model

The Ising model is the simplest discrete spin model in which the field variable is $s_{\mathbf{x}} = \pm 1$, i.e. at each lattice site the spin either points up or down. The partition function of the system is

$$Z = \sum_{\{s_{\mathbf{x}}\}} \exp\left[-\beta\left(\frac{1}{2}\sum_{<\mathbf{x},\mathbf{y}>}(1 - s_{\mathbf{x}}s_{\mathbf{y}}) + B\sum_{\mathbf{x}} s_{\mathbf{x}}\right)\right]. \tag{13.95}$$

Here the first sum is over all possible spin configurations. The sum $< \mathbf{x}, \mathbf{y} >$ is only over pairs of nearest lattice sites; only their spin interaction is counted. Finally, B describes an external magnetic field, which one may also put to zero. $\beta = 1/T$ measures the temperature (we use a natural system of units here).

For $B = 0$, the Ising model shows 2nd-order phase transitions if the number of dimensions is larger than one. Then, below a certain critical temperature, spontaneous magnetization of the medium occurs. For $d = 2$, the transition temperature has been calculated analytically by Onsager, $\beta_c = \ln(1 + \sqrt{2})$, but for higher dimensions, analytic solutions are not known. Here one therefore needs to turn to Monte Carlo simulations.

The simplest approach is to use the Metropolis algorithm in which one selects a single spin $s_{\mathbf{x}}$ at lattice site $\mathbf{x}$ and proposes the opposite spin direction $s'_{\mathbf{x}}$. The selection of the lattice site can be done randomly, or in red-black ordering (or even type-writer ordering, but this is less efficient). One then computes the local energy

functional $E_{\mathbf{x}}$ which involves all the terms in the interaction energy that involve the chosen spin. This gives rise to a change $\delta E_{\mathbf{x}} = E_{\mathbf{x}}(s'_{\mathbf{x}}) - E_{\mathbf{x}}(s_{\mathbf{x}})$ due to the spin flip. The acceptance probability is then given as

$$r = \min\left(1, \exp[-\beta\, \delta E_{\mathbf{x}}]\right), \tag{13.96}$$

and with this one can generate a long MC chain with different states of the system, which will eventually represent thermal equilibrium at the prescribed temperature.

For this simple spin system, one may also use the heat bath update, and choose the new spin direction of the selected site according to

$$p(s_{\mathbf{x}}) = \frac{\mathrm{e}^{-\beta E_{\mathbf{x}}(s_{\mathbf{x}}=+1)}}{\mathrm{e}^{-\beta E_{\mathbf{x}}(s_{\mathbf{x}}=+1)} + \mathrm{e}^{-\beta E_{\mathbf{x}}(s_{\mathbf{x}}=-1)}}. \tag{13.97}$$

Here the inversion is trivially possible; one simply draws a random number u from $[0, 1]$ and picks the $+1$ direction when $u < p(s_{\mathbf{x}})$ and spin equal to -1 otherwise.

Once a very long MCMC chain of the system in thermal equilibrium has been calculated, one can simply compute various thermodynamic quantities of interest by straightforward averaging (which really is Monte Carlo integration with importance sampling). For example:

- Mean magnetization: $M = \frac{1}{V}\sum_i s_i$
- Specific heat: $C_V = \frac{1}{V}\frac{\partial E}{\partial T} = \langle E^2\rangle - \langle E\rangle^2$
- Magnetic susceptibility: $\chi_M = \frac{1}{V}\frac{\partial M}{\partial T} = \langle M^2\rangle - \langle M\rangle^2$

13.7 Monte Carlo Markov Chains in parameter estimation

Another important application of MCMC techniques lies in parameter estimation, in particular in the context of *Baysian data analysis*. Often in physics, we want to use some experimental data

$$\mathbf{z} = (z_1, z_2, \ldots, z_n) \tag{13.98}$$

to infer a number of parameters

$$\theta = (\theta_1, \theta_2, \ldots, \theta_m) \tag{13.99}$$

which describe a physical model/theory. For example, observations of the microwave background radiation (where $\mathbf{z}$ might refer to the pixels with measured temperature fluctuations) are used to estimate parameters such as the mean density and expansion rate of the Universe.

One way of constraining the parameters is to consider the likelihood function

$$p(\mathbf{z}|\theta). \tag{13.100}$$

Here one considers the probability for observing the data given the parameters have certain values. If we have $p(\mathbf{z}|\theta') > p(\mathbf{z}|\theta)$ for two parameter sets θ' and θ, then it is natural to assume that θ' is somehow more plausible. This leads to the idea of determining the best guess for the parameters by determining the point with the maximum likelihood.

However, thinking about this for a bit, one realizes that we are actually not interested in the probability of observing the data given some parameters, instead we want the reverse: Because we have the data and the parameters are unknown, we should ask, what statement can we make about the probability distribution function of the (unknown) parameter values given the data? This means that we really would like to have

$$p(\theta|\mathbf{z}). \tag{13.101}$$

To obtain access to this quantity, we can invoke **Bayes theorem**, which simply states

$$p(\theta|\mathbf{z}) = \frac{p(\theta)\, p(\mathbf{z}|\theta)}{p(\mathbf{z})}. \tag{13.102}$$

In the context of Bayesian inferences, the terms in this expression carry names that elucidate their meaning.

- $p(\theta)$ is the so-called *prior*. It encodes the knowledge we already have about θ even before the data is taken. This can be in the form of trivial constraints, such as knowing that a parameter cannot take on negative values, or it can come from previous experiments. If we do know nothing about the values of the parameters, then the prior would be a flat distribution.

- $p(\mathbf{z}|\theta)$ is the likelihood of certain data given the parameters in our theoretical model. This we can normally calculate if we know the experimental errors.

- $p(\theta|\mathbf{z})$ is the *posterior* probability distribution function. It encodes the information we have gained about the parameters given the data $\mathbf{z}$ has been measured/observed. Our goal with the experiment is to map out $p(\theta|\mathbf{z})$. This allows us to give confidence intervals for any of the parameters, and also determines all of their correlations.

- Finally, $p(\mathbf{z})$ is called the *model evidence* and gives the probability distribution function for the data. This is normally not easily accessible. Fortunately, it turns out that this is not problematic as the evidence drops out when the Monte Carlo chain is constructed.

A useful way to think about Bayes theorem as applied here is in terms of information theoretical concepts: It describes how our prior knowledge is updated by new data. This is encoded in the statement:

$$\text{posterior} \propto \text{prior} \times \text{likelihood}. \tag{13.103}$$

Now back to the problem of how we can determine $p(\theta|\mathbf{z})$. We achieve a sampling of this distribution by applying a Monte Carlo Markov Chain that samples the posterior as an equilibrium distribution. To this end, we apply the Metropolis-Hastings algorithm:

1. We adopt a proposal function $q(\theta \to \theta') = q(\theta'|\theta)$ which may only depend on the current point in the chain.

2. We compute the acceptance probability

$$r = \min\left(1, \frac{p(\theta')\, p(\mathbf{z}|\theta')\, q(\theta|\theta')}{p(\theta)\, p(\mathbf{z}|\theta)\, q(\theta'|\theta)}\right) \tag{13.104}$$

   and keep the proposal with this probability, otherwise we use the old point as new point in the chain. We see here that the evidence $p(\mathbf{z})$ drops out in the Hastings ratio, simply because the data does not change. Also, we actually do not need to know the normalizations of prior and likelihood – they also drop out.

Again, once we have produced a sufficiently long chain, it becomes easily possible to calculate integrals involving the posterior as this can be simply be done as a Monte Carlo integral with importance sampling. This allows for example marginalizations over certain parameters in an easy way.

Finally, we should not forget to mention one important caveat of these MCMC techniques. It is usually quite difficult to decide when a chain has reached a sufficient length to safely trust all the obtained results. This is one of the main disadvantages of the MCMC method. While there are some heuristics to tell when 'convergence' has been reached, doing this rigorously is a difficult problem without simple practical solution.

14 Parallelization techniques

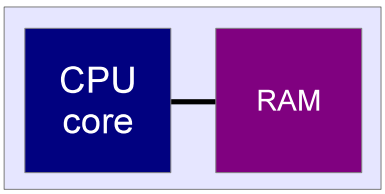
Modern computer architectures offer ever more computational power that we ideally would like to use to their full extent for scientific purposes, in particular for simulations in physics. However, unlike in the past, the speed of individual compute cores, which we may view as serial computers, has recently hardly grown any more (in stark contrast to the evolution a few years back). Instead, the number of cores on large supercomputers has started to increase exponentially. Even on laptops and cell-phones, multi-core computers have become the norm rather than the exception.

However, most algorithms and computer languages are constructed around the concepts of a serial computer, in which a stream of operations is executed sequentially. This is also how we typically think when we write computer code. In order to exploit the power available in modern computers, one needs to change this approach and adopt parallel computing techniques. Due to the large variety of computer hardware, and the many different technical concepts for devising parallel programs, we can only scratch the surface here and make a few basic remarks about parallelization, and some basic techniques that are currently in wide use for it. The interested student is encouraged to read more about this in books and/or in online resources.

14.1 Hardware overview

Let's start first with a schematic overview over some of the main characteristics of current computer architectures.

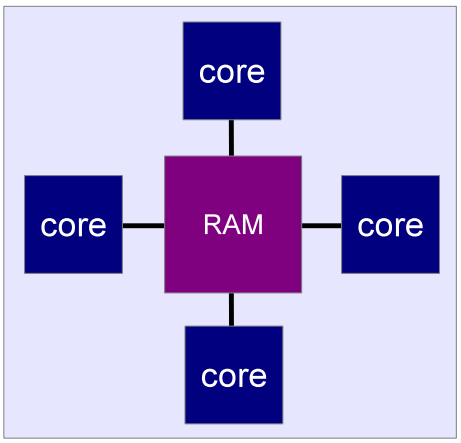
14.1.1 Serial computer



The traditional model of a computer consists of a central processing unit (CPU) capable of executing a sequential stream of load, store, and compute operations, with the data stored in a random access memory (RAM). Branches and jumps in this stream are possible too, but at any given time, only one operation is carried out. The operating system may still provide the illusion that several programs are executed concurrently, but in this case this is reached by time slicing the single compute resource.

Most computer languages are built around this model; they can be viewed as a means to create the stream of serial operations in a convenient way. One can in principle also by-pass the computer language and write down the machine instructions directly (assembler programming), but fortunately, modern compilers make this unnecessary in scientific applications (except perhaps in very special circumstances where extreme performance tuning is desired).

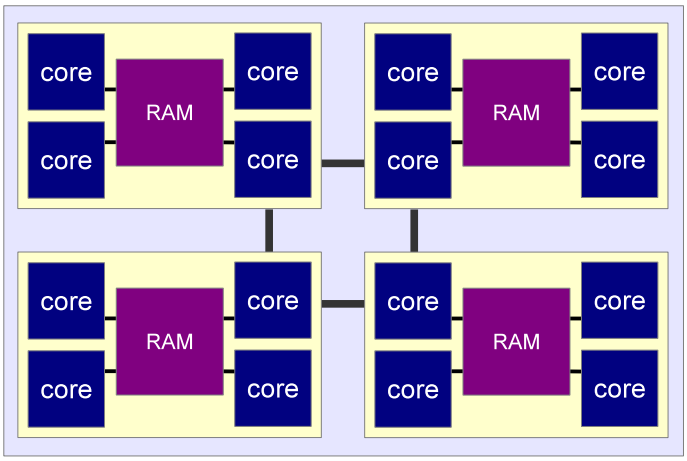
14.1.2 Multi-core nodes



It is possible to attach multiple CPUs to the same RAM, and, especially in recent times, computer vendors have started to add multiple cores to individual CPUs. On each CPU and each core of a CPU, different programs can be executed concurrently, allowing real parallel computations. In machines with uniform memory access, the individual cores can access the memory with the same speed, at least in principle. In this case the distinction between a CPU and a core can become confusing (and is in fact superfluos at some level), because it is ambiguous whether "CPU" refers to a single core, or all the cores on the same die of silicon (recommendation: it's usually best to simple speak about cores to avoid any confusion).

14.1.3 Multi-socket compute nodes

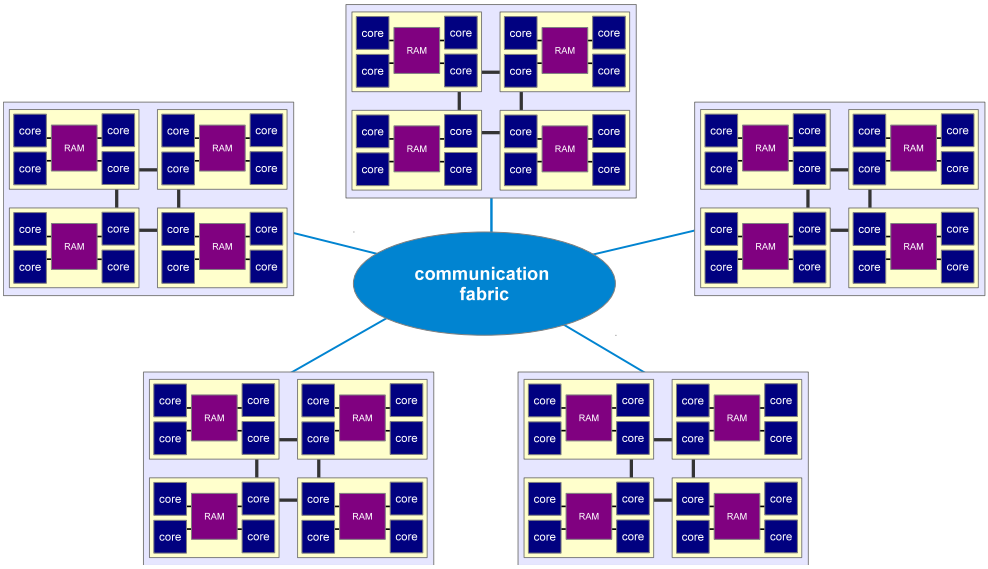
Most powerful compute servers feature a so-called NUMA (non-uniform memory access) architecture these days. Here the full main memory is accessible by all cores, but not all parts of it with the same speed. The compute nodes usually feature individual multi-core CPUs, each with a dedicated memory bank attached. Read and write operations to this part of physical memory are fastest, while accessing the other memory banks is typically noticeably slower and often involves going through special, high-bandwidth interprocessor bus systems.



In such machines, maximum compute performance is only reached when the data that is worked on by a core resides on the "right" memory bank. Fortunately, the operating system will normally try to help with this by satisfying memory requests out of the closest part of physical memory, if possible.

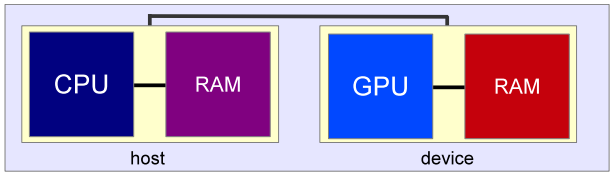
14.1.4 Compute clusters

Very powerful supercomputers used in the field of high-performance computing (HPC) can be formed by connecting many compute nodes through a fast communication network. This can be standard gigabit ethernet in some cases, but usually much faster (and more expensive) communication networks such as infiniband are employed. The leading supercomputers in the world are of this type, and currently reach several 10^5 cores in total.



14.1.5 Device computing

A comparatively new trend is to augment classical compute nodes with special accelerator cards that are particularly tuned to floating point calculations. These cards have much simpler, less flexible compute cores, but the transistors saved on implementing chip complexity can be spent on building more powerful compute engines that can execute many floating point operations in parallel. Graphics processing units (GPUs) have been originally developed with such a design just for the vector operations necessary to render graphics, but now their streaming processors can also be used for general purpose calculations. For certain applications, GPUs can be much faster than ordinary CPUs, but programming them is harder.



In so-called hybrid compute nodes, one has one or several ordinary CPUs coupled to one or several GPUs, or other accelerator cards such as the new Intel Phi. Of course, these hybrid nodes can be clustered again with a fabric to form powerful supercomputers. In fact, the fastest machines in the world are presently of this type.

14.1.6 Vector cores

Another hardware aspect that should not be overlooked is that single compute cores are actually increasingly capable to carry out so-called vector instructions. Here a single instruction (such as addition, multiplication, etc.) is applied to multiple data elements (a vector). This is also a form of parallelization, allowing the calculation throughput to be raised significantly. Below is an example that calculates $x = a \cdot b$ element by element for 4-vectors a and b. This can be programmed explicitly through *intrinsics* in C, which are basically individual machine instructions hidden as macros or function calls within C. (Usually, one does not do this manually though, but rather hopes the compiler emits such instructions somehow automatically.)

```c
#include <xmmintrin.h>

void do_stuff(void)
{
  double a[4], b[4], res[4];

  __m256d x = _mm256_load_pd(a);
  __m256d y = _mm256_load_pd(b);

  x = _mm256_mul_pd(x, y);

  _mm256_store_pd(res, x);
}
```

The newest generation of the x86 processors from Intel/AMD features so-called SSE/AVX instructions that operator on vectors of up to 256 bits. This means that 4 double-precision or 8 single precision operations can be executed with a single such instruction, roughly in the same speed that an ordinary double or single-precision operation takes. So if these instructions can be used in an optimum way, one achieves a speed-up by a factor of up to 4 or 8, respectively. On the Intel Phi chips, the vector length has already doubled again and is now 512 bits, hence allowing another factor of 2 in the performance. Likely, we will see even larger vector lengths in the near future.

14.1.7 Hyperthreading

A general problem in exploiting modern computer hardare to its full capacity is that accessing main memory is very much slower than doing a single floating point operation in a compute core (moving data also costs more energy than doing a floating point calculation, which is becoming an ever more important point too). As a result, a compute core typically spends a large fraction of its cycles waiting for data to arrive from memory.

The idea of hyperthreading as implemented in CPUs by Intel and IBM (Power architecture) is to use this wait time by letting the core do some useful work in the meantime. This is achieved by "overloading" the compute core with several execution streams. But instead of letting the operating system toggle between their execution, the hardware itself can switch very rapidly between these different "hyperthreads". Even though there is still a considerable overhead in changing the execution context from one thread to another, this strategy can still lead to a substantial net increase in the calculational throughput on the core. Effectively, to the operating system and user it appears as if there are more cores (so called virtual cores) than there really are physical cores. For example, the IBM CPU on the Bluegene/Q machine has 16 physical cores with 4-fold hyperthreading, yielding 64 virtual cores. One may then start 64 threads in the user application. Compared with just starting 16 threads, one will then not get four times the performance, but still perhaps 1.8 times the performance or so.

14.2 Ahmdahl's law

Before we discuss some elementary parallelization techniques, it is worthwhile to point out a fundamental limit to the parallel speed-up that may be reached for a given program. We define the speed up here as the ratio of the total execution time without parallelization (i.e. when the calculation is done in serial) to the total execution time obtained when the parallelization is enabled.

Suppose we have a program that we have successfully parallelized. In practice, this parallelization is never going to be fully perfect. Normally there are parts of the calculation that remain serial, either for algorithmic reasons, due to technical

limitations, or we considered them unimportant enough that we have not bothered to parallelize those too. Let us call the *residual serial fraction* f_s, i.e. this is the fraction of the execution time spent in the corresponding code parts when the program is executed in ordinary serial mode.

Then Ahmdahl's law gives the maximum parallel speed up as

$$\text{maximum parallel speed up} = \frac{1}{f_s}. \tag{14.1}$$

This is simply because in the most optimistic case we can assume that our parallelization effort has been perfect, so that the time spent in the parallel parts approaches zero for a sufficiently large number of cores. The serial time remains unaffected by this, however, and does not shrink at all. The lesson is a bit sobering: Achieving large parallel speed-ups, say beyond a factor of 100 or so, also requires extremely tiny residual serial fractions. This is sometimes very hard to reach in practice, and for certain problems, it may even be impossible.

14.3 Shared memory parallelization with OpenMP

- Shared memory parallelization can be used to distribute a computational workload on the multiple available compute cores that have access to the same memory, which is where the data resides.
- UNIX processes are *isolated* from each other – they usually have their own protected memory, preventing simple joint work on the same memory space (data exchange requires the use of files, sockets, or special devices such as /dev/shm). But, a process may be split up into multiple execution paths, called *threads*. Such threads share all the resources of the parent process (memory, files, etc.), and they are the ideal vehicle for efficient shared memory parallelization.
- Threads can be created and destroyed manually, e.g. with the pthreads-libary of the POSIX standard. This is a bit cumbersome in practice. Here is an example how this can look in practice:

```
#include <pthread.h>

void do_stuff(void)
{
  pthread_attr_t attr;
  pthread_t mythread;
  int i, threadid = 1;

  pthread_attr_init(&attr);
  pthread_create(mythreads, &attr, evaluate, &threadid);

  for(i = 0; i < 100; i++)
    some_expensive_calculation(i);
}


void *evaluate(void *p)
{
  int i;

  for(i = 100; i < 200; i++)
    some_expensive_calculation(i);
}
```

- A simpler approach is to use the OpenMP standard, which is a language/compiler extension for C/C++ and Fortran. It allows the programmer to give simple hints to the compiler, instructing it which parts can be executed in parallel sections. OpenMP then automatically deals with the thread creation and destruction, and completely hides this nuisance from the programmer. As a result, it becomes possible to parallelize a code with minimal modifications, and the modified program can still be compiled and executed without OpenMP as a serial code. Here is how the example from above would like like in OpenMP.

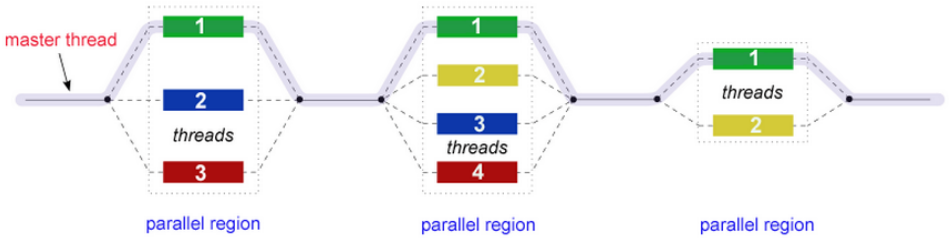
```
#include <omp.h>

void do_stuff(void)
{
  int i;

#pragma omp parallel for
  for(i = 0; i < 200; i++)
    some_expensive_calculation(i);
}
```

14.3.1 OpenMP's "fork-join-model"

The central idea of OpenMP is to let the programmer identify sections in a code that can be executed in parallel. Whenever such a section is encounter, the program execution is split into a number of threads that work in a team in parallel on the work of the section. Often, this work is a simple loop whose iterations are distributed evenly on the team, but also more general parallel sections are possible. At the end of the parallel section, the threads join again onto the master thread, the team is dissolved, and serial execution resumes until the next parallel section starts.



Normally, the number of threads used in each parallel section is constant, but this can also be changed through calls of the OpenMP runtime library functions, as in the above example.

14.3.2 Loop-level parallelism with OpenMP

The simplest and most important use of OpenMP is to parallelize a simple loop. Suppose for example we want to parallelize the loop:

```
for(i = 0; i < 200; i++)
  some_expensive_calculation(i);
```

This can be simply done by placing a special directive for the compiler in front of the loop:

```
#pragma omp parallel for
for(i = 0; i < 200; i++)
  some_expensive_calculation(i);
```

That's basically all. The OpenMP compiler will then automatically wake up all available threads at the beginning of the loop (the "fork"), it will then distribute the loop iterations evenly onto the different threads, and they are then executed concurrently. Finally, once all loop iterations have completed, the threads are put to sleep again, and only the master thread continues in serial fashion. Note that this will only work correctly if there are *no dependencies* between the different loop iterations, or in other words, the order in which they are carried out needs to be unimportant. If everything goes well, the loop is then executed faster by a factor close to the number of threads.

In order for this to work in practice, one has to do a few additional things:

- The code has to be compiled with an OpenMP capable compiler. This feature often needs to be enabled with a special switch, e.g. with gcc,

  ```
  gcc -fopenmp ...
  ```

  needs to be used.

- For some more advanced OpenMP features accessible through calls of OpenMP-library functions, one should include the OpenMP header file

  ```
  #include <omp.h>
  ```

- In order to set the number of threads that are used, one should set the `OMP_NUM_THREADS` environment variable before the program is started. Depending on the shell that is used (here bash is assumed), this can be done for example through

  ```
  export OMP_NUM_THREADS=8
  ```

  in which case 8 threads would be allocated by OpenMP. Normally one should then also have at least eight (virtual) cores available. The `omp_get_num_threads()` function call can be used inside a program to check how many threads are available.

14.3.3 Race conditions

When OpenMP is used, one can easily create hideous bugs if different threads may modify the same variable at the same time. Which thread wins the "race" and gets to modify a variable first is essentially undetermined in OpenMP (because the exact timings on a compute core will vary stochastically due to "timing noise" originating in interruptions from the operating system), so that subsequent executions will seemingly not produce deterministic behaviour.

Example: Double loop

A simple example is the following double-loop:

```
#pragma omp parallel for
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            do_stuff();
            .
            .
         }
      }
```

Here the simple OpenMP directive in the outer loop will instruct the `i`-loop to be split up. However, there is only one variable for `j`, *shared* by all the threads. They are hence not able to carry out the inner loop independent from each other! What is needed here is that each thread gets its own copy of `j`, so that the inner loop can be executed independently. This can be simply achieved by adding a `private` clause to the OpenMP directive of the outer loop, like this

```
#pragma omp parallel for private(j)
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            do_stuff();
            .
            .
         }
      }
```

Example: Reduction

Another common problem are reductions such as done in this example:

```
   int count = 0;

#pragma omp parallel for
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
             .
             .
             if(complicated_calculation(i) > 0)
               count++;
             .
             .
         }
      }
```

Here the loop is nicely parallelized by OpenMP, but we may nevertheless sometimes get an incorrect result for `count`. This is because the increment of this variable is not really carried out as a single instruction. It basically involves a read from memory, an addition of 1, and a write back. If now two threads happen to arrive at this statement at essentially the same time, they will both read `count`, increment it, and then write it back. But in this case the variable will end up being incremented only by one unit and not by two, because one of the threads is ignorant of the change of `count` by the other and overwrites it!

There are different solutions to this problem. One is to serialize the increment of `count` by putting a so-called lock around it. This can be done by enclosing it with a

```
   #pragma omp critical
     {
       count++;
     }
```

construct. But this can cost substantial performance: If one or several threads arrive at the statement at the same time, they have to wait and do the operation one after the other.

A better solution would be to have private variables for `count` for each thread, and only at the end of the parallel section add up the different copies to get the global sum. OpenMP can generate the required code automatically, all that is needed is to add the clause `reduction(+:count)` to the directive for parallelizing the loop:

```
   int count = 0;
```

```
#pragma omp parallel for reduction(+:count)
   for(i = 0; i < N; i++)
     {
       for(j = 0; j < N; j++)
         {
            .
            .
            if(complicated_calculation(i) > 0)
              count++;
            .
            .
         }
      }
```

A more detailed description of the OpenMP standard can for example be found in the very good online tutorial `https://computing.llnl.gov/tutorials/openMP`, or in various textbooks.

14.4 Distributed memory parallelization with MPI

To use multiple nodes in compute clusters, OpenMP is not sufficient. Here one either has to use special languages (like UPC, Co-Array Fortran, etc.) which are popular among theoretical computer scientists (but hardly anyone else), or one turns to the "Message Passing Interface" (MPI). MPI has become the de-facto standard for programming large-scale simulation code.

MPI offers library functions for exchanging messages between different processes running on the same or different compute nodes. The compute nodes do not necessarily have to be physically close, in principle they can also be loosely connected over the internet (although for tightly coupled problems the message latency makes this unattractive). Most of the time, the same program is executed on all compute cores (SPMD, "single program multiple data"), but they operate on different data such that the computational task is put onto many shoulders and a parallel speed up is achieved. Since the MPI processes are isolated from each other, all data exchanges necessary for the computations have to be explicitly programmed – this makes this approach much harder than, e.g., OpenMP. Often substantial program modifications and algorithmic changes are needed for MPI.

Once a program has been parallelized with MPI, it may also be augmented with OpenMP. Such hybrid parallel code may then be executed in different ways on a cluster. For example, if one has two compute nodes with 8 cores each, one could run the program with 16 MPI tasks, or with 2 MPI tasks that each using 8 OpenMP threads, or with 4 MPI tasks and 4 OpenMP threads each. It would not make sense to use 1 MPI task and 16 OpenMP threads, however – then only one of the two compute nodes could be used.

14.4.1 General structure of an MPI program

A template of a simple MPI program in C would look as follows:

```
#include <mpi.h>

int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);
    .
    .
   /* now we can send/receive message to other MPI ranks */
    .
    .
   MPI_Finalize();
}
```

- To compile this program, one would normally use a compiler wrapper, for example `mpicc` instead of `cc`, which simply sets a pathname correctly such that the MPI header files and MPI library files are found by the compiler.
- For executing the MPI program, one would normally use a start-up program such as `mpirun` or `mpiexec`. For example

  ```
  mpirun -np 8 ./mycalc
  ```

  could be used to launch 8 instances of the program `mycalc`.

If a normal serial program is augmented by `MPI_Init` in the above fashion, and if it is started multiple times with `mpirun -np X`, it will simply do multiple times exactly the same thing as the corresponding serial program. To change this behavior and achieve non-trivial parallelism, the execution paths in each copy of the program need to be somewhat different. This is normally achieved by making it explicitly depend on the *rank* of the MPI task. All the N processes of an MPI program form a so-called communicator, and they are labelled with a unique rank-id $0, 1, 2, \ldots, N-1$. MPI processes can then send and receive message from different ranks using these IDs to identify each other.

The first thing an MPI program normally does is therefore to find out how many MPI processes there are in the "world", and what the rank of the program itself is. This is done with the function calls

```
int NTask, ThisTask;

MPI_Comm_size(MPI_COMM_WORLD, &NTask);
MPI_Comm_rank(MPI_COMM_WORLD, &ThisTask);
```

The returned integers `NTask` and `ThisTask` then contain the number of MPI tasks and the rank of the current one, respectively.

14.4.2 A simple point to point message

With this information in hand, we can then exchange simple messages between two different MPI ranks. For example, a send of a message from rank 5 to rank 7 could be programmed like this:

```
int data[50], result[50]
.
.
if(ThisTask == 5)
  MPI_Send(data, 50, MPI_INT, 7,    /* buffer, size, type, destination */
           12345, MPI_COMM_WORLD);  /* message tag, communicator id    */

if(ThisTask == 7)
  MPI_Recv(result, 50, MPI_INT, 5,    /* receive buffer, size, type, sender */
           12345, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  /* tag, comm, status */
.
.
```

Here one sees the general structure of most send/recv calls, which always decompose a message into an "envelope" and the "data". The envelope describes the rank-id of sender/receiver, the size and type of the message, and a user-specified tag (this is the '12345' here), which can be used to distinguish messages of the same length.

Through the if-statements that depend on the local MPI rank, different execution paths for sender and receiver are achieved in this example. Note that if something goes wrong here, for example an MPI rank posts a receive but the matching send does not occur, it is simple to create deadlocks in a program. Here one or several of the MPI tasks get stuck in waiting in vain for messages that are not sent.

It is also possible to make MPI communications non-blocking and achieve asynchronous communication in this way. The MPI-2 standard even contains some calls for one-sided communication operations that do not always require direct involvement of both the sending and receiving sides.

14.4.3 Collective communications

The MPI standard knowns a large number of functions that can be used to conveniently make use of commonly encountered communication patterns. For example, there are calls for *broadcasts* which send the same data to all other MPI tasks in the same communicator. There are also *gather* and *scatter* operations that collect data elements from all tasks, or distribute them as disjoint sets to the other tasks. Finally, there are *reduction* function that allow one to conveniently calculate sums, minima, maxima, etc., over variables held by all MPI tasks in a communicator.

A detailed description of all these possibilities is way passed the scope of these brief lecture notes. Please check out some of the online resources (for example `https://computing.llnl.gov/tutorials/mpi`) or a text book is you want more information about this.

References

Atkinson, K. (1978), *An introduction to numerical analysis*, Wiley

Balsara, D. S., Rumpf, T., Dumbser, M., Munz, C.-D. (2009), *Efficient, high accuracy ADER-WENO schemes for hydrodynamics and divergence-free magnetohydrodynamics*, Journal of Computational Physics, 228, 2480

Barnes, J., Hut, P. (1986), *A Hierarchical O(NlogN) Force-Calculation Algorithm*, Nature, 324, 446

Bauer, A., Springel, V. (2012), *Subsonic turbulence in smoothed particle hydrodynamics and moving-mesh simulations*, MNRAS, 423, 2558

Bertone, G., Hooper, D., Silk, J. (2005), *Particle dark matter: evidence, candidates and constraints*, Physics Reports, 405, 279

Binney, J., Tremaine, S. (1987), *Galactic dynamics*, Princeton University Press

Binney, J., Tremaine, S. (2008), *Galactic Dynamics: Second Edition*, Princeton University Press

Brandt, A. (1977), *Multi-Level Adaptive Solutions to Boundary-Value Problems*, Mathematics of Computation, 31, 333

Briggs, W. L., Henson, V. E., McCormick, S. F. (2000), *A Multigrid Tutorial*, EngineeringPro collection, Society for Industrial and Applied Mathematics (SIAM, Philadelphia)

Campbell, J. E. (1897), *On a law of combination of operators bearing on the theory of continuous transformation groups*, Proc Lond Math Soc, 28, 381

Chandrasekhar, S. (1943), *Dynamical Friction. I. General Considerations: the Coefficient of Dynamical Friction.*, ApJ, 97, 255

Cooley, J. W., Tukey, J. W. (1965), *An algorithm for the machine calculation of complex Fourier series*, Math. Comp., 19, 297

Courant, R., Friedrichs, K., Lewy, H. (1928), *Über die partiellen Differenzengleichungen der mathematischen Physik*, Mathematische Annalen, 100, 32

Dehnen, W. (2000), *A Very Fast and Momentum-conserving Tree Code*, ApJL, 536, L39

Dehnen, W. (2002), *A Hierarchical O(N) Force Calculation Algorithm*, Journal of Computational Physics, 179, 27

Diniz, P., da Silva, E., Netto, S. (2002), *Digital Signal Processing: System Analysis and Design*, Cambridge University Press

Eckart, C. (1960), *Variation Principles of Hydrodynamics*, Physics of Fluids, 3, 421

Field, G. B. (1965), *Thermal Instability.*, ApJ, 142, 531

Gingold, R. A., Monaghan, J. J. (1977), *Smoothed particle hydrodynamics - Theory and application to non-spherical stars*, MNRAS, 181, 375

Goldstein, H. (1950), *Classical mechanics*, Addison-Wesley

Hairer, E., Lubich, C., Wanner, G. (2002), *Geometric numerical integration*, Springer Series in Computational Mathematics, Springer, Berlin

Harten, A., Lax, P. D., Van Leer, B. (1983), *On upstream differencing and Godunov-type schemes for hyperbolic conservation laws*, SIAM Review, 25, 35

Hernquist, L. (1987), *Performance characteristics of tree codes*, ApJS, 64, 715

Hockney, R. W., Eastwood, J. W. (1988), *Computer simulation using particles*, Bristol: Hilger

James, R. A. (1977), *The Solution of Poisson's Equation for Isolated Source Distributions*, Journal of Computational Physics, 25, 71

Kirkwood, J. G. (1946), *The Statistical Mechanical Theory of Transport Processes I. General Theory*, Journal of Chemical Physics, 14, 180

Klypin, A. A., Shandarin, S. F. (1983), *Three-dimensional numerical model of the formation of large-scale structure in the Universe*, MNRAS, 204, 891

Knebe, A., Green, A., Binney, J. (2001), *Multi-level adaptive particle mesh (MLAPM): a c code for cosmological simulations*, MNRAS, 325, 845

Kolmogorov, A. N. (1941), *Dissipation of Energy in the Locally Isotropic Turbulence*, Proceedings of the USSR Academy of Sciences, 32, 16

Landau, L. D., Lifshitz, E. M. (1959), *Fluid mechanics*, Course of theoretical physics, Oxford: Pergamon Press

LeVeque, R. J. (2002), *Finite volume methods for hyperbolic systems*, Cambridge University Press

Lucy, L. B. (1977), *A numerical approach to the testing of the fission hypothesis*, AJ, 82, 1013

Miyoshi, T., Kusano, K. (2005), *A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics*, Journal of Computational Physics, 208, 315

Mo, H., van den Bosch, F. C., White, S. (2010), *Galaxy Formation and Evolution*, Cambridge University Press

Monaghan, J. J. (1992), *Smoothed particle hydrodynamics*, ARA&A, 30, 543

Ollivier-Gooch, C. F. (1997), *Quasi-ENO Schemes for Unstructured Meshes Based on Unlimited Data-Dependent Least-Squares Reconstruction*, Journal of Computational Physics, 133, 6

Pope, S. B. (2000), *Turbulent Flows*, Cambridge University Press

Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P. (1992), *Numerical recipes in C. The art of scientific computing*, Cambridge: University Press, 1992, 2nd ed.

Price, D. J. (2012), *Smoothed Particle Hydrodynamics: Things I Wish My Mother Taught Me*, in *Advances in Computational Astrophysics: Methods, Tools, and Outcome*, edited by R. Capuzzo-Dolcetta, M. Limongi, A. Tornambè, volume 453 of *Astronomical Society of the Pacific Conference Series*, 249

Pringle, J. E., King, A. (2007), *Astrophysical Flows*, Cambridge University Press

Rankine, W. J. M. (1870), *On the thermodynamic theory of waves of finite longitudinal disturbances*, Philosophical Transactions of the Royal Society of London, 160, 277288

Rusanov, V. V. (1961), *Calculation of interaction of non-steady shock waves with obstacles*, J. Comput. Math. Phys. USSR, 1, 267

Saad, Y. (2003), *Iterative Methods for Sparse Linear Systems: Second Edition*, Society for Industrial and Applied Mathematics

Saha, P., Tremaine, S. (1992), *Symplectic integrators for solar system dynamics*, AJ, 104, 1633

Salmon, J. K., Warren, M. S. (1994), *Skeletons from the treecode closet*, J. Comp. Phys., 111, 136

Schaal, K., Bauer, A., Chandrashekar, P., Pakmor, R., Klingenberg, C., Springel, V. (2015), *Astrophysical hydrodynamics with a high-order discontinuous Galerkin scheme and adaptive mesh refinement*, MNRAS, 453, 4278

Shu, F. H. (1992), *The physics of astrophysics. Volume II: Gas dynamics.*, University Science Books, Mill Valley, CA

Springel, V. (2005), *The cosmological simulation code GADGET-2*, MNRAS, 364, 1105

Springel, V. (2010), *Smoothed Particle Hydrodynamics in Astrophysics*, ARA&A, 48, 391

Springel, V., Hernquist, L. (2002), *Cosmological smoothed particle hydrodynamics simulations: the entropy equation*, MNRAS, 333, 649

Stadel, J. G. (2001), *Cosmological N-body simulations and their analysis*, Ph.D. thesis, University of Washington

Stoer, J., Bulirsch, R. (2002), *Introduction to Numerical Analysis*, Texts in Applied Mathematics, Springer

Stone, J. M., Gardiner, T. A., Teuben, P., Hawley, J. F., Simon, J. B. (2008), *Athena: A New Code for Astrophysical MHD*, ApJS, 178, 137

Strang, G. (1968), *On the Construction and Comparison of Difference Schemes*, SIAM J. Numer. Anal., 5, 506

Teyssier, R. (2002), *Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES*, A&A, 385, 337

Toro, E. (1997), *Riemann solvers and numerical methods for fluid dynamics*, Springer

van Leer, B. (1984), *On the Relation Between the Upwind-Differencing Schemes of Godunov, Enguist, Osher and Roe*, SIAM J. Sci. Stat. Comput., 5, 1

van Leer, B. (2006), *Upwind and High-Resolution Methods for Compressible Flow: From Donor Cell to Residual-Distribution Schemes*, Communications in Computational Physics, 1, 192

White, S. D. M., Frenk, C. S., Davis, M. (1983), *Clustering in a neutrino-dominated universe*, ApJL, 274, L1