

Introduction to Computational Physics, 370, taught in Spring 2016

Alexander J. Wagner,
Department of Physics, NDSU, Fargo, ND 58108, USA

January 12, 2016

Chapter 1

Introduction

About me

I was born 1967 in München, Germany. I studied Physics and Mathematics at the University of Bielefeld, Germany. My graduate degree is a D.Phil. in theoretical Physics from Oxford University, U.K. I then worked for two years as a Postdoctoral Researcher at the Massachusetts Institute of Technology in Cambridge, MA and for an additional two years at the University of Edinburgh in Scotland, U.K. Since 2002 I have been a Professor of Physics at North Dakota State University. I am married with two sons (6 & 8) and live with my family in North Fargo.

Contents

1	Introduction	3
1.1	Approach in this lecture	6
2	Introduction to Linux	9
2.0.1	Working remotely	13
3	Writing a report using L^AT_EX	17
4	Introduction to C programming	21
4.1	The bare bones	21
4.2	A bit more interesting	24
4.3	And now with a GUI	27
4.4	C-basics	34
4.4.1	Variables	34
4.4.2	Operators	34
4.4.3	Conditional execution	35
4.4.4	Functions	36
4.5	Fractals: Mandelbrot and Julia sets	36
4.6	Outlook	43
5	Newtonian dynamics	47
5.1	The falling ball	48
5.2	A simple oscillator	52
5.3	Two dimensional projectile motion	56
5.4	Three dimensional motion	60
6	Numerical Algorithms	61
6.1	The Euler algorithm	61
6.2	A second order method	62
7	Particles in a box and arbitrary graphics	69

8 Planetary motion	73
8.1 Adaptive Step Size	81
8.2 More particles	87
8.3 Simulating chaotic motion	97
8.4 Planets with internal structure	117
8.5 Diffusion of particles in a box	135
9 Basic Kinetic Theory and Statistical Mechanics	141
10 Monte Carlo	151
11 Lattice Gases	153
11.1 Hardy, de Pazzis, and Pomeau (HPP)	153
11.2 Frisch, Hasslacher, and Pomeau (FHP)	157
12 The Boltzmann equation	165
12.1 Multi-phase flow	166
12.2 Including a passive scalar	166
12.3 Including temperature as a passive scalar	167
A Programming Exercises	169
A.1 Graph library tips	170
A.2 The Pair correlation function	171

1.1 Approach in this lecture

In this lecture we will explore, how one can use a computer to aid out understanding of natural phenomena¹. We will start with some very simple discrete phenomena like the logistic equation (which maybe inspired by population dynamics) and then move on to continuous problems, like solving Newton's equations.

Soon we will move on to continuous systems for which we will derive the equivalent equations of motion to Newton's equations of motion. This is an area that is close to my own research interests and I will introduce you to the simple and beautiful world of lattice Boltzmann, which will allow us to examine even complex phenomena like eddies and turbulence.

To do this we have to choose a computational platform, and I will introduce you to the Graphical User Interface that we have developed over the last two decades. You will download the code from my website at http://www.ndsu.edu/physics/people/faculty/wagner/graphics_library/.

¹This lecture is a work in progress. Your comments and suggestions are always much welcomed, as are your suggestions for removing misprints from the lecture notes. If you find things boring or cumbersome, please let me know. Also if they are too challenging or if I am assuming background material that you do not know, please let me know right away.

To use this library you have to write your programs in C² on a system that provides the X-Window system. This is done most naturally on a Linux system that (typically) will be able to compile the example codes without any difficulty. Since Apple have now switched to a Unix based operating system as well, the same is true about Apple systems, as long as X and a compiler are installed. On Windows systems it is possible to use the graphics library as well, using cygwin-X, but I have not tested this on the new Windows 8. A web search gives some information that cygwin did not work last April, but there were developers actively on this, so it may have been solved by now.

The class is loosely based on the book “Introduction to Computer Simulation Methods” by H. Gould, J. Tobochnik, and W. Christian. However we do not use their codes, or the graphical libraries. Also none of the currently available textbooks cover lattice Boltzmann methods in any detail yet. So there these lecture notes will cover the missing material. Another useful reference for lattice Boltzmann methods is the Practical introduction to lattice Boltzmann methods, as well as Sauro Succi’s book on lattice Boltzmann.

²That is too strong a statement, some people have used the library with other programming languages, but this becomes more cumbersome.

Chapter 2

Introduction to Linux

Some of you may have used different operating systems in the past. Because of their initial design for multiple processes and multiple users High Performance Computing environments typically employ some version of Unix. Over the last decades most of these proprietary Unix flavors have been replaced by the free software version Linux which is also freely available to you.

If you are not yet familiar with Linux, this course will help you get familiar. There are many options of how to gain access to Linux or Linux-like environments. You will have access to the Linux computers in the classroom and you may get a double boot Linux partition if you have a laptop or a desktop that currently uses the Windows operating system. A second option for you would be Cygwin-X, but from my earlier experiences a double boot option should be preferred. If you use a Mac laptop or Desktop you will find that this is already built on a Unix variety (i.e. the BSD Unix) and you can use a project called fink to install a large variety of Unix programs.

Once you have installed a Linux variety (Unbutu seems to be an easy to install flavor) you need to learn the basic terminal commands. First open up a terminal. The most useful command is likely the manual command. If you remember that there is a list command called ls, but you don't remember the specific options you can call

```
1 $ man ls
```

This results in a lengthy output giving you all the detailed information about this command. It starts with

```
LS(1)                                User Commands
LS(1)
```

```
NAME
```

```
4      ls - list directory contents
```

```
SYNOPSIS
```

```
      ls [OPTION]... [FILE]...
```

9 DESCRIPTION

List information about the FILES (the current directory by default). Sort entries alphabetically **if** none of `-cftuvSUX` nor `--sort` is specified.

Mandatory arguments to long options are mandatory **for** short options too.

14

`-a, --all`
do not ignore entries starting with `.`

19

`-A, --almost-all`
do not list implied `.` and `..`

`--author`
 with `-l`, print the author of each file

24

`-b, --escape`
 print C-style escapes **for** nongraphic characters

29

`--block-size=SIZE`
 scale sizes by SIZE before printing them. E.g
`.., '--block-size=M` prints
 sizes **in** units of 1,048,576 bytes. See SIZE
 format below.

You can scroll through this listing using the up and down keys as well as the space key.
 The description ends with

1 Exit status:
 0 **if** OK,

 1 **if** minor problems (e.g., cannot access
 subdirectory),

 6 2 **if** serious trouble (e.g., cannot access **command-**
 line argument).

AUTHOR

Written by Richard M. Stallman and David MacKenzie.

11 REPORTING BUGS

Report `ls` bugs to bug-coreutils@gnu.org

GNU coreutils home page: <<http://www.gnu.org/software/coreutils/>>
 General **help** using GNU software: <<http://www.gnu.org/gethelp/>>
 Report ls translation bugs to <<http://translationproject.org/team/>>

16

COPYRIGHT

Copyright 2011 Free Software Foundation, Inc.
 License GPLv3+: GNU GPL version 3
 or later <<http://gnu.org/licenses/gpl.html>>.
 This is free software: you are free to change and
 redistribute it. There is NO
 WARRANTY, to the extent permitted by law.

21

SEE ALSO

The full documentation **for** ls is maintained as a
 Texinfo manual. If the info and
 ls programs are properly installed at your site, the
command

26

info coreutils 'ls invocation'

should give you access to the **complete** manual.

31 GNU coreutils 8.12.197-032bb September 2011
 LS(1)

And important section is the SEE ALSO section, where you find hints of where else you may want to look for information. In this case it points you to the very useful info command, that you can use to learn more about ls. To get back to the command prompt, type 'q'.

There are a number of important commands you should learn to use a terminal:

ls (lists files **in** the current directory)
 mkdir (makes a new directory)
 rm (deletes a file, or a directory, or a whole filesystem ...)
 4 rmdir (deletes an empty directory)
 cd (changes the current working directory)

To learn more about these commands use man. If you are running a program from the terminal you can use special signals to interrupt or kill the program. In this context the `^bcdgijkmprsu` character refers to the ctrl key that needs to be pressed simulatenously with the other indicated key.

`^Z` (interrupts program **in** terminal and returns you to **command** prompt),
`bg` (sends program to the background),
`fg` (sends program to the foreground),
`^C` (kills program)

You can also send a program into the background right away by using the `&` character right after the command.

Installing new software (specific to debian and derivatives like ubuntu, on a mac you would use fink for a similar purpose)

`apt-cache` (shows all installation candidates),
`apt-get` (installs the program)

A key program you need is an editor which will allow you to edit programs, latex files, data files, debug programs and much more. In the Unix world there are two key editors called vi and emacs. My preference is for the editor called emacs. You can search for versions of emacs available by calling

`apt-cache search emacs`

which will give you a large number of packages related in some way to emacs. Now install the editor using something like

```
1 $ sudo apt-get install emacs
```

where `sudo` allows you to execute a command as a super-user on the system. Now you can enter the editor by typing

```
$ emacs
```

Once you are in the editor you can create text files like a program in C or a documentation of your project in L^AT_EX. Please go to the 'Help' tab and do the emacs tutorial to start to get familiar with the power of emacs.

To write documentations (like this one) including fancy mathematical formulas (and all documents) like this one:

$$\int_0^\infty \frac{1}{1+x} = \alpha \sum_{n=0}^\infty \frac{\beta^n}{n!} \quad (2.1)$$

you will need to install L^AT_EX using something like

```
$ sudo apt-get install texlive-latex-recommended texlive-latex-recommended-doc
```

The C-compiler is called `gcc` and it should be preinstalled on your Linux system. You can generate C-programs using the editor and compile them with

```
$ gcc
```


or eventually using the more convenient command

```
$ make
```

Additional user programs you will want to install include:

```
xfig (draw figures),
xmgrace (draw graphs from data sets).
```

2.0.1 Working remotely

One advantage of Linux systems with a fixed IP address is that you can access them remotely from your home (or anywhere in the world) and run your programs on different computers. This is particularly useful when you use your program to run for a long time and/or with a large number of different parameters. You will all have logins on the cluster of mini's in 221.

To working remotely you will use the following programs:

```
ssh, scp,
vncserver, vncviewer
```

Let us use an example: you are on your local laptop and you want to access data that you have left on a mini. You can log into that mini using a secure shell:

```
$ ssh -X username@mini04.physics.ndsu.nodak.edu
The authenticity of host 'mini04.physics.ndsu.nodak.edu (134.129.87.72)' can't be
established.
ECDSA key fingerprint is 31:49:5a:08:c9:57:1c:41:f0:61:23:27:9d:13:98:3a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'mini04.physics.ndsu.nodak.edu,134.129.87.72' (ECDSA)
to the list of known hosts.
awagner@mini04.physics.ndsu.nodak.edu's password:
Welcome to Ubuntu 12.04.5 LTS (GNU/Linux 3.2.0-72-generic x86_64)
```

* Documentation: <https://help.ubuntu.com/>

The programs included with the Ubuntu system are free software; the exact distribution terms for each program are described in the individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.

```
username@mini04:~$
```

where 'username' is your username that Paul provided for you. Now you have a shell on the remote computer and you can run all the programs that you can run on the remote computer. The '-X' flag enables the xwindow connection that will be able to open up new windows on your local computer.

Now there are files on this local computer (and you can see them using ls), but how do you get them to your local computer (and back again)? This is where secure copy comes in handy. Say I have a file on the mini, prog.c in the directory ./c in your homedirectory, that I want to have on my local computer. To transfer it you type in a terminal on your local computer:

```
$ scp username@mini04.physics.ndsu.nodak.edu:c/prog.c .
```

Now you have the file on your local computer in your local directory. Maybe you debugged your program or you extended it, and now you want to save a copy back at the remote computer. Do do that you simply do

```
$ scp prog.c username@mini04.physics.ndsu.nodak.edu:c/
```

You can use scp to run a program in the background (remember the '&' after the command) using ssh. However, if the program opens up a window it relies on the connection between your local and the remote machine remaining open. Sometimes that is not what you want. This is a situation where a vitrual network client (VNC) comes in handy.

If you are logged into the computer you want to access (either directly or remotely via ssh) you first set a password to access the vncserver:

```
$ vncpasswd
Using password file /home/username/.vnc/passwd
Password:
Verify:
Would you like to enter a view-only password (y/n)? n
```

Then you start the vncviewer

```
$ vncserver
```

```
New 'X' desktop is mini04:1
```

```
Starting applications specified in /home/username/.vnc/xstartup
Log file is /home/username/.vnc/mini04:1.log
```

To access the computer via vnc you then simply need to download a package called vncviewer, then call

```
$ vncviewer mini05.physics.ndsu.nodak.edu:1
```

```
VNC Viewer Free Edition 4.1.1 for X - built Jan 30 2013 16:07:52
```

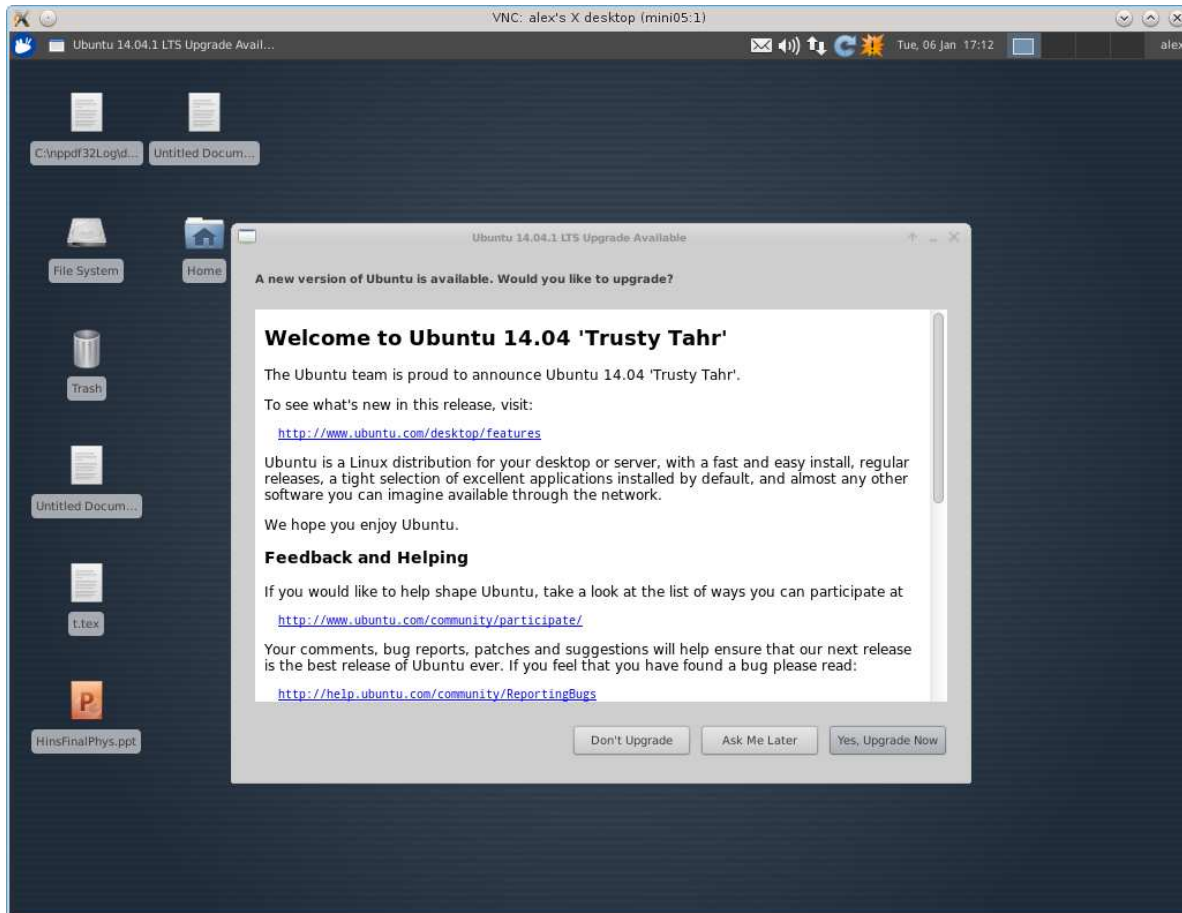


Figure 2.1: A snapshot of a vncscreen that allows you to use a remote computer as if you were sitting at it.

Copyright (C) 2002-2005 RealVNC Ltd.

See <http://www.realvnc.com> for information on VNC.

Tue Jan 6 16:59:29 2015

CConn: connected to host mini05.physics.ndsu.nodak.edu port 5901

CConnection: Server supports RFB protocol version 3.8

CConnection: Using RFB protocol version 3.8

Password:

Tue Jan 6 16:59:32 2015

This should then see a window which should look like Figure 2.1. From there you can control the computer just as if you were sitting at it. Even if you close your vncviewer, you can to back to what you were doing (or check on the progress of a program you started) by simply calling vncviewer yourself. Also, more than one person can control the same vncscreen, if you are sharing a password. (This is most useful if you can also communicate in another way, say via Skype). We can also display such a screen during the lecture on the projector.

Chapter 3

Writing a report using L^AT_EX

Most scientific journals, like the Physical Review, prefer to receive submissions in L^AT_EX, and with good reason. It is the publishing platform that most easily deals with complex mathematical expressions. If you have never used it before it may take a little getting used to.

If you are used to a WYSIWYG editor this will look strange to you at first, but you will get used to it fast enough. Let us write our first L^AT_EX manuscript:

```
\documentclass{article}

\begin{document}
Hello World!
\end{document}
```

Once we have saved this in a file called World.tex we can compile it using the latex compiler:

```
$ latex World.tex
This is pdfTeX, Version 3.1415926-2.4-1.40.13 (TeX Live 2012/Debian)
restricted \write18 enabled.
entering extended mode
(./World.tex
LaTeX2e <2011/06/27>
Babel <v3.8m> and hyphenation patterns for english, dumylang, nohyphenation, lo
aded.
(/usr/share/texlive/texmf-dist/tex/latex/base/article.cls
Document Class: article 2007/10/19 v1.4h Standard LaTeX document class
(/usr/share/texlive/texmf-dist/tex/latex/base/size10.clo))
No file World.aux.
[1] (./World.aux) )
Output written on World.dvi (1 page, 232 bytes).
Transcript written on World.log.
```

This process generates the file World.dvi, which you can now look at using by calling xdv World.dvi. This, unsurprisingly, shows a blank page with the words "Hello World".

Now let us do something more interesting. In the next (somewhat compressed) example we show a full latex document that has a title, an author, some section headings, a figure, equations and references to equations as well as a bibliography and some citations. In short it contains the basic elements you need to write a full paper.

Here we go:

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\title{First Latex document}
\author{Your name here,\\ Department of Physics,\\
North Dakota State University, Fargo ND, USA}
\maketitle % only once you type this will the title appear.
```

```
\begin{abstract}
A short demonstration on how to use \LaTeX.
\end{abstract}
```

```
\section{Introduction}
\LaTeX is great to set formulas like this one
\begin{equation}
i \hbar \frac{\partial \Psi}{\partial t} =
-\frac{\hbar^2}{2m} \nabla^2 \Psi + V(x) \Psi
\label{Schrodinger}
\end{equation}
```

You should note that this equation is automatically numbered, and we can refer back to that number by using: Eqn. (\ref{Schrodinger}) is known as the Schrödinger equation. (to get the reference right, you will have to call latex twice).

```
\section{Graphics}
The next important thing you have to learn is to include figures
into your paper. Here is an example:
\begin{figure}
\includegraphics[width=\textwidth]{CalvinHobbesRelativity.eps}
\caption{This is an alternative explanation of the theory of relativity.}
\label{AlternativeRelativity}
\end{figure}
```

And you can refer to figures in the same way as we see in Fig. \ref{AlternativeRelativity}. One small caveat: traditional L^AT_EX

implementation require that you need to use graphics in encapsulated postscript format. Ideally you will generate graphics in eps format directly. If you have graphics in a different format, however, you can install a utility called 'convert'. You obtain it by installing the package 'imagemagick'.

```
\section{Conclusion}
```

Last but not least every self-respecting paper is citing relevant references\cite{einstein}. These references are provided in the bibiligraphy. This is either typed in directly, or, more conveniently, using bibtex.

```
\bibliography{mybib}{}
```

```
\bibliographystyle{plain}
```

```
\end{document}
```

You need to have a second file, called mybib.bib in this case, that contains the information on the articles you want to cite:

```
@article{einstein,
  author =      "Albert Einstein",
  title =       "{Zur Elektrodynamik bewegter K{\o}rper}. ({German})
    [{0n} the electrodynamics of moving bodies]",
  journal =     "Annalen der Physik",
  volume =      "322",
  number =      "10",
  pages =       "891--921",
  year =        "1905",
  DOI =         "http://dx.doi.org/10.1002/andp.19053221004"
}
```

To compile the latex text you then need to type:

```
$ latex World2
$ bibtex World2
$ latex World2
```

The first latex generates a file that tells bibtex which citations are needed, then bibtex finds those citations and generates a file that is in turn used by latex to include in the bibliography.

This is a very short summary of the most important technical aspects you should know to write scientific papers. In terms of content a paper consists of an abstract that summarizes the main findings of the paper (i.e. why should you care). This is followed

by the introduction that puts the research into the proper scientific context, and this is also the place where you would discuss related research. Then you have the main part of the paper which typically consists of a section setting up the problem and the methods used to address the problem and a results section that shows your actual results. This is followed by a conclusion, that will recapitulate the main results again and maybe indicate which new questions were raised by your results.

Chapter 4

Introduction to C programming

We can't provide a full reference to c-programming here, but I want to give a brief introduction to the basics of getting a program up, so that you all have a starting point where you can use other references to improve your knowledge of C.

4.1 The bare bones

We start with the terminal application, which should exist on all suitably installed operating systems. This terminal will provide you with a prompt, from which you can call an amazingly large number of programs. Typically the terminal will start you up in your home-directory. You can examine the contents of your directory using the `ls` (list) command. Just type “ls”, followed by the return key and you will see the contents of your home-directory.

Next we will want to create a new directory, so that you can keep your own work separate place. Say you want to call this directory “class370”. Type `mkdir class370` and the directory will be created. For the name is easiest if you just use simple letters and numbers as spaces and some characters like %, ? or \$ have special meanings and require extra care if you want to use them in directory names. As a general rule it is best to avoid them.

Next we want to move into the new directory. Change your directory by typing `cd class370`. Now you are in the newly created directory.

Next we need to write our source-code, so we require an editor. My editor of choice is emacs, but there is a large variety of suitable alternatives. Let us open the editor with `emacs first.c`. This will open up an editor in its own window¹.

In the editor emacs you can now start to write your first C program. If you are unfamiliar with emacs I suggest that you go to the “Help” menu and start the emacs tutorial. Now you can write your first c-code:

¹If it opens up in the terminal window, this may be the sign of trouble...

Listing 4.1: first.c

```

#include <stdio.h>

main () {
4   printf("Computational_Physics_rules!\n");
}

```

Next you will have to compile your c-code. In this case you simply type `cc first.c` in the terminal window, and you get the executable, which is called “a.out” by default. To run your program you simply type `./a.out`, and your program will do its job and write out the text you told it to write.

Now occasionally you will do something foolish, and your program will refuse to run in the way that you intend. Let us include an arbitrary error in our program. We need to include a little more program, so we copy the source code to `firstwrong.c`, define three integers in the new source code and (foolishly) divide by zero:

Listing 4.2: firstwrong.c

```

#include <stdio.h>

main () {
    int i,j,k;
5   i=10;
    j=0;
    k=i/j;
    printf("Computational_Physics_rules!_%i\n",k);
}

```

When I compile (`cc firstwrong.c`) there seems to be nothing amiss: the compiler is quite happy² and generates a new a.out. However, when I run this code I get an error:

```

alex$ ./a.out
Floating point exception

```

so, not surprisingly, something went wrong. Typically, however, you will have no idea why something went wrong, and this is where a most powerful tool comes into play: the debugger. To use the debugger you need to provide “hooks” in the code. You do this by compiling the code with the “-g” flag:

```
alex$ cc -g firstwrong.c
```

From within emacs you can then call the debugger with a keystrokes `meta-x gdb`. It will then suggest that you run `gdb` on a.out, which is what you want to do. This will set you up within emacs with a debugger window, which starts at a prompt where you type:

²If you do things that are *syntactically* wrong, the compiler will complain. You will, no doubt, experience this soon.

```
(gdb) run
Starting program: /Users/alex/mytex/committee/syllabus/370_13/a.out
Reading symbols for shared libraries +. done
```

```
Program received signal EXC_ARITHMETIC, Arithmetic exception.
0x0000000100000eee in main () at firstwrong.c:7
```

At the same time the screen will split in half and the debugger will show you exactly where in your program the problem occurred. Now you can examine the relevant variables with

```
(gdb) p i
$1 = 10
(gdb) p j
$2 = 0
(gdb) p k
$3 = 32767
(gdb)
```

and you will probably notice that `j` has the unfortunate value of "0", which caused the arithmetic exception. Of course this is a silly toy example, but when you run into problems with your code you will find that a debugger can save your hours of painful searching.

Problems

4.1.1: Follow all the instructions above, find the terminal, create your directory, call emacs, write your first very simple C programs, compile them and debug the incorrect code.

4.1.2: Play with these tools and look at the emacs tutorial and a gdb-debugger tutorials like this one <http://tedlab.mit.edu/~dr/gdbintro.html>.

4.2 A bit more interesting

Now let us write a program that actually does something. Let us consider some fluffy bunnies on a nice green meadow. Each spring, a pair of bunnies will give birth to a litter of little bunnies. Depending on the size of the breed the average number per litter can vary from 2 to 10. Let us call the number of offspring per pair $2b$. So how does the number of rabbits change from season to season? Let us assume that we have N_0 rabbits in year zero. In the next year we will have $N_1 = N_0 + \frac{N_0}{2}2b = N_0(1+b)$ rabbits. For all subsequent years we can write the number of rabbits as

$$N_{t+1} = N_t(1+b). \quad (4.1)$$

Now let us extend our program to calculate these numbers directly:

Listing 4.3: bunny1.c

```

1 #include <stdio.h>

   int b=1;

   int NextGen(int N){
6   return N*(1+b);
   }

   main () {
       int i,N=2;
11
       printf("Bunny_simulation.\n_year_bunnies\n");
       printf("%i_%i\n",0,N);
       for (i=1;i<10;i++){
           N=NextGen(N);
16   printf("%i_%i\n",i,N);

```

```

    }
}

```

Again we compile and run the code and get the result

```

3$ ./a.out
Bunny simulation.
  year bunnies
0  2
1  4
2  8
3 16
4 32
5 64
6 128
7 256
8 512
9 1024

```

Obviously there is something wrong, as the number of bunnies seems to increase without bound. What we missed is the number of bunnies that die in the time. If this death-rate were simply a constant we could absorb it into a modified birth rate. But the death rate has to depend on the existence of a limit for the carrying capacity of our meadow for the maximal number of rabbits it can sustain before they start to starve. It is reasonable to have a death rate that will go to 1 (i.e. all bunnies will die) when the number of rabbits goes to infinity. So say we have a death rate $d = k * (M + N)$ where $M \gg 1$. This means that for small number of rabbits the death rate will simply be a constant kM , but for larger number of rabbits, the death rate increases. For this to work it makes sense either to make a stochastic model (i.e. the death rate gives a probability for a bunny to die), or we move to an average model, where the number of rabbits now becomes a floatingpoint number. The interpretation would be something like: if we have many bunny populations with N_0 bunnies the average number of bunnies in the next year will be N_1 bunnies, and now N_1 does no longer need to be an integer. Since the second approach is much simpler, and we are still at the beginning of this course, we will take the second route. So now our equation for the evolution of the (continuous) bunny population is

$$N_{t+1} = N_t + bN_t - kN_t(M + N_t) \quad (4.2)$$

$$= N_t(1 + b - kM) - kN_t^2 \quad (4.3)$$

Obviously there are only two relevant parameters in this problem, so let us give them a name $a = 1 + b - kM$ and k . This allows us to write the simple evolution equation

$$N_{t+1} = aN_t - kN_t^2 \quad (4.4)$$

We can easily put this into our program (remembering to switch the integer population to a continuous (i.e. double) population):

Listing 4.4: bunny2.c

```

#include <stdio.h>
2
double a=2,k=0.01;

double NextGen(double N){
    return a*N-k*N*N;
7 }

main () {
    int i;
    double N=2;
12
    printf("Bunny simulation.\nyear bunnies\n");
    printf("%i %5.1f\n",0,N);
    for (i=1;i<10;i++){
        N=NextGen(N);
17    printf("%i %5.1f\n",i,N);
    }
}

```

With this program the result looks more reasonable:

```

$ ./a.out
Bunny simulation.
year bunnies
0 2.0
1 4.0
2 7.8
3 14.9
4 27.6
5 47.6
6 72.6
7 92.5
8 99.4
9 100.0

```

We see that the number of bunnies still increases, but now the population appears to level off, at some sustainable level. Now we can play with the program a bit and we may wonder what the population evolution looks like if we increase the birth rate, i.e. the a factor to something larger, maybe $a = 3$. We then get

```

$ ./a.out
Bunny simulation.
  year bunnies
0   2.0
1   6.0
2  17.5
3  49.5
4 124.0
5 218.2
6 178.4
7 216.9
8 180.2
9 215.9

```

This is interesting, the bunny population no longer simply converges to a large number but it seems to oscillate. We need to explore the program for different parameters.

It is at about this point that you start to run out of patience. Firstly you will want some graphical representation of our results, secondly it is cumbersome to re-compile your program any time you want to look at a new parameter value. This is why we want to put a graphical user interface on our program so that we can freely manipulate the program and get immediate feedback on the results.

4.3 And now with a GUI

Putting on a GUI will complicate the program, but the effort is well worth while, as we will see in a moment. Let us think what we need to do: we need a data structure to display, i.e. the populations at different times. We will need to actually save this data, so that it can be displayed.

Secondly we want to control our parameters a and k and we want to be able to change them, and lastly we want to be able to stop the program, reinitialize it and to start it afresh. A code that does this looks like this:

Listing 4.5: bunnyGUI.c

```

1 #include <stdio.h>
  #include <mygraph.h>

  double a=3,k=0.01;
  double n[100];
6 int size=100,bunnyreq=0;
  double n0=2;

double NextGen(double N){
  return a*N-k*N*N;

```

```

11 }

    void GetPopulationGraph () {
        int i;
        n[0]=n0;
16     for ( i=1;i <100;i++){
            n[i]=NextGen(n[i-1]);
        }
    }

21 main () {
    int i,done=0;
    double N=2;

    DefineGraphN_R("Bunny/year",n,&size,&bunnyreq);
26    StartMenu("Bunny_Program",1);
    DefineDouble("a",&a);
    DefineDouble("k",&k);
    DefineDouble("n0",&n0);
    DefineGraph(curve2d_,"Bunny_Graph");
31    DefineBool("done",&done);
    EndMenu();
    while (!done){
        Events(1);
        DrawGraphs();
36        sleep(1);
        if (bunnyreq){
            bunnyreq=0;
            GetPopulationGraph();
        }
41    else sleep(0);
    }
}

```

To compile the code first you need to install the graphics library on your system, by following the instructions on http://www.ndsu.edu/physics/people/faculty/wagner/graphics_library/. When you now want to compile the code you need to include some directives that tells the compiler where to find the new library and you need to tell it to link the library graph as well as the system library x11 and the math library:

```
$ cc -I ~/include -L ~/lib bunnyGUI.c -lgraph -lm -lX11
```

which should work, if all the libraries were installed correctly. Remember that some systems are picky about the order in which they like the libraries to be listed.

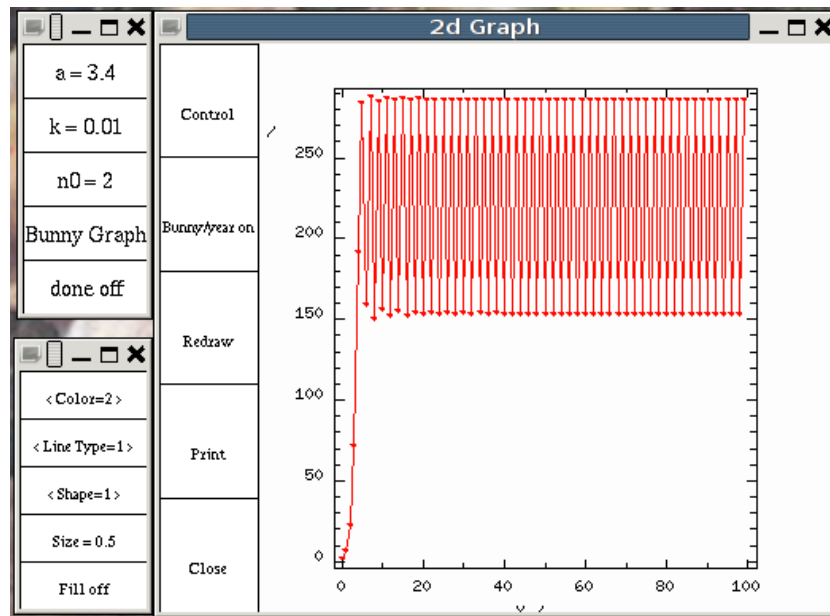


Figure 4.1: Screenshot of your first program with a graphical user interface. The top left window is the main window. The window of the right appeared after you clicked on "Bunny Graph". The bottom left window allows you to control how the graph is displayed and popped up after you right-clicked the "Bunny/year" button.

When you run the program, now a window will appear. This is shown in Figure 4.1. What get first is a window that shows the variables you wanted to see. When you click on one of these windows you can change the numerical value of these parameters. To see the graphics you have to first click on the “Bunny Graph” button you defined. You then see a graphics window which at first shows nothing but some menus on the left. There you have to select the (for now single) available data button “Bunny/year”. Once you have done that you will see a wiggly line, representing the data of the bunnys for each year where we calculated their population. You can change the appearance of the graph by right-clicking the “Bunny/year” button. If you do that another menu appears that lets you change the color, linestyle and a symbol for the data points. This is the situation that you see in Figure 4.1.

Now you can change the parameters, say the value of a and examine how the graph changes. You will notice that for smaller values for a the number of bunnies converges to a limiting value, but for a larger value you will get at first two and then a dizzying variety of values.

Now it would be nice to show these final values change as a function of the parameter a . To do that we need to do a parameter-sweep where we change the parameter a from, say, 1 to 4 and record the values that the bunny populations take. The easiest way of doing this is to initialize the simulation with some value and then let it run for a number of iterations, maybe 1000 or so, and then record the next, say, 50 values. This data would be written into another array, this time a two dimensional one that includes the a value as the first dimension and the n values as the second dimension. Such a could may look like this:

Listing 4.6: bunnyGUI2.c

```

#include <stdio.h>
2 #include <mygraph.h>

double a=3,k=0.01;
double n[100];
int size=100,bunnyreq=0;
7
double sw[200000][2],amin=1,amax=4;
int size2=200000,sweep2req=0;
int size2r=0;
double n0=2;
12
double NextGen(double N){
    return a*N-k*N*N;
}

17 void GetPopulationGraph(){
    int i;
```

```

    n[0]=n0;
    for ( i=1;i <100;i++){
22      n[i]=NextGen(n[i-1]);
    }
}

void GetPopulationSweep() {
    int i,j;
27    double m;

    size2r=0;
    for ( j=0;j<size2/500;j++){
        a=amin+j/(size2/500.0-1.0)*(amax-amin);
32    m=n0;
        for ( i=1;i <1000;i++){
            m=NextGen(m);
        }
        for ( i=0;i <500;i++) {
37            m=NextGen(m);
            sw[size2r][0]=a;
            sw[size2r][1]=m;
            size2r++;
        }
42    /* We put these in here to keep the program responsive
        while the sweep is taking place, and so that we can see
        the data develop */
        Events(1);
        DrawGraphs();
    }
}
47

main () {
    int i,done=0;
    double N=2;

52    DefineGraphN_R("Bunny/year",n,&size,&bunnyreq);
    DefineGraphN_RxR("Bunny/a",&(sw[0][0]),&size2r,&sweep2req);
    StartMenu("Bunny_Program",1);
    DefineDouble("a",&a);
    DefineDouble("k",&k);
57    DefineDouble("n0",&n0);
    DefineGraph(curve2d_,"Bunny_Graph");

```

```

        DefineDouble("amin",&amin);
        DefineDouble("amax",&amax);
        DefineFunction("Final_states",&GetPopulationSweep);
62    DefineBool("done",&done);
        EndMenu();
        while (!done){
            Events(1);
            DrawGraphs();
67        sleep(1);
            if (bunnyreq){
                bunnyreq=0;
                GetPopulationGraph();
            }
72        else sleep(0);
        }
    }

```

We have now defined a new function `GetPopulationSweep` and two new variables `amin` and `amax`. We included both the parameters and the function in the menu, and you can call the function by clicking on the menu window.

Running the program provides some rather spectacular results, which are shown in Figure 4.2. To get the graphs to look this way you have to alter the display options by right-clicking the Bunny/a menu button. Then you get the same menu shown in Figure 4.1. I choose `LineType=0`, `Shape=4`, `Size=0.3`, `Fill off` for the graphs shown in Figure 4.2. To output these graphs there is a print button in the graphics video. Leftclicking on the print button you get another menu. For the output I had to alter the height to 0.7 to change the aspect ratio and I choose `Landscape off`. When you hit the final print button an eps file with the name `graph3d_000.eps` is created. Subsequent prints increase the number, so you have an electronic record of the graphs you printed.

Finally we should now consider the Control button on the top left of the graphics window. When you click this button you can see that the standard choice is for the graphics to self adapt. However, sometimes we want to be able to zoom into a section of the graphics. In this case we can select the x and y values between which we want to view the graphics.

Problems

4.3.1: Show the smaller section between $x \in (3.853, 3.8542)$ and $y \in (52, 53)$, still for $k = 0.01$, and printout the result as an eps file, using the print function.

4.3.2: Show analytically how you expect the results to behave for different values of k . Show that your analysis is correct by numerically verifying it.

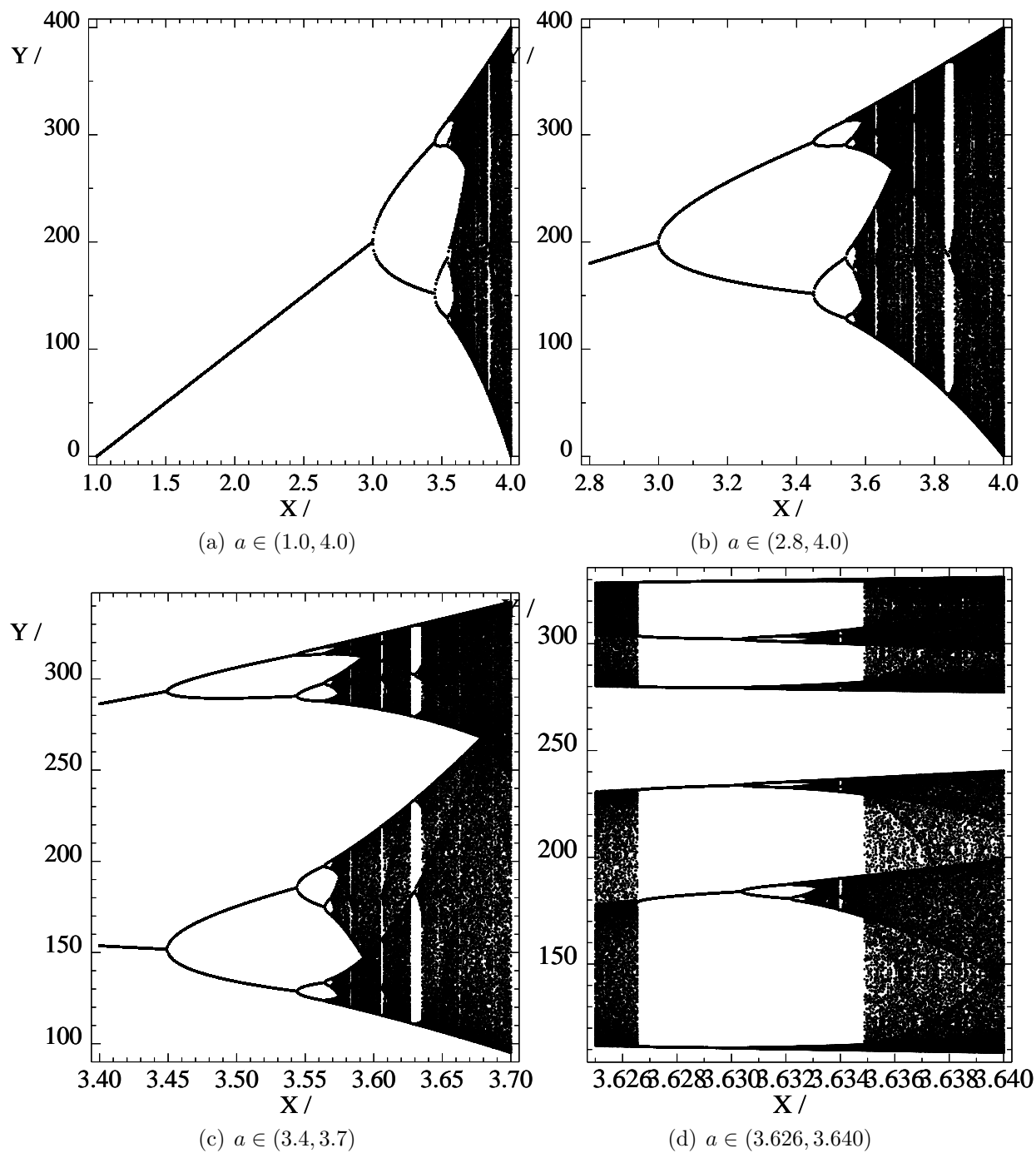


Figure 4.2: The final 500 population densities for different parameters a . These graphs are also known as bifurcation diagrams.

4.3.3: More details on these kinds of problems can be found in chapter 6 of the book. They are mathematically known as one-dimensional maps. Alter the program and examine the behavior of a “bunny function” of $n_{i+1} = n \exp[a(1 - n)]$.

4.4 C-basics

We conclude this section with a brief overview of the most important features of the c-programing language.

4.4.1 Variables

C requires you to define all variables before you are able to use them. You define them to be of a specific type. The two most important types are integer variables declared as `int` and numerical representations of real numbers, e.g. variables of type `double`. Other important variable types are pointers, and we will discuss them shortly.

Examples

```
int i,j; /* defines two integers */
double d,r; /* defines two doubles */
```

Another important feature here is the comment. Anything enclosed between `/*` and `*/` will be ignored by the compiler and should be used to put comments in the code.

Sometimes you will want to refer not to the value of the variable, but to its address in memory. This is achieved by using the address operator.

Example

```
double x=1,*p; // Now x is a double and p is a pointer to a double.
p= &x;         // the & operator gives the address of a variable.
*p=2;          // the * operator references a pointer, so *p is a double
printf("%f, %f",x,*p); // both are 2 now.
```

4.4.2 Operators

The variables can be operated on with a variety of operators and they can be re-assigned through assignments. Assignments are done with the `=` operator. The basic mathematical operators `+`, `-`, `*`, `/` can be used on variables of type `int` and `double`.

Examples

```
i=j+4;
d=r*15;
```

In C it is possible to simplify these expressions above to

```
i+=4;
d*=15;
```

Caution

A common source of errors are integer divisions. In particular you will find that in an integer division $1/10 = 0$, which leads to the unexpected result $(1/10)*10 = 0$. To avoid integer arithmetic where it is not appropriate it is necessary to indicate to the C-compiler that you want it to do real operations: $(1/10.)*10 = 10$, as expected. When you are using variables, you can indicate that you want real operations by starting the calculation by multiplying with 1.0 as in

```
(i/j)*j /* != i in general */
1.0*(i/j)*j /* = i ! */
```

Another important set of operators in C are comparison operators. To check equality use the `==` operator, to check for inequality use the `!=` operator. Two other important comparison operators are `<` and `>`.

There are many more operators available in C. Please consult a C reference book for a complete list.

4.4.3 Conditional execution

The real power of computation comes from re-using parts of code. The simplest example is re-running a bit of code with different values of the variables by using `for` or `while` loops. The syntax of a `for` loop in C is `for (initialization; break condition; incrementor);`

Example

```
for (i=0;i<100;i++){
    r=rmin+i/99. *(rmax-rmin);
    printf("%f*%f = %f",r,r,r*r);
}
```

However, C is quite flexible and you don't actually need the integer here. Instead you could write

```
for (r=rmin;r<=rmax;r+=(rmax-rmin)/99){
    printf("%f*%f = %f",r,r,r*r);
}
```

A while loop is executed while the condition remains true, so we can write the same statement above as:

```
r=rmin;
while (r<=rmax){
    printf( "%f*%f = %f",r,r,r*r);
    r+= (rmax-rmin)/99;
}
```

4.4.4 Functions

It would be cumbersome to write all the programming code in a single block of text. The power of programming lies in reusing bits of code in different circumstances. This power is realized by defining functions. It is good programming style to pass all the variables that the function needs to it through the arguments. It is *important* to realize that any arguments that the function receives are put on a stack (i.e. a different part of memory) and if you alter the arguments in the function, it has no effect on the parameter that you passed the function.

Example

```
double f(double x){
    x=2*x;
    return x*x;
}

main(){
    double x=1,y;
    y = f(x); // now y=4 and x=1. Note that x is unchanged.
}
```

If you need for a function to change the content of one of the parameters, then you have to pass not the parameter but the address of the parameter. **Example**

```
double f(double *x){
    *x=2* *x;
    return *x * *x;
}

main(){
    double x=1,y;
    y = f(&x); // now y=4 and x=2. Note that x has changed.
}
```

4.5 Fractals: Mandelbrot and Julia sets

We are now considering a similar series of the kind

$$x_{n+1} = x_n^2 + c \tag{4.5}$$

For $c = 0$ we know that all series with initial values with $x_0 > 1$ will diverge and all series with initial values with $x_n < 0$ converge to zero. But what if $c \neq 0$? To make

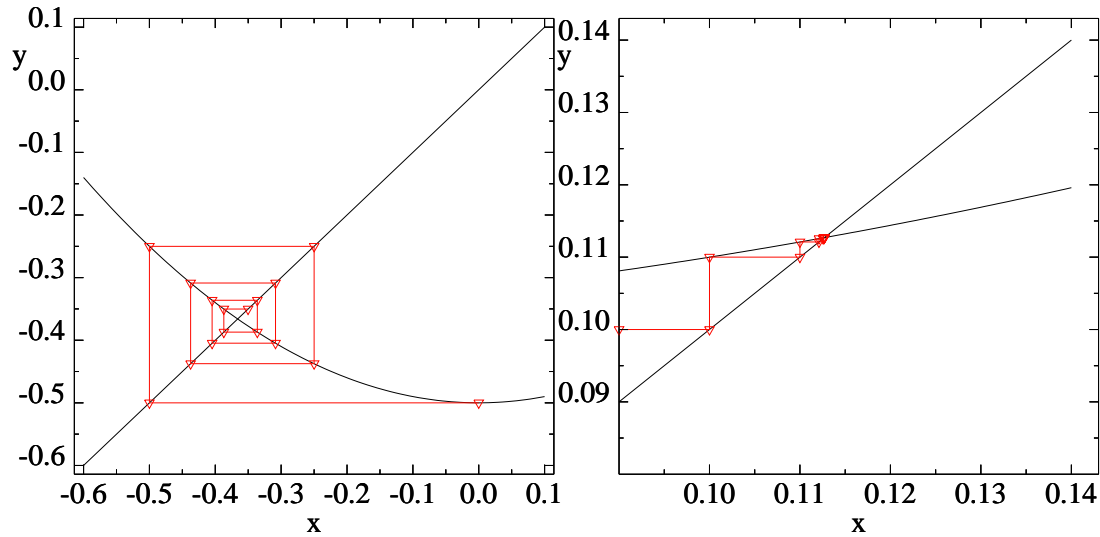


Figure 4.3: Examples of the graphical method of understanding sequences.

this question simpler we will look at the situation of $x_0 = 0$. So our series values will be

$$x_0 = 0 \quad (4.6)$$

$$x_1 = c \quad (4.7)$$

$$x_2 = c + c^2 \quad (4.8)$$

$$x_3 = c + (c^2 + c)^2 \quad (4.9)$$

$$\vdots \quad (4.10)$$

We know that the series of $1/n$ does not converge, whereas the series of $1/n^2$ does converge.

There is actually a very nice way of visualizing the evolution of such a sequence. For your sequence

$$x_{n+1} = f(x_n) \quad (4.11)$$

you can draw a graph of $y = f(x)$ and the diagonal $y = x$. Then you start from your initial value x_0 , advance vertically to the function and find the value of $f(x_0)$. Then you draw a horizontal line from to the diagonal $y = x$, which will give you your new x value. Then you draw another vertical line to the function, and this procedure graphically represents the effect of the sequence of equation (4.11).

An example for this is shown in Figure 4.3. Fixed points are given by the points where the graphs for $y = f(x)$ and $y = x$ intersect. These fixed points can be stable (as the ones shown in the figure), or unstable. (It is easy to see that fixedpoints are unstable is $|f'(x)| > 1$).

This is the program I used to generage the sequences here:

Listing 4.7: sequence.c

```

1  #include<mygraph.h>
   #include <unistd.h>
   #include<math.h>

   #define N 100
6  #define Nf 100

   typedef struct xy {double x;double y;} xy;

   double c=0,d=1,x0=0,xmin=-1,xmax=1;
11 xy gr[2*N] , ff[Nf] , xx[2];
   int done, iteration=5, it2=10, getgr=0, getxx=0, getff=0, Nff=Nf, two
       =2;;

   double (*f)(double);

16 double f2(double x){
       return d*x*x+c;
   }
   void setf2(){ f=&f2; }
   double fsin(double x){
21   return d*sin(x)+c;
   }
   void setfsin(){ f=&fsin; }
   void setx0(){ x0=gr[(iteration-1)*2].y; }

26 void GetGraph(){
       if (getgr){
           getgr=0;
           double x=x0;
           for (int i=0;i<iteration;i++){
31           gr[2*i].x=x;
               x=f(x);
               gr[2*i].y=x;
               gr[2*i+1].x=x;
               gr[2*i+1].y=x;
36       }
       }
       if (getxx){
           getxx=0;
           xx[0].x=xx[0].y=xmin;

```

```

41     xx[1].x=xx[1].y=xmax;
    }
    if (getff){
        getff=0;
        for (int i=0;i<Nf;i++){
46             double x=xmin+i*(xmax-xmin)/(Nf-1);
                ff[i].x=x;
                ff[i].y=f(x);
        }
    }
51 }

void GUI() {
    DefineGraphN_RxR("y=x",&xx[0].x,&two,&getxx);
    DefineGraphN_RxR("y=f(x",&ff[0].x,&Nff,&getff);
56    SetDefaultShape(1);
    SetDefaultColor(2);
    DefineGraphN_RxR("x_n+1=f(x_n",&gr[0].x,&it2,&getgr);
    StartMenu("Sequence_Visulization",1);
    DefineDouble("c",&c);
61    DefineDouble("d",&d);
    DefineDouble("x0",&x0);
    DefineMod("number_of_iterations",&iteration,N);
    DefineFunction("y=d*x^2+c",&setf2);
    DefineFunction("y=d*sin(x)+c",&setfsin);
66    DefineFunction("Set_x0_to_last",&setx0);
    DefineDouble("xmin",&xmin);
    DefineDouble("xmax",&xmax);
    DefineGraph(curve2d_,"Graph");
    DefineBool("done",&done);
71    EndMenu();
}

int main() {
    setf2();
76    GUI();
    while (!done){
        Events(1);
        it2=iteration*2;
        GetGraph();
81    DrawGraphs();
        sleep(1);
    }
}

```

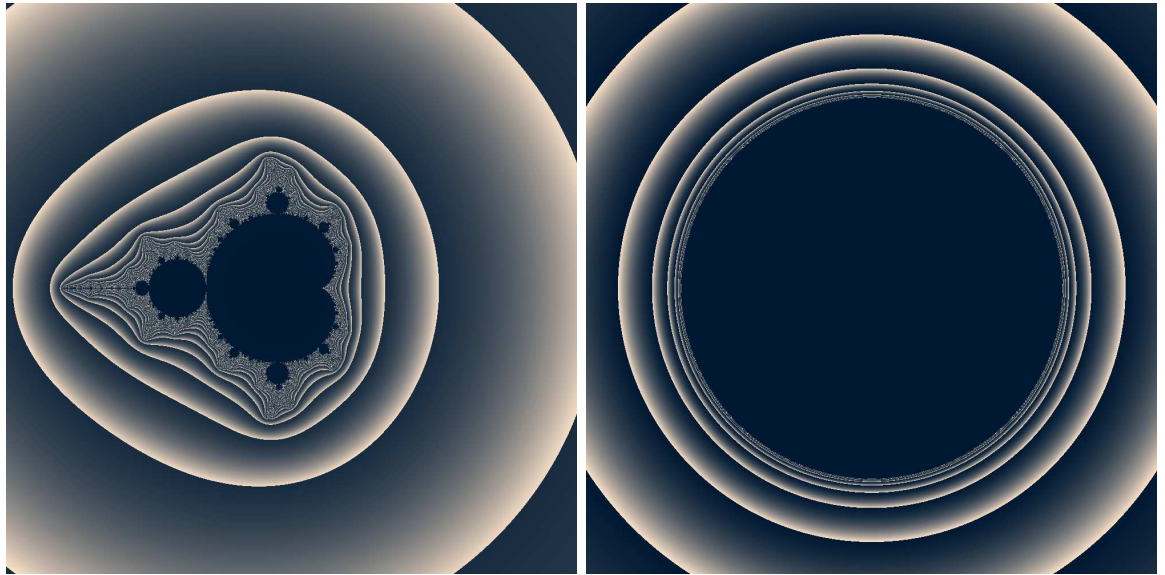


Figure 4.4: A Mandelbrot set around the origin $c = 0$ with a scale factor of 5 and a cutoff radius $|x_n| > 10$ and the corresponding Julia set with a scale factor of 3.

```
}
}
```

Sequences can either converge to a fixed point or a limit cycle, they can fail to converge but remain bounded in a chaotic trajectory, or they can diverge (i.e. grow without limit). For many of these functions it turns out the range of convergence has a tortuous “fractal” boundary. An example for our simple quadratic (interpreting the numbers all as complex values, gives rise to the famous Mandelbrot and Julia sets shown in Figure 4.4.

Now to another computational subject: How do we generate this graphics? Firstly, we should note that C allows for complex variables, which makes the coding easy in this case.

The following is a listing of the complete code that I wrote to generate the images in the following figures:

Listing 4.8: Mandelbrot.c

```
1  /* a short program that shows how one analyzes the convergence
   for the simple iteration  $f(n+1)=f(n)^2+c$ . There are two
   questions one may consider: starting with  $f(0)=0$  where does
   this series lead for different values of  $c$  and given a value
   of  $c$ , where does the series move to for different values of
    $f(0)$ ? */

#include <stdio.h>
```

```

#include <complex.h>
#include <mygraph.h>
6 #define fsize 1000
   double field[fsize][fsize];
   int hund=fsize,MaxIt=100;
   double Far=10;
   complex C0=0,D=0;
11 double ReC=0,ImC=0,ReD=0,ImD=0,mscale=5,jscale=5;
   double f0=0,ffmin=-1,ffmax=1,fimin=-1,fimax=+1;
   complex (*f)(complex,complex);

   complex f2(complex x,complex C){
16   return x*x+C;
   }

   complex fpow(complex x,complex C){
       return D*csin(x)+C;
21 }
   void setfpow(){f=&fpow;}

   complex f3(complex x,complex C){
       return x*x*x+C*x*x+D;
26 }

   complex f3_2(complex x,complex C){
       return x*x*x+D*x*x+C;
   }
31
   void setf2(){
       f=&f2;
   }
   void setf3(){
36   f=&f3;
   }
   void setf3_2(){
       f=&f3_2;
   }
41
   void MB(){
       complex X,C;

       for (int i=0;i<hund;i++)

```

```

46     for (int j=0;j<hund;j++){
        C= ReC+mscale/hund*(-hund/2+i)+(ImC+mscale/hund*(-hund/2+
            j))*I;
        X=0;
        for (int it=0; (it<MaxIt)&&(cabs(X)<Far); it++) X=f(X,C);
        field[i][j]=cabs(X);
51     }
    }

    void Julia(){
        complex X,C;
56     for (int i=0;i<hund;i++)
        for (int j=0;j<hund;j++){
            C=ReC+ImC*I;
            X= jscale/hund*(-hund/2+i)+jscale/hund*(-hund/2+j)*I;
61     for (int it=0; (it<MaxIt)&&(cabs(X)<Far); it++) X=f(X,C);
            field[i][j]=cabs(X);
        }
    }

66 main(){
    int done=0;
    setf2();
    DefineGraphNxN_R("Field",&field[0][0],&hund,&hund,NULL);
    StartMenu("Mandelbrot",1);
71 DefineDouble("Re(C",&ReC);
    DefineDouble("Im(C",&ImC);
    DefineDouble("Re(D",&ReD);
    DefineDouble("Im(D",&ImD);
    DefineDouble("scale_M",&mscale);
76 DefineDouble("scale_J",&jscale);
    DefineInt("MaxIt",&MaxIt);
    DefineDouble("Far",&Far);
    DefineInt("hund",&hund);
    DefineFunction("MB",&MB);
81 DefineFunction("Julia",&Julia);
    DefineGraph(contour2d_,"Field");
    DefineFunction("x^2+c",&setf2);
    DefineFunction("x^3+c",&setf3);
    DefineFunction("x^3+cx^2",&setf3_2);
86 DefineFunction("x^c+c",&setfpow);

```

```

    DefineBool("Done",&done);
    EndMenu();
    while (done==0){
        Events(1);
91    DrawGraphs();
        D=ReD+ImD*I;
    }
}

```

What this code does it iterates the above equation and checks if the value of the numbers is diverging, i.e. if it is larger than some value **Far**. If that is the case, then the iteration is aborted, and a value corresponding to $|x_n|$ is returned. Otherwise we iterate until some maximum number of iterations and conclude that the numbers are not diverging. Again we return the number $|x_n|$. For a Julia set we continue this for different initial values x_0 and record the resulting values in the array `tfield[][]`. For a Mandelbrot set we do this for different values of the constant C , but with the initial condition $x_0 = 0$.

These convergence sets are not limited to the quadratic equation of course. Almost any non-linear equation will do. Two additional equations that I have considered in the code above are

$$x_{n+1} = x_n^2 + C \quad (4.12)$$

$$x_{n+1} = x_n^3 + Cx_n^2 + D \quad (4.13)$$

$$x_{n+1} = x_n^3 + Dx_n^2 + C \quad (4.14)$$

where D is now some arbitrary complex constant that we can set in the program. One important feature of these fractals is that they remain selfsimilar at all smaller lengthscales. This is illustrated in Fig. 4.7.

Problems

4.5.1: Write your own code to generate Julia and Mandelbrot sets.

4.5.2: Now select some non-linear function of your choice, and consider the convergence of the series. Make sure that both converging and non-converging solutions exist.

4.6 Outlook

This ends out brief introduction to programming in C. We will encounter many different kinds of graphs and numerical representations in later chapters. The objects that we

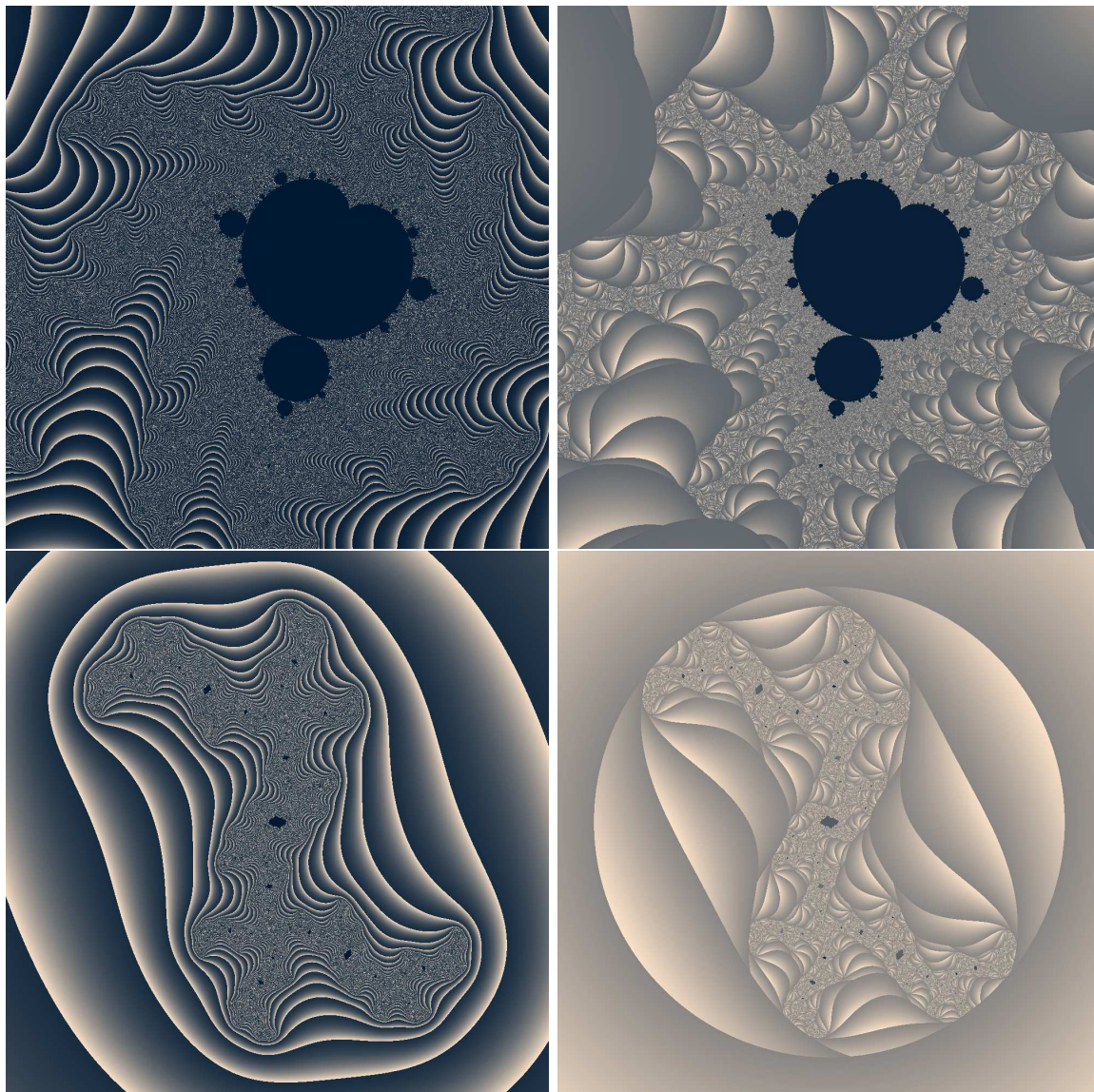


Figure 4.5: A Mandelbrot set around the value $c = 0.386 + 0.569i$ with a scale factor of 0.005. The two graphs show a different cutoff radius for determining the divergence of the series of $|x_n| > 10$ and $|x_n| > 1.2$. In the line below the corresponding Julia set are shown.

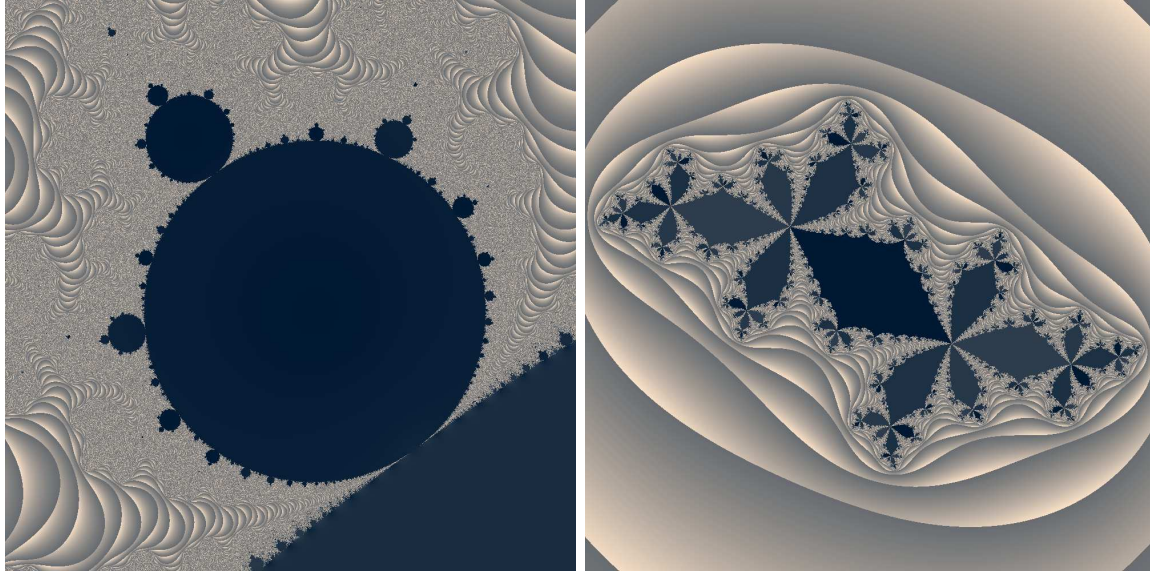


Figure 4.6: A Mandelbrot set around the value $c = -0.51 + 0.569i$ with a scale factor of 0.13 and a cutoff radius $|x_n| > 2$ and the corresponding Julia set.

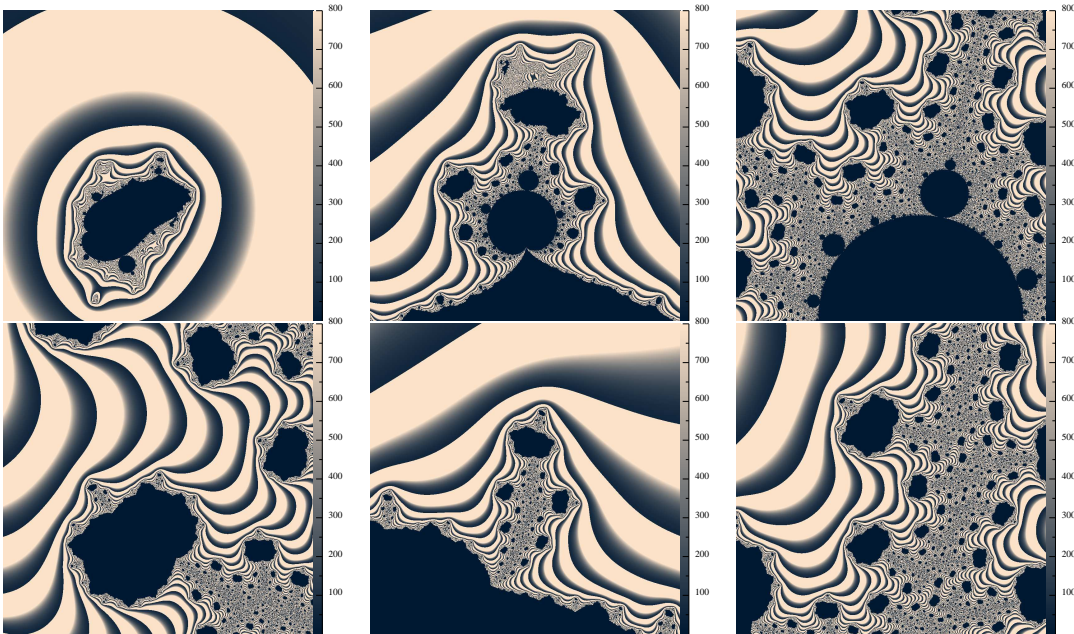


Figure 4.7: Zooming into a small section of a Mandelbrot-like set.

encountered in Figure 4.2 are known as fractals. Such objects have the property of being self-similar, i.e. when you zoom into the bifurcation diagram you can discover many shapes that look very similar to the large picture. In the next chapter we will consider a subject that should be very near and dear to your hearts: Newtonian dynamics, which you have already carefully studied in your first year at NDSU.

Chapter 5

Newtonian dynamics

The bedrock of our understanding of how things move are Newton's equations which were discovered in the 18th century by Sir Isaac Newton. The law you have studied most is Newton's second law stating

$$\frac{d^2\mathbf{x}}{dt^2} = \frac{\mathbf{F}}{m} \quad (5.1)$$

which describes how a point-particle of mass m , located at position \mathbf{x} , which is exposed to the force \mathbf{F} will move. However, since this is a second order differential equation, you also need to know the initial velocity

$$v = \frac{d\mathbf{x}}{dt}. \quad (5.2)$$

To address such a problem numerically, it is much nicer, if it is transformed into a system of first order differential equations:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \quad (5.3)$$

$$\frac{\mathbf{F}}{m} = \frac{d\mathbf{v}}{dt} \quad (5.4)$$

To represent such a problem on a computer we need to *discretize* it. That is we need to find a way of transforming our equation into a discrete equation, much like equation (4.1). We do this, again just as in the bunny problem, by making the time evolution discrete. We say that there is some finite time-step Δt by which we will advance our simulation. We then use a Taylor expansion (which we truncate) to get an expression for the quantity in question at a later time. Let us assume that we know the position and velocity at time t . At time $t + \Delta t$ we then have:

$$\mathbf{v}(t + \delta t) = \mathbf{v}(t) + \frac{d\mathbf{v}(t)}{dt}\Delta t + O(\Delta t^2) \quad (5.5)$$

$$= \mathbf{v}(t) + \frac{\mathbf{F}}{m}\Delta t + O(\Delta t^2) \quad (5.6)$$

and similarly

$$\mathbf{x}(t + \delta t) = \mathbf{x}(t) + \frac{d\mathbf{x}(t)}{dt}\Delta t + O(\Delta t^2) \quad (5.7)$$

$$= \mathbf{x}(t) + \mathbf{v}\Delta t + O(\Delta t^2) \quad (5.8)$$

where the symbol $O(\Delta t^n)$ means that there are terms which are multiplied by Δt^n and higher powers that have been suppressed. If we now make the assumption that $\Delta t \ll 1$ we may hope that we can simply neglect the $O(\Delta t^2)$ terms. If we do that we arrive at the Euler algorithm:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t \quad (5.9)$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{\mathbf{F}}{m}\Delta t \quad (5.10)$$

This is a very robust, if somewhat inaccurate, method of discretizing differential equations. (We will learn about better methods later on).

5.1 The falling ball

When we are faced with the challenging problem of developing a new solution method (i.e. a numerical one) for Newton's equation, we are well advised to start with the simplest problem we can think of. And if this problem has an analytical solution, this will allow us to verify our new method. So we will start with the well worn problem of the falling ball. This is a problem with only one space dimension.

We set up the problem of a ball falling in the standard gravitational field. So the force will be $F = mg$ and we want to find $h(t)$. We all know the theoretical solution $h(t) = \frac{1}{2}gt^2$. To analyze the problem let us look at a ball falling for 1s, and numerically resolve this for a number of different time intervals, say $\Delta t \in (0.1, 0.01, 0.001)$, and then we want to compare this to the analytical solution.

This is a numerical implementation of this problem:

Listing 5.1: Newton1.c

```
1 #include <stdio.h>
   #include <mygraph.h>
   #include <math.h>

   typedef struct part {double x; double v;} part;
6 typedef struct TX {double t; double x;} TX;
   double a=3,k=0.01;
   TX xth[100],x1[11],x2[101],x3[1001];
   int sizeth=100,size1=11,size2=101,size3=1001;
```

```

11  double tmin=0,tmax=1,g=9.81;
    double F(part v){
        return g;
    }

16  double Iterate(part *v, double Dt){
    v->x+=v->v*Dt;
    v->v+=F(*v)*Dt;
    }

21  void GetTrajectory(TX x[],int N,double Dt){
    int i;
    part v;
    x[0].t=tmin;    // t_0 = tmin
    x[0].x=v.x=0;   // x_0 = 0
26  v.v=0;          // v_0 = 0
    for (i=1;i<N;i++){
        Iterate(&v,Dt);
        x[i].t=x[i-1].t+Dt;
        x[i].x=v.x;
31  }
    }

    void Trajectories(){
        int i;
36  double Dt;

        Dt=(tmax-tmin)/(sizeth-1);
        for (i=0;i<sizeth;i++) {
            xth[i].t=tmin+i*Dt;
41  xth[i].x=0.5*g*pow(xth[i].t,2);
        }
        Dt=(tmax-tmin)/(size1-1);
        GetTrajectory(x1,size1,Dt);
        Dt=(tmax-tmin)/(size2-1);
46  GetTrajectory(x2,size2,Dt);
        Dt=(tmax-tmin)/(size3-1);
        GetTrajectory(x3,size3,Dt);
    }

51  main (){
    int i,done=0;

```

```

double N=2;

DefineGraphN_RxR("theory",&(xth[0].t),&sizeh,NULL);
56 DefineGraphN_RxR("x_10",&(x1[0].t),&size1,NULL);
DefineGraphN_RxR("x_100",&(x2[0].t),&size2,NULL);
DefineGraphN_RxR("x_1000",&(x3[0].t),&size3,NULL);
StartMenu("Falling_object",1);
DefineGraph(curve2d_,"Graph");
61 DefineDouble("tmin",&tmin);
DefineDouble("tmax",&tmax);
DefineFunction("Get_Trajectories",&Trajectories);
DefineBool("done",&done);
EndMenu();
66 while (!done){
    Events(1);
    DrawGraphs();
    sleep(1);
}
71 }

```

When we run this code we can compare the results for different values of the discretization. We notice that the results converge to a common curve for small time step Δt . The real test, of course, is to make sure that the curve the algorithm is converging to is actually the correct one. Therefore we also compare the numerical results with the analytical solution.

This is done in Figure 5.1. We see that for $\Delta t = 0.04s$ there is still a noticeable deviation from the theoretical results. For $\Delta t = 0.02$ the difference becomes small and for $\Delta t = 0.01$ the result is nearly indistinguishable from the analytical solution, at least as far as the graphical resolution here is concerned.

To get a better grasp on the convergence it would be helpful to examine a measure of the error and how this error diminishes as Δt is decreased. We simply measure

$$error = h_{\Delta t}(t) - h(t) \quad (5.11)$$

The results of this measurement are shown in Figure 5.2. We see that as we decrease the timestep Δt by a factor of two, the error also decreases by a factor of two. Mathematically this is called linear convergence.

At first one might think that this is quite sufficient. For the simple example of a falling ball we would find it easy to set the timestep Δt small enough to follow the ball quite a distance.

For long term simulations, however, this may not be good enough. If we follow the total energy of the system

$$\frac{E(t)}{m} = \frac{1}{2}v(t)^2 - h(t)g \quad (5.12)$$

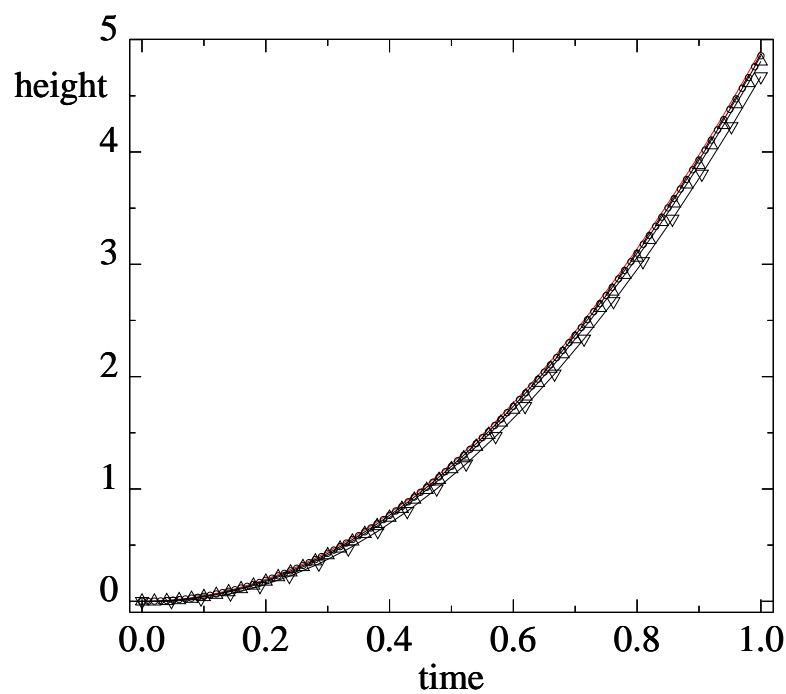


Figure 5.1: The numerical solution of the position of our particle as a function of time simulated with the Euler method using two different timesteps of 0.1 and 0.01, compared to the analytical solution.

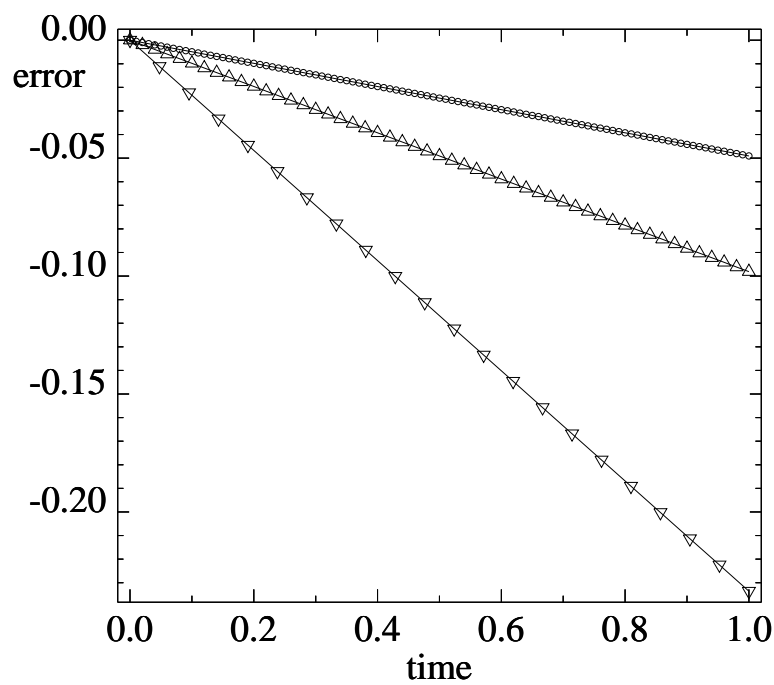


Figure 5.2: Error for the different discretizations of 25, 50, and 100 points.

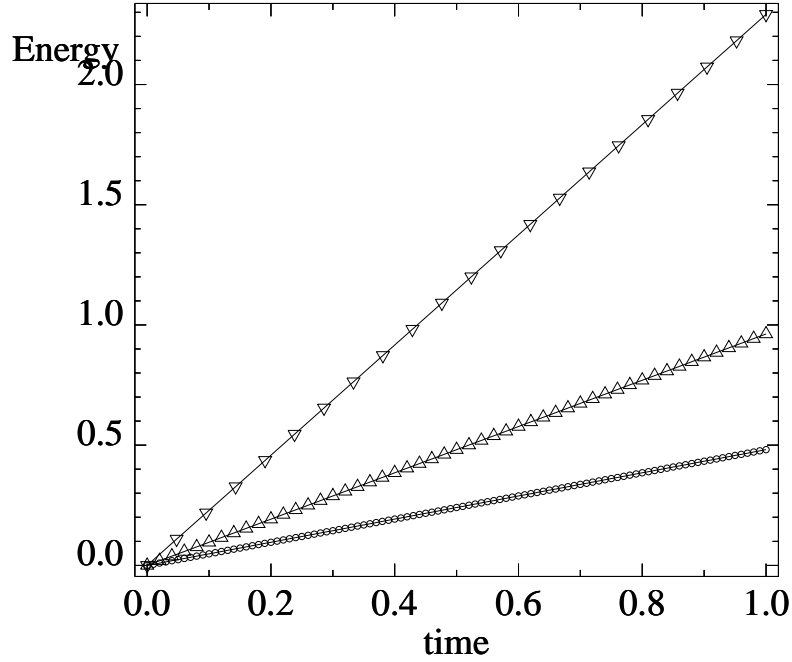


Figure 5.3: Total energy for the different discretizations of 25,50, and 100 points.

(which should be a constant) we find the results of Figure 5.3. For all these discretizations the energy continues to increase. We can reduce the rate of energy increase by decreasing the timestep Δt , but if we want to follow a system for a long time, this is going to cause some trouble. Also the regularity with which the energy increases, suggests that we are making a systematic error. We will return to this issue in a moment.

5.2 A simple oscillator

Let us now consider another common system: a particle in a harmonic potential. Now we have an external potential $V(x) = kx^2/2$ so we have

$$\frac{F}{m} = -\frac{1}{m} \frac{dV}{dx} = -\frac{k}{m}x \quad (5.13)$$

and Newton's equations become

$$\dot{x} = v \quad (5.14)$$

$$\dot{v} = -\frac{k}{m}x. \quad (5.15)$$

The analytical solution of a particle starting at time $t = 0$ at $x(0) = 0$ with initial velocity $\dot{x}(t = 0) = v_0$ is, of course, a harmonic:

$$x(t) = v_0 \sqrt{\frac{m}{k}} \sin \left(\sqrt{\frac{k}{m}} t \right) \quad (5.16)$$

Now we need to extend our algorithm to include a position dependent forcing term. A code may look like this:

Listing 5.2: Newton3.c

```

#include <stdio.h>
#include <mygraph.h>
#include <math.h>
4
typedef struct XV {double x; double v;} XV;
typedef struct TX {double t; double x;} TX;

double m=1,k=1,v0=1;
9 TX xth[100],x1[11],x2[101],x3[1001];
int sizeth=100,size1=11,size2=101,size3=1001;
int errmeasure=0; // Flag to determine if one should measure
the error
double tmin=0,tmax=1,g=9.81;

14 double F(double x){
    return -k/m*x;
}

double Iterate(XV *v, double Dt){
19 v->x+=v->v*Dt;
v->v+=F(v->x)*Dt;
}

void GetTrajectory(TX x[],int N,double Dt){
24 int i;
XV v;
x[0].t=tmin; // t_0 = tmin
x[0].x=v.x=0; // x_0 = 0
v.v=v0; // v_0
29 for (i=1;i<N;i++){
    Iterate(&v,Dt);
    x[i].t=x[i-1].t+Dt;
    if (errmeasure) x[i].x=v.x-v0*sqrt(m/k)*sin(sqrt
(k/m)*x[i].t);
    else x[i].x=v.x;
34 }
}

void Trajectories(){

```

```

    int i;
39    double Dt;

    Dt=(tmax-tmin)/(sizeth-1);
    for (i=0;i<sizeth;i++) {
        xth[i].t=tmin+i*Dt;
44    if (errmeasure) xth[i].x=0;
        else xth[i].x=v0*sqrt(m/k)*sin(sqrt(k/m)*xth[i].t);
    }
    Dt=(tmax-tmin)/(size1-1);
    GetTrajectory(x1,size1,Dt);
49    Dt=(tmax-tmin)/(size2-1);
    GetTrajectory(x2,size2,Dt);
    Dt=(tmax-tmin)/(size3-1);
    GetTrajectory(x3,size3,Dt);
}

54 main () {
    int i,done=0;
    double N=2;

59    DefineGraphN_RxR("theory",&(xth[0].t),&sizeth,NULL);
    DefineGraphN_RxR("x_10",&(x1[0].t),&size1,NULL);
    DefineGraphN_RxR("x_100",&(x2[0].t),&size2,NULL);
    DefineGraphN_RxR("x_1000",&(x3[0].t),&size3,NULL);
    StartMenu("Object_on_spring",1);
64    DefineDouble("m",&m);
    DefineDouble("k",&k);
    DefineDouble("v0",&v0);
    DefineGraph(curve2d_,"Graph");
    DefineDouble("tmin",&tmin);
69    DefineDouble("tmax",&tmax);
    DefineFunction("Get_Trajectories",&Trajectories);
    DefineBool("Error_measure",&errmeasure);
    DefineBool("done",&done);
    EndMenu();
74    while (!done){
        Events(1);
        DrawGraphs();
        sleep(1);
    }
79 }

```

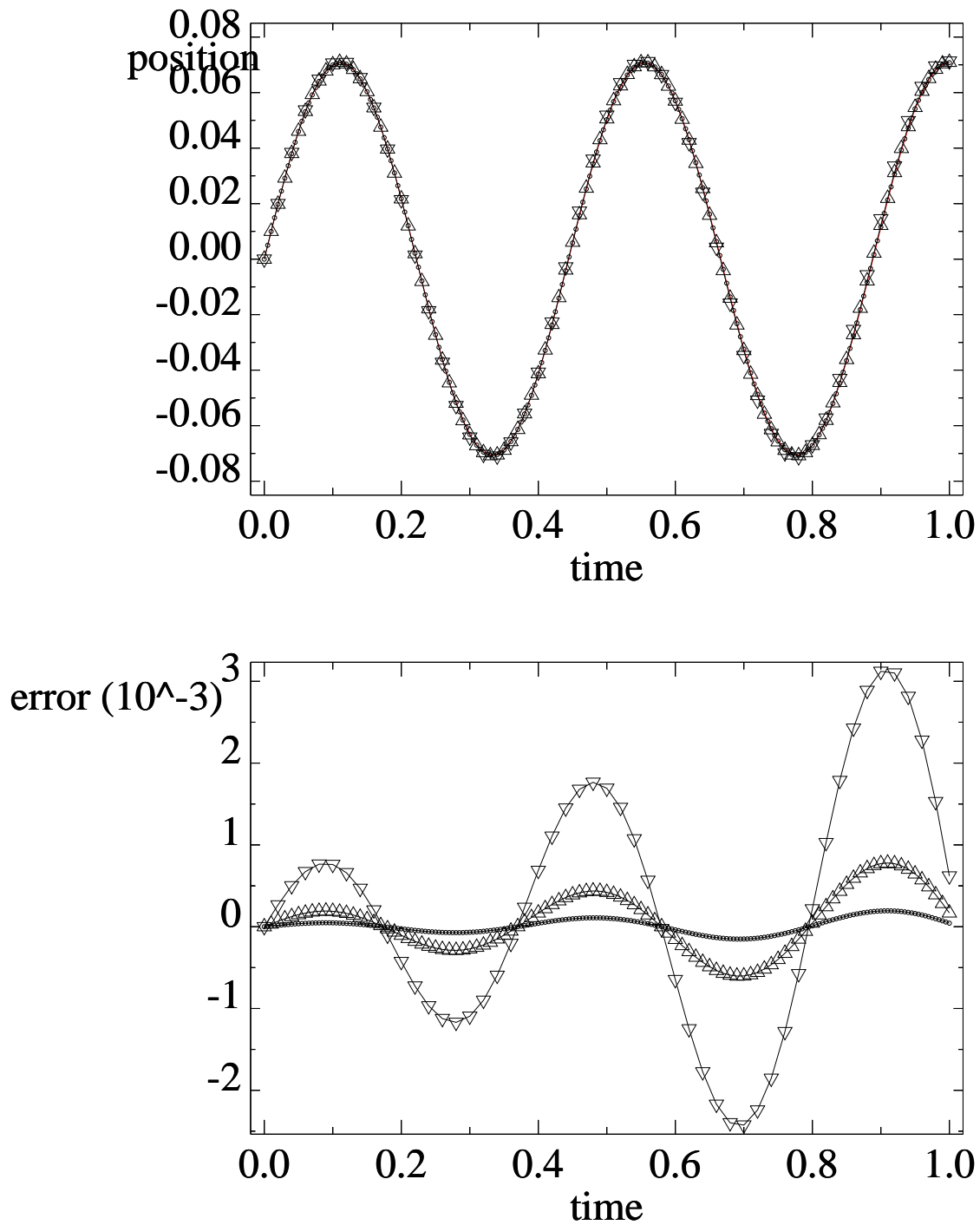


Figure 5.4: The position of a particle in an harmonic oscillator. The results are rather promising, and we get excellent agreement with the theoretical results, even for a poor resolution. However, the error continues to accrue.

We show our numerical results in Figure 5.4. The results are surprisingly good. The errors are very small, but as we can see in the bottom graph, the error is increasing linearly with time. If we want to follow our particle in the harmonic potential for a long time, this will likely spell trouble.

5.3 Two dimensional projectile motion

We can now look at the problem of two dimensional projectile motion. To make the problem more interesting, we can write it such that the force already includes air-friction, so that we can test the code with our analytical solution for the case of no friction and then observe the more interesting case of a projectile with friction:

$$\mathbf{F} = -mg\mathbf{e}_y - \alpha\mathbf{v}|\mathbf{v}| \quad (5.17)$$

where \mathbf{e}_y is a unit vector in the y-direction.

This time we have many functions we may want to observe. We want to be able to display $x(t)$, $y(t)$, $y(x)$, $v_x(t)$, $v_y(t)$, and $E(t)$. The program below implements all these quantities as well as the simple Euler integrator for Newton's equations:

Listing 5.3: Projectile.c

```

1 #include <stdio.h>
  #include <mygraph.h>
  #include <math.h>
  #include <unistd.h>

6 #define pi
   3.1415926535897932384626433832795028841971693993751058209749445

   double m=1,g=9.81,v_in=1,theta=45,alpha=0;
#define NG 6 /* Graphs for display: x(t) y(t) y(x) v_x(t) v_y(t)
   ) E(t) */
   double xth[NG][100][2], x1[NG][51][2], x2[NG][101][2], x3[NG]
   ][201][2];
11 int sizeth=100,size1=51,size2=101,size3=201;
   int errmeasure=0,EnergyMeasure=0; /* Flag to determine if one
   should measure the error
   double tmax=1;

   void FF(double v[4], double F[2]){
16   double vabs=sqrt(v[2]*v[2]+v[3]*v[3]); /* absolute value of $
   \vec{v}$ */
   F[0]=-alpha*vabs*v[2];

```

```

    F[1]=-alpha*vabs*v[3]-m*g;
}

21 void Iterate(double v[4], double Dt){
    double F[2];
    v[0]+=v[2]*Dt;
    v[1]+=v[3]*Dt;
    FF(v,F);
26   v[2]+=F[0]/m*Dt;
    v[3]+=F[1]/m*Dt;
}

void GetTrajectory(double x0[][2],double x1[][2],double x2
    [[2],double x3[][2],double x4[][2],double x5[][2],int N,
    double Dt){
31   int i;
    double v[4];

    v[0]=0; // x_0 = 0
    v[1]=0; // y_0 = 0
36   v[2]=v_in*cos(theta/180*pi);
    v[3]=v_in*sin(theta/180*pi);

    for (i=0;i<N;i++){

41       x0[i][0]=x1[i][0]=x3[i][0]=x4[i][0]=x5[i][0]=i*Dt; // t
        x0[i][1]=x2[i][0]=v[0]; // x
        x1[i][1]=x2[i][1]=v[1]; // y
        x3[i][1]=v[2]; // v_x
        x4[i][1]=v[3]; // v_x
46       x5[i][1]=0.5*m*(pow(v[2],2)+pow(v[3],2))+m*g*v[1];
        // E
        Iterate(v,Dt);
    }
}

51 void Trajectories(){
    int i;
    double Dt;

    Dt=(tmax)/(sizeth-1);
56   for (i=0;i<sizeth;i++) {

```

```

    xth[0][i][0]=i*Dt;
    xth[0][i][1]=i*Dt*v_in*cos(theta/180*pi);
    xth[1][i][0]=i*Dt;
    xth[1][i][1]=i*Dt*v_in*sin(theta/180*pi)-0.5*g*pow(i*Dt,2);
61  xth[2][i][0]=i*Dt*v_in*cos(theta/180*pi);
    xth[2][i][1]=i*Dt*v_in*sin(theta/180*pi)-0.5*g*pow(i*Dt,2);
    xth[3][i][0]=i*Dt;
    xth[3][i][1]=v_in*cos(theta/180*pi);
    xth[4][i][0]=i*Dt;
66  xth[4][i][1]=v_in*sin(theta/180*pi)-g*i*Dt;
    xth[5][i][0]=i*Dt;
    xth[5][i][1]=0.5*m*pow(v_in,2);
}
Dt=(tmax)/(size1-1);
71  GetTrajectory(x1[0],x1[1],x1[2],x1[3],x1[4],x1[5],size1,Dt);
    Dt=(tmax)/(size2-1);
    GetTrajectory(x2[0],x2[1],x2[2],x2[3],x2[4],x2[5],size2,Dt);
    Dt=(tmax)/(size3-1);
    GetTrajectory(x3[0],x3[1],x3[2],x3[3],x3[4],x3[5],size3,Dt);
76 }

main () {
    int i,done=0;
    double N=2;
81
    DefineGraphN_RxR("x(t)_th",&(xth[0][0][0]),&size1,NULL);
    DefineGraphN_RxR("x(t)_10",&(x1[0][0][0]),&size1,NULL);
    DefineGraphN_RxR("x(t)_100",&(x2[0][0][0]),&size2,NULL);
    DefineGraphN_RxR("x(t)_1000",&(x3[0][0][0]),&size3,NULL);
86
    DefineGraphN_RxR("y(t)_th",&(xth[1][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(t)_10",&(x1[1][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(t)_100",&(x2[1][0][0]),&size2,NULL);
    DefineGraphN_RxR("y(t)_1000",&(x3[1][0][0]),&size3,NULL);
91
    DefineGraphN_RxR("y(x)_th",&(xth[2][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(x)_10",&(x1[2][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(x)_100",&(x2[2][0][0]),&size2,NULL);
    DefineGraphN_RxR("y(x)_1000",&(x3[2][0][0]),&size3,NULL);
96
    DefineGraphN_RxR("v_x(t)_th",&(xth[3][0][0]),&size1,NULL);
    DefineGraphN_RxR("v_x(t)_10",&(x1[3][0][0]),&size1,NULL);

```

```

DefineGraphN_RxR("v_x(t)_100",&(x2[3][0][0]),&size2,NULL);
DefineGraphN_RxR("v_x(t)_1000",&(x3[3][0][0]),&size3,NULL);
101
DefineGraphN_RxR("v_y(t)_th",&(xth[4][0][0]),&sizeth,NULL);
DefineGraphN_RxR("v_y(t)_10",&(x1[4][0][0]),&size1,NULL);
DefineGraphN_RxR("v_y(t)_100",&(x2[4][0][0]),&size2,NULL);
DefineGraphN_RxR("v_y(t)_1000",&(x3[4][0][0]),&size3,NULL);
106
DefineGraphN_RxR("E(t)_th",&(xth[5][0][0]),&sizeth,NULL);
DefineGraphN_RxR("E(t)_10",&(x1[5][0][0]),&size1,NULL);
DefineGraphN_RxR("E(t)_100",&(x2[5][0][0]),&size2,NULL);
DefineGraphN_RxR("E(t)_1000",&(x3[5][0][0]),&size3,NULL);
111

StartMenu("Falling_object",1);
DefineDouble("m",&m);
116 DefineDouble("alpha",&alpha);
DefineDouble("theta",&theta);
DefineDouble("v_in",&v_in);
DefineGraph(curve2d_,"Graph");
DefineDouble("tmax",&tmax);
121 DefineBool("Measure_Errors",&errmeasure);
DefineFunction("Get_Trajectories",&Trajectories);
DefineBool("done",&done);
EndMenu();
while (!done){
126   Events(1);
   DrawGraphs();
   sleep(1);
}
}

```

Problems

- 5.3.1:** Examine $v(t)$ for the case of a projectile with air-friction. How does this result differ from the case without air friction? Can you make an analytical prediction for the final velocity (if there is no bottom)? Does the numerical result agree?
- 5.3.2:** How far will a baseball go if it leaves the bat with 70 mph?(look up the friction coefficient for a baseball)

5.3.3: What is the optimal angle for the baseball to be launched at to make it travel as far as possible?

5.4 Three dimensional motion

There is nothing peculiar about problems in three (or more) dimensions, but we need to consider ways of analysing such problems. One way of analyzing three dimensional problems is to examine two-dimensional projections, which is, in fact, all you can do on a two dimensional screen. However, we should keep in mind that all humans can perceive about three dimensional objects stems from the information that reaches the retina of their eyes, which is also two dimensional. Depth perception is generated in a number of ways, some of which we can reproduce on a computer.

The first way to generate depth information is by using *two* eyes. You will notice that by holding a hand in front of your eye you are suddenly loosing depth perception. However, your brain is rather good at camouflaging this deficiency, so you may not notice right away. If we are able to generate two separate images on your two eyes, we can calculate them in such a way as to generate the illusion of a three dimensional object. There are different ways of doing this, but the easiest way of achieving this with a normal setup is by using two-colored glasses.

Chapter 6

Numerical Algorithms

We have talked about using the Euler algorithm to evaluate the evolution of a particle described by Newton's equations. There is nothing in principle that forced us to stick to one particle, and we will soon look at many particle systems. But we already noticed that the Euler algorithm, while converging towards the correct solution for $\Delta t \rightarrow 0$, was not very good at conserving energy. In this chapter we want to analyze how one can improve on this simple Euler algorithm to get a better performance for less computational work. We will see that we can do that by being clever about the discretization of Newton's equations, or ordinary differential equations in general.¹

So we are looking here at the question how to best discretize a set of equations of the form

$$\dot{x} = v \tag{6.1}$$

$$\dot{v} = \frac{F(x, v)}{m} \tag{6.2}$$

where x, v , and $F(x, v)$ are a $3N$ -dimensional vectors for a system with N particles.

6.1 The Euler algorithm

The Euler algorithm of equation (5.10) can be written as

$$x(t + \Delta t) = x(t) + v(t)\Delta t \tag{6.3}$$

$$v(x + \Delta t) = v(t) + \frac{F[x(t), v(t)]}{m}\Delta t \tag{6.4}$$

So what differential equation does this discrete scheme actually solve? It can't really be Newton's equations, since otherwise the energy should be conserved if we only have

¹Please also look at appendix 3A of the book. (It appears that they borrowed this summary themselves from here)

conservative forces. In other words, what can we find out about the errors by looking at this equation?

To answer that question we have to realize that we can write $x(t + \Delta t)$ as a Taylor series. With that we get for the Euler discretization

$$\sum_{n=1}^{\infty} \frac{1}{n!} \frac{d^n x(t)}{dt^n} (\Delta t)^{n-1} = v(t) \quad (6.5)$$

$$\sum_{n=1}^{\infty} \frac{1}{n!} \frac{d^n v(x)}{dt^n} (\Delta t)^{n-1} = \frac{F[x(t), v(t)]}{m} \quad (6.6)$$

So the leading order error for the position is of the form

$$\frac{1}{2} \frac{d^2 x(t)}{dt^2} \Delta t = \frac{1}{2} \frac{dv(t)}{dt} \Delta t + O[(\Delta t)^2] \quad (6.7)$$

$$= \frac{F[x(t), v(t)]}{2m} \Delta t + O[(\Delta t)^2] \quad (6.8)$$

and for the leading order velocity error we have

$$\frac{1}{2} \frac{d^2 v(t)}{dt^2} \Delta t \quad (6.9)$$

$$= \frac{1}{2m} \frac{dF[x(t), v(t)]}{dt} \Delta t + O[(\Delta t)^2] \quad (6.10)$$

$$= \frac{1}{2m} \left\{ \frac{\partial F[x(t), v(t)]}{\partial x} \frac{dx(t)}{dt} + \frac{\partial F[x(t), v(t)]}{\partial v} \frac{dv(t)}{dt} \right\} \Delta t + O[(\Delta t)^2] \quad (6.11)$$

$$= \frac{1}{2m} \left\{ \frac{\partial F[x(t), v(t)]}{\partial x} v(t) + \frac{\partial F[x(t), v(t)]}{\partial v} \frac{F[x(t), v(t)]}{m} \right\} \Delta t + O[(\Delta t)^2]. \quad (6.12)$$

Here we used the order sign $O[(\Delta t)^n]$ which means that we have neglected terms that have pre-factors of $(\Delta t)^n$ and higher powers. It is remarkable that with the help of (6.8) and (6.12) we have been able to represent the error terms in the form of quantities that we already know.

6.2 A second order method

Now that we know the error terms which appear in the second order, we can actively subtract them from the Euler method, to obtain a method where second order errors are removed:

$$x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{F[x(t), v(t)]}{2m} (\Delta t)^2 \quad (6.13)$$

$$\begin{aligned} v(x + \Delta t) = & v(t) + \frac{F[x(t), v(t)]}{m} \Delta t \\ & + \frac{1}{2m} \left\{ \frac{\partial F[x(t), v(t)]}{\partial x} v(t) + \frac{\partial F[x(t), v(t)]}{\partial v} \frac{F[x(t), v(t)]}{m} \right\} (\Delta t)^2 \end{aligned} \quad (6.14)$$

Following the analysis above, this method does only have leading order errors which go as $(\Delta t)^3$. Let us see what this choice means practically: we can also write it as

$$x(t + \Delta t) = x(t) + v(t + \Delta t/2)\Delta t + O(\Delta t^3) \quad (6.15)$$

$$v(x + \Delta t) = v(t) + \frac{F[x(t + \Delta t/2), v(t + \Delta t/2)]}{m}\Delta t + O(\Delta t^3), \quad (6.16)$$

i.e. we are effectively using the average velocity between the two time-steps to advance the position and we are using (approximately, since another Taylor expansion is involved) the average force between the two time-steps to advance the velocity.

Another aspect to consider is that while the continuous Newton's equations are reversible, i.e. if we invert time and all velocities then we will come back to our initial state (at least as long as the force is conservative and, preferably, not dependent on velocity). This is not true for the discrete version since the evolution between two time points t and $t + \Delta t$ only depends on the state of the system at time t . But for the new representation this is now true. In the form of equation (6.16) this is exactly true, but for the actual algorithm proposed here in (6.14) it may only be approximately true (depending on the form of the forcing term). Note that the representation of (6.16) has the disadvantage, that we don't know how to explicitly implement this, as we don't know how to obtain the position and velocity half-way between two time-steps. Several approximations for this exist, all of them accurate to second order. For the algorithm proposed in (6.14) one has to calculate the derivative matrix for the force, which can be a bit tedious. Another popular choice is the so-called velocity Verlet algorithm which is given by

$$x(t + \Delta t) = x(t) + \left(v(t) + \frac{1}{2m}F(t)\Delta t \right) \Delta t \quad (6.17)$$

$$v(x + \Delta t) = v(t) + \frac{F[x(t), v(t)] + F\{x(t + \Delta t), v(t) + F[x(t), v(t)]\Delta t\}}{2m}\Delta t, \quad (6.18)$$

which is also accurate to second order.

Now it is time to implement these algorithms and test their performance. A sample implementation that allows you to compare the Euler, Derivative, and Verlet algorithms is given in Projectile2.c:

Listing 6.1: Projectile2.c

```
#include <stdio.h>
#include <mygraph.h>
#include <math.h>

5 #define pi 3.14159265358979323846264338327950288419716939937

double m=1,g=9.81,v_in=1,theta=45,alpha=0;
#define NG 6 /* Graphs for display: x(t) y(t) y(x) v_x(t) v_y(t)
           ) E(t)*/
```

```

double xth[NG][100][2], x1[NG][51][2], x2[NG][101][2], x3[NG]
    ][201][2];
10 int sizeth=100,size1=51,size2=101,size3=201;
int errmeasure=0,EnergyMeasure=0; // Flag to determine if one
    should measure the error
double tmin=0,tmax=1;

void FF(double v[4], double F[2]){
15 double vabs=sqrt(v[2]*v[2]+v[3]*v[3]); /* absolute value of $
    \vec{v}$ */
    F[0]=-alpha*vabs*v[2];
    F[1]=-alpha*vabs*v[3]-m*g;
}

20 void dFFx(double v[4], double dF[2][4]){ /* Gradient of the
    Force */
    double vabs=sqrt(v[2]*v[2]+v[3]*v[3]); /* absolute value of $
    \vec{v}$ */

    dF[0][0]=0; dF[0][1]=0; dF[0][2]=-alpha*(2*v[2]*v[2]+v[3]*v
        [3])/vabs; dF[0][3]=-alpha*v[2]*v[3]/vabs;
    dF[1][0]=0; dF[1][1]=0; dF[1][2]=-alpha*v[2]*v[3]/vabs; dF
        [1][3]=-alpha*(v[2]*v[2]+2*v[3]*v[3])/vabs;
25 }

void (*Iterate)(double v[4], double Dt)=NULL;

void IterateEuler(double v[4], double Dt){
30 double F[2];
    FF(v,F);
    v[0]+=v[2]*Dt;
    v[1]+=v[3]*Dt;
    v[2]+=F[0]*Dt/m;
35 v[3]+=F[1]*Dt/m;
}

void IterateDeriv(double v[4], double Dt){
    double F[2],dF[2][4];
40 FF(v,F);
    dFFx(v,dF);
    v[0]+=(v[2]+0.5*F[0]*Dt)*Dt;
    v[1]+=(v[3]+0.5*F[1]*Dt)*Dt;

```

```

    v[2]+=(F[0]+0.5/m*(dF[0][0]*v[2]+dF[0][1]*v[3]+dF[0][2]*F[0]/
        m+dF[0][3]*F[1]/m)*Dt)*Dt/m;
45    v[3]+=(F[1]+0.5/m*(dF[1][0]*v[2]+dF[1][1]*v[3]+dF[1][2]*F[0]/
        m+dF[1][3]*F[1]/m)*Dt)*Dt/m;
}

void IterateVerlet(double v[4], double Dt){
    double F[2],Fn[2];
50    FF(v,F);
    v[0]+=(v[2]+0.5*F[0]*Dt)*Dt;
    v[1]+=(v[3]+0.5*F[1]*Dt)*Dt;
    v[2]+=F[0]*Dt/m;
    v[3]+=F[1]*Dt/m;
55    FF(v,Fn);
    v[2]+=0.5*(Fn[0]-F[0])*Dt/m;
    v[3]+=0.5*(Fn[1]-F[1])*Dt/m;
}

60 void SetEuler(){
    Iterate = &IterateEuler;
}

void SetDeriv(){
65    Iterate = &IterateDeriv;
}

void SetVerlet(){
    Iterate = &IterateVerlet;
70 }

void GetTrajectory(double x0[][2],double x1[][2],double x2
    [[2],double x3[][2],double x4[][2],double x5[][2],int N,
    double Dt){
    int i;
    double v[4];
75    v[0]=0; // x_0 = 0
    v[1]=0; // y_0 = 0
    v[2]=v_in*cos(theta/180*pi);
    v[3]=v_in*sin(theta/180*pi);

80    x0[0][0]=x1[0][0]=x3[0][0]=x4[0][0]=x5[0][0]=0; // t
    x0[0][1]=x2[0][0]=v[0]; // x

```

```

x1[0][1]=x2[0][1]=v[1];    // y
x3[0][1]=v[2];             // v_x
x4[0][1]=v[3];             // v_x
85  x5[0][1]=0.5*m*(pow(v[2],2)+pow(v[3],2))+m*g*v[1];    //
    E

    for (i=1;i<N;i++){
        Iterate(v,Dt);

90      x0[i][0]=x1[i][0]=x3[i][0]=x4[i][0]=x5[i][0]=i*Dt;    // t
        x0[i][1]=x2[i][0]=v[0];    // x
        x1[i][1]=x2[i][1]=v[1];    // y
        x3[i][1]=v[2];             // v_x
        x4[i][1]=v[3];             // v_x
95      x5[i][1]=0.5*m*(pow(v[2],2)+pow(v[3],2))+m*g*v[1];
        // E
    }
}

void Trajectories(){
100  int i;
    double Dt;

    Dt=(tmax-tmin)/(sizeth-1);
    for (i=0;i<sizeth;i++) {
105      xth[0][i][0]=i*Dt;
        xth[0][i][1]=i*Dt*v_in*cos(theta/180*pi);
        xth[1][i][0]=i*Dt;
        xth[1][i][1]=i*Dt*v_in*sin(theta/180*pi)-0.5*g*pow(i*Dt,2);
        xth[2][i][0]=i*Dt*v_in*cos(theta/180*pi);
110      xth[2][i][1]=i*Dt*v_in*sin(theta/180*pi)-0.5*g*pow(i*Dt,2);
        xth[3][i][0]=i*Dt;
        xth[3][i][1]=v_in*cos(theta/180*pi);
        xth[4][i][0]=i*Dt;
        xth[4][i][1]=v_in*sin(theta/180*pi)-g*i*Dt;
115      xth[5][i][0]=i*Dt;
        xth[5][i][1]=0.5*m*pow(v_in,2);
    }
    Dt=(tmax-tmin)/(size1-1);
    GetTrajectory(x1[0],x1[1],x1[2],x1[3],x1[4],x1[5],size1,Dt);
120  Dt=(tmax-tmin)/(size2-1);
    GetTrajectory(x2[0],x2[1],x2[2],x2[3],x2[4],x2[5],size2,Dt);

```

```

    Dt=(tmax-tmin)/(size3-1);
    GetTrajectory(x3[0],x3[1],x3[2],x3[3],x3[4],x3[5],size3,Dt);
}
125
main () {
    int i,done=0;
    double N=2;

130    SetEuler();

    DefineGraphN_RxR("x(t)_th",&(xth[0][0][0]),&size1,NULL);
    DefineGraphN_RxR("x(t)_10",&(x1[0][0][0]),&size1,NULL);
    DefineGraphN_RxR("x(t)_100",&(x2[0][0][0]),&size2,NULL);
135    DefineGraphN_RxR("x(t)_1000",&(x3[0][0][0]),&size3,NULL);

    DefineGraphN_RxR("y(t)_th",&(xth[1][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(t)_10",&(x1[1][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(t)_100",&(x2[1][0][0]),&size2,NULL);
140    DefineGraphN_RxR("y(t)_1000",&(x3[1][0][0]),&size3,NULL);

    DefineGraphN_RxR("y(x)_th",&(xth[2][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(x)_10",&(x1[2][0][0]),&size1,NULL);
    DefineGraphN_RxR("y(x)_100",&(x2[2][0][0]),&size2,NULL);
145    DefineGraphN_RxR("y(x)_1000",&(x3[2][0][0]),&size3,NULL);

    DefineGraphN_RxR("v_x(t)_th",&(xth[3][0][0]),&size1,NULL);
    DefineGraphN_RxR("v_x(t)_10",&(x1[3][0][0]),&size1,NULL);
    DefineGraphN_RxR("v_x(t)_100",&(x2[3][0][0]),&size2,NULL);
150    DefineGraphN_RxR("v_x(t)_1000",&(x3[3][0][0]),&size3,NULL);

    DefineGraphN_RxR("v_y(t)_th",&(xth[4][0][0]),&size1,NULL);
    DefineGraphN_RxR("v_y(t)_10",&(x1[4][0][0]),&size1,NULL);
    DefineGraphN_RxR("v_y(t)_100",&(x2[4][0][0]),&size2,NULL);
155    DefineGraphN_RxR("v_y(t)_1000",&(x3[4][0][0]),&size3,NULL);

    DefineGraphN_RxR("E(t)_th",&(xth[5][0][0]),&size1,NULL);
    DefineGraphN_RxR("E(t)_10",&(x1[5][0][0]),&size1,NULL);
    DefineGraphN_RxR("E(t)_100",&(x2[5][0][0]),&size2,NULL);
160    DefineGraphN_RxR("E(t)_1000",&(x3[5][0][0]),&size3,NULL);

```

```

    StartMenu("Falling_object",1);
165   DefineDouble("m",&m);
    DefineDouble("alpha",&alpha);
    DefineDouble("theta",&theta);
    DefineDouble("v_in",&v_in);
    DefineGraph(curve2d_,"Graph");
170   DefineDouble("tmin",&tmin);
    DefineDouble("tmax",&tmax);
    DefineBool("Measure_Errors",&errmeasure);
    DefineFunction("Get_Trajectories",&Trajectories);
    DefineFunction("Set_Euler",&SetEuler);
175   DefineFunction("Set_Deriv",&SetDeriv);
    DefineFunction("Set_Verlet",&SetVerlet);
    DefineBool("done",&done);
    EndMenu();
    while (!done){
180     Events(1);
        DrawGraphs();
        sleep(1);
    }
}

```

A few notes on the code. There is a function `Iteration()` that is nowhere explicitly defined. Instead it is a variable function that gets assigned in the routines `SetEuler()`, `SetDeriv()`, and `SetVerlet()`. Depending on which routine this function pointer is set to, a different algorithm will be used for the iteration.

Problems

6.2.1: Given the simple nature of the problem of Projectile motion we found that the Derivative and Verlet algorithm are actually identical for this system. Show mathematically that this is indeed the case.

6.2.2: Consider the Felix Baumgartner's record of the highest free-fall achieved on October 14, 2012 (see <http://www.sciencedaily.com/releases/2012/10/121014170655.htm>). If you wanted to advise him on his fall, what additional Physics would you have to take into account to predict his free-fall trajectory? Describe which additional effects you considered and whether you judge these effects to be potentially relevant for predicting the fall trajectory. Then implement all of the relevant corrections into our problem and predict the fall of Felix Baumgartner. How does your predicted fall time and maximum velocity agree with the measured values?

Chapter 7

Particles in a box and arbitrary graphics

So far we have discussed rather standard physics problems that could be easily visualized using a few standard data types. However, sometimes problems don't fit these generic representations. As an example let us look at the simple problem of particles of a certain size in a box with additional obstacles. It is fairly obvious what we would a visualization to look like in this case: we want circles of the size of the particles moving in a window which also contains a representation of the walls. So all we need are lines and circles, drawn at the appropriate positions.

The graphics library allows for such representations in an easy way: you can define a graph of type `freedraw` that gives you access to the primitive graphics elements. Here is an example code:

Listing 7.1: `freedraw.c`

```
1 #include <unistd.h>
#include <mygraph.h>
int xoffs=0,yoffs=0;

void Draw(int xdim, int ydim){
6   myselectfont(0,"I_can_write_text!",xdim*0.3);
   mydrawline(1,0,0,xdim,ydim);
   mycircle(2,xdim/2+xoffs,ydim/2+yoffs,0.2*xdim);
   mytext(4,xdim/2+xoffs,ydim/2+yoffs,"I_can_write_text!",1);
}
11
void main (){
   int done=0;
   AddFreedraw("line_and_circle",&Draw);
   StartMenu("Freedraw",1);
16  DefineInt("xoffs",&xoffs);
```

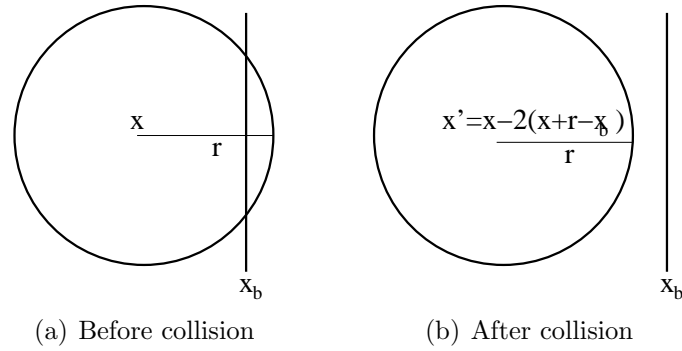


Figure 7.1: Sketch for collisions between balls and a surface aligned with the x direction.

```

DefineInt ("yoffs",&yoffs);
DefineGraph(freedraw_,"Graphics");
DefineBool("done",&done);
EndMenu();
21  while (!done){
        Events(1);
        DrawGraphs();
        sleep(1);
    }
26 }
```

Now we want to consider this particle in a box. We already understand how the particle will move outside away from the boundaries. Now we need to consider what happens when a particle bounces on a wall. For simplicity let us consider the collision with the bottom wall, which is entirely in the x -direction.

In such a collision we find that the orthogonal x component velocity gets inverted where as the parallel y (and z in 3d) component of the velocity remains unchanged. The position of the particle gets reflected back outside the other side of the obstacle. For a particle of radius r at position x and a wall at position x_b the collision will be triggered if $|x - x_b| < r$. This situation is sketched in Figure 7.1. The collision is accomplished by the following process:

$$v_x \rightarrow -v_x \quad (7.1)$$

$$x \rightarrow x - 2(x - x_b + \text{sign}(x_b - x)r) \quad (7.2)$$

In Fig. 7.1 you see the situation for $x_b - x$, but if you consider the other case where the ball is to the right of the obstacle, you will realize the need for the sign function. This will only work correctly if the timestep is such that $v_x \Delta t < r$ for all velocities that occur in the simulation.

Now the question arises how we should deal with arbitrarily aligned obstacles. The easiest way to deal with this problem is outlined in Fig 7.2. A simple rotation around

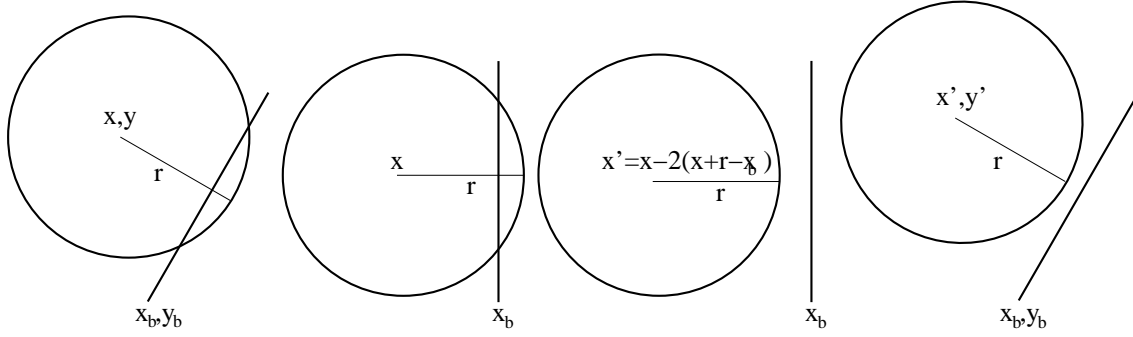


Figure 7.2: Schematic representation of the collision process for balls with walls of arbitrary orientation. We rotate the wall and the particles until they align with the x and y axes, then we perform the collision as described in eqn (7.2), then we rotate back.

one end of the obstacle will reduce this problem to the situation we solved previously. All we need to do is to find the angle as

$$\theta = \text{atan}\left(\frac{\Delta y}{\Delta x}\right) \quad (7.3)$$

There is a small difficulty as the result for the angle would only be found modulo π . There is a function in C, however, that removes this ambiguity: `theta = atan2(deltay,deltax);`. A rotation is achieved by multiplying a vector by a rotation matrix. In two dimensions this is achieved by

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \quad (7.4)$$

After performing the collision, all that is needed is to rotate the coordinates back by an angle of $-\theta$.

This works well, except for a small error in the energy conservation, as shown in Figure 7.3. To fix this we have to take into account the effect of the imposed displacement on the energy:

$$\Delta E_p = mg\Delta y \quad (7.5)$$

The current algorithm conserves the kinetic energy, because it just reflects the velocities leaving $|v|$ unchanged. We now realize that we need to change the velocity by a small amount to account for the change in potential energy. We get

$$\Delta E_k = \frac{1}{2}m(v^2 - (v + \Delta v)^2) = \frac{1}{2}m(-v \cdot \Delta v + (\Delta v)^2) \quad (7.6)$$

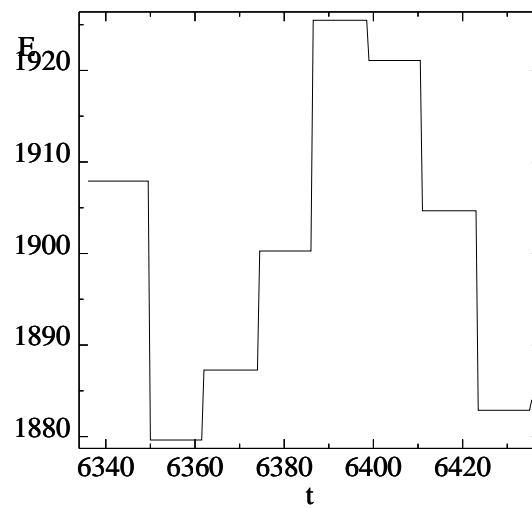


Figure 7.3: We see that there are discrete jumps in the energy value related to bouncing events. This only occurs when we have a finite amount of gravity, suggesting that we see the effect of neglecting the change of potential energy as we are displacing the ball.

Chapter 8

Planetary motion

Let us consider another interesting problem, familiar from introductory Physics. Let us first consider the motion of two celestial objects interacting purely through gravity. Let us assume that the two objects have the masses m_1 and m_2 , respectively. The force between the two objects is then given by Newton's law of gravity¹:

$$\mathbf{F}_{12} = -\frac{m_1 m_2 G}{|\mathbf{r}_{12}|^3} \mathbf{r}_{12} \quad (8.1)$$

By Newton's second law, the second mass will feel the force $\mathbf{F}_{21} = -\mathbf{F}_{12}$. We know how to solve this problem analytically, which makes it a good test case for our algorithms. However, for the only slightly more complicated situation of three masses, there no longer exists a general analytical solution. For this case, then, numerical methods become indispensable.

Implementing this problem numerically is straight forward. Except in stead of one particle, now we have two. We could, of course, separate out the center of mass motion, which would reduce the complexity of the problem by half, and we would recover a one particle problem. But since we are on our way to many particle systems, we will just implement this blindly for now.

This will be the first true simulation, in the sense that we will want to follow the simulation for a long time, not just numerically integrate the equations of motion for a short time. We already know, of course, that for two particles the solution will be a pair of elliptical orbits (at least that is the analytical result, we will have to see if we get the same for our numerical integration), so this won't be too exciting. But if we include more particles the situation changes drastically.

This difference has to be reflected in the structure of the program. Previously we calculated trajectories for a pre-determined time-period. Now we want to continuously follow the simulation and visualize the state of the simulation (and maybe a part of its history). The simulation should be able to continue indefinitely, or until we decide to

¹There is a small correction, that we can obtain from Einstein's general theory of Gravitation. We will neglect this for now.

stop it. To be able to run such a simulation for a long time, we probably don't want to save all intermediate steps. Otherwise we would die of information overload...

So what is a reasonable set of information to retain? We need, of course, the state of the system. The state of the system is determined by all the position and velocity coordinates of the particles. For two particles we can restrict ourselves initially to two-dimensions. For more than two particles, however, we will need to switch to three dimensions.

I suggest that we keep track of, and graphically display,

1. the current position of the particles
2. the current velocities of the particles
3. a short history of positions of the particles
4. a short history of the velocities of the particles
5. a short history of the energy of the system
6. a short history of the angular momentum of the system

Previously we have always integrated the system at different time-steps simultaneously. For this next simulation, we will only use one timestep, but we will be able to change this timestep in the GUI to see the effect of changing it. And we will also implement an algorithm that picks its own time-step according to its needs. We will see that this can be extremely useful.

Two quantities that should be conserved are the energy

$$E = \frac{1}{2}m_1\mathbf{v}_1^2 + \frac{1}{2}m_2\mathbf{v}_2^2 - \frac{m_1m_2G}{|\mathbf{r}_{12}|} \quad (8.2)$$

and the total angular momentum

$$\mathbf{L} = m_1\mathbf{r}_1 \times \mathbf{v}_1 + m_2\mathbf{r}_2 \times \mathbf{v}_2 \quad (8.3)$$

For our initial two-dimensional simulations this angular momentum only has a z-component which is

$$L_z = m_1(r_{1x}v_{1y} - r_{1y}v_{1x}) + m_1(r_{2x}v_{2y} - r_{2y}v_{2x}). \quad (8.4)$$

First let us talk about the choice of the parameters relevant to this simulation: the masses of the celestial objects m_1 and m_2 , the universal Gravitational constant $G = 6.67300 \times 10^{-11}m^3kg^{-1}s^{-2}$, and finally the spatial displacement of the object, which is often another large number. The astronomical unit, roughly the average distance between the earth and the sun, is $AU = 149,597,870,700m$. Typical masses are also very large numbers, when measured in kg. The earth weighs $5.972 \times 10^{24}kg$, the sun $1.9891 \times 10^{30}kg$. Depending on the data type used, these numbers, when combined, can tax the ability of the computer to represent those numbers. For the real number

type `float` we have a range of about 1.2×10^{38} to $3.4 \times 10^{+38}$. For a double we typically have a range from 1.7×10^{308} to $1.7 \times 10^{+308}$. So we see that using the data-type of `float` will give immediate problems, whereas we may be able to get by with the double type.

Regardless, even if possible it is inconvenient to deal with such humungous numbers. Instead we make use of the fact that we can use different units to measure our primitive quantities like mass, time, length, charge etc. So if we want to simulate the earth-sun system we can choose a length scale of AU , a mass scale of the mass of the sun (or the earth) and a time-scale so that the gravitational constant becomes 1. These computational units are much more convenient for programming purposes.

For reasons of definiteness let us define our length-scale as $1AU$, and our mass scale as 1 earth mass. The mass of the sun is then 333,000. The timescale is chosen as to give

$$1 \frac{AU^3}{m_e T} = G \quad (8.5)$$

$$= 6.67300 \times 10^{-11} m^3 kg^{-1} s^{-2} \quad (8.6)$$

$$= 6.67300 \times 10^{-11} \frac{1}{(1.495 \times 10^{11})^3} AU^3 5.972 \times 10^{24} \frac{1}{m_e} \frac{\tau^2}{s^2} \frac{1}{T^2} \quad (8.7)$$

$$= 1.192 \times 10^{-19} \tau^2 \frac{AU^3}{m_e T^2} \quad (8.8)$$

From this we see that our timescale τ is

$$\tau = \sqrt{1.92 \times 10^{19}} s = 139 years \quad (8.9)$$

We will use computational units like these to have resonable number representations. With the analysis above we now also know how to refer back between the computational and the standard units.

Listing 8.1: planets.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
4 #include <mygraph.h>
#include <math.h>

#define pi 3.14159265358979323846264338327950288419716939937

9 double m1=1,m2=1,G=1,v_in=1,theta=45;
#define VC 8 /* Number of components in state vector */
double v[VC]; /* The state vector of the system */
#define FC 4 /* Number of components in Force vector, FC=VC/2 (
    typically) */
```

```

#define NG 6 /* Graphs for display: (x1,y1) (x2,y2) (v1x,v1y) (
    v2x,v2y) E(t) */
14 double x1[NG][100][2];
    char *GrName[NG]={"(x1,y1)", "(x2,y2)", "(v1_x,v1_y)", "(v2_x,v2_y)
        ", "(t,E)", "(t,L)"};
    int size=100;
    double time=0, iterations=0;

19 /* States: we now have two particles with coorcinates
    x1,y1,v1x,v1y,x2,y2,v2x,v2z */

    void FF(double v[VC], double F[FC]) {
        double rabs=sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));
24 F[0]=-G*m1*m2/pow(rabs,3)*(v[0]-v[4]);
        F[1]=-G*m1*m2/pow(rabs,3)*(v[1]-v[5]);
        F[2]=-F[0];
        F[3]=-F[1];
    }

29 void dFFx(double v[VC], double dF[FC][VC]) { /* Gradient of the
    Force */
        double rabs=sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));

        dF[0][0]=0; dF[0][1]=0; dF[0][2]=0;
34 dF[1][0]=0; dF[1][1]=0; dF[1][2]=0;
        dF[1][3]=0;
    }

    void (*Iterate)(double v[VC], double Dt)=NULL;

39 void IterateEuler(double v[VC], double Dt) {
        double F[4];
        FF(v,F);
        v[0]+=v[2]*Dt;
        v[1]+=v[3]*Dt;
44 v[2]+=F[0]*Dt/m1;
        v[3]+=F[1]*Dt/m1;

        v[4]+=v[6]*Dt;
49 v[5]+=v[7]*Dt;
        v[6]+=F[2]*Dt/m2;
        v[7]+=F[3]*Dt/m2;

```



```

    }

54 void IterateDeriv(double v[VC], double Dt){
    double F[FC], dF[FC][VC];
    FF(v, F);
    dFFx(v, dF);
    v[0] += (v[2] + 0.5 * F[0] * Dt) * Dt;
59 v[1] += (v[3] + 0.5 * F[1] * Dt) * Dt;
    v[2] += (F[0] + 0.5 / m1 * (dF[0][0] * v[2] + dF[0][1] * v[3] + dF[0][2] * F
        [0] / m1 + dF[0][3] * F[1] / m1) * Dt) * Dt / m1;
    v[3] += (F[1] + 0.5 / m1 * (dF[1][0] * v[2] + dF[1][1] * v[3] + dF[1][2] * F
        [0] / m1 + dF[1][3] * F[1] / m1) * Dt) * Dt / m1;
    }

64 void IterateVerlet(double v[VC], double Dt){
    double F[FC], Fn[FC];
    FF(v, F);
    v[0] += (v[2] + 0.5 * F[0] / m1 * Dt) * Dt;
    v[1] += (v[3] + 0.5 * F[1] / m1 * Dt) * Dt;
69 v[2] += F[0] * Dt / m1;
    v[3] += F[1] * Dt / m1;

    v[4] += (v[6] + 0.5 * F[2] / m2 * Dt) * Dt;
    v[5] += (v[7] + 0.5 * F[3] / m2 * Dt) * Dt;
74 v[6] += F[2] * Dt / m2;
    v[7] += F[3] * Dt / m2;
    FF(v, Fn); // Could do this right away since F does not depend
        on velocity.
    v[2] += 0.5 * (Fn[0] - F[0]) * Dt / m1;
    v[3] += 0.5 * (Fn[1] - F[1]) * Dt / m1;
79 v[6] += 0.5 * (Fn[2] - F[2]) * Dt / m2;
    v[7] += 0.5 * (Fn[3] - F[3]) * Dt / m2;
    }

void SetEuler(){
84 Iterate = &IterateEuler;
    }

void SetDeriv(){
    Iterate = &IterateDeriv;
89 }

```

```

void SetVerlet(){
    Iterate = &IterateVerlet;
}

94 double E(double v[VC]){
    return 0.5*m1*(v[2]*v[2]+v[3]*v[3])+0.5*m2*(v[6]*v[6]+v[7]*v
        [7])-m1*m2*G/sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));
}

99 double L(double v[VC]){
    return m1*(v[0]*v[3]-v[1]*v[2])+m2*(v[4]*v[7]-v[5]*v[6]);
}

void Initialize(double v[VC],double x[NG][size][2],double *time
){
104     *time = 0;

    v[0]=0; // x1_0 = 0
    v[1]=1; // y1_0 = 1
    v[2]=v_in*cos(theta/180*pi);
109    v[3]=v_in*sin(theta/180*pi);

    v[4]=0; // x2_0 = 0
    v[5]=-1; // y2_0 = -1
    v[6]=-m1/m2*v_in*cos(theta/180*pi);
114    v[7]=-m1/m2*v_in*sin(theta/180*pi);

    /* Initialize the graphics variables */
    for (int i=0; i<size;i++){
        x[0][i][0]=v[0]; x[0][i][1]=v[1];
119        x[1][i][0]=v[4]; x[1][i][1]=v[5];
        x[2][i][0]=v[2]; x[2][i][1]=v[3];
        x[3][i][0]=v[6]; x[3][i][1]=v[7];
        x[4][i][0]=*time;x[4][i][1]=E(v);
        x[5][i][0]=*time;x[5][i][1]=L(v);
124    }
}

void AnalyzeData(double v[VC],double x[NG][size][2],double time
){
    for (int i=0;i<NG;i++) // move data points back in graphics
        array

```

```

129      memmove(&x[i][1][0], &x[i][0][0], (size-1)*2*sizeof(double));

      x[0][0][0]=v[0]; x[0][0][1]=v[1];
      x[1][0][0]=v[4]; x[1][0][1]=v[5];
      x[2][0][0]=v[2]; x[2][0][1]=v[3];
134     x[3][0][0]=v[6]; x[3][0][1]=v[7];
      x[4][0][0]=time; x[4][0][1]=E(v);
      x[5][0][0]=time; x[5][0][1]=L(v);
  }

139 void init(){
      Initialize(v,x1,&time); // Just a little wrapper to call from
                           the menu
  }

  int main (){
144     int i, Paused=1, Step=1, Repeat=1, done=0;
      double Dt=0.01;

      Initialize(v,x1,&time);
      SetEuler();
149     for (i=0; i<NG; i++)
          DefineGraphN_RxR(GrName[i], &(x1[i][0][0]), &size, NULL);

      StartMenu("Two_celestial_bodies", 1);
154     DefineDouble("m1", &m1);
      DefineDouble("m2", &m2);
      DefineDouble("G", &G);
      DefineDouble("theta", &theta);
      DefineDouble("v_in", &v_in);
159     DefineFunction("Reinitialize", &init);
      DefineGraph(curve2d, "Graph");
      DefineFunction("Set_Euler", &SetEuler);
      DefineFunction("Set_Deriv", &SetDeriv);
      DefineFunction("Set_Verlet", &SetVerlet);
164     DefineDouble("Dt", &Dt);
      DefineInt("Repeat", &Repeat);
      DefineBool("Step", &Step);
      DefineBool("Paused", &Paused);
      DefineBool("done", &done);
169     EndMenu();

```

```

while (!done){
    Events(1);
    DrawGraphs();
    if (!Paused || !Step){
174      Step=1;
          for (int i=0;i<Repeat;i++) Iterate(v,Dt);
          time += Repeat*Dt;
          AnalyzeData(v,x1,time);
    }
179    else sleep(1);
}
}

```

We can run this code, but we realize that this code is rather finicky. If we use some randomly chosen initial conditions and timesteps it is likely that the simulation simply blows up, and it takes a little while of trying (and a bit of thinking) to figure out a set of computational parameters that will give you the correct results. We know that we should be able

The main problem is that the force varies widely with the distance.

Problems

8.0.1: The program listing for planets.c does not have the `IterateDeriv()` and `dFFx()` routines correctly implemented. To do this you need to calculate the general gradients of $\partial F/\partial v$ for all the force components and all the state vector components and implement them in the `cFFx` routine. Mathematically you need

$$dF = \begin{pmatrix} \frac{\partial F_{x_1}}{\partial x_1} & \frac{\partial F_{x_1}}{\partial y_1} & \frac{\partial F_{x_1}}{\partial v_{1x}} & \frac{\partial F_{x_1}}{\partial v_{1y}} & \frac{\partial F_{x_1}}{\partial x_2} & \frac{\partial F_{x_1}}{\partial y_2} & \frac{\partial F_{x_1}}{\partial v_{2x}} & \frac{\partial F_{x_1}}{\partial v_{2y}} \\ \frac{\partial F_{y_1}}{\partial x_1} & \frac{\partial F_{y_1}}{\partial y_1} & \frac{\partial F_{y_1}}{\partial v_{1x}} & \frac{\partial F_{y_1}}{\partial v_{1y}} & \frac{\partial F_{y_1}}{\partial x_2} & \frac{\partial F_{y_1}}{\partial y_2} & \frac{\partial F_{y_1}}{\partial v_{2x}} & \frac{\partial F_{y_1}}{\partial v_{2y}} \\ \frac{\partial F_{x_2}}{\partial x_1} & \frac{\partial F_{x_2}}{\partial y_1} & \frac{\partial F_{x_2}}{\partial v_{1x}} & \frac{\partial F_{x_2}}{\partial v_{1y}} & \frac{\partial F_{x_2}}{\partial x_2} & \frac{\partial F_{x_2}}{\partial y_2} & \frac{\partial F_{x_2}}{\partial v_{2x}} & \frac{\partial F_{x_2}}{\partial v_{2y}} \\ \frac{\partial F_{y_2}}{\partial x_1} & \frac{\partial F_{y_2}}{\partial y_1} & \frac{\partial F_{y_2}}{\partial v_{1x}} & \frac{\partial F_{y_2}}{\partial v_{1y}} & \frac{\partial F_{y_2}}{\partial x_2} & \frac{\partial F_{y_2}}{\partial y_2} & \frac{\partial F_{y_2}}{\partial v_{2x}} & \frac{\partial F_{y_2}}{\partial v_{2y}} \end{pmatrix} \quad (8.10)$$

which looks horrible initially, but there are many zeros and symmetries that make the calculations simpler. Once you calculated this matrix of derivatives you need to implement it into the `iterateDeriv()` routine.

Once you have accomplished this, test the relative performance of the deriv, Verlet, and Euler algorithms.

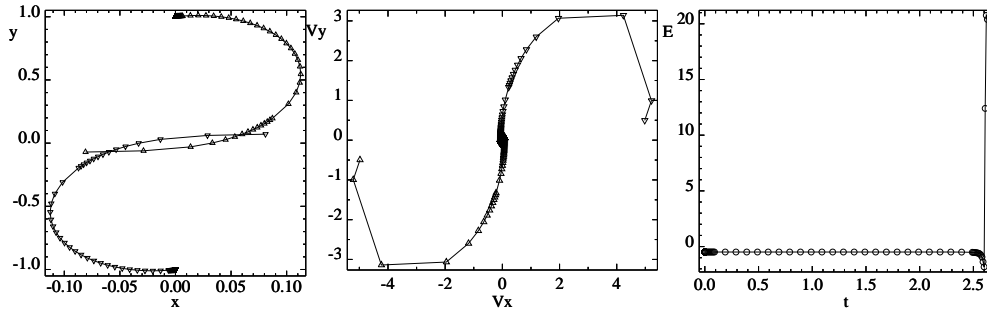


Figure 8.1: Trajectories of two planets, their velocities and the total energy for a simulation with the Euler algorithm. We see that as the planets approach each other the algorithm does not conserve energy. Subsequently the planets will fly away from each other.

8.1 Adaptive Step Size

To get a better execution speed, we need to adapt the step size of the simulation. But what causes the problems, and when do we need to reduce the time-step for the simulation? In Figure 8.1 we show how a simulation using the Euler algorithm with the step-size of 0.01, and an initial velocity of 0.1 fails. For the last data-points before the failure we show the values for each iteration step.

The failure of the algorithm is characterized by large jumps in the position and velocity, and that a failure occurred is most obvious when observing the Energy. There is, maybe surprisingly, no similar error in the angular momentum. So what is going on? One way to look at this is to use an energy argument: when the two planets approach each other the forces become large. If the particle arrives in a low-energy region, a large force is experienced. This force then leads to a large increase in the velocity, and the finite time-step brings the particle out of the energy well without experiencing the corresponding force on the way out of the energy minimum. This scenario also explains why the energy tends to increase strongly in this scenario?

Would using the Verlet algorithm instead of the Euler algorithm help? It depends: on the one hand the position update would be more extreme when the force and the velocity align, because the particle will be moved forward by an additional increment $\mathbf{F}(\Delta t)^2$. On the other hand the velocity will be updated by a more moderate amount of $0.5(\mathbf{F}(x(t)) + \mathbf{F}(x(t + \Delta t)))$. For the example above, shown in Figure 8.2, using the Verlet algorithm is not favorable. This is not an uncommon occurrence: when algorithms fail, higher order algorithms are often less robust than lower order algorithms. This is the reason that for simulations you will hardly ever find methods that are better than second order.

Now what we need to do is to adapt the step-size. This step-size should be chosen such that we don't spend too much time, where a larger time-step would be sufficient, but where we ensure that the time-step becomes small enough when accuracy requires it. This leaves us with the problem of determining of how to chose the timestep.

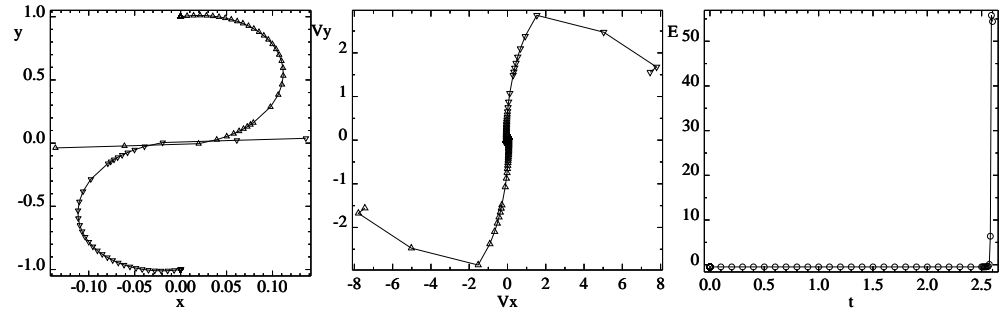


Figure 8.2: Trajectories of two planets, their velocities and the total energy for a simulation with the Verlet algorithm. We see that as the planets approach each other the algorithm does not conserve energy. Subsequently the planets will fly away from each other.

One way of choosing the time-step is to ensure that the displacement in a time-step is small compared to the inter-planet distance. So we require

$$v \Delta t = \epsilon r_{12} \quad (8.11)$$

$$\Leftrightarrow \Delta t = \frac{\epsilon r_{12}}{|v_{max}|} \quad (8.12)$$

where v_{max} is the larger of the two absolute velocities. A possible implementation looks like this:

Listing 8.2: planetsAdapt2.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
4 #include <mygraph.h>
#include <math.h>

#define pi 3.14159265358979323846264338327950288419716939937

9 double m1=1,m2=1,G=1,v_in=1,theta=45;
#define VC 8 /* Number of components in state vector */
double v[VC]; /* The state vector of the system */
#define FC 4 /* Number of components in Force vector, FC=VC/2 (
    typically) */
#define NG 7 /* Graphs for display: (x1,y1) (x2,y2) (v1x,v1y) (
    v2x,v2y) E(t)*/
14 double x1[NG][100][2];
char *GrName[NG]={"(x1,y1)","(x2,y2)","(v1_x,v1_y)","(v2_x,v2_y)
    ","(t,E)","(t,L)","(t,Dt)"};
int size=100;
```

```

double Dtmeas=0.01,Dt=0.0001;
double time=0,Eps=1e-5;
19

/* States: we now have two particles with coorcinates
   x1,y1,v1x,v1y,x2,y2,v2x,v2z */

24 double E(double v[VC]) {
    return 0.5*m1*(v[2]*v[2]+v[3]*v[3])+0.5*m2*(v[6]*v[6]+v[7]*v
        [7])-m1*m2*G/sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));
}

double L(double v[VC]) {
29    return m1*(v[0]*v[3]-v[1]*v[2])+m2*(v[4]*v[7]-v[5]*v[6]);
}

34 double TimeStep(double v[VC], double F[FC]) {
    double rabs=sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));
    double vabs1=sqrt(pow(v[2],2)+pow(v[3],2));
    double vabs2=sqrt(pow(v[6],2)+pow(v[7],2));
    if (vabs2>vabs1) vabs1=vabs2;
39    return Eps*rabs/vabs1;
}

void FF(double v[VC], double F[FC]) {
    double rabs=sqrt(pow(v[0]-v[4],2)+pow(v[1]-v[5],2));
44    F[0]=-G*m1*m2/pow(rabs,3)*(v[0]-v[4]);
    F[1]=-G*m1*m2/pow(rabs,3)*(v[1]-v[5]);
    F[2]=-F[0];
    F[3]=-F[1];
}

49 void (*Iterate)(double v[VC], double *time)=NULL;

void IterateEuler(double v[VC], double *time){
    double F[4];
54    FF(v,F);
    Dt=TimeStep(v,F);
    *time+=Dt;
}

```

```

    v[0] += v[2] * Dt;
59    v[1] += v[3] * Dt;
    v[2] += F[0] * Dt/m1;
    v[3] += F[1] * Dt/m1;

    v[4] += v[6] * Dt;
64    v[5] += v[7] * Dt;
    v[6] += F[2] * Dt/m2;
    v[7] += F[3] * Dt/m2;
}

69 void IterateVerlet(double v[VC], double *time){
    double F[FC], Fn[FC];
    FF(v, F);
    Dt = TimeStep(v, F);
    *time += Dt;
74    v[0] += (v[2] + 0.5 * F[0] / m1 * Dt) * Dt;
    v[1] += (v[3] + 0.5 * F[1] / m1 * Dt) * Dt;
    v[2] += F[0] * Dt/m1;
    v[3] += F[1] * Dt/m1;

    v[4] += (v[6] + 0.5 * F[2] / m2 * Dt) * Dt;
79    v[5] += (v[7] + 0.5 * F[3] / m2 * Dt) * Dt;
    v[6] += F[2] * Dt/m2;
    v[7] += F[3] * Dt/m2;
    FF(v, Fn); // Could do this right away since F does not depend
               on velocity.
84    v[2] += 0.5 * (Fn[0] - F[0]) * Dt/m1;
    v[3] += 0.5 * (Fn[1] - F[1]) * Dt/m1;
    v[6] += 0.5 * (Fn[2] - F[2]) * Dt/m2;
    v[7] += 0.5 * (Fn[3] - F[3]) * Dt/m2;

89 }

void SetEuler(){
    Iterate = &IterateEuler;
}

94 void SetVerlet(){
    Iterate = &IterateVerlet;
}

```



```

99 void Initialize(double v[VC], double x[NG][size][2], double *time
    ){
    *time = 0;

    v[0]=0; // x1_0 = 0
    v[1]=1; // y1_0 = 1
104 v[2]=v_in*cos(theta/180*pi);
    v[3]=v_in*sin(theta/180*pi);

    v[4]=0; // x2_0 = 0
    v[5]=-1; // y2_0 = -1
109 v[6]=-m1/m2*v_in*cos(theta/180*pi);
    v[7]=-m1/m2*v_in*sin(theta/180*pi);

    /* Initialize the graphics variables */
    for (int i=0; i<size; i++){
114 x[0][i][0]=v[0]; x[0][i][1]=v[1];
        x[1][i][0]=v[4]; x[1][i][1]=v[5];
        x[2][i][0]=v[2]; x[2][i][1]=v[3];
        x[3][i][0]=v[6]; x[3][i][1]=v[7];
        x[4][i][0]=*time; x[4][i][1]=E(v);
119 x[5][i][0]=*time; x[5][i][1]=L(v);
        x[6][i][0]=*time; x[6][i][1]=Dt;
    }
}

124 void AnalyzeData(double v[VC], double x[NG][size][2], double time
    ){
    for (int i=0; i<NG; i++) // move data points back in graphics
        array
        memmove(&x[i][1][0], &x[i][0][0], (size-1)*2*sizeof(double));

    x[0][0][0]=v[0]; x[0][0][1]=v[1];
129 x[1][0][0]=v[4]; x[1][0][1]=v[5];
    x[2][0][0]=v[2]; x[2][0][1]=v[3];
    x[3][0][0]=v[6]; x[3][0][1]=v[7];
    x[4][0][0]=time; x[4][0][1]=E(v);
    x[5][0][0]=time; x[5][0][1]=L(v);
134 x[6][0][0]=time; x[6][0][1]=Dt;
}

void init(){

```

```

        Initialize(v,x1,&time); // Just a little wrapper to call from
                               the menu
139 }

    int main () {
        int i, Paused=1, Step=1, done=0;
        double rtime;
144    Initialize(v,x1,&time);
        SetEuler();

        for (i=0;i<NG;i++)
            DefineGraphN_RxR(GrName[i],&(x1[i][0][0]),&size,NULL);
149
        StartMenu("Two_celestial_bodies",1);
        DefineDouble("m1",&m1);
        DefineDouble("m2",&m2);
        DefineDouble("G",&G);
154    DefineDouble("theta",&theta);
        DefineDouble("v_in",&v_in);
        DefineFunction("Reinitialize",&init);
        DefineGraph(curve2d_,"Graph");
        DefineFunction("Set_Euler",&SetEuler);
159    DefineFunction("Set_Verlet",&SetVerlet);
        DefineDouble("Dt",&Dt);
        DefineDouble("Dt_meas",&Dtmeas);
        DefineDouble("Eps",&Eps);
        DefineBool("Step",&Step);
164    DefineBool("Paused",&Paused);
        DefineBool("done",&done);
        EndMenu();
        while (!done) {
            Events(1);
169    DrawGraphs();
            if (!Paused || !Step) {
                Step=1;
                for (rtime=0;rtime<Dtmeas; Iterate(v,&rtime));
                time += rtime;
174    AnalyzeData(v,x1,time);
            }
            else sleep(1);
        }
    }
}

```

This approach gives us a pretty good performance. However, it required a good understanding of the underlying problems. There are more general methods of evaluating the error encountered. These approaches include using results of two methods of different orders and since the higher order method is expected to be closer to the exact result. The difference between the two methods is then a measure for the error of the lower order method. A second approach consists of using the same method but running it at two time-steps. Again the difference between the results gives us a measure for the error.

Problems

8.1.1: Examine what happens if your force law looks different. In particular what happens if your force law is

$$\mathbf{F}_{12} = \frac{Gm_1m_2}{|\mathbf{r}_{12}|^\alpha} \mathbf{r}_{12} \quad (8.13)$$

for $\alpha \neq 3$. A particularly important example occurs for an harmonic oscillator, where the force is linear with the distance. What value of α does that correspond to? What do the trajectories look like for this force potential?

8.1.2: What do the trajectories look like for other exponents α ? Is there an important qualitative difference?

8.2 More particles

So far we have looked at the dynamics for one or two particles. But there are many problems that require us to look at more particles. Interestingly the questions that one might want to ask of such systems changes as we increase the number of particles. First we will consider systems with a few particles. First we will to examine three body problems. They are very interesting as the three body problem usually will not allow an analytical solution. We will also see why that is. We can examine planetary motion where we consider gravitational interaction, the motion of the classical hydrogen atom which behaves quite different because the like charges of the two electrons repel each other. We will also consider a system where we don't simply have a featureless point particle, but a particle with additional degrees of freedom, which we will use as a very simple model of tidal interactions.

But before we do that we have to generalize our program to include more than two, and ideally an arbitrary number of particles. This is a code that includes an arbitrary number of particles:

Listing 8.3: MD.c

```

#include <stdio.h>
2 #include <string.h>
  #include <unistd.h>
  #include <mygraph.h>
  #include <math.h>

7 #define pi 3.14159265358979323846264338327950288419716939937

  #define N 6 /* Number of coordinates (or momenta) */
  double v[N*2], vin[N*2]; /* The state vector of the system,
    initial state */
  /* We define a general state vector consisting of N coordinates
    q and N momenta p. It reads (q1,q2,...,qN,p1,p2,...pN). */
12 double m[N/2], G=1;
  int N02=N/2; /* Needed for the graphics library */
  double Eps=1e-5;

  #define NG 2+N/2 /* Graphs for display: Could be anything E(t),
    L(t) ...*/
17 #define MEM 1000
  double x1[NG][MEM][2];
  char *GrName[NG]={"(t,E)", "(t,L)", "(x1,y1)", "(x2,y2)", "(x3,y3)"
    }; //needs to be more general...
  int size=MEM;
  double time=0, iterations=0;

22 /* States: we now have N/2 particles in two dimensions */

  void FF(double v[N*2], double F[N]) {
    double rx, ry, rabs;
27   int i, j;

    memset(&(F[0]), 0, N*sizeof(double)); /* set the F array to
      zero */

    for (i=0; i<N/2; i++){
32     for (j=i+1; j<N/2; j++){
        rx=v[i*2]-v[j*2];
        ry=v[i*2+1]-v[j*2+1];
        rabs=sqrt(pow(rx,2)+pow(ry,2));
        F[i*2]+=-G*m[i]*m[j]/pow(rabs,3)*rx;

```

```

37      F[i*2+1]+=-G*m[i]*m[j]/pow(rabs,3)*ry;
      F[j*2]+=-F[i*2];
      F[j*2+1]+=-F[i*2+1];
    }
  }
42 }

double TimeStep(double v[N*2]) { // for gravitational
    interaction
    int i, j;
    double rabs=1e38, vabs=1e-10, t;
47    for (i=0; i<N/2; i++){
        t=pow(v[N+i*2],2)+pow(v[N+i*2+1],2);
        if (t>vabs) vabs=t;
        for (j=i+1; j<N/2; j++){
52            t=pow(v[i*2]-v[j*2],2)+pow(v[i*2+1]-v[j*2+1],2);
            if (t<rabs) rabs=t;
        }
    }
    return Eps*sqrt(rabs/vabs);
57 }

void (*Iterate)(double v[N*2], double Dt)=NULL;

62 void IterateEuler(double v[N*2], double Dt){
    double F[N];
    int i;
    FF(v, F);
    for (i=0; i<N; i++){
67        v[i]+=v[N+i]*Dt;
        v[N+i]+=F[i]*Dt/m[i/2];
    }
}

72 void IterateVerlet(double v[N*2], double Dt){
    double F[N], Fn[N];
    int i;

    FF(v, F);
77    for (i=0; i<N; i++){

```

```

        v[i]+=(v[N+i]+0.5*F[i]/m[i/2]*Dt)*Dt;
    }
    FF(v,Fn); /* assuming that forces don't depend on velocity */
    for (i=0;i<N;i++){
82      v[N+i]+=0.5*(F[i]+Fn[i])*Dt/m[i/2];
    }
}

void SetEuler(){
87   Iterate = &IterateEuler;
}

void SetVerlet(){
    Iterate = &IterateVerlet;
92 }

double E(double v[N*2]){
    double Eret=0,rx,ry,rabs;
    int i,j;
97   for (i=0;i<N/2;i++){
        Eret+=0.5*m[i]*(pow(v[N+2*i],2)+pow(v[N+2*i+1],2));
        for (j=i+1;j<N/2;j++){
            rx=v[i*2]-v[j*2];
            ry=v[i*2+1]-v[j*2+1];
102          rabs=sqrt(pow(rx,2)+pow(ry,2));
            Eret+=-G*m[i]*m[j]/pow(rabs,1);
        }
    }
107   return Eret;
}

double L(double v[N*2]){
    double Lzret=0;
112   int i;
    for (i=0;i<N/2;i++){
        Lzret+=m[i]*(v[i*2]*v[N+i*2+1]-v[i*2+1]*v[N+i*2]);
    }
    return Lzret;
117 }

void Initialize(double v[N*2],double x[NG][MEM][2],double *time

```

```

    ){
    int i,j;
    *time = 0;
122    for (i=0;i<2*N;i++) v[i]=vin[i];

    /* Initialize the graphics variables */
    for (int i=0; i<size;i++){
127        x[0][i][0]=*time;x[0][i][1]=E(v);
        x[1][i][0]=*time;x[1][i][1]=L(v);
        for (j=0;j<N/2;j++){
            x[j+2][0][0]=v[2*j];
            x[j+2][0][1]=v[2*j+1];
132        }
    }
}

void AnalyzeData(double v[N*2],double x[NG][size][2],double
    time){
137    for (int i=0;i<NG;i++) // move data points back in graphics
        array
        memmove(&x[i][1][0],&x[i][0][0],(size-1)*2*sizeof(double));

    x[0][0][0]=time; x[0][0][1]=E(v);
    x[1][0][0]=time; x[1][0][1]=L(v);
142    for (int i=0;i<N/2;i++){
        x[i+2][0][0]=v[2*i];
        x[i+2][0][1]=v[2*i+1];
    }
}
147

void SaveState(){
    for (int i=0;i<2*N;i++) vin[i]=v[i];
}

152

void init(){
    Initialize(v,x1,&time); // Just a little wrapper to call from
        the menu
}

157 int main (){

```

```

int i, Paused=1, Step=1, Repeat=1, done=0, Adapt=1;
char str[100];
double Dt=0.01;

162   Initialize(v,x1,&time);
      SetEuler();

      DefineGraphN_RxR("Pos",&v[0],&N02,NULL);
      DefineGraphN_RxR("Vel",&v[N],&N02,NULL);
167   for (i=0;i<NG;i++)
        DefineGraphN_RxR(GrName[i],&(x1[i][0][0]),&size,NULL);

      StartMenu("Two_celestial_bodies",1);
      DefineDouble("G",&G);
172   StartMenu("Initial_State",0);
      for (i=0;i<N/2;i++){
        sprintf(str,"m_%i",i);
        DefineDouble(str,&m[i]);
        sprintf(str,"x_%i",i);
177   DefineDouble(str,&vin[i*2]);
        sprintf(str,"y_%i",i);
        DefineDouble(str,&vin[i*2+1]);
        sprintf(str,"Vx_%i",i);
        DefineDouble(str,&vin[N+i*2]);
182   sprintf(str,"Vy_%i",i);
        DefineDouble(str,&vin[N+i*2+1]);
      }
      DefineFunction("SaveState",&SaveState);
      DefineFunction("Reinitialize",&init);
187   EndMenu();
      DefineGraph(curve2d_,"Graph");
      DefineFunction("Set_Euler",&SetEuler);
      DefineFunction("Set_Verlet",&SetVerlet);
      DefineDouble("time",&time);
192   DefineDouble("Dt",&Dt);
      DefineDouble("Eps",&Eps);
      DefineBool("Adapt",&Adapt);
      DefineInt("Repeat",&Repeat);
      DefineBool("Step",&Step);
197   DefineBool("Paused",&Paused);
      DefineBool("done",&done);
      EndMenu();

```


m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	0	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	1	0
m_2	x_2	y_2	v_{x2}	v_{y2}
1	0	2	0.1	0

Table 8.1: Initial conditions for the three planets shown in Figure 8.3.

```

while (!done){
    Events(1);
202   DrawGraphs();
    if (!Paused || !Step){
        Step=1;
        for (int i=0;i<Repeat;i++){
            time += Dt;
207         Iterate(v,Dt);
            if (Adapt) Dt=TimeStep(v);
        }
        AnalyzeData(v,x1,time);
    }
212   else sleep(1);
}

```

We can now use this code to examine the behavior of a three particle system. A first taste of what these behavior of these systems is, is shown in Figure 8.3. We no longer find simple elliptical path, but when the planets approach each other they scatter and find new path. Not all orbits are as chaotic, however. From experience we know that our planetary system consists of well-spaced planets that are small enough, compared to the sun, to not influence each others planetary obits significantly.

Similarly many of these planets have moons orbiting the planets. We can see if we can simulate a system that is reminiscent of the sun-earth-moon system. We can use parameter sets that have a heavy planet a lighter planet and an even lighter moon. One may wonder what initial conditions will lead to a system where the moon orbits the planet, and where this system again orbits around the sun.

First we try the parameters of table 8.2 where we set all bodies up in a line along the y axis. Then we give them velocities in the x-direction. This can give a system that behaves qualitatively like a system of sun/planet/moon. However, a relatively slight difference in the velocity of the moon, either a little too fast or a little too slow leads to very different systems where the planet and the moon do not remain bound to each other.

In this particular example the system with the least amount of total energy happens

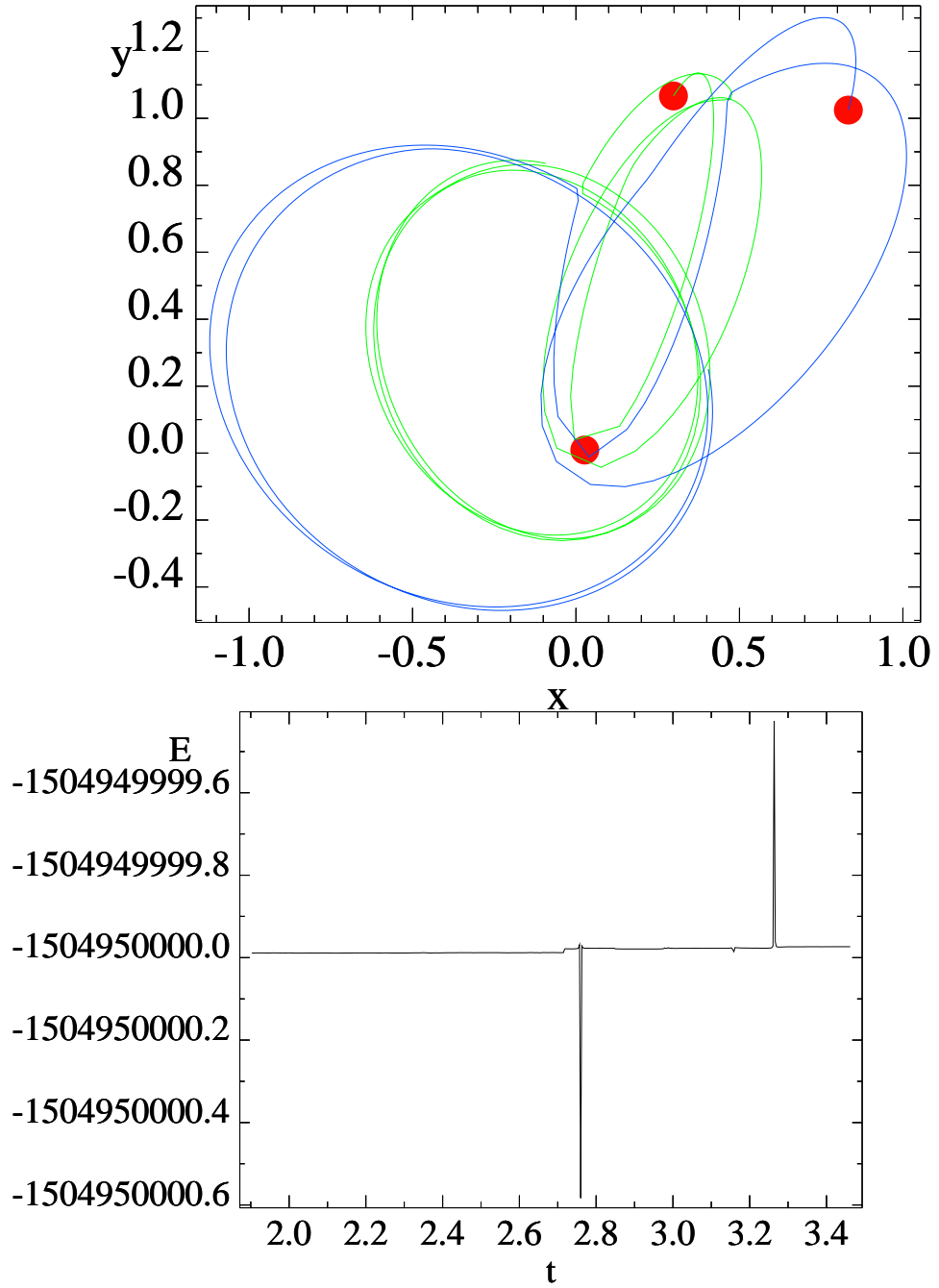


Figure 8.3: A three body simulation. The dots indicate the current position of the planets, and the lines indicate their past positions. The initial conditions for this simulation is shown in table 8.1. The parameters for this simulation were $\Delta t = 10^{-8}$, Repeat=3000, $G=1$, and we used the Verlet algorithm.

m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	-0.1	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	10	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.1	0	1.2	7	0

Table 8.2: Initial conditions for the three planets shown in Figure 8.4.

m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	-0.1	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	10	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.1	0	1.2	7.1	0

Table 8.3: Initial conditions for the three planets shown in Figure 8.4.

to eject the moon from the system entirely. In the problems section you will be asked to examine this system more carefully.

When looking at these complicated path one may ask the question which we have avoided so far: are these orbits truly solutions to our initial conditions? One way to answer this is to recalculate the orbits with a different time-step. If the orbit is truly converged we would expect that the results of the two simulations give us the same answer. For the calculations used here we used the adaptive step-size algorithm, so what we have to vary here is the ϵ value in our program (called **Eps** there). For the initial calculations we used $\epsilon = 10^{-6}$. We can compare three simulations with the values of $\epsilon = \{10^{-5}, 10^{-6}, 10^{-7}\}$. This is shown in Figure 8.5.

Problems

8.2.1: In the sun/planet/moon system, described in the text above I showed that we do only find a very limited range in which can observe a moon that is stably bound

m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	-0.1	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	10	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.1	0	1.2	7.2	0

Table 8.4: Initial conditions for the three planets shown in Figure 8.4.

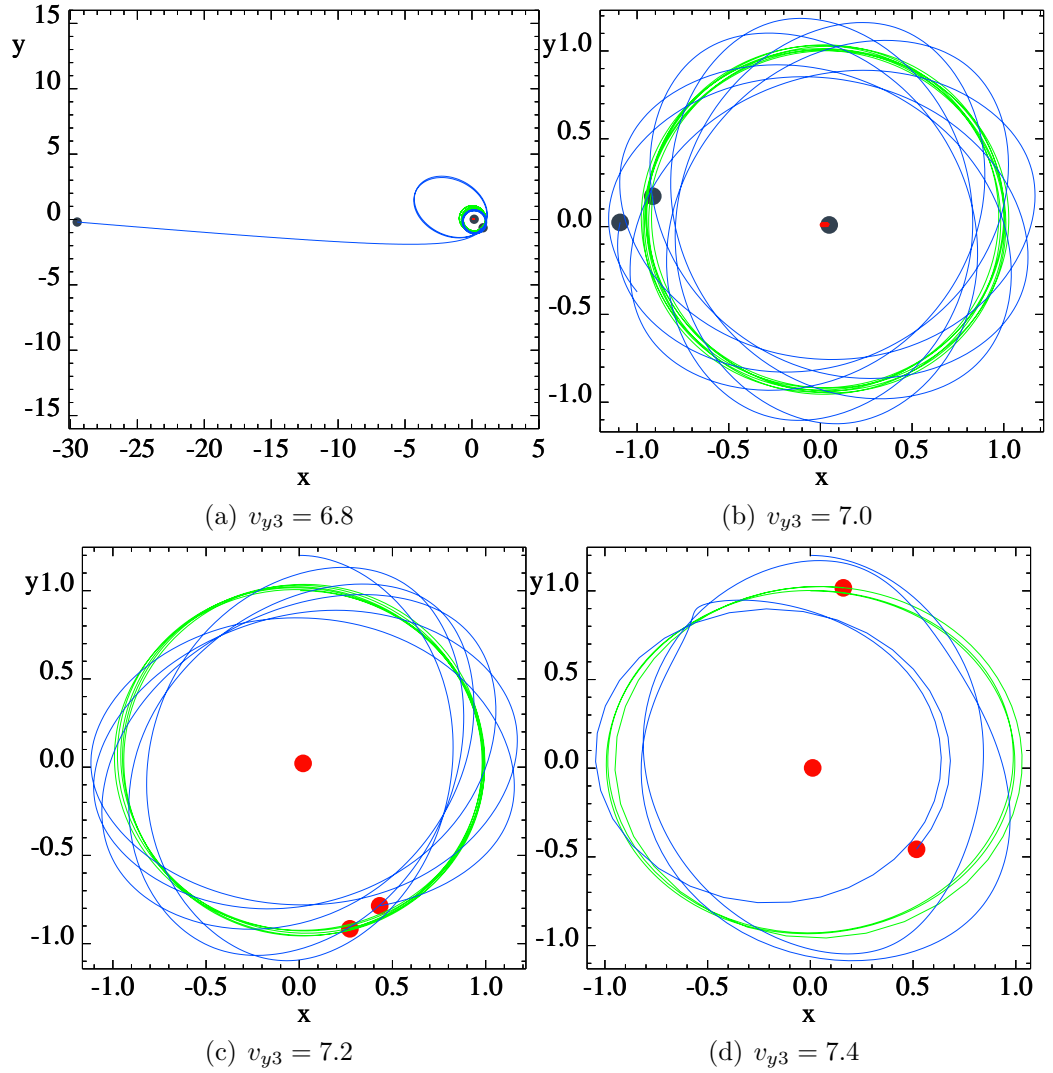


Figure 8.4: Orbits of a potential sun/planet/moon system. The only difference between these systems lies in the slight difference in the initial velocity of the moon, as shown in Tables 8.5, 8.2, 8.3, 8.4.

m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	-0.1	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	10	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.1	0	1.2	6.8	0

Table 8.5: Initial conditions for the three planets shown in Figure 8.4.

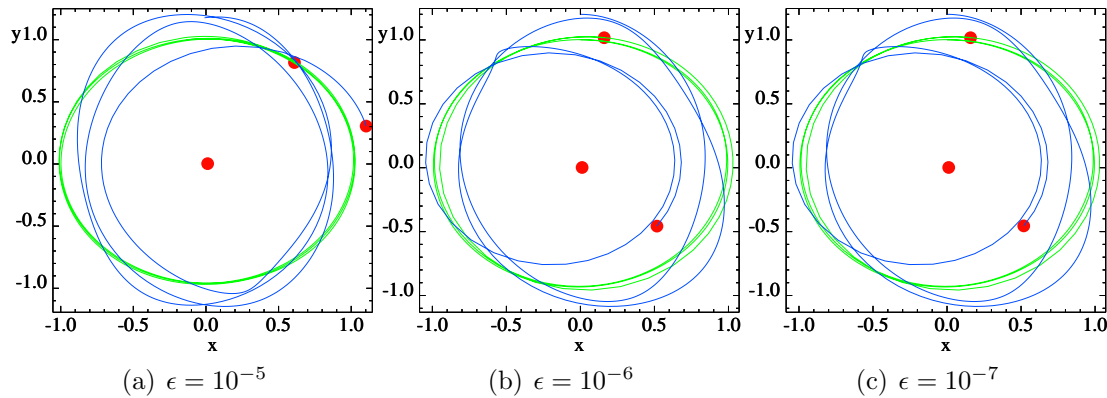


Figure 8.5: Comparison of planetary motion for different time-resolutions ϵ . Other parameters are the same as for Figure 8.4(d).

to the planet. Using the provided program examine the range of stability more carefully. In particular examine the motion of the system for at least three orbits of the planets and determine if the moon is likely to remain bound to the planet. Using the same parameters as proposed in the text, try to determine the limits of v_{3y} for which a bound moon can be observed. Can you give a physical argument for why the moon fails to be bound at either end of the velocity spectrum?

HINT: To do this effectively you want to write a subroutine that scans through a range of initial velocities of the moon and then record a relevant measurement (like in the Bunny program). This relevant measurement could be the minimum and maximum distance between the planet and the moon, averaged over several orbits of the planet.

8.2.2: We observed that all the bound trajectories for the moon were trajectories where the moon rotated in the opposite direction to the rotation of the planet around the sun. Can you find a set of parameters for which the moon rotates in the opposite direction? If yes, please give these parameters, if not, explain why not.

8.3 Simulating chaotic motion

The problem of accuracy we mentioned in the last chapter raises a complication arising from the following observations

1. numerical errors will continue to add up during the simulation
2. if we want to ensure that we obtain an accurate solution for a given initial condition we have to choose the time-step accordingly small

To see why this causes a difficulty assume that you have simulated a system from some initial conditions at time t_0 to time t_1 within some specified error ϵ . Now assume that

m_0	x_0	y_0	v_{x0}	v_{y0}
1	0	0	-0.018	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	0.1	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.1	0	1.4	0.08	0

Table 8.6: Initial conditions for the three planets shown in Figure 8.6.

you also want to know the state the system will be in at some even later time t_2 . If you want to know this state also with the same error of ϵ , then you could not start with the state at time t_1 , which you already know, because you only know this state with a certainty of ϵ . This would (likely) not allow you to know the state of the system at time t_2 with that same accuracy and you would have to start your simulation with the initial condition at time t_0 and iterate your system with a higher accuracy.

This now raises the question as to how the error increases with time. This is an interesting question and the answer strongly depends on the kind of system we are looking at. For “well behaved” systems the error will not increase more than linearly with time, i.e. if we simulate the system twice as long, the error will be twice as big. An example for such a system is the two-body system where we saw that Euler simulations will show an increase of the energy with time and energy increased linearly in time. The same error would have been found if we had looked at the deviation of the position from its analytical value or the velocity.

Many other system, however, have the property that such errors increase not linearly with time but instead exponentially! These systems are called “chaotic”. Many systems, including the three body problem, show this property. But why should the three body problem show such a radically different behavior than the two-body system? After all the forces are similar, the numerical algorithm is the same, so what makes this system different? An example can be found already in Figure 8.5 where we saw that a scattering event of two bodies caused a strong dependence of the scattering angle on the impact parameter.

If you have ever played billiard you know this from personal experience: it is relatively easy to hit one ball. It is a little harder to control the direction the ball you hit will go. Now if you direct this second ball to hit a third one even a professional player will find it next to impossible to control the direction of the third ball with any accuracy.

Let us now consider the situation in Figure 8.6, where three bodies with equal mass and a sufficiently small initial velocity are circulating each other. It is apparent that the motion of these bodies has a somewhat random appearance. But does this mean that the behavior of the system is chaotic in the sense that the behavior is sensitive to the initial conditions?

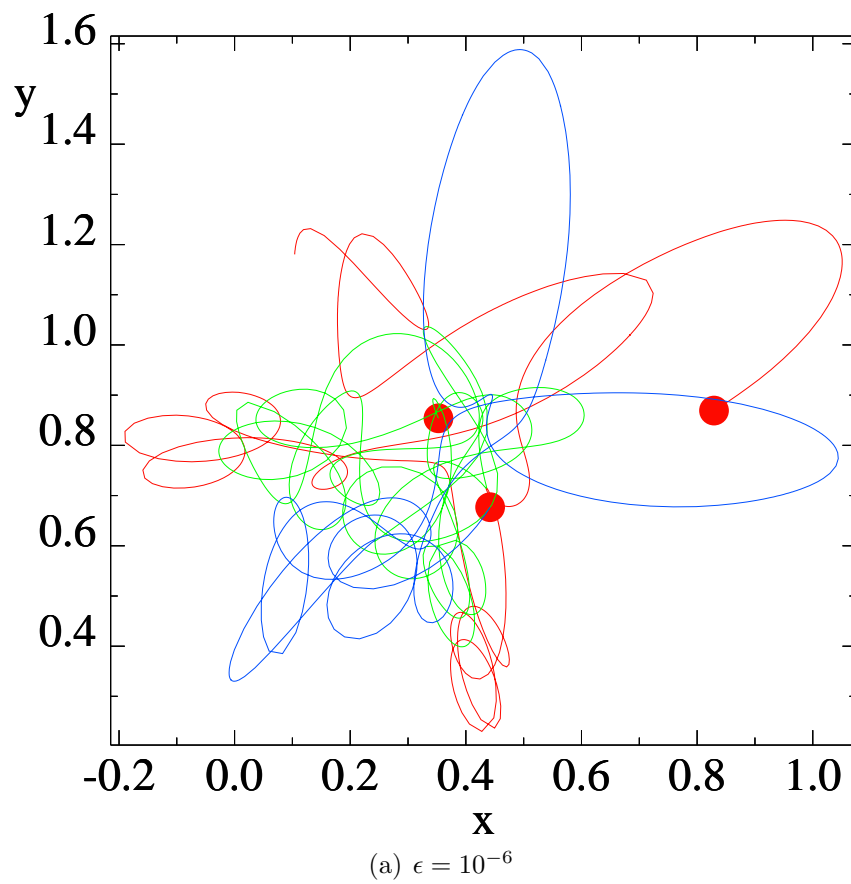


Figure 8.6: Three bodies with equal mass undergoing chaotic motion. The initial conditions are given in Table 8.6.

m_0	x_0	y_0	v_{x0}	v_{y0}
1	-1.359	2.632	-1.3745	7.694
m_1	x_1	y_1	v_{x1}	v_{y1}
1	-1.374	2.637	1.113	-6.795
m_2	x_2	y_2	v_{x2}	v_{y2}
1	-0.652	2.187	-1.096	1.147
	-0.672			
	-0.692			
	-0.74			
	-0.75			

Table 8.7: Initial conditions for the three planets shown in Figure 8.7.

To test this question qualitatively we let the system run for a while and then we save the state. I picked a situation where two particles (red and green) are closely circling each other and then the particle with the blue trace approaches the pair. In the original simulation the blue particle approaches the red particle closely and scatters, then the red particle closely interacts with the green particle and the green particle gets scattered away leaving the red and blue particles to circle each other closely.

Is this a situation where the behavior of the system is strongly dependent on the initial conditions? We can restore the same initial condition and slightly change the initial position of the blue particle and observe what the effect of this change is. The result of this numerical experiment is shown in Figure 8.7. We see that a slight change in the initial position of the blue particle can have a pronounced effect on the outcome of the simulation. If the change is “small enough” the trajectories remain quite similar, but as the change is increased the qualitative behavior of the scattering process changes.

In principle we should examine next is how the distance between two trajectories with similar initial conditions changes as a function of a small displacement. But this is a difficult undertaking: we see that if we change the trajectory only infinitesimally we expect to get a result that will remain quite similar, whereas a larger deviation can quickly lead to a completely different solution.

Mathematically we want to look at the state-vector as a function of time, $v(t)$, as it develops from an initial state $v(t) = v_{in}$. We want to compare this trajectory with another trajectory $v_\epsilon(t)$ that evolves from a slightly different initial condition $v_\epsilon(0) = v_{in} + \epsilon \delta v_{in}$. The difference of this trajectories δv then evolves as

$$\delta v(t) = v(t) - v_\epsilon(t). \quad (8.14)$$

To lowest order in ϵ we then expect this difference to be a linear function of the initial conditions. We can then write

$$\delta v(t) = A(t) \epsilon \delta v_{in} \quad (8.15)$$

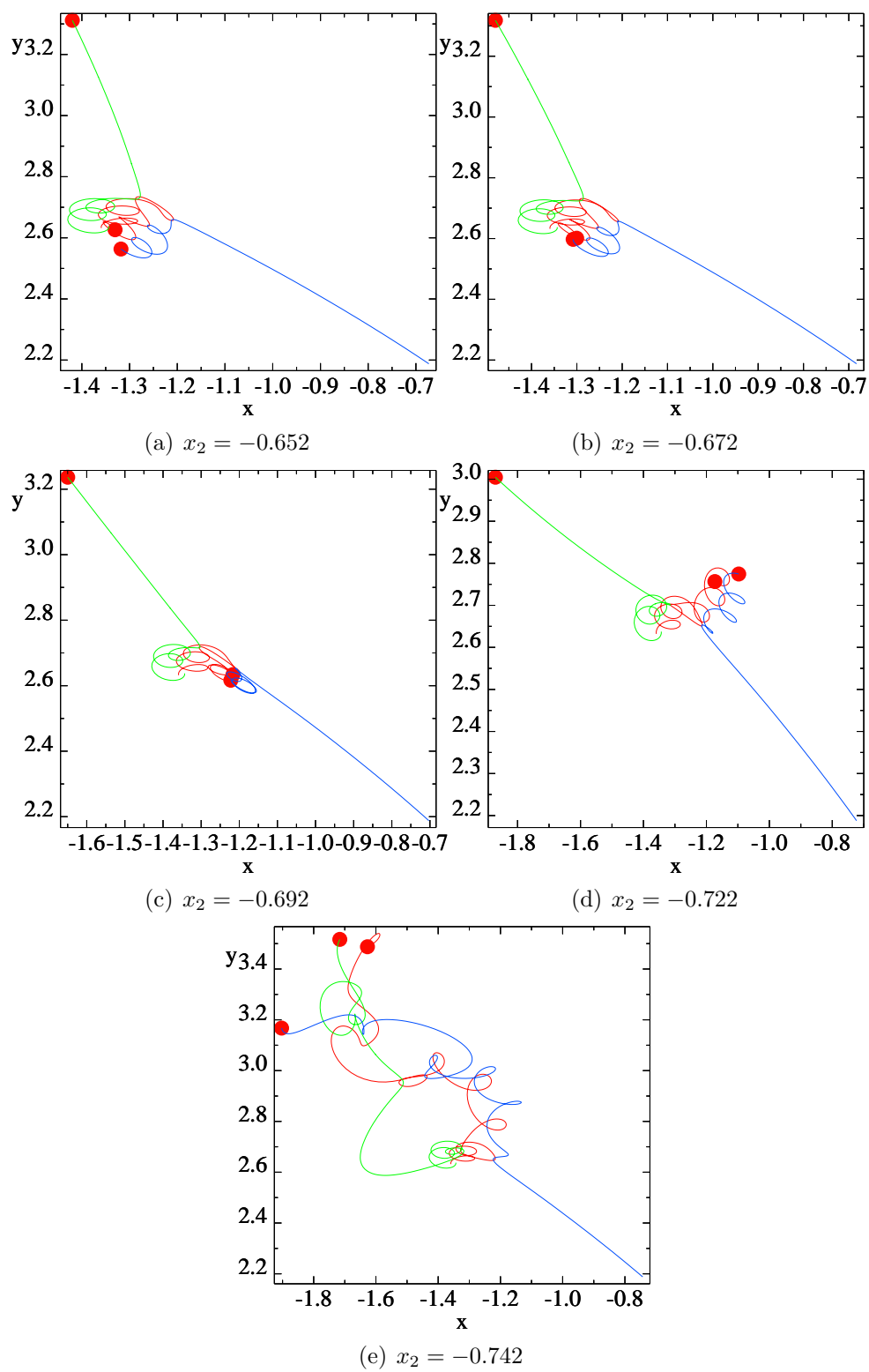


Figure 8.7: Three bodies with equal mass undergoing chaotic motion. The initial conditions are given in Table 8.7.

where it is reasonable to expect the matrix $A(t)$ to be independent of ϵ for small enough ϵ^2 . $A(0)$ is the identity matrix and from there the matrix evolves in time.

Both the trajectories obey the same first order differential equation

$$\frac{dv(t)}{dt} = a(t) \quad (8.16)$$

where $a(t)$ is a vector which contains first the Nd velocities of the N particles in d dimensions, and then the Nd accelerations:

$$a(t) = \begin{pmatrix} v_{1x}(t) \\ v_{1y}(t) \\ \vdots \\ v_{Nx}(t) \\ v_{Ny}(t) \\ F_{1x}(t)/m_1 \\ F_{1y}(t)/m_1 \\ \vdots \\ F_{Nx}(t)/m_N \\ F_{Ny}(t)/m_N \end{pmatrix} \quad (8.17)$$

The differential equation governing the evolution of $\delta v(t)$ is given by

$$\frac{d(\delta v(t))}{dt} = \frac{dv(t)}{dt} - \frac{dv_\epsilon(t)}{dt} = \begin{pmatrix} \partial_\epsilon v_{\epsilon 1x}(t) \\ \partial_\epsilon v_{\epsilon 1y}(t) \\ \vdots \\ \partial_\epsilon v_{\epsilon Nx}(t) \\ \partial_\epsilon v_{\epsilon Ny}(t) \\ \partial_\epsilon F_{\epsilon 1x}(t)/m_1 \\ \partial_\epsilon F_{\epsilon 1y}(t)/m_1 \\ \vdots \\ \partial_\epsilon F_{\epsilon Nx}(t)/m_N \\ \partial_\epsilon F_{\epsilon Ny}(t)/m_N \end{pmatrix} \epsilon \quad (8.18)$$

Averaging over this quantity will give you a measure of how fast a trajectory will diverge. The logarithm of this quantity is related to the Lyapunov exponent. In dependence on the initial disturbance there will be a full matrix of elements and the eigenvalues of this matrix are known as the Lyapunov exponent. A Lyapunov exponent larger than 1 corresponds to the famous “sensitivity to initial conditions” since solutions will diverge exponentially.

²If you are a mathematician you can certainly conjure up situation where this would not be the case, like two point particles in a head-on collision. Any deviation of the path towards the one or the other direction will lead to a very different outcome.

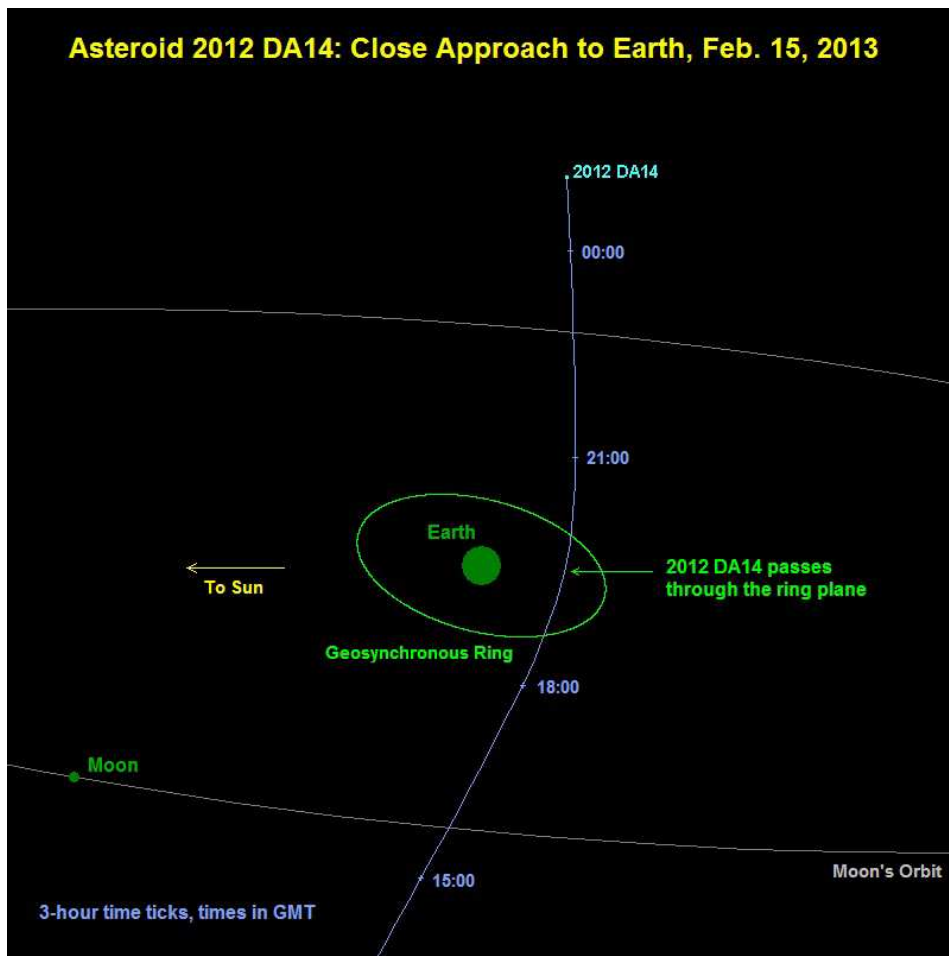


Figure 8.8: Predicted trajectory of Asteroid 2012 DA14. Credit: NASA

However, these mathematical quantities only tell you about infinitesimal disturbances and finite disturbances (i.e. where we cannot neglect higher powers of ϵ) can show a different behavior.

Let us examine this situation with a practical example (from earthsky.org):

Asteroid 2012 DA14 will pass closest on February 15, 2013. As shown in Figure 8.8, it will pass much closer than the orbit of the moon - closer even than orbiting geosynchronous satellites (22,000 miles). So, no, 2012 DA14 won't strike us in 2013. There was a remote possibility it might strike us in 2020, but that possibility has been ruled out also. What will happen when Asteroid 2012 DA14 passes closely in 2013?

What will happen when it passes us? The short answer is nothing. On the day it passes, most of us won't see it or be aware of its passage, in any way. The asteroid won't alter the tides. It won't cause volcanoes. It'll just sweep closely past us as millions of asteroids have done throughout Earth's four-and-a-half-billion-year history some in your own lifetime.

The asteroid will be within range for small telescopes and solidly mounted binoculars, used by experienced observers who have access to appropriate stars charts. Here's what NASA says about its visibility:

On [February 15, 2013], the asteroid will travel rapidly from the southern evening sky into the northern morning sky with its closest Earth approach occurring about 19:26 UTC when it will achieve a magnitude of less than seven, which is somewhat fainter than naked eye visibility. About 4 minutes after its Earth close approach, there is a good chance it will pass into the Earth's shadow for about 18 minutes or so before reappearing from the eclipse. When traveling rapidly into the northern morning sky, 2012 DA14 will quickly fade in brightness.

What do we know about asteroid 2012 DA14?

Asteroid 2012 DA14 is a little guy, compared to some asteroids, although its size has not been pinned down precisely. It is thought to be about 45 meters across (nearly 150 feet across), with an estimated mass of about 130,000 metric tons.

If a space object 150 feet wide were to strike our planet, it wouldn't be Earth-destroying. But it has been estimated that it would produce the equivalent of 2.4 megatons of TNT. How does that compare with other known impact events on Earth? In 1908, in a remote part of Russia, an explosion killed reindeer and flattened trees. But no crater was ever found. Scientists now believe a small comet struck Earth. That event has been estimated at 3 to 20 megatons. So 2012 DA14 is in the same approximate realm as the Tunguska comet (which, actually, might have been an asteroid instead). It would not destroy Earth, but it could flatten a city.

Of course, about 70% of our world is covered by oceans. That means the most likely landing spot of any incoming asteroid is in the water—not on a city or other populated area.

Astronomers at the Observatorio Astronómico de La Sagra in Spain discovered 2012 DA14 in early 2012. We know 2012 DA14's orbit is similar to that of Earth. That is one reason the asteroid eluded astronomers until recently. You can be sure that many astronomers are carefully tracking 2012 DA14 now.

The orbit of 2012 DA14 is an inclined ellipse. In other words, it's tilted slightly with respect to Earth's orbit around the sun, and, like Earth's orbit, it's not circular but elliptical—like a circle that someone sat down on. According to Bad Astronomer Phil Plait, who appears to have used a computer program to look at its orbit:

The asteroid spends most of its time well away from our planet. However, the path of the rock does bring it somewhat close to the Earth twice per orbit, or about every six months. The last time it passed us was on February 16 [2012], when it was about 2.5 million km (1.5 million miles) away, equal to about 6 times the distance to the moon. That's usually about the scale of these encounters—it misses us by quite a margin.

If we know it will miss us in 2013 and in 2020, why are astronomers still watching? In fact, the orbit of 2012 DA14 is not entirely pinned down, although it is known well enough to say for sure: it will not hit us next year, or in 2020.

But it will come close on February 15, 2013! It should be close enough to catch the attention of virtually everyone on Earth in February 2013, on what's sure to be a media field day.

Will 2012 DA14 strike Earth in 2020?

No. In March 2012, when a collision between 2012 DA14 and Earth in 2020 was still remotely possible, I asked astronomer Donald Yeomans to clarify the risk. Yeomans is, among other things, manager of NASA's Near-Earth Object Program Office at NASA's Jet Propulsion Laboratory. In March 2012, he told EarthSky that a 2020 collision between Earth and asteroid 2012 DA14 was

approximately one chance in 83,000, with additional remote possibilities beyond 2020. However, by far the most likely scenario is that additional observations, especially in 2013, will allow a dramatic reduction in the orbit uncertainties and the complete elimination of the 2020 impact possibility.

It turned out they didn't have to wait until 2013. By May, 2012, astronomers had ruled out even the remote possibility of a 2020 collision.

Still, 2012 DA14 and asteroids like it are sobering.

Bottom line: The near Earth asteroid 2012 DA14 will have a very close pass near Earth on February 15, 2013. It will sweep approximately 21,000 miles from us—much closer than the moon's orbit and closer than geosynchronous satellites. It will not strike Earth. Its orbit around the sun can bring it no closer to the Earth's surface on February 15, 2013 than 3.2 Earth radii.

The situation described above can be rephrased in the following terms: we know reasonably well the orbit of the earth and the moon, but the orbit of the asteroid is not yet precisely known. As we observe the asteroid using telescopes we can get an estimate on the asteroid's position and velocity. But these estimates are subject to measurement errors. So we want to know what the possible trajectories are, given the uncertainty in the position and velocity of the asteroid.

How would you go about addressing such a problem?

The simplest answer is to run the program with a number of different initial conditions. Ideally we will be able to see all the solutions at the same time. We can define an array of state-vectors and an array of initial conditions and then show the positions for all the different realizations at the same time. If the uncertainty is irrelevant for the problem at hand we will find that all states corresponding to the different initial conditions remain closely spaced together. In our example above we would expect all the asteroid trajectories to remain closely spaced together.

A new version of our planets program that accomplishes this is included here:

Listing 8.4: MD-ensemble.c

```
1 #include <stdlib.h>
  #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
```

```

#include <mygraph.h>
6 #include <math.h>

#define pi 3.14159265358979323846264338327950288419716939937

#define M 150 /* Maximum number of ensembles */
11 int mp1=M; /* Initial number of ensembles */
#define N 6 /* Number of coordinates (or momenta) */
double v[M+1][N*2],vin[N*2]={0,0,0,1,0,1.2,-0.107,0,10,0,7,0},
    dvin[N*2]; /* The state vector of the system, initial state
    */
/* We define a general state vector consisting of N coordinates
    q and N momenta p. It reads (q1,q2,...,qN,p1,p2,...pN). */
int RandInitial=0;
16 double m[M+1][N/2],dm[N/2],G=1;
int N02=N/2; /* Needed for the graphics library */
double Eps=1e-5;

#define NG 2+N/2 /* Graphs for display: Could be anything E(t),
    L(t) ...*/
21 #define MEM 1000
double x1[NG][MEM][2];
double x2[N/2][M][2]; /* instantaneous position of ensemble
    particles */
char *GrName[NG]={"(t,E)","(t,L)","(x1,y1)","(x2,y2)","(x3,y3)"};
/*needs to be more general...
char *Gr2Name[N/2]={ "Ens1","Ens2","Ens3"};
26 int size=MEM;
double time=0,iterations=0;

/* States: we now have N/2 particles in two dimensions */

31 void FF(double v[N*2],double m[N/2], double F[N]){
    double rx,ry,rabs;
    int i,j;

    memset(&(F[0]),0,N*sizeof(double)); /* set the F array to
        zero */
36
    for (i=0;i<N/2;i++){
        for (j=i+1;j<N/2;j++){
            rx=v[i*2]-v[j*2];

```

```

    ry=v[i*2+1]-v[j*2+1];
41    rabs=sqrt(pow(rx,2)+pow(ry,2));
    F[i*2]+=-G*m[i]*m[j]/pow(rabs,3)*rx;
    F[i*2+1]+=-G*m[i]*m[j]/pow(rabs,3)*ry;
    F[j*2]+=-F[i*2];
    F[j*2+1]+=-F[i*2+1];
46    }
    }
}

double TimeStep(double v[N*2]){ // for gravitational
    interaction
51    int i,j;
    double rabs2,vabs2[N/2],t,dt=1e10;

    for (i=0;i<N/2;i++)
        vabs2[i]=pow(v[N+i*2],2)+pow(v[N+i*2+1],2); //vel of part i
56    for (i=0;i<N/2;i++){
        for (j=i+1;j<N/2;j++){
            rabs2=pow(v[i*2]-v[j*2],2)+pow(v[i*2+1]-v[j*2+1],2); //
                dist i/j
            if (dt>rabs2/vabs2[i]) dt=rabs2/vabs2[i];
            if (dt>rabs2/vabs2[j]) dt=rabs2/vabs2[j];
61        }
    }
    return Eps*sqrt(dt);
}

66 void (*Iterate)(double v[N*2], double m[N/2], double Dt)=NULL;

void IterateEuler(double v[N*2], double m[N/2], double Dt){
    double F[N];
71    int i;
    FF(v,m,F);
    for (i=0;i<N;i++){
        v[i]+=v[N+i]*Dt;
        v[N+i]+=F[i]*Dt/m[i/2];
76    }
}

void IterateVerlet(double v[N*2], double m[N/2], double Dt){

```

```

    double F[N], Fn[N];
81    int i;

    FF(v, m, F);
    for (i=0; i<N; i++){
        v[i] += (v[N+i] + 0.5*F[i]/m[i/2]*Dt)*Dt;
86    }
    FF(v, m, Fn); /* assuming that forces don't depend on velocity
                  */
    for (i=0; i<N; i++){
        v[N+i] += 0.5*(F[i] + Fn[i]) *Dt/m[i/2];
91    }

    void SetEuler(){
        Iterate = &IterateEuler;
    }
96    void SetVerlet(){
        Iterate = &IterateVerlet;
    }

101    double E(double v[N*2], double m[N/2]){
        double Eret=0, rx, ry, rabs;
        int i, j;

        for (i=0; i<N/2; i++){
106            Eret += 0.5*m[i] * (pow(v[N+2*i], 2) + pow(v[N+2*i+1], 2));
            for (j=i+1; j<N/2; j++){
                rx = v[i*2] - v[j*2];
                ry = v[i*2+1] - v[j*2+1];
                rabs = sqrt(pow(rx, 2) + pow(ry, 2));
111            Eret += -G*m[i]*m[j]/pow(rabs, 1);
            }
        }
        return Eret;
    }
116    double L(double v[N*2], double m[N/2]){
        double Lzret=0;
        int i;
        for (i=0; i<N/2; i++){

```



```

161     }
        /* Initialize the graphics variables */
        for (int i=0; i<size; i++){
            x1[0][i][0]=*time; x1[0][i][1]=E(v[0],m[0]);
            x1[1][i][0]=*time; x1[1][i][1]=L(v[0],m[0]);
166         for (j=0; j<N/2; j++){
            x1[j+2][i][0]=v[0][2*j];
            x1[j+2][i][1]=v[0][2*j+1];
        }
    }
171 }

void AnalyzeData(double v[M+1][N*2], double m[M+1][N/2], double
    x1[NG][MEM][2], double x2[N/2][M][2], double time){
    for (int i=0; i<NG; i++) // move data points back in graphics
        array
        memmove(&x1[i][1][0], &x1[i][0][0], (size-1)*2*sizeof(double)
            );
176
    x1[0][0][0]=time; x1[0][0][1]=E(v[0],m[0]);
    x1[1][0][0]=time; x1[1][0][1]=L(v[0],m[0]);
        for (int i=0; i<N/2; i++){
            x1[i+2][0][0]=v[0][2*i];
181         x1[i+2][0][1]=v[0][2*i+1];
        }
        for (int n=0; n<N/2; n++){
            for (int m=0; m<mp1; m++){
                x2[n][m][0]=v[m+1][2*n];
186         x2[n][m][1]=v[m+1][2*n+1];
            }
        }
    }

191 void SaveState(){
    for (int i=0; i<2*N; i++) vin[i]=v[0][i];
}

196 void init(){
    Initialize(v,m,x1,x2,&time); // Just a little wrapper to call
        from the menu
    }

```

```

int main () {
201   int i, Paused=1, Step=1, Repeat=1, done=0, Adapt=1;
      char str[100];
      double Dt=0.01, DelT=0.01;

      Initialize(v,m,x1,x2,&time);
206   SetVerlet();

      DefineGraphN_RxR("Pos",&(v[0][0]),&N02,NULL);
      DefineGraphN_RxR("Vel",&(v[0][N]),&N02,NULL);
      for (i=0;i<NG;i++)
211   DefineGraphN_RxR(GrName[i],&(x1[i][0][0]),&size,NULL);
      for (i=0;i<N/2;i++)
        DefineGraphN_RxR(Gr2Name[i],&(x2[i][0][0]),&mp1,NULL);

      StartMenu("Two_celestial_bodies",1);
216   DefineDouble("G",&G);
      StartMenu("Initial_State",0);
      for (i=0;i<N/2;i++){
        sprintf(str,"m_%i",i);
        DefineDouble(str,&m[0][i]);
221   if (i==2){sprintf(str,"dm_%i",i);
          DefineDouble(str,&dm[i]);}
        sprintf(str,"x_%i",i);
        DefineDouble(str,&vin[i*2]);
        if (i==2){sprintf(str,"dx_%i",i);
226   DefineDouble(str,&dvin[i*2]);}
        sprintf(str,"y_%i",i);
        DefineDouble(str,&vin[i*2+1]);
        if (i==2){sprintf(str,"dy_%i",i);
          DefineDouble(str,&dvin[i*2+1]);}
231   sprintf(str,"Vx_%i",i);
        DefineDouble(str,&vin[N+i*2]);
        if (i==2){sprintf(str,"dVx_%i",i);
          DefineDouble(str,&dvin[N+i*2]);}
        sprintf(str,"Vy_%i",i);
236   DefineDouble(str,&vin[N+i*2+1]);
        if (i==2){sprintf(str,"dVy_%i",i);
          DefineDouble(str,&dvin[N+i*2+1]);}
      }
      DefineBool("Random_Initial_states",&RandInitial);

```

```

241   DefineMod("#_of_Initial_points",&mp1,M);
      DefineFunction("SaveState",&SaveState);
      DefineFunction("Reinitialize",&init);
      EndMenu();
      DefineGraph(curve2d_,"Graph");
246   DefineFunction("Set_Euler",&SetEuler);
      DefineFunction("Set_Verlet",&SetVerlet);
      DefineDouble("time",&time);
      DefineDouble("Dt",&Dt);
      DefineDouble("DelT",&DelT);
251   DefineDouble("Eps",&Eps);
      DefineBool("Adapt",&Adapt);
      DefineInt("Repeat",&Repeat);
      DefineBool("Step",&Step);
      DefineBool("Paused",&Paused);
256   DefineBool("done",&done);
      EndMenu();
      while (!done){
          Events(1);
          DrawGraphs();
261   if (!Paused || !Step){
              Step=1;
              for (int n=0;n<mp1+1;n++){
                  for (double dtime=0;dtime<DelT;dtime+=Dt){
                      if (Adapt) Dt=TimeStep(v[n]);
266   if (dtime+Dt>DelT) Dt=DelT-dtime;
                      Iterate(v[n],m[n],Dt);
                  }
              }
              time+=DelT;
271   AnalyzeData(v,m,x1,x2,time);
          }
          else sleep(1);
      }
  }

```

The key features of this program are that we now have an array of state vectors (line 13). Also we define a level of confidence for the initial state vector. In the initial conditions are now set for all these state vectors in the function `Initialize`. There are two methods: We use random initial conditions (line 136) if the value of `RandInitial` is not zero. To accomplish this we use the random function `rand48()`, which returns a `double` value between 0 and 1. Or we will set the initial values on the corners of the hyper-cube. Since a high-dimensional hyper-cube has many values we will eventually

need more points than we reasonably can (which is why we also introduced the random initial values, which do not suffer from this discrepancy).

In the main routine (line 260) we now loop additionally over all ensembles (i.e. our loop over `m`), can call the `TimeStep()`, and `Iterate()` routines with all relevant state vectors.

There is a small subtlety: for the first time we allow a variable number of points to be displayed. This is done on line 210, where we have the number of ensembles `mp1`. This number is accessible through the menu due to the entry in line 238. We used a new kind of variable `DefineMod`. What this means that in this case we have an integer that can take on the values between 0 and `M`. If someone were to enter a larger number $n > M$ (which would likely crash the program) the GUI will replace the number with $n \% M$.

Another subtlety involves realizing that with variable step sizes, the different ensembles will be at different times after the same number of steps. Therefore I replaced the `Repeat` variable with `DeLT` variable that gives the time intervals that are iterated to until we look at the graphics again and call a routine that examines events (i.e. notices that someone clicked the menu and wants to interact with the program). This ensures that we compare the positions of the planets in the different ensembles at the same times.

Now we should examine some examples. Let us consider the scattering situation of Figure 8.7. We now set the initial condition at $x_2 = 0.7$, but for the third particle (with index 2) we allow for a variation of 0.001 on the positions, the velocity and the mass. For this simulation we leave the coordinates of the other particles unchanged. We then initiated 9 state vectors with random initial conditions within the allowed range of 0.001 in dimensionless units and examine the evolution of the system. Initially there is little difference between the different ensembles, as should be expected. This is shown in Figure 8.9.

Even after the blue particle has scattered for the first time and the green particle it originally scattered with interacted again with the red particle, the introduced scatter remains small and all ensembles follow a very similar trajectory. This is shown in Figure 8.9(b). After two more scattering events at time $t = 0.527$ there is still a coherent swarm of ensembles, but they clearly start to diverge.

The further evolution of our system is now shown in Figure 8.10. Even at time $t = 0.738$ the positions of the Ensembles form a clearly coherent bunch, giving you an estimate of your likely error due to your uncertainty regarding the initial conditions. At time $t = 0.908$ we see that the positions of the green particle in the different ensembles now can be found in two separate patches. If we had included more particles it is likely that our “probability cloud” would not be disjointed, but would remain connected. At this point it is clear that we would need a much large number of ensemble points to get a decent representation of the probability distribution of the particle positions consistent with the uncertainty of our knowledge of the initial conditions.

By time $t = 0.58$ some ensembles are still undergoing a scattering process, others have disengaged (like the sample system for which we are drawing the trajectories) into

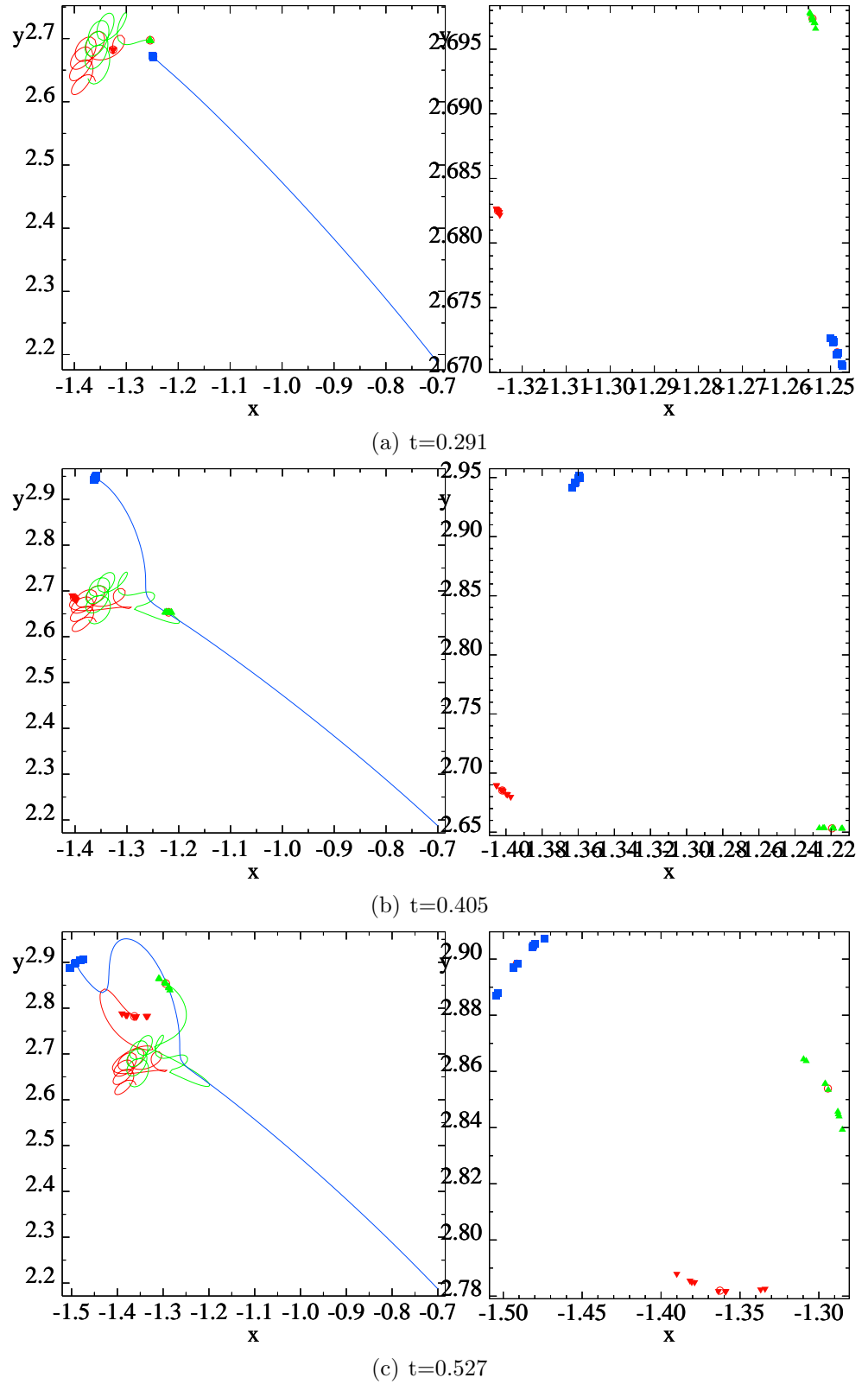


Figure 8.9: Scattering experiment for initial conditions given in Table 8.7 with the exception of $x_2 = 0.7$. Nine different ensembles with a scattering of 0.001 (in dimensionless units) for position, velocity and mass of the third (blue) particle are shown. We see that initially very little scattering is visible but the ensembles diverge as time goes on.

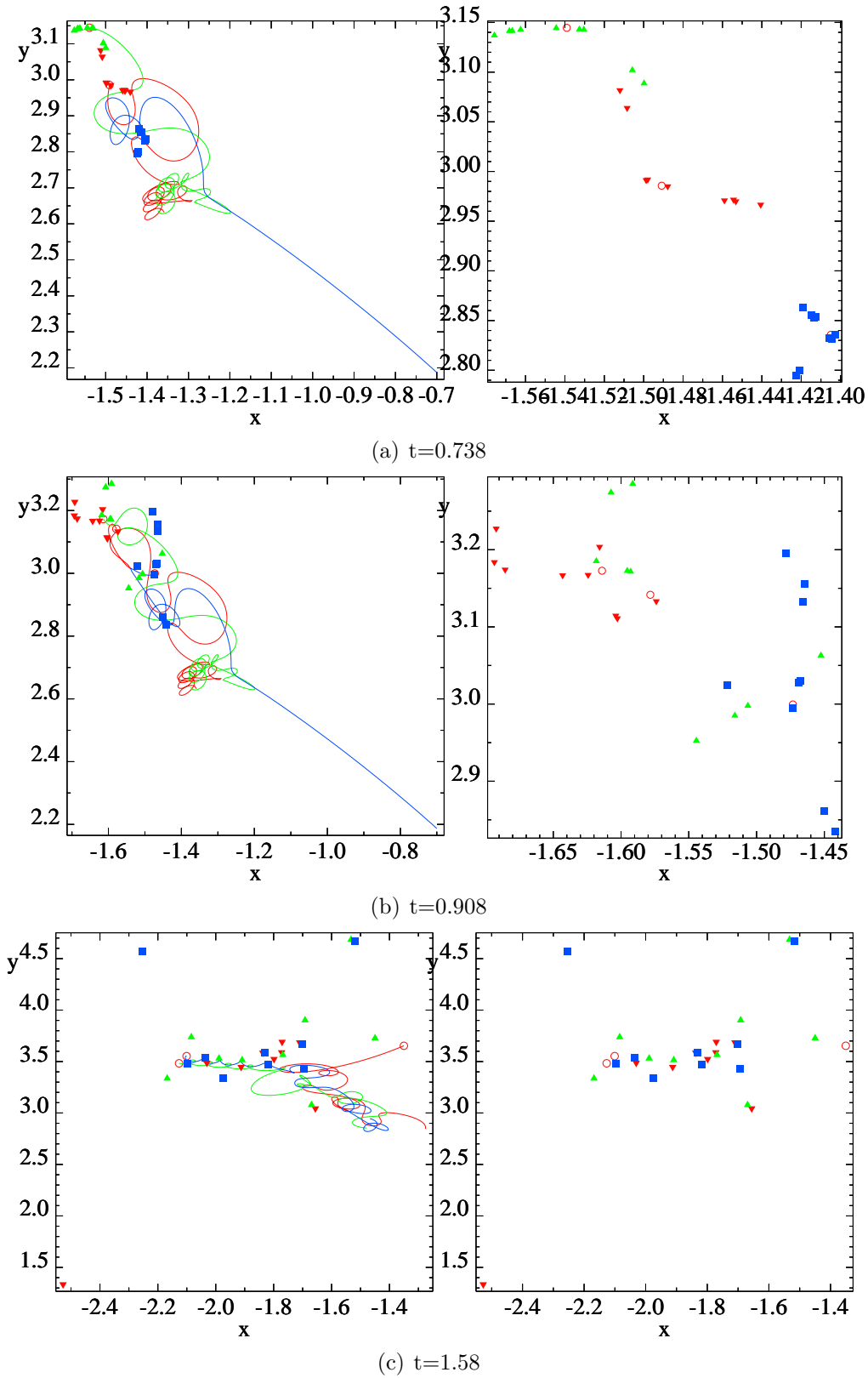


Figure 8.10: Same as Figure 8.9, but at later times. We see that at the last time the ensembles have lost most of their original correlation.

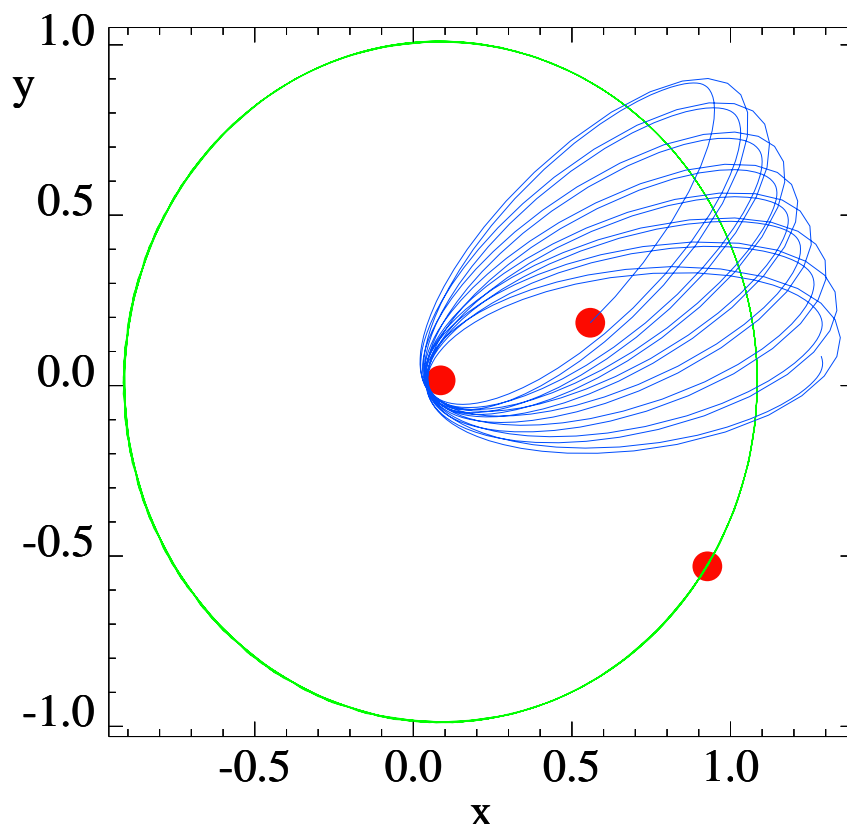


Figure 8.11: Example of non-trivial periodic motion in the three body problem. The parameters for this simulation are given in Table 8.8.

a single scattered particle and a pair of particles orbiting each other.

Note: From our previous numerical experiments one may expect that all three particle systems exhibit chaotic motion. However, this is not always the case. A fun example is shown in Figure 8.11, with parameters similar to the ones we used for the sun-earth-moon system of Figure 8.4, except that the light body (with the blue trace) is a factor of 100 times less massive. The trajectory of the light body is now a spiral which repeats after rounding the heavy “sun” in the middle twice, only with a specific new incline angle. At this point I happened to come across this interesting configuration after playing with the program for a while.

Problems

8.3.1: Use the `MD-ensemble.c` program to simulate a two-particle system with a certain amount of uncertainty in the initial conditions. What do you observe about the divergence of ensembles with different initial conditions?

8.3.2: Use the same program to simulate a three-particle system of your choice. Does

m_0	x_0	y_0	v_{x0}	v_{y0}
100	0	0	-0.1	0
m_1	x_1	y_1	v_{x1}	v_{y1}
1	0	1	10	0
m_2	x_2	y_2	v_{x2}	v_{y2}
0.001	0	1.2	2	0

Table 8.8: Initial conditions for the three planets shown in Figure 8.11.

the divergence behavior of the system appear to change?

8.3.3: To be a bit more quantitative in the analysis above, consider what is a good quantitative measure of the observed divergence of the ensembles. Use the measure you devise to calculate the divergence of the different ensembles in the two systems you examined above. You will want to create a new graph by increasing `NG`, thereby introducing another entry in the `x1` array and making sure that you include the new name in the `GrName[NG]` array.

8.3.4: The observation in the last note gives rise to an interesting observation: the trajectory in Fig. 8.11 shows a clear double periodicity. Can you quantify the periodicity? Once you have done this do a plot (similar to the plot in the bunny graph) of the local minima of the distance as a function of the initial velocity of the light object. For your initial conditions use a window around the initial values given in Table 8.8. Use this to identify the window in which we have a stable configuration and give an idea for what causes unstable orbits or planet ejections. **HINT:** I recommend following the suggestion Tyler made in class regarding finding the minimal distance: simply follow the distances of the two lighter objects and once this distance has gone through a minimum simply record that distance.

8.4 Planets with internal structure

So far we have treated our celestial bodies as point particles. That misses important structure: the particles can hardly ever collide, there are no tidal forces and no means to dissipate energy. For many fundamental problems, like the formation of a planets in a solar system, or even the slowing down of a moon circulating around a planet due to the tidal forces are not accessible in such a system.

We discussed that we wanted to simulate the effect of tides. However, how should one model such an extended planet? There are many discussions that treat the planet basically as a fluid drop which has a viscosity and reacts to the gravitational interaction with another celestial object. However, such treatments are not simple and while it is possible to obtain analytical solutions in simple cases, it is not so easy to write out these forces in all generality.

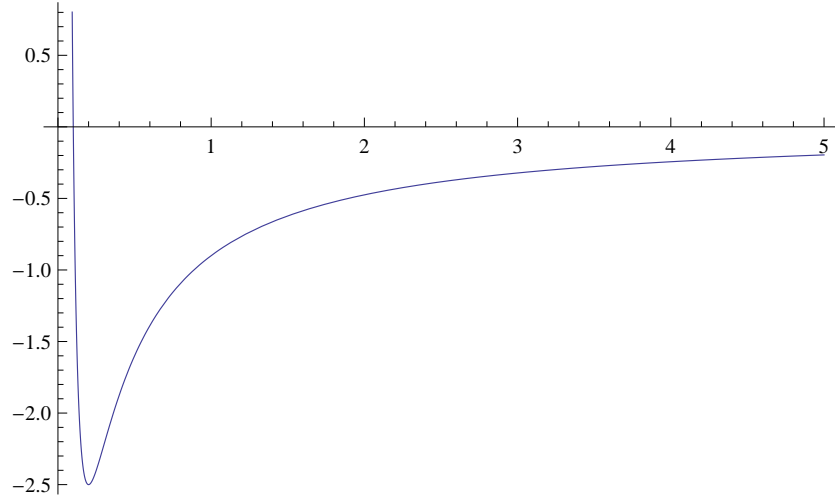


Figure 8.12: Potential for our gravitationally interacting particles with a repulsive core.

Having thought about this problem for a while I decided that we should at least try to take a computational approach to the problem and see how far we can get with this. We could imagine our planet to be made up of a set of mass blocks that show gravitational attraction up to a distance given by their radius, but then show a repulsive interaction that will prevent them from collapsing into each other. We achieve this by altering the potential of these particles so that they are repulsive at short distances.

A possible example of such a potential that behaves like $1/r$ at short distances but becomes repulsive at short distances would be

$$V(r) = Gm_1m_2 \left(\frac{r_0^3}{4r^4} - \frac{1}{r} \right). \quad (8.19)$$

This potential is graphed in Figure 8.12. This potential has a very special distance: at a distance of r_0 there is a minimum in the potential, so it becomes possible for particles to aggregate and find an equilibrium position. This is easily seen: the requirement for an equilibrium position is

$$0 = \frac{dV(r)}{dr} \quad (8.20)$$

$$= Gm_1m_2 \left(-\frac{r_0^3}{r^5} + \frac{1}{r^2} \right) \quad (8.21)$$

$$\Leftrightarrow 1 = \frac{r_0^3}{r^3} \quad (8.22)$$

$$\Leftrightarrow r = r_0 \quad (8.23)$$

$$(8.24)$$

Numerically this is easily achieved. We simply change the interaction force between

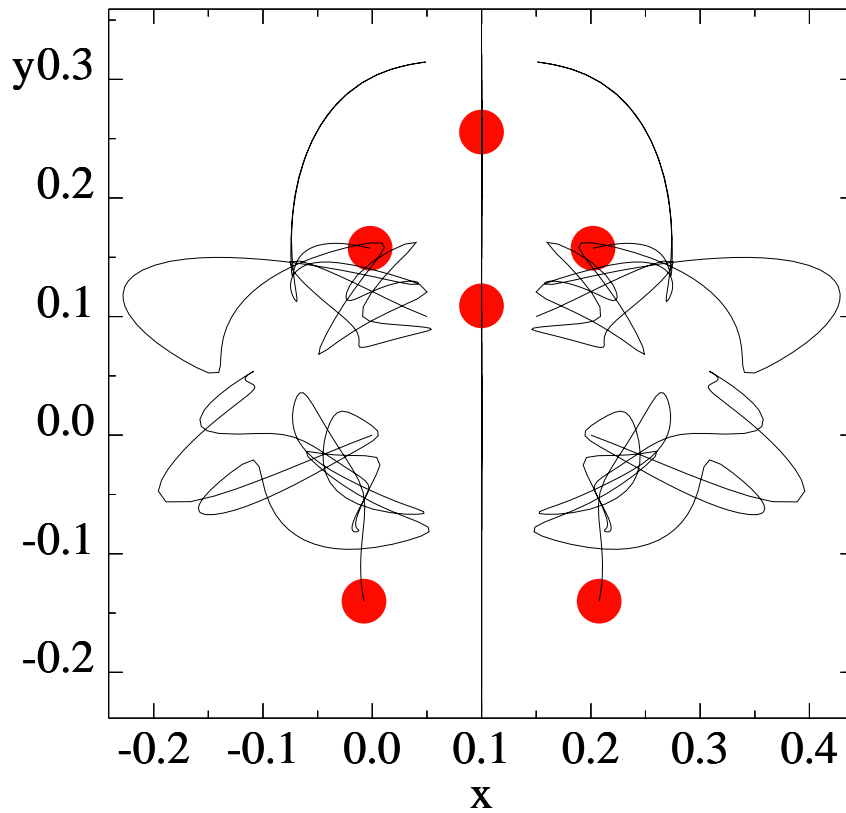


Figure 8.13: The trajectories of 6 particles set up on a triangular lattice initially. They do not settle down to a nice equilibrium shape and continue to move since there is no dissipation that would allow them to settle down.

the particles to

$$F_{1,2} = -Gm_1m_2 \left(-\frac{r_0^3}{r_{1,2}^6} + \frac{1}{r_{1,2}^3} \right) \mathbf{r}_{1,2}. \quad (8.25)$$

When we run a simulation with a number of particles packed closely together at a distance of the distance $r_0 = 0.1$, we see particles busily moving around. Firstly, the reason that the particles are moving around comes from the additional attractive interaction of the particles with other particles that are not their nearest neighbours. So the particles are spaced a little bit too far for their true equilibrium position. This extra energy induces oscillatory motion. Initially this motion is along the lines that they system symmetry requires. However, after a while numerical errors add up, and this symmetric solution becomes unstable. Instead the particles start to undergo a new kind of motion that has a “noisy” feel to it. In fact the particles move in a way that seems appropriate for the thermal motion of Atoms. An example of this is shown in Figure 8.13.

However, this motion does not seem reasonable for parts of a planet. So what is missing in our description, if we imagine the particles to represent parts of a planet? If you imagine that these planet-parts are moving relative to each other we would expect that there is some friction that will eventually bring these particles to rest. This friction will be (at least approximately) proportional to their relative velocity. So now we need to model a velocity dependent friction force.

This should sound familiar, since we introduced something similar when we considered a baseball flying with air resistance. In this case we would only want to have a friction force if the particles are close together, but the friction force should vanish when the particles are separating from each other.

Since we are making some very general assumptions here, we can introduce a velocity dependent friction force. What form does such a force take? To give a well founded answer to this problem we would need to carefully analyze the dynamics of an extended planetary body. That is an extremely difficult task and will depend strongly on its internal composition. A rocky planet will have a very different behavior from a gaseous giant. If we are interested in a generic phenomenon only, however, we can try to develop a minimal model that just includes enough of the Physical phenomena to allow us to see the effects of tidal interactions. So we can assume that the friction force becomes larger if the “planet parts” that are represented by our particles are closer together. Furthermore the force should be proportional to the relative velocity of the two particles. For the friction force \hat{F} between particle i and j we can then choose

$$\hat{F}_{i,j} = \text{Max} \left\{ 1 - \frac{1}{r_1}, 0 \right\} \eta (\mathbf{v}_j - \mathbf{v}_i) \quad (8.26)$$

where η is a friction coefficient and r_1 is the cut-off radius for the frictional particle interaction.

When we put this into the code we see that the thermal motion does indeed stop. Now we can test this code for a rotating planet, and this is when we find a surprise: the “planet” stops rotating! Why is this happening?

It is not unusual, when devising a simulation, to run into this kind of problem. So it is worth while to spend a moment to consider how one should react in such a situation.

My first instinct is to note that the behavior of this sytem clearly violates angular momentum conservation. So why does this system violate angular momentum conservation? To answer this question it is a good idea to recall why we expect angular momentum conservation in the first place. When you look back to your introductory mechanics you will find that angular momentum conservation is prooved fro system that only have central forces between point particles, i.e. forces that do not apply to torque. In our cas the potential interaction force of equation (8.25) is exactly of that form. However, our ad-hoc friction force of equation (8.26) does not have this form. Here the friction force does apply a torque!

At first glance this appears to be an insurmountable problem: if we don't have a friction force that is applied to particles moving sideways with respect to each other, then it is not a physical friction force. So how can we make this more physical?

To answer this question let us consider the simple case of two particles: If they are simply rotating around each other, there should be no friction force, but if they are passing by each other there should be a friction force. However, this is a difficult proposition as the situations look exactly the same as far as the friction forces are concerned.

If we imagine the physical system the difference between the two situations is that the particles are rotating in the case where they are moving around each other and they are not rotating in the case where they are just passing by each other. But in the case of our point-particles, we can't keep track of that difference.

So what is needed to resolve this situation is to include the angular momentum of the particle in the simulation. In a two-dimensional case the angular momentum is always in the direction orthogonal to the plane, so we need a single number to encode the angular momentum. In a three-dimensional case the angular momentum is a (pseudo-)vector, so we need to keep track or three components of the angular momentum.

Now we need to consider the angular momentum of our particles when we calculate the friction force. In equation (8.26) we should now consider the velocity difference of the velocity of the two rotating disks at the midpoint between the disks. This gives a new friction force of the kind

$$\hat{\mathbf{F}}_{i,j} = \text{Max} \left\{ 1 - \frac{1}{r_1}, 0 \right\} \eta \left[\left(\mathbf{v}_j + \boldsymbol{\omega}_j \times \frac{\mathbf{r}_{j,i}}{2} \right) - \left(\mathbf{v}_i + \boldsymbol{\omega}_i \times \frac{\mathbf{r}_{i,j}}{2} \right) \right] \quad (8.27)$$

To update the angular momenta L_i of the two particles we need to additionally calculate the torques they applying on each other. The torques for this simple interaction between two particles are given by

$$\boldsymbol{\tau}_i = -\hat{\mathbf{F}}_{i,j} \times \frac{\mathbf{r}_{i,j}}{2} \quad (8.28)$$

$$\boldsymbol{\tau}_j = \boldsymbol{\tau}_i. \quad (8.29)$$

Now let us consider if this satisfies the condition of angular momentum conservation. The change in angular momentum ΔL , obtained from the forcing terms in eqn. (8.27), is given by

$$\Delta \mathbf{L}^{part} = \hat{\mathbf{F}}_{i,j} \times \mathbf{r}_{i,j} \Delta t \quad (8.30)$$

The internal angular momentum is changed by

$$\Delta \mathbf{L}^{int} = (\boldsymbol{\tau}_i + \boldsymbol{\tau}_j) \Delta t = -\Delta \mathbf{L}^{part} \quad (8.31)$$

so that the angular momentum of the point particles that is destroyed through the friction force is added symmetrically to the internal angular momentum of each particle.

This needs to be put into the code. Here is the modified code version:

Listing 8.5: MD-rep2.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <mygraph.h>
5 #include <math.h>

#define pi 3.14159265358979323846264338327950288419716939937

#define N 14 /* Number of coordinates (or momenta) */
10 #define Nplanet 6 /* number of planetoid parts */
double v[N*2+N/2],vin[N*2+N/2]; /* The state vector of the
    system, initial state */
/* We define a general state vector consisting of N coordinates
    q and N momenta p. It reads (q1,q2,...,qN,p1,p2,...,pN). */
double m[N/2],MI[N/2],G=1,r0=0.1,r1=0.2,eta=0;
int N02=N/2; /* Needed for the graphics library */
15 double Eps=1e-5;

#define NG1 8
#define NG NG1+N/2 /* Graphs for display: Could be anything E(t
    ), L(t) ..*/
#define MEM 1000
20 double x1[NG][MEM][2];
char *GrName[NG]={"(t,E)","(t,L_orb)","(t,L_int)","(t,L_tot)","
    (t,lam1)","(t,lam2)","(t,tet)","(t,tetM)","(x1,y1)","(x2,y2)
    ","(x3,y3)","(x4,y4)","(x5,y5)","(x6,y6)","(x7,y7)"}; //
    needs to be more general...
int size=MEM;
double time=0,iterations=0;
```

```

25  /* States: we now have N/2 particles in two dimensions */

void FF(double v[N*2+N/2], double F[N+N/2]) {
    double rx, ry, rabs, vx, vy, Ft, Fx, Fy;
    int i, j;

30    memset(&(F[0]), 0, (N+N/2)*sizeof(double)); /* set the F array
        to zero */

    for (i=0; i<N/2; i++){
        for (j=i+1; j<N/2; j++){
35            rx=v[i*2]-v[j*2];
            ry=v[i*2+1]-v[j*2+1];
            rabs=sqrt(pow(rx,2)+pow(ry,2));
            Ft=-G*m[i]*m[j]*(-pow(r0,3)/pow(rabs,6)+1/pow(rabs,3));
            Fx=Ft*rx;
40            Fy=Ft*ry;
            if (rabs<r1){
                vx=(v[N+j*2]-v[2*N+j]*ry/2)-(v[N+i*2]+v[2*N+i]*ry
                    /2);
                vy=(v[N+j*2+1]+v[2*N+j]*rx/2)-(v[N+i*2+1]-v[2*N+i]*rx
                    /2);
                Fx+=(1-rabs/r1)*eta*vx;
45                Fy+=(1-rabs/r1)*eta*vy;
                F[N+i]+=0.5*(Fx*ry-Fy*rx);
                F[N+j]+=0.5*(Fx*ry-Fy*rx);
            }

50            F[i*2]+=Fx;
            F[i*2+1]+=Fy;
            F[j*2]-=Fx;
            F[j*2+1]-=Fy;
        }
55    }
}

double TimeStep(double v[N*2+N/2]) { // for gravitational
    interaction
    int i, j;
60    double rabs=1e38, vabs=1e-4, t;

    for (i=0; i<N/2; i++){

```

```

        t=pow(v[N+i*2],2)+pow(v[N+i*2+1],2);
        if (t>vabs) vabs=t;
65    for (j=i+1;j<N/2;j++){
            t=pow(v[i*2]-v[j*2],2)+pow(v[i*2+1]-v[j*2+1],2);
            if (t<rabs) rabs=t;
        }
    }
70    return Eps*sqrt(rabs/vabs);
}

```

```

void (*Iterate)(double v[N*2+N/2], double Dt)=NULL;
75
void IterateEuler(double v[N*2+N/2], double Dt){
    double F[N+N/2];
    int i;
    FF(v,F);
80    for (i=0;i<N;i++){
        v[i]+=v[N+i]*Dt;
        v[N+i]+=F[i]*Dt/m[i/2];
    }
    for (i=0;i<N/2;i++){
85        v[2*N+i]+=F[N+i]*Dt/MI[i];
    }
}

```

```

void IterateVerlet(double v[N*2+N/2], double Dt){
90    double F[N+N/2],Fn[N+N/2];
    int i;

    FF(v,F);
    for (i=0;i<N;i++){
95        v[i]+=(v[N+i]+0.5*F[i]/m[i/2]*Dt)*Dt;
    }
    for (i=0;i<N;i++){
        v[N+i]+=F[i]*Dt/m[i/2];
    }
100    for (i=0;i<N/2;i++){
        v[2*N+i]+=F[N+i]*Dt/MI[i];
    }
    FF(v,Fn); /* assuming that forces do depend on velocity */
    for (i=0;i<N;i++){

```



```

105     v[N+i]+=0.5*(-F[i]+Fn[i])*Dt/m[i/2];
        }
        for (i=0;i<N/2;i++){
            v[2*N+i]+=0.5*(-F[N+i]+Fn[N+i])*Dt/MI[i];
        }
110 }

void SetEuler(){
    Iterate = &IterateEuler;
}

115 void SetVerlet(){
    Iterate = &IterateVerlet;
}

120 double E(double v[N*2+N/2]){
    double Eret=0,rx,ry,rabs;
    int i,j;

    for (i=0;i<N/2;i++){
125     Eret+=0.5*m[i]*(pow(v[N+2*i],2)+pow(v[N+2*i+1],2))
        +0.5*MI[i]*pow(v[2*N+i],2);
        for (j=i+1;j<N/2;j++){
            rx=v[i*2]-v[j*2];
            ry=v[i*2+1]-v[j*2+1];
130     rabs=sqrt(pow(rx,2)+pow(ry,2));
            Eret+=-G*m[i]*m[j]*(-pow(r0,3)/(4*pow(rabs,4))+1/pow(rabs
                ,1));
        }
    }
    return Eret;
135 }

double L(double v[N*2+N/2],int type){
    double Lzret=0;
    int i;
140     for (i=0;i<N/2;i++){
        switch (type){
            case 0: Lzret+=m[i]*(v[i*2]*v[N+i*2+1]-v[i*2+1]*v[N+i*2]);
                break;
            case 1: Lzret+=MI[i]*v[2*N+i];
145         break;
        }
    }
}

```

```

        case 2: Lzret+=m[i]*(v[i*2]*v[N+i*2+1]-v[i*2+1]*v[N+i*2])+
            MI[i]*v[2*N+i];
        break;
    }
}
150 return Lzret;
}

double PlanetMoments(double v[N*2+N/2], double *lam1, double *
    lam2, double *tet){
    double M=0, X[2]={0,0}, xx=0, xy=0, yy=0;
155 int i;
    for (i=0; i<Nplanet; i++){
        M+=m[i];
        X[0]+=m[i]*v[2*i];
        X[1]+=m[i]*v[2*i+1];
160 xx+=m[i]*v[2*i]*v[2*i];
        xy+=m[i]*v[2*i]*v[2*i+1];
        yy+=m[i]*v[2*i+1]*v[2*i+1];
    }
    X[0]/=M;
165 X[1]/=M;
    xx/=M; xy/=M; yy/=M;
    xx-=X[0]*X[0];
    xy-=X[0]*X[1];
    yy-=X[1]*X[1];
170 *lam1=0.5*(xx+yy)+sqrt(pow(0.5*(xx-yy),2)+pow(xy,2));
    *lam2=0.5*(xx+yy)-sqrt(pow(0.5*(xx-yy),2)+pow(xy,2));
    *tet=atan2(xy,xx-*lam2);
    if (*tet<0) *tet+=M_PI;
}
175

void RemoveCMvel(double v[N*2+N/2]){
    double px=0, py=0, M=0;
    for (int i=0; i<N/2; i++){
        px+=m[i]*v[N+2*i];
180 py+=m[i]*v[N+2*i+1];
        M+=m[i];
    }
    px/=M;
    py/=M;
185 for (int i=0; i<N/2; i++){

```

```

    v[N+2*i]-=px;
    v[N+2*i+1]-=py;
  }
}
190
void SetVin(double vin[N*2+N/2]){
  for (int i=0;i<3;i++){
    m[i]=1;
    vin[i*2]=r0*i;
    195    vin[i*2+1]=0;
    vin[N+i*2]=vin[N+i*2+1]=0;
    MI[i]=2./5.*m[i]*pow(0.5*(r0+r1),2);
    vin[2*N+i]=0;
  }
  200  for (int i=3;i<5;i++){
    m[i]=1;
    vin[i*2]=(i-2.5)*r0;
    vin[i*2+1]=sqrt(3./4.)*r0;
    vin[N+i*2]=vin[N+i*2+1]=0;
    205    MI[i]=2./5.*m[i]*pow(0.5*(r0+r1),2);
    vin[2*N+i]=0;
  }
  for (int i=5;i<6;i++){
    m[i]=1;
    210    vin[i*2]=(i-4)*r0;
    vin[i*2+1]=2*sqrt(3./4.)*r0;
    vin[N+i*2]=vin[N+i*2+1]=0;
    MI[i]=2./5.*m[i]*pow(0.5*(r0+r1),2);
    vin[2*N+i]=0;
    215  }
  for (int i=6;i<7;i++){
    m[i]=1;
    vin[i*2]=(i-4)*r0;
    vin[i*2+1]=10*sqrt(3./4.)*r0;
    220    vin[N+i*2]= 3;
    vin[N+i*2+1]=0;
    MI[i]=2./5.*m[i]*pow(0.5*(r0+r1),2);
    vin[2*N+i]=0;
  }
  225  RemoveCMvel(vin);
}

```

```

void Initialize(double v[N*2+N/2],double x[NG][MEM][2],double *
    time){
    int i,j;
230    double l1,l2,t;
    *time = 0;

    for (i=0;i<2*N+N/2;i++) v[i]=vin[i];

235    /* Initialize the graphics variables */
    for (int i=0; i<size; i++){
        x[0][i][0]=*time;x[0][i][1]=E(v);
        x[1][i][0]=*time;x[1][i][1]=L(v,0);
        x[2][i][0]=*time;x[2][i][1]=L(v,1);
240    x[3][i][0]=*time;x[3][i][1]=L(v,2);
        PlanetMoments(v,&l1,&l2,&t);
        x[4][i][0]=*time;x[4][i][1]=l1;
        x[5][i][0]=*time;x[5][i][1]=l2;
        x[6][i][0]=*time;x[6][i][1]=t;
245    x[7][i][0]=*time;x[7][i][1]=atan2(v[N-1],v[N-2]);

        for (j=0;j<N/2;j++){
            for (int k=0;k<size;k++){
                x[j+NG1][k][0]=v[2*j];
250                x[j+NG1][k][1]=v[2*j+1];
            }
        }
    }
255
void AnalyzeData(double v[N*2],double x[NG][size][2],double
    time){
    double l1,l2,t,Xcm=0,Ycm=0,M=0;
    for (int i=0;i<NG;i++) // move data points back in graphics
        array
        memmove(&x[i][1][0],&x[i][0][0],(size-1)*2*sizeof(double));
260
    for (int i=0;i<N/2;i++){
        Xcm+=m[i]*v[2*i];
        Ycm+=m[i]*v[2*i+1];
        M+=m[i];
265    }
    Xcm/=M; Ycm/=M;

```

```

    x[0][0][0]=time; x[0][0][1]=E(v);
    x[1][0][0]=time; x[1][0][1]=L(v,0);
270  x[2][0][0]=time; x[2][0][1]=L(v,1);
    x[3][0][0]=time; x[3][0][1]=L(v,2);
    PlanetMoments(v,&l1,&l2,&t);
    x[4][0][0]=time;x[4][0][1]=l1;
    x[5][0][0]=time;x[5][0][1]=l2;
275  x[6][0][0]=time;x[6][0][1]=t;
    x[7][0][0]=time;x[7][0][1]=atan2(v[N-1]-Ycm,v[N-2]-Xcm);
    for (int i=0;i<N/2;i++){
        x[i+NG1][0][0]=v[2*i];
        x[i+NG1][0][1]=v[2*i+1];
280  }
    }

void SaveState(){
    for (int i=0;i<2*N+N/2;i++) vin[i]=v[i];
285  }

void init(){
    Initialize(v,x1,&time); // Just a little wrapper to call from
                           the menu
290  }

int main (){
    int i, Paused=1, Step=1, Repeat=1, done=0, Adapt=1;
    char str[100];
295  double Dt=0.01;

    SetVin(vin);
    Initialize(v,x1,&time);
    SetVerlet();
300

    SetDefaultColor(2);
    SetDefaultLineType(0);
    SetDefaultShape(4);
    SetDefaultSize(8);
305  SetDefaultFill(1);
    DefineGraphN_RxR("Pos",&v[0],&N02,NULL);
    DefineGraphN_RxR("Vel",&v[N],&N02,NULL);

```

```

    for (i=0;i<NG;i++){
        SetDefaultColor(i+2);
310    SetDefaultShape(0);
        SetDefaultLineType(1);
        SetDefaultFill(0);
        DefineGraphN_RxR(GrName[i],&(x1[i][0][0]),&size,NULL);
    }
315 StartMenu("Multiple_celestial_bodies",1);
    DefineDouble("G",&G);
    DefineDouble("r0",&r0);
    DefineDouble("r1",&r1);
    DefineDouble("eta",&eta);
320 StartMenu("Initial_State",0);
    for (i=0;i<N/2;i++){
        sprintf(str,"m_%i",i);
        DefineDouble(str,&m[i]);
        sprintf(str,"x_%i",i);
325 DefineDouble(str,&vin[i*2]);
        sprintf(str,"y_%i",i);
        DefineDouble(str,&vin[i*2+1]);
        sprintf(str,"Vx_%i",i);
        DefineDouble(str,&vin[N+i*2]);
330 sprintf(str,"Vy_%i",i);
        DefineDouble(str,&vin[N+i*2+1]);
        sprintf(str,"Omega_%i",i);
        DefineDouble(str,&vin[2*N+i]);
    }
335 DefineFunction("SaveState",&SaveState);
    DefineFunction("Reinitialize",&init);
    EndMenu();
    DefineGraph(curve2d_,"Graph");
    DefineFunction("Set_Euler",&SetEuler);
340 DefineFunction("Set_Verlet",&SetVerlet);
    DefineDouble("time",&time);
    DefineDouble("Dt",&Dt);
    DefineDouble("Eps",&Eps);
    DefineBool("Adapt",&Adapt);
345 DefineInt("Repeat",&Repeat);
    DefineBool("Step",&Step);
    DefineBool("Paused",&Paused);
    DefineBool("done",&done);
    EndMenu();

```

```

350  while (!done){
        Events(1);
        DrawGraphs();
        if (!Paused || !Step){
            Step=1;
355      for (int i=0;i<Repeat;i++){
                if (Adapt) Dt=TimeStep(v);
                time += Dt;
                Iterate(v,Dt);
            }
360      AnalyzeData(v,x1,time);
        }
        else sleep(1);
    }
}

```

Firstly we test that angular momentum is now conserved. (Add more text here).

We can now use this code to simulate a compound planet and the effect a circulating moon has on this system. Some of the first results of such a simulation are shown in Figure 8.14. The first observation we should make is that the effect of any tidal forces is very small. There is a small increase in the orbital angular momentum of the system, which obtains its value from the internal rotation of the planetary bits. This is mostly an oscillatory process with a small drift, as one can see in the closeup in the second graph. At the same time the energy of the system slowly decreases, consistent with a small tidal friction. The configuration of the compound planet and its moon is shown in the last panel.

When you see these results of your first simulation you should a) be elated that you see the effect you were looking for, and b) be extremely wary about whether the effect you see is real. The changes we see here are very small, not unlike some of the earlier numerical errors we have seen. When you are confronted with a small effect, you have to be extra-weary about what you actually see in your simulation.

Furthermore, the actual tides are not visible on our roughly discretized planet, so we can't directly observe the phenomenon. So if we want to seriously consider the problem of tides, our work is just now beginning.

How can we see the tides in our compound planet? We would need to see that our compound planet is no longer isotropic. A typical way of looking at deviations from the spherical shape consists of looking at moments of the positions. In particular we can find the center of mass as

$$\mathbf{X} = \frac{\sum_i \mathbf{x}_i m_i}{\sum_i m_i} \quad (8.32)$$

where the sum is over all particles in the compound planet. However, we can also sum over two coordinates to obtain a tensor. To write this down effectively we express the vectors in terms of their coordinates and identify the coordinates by Greek indices. We

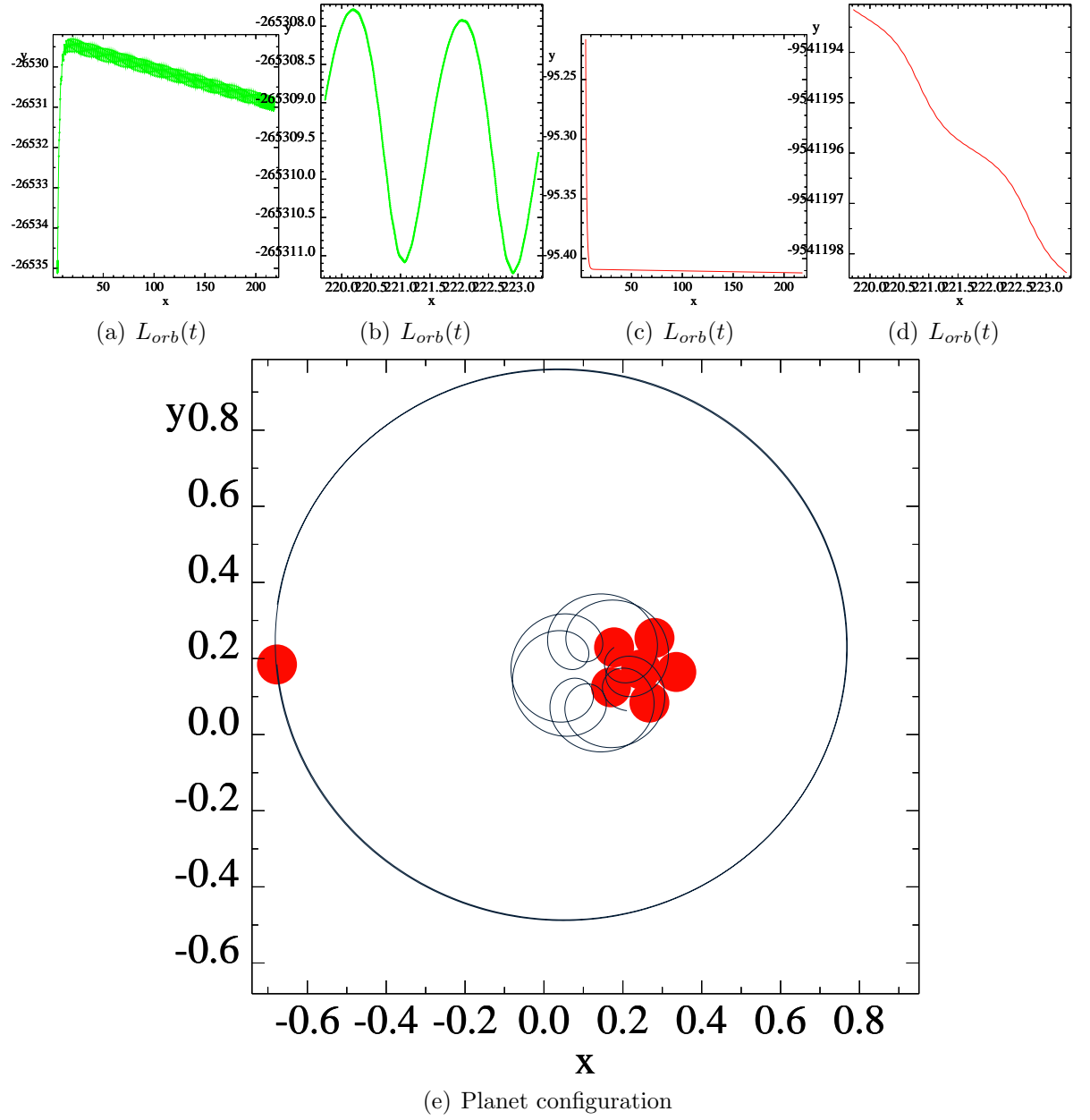


Figure 8.14: First attempt at simulating tidal effects with our new code.

can write (8.32) as

$$X_\alpha = \frac{\sum_i x_{i\alpha} m_i}{\sum_i m_i} \quad (8.33)$$

So we can write for the tensor

$$\Gamma_{\alpha\beta} = \frac{\sum_i (x_{i\alpha} - X_\alpha)(x_{i\beta} - X_\beta) m_i}{\sum_i m_i}. \quad (8.34)$$

By construction this tensor is symmetric, and a symmetric tensor has two real eigenvalues and two real eigenvectors. If the two eigenvalues are not identical, then we have a non-isotropic drop. The eigenvectors give us information about the orientation of the compound planet.

The eigenvectors are defined through the relation

$$\Gamma_{\alpha\beta} e_\beta = \lambda e_\alpha \quad (8.35)$$

and the number of eigenvectors is equal to the dimension of the matrix. From this we can see that

$$(\Gamma_{\alpha\beta} - \lambda \delta_{\alpha\beta}) e_\beta = 0 \quad (8.36)$$

The question we now need to answer is the following: for which values of λ does this equation have a solution? The answer is that the matrix multiplying the eigenvector has to be singular:

$$\det(\Gamma - \lambda I) = 0 \quad (8.37)$$

For our two-dimensional problem above this gives the so-called characteristic equation

$$(\Gamma_{xx} - \lambda)(\Gamma_{yy} - \lambda) - \Gamma_{xy}\Gamma_{yx} = 0 \quad (8.38)$$

$$\lambda^2 - (\Gamma_{xx} + \Gamma_{yy})\lambda + \Gamma_{xx}\Gamma_{yy} - \Gamma_{xy}\Gamma_{yx} = 0 \quad (8.39)$$

$$(8.40)$$

This quadratic equation has the solution

$$\lambda_{1,2} = \frac{\Gamma_{xx} + \Gamma_{yy}}{2} \pm \sqrt{\frac{(\Gamma_{xx} + \Gamma_{yy})^2}{4} - (\Gamma_{xx}\Gamma_{yy} - \Gamma_{xy}\Gamma_{yx})} \quad (8.41)$$

$$= \frac{\Gamma_{xx} + \Gamma_{yy}}{2} \pm \sqrt{\frac{(\Gamma_{xx} - \Gamma_{yy})^2}{4} + \Gamma_{xy}\Gamma_{yx}} \quad (8.42)$$

Once we know the eigenvalues we can solve eqn. (8.35) for the eigenvectors. We get

$$(\Gamma_{xx} - \lambda)e_x + \Gamma_{xy}e_y = 0 \quad (8.43)$$

$$\Gamma_{yx}e_x + (\Gamma_{yy} - \lambda)e_y = 0 \quad (8.44)$$

$$(8.45)$$

Eigenvectors are only defined up to a factor, so we can define the vector as

$$\begin{pmatrix} e_x \\ e_y \end{pmatrix} = \begin{pmatrix} \Gamma_{xy} \\ \Gamma_{xx} - \lambda_{1,2} \end{pmatrix} \quad (8.46)$$

Once we have the eigenvectors we know the orientation of the deformation of the compound planet.

Since the eigenvectors are only defined up to a factor they can be inverted and still be eigenvectors. This means that the angle we define here is only defined up to an arbitrary addition of an angle of 180° .

Now we can examine the angle of the moon, compared to the center of mass of the planet/moon system, to the angle of the eigenvector of the planet. On first look the two angles agree almost completely. Keep in mind that the angle of the eigenvector is only defined up to an arbitrary multiple of π . If there is to be an exchange of angular momentum between the planet and the moon, there has to be a difference in the two angles. Otherwise there could not be a torque, due to the symmetry.

So we want to monitor the difference between the two angles, and for small angles we expect the torque to be proportional to the difference between the two angles. This raises a number of important questions: what happens if there is no friction? Can we still transfer angular momentum between the moon and the planet? With friction, how long will it take for the planet to rotate in sync with the moon?

It is important to keep in mind that this is only a very simple model for tidal effects. However, we will be able to understand quite a bit of the basic physics from this simple model.

Problems

- 8.4.1:** Use the sample program (with minor additions) to measure the difference between the inclination angles for the deformation of the planet and the position of the moon. Explain carefully how the different angles are measured and what the importance of this difference is.
- 8.4.2:** Now measure the total angular momentum of the planet. Does this angular momentum change with time? Does the change of angular momentum depend on the viscous friction force (i.e. η)? Why is that?
- 8.4.3:** Assuming you found that the planet obtained angular momentum from the moon, how long will it take for the planetary rotation and the rotation of the moon to be the same? Try to make a simple model for this and test your prediction.
- 8.4.4:** Now give the planet a large amount of angular momentum. How much angular momentum can your planet have before it disintegrates? Explain this limit analytically.

8.5 Diffusion of particles in a box

So far we have considered particles that will attract each other with a gravitational potential. Such a potential is considered to be “long ranged”. To see what long-ranged means let us consider the following example: first let us consider material of a constant density ρ_1 making up object 1. This object interacts with another smaller body, which we call object 2. Now when object 2 approaches object 1 it makes a big difference if object 1 is big or small. This is so even though any additional mass of object 2 will have to be further away. We can also make this quantitative: if the interaction between object 1 and object 2 is purely gravitational we can express this analytically as

$$\mathbf{F}_{12} = \frac{m_1 m_2 G}{r_{12}^3} \mathbf{r}_{12} \quad (8.47)$$

Now the distance between the two centers depends on the total mass on object 1: say R is the distance between the center of object 2 to the surface of object 1, then $r_{12} = R + r_1$, where r_1 is the radius of object 1. Now we have the relation:

$$m_1 = \rho_1 V_1 = \rho_1 \frac{3}{4} \pi r_1^3 \quad (8.48)$$

and therefore we have

$$r_1 = \left(\frac{4m_1}{3\pi\rho_1} \right)^{\frac{1}{3}} \quad (8.49)$$

and for a given distance R the magnitude of the force is

$$F_{12} = \frac{m_1 m_2 G}{(R + r_1)^2} \quad (8.50)$$

$$= \frac{m_1 m_2 G}{\left(R + \sqrt[3]{\frac{4m_1}{3\pi\rho_1}} \right)^2} \quad (8.51)$$

Now let us assume for simplicity that R is much smaller than r_1 . In fact, let us just put it to zero to see what the force right at the surface of object 1 would be:

$$F_{12}(R = 0) = \frac{m_1^{1/3} m_2 G (3\pi\rho_1)^{2/3}}{4^{2/3}} \quad (8.52)$$

What is important here is that the Force does not converge to a constant value, no matter how big we make object 1. Another way of saying this is that we can never neglect the interaction with objects that are some fixed distance away. This is why we call forces with this property long-range.

A completely different example are the inter-atomic forces that keep molecules together. Imagine some liquid water. This water is held together by intermolecular forces. Now if another small drop approaches the water-surface it will eventually be attracted

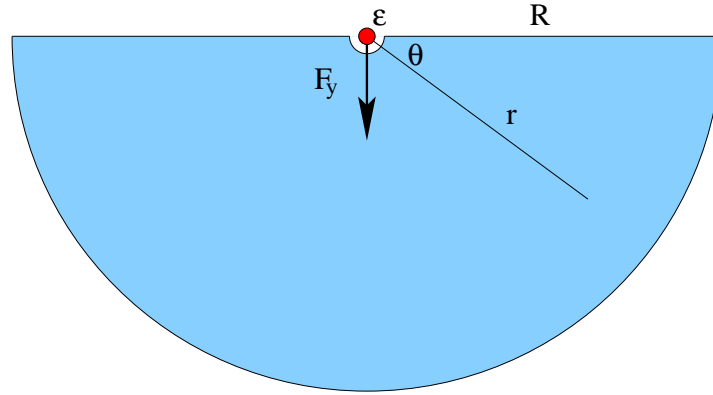


Figure 8.15: Red object is sample that experiences force F_y from the blue substrate. The force is long-ranged if the resulting force on the red object does not become independent of R for large R .

and merge with the water surface. However, the force that accomplishes this only depends on the *local* environment, and if you add much more water to the system this will have not effect on the drops affinity to the water surface. These forces are what we call short-ranged.

Mathematically we can distinguish between long and short range forces in the following way: let us consider a sample attracted by a large substrate, as shown in Figure 8.15. If the forces are long-range, like the gravitational forces we have been dealing with, then the force the red sample is experiencing will increase as the big blue object is increased in size.

However, many other forces don't act in that way. Imagine the red object being a bead that is about to interact with a mass of water. The bead will feel some force, once it is close enough to the water, but that force will not depend on how much water there is. So this raises the question: what distinguishes between long-ranged and short-ranged forces? There are some simple cases, e.g. a potential that has a finite range will ensure that there are no force contributions from any distance further away than that range. But such forces are nonphysical.

Most force follows a power-law (or at least can be written as a sum of power contributions). So let us assume that our particle, located at the origin interacts with particles from the large mass. This large mass consists of many small particles, but we can simplify this here and assume a continuum of particles, each interacting through a pair-potential that follows a simple power-law

$$F_{1,2} = -r_{1,2}^\alpha \frac{\mathbf{r}_{1,2}}{r_{1,2}^3} \quad (8.53)$$

Because the force can diverge for very small distances we will remove a bit of Material from around the red object up to a distance ϵ . So now the blue material stretches in a semi-sphere from the distance ϵ to the distance R . The question we want to answer

is: does the force on the red object converge to some constant, or does it continue to increase?

For this very simple geometry it is not hard to calculate the force on the object. For a two-dimensional object we get for the force

$$F_y(R) = \int_{r=\epsilon}^R dr \int_0^\pi r d\theta \mathbf{F}_{1,2} \cdot \hat{\mathbf{e}}_y \quad (8.54)$$

$$= \int_{r=\epsilon}^R dr \int_0^\pi r d\theta r^\alpha \quad (8.55)$$

$$= 2 \int_{r=\epsilon}^R dr (-r^{\alpha+1}) \quad (8.56)$$

$$= \frac{2}{\alpha+2} [-r^{\alpha+2}]_\epsilon^R \quad (8.57)$$

$$= \frac{2}{\alpha+2} (\epsilon^{\alpha+2} - R^{\alpha+2}) \quad (8.58)$$

Now if we consider the limit of large R we get

$$\lim_{R \rightarrow \infty} F_y(R) = \begin{cases} \frac{2\epsilon^{\alpha+2}}{\alpha+2} & \alpha < -2 \\ -\infty & \alpha \geq -2 \end{cases} \quad (8.59)$$

So for a gravitational potential of $\alpha = -2$ this does show that (for a two-dimensional situation) the gravitational potential leads to a long-range force.

More realistically, however we should consider the three dimensional system. In that case we can consider the equivalent system where the spherical cap now extends not only in two dimensions, but in three instead. Mathematically this simply means that we need to move from a polar coordinate system to a spherical coordinate system. This introduces a new angle ϕ . We then get

$$F_y(R) = \int_{r=\epsilon}^R dr \int_0^\pi r d\theta \int_0^{2\pi} r d\phi \mathbf{F}_{1,2} \cdot \hat{\mathbf{e}}_y \quad (8.60)$$

$$= \int_{r=\epsilon}^R dr \int_0^\pi r d\theta \int_0^{2\pi} r d\phi r^\alpha \quad (8.61)$$

$$= 4\pi \int_{r=\epsilon}^R dr (-r^{\alpha+2}) \quad (8.62)$$

$$= \frac{4\pi}{\alpha+3} [-r^{\alpha+3}]_\epsilon^R \quad (8.63)$$

$$= \frac{4\pi}{\alpha+3} (\epsilon^{\alpha+3} - R^{\alpha+3}) \quad (8.64)$$

Now if we consider the limit of large R we get

$$\lim_{R \rightarrow \infty} F_y(R) = \begin{cases} \frac{4\pi\epsilon^{\alpha+3}}{\alpha+3} & \alpha < -3 \\ -\infty & \alpha \geq -3 \end{cases} \quad (8.65)$$

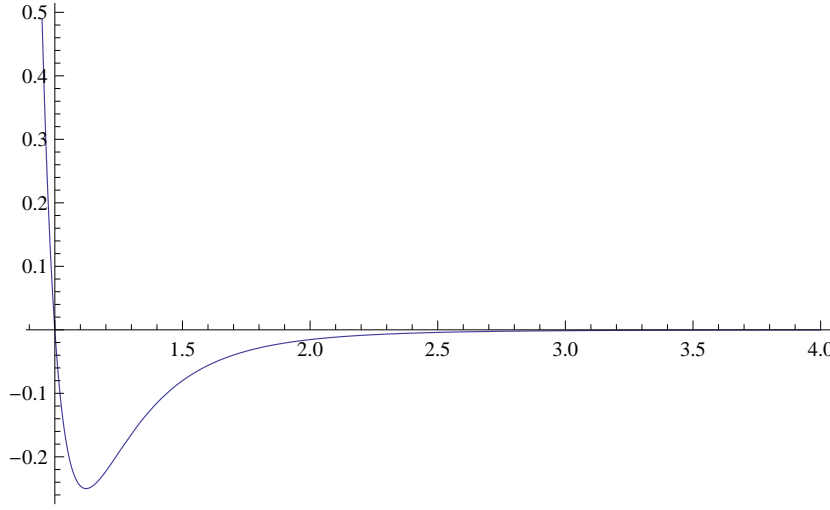


Figure 8.16: Leonard-Jones potential of (8.52) for $a = 1$ and $b = 0$.

So for a three dimensional system the forcing term is long-ranged if $\alpha > -3$. An example for such forces are dipol-forces that go with r^{-3} , and can be considered long-range in this case.

But interactions that go with r^{-4} or higher powers will be short-ranged in the sense defined above.

This leads us a famous toy-potential that people have long used to simulate inter-particle interactions: the Leonard-Jones potential. It is short ranged and given by

$$V(r) = \left(\frac{a}{r^{12}} - \frac{b}{r^6} \right) \quad (8.66)$$

This potential is similar to the potential we used for our planet parts, but the powers for the dependence on the distance is different.

Because this potential is short-ranged we often find that people use a slightly altered potential that is set to exactly zero at some reasonable cut-off radius. The advantage of this procedure is that you don't have to calculate the force if particles are further than this cut-off radius away from each other.

Implementing this in our code is trivial. We simply replace the interaction force with a force corresponding to the potential of equation (8.66)

$$\mathbf{F}(\mathbf{r}) = \left(-\frac{12a}{r^{14}} + \frac{6b}{r^8} \right) \mathbf{r} \quad (8.67)$$

An even easier situation arises when we consider non-aggregating particles, e.g. gas molecules of an ideal gas. Then we only use the repulsive part of the potential and we can observe the diffusion of particles.

When we are interested in bulk behavior of a system we can't possibly simulate all the particles and so we have to somehow contain them. One way to do this would be

to introduce walls into our system. When we do that, however, the system may show special behavior near the walls. This is good for many applications, particularly the simulation of nano-size objects, since we can then simulate them completely.

For larger systems, however, where we are interested of the behavior in the bulk, this is not such a good idea. If we contain the material in some sort of container we need to look only at the behavior a certain distance away from the wall in an effort to separate the wall behavior from the bulk behavior. One simple method to remove this difficulty is to use periodic boundary conditions. In this method we imagine that the large system we are intersted is covered with identical copies of our much smaller simulation. If the system is basically homogenous without large variations of its properties, the resulting system should basically look correct.

Let us look concretely at how one would implement this. Say our simulational system has a finite spatial extend and consists of a box that is spanned by the two corner points $(0, 0, 0)$ and (L_x, L_y, L_z) with edges that are parallel to the x, y, and z-axis. The particles can move freely inside this box and they interact according to Newton's laws. But when their path cross one of the faces of the box they are re-inserted into the box at the equivalent position at the opposing face of the box. So at all times the particles will remain inside the box. Mathematically this is achieved by projecting the positions inside the appropriate intervals:

$$x \rightarrow x \mod L_x \quad (8.68)$$

$$y \rightarrow y \mod L_y \quad (8.69)$$

$$z \rightarrow z \mod L_z \quad (8.70)$$

We also need to consider the interactions of the particles. The do not only interact with the particles in the box, but also with the images of the particles in the adjoining space. Formally, this should be taken to infinity, which would make the whole process completely unfeasible:

$$\mathbf{F}_i = \sum_{images} \sum_{j=1}^N F_{i,j}(\mathbf{x}_i, \mathbf{x}_j) \quad (8.71)$$

$$= \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} \sum_{j=1}^N F_{i,j}(\mathbf{x}_i, \mathbf{x}_j + (kL_x\mathbf{e}_x + lL_y\mathbf{e}_y + mL_z\mathbf{e}_z)) \quad (8.72)$$

Now since the interactions we are interested in are short-ranged, images that are removed from the center box can be neglected. (This would not be the case for gravitational intereactions).

Chapter 9

Basic Kinetic Theory and Statistical Mechanics

Now that we have a simple MD code that can simulate different phases of matter, it would be helpful if we could say something more quantitative about this. We will start by trying to understand the concept of temperature.

Firstly we will consider what the probability is that we will find that a particle has a particular velocity. What does $P(v_x = U_x)$ look like? what we can do is to look at the distribution of velocities that we find over some extended period of time. If we look at different times, we will find that the particles have some particular velocities. At some instance later these velocities will have changed. If we now look at all these velocities we will find that some velocities are more likely than others. How would we go about quantifying this?

A typical way of doing this is by using a velocity histogram. I.e. we just count the number of particles with velocity in a range of v_k and $v_k + \Delta v$ and plot this histogram. To do this we define a velocity range between v_{min} and v_{max} and the number of histogram bars that we would like to have N_H . Then at different times in our simulation we look at all the velocities and check where in the histogram the velocity falls and add one to the total number in that histogram bar. We can then do this again until we find a smooth distribution. There is some finetuning you may want to do determining appropriate values for the min and max values as well as the number of histogram bars.

Now we need to compare this with our expectation. We expect to find that the probability of finding a particular velocity is given by the Maxwell-Boltzmann distribution

$$P(\mathbf{v}) \propto \exp\left(-\frac{(\mathbf{v} - \langle \mathbf{v} \rangle)^2}{2kT}\right) \quad (9.1)$$

where the expectation value of the velocity $\langle \mathbf{v} \rangle$ is just a constant that does not change, because our algorithm conserves mass. The Temperature is given by the kinetic energy per particle:

$$D kT = \left\langle \frac{1}{2} m \mathbf{v}^2 \right\rangle \quad (9.2)$$

where D is the number of spatial dimensions.

In terms of the program we can approximate this quantity by

$$D kT \approx \sum_{i=1}^N \frac{1}{2} m_i \mathbf{v}_i^2 / N \quad (9.3)$$

This quantity will still fluctuate, since energy is exchanged between the potential and kinetic energy, so we get an even better approximation by continuing to average the temperature over different simulation times.

Now let us put this in the program:

Listing 9.1: MDnew.c

```

1 #include <math.h>
#include <time.h>
#include <stdlib.h>
#include <mygraph.h>
#define N 100
6
typedef struct obj {double x;double y; double vx; double vy;
    double m; double r1; double r2; int col1;int col2;} obj;
typedef struct vecd {double x; double y;} vecd;

double Lx=10,Ly=10, amp=0.1, vscale=1;
11 double Etot , Ekin , Epot , GlobalkT=0;

    obj at[N];

#define NH 1000
16 int Nhist=100, DoVhist=0, GlobalkTcount=0;
    vecd Vav;
typedef struct hist {double x; double count;} hist;
    hist vhist[NH], vhistN[NH], vhistTH[NH];
double vmin=-10, vmax=10;
21
void NormalizeHist(hist histin[NH], hist histout[NH], int Nhist){
    double sum=0;
    for (int i=0; i<Nhist; i++){
        sum+=histin[i].count;
26    }
    double fac=Nhist/(histin[Nhist-1].x - histin[0].x)/sum;
    for (int i=0; i<Nhist; i++){
        histout[i].x=histin[i].x;
        histout[i].count = fac*histin[i].count;

```

```

31     }
    }

    double GetkT( obj at [N] ) {
        double M=0,kT=0;
36     Vav.x=0;
        Vav.y=0;

        for (int n=0;n<N;n++){
41     M+=at [ n ].m;
        Vav.x+=at [ n ].m*at [ n ].vx;
        Vav.y+=at [ n ].m*at [ n ].vy;
        }
        Vav.x/=M;
46     Vav.y/=M;
        for (int n=0;n<N;n++){
            kT+=at [ n ].m*(pow( at [ n ].vx-Vav.x,2)+pow( at [ n ].vy-Vav.y,2) );
        }
        kT/=2*N; // number of dimensions
51     return kT;
    }

    void VHistTH( hist vhist [NH], int Nhist, double kT ) {
        for (int i=0;i<Nhist;i++){
56     vhist [ i ].x=vmin+(i+0.5)*(vmax-vmin)/(Nhist);
        vhist [ i ].count=exp(-at [ 0 ].m*pow( vhist [ i ].x-Vav.x,2)/(2*kT))
            ;
            // not very precise since the distribution depends on m for
            each atom
        }
        NormalizeHist( vhist , vhist , Nhist );
61 }

    void VHistInit( hist vhist [NH], int Nhist ) {
        for (int i=0;i<Nhist;i++){
            vhist [ i ].x=vmin+(i+0.5)*(vmax-vmin)/(Nhist);
66     vhist [ i ].count=0;
        }
        GlobalkT=GetkT( at );
        GlobalkTcount=1;
        VHistTH( vhistTH , Nhist , GlobalkT );
    }

```



```

        double r2= rx*rx+ry*ry;
        double r6= r2*r2*r2;
        double r12=r6*r6;
116      E+=1/r12-1/r6;
    }
  }
  return E;
}

121 void Force(obj at[N], vecd F[N]){
    for (int i=0; i<N; i++){F[i].x=F[i].y=0;}
    for (int i=0; i<N; i++)
        for (int j=i+1; j<N; j++){
126          for (int x=-1; x<2; x++)
              for (int y=-1; y<2; y++){
                  double rx=at[i].x-(at[j].x+x*Lx);
                  double ry=at[i].y-(at[j].y+y*Ly);
                  double r2= rx*rx+ry*ry;
131                  if (r2>16) continue;
                  double r8= r2*r2*r2*r2;
                  double r14=r8*r8/r2;
                  double tmp=12/r14-6/r8;
                  double Fx=tmp*rx;
136                  double Fy=tmp*ry;
                  F[i].x+=Fx;
                  F[i].y+=Fy;
                  F[j].x-=Fx;
                  F[j].y-=Fy;
141              }
          }
    }

    void Verlet(obj at[N], double dt){
146      vecd F[N];

      Force(at, F);
      for (int i=0; i<N; i++){
151        at[i].vx+=F[i].x/at[i].m*0.5*dt;
        at[i].vy+=F[i].y/at[i].m*0.5*dt;
        at[i].x +=at[i].vx*dt;
        if (at[i].x<0) at[i].x+=Lx;
        else if (at[i].x>=Lx) at[i].x-=Lx;

```

```

    at[i].y += at[i].vy*dt;
156    if (at[i].y < 0) at[i].y += Ly;
        else if (at[i].y >= Ly) at[i].y -= Ly;
    }
    Force(at, F);
    for (int i=0; i<N; i++){
161    at[i].vx += F[i].x/at[i].m*0.5*dt;
        at[i].vy += F[i].y/at[i].m*0.5*dt;
    }
}

166 void init(obj at[N]) {
    int N2=ceil(sqrt(N)), n=0;
    vecd CMv;
    double M;
    for (int i=0; (i<N2)&&(n<N); i++)
171    for (int j=0; (j<N2)&&(n<N); j++, n++){
        at[n].x = i*Lx/N2;
        at[n].y = j*Ly/N2;
        at[n].vx = amp*random()/RANDMAX;
        at[n].vy = amp*random()/RANDMAX;
176    at[n].m = 1;
        at[n].r1 = 0.5*1;
        at[n].r2 = 0.5*1.122;
        at[n].col1 = 2;
        at[n].col2 = 3;
181    }
    // now remove the center of mass motion
    M=CMv.x=CMv.y=0;
    for (int n=0; n<N; n++){
        M += at[n].m;
186    CMv.x += at[n].m*at[n].vx;
        CMv.y += at[n].m*at[n].vy;
    }
    CMv.x /= M;
    CMv.y /= M;
191    for (int n=0; n<N; n++){
        at[n].vx -= CMv.x;
        at[n].vy -= CMv.y;
    }
    Epot=PE(at);
196    Ekin=KE(at);

```

```

    Etot=Epot+Ekin;
}

void Init(){
201   init(at);
}

void vscalef(obj at[N]){
    for (int n=0;(n<N);n++){
206     at[n].vx*=vscale;
        at[n].vy*=vscale;
    }
}

211 void Vscale(){
    vscalef(at);
}

void TopView(int xdim, int ydim){
216   int xoffs,yoffs;
    double scale;
    // we want to center the display in the center of the window
    if (Lx/xdim>Ly/ydim){
        scale=xdim/Lx;
221     xoffs=0;
        yoffs=(ydim-Ly*scale)/2;
    }
    else {
        scale=ydim/Ly;
226     xoffs=(xdim-Lx*scale)/2;
        yoffs=0;
    }

    for (int i=0;i<N;i++){
231     myfilledcircle(at[i].col2 ,
                    xoffs+at[i].x*scale ,
                    ydim-(yoffs+at[i].y*scale) ,
                    at[i].r2 *scale);
        myfilledcircle(at[i].col1 ,
236     xoffs+at[i].x*scale ,
                    ydim-(yoffs+at[i].y*scale) ,
                    at[i].r1 *scale);
    }

```

```

    }
}
241 void Analysis() {
    Epot=PE(at);
    Ekin=KE(at);
    Etot=Epot+Ekin;
246    if (DoVhist){
        VHist(at, vhist, Nhist);
        NormalizeHist(vhist, vhistN, Nhist);
    }
}
251 int main() {
    int done=0, Repeat=1, cont=0, Slow=1000;
    double dt=0.01;

256    init(at);
    AddFreedraw("Top_view",&TopView);
    DefineGraphN_RxR("Vel_x_Hist",&vhist[0].x,&Nhist,NULL);
    SetDefaultColor(3);
    SetDefaultShape(1);
261    SetDefaultLineType(0);
    DefineGraphN_RxR("Vel_x_Hist_norm",&vhistN[0].x,&Nhist,NULL);
    SetDefaultColor(2);
    SetDefaultShape(0);
    SetDefaultLineType(1);
266    DefineGraphN_RxR("Vel_x_Hist_TH",&vhistTH[0].x,&Nhist,NULL);

    StartMenu("Sol",1);
    DefineGraph(freedraw_,"Views");
    DefineGraph(curve2d_,"Histogram_graph");
271    StartMenu("Energy",0);
    DefineDouble("E_pot",&Epot);
    DefineDouble("E_kin",&Ekin);
    DefineDouble("kT", &GlobalkT);
    DefineDouble("E_tot",&Etot);
276    EndMenu();
    StartMenu("Histogram",0);
    DefineDouble("Vmin",&vmin);
    DefineDouble("Vmax",&vmax);
    DefineMod("N_hist",&Nhist,NH);

```



```

281  DefineFunction("Start_Hist",&StartVHist);
    DefineFunction("Stop_Hist",&StopVHist);
    EndMenu();

    StartMenu("Init",0);
286  DefineDouble("Amp",&amp);
    DefineFunction("Init",&Init);
    EndMenu();
    DefineDouble("velScale",&vscale);
    DefineFunction("Vscale",&Vscale);
291  DefineDouble("Lx",&Lx);
    DefineDouble("Ly",&Ly);
    DefineDouble("dt",&dt);
    DefineBool("cont",&cont);
    DefineInt("Slow",&Slow);
296  DefineInt("Repeat",&Repeat);
    DefineBool("done",&done);
    EndMenu();
    while (!done){
        Events(1);
301  DrawGraphs();
        if (cont){
            for (int i=0; i<Repeat;i++){
                Verlet(at,dt);
                for (int j=0;j<Slow;j++) nanosleep((struct timespec [])
                    {{0, 100000}}, NULL);
306  }
            Analysis();
        }
        else nanosleep((struct timespec []) {{0, 1000000}}, NULL);
    }
311 }

```

The result is rather satisfactory, as is shown in Figure 9.1.

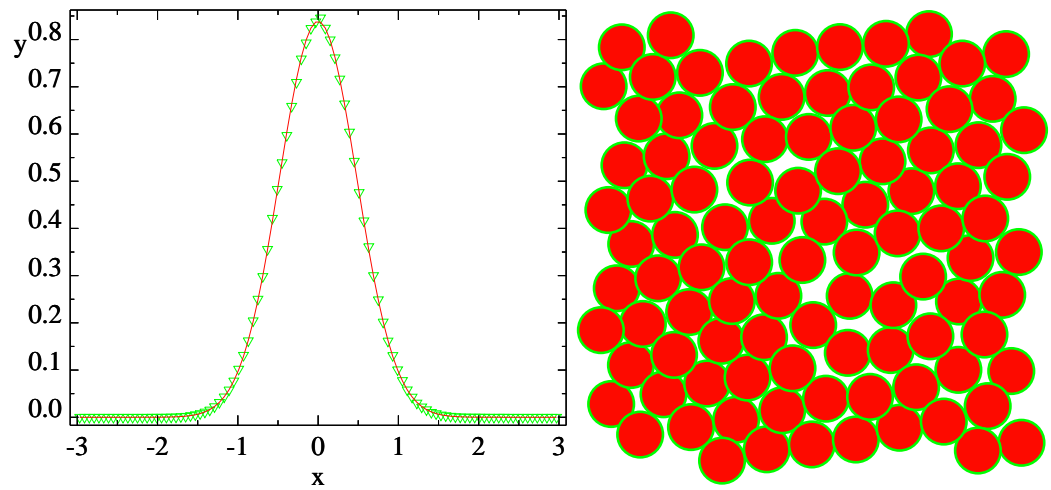


Figure 9.1: Velocity distribution function for the x velocity compared to the theoretical distribution function and a snapshot of the configuration.

Chapter 10

Monte Carlo

$$P(s) \propto \exp\left(-\frac{H(s)}{kT}\right) \quad (10.1)$$

where $H()$ is the Hamiltonian, i.e. a function that gives the total energy of the system. Then if we ask about the probability of finding the system in state s_1 or s_2 it is simply given by

$$\frac{P(s_1)}{P(s_2)} = \exp\left(-\frac{H(s_1) - H(s_2)}{kT}\right) \quad (10.2)$$

Now in equilibrium these probabilities are constant, and if we have an algorithm that has the following detailed balance condition for transition T between states, we can be assured that the probabilities are not changed by the transitions:

$$P(s_1)T_{1\leftarrow 2} = P(s_2)T_{2\leftarrow 1} \quad (10.3)$$

Since time does not matter here, we can multiply the transition rates by an arbitrary constant and that will not change the final probability distribution. Let us assume that $H(s_1) > H(s_2)$, so we have $P(s_1) > P(s_2)$. To get the most accepted transitions we choose

$$T_{2\leftarrow 1} = 1 \quad (10.4)$$

$$T_{1\leftarrow 2} = \frac{P(s_1)}{P(s_2)} = \exp\left(-\frac{\Delta E_{1,2}}{kT}\right) \quad (10.5)$$

which ensures that the detailed balance condition (10.3) is fulfilled.

Chapter 11

Lattice Gases

11.1 Hardy, de Pazzis, and Pomeau (HPP)

The first of the lattice Gases. It uses a square lattice and a simple collision rule: particles colliding head-on end up going in the orthogonal direction. That is all we need. These collisions conserve mass, momentum and (trivially) energy.

However, the model is not sufficiently symmetric to allow for the correct hydrodynamics. (we will see later why that is.

To practically implement this we need to encode a state. Since we can only have (in this model) one particle per lattice velocity we can encode the presence or absence of particles as single bits! This makes the code very memory efficient.

One example program where we have utilized this is given here:

Listing 11.1: LG1.c

```
/*  
  Lattice Gas according to , Hardy, Pomeau and de Pazzis (1973 and  
    1976)  
  */  
4  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
9 #include <mygraph.h>  
#include <math.h>  
  
/* The following should not be necessary , but some older  
   compilers don't support this yet */  
#define Ob0001 1  
14 #define Ob0010 2  
#define Ob0100 4
```

```

#define Ob1000 8
#define Ob0101 5
#define Ob1010 10
19 #define Ob1111 15
#define Ob1110 14
#define Ob1011 11

#define NV 4 // 4 velocities, square lattice, not isotropic.
24 #define LX 1000
#define LY 1000

int n[LX][LY];
/* Meaning of the state: each bit of the integer implies a
   particle at the corresponding velocity. east: 1 north: 10
   west:100 south:1000, (in binary)
29 So one particle travling north and one particle travelling
   south would be 1010=9
   */
double R=10;
/* now some fields to be displayed */
int rhoreq=0, ureq=0,coarse=1,lx=LX,ly=LY;
34 double rho[LX][LY],u[LX][LY][2]; // these could be integers,
   but the graphics is not yet set up for that.

/* For analysis */
double Uxav[LY];
int Uxavreq=0;
39

void iterate(){
static int nn[LX][LY];
/* colliding the particles */
44 for (int x=0;x<LX;++x)
    for (int y=0;y<LY;++y){
        switch (n[x][y]){
            case Ob0101: nn[x][y]= Ob1010; break;
            case Ob1010: nn[x][y]= Ob0101; break;
49         default: nn[x][y]=n[x][y];
        }
    }
/* now we need to move the particles */
for (int x=0;x<LX;++x)

```

```

54     for (int y=0;y<LY;++y){
        n[x][y]=
            (nn[(x+1)%LX][y]& Ob0100)
            +(nn[(x+LX-1)%LX][y]& Ob0001)
            +(nn[x][(y+1)%LY]& Ob1000)
59         +(nn[x][(y+LY-1)%LY]& Ob0010);
    }
}

64 void initcirc(){
    for (int x=0;x<LX;++x)
        for (int y=0;y<LY;++y){
            if (pow(x-LX/2,2)+pow(y-LY/2,2)<R)
                n[x][y]=0;
69         else
            n[x][y]=rand()& Ob1111;
        }
}

74 void initshear(){
    for (int x=0;x<LX;++x)
        for (int y=0;y<LY;++y){
            if (y<LY/2)
                n[x][y]=random()& Ob1011;
79         else
            n[x][y]=random()& Ob1110;
        }
}

84 void GetGraphs(){
    double *rp=&(rho[0][0]),*up=&(u[0][0][0]);
    lx=LX/coarse;
    ly=LY/coarse;
    if (rhoreq){
89     rhoreq=0;
    memset(rp,0,lx*ly*sizeof(double));
    for (int x=0; x<LX;++x)
        for (int y=0; y<LY; ++y)
            rp[x/coarse*ly+y/coarse]
94         +=(n[x][y]& Ob0001)
            +((n[x][y]& Ob0010)>>1)

```

```

        +((n[x][y]& Ob0100)>>2)
        +((n[x][y]& Ob1000)>>3);
    }
99    if (ureq){
        ureq=0;
        memset(up,0,lx*ly*2*sizeof(double));
        for (int x=0; x<LX;++x)
            for (int y=0; y<LY; ++y){
104        up[(x/coarse*ly+y/coarse)*2] +=
            (n[x][y]& Ob0001) - ((n[x][y]& Ob0100)>>2);
            up[(x/coarse*ly+y/coarse)*2+1] +=
            ((n[x][y]& Ob0010)>>1) - ((n[x][y]& Ob1000)>>3);
            }
109    }
    if (Uxavreq){
        Uxavreq=0;
        memset(Uxav,0,ly*sizeof(double));
        for (int x=0; x<LX;++x)
114        for (int y=0; y<LY; ++y){
            Uxav[y/coarse] +=
            (n[x][y]& Ob0001) - ((n[x][y]& Ob0100)>>2);
        }
        for (int y=0; y<ly; ++y){
119        Uxav[y]/=LX*coarse;
        }
    }
}

124 int main () {
    int Paused=1, Step=1, Repeat=1, done=0;

    initcirc();
    DefineGraphN_R("Ux_av",&Uxav[0],&ly,&Uxavreq);
129 DefineGraphNxN_R("Density",&(rho[0][0]),&lx,&ly,&rhoreq);
    DefineGraphNxN_RxR("velocity",&(u[0][0][0]),&lx,&ly,&ureq);

    StartMenu("Square_Lattice_Gas",1);
    DefineDouble("R^2",&R);
134 DefineFunction("init_circ",&initcirc);
    DefineFunction("init_shear",&initshear);
    DefineInt("Coarsegrain",&coarse);
    DefineGraph(contour2d_,"Density_plot");

```



```

DefineGraph ( curve2d_ , "Uav_graph" );
139 DefineInt ( " Repeat", &Repeat );
DefineBool ( " Step", &Step );
DefineBool ( " Paused", &Paused );
DefineBool ( " done", &done );
EndMenu ();
144 while ( !done ) {
    Events ( 1 );
    GetGraphs ();
    DrawGraphs ();
    if ( !Paused || !Step ) {
149     Step = 1;
        for ( int i = 0; i < Repeat; i++ ) {
            iterate ();
        }
    }
154 else sleep ( 1 );
}
}

```

11.2 Frisch, Hasslacher, and Pomeau (FHP)

The discovery of Frisch, Hasslacher, and Pomeau (or Stephen Wolfram, depending on whom you ask) was that one can recover a more symmetrical simulation, when one uses a more symmetric lattice. The simplest lattice that achieves this is the hexagonal lattice, presented in Figure 11.1.

Now we have more collisions that we can consider: there are still head-on collisions of two particles, but there are now two possible outcomes in two different lattice directions rotated by 60° and -60° , respectively. Additionally we consider a three particle collision, where three particles coming in with angles 0° , 120° , 240° will go out with angles 60° , 180° , 300° and vice versa.

For a practical implementation we will need to represent the hexagonal grid in C. We will use an array again, but that requires a square morphology. In figure 11.1 we show the sheared two dimensional lattice as black lines. In the square lattice the black lines now correspond to direct nearest-neighbor connections. However, the hexagonal lattice has two more nearest neighbors, shown as red lines, and these connections are represented as next-nearest neighbor connections in the square lattice.

We can adapt the code for the HPP lattice Gas to simulate the FHP lattice gas:

Listing 11.2: LG2.c

```

/*
Lattice Gas according to Frisch Hasslacher and Pomeau

```

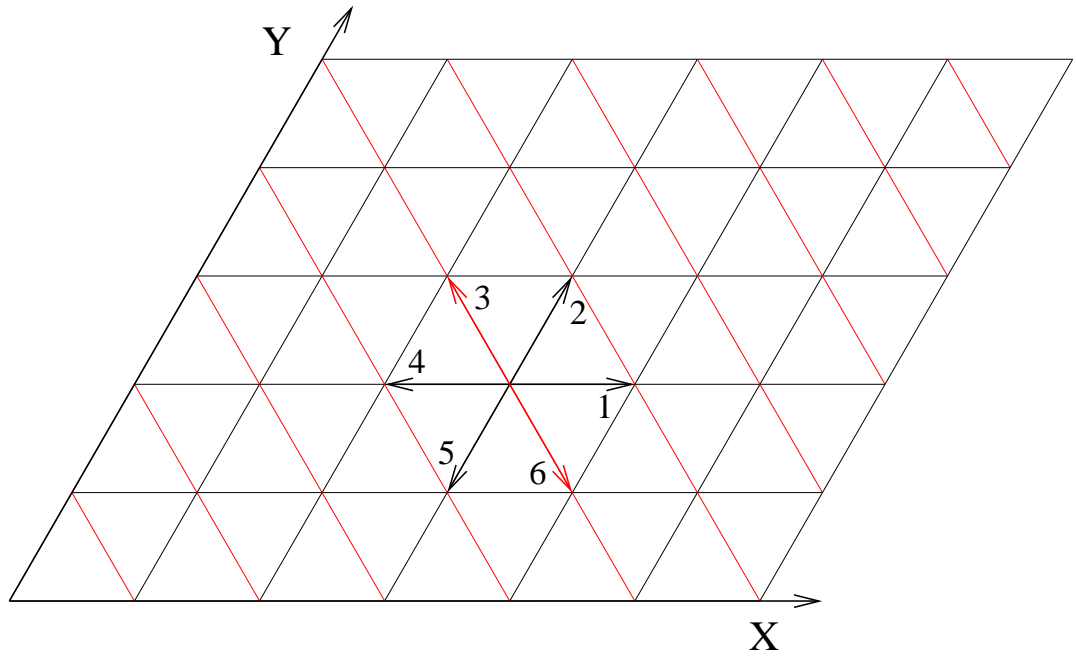


Figure 11.1: The hexagonal lattice of the FHP model.

```

4      */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
9  #include <mygraph.h>
#include <math.h>

/* The following should not be necessary, but some older
   compilers don't support this yet */
#define Ob000001 1
14 #define Ob000010 2
#define Ob000100 4
#define Ob001000 8
#define Ob010000 16
#define Ob100000 32
19 #define Ob001001 9
#define Ob010010 18
#define Ob100100 36
#define Ob010101 21
#define Ob101010 42

```

```

24 #define Ob111111 63

#define NV 6 // 6 velocities, hexagonal lattice.
#define LX 500
29 #define LY 500 // This represents a parallelogram

int n[LX][LY];
/* Meaning of the state: each bit of the integer implies a
   particle at the corresponding velocity. east: 1 north: 10
   west:100 south:1000, (in binary)
   So one particle travling north and one particle travelling
   south would be 1010=9
34 */
double R=10,frac=0.5;
/* now some fields to be displayed */
int rhoreq=0, ureq=0,coarse=1,lx=LX,ly=LY;
double rho[LX][LY],u[LX][LY][2]; // these could be integers,
   but the graphics is not yet set up for that.
39
/* For analysis */
double Uxav[LY];
int Uxavreq=0;

44 void iterate(){
   static int k=153123;
   static int nn[LX][LY];
   /* colliding the particles */
   for (int x=0;x<LX;++x)
49   for (int y=0;y<LY;++y){
       switch (n[x][y]){
           case Ob001001: nn[x][y]= ((++k&1)==0)? Ob100100: Ob010010
               ; break;
           case Ob010010: nn[x][y]= ((++k&1)==0)? Ob001001: Ob100100
               ; break;
           case Ob100100: nn[x][y]= ((++k&1)==0)? Ob010010: Ob001001
               ; break;
54   case Ob010101: nn[x][y]= Ob101010; break;
           case Ob101010: nn[x][y]= Ob010101; break;
           default: nn[x][y]=n[x][y];
       }
   }
}

```

```

59  /* now we need to move the particles */
    for (int x=0;x<LX;++x)
        for (int y=0;y<LY;++y) {
            n[x][y]=
                (nn[(x+1)%LX][y]& Ob001000)
64          |(nn[(x+LX-1)%LX][y]& Ob000001)
                |(nn[x][(y+1)%LY]& Ob010000)
                |(nn[x][(y+LY-1)%LY]& Ob000010)
                |(nn[(x+LX-1)%LX][(y+1)%LY]& Ob100000)
                |(nn[(x+1)%LX][(y+LY-1)%LY]& Ob000100);
69      }
    }

    void init() {
74      for (int x=0;x<LX;++x)
          for (int y=0;y<LY;++y) {
              if (pow((x+0.5*y)-(LX/2.+LY/4.),2)+pow(sqrt(0.75)*(y-LY
                  /2),2)<R)
                  n[x][y]=0;
              else {
79                  n[x][y]=((rand())<frac*RAND_MAX)?Ob000001:0)
                      |((rand())<frac*RAND_MAX)?Ob000010:0)
                      |((rand())<frac*RAND_MAX)?Ob000100:0)
                      |((rand())<frac*RAND_MAX)?Ob001000:0)
                      |((rand())<frac*RAND_MAX)?Ob010000:0)
84                  |((rand())<frac*RAND_MAX)?Ob100000:0);
              }
          }
    }

89 void initshear() {
    for (int x=0;x<LX;++x)
        for (int y=0;y<LY;++y) {
            if (y>LY/2)
                n[x][y]=((rand())<frac*RAND_MAX)?Ob000001:0)
94                |((rand())<0.25*frac*RAND_MAX)?Ob010010:0)
                |((rand())<0.25*frac*RAND_MAX)?Ob100100:0)
                /*|((rand())<frac*RAND_MAX)?Ob001000:0)*/
                |((rand())<0.25*frac*RAND_MAX)?Ob010010:0)
                |((rand())<0.25*frac*RAND_MAX)?Ob100100:0);
99        else

```

```

n[x][y]=/* (rand()<frac*RAND_MAX)?Ob000001:0)
          |*/((rand()<0.25*frac*RAND_MAX)?Ob010010:0)
          |((rand()<0.25*frac*RAND_MAX)?Ob100100:0)
          |((rand()<frac*RAND_MAX)?Ob001000:0)
104      |((rand()<0.25*frac*RAND_MAX)?Ob010010:0)
          |((rand()<0.25*frac*RAND_MAX)?Ob100100:0);

    }
109 }

void GetGraphs() {
    int x,y;
    double *rp=&(rho[0][0]),*up=&(u[0][0][0]);
114    lx=LX/coarse;
    ly=LY/coarse;
    if (rhoreq){
        rhoreq=0;
        memset(rp,0,lx*ly*sizeof(double));
119        for (x=0; x<LX;++x)
            for (y=0; y<LY; ++y)
                rp[x/coarse*ly+y/coarse]
                    +=(n[x][y]& Ob000001)
                    +((n[x][y]& Ob000010)>>1)
124                +((n[x][y]& Ob000100)>>2)
                    +((n[x][y]& Ob001000)>>3)
                    +((n[x][y]& Ob010000)>>4)
                    +((n[x][y]& Ob100000)>>5);
    }
129    if (ureq){
        ureq=0;
        memset(up,0,lx*ly*2*sizeof(double));
        for (x=0; x<LX;++x)
            for (y=0; y<LY; ++y){
134                up[(x/coarse*ly+y/coarse)*2] +=
                    (n[x][y]& Ob000001)
                    +0.5*(((n[x][y]& Ob000010)>>1)-((n[x][y]& Ob000100)
                        >>2))
                    -((n[x][y]& Ob001000)>>3)
                    -0.5*(((n[x][y]& Ob010000)>>4)-((n[x][y]& Ob100000)
                        >>5));
139                up[(x/coarse*ly+y/coarse)*2+1] += sqrt(0.75)*

```

```

        (((n[x][y]& Ob000010)>>1)+((n[x][y]& Ob000100)>>2)
        -((n[x][y]& Ob010000)>>4)-((n[x][y]& Ob100000)>>5));
    }
}
144  if (Uxavreq){
        memset(Uxav,0,ly*sizeof(double));
        for (y=0; y<LY; ++y){
            for (x=0; x<LX; ++x)
                Uxav[y/coarse]+=
149          (n[x][y]& Ob000001)
            +0.5*(((n[x][y]& Ob000010)>>1)-((n[x][y]& Ob000100)
                >>2))
            -((n[x][y]& Ob001000)>>3)
            -0.5*(((n[x][y]& Ob010000)>>4)-((n[x][y]& Ob100000)
                >>5));
        }
154    for (y=0;y<ly;y++){
            Uxav[y]/=LX*coarse;
        }
    }
}
159  int main () {
        int Paused=1, Step=1, Repeat=1, done=0;

        init();
164  DefineGraphN_R("UxLav",&Uxav[0],&ly,&Uxavreq);
        DefineGraphNxN_R("Density",&(rho[0][0]),&lx,&ly,&rhoreq);
        DefineGraphNxN_RxR("velocity",&(u[0][0][0]),&lx,&ly,&ureq);

        StartMenu("Hexagonal_Lattice_Gas",1);
169  DefineDouble("R^2",&R);
        DefineDouble("frac",&frac);
        DefineFunction("init_circ",&init);
        DefineFunction("init_shear",&init_shear);
        DefineInt("Coarsegrain",&coarse);
174  DefineGraph(contour2d_,"Density_plot");
        DefineGraph(curve2d_,"Uav_graph");
        DefineInt("Repeat",&Repeat);
        DefineBool("Step",&Step);
        DefineBool("Paused",&Paused);
179  DefineBool("done",&done);

```

```

EndMenu();
while (!done){
    Events(1);
    GetGraphs();
184    DrawGraphs();
    if (!Paused || !Step){
        Step=1;
        for (int i=0;i<Repeat;i++){
189            iterate();
        }
    }
    else sleep(1);
}

```

To represent this graphically I include another option in the GUI to represent an array as a trapezoidal part of a hexagonal lattice.

Chapter 12

The Boltzmann equation

While we discussed the different lattice gases, Tyler pointed out that the simple pictures we got do not really tell you why the FHP lattice gas is supposed to be so superior over the HHP lattice gas. Sure, in the simulations of the fluid filling in a void we saw a hexagonal structure rather than a cubic structure, but that is not enough evidence to show that the FHP lattice gas really represents a significant advantage.

To understand the reasoning behind this we need to consider the underlying theory of kinetic theory, which will take us all of today's lecture. This is a somewhat involved mathematics, maybe more than you are used to, but it will help us understand the difference between these two lattice gases, as well as the method which as now all but replaced lattice gases: lattice Boltzmann.

We can write the evolution equation for the lattice gas for a general velocity set $\{\mathbf{v}_i\}$ as

$$n_i(\mathbf{x} + \mathbf{v}_i, t+1) = n_i(\mathbf{x}, t) + \sum_{jkl} P_{i,j \rightarrow kl} (n_k n_l - n_i n_j) + \sum_{jklmn} P_{ijk \rightarrow lmn} (n_l n_m n_n - n_i n_j n_k) + \dots \quad (12.1)$$

where $n_i(\mathbf{x}, t)$ is the number of particles moving with velocity v_i at position \mathbf{x} at time t . The effect of the collision is given by first two-particle collisions, then three particle collisions and maybe more. Now we want to examine the macroscopic behavior of the fluid we are simulating. To do that we now define a distribution function f_i , which represents the behavior of a large number of molecules. We can imagine that we look at a coarse-grained system where we look at many particles. In this case, the local fluctuations become unimportant and we are left with a deterministic behavior:

$$f_i(\mathbf{x} + \mathbf{v}_i, t + 1) = f_i(\mathbf{x}, t) + \Omega_i \quad (12.2)$$

where Ω_i is the collision operator.

To understand what the collision operator does, let us consider what the distribution looks like in equilibrium. For a lattice gas model, the solution will widely fluctuate, but for a coarse-grained approach the result will be a well-defined distribution. We will call this the equilibrium distribution. We can determine the equilibrium distribution by

the condition that it does not get altered by the collisions. So what is this equilibrium distribution? Before we can answer this question, we need to consider the important effect that the conserved quantities of mass, momentum and energy (or temperature) cannot be altered by the collisions, so the distribution is expected to depend on these variables. So we are looking for $f_i^0(\rho, \mathbf{u}, \theta)$.

$$f_i^0(\rho, \mathbf{u}, \theta) = \quad (12.3)$$

Now the effect of this collision operator will be to bring the distributions f_i closer to their local equilibrium values f_i^0 , which are consistent with the conserved quantities mass, momentum and (sometimes) energy.

This suggests that we can approximate the collision operator as

$$\Omega_i = \frac{1}{\tau}(f_i^0 - f_i) \quad (12.4)$$

12.1 Multi-phase flow

If the system is not an ideal system, it will have a pressure that is different from the ideal gas pressure. Such systems are described by a Van-der-Waals equation of state:

$$p(\rho) = p_0 \left(\frac{\rho}{3 - \rho} - \frac{9}{8} \rho^2 \theta_c \right) \quad (12.5)$$

We can incorporate such a pressure into the Navier-Stokes equation:

$$\partial_t(\rho \mathbf{u}) + \nabla(\rho \mathbf{u} \mathbf{u}) = -\nabla p + \nabla(\eta(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)) \quad (12.6)$$

We currently have for a pressure term $-\nabla(\rho \theta)$. We can compensate this with a forcing term of the form:

$$F = -\nabla(p - \rho \theta) \quad (12.7)$$

12.2 Including a passive scalar

A passive scalar is a quantity that is simply advected with the fluid flow. This could represent the movement of some tracer particles. Such a second quantity can be encoded by its own lattice Boltzmann equation with some densities g_i . For this density we will have

$$T = \sum_i g_i \quad (12.8)$$

and it will obey the lattice Boltzmann equation

$$g_i(x + v_i, t + 1) = g_i(x, t) \frac{1}{\tau_2} (f_i^0(T, u) - g_i(x, t)). \quad (12.9)$$

The remaining question is what the resulting hydrodynamic equation for this passive tracer density $T(x, t)$ is going to be. Just as in the case of the hydrodynamic equations we will Taylor expand the densities in (12.9)

$$(\partial_t + v_{i\alpha}\partial_\alpha)g_i + \frac{1}{2}(\partial_t + v_{i\alpha}\partial_\alpha)^2g_i + O(\partial^3) = \frac{1}{\tau_2}(f_i^0(T, u) - g_i). \quad (12.10)$$

From this we again get

$$g_i = f_i^0(T, u) - \tau(\partial_t + v_{i\alpha}\partial_\alpha)g_i + O(\partial^2) \quad (12.11)$$

which we substitute into (12.10) to get

$$(\partial_t + v_{i\alpha}\partial_\alpha)g_i^0 - (\tau - \frac{1}{2})(\partial_t + v_{i\alpha}\partial_\alpha)^2g_i^0 + O(\partial^3) = \frac{1}{\tau_2}(f_i^0(T, u) - g_i). \quad (12.12)$$

and then sum this to get

$$\partial_t T + \partial_\alpha(Tu_\alpha) - (\tau - \frac{1}{2})[\partial_t(\partial_t T + \partial_\alpha(Tu_\alpha)) + \partial_\beta(\partial_t(Tu_\beta) + \partial_\alpha(Tu_\alpha u_\beta + T\theta\delta_{\alpha\beta}))] = 0 \quad (12.13)$$

This simplifies to (remembering that $\partial_t u_\alpha + u_\beta\partial_\beta u_\alpha = (1/\rho)\partial_\alpha(\rho\theta) + O(\partial^2)$)

$$\partial_t T + \partial_\alpha(Tu_\alpha) = (\tau_2 - \frac{1}{2})[\theta\partial_\alpha\partial_\alpha T + T\partial_\alpha\frac{1}{\rho}\partial_\alpha(\rho\theta)] \quad (12.14)$$

This we can write as

$$\partial_t T + \nabla(T\mathbf{u}) = (\tau_2 - \frac{1}{2})\theta[\nabla^2 T + T\nabla^2 \ln(\rho)]. \quad (12.15)$$

In most cases the density is assumed to be nearly constant and the last term can be neglected.

12.3 Including temperature as a passive scalar

In principle there is no need to do anything special to include the temperature in the method, since this would simply follow from energy conservation. However, to include the temperature we would need to match the fourth order moments of the equilibrium distribution, and this requires a larger velocity set. There have been some recent efforts in that direction, but we will stick there to another approach, which can also be used for

Appendix A

Programming Exercises

Problems

- 1.1:** Write a C-program that writes out the numbers 1 to 10 to the terminal.
- 1.2:** Write a C-program that writes out the values of $\sin(x/10)$ for x from 0 to 100.
- 1.3:** Write a C-program that writes the values of $\sin(x/10)$ for x from 0 to 100 into an array and then use the graphics library to display this array graphically.
HINT: You can use the subroutine `DefineGraphN_R` to define a one-dimensional array, and use the `DefineGraph` routine with the parameter `curve2d_` to get a menu button to view graphs of that type.
- 1.4:** Write a C-program that writes the values of $\sin(ax)$ for x from 0 to 100 and a variable a into an array and then use the graphics library to display this array graphically. Use the GUI to be able to interactively change the value of a .
- 1.5:** Now write a C-program that fills a double array with the x component of $\sin(ax)$ and the y component with $\cos(bx)$ for values of x from 0 to 99. Include the values of a and b in the GUI and plot the resulting two-dimensional array for various values of a and b . What do these figures look like? What are “reasonable” values of a and b ?
- 1.6:** Now plot the path

$$\begin{pmatrix} \sin(ax + c) \\ \cos(bx + c) \end{pmatrix} \quad (\text{A.1})$$

instead, include the parameter c in the GUI and plot the result. What does the parameter c do to the graph? Can you give an analytical expression of the effect of the parameter?

1.7: Now plot instead the path

$$\begin{pmatrix} \sin(ax + c) \\ \sin(bx + c) \end{pmatrix} \quad (\text{A.2})$$

for different values of a and b . How do these graphs differ from the previous graphs?

1.8: Now plot instead the path

$$\begin{pmatrix} \sin(ax + c) \\ \sin(bx) \end{pmatrix}. \quad (\text{A.3})$$

How do these graphs differ? What is the effect of changing c now?

1.9: Now you want to allow the graph to be continuously updated. You could allow the parameters a , b , and c to be changing in time. Write a program that shows how the graph changes as you (slowly) alter the parameters.

HINT: You could change have an iteration routine that changes the parameters for you and then you can observe the change. You may also want to experiment with the size of your data array to get the most pleasing results.

1.10: Not all data are created equal. Imagine you want to consider the general behavior of a system like the three particle scattering as the function of two different parameters (maybe the initial x position and the initial y velocity). Now you calculate how far apart the particles will be after a certain time. The resulting data will be a two dimensional array of distances. To represent this data, you will have to define a new data type: `DefineGraphNxN.R()`. Use this routine to plot data of the type

$$f(x, y) = \exp(-a(x^2 + y^2)) \sin(b(x^2 + y^2)) \quad (\text{A.4})$$

for different values of a and b . To view data of this type you will need to include a menu item of the form `DefineGraph(contour2d_, 'name')`.

A.1 Graph library tips

To have different graphs displayed in different windows you can use:

```
extern void NewGraph();
extern void SetActiveGraph(int);
```

The first of these commands is used between different `DefineGraph` statements, and the second one is used before the

```
DefineGraph();
```

statements in the GUI.

A.2 The Pair correlation function

A quantity of interest is the pair correlation function $g(r_1, r_2)$, i.e. the probability of finding a particle at position r_2 , given that there is already a particle at position r_1 . In translationally and rotationally invariant systems this quantity will only depend on the total distance $r = |r_1 - r_2|$. We then have the radial distribution function $g(r)$.

This quantity is very important, since it is related (through a Fourier transformation) to the Structure factor $S(q)$, which can be experimentally obtained by X-ray scattering. Here, however, we will focus on how to obtain this radial distribution function in our simulations.

Formally this is straight forward. First let us define a density

$$\rho(r) = \sum_i \delta(r - r_i) \quad (\text{A.5})$$

where r_i is the position of the i th atom. We then define the pair correlation function as

$$g(r) = \int dr_1 dr_2 \rho(r_1) \rho(r_2) \delta((r_1 - r_2) - r) \quad (\text{A.6})$$

This would be a pair correlation function for a specific configuration, and it would be a set of delta functions.

But what we are really interested in is a generic state of the system, not a particular realization. A different way of thinking about this is that we are looking at an ensemble of equivalent systems, and we look at the generic structure, which will be a continuous function, not a set of delta functions:

$$g(r) = \left\langle \int dr_1 dr_2 \rho(r_1) \rho(r_2) \delta((r_1 - r_2) - r) \right\rangle \quad (\text{A.7})$$

where I have not yet defined what I mean by this ensemble average $\langle \dots \rangle$. What we mean by this ensemble is a selection of all systems that correspond to the same “macroscopic” state. The way to think about this is that if I tell you that you have a jar of a given volume full of a gas of a certain density in equilibrium, there are many particle configurations consistent with that information. All of these consistent configurations together make up the “ensemble”.

To practically extract a radial distribution function from a simulation we have another route we can take. What we are after is the distribution of those delta functions, so we will want to “smear out” the result, or put it in some kind of histogram, so that we obtain a continuous looking solution.

$$g(r) = \sum_i \sum_j \theta[|(r_1 - r_2) - r| - \Delta r] \quad (\text{A.8})$$

Practically we might achieve this by defining a $g(r)$ field in our code with dimension $\dim_x = L_x / \Delta x$ etc. We would then calculate an instantaneous pair correlation function through a bit of code like this: