Libraries

Ramses van Zon

PHY1610, Winter 2023

Libraries

Code is bad

There is a big different in the way scientists view code and the way software developer view it.

Scientists	Software developer
Code is an asset.	**Code is a liability.**

- Every line of code you write has potential issues now or in the future and needs to be maintained.
- Scientists will often come up with quick and dirty solutions to get results, which causes headaches later in the development process: Technical Debt.
- Furthermore there is a lot of code that has already been written and that can be reused, so you might be reinventing the wheel.

The solution is to code less!

Reuse and recycle code that is out there by using **libraries**.

Libraries are modules

- So let's start with a modular code.
- Several object files for different modules that need to be linked together.
- Example: `thisapp.cpp` contains the main function and `helper.cpp/helper.h` are a module.

```
# makefile for 'thisapp'
CXX=g++
CXXFLAGS=-O2 -std=c++17
all: thisapp

thisapp.o: thisapp.cpp helper.h
  $(CXX) $(CXXFLAGS) -c -o thisapp.o thisapp.cpp

helper.o: helper.cpp helper.h
  $(CXX) $(CXXFLAGS) -c -o helper.o helper.cpp

thisapp: thisapp.o helper.o
  $(CXX) -o thisapp thisapp.o helper.o
```

- To reuse the module, copy `helper.cpp/.h`
- What if we could use it in another project called without recompiling `helper.cpp`?
- Install .o and .h to separate directories:
  `helper.h -> /base/include/helper.h`
  `helper.o -> /base/lib/helper.o`
- Must let compiler know where they are: Add `-I` flag for include directories.

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O2 -std=c++17
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) -o newapp newapp.o /base/lib/helper.o
```

Libraries, continued

What we just did is a poor man's library building.

Real libraries are similar; they have

- to be installed (and perhaps built first)
- header files (.h or .hpp) in some folder
- library files (object code) in a related folder.

Library filenames start with `lib` & end in `.a`/`.so`.

To avoid explict paths in makefile rules, we specify:

- the path to the library's object using the `-L` option in the `LDFLAGS` variable;
- the object code using `-lNAME` (a lower case `l`!) stored in variable `LDLIBS`.

We're not getting into creating your own libraries here, which requires some system-dependent specialized linking commands.

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O2 -std=c++17
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) -o newapp newapp.o /base/lib/libhelper.a
```

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O2 -std=c++17
LDFLAGS=-L/base/lib
LDLIBS=-lhelper
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) $(LDFLAGS) -o newapp newapp.o $(LDLIBS)
```

Libraries, once more

Adding a clean rule and extracting the common path, the Makefile for newapp will look like this:

```
# makefile for 'newapp'
CXX=g++
HELPERBASE?=/base/
HELPERINC=$(HELPERBASE)include
HELPERLIB=$(HELPERBASE)lib
CXXFLAGS=-I$(HELPERINC) -O2 -std=c++17
LDFLAGS=-L$(HELPERLIB)
LDLIBS=-lhelper

all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) $(LDFLAGS) -o newapp newapp.o $(LDLIBS)

clean:
  $(RM) newapp.o
```

Note:

- C++ standard libaries (`vector`,`cmath`,`...`) do not need any `-l...`'s.
- There are standard directories for libraries that needn't be specified in `-I` or `-L` options (`/usr/include,...`)
- Libraries installed through a package manager end up in standard paths; they just need `-l...` options in LDLIBS.
- You also do not need `-I` or `-L` for libraries accessed using the 'module load' command on the Teach or Niagara clusters.
- If you compile your own libraries in non-standard locations, you do need `-I` and `-L` options.

Installing libraries from source

What to do when your package manager does not have that library, or you do not have permission to install packages in the standard paths?

Or, what if you are on SciNet systems (where you do not have permissions to install using the package manager) and there isn't a module for that library already?

Compile from source code with a "base" or "prefix" directory.

Common installation procedure (but read documentation!):

```
$ ./configure --help
$ ./configure --prefix=<BASE>
$ make -j 4
$ make install
```

```
$ mkdir builddir && cd builddir
$ cmake .. -DCMAKE_INSTALL_PREFIX=<BASE>
$ make -j 4
$ make install
```

You choose the <BASE>, but it should be a directory that you have write permission to, e.g., a subdirectory of your **$HOME**. These are "non-standard" installation directories.

If the documentation says to do **sudo**, it is wrong except for system-wide installations on personal computers.

Using Libraries

- Include its header file(s) in your code.
- Link with `-lLIBNAME`.
- Non-standard installation directory? You need `-I<BASE>/include` and `-L<BASE>/lib` options.
- Alternatively, you can omit these for g++ under linux by setting some environment variables:

```
export CPATH="$CPATH:<BASE>/include"                   # compiler looks here for include files
export LIBRARY_PATH="$LIBRARY_PATH:<BASE>/lib"         # and here for library files
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:<BASE>/lib" # runtime linker looks here
```

  Eenter these commands on the linux prompt before `make` or add to your `~/.bashrc`.

- The `LD_LIBRARY_PATH` is necessary to run application linked against dynamic libraries (`.so`).
- If the library installs binary applications (i.e. commands) as well, you'll also need to set

```
export PATH="$PATH:<BASE>/bin"      # linux shell looks for executables here
```

- **Read the documentation** that came with the library (before searching the web)!

Library Example: GNU Scientific Library

GNU Scientific Library (GSL)

Is a C library containing many useful scientific routines, such as:

- Root finding
- Minimization
- Sorting
- Integration, differentiation, interpolation, approximation
- Statistics, histograms, fitting
- Monte Carlo integration, simulated annealing
- ODEs
- Polynomials, permutations
- Special functions
- Vectors, matrices

Note: C library means we'll likely need to deal with some pointers and casts.

GSL root finding example

Suppose we want to find where $f(x) = a\cos(\sin(v + wx)) + bx - cx^2$ is zero (a *"root"*).

```
// gslrx.cpp
#include <iostream>
#include <gsl/gsl_roots.h>

struct Params {
 double v, w, a, b, c;
};
double examplefunction(double x, void* param){
 Params* p = reinterpret_cast<Params*>param;
 return p->a*cos(sin(p->v+p->w*x))+p->b*x-p->c*x*x;
}

int main() {
 double x_lo = -4.0;
 double x_hi = 5.0;
 Params args = {0.3, 2/3.0, 2.0, 1/1.3, 1/30.0};
 gsl_root_fsolver* solver;
 gsl_function      fwrapper;
 solver = gsl_root_fsolver_alloc(
              gsl_root_fsolver_brent);
 fwrapper.function = examplefunction;
 fwrapper.params = &args;
 gsl_root_fsolver_set(solver,&fwrapper,x_lo,x_hi);

 std::cout << "iter lower upper root err\n";

 int status = 1;
 for (int iter=0; status and iter < 100; ++iter) {
   gsl_root_fsolver_iterate(solver);
   double x_rt = gsl_root_fsolver_root(solver);
   double x_lo = gsl_root_fsolver_x_lower(solver);
   double x_hi = gsl_root_fsolver_x_upper(solver);
   std::cout << iter <<" "<< x_lo <<" "<< x_hi
             <<" "<< x_rt <<" "<<x_hi-x_lo<<"\n";
   status=gsl_root_test_interval(x_lo,x_hi,0,1e-3);
 }

 gsl_root_fsolver_free(solver);
 return status;
}
```

Compilation and linkage

- Lots of `gsl...` stuff.
- All of the algorithms come from the GSL.
- The rest is just wrappers, setting up parameters and calling the appropriate functions.
- There are pointers and typecasts, because we're dealing with a C library.
- ***How to compile on the command line?***

```
$ module load gcc/12 gsl/2.7.1
$ GSLINC=$MODULE_GSL_PREFIX/include
$ GSLLIB=$MODULE_GSL_PREFIX/lib
$ g++ -c -I$GSLINC gslrx.cpp -o gslrx.o
$ g++ gslrx.o -o gslrx -L$GSLLIB -lgsl -lgslcblas
$ ./gslrx
```

Output

```
$ ./gslrx
iter lower      upper    root      err
0    -4        -1.27657 -1.27657  2.72343
1    -1.95919  -1.27657 -1.95919  0.682622
2    -1.75011  -1.27657 -1.75011  0.473542
3    -1.75011  -1.74893 -1.74893  0.0011793
$
```

GSL Makefile usage

```
CXX=g++
GSL_MODULE_PREFIX?=.
GSLINC?=$(MODULE_GSL_PREFIX)/include
GSLLIB?=$(MODULE_GSL_PREFIX)/lib
CXXFLAGS=-I$(GSLINC) -O2 -std=c++17
LDFLAGS=-L$(GSLLIB)
LDLIBS=-lgsl -lgslcblas

all: gslrx
.PHONY: all clean

gslrx.o: gslrx.cpp
  $(CXX) $(CXXFLAGS) -c -o gslrx.o gslrx.cpp

gslrx: gslrx.o
  $(CXX) $(LDFLAGS) -o gslrx gslrx.o $(LDLIBS)

clean: ; $(RM) gslrx.o
```

Compilation on Teach cluster:

```
$ module load gcc/12 gsl/2.7.1
$ make
```

Compilation on your own computer:

```
$ export GSLINC=... # whereever headers are
$ export GSLLIB=... # whereever libs are
$ make
```

or

```
$ export MODULE_GSL_PREFIX=... # with include & lib
$ make
```

You can also set make variables like this:

```
$ make MODULE_GSL_PREFIX=... # with include & lib
```

Don’t Reinvent the Wheel

- There are many possible algorithms to implement for root finding.
- But they are all pretty standard.
- Surely, someone must have done this already? Correct!
- The GNU Scientific Library is one such library.
- Don’t implement this yourself if there is a library that does it for you.
- Even existing solutions like the once in the GSL, can’t really be used until you understand the algorthims on a high level.

Digression: Root Finding

Root finding

- It is not uncommon in scientific computing to want solve an equation numerically.
- If there is one unknown and one equation only, we can always write the equation as

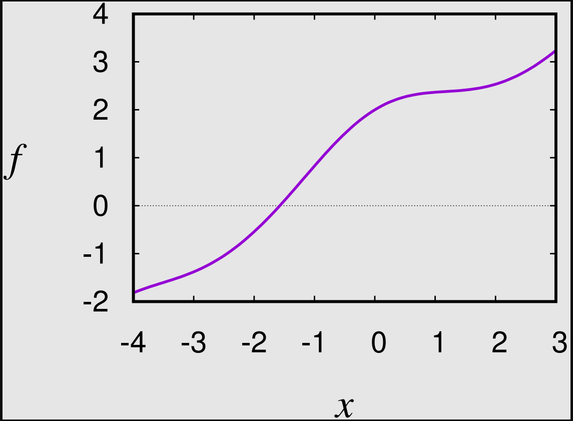
$$f(x) = 0$$

- If x satisfies this equation, it is called a “root’ ’.
- If there’s a set of equations, one can write:

$$\begin{aligned} f_1(x_1, x_2, x_3, \ldots) &= 0 \\ f_2(x_1, x_2, x_3, \ldots) &= 0 \\ &\ldots \end{aligned}$$

The one-dimensional case is considerably easier to solve: First.

1D Root Finding



Algorithms always start from an initial guess and (usually) a bounding interval $[a, b]$ in which the root is to be found.

What's so nice about 1D?

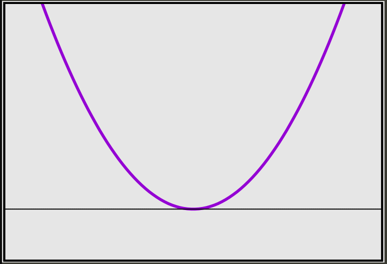
- If $f(a)$ and $f(b)$ have opposite signs, and f is continuous, there must be a root inside the interval: the root is *bracketed*.
- Consecutive refinement of the interval until $a - b < \varepsilon$ guarranteed to find the root.

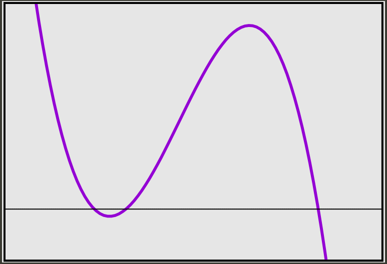
Bracketing

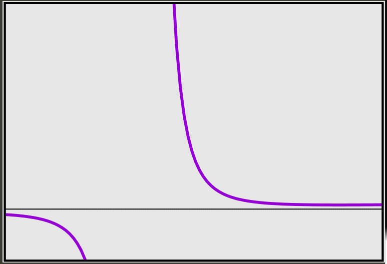
Must find bounding interval first!

- Plot the function.
- Make a conservative guess for $[a, b]$ then *slice up* the interval, checking for sign change of f.
- Make a wild guess and *expand* the interval until f exhibits a sign change.

Trouble makers


No sign change


Easily missed


Singularity

Suppose we've bracketed a root, what's next?

Classic root finding algorithms

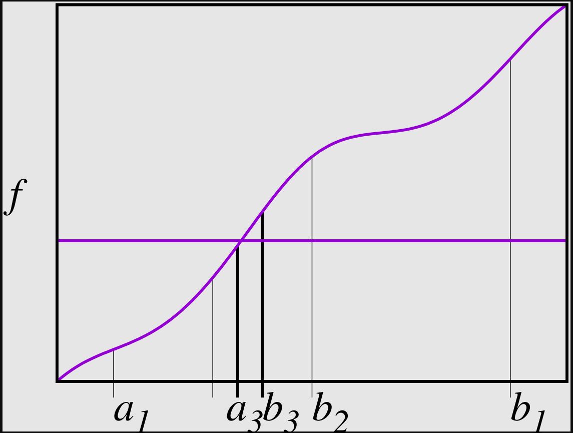
- Bisection
- Secant/False Position
- Ridders'/Brent
- Newton-Raphson (requires derivatives)

All of these focus in on the root, but have different convergence and stability characteristics.

Note:

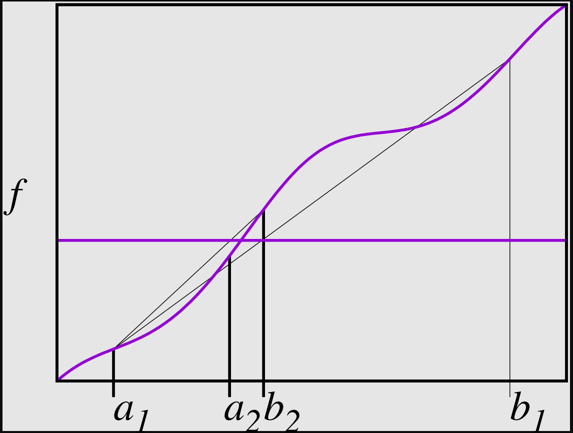
- For polynomial f: specialized routines (Muller, Laguerre)
- For eigenvalue problems: specialized routines (linear algebra)

Bisection method



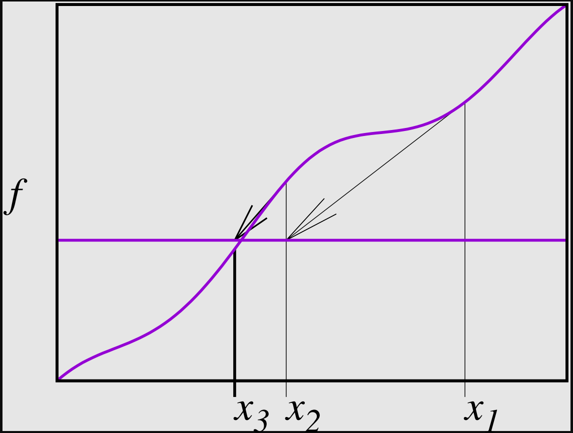
- Sign change: Root must lie in $[a_1, b_1]$
- Find midpoint of interval.
- Compute $f(midpoint)$, and its sign.
- Sign same as upper bound: midpoint becomes new upper bound b_2.
- Find midpoint again.
- Sign same as upper bound: becomes new lower bound a_2.
- Next midpoint becomes new upper bound b_3.
- Next midpoint becomes new lower bound a_3.

False Position Method



- Same start: root within $[a_1, b_1]$
- Linearly approximate/interpolate f.
- Solve for next x.
- Compute $f(x)$, and inspect its sign.
- Get new root bracket.
- Interpolate again.
- Solve for next x.
- Compute $f(x)$, inspect sign: new bracket.
- etc.

Newton-Raphson method



- Guess x.
- From function and derivative, get linear approx.
- Compute approximate root.
- Repeat.
- Fast convergence.

Convergence and Stability

method	convergence	stability
Bisection	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n$	Stable
Secant	$\epsilon_{n+1} = c\epsilon_n^{1.6}$	No bracket guarrantee
False position	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^{1.6}$	Stable
Ridders'	$\epsilon_{n+2} = c\epsilon_n^{2}$	Stable
Brent	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^{2}$	Stable
Newton-Raphson	$\epsilon_{n+1} = c\epsilon_n^{2}$	Can be unstable

Now consider the GSL root finding example again

Suppose we want to find where $f(x) = a\cos(\sin(v + wx)) + bx - cx^2$ is zero (a *"root"*).

```cpp
// gslrx.cpp
#include <iostream>
#include <gsl/gsl_roots.h>

struct Params {
 double v, w, a, b, c;
};
double examplefunction(double x, void* param){
 Params* p = reinterpret_cast<Params*>param;
 return p->a*cos(sin(p->v+p->w*x))+p->b*x-p->c*x*x;
}

int main() {
 double x_lo = -4.0;
 double x_hi = 5.0;
 Params args = {0.3, 2/3.0, 2.0, 1/1.3, 1/30.0};
 gsl_root_fsolver* solver;
 gsl_function      fwrapper;
 solver = gsl_root_fsolver_alloc(
              gsl_root_fsolver_brent);
 fwrapper.function = examplefunction;
 fwrapper.params = &args;
 gsl_root_fsolver_set(solver,&fwrapper,x_lo,x_hi);

 std::cout << "iter lower upper root err\n";

 int status = 1;
 for (int iter=0; status and iter < 100; ++iter) {
   gsl_root_fsolver_iterate(solver);
   double x_rt = gsl_root_fsolver_root(solver);
   double x_lo = gsl_root_fsolver_x_lower(solver);
   double x_hi = gsl_root_fsolver_x_upper(solver);
   std::cout << iter <<" "<< x_lo <<" "<< x_hi
             <<" "<< x_rt <<" "<<x_hi-x_lo<<"\n";
   status=gsl_root_test_interval(x_lo,x_hi,0,1e-3);
 }

 gsl_root_fsolver_free(solver);
 return status;
}
```

Multidimensional Root Finding

$$\vec{f}(\vec{x}) = \vec{0}$$

or

$$\begin{aligned}
f_1(x_1, x_2, x_3, \ldots, x_D) &= 0 \\
f_2(x_1, x_2, x_3, \ldots, x_D) &= 0 \\
&\vdots \\
f_D(x_1, x_2, x_3, \ldots, x_D) &= 0
\end{aligned}$$

- Cannot bracket a root with a finite number of points.
- Roots of each equation define a $D - 1$ hypersurface.
- Looking for possible intersections of hypersurfaces.

Newton-Raphson for Multidimensional Root Finding

Given a good initial guess, Newton-Raphson can work in arbitrary dimensions:

$$\begin{aligned}
\vec{f}(\vec{x}_0 &+ \delta\vec{x}) = \vec{0} \\
\vec{f}(\vec{x}_0) &+ \frac{\partial\vec{f}}{\partial\vec{x}}\cdot\delta\vec{x} = \vec{0} \\
\vec{f}(\vec{x}_0) &= -\frac{\partial\vec{f}}{\partial\vec{x}}\cdot\delta\vec{x} \\
\vec{f}(\vec{x}_0) &= -J\cdot\delta\vec{x} \\
\delta\vec{x} &= -J^{-1}\cdot\vec{f}(\vec{x}_0)
\end{aligned}$$

Requires inverting a $D\times D$ matrix, or at least, solving a linear set of equations: see lecture on linear algebra.

Convergence and Stability

- As in 1D, Newton-Raphson can be unstable.
- Need some safe guard that our iteration steps do not spin out of control.
- Several ways, e.g. make sure that $\|\vec{f}(\vec{x})\|^2$ gets smaller in each time step.
- This can potentially still fail, but usually does the trick.
- Also in the GSL.

Conclusion

- GSL has a lot of other functionality
- Point of this lecture was to use a library
- C libraries may need some ‘boilerplate’, but at least you’re not