

Further C++ Programming

David MacKay
University of Cambridge

January 16, 2008 – version 2008.1.15

This term's tasks are harder than last term's.
It's essential to prepare before each week's programming exercises. Read this manual and sketch out a plan of what you are going to do *before* sitting down at the computer.
The exercises involve physics, so you will need to prepare both physics thoughts and computing thoughts. **Please allow two hours preparation time per week.**

For the latest information about the IB course, please see:

- the course wiki (which requires Raven access):

<https://wiki.csx.cam.ac.uk/cphysics/>

– also reachable via <http://tinyurl.com/32tw9m>

- the course homepage:

<http://www.inference.phy.cam.ac.uk/teaching/comput/C++/>

– also reachable via <http://tinyurl.com/2uffap>

You are advised to attend the laboratory sessions with expert demonstrators, which will take place in the PWF at the Cavendish Laboratory at the following times in weeks 2, 3, and 4:

- | | | |
|----|--------------------|------------------------------|
| A: | Thursdays 11am–1pm | (starting Thursday 24th Jan) |
| B: | Thursdays 2–4pm | (Thursday 24th Jan) |
| C: | Fridays 2–4pm | (Friday 25th Jan) |
| D: | Tuesdays 11am–1pm | (Tuesday 29th Jan) |
| E: | Tuesdays 2–4pm | (Tuesday 29th Jan) |

Aims of the Lent Term Computing Course

Main aims:

- Structures, arrays, dynamic memory allocation.
- Putting useful functions into separate files.
- Use computing to help understand physics better.

Possible further aims that may be addressed in the lectures: recursion; classes; other programming languages; data modelling, likelihood functions.

Organisation

There are 3 two hour laboratory sessions scheduled for the Lent term computing course. You should work through the examples and exercises described in this document.

To benefit most from the laboratory session and from the demonstrators' help you must **read this document before the start of each session.**

This term's tasks are harder than last term's. The exercises involve physics, so you will need to prepare both physics thoughts and computing thoughts. **Please allow two hours preparation time per week.**

Assessment

In this course, your progress will be **self-assessed**.

The self-assessment process is based on a desire to treat you as adults, and on the assumption that you are interested in learning, capable of choosing goals for yourself, and capable of self-evaluation. I expect everyone to achieve full marks.

There are different types of programmers. Some people like to get into the nuts and bolts and write programs from scratch, understanding every detail – like building a house by first learning how to make bricks from clay and straw. Other people are happy to take bricks as a given, and get on with learning how to assemble bricks into different types of building. Other people are happy to start with an existing house and just make modifications, knocking through a wall here, and adding an extension there.

I don't mind what sort of programmer you become. All these different skills are useful. I encourage you to use this course to learn whatever skills interest you. This term's exercises give opportunities for various styles of activity.

As last term, we strongly encourage you to do the exercises at the recommended time, because that's when the expert demonstrators are there to help you. To allow for unexpected complications, the final deadline for assessment of each piece of work will be 9 days after the recommended completion time.

The main form of assessment for all this work will be self-assessment. Ask yourself "have I mastered this stuff?" If not, take appropriate action. Think, read, tinker, experiment, talk to demonstrators, talk to colleagues. Sort it out. Sort it out well before the final deadline.

When you have checked your own work and fully satisfied your self-assessment, you should submit an electronic record of your self-assessment to the course pigeonholes. By submitting your work, you are confirming that you have assessed yourself and achieved the session's objectives. Please do not abuse the electronic pigeonholes. I will make random checks, and all self-assessed submissions will be made available to the IB examiners.

In each session, set yourself one or two goals. When you have finished each session, submit a brief summary of your goals to the electronic course pigeonholes along with a file containing a record of your solution – typically the solution might take the form of one or two compilable C++ programs.

Collaboration

How you learn to program is up to you, but let me make a recommendation:

I recommend that you do most of your programming work in **pairs**, *with the weaker programmer doing the typing*, and the stronger one looking over his/her shoulder.

Working in pairs greatly reduces programming errors. Working in pairs means that there is always someone there to whom you can explain what you are doing. Explaining is a great way of enhancing understanding. It's crucial to have the weaker programmer do the typing, so that both people in the pair understand everything that's going on.

At the end of the day, you must all self-assess *individually*, and you must submit *individual* electronic records of your work to the electronic course pigeonholes.

Copying

When programming in real life, copying is strongly encouraged.

- Copying saves time;
- Copying avoids typing mistakes;
- Copying allows you to focus on your new programming challenges.

Similarly, in this course, copying may well be useful. For example, copy a working program similar to what you want to do; then modify it. Feel free to copy programs from the internet. The bottom line is: “do you understand how to solve this sort of programming problem?” Obviously, copying someone else's perfect answer verbatim does *not* achieve the aim of learning to program. Always self-assess. If you don't understand something you've copied, tinker with it until you do.

'Is this enough?'

If you find yourself asking “have I done enough for this week's session?”, the answer is “please self-assess!” Do you feel you've got a firm grasp of all the highlighted concepts? If so, then that's enough. One thing to check is “*have I developed my understanding of this week's physics topic?*” That's one of the aims this term – to get fresh insights into orbits, dynamics, waves, statistical physics, pressure, temperature, and so forth.

Submitting your self-assessed work

1. Last term we already made folders for everyone. If your user ID is jb007 you can check your folder is still there:

```
ls /ux/physics/part_1b/c++/pigeonhole/jb007
```

You may find it convenient to make a symbolic link (or 'shortcut') to your folder, like this for example

```
ln -s /ux/physics/part_1b/c++/pigeonhole/jb007 ~/submissions
```

2. Then, for each of the three pieces of work this term, make a folder like this:

```
mkdir /ux/physics/part_1b/c++/pigeonhole/jb007/IB.4
```

```
mkdir /ux/physics/part_1b/c++/pigeonhole/jb007/IB.5
```

```
mkdir /ux/physics/part_1b/c++/pigeonhole/jb007/IB.6
```

[or `mkdir ~/submissions/IB.4` if you made the symbolic link]. Put your work (the additional programming goal, and your solution) in that sub-folder: for example, if the work is in files called `c++/README` and `solution.cc`, copy them like this:

```
cp c++/README /ux/physics/part_1b/c++/pigeonhole/jb007/IB.4
```

```
cp solution.cc /ux/physics/part_1b/c++/pigeonhole/jb007/IB.4
```

Your marks for the course depend on your submission of your self-assessments to the course pigeonhole.

Deadline: You should do the first exercise during 'week 2' (between Thursday 24th and Wednesday 30th January). The **recommended completion time** for the first exercise is thus Wednesday 30th January 2007. The **final deadline** is **9 days later**, on Friday 1st January 2007 at 23:59.

Any problems, email `djcm1@cam.ac.uk`.

Feedback

Your feedback on all aspects of this new course is welcome. Feedback given *early* is more valuable than feedback delivered through the end-of-term questionnaires. Early feedback allows any problems arising to be fixed immediately rather than next year!

I'd be especially interested to know whether you feel it was a good idea to split this course into two halves, one in Michaelmas term and one in Lent.

Programming concepts

The new programming tools we'll use this term are arrays (great for representing vectors, or lists of similar things); structures (great for organizing things that belong together, such as a particle's position, mass, and velocity, or a vector and its range of indices); and how to modularize your code.

The next 11 pages cover the following topics.

arrays	how to allocate memory on the fly for things like vectors
modularizing	chopping up your code so you can reuse it elegantly
structures and packages	how to use structures in your elegant modules
formatted output	getting the number of decimal places that you want

The appendices include a guide to **Twenty-nine useful unix commands** (page 29).

Arrays

Last term, we declared arrays of fixed size like this

```
float      position[3];
int        count[100];
for(int i=0; i< 100; i++)
{
    count[i] = 0;
}
```

This fixed-size approach is an ugly way of doing things – it's inflexible, and it hard-wires numbers like '3' and '100' into your code, whereas such numbers should almost always be parameters. It would be more elegant to replace the '100' above by a name (such as *N*), and use that name everywhere, and fix the value of *N* just once at the start of the program. It's even more elegant to allow *N* to be set interactively or on the command-line of the program – in the way that `RandDemo6.cc`, for example, could optionally set the number of points *N* (page 34 of tutorial 1). The ugly way of doing arrays does not allow the creation of arrays of size controlled by a variable.

In C++ we can create variable-sized arrays on the fly while a program is running. We use a special command `new` to allocate the memory for an array, and a complementary command `delete` to free up the memory again when we don't need it any more.

The example program `NewDelete.cc` illustrates how to use `new` and `delete`, and how to pass an array to a function.

```

// NewDelete.cc
#include <iostream>
using namespace std;

void show( double *a , int N ){
    for( int n=0; n<N; n++ )
        cout << "a["<<n<<"]="\t"
            << a[n]
            << endl ;
}

int main()
{
    double *a ;
    // This creates a pointer but doesn't allocate any memory for the array

    int    N = 20 ;

    a = new double[N] ; // allocates memory for a[0]..a[N-1]
    for( int n=0; n<N; n++ )
        a[n] = n*n ;
    show( a , N ) ;
    delete [] a ;      // frees the memory
}

```

In C++, array indices by default run from zero. But if you want an array that runs from, say, 1 to N, then you can do this by creating an array `a[0]...a[N-1]` of the right size, then offsetting the pointer by 1 (using `b=a-1`), so that `b[1]...b[N]` points to the same locations as `a[0]...a[N-1]`. This convenient offsetting is demonstrated in the next example.

We can also allocate memory on the fly for more complex objects such as matrices. A good way to think of an $M \times N$ matrix with typical entry `q[m][n]` is that it consists of M vectors, each of length N . We can allocate memory for *a pointer (q) to an array of pointers* (`q[1]...q[M]`). Each of the pointers `q[1]...q[M]` is just like the pointer `a` in the previous example, pointing to its own personal chunk of memory of size N . This way of handling matrices is demonstrated in several examples on the website.

You can pass arrays to functions only by reference, not by value.

Modularizing

Last term we introduced the idea of chopping a program into small chunks called functions. It's good style to put each conceptual bit of the program in its own function. Try to avoid writing the same piece of code more than once.

In accordance with this principle of writing everything once only, it's also a good idea to put any function that you might want to use again into a file that other programs can make use of. There's a couple of ways to split programs over multiple files.

1. The simple way with `#include`. If you put the directive `#include "blah.cc"` on one line of a C++ program then the compiler will behave exactly as if the contents of the file

blah.cc were there, in your file, at that point. In this way you need to compile only one file – the other files get read in appropriately.

2. The professional way with *linking*. Alternatively, you can split the program into multiple files *each of which is compiled separately*. In this approach, the compiler needs to be run several times, once for each separate file, and then a final time to *link* the compiled files together. *Compiling* each individual .cc file creates a corresponding .o file. *Linking* takes all the .o files and combines them into a single executable. Just one of the .cc files should contain a definition of the main function, which is where the executable starts.

When compiling an individual file, the compiler doesn't need to know anything about the functions in the other files, except for the syntax of any functions that get used in the current file. That syntax is conveyed to the compiler by function declarations. The recommended technique to handle these function declarations is to ensure that all functions in the file blah.cc are *declared* in another file called blah.h, which is #included by blah.cc and by any other files that wish to use blah.cc's functions.

This 'linking' technique is illustrated by the four files that follow. The main program is in package1.cc. This program uses utility functions for memory allocation and array printing, located in the file tinyutils.cc. Both package1.cc and tinyutils.cc include the header file tinyutils.h. And finally, to keep track of what needs compiling when, it is essential to use a makefile. The makefile must contain an explicit statement that "package1 depends on package1.o and tinyutils.o", and (on the following line, *which must start with a single tab character*) an explicit instruction of how to link them together.

```
// tinyutils.h
//   declares the functions defined in tinyutils.cc

using namespace std;

// memory management
double *dvector ( int low, int high ) ;
void   freedvector ( double* a , int low ) ;

// printing
void printVector( double *b , int lo , int hi , int style=1 ) ;
// Note that any default parameter values (such as 'style=1')
// must be specified in the declaration.

// maths
double square( double a ) ;
```

```
# makefile for package1 and package2

CFLAGS = -ansi -g -Wall

LIBS = -l stdc++ -lm

CXX = g++

# These lines define what package1 depends on, and how to make it
package1: package1.o tinyutils.o
$(CXX) $(CFLAGS) $(LIBS) package1.o tinyutils.o -o package1
package2: package2.o tinyutils2.o
$(CXX) $(CFLAGS) $(LIBS) package2.o tinyutils2.o -o package2

%.o: %.cc
$(CXX) $(CFLAGS)      $< -c -o $@

%: %.cc makefile
$(CXX) $(CFLAGS) $(LIBS) $< -o $@
```



```

// package1.cc
// demonstrates how to use functions
// defined in a separately-compiled file (tinyutils.cc)

#include <iostream>
using namespace std;

// Both this file and tinyutils.cc include the
// function declarations from a single header file:
#include "tinyutils.h"

// In this example, we use functions 'dvector',
// 'square', 'printVector', and 'freedvector', all defined in
// tinyutils.cc

int main()
{
    double *b , *a ;
    int     N = 20 ;

    // allocate the space for b[1]..b[N]
    b = dvector( 1 , N ) ;
    a = dvector( 1 , N ) ;

    for ( int n = 1 ; n <= N ; n ++ )
        a[n] = static_cast<double>(n) ;
    for ( int m = 1 ; m <= N ; m ++ )
        b[m] = square( a[m] ) ;
    printVector( b , 1 , N ) ;

    // free the memory
    freedvector( b , 1 ) ;
    freedvector( a , 1 ) ;
    return 0;
}

```

```

// tinyutils.cc
// provides functions for double vectors allocation and clean-up
#include <iostream>
using namespace std;
#include "tinyutils.h"

// allocates memory for an array of doubles, say b[low]..b[high]
// example usage:  b = dvector( 1 , N ) ;
double *dvector ( int low, int high ) {
    int N = high-low+1 ;
    double *a ;
    if ( N <= 0 ) {
        cerr << "Illegal range in dvector: "
              << low << ", " << high << endl ;
        return 0 ; // returns zero on failure.
    }
    a = new double[N] ; // allocates memory for a[0]..a[N-1]
    if(!a) {
        cerr << "Memory allocation failed\n" ;
        return 0 ; // returns zero on failure.
    }
    return (a-low) ; // offset the pointer by low.
} // the user uses b[low]..b[high]

void freedvector ( double *b , int low ) {
    delete [] &(b[low]) ; // The '[]' indicate that what's
} // being freed is an array

// Note that default parameter values (such as 'style=0') have already
// been specified in the function declaration in tinyutils.h.
void printVector( double * b , int lo , int hi , int style ) {
    // style 1 means "all on one line"; style 0 means "in one column"
    for ( int n = lo ; n <= hi ; n ++ ) {
        cout << b[n] ;
        if(style) {
            if ( n == hi )    cout << endl;
            else               cout << "\t" ;
        } else {
            cout << endl;
        }
    }
}

double square( double x ) {
    return x*x ;
}

```

When we type `make package1`, the following things happen.

1. `make` looks at the makefile and learns that `package1` depends on `package1.o` and `tinyutils.o`.
2. If `package1.o` needs to be made, `make` invokes

```
g++ -ansi -pedantic -g -Wall package1.cc -c -o package1.o
```

At this stage, the compiler compiles just the functions that are defined in `package1.cc`.

3. Similarly for `tinyutils.o`, `make` invokes

```
g++ -ansi -pedantic -g -Wall tinyutils.cc -c -o tinyutils.o
```

4. For the final linking step, `make` invokes

```
g++ -ansi -pedantic -g -Wall -l stdc++ -lm package1.o tinyutils.o -o package1
```

yielding the executable `package1`. It's at this stage that the compiler will complain if any functions have been declared but not defined.

Structures and packages

We described, last term, how to use structures. Structures are a great way to organize things that belong together, and that should never really be separated from each other, such as a vector and its index range. By putting such things together into a single object, we can make code briefer (because we just refer to the one object, rather than its parts), and less buggy. The next example shows how to rewrite the previous example's vector-creation and vector-printing using a structure that contains the vector's pointer and its index range. The structure is defined in the header file. Why is it a good idea to use this structure? For this toy example, it doesn't seem like a big deal, but what you can notice is that function-calls that do things with the vector, once it's been created (such as `printDvector(b)`) are simpler and briefer, because we don't need to send along the baggage (`low, high`) that is required in the un-structured approach. The structure contains this baggage, so when we pass the pointer to the structure to other functions, those functions get access to exactly the baggage they need. The only down-side of this structured approach is that when we want to access the contents of the vector, we have to get the pointer to the vector out of the structure, so what used to be `a[n]` in the old approach (where `a` was the pointer to the array) becomes `a.v[n]` in the new approach (where `a` is the structure).

```

// package2.cc
// demonstrates how to use structures and functions
// defined in a separately-compiled file (tinyutils2.cc).
// The structure Dvector is defined in tinyutils2.h

#include <iostream>
using namespace std;

// Both this file and tinyutils2.cc include the
// function declarations from a single header file:
#include "tinyutils2.h"

int main()
{
    Dvector a , b ;
    int     N = 20 ;

    // allocate the space for b.v[1]..b.v[N]
    allocate( b , 1 , N ) ;
    allocate( a , 1 , N ) ;

    for ( int n = 1 ; n <= N ; n ++ )
        a.v[n] = static_cast<double>(n) ;
    for ( int m = 1 ; m <= N ; m ++ )
        b.v[m] = square( a.v[m] ) ;
    printDvector( b ) ;

    // free the memory
    freeDvector( b ) ;
    freeDvector( a ) ;
    return 0;
}

```

```

// tinyutils2.h
//   declares the structures and functions defined in tinyutils2.cc

using namespace std;

struct Dvector {
    double *v ; // the vector itself
    int     low;
    int     high;
} ; // don't forget the semicolon in the structure definition

// memory management
int     allocate( Dvector &a , int low, int high ) ;
void     freeDvector ( Dvector &a );

// printing
void printDvector( Dvector &b , int style=0 ) ;
// Default parameter values (such as 'style=0')
// must be specified in the declaration.

// maths
double square( double a ) ;

```

```

// tinyutils2.cc
// provides functions for double vectors allocation and clean-up
#include <iostream>
using namespace std;
#include "tinyutils2.h"

// allocates memory for an array of doubles. Example: allocate( b, 1 , N ) ;
int allocate ( Dvector &a , int low, int high ) {
    a.low = low ; a.high = high ;
    int N = high-low+1 ;
    if ( N <= 0 ) {
        cerr << "Illegal range in dvector: "
              << low << ", " << high << endl ;
        return 0 ; // returns zero if failure.
    }
    a.v = new double[N] ; // allocates memory for a[0]..a[N-1]
    if(!a.v) {
        cerr << "Memory allocation failed\n" ;
        return 0 ;
    } else {
        a.v -= low ; // offset the pointer by low.
        return 1 ;
    }
}

void freeDvector ( Dvector &b ) {
    delete [] &(b.v[b.low]) ;
    b.high = b.low - 1 ;
}

// Note that default parameter values (such as 'style=0') have already
// been specified in the function declaration in tinyutils2.h.
void printDvector( Dvector &b , int style ) {
    // style 1 means "all on one line"; style 0 means "in one column"
    for ( int n = b.low ; n <= b.high ; n ++ ) {
        cout << b.v[n] ;
        if(style) {
            if ( n == b.high )      cout << endl;
            else                    cout << "\t" ;
        } else {
            cout << endl;
        }
    }
}

double square( double x ) {
    return x*x ;
}

```

Formatted output

We've mainly printed out numbers using `cout` with commands like

```
cout << myint << " " << mydouble << endl ;
```

What if you don't like the way `cout` makes your numbers look, however? Too many decimal places? Too few? Well, `cout` can be bossed around and can be told to serve up numbers with different numbers of decimal places. You can find out more about `cout` by looking in a manual online or on paper.

Here we'll describe another way to control output formatting, using the old-fashioned C function, `printf` (which means print, formatted). It's probably worth learning a bit about `printf` because its syntax is used in quite a few languages.

The commands

```
int age = 84; printf( "his age is %d years" , age ) ;
```

will print the string 'his age is 84 years'. The command

```
printf( "%d %d\n" , i , j ) ;
```

causes the integers `i` and `j` to be printed, separated by a single space, and followed by a newline (just like `cout << i << " " << j << endl ;`). The command

```
printf( "%3d %-3d\n" , i , j ) ;
```

causes the same numbers to be printed out but encourages `i` to take up 3 columns, and to be right-aligned within those 3 columns; and encourages `j` to take up 3 columns and to be left-aligned. Text and numbers can be mixed up. For example,

```
printf( "from %3d to %-3d\n" , i , j ) ;
```

might be rendered as

```
from 123 to 789
```

Notice how you specify the format of the whole string first, then the missing bits, each of which was indicated in the format string by a specifier, *%something*. The format specifier for a single character is `%c`; for a string of characters, `%s`. The special character `\n` is a newline; `\t` is a tab; `\"` gives a double quote; `\\` gives a single backslash. The command

```
printf( "%f %e %g \n" , x , y , z ) ;
```

prints the floating point numbers `x`, `y`, and `z` as follows: `x` is printed as a floating point number with a default number of digits (six) shown after the decimal point; `y` is printed in scientific (exponential) notation; `z` is printed using whichever makes more sense, fixed floating point or scientific notation. The program `Printf.cc` on the website illustrates more examples. I usually use the format

```
printf( "%12.6g %12.6g %12.6g" , x , y , z ) ;
```

to print my real numbers – this format gives each of them 12 columns, and shows 6 digits of precision.

Physics objectives: to better understand Newton's laws, circular motion, angular momentum, planets' orbits, Rutherford scattering, and questions about spacemen throwing tools while floating near space shuttles.

Computing objectives: structures, arrays, using gnuplot; simulation methods for differential equations (Euler, leapfrog).

You are encouraged to work in pairs, with the weaker programmer doing all the typing.

Task

Simulate Newton's laws for a small planet moving near a massive sun. For ease of plotting, assume the planet and its velocity both lie in a two-dimensional plane. Put the sun at the origin $(0,0)$ and let the planet have mass m and initial location $\mathbf{x}^{(0)} = (x_1, x_2)$ and initial velocity $\mathbf{v}^{(0)} = (v_1, v_2)$. The equations of motion are:

$$\begin{aligned}\frac{d\mathbf{x}}{dt} &= \mathbf{v}(t) \\ m\frac{d\mathbf{v}}{dt} &= \mathbf{f}(\mathbf{x}, t),\end{aligned}\tag{1}$$

where, for a standard inverse-square law (gravity or electrodynamics) the force \mathbf{f} is:

$$\mathbf{f}(\mathbf{x}, t) = A \frac{\mathbf{x}}{\left(\sqrt{\sum_i x_i^2}\right)^3},\tag{2}$$

with $A = -GMm$ for gravitation, and $A = Qq/(4\pi\epsilon_0)$ for electrodynamics with two charges Q and q .

Try to write your programs in such a way that it'll be easy to switch the force law from inverse-square to other force laws. For example, Hooke's law says:

$$\mathbf{f}(\mathbf{x}, t) = k\mathbf{x}.\tag{3}$$

How should we simulate Newton's laws (1) on a computer? One simple method called Euler's method makes *small, simultaneous* steps in \mathbf{x} and \mathbf{v} . We repeat lots of times the following block:

- EULER'S METHOD
- 1: Find $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$
 - 2: Increment \mathbf{x} by $\delta t \times \mathbf{v}$
 - 3: Increment \mathbf{v} by $\delta t \times \frac{1}{m}\mathbf{f}$
 - 4: Increment t by δt

We might hope that, for sufficiently small δt , the resulting state sequence (\mathbf{x}, \mathbf{v}) in the computer would be close to the true solution of the differential equations.

Euler's method is not the only way to approximate the equations of motion.

An equally simple algorithm can be obtained by reordering the updates. In the following block, the updates of \mathbf{f} and \mathbf{x} have been exchanged, so the force is evaluated at the new location, rather than the old.

- A: Increment \mathbf{x} by $\delta t \times \mathbf{v}$
- B: Find $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$
- 3: Increment \mathbf{v} by $\delta t \times \frac{1}{m}\mathbf{f}$
- 4: Increment t by δt

It's not at all obvious that this minor change should make much difference, but often it does. This second method, in which position and velocity are updated alternately, is called the Leapfrog method. Here is a precise statement of the leapfrog method.

```

LEAPFROG METHOD
Repeat
{
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
    Find  $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$ 
    Increment  $\mathbf{v}$  by  $\delta t \times \frac{1}{m}\mathbf{f}$ 
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
}

```

In this version, we make a half-step of \mathbf{x} , a full-step of \mathbf{v} , and a half-step of \mathbf{x} . Since the end of every iteration except the last is followed by the beginning of the next iteration, this algorithm with its half-steps is identical to the previous version, except at the first and last iterations.

When simulating a system like this, there are some obvious quantities you should look at: the angular momentum, the kinetic energy, the potential energy, and the total energy.

Techniques to use:

1. Represent the vectors \mathbf{x} , \mathbf{v} , and \mathbf{f} using arrays. Arrays were mentioned at the end of the first term's tutorial, but we didn't use them in any exercises. Make sure you look at some simple examples of arrays before using them for planets.
2. Put all the variables and arrays associated with a single object (such as a planet) in a structure. Structures are a really useful programming tool; they were explained in the first term's tutorial, but we didn't use them in any exercises. Make sure you look at some simple examples of structures.

What to do

There's a lot of choice. This is a self-directed and self-assessed course, and I'd like you to choose a planet-simulating activity that interests you.

Here are some suggestions. **You don't have to do all of these.** The more you do, the more educational it'll be. But do take your pick, and feel free to steal working code (e.g. from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

1. Write code that implements Euler's method and the Leapfrog method. Get it going for one initial condition, such as $\mathbf{x}^{(0)} = (1.5, 2)$, $\mathbf{v}^{(0)} = (0.4, 0)$. Before running your program, predict the correct motion, roughly. Compare the two methods, using `gnuplot` to look at the resulting trajectories, the energies, and the angular momenta. (If you'd like more detailed guidance through a possible approach to this task, **see the appendix on p. 27**. A worked solution for this first part is also available.)

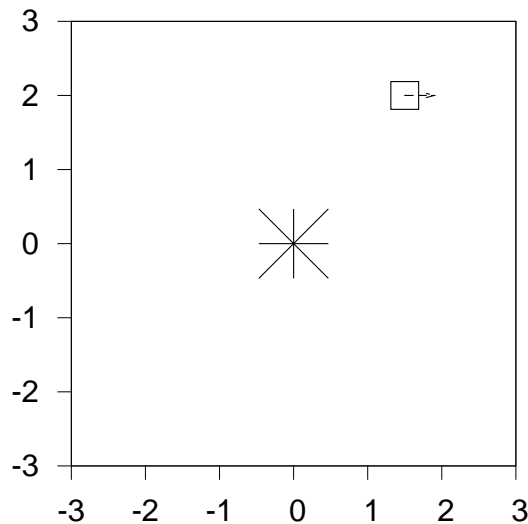


Figure 1: The initial condition $\mathbf{x}^{(0)} = (1.5, 2)$, $\mathbf{v}^{(0)} = (0.4, 0)$.

2. Once you have a trustworthy simulation: take a collection of similar initial conditions, all having the same initial position and initial energy but differing in the direction of the initial velocity. What happens? Show all the evolving trajectories in a single movie.
3. Or take initial conditions that differ slightly in their initial position or velocity, such as “the spaceshuttle and the spaceman who is 100 yards from the spaceshuttle, both orbiting the earth” (in what direction does he need to nudge himself in order to get home?). Or “the spaceman and the hammer” – if he throws his hammer ‘forwards’, where does it end up going? If he throws it ‘back’ or ‘sideways’, where does it end up going? (Here the idea is to simulate two planets orbiting a single sun; to get the spaceman analogy, think of the spaceman and his shuttle being like the two planets, and the earth playing the role of the sun.)
4. Get an initial condition that leads to a perfectly circular orbit. Now perturb that initial condition by giving a little kick of momentum in 8 different directions. Show all nine resulting trajectories in a single movie. Which kicks lead to the period of the orbit changing? Which kicks lead to the period of the orbit staying the same? Which kicks lead to orbits that come back to the place where the kick occurred? If an object is in an elliptical orbit, what kicks do you give it in order to make the orbit larger and circular? What kicks do you give it to make the orbit smaller and circular? What’s the best time, in an elliptical orbit, to give the particle a kick so as to get the particle onto an orbit that never comes back, assuming we want the momentum in the kick to be as small as possible?
5. Take initial conditions coming in from a long way away, particles travelling along equally-spaced parallel lines. What happens? (The distance of the initial line from the sun is called the impact parameter of the initial condition.)
6. People who criticise Euler’s method often recommend the *Runge–Kutta method*. Find out what Runge–Kutta is (Google knows), implement it, and compare with the leapfrog method. Choose a big enough step-size δt so that you can see a difference between the methods. Given equal numbers of computational operations, does Runge–Kutta or leapfrog do a better job of making an accurate trajectory? For very long runs, and again assuming equal numbers of computational operations, does Runge–Kutta or leapfrog do a better job of energy conservation? Of angular momentum conservation?

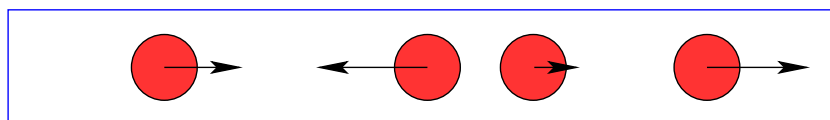
Physics objectives: to better understand collisions, conservation laws, and statistical physics, especially equipartition, properties of ideal gases, the concept of temperature, and the Boltzmann distribution; also the way in which microscopic physics leads to macroscopic phenomena; what happens when a piston compresses an ideal gas; adiabatic expansion; fluctuations and dissipation, equilibration of systems with different temperatures.

Computing objectives: structures, arrays, using gnuplot; memory allocation.

You are encouraged to work in pairs, with the weaker programmer doing all the typing.

Task

We're going to simulate hard spheres colliding with each other elastically in a box. An incredible range of interesting physics phenomena can be studied in this way.



The heart of this computing exercise is going to be a single function – let's call it `collide` – which receives two particles with masses m_1 and m_2 and velocities u_1 and u_2 , and returns the new velocities v_1 and v_2 of the two particles after an elastic collision between the two.

You could start by writing a function that implements this elastic collision. Check your function with special cases that you've solved by hand, and by putting in a range of initial conditions and evaluating whether total momentum and total energy are indeed conserved as they should be.

An elegant approach to this programming task uses a structure to represent each particle – something like this, for a particle moving in one dimension:

```
struct particle {  
    double x      ; // position  
    double p      ; // momentum  
    double im     ; // inverse mass  
    double v      ; // velocity  
    double T      ; // kinetic energy  
    double a      ; // radius of particle  
} ; // Note the definition of a structure ends with a semicolon
```

At this stage it's not crucial, but at some point I recommend making sure all references to the particle's mass use the *inverse mass*, rather than the mass – this allows you to treat the walls of the box as standard particles that just happen to have infinite mass (that is, inverse-mass zero).

Once you have defined a struct like `particle`, you may define an array of particles in just the same way that you define arrays of other objects like ints or doubles. For example

```
a = new particle[N] ;
```

What to do next

There's a lot of choice. This is a self-directed and self-assessed course, and I'd like you to choose a bonking-simulating activity that interests you.

Here are some suggestions. **You don't have to do all of these.** The more you do, the more educational it'll be. But do take your pick, and feel free to steal working code (e.g. from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

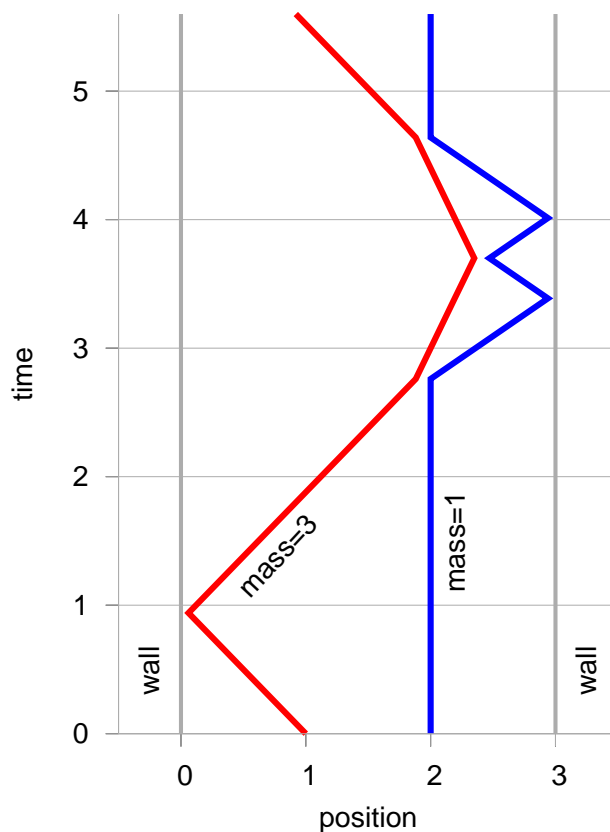
1. Write code that uses a one-dimensional `collide` function to simulate the motion of N particles in a one-dimensional box. Each particle can collide only with its immediate neighbours; each of the two end particles can collide with one wall too. To simulate the dynamics, you must identify the times at which collisions occur, and (assuming you want to make a realistic movie for showing in gnuplot) spit out the state of the simulation *at equally spaced times*. A suggested strategy is to take the current state and figure out which pair of adjacent particles will collide *next*. Then advance the simulation exactly to the moment of that next collision (stopping if appropriate at intermediate times along the way, so as to spit out the required states for the movie, equally spaced in time). Collisions should be handled by passing the pair of particles to the `collide` function. Motion in between collisions is simple (since there are no forces) and it should be handled by another function, `leapForward`, say. Printing out of the state at certain times of interest should be handled by another function, `showState`, say. (A worked solution for this first part is available on the course website.)

[A possible difficulty with this approach of computing all collisions that occur is that it is conceivable that the true number of collisions in a finite time might be very large, a phenomenon known as chattering. You can get the idea of chattering by imagining quickly squashing a moving ping-pong ball between a table-tennis bat and a table.]

2. Testing: Put just two particles in a box, with equal masses. Check that the dynamics are right. Make the two masses unequal. Make a scatter-plot of the positions of the two particles. Make a scatter-plot of the velocities of the two particles. Use more masses. Check that kinetic energy is conserved.
3. Put quite a few *unequal* masses in the box (say, 10 masses, with a variety of masses spanning a range of a factor of 4 in magnitude), run the simulation for a long time, and make histograms of the velocities of two of the particles whose masses are in the ratio 4:1. What do you find? Make histograms of the *positions* of the particles. If you make some of the particles *really heavy* compared to their neighbours, what happens to the histogram of the positions of the neighbours? For example, make all the particles except for the two end particles be much larger; or make half the particles (those on the left hand side) heavy, and the other half light. (In all simulations make sure no two adjacent particles have identical masses.)
4. What happens if the right-hand wall (with infinite mass) is moved at constant speed towards or away from the other wall?

Ye have heard it said that, under some circumstances, $pV^\gamma = \text{constant}$. What is γ for a *one-dimensional* ideal gas? How should the total energy vary with V under these conditions?

5. Set up N_1 light masses to the left of a single big heavy mass, and N_2 more light masses to the right of the heavy mass. Call the heavy mass a piston, if you like, and think of it as separating two ideal gases from each other. The light masses don't need to be identical to each other. An example set of masses for $N_1 = N_2 = 5$ could be (1.1, 1.2, 1.1, 1.3, 1.1, 100.0, 4.1, 4.2, 4.1, 4.8, 4.4) where the 100-mass is the piston. Give randomly chosen velocities to the particles. What should happen? How long does it take for 'equilibrium' to be reached? Give an enormous velocity to the piston and small velocities to the other particles. What should happen? How long does it take for 'equilibrium' to be reached? Can you get the piston to oscillate roughly sinusoidally (before 'equilibrium' is reached)? What is the frequency of such oscillations? Predict the frequency using the theory of adiabatic expansion/compression of gases.



The first six collisions between two particles of masses 3 and 1 and two walls.

Part IB Computing Course Session 6: TSUNAMIS

Physics objectives: to better understand waves, superposition, linearity, dispersion relations and dispersion, wavepackets.

Computing objectives: structures, arrays, matrices (handled using arrays of pointers), using gnuplot; memory allocation

One of the key ideas of physics is *superposition*. If a system is *linear* then a solution for its motion can be represented by the superposition of a linear combination of simple solutions of its motion. This idea is the central idea of normal modes, of waves, and of quantum physics.

What happens when a pebble is thrown in a pond?

The idea of this exercise is to simulate the motion of a water surface corresponding to a stretch of shallow water, or of deep water, or of water with surface tension, for any initial condition; or to simulate a homogeneous medium with almost any properties you like. We'll assume periodic (wrap-around) boundary conditions. One simple motion that such a surface can always make is a standing wave, where the surface is at all times a perfect sinusoid with wavelength $\lambda = L/m$ where L is the box length and m is a positive integer. The amplitude of the standing wave varies sinusoidally in time. The frequency ω of the temporal oscillation depends on the wavenumber $k \equiv 2\pi/\lambda$ in a way that depends on the springiness properties and inertial properties of the medium. The function $\omega(k)$ is called the dispersion relation.

Now, the astonishing assertion of superposition is that the complex motion resulting from any initial condition (for example a little cluster of ripples running away from the point of impact of a pebble) can be represented by a superposition of the simple standing waves.

This astonishing idea can be fruitfully explored on the computer. Take care not to use too much memory, nor to write too-big files to disc. A 1000×1000 matrix of doubles takes about 8 megabytes of memory in a program. Written to file, a million doubles might make a file of size 12 megabytes or so. As a rule of thumb, you usually don't want your program to use more than about 64 megabytes of memory, and it's probably good to keep most file sizes below a megabyte or so. You can use bigger things if you want, but performance may get sluggish.

If writing lots of files, or big files, for plotting with gnuplot, it may be a good idea to write them to the `/tmp` filesystem, which is a local filesystem on your machine. Files in `/tmp` are usually deleted when the computer reboots, so copy any important outputs into your normal filesystem. You can find the size of the `/tmp` file system with the unix command `df /tmp` (`df` means 'disc free'). *Don't ever expect to find your files in /tmp next time you log in.*

You are encouraged to work in pairs, with the weaker programmer doing all the typing. As usual, you don't have to do everything listed here.

1. Make a program that represents a surface by the displacements y_n of N equally-spaced points, and that creates an $M \times N$ matrix containing M sinusoidal functions representing M standing waves. (As you can confirm, the biggest possible value of M is N or $N - 1$, depending whether you allow the zero-wavenumber sinusoid. If you try to make more

sinusoids than this, you'll find the extra ones are identical to ones you already made.) Make sure the sinusoidal functions are orthonormal. Confirm (for small N) that you can decompose any displacement y into a linear combination of N sinusoids. Pick a dispersion relation – for example $\omega = ck$ (shallow water, or slinky) or $\omega \propto \sqrt{k}$ (infinitely deep water). Take y as an initial condition and evolve it forward in time by imagining that each sinusoid oscillates at its own frequency (as defined by the dispersion relation), and re-superposing the sinusoids. Make an initial condition that looks like a small bump. What happens? What happens for each dispersion relation? So far the initial conditions have been static displacements, y ; include an initial velocity v too. (A worked solution for this first part is available on the course website.)

2. Simulate a tsunami (caused by a bit of earth moving underneath the sea, or by a meteor hitting the ocean). Describe what happens from the point of view of someone in a boat a long way from the initial disturbance. Compare your description with first-hand accounts of recent tsunamis. Use the dispersion relation for deep, but not infinitely deep water. (A depth of 5 km might be reasonable.)

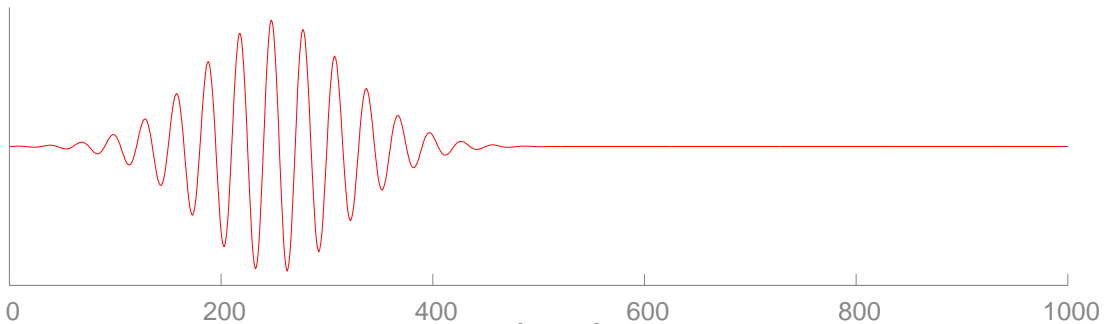


Figure 2: The function $\sin(kx) \exp(-(x - x_0)^2 / (2r^2))$, with $k = 2\pi/30$, $r = 70$, and $x_0 = 250$.

3. Simulate a wavepacket – a disturbance of finite extent where the surface looks locally approximately sinusoidal. What happens for each dispersion relation – for example $\omega = ck$ (shallow water) or $\omega \propto \sqrt{k}$ (infinitely deep water)? Look in detail at the evolution of the wavepacket for $\omega \propto \sqrt{k}$. Do the crests move at the same speed as the wavepacket? Does the wavepacket's width remain constant? Look carefully at the spatial frequency of the parts of the wave at the front and rear of the wavepacket – is the spatial frequency the same at both ends?

Part IB Computing Course Session 7: RECURSION

This is an optional extra.

Physics objectives: to better understand statistical physics by counting some interesting things.

Computing objectives: recursion.

Recursion means defining a function f in terms of the function f . For example we could define the factorial function $f(x)$ by:

1. if $x > 1$ then return $x \times f(x - 1)$
2. otherwise return 1.

A recursive function in a computer program is one that calls itself. Here's an example, following the above definition closely.

```
// factorial calculator - recursive
#include <iostream>
using namespace std;

int factorial (int a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}

int main ()
{
    int number;
    cout << "Please type a number: ";
    cin >> number;
    cout << number << "! = " << factorial(number) << endl;
    return 0;
}
```

Errors in the definition of recursive functions often lead to disaster, since it's all too easy to write a function that keeps calling itself and never stops.

Some programmers find recursion an elegant way to express many programming tasks. Here is an example. The task is 'print all ternary strings of length L '. For $L = 1$ the output should be

0,1,2.

For $L = 2$ the output should be

00,01,02,10,11,12,20,21,22

Here is a solution.


```

// allStrings.cc
// Enumerate all ternary strings by recursion

#include <iostream>
using namespace std;

void appendAllStrings( char *prefix , int remainingLength )
{
    if (remainingLength == 0)
        cout << prefix << endl ;
    else {
        int lp = strlen(prefix) ;
        for ( int i = 0 ; i <= 2 ; i ++ ) { // Extend prefix by one character.
            prefix[lp] = i+'0' ;           // By adding i to the character '0'
                                           // we get the characters '0', '1', '2'
            appendAllStrings( prefix , remainingLength-1 );
        }
        prefix[lp] = '\0' ;               // Remove what was added
                                           // [ '\0' is the null character ]
    }
    return ;
}

int main ()
{
    int length;
    cout << "Please type a length: ";
    cin >> length;
    char *prefix ;
    prefix = new char[length] ;           // Assume this memory is all-null
    appendAllStrings( prefix , length ) ;
    delete [] prefix ;                   // Free the memory
    return 0;
}

```

The function `strlen` returns the length of the string generated so far; the line

```
prefix[lp] = ...
```

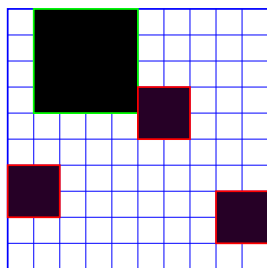
extends the string by one character. The algorithm proceeds by starting from the null string, and extending it by all possible single characters, extending each in turn by all possible characters, and so forth.

Such 'branching processes' are the type of problem for which recursion is especially recommended.

Recursion exercises

Solve each of these tasks using a recursive function.

1. Write a function called `twos` that takes a single integer as its argument and returns the number of factors of 2 in the number. (Hint: odd numbers have no factors of 2, numbers that are twice an odd number have one, numbers that are four times an odd have two, and so on.) For example: `twos(-12)`; should return 2.
2. Write a function called `printWithCommas` that takes a single nonnegative long integer argument and displays it with commas inserted in the conventional way. For example:
 - `printWithCommas(12045670)`; displays 12,045,670.
 - `printWithCommas(1)`; displays 1.
3. INTERESTING PROPERTIES OF A HARD SPHERE GAS.



A legal state of four particles in a 10×10 box.

A square box of size $H \times H$ lattice-points contains one big particle and $T = 3$ little particles. The big particle occupies 4×4 lattice points. Each little particle occupies 2×2 lattice points. Particles may not overlap. All legal states of this $T + 1$ -particle system are equiprobable. How probable are the alternative locations of the big particle? Find the answer for $H = 10$ (by recursively enumerating all legal states, and keeping count). It is a good idea to spit out the answer for smaller values of T along the way towards the answer for the biggest value of T .

If not all locations of the big particle are equiprobable then we can describe the effect of the little particles in terms of an effective ‘force’ acting on the big particle. Such forces are called ‘entropic forces’. Entropic forces in hard-sphere mixtures with a variety of sizes of spheres are an area of industrial research interest.¹

¹Y. Mao, P. Bladon, H. N. W. Lekkerkerker, and M. E. Cates. *Mol. Phys.* **92**, 151 (1997).

R. Dickman, P. Attard, and V. Simonian. ‘Entropic Forces in Binary Hard-Sphere Mixtures: Theory and Simulation’, *J. Chem. Phys.* **107**, 205–213 (1997).

‘Entropic Attraction and Repulsion in Binary Colloids Probed with a Line Optical Tweezer’ J. C. Crocker, J. A. Matteo, A. D. Dinsmore, and A. G. Yodh. *Physical Review Letters*, **82**, 4352–4355 (1999).

Appendices

PLANET: step by step guidance

1. Chop the problem into small pieces.
2. Define a structure that contains the state of the planet. (See page 11 of this tutorial and page 60 of the first term's tutorial.) Your structure could be as simple as:

```
struct particle {  
    double x[2] ; // (x,y) coordinates  
    double v[2] ; // velocity  
} ;
```

but as you continue, you will probably think of other sensible things to add to the structure. You might find it elegant to give a name to the dimensionality of the position space, say D.

```
#define D 2 // number of dimensions  
struct particle {  
    double x[D] ; // (x,y) coordinates  
    double v[D] ; // velocity  
} ; // Note the definition of a structure ends with a semicolon
```

If you then use D everywhere, it makes the meaning of your code clearer (compared with using '2'), and it makes it easier to update your simulation's dimension (say from 2 to 3 dimensions).

3. Write a simple main that defines a particle, and check that it compiles and runs.

```
int main()  
{  
    particle a ;  
  
    a.v[0] = 0.4;  
    a.v[1] = 0.0;  
    a.x[0] = 1.5;  
    a.x[1] = 2;  
  
    return 0;  
}
```

4. Write a function `showState` that prints out the particle's position and velocity. Call it from `main` and check that your program compiles and runs correctly. Here is an example of a function that does the right sort of thing:

```
void showState ( particle &a )
{
    int i=0;
    cout << "some of the state: " << a.x[i] << endl ;
}
```


This could be called by `main` using a command such as

```
showState( a ) ;
```

Notice the use of the ampersand in the function definition

```
void showState ( particle &a ).
```

This ampersand means that we are passing the particle by reference, rather than by value. (See page 45 of last term's tutorial.)

5. Remember the tips from last term about debugging your code. (1) Use a `makefile`. (2) Switch on all the compiler warnings. (3) Whenever you think that your code ought to compile, check that it does so. (4) If you have code that compiles ok, but that doesn't behave as expected when you run it, *run it in a debugger* (for example, `kdbg`). Even if it seems to be running as expected, it might be a good idea to run it in a debugger. When I use `kdbg`, the main things I click on are 'set breakpoint' (right click), 'run', and the 'step over'  icon.
6. Write a function that computes the squared-distance from the origin.
7. Write a function `Force` that computes the force acting on the particle, or the acceleration of the particle. (Perhaps the force or acceleration should be in your structure?)
8. Write a function `PositionStep` that makes a change of a particle's position in the direction of its velocity. One of the arguments of this function should be the required time step size.
9. Write a function `VelocityStep` that makes a change of a particle's velocity in the direction of its acceleration.
10. Write a function that computes the energy (kinetic and potential) of the particle.
11. Write a function that computes the angular momentum about the origin.
12. Write a leapfrog simulator with a loop that calls the above functions appropriately.
13. Write the simulated trajectory to a file using a function like `showState`. To plot the trajectory from a file whose columns are

time, x_1 , x_2 , v_1 , v_2 ,

the following gnuplot commands may be useful.

```
set size ratio -1 ; ## this makes units on the x and y axes have equal size
plot 'tmp' u 2:3 w l, 'tmp' every 10 u 2:3:4:5 w vector
## this plots (x1,x2) with lines, and plots every 10th state using a vector
## of length (v1,v2) based at the point (x1,x2)
```

gnuplot's plot command is very versatile. The website shows how you can make a gnuplot animation using a single data file containing a trajectory.

14. Be aware of the size of the files you are writing. If you give gnuplot a file of more than 1 megabyte, you should expect it to be sluggish. Maybe it would be a good idea to reduce the amount of information written to file. You may be able to get better performance by using the local filesystem of the machine at which you are sitting. You can write anything you want into the folder /tmp. For example,

```
ls > /tmp/mylist
```

The /tmp folder is a good place to put anything large and anything that you want to access frequently or quickly. But don't leave anything important in /tmp – after you log out, the /tmp folder may be cleaned up.

Twenty-nine useful unix commands

This guide, based on a University computing service leaflet, is intended to be as generally valid as possible, but there are many different versions of Unix available within the University, so if you find a command option behaving differently on your local machine you should consult the on-line manual page for that command. Some commands have numerous options and there is not enough space to detail them all here, so for fuller information on these commands use the relevant on-line manual page.

The names of commands are printed in **bold**, and the names of objects operated on by these commands (e.g. files, directories) are printed in *italics*.

Twenty-nine useful unix commands – index

1. **cat** - display or concatenate files
2. **cd** - change directory
3. **chmod** - change the permissions on a file or directory
4. **cp** - copy a file
5. **date** - display the current date and time
6. **diff** - display differences between text files
7. **file** - determine the type of a file
8. **find** - find files of a specified name or type
9. **ftp** - file transfer program
10. **grep** - searches files for a specified string or expression
11. **gzip** - compress a file
12. **help** - display information about bash builtin commands
13. **info** - read online documentation
14. **kill** - kill a process
15. **lpr** - print out a file
16. **ls** - list names of files in a directory
17. **man** - display an on-line manual page
18. **mkdir** - make a directory
19. **more** - scan through a text file page by page
20. **mv** - move or rename files or directories
21. **nice** - change the priority at which a job is being run
22. **passwd** - change your password
23. **ps** - list processes
24. **pwd** - display the name of your current directory
25. **quota** - disk quota and usage
26. **rm** - remove files or directories
27. **rmdir** - remove a directory
28. **sort** - sort and collate lines
29. **ssh** - secure remote login program

1. **cat** – display or concatenate files

cat takes a copy of a file and sends it to the standard output (i.e. to be displayed on your terminal, unless redirected elsewhere), so it is generally used either to read files, or to string together copies of several files, writing the output to a new file.

cat *ex*

displays the contents of the file *ex*.

cat *ex1 ex2 > newex*

creates a new file *newex* containing copies of *ex1* and *ex2*, with the contents of *ex2* following the contents of *ex1*.

2. **cd** – change directory

cd is used to change from one directory to another.

cd *dir1*

changes directory so that *dir1* is your new current directory. *dir1* may be either the full pathname of the directory, or its pathname relative to the current directory.

cd

changes directory to your home directory.

cd ..

moves to the parent directory of your current directory.

3. **chmod** – change the permissions on a file or directory

chmod alters the permissions on files and directories using either symbolic or octal numeric codes. The symbolic codes are given here:-

u	user	+	to add a permission	r	read
g	group	-	to remove a permission	w	write
o	other	=	to assign a permission explicitly	x	execute (for files), access (for directories)

The following examples illustrate how these codes are used.

chmod u=rw *file1*

sets the permissions on the file *file1* to give the user read and write permission on *file1*. No other permissions are altered.

chmod u+x,g+w,o-r *file2*

alters the permissions on the file *file2* to give the user execute permission on *file2*, to give members of the user's group write permission on the file, and prevent any users not in this group from reading it.

chmod u+w,go-x *dir1*

gives the user write permission in the directory *dir1*, and prevents all other users having access to that directory (by using **cd**. They can still list its contents using **ls**.)

chmod g+s *dir2*

means that files and subdirectories in *dir2* are created with the group-ID of the parent directory, not that of the current process.

4. **cp** – copy a file

The command **cp** is used to make copies of files and directories.

cp file1 file2

copies the contents of the file *file1* into a new file called *file2*. **cp** cannot copy a file onto itself.

cp file3 file4 dir1

creates copies of *file3* and *file4* (with the same names), within the directory *dir1*. *dir1* must already exist for the copying to succeed.

cp -r dir2 dir3

recursively copies the directory *dir2*, together with its contents and subdirectories, to the directory *dir3*. If *dir3* does not already exist, it is created by **cp**, and the contents and subdirectories of *dir2* are recreated within it. If *dir3* does exist, a subdirectory called *dir2* is created within it, containing a copy of all the contents of the original *dir2*.

5. **date** – display the current date and time

date returns information on the current date and time in the format shown below:-

```
Wed Jan 9 14:35:45 GMT 2002
```

It is possible to alter the format of the output from **date**. For example, using the command line

date '+The date is %d/%m/%y, and the time is %H:%M:%S.'

at exactly 3.10pm on 9th January 2002, would produce the output

```
The date is 09/01/02, and the time is 15:10:00.
```

6. **diff** – display differences between text files

diff file1 file2 reports line-by-line differences between the text files *file1* and *file2*. The default output will contain lines such as ' n1 a n2,n3 ' and ' n4,n5 c n6,n7 ', (where ' n1 a n2,n3 ' means that *file2* has the extra lines n2 to n3 following the line that has the number n1 in *file1*, and ' n4,n5 c n6,n7 ' means that lines n4 to n5 in *file1* differ from lines n6 to n7 in *file2*). After each such line, **diff** prints the relevant lines from the text files, with < in front of each line from *file1* and > in front of each line from *file2*.

There are several options to **diff**, including **diff -i**, which ignores the case of letters when comparing lines, and **diff -b**, which ignores all trailing blanks.

diff -cn produces a listing of differences within *n* lines of context, where the default is three lines. The form of the output is different from that given by **diff**, with + indicating lines which have been added, - indicating lines which have been removed, and ! indicating lines which have been changed.

diff dir1 dir2 will sort the contents of directories *dir1* and *dir2* by name, and then run **diff** on the text files which differ.

7. **file** – determine the type of a file

file tests named files to determine the categories their contents belong to.

file *file1*

can tell if *file1* is, for example, a source program, an executable program or shell script, an empty file, a directory, or a library, but (a warning!) it does sometimes make mistakes.

8. **find** – find files of a specified name or type

find searches for files in a named directory and all its subdirectories.

find . -name '*.cc' -print

searches the current directory and all its subdirectories for files ending in .cc, and writes their names to the standard output. In some versions of Unix the names of the files will only be written out if the **-print** option is used.

find /local -name core -user user1 -print

searches the directory /local and its subdirectories for files called *core* belonging to the user *user1* and writes their full file names to the standard output.

9. **ftp** – file transfer program

ftp is an interactive file transfer program. While logged on to one machine (described as the local machine), **ftp** is used to logon to another machine (described as the remote machine) that files are to be transferred to or from. As well as file transfers, it allows the inspection of directory contents on the remote machine. There are numerous options and commands associated with **ftp**, and **man ftp** will give details of those.

A simple example **ftp** session, in which the remote machine is the Central Unix Service (CUS), is shown below:-

ftp *cus.cam.ac.uk*

If the connection to CUS is made, it will respond with the prompt:-

Name (cus.cam.ac.uk:user1) :

(supposing *user1* is your username on your local machine). If you have the same username on CUS, then just press *Return*; if it is different, enter your username on CUS before pressing *Return*. You will then be prompted for your CUS password, which will not be echoed.

After logging in using **ftp** you will be in your home directory on CUS. Some Unix commands, such as **cd**, **mkdir**, and **ls**, will be available. Other useful commands are:-

help

lists the commands available to you while using **ftp**.

get *remote1 local1*

creates a copy on your local machine of the file *remote1* from CUS. On your local machine this new file will be called *local1*. If no name is specified for the file on the local machine, it will be given the same name as the file on CUS.

send *local2 remote2*

copies the file *local2* to the file *remote2* on CUS, i.e. it is the reverse of **get**.

quit

finishes the **ftp** session. **bye** and **close** can also be used to do this.

Some machines offer a service called “anonymous ftp”, usually to allow general access to certain archives. To use such a service, enter *anonymous* instead of your username when you ftp to the machine. It is fairly standard practice for the remote machine to ask you to give your email address as a password. Once you have logged on you will have read access in a limited set of directories, usually within the /pub directory tree. It is good etiquette to follow the guidelines laid down by the administrators of the remote machine, as they are being generous in allowing such access. See leaflet G72 for more detailed examples of using ftp.

WARNING! When you use **ftp** the communications between the machines are not encrypted. This means that your password could be snooped when you use it make an **ftp** connection. If available, the commands **sftp** (secure file transfer program) or **scp** (secure remote file copy program) are preferable, as they provide encrypted file transfer. Use **man scp** to get the syntax, which is like **cp**’s. To copy file1 from a remote machine to your local machine and name it file2:

```
scp user@pwf.cam.ac.uk:file1 file2
```

To copy file2 from here to your home directory on a remote machine:

```
scp file2 user@pwf.cam.ac.uk:~
```

10. **grep** – searches files for a specified string or expression

grep searches for lines containing a specified pattern and, by default, writes them to the standard output.

```
grep motif1 file1
```

searches the file *file1* for lines containing the pattern *motif1*. If no file name is given, **grep** acts on the standard input. **grep** can also be used to search a string of files, so

```
grep motif1 file1 file2 ... filen
```

will search the files *file1*, *file2*, ... , *filen*, for the pattern *motif1*.

```
grep -c motif1 file1
```

will give the number of lines containing *motif1* instead of the lines themselves.

```
grep -v motif1 file1
```

will write out the lines of *file1* that do NOT contain *motif1*.

11. **gzip** – compress a file

gzip reduces the size of named files, replacing them with files of the same name extended by .gz. The amount of space saved by compression varies.

```
gzip file1
```

results in a compressed file called *file1.gz*, and deletes *file1*.

```
gzip -v file2
```

compresses *file2* and gives information, in the format shown below, on the percentage of the file’s size that has been saved by compression:-

```
file2 : Compression 50.26% -- replaced with file2.gz
```

To restore files to their original state use the command **gunzip**. If you have a compressed file *file2.gz*, then

```
gunzip file2
```

will replace *file2.gz* with the uncompressed file *file2*.

12. **help** – display info about bash builtin commands

help gives access to information about builtin commands in the **bash** shell. Using **help** on its own will give a list of the commands it has information about. **help** followed by the name of one of these commands will give information about that command. **help history**, for example, will give details about the **bash** shell history listings.

13. **info** – read online documentation

info is a hypertext information system. Using the command **info** on its own will enter the **info** system, and give a list of the major subjects it has information about. Use the command **q** to exit **info**. For example, **info bash** will give details about the **bash** shell.

14. **kill** – kill a process

To kill a process using **kill** requires the process id (PID). This can be found by using **ps**. Suppose the PID is 3429, then

```
kill 3429
```

should kill the process. If it doesn't then sometimes adding **-9** helps...

```
kill -9 3429
```

15. **lpr** – print out a file

lpr is used to send the contents of a file to a printer. If the printer is a laserwriter, and the file contains PostScript, then the PostScript will be interpreted and the results of that printed out.

```
lpr -Pprinter1 file1
```

will send the file *file1* to be printed out on the printer *printer1*. To see the status of the job on the printer queue use

```
lpq -Pprinter1
```

for a list of the jobs queued for printing on *printer1*. (This may not work for remote printers.)

16. **ls** – list names of files in a directory

ls lists the contents of a directory, and can be used to obtain information on the files and directories within it.

ls *dir1*

lists the names of the files and directories in the directory *dir1*, (excluding files whose names begin with .). If no directory is named, **ls** lists the contents of the current directory.

ls -a *dir1*

will list the contents of *dir1*, (including files whose names begin with .).

ls -l *file1*

gives details of the access permissions for the file *file1*, its size in kbytes, and the time it was last altered.

ls -l *dir1*

gives such information on the contents of the directory *dir1*. To obtain the information on *dir1* itself, rather than its contents, use

ls -ld *dir1*

17. **man** – display an on-line manual page

man displays on-line reference manual pages.

man *command1*

will display the manual page for *command1*, e.g **man** *cp*, **man** *man*.

man -k *keyword*

lists the manual page subjects that have *keyword* in their headings. This is useful if you do not yet know the name of a command you are seeking information about, but can produce a lot of output. To refine the output you could, for example, use **man -k** *keyword* | **grep** '(1)' to get a list of user commands with *keyword* in their headings (user commands are in section 1 of the man pages). The | means that the output of **man -k** is piped to (i.e. is used as the input for) **grep**.

man -M*path* *command1*

is used to change the set of directories that **man** searches for manual pages on *command1*

18. **mkdir** – make a directory

mkdir is used to create new directories. In order to do this you must have write permission in the parent directory of the new directory.

mkdir *newdir*

will make a new directory called *newdir*.

mkdir -p can be used to create a new directory, together with any parent directories required.

mkdir -p *dir1/dir2/newdir*

will create *newdir* and its parent directories *dir1* and *dir2*, if these do not already exist.

19. **more** – scan through a text file page by page

more displays the contents of a file on a terminal one screenful at a time.

more *file1*

starts by displaying the beginning of *file1*. It will scroll up one line every time the *return* key is pressed, and one screenful every time the *space* bar is pressed. Type **?** for details of the commands available within **more**. Type **q** if you wish to quit **more** before the end of *file1* is reached.

more -n *file1*

will cause *n* lines of *file1* to be displayed in each screenful instead of the default (which is two lines less than the number of lines that will fit into the terminal's screen).

20. **mv** – move or rename files or directories

mv is used to change the name of files or directories, or to move them into other directories. **mv** cannot move directories from one file-system to another, so, if it is necessary to do that, use **cp** instead – copy the whole directory using **cp -r oldplace newplace** then remove the old one using **rm -r oldplace**.

mv *name1 name2*

changes the name of a file called *name1* to *name2*.

mv *dir1 dir2*

changes the name of a directory called *dir1* to *dir2*, unless *dir2* already exists, in which case *dir1* will be moved into *dir2*.

mv *file1 file2 dir3*

moves the files *file1* and *file2* into the directory *dir3*.

21. **nice** – change the priority at which a job is being run

nice causes a command to be run at a lower than usual priority. **nice** can be particularly useful when running a long program that could cause annoyance if it slowed down the execution of other users' commands. An example of the use of **nice** is

nice gzip *file1*

which will execute the compression of *file1* at a lower priority.

If the job you are running is likely to take a significant time, you may wish to run it in the background, i.e. in a subshell. To do this, put an ampersand **&**, after the name of your command or script. For instance,

rm -r mydir &

is a background job that will remove the directory *mydir* and all its contents.

The command **jobs** gives details of the status of background processes, and the command **fg** can be used to bring such a process into the foreground.

22. **passwd** – change your password

Use **passwd** when you wish to change your password. You will be prompted once for your current password, and twice for your new password. Neither password will be displayed on the screen.

23. **ps** – list processes

ps displays information on processes currently running on your machine. This information includes the process id, the controlling terminal (if there is one), the cpu time used so far, and the name of the command being run.

ps

gives brief details of your own processes in your current session.

To obtain full details of all your processes, including those from previous sessions use:-

ps -fu user1

using your own user name in place of *user1*.

ps is a command whose options vary considerably in different versions of Unix. Use **man ps** for details of all the options available on the machine you are using.

24. **pwd** – display the name of your current directory

The command **pwd** gives the full pathname of your current directory.

25. **quota** – display disk quota and usage

quota gives information on a user's disk space quota and usage.

On some systems using **quota** without options will only give details of where you have exceeded your disk quota on local disks, in which case, use the -v option

quota -v

to get details of your quota and usage on all mounted filesystems.

26. **rm** – remove files or directories

rm is used to remove files. In order to remove a file you must have write permission in its directory, but it is not necessary to have read or write permission on the file itself.

rm file1

will delete the file *file1*. If you use

rm -i file1

instead, you will be asked if you wish to delete *file1*, and the file will not be deleted unless you answer **y**. This is a useful safety check when deleting lots of files.

rm -r dir1

recursively deletes the contents of *dir1*, its subdirectories, and *dir1* itself, and should be used with suitable caution.

rm -rf dir1

is like **rm -r**, except that any write-protected files in the directory are deleted without query. This should be used with even more caution.

27. **rmdir** – remove a directory

rmdir removes named empty directories. If you need to delete a non-empty directory **rm -r** can be used instead.

rmdir *exdir*

will remove the empty directory *exdir*.

28. **sort** – sort and collate lines

The command **sort** sorts and collates lines in files, sending the results to the standard output. If no file names are given, **sort** acts on the standard input. By default, **sort** sorts lines using a character by character comparison, working from left to right, and using the order of the standard character set.

sort -d

uses “dictionary order”, in which only letters, digits, and white-space characters are considered in the comparisons.

sort -r

reverses the order of the collating sequence.

sort -n

sorts lines according to the arithmetic value of leading numeric strings. Leading blanks are ignored when this option is used, (except in some System V versions of **sort**, which treat leading blanks as significant. To be certain of ignoring leading blanks use **sort -bn** instead.).

29. **ssh** – secure remote login program

ssh is used for logging onto a remote machine, and provides secure encrypted communications between the local and remote machines using the SSH protocol. The remote machine must be running an SSH server for such connections to be possible. For example,

ssh -X *linux.phy.pwf.cam.ac.uk*

will commence a login connection to the Physics PWF server.

You can connect using your password for the remote machine, or you can set up a system of passphrases to avoid typing login passwords directly (see the man page for **ssh-keygen** for information on how to create these).

If you have a different user-id on the remote machine, use

ssh -X YourUserID@*linux.phy.pwf.cam.ac.uk*

The optional -X flag makes it possible for the remote machine to open X windows on your local machine.