Physics 5621 Computational Physics Lecture 1

T. Blum

September 12, 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 #%&!=. 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:

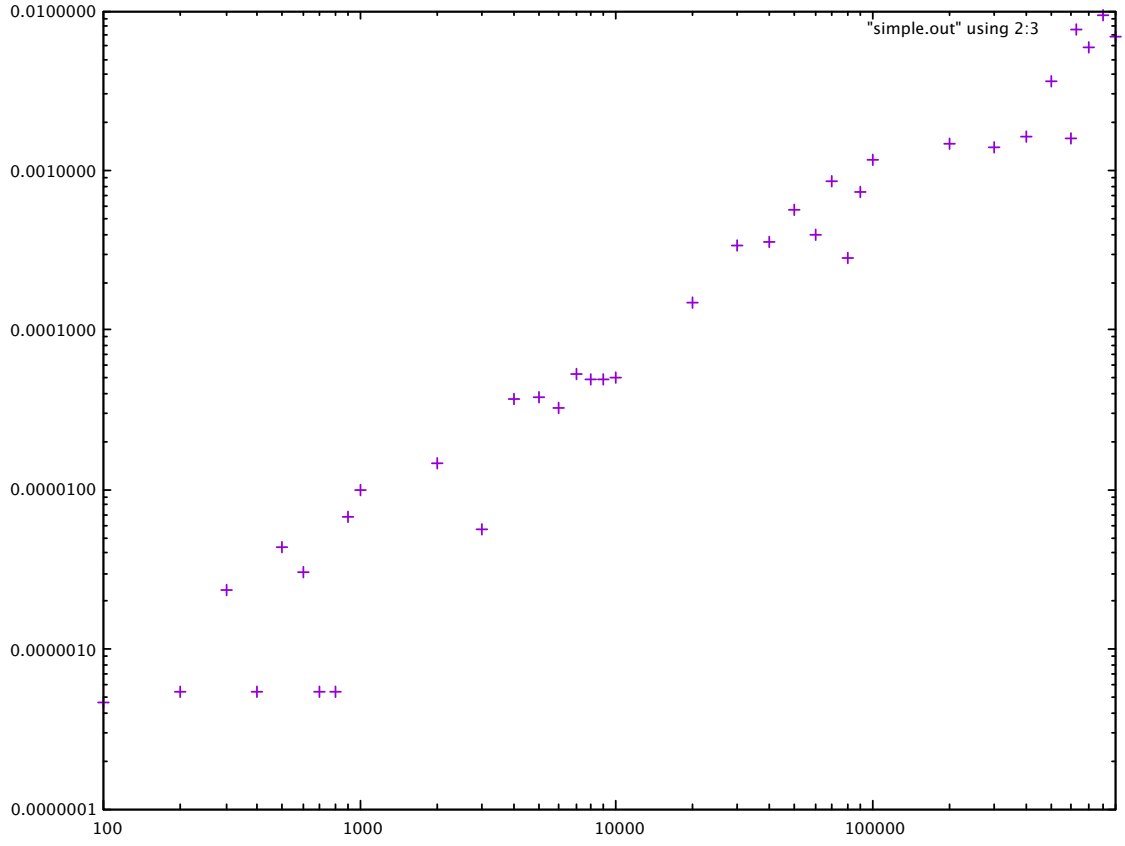

Figure 1: simple plot produced by gnuplot.

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
```

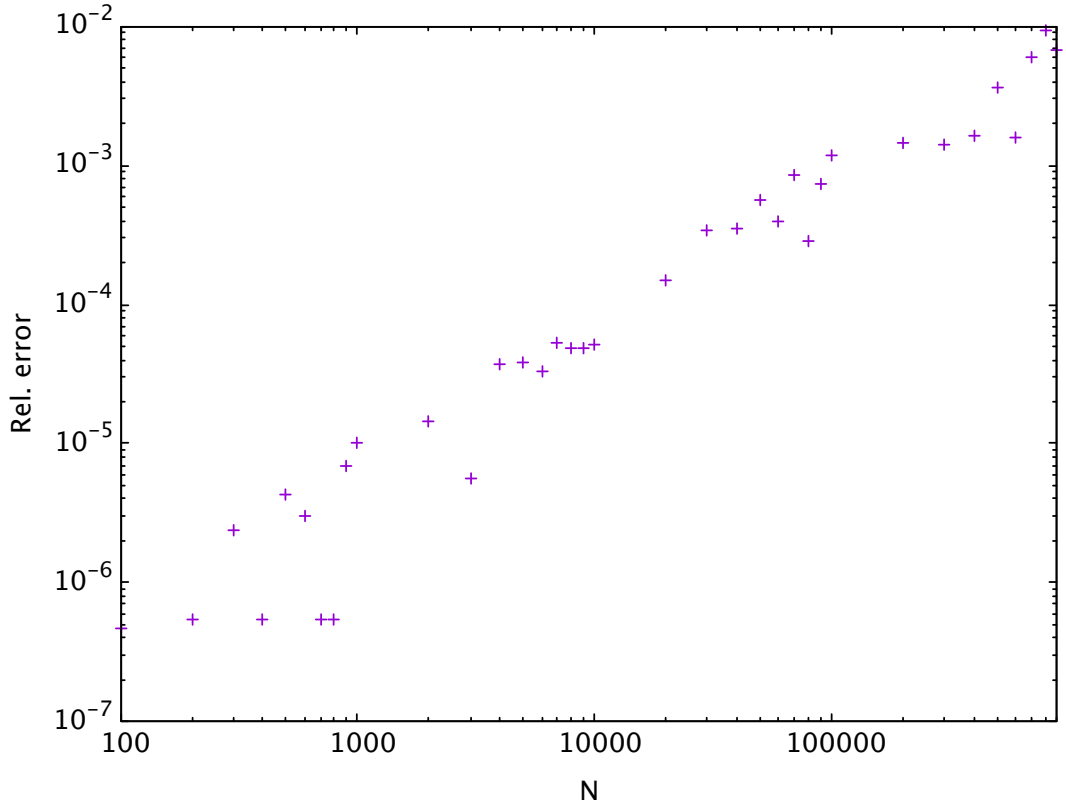

Figure 2: Prettier plot using various controls in gnuplot (see text).

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'
```

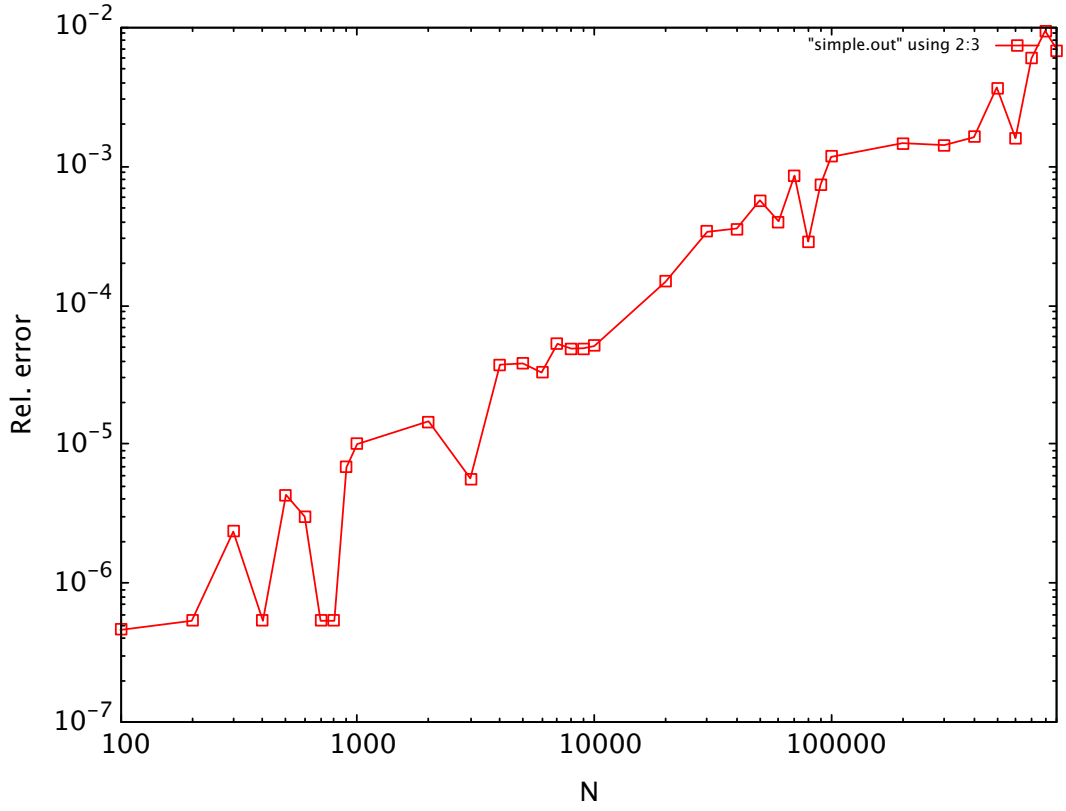

Figure 3: Prettier plot using load command.

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 $a = 0.025$:

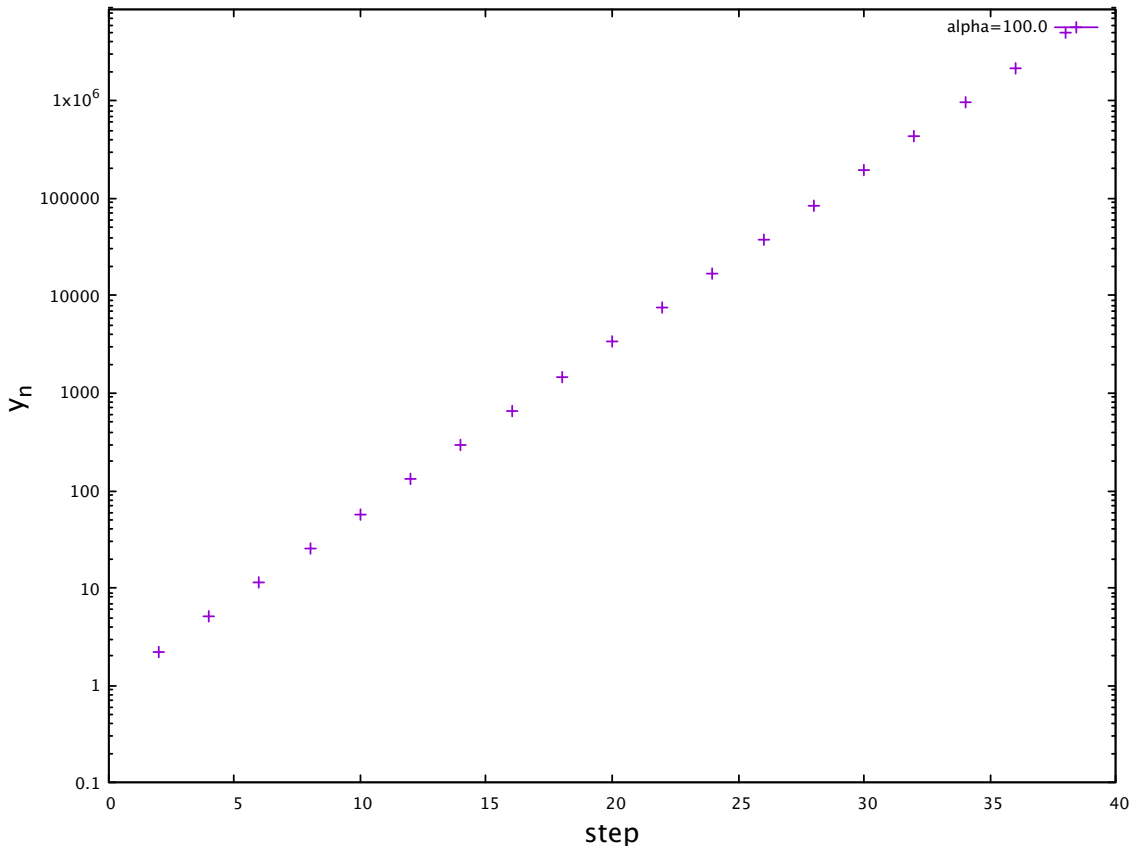

Figure 4: Euler method for solution of $y' = -100.0y$ with lattice spacing $a = 0.025$. The solution diverges exponentially (only even steps shown, odd are less than zero).

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%.

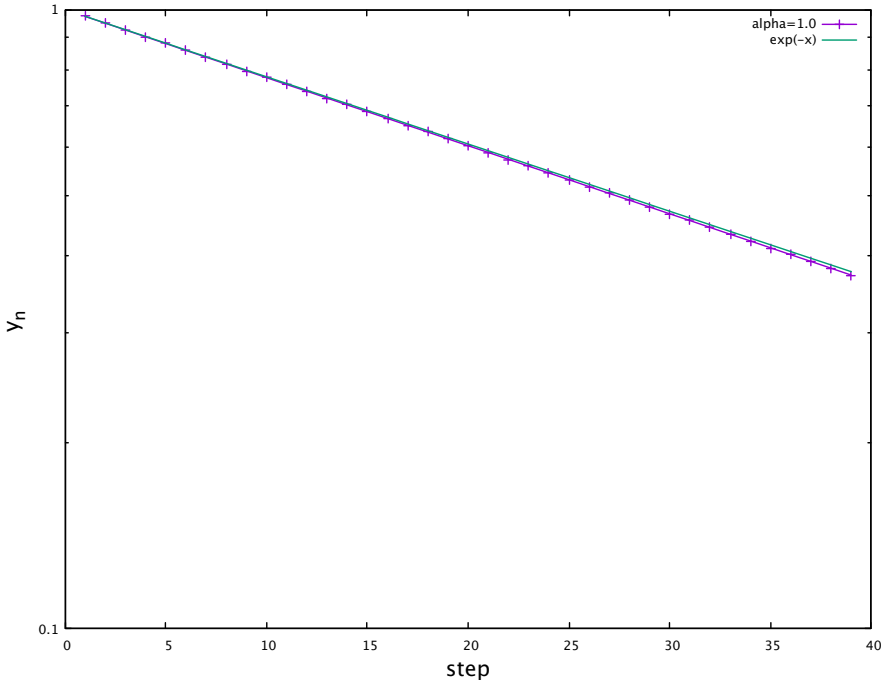

Figure 5: Euler method for solution of $y' = -1.0y$ with lattice spacing $a = 0.025$. The exact solution is also shown.

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 call Runge-Kutta invented in 1901 (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,

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

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

(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%!

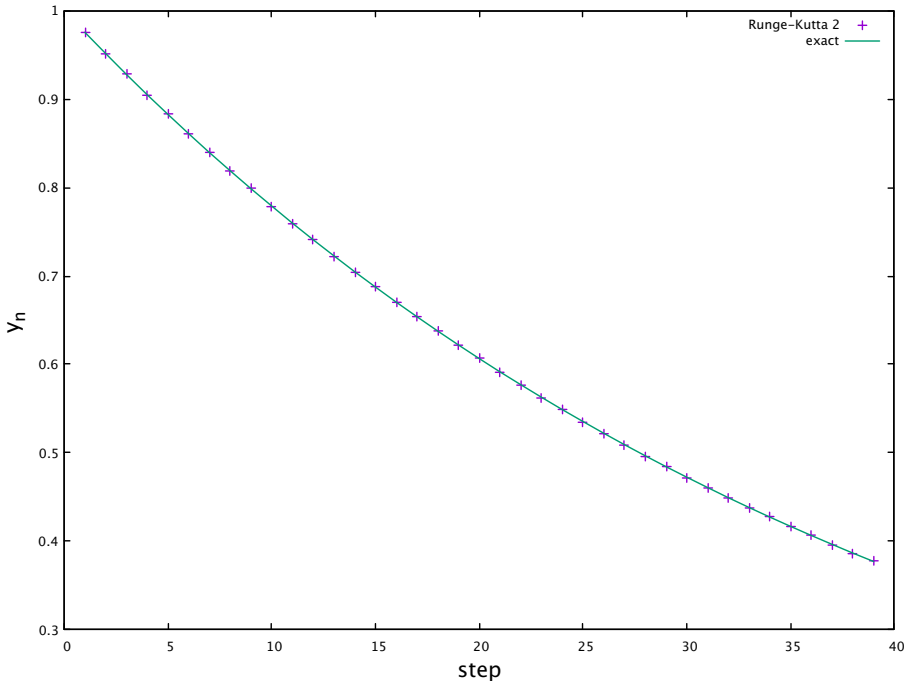

Figure 6: 2nd order Runge-Kutta method for solution of $y' = -1.0y$ with lattice spacing $a = 0.025$. The accuracy at the last step is $O(a)$ smaller than for Euler.

We can do much better with a few more flops. 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 with more weight. Graphically it looks like this.

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:

$$\begin{aligned}
\Delta_+ &\equiv f(x+a) - f(x) && (24)\\
\Delta_+ &\equiv f(x) - f(x-a) && (25)\\
\Delta_-\Delta_+ f(x) &= \Delta_- \frac{(f(x+a) - f(x))}{a} && (26)\\
&= \frac{f(x+a) - f(x) - f(x) + f(x-a)}{a^2} && (27)\\
&= \frac{f(x+a) + f(x-a) - 2f(x)}{a^2} && (28)
\end{aligned}$$

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

$$\begin{aligned}
f(x \pm a) &= f(x) + af' + \frac{a^2}{2} f'' \pm O(a^3) && (29)\\
f(x+a) + f(x-a) - 2f(x) &= a^2 f'' + O(a^4). && (30)
\end{aligned}$$

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'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).

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

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).

10.2 velocity Verlet

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

$$F = m\ddot{x} \tag{35}$$

$$= m\dot{v} \tag{36}$$

$$v = \dot{x} \tag{37}$$

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

$$x_{t+a} = x_t + av_t + \frac{a^2}{2}a_t \tag{38}$$

$$v_{t+a} = v_t + a\frac{a_t + a_{t+a}}{2} \tag{39}$$

where again $a(t) = F(t)/m$.