Numerical Libraries

James Willis, Yohai Meiron

2022 IHPCSS - Athens, Greece

Overview

- Modular programming
- Libraries
- Software modules
- Scientific libraries

1

Don’t reinvent the wheel!

Modular programming

Motivation

- Scientific software can be large, complex and subtle.
- Interactions grow as (number of lines of code)$^{\mathbf{2}}$.
- You're either recoding the same thing, or are copy-pasting with large risk of mistakes.

Modular programming

Motivation

- Scientific software can be large, complex and subtle.
- Interactions grow as (number of lines of code)^2.
- You're either recoding the same thing, or are copy-pasting with large risk of mistakes.

How to

- Design the interface (header files in C++, interface in Fortran).
- Implementation is separate (ideally separate file).
- Enforce boundaries. Avoid globals! Use namespaces where needed (C++).

Modular programming

Motivation

- Scientific software can be large, complex and subtle.
- Interactions grow as (number of lines of code)^2.
- You're either recoding the same thing, or are copy-pasting with large risk of mistakes.

How to

- Design the interface (header files in C++, interface in Fortran).
- Implementation is separate (ideally separate file).
- Enforce boundaries. Avoid globals! Use namespaces where needed (C++).

Advantages

- It allows each module to be tested individually.
- It makes rebuilding software more efficient.
- It makes changing the code easier, and version control more powerful.

More comments

- "Module" can mean a couple of things: *translation unit*, or *software module*.
- The C++20 standard introduced a new (and very different) to work with modules.

Modular programming, compiling and linking

- In modular programming, several object files for different modules need to be linked together.
- In the example below, thiscppapp.cpp/thisfapp.f90 contains the main program and alibrary.cpp/alibrary.h/alibraryf.f90 are a C++ and f90 module.

Modular programming, compiling and linking

- In modular programming, several object files for different modules need to be linked together.

- In the example below, thiscppapp.cpp/thisfapp.f90 contains the main program and alibrary.cpp/alibrary.h/alibraryf.f90 are a C++ and f90 module.

```
g++ -g -O2 -c -o thiscppapp.o thiscppapp.cpp
g++ -g -O2 -c -o alibrary.o alibrary.cpp
g++ -g thiscppapp.o alibrary.o -o thiscppapp
gfortran -g -O2 -c -o thisfapp.o thisapp.f90
gfortran -g -O2 -c -o alibraryf.o alibraryf.f90
gfortran -g thisfapp.o alibraryf.o -o thisfapp
```

(by the way, please use `make` or `cmake` in real life).

- What if we could use our alibrary in another project called newapp, without recompiling alibrary.cpp or alibraryf.f90?

From modular programming to libraries

From modular programming to libraries

- Copy .o and .h to separate directories:

```
alibrary.h     -> /base/include/alibrary.h
alibraryf.mod  -> /base/include/alibraryf.mod
alibrary.o     -> /base/lib/alibrary.o
alibraryf.o    -> /base/lib/alibraryf.o
```

From modular programming to libraries

- Copy .o and .h to separate directories:

```
alibrary.h     -> /base/include/alibrary.h
alibraryf.mod  -> /base/include/alibraryf.mod
alibrary.o     -> /base/lib/alibrary.o
alibraryf.o    -> /base/lib/alibraryf.o
```

- Must let compiler know where they are:
  Add `-I` flag for include directories.
  Absolute path for object file *(only for now!)*.

```
g++ -g -O2 -I/base/include -c -o newapp.o newapp.cpp
g++ -g -o newapp newapp.o /base/lib/alibrary.o
gfortran -g -O2 -I/base/include -c -o newfapp.o newfapp.f90
gfortran -g -o newfapp newfapp.o /base/lib/alibraryf.o
```

Building with Libraries

Real libraries are similar; they have

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

Linux: library filenames start with lib and end in .a or .so.

```
g++ -g -o thiscppapp thiscppapp.o /base/lib/libalibrary.a
gfortran -g -o thisfapp thisfapp.o /base/lib/alibraryf.a
```

Instead of giving the explict path in linker command, we should specify:

- the path to the library's object using the -L option
- the object code using -lNAME (with a lower case letter -l)
- libraries should come after the object files that use them.

```
g++ -g -L/base/lib -o thiscppapp thiscppapp.o -lalibrary
gfortran -g -L/base/lib -o thisfapp thisfapp.o -lalibraryf
```

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

More Notes on Libraries

- 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.
- You usually also do not need `-I` or `-L` for libraries accessed using the 'module load' command on the supercomputers.
- If you compile your own libraries in non-standard locations, you do need `-I` and `-L` options (as well as the `-lNAME` clause).

Example: MPI

```
$ mpicc -show
gcc -I/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/include
 -L/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib/release
 -L/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib
 -Xlinker --enable-new-dtags -Xlinker -rpath -Xlinker
 /opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib/release
 -Xlinker -rpath -Xlinker /opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib
 -lmpifort -lmpi -lrt -lpthread -Xlinker --enable-new-dtags -ldl
```

```
$ mpifc -show
gfortran -I/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/include/gfortran/4.8.0
 -I/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/include
 -L/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib/release
 -L/opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib
 -Xlinker --enable-new-dtags -Xlinker -rpath -Xlinker
 /opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib/release
 -Xlinker -rpath -Xlinker /opt/intel/compilers_and_libraries_2019.3.199/linux/mpi/intel64/lib
 -lmpifort -lmpi -lrt -lpthread -Xlinker --enable-new-dtags -ldl
```

Software modules

Software modules are a very common way to make software available on shared supercomputers.

```
$ module avail
------------------------------ /opt/modulefiles ------------------------------
abaqus/2016         abyss/2.0.2                      anaconda/4.2.0-3.5.2
abaqus/2017         AI/anaconda2-5.1.0_gpu           anaconda2/5.1.0
Abinit/7.10.5       AI/anaconda3-5.1.0_gpu           anaconda2/5.2.0
Abinit/8.0.8b       AI/anaconda3-5.1.0_gpu.2018-08   anaconda3/2019.03
Abinit/8.4.3        AIPS/31DEC16                     anaconda3/5.1.0
abyss/1.5.2         allpaths-lg/52488                anaconda3/5.2.0(default)
...
```

E.g. try `module avail openblas` and `module help openblas` (and `module spider openblas` for systems using lmod).

Must first do a `module load MODULE` before compiling and before running.

Installing libraries from source

What to do when your package manager does not have the library, or it's not in the software module stack, and you do not have permission to install packages in the system paths?

Installing libraries from source

What to do when your package manager does not have the library, or it's not in the software module stack, and you do not have permission to install packages in the system paths?

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

Common installation procedure (but read documentation!):

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

```
$ cmake -DCMAKE_INSTALL_PREFIX=<BASE> ....
$ make
$ 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**.

Using libraries that are not in standard directories

- For libraries that are not in standard directories, you need `-I<BASE>/include` and `-L<BASE>/lib` options in your compilation/link commands.
- Alternatively, you can omit these by setting some linux 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
```

  You either enter these commands on the linux prompt before compiling, or, to set these automatically when you log in, add these lines to the `.bashrc` file in your home folder.

- The last one (`LD_LIBRARY_PATH`) may be necessary to run the application, even when it was successfully built and linked already.
- 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
```

2

So many libraries, so little time. . .

Some libraries for scientific computing (CPU)

BLAS interface for libraries for basic linear algebra operations. C C++ Fortran

LAPACK Linear solvers, eigenvalue problems, SVD, factorization. C C++ Fortran Python

ScaLAPACK Distributed version of LAPACK. C C++ Fortran

ARPACK-NG and SLEPc Eigenvalue problems. C C++ Fortran

Boost Peer-reviewed portable C++ source libraries C++

Eigen Matrices, vectors, numerical solvers, ... C++

FFTW Fourier and related transforms. C C++ Fortran

HDF5 and NetCDF Portable data model, library, and file formats. C C++ Fortran

GSL Numerical analysis library. C C++

PETSc Scalable (parallel) solution of partial differential equations. C C++ Fortran

Armadillo Matrix and vector maths similar to MATLAB. C++

Blaze Dense and sparse arithmetic. C++

Dlib Machine learning algorithms and tools. C++

Mlpack Machine learning algorithms. C++ Python

Trilinos algorithms etc. for the solution of large-scale, complex multi-physics engineering and scientific problems. C C++

Eigenvalue solvers targeting exascale computers

ELPA

EigenExa

Some libraries for scientific computing (GPU)

cuBLAS / rocBLAS GPU-accelerated BLAS implementations

cuSOLVER / rocSOLVER$^{(\beta)}$ / MAGMA GPU-accelerated LAPACK-like libraries

cuSPARSE / rocSPARSE GPU-accelerated sparse matrix libraries

cuFFT / rocFFT GPU-accelerated Fast Fourier Transforms libraries

cuRAND / rocRAND Random number generation on the GPU

NPP (CUDA) Performance primitives for image & video processing

Thrust GPU-accelerated STL-like library (parallel algorithms)

Some libraries for scientific computing (Python)

NumPy Faster arrays for Python. Mind all the lessons from the HPC Python programming session!

SciPy Provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization.

Numba JIT compilation of a subset of Python and NumPy code into fast machine code.

Pandas Tabular data manipulation and analysis.

scikit-learn (sklearn) Machine Learning in Python. Simple and efficient tools for data mining and data analysis.

TensorFlow & PyTorch Neural nets/deep learning libraries.

Stan, PyMC Statistical modelling, data analysis, and Bayesian predictions.

3

Example: GNU Scientific Library (GSL)

Example: 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 (gslrx.cpp)

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

```
#include <iostream>
#include <gsl/gsl_roots.h>
#include <gsl/gsl_errno.h>

struct Params {double v, w, a, b, c;};

double examplefunction(double x, void* param){
 auto [v, w, a, b, c] = *(Params*)param;
 return a*cos(sin(v+w*x))+b*x-c*x*x;
}

int main() {
 double x_lo = -4.0;
 double x_hi =  5.0;
 Params args = {0.3, 2./3., 2., 1./1.3, 1/30.};
 gsl_root_fsolver* solver;
 solver = gsl_root_fsolver_alloc(
             gsl_root_fsolver_brent);
```

```
 gsl_function fwrapper;
 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, iter = 0;
 do {
  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);
 } while (status==GSL_CONTINUE and iter < 100);
 gsl_root_fsolver_free(solver);
}
```

Compilation and linkage

- Lots of `gsl...` stuff.
- 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.

Compilation and linkage

- Lots of `gsl...` stuff.
- 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?

Compilation and linkage

- Lots of `gsl...` stuff.
- 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
g++ gslrx.cpp -o gslrx -lgsl -lgslcblas
```

Compilation and linkage

- Lots of `gsl...` stuff.
- 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
g++ gslrx.cpp -o gslrx -lgsl -lgslcblas
```

1. When using software modules, you don't need the `-I` and `-L` options.
2. GSL is usually a module (not on Bridges2).

Compilation and linkage

- Lots of `gsl...` stuff.
- 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
g++ gslrx.cpp -o gslrx -lgsl -lgslcblas
```

1. When using software modules, you don't need the `-I` and `-L` options.
2. GSL is usually a module (not on Bridges2).

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 0.001179
```

Compilation and linkage

- Lots of `gsl...` stuff.
- 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?
- Even existing solutions like the once in the GSL, can't really be used until you understand the algorithm on a high level.

```
module load gcc
g++ gslrx.cpp -o gslrx -lgsl -lgslcblas
```

1. When using software modules, you don't need the `-I` and `-L` options.
2. GSL is usually a module (not on Bridges2).

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 0.001179
```

4

Using a C or C++ library in Python

But I only like Python!

No problem! Most numerical libraries have a Python ***API***.

But I only like Python!

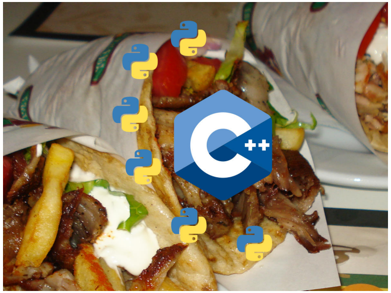
No problem! Most numerical libraries have a Python ***API***.

And you can create your own wrapper using PyBind11, ctypes, or SWIG.

But I only like Python!

No problem! Most numerical libraries have a Python ***API***.

And you can create your own wrapper using PyBind11, ctypes, or SWIG.



(PyBind11 is the tzatziki)

PyBind11 example

Suppose we want to use cuFFT to calculate a Fourier transform, but in Python.

PyBind11 example

Suppose we want to use cuFFT to calculate a Fourier transform, but in Python.

We write a function in C++ that wraps around the library call, and use *PyBind11* to create a Python "package", which we can import and use like any other.

PyBind11 example

Suppose we want to use cuFFT to calculate a Fourier transform, but in Python.

We write a function in C++ that wraps around the library call, and use *PyBind11* to create a Python "package", which we can import and use like any other.

GPU-based FFT functionality (via cuFFT or otherwise) is provided by some packages on PyPI.

PyBind11 example (`pycufft.cu`)

This is the wrapper, defining a C++ function, that calls the cuFFT C function.

```cpp
#include <complex>
#include <cufft.h>
#include <vector>

std::vector<std::complex<float>>
 fft(const std::vector<float>& signal)
{
 const size_t size = signal.size();
 float *signal_d;
 cudaMalloc((void**)&signal_d,
  size*sizeof(float));
 cudaMemcpy(signal_d, signal.data(),
  size*sizeof(float), cudaMemcpyHostToDevice);
 float2 *result_d;
 cudaMalloc((void**)&result_d,
  (size/2+1)*sizeof(float2));
 cufftHandle plan;
 cufftPlan1d(&plan, size, CUFFT_R2C, 1);
 cufftExecR2C(plan, (cufftReal*)signal_d,
  (cufftComplex*)result_d);
```

```cpp
 std::vector<std::complex<float>>
  result(size/2+1);
 cudaMemcpy(result.data(), result_d,
  (size/2+1)*sizeof(float2),
  cudaMemcpyDeviceToHost);
 cudaFree(signal_d);
 cudaFree(result_d);
 cufftDestroy(plan);
 return result;
}

#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <pybind11/complex.h>
PYBIND11_MODULE(pycufft, m) {
    m.doc() = "cuFFT wrapper";
    m.def("fft", &fft, "docstring");
}
```

PyBind11 example (pycufft_example.py)

This is an example of a Python script using our wrapped library.

```python
import pycufft, numpy as np
w0    = 5   # rad/s
t_max = 20  # s
n     = 2048
t = np.linspace(0, t_max, n)
x = np.sin(w0*t)
result = pycufft.fft(x)
mag = np.abs(result)
w_recovered = mag.argmax() * 2*np.pi/t_max
print(f'Recovered angular frequency: {w_recovered} rad/s')
```

Note that the return type of our wrapped function is a list, we can ask PyBind11 to return a NumPy array, but that doesn't fit on one slide

PyBind11 example (how to use on Bridges2)

(1) Copy code for Python session if you haven't done so already:

```
cp -r /jet/home/rzon/hpcpycode ~
```

PyBind11 example (how to use on Bridges2)

(1) Copy code for Python session if you haven't done so already:

```
cp -r /jet/home/rzon/hpcpycode ~
```

(2) Activate environment:

```
source ~/hpcpycode/activate
```

PyBind11 example (how to use on Bridges2)

(1) Copy code for Python session if you haven't done so already:

```
cp -r /jet/home/rzon/hpcpycode ~
```

(2) Activate environment:

```
source ~/hpcpycode/activate
```

(3) Load the CUDA module and compile the shared object file:

```
module load cuda
nvcc -O3 -shared -std=c++17 -Xcompiler="-Wall -fPIC" \
 $(python3 -m pybind11 --includes) pycufft.cu \
 -o pycufft.cpython-38-x86_64-linux-gnu.so -lcufft
```

PyBind11 example (how to use on Bridges2)

(1) Copy code for Python session if you haven't done so already:

```
cp -r /jet/home/rzon/hpcpycode ~
```

(2) Activate environment:

```
source ~/hpcpycode/activate
```

(3) Load the CUDA module and compile the shared object file:

```
module load cuda
nvcc -O3 -shared -std=c++17 -Xcompiler="-Wall -fPIC" \
 $(python3 -m pybind11 --includes) pycufft.cu \
 -o pycufft.cpython-38-x86_64-linux-gnu.so -lcufft
```

(4) Try running the script:

```
srun --partition=GPU-shared --gres=gpu:v100-32:1 \
 --time=00:01:00 python pycufft_example.py
```

Final remarks

- “Do not reinvent the wheel”, i.e. reuse libraries already developed.
- Use **mature** libraries, well-known in the corresponding field/community. They have been developed, maintained, debugged, tested, . . .
- Optimized and really good at the task(s) in question.
- Libraries are a cornerstone element in modularity and professional sofware development.
- Consider including further elements of software enigneering, such as, automation (via `make` or `cmake`) of compilation and linking – helps with compilation flags and cross-platform developments.