Facilitating code distribution: Docker and Generic IO

Robert G. Clapp

ABSTRACT

Disseminating research through computational software is of growing importance in academia. The growth of cloud computing and object-oriented programming has led to the development of new paradigms. We take advantage of two of these paradigms, containers and generalized IO, as a new way to distribute. software to SEP sponsors.

INTRODUCTION

For more than 25 years SEP has included the concept of reproducible research in its mission. SEP's initial approach was to use `Cake` (Claerbout, 1990; Claerbout and Nichols, 1990). Later Schwab and Schroeder (1995) outlined our replacement of `Cake` with the standard Unix tool `make`. Fomel and Hennenfent (2007) translated and extended these reproducible concepts using `Scons`.

Reproducible research serves multiple purposes such as integrity of research results and a way to distribute research ideas. All of the above approaches have significant challenges. First, they require the installation of a significant software stack. Clapp (2016) described initial efforts of overcoming this obstacle using containers. Containers can capture the entire software stack and guarantee reproducibility. The second mission, making code simpler to use by others, is complicated by the fact that the seismic community does not agree on a single software solution. Instead companies and universities use one of a multitude of software packages.

In this paper we attempt to deal with the issue of producing reproducible and interruptible software. To solve the first problem, we build on Clapp (2016), describing how dockers can be used to encapsulate a research paper. We also introduce a library, `genericIO`, a C++ and Fortran 2003 library which abstracts parameter handling and IO. We include three concrete versions of the library using SEPlib, Madgascar, and Java Script Object Notation (JSON). We also show several examples of applications using the library.

GENERIC-IO

Abstraction is a powerful concept in object-oriented programming. Software written to function with a generic abstract object will work with all objects that inherit from the abstract base class. An example of using abstraction is writing solvers. which separate the mathematics of inversion from the physics of operators. Parameter handling and input/output (IO) can also be thought of as being separate from the mathematics/science that application is performing. If all of the IO/parameter handling in a program is written in terms of an abstract base class, all that is required to change to a new processing system is to write a concrete child class that inherits from the abstract base class. The library `genericIO` attempts to create an abstract interface for IO/parameter handling. The entire library is documented using `Doxygen`. In the next portion we describe the basic building blocks of the library in C++ then Fortran2008.

The `paramObj` abstract class reads parameters for the program. It has functions to return `bool, float, int, string`, and vectors of each given the parameter name. It allows the user specify a default if the parameter isn't found.

The `genericRegFile` base class can read the same type of parameters from a file. In addition it has the ability to write parameters to a file, and to read/write float, complex, and byte data to/from a file. The reading and writing can be in terms of stream (read the next X bytes), or a subcube of the data object.

The `genericIO` base class is able to return a `genericRegFile` object given a filename or tag, and how the file is going to be used (in,out, in/out). It also has the ability to return a `genericIrregFile` object which still in the design stage. In addition, it keeps a list of all the files it has opened and when the object goes out of scope closes all of its files.

The `IOModes` class knows about all of the base classes of `genericIO` along which the default IO mode (can be selected at compile time). A programmer asks for either the default or a specific IO mode.

Currently three concrete classes inherit from abstract base classes: `rsfIO, sepIO,` and `JSONIO`. YOU ONLY LIST 3 CLASSES. The first two are simply wrap the corresponding RSF and SEPlib function. The `JSONIO` class uses JSON file that describe command line and file parameters.

Rather than maintaining a parallel FOrtran2003 library for all potential IO types, we used a singleton design pattern to allow us to create a single instance of the `ioModes` object to be created. By writing C functions that all use the singleton ioModes object we were able to support the same IO mechanisms in Fortran while creating a single set of Fortran2003 classes.

An example of using the `genericIO` library to window a dataset can be found in the first appendix.

DOCKERS

Clapp (2016) discussed how containers, specifically dockers, were designed to streamline the development to application process. A docker can be thought of as a virtual machine with minimal overhead. Once a developer has built a docker that runs on their local machine, that same docker will run in a production environment. The latest versions of the `gcc` compilers docker is conventionally built through a `Dockerfile`. A `Dockerfile` starts with a `From` statement which is another docker to build from. For example, we might start from a compact version of an operating system such as a specific version of CentOS.

```
From centos:7
MAINTAINER Bob Clapp <bob@sep.stanford.edu>
```

We then line by line describe how to build our environmnet. Each line represents a different layer in the docker filesytstem A delta of the filesytesm is calculated and stored. The final docker is the summation of all of these deltas. The storing of deltas means that we don't want to create files in one layer that we are going to erase in another layer. As a result we are careful to remove all temporary and unneeded files. For example, every time we run `yum` it creates a temporary cache. So after we install a list of packages with `yum` we clean up these cache files. As an example, imagine we want to build the latest version of the `gcc` compilers. There are several prequisites that we need to install.

```
RUN yum -y install gmp-devel mpfr-devel libmpc-devel;\\
   yum -y clean  all
RUN yum -y install make cpp wget gcc-c++ zlib-devel \\
curl wget; yum -y clean all
```

Once we've installed these packages we can download the latest versions of the compilers. Then configure, compile, install, and finally clean up. We need to do all of this on one line to minimize the delta size.

```
RUN mkdir   /tmp/gcc-build
RUN cd /tmp; \\
wget ftp://208.118.235.20/gnu/gcc/gcc-6.2.0/gcc-6.2.0.tar.gz;  \\
tar xf gcc-6.2.0.tar.gz; cd /tmp/gcc-build;  cd /tmp/gcc-build;\\
 ../gcc-6.2.0/configure --prefix=/usr/local --disable-multilib \\
  -with-system-zlib --enable-languages=c,c++,fortran;  \\
  cd /tmp/gcc-build; \\
  make  -j 12; cd /tmp/gcc-build make install;  rm -rf /tmp/gcc*
```

Dockers for reproducibility

The docker concept of layers is analogous to the way we currently do reproducible research. We start with our current environment, the base image in dockerspeak. We then, through make dependencies, build the software we need, and run a series of commands to build a result. There is a significant level of redundancy in a paper, let alone a report. On the report level, each paper depends on the same basic environment. At the paper level, results often depend on the same executables so those layers will get reused. To convert to using dockers for reproducibility, and to enable others to use our code more easily, we need to significantly improve the way we build our environment.

For years SEP has used autoconf/automake for building SEPlib. The autoconf/automake combination also attempted to configure makefiles to ease the building process for SEP reports. This process worked somewhat successfully inside SEP but proved challenging for outside users due to the incompleteness of the makefile configuration process, the use of additional libraries (such as FFTW), and the vagarities of autoconf. Levin and Clapp (2017) describes our reasoning for converting/methodology of converting from autoconf to `cmake` for building SEPlib. Simplifying the SEPlib building process was the necessary first step in our complete overhaul of how we define reproducible software. The same argument that supports the shift to `cmake` for building SEPlib is true for the software in individual report articles. By switching to `cmake` we make installation at outside locations much simpler.

REPRODUCIBILITY TEMPLATE

The key for internal and external reproducibility will be the combination of a gitlab server and a docker repository. Each reproducible figure will be its own docker. All report articles, stored as git repositories, will contain the Dockerfiles to build each reproducible figure and make commands to build the dockers from these images.

While the bar described above seems high, it should, once internalized, make doing reproducible research easier. We will begin by creating a docker containing the entire SEP environment at report time. The student will then start from this base image, clone from their git repository the source code for the paper and build it using cmake, creating a report article image. The Dockerfile's for each figure starts from this report article image and runs the commands to the create an individual figure.

Their are several advantages of this approach. First, the student can switch between systems with minimal effort[1] as long as docker is installed. Second, guaranteed reproducibility over time. As long as docker exists the results are completely reproducible. If docker does not exist the Dockerfile contains a complete description of how it was created. Third, reprodcubility testing is simplified for both the author and the

[1]Different locations for data/scratch space, etc. will change the docker invocation.

checker. Building a docker is not environment dependent. As long as it builds for the author, it will build for the checker.

FUTURE WORK

One of the longer term goals of this project is for all SEP research papers to be written in this form. To accomplish this goal the generic IO library will be extended to handle irregular dataasts beyond its current capabilities. In addition, we would like to have all of the labs for SEP classes to be done in this genericIO/docker format. An even longer term goal is to replace all of the IO in SEPlib using this genericIO framework. This would allow us to update the core of SEPLib to a more more modern design without affecting any applications.

CONCLUSION

In this paper we described a new way to perform research that is guaranteed to be easily reproducible while it the same time allowing someone to quickly incorporate research ideas into their software environment. We accomplish this goal by writing programs in terms of a generalized IO library, using git and cmake for code development and installation, and dockers to guarantee reproduciblity.

REFERENCES

Claerbout, J. F., 1990, Active documents and reproducible results: SEP-Report, **67**, 139–144.

Claerbout, J. F. and D. Nichols, 1990, Why active documents need cake: SEP-Report, **67**, 145–148.

Clapp, R. G., 2016, Reproducibility through containers : SEP-Report, **165**, 193–198.

Fomel, S. and G. Hennenfent, 2007, Reproducible computational experiments using scons: Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on, IV–1257.

Levin, S. A. and R. G. Clapp, 2017, make, schmake: CMake : SEP-Report, **168**, 309–312.

Schwab, M. and J. Schroeder, 1995, Reproducible research documents using GNUmake: SEP-Report, **89**, 217–226.

A

```
#include<ioModes.h>
using namespace SEP;
std::shared_ptr<hypercube> calc_params(std::shared_ptr<paramObj> par,
  std::shared_ptr<hypercube> hyperIn, const std::vector<int> ng,
```

```
  std::vector<int>&nw,std::vector<int>&fw,std::vector<int>&jw){

  int ndim=(int)ng.size();
  std::vector<axis> axes;
  for(int idim=0,i=1; idim<ndim; idim++,i++){
    axis ain=hyperIn->getAxis(i);
    nw[idim]=par->getInt(std::string("n")+std::to_string(i),nw[idim]);
    fw[idim]=par->getInt(std::string("f")+std::to_string(i),0);
    jw[idim]=par->getInt(std::string("j")+std::to_string(i),1);
    fprintf(stderr,"i=%d n=%d f=%d j=%d \n",idim,nw[idim],fw[idim],jw[idim]);
    if(fw[idim] >= ng[idim]) par->error(std::string("Invalid f")+
     std::to_string(i));
    if(nw[idim] > ng[idim]) par->error(std::string("Invalid n")+
      std::to_string(i));
    if(jw[idim] ==-1) jw[idim]=1;
    if(nw[idim] ==-1) {
      if(fw[idim]==-1){
        fw[idim]==0;
      }
      nw[idim]=(ng[idim]-1-fw[idim])/jw[idim]+1;
    }
    else if(fw[idim]==-1) fw[idim]=0;
    if(ng[idim] < 1+ fw[idim] +jw[idim]*(nw[idim]-1)){
       fprintf(stderr,"f=%d j=%d n=%d ng=%d \n",fw[idim],jw[idim],nw[idim],ng[idim]);
       par->error(std::string("Invalid window parameters axis ")+std::to_string(i));
     }
    float o=fw[idim]*ain.d+ain.o;
    float d=jw[idim]*ain.d;
    axes.push_back(axis(nw[idim],o,d,ain.label));
  }
  for(int idim=ndim; idim<8; idim++){
    nw[idim]=1; fw[idim]=0; jw[idim]=1;
  }
  std::shared_ptr<hypercube> h(new hypercube(axes));
  return h;
}
int main(int argc, char **argv){

  ioModes modes(argc,argv);



  std::shared_ptr<genericIO>  io=modes.getDefaultIO();
  std::shared_ptr<paramObj> par=io->getParamObj();
  std::string in=par->getString(std::string("in"));
```

```
std::string out=par->getString(std::string("out"));

std::shared_ptr<genericRegFile> inp=io->getRegFile(in,usageIn);


std::shared_ptr<hypercube> hyperIn=inp->getHyper();
std::vector<int> ng=hyperIn->getNs();

int ndim=(int)ng.size();

if(ndim>8) par->error("Currently handle maximum of 8 dimensons");
std::vector<int> nw(8,-1),fw(8,-1),jw(8,-1);

std::shared_ptr<hypercube> hyperOut=calc_params(par,hyperIn,ng,nw,fw,jw);

std::shared_ptr<genericRegFile> outp=io->getRegFile(out,usageOut);
outp->setHyper(hyperOut);
outp->writeDescription();

int maxSize=par->getInt(std::string("maxsize"),8);
long long maxS=(long long) maxSize*1024*1024*1024;

std::vector<int> nloop(8,1);
std::vector<int> fsend=fw,jsend=jw,nsend=nw;


long long n123= hyperOut->getN123();
 int iaxis=ndim-1;
 while(n123 > maxS && iaxis>0){
   n123=n123/(long long) ng[iaxis];
   nloop[iaxis]=nw[iaxis];
   iaxis--;
 }
 for(int i=iaxis+1; i<8; i++){
    fsend[i]=0; jsend[i]=1; nsend[i]=1;
  }

 float *buf=new float[n123];
 for(int i8=0; i8 < nloop[7]; i8++){
   if(iaxis<7){fsend[7]=i8*jw[7]+fw[7]; nsend[7]=1;}
 for(int i7=0; i7 < nloop[6]; i7++){
   if(iaxis<6){fsend[6]=i7*jw[6]+fw[6]; nsend[6]=1;}
 for(int i6=0; i6 < nloop[5]; i6++){
   if(iaxis<5){fsend[5]=i6*jw[5]+fw[5]; nsend[5]=1;}
```

```
  for(int i5=0; i5 < nloop[4]; i5++){
    if(iaxis<4){fsend[4]=i5*jw[4]+fw[4]; nsend[4]=1;}
  for(int i4=0; i4 < nloop[3]; i4++){
    if(iaxis<3){fsend[3]=i4*jw[3]+fw[3];  nsend[3]=1;}
  for(int i3=0; i3 < nloop[2]; i3++){
  if(iaxis<2){fsend[2]=i3*jw[2]+fw[2];  nsend[2]=1;}
for(int i2=0; i2 < nloop[1]; i2++){
if(iaxis<1){ fsend[1]=i2*jw[1]+fw[1]; nsend[1]=1;}

inp->readFloatWindow(nsend,fsend,jsend,buf);
outp->writeFloatStream(buf,n123);
}}}}}}
delete [] buf;

}
```