Physics 5621 Computational Physics Lecture 1

T. Blum

November 28, 2018

Contents

1 Goals of the course

- Develop basic programming skills using C/C++ and Python
- Basics of Unix shell environment (sh, csh, bash, ...)
- Basics of gnu compilers (gcc, g++),
- Basics of code management using git
- latex, gnuplot, ...
- Attain proficiency in numerical solution of physics (other) problems, including
  - range of numerical algorithms/techniques
  - numerical errors (roundoff, discretization, ...) and stability
  - basics high performance computing using MPI, threading, simd, BLAS, ...

2 Resources

Class notes based on many sources including

- Homer Reid's lecture notes
- Richard Fitzpatrick's lecture notes
- Branislav Nikolic's lecture notes

- Kernighan and Ritchie's book The C Programming Language
- Bjarne Stroustup's book on C++
- Numerical Recipes
- Yousef Saad's book (iterative methods)
- Computational Fluid Dynamics and Heat Transfer (Anderson, Tannehill, and Pletcher)
- Various journal articles
- the Internet

3 Introduction to C programming

Why C/C++? C is the most widely used programming language. Many other languages and operating systems are written in it, notably the Unix operating system. By writing your own codes in C you will have a deeper understanding numerical computations and eventually the ability to write high performance, efficient code.

C++ is an object oriented extension of C that is very powerful. We won't need all of the bells and whistles, but will avail ourselves of some of the important features (for example standard templates for complex arithmetic).

The following sub-sections closely follow the notes by Fitzpatrick which provide a very nice introduction to C. See also the book by Kernighan and Ritchie.

3.1 Data types in C

- characters

- integers
- floating point numbers
- symbolic constants

Declarations are used in C to define variables of a given type. For example,

int a;

a = 16;

Typically an int is four "bytes". A byte is 8 "bits". A bit is a binary number, either 0 or 1. So a 4 byte int can hold a *signed* integer with a maximum value of $2^{31} - 1 = 2\,147\,483\,647$. The "-1" is a bit subtle: the first integer bit is 2^0, so the last is 2^{30}, and $2^{31} - 1 =$ 1111111 11111111 11111111 11111111

There are also "unsigned", "long" and "long long" int's. The difference is compiler and/or machine dependent. In 32 and 64 bit architectures int and long int are the same.

float b;

b = 4.13;

double d = $1.056e{-}13$ (declaration and initialization in same statement, using scientific notation)

Floating point works more or less the same, except we have upto 8 bytes (64 bits) for double. A floating point number is just what the name implies: the decimal point "floats" as a opposed to a "fixed" point number like 5.00. Typically 52 bits are used for the mantissa (significand), 11 bits for the exponent and 1 sign bit.

The sign in the exponent is determined by subtracting a *bias*, so for double precision the exponent takes values between -1022 and +1023. The 11 bit width allows for a range of numbers between $2^{\pm 1023} \approx 10^{\pm 308}$. If you try use a number bigger or smaller, you will get an *overflow* or *underflow* error.

The 52(+1) bit mantissa gives an absolute precision of about 16 decimal digits, *i.e.*, $2^{-53} \approx 1.11\text{e} - 16$.

We will discuss exactly representable and approximate floating point numbers (integers) when we discuss roundoff error.

char c;

c = 'j'; or c="8"; // and so on

Characters are 1 byte and are typically used to make ascii (or unicode) strings. Depending on the set used, the digits 0-255 correspond to a letter (either upper or lower case, numbers (0-9), or symbols like #%&!ˆ=. There are also special "escape" characters, or sequences which contain a leading "\". \n is an important one (representing the newline character).

Declarations are not limited in length:

double my_data, your_data, everyones_data, ...;

~~But in C they must all appear before the first *executable* statement~~. This will change in C++. Declarations can now appear anywhere in C too.

3.2 variables and names

In the preceding section we used variable names in our declarations. For example "int a;" declared a variable "a" which can be used in our program to hold the value of an integer. Valid variable names can be quite general, like x, y2, pressure, size, name, lattice_constant, latticeConstant, and so on. They are case-sensitive. It's good practice to stay within 31 characters. Special key words like double, void, main, ... can not be used.

3.3 symbolic constants

Sometimes it is useful to define a constant for a numerical value or other text. For example,

#define pi 3.141592653589793

...

mom = 2 * pi / L;

The define statement tells the C compiler to literally replace all instances of "pi" with the text "3.141592653589793" *before* compiling. "#define" is an example of a compiler directive which is handled by the C-preprocessor. The "#" signals that is not a regular statement. We'll see other examples later.

3.4 expressions, statements and operators in C

A program in C is built with a series of *statements* constructed from *expressions* and *operators*. An expression combines data and operators and itself represents a datum, usually a number. For example

x+y (addition operator. also -, *, /)

z = x+y (assignment operator)

u > me (greater than relational operator (boolean datum), also <, >=, <=, and ! =)

u && I (logical "and", boolean datum, either true (1) if both u and i true or false (0) if both or either is false)

me || u (logical "or", true if either me or u is true)

Expressions are still abstract in the sense that the computer (CPU or GPU) hasn't done anything yet. A *statement* tells the computer to do something and is signaled (executed) by a ";" at the end. A statement can extend over any number

of lines but always ends with a ";".

z = x+y; (assign z the value equal to the sum of x and y)

x+y; (not valid since there is no assignment or control instruction)

There are three types of statements in C: expression statements (above), compound statements, and control statements. Compound statements are groups of statements enclosed within braces '{}', like

```
{
    float re = x;
    float im = y;
    float mag_sq = (x*x + y*y);
}
```

These often appear as *struct's* in C which are user defined. When we get to C++ they could also be part of a *class object*.

Control statements control the flow of execution of the program. There are many kinds: "if", "do", "while", "for", and so on. We'll talk more later about each of these.

Many kinds of *operators* can appear in expressions and statements:

- Arithmetic: + - * / % (mod or remainder)
- assignment: =
- (compound) assignment: += -= *= /=
- cast (treat a data type as a different type)
- logical: && (and) || (or)

- relational: < (less than) > (greater than) ≤ (less than or eq) ≥ (greater than or eq)

- unary: - (minus) ++ (increment) –(decrement) *(dereference) &(address) sizeof() ! (not). Unary operators operate on a single expression (datum)

In an expression all variables should (must) be the same type:

float b, m, x ,y;

y = m*x+b;

int i=2; int j=3; float y; float x=2; (long declarations are not easily readable)

y = j / i;

would evaluate to y = 1.0 (since y is a float) but

y = (float) j / (float) i; gives 1.5. This is an example of a *cast* operation, or cast for short. If you write y = i/j; this will evaluate to 0! Integer division is defined by discarding the remainder (sometimes call the floor function)

y= x/j; will evaluate to 0.6666667. In other words the compiler is smart enough to convert int to float, but for more complicated expressions this is not usually the case (especially when we define our own data types or objects). This is called *type promotion* or automatic conversion. The "lower" data type is converted to the "upper" data type before the binary operation. So, int to long int, int to float, float to double, and so on. In complicated expressions you can get unexpected (wrong) results that are hard to track down. You can waste hours tracking down such a "bug" so it's best to cast explicitly!

Operators also have a *precedence* when evaluated in an expression. The highest is evaluated first, lowest last.

Their *associativity* indicates in what order operators of equal precedence in an expression are applied. The following table lists operators and their precedence and

associativity. Associativity denotes the order of evaluation of operators with equal precedence in a single expression.

operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ - -	Postfix increment/decrement (see Note 2)	
++ - -	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Note 1: Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2: Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement y = x * z++; the current value of z is used to evaluate the expression (i.e., z++ evaluates to z) and z only incremented after all else is done.

4 A simple C program

In this section we'll see how to write a simple program, including the "main()" section, a simple function, control statements, and simple i/o. We'll also introduce version control using git and a powerful plotting program called gnuplot. You'll need to choose a text editor. I use emacs and vi(m) which are supported on all (most?) platforms running Unix (linux, MacOS, etc), and even Windows.

To compile and run our program I will operate in a standard Unix operating system environment using open source compilers and tools (gnu). To follow me you will need to be familiar with a simple terminal interface and basic Unix commands. These are all available on department computers. For git we will use the gitlab set up on the Physics Department server astral.

4.1 source

Here is the source listing from a simple program that demonstrates cumulative round-off error by adding number to itself many times. It should plant the seed to always be

thinking about numerics and errors, and it serves to introduce many basic elements of a generic C program. Let's go through it line by line.

```
/*
   simple C program to demonstrate numerical round off error
   See http://homerreid.com/teaching/18.330/Notes/MachineArithmetic.pdf
   for a discussion
*/

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

#define PI 3.141593

int main(int argc, char *argv[]){

  int i;
  int N;
  float X;
  float sum;

  /* divide pi into 1000 bits */
  N=1000;
  X = PI/(float)N;
  /* add them up again */
  sum=0.0;
  for(i=0;i<N;i++){
    sum += X;
  }
  X = fabs(sum-PI)/PI;

  printf("Error on PI(N=%d) = %e \n",N,X);

  return 0;
}
```

As we get more experience we will quickly see that our program can get complicated with many lines of code. To manage the complexity it is common to break up

the program into *functions* (or subroutines) that are called from the main program (and other functions). For example we can move the for loop in main to its own function and try many values of N by using *nested* for loops, or by moving the inner for loop to its own function. The program then looks like

```
/*
   simple C program to demonstrate numerical round off error
   See http://homerreid.com/teaching/18.330/Notes/MachineArithmetic.pdf
   for a discussion
*/

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

#define PI 3.141593

/* function to add a number to itself N times */
float directsum(int N, float X){

  int i;
  float sum=0.0;

  for(i=0;i<N;i++){
    sum += X;
  }
  return sum;
}

int main(int argc, char *argv[]){

  int N;
  float X;

  for(N=0;N<1000;N+=100){

    if(N){
      X = directsum(N,PI/(float)N);
```

```
      X = fabsf(X-PI)/PI;
      printf("X(N)= %d %e \n",N,X);
    }
  }

  return 0;
}
```

Notice that the function "directsum" is defined before it is called in main. If it was defined afterwards, we would need a *function declaration* before main so the compiler knows about it before it is called. If you have many functions to be declared, they can be collected in a *header* file with a ".h" extension, and "#include" ed at the beginning.

There are many "library" functions that you can (will) use without having to write your own. For example we used "fabsf" already. It is defined in the header file "math.h" which is so useful and so common it is always found in the same place: /usr/include/math.h (more later). You can also find cos, sin, log, exp, etc. There is no exponent function for expressions like $3.0^{4.4}$. You must use

$$\text{pow}(3.0, 4.4).$$

Also notice the stdio.h file which contains the "printf" function.

Further improvements are possible. For example, we may tire of changing the value of "N" by hand and recompiling by hand. We can instead use a command line argument to read any value at run time and only compile our code once! This is done using argc and argv. The important parts are

```
int main(int argc, char *argv[]){

  int N;
```

```
float X;
int Nmax;
int Ninc;

if(argc != 3){
  printf("Wrong # of args: Nmax Ninc\n");
  exit(-1);
}

Nmax = atoi(argv[1]);
Ninc = atoi(argv[2]);
```

Here the command line arguments are parsed and read in to the character array *argv[] (actually a pointer to a pointer to char). They should be separated by white space. Note argv[0] is always the name of the executable so argc = 1+number of args.

We have to convert the character strings read into argv. There is also a conversion function atof() for floats (doubles). We have been kind to the user by checking if the number of command line arguments provided is correct, and if not, printing them to the *stdout* (console usually the screen) and exiting with a value "-1". the exit function is in stdlib.h. There are also i/o streams called stdin and stderr. You should think of streams just like you would any file. We'll talk more about them when we get to i/o.

4.2 compiling and running the simple program

To the compile the program we type in a the following command in our terminal window:

gcc -o simple simple.c (followed by carriage return)

Which compiles the source simple.c and creates and executable (object) file "sim-

ple". We can run the program by simply typing

simple

Later we may need to "link" our program to an already compiled library (produce a combined executable). For example,

gcc -o prog prog.c -L/usr/lib -lblas

or

gcc -o prog prog.c -lblas

compiles our program "prog.c" and links the resulting object file to existing basic linear algrebra subroutines (BLAS).

In the 2nd example we assumed that the path /usr/lib is "known" to the system (see environment variable LD_LIBRARY_PATH).

The syntax is the following. The "-L" string tells the linker where to look for the desired library (path) and the "-l" gives the library name, assuming the convention lib"name".a. In our example the full path+name would be

/usr/lib/liblas.a

Libraries are "archive" files, or a collection of object files (.o) which have been compiled already, using a compiler that is compatible with the one we are using. These ".a" files are known as static libraries. The main advantage of static libraries is speed: everything is linked together at compile time. When the program is run, there is no need for the system to find the desired function because a copy of the library function is included at compile time. However, if a change is made to a library function your code has to be recompiled.

Alternatively, *dynamically linked libraries* can be updated anytime without the need to recompile. In UNIX a dynamic library has a ".so" extension. Typically they use less space since no copy is made at compile time.

You can create your own library archive using the "ar" command. There is an

extensive framework to compile large programs called *Make* (Makefiles) and to set up your own detailed compiling environment using tools like *autoconf*. We may touch on these later.

4.3 using gnuplot to view the results of the simple program

Note that we can expect roughly 7 significant digits for a float since there are 24 bits for the significand: $\log_{10}(2^{24}) \approx 7.22$ and 16 digits for double: $\log_{10}(2^{53}) \approx 15.95$. For now we use single precision and set PI=3.141593.

It is almost always useful to present your numerical results graphically, so let's learn a powerful but easy to use package called *gnuplot*. gnuplot produces high quality 2d plots with plenty of control, styles, etc. Typically a program will produce various number of columns of data, for example dependent variable, indep. variable, and error on either or both.

For example our simple program produces three columns,

```
Thomass-MacBook:5621 tblum$ simple 1000 100
X(N)= 100 4.635447e-07
X(N)= 200 5.394357e-07
X(N)= 300 2.360820e-06
X(N)= 400 5.394357e-07
X(N)= 500 4.333986e-06
X(N)= 600 3.027442e-06
X(N)= 700 5.394357e-07
X(N)= 800 5.394357e-07
X(N)= 900 6.838389e-06
Thomass-MacBook:5621 tblum$ simple 1000 100 > simple.out
Thomass-MacBook:5621 tblum$ simple 10000 1000 >> simple.out
Thomass-MacBook:5621 tblum$ simple 100000 10000 >> simple.out
Thomass-MacBook:5621 tblum$ simple 1000000 100000 >> simple.out
```

Note the "redirect" of stdout to a file, ">" and ">>". The first one creates a

new file then writes to it while the latter appends to an existing file or creates a new one if it doesn't exist. Let's use gnuplot to make a 2d plot of the relative error in PI as a function of the size of the small piece which is added to itself N times to get the total. Since the relative error and N range over several orders of magnitude, we should use a log scale on the x and y axes.

Our gnuplot session might look something like

```
Thomass-MacBook:5621 tblum$ gnuplot

G N U P L O T
Version 5.2 patchlevel 4    last modified 2018-06-01

Copyright (C) 1986-1993, 1998, 2004, 2007-2018
Thomas Williams, Colin Kelley and many others

gnuplot home:     http://www.gnuplot.info
faq, bugs, etc:   type "help FAQ"
immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> set logscale xy
gnuplot> plot "simple.out" using 2:3
gnuplot>
```

The plot looks like this on my screen:

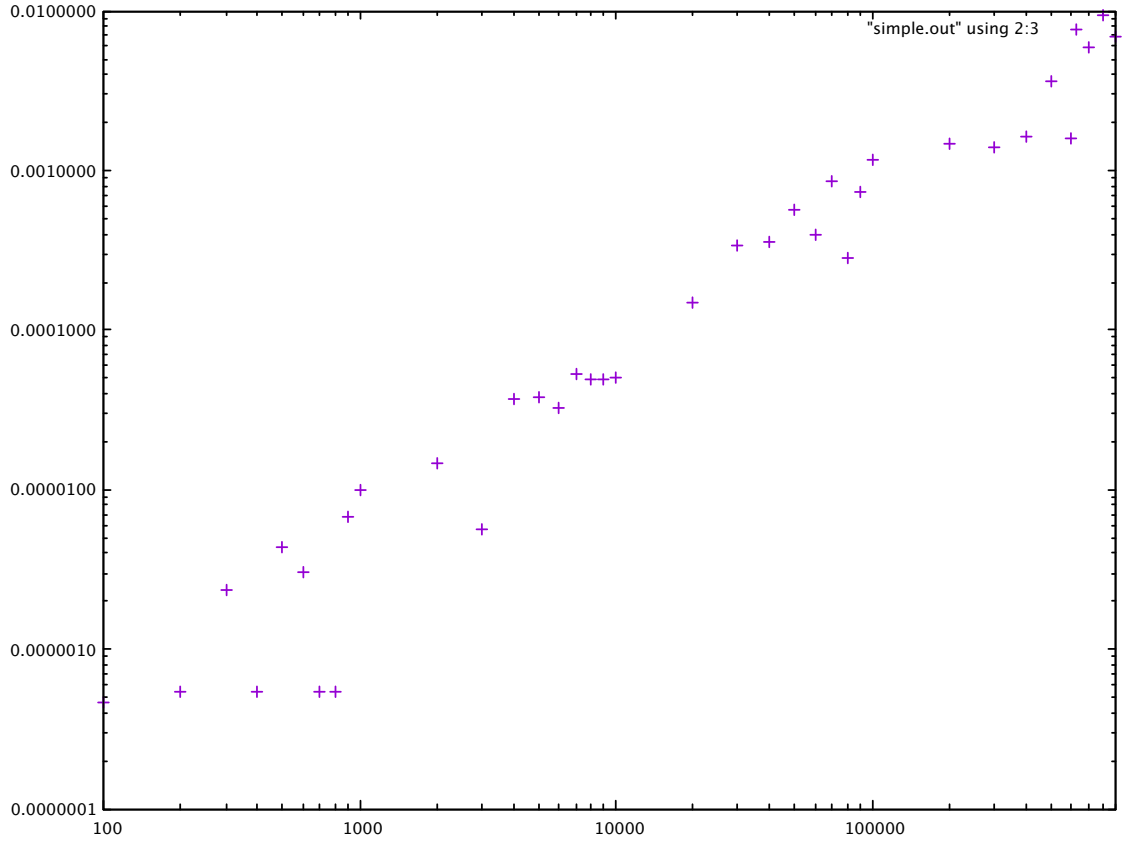

Figure 1: simple plot produced by gnuplot.

We can make it look a bit nicer by adding labels, increasing font size and so on

```
gnuplot> set xlabel 'N' font ",15" offset 0,-1
gnuplot> set ylabel 'Rel. error' font ",15" offset -5.0,0
gnuplot> set xtics font ",15"
gnuplot> set ytics font ",15"
gnuplot> set format y "10^{%L}"
gnuplot> set lmargin 20
gnuplot> set tmargin 4
gnuplot> plot "simple.out" using 2:3
```

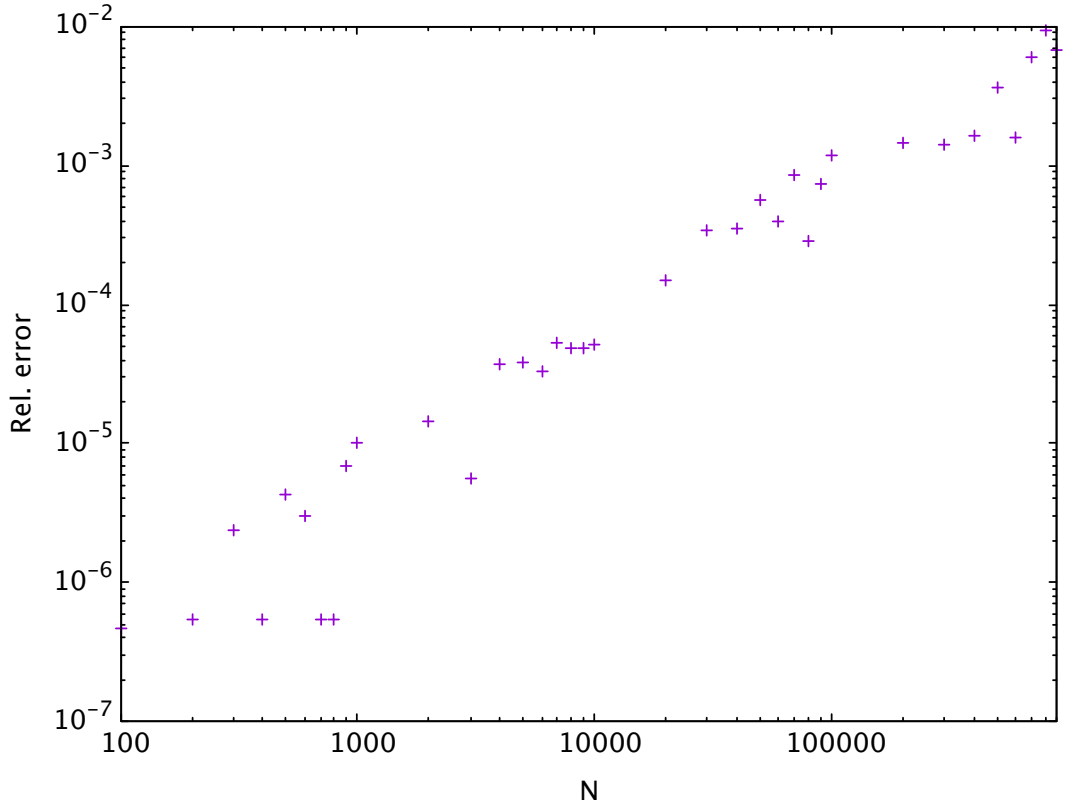

Figure 2: Prettier plot using various controls in gnuplot (see text).

If you're like me you will get tired of typing and re-typing these commands. gnuplot also does "batch" mode. Simply type the above commands into a text file and use the "load" command.

```
Thomass-MacBook:simple tblum$ cat plot-simple
set logscale xy
set xlabel 'N' font ",15" offset 0,-1
set ylabel 'Rel. error' font ",15" offset -5.0,0
set xtics font ",15"
set ytics font ",15"
set format y "10^{%L}"
```

```
set lmargin 20
set tmargin 4
plot "simple.out" using 2:3 with linespoints lc 'red' pt 4
...
gnuplot> load 'plot-simple'
```

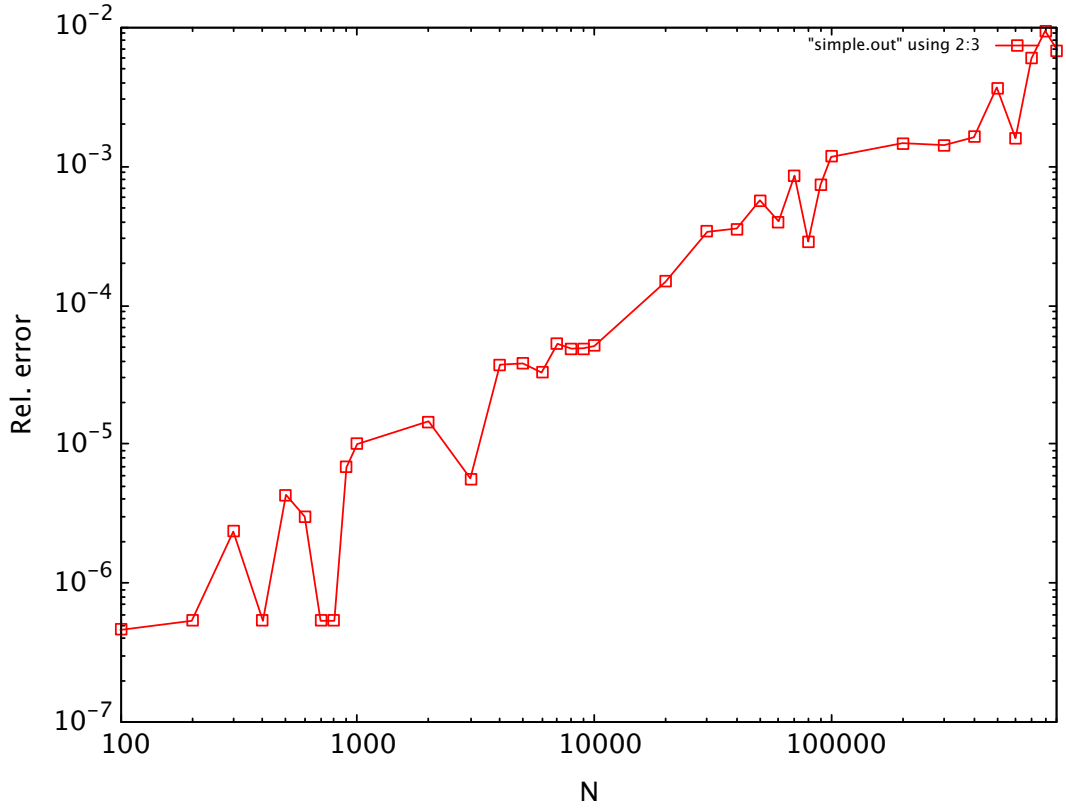

Figure 3: Prettier plot using load command.

On my mac I use a program called 'qt' for my gnuplot 'terminal' which allows me to plot plots directly to my screen. You can also use 'xterm' or many others. Or you can use the 'postscript' terminal to create postscript plots (.ps or .eps) which can be viewed using your postscript viewer (Preview, ...).

It is very handy to have an app installer. On mac, *Homebrew* is nice. For Debian-based linux (Ubuntu) *apt-get*, Redhat (Fedora) *yum*, and Windows *Chocolatey*. For example,

Thomass-MacBook:simple tblum$ brew install gnuplot or

Thomass-MacBook:simple tblum$ brew upgrade gnuplot

Thomass-MacBook:simple tblum$ brew install git

These package managers will check for dependencies, install required ones and the desired package.

4.4 using git to track changes to the simple program

git is a code management system that will track changes, allow multiple versions (branches) and so on. It's very useful for personal projects, or ones developed by a group. We have a git(lab) server in the Physics Department here. To set up an account click on the register tab and use your UConn email address. We'll use gitlab for all of your class related projects.

(demonstrate gitlab server in class– setting permissions, and so on)

Michael Rozman has compiled a list of useful links here.

When you've finished a project or homework assignment (and saved it in a git repository), please grant me *Reader* access so I can view it (my username on the UConn gitlab is tblum). There is a hierarchy of permissions on git that allow various levels of access. git servers set up on github are public unless you pay for privacy.

Gitlab servers are free and private.

Let's start a project for our simple code. We can start a brand new project or begin from an existing one. Let's do the latter since I've already started. I've already set up an ssh key for my git account, so I won't need to type in a password each time I access my gitlab account. You can do this too, and Michael Rozman will help you if necessary. When you create a git repository it sets up some special files and directories under the 'top-level' directory:

```
Thomass-iMac:euler tblum$ git init
Initialized empty Git repository in /Users/tblum/Dropbox/5621/simple/.git/
Thomass-iMac:euler tblum$ more .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
Thom:euler tblum$ git remote add origin https://astral.phys.uconn.edu/tblum/simple.gi
Thomass-iMac:euler tblum$ more .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://astral.phys.uconn.edu/tblum/simple.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

The important line for now is the 'url' one which allows you to communicate with the server via https. Your remote repository will look like mine, except after the '...tblum/euler.git' part. Change yours appropriately. If you set an ssh key for

your gitlab account, you can add an “ssh url” with

```
Thomass-iMac:euler tblum$ vi .git/config
Thomass-iMac:euler tblum$
Thomass-iMac:euler tblum$ more .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = git@astral.phys.uconn.edur:tblum/simple.git
        url = https://astral.phys.uconn.edu/tblum/simple.git
        fetch = +refs/heads/*:refs/remotes/origin/*
```

Git will use the first url it finds in your config file. Now assume you have a file.c that you want to add to the repository.

```
Thomass-MacBook:simple tblum$ git add simple.c
Thomass-MacBook:simple tblum$ git commit simple.c
Thomass-MacBook:simple tblum$ git push
Enter passphrase for key '/Users/tblum/.ssh/id_rsa':
X11 forwarding request failed on channel 0
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 587 bytes | 293.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: The private project tblum/simple was successfully created.
remote:
remote: To configure the remote, run:
remote:   git remote add origin git@astral.phys.uconn.edu:tblum/simple.git
remote:
remote: To view the project, visit:
remote:   https://astral.phys.uconn.edu/tblum/simple
```

```
remote:
To astral.phys.uconn.edu:tblum/simple.git
 * [new branch]      master -> master
Thomass-MacBook:simple tblum$
```

Now, let's say I make some changes to my code. What does git do?

```
Thomass-MacBook:simple tblum$ cp simple-v1.c simple.c
Thomass-MacBook:simple tblum$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   simple.c
Thomass-MacBook:simple tblum$ git diff simple.c
diff --git a/simple.c b/simple.c
index d219b9b..3e41bc8 100644
--- a/simple.c
+++ b/simple.c
@@ -7,26 +7,33 @@
 #include <stdio.h>
 #include <math.h>

-#define PI 3.141593
+#define PI 3.141592

-int main(int argc, char *argv[]){
+/* function to add a number to itself N times */
+float directsum(int N, float X){

-  int N;
-  float X;
-  float sum;
-
-  /* divide pi into 1000 bits */
-  N=1000;
```

```
-  X = PI/(float)N;
-  /* add them up again */
-  sum=0.0;
   int i;
+  float sum=0.0;
+
   for(i=0;i<N;i++){
     sum += X;
   }
-  X = fabs(sum-PI)/PI;
+  return sum;
+}

-  printf("Rel. error on PI(N=%d) = %e \n",N,X);
+int main(int argc, char *argv[]){
+
+  int N;
+  float X;
+
+  for(N=0;N<1000;N+=100){
+
+    if(N){
+      X = directsum(N,PI/(float)N);
+      X = fabsf(X-PI)/PI;
+      printf("X(N)= %d %e \n",N,X);
+    }
+  }

   return 0;
 }
Thomass-MacBook:simple tblum$ git commit -a
Thomass-MacBook:simple tblum$ git push
```

Git shows all the differences between the new edited version and the last commit.

When I "push" a commit I'm updating the (master) branch on the server. git will track the commits locally and on the server. If someone else made changes and pushed them to the server (master) you can get them using the "pull" command.

You can create new branches, delete existing ones, and so on.

Oher useful commands are

- git log (history of all commits)
- git branch (create/delete branch). For example 'git branch masteer' creates the new branch masteer. To delete use 'git branch -D masteer'
- git checkout (switch between branches, update files)
- git clone (cp a complete git repository from the server)
- git –help

What if you accidentally delete a file? use checkout to get it back! Or revert to the previous (or any) version.

5 Pointers and arrays

An important concept in C/C++ is the *pointer*. A pointer variable gives the address of a datum that is stored in memory. Pointers also have types just like ordinary variables. For example

```
/*
   pointer.c C program to demonstrate the use of pointers
*/
#include <stdio.h>
#include <math.h>

int main(){

  int i;
  int *ip;
```

```
  i=7;
  ip = &i;

  printf("int and pointer to int: %d %d %p \n",i,*ip,ip);

  *ip = 6;

  printf("int and pointer to int: %d %d %p \n",i,*ip,ip);

  return 0;
}
```

```
Thomass-MacBook:pointer tblum$ gcc pointer.c
Thomass-MacBook:pointer tblum$ a.out
int and pointer to int: 7 7 0x7ffee2c367c8
int and pointer to int: 6 6 0x7ffee2c367c8
```

- In a declaration, the unary operator * identifies the variable 'ip' as a pointer
- The unary address operator & gives the address to the integer i
- The unary dereferencing operator * gives the *value* of the datum that the pointer points to
- we can also change the variable through the pointer

This leads to another very important and powerful concept in C/C++: the passing of variables *by reference* instead of *by value*. Let's rewrite the above to print out an integer from a function:

```
/*
   pointer1.1.c C program to demonstrate the use of pointers
   and pass by reference
*/
#include <stdio.h>
```

```
#include <math.h>

void mod_i(int j);
void mod_ip(int*);

int main(){

  int i;
  int *ip;

  i=7;
  ip = &i;

  printf("inital i= %d\n",i);

  mod_i(i);
  printf("i= %d\n",i);
  mod_ip(ip);
  printf("*ip: %d \n",*ip);
  i=7;
  printf("*ip: %d \n",*ip);
  mod_ip(&i);
  printf("i: %d\n",i);

  return 0;
}

void mod_i(int k){

  k=6;
  printf("k= %d\n",k);
}
void mod_ip(int *k){

  *k=6;
  printf("k= %d\n",*k);
}
```

```
Thomass-MacBook:pointer tblum$ gcc pointer1.1.c
Thomass-MacBook:pointer tblum$ a.out
inital i= 7
k= 6
i= 7
k= 6
*ip: 6
*ip: 7
k= 6
i= 6
```

Notice that

- the value of i did not change after the call to mod_i(i). In effect all we did was initialize the value of k in the function, then we changed it but only in the function's *scope*. k is completely local to the function. This is called pass by value.
- by passing a pointer to mod_ip(ip) or mod_ip(&i) (pass by reference) we are able to change the value of i in the calling function main after the call to mod_ip.
- ip always points to i, so if we reset i, we reset the value of the thing ip points to
- we defined a new type for our functions: *void* since they do not return a value (no return statement)
- we had to declare both before main (we could have put the actual functions first then no declaration would be necessary)

5.0.1 arrays

An array in C is pretty much like you would expect, at first.

```
/*
   pointer2.c C program to demonstrate the use of pointers
*/

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

int main(){

  int i[3];
  for(int j=0; j<3;j++){
    i[j]=j;
    printf("int and pointer to int: %d %p \n",i[j],&(i[j]));
  }
  for(int j=0; j<3;j++){
    printf("int and pointer to int: %d %p \n",*(i+j),i+j);
  }
  return 0;
}
```

```
Thomass-MacBook:pointer tblum$ gcc pointer.c
Thomass-MacBook:pointer tblum$ a.out
int and pointer to int: 0 0x7ffee319f7bc
int and pointer to int: 1 0x7ffee319f7c0
int and pointer to int: 2 0x7ffee319f7c4
int and pointer to int: 0 0x7ffee319f7bc
int and pointer to int: 1 0x7ffee319f7c0
int and pointer to int: 2 0x7ffee319f7c4
```

Notice the use of the unary address and dereferencing operators &, *:

- The array name is itself a pointer!

- i and &i[0] are pointers (note precedence!) to the first element of the array, i+1 and &i[1] the second, and so on.
- Likewise i[0] and *i are the values of the first element, i[1] and *(i+1) the second and so on.
- Be careful with ()'s: *i + 1 adds 1 to the first element of the array!
- Elements of an array are stored contiguously in memory

Pointer arithmetic is powerful! and works for any defined (even user) data type.

Wait– it gets even better! Let's say you did not know at compile time how big the array is supposed to be. We can pass a variable to our program each time we run it so the array is as big as it needs to be, but not bigger. We use the function *malloc* to accomplish this.

```
/*
   pointer3.c C program to demonstrate the use of pointers
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h> /* malloc */

int main(int argc, char **argv){

  if(argc !=2){
         printf("error: input the size of array\n");
         exit(-1);
  }
  int mysize = atoi(argv[1]);
  int *i;
  i=(int*)malloc(mysize*sizeof(int));

  for(int j=0; j<3;j++){
```

```
    i[j]=j;
    printf("int and pointer to int: %d %p \n",i[j],&(i[j]));
  }
  for(int j=0; j<3;j++){
    printf("int and pointer to int: %d %p \n",*(i+j),i+j);
  }
  for(int j=0; j<3;j++){
    printf("int and pointer to int: %d\n",*i++);
  }

  free(i);

  return 0;
}
```

C/C++ memory management and use through pointers is very powerful but is **fraught with danger**. For each array that we malloc, we need to *free* it when we're done (leave the function), else we will run out of memory! In other words the system will treat the memory as being used until freed. When you call the function again, malloc will allocate a new chunk of memory whether you freed the previous one or not. This is called a "memory leak" and happens all the time. It's a good habit to run your code through the open source program *valgrind* to find memory leaks and other bugs.

```
valgrind a.out 3000
...
int and pointer to int: 2999 0x100dee73c
==42249==
==42249== HEAP SUMMARY:
==42249==     in use at exit: 22,555 bytes in 164 blocks
==42249==   total heap usage: 186 allocs, 22 frees, 43,003 bytes allocated
==42249==
==42249== LEAK SUMMARY:
==42249==    definitely lost: 0 bytes in 0 blocks
==42249==    indirectly lost: 0 bytes in 0 blocks
```

```
==42249==      possibly lost: 72 bytes in 3 blocks
==42249==    still reachable: 200 bytes in 6 blocks
==42249==         suppressed: 22,283 bytes in 155 blocks
==42249== Rerun with --leak-check=full to see details of leaked memory
==42249==
==42249== For counts of detected and suppressed errors, rerun with: -v
==42249== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

but if we forget the free statement, we get

```
...
==42260== LEAK SUMMARY:
==42260==    definitely lost: 12,000 bytes in 1 blocks
...
```

Note: If we use the *i++ line at the bottom we get a memory leak or worse even if we use the free statement! What is going on? It turns out that we have incremented the pointer such that at the end of the loop it is pointing to the memory location just after the last element, so we're freeing memory that wasn't allocated. When we run it we get an error:

```
Thomass-iMac:pointer tblum$ a.out 2
int and pointer to int: 0 0x7fc9f2c027c0
int and pointer to int: 1 0x7fc9f2c027c4
int and pointer to int: 0 0x7fc9f2c027c0
int and pointer to int: 1 0x7fc9f2c027c4
int and pointer to int: 0 0x7fc9f2c027c4
int and pointer to int: 1 0x7fc9f2c027c8
a.out(55068,0x7fffab064380) malloc: *** error for object 0x7fc9f2c027c8:
pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort trap: 6
```

In C and C++ the system has (at least) two partitions of real memory. They are called the "stack" and the "heap". Usually the heap is much bigger than the stack. Things like ordinary variable declarations and even fixed length arrays, are

put on the stack at compile time along with instructions. Variable length arrays, on the other hand, are put on (and taken off of) the heap. As you might guess, access of stack memory is much faster.

In C++ memory allocation for the heap is a bit different, though you can use malloc and free if you want to. In C++ one typically uses new commands called *new* and *delete*.

Multi-dimensional arrays are constructed through pointers to pointers to pointers to ... In fact in the examples above *argv* is a pointer to a char pointer and the size of the array is parsed at run time using argc and the size of each read string. An example of a program using a two dimensional array of floats is given below.

First let's start a new branch called 2d in git by typing "git branch 2d" just to see how it works. Notice only the new code exists in this branch (which I created beforehand). We can toggle back to the master branch with "git checkout master".

Here's the new code:

```
...
  double **a;
  a=(double**)malloc(mysize*sizeof(double*));
  for(int j=0; j<mysize;j++){
    a[j] = (double*) malloc(mysize*sizeof(double));
    if(&a[j]==NULL){
      printf("Not enough memory %p\n", &a[j]);
      exit(-1);
    }
  }
  for(int j=0; j<mysize;j++){
```

```
    for(int k=0; k<mysize;k++){
      a[j][k] = 0.0;
      printf("a[%d][%d] = %e ",j,k,a[j][k]);
    }
    printf("\n");
  }

  for(int j=mysize-1; j>=0;j--)
    free(a[j]);
  free(a);
...
```

```
Thomass-MacBook:pointer tblum$ gcc pointer4.c
Thomass-MacBook:pointer tblum$ a.out 2
a[0][0] = 0.000000e+00 a[0][1] = 0.000000e+00
a[1][0] = 0.000000e+00 a[1][1] = 0.000000e+00
```

There's alot going on here.

- There are two malloc's
- Note the casts **double and *double (malloc returns a (void*), or pointer to anything).
- Notice the order of free is reverse to malloc. This is to avoid the phenomenon of "memory fragmentation" which can be a serious problem for large arrays

Warning: malloc may not barf even if there is not enough memory to alloc your array, so it's good to trap mallocs. (check that the pointer is initialized to a non-NULL address). Later versions of malloc may do this for you.

6 I/O

I'm going to skip I/O for now since we don't yet need more than what we already have.

7 Upgrading to C++

As mentioned, there are many nice features of C++ (as the name implies!), a few of which we will take advantage of eventually. For now we press on.

8 Numerical Solution of ODE's

We are interested in solving ordinary differential equations (ODE's) numerically (as opposed to computing a definite integral via Simpson's rule or the Trapezoid rule). Hence we will work in 1d+time (or analogous coordinate). Later we will solve partial differential equations in higher dimensions. It has been known for a long time how to do this: Euler's method, Runge-Kutta, etc. Today of course we have powerful computers to take advantage.

8.1 Euler's method

We won't actually use this method to solve real problems because it is well known that it has serious deficiencies, namely poor accuracy, and even worse, instability.

However it is very simple and also easy to see how the above two problems arise. I'm taking the following description from Fitzpatrick's lecture notes (the discussion of numerical errors is especially clear), but it is commonly found elsewhere.

The method is one of the oldest and simplest numerical methods available. It was invented by Leonhard Euler in the 18th century.

8.1.1 the method

Let's start with a generic first order ODE,

$$y' = f(x, y) \tag{1}$$

$$y(x_0) = y_0, \tag{2}$$

and let's further imagine that we will numerically solve for the solution on a suitably fine, discrete, set of points x_n, each separated from its neighbors by a small constant spacing a. We can think of this one dimensional chain of discrete points $(x_0, x_1, x_2, \ldots, x_{N-1})$ as a *lattice*. Euler's idea was that for small enough a, the solution at $y(x_n + a)$ is simply and approximately

$$y_{n+1} = y_n + af(x_n, y_n) \tag{3}$$

i.e., a straight line between the points, using the known slope. In practice, to "solve" the ODE, one simply steps from x_0 where y_0 is known to some desired point x_N in N small steps. Aside: conventionally the true, continuous solution is written $y(x_n)$ while for the discrete, approximate version we write y_n.

8.1.2 the problem

.

One should immediately have several questions. How small is small? How good is Euler's approximation, or how much does it differ from the real (continuous) solution? To begin answering these (related) questions, let's look at the Taylor expansion of the actual solution around the point x_n,

$$y(x_n + a) \;=\; y(x_n) + af(x, y) + \frac{a^2}{2} f'(x, y) + \ldots \tag{4}$$

Comparing to Euler's solution, we see he made a "mistake" (error) of $O(a^2)$ in a single step. If $a \ll 1$ this might be ok. Usually we are interested in many steps, on the order of $1/a$, so in total, if we assume the errors from each step add together, the final error is actually $O(a)$. This error is called a truncation, or disrcetization error. And this method is one of a class called *finite difference* methods since the continuous derivative is replaced by a difference,

$$\frac{y_{n+a} - y_n}{a} \;=\; f(x_n, y_n) + O(a^2) \tag{5}$$

Euler's scheme is called an "order a" method because it is accurate up to terms of $O(a)$. We can do much better. But first, let's look more closely at the discretization error and the lattice spacing.

The first thing to realize is the truncation error does *not* result from round off errors. Even a perfect numerical computation with infinitely precise arithmetic using Euler's method induces the $O(a)$ error. Let's define the round-off error incurred after one step to be $O(\eta)$ where $\eta \approx 10^{-16} \ll 1$ for double precision arithmetic. The total error in our solution after $N \sim 1/a$ steps, assuming they accumulate independently, is then

$$\epsilon \sim \frac{\eta}{a} + a \tag{6}$$

If $a \gg \eta$ (typically the case) the truncation error dominates and vice versa. The error is a minimum with respect to a for $a = \sqrt{\eta}$. Typically this means single precision is

not good enough for numerical integration ($\sqrt{10^{-7}} \sim 10^{-4}$).

8.1.3 numerical instability

Let's look at our first example of numerical instability. To be specific, consider the ODE and initial condition

$$y' = -\alpha y, \tag{7}$$

$$y(0) = 1, \tag{8}$$

with $\alpha > 0$. The solution is $y = \exp -\alpha x$, an exponential that is ubiquitous in physics. On the other hand, Euler's solution is

$$y_{n+1} = y_n + ay' \tag{9}$$

$$= (1 - \alpha a)y_n \tag{10}$$

My program looks like this:

```
/*
euler.c:
    simple implementation of 1st order Euler scheme
    to integrate a 1st Order ODE
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define alpha 1.0 /* the exponent */

/* simple exponential */
double yderiv(int k, double *y){

  return -alpha*y[k];
```

```
}

int main(int argc, char **argv){

  if(argc!=3){
    printf("error: 2 args: size and spacing of lattice\n");
    exit(-1);
  }
  int N=atoi(argv[1]);
  double a = atof(argv[2]);

  double *y; /* the solution */
  y=(double*)malloc(N*sizeof(double));

  y[0]=1.0;  /* intial value */

  for(int i=1;i<N;i++){
    y[i]=y[i-1] + a * yderiv(i-1, y);
    printf("y[ %d ]= %e %e\n",i,y[i],exp(-alpha*i*a));
  }
  free(y);

  return 0;
}
```

Here's the numerical solution for $\alpha = 100.0$ and $a = 0.025$:

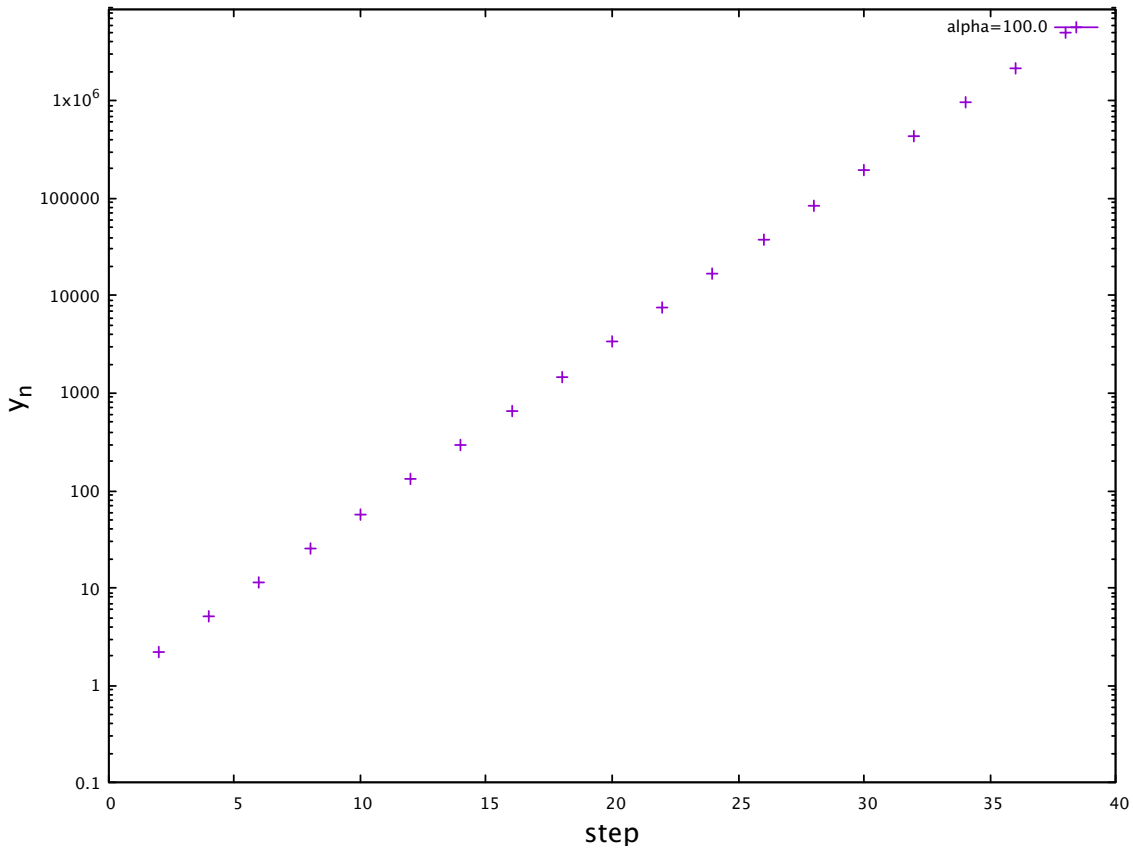

Figure 4: Euler method for solution of $y' = -100.0y$ with lattice spacing $a = 0.025$. The solution diverges exponentially (only even steps shown, odd are less than zero).

What happened?! In fact it's easy to see from the Euler method solution that when $a > 2/\alpha$ then $|y_{n+a}| > |y_n|$: the solution at the next step always grows in magnitude (it also oscillates like $(-1)^n$).

$$\frac{y_{n+a}}{y_n} = (1 - a\alpha)|_{a=2/\alpha} = -1.0, \tag{11}$$

and in our example $a = 0.025 > 2/100.0 = 0.02$. On the other hand, if $\alpha = 1.0$ and $a = 0.02$, the Euler solution is not terrible: the relative error at step 40 is a bit over 1%.

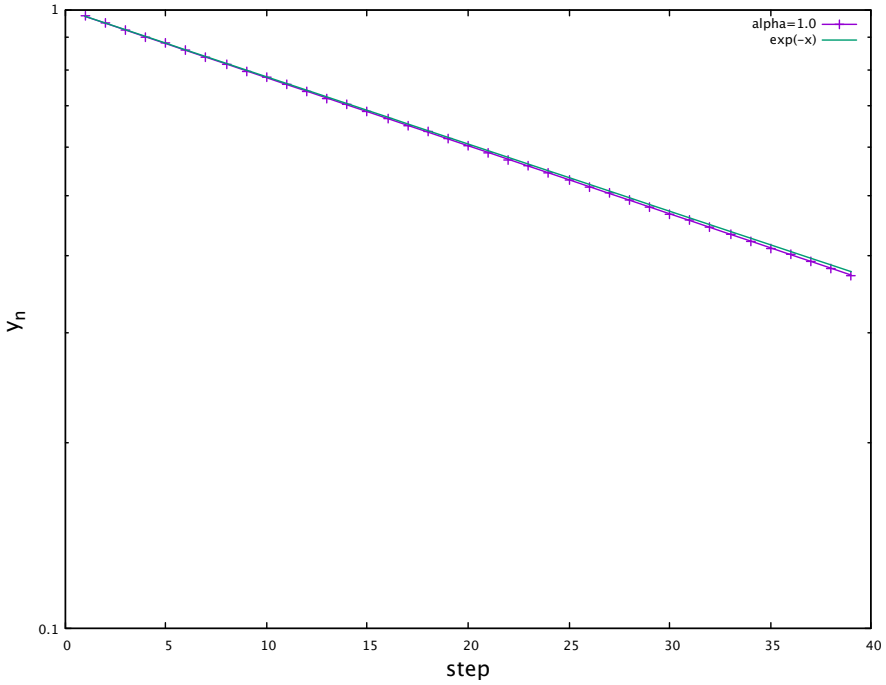

Figure 5: Euler method for solution of $y' = -1.0y$ with lattice spacing $a = 0.025$. The exact solution is also shown.

The above begs the question: can we be sure to get a solution with a given numerical algorithm? The answer could be never, sometimes, or always. We'll look

at this question in more detail as we investigate other algorithms. It should be clear that both *small errors and stability* are important criteria for a good numerical algorithm.

9 Runge-Kutta integrators

There are many schemes available that are better than the 1st order Euler method. For example there is a family of integrators invented in 1901 called Runge-Kutta (after the 19/20th century Germans who invented them. There's a crater on the moon named after Runge).

To see how one might do better, recall why Euler is 1st order in the first place. Start with the Taylor expansion

$$y(x_n + a) = y(x_n) + af(x_n, y) + \frac{a^2}{2} f'(x_n, y) + \dots \tag{12}$$

rearranging,

$$\frac{y(x_n + a) - y(x_n)}{a} = f(x_n, y) + O(a) \tag{13}$$

Euler's difference operator (sometimes called a stencil) is *asymmetric*. In fact it's called a forward difference. We could just as well defined a backward difference that is also 1st order:

$$y(x_n - a) = y(x_n) - af(x_n, y) + \frac{a^2}{2} f'(x_n, y) + \dots \tag{14}$$

or

$$\frac{y(x_n) - y(x_n - a)}{a} = f(x_n, y) + O(a). \tag{15}$$

Notice in the Taylor expansions for $y(x_n \pm a)$ the even (odd) terms have the same (opposite) sign. Subtracting the two leads to a *symmetric*, or *central* difference,

$$\frac{y(x_n + a) - y(x_n - a)}{2a} = f(x_n, y) + O(a^2) \tag{16}$$

The $O(a)$ terms cancel! The central difference is accurate to $O(a^2)$ corrections. Now we see why Euler's method is no good; it uses an asymmetric difference. We should use a symmetric difference instead. It's easy to see that if the slope is evaluated at the midpoint, $x_{n+a/2}$, in Euler's original method instead of at x_n, we get the 2nd order accurate (midpoint) Runge-Kutta method,

$$y_{x_n+a} = y_n + af(x_n + a/2, y_{x_n+a/2}) \quad \text{midpoint} \tag{17}$$

$$y_{x_n+a/2} = y_n + \frac{a}{2} f(x_n, y_n) \quad \text{Euler} \tag{18}$$

(it's equivalent to using a central difference: just Taylor expand about $x_n + a/2$)

The algorithm is

1. evaluate y at the midpoint with the Euler method
2. evaluate the slope at the midpoint using this new value of y
3. advance y a step with the Euler method using the slope at the midpoint

Now the agreement is 0.01%!

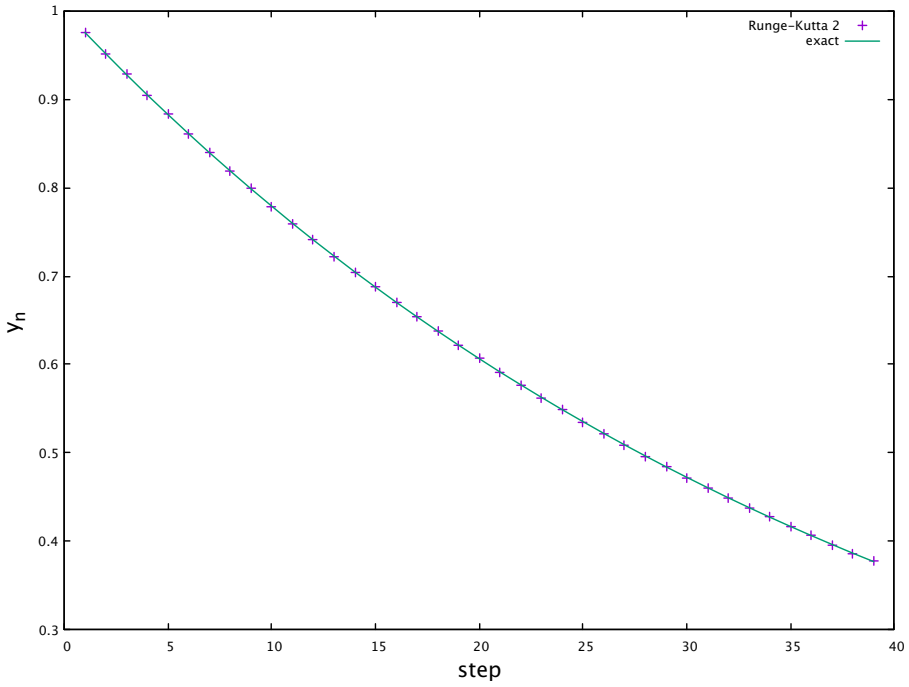

Figure 6: 2nd order Runge-Kutta method for solution of $y' = -1.0y$ with lattice spacing $a = 0.025$. The accuracy at the last step is $O(a)$ better than for Euler (the difference with exact result is smaller).

We can do much better with a few more flops (floating point operations). The four-step Runge-Kutta method is 5th order accurate:

$$
\begin{aligned}
y_{x_n+a} &= y_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) && (19)\\
k_1 &= af(x_n, y_n) && (20)\\
k_2 &= af(x_n + a/2, y_n + k_1/2) && (21)\\
k_3 &= af(x_n + a/2, y_n + k_2/2) && (22)\\
k_4 &= af(x_n + a, y_n + k_3) && (23)
\end{aligned}
$$

The slope is evaluated at the end points and twice in the middle, and the results are averaged (with more weight in the middle). Graphically it looks like this (my definition for k differs by a factor of a).

10 Higher order ODE's

So far we have only looked at first order ODE's. Often we are interested in 2nd or higher order ODE's. For example Newton's 2nd Law is a 2nd order equation. How do we handle this situation? There are several ways. One might think to define a difference operator for a 2nd derivative. In fact this is easy to do:

$$\Delta_+ \equiv f(x+a) - f(x) \quad \text{(forward)} \tag{24}$$

$$\Delta_- \equiv f(x) - f(x-a) \quad \text{(backward)} \tag{25}$$

$$\Delta_-\Delta_+ f(x) = \Delta_- \frac{(f(x+a) - f(x))}{a} \tag{26}$$

$$= \frac{f(x+a) - f(x) - f(x) + f(x-a)}{a^2} \tag{27}$$

$$= \frac{f(x+a) + f(x-a) - 2f(x)}{a^2} \tag{28}$$

This is not unique. You could have done Δ_+ twice, and so on. Notice that

$$f(x \pm a) = f(x) \pm af' + \frac{a^2}{2} f'' \pm O(a^3) \tag{29}$$

$$f(x+a) + f(x-a) - 2f(x) = a^2 f'' + O(a^4). \tag{30}$$

10.1 Verlet integration

The simple central difference for the second derivative leads to the method of *Verlet* integration after the guy who rediscovered it in the 1960's for molecular dynamics (it was invented much earlier). It's also used in computer graphics.

Let's say we want to integrate Newton's equation of motion for some external force (potential) acting on a particle (this easily generalized to multi-dimensions and many particles).

$$\ddot{x} = \frac{F(t)}{m} \tag{31}$$

Applying our central difference operator gives

$$x_{t+a} + x_{t-a} - 2x_t = a^2 \frac{F(t)}{m} \tag{32}$$

$$x_{t+a} = 2x_t - x_{t-a} + a^2 \frac{F(t)}{m} \tag{33}$$

Just update the value of x_{t+a} from the previous two values and the force evaluated at t! Starting is a bit tricky since we need $x(1)$ and $x(0)$. We fudge this by Taylor expanding $x(1)$ about $x(0)$,

$$x(1) = x(0) + av(0) + \frac{a^2}{2}a(0) + O(a^3) \tag{34}$$

where $a(0) = F(0)/m$ and $v(0)$ is the initial velocity (we need two conditions for 2nd order ODE). **Warning**: The error for the Verlet method is $O(a^2)$ which might seem intuitive since it integrates a second order ODE but is not necessarily obvious.

10.1.1 calculating velocities

You may need to know the velocity in your calculation. We can easily compute it from our solution for the position as function of time step,

$$v_t = \frac{x_{t+a} - x_{t-a}}{2a} \tag{35}$$

which is 2nd order accurate but a step *behind* x. One could advance another half-step and still retain 2nd order accuracy,

$$v_{t+a/2} = \frac{x_{t+a} - x_t}{a} \tag{36}$$

or even advance a full step with a forward difference but at the cost of reducing accuracy to 1st order.

10.2 velocity Verlet

You might also think to rewrite the 2nd Law as two first order equations:

$$F = m\ddot{x} \tag{37}$$

$$= m\dot{v} \tag{38}$$

$$v = \dot{x} \tag{39}$$

The last two equations are *coupled*. They can be integrated simultaneously with two initial conditions, $v(0) = v_0$ and $x(0) = x_0$. Discretizing these leads to

$$x_{t+a} = x_t + av_t + \frac{a^2}{2}a_t \tag{40}$$

$$v_{t+a} = v_t + a\frac{a_t + a_{t+a}}{2} \tag{41}$$

where again $a_t = F(t)/m$. Here we solved for x_{t+a} using the Verlet algorithm (after a small rearrangement), then updated the velocity according to the velocity ODE with the midpoint method. One can show this has the same order error as Verlet. According to Wikipedia, the most common implementation is the following "leapfrog" scheme

1. calculate $v_{t+a/2} = v_t + aa_t/2$
2. calculate $x_{t+a} = x_t + av_{t+a/2}$
3. compute $a_{t+a} = F(t+a)/m$
4. calculate $v_{t+a} = v_{t+a/2} + aa_{t+a}/2$

The leapfrog scheme is a *symplectic* integrator, or numerical method for Hamiltonian systems. More importantly the scheme is *reversible*: if you integrate backwards from the final solution by reversing the sign of the time step, you get back to the

initial values. This means the integrator conserves the energy of the (discrete Hamiltonian) system.

You will be asked to use velocity Verlet to solve a molecular dynamics problem in 2d for an upcoming homework.

11 The diffusion equation

Let's look into the solution of our first partial differential equation, the 1d diffusion equation. This will allow us to introduce several important concepts and problems, including von Neumann stability analysis, implicit finite difference schemes, and matrix inversion.

The diffusion equation is

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial^2 x} \tag{42}$$

where $T = T(x,t)$ is the temperature of some material, say, and $D > 0$. We're interested in a finite system with boundaries x_1 and x_2 where the temperature and/or its derivative is specified, and T is specified everywhere at some time t_0. Fixed temperature and temperature gradient boundary conditions are known as Dirichlet and Neumann (not the same as von Neumann), respectively. In general,

$$\alpha_1(t)T(x_1,t) + \beta_1(t)\frac{\partial T(x_1,t)}{\partial x} = \gamma_1(t) \tag{43}$$

$$\alpha_2(t)T(x_2,t) + \beta_2(t)\frac{\partial T(x_2,t)}{\partial x} = \gamma_2(t) \tag{44}$$

$$\tag{45}$$

We take a simple forward difference in time and our central difference in space to get

$$\frac{T_i^{n+1} - T_i^n}{a_t} = D\frac{T_{i+1}^n + T_{i-1}^n - 2T_i^n}{a_x^2}. \tag{46}$$

The boundary conditions become, after a bit of algebra,

$$T_0^n = \frac{a_x \gamma_1^n - \beta_1^n T_1^n}{a\alpha_1^n - \beta_1^n} \tag{47}$$

$$T_{N+1}^n = \frac{a_x \gamma_2^n + \beta_2^n T_1^n}{a\alpha_2^n + \beta_2^n} \tag{48}$$

$$\tag{49}$$

The algorithm as written is clearly $O(a_t, a_x^2)$ and can be solved with a straight forward (explict) step.

$$T_i^{n+1} = T_i^n + C\left(T_{i+1}^n + T_{i-1}^n - 2T_i^n\right), \tag{50}$$

$$C = \frac{a_t}{a_x^2} D. \tag{51}$$

11.1 diffusion example

Here I'm following Fitzpatrick's notes, sections 6.3 and 6.4. For the initial temperature distribution at t_0 he takes a gaussian

$$T(x, t_0) = \exp \frac{-x^2}{4Dt_0} \tag{52}$$

and spatial boundary conditions,

$$T(\pm x_0, t) = \sqrt{\frac{t_0}{t}} \exp \frac{-x_0^2}{4Dt} \tag{53}$$

where $x_1 = -x_0$ and $x_2 = +x_0$. Why did he do that?

11.1.1 code

(aside: to increase/decrease the font in emacs, x̂x̂+/-, or "control-x+-") Here is the code we wrote in class. Try it!

```
/*
Simple diffusion equation example
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double ax;
double at=0.004;
int Nx = 100;
double x0=5.0;
double D=1.0;
double t0=0.1;

//void first_step(double **T);

void first_step(double **T){

  for(int i=0;i<Nx+2;i++){
    double x=-x0+(double)i*ax;
    T[0][i] = exp(-0.25*x*x/D/t0);
  }
}
void next_step(double **T){

  double C = at*D/ax/ax;
  for(int i=1;i<Nx+1;i++){
    double x=-x0+(double)i*ax;
    T[1][i] = T[0][i]+C*(T[0][i+1]+T[0][i-1]-2*T[0][i]);
  }
  /* update boundaries */
  T[1][0] = sqrt((double)n/double(n+1))* T[0][0];
  T[1][Nx+1] = sqrt((double)n/double(n+1))* T[0][0];
}

int main(int argc, char **argv){
```

```
  double **T;
  ax = 2*x0/((double)Nx+2);
  T = (double**)malloc(2*sizeof(double*));
  for(int i=0;i<2;i++){
    T[i] = (double *)malloc((Nx+2)*sizeof(double));
  }

  first_step(T);

  int Nsteps = 1.0/at;
  for(int n=1; n< Nsteps; n++){
    next_step(T,n);
  }

  for(int i=1;i>=0;i--){
    free(T[i]);
  }
  free(T);

}
```

There are several trivial errors in the above, and some bugs. Did you find them all?

The result from code I prepared before class is shown in Fig. 7.

As we saw for the Euler method, the numerical solution of our simple diffusion problem can become unstable. For example if we increase the number of spatial sites (decrease the lattice spacing), we get severe oscillations which clearly signal a breakdown (see Fig. 8) of the method (instability).

By the way, the reason we did first order accurate in time, and not second, is that a central time difference results in an unconditionally *unstable* algorithm! In this case (and many others) there is good reason for the breakdown: for some values of C the difference equation (50) is unphysical (see Anderson, Tannehill, and Pletcher). To

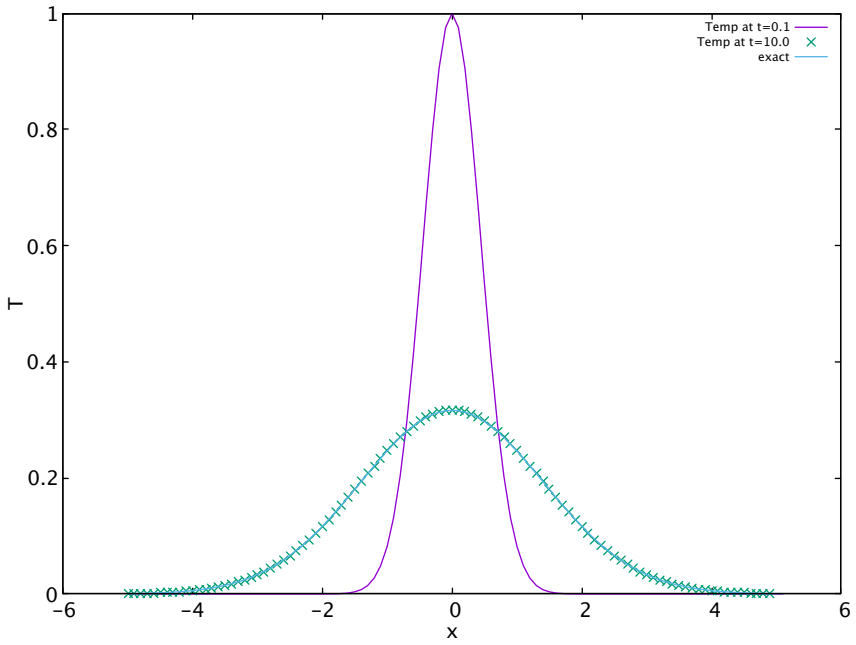

Figure 7: Simple diffusion for initial gaussian temperature distribution at $t = 0.1$. $t = 1.0$, $x_0 = 5.0$, $D = 1.0$, $N_x = 100$, $a_t = 4 \times 10^{-3}$.

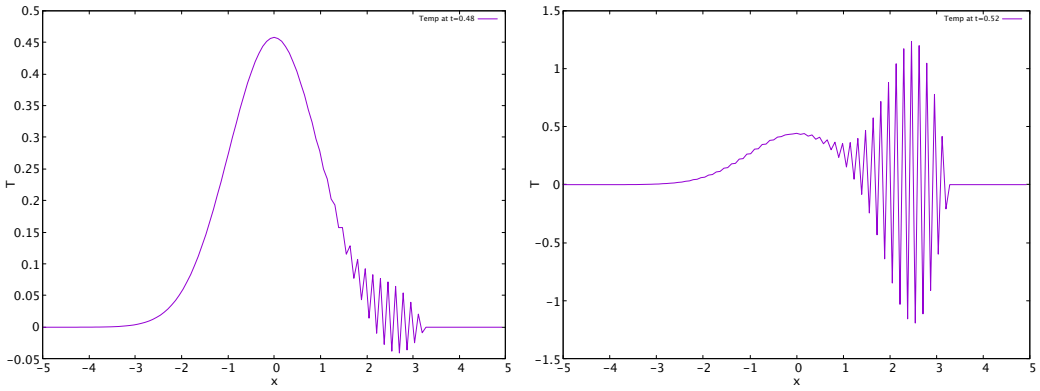

Figure 8: Simple diffusion for initial gaussian temperature distribution at $t = 0.1$. $t = 0.48$ (left), $x_0 = 5.0$, $D = 1.0$, $N_x = 122$, $a_t = 4 \times 10^{-3}$. $t = 0.52$ (right). Both exhibit instability, which increases at later times.

see this suppose at time step n $T^n_{i+1} = T^n_{i-1} = 100$ degrees while $T^n_i = 0$ degrees. If $C = 1$, at the next step $T^{n+1}_i = 200!$ (plug in to Eq. 50) while the maximum temperature we should expect is 100. In the next section we will show rigorously that $C = 1.0$ is indeed unstable.

11.2 von Neumann stability analysis

To see what's going on we perform the well known von Neumann stability analysis of our difference equations. Since we started with a *linear* PDE, we know the solution can be given in terms of a superposition of all possible "modes" allowed on the discrete lattice.

$$T(t,k) \;=\; T(t)e^{ikx} \tag{54}$$

where k is the wave number and $T(t)$ its time-dependent amplitude. For now ignore boundary conditions and insert into the difference equation,

$$\begin{aligned} T^{n+1}e^{ikx_i} &= T^n e^{ikx_i} + C\left(T^n e^{ik(x_i+a_x)} + T^n e^{ik(x_i-a_x)} - 2T^n e^{ikx_i}\right) && (55)\\ T^{n+1} &= (1 + C\left(e^{ika_x} + e^{-ika_x} - 2\right))T^n && (56)\\ &= (1 + 2C(\cos ka_x - 1))T^n. && (57) \end{aligned}$$

The magnitude of the "amplification" factor on the right hand side must be less than one for all k for stability (similar to the Euler case we looked at earlier).

$$|1 + 2C(\cos ka_x - 1)| \;<\; 1 \tag{58}$$

One can see that the most extreme case is when the cosine equaks -1, which translates to the condition $|1 - 4C| < 1$, or $C < 1/2$ for stability. Since $k = 2\pi/\lambda$, we see the wavelength of the least stable mode has $\lambda = a_x/2$, *i.e.* half the grid spacing. The

high modes are less stable. Now we can understand why our solution started to breakdown as we decreased the spatial lattice spacing. For our examples, $C = 0.408$ and 0.605 for $N_x = 100$ and 122, respectively.

11.2.1 Courant-Friedrichs-Lewy criterion

For *hyperbolic* equations (*e.g.* the wave equation) the conditional stability requirement is known as the Courant-Friedrichs-Lewy criterion, or CFL criterion for short. The Courant number is defined as

$$\nu = c\frac{a_t}{a_x} \tag{59}$$

and $|\nu| \leq 1$ for stability. According to Anderson, Tannehil, and Pletcher, the Courant, *et al.* paper (1928) was the first extensive study of stability and convergence, and gave birth to the field of modern numerical methods for PDE's.

11.2.2 Lax's Equivalence Theorem

The above analysis was for a linear PDE/FDE. In fact we should mention in this case that it can be proved that the solution of a *consistent* FDE will converge to the exact PDE solution in the limit of vanishing lattice spacing *iff* the scheme is stable. Here consistency means the truncation error vanishes as $a \to 0$ which is usually the case. Note however schemes can have errors of order $O((a_t/a_x)^n)$ which might not vanish if $a_t/a_x =$ constant. This convergence theorem is known as Lax's Equivalence Theorem. It has not been proven for the non-linear case, though most practitioners assume it is also true for that case.

12 Implicit differencing schemes

We've run into the instability problem a couple of times, and it turns out this a general problem for *explicit* differencing schemes where the right hand side is evaluated at one time step behind the left hand side. This leads to the idea of an *implicit* scheme where the LHS is (partially) evaluated at the same time.

12.0.1 Crank-Nicholson scheme

Our first example is the Crank-Nicholson scheme. The idea is to simply take the average of the central difference operator evaluated at both time steps,

$$\frac{T_i^{n+1} - T_i^n}{a_t} = \frac{D}{2}\left\{\frac{T_{i+1}^n + T_{i-1}^n - 2T_i^n}{a_x^2} + \frac{T_{i+1}^{n+1} + T_{i-1}^{n+1} - 2T_i^{n+1}}{a_x^2}\right\}. \quad (60)$$

The Crank-Nicholson scheme is actually $O(a_t^2, a_x^2)$ accurate. To see this, Taylor expand about $t_n + a_t/2$. It works just like the Euler and RK2 schemes we saw earlier.

Before we think about how we might solve this difference equation for T_i^{n+1}, let's show that it's *unconditionally* stable. Using the von Neumann stability analysis as before,

$$\begin{aligned} T^{n+1}e^{ikx_i} &= \left(e^{ikx_i} + \frac{C}{2}\left(e^{ik(x_i+a_x)} + e^{ik(x_i-a_x)} - 2e^{ikx_i}\right)\right)T^n \\ &+ \frac{C}{2}\left(e^{ik(x_i+a_x)} + e^{ik(x_i-a_x)} - 2e^{ikx_i}\right)T^{n+1} \quad (61) \\ & \quad (62) \end{aligned}$$

or, rearranging,

$$\left(1 - \frac{C}{2}\left(e^{ika_x} + e^{-ika_x} - 2\right)\right)T^{n+1} = \left(1 + \frac{C}{2}\left(e^{ika_x} + e^{-ika_x} - 2\right)\right)T^n \quad (63)$$

$$T^{n+1} = \frac{1 + C\left(\cos ka_x - 1\right)}{1 - C\left(\cos ka_x - 1\right)}T^n \quad (64)$$

which has magnitude less than 1 for all k! It turns out the affect of the boundary conditions does not change the conclusion. See Anderson, Tannehill, and Pletcher to handle boundary conditions. In fact we need a "matrix method" where the boundary conditions appear explicitly. We can then diagonalize the matrix, and it's eigenvalues should obey the von Neumann conditions.

How do we find the solution to our difference equation since T^{n+1} appears on both sides? Let's rearrange again, putting all the terms to be evaluated at $n+1$ on the LHS.

$$T_i^{n+1} - \frac{C}{2}\left(T_{i+1}^{n+1} + T_{i-1}^{n+1} - 2T_i^{n+1}\right) = T_i^n + \frac{C}{2}\left(T_{i+1}^n + T_{i-1}^n - 2T_i^n\right). \quad (65)$$

If we think of T at each time slice as vector that spans the spatial dimension of the lattice, we can write this system of linear equations as a matrix equation,

$$MT^{n+1} = R^n \quad (66)$$

where M in this case is a *sparse* matrix, in fact one that has a special name, a *tridiagonal* matrix.

$$\begin{bmatrix} \cdots & \ddots & \ddots & \ddots & 0 & 0 & \cdots & & \\ \cdots & -\frac{C}{2} & 1+C & -\frac{C}{2} & 0 & 0 & \cdots & & \\ \cdots & 0 & -\frac{C}{2} & 1+C & -\frac{C}{2} & 0 & \cdots & & \\ \cdots & 0 & 0 & -\frac{C}{2} & 1+C & -\frac{C}{2} & 0 & \cdots & \\ \cdots & 0 & 0 & 0 & \ddots & \ddots & \ddots & 0 & \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ T_{j-1}^{n+1} \\ T_j^{n+1} \\ T_{j+1}^{n+1} \\ T_{j+2}^{n+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ R_{i-1}^n \\ R_i^n \\ R_{i+1}^n \\ R_{i+2}^n \\ \vdots \end{bmatrix} \quad (67)$$

and the solution is given by the inverse of M acting on T^n

$$T^{n+1} = M^{-1}R^n \quad (68)$$

$$R^n = (1-C)T_i^n + \frac{C}{2}(T_{i+1}^n + T_{i-1}^n) \quad (69)$$

So the price we pay for unconditional stability is the computation of the inverse of a matrix. By the way, implementing the boundary conditions is not difficult. For Dirichlet and Neumann we simply move the known values from the LHS to the RHS and eliminate the first row and column and the last row and column. If the b.c.'s are (anti-)periodic, entries will appear in the upper right and lower left corners. Then the matrix is no longer tridiagonal, and we have to deal with a slightly more complicated matrix.

13 Inverse of a Matrix I. Exact Method

There are many ways to invert a matrix. Often if the matrix is not too large, it can be done exactly. In general for an $N \times N$ square matrix exact solutions require $O(N^3)$ operations which is too expensive for large N. Special matrices like the tridiagonal one we just saw can be inverted exactly in fewer steps. For general matrixes we will need iterative methods. Lets start with a general exact scheme, LU decomposition which is $O(N^3)$ (there are others but they are also $O(N^3)$ like Gaussian elimination or Gauss-Jordan elimination). It simplifies nicely for the tridiagonal case. Finally we will delve into *Krylov space* solvers for large, sparse matrices.

13.1 LU decompositon

See Numerical Recipes (chapter 2) for a full explanation. Let's suppose we can factorize our matrix to be inverted,

$$M = LU \tag{70}$$

where L is a lower-triangular matrix and U is upper-triangular. This means all elements above (below) the diagonal are zero. If this LU decomposition can be

done, then the inverse can be computed easily using forward substitution followed by backward substitution. Consider the following linear equation,

$$\begin{aligned} Mx = LUx &= b && (71)\\ L(Ux) = Ly &= b \quad \text{(forward sub)} && (72)\\ Ux &= y \quad \text{(backward sub)} && (73) \end{aligned}$$

In other words,

$$\begin{aligned} L_{00}y_0 &= b_0 && (74)\\ L_{10}y_0 + L_{11}y_1 &= b_1 && (75)\\ &\dots \end{aligned}$$

and similarly for x.

$$\begin{aligned} U_{N-1,N-1}x_{N-1} &= y_{N-1} && (76)\\ U_{N-2,N-1}x_{N-1} + U_{N-2,N-2}x_{N-2} &= y_{N-2} && (77)\\ &\dots \end{aligned}$$

You can see that forward and backward substitution each cost $O(N^2)$ multiply and add operations for each b. If you just wanted to invert the entire matrix, that's N b's (each is a unit vector with one nonzero entry), or a total of $O(N^3)$ operations.

To find the LU decomposition for for your matrix, you need *Crout's* algorithm which solves the N^2 equations represented by Eq. 70 for the $N^2 + N$ unknowns comprising L and U (each has a diagonal). Crout's algorithm starts by setting all the diagonal elements of L to 1, and then solves for the rest of the unknowns by a tricky reordering (again, see Numerical Recipes). The whole procedure is deterministic. Overall the algorithm is still $O(N^3)$.

Thankfully, for a tridiagonal matrix, Crout's algorithm simplifies to $O(N)$ equations and the complete inverse, including forward and back substitution can be done in $O(N)$ steps.

A tridiagonal matrix inverter:

```
/*
 invert a tridiagonal matrix with super (sub) diagonal c (a), and diagonal b.
 rhs is the right hand side and x is the solution.
 From Numerical Recipes
*/

#include <stdlib.h>
#include <stdio.h>

void tridiag(int N, double *a, double *b, double *c, double *rhs, double *x){

  double beta = b[0];
  double *gamma = (double *) malloc (N*sizeof(double));
  x[0] = rhs[0]/beta;
  /* LU decomp and forward sub */
  for(int j=1;j<N;j++){
      gamma[j]=c[j-1]/beta;
      beta=b[j]-a[j]*gamma[j];
      if(beta==0.0){
         printf("Error: j=%d beta=%e\n", j, beta);
         exit(-1);
      }
      x[j]=rhs[j]-a[j]*x[j-1];
      x[j]/=beta;
  }
  /* backward sub */
  for(int j=N-2;j>=0;j--){
      x[j] -= gamma[j+1]*x[j+1];
  }
  free(gamma);
}
```

Now we can solve the diffusion equation with the implicit Crank-Nicholson scheme! Implementing the above code in the implicit scheme is your homework. I have checked that it works by writing my own code to invert the simple tridiagonal matrix resulting from the 1-D diffusion equation, and then multiplying the solution for a given rhs by the matrix itself. This gives back the $N \times N$ identity matrix as it should. Try it! It's always a good check when you implement a new inverter.

My solution using the implicit scheme is shown in Fig. 9. I choose $a_t = 0.1$! and you can see there is only a small deviation from the exact solution at $x = 0$.

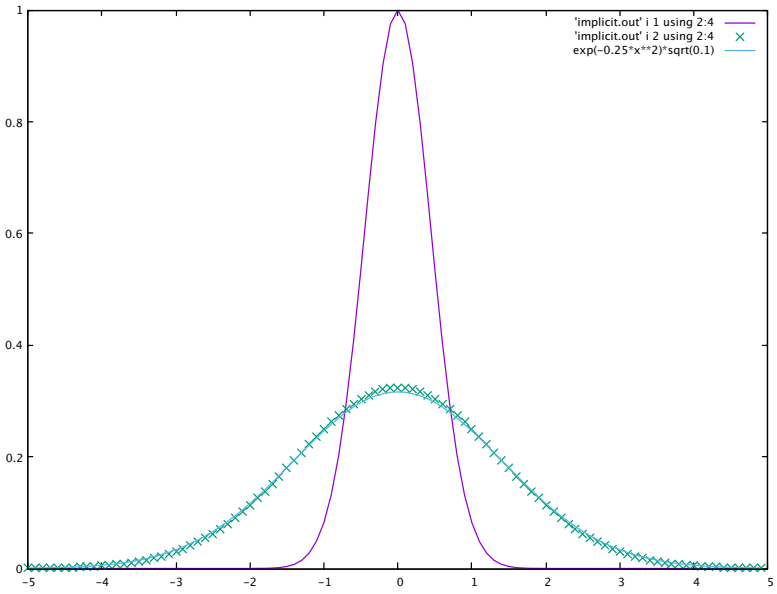

Figure 9: Solution to the 1D diffusion equation using the implicit scheme described in the text. Note that $a_t = 0.1$ compared to 4×10^{-3} for the explicit scheme. The solution at $t = 1.0$ is does not show any hint of instability, as expected.

14 The Diffusion Equation in Two Dimensions

The diffusion equation in 2D reads

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial^2 x} + \frac{\partial^2 T}{\partial^2 y}\right) \tag{78}$$

where of course $T = T(x, y, t)$. It's easy to check that the von Neumann analysis for the explicit method yields

$$Da_t\left(\frac{1}{a_x^2} + \frac{1}{a_y^2}\right) \leq 1/2 \tag{79}$$

as you might expect, or $C \leq 1/4$ if $a_x = a_y$. This is pretty restrictive, so we should try an implicit method.

How does Crank-Nicholson work in this case?

$$\frac{T_i^{n+1} - T_i^n}{a_t} = \frac{D}{2}\left(\delta_x^2 + \delta_y^2\right)\left(T_{ij}^{n+1} + T_{ij}^n\right) \tag{80}$$

$$\delta_x^2 T_{ij} = \frac{T_{i+1,j} + T_{i-1,j} - 2T_{ij}^n}{a_x^2} \tag{81}$$

$$\delta_y^2 T_{ij} = \frac{T_{i,j+1} + T_{i,j-1} - 2T_{ij}^n}{a_y^2} \tag{82}$$

turns out to be $O(a_t^2, a_x^2, a_y^2)$. If we write the set of equations as a matrix, the matrix will no longer be tridiagonal. It is *banded*. It looks like the tridiagonal matrix we had before, but now there are two additional bands for the hopping terms in the y direction. These are displaced, or separated by 0's from the inner bands depending on the size of the lattice in the x direction. Similar bands appear for each new

direction, and the linear size of the matrix grows like $N_x \times N_y \times \ldots$

$$\begin{bmatrix} 1+C & -\frac{C}{2} & 0 & 0 & -\frac{C}{2} & 0 & \ldots & & & 0 \\ -\frac{C}{2} & 1+C & -\frac{C}{2} & 0 & 0 & -\frac{C}{2} & 0 & & & \ldots \\ 0 & -\frac{C}{2} & 1+C & -\frac{C}{2} & 0 & 0 & -\frac{C}{2} & 0 & & \ldots \\ 0 & & -\frac{C}{2} & 1+C & 0 & 0 & 0 & -\frac{C}{2} & 0 & \ldots \\ -\frac{C}{2} & 0 & & 0 & 1+C & -\frac{C}{2} & 0 & 0 & -\frac{C}{2} & \ldots \\ & & & & \vdots & & & & & \\ 0 & \ldots & & & & 0-\frac{C}{2} & 0 & 0 & -\frac{C}{2} & 1+C \end{bmatrix} \begin{bmatrix} T_{1,1}^{n+1} \\ T_{2,1}^{n+1} \\ T_{3,1}^{n+1} \\ T_{4,1}^{n+1} \\ T_{1,2}^{n+1} \\ \vdots \\ T_{4,4}^{n+1} \end{bmatrix} \tag{83}$$

$$= \begin{bmatrix} T_{1,1}^{n} + \frac{C}{2}(T_{0,1}^{n+1} + T_{1,0}^{n+1}) \\ T_{2,1}^{n} + \frac{C}{2}T_{2,0}^{n+1} \\ T_{3,1}^{n} + \frac{C}{2}T_{3,0}^{n+1} \\ T_{4,1}^{n} + \frac{C}{2}(T_{5,1}^{n+1} + T_{4,0}^{n+1}) \\ T_{1,2}^{n} + \frac{C}{2}T_{0,2}^{n+1} \\ \vdots \\ T_{4,4}^{n} + \frac{C}{2}(T_{5,4}^{n+1} + T_{4,5}^{n+1}) \end{bmatrix}$$

This is much harder to invert than the 1D matrix we had before! Typically one resorts to iterative methods. Also notice the right hand side gets a little complicated at the boundaries. Before looking at full blown iterative methods, there is an easier alternative which is described in the next section.

14.1 Alternating Direction Implicit Method

The ADI method goes as the name implies. We start by updating all sites in the x direction, say, implicitly, then update with the y direction implicit. The two step

algorithm looks like this,

$$\frac{T_{ij}^{n+1/2} - T_{ij}^{n}}{a_t/2} = D\left(\delta_x^2 T_{ij}^{n+1/2} + \delta_y^2 T_{ij}^{n}\right) \quad (84)$$

$$\frac{T_{ij}^{n+1} - T_{ij}^{n+1/2}}{a_t/2} = D\left(\delta_x^2 T_{ij}^{n+1/2} + \delta_y^2 T_{ij}^{n+1}\right) \quad (85)$$

or after rearranging (and absorbing $a_x^2 = a_y^2$ into C),

$$T_{ij}^{n+1/2} - \frac{C}{2}\delta_x^2 T_{ij}^{n+1/2} = T_{ij}^{n} + \frac{C}{2}\delta_y^2 T_{ij}^{n} \quad (86)$$

$$T_{ij}^{n+1} - \frac{C}{2}\delta_y^2 T_{ij}^{n+1} = T_{ij}^{n+1/2} + \frac{C}{2}\delta_x^2 T_{ij}^{n+1/2} \quad (87)$$

$$(88)$$

During the 1st step a tridiagonal matrix is inverted for each row of j lattice points, and in the 2nd step for each column of i points. Notice we didn't escape totally unscathed since now we need $O(N_x N_y)$ operations. It turns out that the von Neumann analysis again shows unconditional stability. The above procedure is $O(a_t^2, a_x^2, a_y^2)$ accurate. In Fig. 10 the temperature after a time $t = 1$ is shown.

My code is now in two files, diff2D_implicit.c and tridiag.c. It's useful to split it this way since I can use the tridiag function for any problem that needs to invert a tridiagonal matrix. For example, both 1D and 2D implicit solvers for the diffusion problem! Now my compile statement looks like this:

```
Thomass-iMac:diffusion tblum$ gcc -o diff2D diff2D_implicit.c ../invert/tridiag.c
```

In fact since tridiag shouldn't change, I can compile it once, then link it when needed

```
Thomass-iMac:diffusion tblum$ gcc -c  ../invert/tridiag.c
Thomass-iMac:diffusion tblum$ ls -lt | head -1
total 6320
-rw-r--r--@ 1 tblum  staff    1288 Sep 26 14:13 tridiag.o
Thomass-iMac:diffusion tblum$ gcc -o diff2D diff2D_implicit.c tridiag.o
```

Notice we made a 3d plot in gnuplot! It is just as easy as in 2d:

```
gnuplot> splot 'imp2d.out' i 2 using 2:3:5 w lines title 't=1, t_0=0.1'
```

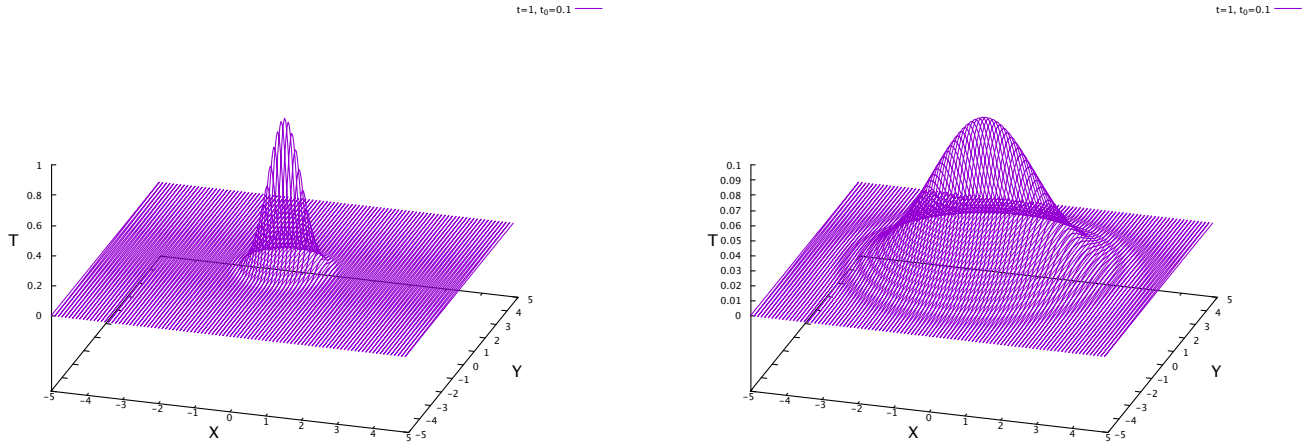

Figure 10: Diffusion in 2D. Initial temperature distribution at $t_0 = 0.1$ (left) and at $t = 1.0$ (right).

15 upgrading to C++

We've already seen some advantages to C++ like type declarations appearing anywhere in our code not just at the top[1] An rigorous disposition on C++ is outside the scope of our course, but let's delve in a bit.

Here's some simple code demonstrating the use of complex numbers:

```
/*
   simple C++ program to demonstrate complex arithmetic
*/

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

[1]By the way, the fact that gcc is now really combined into g++ explains why this worked in nominal C code.

```
#include <stdlib.h>
#include <complex>
#include <vector>
using namespace std;

int main(int argc, char *argv[]){

  complex<double> a(1,1);
  complex<double> b(1.,-1);
  complex<double> c;

  c= a*b;
  printf("This is how C prints c:\n%e %e\n",c.real(),c.imag());
  cout << "This is how C++ prints c:\n" << c << "\n";
  cout << "This is how C++ prints c:\n" << c << endl;
  c= a-b;
  vector<complex<double> > v;
  int N;
  cout << "Enter the size of your vector: ";
  cin >> N;

  v.resize(N);


  for(int i=0;i<N;i++)
    cout << "v[" << i << "]= " << v[i] << endl;
  for(int i=0;i<N;i++)
    cout << "v[" << i << "]= " << v[i].real() << " , " << v[i].imag() << endl;
  return 0;
}
```

There's a lot of useful and/or new stuff here.

- C++ header files isostream, complex, vector
- iostreams
- namespace "std"

- templates
- object classes

First of all, let's begin with the fact that C++ is an *object oriented* programing language. Objects in C++ are called *classes*. Classes have two types of *members*: data and functions. For example the complex class defined in C++ stores the real and imaginary parts and also provides functions to manipulate them. As we see above, the arithmetic ops defined for floating point numbers are also defined for complex. Most objects in C++ are user defined, *i.e.* you make them up. Let's go over the above items, then we will write a simple program to define a wave function class, using what we've learned.

15.0.1 iostreams

C++ uses "iostreams" like C uses printf (and scanf) to get data to and from stdin and out (or files). The stdout is called "cout" and stdin, "cin". That's about all we need for now. When we need file io we'll come back to it.

15.0.2 namespaces

A namespace defines a region, or block, of code where all the type, variable, and function names are defined only within that *scope*. Multiples namespaces are allowed, and switching between them is too. So is using variables and functions defined in one with another. For example in the code below we define two namespaces, block1 and block2, and also use the "std" namespace.

```
/*
   simple C++ program demonstrating namespaces
*/
```

```
#include <iostream>

namespace block1 {
  double a = 1.0;
}
namespace block2 {
  double a = 2.0;
}

using namespace std;

int main(int argc, char *argv[]){

  cout << "in block1 a= " << block1::a << endl;
  cout << "in block2 a= " << block2::a << endl;

  return 0;
}

Thomass-iMac:simple tblum$ g++ namespace.C
Thomass-iMac:simple tblum$ a.out
in block1 a= 1
in block2 a= 2
Thomass-iMac:simple tblum$
```

Notice the use of the "::" scope operator and the namespace std. If the line before main was omitted we would need the following

```
/*
   simple C++ program demonstrating namespaces
*/
#include <iostream>

namespace block1 {
  double a = 1.0;
}
namespace block2 {
  double a = 2.0;
```

```
}

int main(int argc, char *argv[]){

  std::cout << "in block1 a= " << block1::a << std::endl;
  std::cout << "in block2 a= " << block2::a << std::endl;

  return 0;
}

Thomass-iMac:simple tblum$ g++ namespace.C
Thomass-iMac:simple tblum$ a.out
in block1 a= 1
in block2 a= 2
Thomass-iMac:simple tblum$
```

In other words functions are defined in namespaces too. In c++ the *local* or default name space, or scope is defined by curly braces "{}". Sometimes this is called the "unnamed namespace". Sometimes using "using namespace <name>;" is convenient for a large block of code. But you may need a variable or function from another namespace in that block. Then using the scope operator is convenient. Or just use "{}".

```
using namespace std;

int main(int argc, char *argv[]){

  cout << "in block1 a= " << block1::a << endl;
  cout << "in block2 a= " << block2::a << endl;

  {
    double a = 3.0;
    std::cout << "in {} a= " << a << std::endl;
  }
```

```
  return 0;
}

Thomass-iMac:simple tblum$ g++ namespace.C
Thomass-iMac:simple tblum$ a.out
in block1 a= 1
in block2 a= 2
in {} a= 3
```

Later we will see that object classes in C++ also define their own namespaces.

15.0.3 templates

Templates are a way of defining functions that operate on generic types. For example, say you are writing some nifty code for your research project, and you want the flexibility to do calculations in double or single precision. But you're not sure which, and you don't want to write two separate, essentially identical, codes that do the same thing. You can use templates to build in this flexibility from the start. Here is a version of our simple direct sum code using templates that computes the sum using double or single precision.

```
/*
   simple C++ program using templates to demonstrate numerical round off error.
   See http://homerreid.com/teaching/18.330/Notes/MachineArithmetic.pdf
   for a discussion
*/

#include <iostream>
#include <math.h>

#define PI 3.141592653589793

/* function to add a number to itself N times */
```

```
using namespace std;

template <class T>
T directsum(int N, T X){

  int i;
  T sum=0.0;

  for(i=0;i<N;i++){
    sum += X;
  }
  return sum;
}


int main(int argc, char *argv[]){

  int N;
  int Nmax;
  int Ninc;

  if(argc != 3){
    printf("Wrong # of args: Nmax Ninc\n");
    exit(-1);
  }

  Nmax = atoi(argv[1]);
  Ninc = atoi(argv[2]);

  for(N=0;N<Nmax;N+=Ninc){

    if(N){
      {
         float X = directsum<float>(N,PI/(float)N);
         X = fabs(X-PI)/PI;
         cout << "error(float)= " << N << X;
      }
      {
```

```
        double X = directsum<double>(N,PI/(double)N);
        X = fabs(X-PI)/PI;
        cout << " error(double)= " << X;
      }
    }
  }

  return 0;
}
Thomass-iMac:simple tblum$ g++ simple_template.C
Thomass-iMac:simple tblum$ a.out 100 10
error(float)= 101.03719e-07 error(double)= 0
error(float)= 202.55501e-07 error(double)= 0
error(float)= 303.51627e-07 error(double)= 2.82716e-16
error(float)= 403.31392e-07 error(double)= 0
error(float)= 503.31392e-07 error(double)= 8.48148e-16
error(float)= 607.31082e-07 error(double)= 7.0679e-16
error(float)= 702.55501e-07 error(double)= 0
error(float)= 803.31392e-07 error(double)= 1.97901e-15
error(float)= 907.10846e-07 error(double)= 1.13086e-15
```

1. the line "template <class T> always precedes the definition of the templated function
2. note the "T" declaration of the 2nd argument to directsum
3. in main, the templated function is call with a specific instance of the desired type (either float or double)

We can define templates for templated variables like complex too!

```
/*
   simple C++ program using templates to demonstrate numerical round off error.
   See http://homerreid.com/teaching/18.330/Notes/MachineArithmetic.pdf
   for a discussion
*/
```

```
#include <iostream>
#include <math.h>
#include <complex>

//#define PI 3.141593
#define PI 3.141592653589793

/* function to add a number to itself N times */

using namespace std;

template <class T>
T directsum(int N, T X){

  int i;
  T sum=0.0;

  for(i=0;i<N;i++){
    sum += X;
  }
  return sum;
}


int main(int argc, char *argv[]){

  int N;
  int Nmax;
  int Ninc;

  if(argc != 3){
    printf("Wrong # of args: Nmax Ninc\n");
    exit(-1);
  }

  Nmax = atoi(argv[1]);
  Ninc = atoi(argv[2]);
```

```
  for(N=0;N<Nmax;N+=Ninc){

    if(N){
      {
         float X = directsum<float>(N,PI/(float)N);
         X = fabs(X-PI)/PI;
         cout << "error(float)= " << N << X;
      }
      {
         double X = directsum<double>(N,PI/(double)N);
         X = fabs(X-PI)/PI;
         cout << " error(double)= " << X;
      }
      {
         complex<double> X = directsum<complex<double> >(N,(complex<double>)(PI/(doub
         X = abs(X-PI)/PI;
         cout << " error(complex)= " << X << endl;
      }
    }
  }

  return 0;
}
Thomass-iMac:simple tblum$ g++ simple_template.C
Thomass-iMac:simple tblum$ a.out 100 10
error(float)= 101.03719e-07 error(double)= 0 error(complex)= (0,0)
error(float)= 202.55501e-07 error(double)= 0 error(complex)= (0,0)
error(float)= 303.51627e-07 error(double)= 2.82716e-16 error(complex)= (2.82716e-16,0
error(float)= 403.31392e-07 error(double)= 0 error(complex)= (0,0)
error(float)= 503.31392e-07 error(double)= 8.48148e-16 error(complex)= (8.48148e-16,0
error(float)= 607.31082e-07 error(double)= 7.0679e-16 error(complex)= (7.0679e-16,0)
error(float)= 702.55501e-07 error(double)= 0 error(complex)= (0,0)
error(float)= 803.31392e-07 error(double)= 1.97901e-15 error(complex)= (1.97901e-15,0
error(float)= 907.10846e-07 error(double)= 1.13086e-15 error(complex)= (1.13086e-15,0
```

Templates are very powerful.

15.0.4 object classes

This is perhaps the primary reason to extend C to C++. Objects are also very powerful constructs, and the power is compounded by templating. You can think of a class object as a fancy type that not only holds data, but also has function members that manipulate that data. Let's begin by writing a class to define the wave function (solution to Schrödinger equation).

```
/*
   simple C++ program to define a wave function class and demonstrate its use
*/

#include <complex>
#include <vector>

class WaveFunction {

private:
  std::vector <std::complex<double> > v;
public:
  // constructor (for type declaration, e.g.)
  WaveFunction(){};
  WaveFunction(std::vector <std::complex<double> >);
  // take the norm of wave function
  double norm();
  // compute a dot product with another vector
  std::complex<double> dot_product(std::vector<std::complex<double> > a);
  // complex conjugate of wave function
  std::vector<std::complex<double> > conjugate();
  // equal operator
  void operator=(std::vector<std::complex<double > > a);
  // complex dot product operator for wave functions
  std::complex<double> operator*(const WaveFunction &b);
};

//constructor
WaveFunction::WaveFunction(std::vector <std::complex<double> > w){
```

```
  v.resize(w.size());
  for(int i=0;i<v.size();i++){
    v[i] = w[i];
  }
};
// accessor
std::vector <std::complex<double> > WaveFunction::vector(){

  return v;

};
// compute the norm of the wave function, v^* . v
double WaveFunction::norm(){

  double norm=0.0;
  for(int i=0;i<v.size();i++){
    norm+=std::norm(v[i]);
  }

  return sqrt(norm);

};

// return the complex conjugate of the wave function
std::vector<std::complex<double> > WaveFunction::conjugate(){

  std::vector<std::complex<double> > temp;
  temp.resize(v.size());
  for(int i=0;i<v.size();i++){
    temp[i]=std::conj(v[i]);
  }

  return temp;

};

// return the dot product v*.a
std::complex<double> WaveFunction::dot_product(std::vector<std::complex<double> > a){
```

```
  std::complex<double> sum(0.0,0.0);
  for(int i=0;i<v.size();i++){
    sum += std::conj(v[i])*a[i];
  }
  return sum;
}

// = operator
void WaveFunction::operator=(std::vector<std::complex<double > > a){

  v.resize(a.size());
  for(int i=0;i<v.size();i++){
    v[i]=a[i];
  }
};
// complex dot product
std::complex<double> WaveFunction::operator * (const WaveFunction &b){
  int n = this->v.size();
  int m = b.v.size();
  if(n!=m){
    std::cout<< "error! a and b have different lengths" << std::endl;
    exit(-1);
  }
  std::complex<double> c(0.0);
  c = this->dot_product(b.v);
  return c;
};

using namespace std;

# include <iostream>
# include <numeric>

int main(int argc, char *argv[]){

  WaveFunction psi;
  vector<complex<double > > phi;
```

```
  phi.resize(10);
  for(int i=0;i<10;i++){
    phi[i]=complex<double>(1,1);
  }

  psi = phi;

  double norm = psi.norm();
  cout << "norm of psi " << norm << endl;

  complex<double> x = psi.dot_product(phi);
  cout << "psi.phi= " << x << endl;
  cout << "psi.phi using * operator= " << psi * phi << endl;

  cout << "vector has it's own inner product: phi.phi= " << std::inner_product(phi.be

  WaveFunction chi(phi);
  x = chi.dot_product(phi);
  cout<< "use the other constructor to init wave func" << endl;
  cout << "chi.phi= " << x << endl;

  //access the data member of wavefunction
  std::vector<complex<double> > temp = chi.conjugate();
  complex<double> c(0.,0.);
  for(int i=0;i<temp.size();i++)
    c+= temp[i]*temp[i];
  cout << "c= temp . temp " << c << endl;

  return 0.0;
}
Thomass-iMac:Schrodinger tblum$ g++ wave_function.C
Thomass-iMac:Schrodinger tblum$ a.out
norm of psi 4.47214
psi.phi= (20,0)
vector has it's own inner product: phi.phi= (0,20)
chi.phi= (20,0)
Thomass-iMac:Schrodinger tblum$ g++ wave_function.C
Thomass-iMac:Schrodinger tblum$ a.out
```

```
norm of psi 4.47214
psi.phi= (20,0)
vector has it's own inner product: phi.phi= (0,20)
use the other constructor to init wave func
chi.phi= (20,0)
```

There's a lot happening here! Notice the "class" definition statement. It says what members are in the class. These can be "private", "public", or protected. The first means only accessible within the class. public means any function can access it, and protected is only for "friends" (as well as the class itself). Since most data members are private, one typically has to make an "accessor" function(s) to propagate the data outside of the class.

Notice that in C++ we can use the same name for different functions. This is called *overloading*. For example, in the above two constructors are given that are both named WaveFunction(...) but have different argument lists.

You will have to "complexify" our simple tridiagonal matrix inverter to use in the implicit solution of the Schrödinger equation. You can do it by changing (almost) all of the doubles to complex<double>'s and changing the 1d arrays to vector's. Note you will not need malloc here! C++ has done it for you. You don't need to use classes either. But eventually they will be needed and/or will be convenient.

Messages of the day, Monday Oct 8 2018

- New hw on line due in 2 weeks!
- Going forward, start working on good habits, style (*i.e.*, comments, proper indentation, no global variables, proper functions, main, etc)
- do not take code from sources other than these notes or that provided by me (you can work together, but you must produce your own code).

- Come see me if you get stuck! (start early!)
- Great colloquium on Friday– condensed matter and climate physics (actually topological similarities)
- Hike Monadnock on Saturday!

15.0.5 passing by reference in C++ and C

C++ introduces a new method of passing by reference. Recall, in C, passing by reference is done through pointers. This is a powerful tool as we have seen, but sometimes it is (more than) a bit complicated and opaque. C++ addresses this with a new type called a reference which accomplishes the same result while treating the variables more like ordinary types.

In C++ we can use the "&" reference operator to pass a variable (*not* to be confused with the "&" unary address operator!).

```
/*
   reference.C C++ program to demonstrate
   the difference between pointers and references
*/
#include <iostream>

void val(int *a)
{
  *a = 6;
};
void val(int &a)
{
  a=6;
};

using namespace std;
```

```
int main(){

  int i;
  int *ip;

  i=7;
  cout << "inital val of i= " << i << endl;
  val(&i);
  cout << "use pointer to pass/change val of i= " << i << endl;
  val(i);
  cout << "use reference to pass/change val of i= " << i << endl;

  return 0;
}

Thomass-iMac:pointer tblum$ g++ reference.C
Thomass-iMac:pointer tblum$ a.out
inital val of i= 7
use pointer to pass/change val of i= 6
use reference to pass/change val of i= 6
```

We can have pointers to class objects. Then how do we access the data and function members? Let's make another simple example to show some possibilities.

```
/*
   reference.C: C++ program to demonstrate
   using pointers to objects
*/
#include <iostream>

class val
{
private:
  int a;
public:
  val(int);
  int access();
```

```
  void operator=(val&);
};

// constructor
val::val(int b){a=b;};
// accesor
int val::access(){return a;};
// copy constructor
void val::operator=(val& b){a=b.access();};

using namespace std;

int main(){

  int i=7;

  val v(i);
  cout << "object, i= " << v.access() << endl;
  val *pv=&v;
  cout << "pointer to object, i= " << pv->access() << endl;

  val v2=v;
  cout << "reference to object, i= " << v2.access() << endl;


  return 0;
}

Thomass-MacBook:pointer tblum$ g++ reference2.C
Thomass-MacBook:pointer tblum$ a.out
object, i= 7
pointer to object, i= 7
reference to object, i= 7
```

So the effect is the same for pointers and references, and in fact C++ is passing a pointer to the variable in both cases. The differences are summarized by the

following.

- A pointer can be reassigned while a reference cannot, and must be assigned at initialization only
- A pointer can be assigned a value of NULL, whereas a reference cannot
- Pointers can be iterated over an array: we can use ++ to increment the pointer to the next address in the array
- A pointer is a variable whose value is an address. A reference has the same memory address as the item it references
- A pointer to a class/struct object uses '$->$' (arrow operator) to access it's members whereas a reference uses a '.' (dot operator) like any regular class object.
- A pointer needs to be dereferenced with the * operator to access the value of the variable to which it points, whereas a reference can be used directly like an ordinary variable.

So a reference combines the convenience of ordinary variables with (some of) the power of pointers.

15.0.6 inheritance and derived classes

Not now.

16 The Schrödinger Equation

With some pretty powerful tools at our disposal already, we now turn to solution of the Schrödinger equation which is almost identical to the diffusion equation except for a factor of "i" (adding a source term to the diffusion equation is like the potential).

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = H\Psi(x,t) \tag{89}$$

$$= -\frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2} + V(x)\Psi(x,t). \tag{90}$$

$\Psi(x,t)$ is of course the wave function which is in general *complex*. Rather than write our own library functions to deal with complex numbers and arithmetic, let's upgrade to C++ to take advantage of the stdlib's type definitions. See last section.

16.1 Numerical solution of Schrödinger's Equation

There are many ways and a huge literature on how to do this! We are going to stick with our finite difference methods for now, but there are other methods which we may introduce later. In particular we will use some ADI-based methods developed recently. See here and here. The second paper looks really interesting: it discusses the solution of the non-linear Schrödinger Eq. which describes the dynamics of a generalized Bose-Einstein condensate (BEC) which we will explore in some detail. Another good reference which predates these but gives a lot more explanation on the physics and many more numerical techniques is here. You should have access to these if you are on the UConn internet or set up a UConn VPN from home.

Let's look at the three references linked above. Think of this lecture as a journal club.

16.1.1 nonlinear Schrödinger equation

The nonlinear Schrödinger equation is called the Gross-Pitaevskii equation (GPE) after the two physicists who derived it independently. As mentioned above, the physics of the BEC (GPE) is discussed at length in here along with several numerical methods for it's solution.

We will study section 2, which deals with confined BEC's in 1, 2 or 3 dimensions at zero or very low temperature, in the dilute gas limit. The GPE is valid in this case. In fact the solution to the GPE is nothing but the time dependent order-parameter, or mean field, describing an inhomogeneous BEC.

For the imaginary time GPE, we'll use this as our source. It will provide information for the next home work assignment. Notice that here they use an explict algorithm to solve the GPE in imaginary time. We'll use our implicit solver.

A brief explanation of many-body physics formalism can be found here.

16.1.2 2d linear example

In Fig. 11 I show the numerical results from my code (uploaded to Husky CT) for the ADI method corresponding to example 1 in the Xu and Zhang paper. They agree well and also agree well with the exact solution. The steps to plot in gnuplot:

```
gnupolt> set palette defined ( 0 '#000090', 1 '#000fff', 2 '#0090ff',3 '#0fffee',\
             4 '#90ff70',5 '#ffee00',6 '#ff7000',7 '#ee0000',8 '#7f0000')
gnuplot> set size square
gnuplot> set view map
gnuplot> splot 'out_t0.25' i 1 using 3:4:5 with image
```

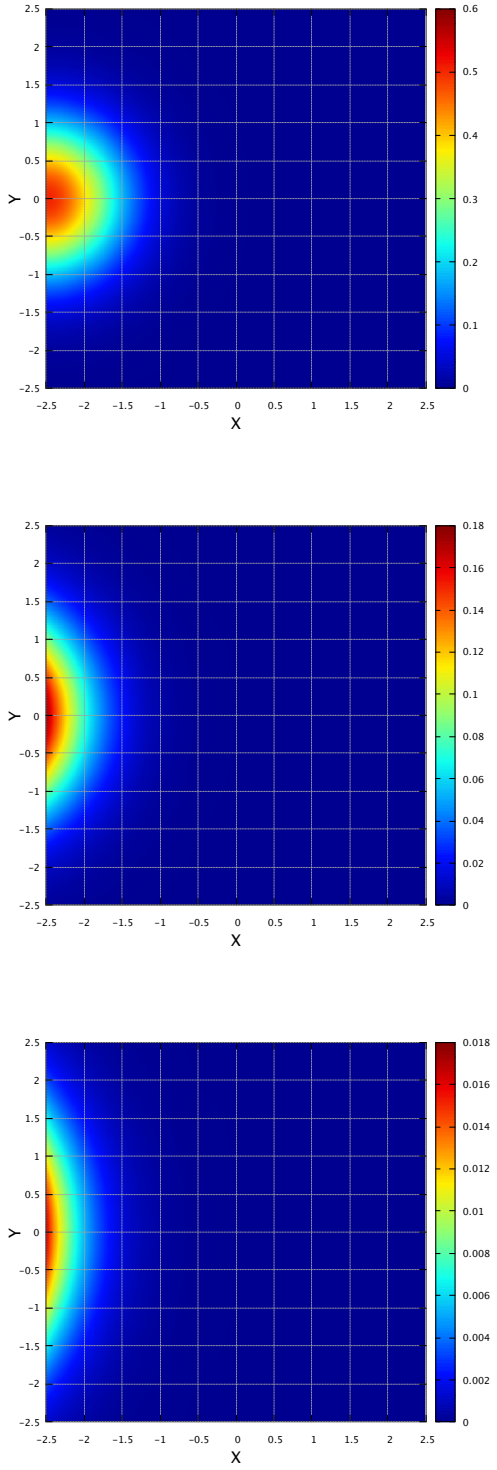

Figure 11: ADI solution of 2D Schrödinger equation for initial conditions and boundary conditions described in the text. Top to bottom, $t = 0.25$, 0.35, and 0.5.

16.1.3 nonlinear example

Let's look at the imaginary time problem in more detail. We start with Eq. (4) in the referred to paper

$$i\hbar\frac{\partial\psi(z,t)}{\partial t} = \left(-\frac{\hbar^2}{2M}\frac{\partial^2}{\partial z^2} + u_{\rm ext}(z) + u_I|\psi(z,t)|^2\right)\psi(z,t), \tag{91}$$

and let $t \to \tau = it$

$$\hbar\frac{\partial\psi(z,\tau)}{\partial\tau} = -\left(-\frac{\hbar^2}{2M}\frac{\partial^2}{\partial z^2} + u_{\rm ext}(z) + u_I|\psi(z,\tau)|^2\right)\psi(z,\tau). \tag{92}$$

First, let's get this into dimensionless form by dividing both sides by $\hbar\omega$,

$$\frac{\partial\psi(z,\tau)}{\partial\omega\tau} = -\left(-\frac{\hbar}{2M\omega}\frac{\partial^2}{\partial z^2} + \frac{u_{\rm ext}(z)}{\hbar\omega} + \frac{u_I}{\hbar\omega}|\psi(z,\tau)|^2\right)\psi(z,\tau) \tag{93}$$

or (94)

$$\frac{\partial\psi(\tilde{z},\tilde{\tau})}{\partial\tilde{\tau}} = -\left(-\frac{\partial^2}{\partial\tilde{z}^2} + u_{\rm ext}(\tilde{z}) + \tilde{u}_I|\psi(\tilde{z},\tilde{\tau})|^2\right)\psi(\tilde{z},\tilde{\tau}), \tag{95}$$

where

$$\tilde{\tau} \equiv \omega\tau, \tag{96}$$

$$\tilde{z} \equiv \sqrt{\frac{2M\omega}{\hbar}}z, \tag{97}$$

$$\tilde{a} = \gamma a = \left(\frac{5}{2\pi}\right)^{3/5}\frac{a}{(24\pi Na)^{2/5}}\left(\frac{2M\omega}{\hbar}\right)^{4/5}, \tag{98}$$

$$\tilde{u}_I = \frac{4\pi\hbar\tilde{a}N}{M\omega} \tag{99}$$

$$= (10aN)^{3/5}\left(\frac{2}{3}\sqrt{\frac{\hbar}{M\omega}}\right)^{2/5} \tag{100}$$

where a is the scattering length, N is the number of atoms in the trap, M is the mass of each atom, and ω is the frequency of the harmonic trap.

$$u_{\rm ext}(\tilde{z}) = \frac{M\omega^2}{2}z^2\frac{1}{\hbar\omega} = \frac{\tilde{z}^2}{4}. \tag{101}$$

Notice that M and ω completely drop everywhere except in $\tilde{u}_I$ which has dimensions of length. That's consistent with $|\psi|^2$ having dimensions of inverse length (probability density in 1d). So, we should modify the equation one more time to make the wave function dimensionless as well.

$$\psi(\tilde{z},\tilde{\tau}) \rightarrow \left(\frac{\hbar}{2M\omega}\right)^{1/4}\psi(\tilde{z},\tilde{t}) = \tilde{\psi}(\tilde{z},\tilde{\tau}) \tag{102}$$

$$\frac{\partial\tilde{\psi}(\tilde{z},\tilde{\tau})}{\partial\tilde{\tau}} = -\left(-\frac{\partial^2}{\partial\tilde{z}^2} + u_{\text{ext}}(\tilde{z}) + \tilde{u}_I\sqrt{\frac{2M\omega}{\hbar}}|\tilde{\psi}(\tilde{z},\tilde{\tau})|^2\right)\tilde{\psi}(\tilde{z},\tilde{\tau}), \tag{103}$$

Let's take the same set up and parameters as used in the paper: ^{87}Rb atoms with scattering length $a = 110$ Bohr radii and $N = 10^4$ to 10^6. I had to "tune" the trap frequency $\omega = 43.0515\ s^{-1}$ to find results close to those in the paper.

The initial condition is taken as a gaussian,

$$\psi(z,\tau) \propto \exp -M\omega z^2/4\hbar \tag{104}$$

$$= \exp -\tilde{z}^2/8 \tag{105}$$

In Figs. 12-14 the results from my code are shown (see HuskyCT).

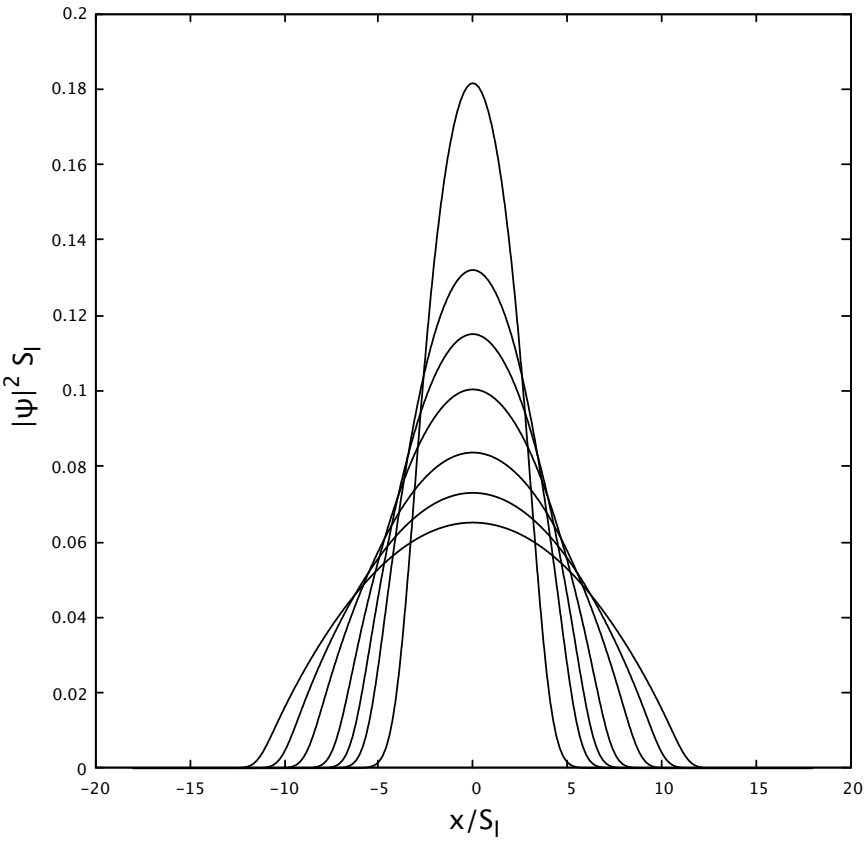

Figure 12: Density profiles for ground state BEC's for (top to bottom) $N = 10^4$, 5×10^4, 1×10^5, 2×10^5, 5×10^5, 1×10^6, and 2×10^6 trapped atoms.

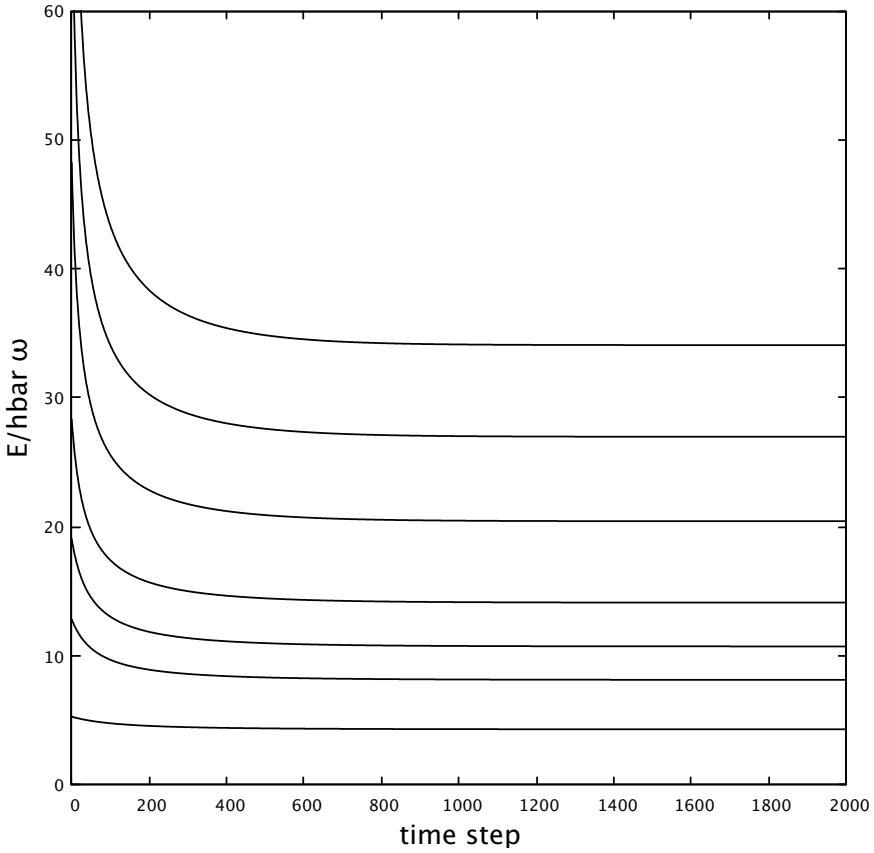

Figure 13: BEC ground state energy from imaginary time integration (bottom to top) for $N = 10^4$, 5×10^4, 1×10^5, 2×10^5, 5×10^5, 1×10^6, and 2×10^6 trapped atoms. Total time is 2.0.

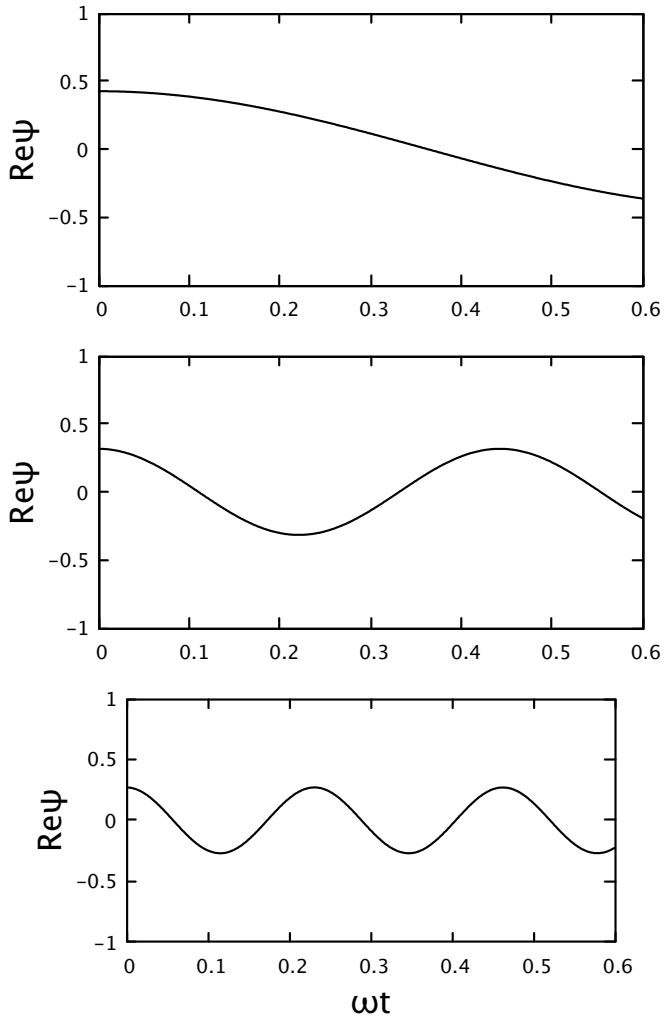

Figure 14: BEC real time evolution of the ground state (top to bottom) for $N = 10^4$, 2×10^5, and 1×10^6 trapped atoms.

17 Krylov Subspace Methods

These encompass a broad class of *iterative* methods that are very popular (and efficient) for large sparse matrices. The central characteristic of such methods is the vector space constructed by $n-1$ successive applications of the matrix to an arbitrary vector. The resulting space is a so-called n-dimensional Krylov subspace. I will follow chapter 6 in the book by Saad and various wikipedia pages. The two important examples that we will study are the Lanczos algorithm to diagonalize the matrix and the conjugate gradient algorithm to determine it's inverse.

A general *projection* method seeks an approximate solution in a subspace $x_0+\mathcal{K}_m$ of dimension m, x_m, to the linear system $Ax=b$ by imposing (the Petrov-Galerkin) condition

$$b - Ax_m \perp \mathcal{L}_m \tag{106}$$

where $\mathcal{L}_m$ is another subspace of dimension m. x_0 is a starting guess. A Krylov space method is one where $\mathcal{K}_m \in \text{span}\{r_0, Ar_0, A^2r_0 \dots A^{m-1}r_0\}$ where $r_0 = b - Ax_0$. r is sometimes called the *residual* vector. The basic idea is that as $m \gg 1$, $r_m \to 0$ and $x_m \to x$. In other words we seek a solution in a subspace spanned by polynomials of A, or $A^{-1}b$ is approximated by $p(A)b$.

Krylov methods are those for which $\mathcal{L}_m = \mathcal{K}_m$.

17.1 properties of Krylov spaces

See section 6.2 of Saad's book for a detailed discussion. A Krylov subspace is defined as

$$\mathcal{K}_m(A, v) \equiv \text{span}\left\{v, Av, A^2v, \dots, A^{m-1}v\right\} \tag{107}$$

where A is matrix and v is arbitrary. $\mathcal{K}_m$ is a subspace of all vectors in $\mathbb{R}^n$ (or $\mathbb{C}^n$ if v is complex) that can be written as a polynomial of degree $m-1$ or less, $x = p(A)v$. The dimension of $\mathcal{K}_m$ is m iff the *grade* of v with respect to A is not less than m. The grade of v with respect to A is the degree of the minimal monic polynomial such that $p(A)v = 0$.

In the following we discuss iterative methods that build up the Krylov space basis by successive applications of the matrix of interest to a starting vector.

17.2 Lanczos algorithm

The algorithm is designed to transform an $n \times n$ Hermitian matrix A into a real, symmetric tridiagonal one (T) which can then be diagonalized. The eigenvalues and eigenvectors of T are the same as the original matrix since the Lanczos algorithm effects a unitary transformation on A.

$$A = VTV^\dagger \tag{108}$$

The Lanczos algorithm is very efficient for large, sparse matrices like the ones that often show up in physically interesting and/or practical applications. In practice however, the matrices are very large, so that only a small number of eigenvalues/vectors can be computed. Many times this is enough. However, in this case the eigensystems of T are only approximately equal to the ones of A.

Let's say that we can afford to compute $m \ll n$ eigenvectors of A (both compute resources and memory are a concern). Then the Lanczos algorithm is the following.

1. let $v_1 \in \mathbb{C}^n$ with norm $||v_1|| = 1$. v_1 is an arbitrary vector

2. first step

(a) $w_1' = Av_1$

(b) $\alpha_1 = w_1'^{\dagger} \cdot v_1$ (projection in direction v_1)

(c) $w_1 = w_1' - \alpha_1 v_1$ (orthogonal to v_1)

3. For $j = 2$ to m do:

(a) $\beta_j = ||w_{j-1}||$

(b) $v_j = w_{j-1}/\beta_j$

(c) $w_j' = Av_j$

(d) $\alpha_j = w'^{\dagger} \cdot v_j$

(e) $w_j = w_j' - \alpha_j v_j - \beta_j v_{j-1}$

Combining 3.b, 3.c and 3.e we find

$$Av_j = w_j + \alpha_j v_j + \beta_j v_{j-1} \tag{109}$$

$$= \beta_{j+1} v_{j+1} + \alpha_j v_j + \beta_j v_{j-1} \tag{110}$$

which is nothing but the form of a symmetric tridiagonal matrix times a vector for $1 < j < m$.

Now we have many algorithms at our disposal to diagonalize the tridiagonal matrix. As already mentioned, if $m = n$, then eigenvalues of T are also eigenvalues of A with eigenvectors Vx.

$$Tx = \lambda x \tag{111}$$

$$Ay = AVx \tag{112}$$

$$= VTV^{\dagger}Vx \tag{113}$$

$$= VTx = \lambda Vx = \lambda y \tag{114}$$

Even if $m \ll n$ we can still obtain the (largest) eigenvalues/vectors of A from T. Note that in this case V is an $n \times m$ matrix (not unitary). We'll return to diagonalizing T after we derive the above algorithm.

17.2.1 power method

A derivation of the Lanczos algorithm begins with a simpler method for finding the (largest) eigenvalues of A (which need not be Hermitian) called the power method which is essentially just repeated application of A to a random vector.

Pick a random vector and write it in the eigenvector basis of A ($\lambda_1 > \lambda_2 > \cdots > \lambda_n$,

$$u_1 = c_1|1\rangle + c_2|2\rangle + \cdots + c_n|n\rangle \tag{115}$$

A can be written in a spectral decomposition,

$$A = UDU^\dagger = \sum_i \lambda_i |i\rangle\langle i| \tag{116}$$

where U is the unitary matrix whose columns are the eigenvectors of A and D is the corresponding diagonal matrix whose elements are the eigenvalues. Assuming $c_1 \neq 0$,

$$\begin{aligned} Au_1 &= \sum_i \lambda_i |i\rangle\langle i|(c_1|1\rangle + c_2|2\rangle + \cdots + c_n|n\rangle) && (117)\\ &= c_1\lambda_1|1\rangle + c_2\lambda_2|2\rangle + \cdots + c_n\lambda_n|n\rangle && (118)\\ &= \lambda_1\left(c_1|1\rangle + c_2\frac{\lambda_2}{\lambda_1}|2\rangle + \cdots + c_n\frac{\lambda_n}{\lambda_1}|n\rangle\right) && (119) \end{aligned}$$

since $\lambda_1 > \lambda_2 > \ldots$, after many applications,

$$A^j u_1 \approx \lambda_1^j c_1|1\rangle \tag{120}$$

which projects out the largest eigenvalue. This leads to the following algorithm for the power method,

1. pick a random vector u_1 with norm $||u_1|| = 1$

2. For $j = 1$ to m do

   (a) $u'_{j+1} = Au_j$

   (b) $u_{j+1} = u'_{j+1}/||u'_{j+1}||$

In the large j limit u_j is the normalized eigenvector of A corresponding to the largest eigenvalue.

This algorithm is not very efficient. Indeed, usually one overwrites u_j with u_{j+1}. Lot's of info is lost in the process. Does the iteration remind you of something? Krylov subspace! The sequence of u_j's is almost the basis of a Krylov subspace. The u's themselves can't serve this purpose well since they are not orthogonal (by design). But we can easily make them so by adding a Gram-Schmidt orthogonalization step in our algorithm,

1. Start with random u_1, $||u_1|| = 1$, $v_1 = u_1$

2. For $j = 1, \ldots, m-1$ do

   (a) $u'_{j+1} = Au_j$

   (b) For $k = 1, \ldots, j$ $g_{kj} = v_j^\dagger \cdot u'_{j+1}$ (Gram-Schmidt)

   (c) $w_{j+1} = u'_{j+1} - \sum_{k=1}^{j} g_{kj} v_k$ (Gram-Schmidt)

   (d) $u_{j+1} = u'_{j+1}/||u'_{j+1}||$

   (e) $v_{j+1} = w_{j+1}/||w_{j+1}||$

Using equations in items 3, 5, and 7, we see

$$Au_j = w_{j+1} + \sum_{k=1}^{j} g_{kj} v_k \tag{121}$$

$$= ||w_{j+1}|| v_{j+1} + \sum_{k=1}^{j} g_{kj} v_k \tag{122}$$

which tells us how to get the v's from the u's. But a little more thought reveals we don't need the u's at all to get the v's!

The difference $u_j - v_j \in \text{span}\{v_1, \cdots, v_{j-1}\}$, so the difference $u'_{j+1} - w'_{j+1} = A(u_j - v_j) \in \text{span}\{v_1, \cdots, v_j\}$ is just the Gram-Schmidt orthogonalization (step 2.c). In other words the power method plus GS just gives the Krylov subspace spanned by the v's. Nothing is lost by changing basis. So we should omit the u vectors completely:

1. Start with random v_1, $||v_1|| = 1$
2. For $j = 1, \ldots, m-1$ do
   (a) $w'_{j+1} = Av_j$
   (b) For $k = 1, \ldots, j$ $h_{kj} = v_k^\dagger \cdot w'_{j+1}$
   (c) $w_{j+1} = w'_{j+1} - \sum_{k=1}^{j} h_{kj} v_k$ (Gram-Schmidt)
   (d) $h_{j+1,j} = ||w_{j+1}||$
   (e) $v_{j+1} = w_{j+1}/h_{j+1,j}$

From 2.a and 2.c-e we see that $Av_j = \sum_{k=1}^{j+1} h_{kj} v_k$ for all $j < m$ and

$$v_{j+1}^\dagger w'_{j+1} = v_{j+1}^\dagger \left(w_{j+1} + \sum_{k=1}^{j} h_{kj} v_k \right) \tag{123}$$

$$= v_{j+1}^\dagger w_{j+1} = h_{j+1,j} w_{j+1}^\dagger w_{j+1} \tag{124}$$

$$= h_{j+1,j} = ||w_{j+1}|| \tag{125}$$

as expected from 2.b and 2.d.

The final algorithm is called the Arnoldi procedure, or iteration. If we now specialize to a Hermitian matrix, then

$$\begin{aligned} h_{k,j} &= v_k^\dagger \cdot w'_{j+1} = v_k^\dagger A v_j = v_k^\dagger A^\dagger v_j && (126) \\ &= (Av_k)^\dagger \cdot v_j && (127) \end{aligned}$$

For $k < j-1$ $Av_k \in \text{span}\{v1, \ldots, v_{j-1}\}$ and by construction v_j is orthogonal, so $h_{k,j} = 0$. For $k = j-1$,

$$\begin{aligned} h_{j-1,j} &= (Av_{j-1})^\dagger \cdot v_j && (128) \\ &= (v_j^\dagger \cdot Av_{j-1})^* = h_{j,j-1}^* = h_{j,j-1}. && (129) \end{aligned}$$

The last equality holds since h is the norm of a vector and therefore is real. Finally,

$$\begin{aligned} h_{j,j} &= (Av_j)^\dagger \cdot v_j && (130) \\ &= (v_j^\dagger \cdot Av_j)^* = h_{j,j}^* && (131) \end{aligned}$$

is also real.

Now, take the columns of V to be the vectors $v_1, \cdot, v_m$. Then the $h_{k,j}$ are the elements of the matrix $H = V^\dagger A V$ with $h_{k,j} = 0$ for $k > j+1$. This is called an *upper Hessenberg* matrix. An upper Hessenberg matrix has one subdiagonal with all other elements below it equal zero. But H is Hermitian which implies it is also lower Hessenberg. So H is a real, symmetric, tridiagonal matrix! just like we set out to show. The Arnoldi procedure on a Hermitian matrix is the Lanczos algorithm.

17.2.2 convergence of the Lanczos procedure

After m iterations of the Lanczos procedure A is a real, symmetric, tridiagonal matrix with eigenvalues $\theta_1 > \theta_2 > \cdots > \theta_m$. How close are they to the eigenvalues λ of the

original matrix A? It turns out the converge is usually faster than the power method by several orders of magnitude. Without going into all the details, the convergence for λ_1 is

$$\lambda_1 - \theta_1 \leq \#\frac{(\lambda_1 - \lambda_n)}{R^{2(m-1)}} \tag{132}$$

$$R = 1 + 2\rho + 2\sqrt{\rho^2 + \rho} \tag{133}$$

$$\rho = \frac{\lambda_1 - \lambda_2}{\lambda_1 - \lambda_n} \tag{134}$$

For $\rho \gg 1$ this is about $(1 + 4\rho/1 + 2\rho)^2$ faster than the power method. For $\rho << 1$ the speed up is much larger.

17.2.3 stability

The above discussion was for exact arithmetic. Because of roundoff error the Gram-Schmidt step may lose accuracy and orthogonality will be lost. To avoid this in practice, a modified Gram-Schmidt procedure should be used in the Arnoldi procedure,

1. Start with random v_1, $||v_1|| = 1$
2. For $j = 1, \ldots, m-1$ do
   (a) $w'_{j+1} = Av_j$
   (b) For $k = 1, \ldots, j$ do (Gram-Schmidt)
      i. $h_{kj} = v_k^\dagger \cdot w'_{j+1}$
      ii. $w_{j+1} = w'_{j+1} - h_{kj}v_k$
   (c) $h_{j+1,j} = ||w_{j+1}||$
   (d) $v_{j+1} = w_{j+1}/h_{j+1,j}$

This leads to our final version of the Lanczos algorithm

1. Start with random $q_1 = v/||v||$, $\beta_0 = 0$, $q_0 = 0$
2. For $j = 1, \ldots, k$ do
3. $w = Aq_j$
4. $\alpha_j = q_j^\dagger \cdot w$
5. $w = w - \alpha_j q_j - \beta_{j-1} q_{j-1}$
6. full Gram-Schmidt, w on q_i $(i < j)$
7. $\beta_j = ||w||$
8. $q_{j+1} = w/\beta_j$

where Gram-Schmidt is

$$w \quad = \quad w - \sum_{i=1}^{j-1} (w^\dagger \cdot q_i) q_i \tag{135}$$

finally, compute eigenvalues of the tridiagonal matrix.

17.2.4 diagonalization of a tridiagonal matrix

Normally I'd say check out Numerical Recipes, but as we'll see I ran into some difficulty. There are many ways to do this. The factorization algorithms QR and QL are popular, efficient methods. In fact they can be combined with Lanczos to produce an "implicitly restarted" lanczos algorithm that is efficient for large sparse matrices where many eigenvalues and eigenvectors are needed. Added bells and whistles include shifts and polynomials to handle (nearly) degenerate eigenvalues.

The following discussion follows Numerical Recipes 2nd edition, sectioin 11.3. Start by assuming a factorized form

$$A = QR \tag{136}$$

where Q is an orthogonal (unitary) matrix and R is upper triangular (construct both via a Householder transformation). (R means the right hand side of the matrix is non-zero). Next define

$$A' = RQ \tag{137}$$

$$= Q^\dagger A Q \tag{138}$$

which means A' is a unitary transformation on A ($i.e.$ has all the important properties of the original matrix).

This works just as well for $A = QL$ where L is lower triangular. Numerical recipes proceeds with QL (because it has smaller roundoff error). The algorithm consists of a sequence of unitary (Householder) transformations,

$$A_s = Q_s L_s \tag{139}$$

$$A_{s+1} = L_s Q_s \quad (= Q_s^\dagger A_s Q_s). \tag{140}$$

There is a theorem that as $s \to \infty$ (i) if A has non-degenerate eigenvalues λ_i, the $A_s \to$ lower triangular form. The λ_i appear on the diagonal in increasing order of magnitude. (ii) if A has a degeneracy n, the above limit is the same except for a diagonal block of size n for the degenerate eigenvalue.

The work load is $O(N^3)$, $O(N^2)$ and $O(N)$ for general, hessenberg, and tridiagonal matrices, respectively.

Now consider the case of a real, symmetric, tridiagonal matrix. In the case of degenerate eigenvalues, the theorem says there are $n-1$ zeros on the sub(super)

diagonals which means the matrix can be split into blocks which are diagonalized separately.

The proof of the theorem reveals that super-diagonal elements vanish like

$$a_{ij} \sim \left(\frac{\lambda_i}{\lambda_j}\right)^s \tag{141}$$

where $\lambda_i < \lambda_j$. For dense spectra, this can be very slow. Convergence is accelerated by working with a *shifted* matrix $A - kI$.

$$A_s - k_s I = Q_s L_s \tag{142}$$

$$A_{s+1} = L_s Q_s + k_s I \tag{143}$$

$$= Q_s^\dagger A_s Q_s \tag{144}$$

and the convergence is then

$$a_{ij} \sim \left(\frac{\lambda_i - k_s}{\lambda_j - k_s}\right)^s \tag{145}$$

The shifts can be chosen at each step. For example $k_1 \approx \lambda_1$ is a good choice to quickly eliminate the first row of off-diagonals. Usually we don't know what that is beforehand! A good procedure in practice is to diagonalize the leading 2×2 submatrix on the diagonal and choose the shift equal to the eigenvalue closest to $a_1 1$. To improve convergence even further, one can *deflate* the matrix by eliminating the $r - 1$ rows and columns where r is the number of eigenvalues you have converged at the current step. Then k_s is taken as the eigenvalue of the remaining leading 2×2 sub matrix

Adding shifts means that eigenvalues no longer appear in order of increasing magnitude, so they may need to be sorted at each step.

A final flourish is to avoid roundoff errors induced by large shifts when the spectrum spans a large range. The idea is known as *implicit shifts*...

17.2.5 Code I.

The following is my simple implementation of the Lanczos algorithm given at the end of the section 17.2.3. I tried adding the tridiagonal matrix diagonalization routing tqli from Numerical Recipes. However it seems this does not work, and I haven't figured out why.

To diagonalize the tridiagonal matrix produced by the lanczos procedure, I used the free software Eigen package. There is a tutorial. I don't know a lot about it, but it seems to work, and seems to be in wide use. Eigen is comprised entirely in header files, so it is particularly easy to use. Just include the requisite header in your code. No additional compilation is needed.

```
// a simple implementation of the Lanczos algorithm
// reduces a hermitian matrix to tridiagonal form
// includes full Gram-Schmidt orthogonalization
// Test on diagonal matrix with gaussian distributed random numbers
// Diagonalize the tridiagonal matrix with Eigen routine

#include <complex>
#include <iostream>
#include <vector>
#include <random>
#include "/Users/tblum/Dropbox/Eigen/Eigen/Eigenvalues"
#define tol 1e-16

using namespace std;

// the matrix we are diagonalizing acting on a vector
void multOp(vector<complex<double> > &in,
    vector<complex<double> > &out,
    vector<double> &Op
    ){
  for(int i=0;i<in.size();i++){
    out[i] = Op[i]*in[i];
  }
```

```
}
// real norm of a complex vector
double norm(vector<complex<double> > &v){
  double sum=0.0;
  for(int i=0;i<v.size();i++){
    sum+=(conj(v[i])*v[i]).real();
  }
  return sqrt(sum);
}
//dot product of two complex vectors: a^* . b
complex<double> dot(vector<complex<double> > &a,
    vector<complex<double> > &b
    )
{
  if(a.size() != b.size()){
    cout << "error: a and b different sizes in dot" << endl;
    exit(-1);
  }
  complex<double>result(0.0,0.0);
  for(int i=0;i<a.size();i++){
    result+= conj(a[i])*b[i];
  }
  return result;
}
//dot product of two complex vectors: a^* . b
complex<double> cdot(vector<complex<double> > &a,
     vector<complex<double> > &b
     )
{
  if(a.size() != b.size()){
    cout << "error: a and b different sizes in cdot" << endl;
    exit(-1);
  }
  complex<double>result(0.0,0.0);
  for(int i=0;i<a.size();i++){
    result+= a[i]*b[i];
  }
  return result;
```

```
}
// Gram-Schmit full orthogonalization of vector w to all vectors v[i]
void gram_schmidt(vector<vector<complex<double> > > &v,
  vector<complex<double> > &w,
  int k)
{
  for(int j=0; j<k; j++){
    complex<double> c=cdot(w,v[j]);
    for(int i=0; i<v[0].size(); i++){
      w[i] = w[i]-c*v[j][i];
    }
  }
}

// eigenvalues only of a tridiagonal matrix
// from Numerical Recipes
// appears not to work. Use Eigen instead.
#define SIGN(a,b) ((b)<0 ? -fabs(a) : fabs(a))
int tqli(vector<double> &d, vector<double> &e, int n)
{
  int     m, l, iter, i, k;
  double  s, r, p, g, f, dd, c, b;

  for (i = 1; i < n; i++) e[i - 1] = e[i];
  e[n] = 0.0;
  for (l = 0; l < n; l++)
    {
      iter = 0;
      do
        {
  for (m = l; m < n - 1; m++)
            {
      dd = fabs(d[m]) + fabs(d[m + 1]);
      if (fabs(e[m]) + dd == dd)
break;
            }
  if (m != l)
            {
```

```
      if (iter++ == 30)
return (iter);
      g = (d[l + 1] - d[l]) / (2.0 * e[l]);
      r = sqrt((g * g) + 1.0);
      g = d[m] - d[l] + e[l] / (g + SIGN(r, g));
      s = c = 1.0;
      p = 0.0;
      for (i = m - 1; i >= l; i--)
                {
  f = s * e[i];
  b = c * e[i];
  if (fabs(f) >= fabs(g))
                    {
      c = g / f;
      r = sqrt((c * c) + 1.0);
      e[i + 1] = f * r;
      c *= (s = 1.0 / r);
                    } else
                    {
      s = f / g;
      r = sqrt((s * s) + 1.0);
      e[i + 1] = g * r;
      s *= (c = 1.0 / r);
                    }
  g = d[i + 1] - p;
  r = (d[i] - g) * s + 2.0 * c * b;
  p = s * r;
  d[i + 1] = g + p;
  g = c * r - b;
  //for (k = 0; k < n; k++)
  //{
  //  f = z[k][i + 1];
  //  z[k][i + 1] = s * z[k][i] + c * f;
  //  z[k][i] = c * z[k][i] - s * f;
  //}
                }
      d[l] = d[l] - p;
      e[l] = g;
```

```
      e[m] = 0.0;
           }
        } while (m != l);
    }

  return (iter);
}

//Do the Lanczos iteration m times
int lanczos(int m,
    vector<vector<complex<double> > > &evec,
    vector<double> &Op
    ){

  // tridiag vectors
  vector<double> alpha;
  alpha.resize(m);
  vector<double> beta;
  beta.resize(m);
  beta[0] = 0;

  int N = evec[0].size();
  // initialize, set starting vector to a uniform vector
  for(int i=0;i<N;i++){
    evec[0][i] = complex<double>(1.,0.);
    evec[0][i] /= sqrt((double)N);
  }

  // The first step
  vector <complex<double> > w;
  w.resize(N);
  multOp(evec[0],w,Op);
  alpha[0]=real(dot(w,evec[0]));
  for(int j=0;j<N;j++)
    w[j] = w[j] - alpha[0]*evec[0][j];
  beta[0] = norm(w);

  cout << "alpha, beta i = "<< 0 << ' ' << alpha[0] << ' ' << beta[0] << endl;
```

```
  for(int j=0;j<N;j++)
    evec[1][j] = w[j]/beta[0];


  // lanczos iteration
  for(int i=1;i<m;i++){

    multOp(evec[i], w, Op);
    for(int j=0;j<N;j++)
w[j] = w[j] - beta[i-1]*evec[i-1][j];
    alpha[i] = real(dot(w,evec[i]));
    for(int j=0;j<N;j++)
w[j] = w[j] - alpha[i]*evec[i][j];
    gram_schmidt(evec,w,i+1);
    beta[i] = norm(w);
    cout << "alpha, beta i = "<< i << ' ' << alpha[i] << ' ' << beta[i] << endl;
    if(abs(beta[i]) < tol){
      cout << "beta less than tol at step " << i << endl;
      //return i;
    };
    if(i==m-1)break;
    for(int j=0;j<N;j++)
      evec[i+1][j] = w[j]/beta[i];

  }
  //diagonalize the tridiagonal matrix
  // int tqli_iters = tqli(alpha,beta,m);
  // cout << "tqli iters= " << tqli_iters << endl;
  // sort(alpha.begin(),alpha.end());
  // reverse(alpha.begin(),alpha.end());
  Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> mat;
  Eigen::VectorXd a(m); for(int i=0; i<m;i++)a[i] = alpha[i];
  Eigen::VectorXd b(m); for(int i=0; i<m;i++)b[i] = beta[i];
  mat.computeFromTridiagonal(a,b,Eigen::EigenvaluesOnly);
  a = mat.eigenvalues();
  for(int j=0;j<m;j++){
    cout << "m lambda j = "<< m << ' ' << j << ' ' << a[j] << endl;
```

```
  }

  return m;
}

int main(int argc, char **argv){

  // linear size of matrix
  int N;
  // number of eigenvalues to find
  int m;
  cout << "enter matrix size N, lanczos iters m" << endl;
  cin >> N >> m;

  //set up random gaussian number distribution
  vector<double> Op;
  Op.resize(N);
  default_random_engine generator;
  normal_distribution<double> distribution(0.0,1.0);
  for(int i=0;i<N;i++)
    Op[i] = distribution(generator);
  // sort from largest to smallest
  sort(Op.begin(), Op.end());
  reverse(Op.begin(), Op.end());
  for(int i=0;i<N;i++)
    cout << "gauss rand i= " << i << "\t" << Op[i] << endl;

  // call the lanczos iteration
  vector<vector<complex<double> > > evec;
  evec.resize(m);
  for(int i=0;i<m;i++)
    evec[i].resize(N);
  lanczos(m,evec,Op);

  return 0;
}
```

In the above code, there is a call to Eigen::SelfAdjointSolver which uses the symmetric QR algorithm to find the eigenvalues (and eigenvectors) of the tridiagonal matrix.

17.2.6 example

A simple but very useful example is to set the original matrix to a diagonal matrix with gaussian distributed values. Results for various numbers of Lanczos iterations are shown in Fig. 15. Notice that the *extreme* eigenvalues converge the fastest while the lowest are the slowest. (Why?).

17.2.7 Finding eigenvectors

So far our Lanczos+QR algorithm has only been used to compute eigenvalues of hermitian matrix. What about the eigenvectors? Recall that the Lanczos algorithm is a similarity transformation on the matrix A:

$$T = V^\dagger A V \tag{146}$$

The columns of V are the orthogonal Lanczos vectors (Krylov space basis vectors). The QR algorithm diagonalizes the tridiagonal matrix with another transformation.

$$D = U^\dagger T U \tag{147}$$

$$= U^\dagger V^\dagger A V U \tag{148}$$

The combined transformation VU diagonalizes the original matrix. It's columns are the corresponding eigenvectors of A. In practice the eigenvectors are only estimates of the true eigenvectors of A which become more accurate as the Lanczos procedure converges.

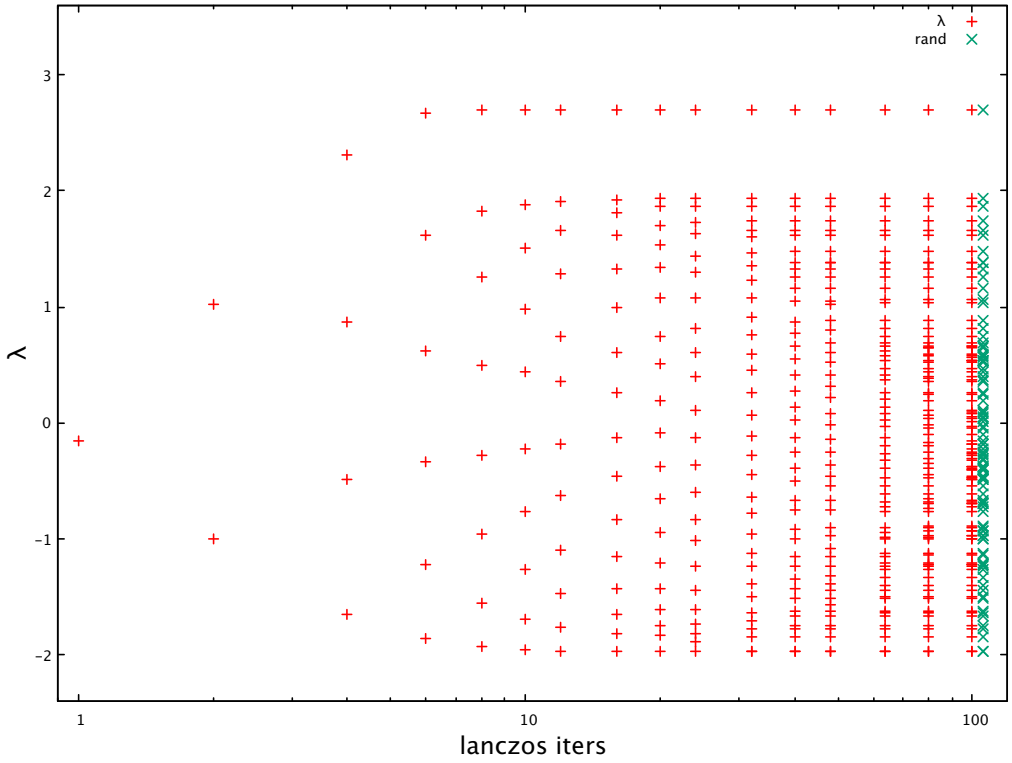

Figure 15: Convergence of Lanczos eigenvalues. The matrix is size $N = 100$ with diagonal elements only, taken from a gaussian distribution with zero mean and width 1.0. Red plusses are computed eigenvalues and the green crosses are the exact values (they agree to seven digits). Notice that the extreme eigenvalues converge fastest, lowest slowest.

In the above Lanczos code, I added the following lines to compute the eigenvectors,

```
...
  //mat.computeFromTridiagonal(a,b,Eigen::EigenvaluesOnly);
  mat.computeFromTridiagonal(a,b,Eigen::ComputeEigenvectors);
  Eigen::MatrixXd V(m,m);
  V = mat.eigenvectors();
  a = mat.eigenvalues();
  for(int j=0;j<m;j++){
    cout << "m lambda j = "<< m << ' ' << j << ' ' << a[j] << endl;
  }
  // eigenvectors
  vector<vector<complex<double> > > final_evec;
  final_evec.resize(m);
  for(int i=0;i<m;i++){
    final_evec[i].resize(N);
    for(int j=0;j<N;j++){
      final_evec[i][j] = complex<double>(0.0,0.0);
    }
  }
  for(int j=0;j<N;j++){
    for(int i=0;i<m;i++){
      for(int k=0;k<m;k++){
final_evec[i][j] += evec[k][j] * V(k,i);
      }
    }
  }
  for(int i=0;i<m;i++){
    cout << "evec " << i << endl;
    for(int j=0;j<N;j++){
      cout << j << ' ' << final_evec[i][j] << endl;
    }
  }
  //test orthogonality
  for(int i=0;i<m;i++){
    for(int j=0;j<m;j++){
    complex<double> c = dot(final_evec[i],final_evec[j]);
```

```
    cout << i << " dot " << j << ' ' << c << endl;
    }
  }
  return m;
}
```

Running the code for $N = 10$ and $m = 3$, 8, and 10, we find

```
Thomass-iMac:lanczos tblum$ g++ lanczos_v3.C
Thomass-iMac:lanczos tblum$ a.out
enter matrix size N, lanczos iters m
10 3
gauss rand i= 0 1.87065
gauss rand i= 1 1.61201
gauss rand i= 2 1.3828
gauss rand i= 3 0.399771
gauss rand i= 4 0.0523187
gauss rand i= 5 -0.407472
gauss rand i= 6 -0.688081
gauss rand i= 7 -0.904147
gauss rand i= 8 -1.23974
gauss rand i= 9 -1.66043
alpha, beta i = 0 0.041769 1.17924
alpha, beta i = 1 0.335458 0.936567
alpha, beta i = 2 -0.225068 0.864837
m lambda j = 3 0 -1.38947
m lambda j = 3 1 -0.118438
m lambda j = 3 2 1.66007
evec 0
0 (0.063255,0)
1 (-0.0125578,0)
2 (-0.0628565,0)
3 (-0.0986243,0)
4 (-0.0414574,0)
5 (0.0902421,0)
6 (0.201992,0)
7 (0.304244,0)
8 (0.491017,0)
```

```
9 (0.7732,0)
evec 1
0 (0.150093,0)
1 (-0.0315339,0)
2 (-0.168044,0)
3 (-0.492985,0)
4 (-0.506771,0)
5 (-0.443871,0)
6 (-0.360061,0)
7 (-0.272068,0)
8 (-0.0949243,0)
9 (0.196699,0)
evec 2
0 (0.691831,0)
1 (0.554121,0)
2 (0.444014,0)
3 (0.09892,0)
4 (0.0262658,0)
5 (-0.030281,0)
6 (-0.0426257,0)
7 (-0.040682,0)
8 (-0.0179122,0)
9 (0.0445758,0)
i dot j (1,0)
i dot j (8.32667e-17,0)
i dot j (2.42861e-16,0)
i dot j (8.32667e-17,0)
i dot j (1,0)
i dot j (-2.77556e-17,0)
i dot j (2.42861e-16,0)
i dot j (-2.77556e-17,0)
i dot j (1,0)
Thomass-iMac:lanczos tblum$ g++ lanczos_v3.C
Thomass-iMac:lanczos tblum$ a.out
enter matrix size N, lanczos iters m
10 3
gauss rand i= 0 1.87065
gauss rand i= 1 1.61201
```

```
gauss rand i= 2 1.3828
gauss rand i= 3 0.399771
gauss rand i= 4 0.0523187
gauss rand i= 5 -0.407472
gauss rand i= 6 -0.688081
gauss rand i= 7 -0.904147
gauss rand i= 8 -1.23974
gauss rand i= 9 -1.66043
alpha, beta i = 0 0.041769 1.17924
alpha, beta i = 1 0.335458 0.936567
alpha, beta i = 2 -0.225068 0.864837
m lambda j = 3 0 -1.38947
m lambda j = 3 1 -0.118438
m lambda j = 3 2 1.66007
evec 0
0 (0.063255,0)
1 (-0.0125578,0)
2 (-0.0628565,0)
3 (-0.0986243,0)
4 (-0.0414574,0)
5 (0.0902421,0)
6 (0.201992,0)
7 (0.304244,0)
8 (0.491017,0)
9 (0.7732,0)
evec 1
0 (0.150093,0)
1 (-0.0315339,0)
2 (-0.168044,0)
3 (-0.492985,0)
4 (-0.506771,0)
5 (-0.443871,0)
6 (-0.360061,0)
7 (-0.272068,0)
8 (-0.0949243,0)
9 (0.196699,0)
evec 2
0 (0.691831,0)
```

```
1 (0.554121,0)
2 (0.444014,0)
3 (0.09892,0)
4 (0.0262658,0)
5 (-0.030281,0)
6 (-0.0426257,0)
7 (-0.040682,0)
8 (-0.0179122,0)
9 (0.0445758,0)
0 dot 0 (1,0)
0 dot 1 (8.32667e-17,0)
0 dot 2 (2.42861e-16,0)
1 dot 0 (8.32667e-17,0)
1 dot 1 (1,0)
1 dot 2 (-2.77556e-17,0)
2 dot 0 (2.42861e-16,0)
2 dot 1 (-2.77556e-17,0)
2 dot 2 (1,0)
Thomass-iMac:lanczos tblum$ a.out
enter matrix size N, lanczos iters m
10 8
gauss rand i= 0 1.87065
gauss rand i= 1 1.61201
gauss rand i= 2 1.3828
gauss rand i= 3 0.399771
gauss rand i= 4 0.0523187
gauss rand i= 5 -0.407472
gauss rand i= 6 -0.688081
gauss rand i= 7 -0.904147
gauss rand i= 8 -1.23974
gauss rand i= 9 -1.66043
alpha, beta i = 0 0.041769 1.17924
alpha, beta i = 1 0.335458 0.936567
alpha, beta i = 2 -0.225068 0.864837
alpha, beta i = 3 0.0969014 1.01879
alpha, beta i = 4 0.44 0.793629
alpha, beta i = 5 -0.264974 0.655872
alpha, beta i = 6 0.363399 0.939007
```

```
alpha, beta i = 7 0.417257 0.57213
m lambda j = 8 0 -1.66
m lambda j = 8 1 -1.21849
m lambda j = 8 2 -0.763123
m lambda j = 8 3 -0.25068
m lambda j = 8 4 0.322062
m lambda j = 8 5 1.32932
m lambda j = 8 6 1.57703
m lambda j = 8 7 1.86862
evec 0
0 (-0.000228446,0)
1 (0.00108237,0)
2 (-0.00128895,0)
3 (0.00403781,0)
4 (-0.00791017,0)
5 (0.00683541,0)
6 (0.00542621,0)
7 (-0.0144141,0)
8 (0.00929105,0)
9 (0.999774,0)
evec 1
0 (0.00138893,0)
1 (-0.00665592,0)
2 (0.00802049,0)
3 (-0.0273398,0)
4 (0.0566983,0)
5 (-0.0561566,0)
6 (-0.0528926,0)
7 (0.184373,0)
8 (0.97773,0)
9 (-0.00518014,0)
evec 2
0 (0.00280882,0)
1 (-0.0136762,0)
2 (0.0167633,0)
3 (-0.0655976,0)
4 (0.152349,0)
5 (-0.220796,0)
```

```
6 (-0.644586,0)
7 (-0.708593,0)
8 (0.0751422,0)
9 (-0.00439891,0)
evec 3
0 (-0.00508448,0)
1 (0.0254253,0)
2 (-0.0321079,0)
3 (0.170988,0)
4 (-0.597785,0)
5 (-0.730208,0)
6 (-0.161237,0)
7 (0.222955,0)
8 (-0.0527936,0)
9 (0.00408222,0)
evec 4
0 (-0.00436339,0)
1 (0.0230007,0)
2 (-0.0309758,0)
3 (0.896642,0)
4 (0.420669,0)
5 (-0.098317,0)
6 (-0.0437387,0)
7 (0.0744356,0)
8 (-0.0209451,0)
9 (0.00181858,0)
evec 5
0 (-0.019585,0)
1 (0.164678,0)
2 (-0.96402,0)
3 (-0.117608,0)
4 (0.13942,0)
5 (-0.0647962,0)
6 (-0.0343621,0)
7 (0.0641194,0)
8 (-0.0199783,0)
9 (0.00189205,0)
evec 6
```

```
0 (-0.0263822,0)
1 (0.972439,0)
2 (0.193934,0)
3 (-0.0678503,0)
4 (0.0853183,0)
5 (-0.0414343,0)
6 (-0.0223613,0)
7 (0.0421721,0)
8 (-0.0133136,0)
9 (0.00127667,0)
evec 7
0 (0.998795,0)
1 (0.0348,0)
2 (-0.0203562,0)
3 (0.0142773,0)
4 (-0.0188036,0)
5 (0.00948465,0)
6 (0.00520123,0)
7 (-0.0099076,0)
8 (0.00316748,0)
9 (-0.000307484,0)
0 dot 0 (1,0)
0 dot 1 (-8.23994e-17,0)
0 dot 2 (-9.80119e-17,0)
0 dot 3 (-4.59702e-17,0)
0 dot 4 (3.38271e-17,0)
0 dot 5 (2.05998e-17,0)
0 dot 6 (-4.2067e-17,0)
0 dot 7 (-1.95698e-17,0)
1 dot 0 (-8.23994e-17,0)
1 dot 1 (1,0)
1 dot 2 (1.89698e-16,0)
1 dot 3 (-5.4244e-18,0)
1 dot 4 (-2.75879e-17,0)
1 dot 5 (1.95539e-16,0)
1 dot 6 (-1.07449e-16,0)
1 dot 7 (4.03345e-16,0)
2 dot 0 (-9.80119e-17,0)
```

```
2 dot 1 (1.89698e-16,0)
2 dot 2 (1,0)
2 dot 3 (2.49485e-16,0)
2 dot 4 (-1.36216e-16,0)
2 dot 5 (6.3192e-17,0)
2 dot 6 (1.51111e-17,0)
2 dot 7 (-6.56451e-17,0)
3 dot 0 (-4.59702e-17,0)
3 dot 1 (-5.4244e-18,0)
3 dot 2 (2.49485e-16,0)
3 dot 3 (1,0)
3 dot 4 (-1.91675e-17,0)
3 dot 5 (-4.97141e-17,0)
3 dot 6 (-2.18272e-17,0)
3 dot 7 (9.91569e-17,0)
4 dot 0 (3.38271e-17,0)
4 dot 1 (-2.75879e-17,0)
4 dot 2 (-1.36216e-16,0)
4 dot 3 (-1.91675e-17,0)
4 dot 4 (1,0)
4 dot 5 (8.70009e-17,0)
4 dot 6 (-4.21526e-18,0)
4 dot 7 (-1.19364e-16,0)
5 dot 0 (2.05998e-17,0)
5 dot 1 (1.95539e-16,0)
5 dot 2 (6.3192e-17,0)
5 dot 3 (-4.97141e-17,0)
5 dot 4 (8.70009e-17,0)
5 dot 5 (1,0)
5 dot 6 (-2.2863e-16,0)
5 dot 7 (1.01052e-17,0)
6 dot 0 (-4.2067e-17,0)
6 dot 1 (-1.07449e-16,0)
6 dot 2 (1.51111e-17,0)
6 dot 3 (-2.18272e-17,0)
6 dot 4 (-4.21526e-18,0)
6 dot 5 (-2.2863e-16,0)
6 dot 6 (1,0)
```

```
6 dot 7 (2.11683e-16,0)
7 dot 0 (-1.95698e-17,0)
7 dot 1 (4.03345e-16,0)
7 dot 2 (-6.56451e-17,0)
7 dot 3 (9.91569e-17,0)
7 dot 4 (-1.19364e-16,0)
7 dot 5 (1.01052e-17,0)
7 dot 6 (2.11683e-16,0)
7 dot 7 (1,0)
Thomass-iMac:lanczos tblum$ a.out
enter matrix size N, lanczos iters m
10 10
gauss rand i= 0 1.87065
gauss rand i= 1 1.61201
gauss rand i= 2 1.3828
gauss rand i= 3 0.399771
gauss rand i= 4 0.0523187
gauss rand i= 5 -0.407472
gauss rand i= 6 -0.688081
gauss rand i= 7 -0.904147
gauss rand i= 8 -1.23974
gauss rand i= 9 -1.66043
alpha, beta i = 0 0.041769 1.17924
alpha, beta i = 1 0.335458 0.936567
alpha, beta i = 2 -0.225068 0.864837
alpha, beta i = 3 0.0969014 1.01879
alpha, beta i = 4 0.44 0.793629
alpha, beta i = 5 -0.264974 0.655872
alpha, beta i = 6 0.363399 0.939007
alpha, beta i = 7 0.417257 0.57213
alpha, beta i = 8 -0.185068 0.261867
alpha, beta i = 9 -0.601984 8.60537e-32
beta less than tol at step 9
m lambda j = 10 0 -1.66043
m lambda j = 10 1 -1.23974
m lambda j = 10 2 -0.904147
m lambda j = 10 3 -0.688081
m lambda j = 10 4 -0.407472
```

```
m lambda j = 10 5 0.0523187
m lambda j = 10 6 0.399771
m lambda j = 10 7 1.3828
m lambda j = 10 8 1.61201
m lambda j = 10 9 1.87065
evec 0
0 (3.82466e-16,0)
1 (-1.35186e-16,0)
2 (-3.31115e-16,0)
3 (-2.13696e-16,0)
4 (-1.44416e-16,0)
5 (-1.06685e-16,0)
6 (3.03577e-17,0)
7 (2.25514e-16,0)
8 (1.11315e-15,0)
9 (-1,0)
evec 1
0 (2.57498e-17,0)
1 (3.92481e-17,0)
2 (-2.66714e-17,0)
3 (-2.34188e-17,0)
4 (1.00614e-16,0)
5 (9.02056e-17,0)
6 (3.46945e-17,0)
7 (-3.33067e-16,0)
8 (1,0)
9 (1.00842e-15,0)
evec 2
0 (8.26162e-17,0)
1 (8.76035e-17,0)
2 (-1.59595e-16,0)
3 (9.02056e-17,0)
4 (4.16334e-17,0)
5 (3.60822e-16,0)
6 (-1.11022e-16,0)
7 (1,0)
8 (1.2837e-16,0)
9 (-9.54098e-18,0)
```

```
evec 3
0 (-8.06646e-17,0)
1 (-5.20417e-17,0)
2 (2.08167e-17,0)
3 (-1.11022e-16,0)
4 (2.77556e-16,0)
5 (-1.11022e-16,0)
6 (1,0)
7 (-1.66533e-16,0)
8 (2.77556e-17,0)
9 (-6.50521e-17,0)
evec 4
0 (7.50268e-17,0)
1 (2.25514e-17,0)
2 (-3.81639e-17,0)
3 (3.46945e-17,0)
4 (1.94289e-16,0)
5 (-1,0)
6 (1.66533e-16,0)
7 (0,0)
8 (1.249e-16,0)
9 (1.01048e-16,0)
evec 5
0 (-3.7405e-17,0)
1 (5.20417e-17,0)
2 (-1.53523e-16,0)
3 (3.29597e-16,0)
4 (1,0)
5 (1.66533e-16,0)
6 (-1.11022e-16,0)
7 (-8.32667e-17,0)
8 (-3.1225e-16,0)
9 (1.59161e-16,0)
evec 6
0 (8.83083e-17,0)
1 (1.43115e-16,0)
2 (-1.48753e-16,0)
3 (-1,0)
```

```
4 (-3.46945e-17,0)
5 (-6.245e-17,0)
6 (-1.66533e-16,0)
7 (-3.05311e-16,0)
8 (8.32667e-17,0)
9 (2.37982e-17,0)
evec 7
0 (1.01644e-18,0)
1 (-1.36013e-16,0)
2 (1,0)
3 (-4.22839e-16,0)
4 (2.42861e-17,0)
5 (-3.64292e-17,0)
6 (-1.59595e-16,0)
7 (-1.50921e-16,0)
8 (7.37257e-17,0)
9 (2.53907e-16,0)
evec 8
0 (-1.57449e-15,0)
1 (-1,0)
2 (-5.17219e-16,0)
3 (-1.4962e-17,0)
4 (-1.12757e-17,0)
5 (-2.25514e-17,0)
6 (2.60209e-17,0)
7 (-5.20417e-18,0)
8 (-2.43512e-16,0)
9 (-1.26933e-16,0)
evec 9
0 (1,0)
1 (-2.0192e-15,0)
2 (-2.07069e-16,0)
3 (4.31241e-16,0)
4 (3.15828e-16,0)
5 (2.48065e-16,0)
6 (2.17274e-16,0)
7 (1.35742e-16,0)
8 (-1.30104e-17,0)
```

```
9 (1.61709e-16,0)
0 dot 0 (1,0)
0 dot 1 (1.04734e-16,0)
0 dot 2 (2.35055e-16,0)
0 dot 3 (9.54098e-17,0)
0 dot 4 (5.63785e-18,0)
0 dot 5 (-3.03577e-16,0)
0 dot 6 (1.89898e-16,0)
0 dot 7 (-5.85022e-16,0)
0 dot 8 (2.62119e-16,0)
0 dot 9 (2.20757e-16,0)
1 dot 0 (1.04734e-16,0)
1 dot 1 (1,0)
1 dot 2 (-2.04697e-16,0)
1 dot 3 (6.245e-17,0)
1 dot 4 (3.46945e-17,0)
1 dot 5 (-2.11636e-16,0)
1 dot 6 (1.06685e-16,0)
1 dot 7 (4.70544e-17,0)
1 dot 8 (-2.8276e-16,0)
1 dot 9 (1.27394e-17,0)
2 dot 0 (2.35055e-16,0)
2 dot 1 (-2.04697e-16,0)
2 dot 2 (1,0)
2 dot 3 (-2.77556e-16,0)
2 dot 4 (-3.60822e-16,0)
2 dot 5 (-4.16334e-17,0)
2 dot 6 (-3.95517e-16,0)
2 dot 7 (-3.10516e-16,0)
2 dot 8 (-9.28077e-17,0)
2 dot 9 (2.18358e-16,0)
3 dot 0 (9.54098e-17,0)
3 dot 1 (6.245e-17,0)
3 dot 2 (-2.77556e-16,0)
3 dot 3 (1,0)
3 dot 4 (2.77556e-16,0)
3 dot 5 (1.66533e-16,0)
3 dot 6 (-5.55112e-17,0)
```

```
3 dot 7 (-1.38778e-16,0)
3 dot 8 (7.80626e-17,0)
3 dot 9 (1.36609e-16,0)
4 dot 0 (5.63785e-18,0)
4 dot 1 (3.46945e-17,0)
4 dot 2 (-3.60822e-16,0)
4 dot 3 (2.77556e-16,0)
4 dot 4 (1,0)
4 dot 5 (2.77556e-17,0)
4 dot 6 (2.77556e-17,0)
4 dot 7 (-1.73472e-18,0)
4 dot 8 (-1.31352e-31,0)
4 dot 9 (-1.73039e-16,0)
5 dot 0 (-3.03577e-16,0)
5 dot 1 (-2.11636e-16,0)
5 dot 2 (-4.16334e-17,0)
5 dot 3 (1.66533e-16,0)
5 dot 4 (2.77556e-17,0)
5 dot 5 (1,0)
5 dot 6 (-3.64292e-16,0)
5 dot 7 (-1.29237e-16,0)
5 dot 8 (-6.33174e-17,0)
5 dot 9 (2.78423e-16,0)
6 dot 0 (1.89898e-16,0)
6 dot 1 (1.06685e-16,0)
6 dot 2 (-3.95517e-16,0)
6 dot 3 (-5.55112e-17,0)
6 dot 4 (2.77556e-17,0)
6 dot 5 (-3.64292e-16,0)
6 dot 6 (1,0)
6 dot 7 (2.74086e-16,0)
6 dot 8 (-1.28153e-16,0)
6 dot 9 (-3.42933e-16,0)
7 dot 0 (-5.85022e-16,0)
7 dot 1 (4.70544e-17,0)
7 dot 2 (-3.10516e-16,0)
7 dot 3 (-1.38778e-16,0)
7 dot 4 (-1.73472e-18,0)
```

```
7 dot 5 (-1.29237e-16,0)
7 dot 6 (2.74086e-16,0)
7 dot 7 (1,0)
7 dot 8 (-3.81205e-16,0)
7 dot 9 (-2.06053e-16,0)
8 dot 0 (2.62119e-16,0)
8 dot 1 (-2.8276e-16,0)
8 dot 2 (-9.28077e-17,0)
8 dot 3 (7.80626e-17,0)
8 dot 4 (-1.31352e-31,0)
8 dot 5 (-6.33174e-17,0)
8 dot 6 (-1.28153e-16,0)
8 dot 7 (-3.81205e-16,0)
8 dot 8 (1,0)
8 dot 9 (4.44713e-16,0)
9 dot 0 (2.20757e-16,0)
9 dot 1 (1.27394e-17,0)
9 dot 2 (2.18358e-16,0)
9 dot 3 (1.36609e-16,0)
9 dot 4 (-1.73039e-16,0)
9 dot 5 (2.78423e-16,0)
9 dot 6 (-3.42933e-16,0)
9 dot 7 (-2.06053e-16,0)
9 dot 8 (4.44713e-16,0)
9 dot 9 (1,0)
```

The eigenvectors converge as claimed.

17.2.8 Bose-Hubbard model

A problem that is very interesting which we have already encountered in the form of the GPE is the Bose-Hubbard model. With the Lanczos algorithm developed in the previous sections, we can solve this model by exact (to numerical precision) diagonalization instead of resorting to mean field theory (GPE). The model is a variation of the original Hubbard model which was developed to describe superconductivity in

fermion systems. This model describes superfluid transitions in systems of bosons.

The Bose-Hubbard Hamiltonian is

$$H = -t\sum_{\langle i,j\rangle}\hat{a}_i^\dagger\hat{a}_j + \frac{U}{2}\sum_i \hat{n}_i(\hat{n}_i - 1) - \mu\sum_i \hat{n}_i, \tag{149}$$

where $\hat{a}_i$ is the boson annihilation operator at site i, t controls the mobility of bosons on the lattice, U is the onsite interaction strength ($U < (>)0$ is attractive (repulsive)), and μ is the chemical potential. As usual $\langle i,j\rangle$ means sum over nearest neighbor pairs.

The major computational difficulty associated with the BH model is that the dimension of the associated Hilbert space (and the therefore the dimension of H) grows like

$$D = \frac{(N+L-1)!}{(N)!(L-1)!} \tag{150}$$

where N is the number of bosons and L is the number of lattice sites. At fixed N or L, the growth is polynomial. But if the *density* of bosons is fixed, $n = N/L$, the growth is exponential! Therefore we will be restricted to small lattices with a few bosons. Typically in experiments the number of lattice sites is $O(10^6)$ with average *filling* $N/L > 1$ (*c.f.* your homework for GPE!).

Following previous practice, I have found a pedagogical paper that explains the model and it's exact diagonalization in more detail which we will use as a guide. Note the wiki page on many body systems that we looked at for the GPE is also useful here.

The idea is to work in the occupation number basis, $|n_1, n_2, \ldots, n_{L-1}\rangle$ which simply encodes the number of particles at each site in the given state,

$$\hat{n}_i|n_1, n_2, \ldots, n_{L-1}\rangle = n_i|n_1, n_2, \ldots, n_{L-1}\rangle. \tag{151}$$

If we work with a fixed number of particles, N, then we impose the constraint

$$N = \sum_{i=0}^{L-1} n_i. \tag{152}$$

In the paper mentioned above, there are 24, 310 states for the case $N = L = 9$, and 5, 200, 300 for $N = L = 13$! However, it will turn out the number of *non-zero* matrix elements of the hamiltonian between these states is very small. So the problem is to diagonalize a sparse matrix, albeit one with large dimension. Perfect for Lanczos!

The problem now boils down to efficiently enumerating all of the states (for fixed N, say) and then computing the Hamiltonian matrix whose elements are mostly zero. There are neat ideas on how to do both.

First, the states can be enumerated by establishing a *lexographic* ordering where states with larger n at the beginning come before ones with larger n at the end. In other words, assume you have two states, $|n_0, n_1, \ldots, n_{L-1}\rangle$ and $|n'_0, n'_1, \ldots, n'_{L-1}\rangle$ with identical filling up to site $0 \leq k-1 < L$ but at site k, $n_k > n'_k$. Then the first state is ordered before the second.

With this rule in hand, the following simple algorithm allows an efficient enumeration starting with $|n_0, n_1, \ldots, n_k, 0, \cdots, 0\rangle$ where $n_k \neq 0$ but $n_i = 0$ for $i > k$:

1. while $(n_{L-1} < N)$ do
2. $n'_i = n_i$ for all $i < k$
3. $n'_k = n_k - 1$
4. $n'_{k+1} = N - \sum_{i=0}^{k} n'_i$
5. $n'_i = 0$ for $i > k+2$
6. end do

i.e., start with $|N, 0, \ldots, 0\rangle$ and end with $|0, \cdots, N\rangle$.

As an example, see Tab. 1 in the paper for $N = L = 3$. There are 10 states in this case.

Next, one needs to calculate all of the matrix elements $\langle i|H|j\rangle$. While it is clear how H acts on any basis vector, we don't want to simply loop over all D^2 possibilities since this would be costly in compute time and memory for large systems. And as already mentioned, most of the elements are zero. In fact only the kinetic part of the hamiltonian is non-trivial since the interaction term is (ultra) local (only diagonal terms enter).

The clever idea is to ask, if the H_{kin} acts on a basis vector $|v\rangle$, which basis vectors appear in the result? And what are the coefficients of the resulting linear combination? Remember that H_{kin} is a sum over nearest neighbor pairs $\langle ij\rangle$ of terms $a_i^\dagger a_j$. Each term will either annihilate $|v\rangle$ or turn it into another basis vector. We can easily calculate the new vector from the old one, but we need to know which vector it is so we can fill in that entry in the matrix H.

It turns out that a good way to answer this question is to use a *hash* table for the basis vectors where each basis vector has a unique hash (key) that is easy to look up in the table. For each basis vector $|v\rangle$ define the hash function

$$T(v) \;=\; \sum_{i=0}^{L-1} \sqrt{p_i} v_i \tag{153}$$

where p_i is a prime number given by $p_i = 100 * i + 3$, and v_i is the i-th component of $|v\rangle$, $v = 0, \ldots, D-1$. Given a vector with *unknown* v, calculate it's tag and look it up in the table $T(v)$ to find v, *i.e.* the vector you just generated. To make the look up efficient, the table should be sorted in ascending or descending order according to the tags. Then a method like the Newton binary method can be used to find an element in $O(\log_2 D)$ steps instead of $O(D)$ steps doing it the naive way. Thus you

have to have another map of indices between the sorted and unsorted look up tables, ind,

$$T(\mathrm{ind}(i)) = T_{\mathrm{sorted}}(i), \tag{154}$$

$$i = 0, \ldots, D-1. \tag{155}$$

Now the elements $\langle i|H|j\rangle$ can be calculated. For an arbitrary basis vector $|v\rangle$ with $v_j \geq 1$,

$$a_i^\dagger a_j |v\rangle = \sqrt{(v_i+1)v_j}\,|\ldots, v_i+1, \ldots, v_j-1, \ldots\rangle \tag{156}$$

$$= \sqrt{(v_i+1)v_j}\,|w\rangle \tag{157}$$

where $|w\rangle$ is another basis vector. To fill the matrix we simply compute $T_{\mathrm{sorted}}(w)$, $\mathrm{ind}(w) = u$, and we can fill in H_{wv}. This process is repeated for all $v = 0, \ldots, D-1$. The compute time apparently scales like $LD\log_2 D$. See Fig. 1 in the paper for an example of the sparsity pattern for H for $L = N$.

To find the ground state of the system we can use the Lanczos solver developed earlier. However we need to add another element to our code since Lanczos will find the extreme eigenvalues in practice. This may be beyond our present abilities, so perhaps we will use the MATLAB eig routine as suggested in the paper.

The BH model has symmetries (associated with, *e.g.*, particle number conservation, momentum conservation) that render the hamiltonian to be block-diagonal. When N, L are large it's necessary to take these into account, so we only work with sub-blocks of H. For example it's easy to work in momentum space. See the paper.

17.3 The Conjugate Gradient algoritm

Another powerful Krylov space method that is commonly used is the conjugate gradient (CG) method to solve large systems of equations $Ax = b$, or equivalently

to invert a large (sparse) matrix. The method works for Hermitian, positive definite matrices. If your matrix is not hermitian or positive definite, solve $A^\dagger Ax = A^\dagger b$ instead. The CG, like lanczos, is an iterative method that can be shown to converge in exactly n steps (for exact arithmetic), where n is the size of the matrix. In fact it usually converges (to machine precision) in many fewer steps.

According to Saad, "the method is a realization of an orthogonal projection technique on the Krylov subspace $\mathcal{K}_m(r_0, A)$ where r_0 is the residual vector $r_0 = Ax_0 - b$. Saad's derivation of the algorithm is a bit terse, as usual, so let's follow the above wiki-page instead. The algorithms derived are the same. In the following $r_k = b - Ax_k$ is the residual vector at the kth step of the iteration, and p_k is the "search direction" in the Krylov space where we are looking for the solution x_{k+1}. The p_k are the usual basis vectors in the Krylov space. The r_i also span the same Krylov space (and can be made orthogonal).

17.3.1 coefficients α_k and β_k

The computation of the necessary coefficients in the algorithm can be made efficient with a bit of linear algebra (following Saad and/or wiki). First, starting from the new solution at step $j+1$ which turns along p_j

$$\begin{aligned} x_{j+1} &= x_j + \alpha_j p_j && (158)\\ Ax_{j+1} &= Ax_j + \alpha_j Ap_j && (159)\\ Ax_{j+1} - b &= Ax_j - b + \alpha_j Ap_j && (160)\\ r_{j+1} &= r_j - \alpha_j Ap_j && (161) \end{aligned}$$

and forcing the r's to be orthogonal,

$$\begin{aligned} r_j^\dagger r_{j+1} &= r_j^\dagger (r_j - \alpha_j A p_j) && (162)\\ &= r_j^\dagger r_j - \alpha_j r_j^\dagger A p_j && (163)\\ \alpha_j &= \frac{r_j^\dagger r_j}{r_j^\dagger A p_j} && (164) \end{aligned}$$

Finally, since $r_{j+1} = p_{j+1} - \beta_j p_j$ and the p_j are A-orthogonal,

$$\alpha_j = \frac{r_j^\dagger r_j}{p_j^\dagger A p_j} \quad (165)$$

which is the final result for the α's. Now, rewriting $p_{j+1} = r_{j+1} + \beta_j p_j$ and using $(Ap_j, p_j) = 0$ again,

$$\begin{aligned} p_{j+1} &= r_{j+1} + \beta_j p_j && (166)\\ p_j^\dagger A p_{j+1} &= p_j^\dagger A r_{j+1} + \beta_j p_j^\dagger A p_j && (167)\\ 0 &= p_j^\dagger A r_{j+1} + \beta_j p_j^\dagger A p_j && (168)\\ \beta_j &= -\frac{p_j^\dagger A r_{j+1}}{p_j^\dagger A p_j} && (169) \end{aligned}$$

From (161)

$$Ap_j = -\frac{1}{\alpha_j}(r_{j+1} - r_j) \quad (170)$$

so

$$\begin{aligned} \beta_j &= -\frac{p_j^\dagger A r_{j+1}}{p_j^\dagger A p_j} && (171)\\ &= \frac{1}{\alpha_j}\frac{(r_{j+1} - r_j)^\dagger r_{j+1}}{p_j^\dagger A p_j} && (172)\\ &= \frac{r_{j+1}^\dagger r_{j+1}}{r_j^\dagger r_j} && (173)\\ & && (174) \end{aligned}$$

For completeness, the conventional algorithm for the CG from Saad is

1. compute $r_0 := b - Ax_0$, $p_0 := r_0$
2. For $j = 0, 1, \ldots$ until convergence Do
3. $\alpha_i := r_j^\dagger r_j$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j A p_j$
6. $\beta_j := r_{j+1}^\dagger r_{j+1} / r_j^\dagger r_j$
7. $p_{j+1} := r_{j+1} + \beta_j p_j$
8. EndDo

Notice that as the algorithm progresses, one searches directions in the Krylov space and builds residual vectors that have no overlap in the Krylov space. In other words, at convergence, the solution is in the Krylov subspace and the residual is outside (in the complement subspace).

The CG converges monotonically to the exact solution with a rate that depends on the condition number of the matrix. Many times a *preconditioning* matrix can be applied to achieve a smaller condition number, hence faster convergence.

18 Monte Carlo Methods

Here I will follow the excellent grad text by Binder and Heermann, Monte Carlo Simulation in Statistical Physics.

18.1 Motivation

In many cases in physics we are interested in systems with many degrees of freedom (Avagadro's number of them!) whose (model) hamiltonians are known. However, even for many simple models exact solutions do not exist (3d Ising, Hubbard, ...). Further, in statistical physics many times we are interested in simple averages of basics quantities (energy, magnetization, ...) which follow from very large dimension integrals. Though we will not cover quantum field theories, it is well known that their Euclidean space-time versions closely resemble statistical mechanical systems in one higher dimension.

Numerical Monte Carlo methods provide first principles answers to these problems, albeit with statistical and systematic (finite volume, lattice spacing) errors. These errors are *systematically* improvable and can be made arbitrarily small with enough compute power. Thus numerical solutions provide essentially exact answers to problems of interest in modern physics.

To see how this going to work, let's examine how to find π using a simple numerical integration technique. Consider a square with sides of length two centered at the origin, and a circle of radius one that inscribes it. The ratio of areas = $\pi/4$. Now, imagine drawing pairs of random numbers from a uniform distribution between -1 and 1. Take each pair as a coordinate and plot inside the square, keeping track of whether or not it is also inside the circle. After many pairs have been drawn, ask what percentage lie in the circle. You have just estimated π! It should be intuitive that the more trials you make, the better your estimate.

18.2 simple sampling

The example of computing π in the last section is an example of simple sampling Monte Carlo. One simply computes points in the integrand with uniform probability and sums the results. We'll see later that this is very inefficient in most interesting cases, but it's best to start with the simplest case.

In fact, why do we need a Monte Carlo (random) method at all? You probably remember from elementary calculus that a simple uniform set of grid points works just fine,

$$\int_{x_i}^{x_f} f(x) = \lim a \to 0 \sum_{i=0}^{N} f(x_i + ia) \tag{175}$$

$$a = \frac{x_f - x_i}{N} \tag{176}$$

A surprising problem arises because we are usually interested in very high dimension integrals resulting from problems with many (possibly internal) degrees of freedom. Binder has a nice example. Consider the XY spin model in 2d,

$$H_{XY} = -J \sum_{\langle i,j \rangle} (S_i^x S_j^x + S_i^y S_j^y) - h_x \sum_i S_i^X, \tag{177}$$

$$(S_i^x)^2 + (S_i^y)^2 = 1. \tag{178}$$

The spins live on sites of a regular lattice in space, say N points in all. At each site the spin can point in any direction, $\phi \in \{0, 2\pi\}$. Take $S_i^x = \cos(\phi_i)$ and $S_i^y = \sin(\phi_i)$. Then the simple 1d integral above becomes, in this case, a product of integrals, one at each site, $\Pi_i \int d\phi_i ...$ Let's say we break the integrals into p finite bits,

$$\phi_i^\gamma = (\gamma_i/p)2\pi$$

$$\gamma_i = 1, 2, \ldots, p$$

which defines a grid in our multi-dimensional space with total number of grid points p^N. Already this is a large number of points even for small p (exponential in N!). Even if we could somehow work with a relatively large p, another problem occurs: almost all of the points on our uniform grid lie on the *surface* of our integration hypercube, with almost none in the interior! In any direction the high dimensional space there are p points, $p-2$ in the "interior". The total fraction of points in the interior is then

$$\left(\frac{p-2}{p}\right)^N = \left(1-\frac{2}{p}\right)^N \tag{179}$$

$$= \exp\left(N\log\left(1-2/p\right)\right) \tag{180}$$

$$\approx \exp\left(-N2/p\right) \tag{181}$$

where the last line is valid for large N, and which vanishes at $N\to\infty$! The solution is to *choose the points randomly in a uniform way* over the entire domain, just like we did for the estimate of π. Note in this case though, the Monte Carlo method is indispensable.

To carry out the method in practice, we need a random number generator that can be included into our code. Such a generator is based on a recursion formula and is not actually completely random. Instead it is referred to as a *pseudo*-random number generator. In practice pseudo-random generators can be quite good in providing a sequence of random numbers, but there are pitfalls. Especially when the number of required random numbers is large as is usually the case.

18.2.1 random walks and random number generators

As an example where simple sampling is effective, and to look into random number generators in more detail, consider a simple random walk problem in two dimensions (exercise 3.3 in BH) which may be used as a model for understanding a simple

polymer. At each step the walker can go in 1 of 4 directions with uniform probability of 1/4. The idea is to compute the average end-to-end distance of a random walker(s) after say N steps, starting from the origin. Analytically we know in the large N limit that

$$\langle R^2 \rangle \quad = \quad N \tag{182}$$

To compute this average using simple sampling Monte Carlo, use the following algorithm (BH3.2),

1. do for sample := 1 to $N_{\rm samples}$
2. x:=0, y:=0
3. do step:= 1 to N
4. ir:= random(iseed) * 4
5. case ir

   0: x:=x+1

   1: y:=y+1

   2: x:=x-1

   3: y:=y-1
6. enddo
7. save end-to-end distance (X, Y)
8. enddo

The core of the calculation relies on the random number ir generated in step 4 which returns a random number between 0 and 1. BH provide a generator (BH3.3):

```
real function random(ibm)
   integer mult, modulo, ibm;
   real rmodulo;
   begin
       mult :=1277;
       modulo := 2^17;
       rmodulo :=modulo;
       ibm := ibm * mult;
       ibm := mod(ibm, modulo);
       random := ibm/rmodulo
    end
```

When first called, the function random takes the input "seed" to start the generation of the sequence of random numbers. It is subsequently modified by each call. The number 1277 and the exponent 17 are fixed and should not be changed. The idea of this "modulo" generator is the following "folding" mechanism which generates a sequence of numbers that appear to be highly unpredictable. Take four integers, m, a, c and x_0 with $m > x_0$ and a, c, $x_0 > 0$. The sequence of (pseudo) random numbers is generated by the following recursion relation,

$$x_i = (ax_{i-1} + c)\%m. \tag{183}$$

The success of the generator depends on the choice of m, a, and c. BH refer us to the literature as to what they should be. Essentially, a good choice results in a sequence with a long *period* before repeating. Let's see what happens in the case of

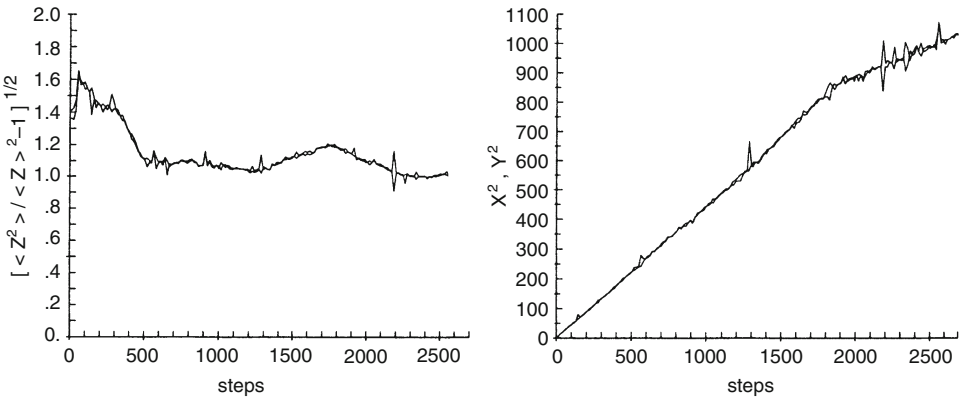

Fig. 3.2 *Left*: The error of the end-to-end distance, which should be constant, as a function of the step length. Here z stands for either x^2 or y^2, respectively. *Right*: The corresponding average end-to-end distances. The results were obtained using the random number generator of Algorithm 3.3

Figure 16:

our generator. See Fig. 16. Look at the right panel first. It looks ok for steps up to about 1500, then bad stuff happens! There is a kink. Apparently we've reached the period of the generator. The left panel is even more alarming. After a few hundred steps the error drops dramatically. It should be constant in this case. Some quantities are more sensitive.

The lesson is that random number generators should always be checked! preferably by several observables. Even if it is a "trusted" generator. Especially if your calculation needs lots of random numbers.

Going forward we will use canned generators like the one in random.h and ones in Python.

18.3 importance sampling

In many cases of interest simple sampling fails because the points in the integrand are distributed uniformly. For instance, many times we're interested in a thermal average or expectation value of some observable O (think of a gas of atoms in a box where the energy is sum of all pairwise interactions). Statistical mechanics tells us to average over all configurations with the weight $e^{-E/kT}$,

$$\langle O \rangle = \frac{\int [dx_0, \ldots, dx_N] O e^{-E/kT}}{\int [dx_0, \ldots, dx_N] e^{-E/kT}} \tag{184}$$

with coordinates (degrees of freedom) $x_0, \ldots, x_N$. In general both O and E depend on the coordinates. The denominator is recognized as the partition function, and the probability of a configuration with a given set of coordinates appearing is proportional to $e^{-E/kT}$.

Written as an integral over E with a density of states, the distribution of "energy points" in the integrand is the product of two exponentials, one falling and one rising which gives rise to a distribution peaked at the most probable energy which clearly depends on T.

In other words our integrand represents a non-uniform distribution. If we sample it uniformly, we waste almost all our time on points that don't contribute much to the integral!

The way out is to sample points according to the given distribution. This is known as "importance" sampling. This technique widely used in Monte Carlo "simulations". In the above, our integral becomes

$$\langle O \rangle = \frac{\sum_i^N O_i}{\sum_i^N 1} = \frac{1}{N} \sum_i^N O_i \tag{185}$$

where O_i is the observable calculated on the ith configuration, and we sum over N configurations generated according to the distribution $e^{-E/kT}$. Of course, this will

work for all kinds of systems like spin models for ferromagnets, quantum field theories in Euclidean spacetime, and so on.

While the sum we just wrote down looks very easy to compute, it's not that simple. We have to produce a set of configurations according to the desired probability distribution. We can't just write them down since that would amount to solving the problem in the first place! Luckily there is a theorem that guarantees the correct distribution can be obtained by following a simple procedure.

18.3.1 Markov processes

The procedure, or algorithm to generate the configurations is called a Markov process, and produces a Markov chain of configurations starting from an arbitrary one. Each configuration in the chain is given by a set of transition probabilities that takes us from the current configuration to the next. If they obey certain conditions, the following theorem guarantees that if the procedure is carried out many times, we end up generating configurations with the correct distribution.

The following two part theorem is proved in D. Toussaint, *Algorithms for simulations and their application to QCD*, but appears in most stat mech texts where the H theorem is proved. Capital letters A,B correspond to possible configurations. $P_n(A)$ is the probability to be in A at step n. P_∞ is the desired, or equilibrium distribution. $P(A \to B)$ is the probability to be in B at n+1 if at n we are in A.

The first assertion of the theorem is *if for all pairs A and B*

$$\frac{P(A \to B)}{P(B \to A)} = \frac{P_\infty(B)}{P_\infty(A)} \tag{186}$$

and at step n $P_n(A) = P_\infty(A)$ *for all* A, then at step n+1 $P_{n+1}(A) = P_\infty(A)$. To see

this, consider the probability of being in A at n+1 if at n we are in A,

$$\begin{aligned} P_{n+1}(A) &= P_n(A) - \sum_{B\neq A} P_n(A)P(A\to B) + \sum_{B\neq A} P_n(B)P(B\to A) \\ &= P_n(A) + \sum_{B\neq A} [P_n(B)P(B\to A) - P_n(A)P(A\to B)] \\ &= P_\infty(A) + \sum_{B\neq A} [P_n(B)P(B\to A) - P_\infty(B)P(B\to A)] \\ &= P_\infty(A) \end{aligned}$$

The second to last line comes from (186) and the last from $P_n(A) = P_\infty(A)$ for all A. (186) is nothing but the condition of *detailed balance*.

The 2nd assertion is that if detailed balance is satisfied, then at step n+1 the distribution P_{n+1} is "closer" (in a sense to be explained) to P_∞ than P_n. This part of the theorem requires another assumption: *ergodicity*, the condition that starting from any configuration, we can get to any other with finite probability. This does not mean $P(A\to B)$ is non-zero for all A,B, but that some (possibly very long) sequence of transitions can get from one to the other.

Let's define $\tilde{P}_n(A) = P_n(A)/P_\infty(A)$ and divide the second equation above by $P_\infty(A)$,

$$\tilde{P}_{n+1}(A) = \tilde{P}_n(A) + \sum_{B\neq A} [\tilde{P}_n(B)\frac{P_\infty(B)}{P_\infty(A)}P(B\to A) - \tilde{P}_n(A)P(A\to B)] \quad (187)$$

$$= \tilde{P}_n(A) + \sum_{B\neq A} P(A\to B)[\tilde{P}_n(B) - \tilde{P}_n(A)] \quad (188)$$

Now consider the configuration with the largest $\tilde{P}_n(A)$ which can be greater than one. The difference in the []'s is always negative and $P(A\to B)$ is always positive. At step $n+1$ $\tilde{P}(A)$ is therefore smaller. Likewise the smallest $\tilde{P}$ is bigger. The $\tilde{P}$'s

are squeezed to one! Another way of seeing this is to rearrange the above,

$$\tilde{P}_{n+1}(A) = \left(1 - \sum_{B \neq A} P(A \to B)\right) \tilde{P}_n(A) + \sum_{B \neq A} P(A \to B)\tilde{P}_n(B) \quad (189)$$

and note that $\tilde{P}_{n+1}$ is a weighted average of all $\tilde{P}(B)$'s with positive coefficients ($\sum P(A \to B) = 1$, including $A \to A$), so $\tilde{P}_{n+1}$ is less than the largest $\tilde{P}_n(A)$, or

$$\min_B \tilde{P}(B) \leq \tilde{P}_{n+1}(A) \leq \max_B \tilde{P}(B)$$

for all A. Again, squeezed to 1.

18.3.2 Metropolis method

A simple but effective update algorithm to realize the theorem in the previous section is the Metropolis update. Start with a current configuration, A and generate a *trial* configuration B'. We suggest the new configuration with probability

$$P_{\text{suggest}}(A \to B') = P_{\text{suggest}}(B' \to A).$$

For example, take the Ising model where the spin on a lattice site can either be "up" or "down" (± 1). We may suggest a new configuration by choosing a site randomly with uniform probability, and then flip it's spin (we need not choose a random site, we can also visit each site one-by-one, or in some other prescribed order). As another example consider a system with degrees of freedom that are real numbers (position, momentum, ...). Here we might suggest $x_i' = x_i + R(-\delta, \delta)$ where $R(-\delta, \delta)$ is a function that returns a random real number in the range $(-\delta, \delta)$.

Next, we *accept* the new configuration if $P(B') > P(A)$, so $B = B'$. However if $P(B') < P(A)$, we generate a random number R in the range (0,1) and accept with probability $P(B')/P(A)$. That is, if $R < P(B')/P(A)$, we accept $B' = B$, otherwise

we *reject*, $B = A$. Note that if we reject, we have to take the "new" configuration as the old one, in other words it appears successively in the Markov chain.

In the usual case where $P(A) \sim e^{-E(A)/kT}$, we always accept if the suggested configuration *lowers* the energy, and we accept conditionally those that *raise* the energy according to the exponential of the *energy difference*.

Now, $P(A \to B)$ is just the product of probabilities to suggest a configuration and then accept it. Thus, for $P(B) > P(A)$,

$$P(A \to B) = P_{\text{suggest}}(A \to B) \times 1 \tag{190}$$

$$P(B \to A) = P_{\text{suggest}}(B \to A)\frac{P(A)}{P(B)}. \tag{191}$$

Since the two P_{suggest}'s are the same by construction, detailed balance (186) is satisfied, and similarly for $P(B) < P(A)$.

The procedure just described is called an "update" of one variable (*e.g.* the spin at site i, s_i). Typically we "sweep" over the sites of a lattice in a regular order, updating the variable at each site as we go. We usually count configurations by one sweep of the entire lattice.

We also note that it is not necessary to suggest a new configuration by changing only one variable. However for models like the Ising model, the Hamiltonian is ultra-local: the change in energy of the entire configuration from flipping a single spin only depends on 2^d sites and is very cheap to compute.

18.3.3 refreshed molecular dynamics

For systems where the energy (Hamiltonian) is non-local, single dof updates like Metropolis become very inefficient. Molecular dynamics updates (with refreshing) are slow, but usually perform better in these cases.

Consider our earlier example expectation value (thermal average) of some observable O,

$$\langle O \rangle = \frac{\int [dx] O e^{-E(x)/kT}}{\int [dx] e^{-E(x)/kT}} \tag{192}$$

where the coordinates could be real coordinates, or spins, or even fields in an Euclidean QFT. Now introduce auxiliary "momenta" p_i, one for each coordinate,

$$\langle O \rangle = \frac{\int [dx] O e^{-E(x)/kT}}{\int [dx] e^{-E(x)/kT}} \frac{\int [dp] e^{-1/2 \sum_i p_i^2}}{\int [dp] e^{-1/2 \sum_i p_i^2}} \tag{193}$$

$$= \frac{\int [dx\, dp] O(x) e^{-(1/2 \sum_i p_i^2 + E(x)/kT)}}{\int [dx\, dp] e^{-(1/2 \sum_i p_i^2 + E(x)/kT)}} \tag{194}$$

(we just multiplied by 1). $1/2 \sum_i p_i^2 + E(x)/kT \to 1/2 \sum_i p_i^2 + V(x)$ looks like a hamiltonian for a non-relativistic system with coordinates x_i and momenta p_i. The fictitious hamiltonian satisfies Hamilton's equations of motion,

$$\dot{x}_i = p_i \tag{195}$$

$$\dot{p}_i = -\frac{\partial V(x)}{\partial x_i} \tag{196}$$

which can be integrated in *simulation time* using known techniques like leapfrog or velocity Verlet! (note: the momenta are in no sense real momenta, but momenta in simulation time).

Now we have to devise an algorithm that satisfies our fundamental theorem. Given a set of p's and x's called configuration A, try

1. flip sign of all momenta (at once) with probability 1/2

2. integrate Hamilton's equations for some amount of time

If we don't change sign of momenta, say, we have $A \to B$, or $P(A \to B) = 1/2$. On the other hand, if we are at B and do flip the sign, we get back to A! or $P(B \to$

$A) = 1/2 = P(A \to B)$. Notice the total "energy" is conserved (as is the density in phase space (Louiville's theorem)), so the fundamental theorem is satisfied. Except the algorithm is not ergodic! We never reach configurations with different "energy". This is easily fixed by adding in another type of update that does change the energy, like Metropolis updates on either the p's or x's, or we can *refresh* the momenta with a set of Gaussian random numbers (the update of the p_i is trivial since they are independent of each other and all x_i).

The refreshed molecular dynamics consists of a "heat bath" update of the momenta (set each to Gaussian random number) followed by integration of Hamilton's equations. No sign flip is necessary since the Gaussian distributed momenta are just as likely to generate $-p_i$ as p_i.

This kind of refreshed molecular dynamics is used in Lattice QCD to update the (non)abelian gauge fields. It is perhaps helpful to keep the following picture in mind (see Fig. 17) as the update algorithm moves us through the target space of variables. Too much refreshing results in a random walk which is not efficient. Not enough refreshing is also inefficient because we don't stray far from periodic orbits. One needs just enough to make large excursions in coordinate space.

18.3.4 auto-correlations in simulation time

So far we have discussed generation of ensembles of "configurations" that are Markov chains where the next configuration in the chain depends on the previous one. This necessarily leads to *auto-correlations* in simulation time of observables calculated on the ensemble of configurations making up the chain. Understanding these correlations are important, because the statistical error on observables depends on the number of *independent* measurements like $1/\sqrt{N}$.

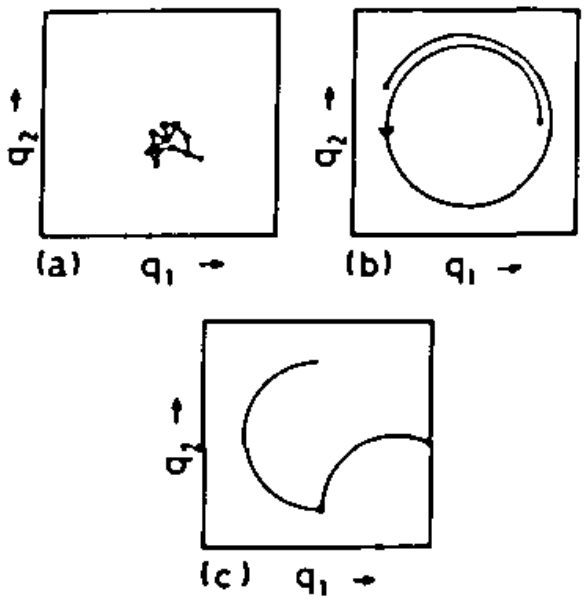

Fig. 5. Movement of a system through coordinate space in the refreshed molecular dynamics algorithm. (a) illustrates the effect of frequent refreshing – we do a random walk in coordinate space. (b) illustrates the effect of too little refreshing – we waste time in periodic or near periodic orbits. (c) is roughly optimal.

Figure 17: Reproduced from D. Toussaint, *Algorithms for simulations and their application to QCD.*

The (normalized) correlation can be obtained from

$$C_O(t) \;=\; \frac{\langle O(t+\tau)O(t)\rangle - \langle O(\tau)\rangle^2}{\langle O(\tau)^2\rangle - \langle O(\tau)\rangle^2} \tag{197}$$

where τ is the simulation time and t is the difference in simulation times. We usually count t in units of sweeps over the entire lattice as described above. In general correlation functions are sums of exponentials (*e.g.* the imaginary time GPE wave function in your homework!). For long times the correlation will be dominated by the mode with longest time constant,

$$C_O(t) \;\sim\; e^{-t/t_c}. \tag{198}$$

t_c for the slowest mode is taken at the auto-correlation time for O.

In general t_c is characteristic of the particular algorithm used to generate the chain, and depends on the slowest decaying mode in the difference of probability ensembles with the equilibrium ensemble $P - P_\infty$. An exception is an observable "protected" by a symmetry. For example, for Metropolis, the slowest update mode corresponds to flipping the sign of all the spins. In the Ising model, however, flipping the signs of all the spins leaves the energy unchanged, so the auto-correlation of the energy (and other even operators) can be shorter than for the magnetization.

In systems with local energy (like Ising), the auto-correlation time for Metropolis (and others) is related to the physical correlation length of the system, ξ, as

$$t_c \;=\; \xi^x \tag{199}$$

where $x \approx 2$. In the vicinity of a critical point, ξ diverges, and the algorithm slows down (critical slowing down). This is a big problem (for statistical noise) since we are usually interested in precisely this point!

18.3.5 Ising model

Let's take what we have learned in the last section and investigate the Ising model. The model is very simple, but captures many of the important features of a paramagnetic system, especially spontaneous symmetry breaking. The Hamiltonian is

$$H = -J \sum_{\langle i,j \rangle} s_i s_j - h \sum_i s_i \tag{200}$$

where J is a coupling constant. $J > 0$ describes ferromagnetic systems and $J < 0$, anti-ferromagnetic. The Hamiltonian as written could be in any dimension, but we will specialize to 2d where the analytic solution is known (Onsager, Yang, etc.) but very complicated.

The physics of the Ising model is well known... 2nd order phase transition, order parameter, etc...

The magnetization and susceptibility are computed from

$$M = \sum_i s_i \tag{201}$$

on each configuration. Ensemble averages for N configurations are

$$\langle M \rangle = \frac{1}{N} \sum_i M_i \tag{202}$$

$$\langle \chi \rangle = \frac{1}{N} \left(\sum_i M_i \right)^2 - \langle M \rangle^2 \tag{203}$$

Since these quantities are extensive, we should normalize them by the system size (number of sites).

Since I will ask you to code up the 2d Ising model, and compute several observables, a few comments are in order.

- We will use the Metropolis update, and 1 sweep is defined as visiting each site once, in order, to attempt to flip the spin at that site.

- Typically one starts the Markov chain from an ordered lattice: all spins up or down, which corresponds to $T = 0$. Alternatively on can start with a randomly disordered lattice (corresponding to $T = \infty$)

- Some number of sweeps must be discarded for thermalization. Usually the convergence to the equilibrium distribution is exponential away from the critical temperature (power-like near it). You can test how many to throw away empirically (see Fig. 19).

- The finite size effects are severe, especially near the critical point ($T = T_c$) so several lattice sizes will need to be investigated.

- All the interesting stuff happens near T_c. From the exact solution we know

$$J/kT_c = \frac{1}{2}\ln\left(1+\sqrt{2}\right) \tag{204}$$

- In the simulation, as usual, we work with dimensionless quantities. In this case they are J/kT and h. For the latter, your simulations should focus on values of T above and below T_c. Note there is no need to vary J and T separately: there is really only one dimensionless parameter.

- The statistical error on your results will depend on how many *independent* measurements you make on a particular quantity. Since there are auto-correlations, measurements should be made relatively seldom, *i.e.* not on consecutive configurations. After some separation in simulation time the measurements become

de-correlated. We can calculate the auto-correlation time in the system to optimize our results. In any case one must account for correlations when quoting a statistical error.

As mentioned, we will use the Metropolis update to solve the Ising model in 2d. (Following BH) At the heart is the accept-reject step which depends on the energy difference between the suggested configuration and the current one. At site i,

$$\begin{aligned} \Delta H &= E_{\text{suggest}} - E & (205)\\ &= -\frac{J}{kT}\left((-s_i)\sum_{nn} s_j - s_i \frac{J}{kT}\sum_{nn} s_j\right) & (206)\\ &= 2s_i \frac{J}{kT}\sum_{nn} s_j & (207) \end{aligned}$$

where all spins are evaluated for the current configuration. We accept with probability

$$W(\Delta H) = \min\{1, \exp -\Delta H\}. \quad (208)$$

In other words, draw a (pseudo) random number r between 0 and 1 and flip the spin if $r < W(\Delta H)$.

There is a nifty trick in BH: a lookup table for W. A little thought shows there are only 5 possible values of the nearest neighbor sum times s_i: $\pm 4, \pm 2, 0$. The first three values, -4, -2, 0 all give 1. The last two are exponentials. Simply calculate sum at each site, multiply by s_i and return the corresponding value! Note, you don't want to calculate exponentials at each step: precompute them!

Finally we need to say a word about boundary conditions.

Figure 20 shows the magnetization and susceptibility for a small lattice of $L = 5$ sites in each direction. I computed this using the Metropolis algorithm and a Jupyter notebook!

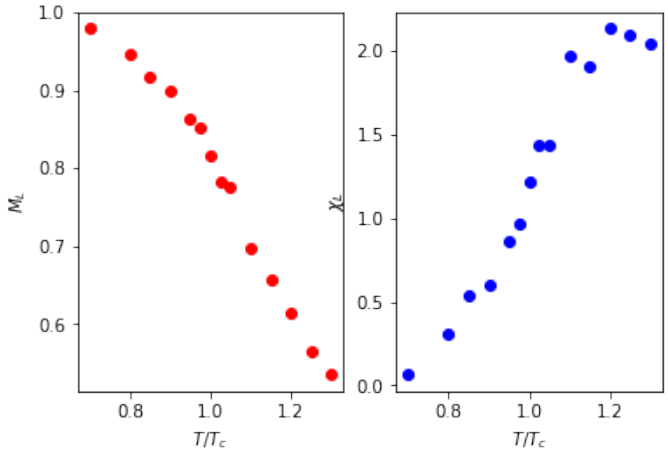

Figure 18: Magnetization and susceptibility in the 2d Ising model on a 5×5 site lattice. Statistical errors are not shown.

.

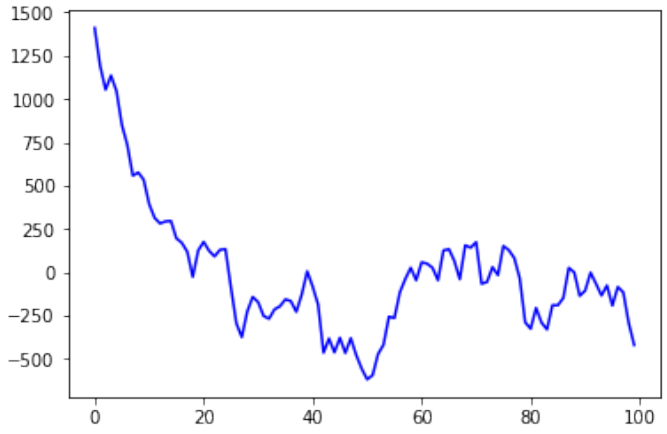

Figure 19: Thermalization of the magnetization at $T/T_c = 1.15$. $L = 40$.

.

What do you make of the following in Fig. 20?

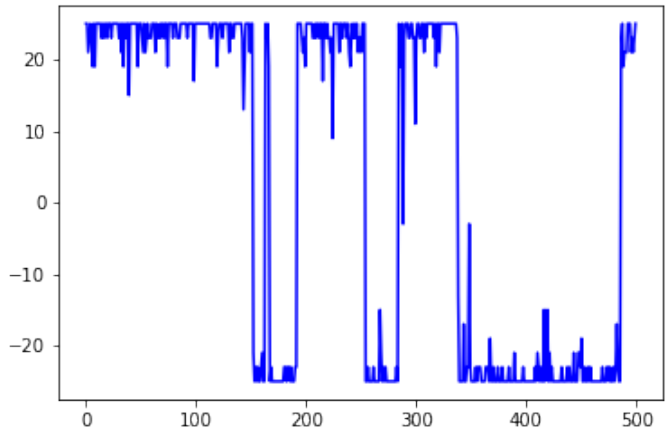

Figure 20: Magnetization at $T/T_c = 0.8$ every 20 Metropolis sweeps. $L = 5$.

.

18.3.6 dictionary of critical exponents

See for example, Statistical Mechanics by K. Huang, section 16.3.

Define the reduced temperature,

$$t \quad = \quad \frac{T - T_c}{T_c}. \tag{209}$$

As $t \to 0$ thermodynamic quantities contain finite parts (corrections) plus parts that diverge (or have divergent derivatives). The singular parts are proportional to t to some power, given by the corresponding critical exponent. See Tab. 1 for common examples.

Many experiments and theory indicate the critical exponents are the same for completely different physical systems with different critical temperatures (or coupling constants) that are in the same *universality class*. This phenomenon of universality is

Table 1: Critical exponents for various quantities.

Correlation length:	$\xi \sim \|t\|^{-\nu}$
Power-law decay:	$\lim_{t\to 0} \Gamma(r) = r^{-d+2-\eta} e^{-r/\xi}$
Heat capacity:	$C \sim \|t\|^{-\alpha}$
Order parameter:	$M \sim \|t\|^{\beta}$
Susceptibility:	$\chi \sim \|t\|^{-\gamma}$
Equation of state ($t = 0$):	$M \sim H^{-1/\delta}$

what makes critical exponents/scaling so powerful and interesting. The universality is not obvious until all the data are reduced to scaled variables. It turns out the exponents are not all independent: they satisfy various scaling laws.

Underlying the whole framework is the "scaling hypothesis". Dimensionful quantities are expressed in terms of a standard length scale and so change when the standard changes. Near T_c the assumption is that the *only* relevant (physical) length scale is the correlation length, in which all other lengths must be measured. Then physics can be cast in a scale-invariant form.

18.3.7 finite size scaling analysis

Following BH (sections 2.3.3 and 2.3.4), we will describe the general situation for thermally driven (2nd order) phase transitions with spontaneous symmetry breaking like the Ising model. As we have discussed, above T_c the magnetization (order parameter) vanishes and below the system spontaneously magnetizes. This is true only for an infinite system size (thermodynamic limit). For a finite system there is always finite probability that the magnetization will flip sign, and so the expectation of the

magnetization is always zero when computed on a finite lattice for all temperatures.

If we put the system in an external field H, it will magnetize, even for small H. Strictly we should define the magnetization as

$$M_{sp} = \lim_{H\to 0}\lim_{N\to\infty} M(T,H) \tag{210}$$

where the system size is taken to ∞ first then $H \to 0$.

Typically the problem is avoided by computing the rms value of the magnetization,

$$M_{\rm rms} = \sqrt{\langle M^2\rangle_T} = \left\langle \left(\sum S_i/N\right)^2 \right\rangle_T^{1/2} \tag{211}$$

$$= \frac{1}{N}\left\langle \sum_i \sum_j S_i S_j \right\rangle_T^{1/2} \tag{212}$$

where the correlation function is

$$G(\vec{r}_{ij}) = \frac{1}{N}\langle S_i S_j\rangle_T . \tag{213}$$

If we use periodic b.c.'s then the correlation function is translationally invariant, and

$$M_{\rm rms} = \left(\sum_i \langle S_i S_j\rangle_T^{1/2}/N\right)^{1/2} . \tag{214}$$

At the critical temperature, the correlation function scales like

$$\lim_{|\vec{r}_{ij}|\to\infty} G(\vec{r}_{ij}) = \hat{G}|\vec{r}_{ij}|^{-(d-2+\eta)} \tag{215}$$

where $\hat{G}$ is a critical amplitude and η is a critical exponent. In a finite system, and for $|\vec{r}_{ij} < L/2|$,

$$\sum_i \langle S_i S_j\rangle_T \propto \int_0^{L/2} r_{ij}^{d-1} dr_{ij} \langle S_i S_j\rangle_T \tag{216}$$

$$\propto \int_0^{L/2} r_{ij}^{-(d-2+\eta)+d-1} dr_{ij} \propto L^{2-\eta} \tag{217}$$

and the magnetization becomes for $N = L^d$,

$$M_{\rm rms}^{T=T_c} \propto (L^{2-d-\eta})^{1/2} \propto L^{-\beta/\nu} \tag{218}$$

where ν and β are other universal critical exponents, and we have used the scaling laws $2 - \eta = \gamma/\nu$ (Fisher), and $\mathrm{d}\nu = 2\beta + \gamma$ (Rushbrooke plus Josephson). The result for the magnetization is already similar to other scaling "laws" for completely different systems.

Let's look into the theory in more detail, first away from T_c ($L \gg \xi$), then near T_c. The probability distributions for the order parameter (call it s) for $T > T_c$ and $T < T_c$ ($H = 0$) are given by Gaussians

$$P_L(s) = L^{d/2} (2\pi k T \chi_L)^{-1/2} \exp\left[-s^2 L^d/(2kT\chi_L)\right] \tag{219}$$

and

$$P_L(s) = L^{d/2} (2\pi k T \chi_L)^{-1/2} \exp\left[-(s - M_L)^2 L^d/(2kT\chi_L)\right] \tag{220}$$

$$+ \; L^{d/2} (2\pi k T \chi_L)^{-1/2} \exp\left[-(s + M_L)^2 L^d/(2kT\chi_L)\right] \tag{221}$$

Notice that in the thermodynamic limit the width $\to 0$, and the expectation of the magnetization (order-parameter) is fixed to a definite value that does not fluctuate.

In the spontaneously broken phase, there is a small but finite value of P_L near $s = 0$ that allows the system to "tunnel" between the two minima $\pm M_L$. As $L \to \infty$ this becomes increasingly unlikely, and the system does not "sample" the full symmetric distribution but samples only around one or the other. Thus the expectation value of the order parameter becomes

$$\langle s \rangle' = \frac{\int_0^\infty ds\, s P_L(s)}{\int_0^\infty ds\, P_L(s)} = \langle |s| \rangle. \tag{222}$$

and we expect

$$\lim_{L\to\infty} M_L \;=\; \lim_{L\to\infty} \langle |s| \rangle_L = \lim_{L\to\infty} \langle s^2 \rangle_L^{1/2} = M_{\rm sp} \tag{223}$$

In your home work for 2d Ising model you can use $\langle |s| \rangle_L$.

The susceptibility is given from the fluctuation-dissipation theorem and the peak heights (half-width at max),

$$\lim_{L\to\infty} \frac{\langle s^2 \rangle L^d}{kT} \;=\; \lim_{L\to\infty} \frac{P_L^{-2}(0) L^d}{2\pi kT} = \lim_{L\to\infty} \frac{(\Delta s)^2 L^d}{8kT \ln 2} = \chi \tag{224}$$

and

$$\lim_{L\to\infty} \frac{\langle s^2 \rangle - \langle |s| \rangle^2}{kT} L^d \;=\; \lim_{L\to\infty} \frac{P_L^{-2}(M_L) L^d}{2\pi kT} = \lim_{L\to\infty} \frac{(\Delta s)^2 L^d}{8kT \ln 2} = \chi \tag{225}$$

Since usually we don't know what the correlation length is, we don't know *a priori* if $L \gg \xi$. But we can get a handle on the Gaussian character of the distribution by studying the 4th order cumulant,

$$U_L \;=\; 1 - \frac{\langle s^4 \rangle_L}{3 \langle s^2 \rangle_L^2} \tag{226}$$

which *vanishes* for a Gaussian distribution and $L \to \infty$. In fact one can show that $U_L \propto L^{-d} \to 0$ for $T > T_c$ and Ł $\gg \xi$. For $T < T_c$ $\lim_{L\to\infty} U_L = 2/3$. On the other hand when $\xi \gg L$ U_L depends weakly on T and L, and remains close to the fixed-point U^*. The behavior of U_L makes it very useful in estimating T_c in simulations, independent of any assumptions about critical exponents. It turns out one may simply plot ratios of U_L between different L for several pairs and read off T_c from their universal crossing point.

Near the critical point, the correlation length scales like $\xi \propto |1 - T/T_c|^{-\nu}$. When $L \sim \xi$, $P_L(s)$ is far from Gaussian. However we can write physical quantities in terms of the *scaling* variable L/ξ and s instead of s, ξ and L separately (note T

enters through ξ). This not obvious but results from the behavior of the system at criticality where fluctuations appear on all scales (the physics behind the math of the renormalization group; at the critical point physics is scale invariant).

$$P_L(s) = \xi^{\beta/\nu} P\left(L/\xi, s\xi^{\beta/\nu}\right) = L^{\beta/\nu}\tilde{P}\left(L/\xi, sL^{\beta/\nu}\right), \quad (227)$$

$$sL^{\beta/\nu} = s\xi^{\beta/\nu}(L/\xi)^{\beta/\nu}, \quad (228)$$

i.e., there are power-law prefactors. As $L, \xi \to \infty$ but L/ξ finite, the above becomes arbitrarily accurate. Then the usual finite-size scaling relations result,

$$\langle|s|\rangle_L = L^{-\beta/\nu}\tilde{M}(L/\xi) \quad (229)$$

$$\chi'(L,T) \equiv L^d \frac{\langle s^2\rangle_L - \langle|s|\rangle_L^2}{kT} = L^{\gamma/\nu}\tilde{\chi}(L/\xi), \quad (230)$$

$$U_L = 1 - \frac{\tilde{\chi}_4(L/\xi)}{3(\tilde{\chi}_2(L/\xi))^2}, \quad (231)$$

where

$$\langle s^2\rangle_L = L^{-2\beta/\nu}\tilde{\chi}_2(L/\xi), \quad (232)$$

$$\langle s^4\rangle_L = L^{-4\beta/\nu}\tilde{\chi}_4(L/\xi), \quad (233)$$

and χ' approaches the standard susceptibility in the thermodynamic limit for $T < T_c$, and $U^* = 1 - \tilde{\chi}_4(0)/3(\tilde{\chi}_2(0))^2$.

18.3.8 finite size and relaxation time in MC simulation

Following section 2.3.5 in BH. As previously mentioned, the relaxation time in our MC simulation (Markov process) grows like a positive power of the correlation length,

$$\tau \propto \xi^z \propto |1 - T/T_c|^{-\nu z}. \quad (234)$$

Close to T_c and for finite L this becomes $\tau \propto L^z$. This means the statistical error on the magnetization, for example, will take its maximum near T_c. To see this recall,

$$\langle \delta_A^2 \rangle = \frac{1}{n} \left(\langle A^2 \rangle - \langle A \rangle^2 \right) \left(1 + 2 \frac{\tau_A}{\delta t} \right) \tag{235}$$

where n is the number of measurements of A, τ_A its relaxation time and δt the simulation time between measurements. If the "observation time" is much greater than the relaxation time (time for independent measurement), $t_{\rm obs} = n\delta t \gg \tau_A$,

$$\langle \delta_A^2 \rangle \approx 2 \left(\langle A^2 \rangle - \langle A \rangle^2 \right) \frac{\tau_A}{t_{\rm obs}}. \tag{236}$$

Then for the magnetization at T_c,

$$\langle \delta M^2 \rangle = \frac{2\tau_{max}}{t_{\rm obs}} \left(\langle M^2 \rangle - \langle M \rangle^2 \right), \tag{237}$$

$$= \frac{2\tau_{max}}{t_{\rm obs}} \frac{\chi' k T_c}{L^d} \propto \frac{L^{z+\gamma/\nu-d}}{t_{\rm obs}}. \tag{238}$$

On your scalar machine, compute time for a MC sweep over the lattice scales like the size, L^d. Plugging in the known critical exponents, this time scales like $L^{z+\gamma/\nu} \approx L^4$ for $d \leq 4$. So an increase in L by a factor of ten is 10000 times more expensive! (for the same error, or $t_{\rm obs}$). Using parallel algorithms will become mandatory for realistic simulations.

It turns out that for $T \geq T_c$ $\tau_{\rm max}$ is simply the largest relaxation time in the system. Below T_c it's more complicated (see BH). The upshot for us is that below T_c (ordered phase) we should start our Markov chain from an ordered configuration.

18.3.9 finite size and statistical errors in MC simulation

Again, following BH (section 2.3.8). In a MC simulation we calculate statistical errors on a quantity A via

$$\Delta(n, L) = \frac{1}{\sqrt{n}} \sqrt{\langle A^2 \rangle_L - \langle A \rangle_L^2} \tag{239}$$

where n is the number of *independent* measurements of A in our simulation. The central question is how the error depends on L? One might imagine that as $L \to \infty$ the error *vanishes*. If it reaches an L-independent limit instead, we say A lacks the property of *self-averaging*. If it does vanish like

$$\langle A^2 \rangle_L - \langle A \rangle_L^2 \propto L^d \tag{240}$$

we say it is strongly self-averaging. If it vanishes more slowly, we say it is weakly self-averaging.

From statistical mechanics you may remember that densities like energy or magnetization per site (atom) exhibit Gaussian distributions for $\delta A = A - \langle A \rangle$ (for $L \gg \xi$),

$$P_L(\delta A) \;=\; L^{d/2}(2\pi C_A)^{-1/2} \exp\left[-(\delta A)^2 L^d / 2C_A\right] \tag{241}$$

where C_A controls the width of the distribution. $C_A = kT\chi$ and kT^2C for the magnetization and energy, respectively. Computing $\langle (\delta A)^2 \rangle$, we find

$$\langle (\delta A)^2 \rangle \;=\; \langle A^2 \rangle - \langle A \rangle^2 = C_A / L^d, \tag{242}$$

i.e. strong self-averaging, assuming $\lim_{L\to\infty} C_A$ exists. However when the quantity is not a density, but something that is derived from fluctuations like a susceptibility or heat capacity ($\delta A^2 L^d$), we find a different result,

$$\Delta A(n, L) \;=\; \frac{L^d}{\sqrt{n}} \sqrt{\langle \delta A^4 \rangle - \langle \delta A^2 \rangle^2}. \tag{243}$$

For a Gaussian-distributed quantity, the fourth-order cumulant vanishes, so $\langle \delta A^4 \rangle = 3\langle \delta A^2 \rangle^2$, and

$$\Delta A(n, L) \;=\; \frac{L^d}{\sqrt{n}} \langle \delta A^2 \rangle \sqrt{2} = C_A \sqrt{2/n}. \tag{244}$$

So the relative error depends only on n. The upshot is that increasing L on quantities like the energy and magnetization will strongly improve (reduce) the statistical error, but has no effect on susceptibilities and the like. For the latter, it is best to choose L as small as possible and increase n.

Interestingly we can consider computing these quantities not by sampling fluctuations, but by taking numerical derivatives. For example, assuming $\Delta H \ll 1$,

$$\chi \approx \frac{M(H + \Delta H) - M(H)}{\Delta H} \tag{245}$$

and assuming the errors on M are much smaller than the difference,

$$\frac{\Delta\chi}{\chi} \approx \sqrt{2}\frac{\Delta_M(n, L)}{\Delta H \chi} = \sqrt{2}\sqrt{\frac{kT\chi}{L^d}}\frac{1}{\Delta H \chi}, \tag{246}$$

$$= \sqrt{\frac{2kT}{L^d\chi}}\frac{1}{\Delta H}, \tag{247}$$

which is strongly self-averaging!

Finally, what about when $\xi \gg L$? Then we can not assume Gaussian-distributed fluctuations, and must use the critical ones instead. For example, from above,

$$\Delta M = \sqrt{\frac{\langle M^2\rangle - \langle|M|\rangle^2}{n}} \tag{248}$$

$$= \frac{\sqrt{\tilde{\chi}(L/\xi)}}{\sqrt{n}L^{\beta/\nu}} \tag{249}$$

It turns out the $\tilde{\chi}(0)$ is finite so the magnetization exhibits weak self-averaging near the critical temperature since $\beta/\nu \ll d$. The susceptibility again lacks self-averaging,

$$\frac{\Delta\chi}{\chi} = \sqrt{\frac{2 - 3U_L(L/\xi)}{n}}. \tag{250}$$

18.3.10 Determining critical exponents

In practice determining critical exponents is difficult. Typically on tries to fit all of the reduced data near T_c to a set of scaling functions, for example,

$$x \equiv (T - T_c)L^{1/\nu} \tag{251}$$

$$y \equiv m(T, L)L^{\beta/\nu} \tag{252}$$

As a check, you can plot your data for the 2d Ising model according to the above with the known critical exponents $\nu =$ and $\beta =$.

19 Parallel high performance computing

19.1 Multprocessing (MP)

We are going to watch a series of YouTube videos by one of the inventors of openMP, Tim Mattson from Intel. There are several exercises to get us started. As you'll see, omp is straightforward to implement, mostly using compiler directives (#pragma omp ...). A set of presentation slides for the videos can be found here.

There are runtime library functions too, so to compile your openMP code, you'll need a later version of gcc/g++ (simple), or the later llvm libraries if you use clang on macOS. For example, the following worked on my imac/macbook running macOS 10.13.6.

```
Thomass-iMac:parallel tblum$ brew install llvm
...
Thomass-iMac:parallel tblum$ clang++ hello.C -L/usr/local/opt/llvm/lib -lomp
Thomass-iMac:parallel tblum$ clang hello_c.c -L/usr/local/opt/llvm/lib -lomp
```

```
Thomass-iMac:parallel tblum$ g++ hello.C -fopenmp
Thomass-iMac:parallel tblum$ gcc hello_c.c -fopenmp
```

The serial Pi program looks like this,

```
 static long num_steps = 100000;
 double step;
 int main (){
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   for (i=0;i< num_steps; i++){
   x = (i+0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
 }
  pi = step * sum;
}
```

19.2 Message Passing Interface (MPI)

19.3 GPU programming

References