PH30056
Computational Physics B

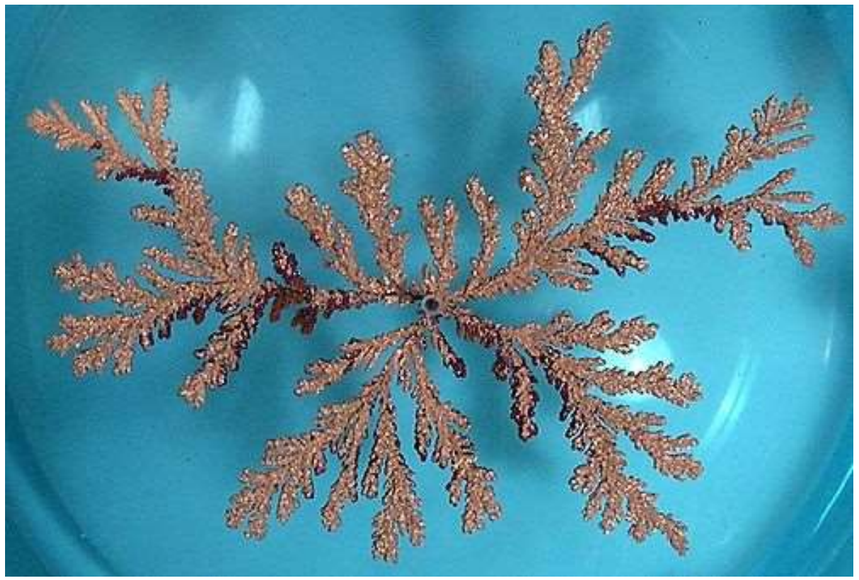
Alain Nogaret (A.R.Nogaret@bath.ac.uk)

Lecture 1
8 February 2024
Semester 2, academic year 2023/24

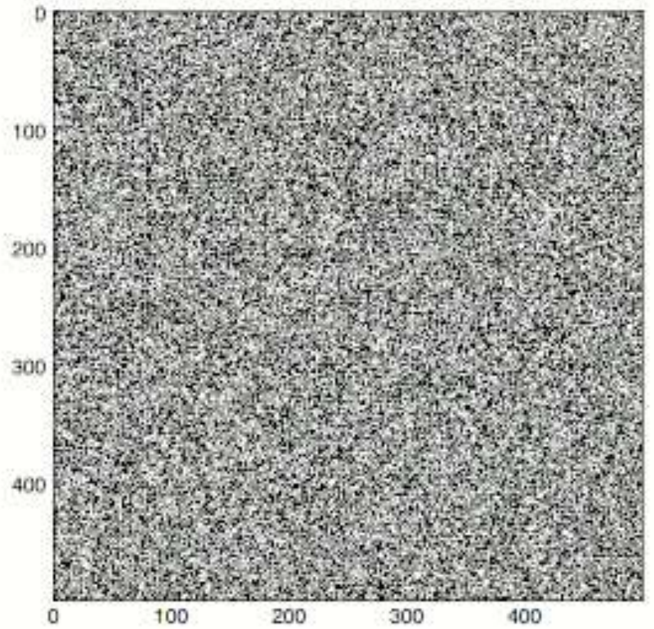
Co-instructor: David Tsang (dcwt21@bath.ac.uk)

Purpose of the unit

Learn and apply techniques to numerically simulate physical systems.



Topics:



1. Diffusion Limited Aggregation (DLA)

2. Ising model of magnetism

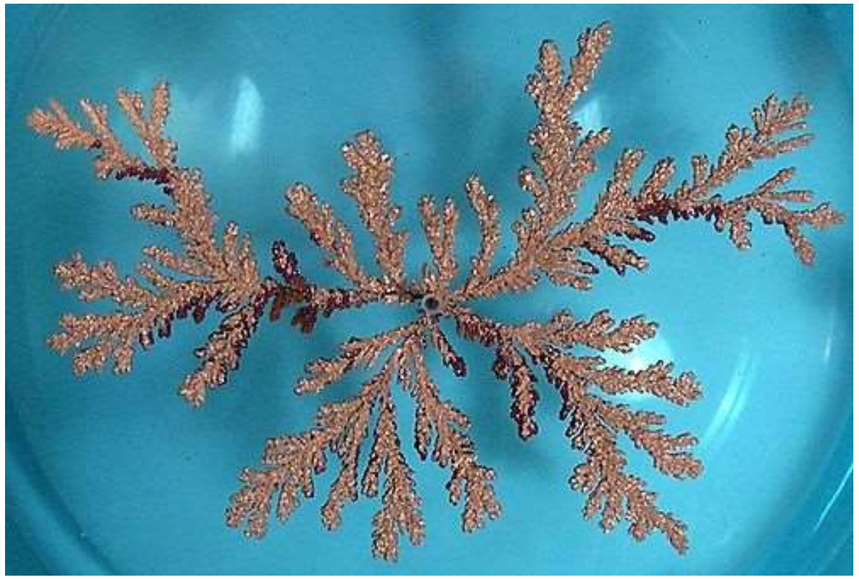
Focus: Physics

Tool: Computers (C++ code, visualisation libraries)

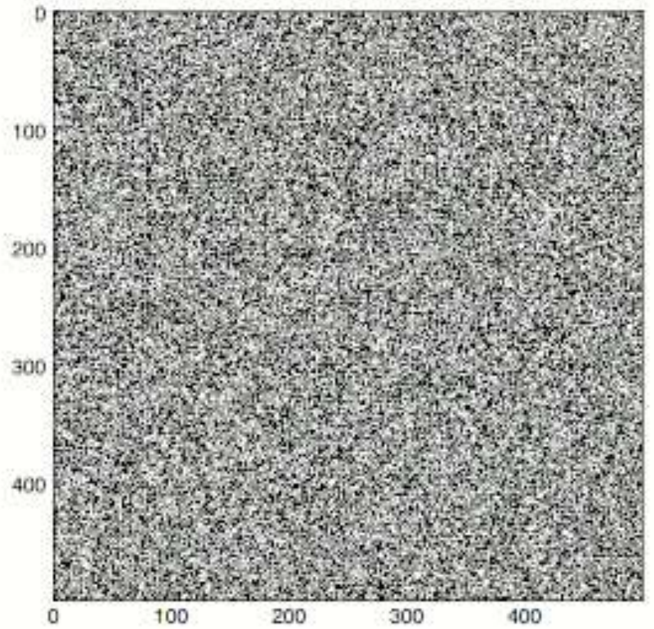
(Images: Wikipedia)

Purpose of the unit

Learn and apply techniques to numerically simulate physical systems.



Topics:



1. Diffusion Limited Aggregation (DLA)

2. Ising model of magnetism

Focus 2: Self-management and independent project implementation

(Images: Wikipedia)

The physics of Diffusion Limited Aggregation

Recommended textbook:
Chapter 16: Fractals and Statistical Growth Models
Computational physics: problem solving with Python
Rubin H. Landau, Manuel J. Paez and Cristian C. Bordeianu.

Available online through library (see Moodle).

Diffusion-limited aggregation (DLA)

K-F Braun *et al*., Surface Science **454-456**, 750 (2000)
http://dx.doi.org/10.1016/S0039-6028(00)00178-3

Vapour deposition of lithium fluoride (pale grey) on a silver metal surface (dark grey)

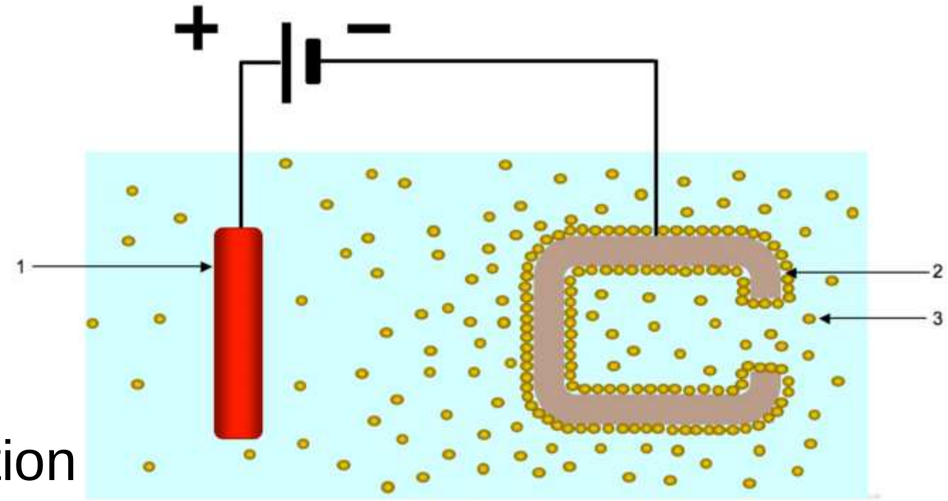
Scanning Tunnelling Microscope STM image, area: 240nm x 240 nm

What is the origin of these patterns?

How should we analyse them?

Why study diffusion-limited aggregation (DLA)?

Dielectric breakdown



Electrodeposition

Industrial use
(often to be avoided)

Dielectric breakdown
(closely related dielectric breakdown model)

Battery failure

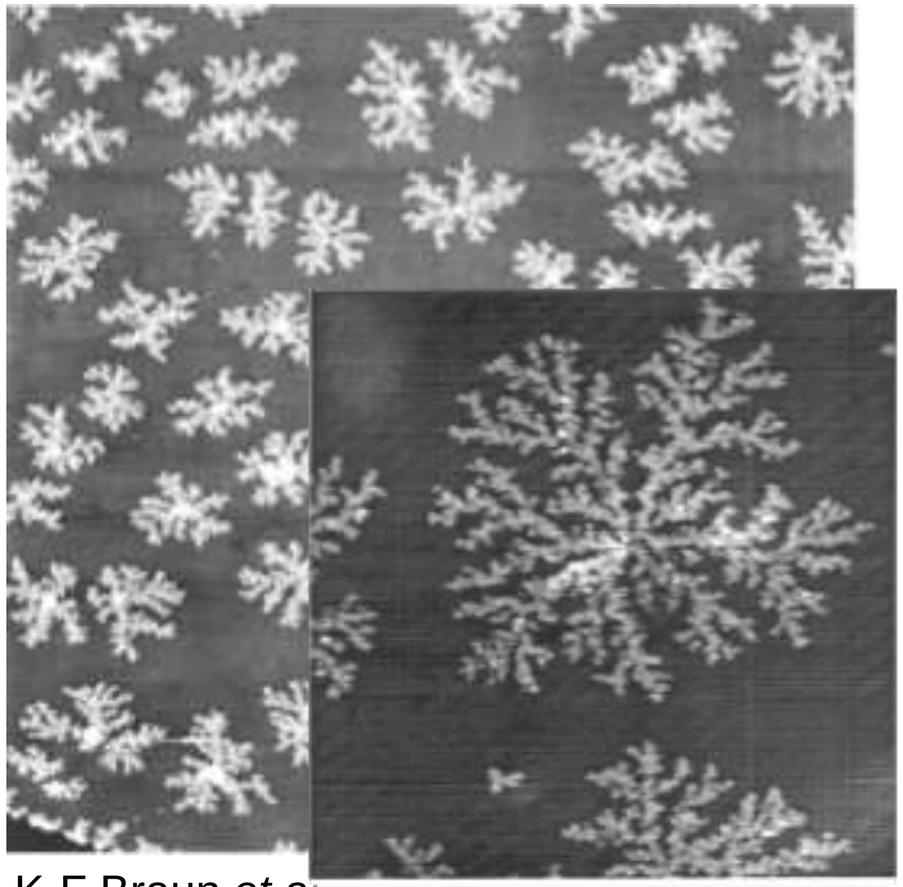
Mineral deposits

Deposition:
For example, electrodeposition

(wikipedia)

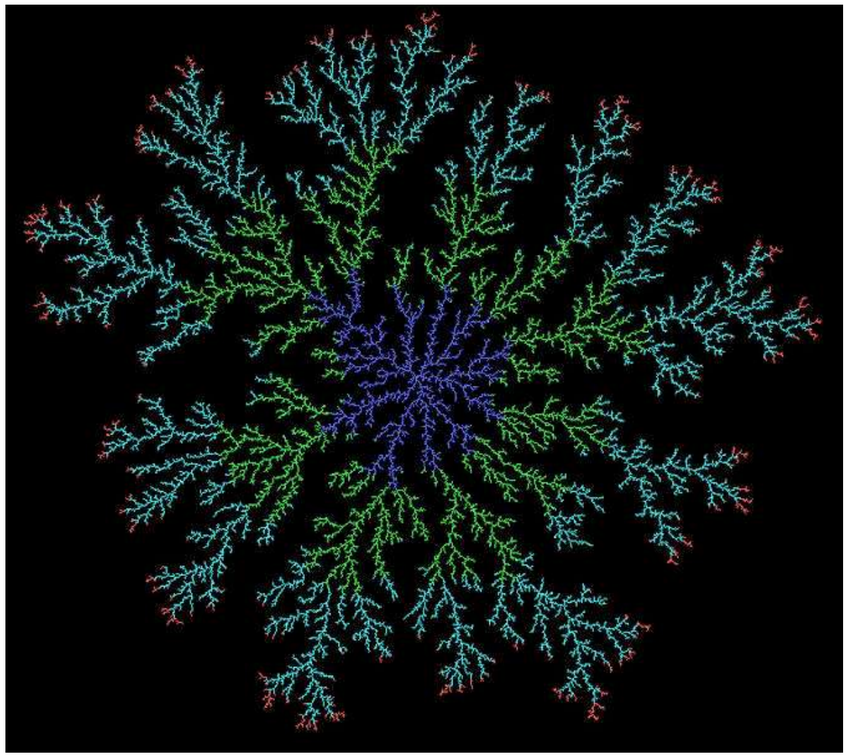
Diffusion-limited aggregation (DLA)

Experiment



K-F Braun *et al*.,
Surface Science **454-456**, 750 (2000)

Computational model

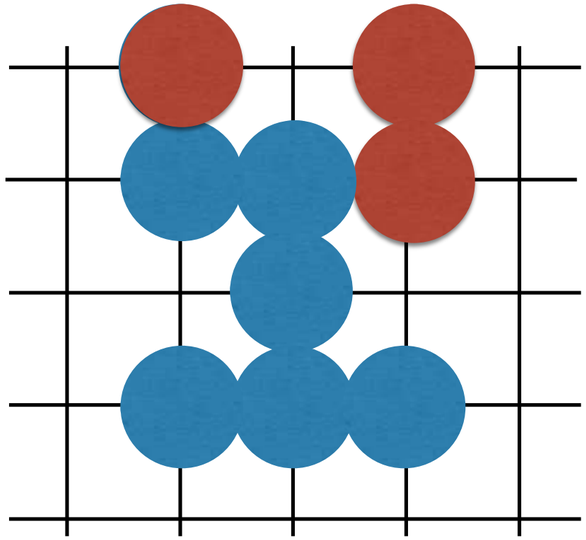


https://commons.wikimedia.org/wiki/File:Of7_p0001_15h.jpg

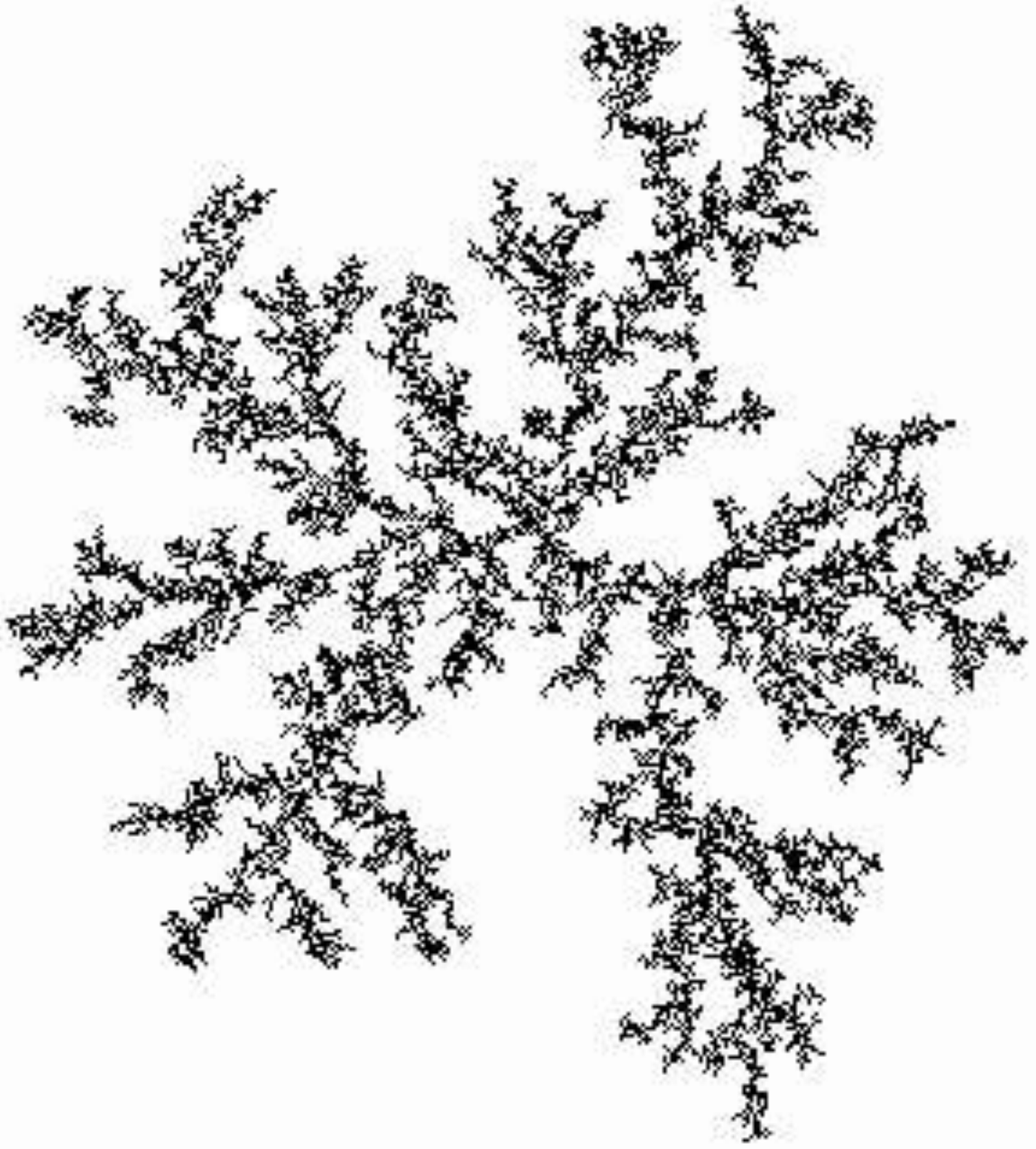
Diffusion-limited aggregation (DLA)

Simple idea: [TA Witten and LM Sander, Phys. Rev. Lett. **47**, 1400 (1981)]

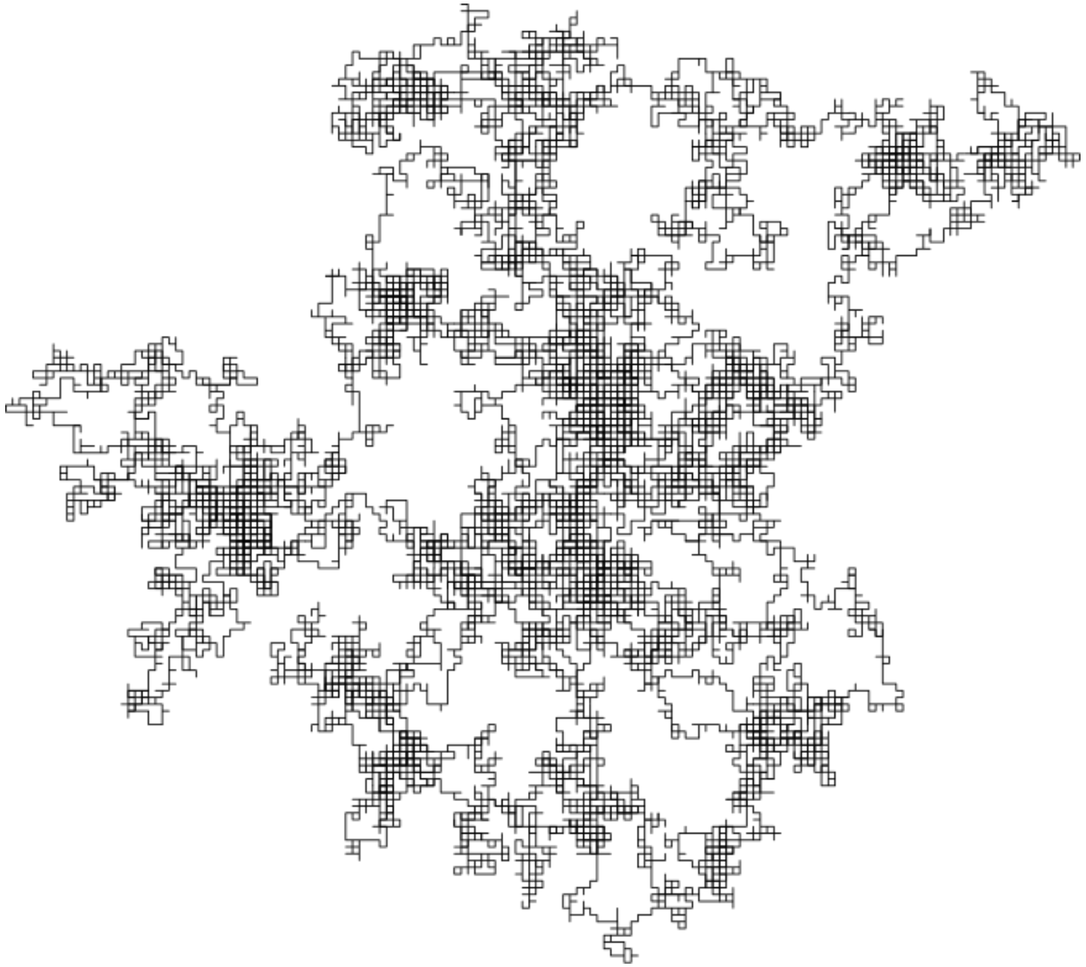
1. Consider a large lattice (or grid), where each site can contain at most one particle
2. Start with one stationary particle at the centre of the lattice.
3. Add another particle, far from the origin, and let it wander around at random until it touches an existing particle. When this happens, the particle immediately becomes stationary and never moves again.
4. Repeat step 3 many times



… etc

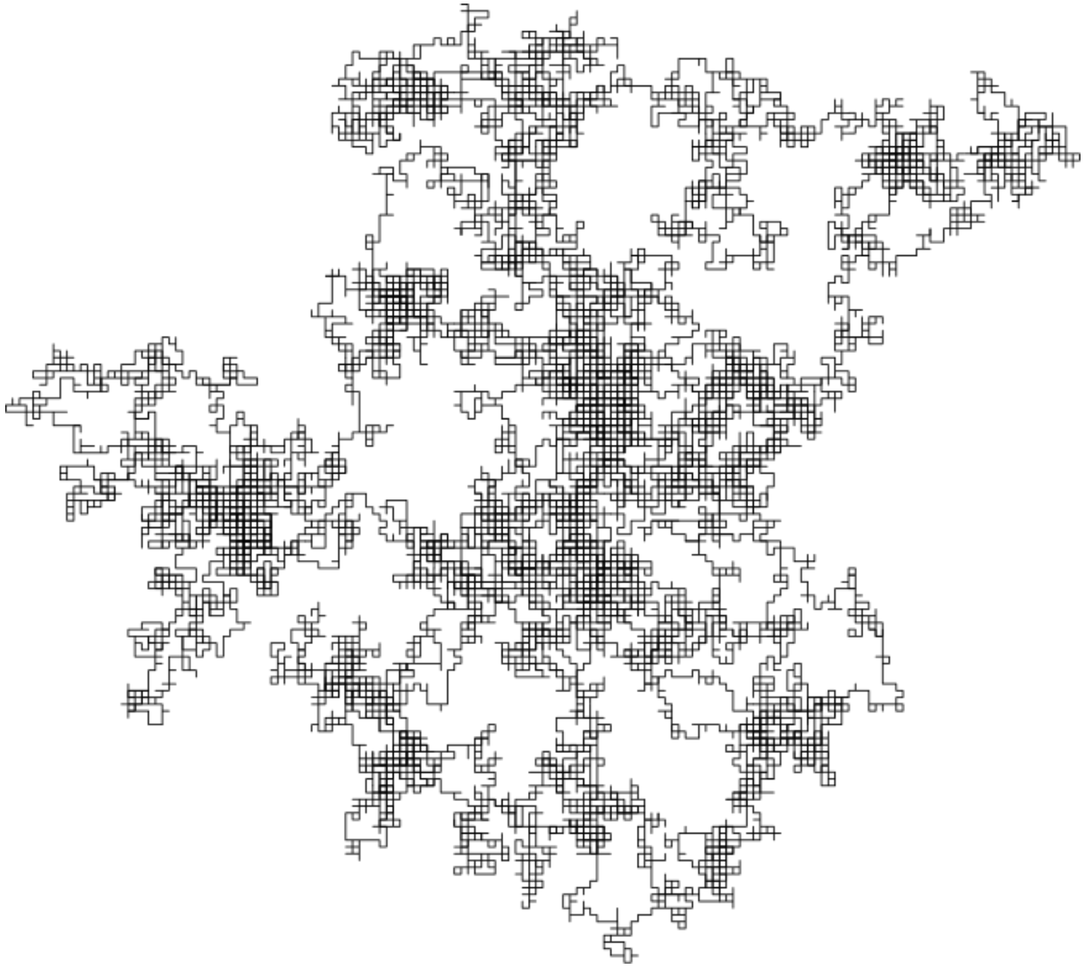


Diffusion



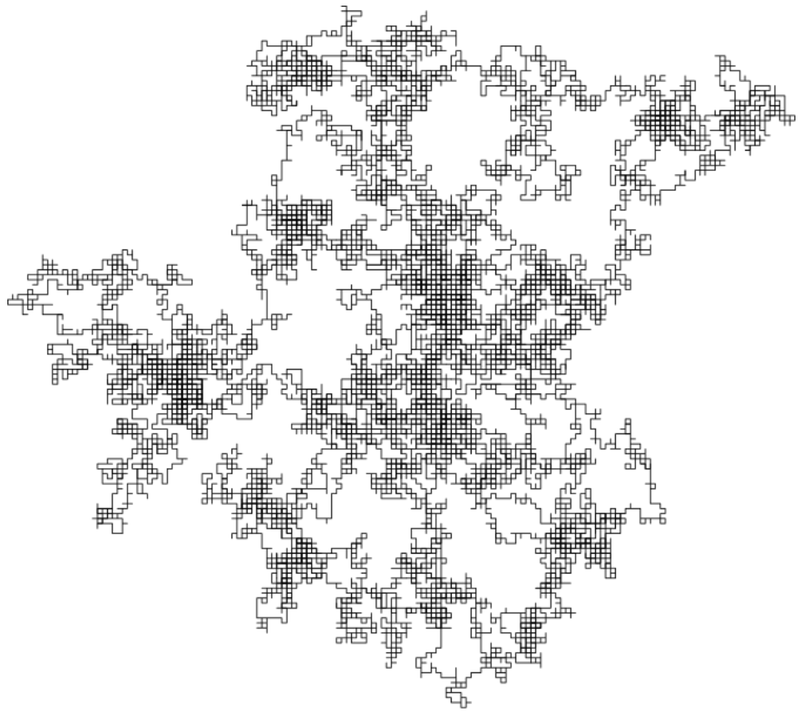
A single particle starts at the origin: where will it go?

Diffusion



By considering the average displacement (squared) of particle, we can go from statistical model to deterministic law

Diffusion: calculations



$\vec{r}_n$ is the (random) displacement at each step

Average of any individual $\vec{r}_n$ is $\langle \vec{r}_n \rangle = 0$
(random directions)

Hence $\langle \vec{R} \rangle = \sum_{n=1}^{N} \langle \vec{r}_n \rangle = 0$

$$\langle |\vec{R}|^2 \rangle = \left\langle \left| \sum_{n=1}^{N} \vec{r}_n \right|^2 \right\rangle = \left\langle \sum_{n=1}^{N} \sum_{m=1}^{N} \vec{r}_m \cdot \vec{r}_n \right\rangle$$

$$= \sum_{n=1}^{N} \langle \vec{r}_n \cdot \vec{r}_n \rangle = Na^2$$

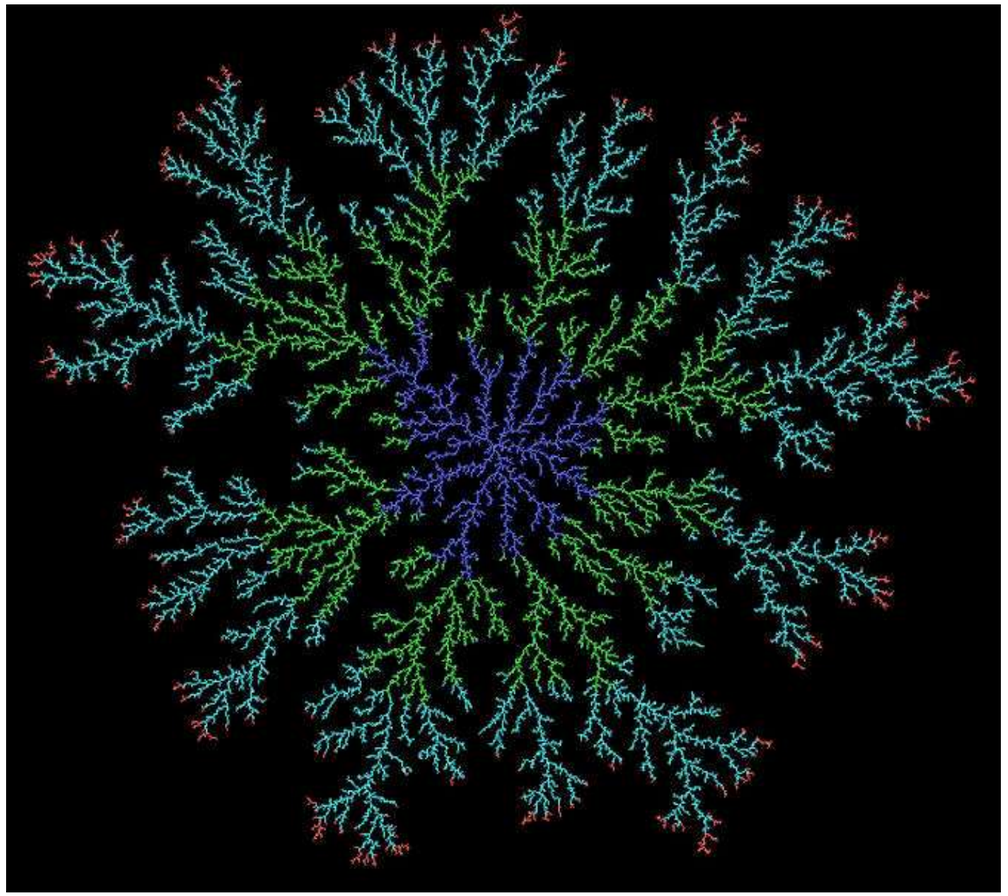
Root-mean-square "size" is
$R_p = \langle |\vec{R}|^2 \rangle^{1/2} = aN^{1/2}$

Used $\langle \vec{r}_n \cdot \vec{r}_m \rangle = 0$ for $n \neq m$,
because link directions are independent.

How to analyse the shape of a (random) cluster?

Model needs to be verifiable:
Calculate statistical properties of cluster shape and compare with experiment.

Cluster shape is self-similar (zoom in on each arm: looks like a shrunk copy).
This is a **fractal**!



Cluster from diffusion limited aggregation model

Diffusion-limited aggregation (DLA)

Questions to think about

- What does it mean if the results of the model look the same as the experiment(s)?

  (Probably the experimental setups are not as simple as the model. Which features of the experiments does the model get right? Which does it get wrong?)

- Why does the cluster have this shape? Can we use theory to understand this?

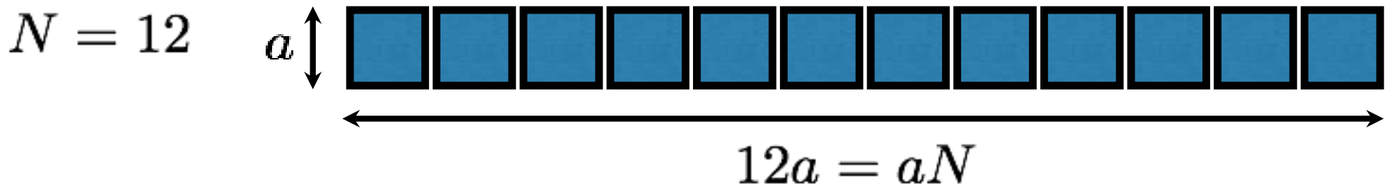
- How do the results of the model change if we make small changes to the rules?

- Which quantities are most interesting to measure (and compare with experiment)?

What is a *dimension*?

Consider a line built from many small squares

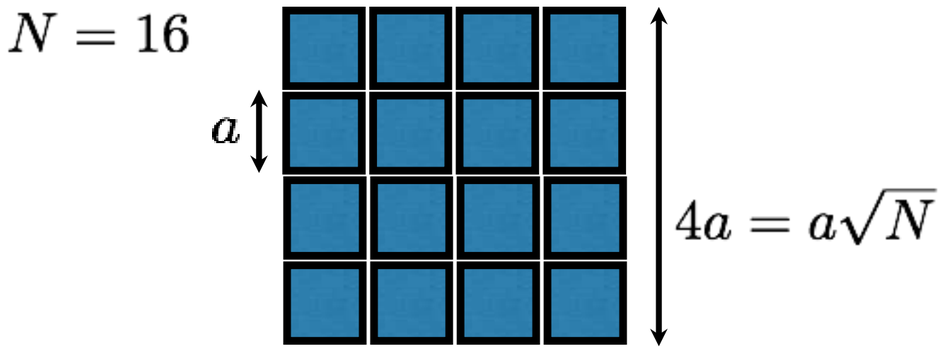
If the number of squares is N and each one has sides of length a, then they form a long line of length aN.



d = 1

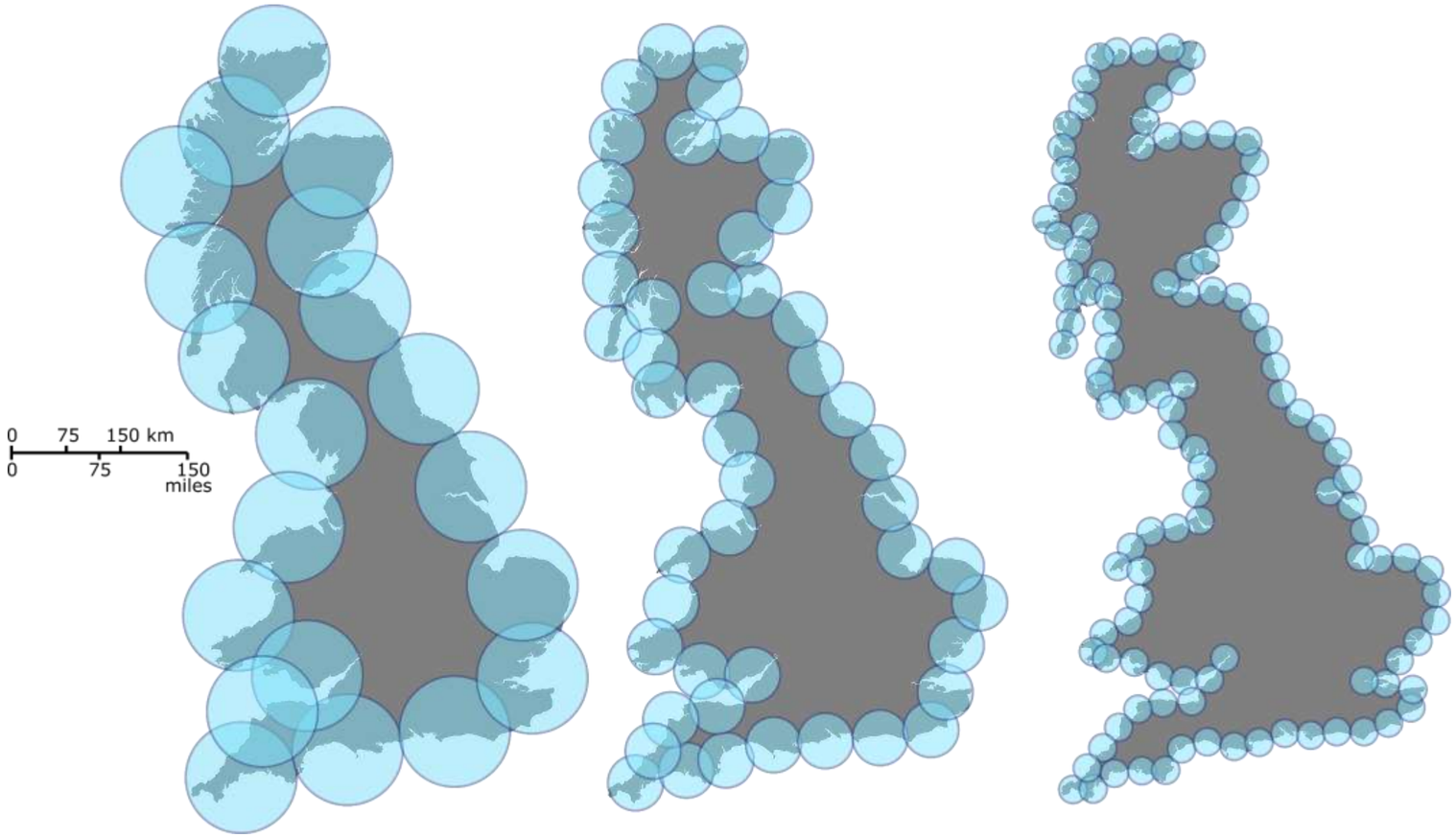
Consider a square built from many smaller squares

If the number of squares is N and each one has sides of length a, then they form a big square with sides of length $a\sqrt{N} = aN^{1/2}$.



d = 2

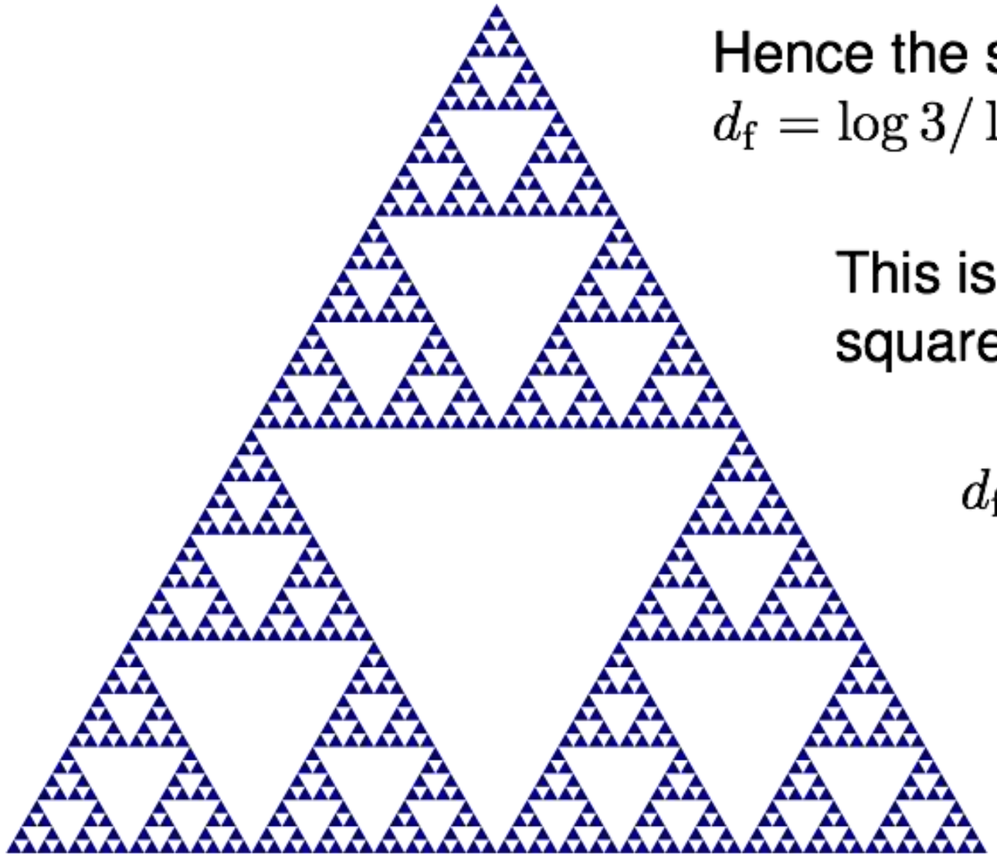
What is a *dimension*?



d ≈ 1.25

Fractal example: Sierpinski triangle

This shape is generated by a hierarchical algorithm. At stage n of the algorithm it consists of $N = 3^n$ triangles (each of side length a) and has side length $a \times 2^n$.



Hence the size is aN^{1/d_f} with $d_\mathrm{f} = \log 3 / \log 2 \approx 1.585$.

This is in between a line ($d_\mathrm{f} = 1$) and a square (or triangle) for which $d_\mathrm{f} = 2$.

d_f is called a “fractal dimension”

Note if the size is R, the density is $N/R^2 \approx N^{-0.415}/a^2$ which tends to zero as $N \to \infty$.

Measuring fractal dimension

In the situations considered so far we have $R = aN^{1/d_\mathrm{f}}$ for some "microscopic" length parameter a

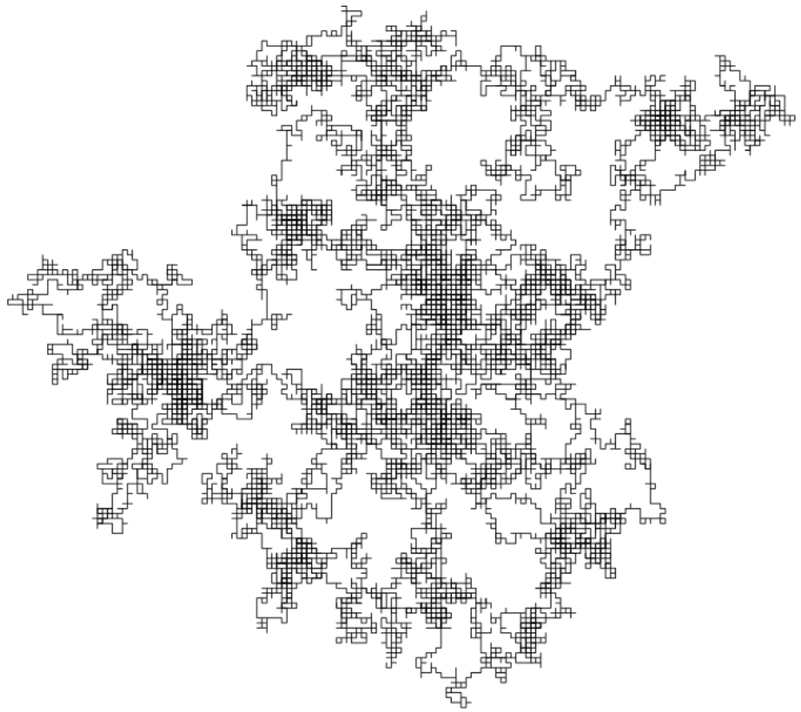
In general for fractals we expect $R = aN^{1/d_\mathrm{f}} + (\dots)$ where a is a "microscopic" length scale, and $(\dots)$ stands for correction terms that are irrelevant for large N

Two ways to estimate d_f:

Assume a is known and neglect $(\dots)$ in which case $d_\mathrm{f} = \frac{\ln N}{\ln(R/a)}$
or
Neglect $(\dots)$, plot $\ln N$ against $\ln R$ and take the gradient.

Note: in general a is not known so the second is preferable; in either case one should use N as large as possible to avoid effects of $(\dots)$

Diffusion: fractal dimension



$\vec{r}_n$ is the (random) displacement at each step

Average of any individual $\vec{r}_n$ is $\langle \vec{r}_n \rangle = 0$
(random directions)

Hence $\langle \vec{R} \rangle = \sum_{n=1}^{N} \langle \vec{r}_n \rangle = 0$

$$\langle |\vec{R}|^2 \rangle = \left\langle \left| \sum_{n=1}^{N} \vec{r}_n \right|^2 \right\rangle = \left\langle \sum_{n=1}^{N} \sum_{m=1}^{N} \vec{r}_m \cdot \vec{r}_n \right\rangle$$

$$= \sum_{n=1}^{N} \langle \vec{r}_n \cdot \vec{r}_n \rangle = Na^2$$

Root-mean-square "size" is
$R_p = \langle |\vec{R}|^2 \rangle^{1/2} = aN^{1/2}$

Used $\langle \vec{r}_n \cdot \vec{r}_m \rangle = 0$ for $n \neq m$,
because link directions are independent.

Hence $d_f = 2$
(in 3d space)

Why measure the fractal dimension?

Verify model: We want to compare cluster shapes between theory and experiment

DLA happens on a grid and has simple rules for attachment, etc
… *details* of particle arrangement will *not* match experiment

In some situations, fractal dimensions are “universal”
… the idea is that large-scale structure may not depend on microscopic details of the model / system (e.g., type of lattice)

If this is the case, then experimental fractal dimensions *can* be compared with theory / computation for “simple” models…
… if they match then the model seems to capture some of the important features of the experiment

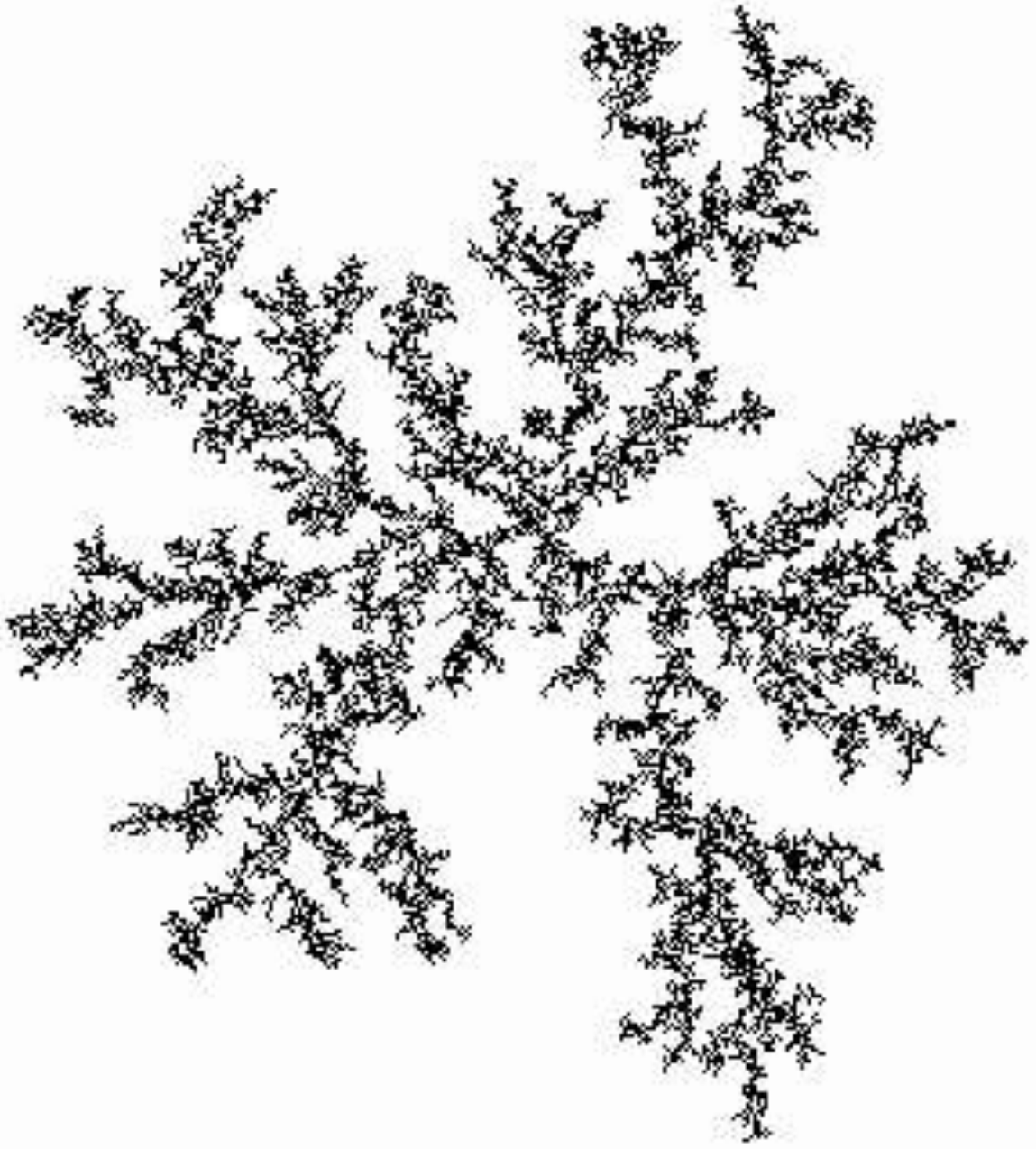
Summary

Diffusion-limited aggregation is a simple model that leads to growing clusters with branching shapes

The fractal dimension is a useful way to analyse objects' shapes, and to compare theory with experiment
… this is especially true if fractal dimensions are "universal"

Scaling theory is a powerful framework for analysing dynamical processes such as diffusion

One question behind your coursework :
to what extent do DLA clusters have "universal" fractal dimensions?



PH30056
Computational Physics B

Alain Nogaret (A.R.Nogaret@bath.ac.uk)

Lecture 2
Semester 2, academic year 2023/24

Co-instructor: David Tsang (dcwt21@bath.ac.uk)

Comments before starting

This course uses C++, an object-oriented programming (OOP) language

This lecture is about OOP in C++

BUT this unit is not about OOP in C++ (!)

This unit is about *physics*. The idea is to use computer programs to understand systems' behaviour.

You will need to spend some effort learning C++ but the main focus of the unit (and the marking) is on the physics. So you need to balance your time…

Some literature

About computational physics:

- K. N. Anagnostopoulos: Computational Physics – A Practical Introduction to Computational Physics and Scientific Computing using C++, 2016 (Online for free)
- R. Fitzpatrick: Computational Physics (Online for free)

About C++ programming:

- A. C. Kak: Programming with Objects – A Comparative Presentation of Object-Oriented Programming with C++ and Java, 2003 (Online for free, through the Library)
- https://www.cprogramming.com/tutorial/c++-tutorial.html
- http://www.cplusplus.com/doc/tutorial/
- https://isocpp.org/faq

Hello world! in C and C++

```c
#include <stdio.h>

int main() {
    printf( “Hello world! \n” );
    getchar();
    return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    cin.get();
}
```

Hello world! in C++

```
#include <iostream>

using namespace std;

int main() {
    // Comment line
    cout << "Hello world!" << endl;
    cin.get();
}
```

- `#include` is a pre-processor directive. This adds the content of `iostream` header file
- `iostream` includes the declarations of functions performing standard input and output operations
- `std` stands for the C++ Standard Library and it is needed because `cout, cin` and `endl` are part of it

Hello world! in C++

```
#include <iostream>

using namespace std;

int main() {
    // Comment line
    cout << "Hello world!" << endl;
    cin.get();
}
```

- `int` specifies that the variable returned by function main( ) is an integer
- `main()` is the entry point of the C++ program.
- `main()` is a function with body `{ }`
- Brackets ( ) are used to hold the argument of a function.
- `//` is used for commenting, also /* insert comment */ works

Hello world! in C++

```
#include <iostream>

using namespace std;

int main() {
    // Comment line
    cout << "Hello world!" << endl;
    cin.get();
}
```

- `cout` stands for “console output”
- `<<` means that we write to `cout`
- The string is within “ “ and `endl` denotes the end of line
- Note that all the command lines end with semicolon `;`
- `cin.get()` is used here simply to stop the program

Hello world! in C++

```
#include <iostream>

int main() {
    // Comment line
    std::cout << "Hello world!" << std::endl;
    std::cin.get();
}
```

- Note: if we omit `using namespace std`, we can still access this namespace by adding the `std::` prefix
- Note: `main` returns an integer, and C++ automatically returns 0 for the `main` if it executes succesfully.

If statements in C++

```
if ( TRUE ) {
    // statements
}
else if ( TRUE ) {
    // statements
}
else {
    // statements
}
```

Loops in C++

```
for ( int x = 0; x < 10 ; x++ ) {
    // statements
}
```

```
int x = 0;
while ( x < 10 ) {
    // statements
    x++;
}
```

```
int x = 0;
do {
    // statements
    x++;
} while ( x < 10 );
```

Switch case in C++

```
int nmbr;

cout << “Give an integer number between 1 – 3.” << endl;
cin >> nmbr;

switch ( nmbr ) {
case 1:
    // statement
    break;
case 2:
    // statement
    break;
case 3:
    // statement
    break;
}
```

“Call by reference” in C++

You might remember from C that if functions need to modify their arguments in the calling program, you need a pointer.

Notation is simplified further in C++, where the address &x of argument x is passed to the function

```
void cubeMe(double& x) {     // This is C++ code
  x = x * x * x;
}
                                // Note the  &  sign!
int main() {
  double myVar=2.0;
  // pass the variable myVar to the function
  cubeMe( myVar );
  cout << myVar << endl;
  // now myVar has the value 8
}
```

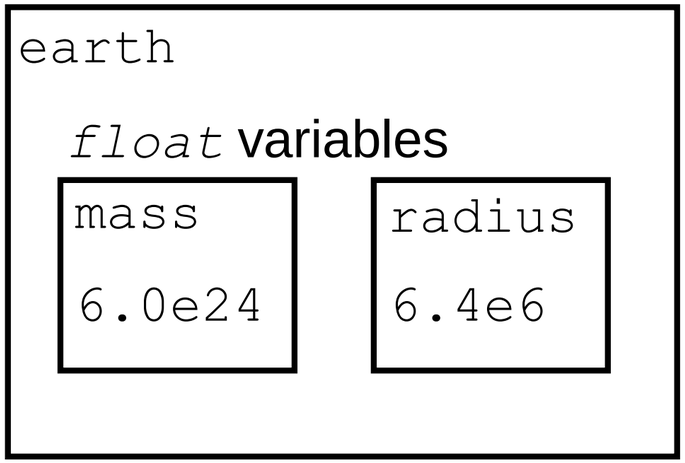
Objects in C++

- C++ is an object oriented programming (OOP) language
- For example, in the Hello world! program, `cout`, `cin` were objects.

- To understand what objects are let's first recall what was `struct` in C programming

Example of C structure

Planet structure



In C, a `struct` is a data structure that includes several (or many) *member variables*, each identified by a name.

For example, `earth.mass` might be the mass (member) of the Earth, where `earth` is an instance of a `Planet` structure.

```
typedef struct pt {
   float mass;
   float radius;
} Planet;


main() {
   // declare earth
   Planet earth;

   // set member variables
   earth.mass  = 6.0e24;
   earth.radius = 6.4e6;

   // rest of program goes here
}
```

Classes (objects) in C++

A planet is an example of a *physical* object

In object-oriented programming, an “object” is a **data structure** (*struct*) that includes the **operations** (*methods*) that manipulate these data (for example, a menu, an equation solver, …)

Objects are represented by `classes`.
Classes are similar to `structs` except that a `class` can include member *functions* (called *methods*) as well as member *variables*

The idea is to group together bits of code that are responsible for different tasks. You can use objects as inputs and outputs to functions, etc, just like `structs`

This is very useful when planning and writing complicated programs…
(… but it can be confusing at first)

Example of Planet class in C++

Planet class in C++

earth

float variables

mass	radius
6.0e24	6.4e6

Methods:

`float calcVolume();`

returns $\frac{4}{3}\pi R^3$

`float calcDensity();`

returns $M/(\frac{4}{3}\pi R^3)$

See later for
"public" / "private"

We might now write in C++:

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};
```

Example of Planet class in C++

Definition of Planet class

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};
```

Information about *what a Planet is* and *what it does*

A set of computations that we want to perform

… main program

```
int main() {
  // declare earth as a Planet
  Planet earth;
  // set member variables
  earth.mass  =6.0e24;
  earth.radius=6.4e6;

  float rho;
  rho=earth.calcDensity();

  // rest of program..
}
```

Example of Planet class in C++

Definition of Planet class

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};
```

Why not 4 / 3 ?

The class definitions end with ;

A set of computations that we want to perform

… main program

```
int main() {
  // declare a Planet
  Planet earth;
  // set member variables
  earth.mass  =6.0e24;
  earth.radius=6.4e6;

  float rho;
  rho=earth.calcDensity();

  // rest of program..
}
```

Class definition inside the main C++ file

```
#include <iostream>
using namespace std;

class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};

int main() {
   Planet earth; // declare a Planet
   earth.mass  =6.0e24; // set member variables
   earth.radius=6.4e6;

   float rho;
   rho = earth.calcDensity();

  // rest of program..
}
```

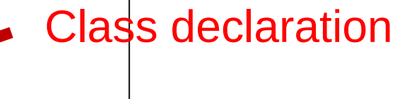
The class declarations / definitions can be, for example, written in the same cpp-file before the main program…

Class in a header file in C++

This class may be declared in a header file Planet.h without defining the functions

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};
```

Source.cpp –file has the main program




```
#include <iostream>
#include "Planet.h"
// include Planet header
// note the use of  "  "
using namespace std;

int main() {

Planet earth;
  // set member variables
earth.mass  =6.0e24;
earth.radius=6.4e6;

float rho;
rho=earth.calcDensity();
    // rest of program..
}
```

Multiple files for readability

Planet.h has the class declaration

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume();
    float calcDensity();
};
```

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24; // default values
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}
// function definition
float Planet::calcVolume() {
    float R = radius;
    return (4 / 3.0)*3.141*R*R*R;
}
// function definition
float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume();
    float calcDensity();
};
```

The main program below.

```
#include <iostream>
#include "Planet.h"
using namespace std;

int main() {
     // our program
}
```

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24; // default values
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}
// function definition
float Planet::calcVolume() {
    float R = radius;
    return (4 / 3.0)*3.141*R*R*R;
}
// function definition
float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Objects in C++

A **class** is a way to collect together variables and functions that are involved in similar tasks

For example: physical particles might be defined as objects. Relevant functions might be concerned with moving particles, drawing particles on the screen, etc.

Other classes are associated with menus, random number generators, etc.

The idea is to provide structure for the code and keep it organised…
… to make use of this *you* will have to think and plan carefully as you edit and improve the code

There are many built-in classes in C++ such as `ofstream`

```
#include <iostream>
#include <fstream> // for interacting with files

using namespace std;

int main() {

    char fname[11] = "opfile.txt"; /* Note: opfile.txt has
                      only 10 characters.
                      Strings end with "\0" null character,
                      therefore, one extra character. */
    ofstream logfile; // Here we create an ofstream object.
    logfile.open(fname); // We open the file

    logfile << "Hello world!" << endl;
    // We write in the file
    logfile.close(); // We close the file
}
```

Writing in a file in C++ using the built-in `ofstream` class

We first created an `ofstream` object and we called it `logfile`, and then we could carry out operations like opening and closing files

```
char fname[11] = "opfile.txt";
ofstream logfile;
logfile.open(fname)
logfile.close();
```

We can also check whether the file is still open before we close it using an if statement

```
if ( logfile.is_open() )
{
    logfile.close();
}
```

Pointers in C++

As you may remember, pointers are used to store addresses of other variables. Here is a simple example of an integer pointer.

```
int main() {
   int firstVar, secondVar;
   int *ptr; /* declare pointer to an integer */
   firstVar = 5; secondVar = 2;

   /* store the address of the first variable */
   ptr = &firstVar;

   /* set in secondVar the value that ptr points to */
   secondVar = *ptr;
   /* secondVar now has the value 5 */
}
```

Pointers to objects in C++

Declaration of Planet class

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
};
```

```
/* we got the same result
earlier by writing */
float rho;
rho=earth.calcDensity();
```

… main program

```
main() {
   // declare a Planet
 Planet earth;
   // set member variables
 earth.mass  =6.0e24;
 earth.radius=6.4e6;
   // Planet pointer
 Planet *ptrEarth;
   // store the address
 ptrEarth = &earth;

 float rho;
  /* compute density using
  the pointer */
 rho=ptrEarth->calcDensity();

  // rest of program..
}
```

Users and developers

An idea in object-oriented programming (OOP) is that programmers can interact with classes in *two different ways*

If you *create a class* or *edit the details of how it works*, you are a **developer**

If you *use the class* for some purpose, without caring how it works, you are a **user**

Examples:
You will be **users** of a `Window` class in C++
You will be **developers** of a `Particle` class, and a `DLASystem` class (in the DLA project)

`public` / `private`

When **developing** a class, it is useful to think about how you (or other people) will **use** it in future.

C++ (and other OOP languages) can help with this, using keywords such as `public` and `private`

Member variables and methods can be `public` or `private`

`public` / `private`

When **developing** a class, it is useful to think about how you (or other people) will **use** it in future.

C++ (and other OOP languages) can help with this, using keywords such as `public` and `private`

Member variables and methods can be `public` or `private`

Loosely speaking, `private` members of a class can only be accessed from **inside** the class, while `public` members can be accessed from **anywhere**

You can think of `private` as “for developers” and `public` as “for users”

(There are other slightly more complicated options but `public` and `private` should be sufficient for our discussion.)

public / private

```
class myClass {
 public:
  void userFunc() {
    int x=3;
    developerFunc(x); // this is OK (inside class)
  }
 private:
  void developerFunc(int x) {
    // restricted access function…
  }
};
```

```
int main(void) {
    myClass myObj;       // create an instance of myClass
    myObj.userFunc();       // this is OK
    myObj.developerFunc(); // NOT ok (will not compile)
                           // (we are outside the class)
}
```

Function “overloading”

Sometimes one might have several functions that do *similar jobs*, but with *different types of input*

C++ (and most OOP languages) allow you to define functions that have *the* **same name** but **different input types**
(maybe think of different versions of the same function)

Roughly speaking, the compiler works out which version of the function to use, according to the type of input

This can be painful for the *developer* but it is great for the *user* … it provides flexibility as to how the class is used

Function “overloading”

Simple example of a counter that uses a private variable…

```
class Counter {
   private:
    // pCount is like a read-only variable
    double pCount;

   public:
    // read the variable
    double readCount() { return pCount; }

    // reset counter to zero
    void   setZero() { pCount = 0.0; }

    // three methods for incrementing the counter
    void Increment() { pCount += 1.0; }
    void Increment(double incr) { pCount += incr;}
    void Increment(int incr)    { pCount += incr;}
};
```

Function “overloading”

```
class Counter {
   private:
    // pCount is like a read-only variable
    double pCount;
   public:
    double readCount() { return pCount; }
    void   setZero() { pCount = 0.0; }
    // three methods for incrementing the counter
    void Increment() { pCount += 1.0; }
    void Increment(double incr) { pCount += incr;}
    void Increment(int incr)    { pCount += incr;}
};
```

Repeated from previous page

```
int main(void) {
  Counter myCounter;          // declare a counter

  myCounter.setZero();        // set it to zero
  myCounter.Increment();      // increase by 1.0
  myCounter.Increment(3.0); // increase by 3.0
  cout << “count “ << myCounter.readCount() << endl;
}
```

Function “overloading”

It is also possible to have “default inputs”, for example:

```
class Counter {
  public:
    // … initial part of class not shown…

    // methods for incrementing the counter
    // increment by a double, use 1.0 by default
    void Increment(double incr = 1.0) { pCount += incr;}

    // this one is as before
    void Increment(int incr)     { pCount += incr;}

    // this one is now redundant
    // void Increment() { pCount += 1.0; }
};
```

Using this definition, the previous example program does exactly the same, but the class definition is slightly simpler…

Function “overloading”

```
class Counter {
   private:
    // pCount is like a read-only variable
    double pCount;
   public:
    double readCount() { return pCount; }
    void   setZero() { pCount = 0.0; }
    // two methods for incrementing the counter
    void Increment(double incr = 1.0) { pCount += incr;}
    void Increment(int incr)          { pCount += incr;}
};
```

new version…
… one line shorter

```
int main(void) {
  Counter myCounter;          // declare a counter

  myCounter.setZero();        // set it to zero
  myCounter.Increment();      // increase by 1.0
  myCounter.Increment(3.0); // increase by 3.0
  cout << “count “ << myCounter.readCount() << endl;
}
```

same as before…

Constructors

When declaring (or “creating”) new instances, there are often general tasks that need to be done

C++ (and most OOP languages) can help with this via special functions called **constructors** which are called when new instances are created

Constructor functions have the same name as the class itself(!) and they have no output type

Constructors are not compulsory but they are useful.
A class can have multiple constructors (“overloading”)

There are also **destructors**, see later

Constructors: example

```
class Planet {
  public:
    double mass;
    double radius;
    // constructor
    Planet(double _mass, double _radius) {
      mass = _mass; radius = _radius;
    }
};
```

there is nothing special about the _ character, its just useful to avoid making up new names

```
int main(void) {
  // create a new planet and set up its variables,
  // all at the same time
  Planet earth(6e24, 6.4e6);

  cout << "mass is " << earth.mass << endl;
  // will print (something like) "mass is 6.0e24"
}
```

Overloading constructors

It can be useful to have several constructors:
when creating a new class, one might want to use different types of data for initialisation

```
class Planet {
  public:
    double mass;
    double radius;
    Planet(double _mass, double _radius) {
      mass = _mass; radius = _radius;
    }
    // "default" constructor
    Planet() {
      mass = 1.0; radius = 1.0;
    }
    // copy planet at address p
    // (not the same as a "copy constructor")
    Planet( Planet *p ) {
      mass = p->mass; radius = p->radius;
    }
};
```

Note: `p->mass` is a pointer reference to the mass variable in the Planet class!

Dynamical allocation and the `new` keyword

There are several ways to create objects in C++

```
int main(void) {
  // create a new Planet called earth
  // (use the default constructor, if one exists)
  Planet earth;

  // create a planet using a constructor
  Planet jupiter(1.9e27,7e7);

  // create a pointer to a planet
  Planet *planetPtr;

  // create a new planet and use the pointer to
  // store its address. (use default constructor.)
  planetPtr = new planet();

  // rest of program goes here
}
```

Dynamical allocation…

If we use `new` then the program allocates up some memory
… we have to take care to tell the computer when we are done

```
int main(void) {
  Planet *planetPtr;

  // create a new planet and use the pointer to
  // store its address. (use default constructor.)
  planetPtr = new Planet();

  // rest of program goes here

  // when we have finished we should delete
  // the Planet that is pointed to by the pointer
  delete planetPtr;
}
```

Dynamical allocation…

We can also use `new` to create arrays, and `delete[]` to destroy them (as malloc does in C)

```
int main(void) {
  cout << "how big is your list?" << endl;

  int listLength;
  // this is a way to read an integer
  cin >> listLength;

  double *listPtr;  // this is a pointer
  // make a new array
  listPtr = new double[listLength];

  // now we can use the array, for example
  listPtr[0] = 3.0;
  // .. main program goes here

  // free the memory at the end
  // we need to include [] when deleting arrays
  delete[] listPtr;
}
```

Dynamical allocation...

Destructor functions have names that match the class name, but begin with a tilde: ~

These functions are called (automatically) whenever an object is deleted

Destructors (obviously) have no inputs

```
class Particle {
  public:
    int dimension;          // are we in (eg) 2d or 3d?
    double *position;       // this will be the position vector

    Particle(int dim) {  // constructor
      dimension = dim;
      position = new int[dimension];  // array for position
    }

    ~Particle() {       // destructor, free up the memory
      delete[] position;
    }
};
```

for every `new` there should be a `delete`

Conclusions

Object oriented programming is powerful.

It can make your programs **easier to read**

It can make life easier for users through **flexibility** (eg with overloading) and by hiding complicated stuff that we don't need to worry about (with private variables etc)

However, there are often many different ways to write programs that do almost the same thing. Some ways are good and some are bad, it takes time and experience to learn which are which.
(The most common problem is that you write a program to do one thing, and then you want to modify it in order to do something slightly different. Good programs are easier to modify and adapt.)

For the purposes of the courseworks, the most important thing is the **physics** (and not the programming)!

PH30056: Computational Physics B: Lecture 3

Semester 2, Academic year 2024/2025

Dr. David Tsang (D.Tsang@bath.ac.uk)

Co-instructors: Prof. Alain Nogaret (A.R.Nogaret@bath.ac.uk); Prof Alison Walker (psyabw@bath.ac.uk)

Content of this lecture

- Revision on the topics of the previous lecture
  - Functions
  - Classes and objects
  - Arrays, pointers and strings
  - Dynamic allocation of memory
- Vectors
- Streams and stringstreams
- Structure of the DLA C++ code

[The slides will be available on Moodle!]

Function calls in C++

```
void cubeMe(double& x) {
  x = x * x * x;
}

int main(void) {
  double myVar = 2.0;
  cubeMe( myVar );
  cout << myVar << endl;
  // What will come out?
}
```

```
void cubeMe(double x) {
  x = x * x * x;
}

int main(void) {
  double myVar = 2.0;
  cubeMe( myVar );
  cout << myVar << endl;
  // What will come out?
}
```

Function calls in C++

```
void cubeMe(double& x) {
  x = x * x * x;
}

int main(void) {
  double myVar = 2.0;
  cubeMe( myVar );
  cout << myVar << endl;
  // returns 8
}
```

This will return 8 because the cubeMe function is allowed to modify its input (pass by reference).

```
void cubeMe(double x) {
  x = x * x * x;
}

int main(void) {
  double myVar = 2.0;
  cubeMe( myVar );
  cout << myVar << endl;
  // returns 2
}
```

The will return 2 because the cubeMe function is not allowed to modify its input (pass by value).

If we use a “pass by value” function, how we should modify the code to return 8?

Function calls in C++

```
void cubeMe(double& x) {
  x = x * x * x;
}

int main(void) {
  double myVar = 2.0;
  cubeMe( myVar );
  cout << myVar << endl;
  // returns 8
}
```

```
double cubeMe(double x) {
  return x * x * x;
}

int main(void) {
  double myVar1 = 2.0;
  double myVar2;
  myVar2 = cubeMe( myVar1 );
  cout << myVar2 << endl;
  // now it returns 8
}
```

We need to give it a returning type and return something.

Function calls in C++

```
double cubeMe( double x = 4.0 ) {
  return x * x * x;
}

int main(void) {
  double myVar1 = 2.0;
  double myVar2, myVar3;
  myVar2 = cubeMe( myVar1 );
  myVar3 = cubeMe();
  cout << myVar2 << endl; // what do we get?
  cout << myVar3 << endl; // what do we get?
}
```

Function calls in C++

```
double cubeMe( double x = 4.0 ) {
  return x * x * x;
}

int main(void) {
  double myVar1 = 2.0;
  double myVar2, myVar3;
  myVar2 = cubeMe( myVar1 );
  myVar3 = cubeMe();
  cout << myVar2 << endl; // we get 8
  cout << myVar3 << endl; // we get 64
}
```

For a “call by value” function, we can set a default input which is used if the user does not set any input.

Classes and objects in C++

- What is the difference between a class and an object?

Classes and objects in C++

- What is the difference between a class and an object?

  Class is a user defined data type with member variables and member functions (i.e. a piece of code)

  Object is an instance of the class (the code of the class is executed and a new variable, i.e. object, is created)

Inline class in a header file in C++

Planet.h (header file) contains full definitions.

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
 private:
  float period;
};
```

Inline class in a header file in C++

Planet.h (header file) contains full definitions.

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
 private:
  float period;
};
```

- `public` (user) or `private` (developer):
  - `public` allows any part of the program (including parts outside the class) to access these members
  - `private` members can be accessed only within the class.
- By default, everything is `private` if not stated otherwise

Inline class in a header file in C++

Planet.h (header file) contains full definitions.

```
class Planet {
 public:
  float mass;
  float radius;
  float calcVolume() {
    float R = radius;
    return (4/3.0)*3.141*R*R*R;
  }
  float calcDensity() {
    return mass/calcVolume();
  }
 private:
  float period;
};
```

Source.cpp –file has the main program

```
#include <iostream>
#include "Planet.h"
// include Planet header
// note the use of  "  "
using namespace std;

int main() {

    Planet earth;
    earth.mass = 1.0;
    earth.radius = 1.0;

    earth.period = 1.0;
    /* this does NOT work
    because period is
    private! */
}
```

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume();
    float calcDensity();
};
```

The main program below.

```
#include <iostream>
#include "Planet.h"
using namespace std;

int main() {
     // our program
}
```

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24; // default values
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}
// function definition
float Planet::calcVolume() {
    float R = radius;
    return (4 / 3.0)*3.141*R*R*R;
}
// function definition
float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume() {
     float R = radius;
     return (4/3.0)*3.141*R*R*R;
    }
    float calcDensity();
};
```

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24;
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}

// function definition
float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Will this work?

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume() {
     float R = radius;
     return (4/3.0)*3.141*R*R*R;
    }
    float calcDensity();
};
```

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24;
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}

// function definition
float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Will this work? Yes, but it is not a very nice syntax…

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume() {
     float R = radius;
     return (4/3.0)*3.141*R*R*R;
    }
    float calcDensity();
};
```

Will this work?

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24;
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}

float Planet::calcVolume() {
  float R = radius;
  return (4/3.0)*3.141*R*R*R;
}

float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Multiple files for readability

Planet.h has the class declarations

```
class Planet {
public:
    Planet();
    ~Planet();
    float mass;
    float radius;
    float calcVolume() {
     float R = radius;
     return (4/3.0)*3.141*R*R*R;
    }
    float calcDensity();
};
```

Will this work? No, the same function must not be defined twice!

Planet.cpp has the function definitions

```
#include "Planet.h"

// constructor
Planet::Planet() {
    mass = 6e24;
    radius = 6.4e6;
}
// destructor
Planet::~Planet() {
}

float Planet::calcVolume() {
  float R = radius;
  return (4/3.0)*3.141*R*R*R;
}

float Planet::calcDensity() {
    return mass / calcVolume();
}
```

Arrays

Arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

```
int array1[10]; // This declares an array
int twodimensionalarray2[4][4]; /* This declares a two
                                   dimensional array */

array1[0] = 5;  /* modifies the first value
                   note: indexing starts from 0 ! */

Planet pltArray[5]; // array of 5 Planet objects
```

Pointers

What are pointers?

Why are pointers useful?

Pointers

What are pointers? Pointers are memory addresses of variables

Why are pointers useful?

- Arrays are basically pointers (array name is a pointer to its first element)
- In function calls, pointers are needed if the values of the input variables need to be changed
- In function calls, it is often much more memory efficient to send the memory address of the variable instead of sending the data contained by the variable (which may be a large structure or object, for example)
- Pointers are needed for dynamic allocation of memory (later in the slides)

Pointers: simple example

```
int *ptr; // declares a pointer to an integer variable

int Variable1 = 5; // integer variables
int Variable2;

ptr = &Variable1; /* saves the memory address of
                     Variable1 to the pointer */

Variable2 = *ptr; /* de-referencing the pointer:
                  extracts the value pointed by the pointer,
                  i.e. Variable2 has now value 5 */
```

Arrays and pointers

```
int main() {

    int arr[5] = {1,2,3,4,5}; // create array of length 5
    int *ptr;

    ptr = arr; /* array name is also a pointer to the 1st
                  element of the array and we do not need to
                  use  &  sign. */

    cout << arr[0] << endl; // ?
    cout << ptr << endl; // ?
    cout << *ptr << endl; // ?

 }
```

Arrays and pointers

```
int main() {

    int arr[5] = {1,2,3,4,5}; // create array of length 5
    int *ptr;

    ptr = arr; /* array name is also a pointer to the 1st
                  element of the array and we do not need to
                  use  &  sign. */

    cout << arr[0] << endl; /* prints 1 (the first element of
                                        the array) */
    cout << ptr << endl; /* prints memory address
                                    (e.g. 008FF724)  */
    cout << *ptr << endl; /* dereferencing pointer, prints the
                                    content, 1 */


}
```

Pointers and objects

- When a member variable or function is called using a pointer, the correct syntax is to use `->` symbol
- `ptr->mv` : go to the object pointed to by `ptr` and look at the member variable that is called `mv`
- `ptr->mf()` : go to the object pointed to by `ptr` and execute the member function `mf()`

```
main() {
    // Planet pointer
 Planet *ptrPlanet;
    // dynamic allocation
    // default constructor
 ptrPlanet = new Planet();

 float ms;
 float rho;
  /* compute density using
  the pointer */
 ms =ptrPlanet->mass;
 rho=ptrPlanet->calcDensity();

  // rest of program..
}
```

Dynamic allocation of memory

Why is this useful?

Dynamic allocation of memory

Why is this useful?

Declaring an array with a fixed size can result in many problems: for example, one can exceed the maximum length of the array.

One solution would be to allocate large amount of memory for the array (to make the array long enough). But this is not efficient use of memory.

Dynamic allocation of memory is a better option since it allows the user to input the required array size.

Dynamic allocation of Arrays

A dynamically allocated array is declared as a pointer (memory is not yet allocated).

When the size of an array is known (input by the user), the memory is allocated with the `new` operator and the address is saved in the pointer.

Afterwards, we need to free the allocated memory with `delete`.

```
int *arr; // pointer to int, initialize to nothing.
int n; // variable for the length of the array

cout << "Give length of the array " << endl;
cin >> n; // the user inputs the size

arr = new int[n]; /* allocate an array of n integers and save
                     address in ptr */
   // rest of the code...

delete[] arr; /* when done, free the memory (deletes the
                                              array) */
```

Dynamic allocation of Objects

The dynamic allocation of objects is carried out with the help of pointers and the `new` keyword (similar to arrays). Afterwards, `delete` releases the allocated memory.

```
int main(void) {
    // create a pointer to a planet
  Planet *planetPtr;
    // create a new planet using the pointer to
    // store its address. (use default constructor.)
  planetPtr = new Planet();

    //  --- rest of the program here ---

    // when we have finished we release the memory
  delete planetPtr;
}
```

C-strings

Fixed size string:

```
char str[256] /* declaration of a string with a length
                 of 256 characters */

cout << "Give text" << endl;
cin >> str; // input text (do not exceed the size n !)
   // rest of the program...
```

Note: C strings are always **terminated** by a **null** character (i.e. the example above has in total 255 characters + 1 null)

C++ Strings

C++ style string (remember to add library `#include<string>`)

Declaring a C++ string:

```
string my_str1;
string my_str2("Initial text"); // constructor
string my_str3;
```

Note that C++ strings do not end with a null character!

Strings are supported by `cin`:

```
cin >> my_str1;
```

To store an entire line (that ends with the enter key):

```
getline(cin, my_str3);
```

There are many member functions in string class. For example, to return the length of the string, you can use the `length` function

```
int L = my_str2.length(); // L equals to 12
```

Streams

We have previously seen a command like this

```
cout << "value of myVar is " << myVar << endl;
```

Remember, `cout` stands for "console output" and you can think of << as passing information to the output

`cout` is an example of an output stream, other examples are files and `stringstreams`

```
// remember to add #include <fstream> in the beginning !!
// these commands could all be inside main()

  int myInt = 5; // data that needs to be saved

   ofstream myFile; // declare an output file stream object

     // create/open a file
   myFile.open("outputFile.txt");
     // print to file
  myFile << "print value to file " << myInt << endl;
    // remember to close the file in the end !!
  myFile.close();
```

Streams

C++ doesn't always have nice syntax but you can do lots of things… Here, `out` means that the file is open for writing (ofstream object is actually, by default, open for writing), and `app` means we go to the end of the file and append to its existing content (i.e. we do not overwrite the content)

```
// these commands could all be inside main()

// declare an output file stream
  ofstream myFile;

// open an existing file for "appending"
// (adds the data in the end of the file)
  myFile.open("outputFile.txt", fstream::out | fstream::app);

// print to the file, then close it
  myFile << "print value to file " << myInt << endl;

  myFile.close();
```

String streams

```
// you need to #include <sstream>
// and         #include <string>   in the beginning !!

// these commands could all be inside main()

 string       myString;     // instance of string class
 stringstream stringMaker;  // instance of stringstream class

 // use the stringstream to assemble our new string
stringMaker << "string contents: " << myInt;
                        // note: there is no  << endl;  here !!
 // set myString to store this new string
 // the str() member function returns the string itself
 myString = stringMaker.str();

// print the string
cout << myString << endl;
```

Vectors

Container classes are objects that hold other objects.

Vectors have nice properties: for example, their size can be changed and they accept different data types.

```
#include <vector> // remember to add this !
#include <iostream>
using namespace std;

// this will print the even numbers 0-18
int main(void) {
  vector<int> myList(10);  // list of 10 integers
                           // similar to int myList[10];
  for (int i=0;i<10;i++)
    myList[i] = 2*i;

  for (int i=0;i<10;i++)
    cout << myList[i] << endl;

  return 0;
}
```

Vectors

```
  // these commands could all be inside main()

  // make a list of 10 integers all with value 3
  vector<int> myList(10, 3);

  myList.resize(20);  // change the size of the list to 20

    // print the size of the list (this will be 20)
  cout << "list size " << myList.size() << endl;

  // add the number 4 to the end of the list
  // (and increase the size by 1)
  myList.push_back(4);

  // delete the last entry in the list
  // (and decrease size by 1)
  myList.pop_back();
}
```

push: add to list

pop: remove from list

Vectors

You can have any kind of class inside < >, for example a pointer to an object…

```
// make a list of doubles
// since no size is given then we start with an empty list
// (i.e. the initial size will be zero)
vector<double> dList;

// make a list of pointers to Particle objects
vector<Particle*> particleList;
```

This last example appears in the 1st course work.

General: You can *always* use a `vector` instead of an array, this can often be useful…

DLA project: Visual Studio 2022

- All the files need to be associated with a project:

  New Project → Empty Project Visual C++

  or Open Project (if you already have one)

- If you want to create a new main file

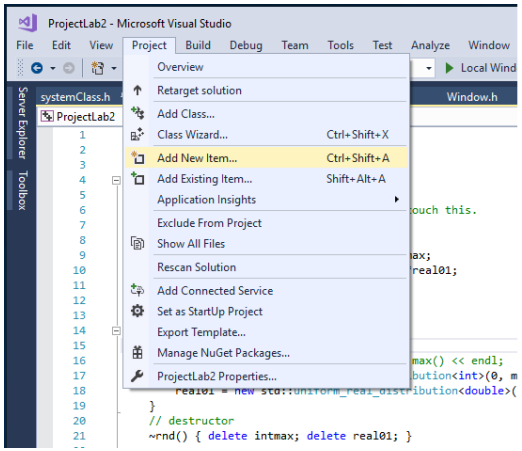
  Project menu → Add New Item → C++ File (.cpp)
  But remember to have only one file with `main()` !

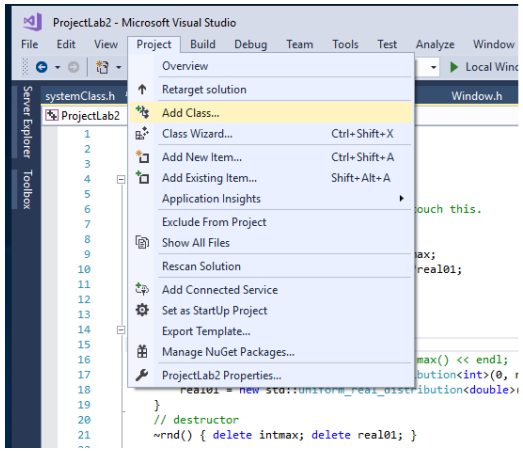
- If you want to create a new class

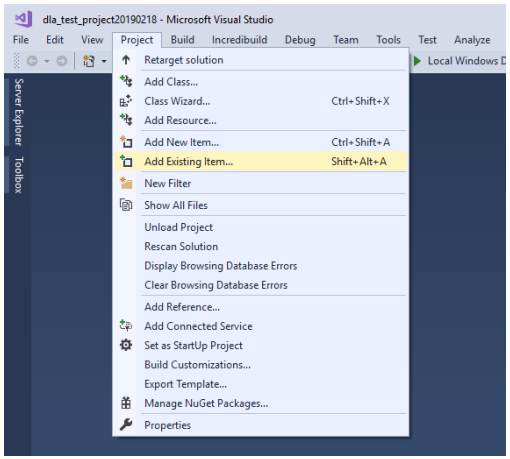
  Project menu → Add Class

- Alternatively, use existing files

  Project menu → Add existing Item







DLA project: structure of codes

MAIN

- set-up DLA object (using global pointer *sys)
- create Window object (using pointer *win)
- user interface to interact with DLA object (handleKeypress)
- update the visualization

DLASystem class
- set-up grid
- create / remove particle object
- move particle
- data for visualization
- (write data in a file)

Particle class:
- size
- position

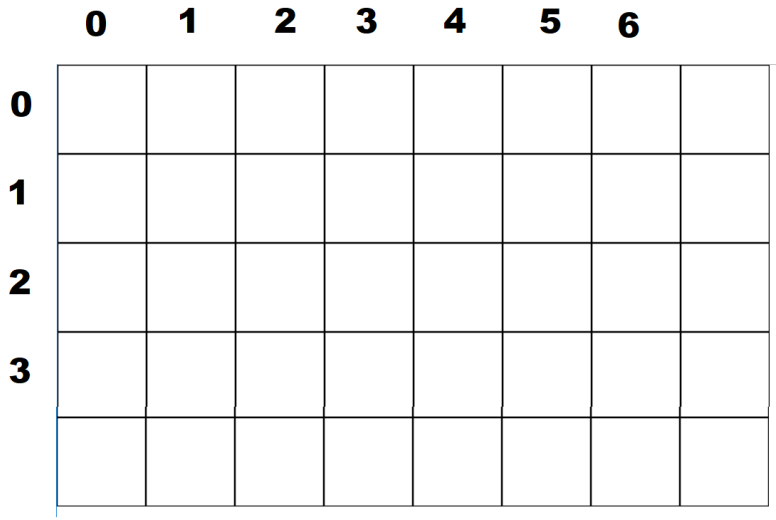
rnd class:
Random number generator for moving the particle

Window class:
- size
- position
- visualization (drawSquares)

Grid variable (in DLASystem class)

In the simulation, the particle moves randomly on a grid. This grid is discrete and in practise it is a matrix (an array of arrays).

Therefore, the position of the particle is characterized by two integers i.e. the row and column of the matrix



However, you might have noticed during debugging that Visual Studio complains about loss of precision while converting `double` to `int`. This is because doubles are used to create random coordinates for the new particles and these coordinates are then rounded to integers later.

Particle class

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

Included in the program only once (`include "Particle.h"`)

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

Syntax of a class:
- `class`, name, and the code within `{ }`
- `public:` in the beginning (otherwise, everything is `private` by default)
- class declaration ends with `};`

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

`static const int`:

- `static` means, in this case, that the variable is created only once and it is shared by all the objects of type `Particle`
- `const` means that the value cannot be changed

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

dynamic allocation of memory:

- first a pointer (in which the memory address of the variable will be stored) is created
- the memory is allocated by using the keyword `new`
- `delete` is used to free the allocated memory

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos; // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

constructor:

- constructors are functions that have the same name as the class
- constructors are used to initialize the objects
- here, there are two constructors: one which simply allocates the memory, and another which also sets coordinates

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial positior
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] p
20  };
```

```
void main() {
    Particle prt1;
}
```

This line uses the default constructor and simply allocates the memory.

This line uses the second constructor. It allocates the memory and sets initial values for the pos array

```
void main() {
  double pos_array[2] = { 1.0, 2.0 };
  Particle prt1(pos_array);
}
```

destructor has ~ in front and the same name as the class

- it basically destroys the object and frees the memory
- normally, you do not need to explicitly call the destructor; it is automatically (implicitly) called when you exit the program

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

DLASystem class (header file)

```
20
21  class DLASystem {
22  private:

27      // list of particles
28      vector<Particle*> particleList;
29      int numParticles;

34      // size of cluster
35      double clusterRadius;
36      // these are related to the DLA algorithm
37      double addCircle;
38      double killCircle;

63  public:
64      // these are public variables and functions

82      // constructor
83      DLASystem(Window *set_win);
84      // destructor
85      ~DLASystem();

131     // add a particle at pos
132     void addParticle(double pos[]);
133     // add a particle at a random point on the addCircle
134     void addParticleOnAddCircle();
135

154 };
155
```

```
20
21  class DLASystem {
22  private:

27      // list of particles
28      vector<Particle*> particleList;
29      int numParticles;

34      // size of cluster
35      double clusterRadius;
36      // these are related to the DLA algorithm
37      double addCircle;
38      double killCircle;

63  public:
64      // these are public variables and functions

82      // constructor
83      DLASystem(Window *set_win);
84      // destructor
85      ~DLASystem();

131     // add a particle at pos
132     void addParticle(double pos[]);
133     // add a particle at a random point on the addCircle
134     void addParticleOnAddCircle();
135

154 };
155
```

`DLASystem` class has both `private` and `public` variables:

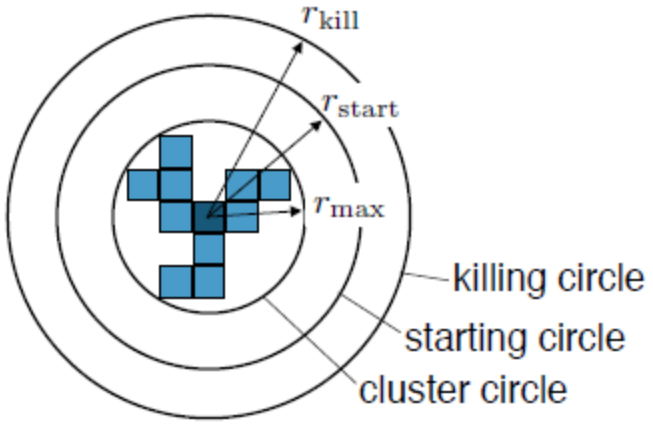
- `private` variables can be accessed only within the DLASystem.h and DLASystem.cpp files
- `public` variables can be accessed also from elsewhere, e.g. from the main

```
void main() {
    // from here !
}
```

```
20
21  class DLASystem {
22  private:

27      // list of particles
28      vector<Particle*> particleList;
29      int numParticles;

34      // size of cluster
35      double clusterRadius;
36      // these are related to the DLA algorithm
37      double addCircle;
38      double killCircle;

63  public:
64      // these are public variables and functions

82      // constructor
83      DLASystem(Window *set_win);
84      // destructor
85      ~DLASystem();

131     // add a particle at pos
132     void addParticle(double pos[]);
133     // add a particle at a random point on the addCircle
134     void addParticleOnAddCircle();
135

154 };
155
```

The DLA code involves three circles, all centred on the initial seed particle:

- r_{max} (`clusterRadius`) is the distance from the starting particle to the stationary particle that is furthest away from it
- When new particles are introduced into the system, they appear on the starting circle (`addCircle`)
- During random walk, if a particle reaches the killing circle (`killCircle`), it is removed and a new particle is created to replaces it



```
20
21  class DLASystem {
22  private:

27      // list of particles
28      vector<Particle*> particleList;
29      int numParticles;

34      // size of cluster
35      double clusterRadius;
36      // these are related to the DLA algorithm
37      double addCircle;
38      double killCircle;

63  public:
64      // these are public variables and functions

82      // constructor
83      DLASystem(Window *set_win);
84      // destructor
85      ~DLASystem();

131     // add a particle at pos
132     void addParticle(double pos[]);
133     // add a particle at a random point on the addCircle
134     void addParticleOnAddCircle();
135

154 };
155
```

Here is a `vector` object that is a container template i.e. it is an object that can contain other objects

- vectors are beneficial because their size can be changed during the program (see, next slides)
- (remember that normally you need to set fixed sizes to arrays etc.)

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
  5   #include "DLASystem.h"

 97   // add a particle to the system at a specific position
 98   void DLASystem::addParticle(double pos[]) {
 99       // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coordinates should be -gridSize/2 < x < gridSize/2
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the system at a random position on the addCircle
110   // if we hit an occupied site then we do nothing except print a message
111   // (this should never happen)
112   void DLASystem::addParticleOnAddCircle() {
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[0] << " " << pos[1] << endl;
121   }
```

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
 5    #include "DLASystem.h"

97    // add a particle to the system at a specific position
98    void DLASystem::addParticle(double pos[]) {
99        // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back mean
102       particleList.push
103       numParticles++;
104
105       // pos coordinate
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the system at a random position on the addCircle
110   // if we hit an occupied site then we do nothing except print a message
111   // (this should never happen)
112   void DLASystem::addParticleOnAddCircle() {
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[0] << " " << pos[1] << endl;
121   }
```

dynamic allocation of memory using `Particle(double set_pos[]);` constructor

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
 5    #include "DLASystem.h"

97    // add a particle to the system at a specific position
98    void DLASystem::addParticle(double pos[]) {
99        // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coordinates shoul
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the sys
110   // if we hit an occupied sit
111   // (this should never happen
112   void DLASystem::addParticleO
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[0] << " " << pos[1] << endl;
121   }
```

Here, we use `push_back` property of vectors to add one `Particle` type of pointer in the end of `particleList` vector and increase its size by one

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
 5    #include "DLASystem.h"

97    // add a particle to the system at a specific position
98    void DLASystem::addParticle(double pos[]) {
99        // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coo
106       setGrid(po
107   }
108
109   // add a parti
110   // if we hit a                                    ge
111   // (this shoul
112   void DLASystem::addParticleOnAddCircle() {
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[0] << " " << pos[1] << endl;
121   }
```

numParticles++;
means the same as
numParticles = numParticles + 1;

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
  5   #include "DLASystem.h"

 97   // add a particle to the system at a specific position
 98   void DLASystem::addParticle(double pos[]) {
 99       // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coordinates should be -gridSize/2 < x < gridSize/2
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the s
110   // if we hit an occupied s
111   // (this should never happ
112   void DLASystem::addParticl
113       double pos[2];
114       double theta = rgen.ra
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[0] << " " << pos[1] << endl;
121   }
```

function that sets that the position `pos` is now occupied (= 1) in the grid

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
 5    #include "DLASystem.h"

97    // add a particle to the system at a specific position
98    void DLASystem::addParticle(double pos[]) {
99        // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coordinates should be -gridSize/2 < x < gridSize/2
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the system at a random position on the addCircle
110   // if we hit an occupied site then we do nothing except print a message
111   // (this should never happen)
112   void DLASystem::addParticleOnAddCircle() {
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle * sin(theta));
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL "
121   }
```

random coordinates

`floor` command rounds down the decimal number to the closest integer (why is this necessary?)

DLASystem class (cpp file)

```
class DLASystem {
private:

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

    // size of cluster
    double clusterRadius;
    // these are related to the DLA algorithm
    double addCircle;
    double killCircle;

public:
    // these are public variables and functions

    // constructor
    DLASystem(Window *set_win);
    // destructor
    ~DLASystem();

    // add a particle at pos
    void addParticle(double pos[]);
    // add a particle at a random point on the addCircle
    void addParticleOnAddCircle();

};
```

```
 5    #include "DLASystem.h"

97    // add a particle to the system at a specific position
98    void DLASystem::addParticle(double pos[]) {
99        // create a new particle
100       Particle * p = new Particle(pos);
101       // push_back means "add this to the end of the list"
102       particleList.push_back(p);
103       numParticles++;
104
105       // pos coordinates should be -gridSize/2 < x < gridSize/2
106       setGrid(pos, 1);
107   }
108
109   // add a particle to the system at a random position on the addCircle
110   // if we hit an occupied site then we do nothing except print a message
111   // (this should never happen)
112   void DLASystem::addParticleOnAddCircle() {
113       double pos[2];
114       double theta = rgen.random01() * 2 * M_PI;
115       pos[0] = floor(addCircle * cos(theta));
116       pos[1] = floor(addCircle *
117       if (readGrid(pos) == 0)
118           addParticle(pos);
119       else
120           cout << "FAIL " << pos[
121   }
```

if this position in the grid is not occupied, then execute `addParticle(pos);`

Conclusion

C++ has a lot of “standard” classes available that can be helpful, e.g.

```
vector, string, ofstream, ifstream, stringstream
```

You need to find the right way to use these classes, and the right header files to `#include`…

C++ is not the easiest programming language but it is general, flexible, powerful, and can be used to write efficient (fast) programs

… if you wish to learn more there is lots of documentation online…

https://www.cprogramming.com/tutorial/c++-tutorial.html
http://www.cplusplus.com/doc/tutorial/
https://isocpp.org/faq

Remember: the main aim of this course is physics and studying physical phenomena through computational models (not programming) !

- you will probably spend more time on analysing the simulation results and writing the reports than C++ coding
- correct physical interpretation of simulation results is important (not fancy C++ coding)

PH30056: Computational Physics B: Lecture 4

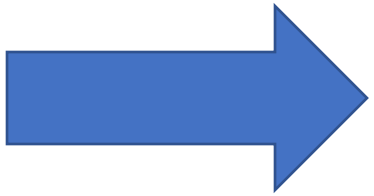
Semester 2, Academic year 2024/2025

Lectured by David Tsang (dcwt21@bath.ac.uk)

Co-lecturers: Prof Alain Nogaret, Prof Alison Walker

Lecture 4: How to write a report

10101101010100010111001110011010101010101010101010101010101010010100000011111001101010101010101010101



(study.com)

Data

Story

David Tsang D.Tsang@bath.ac.uk

Report marking scheme

See Moodle for details

Presentation (25 marks): scientific writing style, correct terminology, clear descriptions and explanations, logical structure, well designed and chosen figures and tables

Background and context (10 marks): physical background, clear objectives, connection to real-life applications

Achievements (35 marks): quality and quantity of results, clear presentation of results, creativity

Analysis and scientific understanding (30 marks): analysis of the results, comparisons to other works, errors and uncertainties, discussion on the characteristics of the techniques

Outline

1. Strategy and tactics for scientific writing

2. How to analyse and present data

Lecture 4: How to write a report

One or two sentences providing a **basic introduction** to the field, comprehensible to a scientist in any discipline.

Two to three sentences of **more detailed background**, comprehensible to scientists in related disciplines.

One sentence clearly stating the **general problem** being addressed by this particular study.

One sentence summarizing the main result (with the words "**here we show**" or their equivalent).

Two or three sentences explaining what the **main result** reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more **general context**.

Two or three sentences to provide a **broader perspective**, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion. Under these circumstances, the length of the paragraph can be up to 300 words. (This example is 190 words without the final section, and 250 words with it).

During cell division, mitotic spindles are assembled by microtubule-based motor proteins[1,2]. The bipolar organization of spindles is essential for proper segregation of chromosomes, and requires plus-end-directed homotetrameric motor proteins of the widely conserved kinesin-5 (BimC) family[3]. Hypotheses for bipolar spindle formation include the 'push–pull mitotic muscle' model, in which kinesin-5 and opposing motor proteins act between overlapping microtubules[2,4,5]. However, the precise roles of kinesin-5 during this process are unknown. Here we show that the vertebrate kinesin-5 Eg5 drives the sliding of microtubules depending on their relative orientation. We found in controlled *in vitro* assays that Eg5 has the remarkable capability of simultaneously moving at ~20 nm s^{-1} towards the plus-ends of each of the two microtubules it crosslinks. For anti-parallel microtubules, this results in relative sliding at ~40 nm s^{-1}, comparable to spindle pole separation rates *in vivo*[6]. Furthermore, we found that Eg5 can tether microtubule plus-ends, suggesting an additional microtubule-binding mode for Eg5. Our results demonstrate how members of the kinesin-5 family are likely to function in mitosis, pushing apart interpolar microtubules as well as recruiting microtubules into bundles that are subsequently polarized by relative sliding. We anticipate our assay to be a starting point for more sophisticated *in vitro* models of mitotic spindles. For example, the individual and combined action of multiple mitotic motors could be tested, including minus-end-directed motors opposing Eg5 motility. Furthermore, Eg5 inhibition is a major target of anti-cancer drug development, and a well-defined and quantitative assay for motor function will be relevant for such developments.

(study.com)

Story

nature.com

One or two sentences providing a **basic introduction** to the field, comprehensible to a scientist in any discipline.

Two to three sentences of **more detailed background**, comprehensible to scientists in related disciplines.

One sentence clearly stating the **general problem** being addressed by this particular study.

One sentence summarizing the main result (with the words "**here we show**" or their equivalent).

Two or three sentences explaining what the **main result** reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more **general context**.

Two or three sentences to provide a **broader perspective**, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion. Under these circumstances, the length of the paragraph can be up to 300 words. (This example is 190 words without the final section, and 250 words with it).

During cell division, mitotic spindles are assembled by microtubule-based motor proteins[1,2]. The bipolar organization of spindles is essential for proper segregation of chromosomes, and requires plus-end-directed homotetrameric motor proteins of the widely conserved kinesin-5 (BimC) family[3]. Hypotheses for bipolar spindle formation include the 'push–pull mitotic muscle' model, in which kinesin-5 and opposing motor proteins act between overlapping microtubules[2,4,5]. However, the precise roles of kinesin-5 during this process are unknown. Here we show that the vertebrate kinesin-5 Eg5 drives the sliding of microtubules depending on their relative orientation. We found in controlled *in vitro* assays that Eg5 has the remarkable capability of simultaneously moving at ~20 nm s^{-1} towards the plus-ends of each of the two microtubules it crosslinks. For anti-parallel microtubules, this results in relative sliding at ~40 nm s^{-1}, comparable to spindle pole separation rates *in vivo*[6]. Furthermore, we found that Eg5 can tether microtubule plus-ends, suggesting an additional microtubule-binding mode for Eg5. Our results demonstrate how members of the kinesin-5 family are likely to function in mitosis, pushing apart interpolar microtubules as well as recruiting microtubules into bundles that are subsequently polarized by relative sliding. We anticipate our assay to be a starting point for more sophisticated *in vitro* models of mitotic spindles. For example, the individual and combined action of multiple mitotic motors could be tested, including minus-end-directed motors opposing Eg5 motility. Furthermore, Eg5 inhibition is a major target of anti-cancer drug development, and a well-defined and quantitative assay for motor function will be relevant for such developments.

One or two sentences providing a **basic introduction** to the field, comprehensible to a scientist in any discipline.

Two to three sentences of **more detailed background**, comprehensible to scientists in related disciplines.

One sentence clearly stating the **general problem** being addressed by this particular study.

One sentence summarizing the main result (with the words "**here we show**" or their equivalent).

Two or three sentences explaining what the **main result** reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

Two or three sentences explaining what the **main result** reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more **general context**.

Two or three sentences to provide a **broader perspective**, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion. Under these circumstances, the length of the paragraph can be up to 300 words. (This example is 190 words without the final section, and 250 words with it).

During cell division, mitotic spindles are assembled by microtubule-based motor proteins[1,2]. The bipolar organization of spindles is essential for proper segregation of chromosomes, and requires plus-end-directed homotetrameric motor proteins of the widely conserved kinesin-5 (BimC) family[3]. Hypotheses for bipolar spindle formation include the 'push–pull mitotic muscle' model, in which kinesin-5 and opposing motor proteins act between overlapping microtubules[2,4,5]. However, the precise roles of kinesin-5 during this process are unknown. Here we show that the vertebrate kinesin-5 Eg5 drives the sliding of microtubules depending on their relative orientation. We found in controlled *in vitro* assays that Eg5 has the remarkable capability of simultaneously moving at ~20 nm s^{-1} towards the plus-ends of each of the two microtubules it crosslinks. For anti-parallel microtubules, this results in relative sliding at ~40 nm s^{-1}, comparable to spindle pole separation rates *in vivo*[6]. Furthermore, we found

capability of simultaneously moving at ~20 nm s^{-1} towards the plus-ends of each of the two microtubules it crosslinks. For anti-parallel microtubules, this results in relative sliding at ~40 nm s^{-1}, comparable to spindle pole separation rates *in vivo*[6]. Furthermore, we found that Eg5 can tether microtubule plus-ends, suggesting an additional microtubule-binding mode for Eg5. Our results demonstrate how members of the kinesin-5 family are likely to function in mitosis, pushing apart interpolar microtubules as well as recruiting microtubules into bundles that are subsequently polarized by relative sliding. We anticipate our assay to be a starting point for more sophisticated *in vitro* models of mitotic spindles. For example, the individual and combined action of multiple mitotic motors could be tested, including minus-end-directed motors opposing Eg5 motility. Furthermore, Eg5 inhibition is a major target of anti-cancer drug development, and a well-defined and quantitative assay for motor function will be relevant for such developments.

The main story arc

This story arc repeats in:

- The abstract
- The introduction
- The report as a whole

(study.com)

Story arcs throughout the report

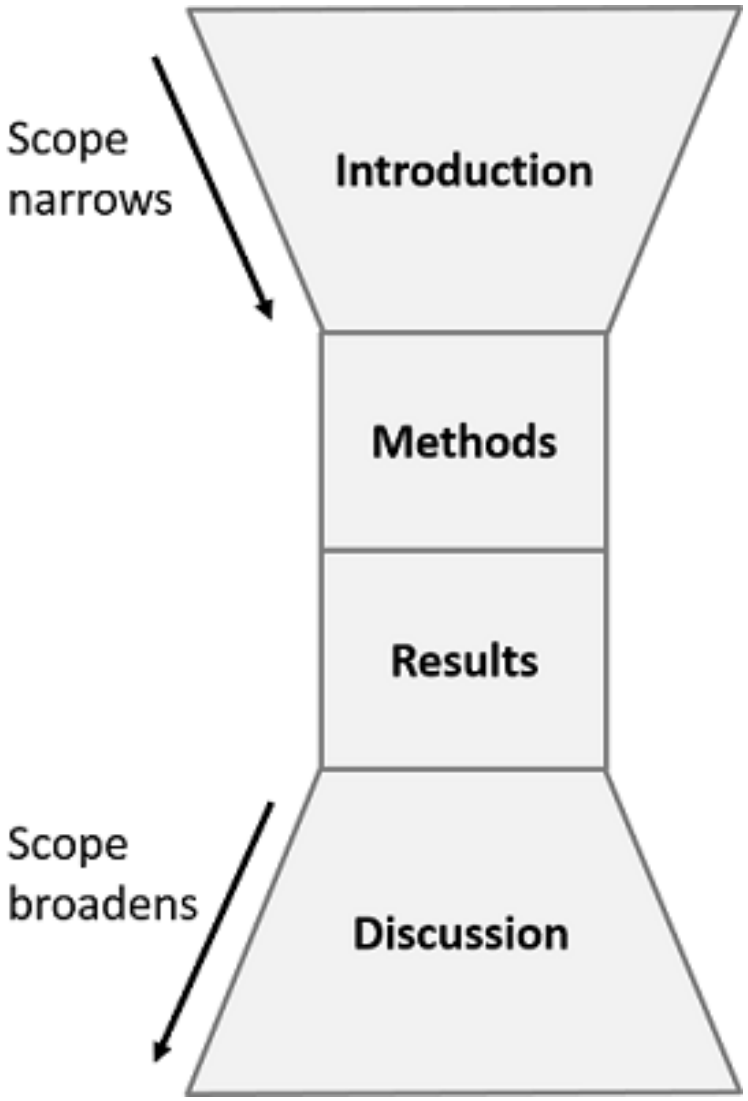
This story arc repeats in:

- The abstract
- The introduction
- The report as a whole

(study.com)

A story arc occurs in every paragraph

Funnel



miamioh.edu

(study.com)

Story

One or two sentences providing a **basic introduction** to the field, comprehensible to a scientist in any discipline.

Two to three sentences of **more detailed background**, comprehensible to scientists in related disciplines.

One sentence clearly stating the **general problem** being addressed by this particular study.

One sentence summarizing the main result (with the words "**here we show**" or their equivalent).

Two or three sentences explaining what the **main result** reveals in direct comparison to what was thought to be the case previously, or how the main result adds to previous knowledge.

One or two sentences to put the results into a more **general context**.

Two or three sentences to provide a **broader perspective**, readily comprehensible to a scientist in any discipline, may be included in the first paragraph if the editor considers that the accessibility of the paper is significantly enhanced by their inclusion. Under these circumstances, the length of the paragraph can be up to 300 words. (This example is 190 words without the final section, and 250 words with it).

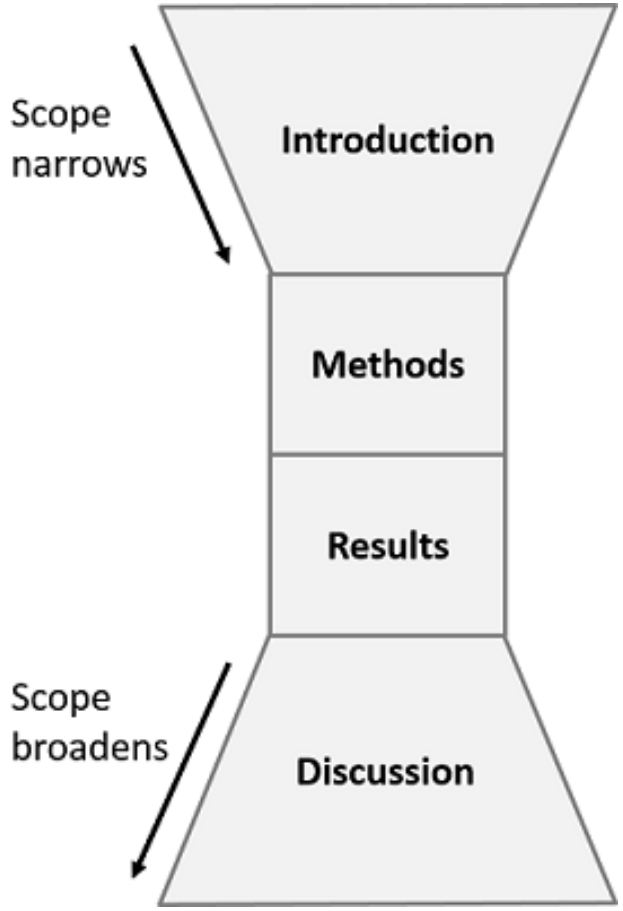
During cell division, mitotic spindles are assembled by microtubule-based motor proteins[1,2]. The bipolar organization of spindles is essential for proper segregation of chromosomes, and requires plus-end-directed homotetrameric motor proteins of the widely conserved kinesin-5 (BimC) family[3]. Hypotheses for bipolar spindle formation include the 'push–pull mitotic muscle' model, in which kinesin-5 and opposing motor proteins act between overlapping microtubules[2,4,5]. However, the precise roles of kinesin-5 during this process are unknown. Here we show that the vertebrate kinesin-5 Eg5 drives the sliding of microtubules depending on their relative orientation. We found in controlled *in vitro* assays that Eg5 has the remarkable capability of simultaneously moving at ~20 nm s^{-1} towards the plus-ends of each of the two microtubules it crosslinks. For anti-parallel microtubules, this results in relative sliding at ~40 nm s^{-1}, comparable to spindle pole separation rates *in vivo*[6]. Furthermore, we found that Eg5 can tether microtubule plus-ends, suggesting an additional microtubule-binding mode for Eg5. Our results demonstrate how members of the kinesin-5 family are likely to function in mitosis, pushing apart interpolar microtubules as well as recruiting microtubules into bundles that are subsequently polarized by relative sliding. We anticipate our assay to be a starting point for more sophisticated *in vitro* models of mitotic spindles. For example, the individual and combined action of multiple mitotic motors could be tested, including minus-end-directed motors opposing Eg5 motility. Furthermore, Eg5 inhibition is a major target of anti-cancer drug development, and a well-defined and quantitative assay for motor function will be relevant for such developments.

Structure of a report

A standard structure of a report:

1. Title
2. Abstract
3. Introduction
4. Methods
5. **Results**
6. **Discussion**
7. Conclusion
8. Bibliography
9. (Possibly Acknowledgements and Appendices)

(You are not forced to use this structure)



miamioh.edu

5. Results

What are the main findings?

Good and clear visualisation is very important:
If the reader has a brief look at the figures, she/he should get a quick idea what you have done

Figure guidelines

Label your axis and what you are presenting

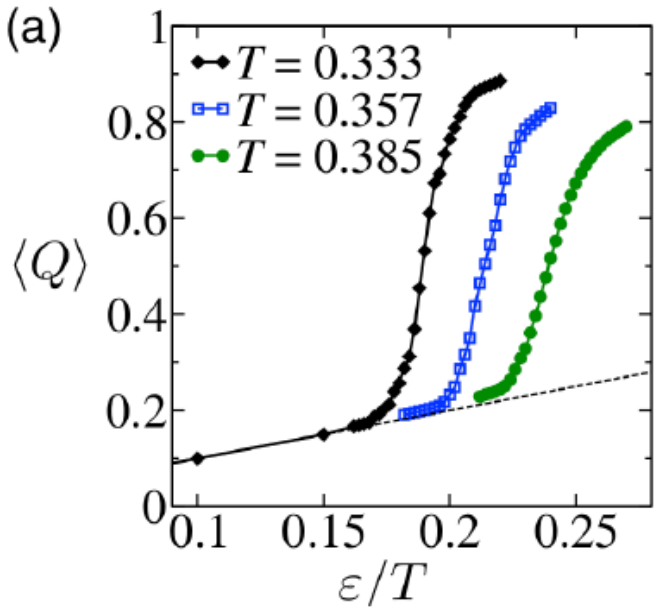


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Label your axis and what you are presenting

Show your data with points

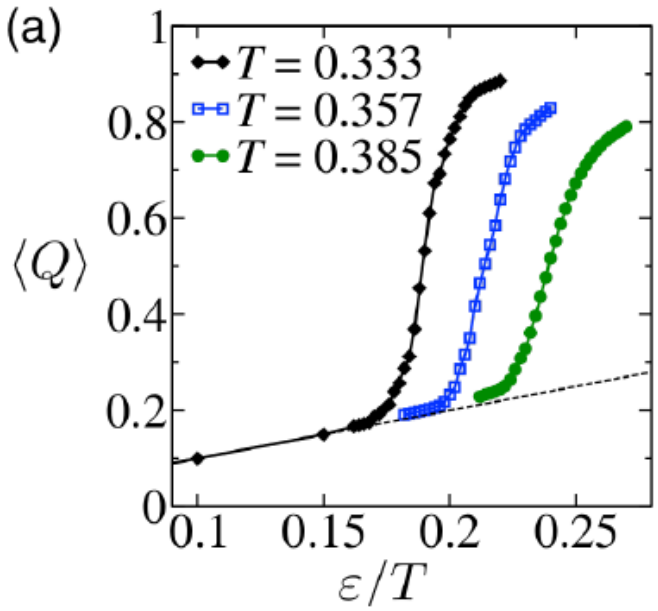


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Label your axis and what you are presenting

Show your data with points

Include error bars (or state that they are "comparable to symbol sizes")

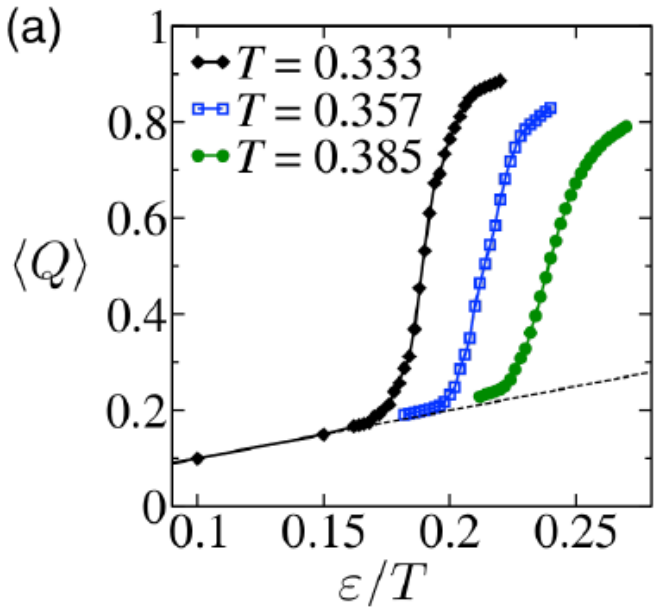


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Label your axis and what you are presenting

Show your data with points

Include error bars (or state that they are "comparable to symbol sizes")

Either connect adjacent points with straight line segments *or* do not connect them at all

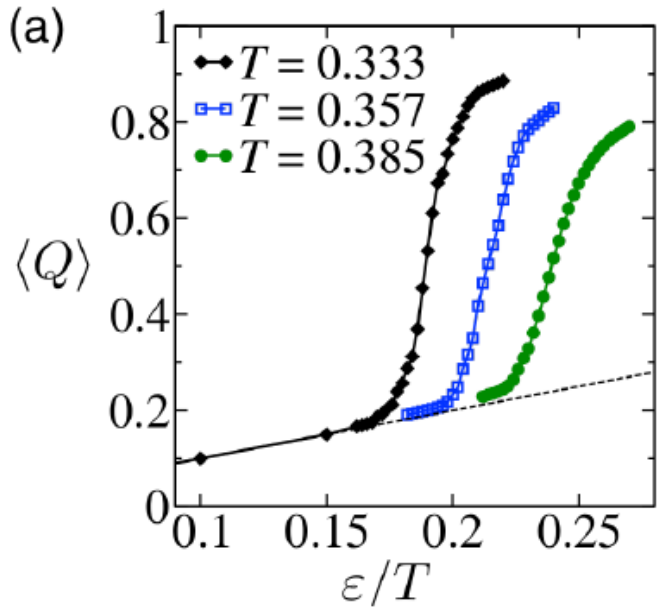


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Label your axis and what you are presenting

Show your data with points

Include error bars (or state that they are "comparable to symbol sizes")

Either connect adjacent points with straight line segments *or* do not connect them at all

If you are going to **fit** your data to a model then you should do this only to test a theory of hypothesis. You should justify the fitting function that you use.
(Do not draw "polynomial trend lines"!)

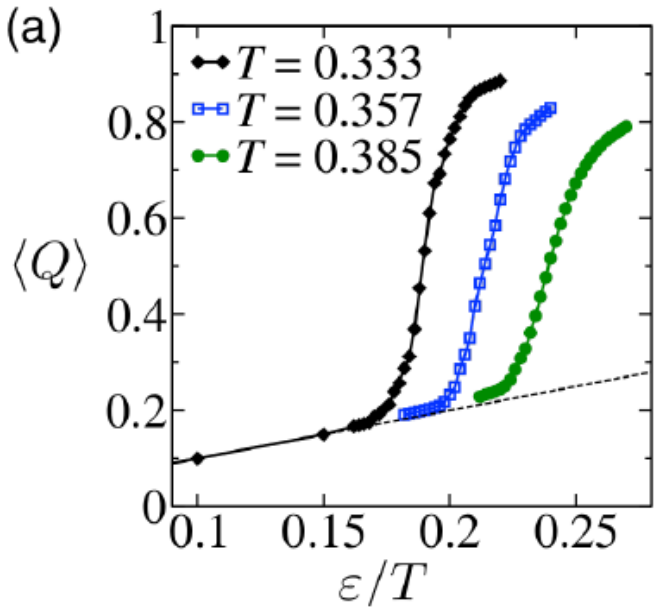


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Label your axis and what you are presenting

Show your data with points

Include error bars (or state that they are "comparable to symbol sizes")

Either connect adjacent points with straight line segments *or* do not connect them at all

If you are going to **fit** your the data to a model then you should do this only to test a theory of hypothesis. You should justify the fitting function that you use.
(Do not draw "polynomial trend lines"!)

To compare results with different parameters, it often helps to plot them all on the same graph

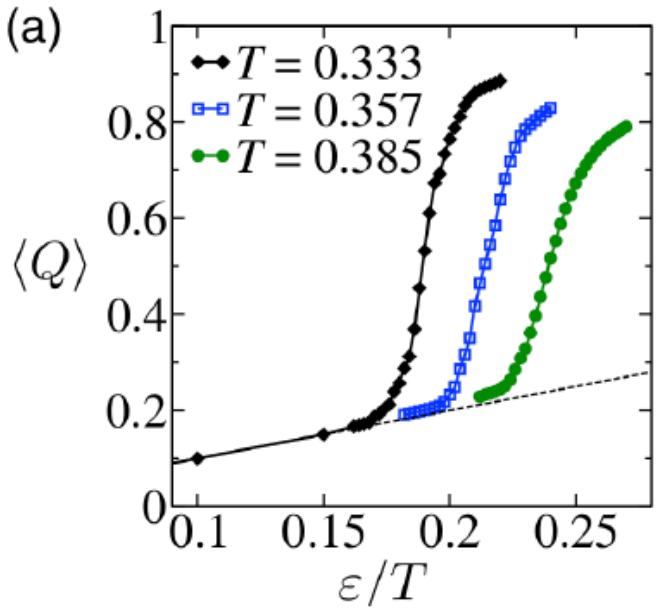


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Write a self-explanatory caption

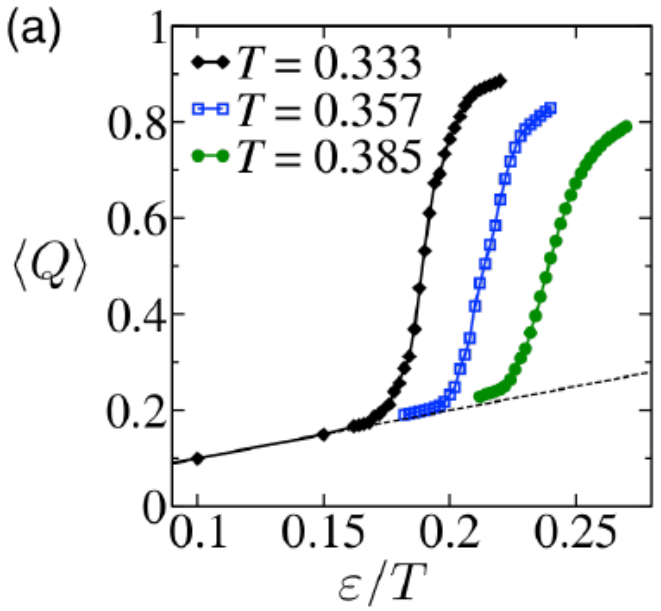


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Write a self-explanatory caption

Graphs should convey a clear message and be easy to understand

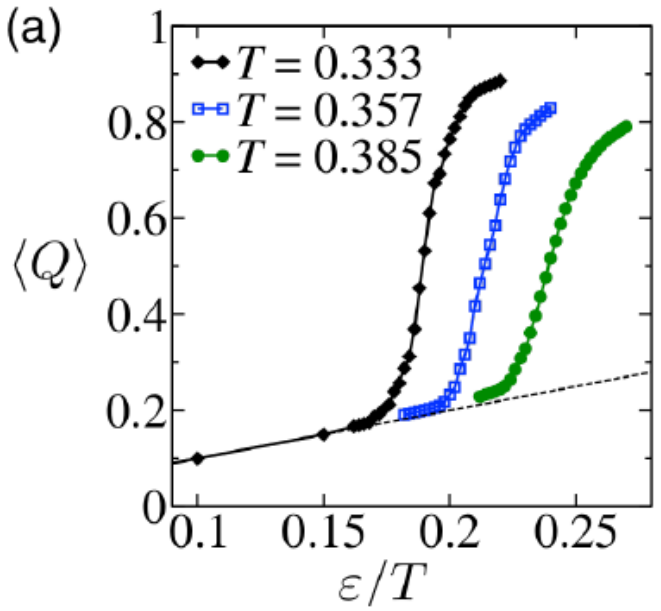


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Write a self-explanatory caption

Graphs should convey a clear message and be easy to understand

Font sizes in all labels should be comparable to the main text in the report

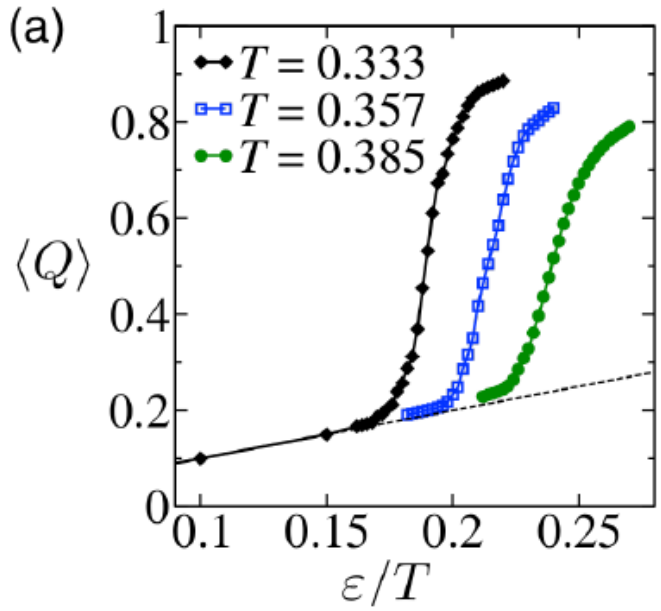


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

Figure guidelines

Write a self-explanatory caption

Graphs should convey a clear message and be easy to understand

Font sizes in all labels should be comparable to the main text in the report

If you copy a figure from a paper or website you need to write, for example, “Reproduced from [1]” in the caption, where [1] is the reference. (A simple citation would be enough in case you only got the idea from a paper but you made your own figure.)

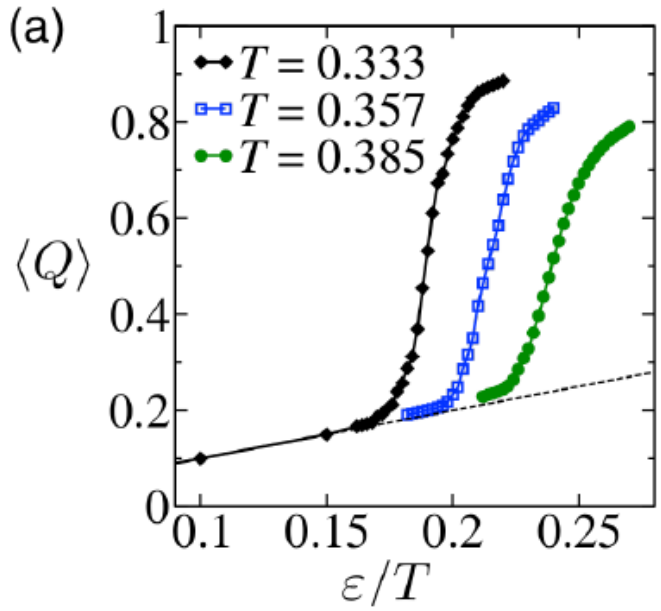


Fig 1a: Sample graph, reproduced from [Jack and Garrahan, Phys Rev Lett 116, 057702 (2016)]. Adjacent data points are connected by straight lines (these lines could also be omitted). The dashed line is a theoretical prediction that is valid for small ε. Error bars are similar to the symbol sizes so they have not been plotted in this case.

6. Discussion

What do the results mean (physical explanations)?

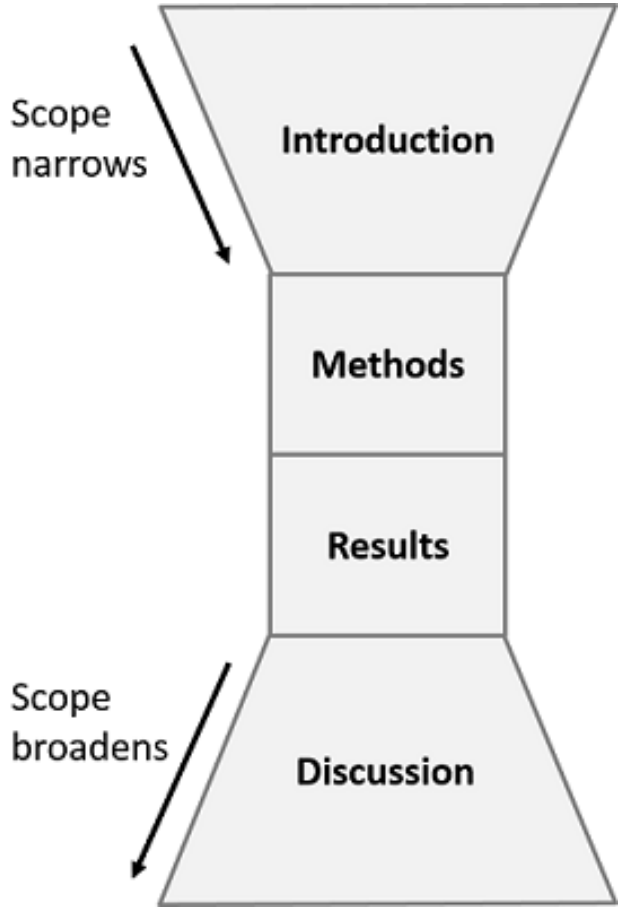
How good and reliable are the methods used?

Comparisons to other (earlier) works

Limitations of the analysis or the methods

Possible improvements in the future

Can be the most important section



miamioh.edu

1. Title

Title should be a short and catchy description of the work.
Where could the following titles appear?

“Fractals in nature”

“Investigating fractal properties of diffusion-limited aggregates in 2D”

“Fractal dimension of dielectric breakdown”

“Modelling and simulation of nanoparticle aggregation in colloidal systems”

1. Title

Title should be a short and catchy description of the work.
Where could the following titles appear?

“Fractals in nature”
(title in Cosmos magazine)

“Investigating fractal properties of diffusion-limited aggregates in 2D”
(title of a DLA course work report)

“Fractal dimension of dielectric breakdown”
(title of a research article in Physical Review Letters)

“Modelling and simulation of nanoparticle aggregation in colloidal systems”
(title of a Master’s thesis)

7. Conclusion and 3. Introduction

Conclusion:

- Highlight the most important findings (“the take-home message”)
- Overall significance of the work

Introduction:

- Background of the physics
- How the work is related to earlier works (brief literature review)
- How the work is related to real-life applications (significance)
- Motivation (why this study is needed) and objectives of the project (how the current know/how is improved)
- What was done in the project

8. Bibliography and 9. Appendices

Bibliography:

Remember to cite all the sources that you have used

Appendices:

Material that is relevant but it does not exactly fit in the main text

Material that makes the main text difficult to read (e.g., details of Methods)

For example, selected pieces of the used C++ codes, additional data, tables or figures

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

Try to be as **precise** as possible. If you read a sentence at random from the middle of the report, is it a true statement? If you are making an argument, does each sentence follow logically from the previous one?

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

Try to be as **precise** as possible. If you read a sentence at random from the middle of the report, is it a true statement? If you are making an argument, does each sentence follow logically from the previous one?

Be careful with **units** and dimensionless variables

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

Try to be as **precise** as possible. If you read a sentence at random from the middle of the report, is it a true statement? If you are making an argument, does each sentence follow logically from the previous one?

Be careful with **units** and dimensionless variables

Use **scientific language**, not informal terms like *wiggly*, *clumps, "this seems suspicious", "the results look correct"*

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

Try to be as **precise** as possible. If you read a sentence at random from the middle of the report, is it a true statement? If you are making an argument, does each sentence follow logically from the previous one?

Be careful with **units** and dimensionless variables

Use **scientific language**, not informal terms like *wiggly*, *clumps, "this seems suspicious", "the results look correct"*

Be careful with phrases like *this behaviour is not realistic.*
(Why? This usually makes sense only if you compare with a specific example).

General guidelines on reports

Writing a good report is not easy, and we are fairly demanding when we assess third-year units.

For background and context, **explain all technical terms** that you use.

Try to be as **precise** as possible. If you read a sentence at random from the middle of the report, is it a true statement? If you are making an argument, does each sentence follow logically from the previous one?

Be careful with **units** and dimensionless variables

Use **scientific language**, not informal terms like *wiggly*, *clumps, "this seems suspicious", "the results look correct"*

Be careful with phrases like *this behaviour is not realistic.*
(Why? This usually makes sense only if you compare with a specific example).

Maximum is 5 pages for first coursework (7 pages for 2nd)

Concentrate on the physics

(i) Describe and explain the behaviour of the model system you have been studying

(ii) Explain how your results are related to the properties of physical materials, or the outcomes of experiments

Concentrate on the physics

(i) Describe and explain the behaviour of the model system you have been studying

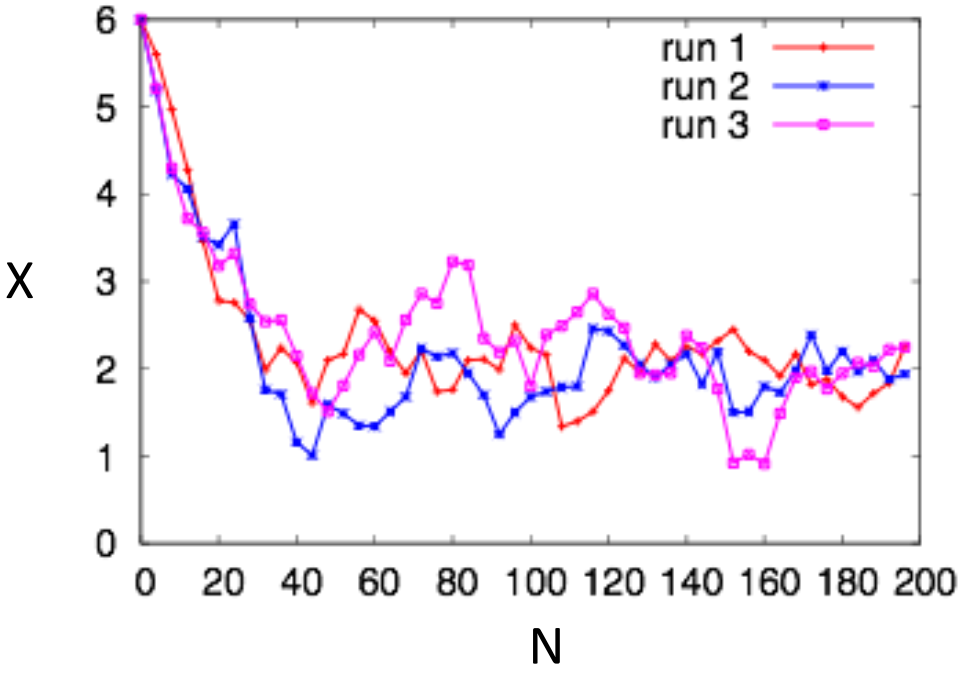
(ii) Explain how your results are related to the properties of physical materials, or the outcomes of experiments

Make sure you **explain what the model represents**.
For example, if you write about DLA, explain how your simulation of particles is related to any experimental examples that you mention.

Analysis of errors is important. If you have not estimated your errors then your conclusions are not justified. (This is especially true in simulations where random numbers are used).

Errors and uncertainties

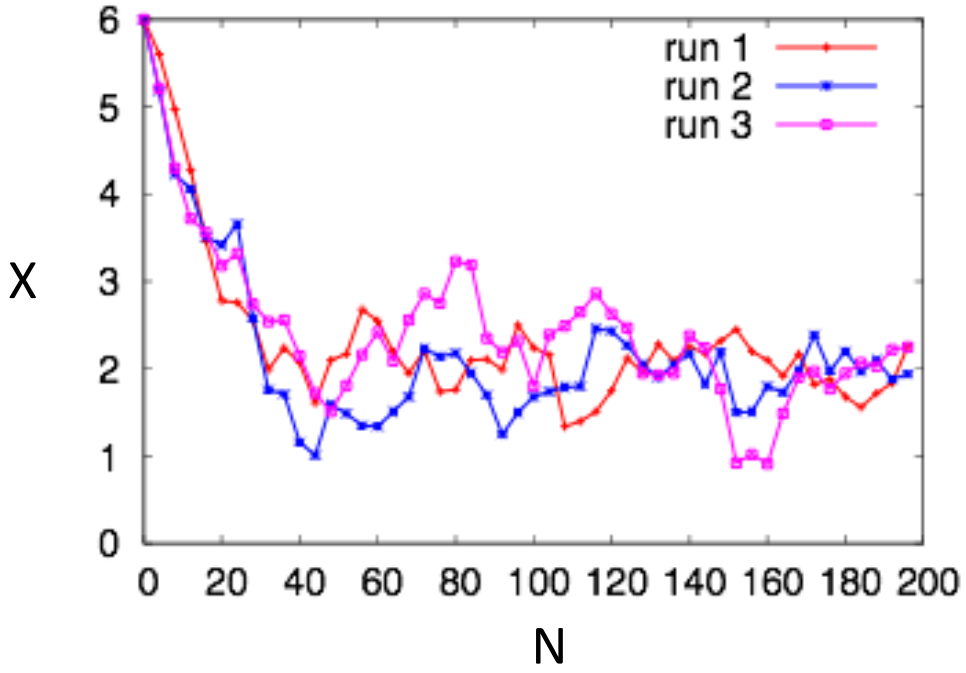
Illustration of error analysis with computational data



If you run your programs several times, you typically get different results.

Errors and uncertainties

Illustration of error analysis with computational data

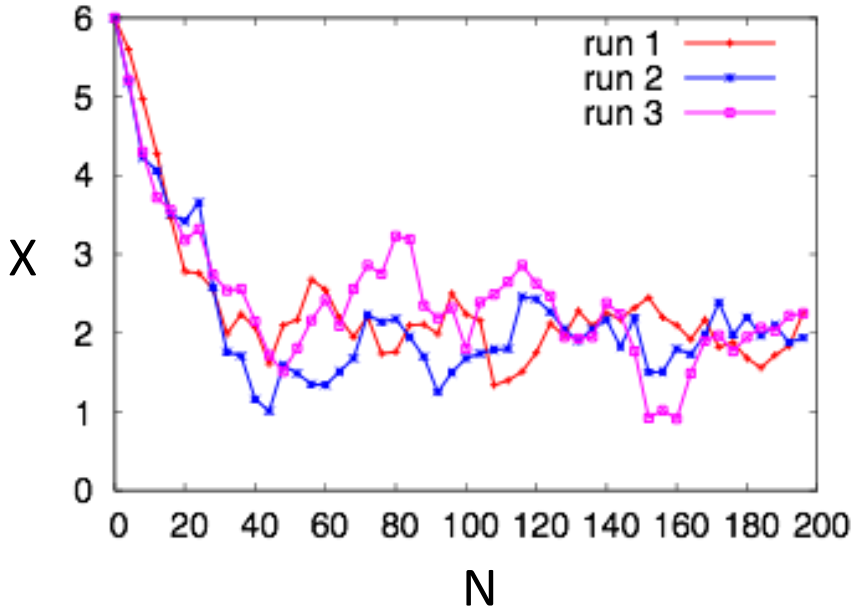


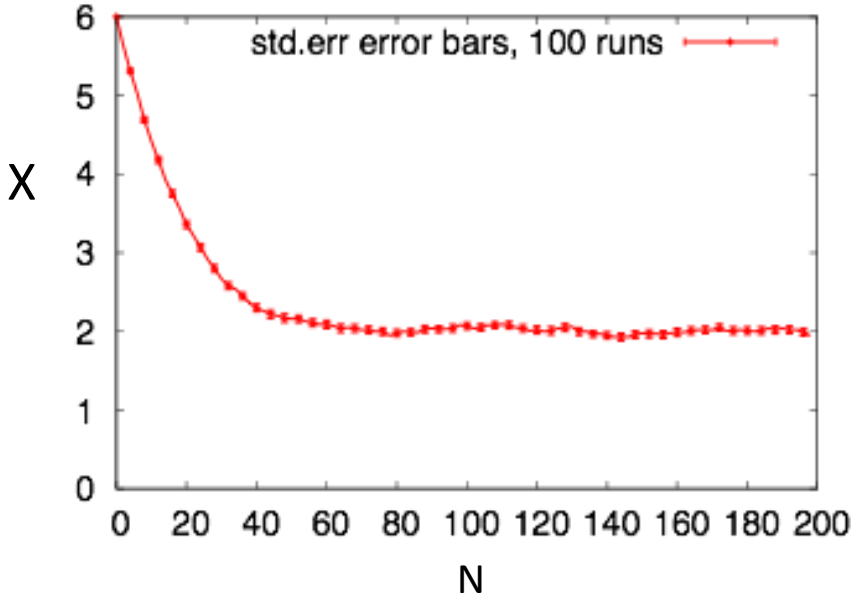
If you run your programs several times, you typically get different results.

How should we think about errors here? What graphs should we plot? What numbers should be calculate?

Do we want to estimate the mean behaviour? Uncertainty of this mean? Or to describe the range of variation?

Errors and uncertainties



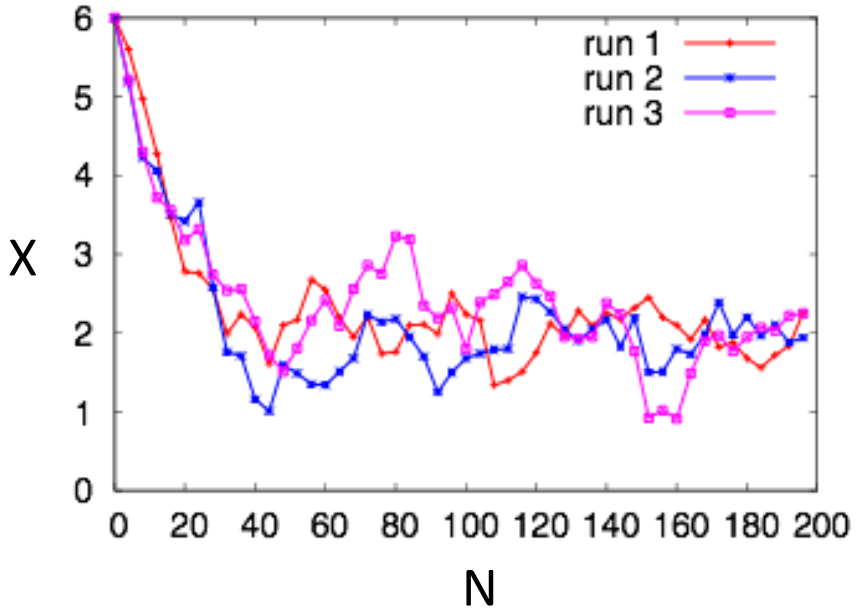


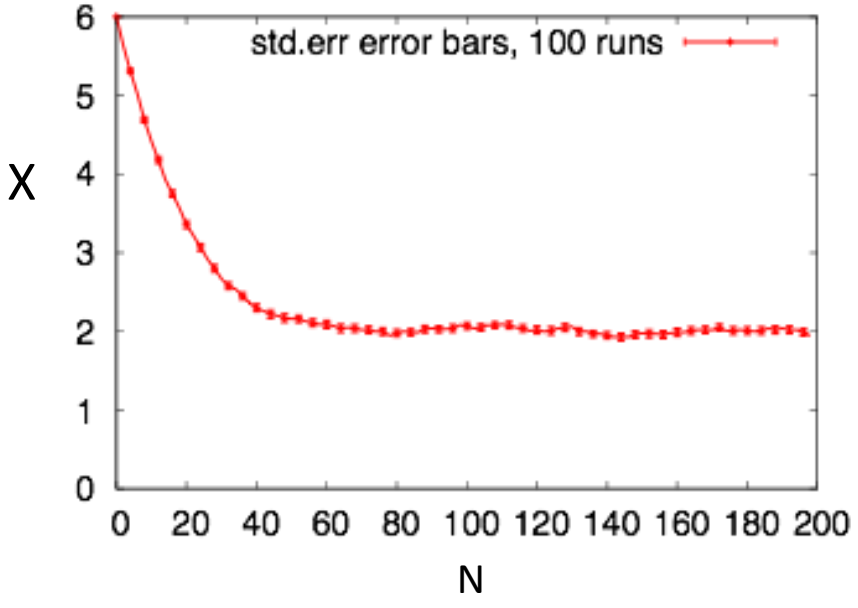
Average over runs $i = 1 \ldots n$, at fixed time:

$$\overline{X(t)} = \frac{1}{n} \sum_i X_i(t)$$

where $X_i(t)$ is the measurement of X in run i at time t.

Errors and uncertainties





Average over runs $i = 1 \ldots n$, at fixed time:

$$\overline{X(t)} = \frac{1}{n} \sum_i X_i(t)$$

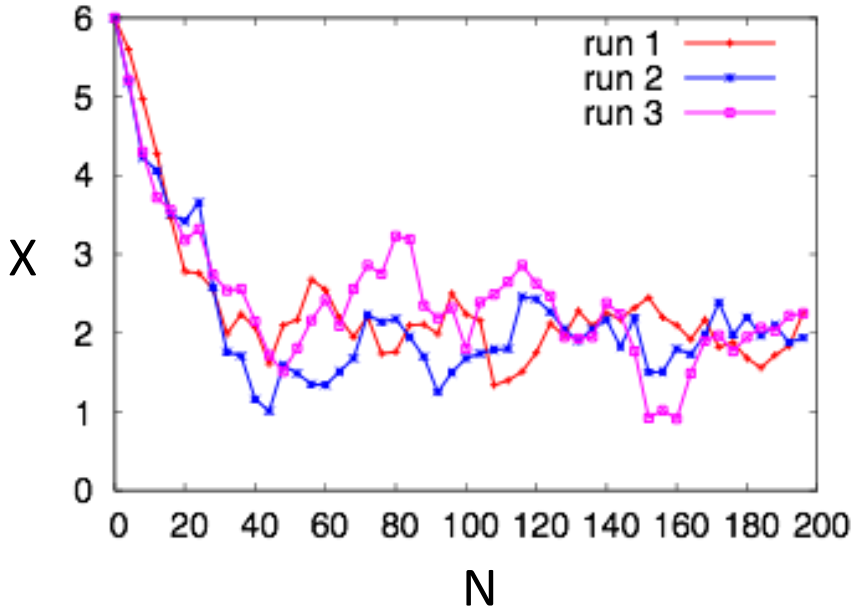
where $X_i(t)$ is the measurement of X in run i at time t.

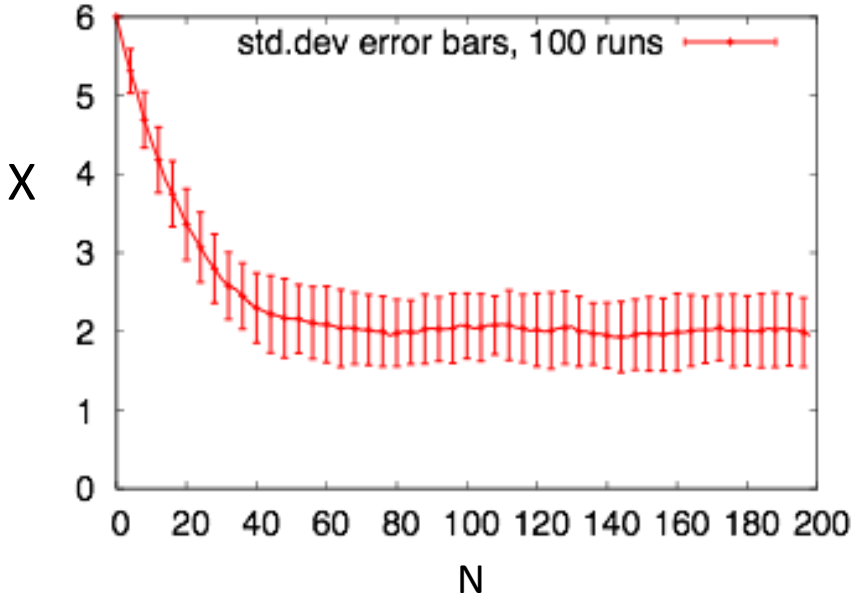
Standard error:

$$\sigma_{\bar{X}} = \frac{1}{\sqrt{n-1}} \left[\overline{X(t)^2} - \overline{X(t)}^2 \right]^{1/2}$$

Standard error is an estimate of how far your answer is from the "true" average value. The more simulations you run (*n* gets larger), the smaller the standard error gets.

Errors and uncertainties



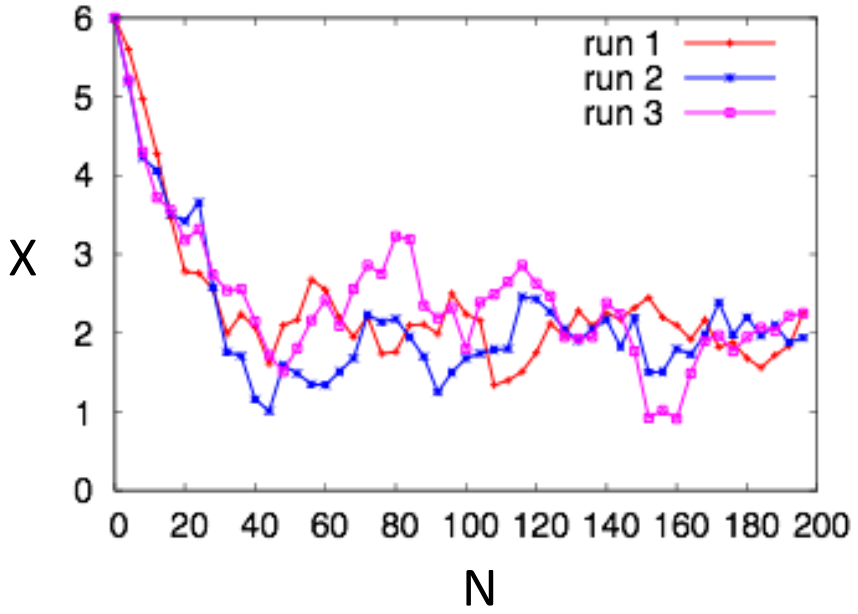


Average over runs $i = 1 \ldots n$, at fixed time:

$$\overline{X(t)} = \frac{1}{n} \sum_i X_i(t)$$

where $X_i(t)$ is the measurement of X in run i at time t.

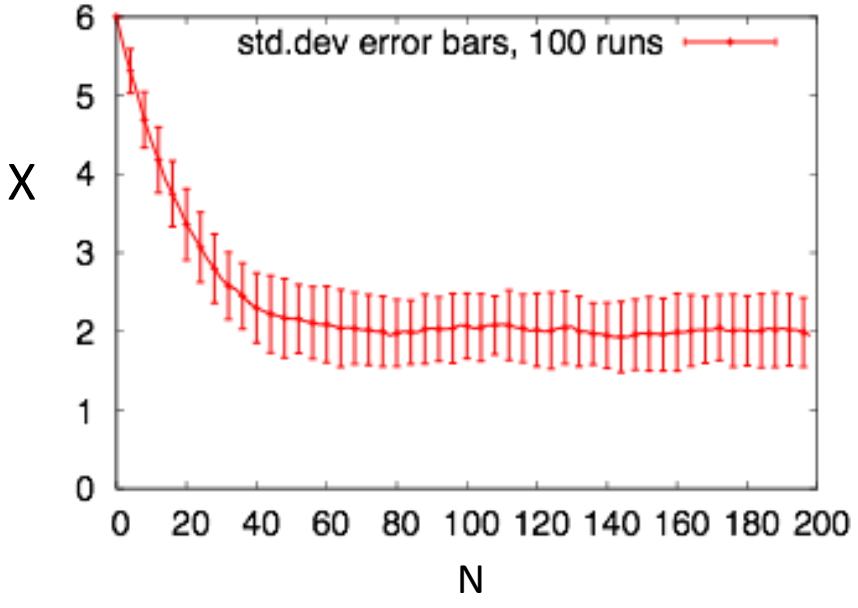
Errors and uncertainties



Average over runs $i = 1 \ldots n$, at fixed time:

$$\overline{X(t)} = \frac{1}{n} \sum_i X_i(t)$$

where $X_i(t)$ is the measurement of X in run i at time t.



Standard deviation:

$$\sigma_X = \left[\overline{X(t)^2} - \overline{X(t)}^2 \right]^{1/2}$$

Standard deviation tells you how much variation there is between different runs of the simulation. It does not get smaller with *n*.

Errors and uncertainties

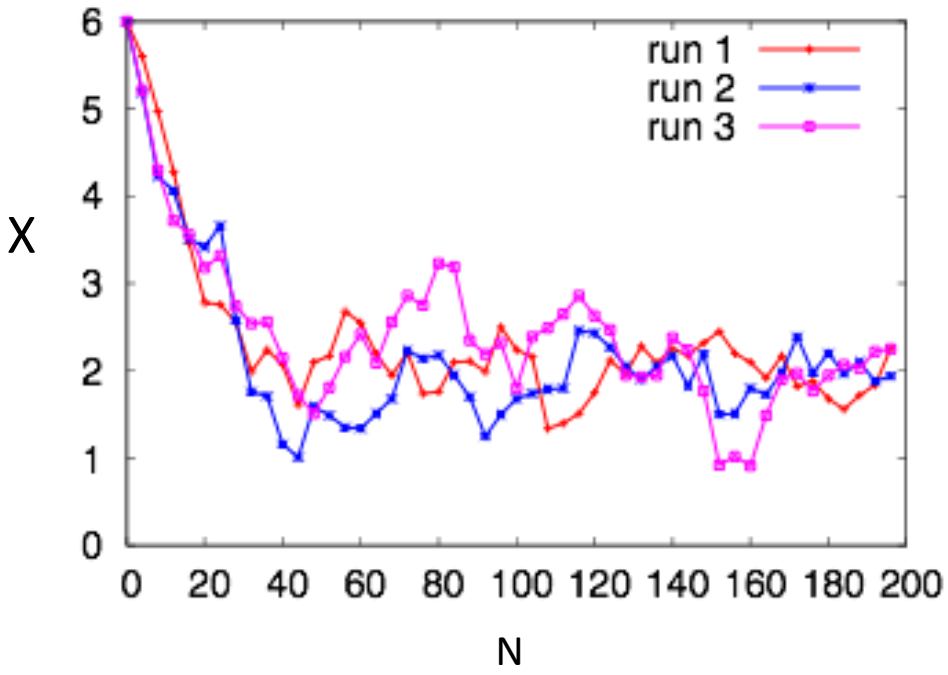
The *standard error* tells you how uncertain you are about the estimated average. This is the usual quantity to use for an error bar.

The *standard deviation* tells you how different are the repeated runs of the simulation or experiment. (This can also be used for an error bar in some cases, but you need to make it clear that it shows a *range of behaviour*, not an *uncertainty*.)

Note: the standard error formula assumes that all measurements are *independent*. This is true for independent runs of the same simulation, but this is not always the case.

Distributions and histograms

Sometimes we really care about the *range* of behaviour in our system

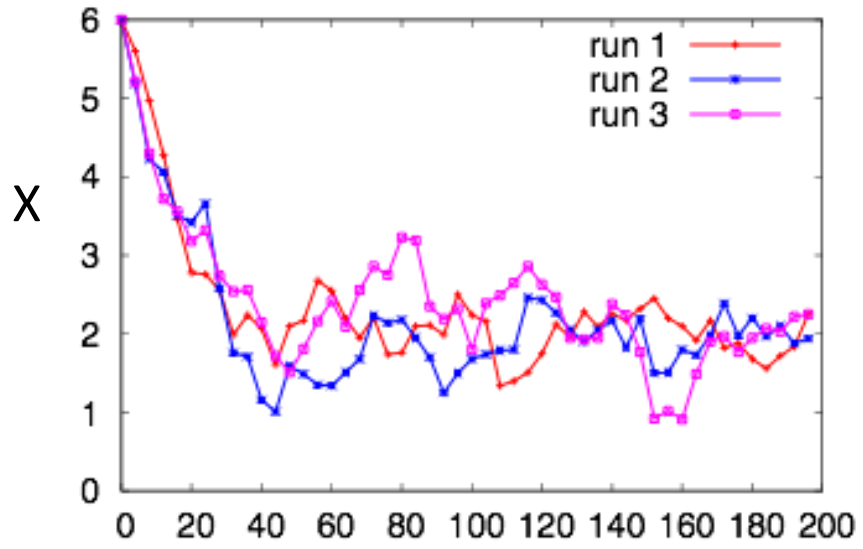


The standard deviation gives an idea of the spread of data

But can we analyse this in more detail? For example, can we say how often particular values appear?

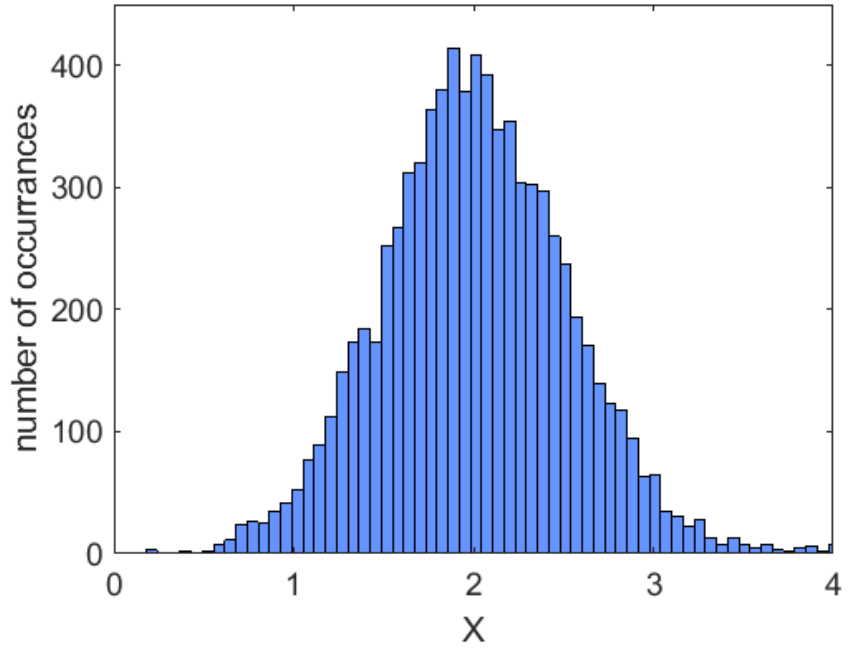
For this purpose, we can draw a **histogram**. Divide the range of values of *X* into n "bins" and count the number of times *X* falls within each bin.

Distributions and histograms



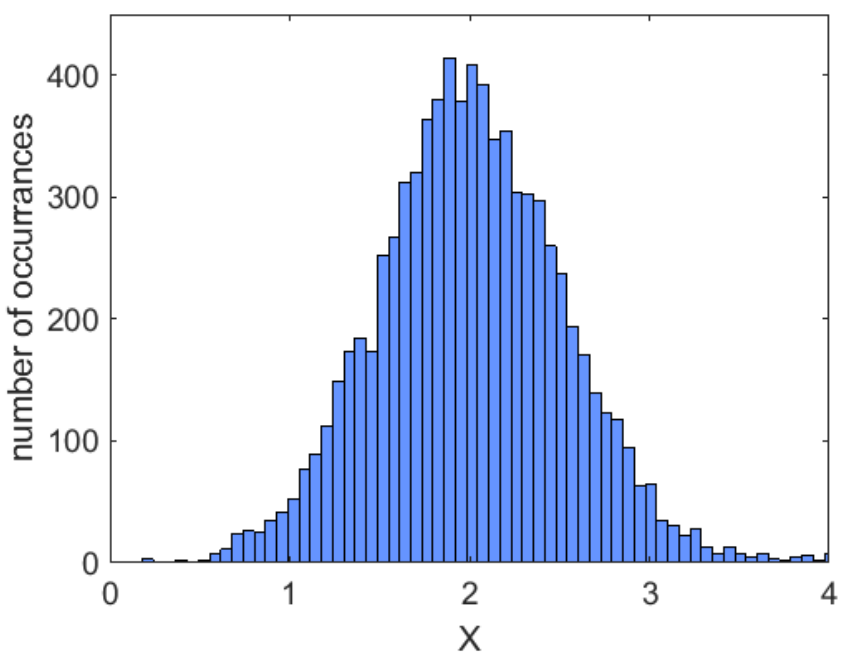
Let's take data from many runs…

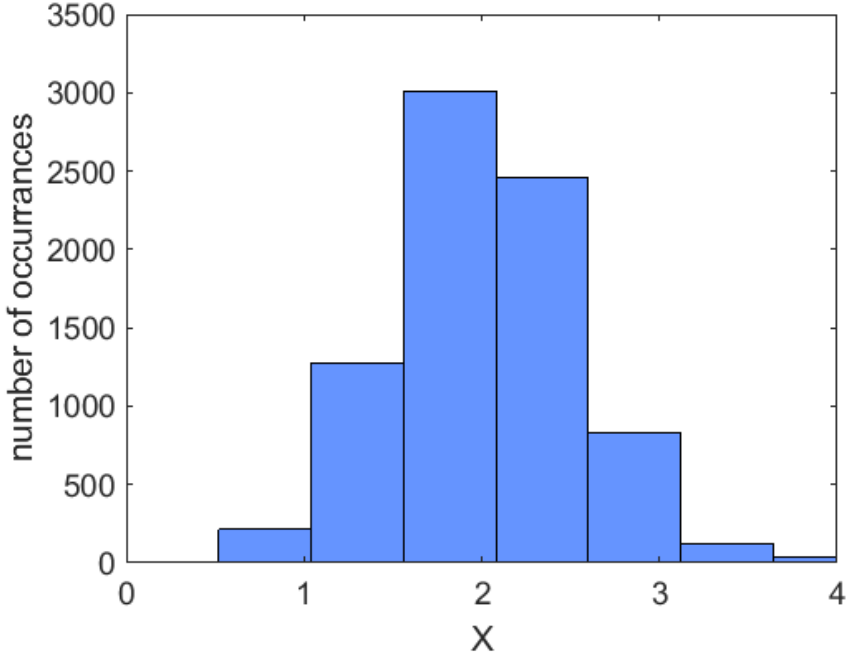
We divide the interval between 0 and 4 into 80 "bins" of width 0.05, and count (with some software) how many times *X* occurs within each "bin".



From the histogram, we can see that the most likely value of *X* is about 2, and the "spread" is about ± 0.5 (which is inline with the standard deviation)

Distributions and histograms





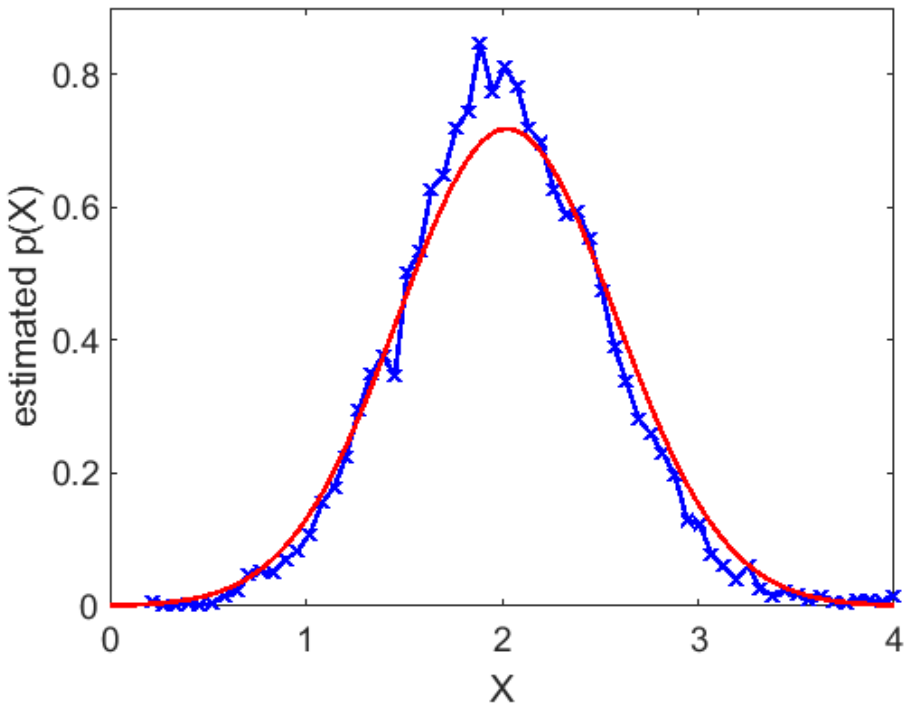
Note how the number of bins affects the visualization. The figure on the left has 80 bins and the figure on the right 10.

Distributions and histograms

A *probability density function* $p(X)$ is a positive function that tells me the fraction of observations are expected to fall in a given range.

If I make many measurements of X, I expect that the fraction falling between X_1 and X_2 should be

$$P(X_1 < X < X_2) = \int_{X_1}^{X_2} p(X)\mathrm{d}X$$



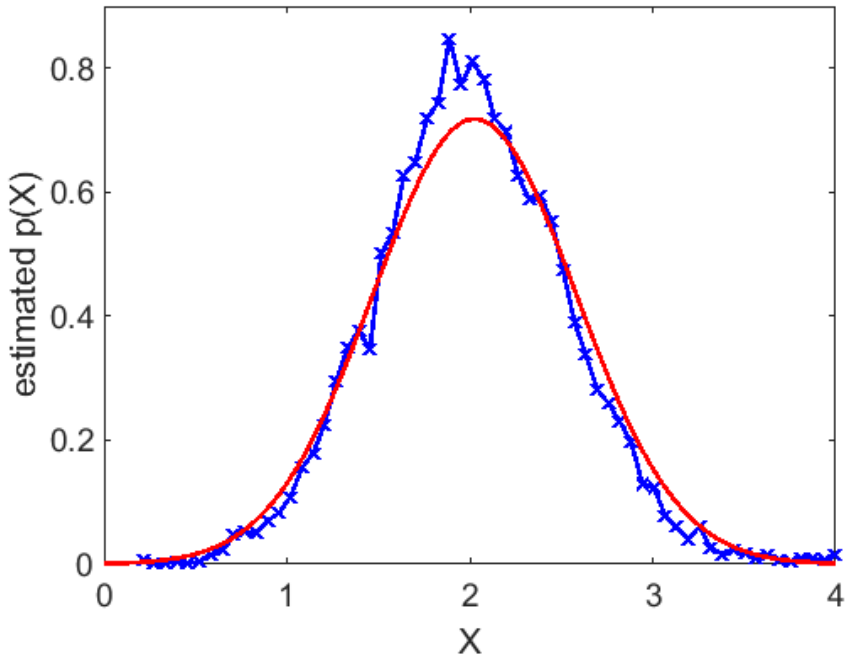
$p(X)$ can be estimated from a histogram of X by dividing the number of occurrences in each bin by the width of the bin, and then dividing by the total number of samples.

In this case $p(X)$ fits well to a Gaussian. . .

Distributions and histograms

Remember the fraction of measurements expected between X_1 and X_2 is

$$P(X_1 < X < X_2) = \int_{X_1}^{X_2} p(x)\mathrm{d}x$$

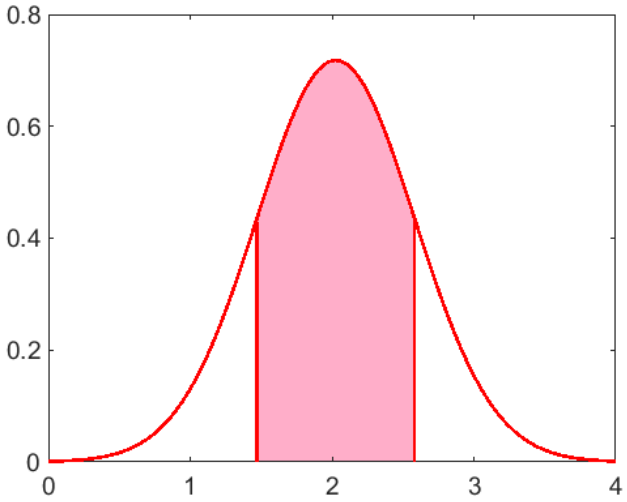


The fraction of measurements in any range corresponds to the *area under the curve*, within that range

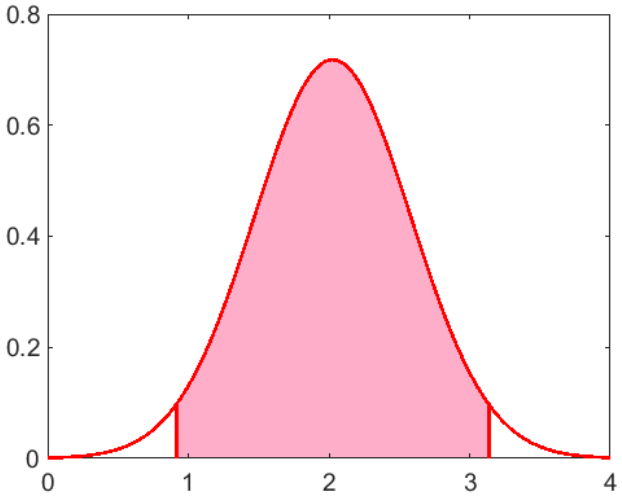
The total area under the curve must be 1 (due to normalisation)

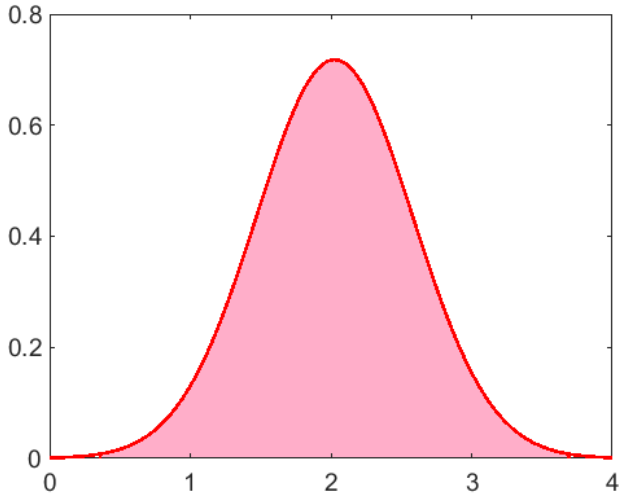
Estimating distributions in this way is a powerful tool when characterising how random observables behave in a system (for example, DLA cluster radius with fixed number of particles…)

Distributions and histograms

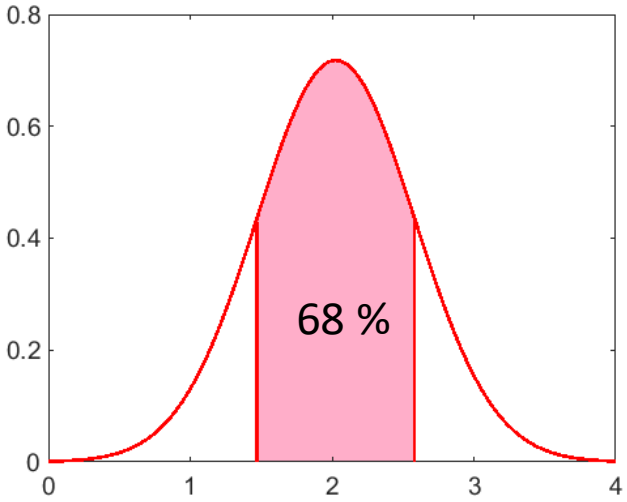


For a Gaussian probability density function, do you remember what percentage of samples falls within the 1, 2 and 3 standard deviation ranges around the mean?

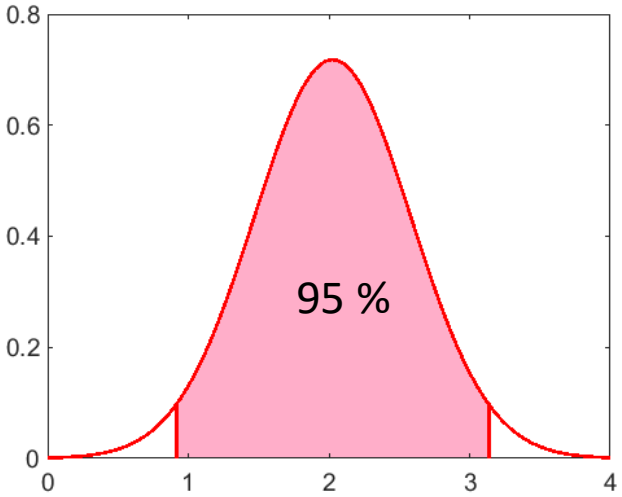


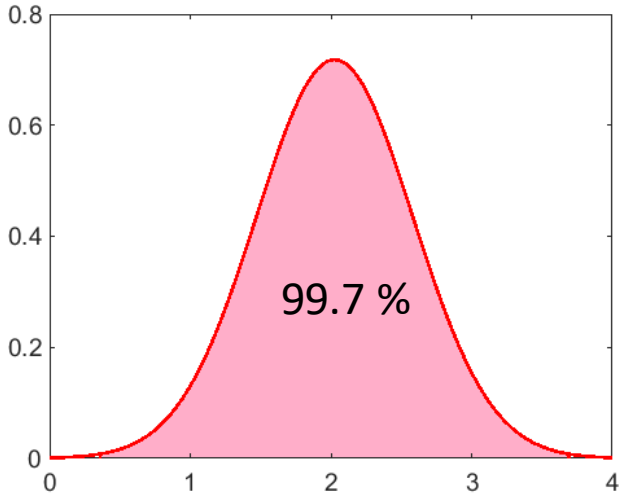


Distributions and histograms



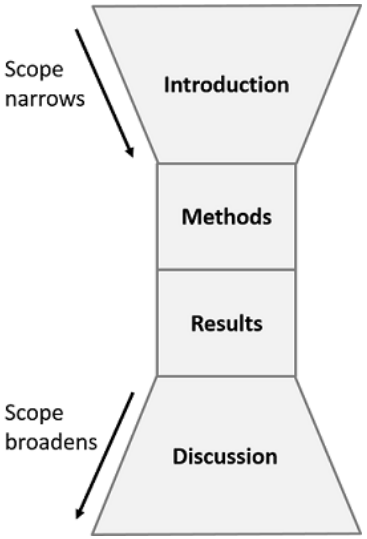
For a Gaussian probability density function, do you remember what percentage of samples falls within the 1, 2 and 3 standard deviation ranges around the mean?





Summary

- Try to use the data from your simulations to tell a story
- A clear report should be engaging, precise, scientifically accurate, and accessible to a third-year student not taking this unit
- Figures can be one of the main pieces of the report, if relevant and well designed. (They do take effort!)
- Simulations are numerical experiments, where errors and uncertainties are important.



There are some guidelines in the report template.
Reading scientific papers can also show you what kind writing and analysis is expected.

PH30056: Computational Physics B: Lecture 5

Semester 2, Academic year 2024/2025

Lectured by David Tsang (dcwt21@bath.ac.uk)

Co-lecturers: Prof Alain Nogaret, Prof Alison Walker

Content of this lecture

Some revision on function overloading and constructors/destructors.

How to use matplotlib for data-analysis and visualization

[SLIDES WILL BE AVAILABLE IN MOODLE !]

Function overloading

Function overloading: C++ allow you to define multiple functions that have *the same name* but *different input types* (roughly speaking, different versions of the same function). Based on the input, the compiler works out which version of the function to use.

Function overloading

```
class Counter {
   private:
      // pCount is like a read-only variable
      double pCount;
   public:
      void   setZero() { pCount = 0.0; }
      // three methods for incrementing the counter
      void Increment() { pCount += 1.0; }
      void Increment(double incr) { pCount += incr;}
};
```

```
int main(void) {
  Counter myCounter;          // declare a counter

  myCounter.setZero();        // set it to zero
  myCounter.Increment();
  myCounter.Increment(3.0);

}
```

Function overloading

```
class Counter {
   private:
      // pCount is like a read-only variable
      double pCount;
   public:
      void   setZero() { pCount = 0.0; }
      // three methods for incrementing the counter
      void Increment() { pCount += 1.0; }
      void Increment(double incr) { pCount += incr;}
};
```

```
int main(void) {
  Counter myCounter;          // declare a counter

  myCounter.setZero();        // set it to zero
  myCounter.Increment();      // increase by 1.0
  myCounter.Increment(3.0); // increase by 3.0

}
```

Overloading and constructors

Function overloading: C++ allow you to define multiple functions that have *the same name* but *different input types* (roughly speaking, different versions of the same function). Based on the input, the compiler works out which version of the function to use.

Repeated...

Constructors: When creating new objects, one can set-up initial values for its member variables. This can be done through constructor functions. Constructors have the same name as the class itself. A class can have multiple constructors (= overloading). Constructors can be used in....

Dynamical allocation of memory

Sometimes we do not know how much memory we need (e.g. how many particles we need to create in our DLASystem)

We can use `new` to allocate memory dynamically (while the program is running). This calls the constructor function. When we finish, we have to take care to free the allocated memory.

```
int main(void) {
  Particle *particlePtr;

  // create a new particle and use the pointer to
  // store its address. This calls the (default) constructor!
  particlePtr = new Particle();

  // rest of program goes here...

  // when we have finished we should delete
  // the Particle that is pointed to by the pointer
  delete particlePtr; // this calls the destructor!
}
```

Particle class (Particle.h)

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

Function overloading: two constructors; they both have the same name, but different inputs

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

Command

`particlePtr = new Particle();`

would use this constructor to create a variable of data type `Particle`

This function calls the second constructor. It creates a new `Particle` and gives initial values for the `pos` array

```
// add a particle to the system at a specific position
void DLASystem::addParticle(double pos[]) {
    // create a new particle
    Particle * p = new Particle(pos);
    // push_back means "add this to the end of the list"
    particleList.push_back( p );
    numParticles++;

    // pos coordinates should be -gridSize/2 < x < gridSize/2
    setGrid(pos,1);
}
```

```
1   #pragma once
2
3   class Particle {
4   public:
5       static const int dim = 2;  // we are in two dimensions
6       double *pos;  // pointer to an array of size dim, to store the position
7
8       // default constructor
9       Particle() {
10          pos = new double[dim];
11      }
12      // constructor, with a specified initial position
13      Particle(double set_pos[]) {
14          pos = new double[dim];
15          for (int d = 0; d < dim; d++)
16              pos[d] = set_pos[d];
17      }
18      // destructor
19      ~Particle() { delete[] pos; }
20  };
```

Destructor function has ~ in front of its name, it takes no input, and its job is to delete the object.

This destructor is (implicitly) called by `clearParticles()` function inside DLAsystem.cpp

DLASystem.h (header file)

```
20
21  class DLASystem {
22  private:

27      // list of particles
28      vector<Particle*> particleList;
29      int numParticles;

34      // size of cluster
35      double clusterRadius;
36      // these are related to the DLA algorithm
37      double addCircle;
38      double killCircle;

63  public:
64      // these are public variables and functions

82      // constructor
83      DLASystem(Window *set_win);
84      // destructor
85      ~DLASystem();

131     // add a particle at pos
132     void addParticle(double pos[]);
133     // add a particle at a random point on the addCircle
134     void addParticleOnAddCircle();
135

154 };
155
```

mainDLA.cpp

```
// this is a global pointer, which is how we access the system itself
DLASystem *sys;
// create the system
sys = new DLASystem(win);
```

These lines in the mainDLA.cpp dynamically allocate memory (`new`) and create the DLA system by calling the constructor

mainDLA.cpp

```
// this is a global pointer, which is how we access the system itself
DLASystem *sys;
// create the system
sys = new DLASystem(win);
```

These lines in the mainDLA.cpp dynamically allocate memory (`new`) and create the DLA system by calling the constructor

DLASystem.cpp

```
// constructor
DLASystem::DLASystem(Window *set_win) {
  cout << "creating system, gridSize " << gridSize << endl;
  win = set_win;
  numParticles = 0;
  endNum = 1000;

  // allocate memory for the grid, remember to free the memory in destructor
  grid = new int*[gridSize];
  for (int i=0;i<gridSize;i++) {
    grid[i]=new int[gridSize];
  }
  slowNotFast = 1;
  // reset initial parameters
  Reset();

  addRatio  = 1.2;   // how much bigger the addCircle should be, compared to cluster radius
  killRatio = 1.7;   // how much bigger is the killCircle, compared to the addCircle

  // this opens a logfile, if we want to...
  logfile.open("opfile.txt");
}
```

These operations will be executed when the constructor is called

DLASystem.cpp

```
// destructor
DLASystem::~DLASystem() {
  // strictly we should not print inside the destructor but never mind...
  cout << "deleting system" << endl;
  // delete the particles
  clearParticles();
  // delete the grid
  for (int i=0;i<gridSize;i++)
    delete[] grid[i];
  delete[] grid;

  if ( logfile.is_open() )
    logfile.close();
}
```

Destructor of the DLASystem class.

DLASystem.cpp

```
// destructor
DLASystem::~DLASystem() {
  // strictly we should not print inside the destructor but never mind...
  cout << "deleting system" << endl;
  // delete the particles
  clearParticles();
  // delete the grid
  for (int i=0;i<gridSize;i++)
    delete[] grid[i];
  delete[] grid;

  if ( logfile.is_open() )
    logfile.close();
}
```

Destructor of the DLASystem class.

mainDLA.cpp

```
// openGL function deals with the keyboard
void drawFuncs::handleKeypress(unsigned char key, int x, int y) {
    switch (key) {
  case 'h':
    drawFuncs::introMessage();
    break;
  case 'q':
  case 'e':
      cout << "Exiting..." << endl;
    // delete the system
    delete sys;
    exit(0);
    break;
  case 'p':
    cout << "pause" << endl;
    sys->pauseRunning();
    break;
```

Here, `delete sys;` command calls the destructor.

All the remaining used memory is freed when `exit(0);` is called

Few notes on vector objects

Used as containers of other objects. Their sizes can be changed while the program is running...
Useful if you cannot tell how many particles you need to create in your DLA program

```
// these commands could all be inside main()

// make a list of 10 integers all with value 3
vector<int> myList(10, 3);

myList.resize(20);  // change the size of the list to 20

  // print the size of the list (this will be 20)
cout << "list size " << myList.size() << endl;

// add the number 4 to the end of the list
// (and increase the size by 1)
myList.push_back(4);

// delete the last entry in the list
// (and decrease size by 1)
myList.pop_back();
```

```
class DLASystem {
  private:
  // these are private variables and functions that the user will not see

    Window *win;  // window in which the system is running

    // list of particles
    vector<Particle*> particleList;
    int numParticles;
```

DLASystem.h

vector container that contains pointers to Particle objects is created

```
class DLASystem {
  private:
  // these are private variables and functions that the user will not see

    Window *win;  // window in which the system is running

    // list of particles
    vector<Particle*> particleList;
    int numParticles;

     // add a particle to the system at a specific position
    void DLASystem::addParticle(double pos[]) {
      // create a new particle
      Particle * p = new Particle(pos);
      // push_back means "add this to the end of the list"
      particleList.push_back( p );
      numParticles++;

      // pos coordinates should be -gridSize/2 < x < gridSize/2
      setGrid(pos,1);
    }
```

DLASystem.h

vector container that contains pointers to Particle objects is created

DLAsystem.cpp

.push_back(p) is used to add a new pointer to a Particle in the end of the vector container

DLASystem.h

```
class DLASystem {
  private:
  // these are private variables and functions that the user will not see

    Window *win;  // window in which the system is running

    // list of particles
    vector<Particle*> particleList;
    int numParticles;
```

vector container that contains pointers to Particle objects is created

DLAsystem.cpp

```
    // add a particle to the system at a specific position
    void DLASystem::addParticle(double pos[]) {
      // create a new particle
      Particle * p = new Particle(pos);
      // push_back means "add this to the end of the list"
      particleList.push_back( p );
      numParticles++;

      // pos coordinates should be -gridSize/2 < x < gridSize/2
      setGrid(pos,1);
    }
```

.push_back(p) is used to add a new pointer to a Particle in the end of the vector container

```
    // make a random move of the last particle in the particleList
    void DLASystem::moveLastParticle() {
      int rr = rgen.randomInt(4);  // pick a random number in the range 0-3
      double newpos[2];

      Particle *lastP = particleList[ numParticles-1 ];

      setPosNeighbour(newpos,lastP->pos,rr);

      if (distanceFromOrigin(newpos) > killCircle ) {
        //cout << "#deleting particle" << endl;
        setGrid( lastP->pos, 0);
        particleList.pop_back();  // remove particle from particleList
        numParticles--;
        setParticleInactive();
      }
```

.pop_back() is used to remove the last pointer to a Particle from the list (if the Particle goes beyond the killing circle

About C++ programming

Few good websites about C++ programming
https://www.cprogramming.com/tutorial/c++-tutorial.html
http://www.cplusplus.com/doc/tutorial/
https://isocpp.org/faq

Remember to allocate enough time for the data-analysis, plotting graphs and writing the report!

Since this is a physics course, concentrate on writing about the physics in your report (not programming).

Matplotlib

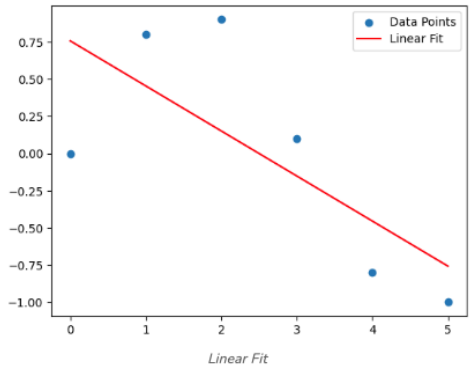
Matplotlib is the matlab-like plotting library in python. It is very powerful, and free! You should have already used this a little bit in your 1st and 2nd year computing.

Very useful guide on Matplotlib plotting with plenty of gorgeous examples you can steal from: https://github.com/rougier/scientific-visualization-book

To fit a polynomial

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0, 0.8, 0.9, 0.1, -0.8, -1.0])

# Perform linear fit
coefficients = np.polyfit(x, y, 1)
print("Linear Fit Coefficients:", coefficients)

# Create polynomial function
p = np.poly1d(coefficients)

plt.scatter(x, y, label='Data Points')
plt.plot(x, p(x), label='Linear Fit', color='red')
plt.legend()
plt.show()
```

```
Linear Fit Coefficients: [-0.30285714  0.75714286]
```

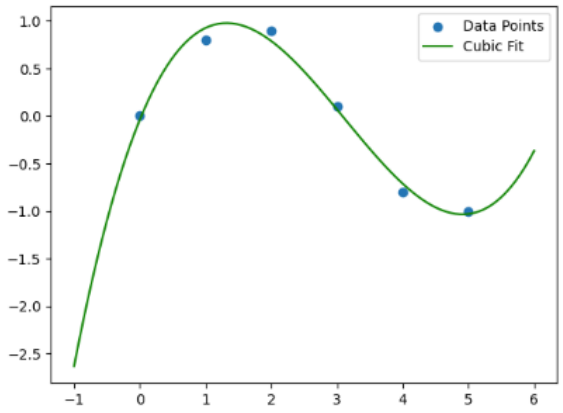


Linear Fit

To fit a polynomial

```python
# Perform cubic fit
coefficients = np.polyfit(x, y, 3)
print("Cubic Fit Coefficients:", coefficients)

# Create polynomial function
p = np.poly1d(coefficients)

# Plot data and fit
xp = np.linspace(-1, 6, 100)
plt.scatter(x, y, label='Data Points')
plt.plot(xp, p(xp), label='Cubic Fit', color='green')
plt.legend()
plt.show()
```

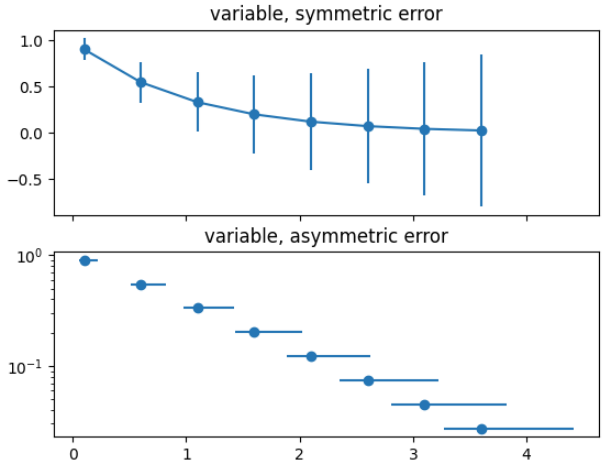
```
Cubic Fit Coefficients: [ 0.08703704 -0.81349206  1.69312169 -0.03968254]
```



Higher-Degree Polynomial Fit

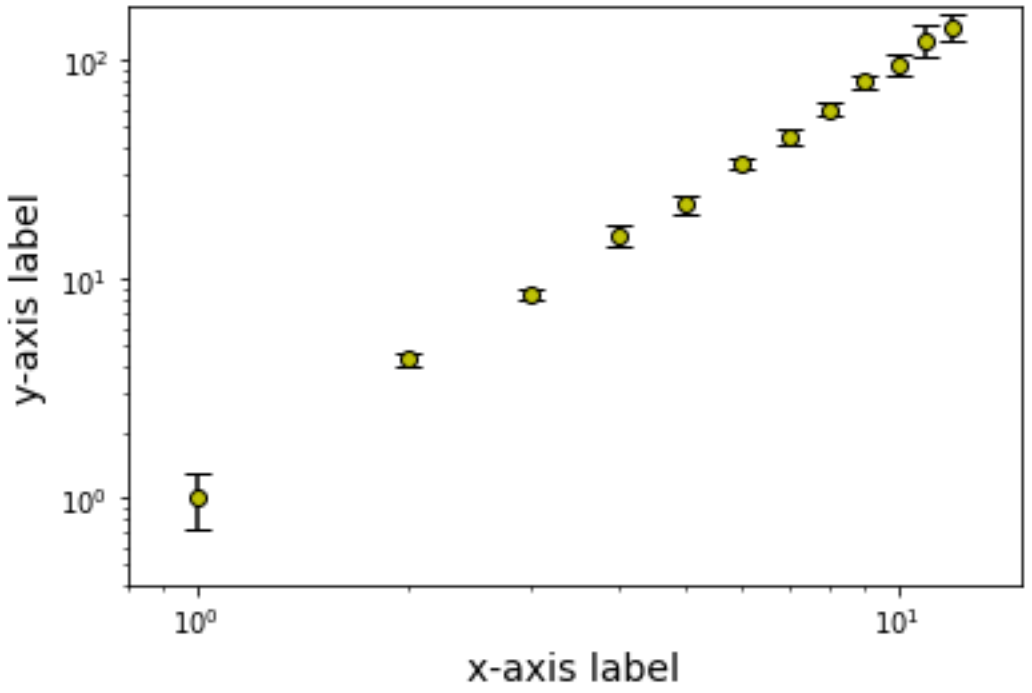
To draw errorbars

```
import matplotlib.pyplot as plt
import numpy as np

# example data
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)

# example error bar values that vary with x-position
error = 0.1 + 0.2 * x

fig, (ax0, ax1) = plt.subplots(nrows=2, sharex=True)
ax0.errorbar(x, y, yerr=error, fmt='-o')
ax0.set_title('variable, symmetric error')

# error bar values w/ different -/+ errors that
# also vary with the x-position
lower_error = 0.4 * error
upper_error = error
asymmetric_error = [lower_error, upper_error]

ax1.errorbar(x, y, xerr=asymmetric_error, fmt='o')
ax1.set_title('variable, asymmetric error')
ax1.set_yscale('log')
plt.show()
```



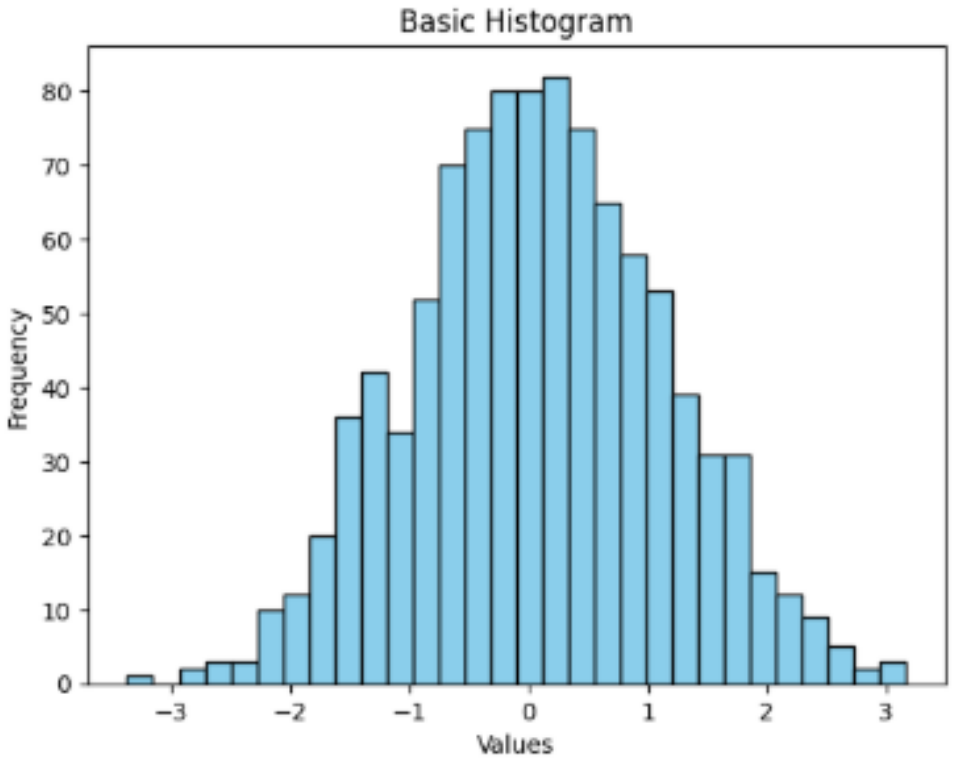
To draw a loglog plot

```
import numpy as np
import matplotlib.pyplot as plt

# Here's an error bar plot with both axes on a log scale.
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
y = [1.02, 4.3, 8.6, 16, 22, 34, 45, 60.2, 80.1, 96, 125, 144]
dy = [.3, .3, .5, 2, 2, 2, 4, 4, 6, 10, 20, 20]
formattedPlot = plt.errorbar(x, y, dy, fmt = 'ko', markersize = 6,\
    capsize = 5, linewidth = 1.5, markerfacecolor = 'y');
plt.xlabel('x-axis label', fontsize = 14, fontname = 'Times')
plt.ylabel('y-axis label', fontsize = 14, fontname = 'Times')
plt.xscale('log')
plt.yscale('log')
plt.axis((0.8, 15, 0.4, 180));
```



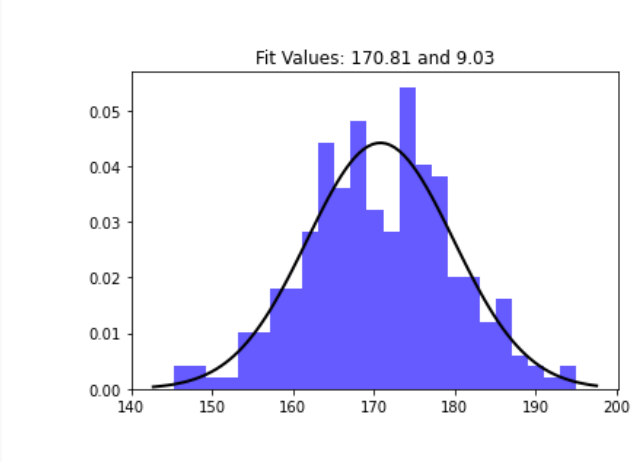
To draw histograms

```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data for the histogram
data = np.random.randn(1000)

# Plotting a basic histogram
plt.hist(data, bins=30, color='skyblue', edgecolor='black')

# Adding labels and title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Basic Histogram')

# Display the plot
plt.show()
```

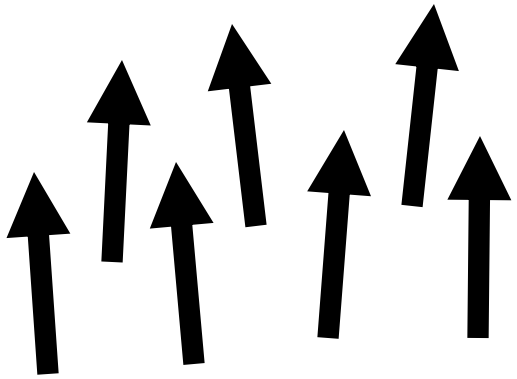


To draw Normal distribution

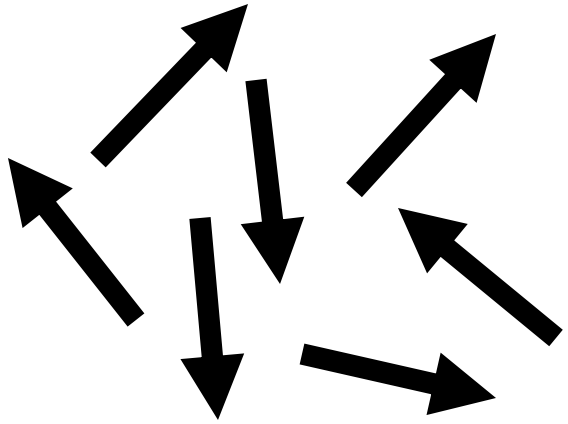
```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

# Generate some data for this
# demonstration.
data = np.random.normal(170, 10, 250)

# Fit a normal distribution to
# the data:
# mean and standard deviation
mu, std = norm.fit(data)

# Plot the histogram.
plt.hist(data, bins=25, density=True, alpha=0.6, color='b')

# Plot the PDF.
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = norm.pdf(x, mu, std)

plt.plot(x, p, 'k', linewidth=2)
title = "Fit Values: {:.2f} and {:.2f}".format(mu, std)
plt.title(title)

plt.show()
```



Lecture 7: Introduction to the Ising model



Ferromagnet
(low temperature)



Paramagnet
(high temperature)

Alison Walker pysabw@bath.ac.uk

Content of this lecture

Brief reminder of ferromagnetism, and introduction to the Ising model

How to simulate the Ising model (Metropolis Monte Carlo algorithm)

Application of statistical mechanics to the Ising model (Boltzmann distribution)

How to use computer simulations to measure/predict physical observables such as energy or magnetisation, as a function of temperature (including a note on errors)

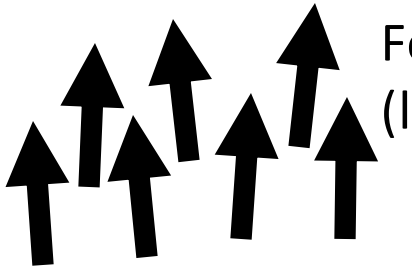
Ferromagnets

Electron spins in metals can lead to *ferromagnetism* (permanent magnets). The reason is that if the spins of electrons in nearby atoms align in order to lower their energy.

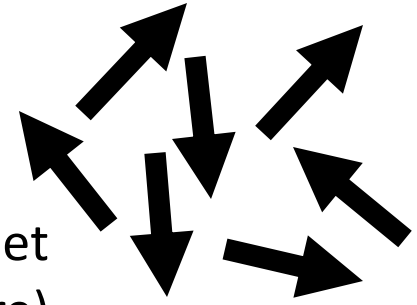
This means that at room temperature (and at lower temperatures), materials like iron prefer to have all their spins aligned.

At higher temperatures, systems prefer to increase their *entropy*, so all the spins point in random directions.

These high and low temperature regimes are separated by a special point, called a *phase transition.*

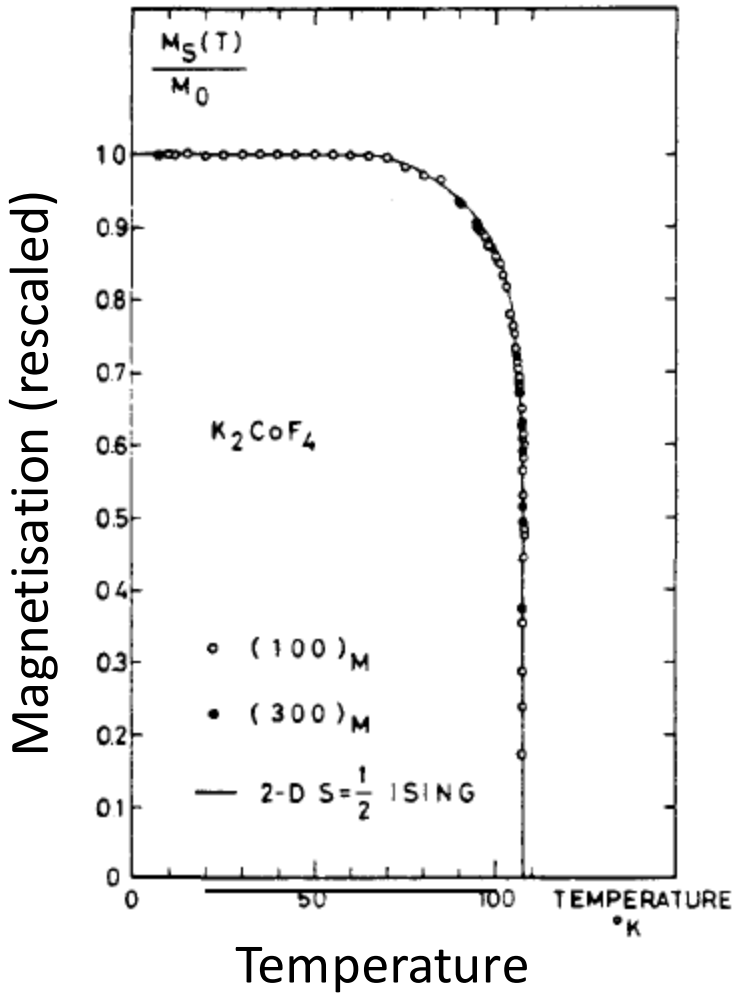


Ferromagnet
(low temperature)



Paramagnet
(high temperature)

Phase transition

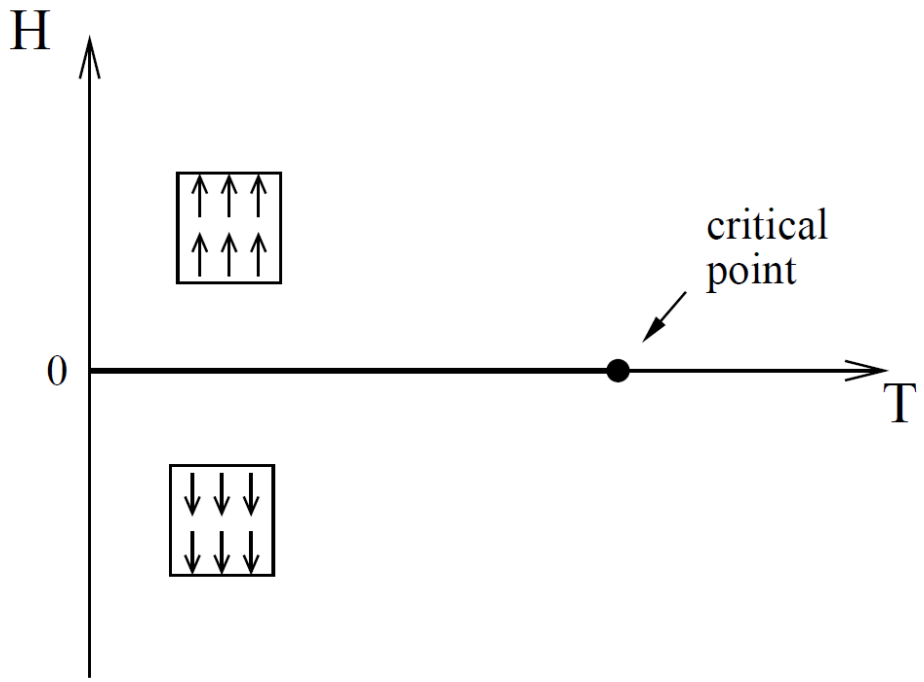


Experimental data for a magnetic material consisting of potassium, cobalt and fluorine

Sharp increase in magnetisation at the phase transition (approx 108K)

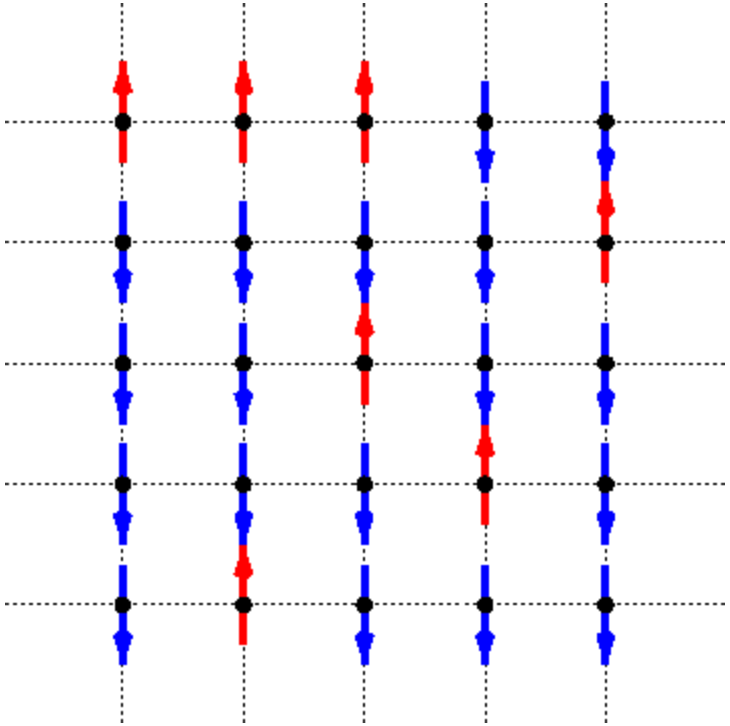
H Ikeda and K Hirakawa, Solid State Communications 14, 529 (1974)

Phase diagram

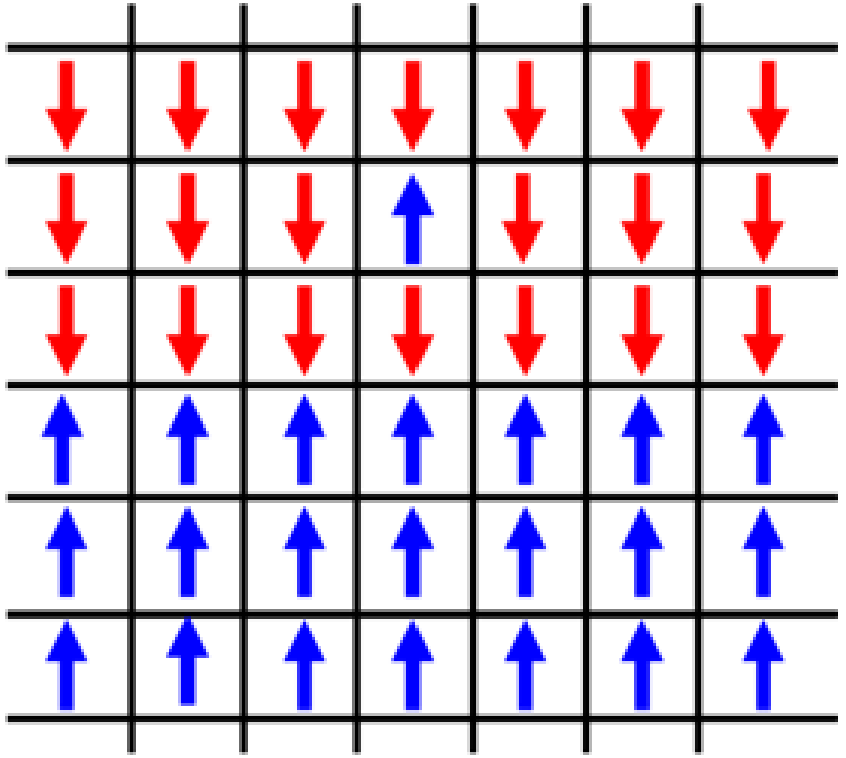


Up and down phases in a magnetic field-temperature phase diagram

Ising model of a ferromagnet



Mostly spin down
with some spin up

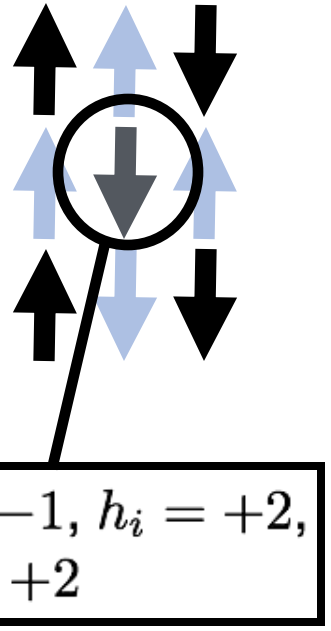


Domain wall between
spin down and spin up

Ising model

[E Ising, Zeitschrift fuer Physik A, **31**, 253 (1925)]

For simplicity, consider N spins on a square grid.
Let s_i be the state of spin i.
For simplicity, either $s_i = 1$ (up) or $s_i = -1$ (down)



Magnetisation $M = \frac{1}{N} \sum_i s_i$, is between -1 (all down) and $+1$ (all up)

The energy of each spin depends only on its nearest neighbours (n.n.) as $E_i = -h_i s_i$, with "local magnetic field"

$$h_i = J \sum_{\text{n.n. } j \text{ of } i} s_j$$

where J is the strength of the interaction.

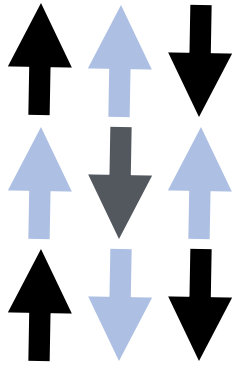
[see also the coursework handout]

Ising model

Repeated from previous page

The energy of each spin depends only on its nearest neighbours (n.n.) as $E_i = -h_i s_i$, with "local magnetic field"

$$h_i = J \sum_{\text{n.n. } j \text{ of } i} s_j$$



The total energy is $E = \frac{1}{2}\sum_i E_i$ The factor of $\frac{1}{2}$ is just for convenience, it means that we can write

$$E = -J \sum_{\langle ij \rangle} s_i s_j$$

where the sum runs over all pairs of nearest neighbours (with each pair counted once)

Physically, the system at low temperatures should have low energy, expect $M \approx \pm 1$.

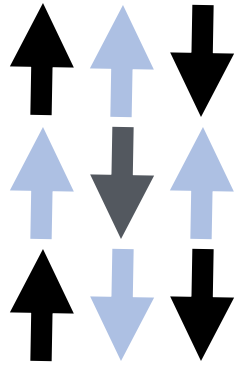
At high temperatures, symmetry means that up and down spins are equally likely, expect $M = 0$.

Ising model

Repeated from previous page

The energy of each spin depends only on its nearest neighbours (n.n.) as $E_i = -h_i s_i$, with "local magnetic field"

$$h_i = J \sum_{\text{n.n. } j \text{ of } i} s_j$$



The total energy is $E = \frac{1}{2}\sum_i E_i$ The factor of $\frac{1}{2}$ is just for convenience, it means that we can write

$$E = -J \sum_{\langle ij \rangle} s_i s_j$$

where the sum runs over all pairs of nearest neighbours (with each pair counted once)

$$E = -J \sum_{<ij>} s_i s_j - H \sum_i s_i$$

External field H

Ising model

In the coursework, the Ising model is studied by using:

1) Metropolis algorithm
   - Computational approach based on Monte Carlo simulations
   - More details in this lecture (see, also the coursework handout)

2) Theoretical solutions
   - Exact solution for 2d Ising model by L. Onsager (1944)
   - Mean-field (Curie-Weiss) theory
   - We'll see more details about these in the next lecture (see, also the coursework handout)

Both options have their benefits and limitations. One of your tasks in the coursework is to compare the numerical and theoretical results.

Metropolis Monte Carlo

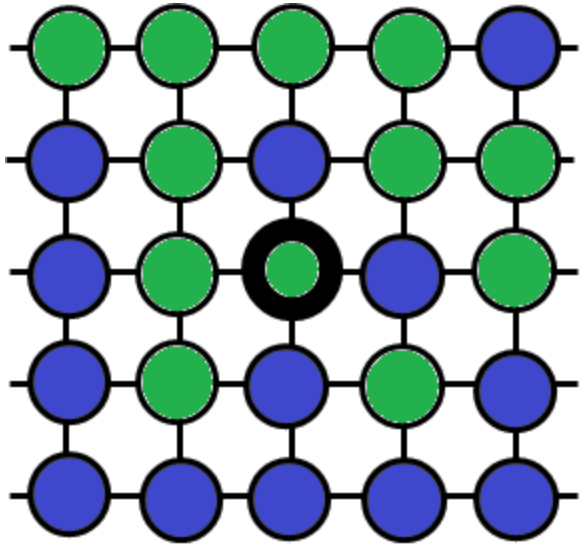
[N Metropolis *et al*, J. Chem Phys. **21**, 1087 (1953)]

We want to investigate the Ising model at *thermal equilibrium* (i.e. in a steady state given by the Boltzmann distribution)

Algorithm:

Pick randomly a spin site:

- Calculate the energy change $\Delta E_i = 2h_i s_i$ associated with the "flipping" the spin *i* (changing its colour)
- If $\Delta E_i < 0$, then flip the spin
- If $\Delta E_i > 0$ then flip the spin with probability $p_i = e^{-\Delta E_i/(k_B T)}$, where *T* is temperature



Repeat the process many times until thermal equilibrium is achieved

With this choice, the steady state of the model is consistent with the Boltzmann distribution

Boltzmann distribution

If we specify all the spins in the system, we say that the system is in a *microstate,* $S = \{s_1, s_2, \ldots, s_N\}$

In the equilibrium *macrostate* at a given temperature *T*, the system is in a microstate S with probability

$$p_B(S) = \frac{1}{Z(T)} e^{-E(S)/(k_B T)}$$

where $E(S)$ is the energy of the microstate and $Z(T) = \sum_S e^{-E(S)/(k_B T)}$ is the partition function (normalization constant)

Thermal equilibrium

Let's take a two microstate $S_1 = \{s_1, s_2, \ldots, si, \ldots, sN\}$ and $S_2 = \{s_1, s_2, \ldots, -si, \ldots, sN\}$ which are otherwise similar except the spin i is flipped, and that S_2 is higher in energy by ΔE.

Now, the probability of going from microstate S_1 to S_2 is

$$P(S_1 \rightarrow S_2) = pB(S_1) \times \frac{1}{N} \times e^{-\Delta E/(k_B T)}$$

probability of starting in state S_1

probability of choosing spin i

probability of accepting the move

On the other hand, the probability of going from microstate S_2 to S_1 is

$$P(S_2 \rightarrow S_1) = pB(S_2) \times \frac{1}{N} \times 1$$

because the state S_1 has lower energy.

Thermal equilibrium

The principle of *detailed balance* states that in thermal equilibrium the rate at which the system makes transitions from microstate S_1 to S_2 is equal to the rate at which the system makes reverse transitions

You can see that $P(S_1 \rightarrow S_2) = P(S_2 \rightarrow S_1)$ if

$$\frac{p_B(S_2)}{p_B(S_1)} = e^{-\Delta E/(k_B T)}$$

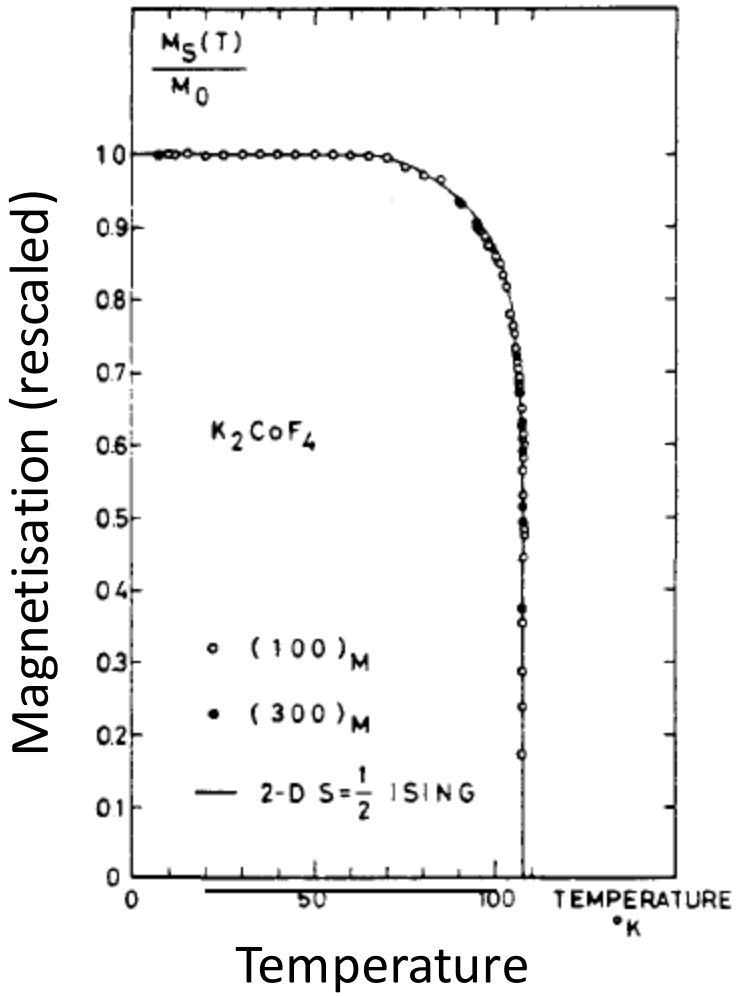
This indeed is true based on the definition of the Boltzmann distribution

$$p_B(S_1) = \frac{1}{Z(T)} e^{-E(S_1)/(k_B T)} \text{ and } p_B(S_2) = \frac{1}{Z(T)} e^{-E(S_2)/(k_B T)}$$

Hence the Metropolis algorithm obeys detailed balance

This can be used to prove that the method converges to the Boltzmann distribution

Ising model and real magnets



Experimental data for a magnetic material consisting of potassium, cobalt and fluorine (the *points* in the figure)

The *line* is a fit to results from an Ising model.

The model gets the shape of the curve precisely right…

H Ikeda and K Hirakawa, Solid State Communications 14, 529 (1974)

Outlook

Things to think about

It is clear that real magnetic materials are not as simple as the Ising model. Can we expect the model to agree with experiment?

Which quantities are most interesting to measure (and compare with experiment)?

Is there really a special temperature (phase transition) or can the system transform smoothly from M close to zero at high temperature to large M at low temperature?

Can we use theory to understand the model, and does this help us to understand the experiments?

Statistical Mechanics

Experimentally-observable quantities are obtained as averages.

For example, the average energy is

$$\langle E \rangle = \sum_S E(S) p(S) = \frac{1}{Z(T)} \sum_S E(S) \mathrm{e}^{-E(S)/k_\mathrm{B}T}$$

For the Ising model we can use our formula for the energy to write

$$p(S) = \frac{1}{Z(T)} \exp\left(-\beta_0 \sum_{\langle ij \rangle} s_i s_j\right)$$

where $\beta_0 = J/(k_\mathrm{B}T)$ is a dimensionless measure of the interaction strength.

Note: p is dimensionless, it can only depend on dimensionless variables such as β_0 (which can also be interpreted as an inverse temperature).

Ergodicity

So far, we explained that the system at equilibrium should be described by a Boltzmann distribution

Another important idea in statistical mechanics is that instead of averaging over many copies of a system (an *ensemble*), we can take averages over some long time period...

... we (usually) expect these two averages to be the same...

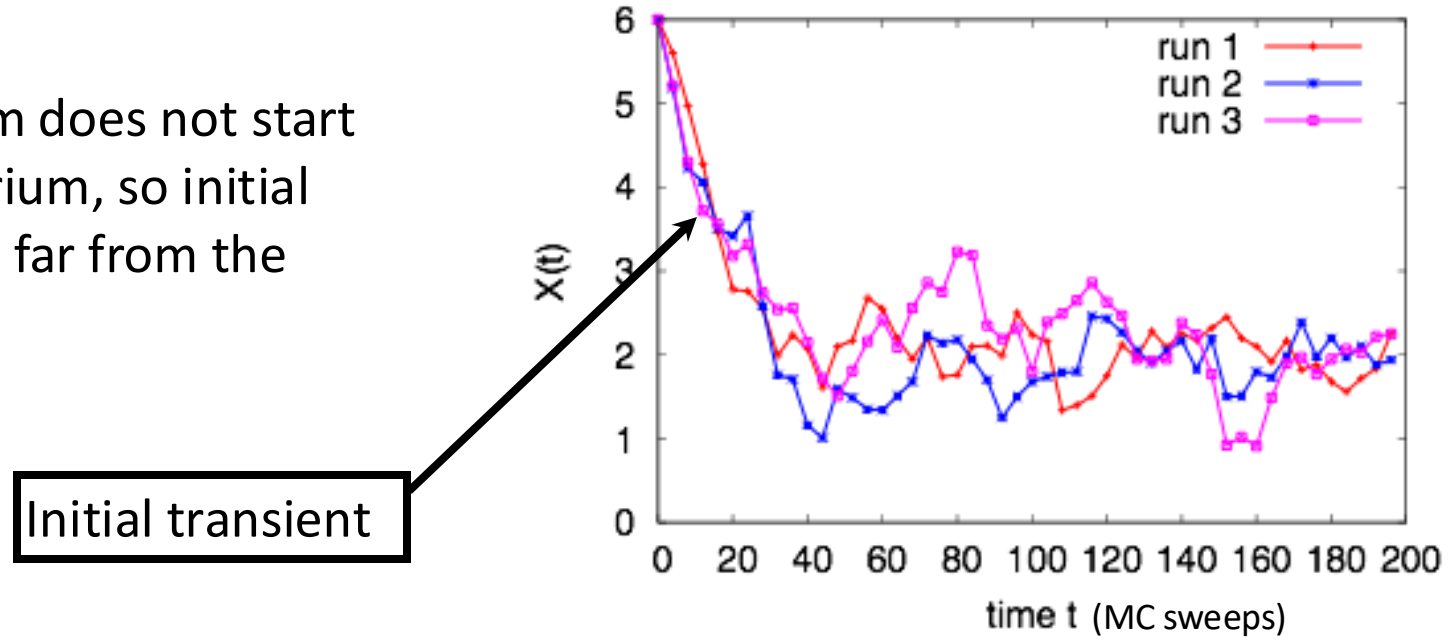
This is useful in *computational physics* because we can *simulate* the system over a long time, and use these data to estimate averages...

... e.g. measure how the (average) energy and magnetisation vary as a function of temperature

Ergodicity

Consider these example data where we measure some quantity X as a function of simulation time t, in three separate runs

The system does not start at equilibrium, so initial values are far from the average

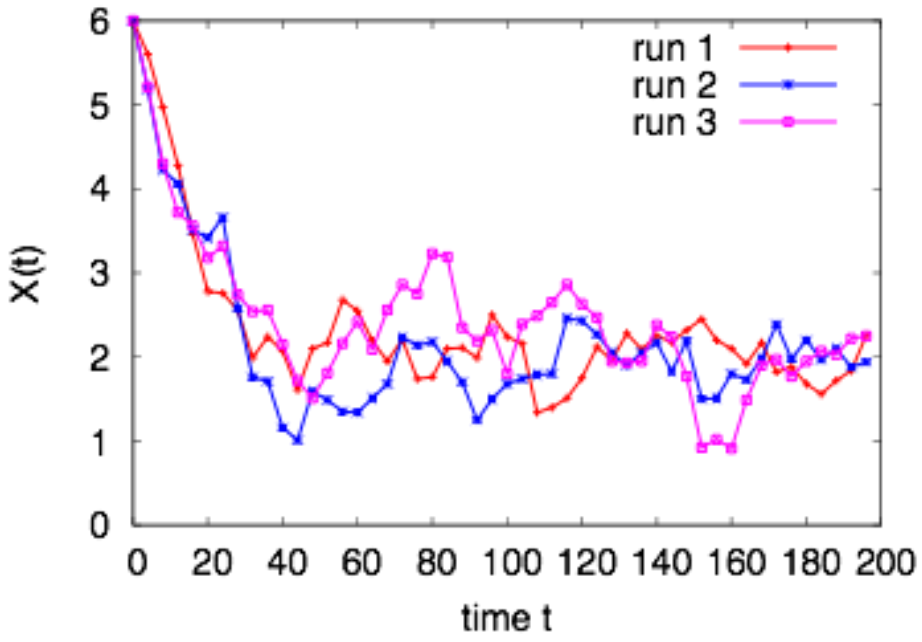


The idea of ergodicity is that if we ignore the transient, we can either average over different runs at the same time (e.g. t = 100) or we can average the data from a single runs at different times

[For example, average all the blue data for all times t > 50]

Time averages

Data points at nearby times are not independent (e.g. if one data point is above average, the next one is also likely to be above average)



To have an efficient estimate of the average, it is useful to have (more-or-less) independent measurements...

Suggested method for estimating averages: discard the first n_0 MC sweeps. Then take measurements every n MC sweeps. Do this until you have m measurements.

Overall you need to run for $n_0 + nm$ MC sweeps. For accuracy one should take n_0 and m large. For efficiency, it's good to have n large enough that your measurements are (roughly) independent

A note on errors...

Suggested method for estimating averages: discard the first n_0 MC sweeps. Then take measurements every n MC sweeps. Do this until you have m measurements.

Repeated from previous page

Your estimate of $\langle X \rangle$ will be

$$\overline{X} = \frac{1}{m} \sum_{i=1}^{m} X_i$$

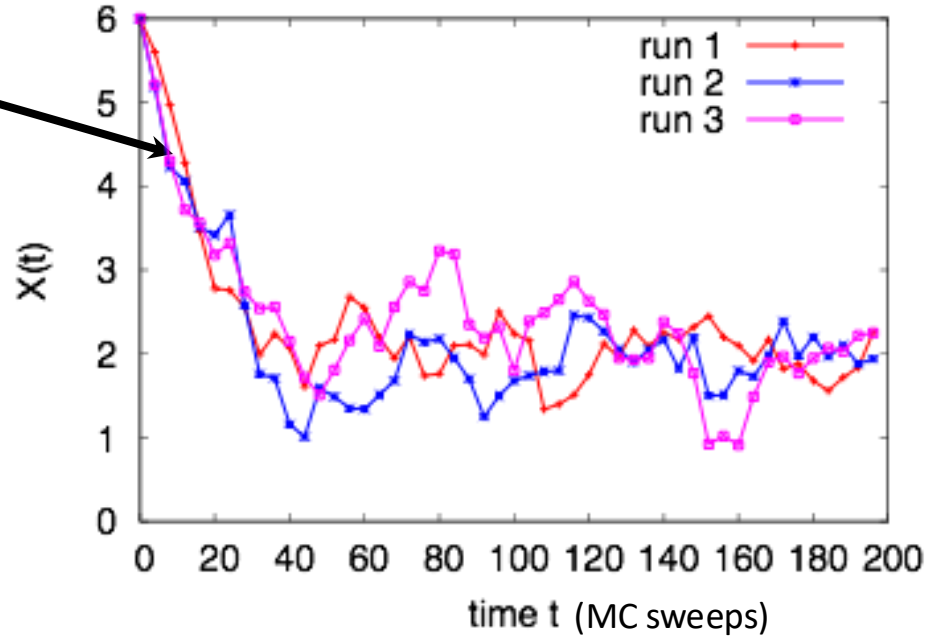
where X_i is the ith measurement.

If the measurements are independent then the standard error gives a good estimate of the uncertainty:

$$\overline{\sigma}_X = \frac{1}{\sqrt{m-1}} \sqrt{\frac{1}{m} \sum_{i=1}^{m} (X_i - \overline{X})^2}$$

But note: if measurements are not independent then $\overline{\sigma}_X$ is an underestimate of your uncertainty about the average.

Initial transients



Initial transients can also be interesting in themselves...

... if a system takes a long time to converge to its equilibrium state, this may be an indication that some interesting collective motion is happening...

... for example "coarsening", "aging"... slow transients in experiments can be compared with theories and models, to understand how different materials behave...

Note also: failing to exclude the transient when estimating averages can lead to *systematic errors*

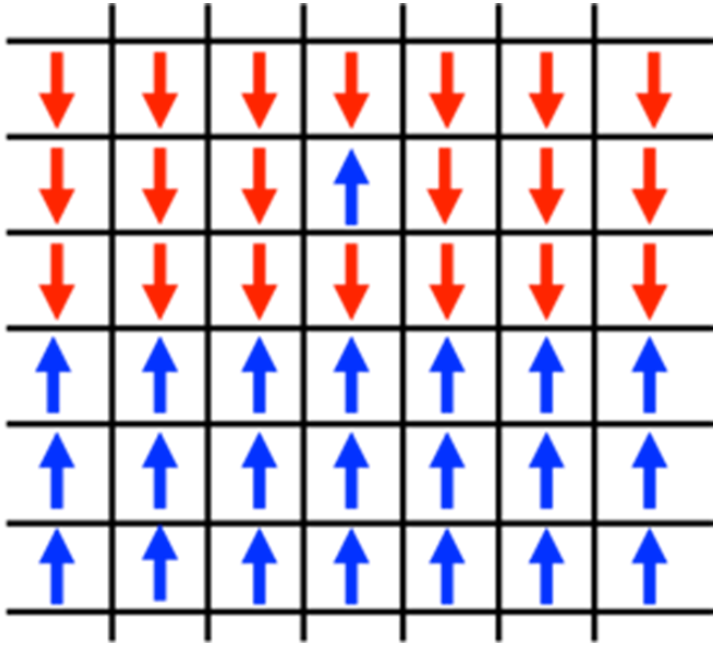
In general

The two dimensional Ising model is complicated enough to have non-trivial properties but simple enough to have an analytic solution

Ising model describes also other phase transitions (for example condensation) and other random systems that consist of elements that have two states (for example, neurons and neural networks in the brain)

Metropolis algorithm has many variants, for example Metropolis-Hastings algorithm which is used to sample multidimensional probability distributions

Summary



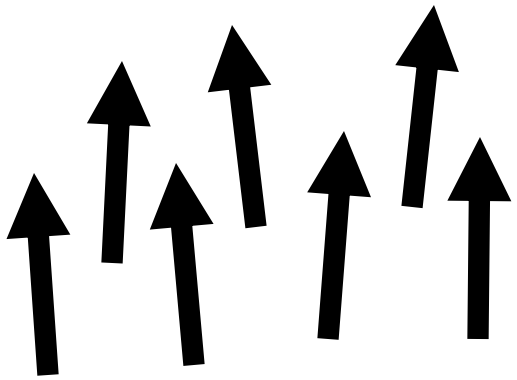
The Ising model is a very simple model of a magnet, in which nearby spins have low energy if they are aligned

The model has an equilibrium state described by the Boltzmann distribution

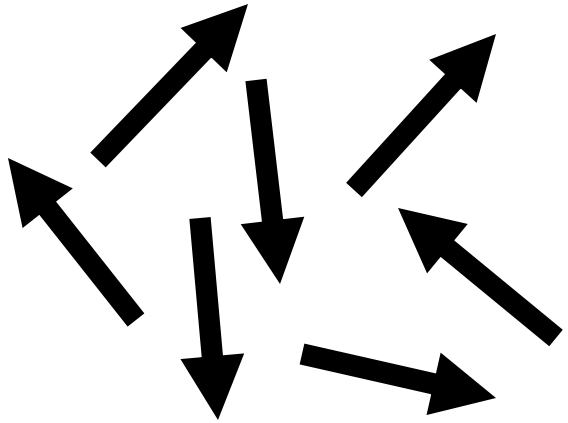
We can simulate the model using the Metropolis algorithm, and ergodicity means that we can use these simulations to estimate averages of observable quantities

When making such estimates and analysing their errors, we need to think about the initial transient and about using independent samples.

Lecture 8: Mean field theory of the Ising model



Ferromagnet
(low temperature)



Paramagnet
(high temperature)

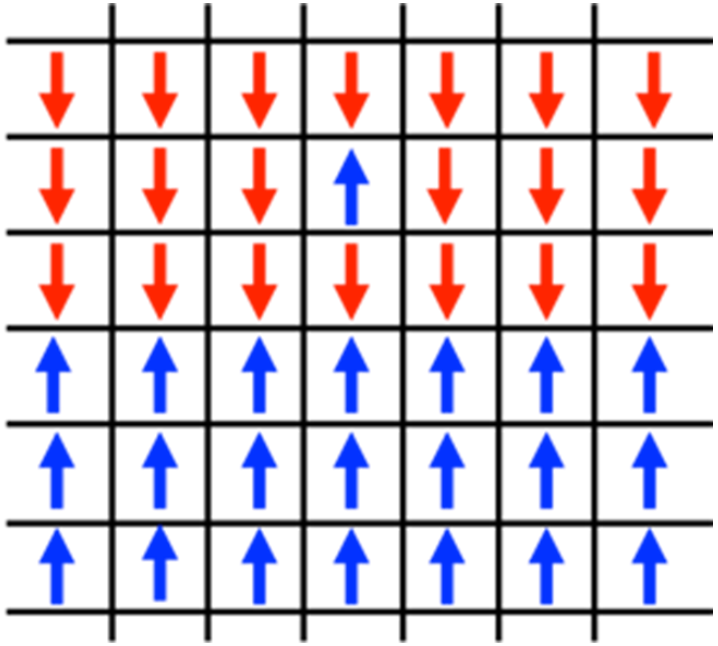
Content of this lecture

Revision of the Ising model

Mean field theory of the Ising model

Dimensionless variables

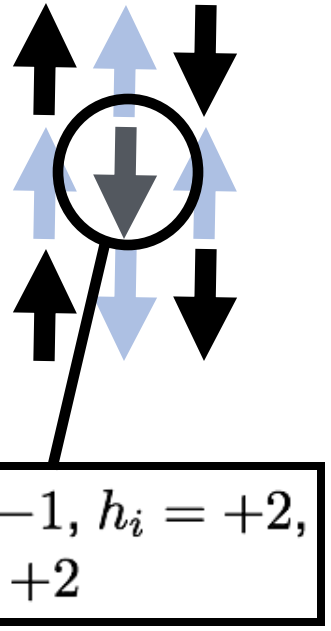
Ising model



Ising model

[E Ising, Zeitschrift fuer Physik A, **31**, 253 (1925)]

For simplicity, consider N spins on a square grid.
Let s_i be the state of spin i.
For simplicity, either $s_i = 1$ (up) or $s_i = -1$ (down)



Magnetisation $M = \frac{1}{N} \sum_i s_i$, is between -1 (all down) and $+1$ (all up)

The energy of each spin depends only on its nearest neighbours (n.n.) as $E_i = -h_i s_i$, with "local magnetic field"

$$h_i = J \sum_{\text{n.n. } j \text{ of } i} s_j$$

where J is the strength of the interaction.

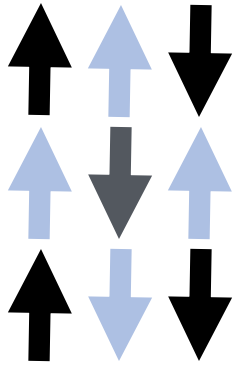
[see also the coursework handout]

Ising model

Repeated from previous page

The energy of each spin depends only on its nearest neighbours (n.n.) as $E_i = -h_i s_i$, with "local magnetic field"

$$h_i = J \sum_{\text{n.n. } j \text{ of } i} s_j$$



The total energy is $E = \frac{1}{2}\sum_i E_i$ The factor of $\frac{1}{2}$ is just for convenience, it means that we can write

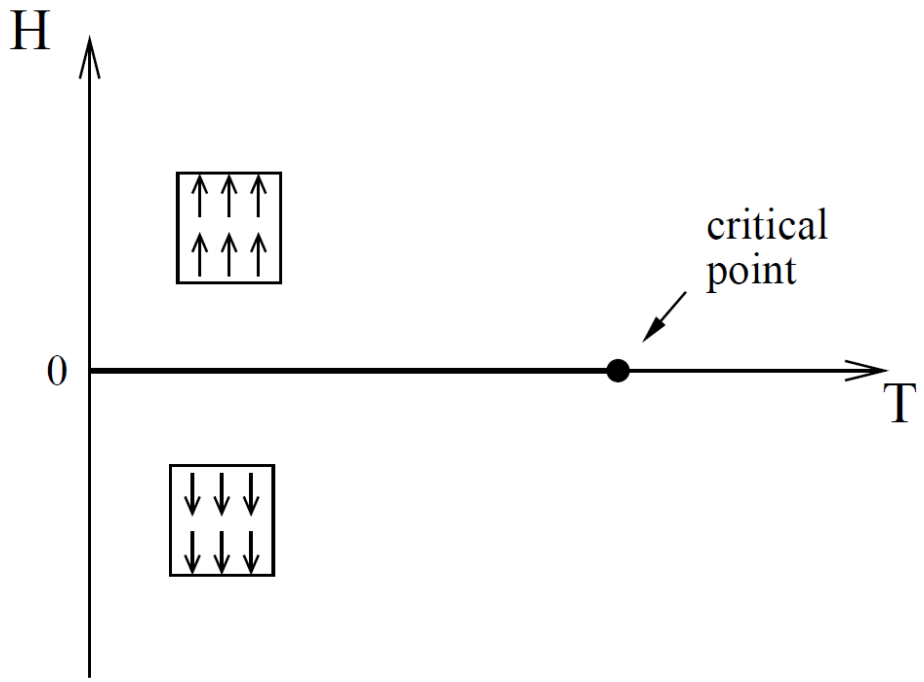
$$E = -J \sum_{\langle ij \rangle} s_i s_j$$

where the sum runs over all pairs of nearest neighbours (with each pair counted once)

Physically, the system at low temperatures should have low energy, expect $M \approx \pm 1$.

At high temperatures, symmetry means that up and down spins are equally likely, expect $M = 0$.

Phase diagram



Up and down phases in a Magnetic field vs Temperature phase diagram

Statistical Mechanics

Experimentally-observable quantities are obtained as averages.

For example, the average energy is

$$\langle E \rangle = \sum_S E(S) p(S) = \frac{1}{Z(T)} \sum_S E(S) \mathrm{e}^{-E(S)/k_\mathrm{B}T}$$

For the Ising model we can use our formula for the energy to write

$$p(S) = \frac{1}{Z(T)} \exp\left(-\beta_0 \sum_{\langle ij \rangle} s_i s_j\right)$$

where $\beta_0 = J/(k_\mathrm{B}T)$ is a dimensionless measure of the interaction strength.

Note: p is dimensionless, it can only depend on dimensionless variables such as β_0 (which can also be interpreted as an inverse temperature).

Ergodicity

So far, we explained that the system at equilibrium should be described by a Boltzmann distribution

Another important idea in statistical mechanics is that instead of averaging over many copies of a system (an *ensemble*), we can take averages over some long time period...

... we (usually) expect these two averages to be the same...

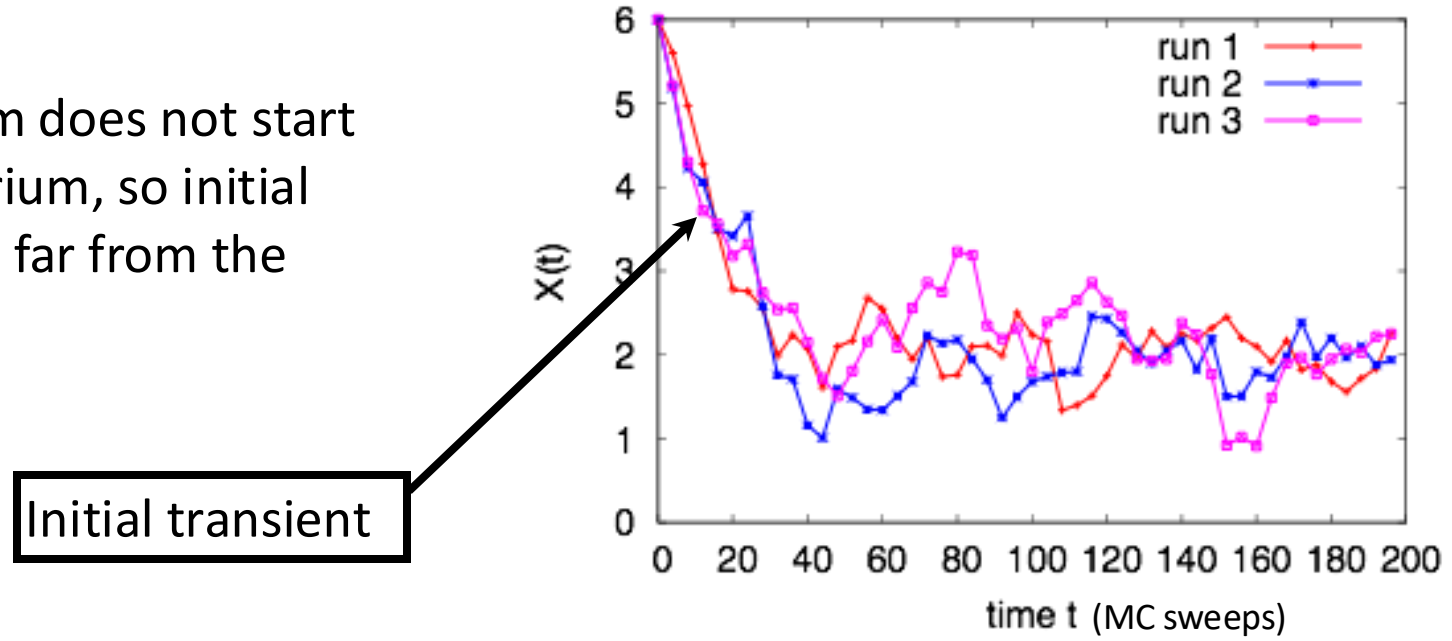
This is useful in *computational physics* because we can *simulate* the system over a long time, and use these data to estimate averages...

... e.g. measure how the (average) energy and magnetisation vary as a function of temperature

Ergodicity

Consider these example data where we measure some quantity X as a function of simulation time t, in three separate runs

The system does not start at equilibrium, so initial values are far from the average

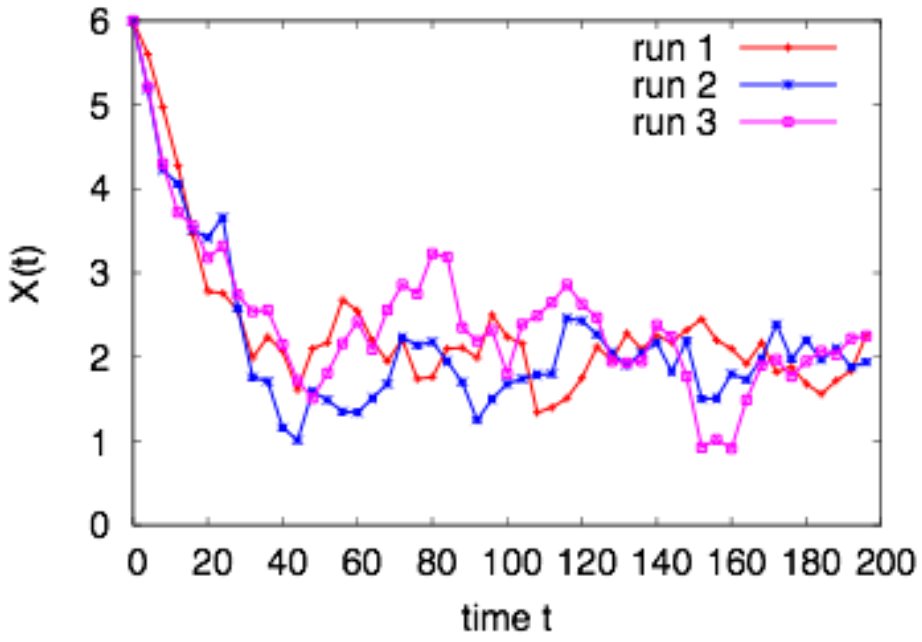


The idea of ergodicity is that if we ignore the transient, we can either average over different runs at the same time (e.g. t = 100) or we can average the data from a single runs at different times

[For example, average all the blue data for all times t > 50]

Time averages

Data points at nearby times are not independent (e.g. if one data point is above average, the next one is also likely to be above average)



To have an efficient estimate of the average, it is useful to have (more-or-less) independent measurements...

Suggested method for estimating averages: discard the first n_0 MC sweeps. Then take measurements every n MC sweeps. Do this until you have m measurements.

Overall you need to run for $n_0 + nm$ MC sweeps. For accuracy one should take n_0 and m large. For efficiency, it's good to have n large enough that your measurements are (roughly) independent

A note on errors...

Suggested method for estimating averages: discard the first n_0 MC sweeps. Then take measurements every n MC sweeps. Do this until you have m measurements.

Repeated from previous page

Your estimate of $\langle X \rangle$ will be

$$\overline{X} = \frac{1}{m} \sum_{i=1}^{m} X_i$$

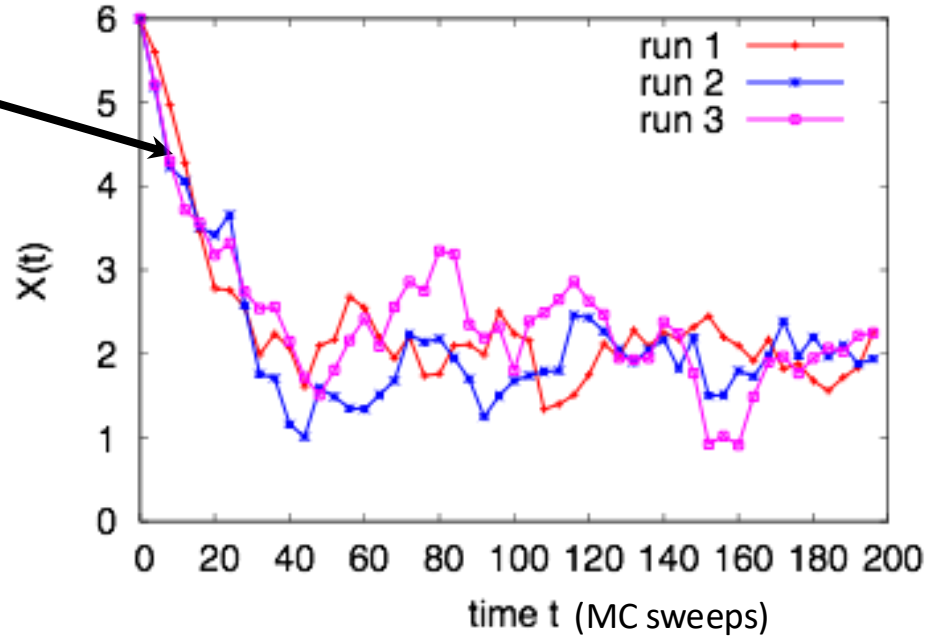
where X_i is the ith measurement.

If the measurements are independent then the standard error gives a good estimate of the uncertainty:

$$\overline{\sigma}_X = \frac{1}{\sqrt{m-1}} \sqrt{\frac{1}{m} \sum_{i=1}^{m} (X_i - \overline{X})^2}$$

But note: if measurements are not independent then $\overline{\sigma}_X$ is an underestimate of your uncertainty about the average.

Initial transients



Initial transients can also be interesting in themselves...

... if a system takes a long time to converge to its equilibrium state, this may be an indication that some interesting collective motion is happening...

... for example "coarsening", "aging"... slow transients in experiments can be compared with theories and models, to understand how different materials behave...

Note also: failing to exclude the transient when estimating averages can lead to *systematic errors*

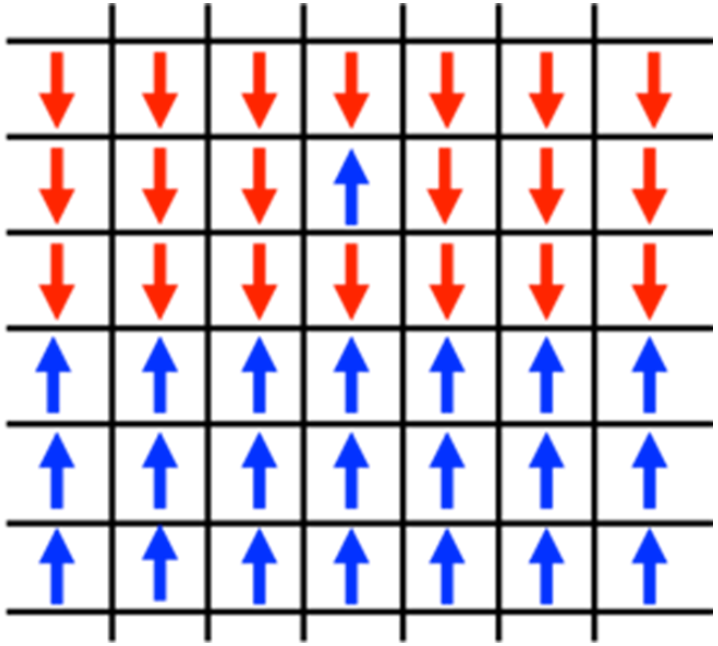
In general

The two dimensional Ising model is complicated enough to have non-trivial properties but simple enough to have an analytic solution

Ising model describes also other phase transitions (for example condensation) and other random systems that consist of elements that have two states (for example, neurons and neural networks in the brain)

Metropolis algorithm has many variants, for example Metropolis-Hastings algorithm which is used to sample multidimensional probability distributions

Summary



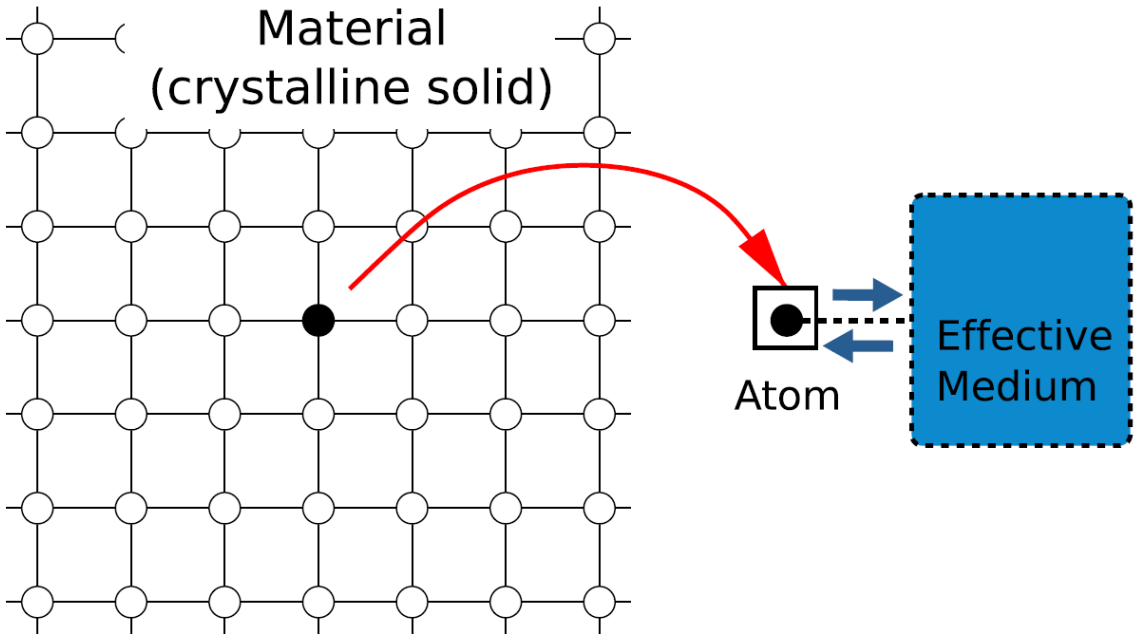
The Ising model is a very simple model of a magnet, in which nearby spins have low energy if they are aligned

The model has an equilibrium state described by the Boltzmann distribution

We can simulate the model using the Metropolis algorithm, and ergodicity means that we can use these simulations to estimate averages of observable quantities

When making such estimates and analysing their errors, we need to think about the initial transient and about using independent samples.

Schematic of mean-field theory



Qualitatively correct, but often quantitatively wrong

C. R. Physique 17 (2016) 430–446

Mean-field theory

[see Section 4 of coursework handout]

Mean-field theory (or Curie-Weiss theory) gives an approximate solution

It is simple and relatively accurate (for a range of temperatures). However, it is inaccurate around the phase transition.

An assumption of mean-field theory is that on average every spin behaves similarly.

Within this assumption, we replace local magnetisation by average magnetisation, *m*:

$$m = < M > = \frac{1}{N} \sum_i s_i = < s_i >$$

Mean-field theory

[see Section 4 of coursework handout]

We can use the Boltzman distribution to determine $< s_i >$. Let's consider the single spin *i* in "magnetic field" h_i and energy $E(s_i) = E_i = -h_i s_i$. There are exactly two microstates with energies $+h_i$ and $-h_i$ (because $s_i = \pm 1$)

$$p(s_i) = \frac{1}{Z(T)} e^{-E(s_i)/(k_B T)} = \frac{1}{Z(T)} e^{h_i s_i/(k_B T)}$$

where $Z(T) = \sum_{s_i=\pm 1} e^{-E(s_i)/(k_B T)} = e^{h_i/(k_B T)} + e^{-h_i/(k_B T)}$

$$< s_i > = \sum_{s_i=\pm 1} s_i p(s_i) = \frac{+1}{Z(T)} e^{\frac{(h_i)(+1)}{k_B T}} + \frac{-1}{Z(T)} e^{-\frac{(h_i)(-1)}{k_B T}}$$

Mean-field theory

By combining the results, we get

$$\langle s_i \rangle = \frac{(\mathrm{e}^{h_i/k_\mathrm{B}T} - \mathrm{e}^{-h_i/k_\mathrm{B}T})}{\mathrm{e}^{h_i/k_\mathrm{B}T} + \mathrm{e}^{-h_i/k_\mathrm{B}T}} = \tanh(h_i/k_\mathrm{B}T)$$

In the Ising model $h_i = J \sum_{\text{n.n. } i \text{ of } j} s_j$

If each spin has $z = 4$ neighbours, then $\langle h_i \rangle = Jz\langle s_i \rangle$
Note, that all the spins are assumed to have the same statistics

Each spin feels the same local field (mean field) $h_i = < h_i > = zJ < s_i >$

$$\langle s_i \rangle = \tanh(\langle h_i \rangle / k_\mathrm{B}T) = \tanh\left(\frac{zJ}{k_\mathrm{B}T}\langle s_i \rangle\right)$$

How can you study this formula?

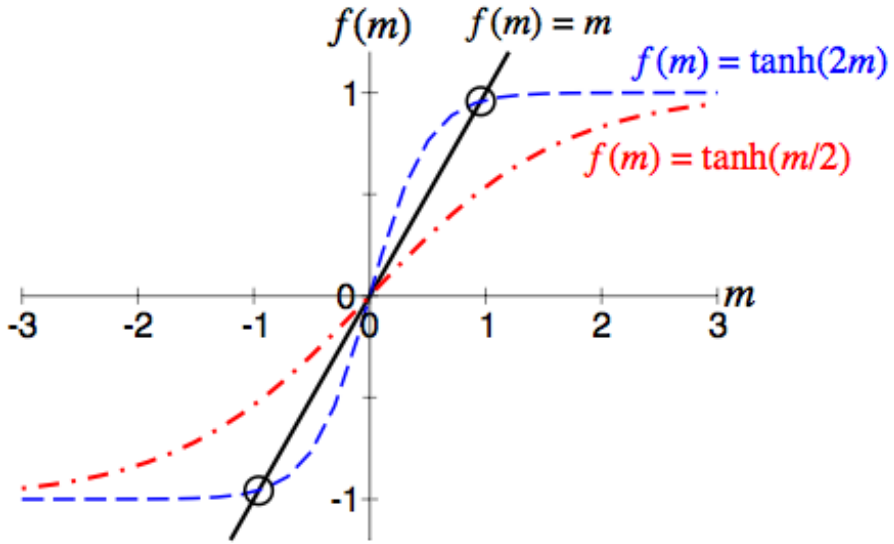
Mean-field theory

Define the average magnetisation as $m = \langle M \rangle = \langle s_i \rangle$.

Within mean-field theory we have
$m = \tanh\left(\frac{zJ}{k_B T} m\right)$.

One possible solution is $m = 0$.

However, there may also be other solutions, as we can see by plotting a graph…



For high temperatures (eg red curve), the only solution is $m = 0$.

For low temperatures (eg blue curve), there are 3 solutions.
The solutions with $m \neq 0$ represent ferromagnetic states.

Mean-field theory

Predictions of mean-field theory:

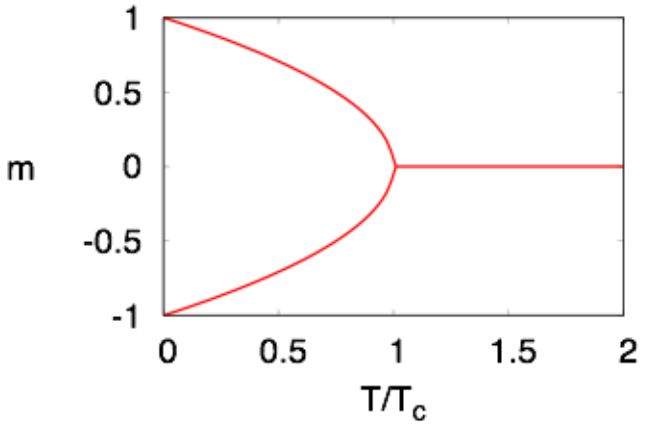
There is a critical temperature $T_c^{\mathrm{MF}} = zJ/k_{\mathrm{B}}$

For $T \geq T_c^{\mathrm{MF}}$ then $m = 0$.

For $T < T_c^{\mathrm{MF}}$ then there are two possible ferromagnetic states with equal and opposite values of m

As $T \to 0$ then $m \to \pm 1$.

If $(T_c^{\mathrm{MF}} - T)$ is small and positive then $m \approx \pm\sqrt{3(T_c^{\mathrm{MF}} - T)/T_c^{\mathrm{MF}}}$.



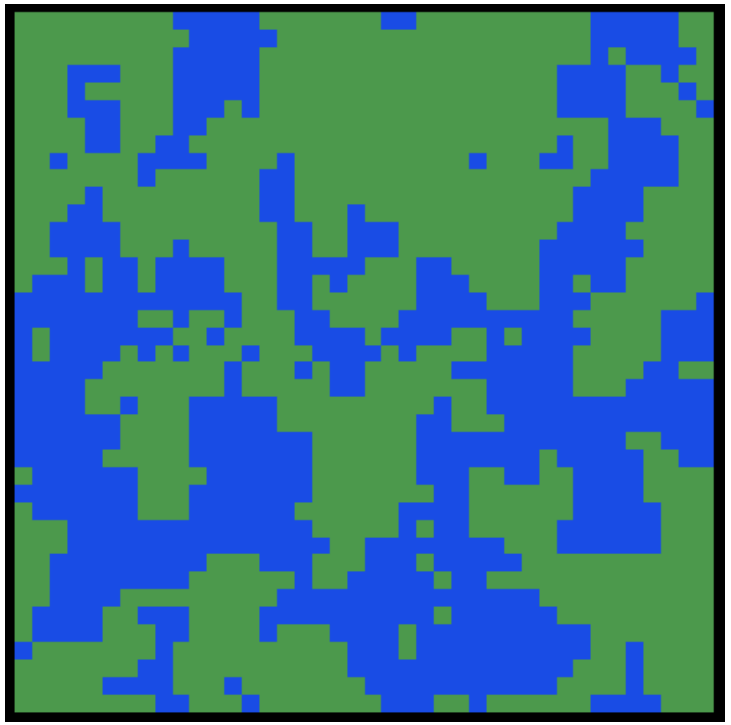
Note, this is all based on an approximation.

However, many of the qualitative features are correct.

Mean-field theory

The mean-field theory neglects correlations between spins. It assumes that all spins are situated in identical environments.

Monte Carlo simulations suggest that near the phase transition, magnetic spins aggregate into blobs of spins pointing in the same direction, while different blobs point in different ways.



In the coursework, one of the exercises is to study how well the mean-field theory and the Monte Carlo simulations match.

Especially interesting is what happens near the phase transition.

Dimensionless variables

In computational physics, we (nearly) always work with *dimensionless variables*.

Usually, all numbers within your program are dimensionless quantities.

> For example, in the DLA program the radius of the cluster might seem like it has units of length, but in fact it should be interpreted as the *ratio* of the size of a single particle and the size of the cluster.

Dimensionless quantities allow comparison between theory and experiment

The general rule is that if we calculate a dimensionless quantity, it can only depend on dimensionless parameters of the model (otherwise the units can’t balance).

Dimensionless variables

The critical temperature for mean-field theory is $T_c^{\mathrm{MF}} = zJ/k_{\mathrm{B}}$

Rearranging, we get $\frac{J}{k_{\mathrm{B}}T_c^{\mathrm{MF}}} = \frac{1}{z}$

Both sides of this equation are dimensionless

It is a *general* prediction
. . . it means that if we scale J by some number then T_c^{MF} just gets scaled by the same number

In fact, if we change J and T but always keep the same value of $\beta_0 = J/k_{\mathrm{B}}T$ then we know that the Boltzmann distribution stays the same, so the (average) magnetisation must be the same.

Moreoever, if we keep β_0 fixed then the whole Metropolis algorithm is independent of the specific values of J and T.
That is why the we have just one variable `inverseTemperatureBeta` instead of having separate variables for J and one for T.

Measuring responses

Suppose we want to apply a magnetic field h to our Ising model.

We expect the magnetisation $m = \langle M \rangle$ to depend on the magnetic field.

An interesting quantity to measure is the susceptibility $\chi = \frac{\partial m}{\partial h}$

. . . we could do this from the gradient of a plot of m against h

In fact, there is a nicer way:

We can prove that $\chi = \frac{N}{k_B T} \mathrm{Var}(M)$

where $\mathrm{Var}(M) = \langle M^2 \rangle - \langle M \rangle^2$ is the *variance* of the magnetisation

[see the coursework handout for the calculation]

We can measure this quantity without ever applying a magnetic field! This comes from properties of the Boltzmann distribution.

Mean-field data on Moodle

On Moodle, you are given dimensionless data

Temperature: T / T_c

Magnetisation: m

Energy: E/NJ

Heat capacity: c/k_{B}

Magnetic susceptibility: $\chi\, J$

For example,

$T/T_c = T\, k_{\mathrm{B}} / zJ$ where $z = 4$ for Mean-field

$T/T_c \approx T\, k_{\mathrm{B}} / (2.27\, J)$ for Onsager's solution exact solution (for infinite system size)

What is the value of T_c in the Monte Carlo simulations?

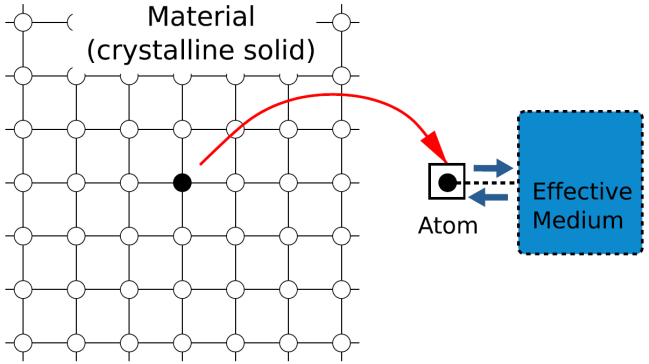
Summary

The Ising model is simple enough to do theoretical calculations and it exhibits a paramagnetic-ferromagnetic phase transition

The Ising model has many exact solutions in one dimension. In two-dimensions, the exact solution is difficult and it was first given by Lars Onsager. However, in three-dimensions there is no solution that is known.

Approximations such as the mean field theory can be used. Qualitatively correct, but often quantitatively wrong.

We can calculate responses (e.g. to magnetic fields) by calculating variances of observable quantities.



$$\chi = \frac{N}{k_{\mathrm{B}}T}\mathrm{Var}(M)$$

About the coursework

There are 5 exercises

- convergence to equilibrium
- measuring equilibrium averages
- comparison with mean-field theory
- specific heat capacity and magnetic susceptibility
- correlation function

Remember that you should write in such a way that another 3rd year student (who is not participating this course) can understand your report