Computational Science I

Lecture Notes for CAAM 519

Matthew G. Knepley

Department of Computational and Applied Mathematics
William Marsh Rice University

December 9, 2022

I dedicate this book to my wonderful wife Margarete, without whose patience and help it could never have been written.

Acknowledgements

There would be no book, and indeed no PETSc, without the pioneering work of Barry Smith, refashioning scientific software into a recognizable discipline. Jed Brown has provided all the version control wisdom in these pages. Mark Adams taught me about multigrid, and in particular the underappreciated utility of FMG. I have had countless productive discussions with Gerard Gorman, David Ham, Patrick Farrell, Michael Lange, and Lawrence Mitchell about Firedrake, solvers, parallel meshing, and software design. Ridgway Scott, Andy Terrel and Rob Kirby have taught me what I know about finite elements and Texas. Hans Petter Langtangen, Anders Logg, Marie Rognes and the entire FEniCS team have been invaluable collaborators and true innovators in the scientific software world. Jay Bardhan has challenged me to apply algorithmics to practical biological problems and been my partner for a decade in becoming mathematical modelers.

Contents

Chapter 1

Programming Basics

1.1 Scientific Computing

I want to begin this course with a brief look back at the history of computational science. I think the physical sciences have always recognized the value of experience better than mathematics. The hallmark of physical science is its interaction with the physical world, and it is long experience with this world that informs and guides theory, and now computation. Computational mathematics, or computational science, stands between the worlds of mathematics and physical science, and thus must deal with these real world constraints. The speed of light, capacitive coupling, leakage current, and the limits of photolithography all constrain the kinds of computations we can do efficiently. And these tradeoffs do change depending on advances in materials science. In the 1960s and 1970s when many foundational numerical algorithms were developed, machines had very little memory compared to computing power, whereas today memory limitations impact very few algorithms. Our assumptions and rules of thumbs must be rethought for each generation of computing hardware.

For pure mathematicians, the *sine non qua* of technical communication is the journal paper, although people like Terence Tao and Timothy Gowers have clearly shown that blogging and the polymath project can play a significant role. However, more then 40 years ago, computational mathematicians created a new way to disseminate their results, namely the high quality numerical library. It is now a commonplace that a great part of your interaction with physical sciences, engineering, and other fields can be mediated by software you produce and maintain. I will argue that the most effective form of software communication is the library.

1.1.1 Libraries

The main impact of computational mathematics is in design and analysis of algorithms for simulation and data analysis. This is where elements of computer science enter in, since software is the transmission mechanism from math-

ematicians and computational scientists to rest of the research community. The description of an algorithm present in a paper is usually schematic so that important pieces of the (local) convergence or complexity proof can be matched to algorithmic operations. Many thing remain unspecified, either because they are incidental to the main point (precise language and data structure choices), cumbersome to describe (exact line search and continuation strategies), difficult to reproduce (optimizations for large, expensive machines), or a host of other reasons. In the end, it is nearly impossible to reproduce results from a computational publication without the original source, test inputs, and outputs to serve as a reference. The best way to assure that others have access to this material and that it does not place an undue burden on the author, is to move most operations into supported community code, reducing the author contribution to an easily manageable set.

The best way to create robust, efficient and scalable, maintainable scientific codes, is to use *libraries*. The primary job of software design is to contain implementation complexity, and libraries provide a systematic, hierarchical strategy for this. They can hide the details associated with different hardware architectures. For example, the MPI libraries (Forum 2012; W. Gropp, Lusk, and Skjellum 1994; W. Gropp and et. al. n.d.) we will work with later in the course hide the details of network interfaces and machine data representations behind generic interfaces. PETSc (Balay, Abhyankar, Adams, Benson, Brown, Brune, Buschelman, E. Constantinescu, et al. 2022; Balay, Abhyankar, Adams, Benson, Brown, Brune, Buschelman, E. M. Constantinescu, et al. 2022) hides complex data structures used for sparse matrix implementations on parallel architectures.

Libraries not only hide complexity for the individual user, but they serve to accumulate best practices in the field. There are very often complex tradeoffs associated with algorithmic choices, so that no *best* algorithm exists (Nachtigal, Reddy, and Trefethen 1992) and determining the best algorithm for a particular problem is costly or undecideable. Thus determining effective and robust defaults can save large amounts of time and computing. For example, by default PETSc uses classical Gram-Schmidt orthogonalization with selective reorthogonalization (Björck 1994) rather than the more robust but far less scalable modified Gram-Schmidt. Moreover, the capabilities of an application using a generic library interface can increase without any code changes. An application using PETSc would benefit from the addition of a new matrix format for specific hardware, Krylov solver, or Implicit-Explicit (IMEX) time integration without any changes to the application itself.

This last point about improvement is the edge of the elusive concept of *extensiblity*. It is not enough to make a fantastic, working code. Users will need the ability to change your approach to fit their problem, or a new hardware or software environment, or to interoperate with another library. A library must separate its core concepts cleanly so that they can be recombined in novel ways to produce more powerful and complex algorithms. For example, in PETSc both multigrid solvers and block solvers exist separately, but they can be combined, even recursively and hierarchically, to produce optimal solvers for complex, multiphysics problems (Brown et al. 2012). The parallel structured grid abstraction,

DMDA, has been reused as the infrastructure underlying both parallel finite volume methods in PyCLAW (Ketcheson et al. 2012; Mandli, Ketcheson, et al. 2012) and parallel isogeometric analysis in PetIGA (Nathan Collier, Lisandro Dalcin, and V. M. Calo 2013; N. Collier, L. Dalcin, and V. Calo 2014).

A library is a bundle of code presenting a uniform interface to the user. Larger libraries, such as PETSc, present many layers of interfaces, often hierarchically, and use their own interface internally when refering to other parts of the library. In order to make the library consistent across architectures, environments, and builds, it should have a consistent Application Binary Interface (ABI) across different builds, for example debugging and optimized.

Bill Gropp gives a thoughtful list of mistakes to avoid when designing a library (W. D. Gropp 1999):

- Namespace pollution and monolithic library structure
- Printing error messages or exiting
- Requiring interactive input or `main()`
- Requiring running on all processes
- Lack of portability, testing, documentation, examples
- Ignorance of standards

A short timeline of early numerical libraries is given below:

71 Handbook for Automatic Computation: Linear Algebra, J. H. Wilkinson and C. Reinch

73 EISPACK, Brian Smith et.al.

79 BLAS, Lawson, Hanson, Kincaid and Krogh

90 LAPACK, many contributors

91 PETSc, Gropp and Smith

All of these packages had their genesis in the Mathematics and Computer Science Division (MCS) (Yood 2005) of Argonne National Laboratory.

1.1.2 Knowledge and Creativity

The purpose of computing is insight, not numbers.

— Richard Hamming, *Numerical Methods for Scientists and Engineers*

Controlling complexity is the essence of computer programming.

— Brian Kernighan

Factual recall and canned examples are not as impressive or important as they once were. Richard Feynman could get an invitation to the Manhattan

Project largely on the strength of his uncanny ability to calculate mentally, but today Wolfram Alpha can fill that role. The ability to make new connections, to draw together disparate sources of information and integrate them, and to propose a new synthesis are more valuable skills today.

In this course, we will rely on internet resources to provide much of the factual background and tutorial examples. However, we have not obviated the need for memorization. It is still quite important to hold a firm mental grasp on collections of information *in order to make new connections*, but initially locating the information, supplementing it, and refreshing yourself have become much easier with the advent of large, decentralized information storage and efficient search.

We are concerned with Why rather than How, which has been satisfactorily explained in the Numerical Analysis course. In his book *Code*, Lawrence Lessig demonstrates the parallels between programming and constitutional law. Both establish ground rules within which a system works. When a court decides a question of constitutional law, they look for an underlying principle which controls the decision. This is exactly the model which we use to compress our understanding of a multitude of situations with similar characteristics, as well as make predictions about new situations. In this course, I want to give you the tools to answer questions such as:

- Which numerical method should I use, and why?
- Is this method accurate for this problem?
- Can my method be made more efficient on this hardware? More scalable?
- Will my method work on this related problem? Will it perform as well?
- Can I write my code such that I can experiment with a range of methods?
- Can I experiment with a range of models?

1.2 Version Control

A mistake most beginning git users make is thinking git is a complete revision control system; it is not. Git, plus an organized mailing list that people actually respond to action items on, plus a rigid set of stylized bookkeeping done by each developer, plus a tyrannical manager of the entire software development process is a complete revision control system.

— Barry Smith, *PETSc Mailing List*

The essence of this class is not to compute things, but rather to learn how to choose the computational method. This class will teach you how to make informed decisions. This is a skill you will also need when it comes time to set a research agenda, choose productive collaborators, and pick tools which minimize work and maximize scientific output. The aim of this lecture is not to show you

how to use Git. There are many excellent web resources for that purpose. This lecture will convince you that Git is the right tool for computational research, among many other things.

1.2.1 Why use Version Control?

Version control is a representation of a groupoid (Wikipedia 2015). The elements of the groupoid are the things being *versioned*, such as source files, and the group operation (which is a partial function), are the changes or *diffs* that map one element to another, such as a textual change to a source file.

A version control system (VCS) allows the user to recall any specific version, which can be thought of as a vertex in a change graph, whose edges represent diffs. We can restore the previous state of a file or set of files, compare changes over time, see who last modified something that might be causing a problem, or trace a set of changes marked for a specific purpose (a branch). It greatly improves the robustness of our storage, since we can recover from accidental changes or deletions even long after the fact, but it can also improve the robustness of our analyses since we can restore the exact state of any computation we performed. In addition, this is all possible with very little overhead.

1.2.2 Why use Distributed Version Control?

The original VCSes used a central server to allow communication among many developers. Each developer in turn would be allowed to modify the master copy. This model has severe problems with availability, reliability, and scalability. Using a distributed VCS, such as Git, Mercurial or Darcs, clients do not check out the latest snapshot of the files, but rather fully independent copies of the repository which communicate among themselves by exchanging sets of changes (the groupoid operation). This removes the server as a single point of failure, since each repository contains all the history data, and as a bottleneck for query and store operations, since there is not central repository through which all communication must flow. Moreover, the repository is fully functional in isolation, such as on an airplane.

In a distributed VCS, each repository is a fully autonomous participant, and thus you can assemble much more complex arrangements among them than the simple star graph of centralized systems. For example, we can create hierarchical workflows which successive migrate changes up the chain as testing is completed, or more general graphs which allow staging areas to combine changes from different developers before merging to the main development line.

1.2.3 Why use Git?

Choosing among VCSes has much in common with choosing among numerical algorithms for a given problem. The accuracy guarantees for VCSes are largely the same, as often happens with algorithms, and thus we are led to also consider its performance, flexibility, and extensibility. Git is quite fast compared to other

VCSes, as well as being scalable to large projects with many contributors, such as the Linux kernel. It has good integration with other tools, such as Emacs and web browsers, and there is a very active tool developer community based on the Git model.

An area where the flexibility of Git really pays off is developer collaboration. The lightweight branching model in Git makes it easy to start a new line of development, incorporate changes from another branch, and manage a complex library workflow. Moreover, once a branch is deleted, it leaves no permanent trace so that history can be cleanly maintained.

1.2.4 How do I learn Git?

One virtue of Git is that is has outstanding, freely available documentation. A comprehensive manual, complete with tutorial, exists online at http://git-scm.com/documentation. In addition, the difficult subject of branching and merging is covered by an interactive, graphical web application which is really fantasic and easy to use. There is even a brief history of Git.

I will assign a few simple exercises to start you off with Git, however the best way to learn anything is to do something truly useful with it. I recommend picking a smaller, but important and fun task to manage with Git. For example, writing a paper, organizing your personal library, or keeping track of your homework assignments in this class. The online tutorials and documentation can provide all the technical help necessary to get such an effort off the ground.

1.2.5 How do I submit a Pull Request?

When many developers are collaborating on a project, especially when they are geographically distributed, it becomes more difficult to coordinate the work, and you need new, higher level organizational mechanisms. A *pull request* is a way both to alert other developers that a particular branch should be merged into the main development line, and to discuss/amend that branch. There are good introductions to pull requests on both Bitbucket and Github.

However, the shortcomings of the Git interface are apparent when dealing with pull requests. Git itself provides no facility for managing or enforcing policy decisions, such as which branches are appropriate for merging what kinds of changes. These policies are now merely documented, such as the PETSc policy for integrating pull requests. This relies in turn on the organization of branches in PETSc shown in Fig. 1.1.

1.2.6 Why use a multi-branch workflow?

The integration workflow known as *gitworkflows* is a multi-branch organization developed to accomodate incoming patches from many distributed contributors. It is nicely explained on the gitworkflows(7) manpage, although the explanation is couched in the language of a patch-based system rather than a pull-request system. The maintainer gets a patch series on the mailing list and pipes the

mailbox through 'git am' to apply that patch series into local branches. The local branch should start from `master` for new features, and `maint` for bug fixes relevant to the last release. Patches frequently go through several revisions before being finalized, but in the meantime, the maintainer puts promising-looking series into named topic branches and merges those into a throw-away integration branch called `pu` (proposed updates). Patches there will probably be reworked before they become permanent history, and `pu` gets rebuilt frequently (every day or two). This organization provides an easy way for people to try out a system with all the intermediate work from other developers. A throw-away merge into `pu` lets a developer find out if their work might conflict with someone else's work without having to monitor the mailing list. You can see the maintainer branches from Git at https://github.com/gitster/git/branches.

Only the `maint`, `master`, `next`, and `pu` branches are in normal repositories. When a topic branch is thought to be complete (perhaps after being re-rolled or fixed up in review), it is merged to `next` where it undergoes testing. If this is successful, then the topic branch is merged to `master`, a step often called "graduation". If all branches graduate in a release cycle, then

```
'git␣log␣master..next'
```

which shows what is in `next`, but not in `master`, would show only merge commits from when the topics were tested.

Mailing list review and integration is an effective workflow for mature projects in which all participants are competent with their tools. However, it takes considerable discipline to structure commits as to be easily reviewable in discrete units and requires using sophisticated email clients. In PETSc, we follow the same principle of testing in `next` before graduating to `master`, but we do not designate a sole integrator. Instead, we all push our topic branches to the same repository, which means that running `git fetch` automatically gets all current development branches. Pushing a branch to the shared repository without merging it into `next` offers the opportunity for passive review. PETSc also accepts pull requests which are merged to named topic branches, and thereafter look like normal development. A group of core PETSc developers have historically done all the integration, where this means merging to any of `next`, `master`, or `maint`, but that circle is expanding. If named topic branches reside on a central server, as is done for PETSc or https://github.com/gitster/git, then

```
comm -12 <(git branch -r --no-merged master | sort) <(git branch -r --merged next | sort)
```

shows the name of all the topic branches that have not yet graduated. This can be used in combination with the decorated log command below which locates change sets in `next` but not in `master`.

```
git log --graph --decorate --oneline --topo-order --no-merges master..next
```

Another popular organization is known as git-flow, which does away with the integration branches, `next` and `pu`, keeping just a `develop` branch (like `master` above) and `master` (like `maint` above). However, this means that new features do not interact before they are merged to `develop`. Instead of having topic branches tested together in `next`, an integrator has to be make the difficult decision of

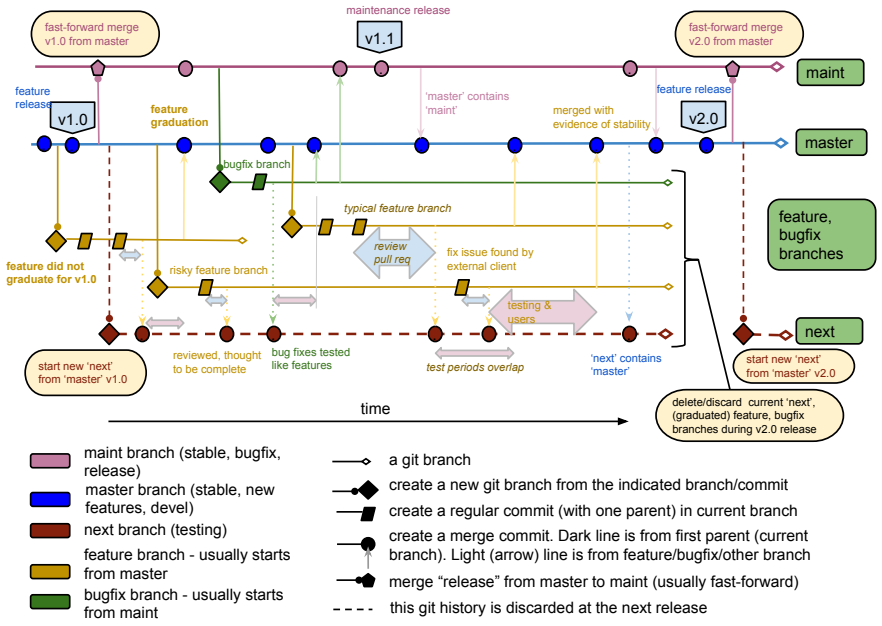


Figure 1.1: This shows the PETSc development branch organization, together with the associated workflow.

whether a topic branch that has only been tested in isolation should be merged into `develop`, or in the case of bugfixes `master`. This generally means that a gitflow `develop` is less stable than a gitworkflows `master`, so it's more common to be working on a new feature and find bugs that were there when you started. This is disruptive to development and its unpleasant for advanced users who often follow the development branch rather than discrete releases. Also any decision to merge a topic branch comes with the pressure that any bugs introduced in the merge (possibly through indirect semantic conflicts with other work) will disrupt developers starting new work between the merge and the time a fix is provided.

In PETSc, we have chosen to use a `maint`/`master`/`next` model, eliding `pu`, since it is simple, produces clean history, and places less burden on the integrator to never make mistakes. No changes make it to `master` or `maint` without first being tested *in combination* with other new features in `next`. Since all new development starts from either `maint`, in the case of bug fixes, or `master` for features, bugs in `next` only affect the integration process, not the development of new code. Starting named topic branches from a stable state is important so that the developer knows that any new bugs have been introduced only in that branch, which prevents needing to merge from upstream in order to fix bugs introduced prior to the branch.

1.3 Configuring and Building

Much like the languages themselves, the mechanics of compiling source and linking executables is dealt with exhaustively online, and many sources are cited on the webpage. I would like to discuss the reasons behind different build architectures and how they help us manage the complexity inherent in writing and maintaining portable code.

1.3.1 What is a build?

The build stage compiles source to object files, stores them somehow (usually in archives), and links shared libraries and executables. These are mechanical operations that reduce to applying a construction rule to sets of files. The make tool is great at this job. However, other parts of make are not as useful, and we should distinguish the two.

Make uses a single predicate, *older than*, to decide whether to apply a rule. This is a disaster. A useful upgrade to make would expand the list of available predicates, including things like *md5sum has changed* and *flags have changed*. There have been attempts to use make to determine whether a file has changed, for example by using stamp files. However, it cannot be done without severe contortions which make it much harder to see what make is doing and maintain the system. Right now, we can combine make with the ccache utility to minimize recompiling and relinking.

1.3.2 Why is configure necessary?

The *configure* program is designed to assemble all information and preconditions necessary for the build stage. This is a far more complicated task, heavily dependent on the local hardware and software environment. It is also the source of nearly every build problem. The most crucial aspect of a configure system is not performance, scalability, or even functionality, it is *debuggability*. Configuration failure is at least as common as success, due to broken tools, operating system upgrades, hardware incompatibilities, user error, and a host of other reasons. Problem diagnosis is the single biggest bottleneck for development and maintenance time. Unfortunately, current systems are built to optimize the successful case rather than the unsuccessful. In PETSc, we have developed the BuildSystem package (BS) to remedy the shortcomings of configuration systems such as Autoconf, CMake, and SCons.

First, BS provides consistent namespacing for tests and test results. Tests are encapsulated in modules, which also hold the test results. Thus you get the normal Python namespacing of results. Anyone familiar with Autoconf will recall the painful, manual namespacing using text prefixes inside the flat, global namespace. Also, this consistent hierarchical organization allows external command lines to be built up in a disciplined fashion, rather than the usual practice of dumping all flags into global reservoirs such as the INCLUDE and LIBS variables. This encapsulation makes it much easier to see which tests are responsible for

donating offending flags and switches when tests fail, since errors can occur far away from the initial inclusion of a flag.

1.3.3 Why use PETSc BuildSystem?

PETSc provides a fully functional configure model implemented in Python, named BuildSystem (BS), which has also been used as the configuration tool for other open sources packages. As more than a few configuration tools currently exist, it is instructive to consider why PETSc would choose to create another from scratch. Below we list features and design considerations which lead us to prefer BuildSystem to the alternatives.

Namespacing BS wraps collections of related tests in Python modules, which also hold the test results. Thus results are accessed using normal Python namespacing. As rudimentary as this sounds, no namespacing beyond the use of variable name prefixes is present in SCons, CMake, or Autoconf. Instead, a flat namespace is used, mirroring the situation in C. This tendency appears again when composing command lines for extenral tools, such as the compiler and linker. In the traditional configure tools, options are aggregated in a single bucket variable, such as INCLUDE or LIBS, whereas in BS you trace the provenance of a flag before it is added to the command line. CMake also makes the unfortunate decision to force all link options to resolve to full paths, which causes havoc with compiler-private libraries.

Explicit control flow The BS configure modules mention above, containing one configure object per module, are organized explicitly into a directed acyclic graph (DAG). The user indicates dependence, an *edge* in the dependence graph, with a single call, `requires('path.to.other.test', self)`, which not only structures the DAG, but returns the configure object. The caller can then use this object to access the results of the tests run by the dependency, achieving test and result encapsulation simply.

Multi-languages tests BS maintains an explicit language stack, so that the current language can be manipulated by the test environment. A compile or link can be run using any language, complete with the proper compilers, flags, libraries, etc with a single call. This kind of automation is crucial for cross-language tests, which are very thinly supported in current tools. In fact, the design of these tools inhibits this kind of check. The `check_function_exists()` call in Autoconf and CMake looks only for the presence of a particular symbol in a library, and fails in C++ and on Windows, whereas the equivalent BS test can also take a declaration. The `try_compile()` test in Autoconf and CMake requires the entire list of libraries be present in the LIBS variable, providing no good way to obtain libraries from other tests in a modular fashion. As another example, if the user has a dependent library that requires `libstdc++`, but they are working with a C project, no straightforward method exists to add this dependency.

Subpackages The most complicated, but perhaps the most useful part of BS is the support for dependent packages. It provides an object scaffolding for including a 3rd party package (more than 60 are now available) so that PETSc downloads, builds, and tests the package for inclusion. The native configure and build system for the package is used, and special support exists for GNU and CMake packages. No similar system exists in the other tools, which rely on static declarations, such as `pkg-config` or `FindPackage.cmake` files, that are not tested and often become obsolete. They also require that any dependent packages use the same configuration and build system.

Batch environments Most systems, such as Autoconf and CMake, do not actually run tests in a batch environment, but rather require a direct specification, in CMake a "platform file". This requires a human expert to write and maintain the platform file. Alternatively, Buildsystem submits a dynamically generated set of tests to the batch system, enabling automatic cross-configuration and cross-compilation.

Caching Caching often seems like an attractive option since configuration can be quite time-consuming, and both Autoconf and CMake enable caching by default. However, no system has the ability to reliably invalidate the cache when the environment for the configuration changes. For example, a compiler or library dependency may be upgraded on the system. Moreover, dependencies between cached variables are not tracked, so that even if some variables are correctly updated after an upgrade, others which depend on them may not be. Moreover, CMake mixes together information which is discovered automatically with that explicitly provided by the user, which is often not tested.

1.3.4 Dealing with Errors

The most crucial piece of an effective configure system is good error reporting and recovery. Most of the configuration process involves errors, either in compiling, linking, or execution, but it can be extremely difficult to uncover the ultimate source of an error. For example, the configuration process might have checked the system BLAS library, and then tried to evaluate a package that depends on BLAS such as PETSc. It receives a link error and fails complaining about a problem with PETSc. However, close examination of the link error shows that BLAS with compiled without position-independent code, e.g. using the `-fPIC` flag, but PETSc was built using the flag since it was intended for a shared library. This is sometimes hard to detect because many 32-bit systems silently proceeed, but most 64-bit systems fail in this case.

When test command lines are built up from options gleaned from many prior tests, it is imperative that the system keep track of which tests were responible for a given flag or a given decision in the configure process. This failure to preserve the chain of reasoning is not unique to configure, but is ubiquitous in software and hardware interfaces. When your Wifi receiver fails to connect

to a hub, or your cable modem to the ISP router, you are very often not told the specific reason, but rather given a generic error message which does not help distinguish between the many possible failure modes. It is essential for robust systems that error reports allow the user to track back all the way to the decision or test which produced a given problem, although it might involve voluminous logging. Thus the system must either be designed so that it creates actionable diagnostics when it fails or it must have unfailingly good support so that human inervention can resolve the problem. The longevity of Autoconf I think can be explained by the ability of expert users to gain access to enough information, possibly by adding `set -x` to scripts and other invasive practices, to act to resolve problems. This ability has been nearly lost in follow-on systems such as SCons and CMake.

Concision is also an important attribute, as the cognitive load is usually larger for larger code bases. The addition of logic to Autoconf and CMake is often quite cumbersome as they do not employ a modern, higher level language. For example, the Trilinos/TriBITS package from Sandia National Laboratory is quite similar to PETSc in the kinds of computations it performs. It contains 175,000 lines of CMakescript used to configure and build the project, whereas PETSc contains less than 30,000 lines of Python code to handle configuration and regression testing and one GNU Makefile of 130 lines.

1.3.5 How do I use make simply and effectively?

We would like a rational, scalable paradigm for building a large, collaborative project. I think this naturally leads to design which centralizes the rules, but distributes the data. A possible answer to this is the recursive build structure employed in PETSc before release 3.5, where makefiles in each source directory are called recursively. Then a developer only modifies the local makefile. However, the arguments given in Recursive Make Considered Harmful are quite convincing. In particular, recursive make does not maintain a consistent DAG so that all dependencies can be checked. PETSc has now moved to a system which uses the old makefile in each directory to list the source files to be built, and then a toplevel makefile. For our example we will use a cleaner version developed for the IBAMR package (Bhalla et al. 2013), using a small `local.mk` file in each subdirectory.

For example, a local makefile lists source files that should be included in the build

```
srcs-core.c += $(call thisdir, \
       fas.c \
       fasfunc.c \
       fasgalerkin.c \
       )
```

where `core` specifies that these objects will be fed into the `libcore` library, and the `.c` suffix indicates the build rules which should be used. The `thisdir` function gives the current directory, so that we get the full path for the source files. For non-terminal directories, we indicate that the inclusion should recurse into the subdirectories,

```
include $(call incsubdirs, interface impls utils)
```

After placing `local.mk` in each subdirectory which will assemble the source list, we want to construct the toplevel control.

```
IBAMR_ARCH := $(if $(IBAMR_ARCH),$(IBAMR_ARCH),build)

all : $(IBAMR_ARCH)/conf/configure.log $(IBAMR_ARCH)/gmakefile
	$(MAKE) -C $(IBAMR_ARCH) -f gmakefile
	@echo "Build␣complete␣in␣$(IBAMR_ARCH).␣␣Use␣make␣test␣to␣test."

$(IBAMR_ARCH)/conf/configure.log:
	./configure.new --IBAMR_ARCH=$(IBAMR_ARCH) --download-muparser \
  --download-eigen --download-silo --download-hdf5 --download-samrai \
  --with-mpi-dir=$(PETSC_DIR)/$(PETSC_ARCH)

$(IBAMR_ARCH)/gmakefile: ./config/gmakegen.py
	$(MAKE) -C $(IBAMR_ARCH) -f bootstrap.mk gmakefile

test : all
	$(MAKE) -C $(IBAMR_ARCH) test

clean :
	$(MAKE) -C $(IBAMR_ARCH) clean

.PHONY: all test clean
```

We notice that two environment variables are used to control the placement of the build, `IBAMR_DIR` specifies the root of the sourse tree, and `IBAMR_ARCH` specifies the build directory for this configuration. Also, we need two files created by the configure process, a `bootstrap.mk` which enables us to access the configure information,

```
gmakefile: ../config/gmakegen.py
	$(PYTHON) ../config/gmakegen.py

include conf/ibamrvariables
```

and the `$PETSC_ARCH/conf/ibamrvariables` file which holds the specialized make variables output by the configure process. The `gmakegen.py` generates a makefile, `gmakefile`, which just has simple toplevel information, and then includes the control file. The current IBAMR version is shown below, which handles package dependecies. This information could, of course, be put into the `ibamrvariables` file, but this divison is sometimes cleaner.

```
PYTHON = ${HOME}/MacSoftware/bin/python2.7

PETSC_DIR = /PETSc3/petsc/petsc-pylith
PETSC_ARCH = arch-next-ibamr-debug
include ${PETSC_DIR}/conf/variables

SAMRAI_DIR = /PETSc3/fluids/IBAMR

BOOST_DIR = /PETSc3/fluids/IBAMR/IBAMR/ibtk/contrib/boost

EIGEN_DIR = /PETSc3/fluids/IBAMR/IBAMR/ibtk/contrib/eigen-3.2.1

MUPARSER_DIR = /PETSc3/fluids/IBAMR/IBAMR/ibtk/contrib/muparser_v2_2_3

IBAMR_DIR = /PETSc3/fluids/IBAMR/IBAMR
IBAMR_ARCH = arch-next-ibamr-debug
IBAMR_INCLUDES = -I${IBAMR_DIR}/${IBAMR_ARCH}/include -I${IBAMR_DIR}/include \
  -I${IBAMR_DIR}/ibtk/include -I${SAMRAI_DIR}/include -I${BOOST_DIR} \
```

```
  -I${EIGEN_DIR} -I${MUPARSER_DIR}/include
CFLAGS += ${IBAMR_INCLUDES} -DNDIM=2
include ${IBAMR_DIR}/${IBAMR_ARCH}/conf/ibamrvariables

include ${IBAMR_DIR}/base.mk
```

Our last requirement is the `base.mk` file which contains toplevel control information. Since it is somewhat long, we will explain it in stages. First we start by configuring make itself, setting up our directory structure, and working around a Cygwin bug when debugging the build process.

```
.SECONDEXPANSION: # to expand $$(@D)/.DIR
.SUFFIXES: # Clear .SUFFIXES because we don't use implicit rules
.DELETE_ON_ERROR: # Delete likely-corrupt target file if rule fails

OBJDIR := $(abspath obj)
LIBDIR := $(abspath lib)
BINDIR ?= bin
INCDIR ?= include

##### Workarounds for Broken Architectures #####
# old cygwin versions
ifeq ($(PETSC_CYGWIN_BROKEN_PIPE),1)
ifeq ($(shell basename $(AR)),ar)
  V ?=1
endif
endif
ifeq ($(V),)
  quiet_HELP := "Use␣\"$(MAKE)␣V=1\"␣to␣see␣the␣verbose␣compile␣lines.\n"
  quiet = @printf $(quiet_HELP)$(eval quiet_HELP:=)"␣␣%10s␣%s\n" "$1$2" "$@"; $($1)
else ifeq ($(V),0)  # Same, but do not print any help
  quiet = @printf "␣␣%10s␣%s\n" "$1$2" "$@"; $($1)
else    # Show the full command line
  quiet = $($1)
endif
```

Next we setup the library versioning by extracting the version number from a header written by the configure process (or possibly consrtucted by hand).

```
##### Versioning #####
IBAMR_VERSION_MAJOR := $(shell awk '/\#define IBAMR_VERSION_MAJOR/{print $$3;}' \
  ./include/ibamrversion.h)
IBAMR_VERSION_MINOR := $(shell awk '/\#define IBAMR_VERSION_MINOR/{print $$3;}' \
  ./include/ibamrversion.h)
IBAMR_VERSION_SUBMINOR := $(shell awk '/\#define IBAMR_VERSION_SUBMINOR/{print $$3;}' \
  ./include/ibamrversion.h)
IBAMR_VERSION_RELEASE := $(shell awk '/\#define IBAMR_VERSION_RELEASE/{print $$3;}' \
  ./include/ibamrversion.h)
ifeq ($(IBAMR_VERSION_RELEASE),0)
  IBAMR_VERSION_MINOR := 0$(IBAMR_VERSION_MINOR)
endif
libibamr_abi_version := $(IBAMR_VERSION_MAJOR).$(IBAMR_VERSION_MINOR)
libibamr_lib_version := $(libibamr_abi_version).$(IBAMR_VERSION_SUBMINOR)
```

We then define some functions for name manipulation. Note that the `thisdir` and `incsubdirs` functions we saw before are defined here.

```
##### Functions #####
# Function to name shared library $(call SONAME_FUNCTION,libfoo,abiversion)
SONAME_FUNCTION ?= $(1).$(SL_LINKER_SUFFIX).$(2)
soname_function = $(call SONAME_FUNCTION,$(1),$(libibamr_abi_version))
libname_function = $(call SONAME_FUNCTION,$(1),$(libibamr_lib_version))
# Function to link shared library
# $(call SL_LINKER_FUNCTION,libfoo,abiversion,libversion)
SL_LINKER_FUNCTION ?= -shared -Wl,-soname,$(call SONAME_FUNCTION,$(notdir $(1)),$(2))
basename_all = $(basename $(basename $(basename $(basename $(1)))))
```

```
sl_linker_args = $(call SL_LINKER_FUNCTION,$(call basename_all,$@),\
                 $(libibamr_abi_version),$(libibamr_lib_version))
# Function to prefix directory that contains most recently-parsed
# makefile (current) if that directory is not ./
thisdir = $(addprefix $(dir $(lastword $(MAKEFILE_LIST))),$(1))
# Function to include makefile from subdirectories
incsubdirs = $(addsuffix /local.mk,$(call thisdir,$(1)))
# Function to change source filenames to object filenames
srctoobj = $(patsubst $(SRCDIR)/%.c,$(OBJDIR)/%.o,$(filter-out $(OBJDIR)/%,$(1)))
```

We must name the libraries to be built, distinguishing between static and shared builds. Here we will build a single library, `libibamr`.

```
##### Libraries #####
libibamr_shared := $(LIBDIR)/libibamr.$(SL_LINKER_SUFFIX)
libibamr_soname := $(call soname_function,$(LIBDIR)/libibamr)
libibamr_libname := $(call libname_function,$(LIBDIR)/libibamr)
libibamr_static := $(LIBDIR)/libibamr.$(AR_LIB_SUFFIX)
libibamr := $(if $(filter-out no,$(BUILDSHAREDLIB)),$(libibamr_shared) \
                  $(libibamr_soname),$(libibamr_static))
```

Then for each package, or unit in the library, we must define the source list. Here we have two packages, `core` and `ibtk`. We are defining them for C++, but we could use C, Fortran, or any other language extension.

```
##### Must define these, or thisdir does not work ######
srcs-core.cpp :=
srcs-ibtk.cpp :=
```

Finally, we are ready to define the source lists by recursively including all the `local.mk` files.

```
##### Inclusions #####
# Recursively include files for all targets, need to be defined before source rules
include $(IBAMR_DIR)/local.mk
```

After all the setup, we can define the compilation rules using make variables from `conf/ibamrvariables` determined by the congiure process.

```
##### Rules #####
all : $(libibamr)

# make print VAR=the-variable
print:
	@echo $($(VAR))

IBAMR_COMPILE.c = $(call quiet,$(cc_name)) -c $(PCC_FLAGS) $(CFLAGS) $(CCPPFLAGS) \
                  $(C_DEPFLAGS)
IBAMR_COMPILE.cxx = $(call quiet,CXX) -c $(CXX_FLAGS) $(CFLAGS) $(CCPPFLAGS) \
                  $(CXX_DEPFLAGS)
IBAMR_COMPILE.cu = $(call quiet,CUDAC) -c $(CUDAC_FLAGS) \
                  --compiler-options="$(PCC_FLAGS)␣$(CXXFLAGS)␣$(CCPPFLAGS)"
IBAMR_GENDEPS.cu = $(call quiet,CUDAC,.dep) --generate-dependencies $(CUDAC_FLAGS) \
                  --compiler-options="$(PCC_FLAGS)␣$(CXXFLAGS)␣$(CCPPFLAGS)"
IBAMR_COMPILE.F = $(call quiet,FC) -c $(FC_FLAGS) $(FFLAGS) $(FCPPFLAGS) \
                  $(FC_DEPFLAGS)
```

Now we define a set a packages, supported languages, and the complete object list for the library. Then the library link rule is straightforward, as is the archiving rule (putting object files into a .a file), however there is some fixup to support the Cygwin OS.

```
pkgs := ibtk core
langs := c cu cpp F
```

```
concatlang = $(foreach lang, $(langs), $(srcs-$(1).$(lang):%.$(lang)=$(OBJDIR)/%.o))
srcs.o := $(foreach pkg, $(pkgs), $(call concatlang,$(pkg)))

$(libibamr_libname) : $(srcs.o) | $$(@D)/.DIR
       $(call quiet,CLINKER) $(sl_linker_args) -o $@ $^ $(PETSC_EXTERNAL_LIB_BASIC)
ifneq ($(DSYMUTIL),true)
       $(call quiet,DSYMUTIL) $@
endif

$(libibamr_static) : obj := $(srcs.o)

define ARCHIVE_RECIPE_WIN32FE_LIB
  @$(RM) $@ $@.args
  @cygpath -w $^ > $@.args
  $(call quiet,AR) $(AR_FLAGS) $@ @$@.args
  @$(RM) $@.args
endef

define ARCHIVE_RECIPE_DEFAULT
  @$(RM) $@
  $(call quiet,AR) $(AR_FLAGS) $@ $^
  $(call quiet,RANLIB) $@
endef

%.$(AR_LIB_SUFFIX) : $$(obj) | $$(@D)/.DIR
       $(if $(findstring win32fe lib,$(AR)),$(ARCHIVE_RECIPE_WIN32FE_LIB),\
     $(ARCHIVE_RECIPE_DEFAULT))
```

We can now declare the dependence relations, including a switch to allow relinking everything. There is a significant optimization at the end for dependencies arising from `*.d` files. These are produces by the compiler to automatically track dependecies arising in the source code.

```
# The package libraries technically depend on each other (not just in an order-only
# way), but only ABI changes like new or removed symbols requires relinking the
# dependent libraries. ABI should only occur when a header is changed, which would
# trigger recompilation and relinking anyway.
# RELINK=1 causes dependent libraries to be relinked anyway.
ifeq ($(RELINK),1)
  libdep_true = $$(libdep)
  libdep_order =
else
  libdep_true =
  libdep_order = $$(libdep)
endif
$(libpetscpkgs_libname) : $(libdep_true) | $(libdep_order) $$(@D)/.DIR
       $(call quiet,CLINKER) $(sl_linker_args) -o $@ $^ $(PETSC_EXTERNAL_LIB_BASIC)
ifneq ($(DSYMUTIL),true)
       $(call quiet,DSYMUTIL) $@
endif

%.$(SL_LINKER_SUFFIX) : $(call libname_function,%)
       @ln -sf $(notdir $<) $@

$(call soname_function,%) : $(call libname_function,%)
       @ln -sf $(notdir $<) $@

$(OBJDIR)/%.o : %.c | $$(@D)/.DIR
       $(IBAMR_COMPILE.c) $(abspath $<) -o $@

$(OBJDIR)/%.o : %.cpp | $$(@D)/.DIR
       $(IBAMR_COMPILE.cxx) $(abspath $<) -o $@

$(OBJDIR)/%.o : %.cu | $$(@D)/.DIR
    # Compile first so that if there is an error, it comes from a normal compile
       $(IBAMR_COMPILE.cu) $(abspath $<) -o $@
    # Generate the dependencies for later
```

```
	@$(IBAMR_GENDEPS.cu) $(abspath $<) -o $(@:%.o=%.d)

$(OBJDIR)/%.o : %.F | $$(@D)/.DIR
ifeq ($(FC_MODULE_OUTPUT_FLAG),)
	cd $(MODDIR) && $(FC) -c $(FC_FLAGS) $(FFLAGS) $(FCPPFLAGS) $(FC_DEPFLAGS) \
     $(abspath $<) -o $(abspath $@)
else
	$(IBAMR_COMPILE.F) $(abspath $<) -o $@ $(FC_MODULE_OUTPUT_FLAG)$(MODDIR)
endif

%/.DIR :
	@mkdir -p $(@D)
	@touch $@

.PRECIOUS: %/.DIR

allobj.d := $(srcs.o:%.o=%.d)
# Tell make that allobj.d are all up to date. Without this, the include
# below has quadratic complexity, taking more than one second for a
# do-nothing build of PETSc (much worse for larger projects)
$(allobj.d) : ;

-include $(allobj.d)
```

GNU make is installed on virtually every UNIX system in existence. Moreover, compatible make clones exist on most other systems as well. This unbiquity, and its efficiency, make it a very attractive option. However, the fact that it recognizes only a single predicate, "older than", is a significant drawback. In order to remedy this, most practitioners replace the compiler, say `cc`, with ccache, as `ccache cc`, by providing that combination to configure. The ccache program hashes the entire command string and file, so that rebuilds are only performed when an actual change is made.

1.3.6 What about pkg-config?

An alternative to using the PETSc make files for variable definition is the pkg-config program. PETSc writes out a pkg-config file, which can then be read by the program, which gives the users access to the variables. Here is a sample makefile using pkg-config to access the variables

```
PETSc.pc := $(PETSC_DIR)/$(PETSC_ARCH)/lib/pkgconfig/PETSc.pc

CC := $(shell pkg-config --variable=ccompiler $(PETSc.pc))
CXX := $(shell pkg-config --variable=cxxcompiler $(PETSc.pc))
FC := $(shell pkg-config --variable=fcompiler $(PETSc.pc))
CFLAGS := $(shell pkg-config --variable=cflags_extra $(PETSc.pc)) $(shell pkg-config --cflags-only-other $(PETSc.pc))
CPPFLAGS := $(shell pkg-config --cflags-only-I $(PETSc.pc))
LDFLAGS := $(shell pkg-config --libs-only-L --libs-only-other $(PETSc.pc))
LDFLAGS += $(patsubst -L%, $(shell pkg-config --variable=ldflag_rpath $(PETSc.pc))%, $(shell pkg-config --libs-only-L $(PETSc.pc)))
LDLIBS := $(shell pkg-config --libs-only-l $(PETSc.pc)) -lm

print:
	@echo CC=$(CC)
	@echo CFLAGS=$(CFLAGS)
	@echo CPPFLAGS=$(CPPFLAGS)
	@echo LDFLAGS=$(LDFLAGS)
	@echo LDLIBS=$(LDLIBS)
```

The only minor sophistication here is the use of `patsubst` to replace `-L` with the `rpath` flags for shared libraries.

1.3.7 How do I debug?

All debugging comes down to a search for problematic code, configuration, or input. Effective debugging cuts down on the possible search space as much as possible. Categorizing bugs can help with this, as can employing effective tools.

The most common type of bug in C is probably the memory overwrite, often generating an SEGV or SIGILL signal. If the error is proximate to the ultimate cause, then running in the debugger is the best way to deal with this situation. For example, suppose that a NULL pointer is passed into your routine instead of a valid pointer. A write to this location (0x0) causes a SEGV which the debugger will catch and take you directly to the offending line. Using a stack trace, you can easily locate the origin of the NULL pointer. The debugger will also allow you to print the value of local variables, step through the code, and call arbitrary functions.

However, it is often the case that errors are far removed from their ultimate cause. For example, suppose that a memory overwrite occurs not into space owned by the kernel, such as NULL, but into memory owned by the application itself. This could result in invalid values, such as array indices, which then cause a subsequent SEGV. Finding the original overwrite which corrupted the index using the debugger can be quite challenging. For this type of problem, and many others, the best tool is valgrind. It will flag any out of bounds read or write, as well as other common errors such as a conditional dependent on an uninitialized value.

Valgrind has an extensible module interface, allowing it to encompass many different tools:

Tool	Use
Memcheck	Check for memory overwrite and illegal use
Callgrind	Generate call graphs
Cachegrind	Monitor cache usage
Helgrind	Check for thread race conditions
Massif	Monitor memory usage

The most commonly used tool for debugging, Memcheck, will catch illegal reads and writes to memory, uninitialized values, illegal frees, overlapping copies, and memory leaks. We can try a simple experiment

```
# Get the tutorial repository
git clone http://bitbucket.org/knepley/simplepetsctutorial.git
git checkout b354cfc
make
# Memcheck is the default tool
valgrind --trace-children=yes --suppressions=bin/simple.supp ./bin/ex5 -use_coords
# Try it for multiple processes
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  $PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 2 ./bin/ex5 -use_coords
```

which generates the following error,

```
==13697== Invalid read of size 8
==13697== at 0x100005263: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13697== by 0x100004447: main (ex5.c:202)
==13697== Address 0x103dc6fa0 is 0 bytes after a block of size 48 alloc'd
```

```
==13697==␣␣␣␣at␣0x10001ED75:␣malloc␣(vg_replace_malloc.c:236)
==13697==␣␣␣␣by␣0x1005CABC4:␣PetscMallocAlign(unsigned␣long,␣...)␣(mal.c:37)
==13697==␣␣␣␣by␣0x1009CC07D:␣VecGetArray2d(_p_Vec*,␣...)␣(rvector.c:1739)
==13697==␣␣␣␣by␣0x10030D980:␣DMDAVecGetArray(_p_DM*,␣_p_Vec*,␣void*)␣(dagetarray.c:72)
==13697==␣␣␣␣by␣0x100005102:␣MyInitialGuess(AppCtx*,␣_p_Vec*)␣(myStuff.c:38)
==13697==␣␣␣␣by␣0x100004447:␣main␣(ex5.c:202)
==13697==
==13697==␣Invalid␣read␣of␣size␣8
==13697==␣␣␣␣at␣0x100005273:␣MyInitialGuess(AppCtx*,␣_p_Vec*)␣(myStuff.c:45)
==13697==␣␣␣␣by␣0x100004447:␣main␣(ex5.c:202)
==13697==␣␣Address␣0x18␣is␣not␣stack'd, malloc'd␣or␣(recently)␣free'd
==13697==
==13698== Use of uninitialised value of size 8
==13698== at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698== by 0x100004447: main (ex5.c:202)
==13698==
==13698== Invalid read of size 8
==13698== at 0x10000529D: MyInitialGuess(AppCtx*, _p_Vec*) (myStuff.c:45)
==13698== by 0x100004447: main (ex5.c:202)
==13698== Address 0x6f5c300000018 is not stack'd,␣malloc'd or (recently) free'd
```

due to indexing outside of an array. We are trying to retrieve the coordinates for a ghost point rather than an owned point, and thus we have to use properly ghosted coordinates.

```
git checkout 1a1c683
make
valgrind --trace-children=yes --suppressions=bin/simple.supp \
  $PETSC_DIR/$PETSC_ARCH/bin/mpiexec -n 2 ./bin/ex5 -use_coords
```

Note that the `--trace-children=yes` flag is used to follow all child processes. This is very useful when running with MPI, as it typically forks before running the executable. In order to make the output more readable, a *suppressions* file can be used to remove output for errors or warnings which we do not wish to see. This file can be generated automatically by piping the output of the following command.

```
valgrind --trace-children=yes --gen-suppressions=all ./bin/ex5 -ksp_rtol 1.0e-9
```

The Massif tool for memory logging is similarly easy to use

```
# Memcheck is the default tool
valgrind --tool=massif --trace-children=yes --massif-out-file=ex5.massif \
  ./bin/ex5 -da_grid_x 100 -da_grid_y 100 -ksp_rtol 1.0e-9
# Turn on stack profiling
valgrind --tool=massif --trace-children=yes --massif-out-file=ex5.massif --stacks=yes \
  ./bin/ex5 -da_grid_x 100 -da_grid_y 100 -ksp_rtol 1.0e-9
# Visualize output
ms_print --threshold=10.0 ex5.massif
```

At a much coarser level, PETSc will provide a list of all unfreed memory, along with stack traces for context, with the option `-malloc_dump`.

In addition to debugging for correctness, we will also debug performance problems. The massif tool, incorporated into valgrind, can generate a view of the heap over time, allowing the user to pinpoint large allocations or unfreed memory. The output of callgrind can be rendered graphically using the gprof2dot tool

```
gprof2dot.py -f callgrind callgrind.out.x | dot -Tsvg -o output.svg
```

There are tools dedicated to performance analysis, such as callgrind and cachegrind inside valgrind and many more are suggested here. However these

tools tend to provide very low level data, and obscure the larger performance patterns in the code. These low level tools should be complemented with a high level, coarse grained timing of functional units, incorporating figures of merit for parallelism such as message sizes and number of reductions. PETSc provides a logging suite for this purpose which allows custom events, user specified event aggregation, and counter for parallel operations (Balay, Abhyankar, Adams, Benson, Brown, Brune, Buschelman, E. Constantinescu, et al. 2022).

Most debugging depends on taking a discplined, step-by-step approach to tracking down errors. Ruthlessly simplify the problem being run, as well as the environment, and remove extraneous detail in the computation. As a simple checklist for debugging, always

- Run in serial,
- Reduce the problem size,
- Run with valgrind, and
- Run the code with a manufactured solution so that you can calculate an error.

Simple Example We will illustrate the use of these debugging tools with a simple example based upon `SNES ex5`. We first clone the repository for the example,

```
git clone https://bitbucket.org/knepley/simplepetscexample.git example
cd example
```

and then skip forward after a user modification which introduces a bug

```
git checkout 857854f
```

We can examine the changeset

```
> git log -1
commit 857854fffca425a1c2673a5a0b4c229a4a12e3fb
Author: Matthew G. Knepley <knepley@gmail.com>
Date: Thu Aug 13 05:07:20 2015 -0500

    Added MyInitialGuess() which uses coordinates
    - Added logging
    - Moved declarations to a header
```

which gives us an idea where the bug may lie.

Now we build the executable, after checking that `PETSC_DIR` is properly defined in our environment,

```
  make
```

and we can run the example using the new code,

```
> ./bin/ex5 -snes_monitor -use_coords
  0 SNES Function norm 9.875766933286e-01
  1 SNES Function norm 7.968273013801e-01
  2 SNES Function norm 2.649139987149e-01
  3 SNES Function norm 1.130203891627e-01
  4 SNES Function norm 9.649349335803e-03
  5 SNES Function norm 6.634259652040e-05
  6 SNES Function norm 4.669629893639e-09
```

with no errors. However, when we run in parallel

```
> mpiexec -n 2 ./bin/ex5 -snes_monitor -use_coords
[0]PETSC ERROR: ------------------------------------------------------------------------
[0]PETSC ERROR: Caught signal number 11 SEGV: Segmentation Violation, probably memory access out of range
[0]PETSC ERROR: Try option -start_in_debugger or -on_error_attach_debugger
[0]PETSC ERROR: or see http://www.mcs.anl.gov/petsc/documentation/faq.html#valgrind
[0]PETSC ERROR: or try http://valgrind.org on GNU/linux and Apple Mac OS X to find memory corruption errors
[0]PETSC ERROR: likely location of problem given in stack below
[1]PETSC ERROR: ------------------------------------------------------------------------
[1]PETSC ERROR: Caught signal number 11 SEGV: Segmentation Violation, probably memory access out of range
[1]PETSC ERROR: Try option -start_in_debugger or -on_error_attach_debugger
[1]PETSC ERROR: or see http://www.mcs.anl.gov/petsc/documentation/faq.html#valgrind
[1]PETSC ERROR: or try http://valgrind.org on GNU/linux and Apple Mac OS X to find memory corruption errors
[1]PETSC ERROR: likely location of problem given in stack below
[1]PETSC ERROR: --------------------- Stack Frames ------------------------------------
[1]PETSC ERROR: Note: The EXACT line numbers in the stack are not available,
[1]PETSC ERROR: INSTEAD the line number of the start of the function
[1]PETSC ERROR: is given.
[1]PETSC ERROR: [1] MyInitialGuess line 24 /PETSc3/classes/CAAM519/simplepetscexample/src/myStuff.c
[1]PETSC ERROR: --------------------- Error Message --------------------------------------------------------------
[1]PETSC ERROR: Signal received
[1]PETSC ERROR: See http://www.mcs.anl.gov/petsc/documentation/faq.html for trouble shooting.
[1]PETSC ERROR: Petsc Development GIT revision: v3.7.2-669-gecdcb5d GIT Date: 2016-06-16 08:48:26 -0500
[1]PETSC ERROR: ./bin/ex5 on a arch-c-exodus-master named localhost by knepley Thu Aug 18 11:48:08 2016
[1]PETSC ERROR: Configure options --with-shared-libraries
[1]PETSC ERROR: #1 User provided function() line 0 in unknown file
application called MPI_Abort(MPI_COMM_WORLD, 59) - process 1
[cli_1]: aborting job:
```

we encounter an SEGV signal, probably arising from a memory overwrite. Here we are somewhat lucky that our memory overwrite has occurred in a protected memory region. We could have written into another part of memroy owned by the process and silently carried on with wrong data.

With a segfault (SEGV), we can often use a debugger, such as `gdb`, to track down the cause. PETSc can automatically spawn a debugger in a different window and attach it to the running process. When running on the Mac I use `lldb`,

```
mpiexec -n 2 ./bin/ex5 -snes_monitor -use_coords -start_in_debugger lldb
```

which spawns two X-windows running the debugger, one for each process. If we `continue` in both windows, we hit the offending line

```
(lldb) cont
Process 67390 resuming
Process 67390 stopped
* thread #1: tid = 0x1c17b8f, 0x0000000108d0ef6c ex5`MyInitialGuess(da=0x00007ff98589b260,
  user=0x00007fff56ef9760, X=0x00007ff9850b0a60) + 1884
  at myStuff.c:45, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0xf0e0d0e1)
   42 /* boundary conditions are all zero Dirichlet */
   43 x[j][i] = 0.0;
   44 } else {
-> 45 x[j][i] = temp1*sqrt(2.0*PetscMin(coords[j][i+1].x + coords[j][i-1].x, coords[j+1][i].y + coords[j-1][i].y));
   46 }
   47 }
   48 }
```

Now it seems likely that there is a problem with the indexing into the `coords` array, and indeed this is the problem. We can see the fix by looking at the following changeset

```
> git log -1 bc2dc5b -u
commit bc2dc5b2c576306274f8e722190042e899e0a7b3
```

```
Author: Matthew G. Knepley <knepley@gmail.com>
Date: Thu Aug 13 05:08:45 2015 -0500

    Fixed memory error after debugging

diff --git a/src/myStuff.c b/src/myStuff.c
index 3995123..123c670 100644
--- a/src/myStuff.c
+++ b/src/myStuff.c
@@ -32,7 +32,7 @@ PetscErrorCode MyInitialGuess(DM da, AppCtx *user, Vec X) {
   temp1 = lambda/(lambda + 1.0);

   ierr = DMGetCoordinateDM(da,&cda);CHKERRQ(ierr);
- ierr = DMGetCoordinates(da,&c);CHKERRQ(ierr);
+ ierr = DMGetCoordinatesLocal(da,&c);CHKERRQ(ierr);
   ierr = DMDAGetCorners(da,&xs,&ys,PETSC_NULL,&xm,&ym,PETSC_NULL);CHKERRQ(ierr)
   ierr = DMDAVecGetArray(da,X,&x);CHKERRQ(ierr);
   ierr = DMDAVecGetArray(cda,c,&coords);CHKERRQ(ierr);
```

which replaces the global vector from `DMGetCoordinates()` with the local vector from `DMGetCoordinatesLocal()` which contains ghost values. Now the indices $i \pm 1$ and $j \pm 1$ do not fall outside the local patch.

We could also have found this problem using `valgrind`. We must be careful to use the `trace-children` option since MPI spawns additional processes after the initial fork. Running

```
valgrind --trace-children=yes mpiexec -n 2 ./bin/ex5 -snes_monitor -use_coords
```

does work but there is a lot of extraneous output. We can filter this using a suppressions file,

```
> git checkout 6583927
> valgrind --trace-children=yes --suppressions=./binsimple/supp mpiexec -n 2 ./bin/ex5 -snes_monitor -use_coords
==68238== Memcheck, a memory error detector
==68238== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==68238== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==68238== Command: ./bin/ex5 -snes_monitor -use_coords
==68238==
==68238== Invalid read of size 8
==68238==    at 0x100008F61: MyInitialGuess (myStuff.c:45)
==68238==    by 0x100001FB5: main (ex5.c:148)
==68238==  Address 0x10013f9f0 is 0 bytes after a block of size 48 alloc'd
==68238==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==68238==    by 0xF0B31: PetscMallocAlign (mal.c:34)
==68238==    by 0x3D19F0: VecGetArray2d (rvector.c:2235)
==68238==    by 0xCF715C: DMDAVecGetArray (dagetarray.c:75)
==68238==    by 0x100008E07: MyInitialGuess (myStuff.c:38)
==68238==    by 0x100001FB5: main (ex5.c:148)
```

and we see that an invalid read has occurred at the same problematic line of code. In addition, we see that the invalid read is looking at memory allocated by the `DMDAVecGetArray()` call. It is quite useful to see what memory is the target of an invalid read or write.

We can also use massif to look at the memory allocation of this run, although it appears that we must generate one massif output file for every process,

```
mpiexec -n 2 valgrind --tool=massif ./bin/ex5 -snes_monitor -da_grid_x 100 -da_grid_y 100
```

produces output files named `massif.out.<procid>`. We can process these using `msprint` or `pymassif`, although the latter does not seem to be well supported. In Figure ref-fig:ex5massif, we see the massif graphical output, giving a timeline for alloca-

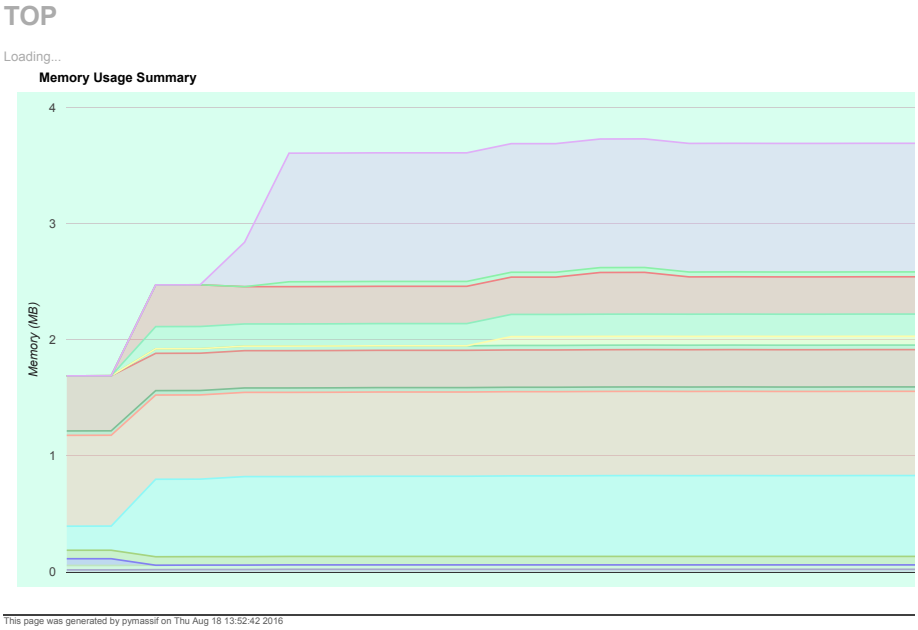


Figure 1.2: Massif output from a parallel run of ex5

tions. In the browser we could mouse-over each snapshot to get the allocation location.

1.4 Problems

Problem I.1 Following the online directions, install the latest release of PETSc.

Problem I.2 Clone my sample repository of PETSc code onto your local machine, https://bitbucket.org/knepley/simplepetscexample. Checkout the ChangeSet with comment "Initial checkin of source".

Problem I.3 Create a repository on Bitbucket in which you will store writing assignments for this course. Commit the LaTeX paragraph you write for Problem II.1 and push it to the repository hosted at Bitbucket.

Problem I.4 Write a makefile that compiles the code in the sample repository from Problem 2 and commit it to your local repository.

Problem I.5 The simple Python script below, in the repository as `bin/plotPerf.py`, runs the sample `ex5` from Problem 2 for a range of problem sizes and plots the

timing.

```
#! /usr/bin/env python
import os

sizes = []
times = []
for k in range(5):
  Nx = 10 * 2**k
  modname = 'perf%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname]
  os.system('./bin/ex5 '+' '.join(options))
  perfmod = __import__(modname)
  sizes.append(Nx ** 2)
  times.append(perfmod.Stages['Main Stage']['SNESSolve'][0]['time'])
print zip(sizes, times)

from pylab import legend, plot, loglog, show, title, xlabel, ylabel
plot(sizes, times)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Time (s)')
show()

loglog(sizes, times)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Time (s)')
show()
```

Notice that the logging information is output in a Python module named `perf1.py` for $k = 1$. Each time we output a module, it must have a different name since Python caches module contents by name.

Modify this Python script to report the linear solver time (`KSPSolve`) instead of the nonlinear solve time (`SNESSolve`), and plot it for the GMRES/ILU (`-ksp_type gmres -pc_type ilu`) and GMRES/GAMG (`-ksp_type gmres -pc_type gamg`) solvers on the same graph. For extra credit, look at the performance as the number of processes increases.

Problem I.6 Modify the script from Problem 5 to report the assembly time (`SNESFunctionEval` and `SNESJacobianEval`), of both the residual and Jacobian, instead of the nonlinear solve time (`SNESSolve`), and plot it for the GMRES/ILU and GMRES/GAMG solvers on the same graph.

References

Forum, Message Passing Interface (2012). *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS).

Gropp, William, Ewing Lusk, and Anthony Skjellum (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

Gropp, William and et. al. (n.d.). *MPICH Web page*. http://www.mpich.org. URL: http://www.mpich.org.

Balay, Satish, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, et al. (2022). *PETSc/-TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.18. Argonne National Laboratory.

Balay, Satish, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, et al. (2022). *PETSc Web page*. https://petsc.org/. URL: https://petsc.org/.

Nachtigal, N.M., S.C. Reddy, and L.N. Trefethen (1992). "How Fast are Nonsymmetric Matrix Iterations?" In: *SIAM Journal on Matrix Analysis and Applications* 13, p. 778.

Björck, Åke (1994). "Numerics of Gram-Schmidt orthogonalization". In: *Linear Algebra and Its Applications* 197, pp. 297–316.

Brown, Jed, Matthew G. Knepley, David A. May, Lois C. McInnes, and Barry F. Smith (2012). "Composable linear solvers for multiphysics". In: *Proceeedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*. IEEE Computer Society, pp. 55–62. DOI: 10.1109/ISPDC.2012.16.

Ketcheson, David I., Kyle T. Mandli, Aron J. Ahmadia, Amal Alghamdi, Manuel Quezada de Luna, Matteo Parsani, Matthew G. Knepley, and Matthew Emmett (2012). "PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems". In: *SIAM Journal on Scientific Computing* 34.4. http://arxiv.org/abs/1111.6583, pp. C210–C231.

Mandli, Kyle T., David I. Ketcheson, et al. (2012). *PyClaw software*. http://clawpack.github.io/doc/pyclaw/. URL: http://clawpack.github.io/doc/pyclaw/.

Collier, Nathan, Lisandro Dalcin, and Victor M. Calo (2013). "PetIGA: High-Performance Isogeometric Analysis". In: *arxiv* 1305.4452. http://arxiv.org/abs/1305.4452.

Collier, N., L. Dalcin, and V.M. Calo (2014). *PetIGA Web page*. https://bitbucket.org/dalcinl/petiga/. URL: https://bitbucket.org/dalcinl/petiga/.

Gropp, William D. (1999). "Exploiting Existing Software in Libraries: Successes, Failures, and Reasons Why". In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM, pp. 21–29.

Yood, Charles Nelson (Aug. 2005). "Argonne National Laboratory and the Emergence of Computer and Computational Science, 1946–1992". PhD thesis. The Pennsylvania State University.

Wikipedia (2015). *Groupoid*. http://en.wikipedia.org/wiki/Groupoid. URL: https://en.wikipedia.org/wiki/Groupoid.

Bhalla, Amneet Pal Singh, Rahul Bale, Boyce E Griffith, and Neelesh A Patankar (2013). "A unified mathematical framework and an adaptive numerical method for fluid–structure interaction with rigid, deforming, and elastic bodies". In: *Journal of Computational Physics* 250, pp. 446–476.

Chapter 2

Finding and Relating Information

2.1 Self-Teaching

The internet has become a repository of mathematical and scientific knowledge every bit as important as books, journals, professors, and colleagues. Its reach is much broader than any single source or even collection of sources, and free preprint services such as arXiv provide an invaluable opportunity to keep abreast of the latest research in a large number of fields. This is especially important for computational science since interdisciplinary understanding is an integral part of the field.

Mathematics in particular, perhaps due to the unanimity in the field, has outstanding web resources, including Wikipedia, Math Geneaology, MathOverflow and its companion SciComp. Wikipedia especially posssess not just sets of facts, but in depth treatments of advanced topics, complete with diagrams and sample code, that rival and sometimes surpass textbooks. As a tool for navigating the literature, Google Scholar is now unequaled. Bibliographic chains can be followed with a few clicks, and now full BibTeX entries are also available directly.

The internet, and in particular Google and Google Scholar, should be your first stop for:

- Resolving compile and link errors,
- Finding packages containing headers or libraries you are missing,
- Finding package documentation or examples,
- Getting BibTeX for missing references,
- Achieving a given effect in LaTeX (see http://tex.stackexchange.com/),
- Achieving a given effect in TikZ,

in addition to all the mathematical and computational resources.

2.2 TEX and LATEX

TEX (Knuth and Bibby 1986), and in particular LATEX (Lamport 1986), is the most important piece of technology for scientific communication. It enables the digital interchange and archiving of scientific communications, including papers, books, reports, posters, talks, etc. In addition, the TEXBook (Knuth and Bibby 1986) is the most outstanding achievements in literature on programming. While LATEX is an excellent typesetting system, some high-level organizational principles can aid writing and maintaining documents, especially those shared with others.

Use a professional style For writing articles and notes, I recommend the SIAM style. It has a bibliographic style which automatically creates links and great math support. It also makes any eventual submission to a journal much easier. I recommend segregating the main text into a separate file so that different styles can be tried in an outer file. For example, I usually submit to a journal, and then put my paper on the arXiv.

Use a preamble I `\input` a file named `preamble.tex` which holds all the TEX code common to my various documents. It includes packages, defines text styles (e.g. for urls), sets colors and fonts, defines custom commands, and defines my `listing` package styles for typesetting code. I also recommend using AMS math package `amsmath`.

The best way to create PDF from LATEX is to use `pdflatex`,

```
pdflatex essay.tex
bibtex essay
pdflatex essay.tex
pdflatex essay.tex
```

where the repetition is necessay to assure that the metadata stored in auxiliary files is consistent. This sequence has traditionally been put into a makefile. However, the process can be handled in a more elegant way by using the `latexmk` program,

```
latexmk -pdf essay.tex
```

If you rely on TEX source or BIBTEX files in other locations, you can use

```
TEXINPUTS=${TEXINPUTS}:/path/to/tex BIBINPUTS=${BIBINPUTS}:/path/to/bib
  latexmk -pdf essay.tex
```

2.3 Problems

Problem II.1 It is quite likely that no matter what profession you choose to pursue after this course, expository writing will form a large part of your workload. Please write a paragraph or two describing what you hope to learn from

this course, suggestions for upcoming units, or broader thoughts on scientific computing and its progress as a discipline. Typeset your work in LaTeX and include at least one citation using BibTeX.

Problem II.2 Create a PDF file from your essay source and submit it by email with the subject *[CAAM 519] Essay I.*

Problem II.3 Using any internet resources available, answer the following questions, providing proper citation for the information you provide:

1. Give the generating function for the sequence $1, 1, 2, 2, 3, 5, 5, 7, 10, 15, 15, 20, 27, 37, \ldots$.
2. Give an asymptotic expansion for the Gamma function $\Gamma(z)$ as $z \to \infty$ with error term.
3. On Ubuntu systems, why can you get the error `ImportError: No module named _md5` when using `import hashlib` in Python?
4. Does the popular nonlinear solver deserve to be called the *Newton-Raphson* method? Why or why not?
5. Who is Leonid Kantorovich?
6. How do I solve the semiconductor equations?

Problem II.4 Using any internet resources available, answer the following questions, providing proper citation for the information you provide:

1. What relation generates the sequence $8, 12, 16, 24, 32, 36, 48, 96, 128, 160, 192, 288, 768, \ldots$?
2. Give an asymptotic expansion for the complete elliptic integral $E(k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$ as $k \to 1$.
3. In Linux, if you receive a linker error with the text "`relocation R_X86_64_32S against symbol` ...", what has happened?
4. Has Hilbert's 13th Problem been solved? If so, who solved it and when.
5. Who invented the Python language? What language did this person work on prior to Python?
6. If I am simulating an incompressible flow, what discretization would be "mass conservative" for these equations?

Problem II.5 Make a contribution to Wikipedia and send the link to your edit.

References

Knuth, Donald Ervin and Duane Bibby (1986). *The TEXBook*. Vol. 1993. Addison-Wesley Reading, Massachusetts.

Lamport, Leslie (1986). *LaTeX: A document preparation system*. Addison-Wesley.

Chapter 3

PETSc Introduction

Change alone is unchanging

— Heraclitus, 544–483 BC

3.1 Numerical Libraries

For pure mathematicians, the *sine non qua* of technical communication is the journal paper, although people like Terence Tao and Timothy Gowers have clearly shown that blogging and the polymath project can play a significant role. However, more then 40 years ago, computational mathematicians created a new way to disseminate their results, namely high quality numerical libraries. It is now a commonplace that a great part of your interaction with physical sciences, engineering, and other fields can be mediated by software you produce and maintain. I will argue that the most effective form of software communication is the library. In fact, the best way to create robust, efficient and scalable, maintainable scientific codes, is to use *libraries*.

Why Libraries? The library organization has advantages over a monolithic application code, although any well-designed application can be sufficiently *library-like* to accomplish these. Libraries hide hardware details from the user. For example, the MPI library hides network details, although the user can specify different harware configurations (shared memory vs. socket connections) on startup. More generally, libraries hide implementation complexity from the user. PETSc has more than 50 matrix formats, many optimized for particular computer acrchitectures, but the user interacts with a single `Mat` interface. As before, the user can obtain control over the implementation by selecting a particular concrete class, such as a block matrix, but will still be shielded from the particular algorithms and data structures needed for that type. Beyond encapsulation, libraries accumulate best practice decisions from experts in the field, another form of algorithm hiding. For example, when orthogonlizing a set

of vectors, PETSc defaults to classical Gram-Schmidt orthogonalization with selective reorthogonalization rather than modified Gram-Schmidt as it is faster in practice on modern architectures. Over time, this can lead to improvements in user code without code changes. The addition of optimized matrix formats and improved time-stepping algorithms to PETSc greatly improved user performance with precisely the same application code as before.

Why is it not just good enough to make a fantastic working code? **Extensibility!** Users need the ability to change your approach to fit their problem. For example, PETSc decided on a block sovler interface that allowed division of a problem into sets (analysis) and combination of solves (synthesis) using additive, multiplicative, or Schur complement means. Since it did not mandate dividing the problem at the top level, it could be used for traditional CFD block sovlers (May and Moresi 2008; Zhong et al. 2015), but also for multigrid with block smoothers (Brown et al. 2012). Since the multigrid interface was extensible, Dave May could design a system to gradually reduce the process set for multigrid levels without changing existing code (May, Sanan, et al. 2016). The structured grid interface (`DMDA`) has enabled an implementation of Isogeometric Analysis (Collier, Dalcin, and Calo 2013; Dalcin et al. 2016).

Early Numerical Libraries Most of the early numerical libraries had their genesis in the Mathematics and Computer Science division (MCS) at Argonne National Laboratory (ANL). In 1971, James H. Wilkinson and Carl Reinch released a set of ALGOL routines as the Handbook for Automatic Computation: Linear Algebra. Wilkinson had a long-standing relationship with ANL, and would often visit for long periods. This collaboration led directly, in 1973, to the EISPACK project led by Brian Smith at ANL. EISPACK was a library of routines written in Fortran to solve the serial eigenproblem for dense matrices. This was followed in 1979, also at ANL, by the BLAS libraries, or Basic Linear Algebra Subroutines. It is one of the foundational libraries for computational science and still widely used today, although it has received a substantial update through the BLIS project Van Zee and van de Geijn 2015. In 1990, a large group of numerical analysts produced the LAPACK library for high level dense linear algebra operations, such as QR factorization and structured updates. And in 1991, again at ANL, Bill Gropp and Barry Smith produced the PETSc library, designed to solve sparse systems of algebraic equation in parallel, usually those arising from the discretization of PDEs.

3.2 Numerical Linear Algebra

3.2.1 Introduction

Numerical linear algebra is one of the most developed parts of numerical analysis. It is also the solid foundation of numerical computing. The basic outline of linear algebra has been clear since at least Grassman's 1862 treatment (Fearnley-Sander 1979). A vector space V over a field F is defined by the axioms in

Axiom	Signification
Associativity of addition	$u + (v + w) = (u + v) + w$
Commutativity of addition	$u + v = v + u$
Vector identity element	$\exists 0 \in V \mid v + 0 = v \quad \forall v \in V$
Vector inverse element	$\forall v \in V, \exists - v \in V \mid v + (-v) = 0$
Distributivity for vector addition	$a(u + v) = au + av$
Distributivity for field addition	$(a + b)v = av + bv$
Scalar and field multiplication	$a(bv) = (ab)v$
Scalar identity element	$1v = v$

Table 3.1: The definition of a vector space (Wikipedia 2015)

Table 3.2.1, and in everything we do F will be either the real or complex numbers. In addition, linear algebra studies mappings between vector spaces that preserve the vector-space structure. Given two vector spaces V and W, a linear operator is a map $A : V \to W$ that is compatible with vector addition and scalar multiplication,

$$A(u + v) = Au + Av, \quad A(av) = aAv \qquad \forall u, v \in V, a \in F. \tag{3.1}$$

This should have been covered in detail in your linear algebra courses.

There are two principal jobs in scientific computing: design of the interface in order to control complexity, and efficiency of the implementation. In this unit we will try to indicate why the current interface has become the standard, and what pieces of it are likely to continue going forward. In a later unit, we will analyze the runtime performance of various implementations. However, none of this can be accomplished without the ability to run a linear algebra code.

3.2.2 PETSc

There are many well-known packages which support numerical linear algebra, including BLAS/LAPACK (Lawson et al. 1979; E. Anderson et al. 1990), Hypre (Falgout 2017; Falgout n.d.), Trilinos (Heroux and Willenbring 2003; Heroux et al. n.d.), DUNE (Bastian et al. 2015), Eigen (Jacob and Guennebaud 2015), and Elemental (Poulson et al. 2013; Poulson 2015). We will use the PETSc libraries (Balay, Abhyankar, Adams, Benson, Brown, Brune, Buschelman, E. Constantinescu, et al. 2022; Balay, Abhyankar, Adams, Benson, Brown, Brune, Buschelman, E. M. Constantinescu, et al. 2022; Balay, W. D. Gropp, et al. 1997) for a number of reasons. PETSc supports scalable, distributed sparse linear algebra, which will be our focus since we will be concerned with larger problems that cannot be contained in a single machine memory and mainly with PDE or graph problems which have a sparse structure. For dense linear algebra problems, we will use Elemental. PETSc is designed as a hierarchical set of library interfaces, and uses C to enhance both portability and language interoperability. A discussion of tradeoffs involved in language choice can be found in (Knepley 2012).

PETSc is intended to be a complete development environment for both algorithms and applications, and should replace the prevalent workflow where a user develops a toy program in Matlab or Mathematica and then completely recodes it for a high performance, parallel environment. The user initially develops a desktop code to test physics, boundary conditions, algorithmic convergence, and analysis, but the same code can be moved to a large parallel platform, and optimized usually using only command line options. Since the algorithms and data structures not hardwired, but rather chosen at runtime, they can be changed to fit the problem and machine architecture without a costly development process. This kind of composable and extensible interface is the key to successful library development. PETSc also incorporates automatic profiling, which can be output to the screen using `-log_view` and as a Python module using `-log_view ascii:log.py:ascii_info_detail`.

We will begin with the simplest PETSc program:

```
static char help[] = "Simple PETSc program";

int main(int argc, char **argv)
{
  PetscErrorCode ierr;

  ierr = PetscInitialize(&argc, &argv, NULL, help);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD, "Hello World!\n");CHKERRQ(ierr);
  ierr = PetscFinalize();
}
```

All PETSc functions return an error code, of type `PetscErrorCode`, which can be checked using the macro `CHKERRQ`. This propagates the error up the call stack, giving the user this information and the ability to recover. Libraries should never print errors directly to the screen or abort suddenly (W. D. Gropp 1999). Hereafter, we will suppress this in code samples, but it should be understood that every call returns an error code and it is checked.

PETSc depends on the Message Passing Interface (MPI) standard (Forum 2012; W. Gropp, Lusk, and Skjellum 1994) for parallelism. However, it allows most users to ignore most of MPI. Each PETSc object has an associated MPI communicator (comm), which we can think of as a *process scope*. The comm determines which processes are involved with that object, and we can also use it to hold data particular to that group of processes such as an outpt object. We begin with the top and bottom of the lattice of possible process sets. `PETSC_COMM_WORLD` is the set of all processes and `PETSC_COMM_SELF` is the process set consisting of only the current process. Other communicators can be formed by set operations such as union and difference. The `PetscInitialize()` and `PetscFinialize()` calls also automatically initialize and shutdown MPI. The `PetscPrintf()` call takes a communicator, and only prints from the first process in the group, eliminating the problem of jumbled output from many cooperating processes.

Function Name	Operation
VecAXPY(Vec y, PetscScalar a, Vec x)	$y = y + a * x$
VecAYPX(Vec y, PetscScalar a, Vec x)	$y = x + a * y$
VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)	$w = y + a * x$
VecScale(Vec x, PetscScalar a)	$x = a * x$
VecCopy(Vec y, Vec x)	$y = x$
VecPointwiseMult(Vec w, Vec x, Vec y)	$w_i = x_i * y_i$
VecMax(Vec x, PetscInt *idx, PetscScalar *r)	$r = \max r_i$
VecShift(Vec x, PetscScalar r)	$x_i = x_i + r$
VecAbs(Vec x)	$x_i = \|x_i\|$
VecNorm(Vec x, NormType type, PetscReal *r)	$r = \|\|x\|\|$

3.2.3 Vectors

A *vector* is a member of a vector space, as defined in Section 3.2.1. They are the fundamental objects representing solutions, right-hand sides, material coefficients, and in general any discrete function in our problems. The vector is defined by its interface, rather than by its data structure, and this is the heart of object-oriented programming. The opposite choice was made by BLAS/LAPACK, making it unable to effectively adapt to changing architectures and problems. Common operations for PETSc vectors are shown in Table 3.2.3.

A vector can be created in the same way as all other PETSc objects. First, a generic creation function, or constructor, is called which gives the object a communicator. This call is *collective* in the sense that all processes in that communicator must call it. Thus, each process is the communicator is attached to this particular object. Next, the object is customized using API calls. In the case of vectors, we must set the size, but for other objects, such as solvers, this can be quite involved. The two sizes used are the *local*, meaning the number of entries stored by this particular process, and *global*, meaning the total number of entries, size for the vector. The global size can be `PETSC_DETERMINE`, in which case it is automatically calculated by PETSc. Alternatively, the local size can be `PETSC_DETERMINE` and PETSc will choose an even distribution for the vector across processes. At least one of the entires, however, must be a number. Lastly, we can allow customization from the command-line by calling the `SetFromOptions()` function for that class. The full sequence is shown below.

```
Vec x;

VecCreate(comm, &x);
VecSetSizes(x, n, N);
VecSetFromOptions(x);
```

Note that the default object implementation type is set by the `SetFromOptions(x)` call, and can be customized using an option like `-vec_type viennacl`. This can be accomplished through the API using `VecSetType(x, VECMPI)`. However, it is rare to create a vector from scratch. Much more often, we are creating a

vector compatible with an input vector, using `VecDuplicate(x, &y)`, or with an input matrix, using `MatCreateVecs(A, &x, &y)`, or as we will see in later lectures compatible with a given discretized domain. When we are finished with an object, it must be destroyed in order to release the resources it allocated,

```
VecDestroy(&x);
```

For the default PETSc vectors, each process locally owns a subvector of contiguous global data, although this is not a requirement. This restriction allows no-copy access to the raw value storage using,

```
Vec x;
PetscScalar *a;
const PetscScalar *ca;

VecGetArrayRead(x, &ca);
/* Use values in calculation */
VecRestoreArrayRead(x, &ca);

VecGetArray(x, &a);
/* Change values in vector */
VecRestoreArray(x, &a);
```

This direct access is also available in F77, F90, Python, and Julia. Implementations which do not require continuity, for example the SAMRAI (Hornung and S. R. Kohn 2002; S. Kohn et al. n.d.) adaptive mesh refinement framework, must copy-in and copy-out to provide this interface.

3.2.4 Matrices

Matrices are finite dimensional linear operators, which satisfy the requirements in (3.1). They are the fundamental objects used for representing stiffness matrices, Jacobians, linear measurements, etc. Just as with vectors, they are defined by their interface rather than storage scheme. However, in the case of matrices, this distinction is much more crucial. While the flat data structure for vectors can perform well almost anywhere, matrices need a variety of data structures to match both the architecture and the problems, such as AIJ, Block AIJ, Symmetric AIJ, Block Matrix, Jagged Diagonal, etc. Matrix creation proceeds just as it did for a vector, except now we must give local and global sizes for both the rows and columns.

```
Mat A;

MatCreate(comm, &A);
MatSetSizes(A, m, n, M, N);
MatSetFromOptions(A);
```

Note that it is very common for each process to locally own a contiguous set of `m` rows, however, this is not a requirement. In the same way as the vector `VecGetArray()` call, this allows the optimization of the `MatGetRow()` call.

Now values can be inserted into the matrix using calls to `MatSetValues()`, which inserts a logically dense block of values. An example is shown below.

```
PetscInt nrows = 2;
PetscInt ncols = 3;
PetscInt rows[2] = {15, 18};
PetscInt cols[3] = {1, 5, 7};
PetscScalar vals[6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

MatSetValues(A, nrows, rows, ncols, cols, vals, ADD_VALUES);
```

This call adds `nrows * ncols` values into the matrix (insertion would use `INSERT_VALUES`), where the values are indexed in a two-dimensional fashion,

	1	5	7
15	1.0	2.0	3.0
18	4.0	5.0	6.0

The user may make as many calls to `MatSetValues()` as required, and then calls

```
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

This will automatically communicate any values not owned by this process to the owning process so that the user need not think about parallel decompositions when creating the matrix. If you wish to mingle calls using `INSERT_VALUES` with `ADD_VALUES`, you must separate the calls using

```
MatAssemblyBegin(A, MAT_FLUSH_ASSEMBLY);
MatAssemblyEnd(A, MAT_FLUSH_ASSEMBLY);
```

The above sequence is enough to create a matrix since PETSc sparse matrices are dynamic data structures that can add additional nonzeros freely. However dynamically adding nonzeros requires additional memory allocations, along with memory copies, which can kill performance. Thus it is very common to optimize matrix creation by telling the it how many entries will be input. This is because memory allocation can be quite expensive, so we would like to minimize the number of `malloc()` calls, and also we used packed data structures for maximum performance, which complicates the job of dynamic allocation. The call below allows the matrix to be allocated with a single call and is applicable also to symmetric storage formats since it can separately specify the upper triangle, although the user may just pass `NULL` if this information is not available.

```
PetscInt bs; /* Matrix block size */
PetscInt *dnz; /* Nonzeros in the diagonal block for each row */
```

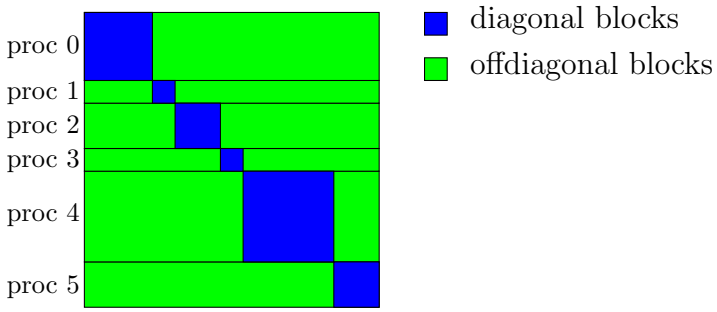


Figure 3.1: The layout of a parallel sparse matrix using AIJ storage.

```
PetscInt *onz; /* Nonzeros not in the diagonal block for each row */
PetscInt *dnzu; /* Like dnz, but only for the upper triangle */
PetscInt *onzu; /* Like onz, but only for the upper triangle */

MatXAIJPreallocation(A, bs, dnz, onz, dnzu, onzu);
```

In Figure 3.1, we show the layout of a parallel sparse matrix, indicating the diagonal and off-diagonal portions. The easiest solution for most codes is to replicate the matrix assembly code, removing the computation but preserving the indexing code and storing the columns for each row in order to get the preallocation information. In almost every case, this time is negligible. Alternatively, this information can be determined from the structure of the problem (mesh and discretization), which is how the PETSc DM object preallocates operators.

After the matrix is created and the values are inserted, we are ready to use it. In Table 3.2, we show common matrix analogous to the vector case. Below is a complete code which will multiply the constant vector by a diagonal matrix,

```
#include <petsc.h>

int main(int argc, char **argv)
{
  Mat A;
  Vec x, y;
  PetscInt N = 10;
  PetscErrorCode ierr;

  PetscInitialize(&argc, &argv, NULL, NULL);
  MatCreate(PETSC_COMM_WORLD, &A);
  MatSetSizes(A, PETSC_DETERMINE, N, PETSC_DETERMINE, N);
  MatSetFromOptions(A);
  MatSetUp(A);
  MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

Function Name	Operation
MatAXPY(Mat A, PetscScalar a, Mat B)	$A = B + a * A$
MatScale(Mat A, PetscScalar a)	$A = a * A$
MatDiagonalScale(Mat, Vec l, Vec r)	$A = \text{diag}(l)\, A\, \text{diag}(r)$
MatCopy(Mat B, Mat A)	$B = A$
MatTranspose(Mat A, MatReuse reuse, Mat *B)	$B = A^T$
MatMult(Mat A, Vec x, Vec y)	$y = Ax$
MatMultTranspose(Mat A, Vec x, Vec y)	$y = A^T x$
MatMatMult(Mat A, Mat B, ..., Mat *C)	$C = AB$
MatNorm(Mat A, NormType type, PetscReal *r)	$r = \|\|A\|\|$

Table 3.2: The PETSc interface for common matrix operations.

```
  MatShift(A, 2.0);
  PetscObjectViewFromOptions((PetscObject) A, NULL, "-A_mat_view");
  MatCreateVecs(A, &x, &y);
  VecSet(x, 1.0);
  PetscObjectViewFromOptions((PetscObject) x, NULL, "-x_vec_view");
  MatMult(A, x, y);
  PetscObjectViewFromOptions((PetscObject) y, NULL, "-y_vec_view");
  VecDestroy(&x);
  VecDestroy(&y);
  MatDestroy(&A);
  PetscFinalize();
  return 0;
}
```

3.3 Correctness and Performance Debugging

The overwhelming amount of time spent developing computational software is spent in design and debugging. That is why you should always be willing to throw away your code, just not your design documents or tests. Even a package as large as PETSc should be reproducible in a few months with good developer documentation and tests. We will focus on design in Section 3.4, whereas this lecture concerns both correctness and performance debugging. Performance debugging is often relegated to an afterthought, but it is crucially important for computational libraries. Existing code should always be profiled prior to changing an algorithm or adding functionality in order to establish a baseline for comparison, and performance differences should be flagged in regression tests in the same way as accuracy differences.

3.3.1 Debugging

Simply the best tool today for memory profiling and debugging is valgrind (Seward 2012). It checks memory access, cache performance, memory usage, and many other things through plugins. Usage of valgrind has virtually eliminated memory errors in PETSc. It also interoperates with MPI, although you will need to run with the `--trace-children=yes` option. Valgrind should be run regularly on every code you develop.

Valgrind, however, introduces enough overhead that it is inconvenient to use on every run. Thus PETSc provides simple utilities for debugging. It generates a stack trace on error or signal, exactly as a debugger would, and can optionally call a user-defined error handler. In addition, when using `PetscMalloc()` and `PetscFree()`, it puts sentinels at the beginning and end of memory allocations to try and detect corruption. The `CHKMEMQ` macro forces a check of all allocated memory. One way to locate memory overwrites without appealing to valgrind is to bracket them with `CHKMEMQ` statements. This interface can also detect memory leaks, and prints any unfreed memory on PetscFinalize() with `-malloc_dump`.

PETSc integrates the debugger so that it can be launched automatically at the start of the run, using `-start_in_debugger`, or when an error occurs, using `-on_error_attach_debugger`. In parallel, a debugging xterm is spawned and connected to each process automatically. A debugging xterm can be connected to only a few processes using `-debugger_nodes 0,1,5` where the argument is a list of process MPI ranks. It is often necessary to set the xterm display in order to have it function in a cluster environment using `-display khan.mcs.anl.gov:0.0`.

3.3.2 Profiling

PETSc has and integrated profiling system which is completely extensible by the user. A comprehensive report can be output during PetscFinalize() with the `-log_view` option. Alternatively, the user may call `PetscLogViewFromOptions()` to output the report at any time. Viewing options in PETSc have a standard argument structure

```
-log_view type:filename:format:filemode
```

where any of the arguments may be omitted. If the `format` is `ascii_info_detail`, then a Python module containing the data will be output. Using the API, the user may call `PetscLogBegin()` initially and then `PetscLogView()` to generate this report at any time.

PETSc profiling is organized into *events* which designate a particular window of execution. All events are created using `PetscLogEventRegister()` so that user-defined events are on equal footing with default PETSc events. The event window is defined by calls to `PetscLogEventBegin()` and `PetscLogEventEnd()`, during which it records elapsed time, number of calls, executed flops, MPI messages, MPI message length, and MPI reductions. Memory usage is currently

associated only with an object, so it does not appear. Flops may be logged manually using `PetscLogFlops()` since there is no portable way to do this automatically. If an event is defined multiple times then these measures are aggregated. Events may be nested and will aggregate in a nested fashion.

The separation into event, however, may be too coarse. For example, suppose I have two linear solves, one of which is much larger than the other (say velocity and pressure). Then the `MatMult` event information averaged between the two would be hopelessly garbled. We would really like to aggregate information for the two solves separately, which is exactly what a `PetscLogStage` does. We can create a new stage using `PetscLogStageRegister()`, and the stage window is defined using `PetscLogStagePush()` and `PetscLogStagePop()`. Event statistics are aggregated within the current stage only. Stages may be nested, but will not aggregate in a nested fashion. Using our hypothetical above as an example, we might have

```
Mat A, C;
Vec vx, vy, px, py;
PetscLogStage stageVel, stagePres;

PetscLogStageRegister(&stageVel, "Velocity Stage");
PetscLogStageRegister(&stagePres, "Pressure Stage");
PetscLogStagePush(stageVel);
/* Carry out velocity solve */
MatMult(A, vx, vy);
PetscLogStagePop();
PetscLogStagePush(stagePres);
/* Carry out pressure solve */
MatMult(C, px, py);
PetscLogStagePop();
```

where now the `MatMult` event will be listed twice in the profiling report, once for each stage.

3.4 PETSc Design

There are two paramount issues for the design of computational libraries: control of complexity and performance. Design of the interface largely controls the complexity, but bad design can prevent performant implementations. For instance, an interface operating on a single element at a time rather than batches can prevent both vectorization and tiling. Below, we will mainly talk about control of complexity as it has received far too little attention in this domain.

I think the notion of hierarchy is essential to managing a large system and reducing the complexity of its component parts. The use of abstraction to understand a group of related concepts as a single entity at a higher level allows us to create simple systems whose individual pieces are nevertheless quite complex. Mathematics operates in just this fashion creating a simpler notions not only

of entities (group, ring, ultrafilter), but also of proof strategies (compactness proof, tensor product trick). While most practitioners try to design algorithms in this way, it is easy to lose sight of this principle when designing the interface.

PETSc explicitly uses a hierarchy of interfaces, internally as well as externally, and encourages users to interact with the library in the same way. For example, a user could interact only with a nonlinear solver, or could extract the linear solver and preconditioner objects, could override domain subsolvers within a decomposition preconditioner, could provide a different matrix action to the solver, or even write a different vector implementation, depending on their focus (engineering design, algorithm development, architectural optimization). In fact, I can extract pieces of a solver and recombine them to form a new higher level object, say a solver for delay differential equations.

Constrast this with the common situation for compilers. The user interacts only through command line options, and has no access to the lower level pieces. I cannot choose to access only the use-definition chain to see what variables are live in a code block I have just written. I cannot easily repurpose parts to the compiler to participate in a small code generation engine I have written to reorder operations at a high level for better vectorization, although the LLVM project is attempting to change this. The monolithic interface for compilers greatly limits their use in library software and the incorporation of compiler techniques.

3.4.1 Language Choice

I have discussed the relative merits of many languages in (Knepley 2012).

C++ The advertised gains from using C++ are largely illusory, and more than outweighed by its significant drawbacks. The structure of object-orientation in C++ necessitates the introduction of a huge number of new types at the interface level, leading to complicated, fragile code with typecasts everywhere. This also brings in the name mangling which sabotages language interoperability. The final insult is the tenfold increase in compile time over C.

Templates do seem to offer genuine advances when dealing with multiple types, especially basic types. However, the mechanism in C++ is clunky. It discards type safety, vastly complicates interoperability with the opaque instantiation mechanism, and the error messages are horrendously long and inpenetrable. Moreover, code reuse very often does not happen beyond basic types since variant methods are needed for other complex types and it functions more like dispatch. Also for complex types, such as meshes, iterators turn out to be a bad design.

Python Python is a flexible language with a good module system, dynamic typing, metaprogramming tools, and an extensive libraries of modules. However, for scientific computing there are still some significant hurdles. Python cannot perform tight loops efficiently in native mode, so scientific computing

almost always requires a form of code generation, provided by packages like Cython. However, this immediately brings in the issue of cross-language debugging. There are some good steps toward this, but it is still very hard. In addition, this introduces indecision about where to place code, in C or in Python, and moving between implementations is still significant work.

Julia Julia is a very promising language. Importantly, they seem to have gotten the foreign function interface right. I would encourage students to experiment with this language. However, the language is very young and does not have the large collection of debugging and profiling tools available for C. This is likely to change over time.

Wish List A small, lightweight system which implemented generics only for basic types, that had smooth conversions at the interface level, and managed multiple versions of the storage seamlessly. Moreover, it would need the same fleixble language interoperability that C enjoys.

3.5 Problems

Problem III.1 Consider the fixed point problem

$$x = Gx \qquad x \in \mathcal{B} \tag{3.2}$$

where $\mathcal{B}$ is some Banach space. It is very common to solve these problems using an iterative method

$$x_{i+1} = \mathcal{M}(x_i, \ldots, x_{i-m}) \tag{3.3}$$

where x_{i+1} is the next approximate solution, $\{x_i, \ldots, x_{i-m}\}$ are previous approximate solutions, and $\mathcal{M}$ is some function defining the method. In (D. G. Anderson 1965), Anderson proposed an iterative method for systems of nonlinear equations, in which the next approximate solution x_{i+1} is chosen to satisfy a minimzation problem involving k prior solutions. However, we will restrict ourselves to the case of *no* prior solutions, or what is called *simple mixing* (Fang and Saad 2009),

$$x_{i+1} = x_i + \beta f_i, \tag{3.4}$$

where f_i is the residual vector at the ith iterate. In ex5 from Problem 2, implement a simple mixing solver using the `SNESSHELL` type in PETSc. Compare the convergence of simple mixing to Newton's method for the default initial guess. Plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the runtime. The y-axis shows the precision of the result, and here we will use the problem size as a proxy for the precision, an acceptable approach for this well-conditioned problem.

Problem III.2 The QR decomposition is a representation of a matrix A in terms of an orthogonal matrix Q and an upper triangular matrix R,

$$A = QR. \tag{3.5}$$

It is described in detail on Wikipedia, and also in many textbooks (Trefethen and Bau, III 1997). Implement the QR decomposition in PETSc for arbitrary matrix dimension using the Gram-Schmdit process. In your code, include a test of the routine for some matrix A which reports $||A - QR||$.

Extra Credit: Implement QR using Householder reflectors or Givens rotations.

Extra Extra Credit: Implement the TSQR Algorithm.

References

May, Dave A. and Louis Moresi (2008). "Preconditioned iterative methods for Stokes flow problems arising in computational geodynamics". In: *Physics of the Earth and Planetary Interiors* 171.1-4. Recent Advances in Computational Geodynamics: Theory, Numerics and Applications, pp. 33–47. ISSN: 0031-9201. DOI: 10.1016/j.pepi.2008.07.036.

Zhong, Shijie, David A. Yuen, Louis N. Moresi, and Matthew G. Knepley (2015). "Numerical Methods for Mantle Convection". In: *Treatise on Geophysics*. Ed. by Gerald Schubert. Second Edition. Vol. 7. Elsevier.

Brown, Jed, Matthew G. Knepley, David A. May, Lois C. McInnes, and Barry F. Smith (2012). "Composable linear solvers for multiphysics". In: *Proceeedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*. IEEE Computer Society, pp. 55–62. DOI: 10.1109/ISPDC.2012.16.

May, Dave A., Patrick Sanan, Karl Rupp, Matthew G. Knepley, and Barry F. Smith (2016). "Extreme-Scale Multigrid Components within PETSc". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. PASC '16. Lausanne, Switzerland: ACM, 5:1–5:12. ISBN: 978-1-4503-4126-4. DOI: 10.1145/2929908.2929913.

Collier, Nathan, Lisandro Dalcin, and Victor M. Calo (2013). "PetIGA: High-Performance Isogeometric Analysis". In: *arxiv* 1305.4452. http://arxiv.org/abs/1305.4452.

Dalcin, Lisandro, Nathan Collier, Philippe Vignal, AMA Côrtes, and Victor M. Calo (2016). "PetIGA: A framework for high-performance isogeometric analysis". In: *Computer Methods in Applied Mechanics and Engineering*.

Van Zee, Field G. and Robert A. van de Geijn (June 2015). "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41.3, 14:1–14:33. DOI: 10.1145/2764454.

Fearnley-Sander, Desmond (1979). "Hermann Grassmann and the Creation of Linear Algebra". In: *The American Mathematical Monthly* 86, pp. 809–817. URL: http://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/DesmondFearnleySander.pdf.

Wikipedia (2015). *Linear Algebra*. https://en.wikipedia.org/wiki/Linear_algebra. URL: https://en.wikipedia.org/wiki/Linear_algebra.

Lawson, C. L., R. J. Hanson, D. Kincaid, and F. T. Krogh (1979). "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software* 5, pp. 308–323.

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (May 1990). *LAPACK: A portable linear algebra library for high-performance computers*. Tech. rep. CS-90-105. Computer Science Dept., University of Tennessee.

Falgout, R. (2017). *hypre Users Manual*. Tech. rep. Revision 2.11.2. Lawrence Livermore National Laboratory.

— (n.d.). *hypre Web page*. http://www.llnl.gov/CASC/hypre.

Heroux, Michael A. and James M. Willenbring (2003). *Trilinos Users Guide*. Tech. rep. SAND2003-2952. Sandia National Laboratories. URL: http://trilinos.sandia.gov/.

Heroux et al., M. (n.d.). *Trilinos Web page*. `http://trilinos.sandia.gov/`.

Bastian, Peter, Markus Blatt, Andreas Dedner, Christian Engwer, Jorrit Fahlke, Christoph Gersbacher, Carsten Gräser, Christoph Grüninger Robert Klöfkorn, Steffen Müthing, Martin Nolte, Mario Ohlberger, and Oliver Sander (2015). *DUNE Web page*. http://www.dune-project.org/. URL: http://www.dune-project.org/.

Jacob, Benoit and Gaël Guennebaud (2015). *Eigen Web page*. http://eigen.tuxfamily.org/. URL: http://eigen.tuxfamily.org/.

Poulson, Jack, Bryan Marker, Jeff R. Hammond, Nichols A. Romero, and Robert van de Geijn (2013). "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Transactions on Mathematical Software* 39.2.

Poulson, Jack (2015). *Elemental: Distributed memory dense linear algebra*. http://libelemental.org. URL: http://libelemental.org/.

Balay, Satish, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, et al. (2022). *PETSc/-TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.18. Argonne National Laboratory.

Balay, Satish, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, et al. (2022). *PETSc Web page*. https://petsc.org/. URL: https://petsc.org/.

Balay, Satish, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith (1997). "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhauser Press, pp. 163–202.

Knepley, Matthew G. (2012). "Programming Languages for Scientific Computing". In: *Encyclopedia of Applied and Computational Mathematics*. Ed. by Björn Engquist. Springer. DOI: 10.1007/978-3-540-70529-1. URL: http://arxiv.org/abs/1209.1711.

Gropp, William D. (1999). "Exploiting Existing Software in Libraries: Successes, Failures, and Reasons Why". In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM, pp. 21–29.

Forum, Message Passing Interface (2012). *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS).

Gropp, William, Ewing Lusk, and Anthony Skjellum (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

Hornung, Richard D. and Scott R. Kohn (2002). "Managing Application Complexity in the SAMRAI Object-Oriented Framework". In: *Concurrency and Computation: Practice and Experience* 14, pp. 347–368.

Kohn, Scott, Xabier Garaiza, Rich Hornung, and Steve Smith (n.d.). "SAMRAI Web Page". See `http://www.llnl.gov/CASC/SAMRAI`.

Seward, Julian (2012). *Valgrind*. http://valgrind.org/. URL: http://valgrind.org/.

Anderson, Donald G (1965). "Iterative procedures for nonlinear integral equations". In: *Journal of the ACM (JACM)* 12.4, pp. 547–560.

Fang, Haw-ren and Yousef Saad (2009). "Two classes of multisecant methods for nonlinear acceleration". In: *Numerical Linear Algebra with Applications* 16.3, pp. 197–221. DOI: 10.1002/nla.617.

Trefethen, Lloyd N. and David Bau, III (1997). *Numerical Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, pp. xii + 361. ISBN: 0-89871-361-7.

Chapter 4

Parallelism

The more any quantitative social indicator is used for social decision-making, the more subject it will be to corruption pressures and the more apt it will be to distort and corrupt the social processes it is intended to monitor.

— Donald T. Campbell

4.1 MPI Basics

The Message Passing Interface (MPI) is an interface standard, specified in several languages, for distributed memory communication, although it has extensions for shared memory. MPI, which originated at Argonne National Laboratory (ANL), is one of the most successful scientific software projects in history. The first implementation of the standard, MPICH, was done by Bill Gropp and Rusty Lusk at ANL. MPI succeeded both because message passing is a very clear conceptual model, and because it had a high quality implementation from day one. There are many excellent references (M. Forum 1994; Message Passing Interface Forum 2012; Snir et al. 1995), tutorials (Gropp, Lusk, and Skjellum 1994; Gropp, Lusk, and Thakur 1999), how-to guides, and other resources (Gropp and et. al. n.d.; Gropp, Lusk, Doss, et al. 1996) dealing with MPI. Thus we will not present a general tutorial, but rather focus on certain aspects which are germane for scientific computation.

MPI has a shared-none memory model, meaning that each process (thread of execution) has an independent memory which it only can access. Processes coordinate with each other by sending messages. All other functionality, such as gathers, scatters, reductions, can be built upon simple message sends and receives, although specialized implementations are common for optimization purposes.

MPI provides great control for the user over every aspect of the communication and synchronization of the operations, which makes optimization possible. However, the important aspects for library writers are the mechanisms for en-

capsulation. Foremost among these is the communicator object. An MPI communicator can be thought of as a scope for parallel operations. It determines the group of processes which particpate in an action and gives them a total order. It can also hold auxiliary information necessary for the implementation. It allows different libraries to coordinate parallelism, as well as send messages guaranteed not to interfere with each other.

An important distinction which arises between MPI operations is the level of process synchronization required. For example, if process p sends a message to process q, only p and q participate, leaving other processes free to compute or communicate as they please. This type of communication is called *point-to-point*, and often abbreviated p2p. On the other hand, if we want the sum of a set of numbers from each process, then all processes must contribute a value and wait for the result. This is an example of *collective* communication, and often happens for reduction operations such as our summation example. However, the barrier operation is also collective, and only requires that every process reach it before any can continue.

4.1.1 Using MPI

In PETSc, each object has an MPI communicator which we will call a *comm*. The comm determines which processes will be involved for any collective operation on the object (these are indicated in the PETSc manpage documentation). For example, suppose we ask for the dot product of two parallel vectors. Even though most of the computation may be performed concurrently by a set of processes, they all must agree on the scalar answer at the end, which makes the result collective. If the user creates two `SNES` nonlinear solver objects using communicators that have disjoint process sets, they can execute two solves concurrently.

MPI uses its own set of data types in order to assure uniformity across languages and operating systems. PETSc provides the `MPIU_INT` and `MPIU_SCALAR` MPI datatypes to match the `PetscInt` and `PetscScalar` C data types, as well as the `PetscMPIInt` C datatype to match `MPI_INT`. Moreover, it provides conversion functions `PetscDataTypeToMPIDataType` and `PetscMPIDataTypeToPetscDataType` to convert between the PETSc binary specification and MPI.

Unfortunately, most large systems that you run on will have a complicated batch submission system. However, when running on your laptop or a friendly local cluster, you can use the `mpiexec` tool to launch parallel jobs. For example, to test PETSc, we could use

```
cd ${PETSC_DIR}/src/snes/examples/tutorials
make ex5
mpiexec -n 2 ./ex5
mpiexec -n 5 ./ex5
mpiexec -n 20 ./ex5
```

to run an example on 2, 5, and 20 processes. There is often an option to manage the memory affinity, such as

```
mpiexec -n 20 -bind-to socket ./ex5
```

FoM	Unit	Rate	Unit
Time	s		
Flops	F	Flop Rate	F/s
Memory	B	Memory Bandwidth	B/s
Energy	J	Power	W
Concurrency	—		

Table 4.1: Figures of merit which can be measured for normal code execution.

on MPICH, or `--bind-to-socket` on OpenMPI.

4.2 Computational Scaling[1]

Computational scaling is the analysis of algorithmic performance as the run environment is varied. This is merely another kind of engineering design. Just as a chemical or process engineer might ask how a test plant reaction will scale up to a production facility, or what step in the process is rate-limiting, we will ask the same kinds of questions of our software. To begin, we need to understand what figures of merit (FoM) are available for our code. We list some FoM which can be measured in Table 4.1.

Concurrency is particularly important recently with the end of Dennard Scaling (Dennard et al. 1974). Dennard noted that chip performance per Watt was scaling at roughly the same rate as Moore's Law for chip integration, and this increase was based on the relation that power (W) scales as capacitance (C) times frequency (ω) times voltage (V) squared

$$W \propto C\omega V^2. \tag{4.1}$$

Thus as the capacitance and voltage declined with chip area, the frequency could be increased at constant power. The problem has become that this simple model ignores both threshold voltage, that some minimum voltage is necessary for switching, and leakage current. The resulting "power wall" has limited clock frequencies to about 4 GHz since 2006. Therefore further increases in performance need to come from other places, most notably more concurrent computing cores. Thus we are being driven to produce algorithms that can handle much more concurrency than before.

Speedup measures the ratio of performance between two executions. These runs could compare implementations in two different languages, such as C and Python, or two different algorithms, such as bubble and quicksort, or different instruction optimizations, such as a scalar and vectorized version. However, we will be concerned with the same problem solved on different numbers of processors. We can run exactly the same algorithm on different process sets to measure the scalability of the implementation. However, in order to look at the

[1] I would like the ackowledge the debt to William Gropp's slides from his CS598

gain realized by a user, we should compare the best algorithm/implementation for each process set. This illustrates the problem of choosing the baseline which we compare against. This can be tricky in the parallel case since we must consider algorithms, implementations, and decomposition strategies.

4.2.1 Strong Scaling

We will define the parallel speedup S_p to be

$$S_p = \frac{T_1}{T_p} \tag{4.2}$$

where T_p is the time taken on p processes. Keeping the total problem size fixed while we increase the number of processes is known as *strong scaling*. Clearly $S_p \geq 1$ since we can just ignore extra processes. However, can $S_p > p$? This can indeed happen if the performance on a given architecture varys with the size of the workload. For example, suppose we have a problem which is memory bound, meaning the computational bottleneck is loading data from memory. If it loads M words from memory many times, and M/p fits into cache for some p, the time to access memory will be different in the two cases since T_1 uses the STREAM main memory bandwidth and T_p uses the appropriate cache bandwidth. This raises the question, is there an upper limit on speedup?

Amdahl's Law

Amdahl's Law (Wikipedia 2015) was formulated by Gene Amdahl in 1967, and analyzes the speedup for a program having a fixed serial fraction, meaning a part of the program that will never improve. Let us define the time T_p it takes for the entire algorithm on p processes,

$$T_p = fT_1 + (1-f)\frac{T_1}{s_p} \tag{4.3}$$

where $f \in [0, 1)$ is the serial fraction of the computation, and s_p is the speedup for the parallel portion of the algorithm. The speedup is then given by

$$S_p = \frac{T_1}{T_p} \tag{4.4}$$

$$= \frac{T_1}{fT_1 + (1-f)\frac{T_1}{s_p}} \tag{4.5}$$

$$= \frac{1}{f + (1-f)s_p^{-1}}. \tag{4.6}$$

In the limit of infinite speedup for the parallel portion, $s_p \to \infty$, the total speedup is limited to

$$S_\infty = \frac{1}{f}. \tag{4.7}$$

For example, suppose that a program has 1% overhead that cannot be sped up by using more processes. Then the total speedup on the largest computer is limited to

$$S_\infty = \frac{1}{\frac{1}{100}} = 100. \tag{4.8}$$

This appears to be a very stringent bound on the possible speedup. Parallel workloads can be made quite large, but this tends to exceed the capacity of one node. We will look at an alternative analysis in the next section.

We can also look at the approach to the asymptotic limit to get an idea how fast the required resources increase as we demand more speedup. For example, at what value of p is half the asymptotic speedup realized? We can solve for this p,

$$S_p = \frac{1}{2} S_\infty \tag{4.9}$$

$$\frac{1}{f + (1-f)s_p^{-1}} = \frac{1}{2f} \tag{4.10}$$

$$f + (1-f)s_p^{-1} = 2f \tag{4.11}$$

$$s_p = \frac{1-f}{f}. \tag{4.12}$$

Thus, for our example above with $f = 0.01$ assuming perfect speedup $s_p = p$, it would take 99 processes to achieve a speedup of 50, so the returns on additonal processes rapidly diminish. Another way to see this is to look at the derivative of speedup with respect to the number of processes,

$$\frac{\partial S_p}{\partial p} = \frac{\partial}{\partial p} \frac{1}{f + (1-f)\frac{1}{p}} \tag{4.13}$$

$$= \frac{\partial}{\partial p} \frac{p}{pf + (1-f)} \tag{4.14}$$

$$= \frac{1}{pf + (1-f)} - \frac{pf}{(pf + (1-f))^2} \tag{4.15}$$

$$= \frac{1-f}{(pf + (1-f))^2} \tag{4.16}$$

which as $p \to \infty$ can be approximated by

$$\frac{1-f}{f^2 p^2} \tag{4.17}$$

which shows the swift decline is marginal efficacy of more processes. The log derivative gives the relative efficiency,

$$\frac{1}{S_p}\frac{\partial S_p}{\partial p} = \frac{pf + (1-f)}{p} \frac{1-f}{(pf + (1-f))^2} \tag{4.18}$$

$$= \frac{1-f}{p^2 f + p(1-f)} \tag{4.19}$$

so at $f = 0.01$, it is $\frac{99}{p^2+99p}$.

4.2.2 Weak Scaling

We can conceive of problems too big to fit on a single process, so what should we do in this case? We could change the definition of speedup to start at k processes instead of 1. We could also take this idea farther and ask what happens if we continue to increase the problem size each time we increase the number of processes. Suppose that the parallel work size on each process was fixed, so that the total parallel work increased linearly with the number of processes. Thus a perfect algorithm would keep the time fixed, rather than decreasing the time linearly with p as we expect in strong scaling arguments. Since the work per process is kept fixed, we will model the time on p processes as the fixed work, along with an overhead that increases more slowly than p and which we will take to be proportional to the local work,

$$T_p = T_1 + o(p)T_1. \tag{4.20}$$

We can define a speedup as the ideal time for the largest problem on one process divided by the time on p processes,

$$S_p = \frac{pT_1}{T_p}, \tag{4.21}$$

but it is perhaps more appropriate to focus on the *efficiency*

$$E_p = \frac{T_1}{T_p} \equiv \frac{S_p}{p}. \tag{4.22}$$

This tells the implementor whether the problem decomposition assumption is true and the overheads have been controlled. From our previous definitions,

$$E_p = \frac{T_1}{T_1 + o(p)T_1} \tag{4.23}$$

$$= \frac{1}{1 + o(p)}. \tag{4.24}$$

We can see that the efficiency approaches zero in the limit of large p unless the overhead per process is a fixed constant α, which is the usual assumption for analysis,

$$S_p = p\frac{1}{1+\alpha} = p(1-\alpha'). \tag{4.25}$$

In this case the efficiency asymptotically approaches $1 - \alpha'$ instead of decaying rapidly to zero as in the case of Amdahl's Law. This phenomenon is known as *weak scaling*.

The appeal to weak scaling rests on the somewhat dubious physical assumption that more resolution is always desirable. While there is definitely a regime

in most problems where this is true, it is not at all clear that scientific insight or engineering practice benefits from the increased resolution on many problems run on large machines today. Moreover, the analysis assumes an $\mathcal{O}(N)$ algorithm for the workload, without which the time cannot keep pace with the increase in size. For example, an $\mathcal{O}(N^3)$ algorithm with twice the concurrency can only solve a problem 26% bigger on constant time.

A more serious flaw in the application of weak scaling is that it discounts the most important mechanism for data compression and scalability of solvers, which is modeling. The idea is not to take a single, basic model and scale it to the ends of the earth, but rather to apply a given model in its optimal domain of applicability, at which point we transition to a higher level of abstraction and another model. Thus, the weak scaling analysis is very useful within a limited domain, and perhaps in modern HPC has been asked to do more than it is really capable.

4.2.3 Machine Scaling

Suppose that instead of starting with a fixed problem, we start with a fixed parallel machine. We would like to understand how this machine handles different workloads. Suppose that I run a range of problem sizes on this machine and then plot the time to solution (T) against the rate of solution (N/T), which we will call the performance spectrum. If the machine is performing optimally, then it will perform all these different computations at the same rate, and our spectrum will be a flat horizontal line. However, as the problem on each computing element (core) becomes small, overheads tend to overwhelm the computation so that the rate decreases and the total time stalls, leading to a tailing off of the line at the left edge. This turning point is the lower bound on turnaround time for a job on this machine. Note that since the parallelism is constant, machine latencies will be constant, so all latency effects will come from algorithmic scaling. If the algorithm has suboptimal scaling, meaning cost that is not in $\mathcal{O}(N)$, then we will see a tailing off on the right edge of the spectrum as well. A stepwise degradation on the right could also indicate NUMA performance issues, such as dropping out of cache for a larger problem size.

4.2.4 Reliability of Scaling Measures

In the 1970s, social scientist Donald Campbell wrote that any metric of quality can become corrupted if people start prioritizing the metric itself over the traits it supposedly reflects (Campbell 1976). According to subsequent mathematical models (Smaldino and McElreath 2016), his argument works even if individuals aren't trying to maximize their metrics, but rather just following broad incentives. This is also true of algorithmic scaling measurements.

The guide lines for "perfect" scaling above are all relative to a base case. If the base case is inefficient, the scaling looks better. This makes it trivial to show near-perfect scaling despite having a poor algorithm or implementation – the scaling gets more perfect as the algorithm gets worse. Demonstrating

that the base case is efficient is not easy, rarely actually the case, and often overlooked by authors and readers alike. Thus, scaling plots need a great deal of context and careful reading of scales to be meaningful. This is perhaps the most critical thing to recognize about the classical strong and weak scaling plots. Even machine scaling plots can be made to look flat by the introduction of useless work, but the subsequent minimum turnaround time will increase, and thus this plot makes gaming easier to detect.

4.3 Scaling on Heterogeneous Machines

Heterogeneous parallelism has become the norm for high-end parallel machines, and new developments at the top end almost invariably trickle down to the consumer market after a decade or so (witness the rise of the multicore laptop). The current world record holder, Tianhe-2 located in China's National Supercomputer Center in Guangzhou, has 32,000 Intel Ivy Bridge Xeon processors, but also 48,000 Xeon Phi 31S1P co-processors, meaning each node has 2 Xeons and 3 Phis. The next-generation Department of Energy (DOE) supercomputer Cori, which recently entered production at Lawrence Berkeley National Laboratory, has about 2,000 Intel Haswell processors nodes (with two sockets each) and 10,000 Intel Knights Landing nodes (with one processor node). In addition, most high-end laptops now include a Graphics Processing Unit (GPU) capable of handling general purpose computing.

With this change in hardware organization, we must re-examine the decision to use MPI as the basis for scientific computing libraries. Is it possible that a heterogeneous software stack would better match the heterogeneous computing platform? We can ask some initial questions about our application in order to guide our choices:

- What are the per-process memory requirements as a function of problem size and scale?
  If memory is a limitation, is that because there is a fundamental requirement for this much memory per process or because data is replicated unnecessarily?

- Is on-node communication currently limiting the performance and scalability?
  For example, is MPI communication on high core count Intel Xeon processors, such as 2×18-core Haswell, a limitation?

If memory per process or intranode communication are not fundamentally limiting factors in the application, the canonical flat MPI paradign will continue to scale nicely on today's largest heterogeneous machines.

4.3.1 MPI+OpenMP

You could imagine two endpoints in the design space for a parallel application. One is a "provisioned" model, like the typical MPI launcher, where a number

of threads of execution are started at the beginning on the run and continue until the end, being given work as necessary. The other is an "on-demand" or fork-join model in which threads of execution are created as needed for parallel sections of a code and then vanish when that section ends, which is the more common OpenMP model. OpenMP can certainly be run using a provisioned or thread pool, however the differences between it and MPI then recede.

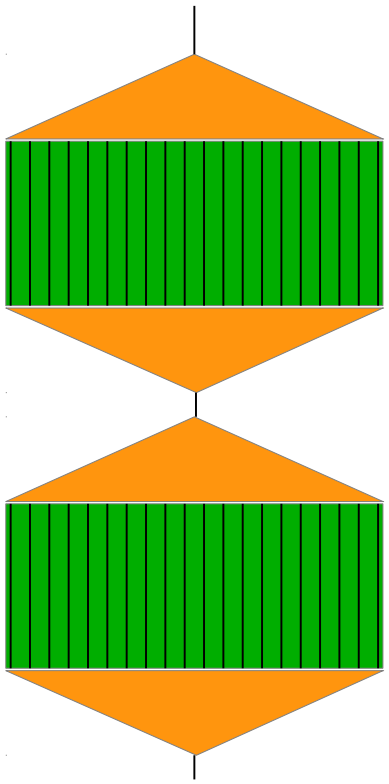
In Fig. 4.3.1, we show the work pattern graphically for the fork-join and provisioned approaches, as well as the typical code organization. The overhead of a fork-join in OpenMP can range from microseconds to a millisecond, especially as the thread count grows (Wang and Parmer 2014). Thus in order to realize significant speedup, we will need long lived workloads. This is a well-known phenomenon, and most OpenMP optimization guides advocate a provisioned approach. This is then very close to the MPI provisioned model.

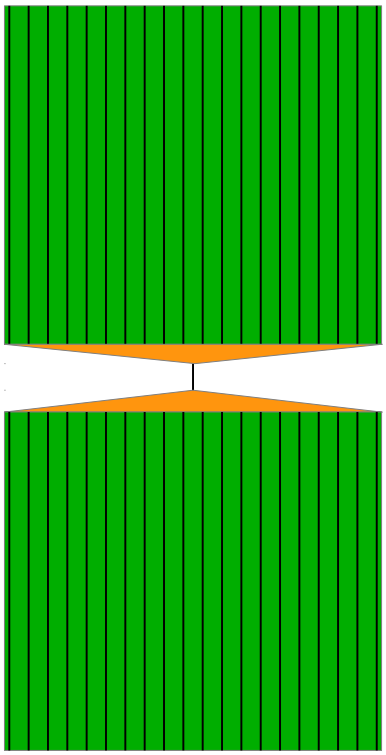
In addition, the interaction of the MPI runtime with threads introduces either serialization (using `MPI_THREAD_FUNNELED`) or large overheads (using `MPI_THREAD_MULTIPLE`). The MPI-4 release, however, is slated to add a feature called *endpoints*, an object which can participate in communication. This construct allows communication to be associated with a thread rather than just with a process. There is still problem for library design here in that a user would typically be calling the library from within an OpenMP parallel region, and more importantly, library callbacks, such as those for residual and Jacobian evaluation in PETSc, would return within the same OpenMP parallel region, but few applications are structured in this way. Instead they embed OpenMP parallel directives inside their residual and Jacobian functions, which is incompatible with the provisioned model, so that it is impossible to maintain the interface semantics without large penalities for synchronization. The application could be rewritten to use a shared-nothing model, but that is most of the way back toward a pure MPI model. Note that this also exposes the difficulty of developing robust libraries using OpenMP in that there is too much implicit or assumed about the relationship between threads and scope of operations. Moreover, OpenMP provides none of the encapsulation methods which are crucial to good library design, making interoperating with other packages using OpenMP or threads very painful. The fine-grained control over the synchronization and buffering behavior in MPI is also absent. A thorough case against the use of threads is made in The Problem with Threads by Edward A. Lee of UC Berkeley EECS.

MPI, however, can suffer when the shared-none memory model incurs overhead. We can model this simply in 1D using an N node torus with a single dof on each node. If we divide this into P partitions, and duplicate the unknowns on each interface for the local spaces, the memory used by all the local spaces is

$$\left(\frac{N}{P} + 2\right) P = N + 2P \tag{4.26}$$

If we imagine that the total memory is fixed, then in the strong scaling limit, the memory overhead of a shared-none partition may exceed the total memory





```
/* thread-unsafe */
#pragma omp parallel for
{
  /* threaded loops */
}

/* thread-unsafe */

#pragma omp parallel for
{
  /* threaded loops */
}
/* thread-unsafe work */
```

```
#pragma omp parallel
{
  /* thread-safe */

  #pragma omp single
  /* thread-unsafe */

  #pragma omp for
  /* threaded loops */

  #pragma omp sections
  /* threaded work */
}
```

Figure 4.1: This figure illustrates two different methods of using OpenMP for parallelization, Fork-Join and Parallelize-Serialize. On the left, each loop is individually parallelized using the pragma for parallel for, whereas on the right threads are spawned at the start of the program, and critical section are protected. The paradigm on the right avoids the overhead of repeated fork-join. Image taken from Hammond.

capacity. This effect is exacerbated by the increase in the surface to volume ratio as subdomain size shrinks, and also with higher order methods which have many unknowns on the surface. If we instead assume that the amount of memory per node is fixed (weak memory scaling), we have similar limitations on node-level parallelism. If the memory per process is fixed, meaning also that we do not over-decompose the problem, then we have a limit on subdomain size which may impact preconditioning. In any case, the shared-none memory model could become a bottleneck. The MPI-3 standard incorporates shared memory regions, allowing processes on the same node access to the same memory much like threads, without the huge cognitive overhead of non-determinism in the entire program (Lee 2006).

If an application is limited by on-node MPI communication, it can now appeal to the MPI-3 neighborhood collectives mechanism. This API allows the user to provide a communication topology directly to the library, enabling many optimizations (Hoefler and Schneider 2012). These are available in both scatter/gather and all-to-all versions within the process set specified.

4.3.2 MPI+CUDA/OpenCL

CUDA, or its standardization OpenCL, look superficially like an on-demand model for parallelism. However it incurs the same overheads as OpenMP. Kernel launch time is the primary bottleneck for computations with small workload, and thus larger kernel computations are recommended. In addition, the latency penalty for access to global shared memory is very high (600+ cycles) so that optimized kernels perform a single, coalesced load into local memory and operate in a shared-none fashion. The global memory can then be thought of as a message medium, similar to an interconnect, and we have essentially the MPI model again, albeit with a dynamic process pool. However, the kernel launch prescribes the number of thread blocks up front, just as we do in an MPI launch.

4.3.3 Verdict

It does not appear that for most workloads, and especially for weak scaling, libraries will have to abandon the current MPI infrastructure. However, for applications like climate simulations which have stringent turn-around time demands and operate in the strong scaling limit, a hybrid model might still offer some, yet to be conclusively demonstrated, relief. However, forthcoming features of MPI, shared memory, neighborhood collectives, and endpoints, will reduce or eliminate these advantages. The core PETSc team has come to the consensus that pure MPI using neighborhood collectives and the judicious using of MPI shared memory, for data structures that you may not wish to have duplicated on each MPI process due to memory constraints, will provide the best performance for HPC simulation needs on current generation systems, next generation systems and exascale systems. It is also a much simpler programming model then MPI + Threads, leading to simpler, more maintainable code.

References

Forum, MPI (1994). "MPI: A Message-Passing Interface Standard". In: *International J. Supercomputing Applications* 8.3/4.

Forum, Message Passing Interface (2012). *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS).

Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra (1995). *MPI: The Complete Reference*. MIT Press.

Gropp, William, Ewing Lusk, and Anthony Skjellum (1994). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.

Gropp, William, Ewing Lusk, and Rajeev Thakur (1999). *Using MPI 2: Advanced Features of the Message Passing Interface*. MIT Press.

Gropp, William and et. al. (n.d.). *MPICH Web page*. http://www.mpich.org. URL: http://www.mpich.org.

Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum (1996). "A high-performance, portable implementation of the MPI Message Passing Interface standard". In: *Parallel Computing* 22, pp. 789–828.

Dennard, Robert H, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc (1974). "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5, pp. 256–268.

Wikipedia (2015). *Amdahl's Law*. https://en.wikipedia.org/wiki/Amdahl's_law. URL: https://en.wikipedia.org/wiki/Amdahl's_law.

Campbell, Donald T. (1976). "Assessing the Impact of Planned Social Change". In: *Occasional Paper Series, The Public Affairs Center* 8.

Smaldino, Paul E. and Richard McElreath (2016). "The natural selection of bad science". In: *Royal Society Open Science* 3.9. DOI: 10.1098/rsos.160384. eprint: http://rsos.royalsocietypublishing.org/content/3/9/160384.full.pdf. URL: http://rsos.royalsocietypublishing.org/content/3/9/160384.

Wang, Qi and Gabriel Parmer (2014). "Fjos: Practical, predictable, and efficient system support for fork/join parallelism". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, pp. 25–36.

Lee, Edward A (2006). "The problem with threads". In: *Computer* 39.5, pp. 33–42. URL: http://ptolemy.eecs.berkeley.edu/publications/papers/06/problemwithThreads/.

Hoefler, Torsten and Timo Schneider (2012). "Optimization principles for collective neighborhood communications". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. ISSN: 21674329. DOI: 10.1109/SC.2012.86.

Chapter 5

Data Layout and Discretization I

5.1 Sparse Data

Sparsity, the fact that only a few important factors determine how a system behaves, is what makes the universe understandable. If the influence of every part mattered equally, we would have no way of doing practical calculations or making predictions. When we discuss PDEs, sparsity in the operators comes from the fact that physics in pieces of the domain influences only immediately adjacent pieces through the differential operator. In integral equations, all parts of the domain are coupled, but the information (strength) contained in distant couplings is far less than close couplings.

If we knew what variables were actually important, we could always write small dense problems in this basis. However, it is difficult or impossible to pick this set for most problems, it could change over the course of the solve, and its not easy to relate to experiments or experience. Thus, we often operate in a situation where we have a large basis set describing a physical configuration, but only a very few variables are coupled to one another. We encode this situation using sparse matrices and sparse grid structures. Below, we will discuss strategies for data representation and manipulation.

A more subtle difficulty is that, phrased in a reduced basis, the problem may be much more ill-conditioned then the original sparse problem, mandating the use of a direct solver. Due to the disparity in complexity, it is possible that an optimal iterative method for the original sparse problem can outperform a dense method for the reduced problem. The same kind of conditioning problem can also arise when transforming a non-orthogonal basis to a smaller orthogonal set. Thus, sparse methods are likely to be practical on a subset of problems for a long time to come.

5.1.1 Sparse Matrices

A matrix is a set of numbers $\{a_{ij}\}$ which represents the action of a linear operator in certain bases. For example, we have an operator $\mathcal{A} : X \to Y$, and I wish to know that action $y = \mathcal{A}x$ where $x \in X$ and $y \in Y$. Now suppose I have orthonormal bases $\{\phi_j\}$ and $\{\psi_i\}$ for X and Y respectively. Then I can rewrite the action above as

$$y = \mathcal{A}x \tag{5.1}$$
$$\sum_k y_k \psi_k = \mathcal{A} \sum_j x_j \phi_j \tag{5.2}$$
$$\psi_i^* \sum_k y_k \psi_k = \psi_i^* \mathcal{A} \sum_j x_j \phi_j \tag{5.3}$$
$$\sum_k \delta_{ik} y_k = \sum_j \psi_i^* \mathcal{A} \phi_j x_j \tag{5.4}$$
$$y_i = \sum_j a_{ij} x_j \tag{5.5}$$
$$y = Ax \tag{5.6}$$
$$\tag{5.7}$$

where

$$a_{ij} = \psi_i^* \mathcal{A} \phi_j. \tag{5.8}$$

Thus we have expressed our linear operator as a matrix, and our functions as vectors. We can use this prescription to translate all our linear algebra problems to matrix language.

Often we know that only some of the entries in A are nonzero. For example, in finite element discretizations, only basis functions from adjacent elements are coupled by differential operators, and similarly for finite volume discretizations. In this case, representing A with a dense array of coefficients is very wasteful, and can greatly increase the asymptotic complexity of our algorithm. In the case of finite elements, the dense matrix can be applied in $\mathcal{O}(N^2)$ for a basis of size N, whereas the sparse matrix can be applied in $\mathcal{O}(N)$ time.

There is an excellent discussion of sparse matrix formats in (Saad 2003), and the older edition (Saad 1996) is freely available online. We will just discuss the main techniques in order to prepare for performance analysis and generalization of this strategy. If there are very few entries, say $\mathcal{O}(1)$, then we can just represent their coordinates as a set of triples $\{(i, j, a_{ij})\}$, and this is refered to as Coordinate Format (COO). However, if we have more entries, such as $\mathcal{O}(N)$ in a finite element matrix, we can imagine that many entries have repeated indices. For example, all entries in same row have i the same, and similarly for columns.

We can apply a common compression technique, run length encoding (RLE), to the sequence $\{(i, j, a_{ij})\}$. Now we just keep track of the number of items for each row $\{(i, \#i)\}$, along with the original list of columns and values $\{(j, a_{ij})\}$.

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix} \qquad \begin{aligned} i &= \begin{pmatrix} 0 & 2 & 5 & 9 & 11 & 12 \end{pmatrix} \\ j &= \begin{pmatrix} 0 & 3 & 0 & 1 & 3 & 0 & 2 & 3 & 4 & 2 & 3 & 4 \end{pmatrix} \\ a &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix} \end{aligned}$$

Figure 5.1: A sparse matrix encoded in AIJ format.

If we store the prior data in a dense array, the row i is implicitly given by the array index. Thus, the AIJ sparse matrix format has three arrays: row offsets, columns, and values. An example is shown in Figure 5.1.1.

Lets look at the process of sparse matrix multiplication, which is actually quite simple. We loop over each sparse row, get the relevant rows from the vector we are acting on, and do a sparse dot product. In C code, this is

```
for (i = 0; i < m; ++i) {
  n = roffset[i+1] - roffset[i];
  c = cols[roffset[i]];
  v = vals[roffset[i]];
  sum = 0.0;
  PetscSparseDensePlusDot(sum, x, v, c, n);
  y[i] = sum;
}
```

where `x` is the input array, `y` is the output array, and `PetscSparseDensePlusDot()` is the operation

$$\text{sum} = \sum_{k=1}^{n} \text{v}[k] * \text{x}[\text{c}[k]]. \tag{5.9}$$

This can be expanded in a straightfoward way

```
PetscInt __i;
for (__i = 0; __i < n; ++__i) sum += v[__i] * x[c[__i]];}
```

or with loop unrolling to try and promote vectorization

```
if (n > 0) {
  switch (n & 0x3) {
  case 3: sum += *v++ *x[*c++];
  case 2: sum += *v++ *x[*c++];
  case 1: sum += *v++ *x[*c++];
  n -= 4;
  }
  while (n > 0) {
    sum += v[0] * x[c[0]] + v[1] * x[c[1]] +
           v[2] * x[c[2]] + v[3] * x[c[3]];
```

```
    v += 4; c += 4; n —= 4;
  }
}
```

Almost all of the changes to the AIJ format are made to try and vectorize code for different architectures, since this is just about the only way to increase performance. However, this only matters if you are currently running slower than the speed that values can be fetched from memory, the *bandwidth bound*, as we will see in Chapter 9.4. The other common optimizations promote good cache reuse. If there are rows with the same index structure, say for a block matrix, we would like to keep those indices in cache and not have them evicted by the vector entries streaming by. In addition, there is benefit to prefetching the indices and values for the next block row. This is the kind of simple, semi-quantitative thinking we want to employ when initially judging algorithms, and it should lead to the construction of illustrative performance models.

5.2 General Data Layout

A sparse matrix assigns some data (columns and values) to each basis vector (row). Suppose we want to generalize this to a scheme which can assign data to a more general entity, such as finite element basis coefficients to edges, or cell averages to cells and field fluxes to faces, or finite difference coefficients to vertices. How would we design such as thing? In PETSc, a general data layout is encoded in the `PetscSection` class, which is mainly a map from an index space which we call *points* to space of (size, offset) pairs, mirroring the AIJ format. This structure is simple, and yet flexible enough to encoded a huge variety of data distributions.

For example, the parallel data layout of a vector can be represented by choosing the point space as the set of process ranks, and the data space as (local size, global start) pairs. It can represent a finite element data layout over an unstructured mesh by choosing the point space to be a numbering of each mesh piece (vertices, edges, faces, cells) and the data space to be (dofs, offset) pairs. Simmilarly, for finite volume discretizations, this same scheme works, although the point numbering is likely to include only faces and cells.

This kind of arrangement can also be seen as the discrete analogue of the mathematical fiber bundle. A fiber bundle makes precise the idea of one topological space, called a fiber, being "parameterized" by another topological space, called a base. In our case, the point space is the base, and data size indicates the dimension of the fiber space (and also the storage for a representative element). We will want to be slightly more general in order to accommodate mixed discretizations and multicomponent problems. If we allow the fiber spaces to be different, but related to a larger total space, then we have a fibration. A `PetscSection` can have different sizes for each point, and at a given point, we allow the size to be split up between some number N_F of *fields*. Thus the data tuple would now be given by (size_0, ..., size_{N_F-1}, offset), since the field offsets

Discretization	Dof/Dimension	Discretization	Dof/Dimension
P_0	[0 0 0 1]	Q_0	[0 0 0 1]
P_1	[1 0 0 0]	Q_1	[1 0 0 0]
P_2	[1 1 0 0]	Q_2	[1 1 1 1]
P_3	[1 2 1 0]	Q_3	[1 2 4 8]
P_1^{disc}	[0 0 0 4]	RT_0	[0 0 3 0]

Table 5.1: Storage specification for familiar scalar Galerkin finite elements.

can be inferred from the sizes and initial offset.

For the case of conforming finite elements, we can simplify the specification since we have strong symmetry requirements for the `PetscSection` structure. In order to define a continuous interpolant, we must have the same number of unknowns on each kind of mesh point (vertex, edge, face, cell). Thus, it is enough to specify the number of dofs for each dimension. Example for familiar elements are shown in Table 5.1. From these specification, it is easy to construct the `PetscSection` by looping over each mesh point and adding the appropriate number of dofs.

5.2.1 Boundary Conditions

When we discuss the computational aspect of applying boundary conditions, we will divide them into two classes, somewhat similar to the elementary division into Dirichlet and Neumann conditions. We will call *essential* boundary conditions those which alter the space of function that we will use to approximate the solution, of which Dirichlet is one type. We will call *natural* boundary conditions those which augment or change the equations we use to define the solution, such as Neumann conditions, but also Robin conditions, transmission conditions, and Nitsche conditions. In simpler terms, I can either fix the solution value on the boundary or write an equation for it.

Why do Neumann conditions fall into the second category if I am simply fixing the value of the solution derivative on the boundary? This is because the derivative is not a local concept, but depends on solution values in a (small) neighborhood, and thus the boundary value depends on solution values in the interior, just like all the other types of natural conditions, and therefore I can write an equation that expresses this dependence.

There are two common ways of imposing essential boundary conditions: remove the variables from the global system, or modify the global system to impose the boundary values. For the later method, we would need to change the residual calculation so that it only computes the distance from the boundary value, $u - u_\Gamma$, but also any linearized update equations so that they fix these boundary values, for which the `MatZeroRowsColumns()` function is often used. This has been the favored method since people can easily see where the condition is imposed, but it has some drawbacks. Very often users create asymmetry in the equations, but even if they are careful to preserve symmetry, scaling of

the boundary unknowns can be an issue. It also imposes an extra step outside of the normal assembly operations, making the code more complex and harder to optimize at a low level.

An alternative way to handle boundary conditions is to remove these unknowns from the global problem. But how will the effect of the boundary values be incorporated into the residual and update rhs? The constrained unknowns will actually be present in a *local* space which we will use to compute updates to the residual, Jacobian, and rhs. These updates will be assembled in the *global* space, which is the one we are most familiar with and in which the solver operates. The local space will contain boundary values, as well as all ghost values that arise from parallelism or overlapping discretization methods. The global space will consist only of the independent dofs which are being solved for. We will show how this works in detail for a very simple problem in order to elucidate the difference in strategies.

Suppose that we have a triangular domain discretized using a single P_1 element, so that there are unknowns associated with each vertex, and we solve the Laplace equation on this domain,

$$-\Delta u = 0 \qquad \text{on } \Omega \tag{5.10}$$

$$u = 5 \qquad \text{on } \partial\Omega_D \tag{5.11}$$

where the boundary $\partial\Omega_D$ consists of only the first vertex. Since we constrain the value at one vertex, our global problem will have only two unknowns, but when we assemble in the local space all three unknowns will be present. When we assemble the residual for an initial guess of zero with exact boundary conditions, we have

$$\vec{f}(\vec{u}) = \begin{pmatrix} 0.5 & 0 & -0.5 \\ 0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{pmatrix} \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix} \tag{5.12}$$

$$= \begin{pmatrix} 2.5 \\ 0 \\ -2.5 \end{pmatrix} \tag{5.13}$$

which is mapped to the global residual (since we ignore the constrained vertex),

$$\vec{F}(\vec{U}) = \begin{pmatrix} 0 \\ -2.5 \end{pmatrix}. \tag{5.14}$$

Note that the local vector $\vec{u}$ has the boundary values inserted into it. The Newton update equation is then

$$\begin{pmatrix} 0.5 & -0.5 \\ -0.5 & 1.0 \end{pmatrix} \delta\vec{u} = \begin{pmatrix} 0 \\ 2.5 \end{pmatrix} \tag{5.15}$$

$$\delta\vec{u} = \begin{pmatrix} 5.0 \\ 5.0 \end{pmatrix}, \tag{5.16}$$

so that $\vec{U} = (5.0, 5.0)$ exactly as we expect. If we use the alternative approach, then we start with a residual

$$\vec{F}(\vec{U}) = \begin{pmatrix} 0.5 & 0 & -0.5 \\ 0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{pmatrix} \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix} \tag{5.17}$$

$$= \begin{pmatrix} 2.5 \\ 0 \\ -2.5 \end{pmatrix}, \tag{5.18}$$

and Newton update equation

$$\begin{pmatrix} 0.5 & 0 & -0.5 \\ 0 & 0.5 & -0.5 \\ -0.5 & -0.5 & 1.0 \end{pmatrix} \delta\vec{U} = \begin{pmatrix} -2.5 \\ 0 \\ 2.5 \end{pmatrix}. \tag{5.19}$$

We then reset the first equation and put a zero on the rhs, obtaining the same solution

$$\begin{pmatrix} 1.0 & 0 & 0 \\ 0 & 0.5 & -0.5 \\ 0 & -0.5 & 1.0 \end{pmatrix} \delta\vec{U} = \begin{pmatrix} 0.0 \\ 0 \\ 2.5 \end{pmatrix} \tag{5.20}$$

$$\delta\vec{U} = \begin{pmatrix} 0.0 \\ 5.0 \\ 5.0 \end{pmatrix}. \tag{5.21}$$

and then insert the boundary values into the solution

$$\vec{U} = \begin{pmatrix} 5.0 \\ 5.0 \\ 5.0 \end{pmatrix}. \tag{5.22}$$

If we had known the problem was exactly linear, we could insert the boundary value itself in the rhs.

5.2.2 Parallelism

TODO Explain SF (**KnepleyLangeGorman2015**; Brown 2011)

5.3 Problems

Problem V.1 The schemes detailed above are designed for the situation in which we fix the value of a single unknown. However, it is easy to imagine that we would like to fix the value of a combination of unknowns. For example, suppose we are solving a problem for fluid flow in a cavity where the flow obeys the Euler equations, so that the constraint specifies that the fluid has no velocity normal to the boundary. If the boundary aligns with the coordinates axes, we

can simply constrain that component of velocity. Now let our boundary be inclined at the 45° angle (see Fig. **??**) so that now our constraint equation becomes

$$\begin{aligned} \vec{u} \cdot \hat{n} &= 0 \\ \begin{pmatrix} u \\ v \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} &= 0 \\ u + v &= 0. \end{aligned}$$

Design a system for enforcing this boundary condition in both cases above, namely eliminating constrained unknowns and replacing redundant equations.

References

Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems*. 2nd. SIAM. DOI: 10.1016/S1570-579X(01)80025-2.

— (1996). *Iterative Methods for Sparse Linear Systems*. Boston: PWS Publishing Company.

Brown, Jed (2011). *Star forests as a parallel communication model*. URL: https://jedbrown.org/files/StarForest.pdf.

Chapter 6

Simple Finite Differences

6.1 Structured Meshes

Despite trememdous theoretical and practical success in computational mechanics, traditional PDE codes (Hughes 2000) can rarely compare different discretizations, for instance different orders of finite element, or different types of elements, or finite volume and finite element discretizations. Nor can they compare different mesh types, such as simplicial, hexahedral, or polyhedral. They cannot easily run different dimensional problems, so that some parts of the computation can be repalced by lower dimensional models, or that reductions can be compared against the full problem.

These problems stem from mesh representations which are inflexible and discretization interfaces which hopelessly tangled in the mesh interface. The traditional mesh interface is too specific in that it carrys along a lot of unnecessary data and categories. Code for the assembly of vectors, such as residuals, and matrices, such as Jacobians, is specialized to each dimension, cell shape, and approximation space. Internal iterations make explicit reference to element type, such as `getVertices(faceID)` or `getAdjacency(edgeID, VERTEX)` or `getAdjacency(edgeID, dim = 0)`, but it is possible to have a single interface for all of these cases. Furthermore, there is no interface for transitive closure, making code instead depend on awkward nested loops which are different in each dimension.

Moreover, in order to enable an optimal solver, we often need to separate the contribution of different fields to the overall problem, and assemble auxiliary operators not directly involved in the solution. The traditional interface between the mesh and solver is too general in that it omits crucial information. The linear/nonlinear solver is not told about the decomposition of the solution space into fields, but merely given a vector or matrix. This means the solver cannot take advantage of this structure, say for block operations or perhaps saddle-point structure. It is also difficult to use auxiliary data, such as eigen-estimates for smoothers or near null spaces for coarse problems.

In PETSc, the interface between meshes and solvers is handled by the DM object, which has three principal jobs. It must represent the topology of the underlying mesh, layout data over that topology, and manage the map between the local assembly space and the global solver space. Each of these jobs is vastly simpler in the case of structured grids, as opposed to unstructured grids, and thus we will begin our exposition there.

6.1.1 Structured Grids

We will understand structured grids to be Cartesian products of one-dimensional chains, and use vertices to define our mesh. You could equally well take the dual point of view, and define the mesh by cells, since this dual mesh has the same topology. Each vertex will be refered to using a tuple (i, j, k) where indices beyond the dimension of the mesh are ignored. This representation makes the toplogy apparent, in that a vertex (i, j, k) has six unit distance neighbors in 3D, or 26 using the full box stencil, identified by $(i \pm 1, j \pm 1, k \pm 1)$. In general, the distance d neighbors are given by $(i \pm d', j \pm d', k \pm d')$ for $0 < d' \leq d$.

It is possible to have a very general data layout over a structured mesh using a PetscSection object. However, support for this in PETSc is incomplete, so we will postpone a discussion of general data layouts until Chapter 7 on unstructured data layout. We will confine ourselves to co-located data layouts on structured grids, meaning that dofs may only be attached to vertices. It is possible to represent staggered discretizations this way by imagining some fields as lying on cells rather than vertices, and including appropriate buffer regions, but we will not review this approach.

With the aforementioned restrictions, the relation between the local assembly space and global solver space is particularly simple. The local space will be defined as the union of the set of vertices local to a given process and a halo region of distance s, which stands for stencil width, as shown in Fig 6.1. The global space is just space of dofs on all vertices, or equivalently the direct sum of the local spaces without the halo regions. The DM provides the DMGlobalToLocalBegin()/DMGlobalToLocalEnd() pair, and its converse DMLocalToGlobalBegin()/DMLocalToGlobalEnd(), in order to map from a vector in one space to another. Since the local space redundantly represents some dofs, the map from the local space to the global space has an InsertMode which dictates how redundant data should be combined. This mapping can also be extracted, using DMGetLocalToGlobalMapping(), as a list of the global index for every local unknown.

Note here that PETSc not only handles ghost value coherence with the above functions, but also all parallelism inherent in the problem. The user can completely assemble the residuals and Jacobians in the sequential local space (see Chapter 8), and then let PETSc construct the global objects appropriate for the linear and nonlinear solvers.

The DM provides methods to create a parallel vector in the global space, DMCreateGlobalVector(), a sequential vector in the local space, DMCreateLocalVector(), and a matrix in the global space, DMCreateMatrix(). It would perhaps make

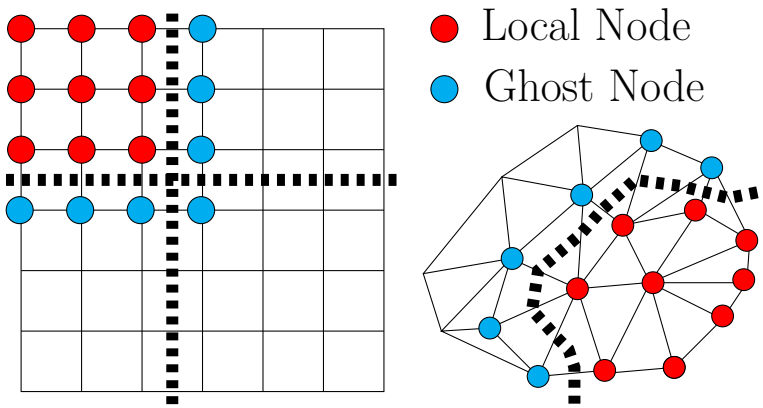


Figure 6.1: Structured and unstructured meshes using an overlapping vertex partition.

sense to also allow matrices in the local space, which are now handled through the somewhat obscure MATIS implementation.

Instead of making geometry intrinsic to the mesh data structure, as is commonly done, we treat it in the same way as other data over the mesh, such as the solution field. Thus we have another DM with the same topology, retrieved using DMGetCoordinateDM(), that describes the layout of the coordinate data. The coordinates themselves can be retrieved in the global space using DMGetCoordinates(), and in the local space using DMGetCoordinatesLocal().

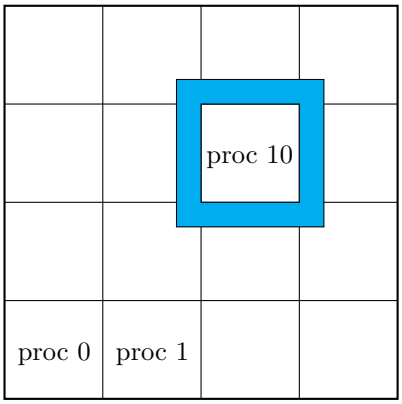
6.1.2 Examining DMDA in PETSc

The DMDA class is a specialization of DM interface to structured grids. It employs an overlapping vertex-based partition, as shown in Fig. 6.1. The parallel layout is specified completely in the constructor, so we will go over each argument in detail. The constructor for a two dimensional DA is given by

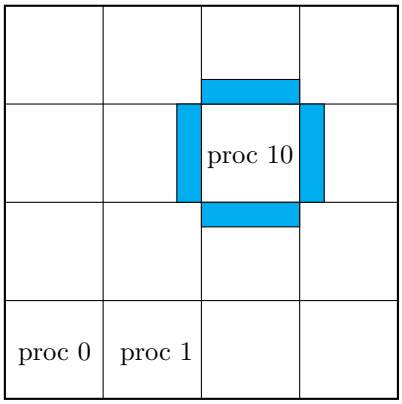
```
DMDACreate2d(comm, bdX, bdY, stype, M, N, m, n, dof, s, lm, ln, &da);
```

The first argument is the communicator, specifying the group of processes which will handle this grid. The next two arguments set the type of boundary behavior, meaning the global topology of the grid. Notice that if this was a three dimensional grid, there would be a third bdZ arugment. If this is just a square, we use DM_BOUNDARY_NONE, whereas we can create a cylinder by giving DM_BOUNDARY_PERIODIC for bdX, or a Möbius strip using DM_BOUNDARY_TWIST. Providing DM_BOUNDARY_PERIODIC for both arguments would give a torus. Using DM_BOUNDARY_GHOSTED creates a layer of vertices s (the stencil width) deep along that boundary. These *ghost vertices* are not present in the global space, only in the local space, exactly as the partition overlap.

The `stype` argument specifies the stencil type, `DMDA_STENCIL_BOX` or `DMDA_STENCIL_STAR`, as shown below, and the stencil width is given by `s`.



Box Stencil



Star Stencil

The `dof` argument gives the number of dofs on each vertices. The `m,n` and `M,N` arguments for the number of local and global vertices in the x and y directions function exactly as the size arguments to a PETSc `Mat` object, and `PETSC_DETERMINE` is also an acceptable input. The `lm` and `ln` arguments, arrays of size m and n, let the user completely specify the grid partition. The `lm` array holds the number of vertices in the x-direction for each partition. In order to be consistent, the entries in `lm` must sum to M, and similarly for `ln`. In the usual case, PETSc partitions the grid automatically, and the user passes `NULL`. However, these arguments are normally used in order to make the parallel division agree with another existing structure. For example, a `DMDA` can be forced to align with a existing `DMDA` with a different number of vertices, or with a user-defined structure.

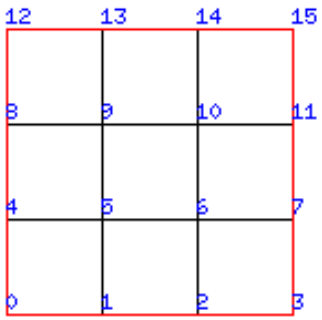


Figure 6.2: Graphical output from `-dm_view draw` in SNES ex5.

The constructor call is merely a convenience function, and the user can accomplish the same thing using the generic create function and API calls:

```
DMDACreate(comm, &da);
DMSetDimension(da, 2);
DMDASetSizes(da, M, N, 1);
DMDASetNumProcs(da, m, n, PETSC_DECIDE);
DMDASetBoundaryType(da, bx, by, DM_BOUNDARY_NONE);
DMDASetDof(da, dof);
DMDASetStencilType(da, stype);
```

```
DMDASetStencilWidth(da, s);
DMDASetOwnershipRanges(da, lx, ly, NULL);
DMSetFromOptions(da);
DMSetUp(da);
```

It is clear from above how DMDA differs in the 1D and 3D cases.

We can use SNES example 5 (ex5) to demonstrate the creation of a DMDA object. We first build the example

```
cd ${PETSC_DIR}/src/snes/examples/tutorials
make ex5
```

then we can get a text representation of the DMDA

```
./ex5 -dm_view
```

which shows both the DMDA and coordinate DMDA

```
DM Object: 1 MPI processes
  type: da
Processor [0] M 4 N 4 m 1 n 1 w 1 s 1
X range of indices: 0 4, Y range of indices: 0 4
DM Object: 1 MPI processes
  type: da
Processor [0] M 4 N 4 m 1 n 1 w 2 s 1
X range of indices: 0 4, Y range of indices: 0 4
```

Notice that the first grid has a single dof since it is the scalar Laplace problem, whereas the second grid has two dof since there are two coordinates per vertex. Using

```
./ex5 -da_grid_x 5 -da_grid_y 5 -dm_view draw -draw_pause -1
```

we get a graphical representation, where the numbers are a total order on the vertices, as shown in Figure 6.2.

We have also shown how to change the initial grid size. If we run on four processes

```
mpiexec -n 4 ./ex5 -da_grid_x 5 -da_grid_y 5 -dm_view draw -draw_pause -1
```

the ordering has changed. This occurs because PETSc renumbers the global problem so that vertices within each partition are contiguous, simplifying the indexing for global vectors and matrices. The numbering is reproduced below.

Proc 2			Proc 3	
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4
Proc 0			Proc 1	

Natural numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

PETSc numbering

6.2 Residual Evaluation

The DM assembly interface is based upon local callback functions which calculate the residual and Jacobian. In PETSc examples these are often called `FormFunctionLocal()` and `FormJacobianLocal()`. A DM is first associated with a solver, e.g using `SNESSetDM()` or `TSSetDM()`, and then callbacks are registered with `DMSNESSetFunctionLocal()` or `DMTSSetJacobianLocal()`. When PETSc needs to evaluate the nonlinear residual $\vec{F}(\vec{x})$,

- PETSc maps the global input vector $\vec{x}$ to a local vector $\vec{x}_l$ using `DMGlobalToLocal()`
- Each process evaluates the local residual $\vec{f}_l$
- PETSc assembles the global residual $\vec{f}$ from $\vec{f}_l$ using `DMLocalToGlobal()`

For structured grids, PETSc introduces another level of reordering, for the local space, so that the domain again appears to be a rectangular grid section and can be accessed through a multidimensional C array. The reordering for our ex5 run with a 5×5 grid is shown below with the ghost values in cyan.

Proc 2			Proc 3	
X	X	X	X	X
X	X	X	X	X
12	13	14	15	X
8	9	10	11	X
4	5	6	7	X
0	1	2	3	X
Proc 0			Proc 1	

Local numbering

Proc 2			Proc 3	
21	22	23	28	29
18	19	20	26	27
15	16	17	24	25
6	7	8	13	14
3	4	5	11	12
0	1	2	9	10
Proc 0			Proc 1	

Global numbering

In 2D, the user provided function which calculates the nonlinear residual has signature

```
(*lr)(DMDALocalInfo *info, PetscScalar **x, PetscScalar **r, void *ctx)
```

where `info` is a structure containing all layout and numbering information, `x` is the current local solution represented as a multidimensional array, `r` holds the local residual as another array, and `ctx` is an optional user context passed to `DMDASNESSetFunctionLocal()`. In SNES ex5, we solve the Bratu equation, a mildly nonlinear eigenvalue problem,

$$\Delta u + \lambda e^u = 0. \tag{6.1}$$

We will fix λ and treat it as a nonlinear system of equations, so that the local residual function is given by

```
lr(DMDALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx)
{
  PetscReal lambda = ((AppCtx *) ctx)->param;
```

```
  PetscReal hx, hy, hxdhy, hydhx;
  PetscInt i, j;

  hx = 1.0/(PetscReal)(info->mx-1);
  hy = 1.0/(PetscReal)(info->my-1);
  hxdhy = hx/hy;
  hydhx = hy/hx;
  for(j = info->ys; j < info->ys+info->ym; ++j) {
    for(i = info->xs; i < info->xs+info->xm; ++i) {
      PetscScalar u;

      u = x[j][i];
      /* Apply homogeneous Dirichlet conditions */
      if (i==0 || j==0 || i == M || j == N) {
        f[j][i] = 2.0*(hydhx+hxdhy)*u; continue;
      }
      u_xx = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
      u_yy = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
      f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
    }
  }
}
```

If the solution field is no longer scalar, but has multiple components, another index is added for the field components, as shown in SNES ex19.

When using DMDA and a finite difference approximation, it is convenient to use the first method for application of Dirichlet boundary conditions, namely replacing equations for the constrained unknowns. Above, we replace the equations for boundary unknowns with the deviation from the boundary value, $u-0$, suitably scaled. Note that the (i, j, k) vertex indices used in the function are global, so if the local patch does not contain the boundary, the branch will never be taken.

6.3 Jacobian Evaluation

The user callabck for Jacobian assembly is largely the same as the residual callback, except that the matrix is an Mat object rather than a raw array.

```
(*lj)(DMDALocalInfo *info, PetscScalar **x, Mat J, void *ctx)
```

The Bratu Jacobian can also be calculated with simple serial code.

```
lj(DMDALocalInfo *info, PetscScalar **x, Mat J, void *ctx)
{
  for(j = info->ys; j < info->ys + info->ym; j++) {
    for(i = info->xs; i < info->xs + info->xm; i++) {
```

```
      row.j = j; row.i = i;
      if (i == 0 || j == 0 || i == mx-1 || j == my-1) {
        v[0] = 1.0;
        MatSetValuesStencil(J, 1, &row, 1, &row, v, INSERT_VALUES);
      } else {
        v[0] = -(hx/hy); col[0].j = j-1; col[0].i = i;
        v[1] = -(hy/hx); col[1].j = j; col[1].i = i-1;
        v[2] = 2.0*(hy/hx+hx/hy) - hx*hy*lambda*PetscExpScalar(x[j][i]);
        v[3] = -(hy/hx); col[3].j = j; col[3].i = i+1;
        v[4] = -(hx/hy); col[4].j = j+1; col[4].i = i;
        MatSetValuesStencil(J, 1, &row, 5, col, v, INSERT_VALUES);
      }
    }
  }
}
```

However since we are inserting values into a global `Mat` Object, we would need to use global row and column indices for `MatSetValues()`. We could use local indices instead by calling `MatSetValuesLocal`, but we would still need to calculate the row indices from the vertex indices. PETSc provides a call which does this conversion automatically,

```
MatSetValuesStencil(Mat A, m, MatStencil idxm[], n, MatStencil idxn[],
                    PetscScalar values[], InsertMode mode);
```

where rather than a row or column index, the user passes a MatStencil structure which holds the global vertex indices and field component. These are converted to global indices, and the values remain in the same logically dense block. This same insertion scheme will be used for unstructured grids, but will allow cells, edges, and faces in addition to vertices.

We can view the parallel Jacobians produced by ex5 using options

```
mpiexec -n 2 ./ex5 -da_grid_x 10 -da_grid_y 10 -mat_view draw -draw_pause -1
```

as well as the denser Jacobian produced by SNES ex48,

```
mpiexec -n 3 ./ex48 -mat_view draw -draw_pause 1 -da_refine 3 -mat_type aij
```

It is often that case that a user is unwilling or unable to code the Jacobian for a given set of equations. It could be too complex or expensive to warrant assembly. In this case, PETSc provides a finite difference approximation to the true Jacobian, which is usually accurate to single precision. It uses the simple formula for the Jacobian action,

$$J(\vec{u})\vec{v} \approx \frac{1}{h}\left(F(\vec{u}+h\vec{v}) - F(\vec{u})\right), \tag{6.2}$$

where there are some heuristic for choosing h (Brown and Saad 1990). If we act on a unit vector, $\vec{v} = e_i$, we get the ith column of the Jacobian. Thus, naively we would need N actions to assemble the entire Jacobian. However, if we can

perturb two values, say i and j, such that they do not generate values in the same row, we can calculate them simultaneously. A vertex i interacts with all vertices a distance s away. If vertices i and j are both a distance s from vertex k, then a perturbation in either can contribute to rwo k in the Jacobian. Thus, we must group vertices into sets which are at least distance $2s$ from each other. For the common case $s = 1$, this is called a 2-coloring.

6.4 Code Verification

There is an old numerical saying, originating with the von Neumann analysis of timestepping methods, that

consistency + stability = convergence

which we will illustrate in the context of linear equations. Convergence means that our approximate solutions x become closer and closer to the true solution x^*. Thus we start by bounding the *error*, $x - x^*$, in terms of the residual, $r = Ax - b$,

$$||x - x^*|| = ||A^{-1}A(x - x^*)||, \tag{6.3}$$
$$= ||A^{-1}r||, \tag{6.4}$$
$$\leq ||A^{-1}||\,||r||. \tag{6.5}$$

We can see that *stability*, namely the invertibility of our linear operator A, combined with *consistency*, or the ability to solve our equations, produces convergence. We also have

$$||b|| = ||Ax^*|| \tag{6.6}$$
$$||b|| \leq ||A||\,||x^*|| \tag{6.7}$$
$$\frac{1}{||x^*||} \leq \frac{||A||}{||b||} \tag{6.8}$$

so that the version with relative errors and residuals is also true

$$\frac{||x - x^*||}{||x^*||} \leq ||A||||A^{-1}||\frac{||r||}{||b||} \tag{6.9}$$
$$\leq \kappa(A)\frac{||r||}{||b||} \tag{6.10}$$

where $\kappa(A)$ is the *condition number* of the operator.

We would like to test our discretization, physics, and solver. We could do that if we had an exact solution to the continuum problem. In order to test the discretization, we could look at the L_2 (or other suitable function norm) difference between the exact solution and its discretization on a series of refined meshes to quantify the discretization error. It is very common to look at the slope on a log-log graph, since we usually have a bound like

$$\|u - u_h\| < C(u)h^k, \tag{6.11}$$

where k is the order of convergence, and C is independent of h, coming from Theorem (5.4.8) of (Brenner and Scott 2002). If we consider the Laplace problem using P_1 elements, this gives a bound for the solution

$$\|u - u_h\|_2 < Ch^2 |u|_{H^k} \tag{6.12}$$

where the H^k norm looks like a norm of the kth derivative. This is also a good way to look at other quantities of interest. For example, if we are solving the Stokes problem, we might also want to look at a measure of the mass conservation, $\|\nabla \cdot \vec{u}\|$.

We can attempt to check the physical equations by looking the residual of the projected exact solution, $F(u_h)$. How close will this be zero? To answer this, lets look at some facts about Galerkin discretizations of elliptic equations $F(u_h) = a(u_h, v_h) - f(v_h)$, and we will eventually use the Laplacian to make the derivation simpler. First, we have Galerkin orthogonality, which says that the error in the finite dimensional solution, $u - u_n$, is orthogonal to any member of the finite dimensional approximation space V,

$$a(u - u_h, v_n) = a(u, v_n) - a(u_h, v_n) = f(v_n) - f(v_n) = 0 \tag{6.13}$$

since $v_n \in V_n \subset V$. The operator is elliptic because

$$a(u, u) \geq \alpha \|u\|^2 \tag{6.14}$$

and also normally satisfies a boundedness condition

$$a(u, v) \leq C\|u\|\|v\| \tag{6.15}$$

where C and α are positive. We can now prove Cea's Lemma

$$\begin{aligned}
\alpha\|u - u_h\|^2 &\leq a(u - u_h, u - u_h) && \text{from Eq. (6.14)} && (6.16)\\
&\leq a(u - u_h, u - v_n) && \text{from Eq. (6.13)} && (6.17)\\
&\leq C\|u - u_h\| \, \|u - v_n\| && \text{from Eq. (6.15)} && (6.18)\\
\|u - u_h\| &\leq \frac{C}{\alpha}\|u - v_n\| && && (6.19)\\
\|u - u_h\| &\leq \inf_{v_n \in V_n} \frac{C}{\alpha}\|u - v_n\|. && && (6.20)\\
& && && (6.21)
\end{aligned}$$

For the case of the Laplacian in the H_1 seminorm, $C = \alpha = 1$, which means that the orthogonal H_1 projection of u is the Galerkin solution u_h, so that the residual should be zero. However, in general the Galerkin solution and the projected exact solution do not coincide, and we can only bound one by the other using the condition number $\frac{C}{\alpha}$. We do have that u_h and $P_h u$ are both within h^k of the true solution u, so that

$$\|A_h(u_h - P_h u)\| \leq C\|u_h - P_h u\| \leq C' h^k \|u\|. \tag{6.22}$$

Thus, we cannot expect $F(P_h u)$ to be too small outside of the Laplacian and mass matrix without some more delicate estimates.

Finally, we can check the solver by actually solving the system and then looking at the L_2 error using the exact solution. But coming up with analytic solutions to arbitrary PDEs can be extremely challenging. However, suppose that we allow ourselves to change both the boundary conditions and add a forcing function on the rhs. For example, suppose we have a bilinear form a, and we choose a solution s. Then we let the rhs be $(b_s, v) = a(s, v)$, so that

$$a(u, v) - (b_s, v) = 0. \tag{6.23}$$

Now the solve for u should produce an approximation to s. This called the Method of Manufactured Solutions (MMS), and there is a wonderful and concise description by Roache in (Roache 2002).

6.4.1 MMS in PETSc

We will use SNES ex5, which employs a simple first order finite difference discretization, to demonstrate MMS in the simplest setting. We first choose an exact solution

$$u^* = x(1-x)y(1-y) \tag{6.24}$$

which makes things a little easier since we do not have to change the boundary conditions as u is already zero when $x, y = 0, 1$. Next, we look at the residual of the exact solution

$$-\Delta u^* - \lambda e^{u^*} = 2x(1-x) + 2y(1-y) - \lambda e^{x(1-x)y(1-y)} \tag{6.25}$$

which we will subtract from the residual. Note that the Jacobian is unchanged because ths term is independent of the solution u. The modified residual function is

```
lr(DMDALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx)
{
  DM coordDA;
  Vec coordinates;
  DMDACoor2d **coords;
  PetscReal lambda = ((AppCtx *) ctx)->param;
  PetscReal hx, hy, hxdhy, hydhx;
  PetscInt i, j;

  hx = 1.0/(PetscReal)(info->mx-1);
  hy = 1.0/(PetscReal)(info->my-1);
  hxdhy = hx/hy;
  hydhx = hy/hx;
  /* Retrieve coordinates */
  DMGetCoordinateDM(info->da, &coordDA);
```

```
  DMGetCoordinates(info->da, &coordinates);
  DMDAVecGetArray(coordDA, coordinates, &coords);
  for(j = info->ys; j < info->ys+info->ym; ++j) {
    for(i = info->xs; i < info->xs+info->xm; ++i) {
      PetscScalar u;

      u = x[j][i];
      x = coords[j][i].x;
      y = coords[j][i].y;
      /* Apply homogeneous Dirichlet conditions */
      if (i==0 || j==0 || i == M || j == N) {
        f[j][i] = 2.0*(hydhx+hxdhy)*u; continue;
      }
      u_xx = (2.0*u - x[j][i-1] - x[j][i+1])*hydhx;
      u_yy = (2.0*u - x[j-1][i] - x[j+1][i])*hxdhy;
      f[j][i] = u_xx + u_yy - hx*hy*(lambda*exp(u)
        + 2*x*(1 - x) + 2*y*(1 - y) - lambda*exp(x*(1 - x)*y*(1 - y)));
    }
  }
  DMDAVecRestoreArray(coordDA, coordinates, &coords);
}
```

We can check the error using the L_2 norm, also called the Root Mean Square deviation (RMS),

```
Vec e;
PetscReal errorl2, errorinf;
PetscInt N;

VecDuplicate(x, &e);
if (MMS == 1) {FormExactSolution1(da, &user, e);}
else {FormExactSolution2(da, &user, e);}
VecAXPY(e, -1.0, x);
VecNorm(e, NORM_2, &errorl2);
VecNorm(e, NORM_INFINITY, &errorinf);
VecGetSize(e, &N);
PetscPrintf(PETSC_COMM_WORLD, "N: %D error L2 %g Linf %g\n", N,
  (double) errorl2/PetscSqrtReal(N), (double) errorinf);
VecDestroy(&e);
```

Now we can run a series of tests with increasing mesh size

```
for i in `seq 1 6`;
do
  ./ex5 -pc_type mg -pc_mg_levels 3 -pc_mg_galerkin \
    -da_grid_x 17 -da_grid_y 17 -da_refine $i \
    -mg_levels_ksp_norm_type unpreconditioned -mg_levels_ksp_chebyshev_esteig 0.5,1.1 \
    -mg_levels_pc_type sor -pc_mg_type full -mms 1
done
```

which outputs

```
N: 1089 error L2 1.35637e-14 Linf 3.02883e-14
N: 4225 error L2 1.36056e-14 Linf 2.98858e-14
N: 16641 error L2 3.05717e-13 Linf 8.70887e-13
N: 66049 error L2 5.53919e-14 Linf 1.83256e-13
N: 263169 error L2 1.96731e-07 Linf 4.24471e-07
N: 1050625 error L2 1.96912e-07 Linf 4.24397e-07
```

This illustrates a potential pitfall of MMS, namely that the solution will not be "generic" enough. Here we get superconvergence at the vertices due to the symmetric nature of the solution. If instead we choose

$$u^* = \sin(\pi x)\sin(\pi y), \tag{6.26}$$

which also fulfills the boundary conditions, and run with `-mms 2`, we get

```
N: 1089 error L2 0.00108949 Linf 0.0023403
N: 4225 error L2 0.000275821 Linf 0.000583337
N: 16641 error L2 6.94434e-05 Linf 0.000145726
N: 66049 error L2 1.74255e-05 Linf 3.64248e-05
N: 263169 error L2 4.36468e-06 Linf 9.10578e-06
N: 1050625 error L2 1.09222e-06 Linf 2.27641e-06
```

We can plot this using Python tools

```
import numpy as np
from pylab import legend, loglog, show, title, xlabel, ylabel

N = np.array([1089, 4225, 16641, 66049, 263169, 1050625])
eL2 = np.array([0.00108949, 0.000275821, 6.94434e-05, 1.74255e-05,
                4.36468e-06, 1.09222e-06])
einf = np.array([0.0023403, 0.000583337, 0.000145726, 3.64248e-05,
                9.10578e-06, 2.27641e-06])

loglog(N, eL2, 'r', N, 0.9 * N ** -1.0, 'r--',
       N, einf, 'g', N, 2.0 * N ** -1.0, 'g--',
       N, eL2 * np.sqrt(N), 'b', N, 0.9 * N ** -0.5, 'b--',)
title('SNES ex5 MMS $u = \sin(\pi x) \sin(\pi y)$')
xlabel('Number of Dof $N$')
ylabel('Solution Error $e$')
legend(['$L_2$', '$h^2 = N^{-1}$',
        '$L_\infty$', '$h^2 = N^{-1}$',
        '$\ell_2$', '$h = N^{-1/2}$'], 'upper right')
show()
```

We see that the ℓ_2 norm has a slower rate of convergence[1]. A slightly slicker script would parse the output of our simulation and plot it automatically, generating Fig. 6.3.

[1] Shown in Pascal Frey's lecture notes for MA691, p.85

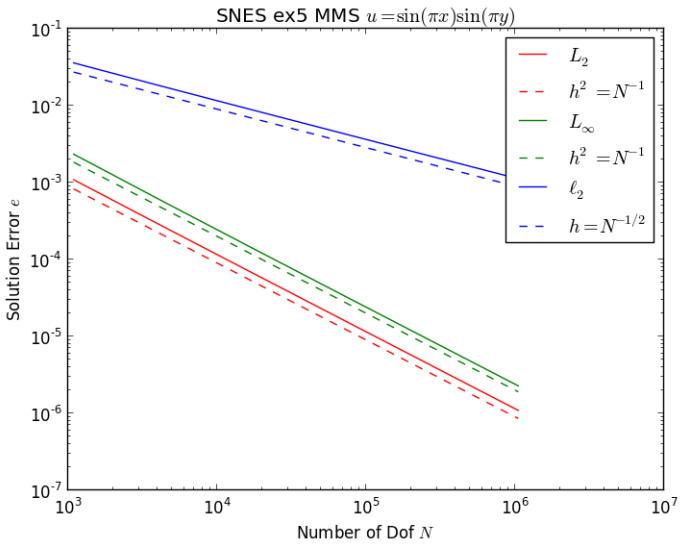


Figure 6.3: Using MMS in SNES ex5, we show that we get the expected rate of convergence of the error in the L_2, ℓ_2 and L_∞ norms.

6.5 Problems

Problem VI.1 Change SNES ex5 to run in 1, 2, and 3 spatial dimensions. Produce a mesh convergence graph similar to Fig. 6.3 for each dimension. Comment on the dimension dependence of the convergence for each norm.

Problem VI.2

Part I Carry out the MMS procedure for a modified Bratu equation which incorporates an inhomogeneous coefficient,

$$-\nabla \cdot \left(\tanh\left(x - \frac{1}{2} \right) \nabla u \right) - \lambda e^u = 0, \tag{6.27}$$

where $\lambda = 6.0$ as we used in the previous exercises. Use the same exact solutions as shown in the text,

$$u^* = x(1-x)y(1-y) \qquad\qquad u^* = \sin(\pi x)\sin(\pi y). \tag{6.28}$$

Create convergence graphs for the solutions in both the ℓ_2 and ℓ_∞ norms, as in the text. This problem becomes very hard to solve as the mesh size is increased, the opposite of the behavior we saw with the original equation. For example, if we use our script from before

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls \
    -da_grid_x 17 -da_grid_y 17 -da_refine $i \
    -pc_type mg -pc_mg_levels 3 -pc_mg_galerkin \
    -mg_levels_ksp_norm_type unpreconditioned -mg_levels_ksp_chebyshev_esteig 0.5,1.2 \
    -mg_levels_pc_type sor -pc_mg_type full -mms 3 \
  -snes_monitor -snes_converged_reason -ksp_converged_reason
done
```

we cannot even converge the first system.

```
  0 SNES Function norm 4.865204926677e-01
  Linear solve converged due to CONVERGED_RTOL iterations 3
  1 SNES Function norm 5.168062931528e-02
  Linear solve converged due to CONVERGED_RTOL iterations 15
  2 SNES Function norm 4.184362140608e-02
  Linear solve converged due to CONVERGED_RTOL iterations 23
  3 SNES Function norm 3.866392229628e-02
  Linear solve converged due to CONVERGED_RTOL iterations 25
  4 SNES Function norm 3.606888512126e-02
  Linear solve converged due to CONVERGED_RTOL iterations 30
  5 SNES Function norm 3.410310612571e-02
  Linear solve converged due to CONVERGED_RTOL iterations 58
  6 SNES Function norm 3.294064502174e-02
  Linear solve converged due to CONVERGED_RTOL iterations 57
  7 SNES Function norm 3.291316232478e-02
  Linear solve converged due to CONVERGED_RTOL iterations 54
Nonlinear solve did not converge due to DIVERGED_LINE_SEARCH iterations 7
```

Since we have control over the grid refinement, one common technique is to solve a smaller problem and use this as the initial guess for the larger problem, which is called *grid sequencing*. PETSc provides grid sequencing automatically using the option `-snes_grid_sequence`, however we will have to alter our error checking code to extract the final solution and grid by adding

```
  ierr = SNESGetSolution(snes, &y);CHKERRQ(ierr);
  ierr = SNESGetDM(snes, &dm);CHKERRQ(ierr);
```

Starting from a smaller grid,

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls -snes_grid_sequence $i -da_refine 1 \
    -ksp_rtol 1e-9 -pc_type mg -pc_mg_levels 3 -pc_mg_galerkin \
    -mg_levels_ksp_norm_type unpreconditioned -mg_levels_ksp_chebyshev_esteig 0.5,1.2 \
    -mg_levels_pc_type sor -pc_mg_type full -mms 3 \
    -snes_monitor -snes_converged_reason -ksp_converged_reason
done
```

we can converge the first five systems,

```
            0 SNES Function norm 6.805338772655e-01
            Linear solve converged due to CONVERGED_RTOL iterations 4
            1 SNES Function norm 5.154734539031e-01
            Linear solve converged due to CONVERGED_RTOL iterations 4
            2 SNES Function norm 5.899227765372e-02
            Linear solve converged due to CONVERGED_RTOL iterations 2
            3 SNES Function norm 4.111593886420e-03
            Linear solve converged due to CONVERGED_RTOL iterations 3
            4 SNES Function norm 2.234476660521e-05
            Linear solve converged due to CONVERGED_RTOL iterations 3
            5 SNES Function norm 1.179499962334e-09
          Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 5
          0 SNES Function norm 9.350819446678e-03
```

```
          Linear solve converged due to CONVERGED_RTOL iterations 4
          1 SNES Function norm 8.975872639470e-07
          Linear solve converged due to CONVERGED_RTOL iterations 4
          2 SNES Function norm 1.581449936603e-12
        Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
        0 SNES Function norm 4.820528651869e-03
        Linear solve converged due to CONVERGED_RTOL iterations 5
        1 SNES Function norm 2.839396976835e-08
        Linear solve converged due to CONVERGED_RTOL iterations 4
        2 SNES Function norm 3.513911836431e-15
      Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
      0 SNES Function norm 2.461494407946e-03
      Linear solve converged due to CONVERGED_RTOL iterations 10
      1 SNES Function norm 8.896591851778e-10
      Linear solve converged due to CONVERGED_RTOL iterations 10
      2 SNES Function norm 9.442247176540e-17
    Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
    0 SNES Function norm 1.244744690374e-03
    Linear solve converged due to CONVERGED_RTOL iterations 2807
    1 SNES Function norm 2.782787902882e-11
    Linear solve converged due to CONVERGED_RTOL iterations 2556
    2 SNES Function norm 1.850505579361e-16
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
  0 SNES Function norm 6.259834963960e-04
  Linear solve converged due to CONVERGED_RTOL iterations 7049
  1 SNES Function norm 8.694365205916e-13
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
0 SNES Function norm 3.139064083951e-04
Linear solve did not converge due to DIVERGED_ITS iterations 10000
Nonlinear solve did not converge due to DIVERGED_LINEAR_SOLVE iterations 0
```

which reveals that the problem is becoming quite ill-conditioned and our geometric multigrid solver cannot cope with the variation in coefficient. For these smaller serial problems, LU can be effective for checking convergence

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls -snes_grid_sequence $i \
    -da_refine 1 -ksp_rtol 1e-9 \
    -pc_type lu -mms 3 \
    -snes_converged_reason
done%$
```

and we see that if the linear systems can be solved, the nonlinear equation is effectively preconditioned with grid sequencing.

```
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 169 error l2 2.20735e-06 inf 7.9559e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 625 error l2 3.081e-07 inf 2.42794e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 2401 error l2 4.0427e-08 inf 8.29001e-06
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 9409 error l2 5.16863e-09 inf 2.62871e-06
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
N: 37249 error l2 6.53061e-10 inf 7.96102e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
N: 148225 error l2 8.20737e-11 inf 2.33813e-07
```

Part II Update the flop counting in the residual and Jacobian evaluation to account for the additional flops in the new equation. Using the total flop count output for each run, plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the flops executed. The

y-axis shows the precision of the result, and here we will use the error. Make this plot comparing two different solvers: SNES Newton using LU with grid sequencing, and SNES Newton using GMRES/GMG with grid sequencing.

Extra Cedit Use a higher order approximation of the derivative, predict the enhanced convergence, and demonstrate it with a convergence graph.

Problem VI.3

Part I Carry out the MMS procedure for an equation similar to Lane-Emden Equation by modifying the Bratu example,

$$-\Delta u - u^{\lambda} = 0, \tag{6.29}$$

where $\lambda = 1, 2, 5$. Use the same exact solutions as shown in the text,

$$u^* = x(1-x)y(1-y) \qquad u^* = \sin(\pi x)\sin(\pi y). \tag{6.30}$$

Create convergence graphs for the solutions in both the ℓ_2 and ℓ_∞ norms, as in the text.

Part II Update the flop counting in the residual and Jacobian evaluation to account for the additional flops in the new equation. Using the total flop count output for each run, plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the flops executed. The y-axis shows the precision of the result, and here we will use the error. Make this plot comparing two different solvers: SNES Newton using LU, and SNES Newton using GMRES/GMG.

Extra Cedit Use a higher order approximation of the derivative, predict the enhanced convergence, and demonstrate it with a convergence graph.

References

Hughes, Thomas J. R. (2000). *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Civil and Mechanical Engineering. Dover, p. 704. ISBN: 978-0486411811.

Brown, Peter N. and Youcef Saad (1990). "Hybrid Krylov Methods for Nonlinear Systems of Equations". In: *SIAM J. Sci. Stat. Comput.* 11, pp. 450–481.

Brenner, Susanne C. and L. Ridgway Scott (2002). *The mathematical theory of finite element methods*. Springer, p. 361. ISBN: 0-38795-451-1.

Roache, Patrick J. (2002). "Code verification by the method of manufactured solutions". In: *Journal of Fluids Engineering* 124.1, pp. 4–10.

Chapter 7

Data Layout and Discretization II

7.1 Unstructured Meshes

The PETSc `DMPlex` class is a generic interface for manifold topology, more precisely it encodes a CW complex (Wikipedia 2015a; Hatcher 2002). It is designed to orthogonalize the concerns of topology/geometry, discretization, data layout, and the solver. In most PDE codes, these are all inextricably entangled when evaluating residuals and Jacobians, such that a change in one feature, say the discretization, entails rewriting the majority of the code. This point of view is critical for library development, rather than the construction of monolithic, inflexible applications.

The ability to compare algorithms which use different meshes, discretizations, problem formulations, and solvers is crucial to the advance of computational science, and its maturation as a discipline. One of the most important contributions of PETSc was to make it easy to compare different Krylov solvers, as well as a subset of domain decomposition preconditioners. Not coincidentally, publication in the field of Krylov solvers was nearly wiped out as it became obvious that the available gains were quite small. It would be healthy for computational science to see a similar decline in publications for rival discretizations and meshes once the obvious candidates for important problems are identified by comparison.

The main advantage of DMPlex in representing topology is that it treats all the different pieces of a mesh, e.g. cells, faces, edges, vertices in exactly the same way. This allows the interface to be very small and simple, while remaining flexible and general. This also allows *dimension independent programming*, which means that the same algorithm can be used unchanged for meshes of different shapes and dimensions.

All pieces of the mesh are identified as mesh *points*, represented internals as just a `PetscInt` so that there is a total order on the topology. A mesh is

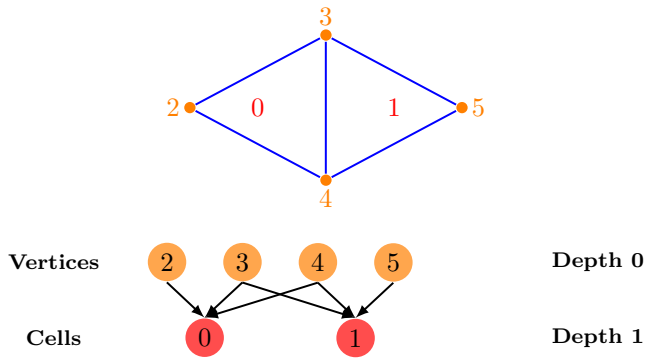


Figure 7.1: A 2D doublet mesh, two triangles sharing an edge along with the Hasse diagram for this mesh, expressed as a DAG.

built by relating points to other points, in particular specifying a "covering" relation among the points. For example, an edge is defined by being covered by two vertices, and a triangle can be defined by begin covered by three edges (or even by three vertices). In fact, this structure has been known for a long time. It is a Hasse Diagram (Wikipedia 2015b), which is a Directed Acyclic Graph (DAG) representing a cell complex using the covering relation. The graph edges represent the relation, which is a poset (partially-ordered set), and when it is segregated by dimension (or breadth-first level) a ranked poset.

For example, we can encode the triangular doublet mesh in Fig. 7.1 using the DAG shown below it. We can use the same scheme to represent the edges as well, show in Fig. 7.2. We can also represent a quadrilateral mesh, Fig. 7.3 and Fig. 7.4, and a tetrahedral mesh, Fig. 7.5. In fact, any CW-complex can be represented, which includes all conforming meshes we have seen used in computational science. Recent work **IsaacKnepley2016** extends the representation to cover non-conforming meshes as well.

We can use the PETSc API directly to construct a mesh, for example one in Fig. 7.2. We first consecutively number the mesh pieces. The PETSc convention is to number first cells, then vertices, then faces, and then edges. This convention can be violated, but many of the higher level tools require it. The set of points present in a mesh is declared using,

```
DMPlexSetChart(dm, 0, 11);
```

We then define the covering relation, which we call the *cone*. For any point p in the DAG, its cone consists of the in-edges. In order to preallocate our data structure, we first setup sizes,

```
DMPlexSetConeSize(dm, 0, 3);
DMPlexSetConeSize(dm, 1, 3);
DMPlexSetConeSize(dm, 6, 2);
DMPlexSetConeSize(dm, 7, 2);
```

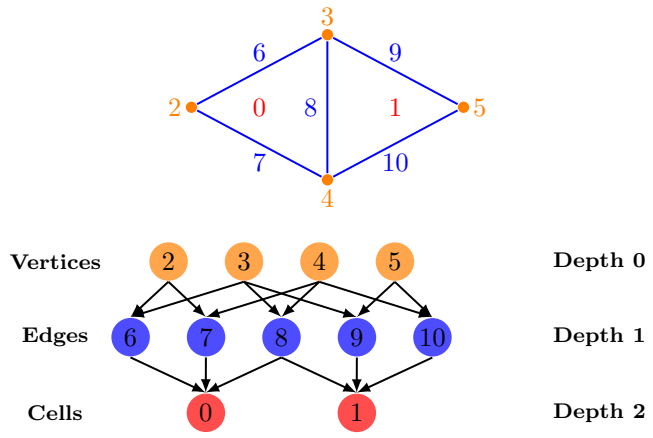


Figure 7.2: The 2D doublet mesh from Fig. 7.1, now incorporating edges, along with its Hasse diagram.

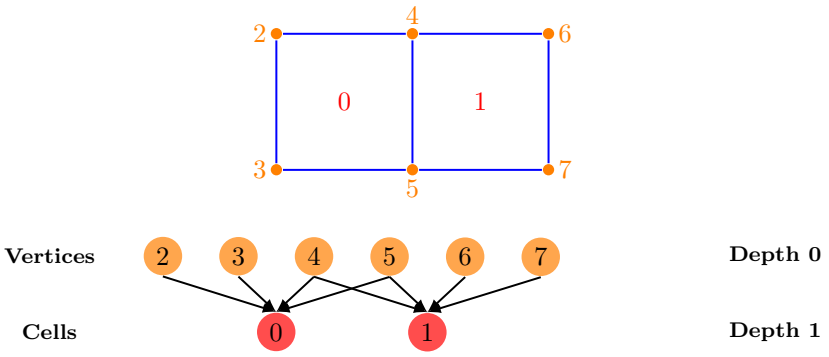


Figure 7.3: A 2D doublet mesh, two quadrilaterals sharing an edge along with the Hasse diagram for this mesh, expressed as a DAG.

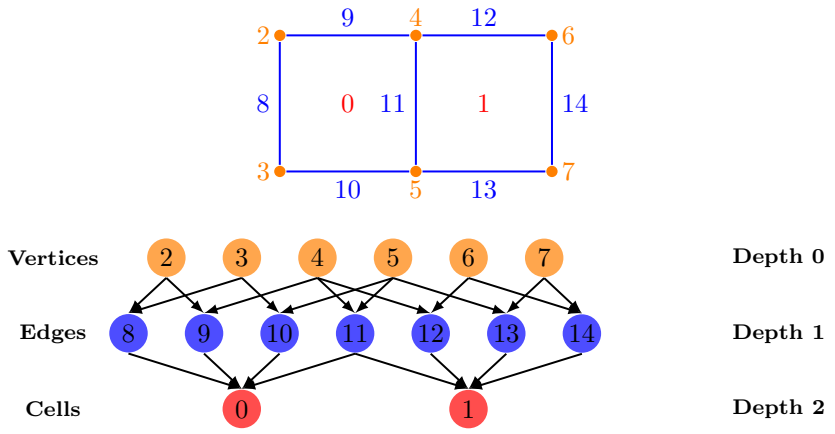


Figure 7.4: The 2D doublet mesh from Fig. 7.3, now incorporating edges, along with its Hasse diagram.

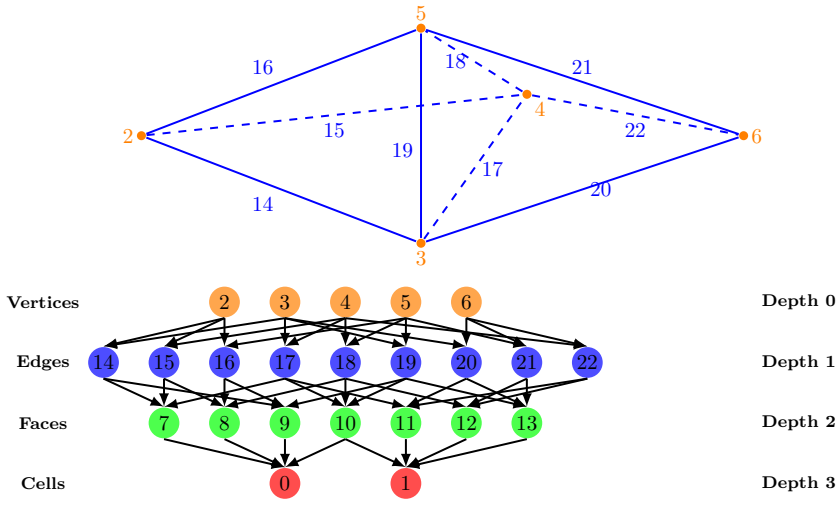


Figure 7.5: A 3D interpolated doublet mesh, two tetrahedra sharing a face, along with its Hasse diagram.

```
DMPlexSetConeSize(dm, 8, 2);
DMPlexSetConeSize(dm, 9, 2);
DMPlexSetConeSize(dm, 10, 2);
DMSetUp(dm);
```

and then point values,

```
DMPlexSetCone(dm, 0, [6, 7, 8]);
DMPlexSetCone(dm, 1, [8, 9, 10]);
DMPlexSetCone(dm, 6, [2, 3]);
DMPlexSetCone(dm, 7, [3, 4]);
DMPlexSetCone(dm, 8, [4, 2]);
DMPlexSetCone(dm, 9, [4, 5]);
DMPlexSetCone(dm, 10, [5, 2]);
```

where the arrays arguments are mnemonic, in that the actual call would have a pointer to that array. There is also an API for the dual relation, using DMPlexSetSupportSize() and DMPlexSetSupport(), but this can be calculated automatically by calling `DMPlexSymmetrize(dm)`. In order to support efficient queries, we also want to construct fast search structures, indices, for the different types of points, which is done using `DMPlexStratify(dm)`.

It is unusual for the user to construct the mesh at such a low level. Instead, a typical construction would use `DMPlexCreateFromCellList()` which takes the cell-vertex list and coordinates as input, or `DMPlexCreateFromFile()` which supports a number of formats such as Gmsh, ExodusII, and Fluent CAS.

By focusing on the key topological relations, the interface can be both concise and quite general (Knepley and Karpeev 2009). It is expressed as a single relation over the points, "covering", and ignores the extraneous detail that plagues other interfaces and implementations. There is no large enumeration of cell types, separate interface functions are not necessary for pieces of different dimension, and the implementation is quite small and easy to optimize since it is built from common graph operations. Moreover, many complex operations become much simpler. For example, the mesh dual can be obtained simply by reversing arrows in the DAG.

7.1.1 Basic Operations

The usefulness of `DMPlex` would not extend much beyond being a canonical representation if important solution strategies, such a finite elements or finite volumes for PDEs, could not be expressed by generic operations over the structure. In fact, we will strive to write *dimension independent* algorithms, so that our code will not change depending on spatial dimension, cell shape, or mesh composition.

The basic operations on a `DMPlex` object can be reduced to graph operations on its Hasse diagram. For example, to find the set of points which cover another point we use the `DMPlexGetCone()` operation, which is just the set of in-edges for the DAG vertex, as shown in Fig. 7.6. Also in that figure, we see that the

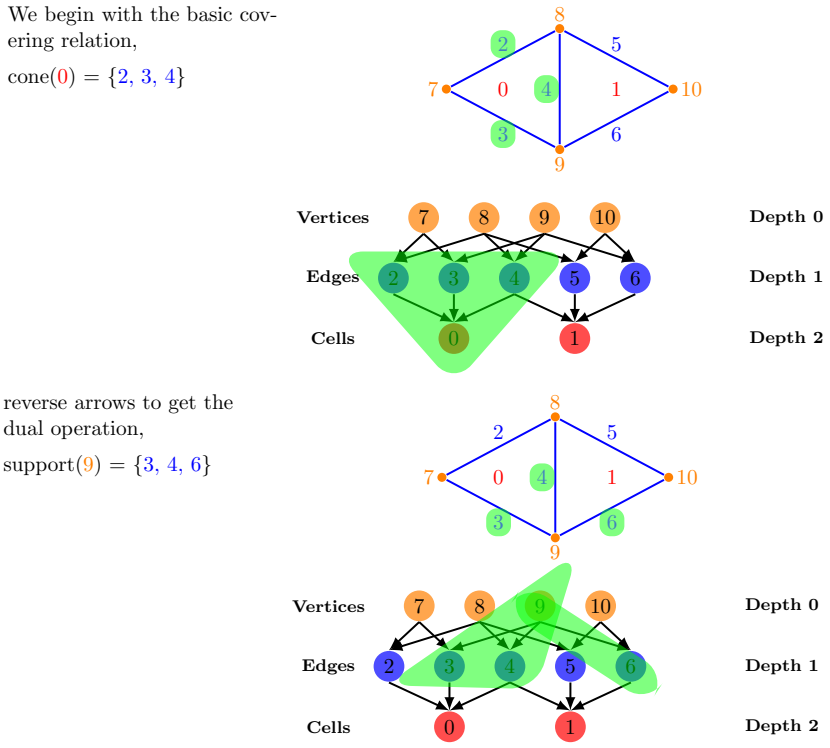


Figure 7.6: The basic operations are illustrated on a doublet mesh.

dual operation, the set of points covered by a given point, is just the set of out-edges. Thinking in this way makes it easy to write code which is independent of the particular features of a mesh. For example, suppose we wanted to get the set of neighbors for a given cell, meaning that they share a face. For a point p, this set is just $\mathtt{supp}(\mathtt{cone}(p))$. Notice that this works for triangles, quads, tetrahedra, hexes, and any other shape or dimension. It also works if I am asking the question about a boundary face in a higher dimensional mesh, and all using the same code.

Very often we are interested not only in a region of the domain, such as a cell, but in its closure, especially in the context of finite element methods. `DMPlex` provides the transitive closure of the basic operations `DMPlexGetCone()` and `DMPlexGetSupport()` as `DMPlexGetClosure()` and `DMPlexGetStar()`, respectively. These operations are illustrated in Fig. 7.7. Using closure can eliminate awkward nested loop structures, and is dimension independent whereas the number of loops often depends on the dimension.

Some queries are not easily answered using just adjacency information, but really require a set operation. For example, do two cells shared a common face, and if so what is it? We can answer these queries by imposing a lattice structure on the poset, and adding the associated *meet* and *join* operations. These are

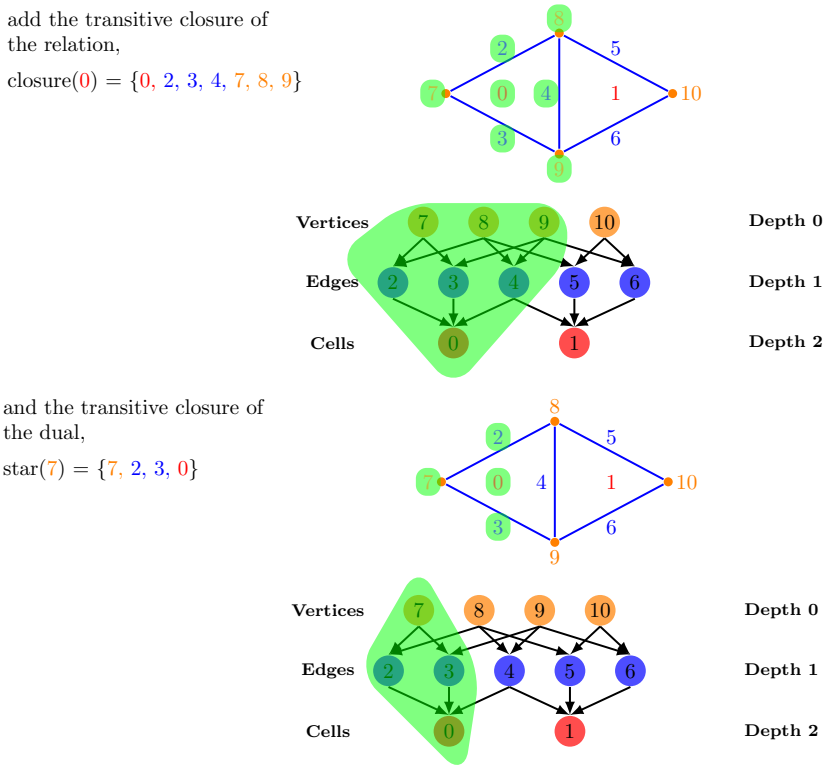


Figure 7.7: The transitive closure operations are illustrated on a doublet mesh.

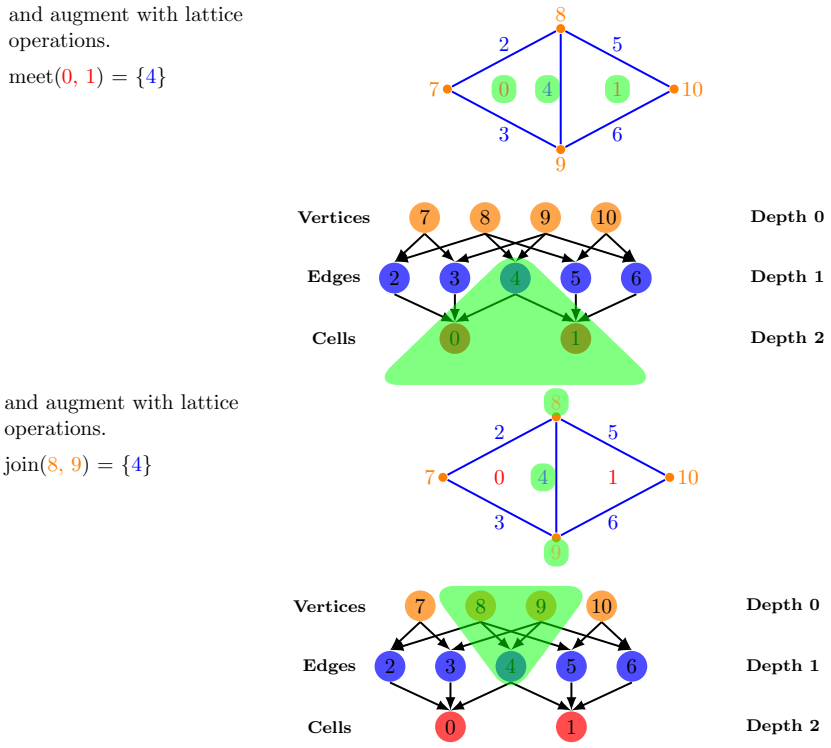


Figure 7.8: The lattice operations are illustrated on a doublet mesh.

illustrated in Fig. 7.8.

7.1.2 Labels

It is very common for mesh subsets to be labeled in some way. For example, if a boundary condition should be applied to a section of the mesh boundary, or refinement should take place over part of the domain, or if part of the mesh should be communicated to another process, and for many other reasons, we would like to mark a subset of mesh points, and perhaps have many such sets. PETSc provides the `DMLabel` class to represent these markings.

A `DMLabel` is a bi-directional map from integers to sets of mesh points, often called a multimap. While it is possible to assign a given point to more than one integer value, only a single value will be returned on reverse lookup. The implementation uses hash tables for efficient creation, and sorted arrays for optimized lookup. If explicit bounds on the points are given, using the interface method `DMLabelCreateIndex(label, pStart, pEnd)`, extremely fast membership queries are possible using bitmaps.

7.1.3 Using DMPlex in PETSc

The most common way to create a DMPlex is to read in the output of a mesh generator

```
DMPlexCreateFromFile(comm, filename, interpolate, &dm);
```

where the interpolate flag tells PETSc whether to create the intermediate edges and faces in the mesh, or just return cells and vertices. Some mesh generators, Triangle (J. R. Shewchuk 1996; J. Shewchuk 2005) and TetGen (Si 2015; Si 2005), are accessible at the library level. If the user has a boundary DM, it can be meshed in using

```
DMPlexGenerate(dmBoundary, NULL, interpolate, &dm);
```

and this strategy is used to make stock meshes of the unit cube for simplices, DMPlexCreateBoxMesh(comm, dim, interpolate, &dm). For tensor product cells, the interface is much closer to the DMDA,

```
DMPlexCreateHexBoxMesh(comm, dim, cells,
  DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, &dm);
```

where the user sets the number of cells and the boundary behavior for each dimension. A mesh can also be stored and loaded from an HDF5 file,

```
DMCreate(comm, &dm);
DMSetType(dm, DMPLEX);
DMLoad(dm, viewer);
```

where the mesh has been previously saved using DMView().

New meshes can be constructed by refinement using the generic DM interface,

```
DMRefine(dm, comm, &dmRef);
if (dmRef) {
  DMDestroy(&dm);
  dm = dmRef;
}
```

A volume constraint on the largest cell can be set using DMPlexSetRefinementLimit(), or for nonuniform refinement using DMPlexSetRefinementFunction(). However, this mechanism uses the mesh generators themselves, and is thus restricted to serial meshes. The eventual incorporation of Pragmatic (Rokos and Gorman 2013) functionality will remove this restriction. Parallel uniform refinement is available by first setting the a flag, DMPlexSetRefinementUniform(dm, PETSC_TRUE), and then calling DMRefine(), and it can also be accomplished with the generic DM option -dm_refine num where num is the number of refinement levels.

A serial mesh can be automatically partitioned and distributed using,

```
DMPlexDistribute(dm, 0, NULL, &dmDist);
if (dmDist) {
```

```
  DMDestroy(&dm);
  dm = dmDist;
}
```

where the 0 is an amount of overlap for the partition, in terms of number of cells, and `NULL` can optionally be a `PetscSF` pointer that returns the communication pattern for shared points. This call uses the `PetscPartitioner` class underneath, and will be discussed further in Section 7.4.

When reading the mesh from a file, sets of mesh points are read in as `DMLabel` objects which are stored in the `DMPlex`. If none exist, the mesh boundary can be automatically marked using

```
DMLabel label;

DMPlexCreateLabel(dm, "boundary");
DMPlexGetLabel(dm, "boundary", &label);
DMPlexMarkBoundaryFaces(dm, label);
DMPlexLabelComplete(dm, label);
```

The third call marks only boundary faces (edges in 2D), and the fourth call adds the closure of each face to the label.

7.2 Data Layout

The strongest links between solvers and discretizations have to do with the the layout of data over a mesh. For instance,

- parallel partitioning, perhaps with overlap,
- division into fields,
- blocking of unknowns, and
- ordering of unknowns,

all spring from the data layout. In order to enable modularity, we encode all the information above in a simple data structure that can be understood by the linear algebra engine in PETSc without any reference to the mesh (topology) or discretization (analysis), the `PetscSection` object.

The name *section* was chosen to draw parallels between the section of a fiber bundle from analysis on manifolds and the linear algebraic vector with additional structure in PETSc. We can imagine that the additional structure in the large linear algebra space comes from gluing together many small spaces representing the partitions. This is exactly what happens in a fiber bundle, which glues together spaces associated with each point of a base space. The section is a thing that chooses values in each of the small spaces, just as our vector does. The analogy is not exact because we do not require that the vector satisfy additional constraints (e.g. trivialization), but this is a topic for future exploration.

7.2.1 Data Layout on Unstructured Grids

Data is associated to a mesh using the PetscSection object, which can be thought of as a generalization of PetscLayout, in the same way that a fiber bundle is a generalization of the normal Euclidean basis used in linear algebra. With PetscLayout, we associate a unit vector (e_i) with every point in the space, and just divide up points between a set of processes, which we will represent as a contiguous set of points indexed by MPI rank. Thus the full space is just the direct sum of the spaces for each partition. Using PetscSection, we can associate a set of dofs, a small space $\{e_k\}$, with each point, and though our points must be contiguous like PetscLayout, now they can be in any range [pStart, pEnd) since they are abstract entities. For mesh layout, the point space will usually be a subset of the mesh points, but we can use any index space, such as the set of partitions or the set of unknowns (we do this when describing adjacency for Jacobian construction).

The sequence for setting up a PetscSection is the following:

1. Specify the *chart*, or range [pStart, pEnd),
2. Specify the number of dofs per point, and
3. Call PetscSectionSetUp().

For example, using the triangular doublet mesh from Fig. 7.2, we can layout data for a continuous Galerkin P_3 finite element method, shown in Figure 7.9,

```
PetscInt pStart, pEnd, cStart, cEnd, c, vStart, vEnd, v, eStart, eEnd, e;

DMPlexGetChart(dm, &pStart, &pEnd);
DMPlexGetHeightStratum(dm, 0, &cStart, &cEnd);
DMPlexGetDepthStratum(dm, 1, &eStart, &eEnd);
DMPlexGetDepthStratum(dm, 0, &vStart, &vEnd);
PetscSectionSetChart(s, pStart, pEnd);
/* One dof on each vertex */
for (v = vStart; v < vEnd; ++v) PetscSectionSetDof(s, v, 1);
/* Two dofs on each edge */
for (e = eStart; e < eEnd; ++e) PetscSectionSetDof(s, e, 2);
/* One dof on each cell */
for (c = cStart; c < cEnd; ++c) PetscSectionSetDof(s, c, 1);
PetscSectionSetUp(s);
```

Now a local vector can be created manually using this layout,

```
Vec lv;
PetscInt n;

PetscSectionGetStorageSize(s, &n);
VecSetSizes(lv, n, PETSC_DETERMINE);
VecSetFromOptions(lv);
```

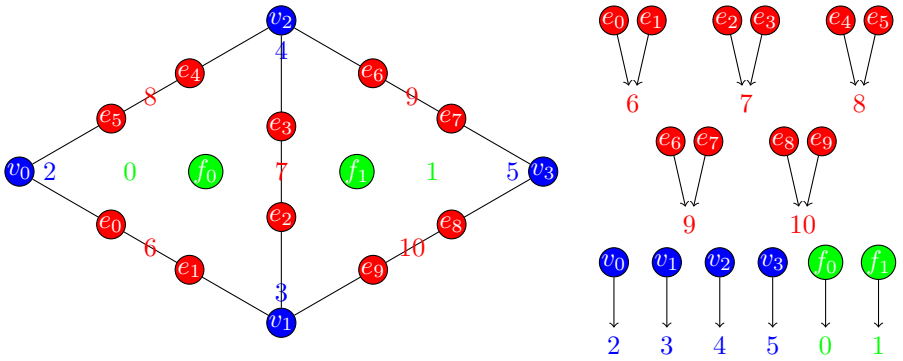


Figure 7.9: Data layout over a triangular doublet mesh using a P_3 continuous Galerkin finite element. The left portion shows the topological data layout, and the right shows the section corresponding to it.

however, it is usually easier to use the `DM` directly, which also provides global vectors,

```
Vec lv, gv;

DMSetDefaultSection(dm, s);
DMGetLocalVector(dm, &lv);
DMGetGlobalVector(dm, &gv);
```

7.2.2 Boundary Conditions

Essential boundary conditions can be represented by marking unknowns which are constrained. When we set the number of dofs on a given mesh point, we also set the number of constraints

```
PetscSectionSetConstraintDof(section, point, numConstraints);
```

and we can also associate constraints with a given field

```
PetscSectionSetFieldConstraintDof(section, point, field, numConstraints);
```

When `PetscSectionSetUp()` is finally called, another section with this information is created. After that, we can let the section know which local dofs on a mesh point are actually constrained. For example, we might fix the x- and z-coorindates of displacement for an elasticity problem,

```
PetscSectionSetConstraintIndices(section, point, [0, 2]);
```

With this information, PETSc can now construct the proper local-to-global mapping.

In the future, PETSc should have an interface which allows the user to prescribe a linear transformation between the local and global basis. This would allow fixing a linear combination of local unknowns instead of just unknowns themselves. Right now, PETSc supports a very limited form of this for hanging node constraints using the `DMForest` class, which represents non-conforming octree meshes. Hanging node constraints are implemented by having the `DM` automatically calculate the constraint matrices mapping from local to global unknowns. This interface could be expanded to allow users to soecify both the connectivity pattern of the constraints and the constraint mapping.

7.3 Using Unstructured Data in PETSc

Most discretizations do not manually construct a data layout, but instead have a rule for assigning dofs to certain parts of a mesh. For example, finite elements assign dofs based upon the dimension of the local space and the continuity requirements for the interpolant. The `PetscFE` object encapsulates this description, as does `PetscFV`, and they can be used to generate a section automatically when `DMGetDefaultSection()` is called for the first time. In SNES ex12, we solve the Poisson equation with optional nonlinear permittivity. A `PetscFE` object is created and inserted into a `PetscDS`, which holds a discretization for each field in the solution. The `PetscDS` allows the `DM` to construct a `PetscSection` for the data layout automatically from the discretization information. In SNES ex62, the same thing is done for the Stokes problem, which has multiple fields.

Checkpoint and visualizing complex data layouts can be difficult, and there is still no standard way to handling this. It seems that current best practice is to use HDF5 (The HDF Group 2015) to store both the original data, and a subsampled version appropriate for visualization. Then the subsampled data can be exposed to a visualization program, such as ParaView (Kitware 2015a), using XDMF metadata (Kitware 2015b). Note here that when checkpointing and visualizing, we want to insert boundary values and output in the canonical basis, rather than the global basis we might use to solve the problem.

PETSc objects can be easily output to HDF5 using builtin viewers. First, we place optional view statements in the code,

```
PetscObjectViewFromOptions((PetscObject) dm, NULL, "-domain_view");
PetscObjectViewFromOptions((PetscObject) u, NULL, "-sol_view");
```

and then on the command line we use the PETSc viewer syntax

```
-domain_view hdf5:sol.h5 -sol_view hdf5:sol.h5::append
```

so that both objects are stored to `sol.h5`. The `DM` output routine also stores a subsampled version of the data, so that we can run the PETSc XMDF generator on the file,

```
${PETSC_DIR}/bin/petsc_gen_xdmf.py sol.h5
```

to produce `sol.xdmf`, which can be visualized with ParaView. The full syntax for a view option is

```
-foo_view <type>:<filename>:<format>:<mode>
```

where ommitted arguments just use the default.

7.4 Parallel Operations for Unstructured Meshes

References

Wikipedia (2015a). *CW complex*. http://en.wikipedia.org/wiki/CW_complex. URL: http://en.wikipedia.org/wiki/CW_complex.

Hatcher, Allen (2002). *Algebraic Topology*. Cambridge University Press.

Wikipedia (2015b). *Hasse Diagram*. http://en.wikipedia.org/wiki/Hasse_diagram. URL: http://en.wikipedia.org/wiki/Hasse_diagram.

Knepley, Matthew G. and Dmitry A. Karpeev (2009). "Mesh Algorithms for PDE with Sieve I: Mesh Distribution". In: *Scientific Programming* 17.3. http://arxiv.org/abs/0908.4427, pp. 215–230. DOI: 10.3233/SPR-2009-0249. URL: http://arxiv.org/abs/0908.4427.

Shewchuk, Jonathan R. (May 1996). "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator". In: *Applied Computational Geometry: Towards Geometric Engineering*. Ed. by Ming C. Lin and Dinesh Manocha. Vol. 1148. Lecture Notes in Computer Science. From the First ACM Workshop on Applied Computational Geometry. Springer-Verlag, pp. 203–222.

Shewchuk, J. (2005). *Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. http://www-2.cs.cmu.edu/~quake/triangle.html.

Si, Hang (Feb. 2015). "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator". In: *ACM Trans. on Mathematical Software* 41.2. DOI: 10.1145/2629697.

— (2005). *TetGen: A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*. http://tetgen.berlios.de.

Rokos, Georgios and Gerard Gorman (2013). "PRAgMaTIc - Parallel Anisotropic Adaptive Mesh Toolkit". In: *Facing the Multicore-Challenge III*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Vol. 7686. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 143–144. ISBN: 978-3-642-35892-0. DOI: 10.1007/978-3-642-35893-7_22.

The HDF Group (2015). *The HDF5 Homepage*. https://www.hdfgroup.org/HDF5/.

Kitware (2015a). *The ParaView Homepage*. http://www.paraview.org/.

— (2015b). *The Extensible Data Model and Format (XDMF)*. http://www.xdmf.org/.

Chapter 8

Simple Finite Elements

8.1 Bases

Consider the space of linear polynomials P_1 on the 2D simplex S. The monomial basis for P_1 is just $\{1, x, y\}$, however this is not the most convenient basis for computing. We often want an interpolatory basis from which we can read off the value at a given point. If our simplex has vertices $(-1,-1)$–$(1,-1)$–$(-1,1)$ which we label v_i, then the basis

$$\phi_0 = -\frac{x+y}{2}, \tag{8.1}$$

$$\phi_1 = \frac{1+x}{2}, \tag{8.2}$$

$$\phi_2 = \frac{1+y}{2}, \tag{8.3}$$

has the property that ϕ_i is 1 at v_i and zero at the other vertices, which we can express as

$$\phi_i(v_j) = \delta_{ij}. \tag{8.4}$$

We would like a basis for the dual space P_1' of linear functionals on P_1, $\{\psi_i\}$, with a diagonal Vandermonde matrix,

$$\psi_i(\phi_j) = \delta_{ij}. \tag{8.5}$$

We can use Eq. 8.4 to see that point evaluation functionals at the vertices will give us exactly what we want,

$$\psi_i(f) = \int_S \delta(\vec{x} - v_i) f(\vec{x}) \tag{8.6}$$

so that

$$\psi_i(\phi_j) = \phi_j(v_i) = \delta_{ij}. \tag{8.7}$$

However, we could instead use elements of P_1 itself since by the Riesz Representation Theorem (Fréchet 1907; Riesz 1907; Riesz 1909), we have an identification of the dual space with the original space. We can take a generic linear polynomial with three free parameters and require conjugacy, Eq. 8.5. The first equation is

$$1 = \int_S \phi_0 \cdot (ax + by + c) \tag{8.8}$$

$$= -\frac{1}{3}a - \frac{1}{3}b + \frac{2}{3}c. \tag{8.9}$$

The other integrals are

$$0 = \int_S \phi_1 \cdot (ax + by + c) \tag{8.10}$$

$$= -\frac{1}{3}b + \frac{2}{3}c, \tag{8.11}$$

and

$$0 = \int_S \phi_2 \cdot (ax + by + c) \tag{8.12}$$

$$= -\frac{1}{3}a + \frac{2}{3}c. \tag{8.13}$$

We can solve this linear system

$$\frac{1}{3}\begin{pmatrix} -1 & -1 & 2 \\ 0 & -1 & 2 \\ -1 & 0 & 2 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \tag{8.14}$$

which gives the polynomial

$$\psi_0 = -x - y - \frac{1}{2}. \tag{8.15}$$

A similar thing can be done for the other functions by solving with rhs $(0, 1, 0)$ and $(0, 0, 1)$, giving

$$\psi_1 = x + \frac{1}{2}, \tag{8.16}$$

$$\psi_2 = y + \frac{1}{2}. \tag{8.17}$$

An alternative way to obtain these functions is to use the Riesz map, that is to solve

$$-\Delta u_i = \psi_i. \tag{8.18}$$

From before, we have the discretization of Δ in the P_1 basis, and solving with the first basis functional as the rhs forcing,

$$\begin{pmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}. \tag{8.19}$$

The system is singular, so we make the rhs consistent by removing the component in the null space,

$$\begin{pmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ -1 \end{pmatrix}. \tag{8.20}$$

We can do the same with the other functionals, and we obtain exactly the same basis as before,

$$\begin{pmatrix} \frac{3}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \end{pmatrix} \Longrightarrow \frac{3}{2}\phi_0(x,y) - \frac{1}{2}\phi_1(x,y) - \frac{1}{2}\phi_2(x,y) \tag{8.21}$$

$$= -x - y - 1/2, \tag{8.22}$$

$$\begin{pmatrix} -\frac{1}{2} \\ \frac{3}{2} \\ -\frac{1}{2} \end{pmatrix} \Longrightarrow \frac{3}{2}\phi_0(x,y) - \frac{1}{2}\phi_1(x,y) - \frac{1}{2}\phi_2(x,y) \tag{8.23}$$

$$= x + 1/2, \tag{8.24}$$

$$\begin{pmatrix} -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{3}{2} \end{pmatrix} \Longrightarrow \frac{3}{2}\phi_0(x,y) - \frac{1}{2}\phi_1(x,y) - \frac{1}{2}\phi_2(x,y) \tag{8.25}$$

$$= y + 1/2. \tag{8.26}$$

When we represent the dual basis in our code, we can make use of the Riesz-Markov-Kakutani Representation Theorem (Riesz 1909; Markov 1938; Kakutani 1941; Wikipedia 2015) which says that any positive linear functional ψ on $C_c(X)$, the space of continuous compactly supported complex-valued functions on a locally compact Hausdorff space X,there is a unique regular Borel measure μ on X such that

$$\psi(f) = \int_X f(x) d\mu(x). \tag{8.27}$$

Since we are operating in the purely discrete world of the computer, we will represent these measures by a quadrature rule

$$\int_X f(x) d\mu(x) \approx \sum_q w_q f(x_q). \tag{8.28}$$

Thus our dual space bases may all be stored internally as sets of quadrature rules supported on a cell and its boundary.

8.2 Evaluating Residuals

The evaluation of a residual, or Jacobian, for most discretizations has the following general form:

- Traverse the mesh, picking out pieces which in general overlap,
- Extract some values from the solution vector, associated with this piece,
- Calculate some values for the piece, and
- Insert these values into the residual vector

DMPlex separates these different concerns by passing sets of points, which are just integers, from mesh traversal routines to data extraction routines and back. In this way, the PetscSection which structures the data inside a Vec does not need to know anything about the mesh inside a DMPlex (Knepley et al. 2013).

The most common mesh traversal is the transitive closure of a point, which is exactly the transitive closure of a point in the DAG using the covering relation. Note that this closure can be calculated orienting the arrows in either direction. For example, in a finite element calculation, we calculate an integral over the closure of each element, and then sum up the contributions to the basis function coefficients. The closure of the element can be expressed discretely as the transitive closure of the element point in the mesh DAG, where each point also has an orientation. Then we can retrieve the data using PetscSection methods,

```
PetscScalar *a;
PetscInt numPoints, *points = PETSC_NULL, p;

VecGetArray(u, &a);
DMPlexGetTransitiveClosure(dm, cell, PETSC_TRUE, &numPoints, &points);
for (p = 0; p < numPoints*2; p += 2) {
  PetscInt dof, off, d;

  PetscSectionGetDof(section, points[p], &dof);
  PetscSectionGetOffset(section, points[p], &off);
  for (d = 0; d < dof; ++d) {
    myfunc(a[off+d]);
  }
}
DMPlexRestoreTransitiveClosure(dm, cell, PETSC_TRUE, &numPoints, &points);
VecRestoreArray(u, &a);
```

Note that the closure operation return both points and and orientation for each point. We ignore the orientations here since we are just retrieving data on each point, and the closure operation itself has used the orientation information to properly order the points. This operation is so common, that we have built a convenience method around it which returns the values in a contiguous array, correctly taking into account the orientations of various mesh points,

```
const PetscScalar *values;
PetscInt csize;

DMPlexVecGetClosure(dm, section, u, cell, &csize, &values);
/* Do integral in quadrature loop */
DMPlexVecRestoreClosure(dm, section, u, cell, &csize, &values);
DMPlexVecSetClosure(dm, section, residual, cell, &r, ADD_VALUES);
```

A simple example of this kind of calculation is in `DMPlexComputeL2Diff()`. Note that there is no restriction on the type of cell or dimension of the mesh in the code above, so it will work for polyhedral cells, hybrid meshes, and meshes of any dimension, without change. We can also reverse the covering relation, so that the code works for finite volume methods where we want the data from neighboring cells for each face

```
PetscScalar *a;
PetscInt points[2*2], numPoints, p, dofA, offA, dofB, offB;

VecGetArray(u, &a);
DMPlexGetTransitiveClosure(dm, cell, PETSC_FALSE, &numPoints, &points);
assert(numPoints == 2);
PetscSectionGetDof(section, points[0*2], &dofA);
PetscSectionGetDof(section, points[1*2], &dofB);
assert(dofA == dofB);
PetscSectionGetOffset(section, points[0*2], &offA);
PetscSectionGetOffset(section, points[1*2], &offB);
myfunc(&a[offA], &a[offB]);
VecRestoreArray(u, &a);
```

This kind of calculation is used in TS ex11, which solves hyperbolic problems such as advection or the shallow water equations.

8.3 MMS

References

Fréchet, Maurice René (1907). “Sur les ensembles de fonctions et les opérations linéaires”. In: *C. R. Acad. Sci. Paris* 144, pp. 1414–1416.

Riesz, Frigyes (1907). “Sur une espéce de géométrie analytique des systémes de fonctions sommables”. In: *C. R. Acad. Sci. Paris* 144, pp. 1409–1411.

— (1909). “Sur les opérations fonctionnelles linéaires”. In: *C. R. Acad. Sci. Paris* 149, pp. 974–977.

Markov, Andrey (1938). “On mean values and exterior densities”. In: *Rec. Math. Moscou* 4, pp. 165–190.

Kakutani, Shizuo (1941). "Concrete representation of abstract (M)-spaces. (A characterization of the space of continuous functions.)" In: *Ann. of Math.* 2.42, pp. 994–1024. DOI: 10.2307/1968778.

Wikipedia (2015). *Riesz-Markov-Kakutani Representation Theorem*. http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani_representation_theorem. URL: http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani_representation_theorem.

Knepley, M. G., J. Brown, K. Rupp, and B. F. Smith (Sept. 2013). "Achieving High Performance with Unified Residual Evaluation". In: *ArXiv e-prints*. arXiv: 1309.1204 [cs.MS].

Chapter 9

Performance Modeling

Without a model, performance measurements are meaningless!

— Matthew G. Knepley

Physical models are essential to our understanding of reality. They allow us to subsume a huge amount of experimental data into an understandable framework, which we use for validation and predicition. In the same way, performance models are required in order to make sense of the mass of performance data. For example, if we measure the time it takes for a code to run, is that fast or slow? Can it be improved? Will it run in the same time for a related problem? These are all questions which cannot be answered without a model of the performance.

When choosing metrics, its always best to start with a goal, and then explain how to achieve that goal. For example, if the goal is to "Run a particular problem as fast as possible in parallel", then strong scaling is a good metric. It shows you how much time improvement you get by adding more processors. However, suppose that you insert redundant work into my solver which is perfectly parallel, such as recalculation of coefficients which can be stored. Now I can show perfect strong scaling, but this is not the most *efficient* way to implement this solver. In the same way, I can add more streaming operations to make my computation bandwidth constrained and achieve the bandwidth peak. Thus, we should keep in mind that any performance metric, with the possible exception of time to solution, can be *gamed*, in the sense that it can be maximized at the expense of another desirable feature. How can I undercover this kind of underhanded benchmarking, and more importantly, how can I know when my algorithm is "improvable"? You must appeal to performance models, which demonstrates their absolute necessity in benchmarking. In addition, this makes it important to use several complementary measures of code performance and accuracy, and to express the tradeoffs visually if possible.

9.1 The STREAM Benchmark

STREAM is a simple benchmark program measuring sustainable memory bandwidth, available from http://www.cs.virginia.edu/stream/. It does this by performing a collection of vector operations, such as Triad (WAXPY): $\mathbf{w} = \mathbf{y} + \alpha\mathbf{x}$, for which the datasets outstrip cache size. In Table 9.1, we show some representative STREAM numbers, along with the maximum flop rate for the triad operation as a percentage of peak performance. Clearly, for these vector operations we cannot hope to achieve much at all of peak performance, and this trend will prove robust throughout numerical computing.

Machine	Peak (MF/s)	Triad (MB/s)	MF/MW	Eq. MF/s
Matt's Laptop	1700	1122.4	12.1	93.5 (5.5%)
Intel Core2 Quad	38400	5312.0	57.8	442.7 (1.2%)
Tesla 1060C	984000	102000.0*	77.2	8500.0 (0.8%)

Table 9.1: Bandwidth limited machine performance

The STREAM benchmark can be run from PETSc using the make system,

```
cd ${PETSC_DIR}
make NPMAX=4 streams
```

where 4 is replaced by the number of cores on the machine in question. This, however, can produce substandard results on multicore machines due to infelicitous mapping of memory to different sockets. This can be alleviated using a flag for the MPI launcher (below we use MPICH installed by PETSc),

```
make NPMAX=4 MPIEXEC="${PETSC_DIR}/${PETSC_ARCH}/bin/mpiexec␣--bind-to␣socket" streams
```

and the OpenMPI equivalent is `--bind-to-socket`.

9.2 Building a Model

A performance model should include measures such as

- computation,
- memory usage,
- communication,
- bandwidth,
- achievable concurrency,

and perhaps others. It allows us both to verify the behavior of a particular implementation or architecture, and also predict the behavior for a new machine or related problem. In this class, we will mainly be interested in the tradeoff between computation and communication, which controls the performance of

most of scientific computing. Our key performance indicator, which we will call the *arithmetic intensity* β, will be the ratio of flops executed to bytes transferred. We will designate the unit $\frac{\mathrm{F}}{\mathrm{B}}$ as the *Keyes*, denoted Ky. The balance factor is a characteristic of the implementation of an algorithm.

Let us begin with an implementation A of an algorithm $\mathcal{A}$. Suppose that we are running our implementation on a machine with peak flop rate r_{peak}. The bandwidth required to run A at this peak rate, b_{req}, is given by

$$b_{\text{req}} = \frac{r_{\text{peak}}}{\beta}. \tag{9.1}$$

It could be that our machine bandwidth b_{peak} is less than b_{req}, which means we cannot achieve the peak flop rate. This is the case for almost all numerical computing today, with the exception of BLAS 3 computations such as DGEMM. Using the peak bandwidth b_{peak}, we can get the maximum flop rate r_{max} for an algorithm

$$r_{\text{max}} = \beta b_{\text{peak}} \tag{9.2}$$

We will first construct a model for a simple BLAS operation, AXPY, which adds two vectors element-wise and updates an input vector with the sum,

$$\vec{y} \leftarrow \alpha\vec{x} + \vec{y}. \tag{9.3}$$

If the vectors are both of length N and store b-byte numbers, then this operation performs $2N$ flops and accesses $(3N + 1)b$ bytes of data, since N bytes are retrieved for x and y, 1 byte for α, and N bytes are written back to y. Here we simplify the problem by not distinguishing loads and stores. The balance factor for this computation is then

$$\beta = \frac{2N\mathrm{F}}{(3N+1)b\mathrm{B}} \approx \frac{2}{3b}\mathrm{Ky} \tag{9.4}$$

which for double precision numbers is $\frac{1}{12}$. For my Mac Air laptop, $r_{\text{peak}} = 1700$ MF/s, implying $b_{\text{req}} = 2550b$ MB/s which is much greater than b_{peak}. The peak bandwidth $b_{\text{peak}} = 1122$ MB/s on the Air implies that $r_{\text{max}} = \frac{748}{b}$ MF/s, 93.5 MF/s which is 5.5% of r_{peak}. This dismal number is all too common in sparse algorithms, where most computations today run at several percent of peak floating point performance.

9.3 The Roofline Model

The Roofline model (Williams, Waterman, and Patterson 2009; Williams and et. al. n.d.) is a performance analysis framework which has the stated purpose to "integrate in-core performance, memory bandwidth, and locality into a single readily understandable performance figure", and is well explained in this talk (Williams and Patterson 2008). However, all locality information is

contained in the *achievable memory bandwidth*, so it is not as fine-grained as the external memory model (Aggarwal and Vitter 1988) which we will see in Section 9.5. The "roof" in the model is the bound on computation performance we have from Section 9.2,

$$r = \min \begin{cases} r_{\text{peak}} \\ b_{\text{peak}}\beta \end{cases} \tag{9.5}$$

and when plotted it looks like the side of a roof on a house, as in Fig 9.1. The flat section at the top is the bound on peak computational performance coming from the processor itself, and as Fig 9.1 demonstrates, this bound is sensitive to optimizations such as use of SIMD instructions, or vectorization, fused instructions such a fused-multiply-add (FMA), and instruction level parallelism. The sloping part of the graph is the performance limited by memory bandwidth. On a log-log plot, this has unit slope (since the exponent of β is unity) and the intercept is determined by b_{peak}. Thus optimizations such as prefetching, unit stride, and tiling will translate this line.

Like any performance measure, however, this can be "gamed" in the sense that we can produce an algorithm which appears to be optimal by this measure, but is in fact unusable. For example, suppose that we have bandwidth-limited algorithm to solve a linear system, such as classical multigrid for the Poisson equation. We could add many small DGEMM calls to shift the arithmetic intensity and obtain almost r_{peak}. Since we nowhere compare algorithms in terms of time-to-solution, it is difficult to see that we are artificially inflating the computational demands. Alternatively, we may have an algorithm with a larger β which is difficult to optimize, such as a Q_2 finite element assembly and solve for a complex PDE. We can choose to parallelize the computation by vectorizing over basis functions, taking each quadrature point in turn. This vectorizes beautifully and requires very little local memory, but pulls all finite element coefficients from global memory for each quadrature point, rather than a single time. Thus β decreases, but our bandwidth utilization goes up, making the overall algorithm look more efficient, even though it is much slower.

9.4 Sparse Matrix-Vector Product (SpMV)

The Sparse Matrix-Vector Product (SpMV) is today a workhorse of scientific computing. It is a central kernel is iterative linear and nonlinear solvers for PDE, and now for many graph algorithms. An excellent performance analysis of SpMV is given in (Gropp et al. 1999), which we will follow closely. For this simple model, we assume that there are no conflict cache misses, meaning each matrix and vector entry is loaded into cache only once, and no latency for memory references, so each load or store takes a single cycle. The SpMV for a matrix with m rows and nz non-zero elements using AIJ storage being multiplied with V vectors is shown below, annotated with the kind of assembly instructions generated by each statement,

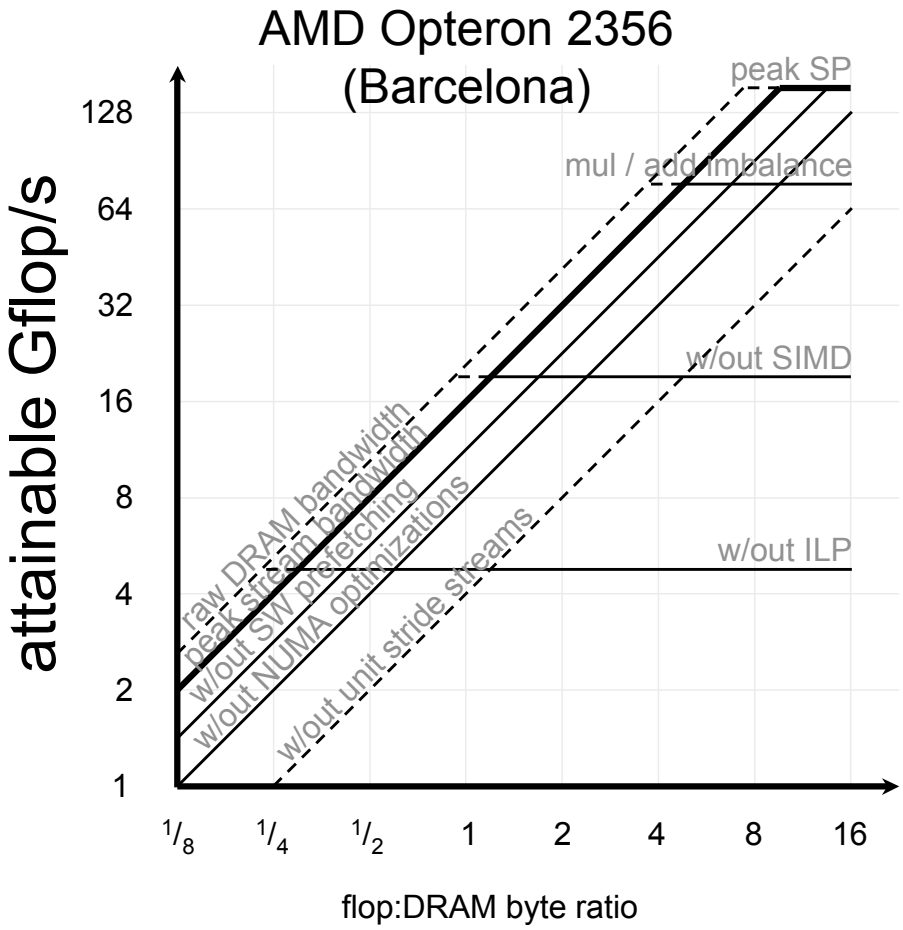


Figure 9.1: A roofline plot for the AMD Opteron 2356, showing the effect of many program optimizations. (Image taken from (Williams and Patterson 2008))

```
for (i = 0, i < m; i++) {
  jrow = ia(i+1) // 1 Of, AT, Ld
  ncol = ia(i+1) - ia(i) // 1 Iop
  Initialize, sum1, ..., sumV // V Ld
  for (j = 0; j < ncol; j++) { // 1 Ld
    fetch ja(jrow), a(jrow),
          x1(ja(jrow)), ..., xV(ja(jrow)) // 1 Of, V+2 AT, V+2 Ld
    do V fmadd (floating multiply add) // 2V Fop
    jrow++
  } // 1 Iop, 1 Br
  Store sum1, ..., sumV in
    y1(i), ..., yV(i) // 1 Of, V AT, V St
} // 1 Iop, 1 Br
```

where `AT` is address translation, `Br` is branch, `Iop` is integer operation, `Fop` is floating-point operation, `Of` is offset calculation, `Ld` is load, and `St` is store. In each outer loop, we load the column offset and store V vector elements. In each inner loop, we need to transfer one integer (`ja` column array) and $V+1$ doubles (one matrix element and V vector elements) and we do V floating-point multiply-add (fmadd) operations or $2V$ flops. We therefore communicate

$$m\left(b_{\text{int}} + Vb_{\text{double}}\right) + nz\left(b_{\text{int}} + (V+1)b_{\text{double}}\right) \tag{9.6}$$
$$= 4(m+nz) + 8(mV + nz(V+1)) \tag{9.7}$$
$$\approx 4(m+nz) + 8(2mV + nz) \tag{9.8}$$

where the third line follows from the perfect cache assumption (since elements of the vectors will not be loaded more than once), so that

$$\beta = \frac{2nzV}{4(m+nz) + 8(2mV+nz)}\text{Ky} = \frac{1}{\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V}}\text{Ky}. \tag{9.9}$$

From Eq. 9.2, the peak performance we can expect is

$$\frac{b_{\text{peak}}}{\left(8 + \frac{2}{V}\right)\alpha^{-1} + \frac{6}{V}}\text{F/s}. \tag{9.10}$$

where α is the average row occupancy of the matrix.

We can look at the implications of this model. For a single matvec with 3D FD Poisson, Matt's laptop can achieve at most

$$\frac{1122.4\text{ MB/s}}{(8+2)\frac{1}{7} + 6\text{B/F}} = 151\text{ MF/s}, \tag{9.11}$$

which is a dismal 8.8% of peak. With a box stencil, the row occupancy is now 27, giving 176 MF/s. If we use 4 vectors instead of 1, we can achieve 414 MF/s, so we see the larger rows have much less effect than using more vectors. We can

ask how many vector would we need to use in order to achieve theoretical peak on this machine,

$$\beta(V) = \frac{r_{\text{peak}}}{b_{\text{peak}}} \tag{9.12}$$

$$\frac{1}{\left(8 + \frac{2}{V}\right)\frac{1}{7} + \frac{6}{V}} = \frac{1700}{1122.4}\text{Ky} \tag{9.13}$$

$$\frac{44}{7V} = -0.48\text{Ky} \tag{9.14}$$

$$\tag{9.15}$$

so it is impossible for this machine. The peak attainable performance with this matrix, $V \to \infty$, is 982 MF/s. However, this shows the limits of the model, since operation issue limitations would dominate long before we reached this limit. We could also improve performance by blocking since it would bring in fewer column offsets and indices, altering the coefficients on the $1/V$ terms, but not the ultimate performance limit given by the 8.

In order to overcome the inherent limitations for performance in SpMV, we would have to adopt an approach having N^k computation for N data where $k > 1$, such as unassembled operator application or nonlinear evaluation. When choosing a method, we must consider tradeoffs betweem storage, bandwidth, and cycles. The assembled operator action (SpMV) trades cycles for memory bandwidth and storage. Whereas unassembled operator action trades bandwidth and storage for cycles in application. In fact, for high order spectral elements, storage is impossible. Partial assembly of the matrix could give even finer control over these tradeoffs, but perhaps introduce new parallel costs, such as load balance.

9.5 Serial Computation

We will not develop models of serial performance in this class. However, they incorporate two main effects which help explain performance on modern processors. First, and usually most important for scientific computing, is *vectorization*. The peak flop rate r_{peak} on modern CPUs is attained through the use of a SIMD multiply-accumulate instruction on special 256-bit registers. SIMD MAC operates in the form of 8 simultaneous operations (4 adds and 4 multiplies):

$$c_1 = c_1 + a_1 * b_1 \tag{9.16}$$

$$c_2 = c_2 + a_2 * b_2 \tag{9.17}$$

$$c_3 = c_3 + a_3 * b_3 \tag{9.18}$$

$$c_4 = c_4 + a_4 * b_4 \tag{9.19}$$

You will degrade performance from the peak if any operations are missing. In the worst case, you are reduced to 12.5% efficiency if your algorithm performs naive

summation or products. In addition, memory alignment is also crucial when using these instructions (AVX on Intel processors). The instructions used to load and store from the 256-bit registers throw very costly alignment exceptions when the data is not stored in memory on 32 byte (256 bit) boundaries.

The other effects are differences in behavior for different types of memory, such as fast local memory, slower global memory, or even slower remote memory on another process. When retrieving data from memory, or over a network, not only are there bandwidth limits, but there is a latency associated with the request and possibly an overhead for each transaction. These factors are incorporated into the LogP model of parallel computation (Culler et al. 1993). In addition, we may have different kinds of memory, for example fast cache memory and slow main memory. The external memory model (Aggarwal and Vitter 1988) models the memory system as a small fast memory consisting of Z words, and an infinite main memory. Computations can only be performed on data in fast memory, and data can only be transfered from main to fast memory in chunks of L words. Then an algorithm analysis will try to predict $Q_{Z,L}(N)$, the number of transfers between main and fast memory for a computation $W(N)$ of size N, and $\beta(N) = W(N)/Q_{Z,L}(N)$ (Czechowski et al. 2011).

9.6 Problems

Problem IX.1 The following pseudocode is a naive implementation of a dense matrix-vector multiplication (assuming matrix dimension of $N \times N$, vector dimension of N).

C

```
for (i=0; i < N; ++i) {
  double sum = 0.0;
  for (j=0; j < N; ++j) {
    sum += a[i*N+j]*b[j];
  }
  c[i] = sum;
}
```

Fortran

```
for(i=1,N)
  sum = 0
  for(j=1,N)
    sum = sum + a(i,j)*b(j)

  c(i) = sum
```

Based on the above code, answer the following questions:

1. Count the total number of floating point operations, in terms of N. The unit will be a FLOP, abbreviated with F.
2. Count the total number of bytes transferred to/from memory if each floating point number is 8 bytes, abbreviated with B, in terms of N.
3. Compute the arithmetic intensity, meaning the ratio of floating point operations to total bytes transferred, and approximate for large N. (This will give you a value in F/B, where F is a FLOP and B is a byte).
4. If a processor flop rate is 2 GF/s, and memory bandwidth is 8 GB/s, is the program flop rate limited, or memory bandwidth limited?

5. What fraction of peak performance do you estimate can be obtained?

Problem IX.2 Consider the Gram-Schmidt Orthogonalization process. Starting with a set of vectors $\{v_i\}$, create a set of orthonormal vectors $\{n_i\}$.

$$n_1 = \frac{v_1}{||v_1||} \tag{9.20}$$

$$n_2 = \frac{w_2}{||w_2||} \text{ where } w_2 = v_2 - (n_1 \cdot v_2)n_1 \tag{9.21}$$

$$n_k = \frac{w_k}{||w_k||} \text{ where } w_k = v_k - \sum_{j<k}(n_j \cdot v_k)n_j \tag{9.22}$$

What is

1. the balance factor β for this algorithm?
2. the bandwidth required to run at peak (b_{req}) on your computer?
3. the maximum achievable flop rate ($r_{\max}$) on your computer?

Extra Credit Can this algorithm be improved?

Problem IX.3 Change the performance model for sparse matrix-vector multiplication (SpMV) so that the loads from memory are uncached. How does the dependence on row occupancy change?

Problem IX.4 Run the STREAMS benchmark on your personal computer and graph the results as a function of core count. What do the results tell you about the architecture of your machine?

References

Williams, Samuel, Andrew Waterman, and David Patterson (2009). "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4, pp. 65–76.

Williams, Samuel and et. al. (n.d.). *Roofline Performance Model*. http://crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/roofline/. URL: http://crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/roofline/.

Williams, Samuel and David Patterson (2008). *Roofline Performance Model*. http://crd.lbl.gov/assets/pubs_presos/parlab08-roofline-talk.pdf. ParLab Summer Retreat. URL: http://crd.lbl.gov/assets/pubs_presos/parlab08-roofline-talk.pdf.

Aggarwal, Alok and Jeffrey Vitter (1988). "The input/output complexity of sorting and related problems". In: *Communications of the ACM* 31.9, pp. 1116–1127.

Gropp, William D., Dinesh K. Kaushik, David E. Keyes, and Barry F. Smith (1999). "Towards Realistic Performance Bounds for Implicit CFD codes". In: *Proceedings of Parallel CFD'99*. Elsevier. URL: http://www.cs.odu.edu/~keyes/papers/pcfd99_gkks.pdf.

Culler, David, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken (1993). "LogP: Towards a realistic model of parallel computation". In: *SIGPLAN Notices* 28.7, pp. 1–12.

Czechowski, Kenneth, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard Vuduc (May 2011). "Balance principles for algorithm-architecture co-design". In: *Proc. USENIX Wkshp. Hot Topics in Parallelism (HotPar)*. Berkeley, CA, USA. URL: http://www.usenix.org/events/hotpar11/tech/final_files/Czechowski.pdf.

Chapter 10

Linear Solvers

A linear solver is an operation which given the equation

$$Ax = b \tag{10.1}$$

where the matrix A and vector b are specified, will return a vector $\tilde{x}$ which is close to x, usually in a normwise sense. We call the measure of proximity a *tolerance*, and distinguish an absolute tolerance where the error is smaller than a fixed size

$$\|x - \tilde{x}\| < \epsilon_a, \tag{10.2}$$

from a relative tolerance

$$\|x - \tilde{x}\| < \epsilon_r \|x\|. \tag{10.3}$$

In practice, we cannot evaluate the error, so the residual is used instead

$$\|A(x - \tilde{x})\| = \|b - A\tilde{x}\|. \tag{10.4}$$

We usually divide these solvers into two classes: *direct* and *iterative*. Direct solvers are supposed to reach a given tolerance using a fixed amount of computation determined by the problem size, and iterative methods take an indeterminant number of iterates which improve the approximate solution until the tolerance is satisfied. The prototypical direct solver is Gaussian elimination (or LU factorization), and the Gauss-Seidel iteration can stand for a broad class of iterative solvers. However, this definition is somewhat fuzzy. The accuracy of direct methods can be limited by the condition number of A, $\kappa(A) = \|A\|\|A^{-1}\|$, and often the solutions are improved by iterative refinement, discussed in Section 10.1.2. On the other hand, the iterative multigrid algorithm from Section 10.3 can be considered a direct solver for the Laplace equation since the amount of computing can be explicitly bounded and accuracy quantified.

10.1 Direct Solvers

10.1.1 Schur complement

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{10.5}$$

We can solve the first equation for u in terms of v,

$$u = f - b/av, \tag{10.6}$$

plug into the second equation to get v

$$c(f - b/av) + dv = g, \tag{10.7}$$

$$\left(d - \frac{cb}{a}\right) v = g - cf, \tag{10.8}$$

and then plug in v to recover u. If we let these entries be matrices, rather than scalar values, we have

$$\left(D - CA^{-1}B\right) v = g - Cf, \tag{10.9}$$

where the matrix $S = D - CA^{-1}B$ is called the *Schur Complement*.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & b/a \\ c & d \end{pmatrix} \tag{10.10}$$

$$= \begin{pmatrix} a & 0 \\ c & d - cb/a \end{pmatrix} \begin{pmatrix} 1 & b/a \\ 0 & 1 \end{pmatrix} \tag{10.11}$$

$$\begin{pmatrix} \alpha & b^T \\ c & D \end{pmatrix} = \begin{pmatrix} \alpha & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & b^T/\alpha \\ c & D \end{pmatrix} \tag{10.12}$$

$$= \begin{pmatrix} \alpha & 0 \\ c & D - cb^T/\alpha \end{pmatrix} \begin{pmatrix} 1 & b/\alpha \\ 0 & I \end{pmatrix} \tag{10.13}$$

Work: $b/\alpha \longrightarrow N-1$ flops $D - \frac{1}{a}cb^T \longrightarrow 3(N-1)^2$ Total: $\sum_{k=1}^{n} k^2 = n(n+1)(2n+1)/6$ so to leading order it is $\mathcal{O}(n^3)$. Can actually prove that is scales like matrix-matrix multiply (Bunch and Hopcroft 1974).

10.1.2 Iterative Refinement

10.2 Krylov Solvers

(Saad 2003) - Why do you want to do low-order/incomplete PC and MF action?

Figure 10.1: The multiplicative full multigrid iteration.

Why Krylov methods? Both Chebyshev and Krylov methods build polynomial approximations to the solution of linear equations. The Chebyshev method solves a continuous problem for the coefficients, so that they are known without computation, but this requires an estimate for the spectrum of the operator. On the other hand, a Krylov method solves the discrete minimization problem for the coefficients, which requires inner products (as many as k^2 in GMRES), that can severely limit the scalability on machines with appreciable latency, such as massively parallel machines. This points out the central strength of Krylov methods.

We can see that as $\kappa \to \infty$, Krylov methods make no progress, but our PDEs have increasing κ with increasing resolution (for the Laplacian, $\kappa \propto h^{-2}$). Thus, it seems Krylov methods alone cannot be an approriate solver. However, Krylov methods have a signal strength in that they are "spectrally adaptive", meaning that the user does not need to know the spectrum in order to use them effectively. When combined with a method that needs spectral information, such as multigrid, Krylov methods can fill holes left by approximate or inappropriate spectral descriptions.

10.3 Multigrid Methods

10.3.1 V-cycle

10.3.2 Full Multigrid

The Full Multigrid algorithm (FMG) can be thought of as applying a multigrid V-cycle at each level at each level of the multigrid hierarchy, and then interpolating to the next finer grid. You can see the structure in Figure 10.1. This may seem like a lot more work than a simple V-cycle, but we will show that it is, in fact, just a small percentage more.

Since FMG is just made up of V-cycles and interpolation operations, at each level the cost is linear in the number of unknowns. Thus if the cost of the final V-cycle is C_V, then the cost of the next coarsest cycle is $\frac{1}{2^d} C_V$, and the entire

cost C_{FMG} is given by

$$C_{FMG} = \left(1 + \frac{1}{2^d} + \frac{1}{2^{2d}} + \dots\right) C_V \tag{10.14}$$

$$= \sum_{n=0}^{\infty} \frac{1}{2^{nd}} C_V \tag{10.15}$$

$$= \frac{2^d}{2^d - 1} C_V \tag{10.16}$$

$$= \left(\frac{2^d}{2^d - 1}\right)^2 C_v. \tag{10.17}$$

In one dimension, the cost of FMG is twice that of a V-cycle, but in three dimensions it is only $\frac{8}{7}$ of a single V-cycle, and only about 30% more expensive than a single two-level iteration.

Full multigrid can solve to discretization error with just one iteration. In order to understand this argument, we must have a model of discretization error E_d. We suppose that

$$\|x - x_h\| < Ch^\alpha \tag{10.18}$$

where $\alpha > 0$. For example, our simple FD method has error which is in $\mathcal{O}(h^2)$, so $\alpha = 2$ and it is a second order method. This is also true for the P_1 linear finite element method. Suppose that we have just finished the V-cycle for the grid with $2h$ spacing, which gives the coarse grid correction for the top level grid of spacing h. The error E is bounded by the sum of the discretization error and the algebraic error E_a, which is the error due to a finite precision iterative solve. We will suppose that our iterative tolerance is such that our algebraic error is within a constant factor $r < 1$ of our discretization error, since we are free to choose any tolerance. This means that

$$E \leq E_d + E_a = (1 + r)C(2h)^\alpha, \tag{10.19}$$

where the reduction factor $r = E_a/E_d$ is the solver tolerance relative to discretization error. Now we perform the final V-cycle for fine grid of spacing h. In order that we make the required reduction r in algebraic error, we need the V-cycle to have error reduction factor η, which satisfies

$$\eta E_a < rCh^\alpha \tag{10.20}$$

$$\eta\left(E - E_d\right) < rCh^\alpha \tag{10.21}$$

$$\eta\left((1 + r)C(2h)^\alpha - Ch^\alpha\right) < rCh^\alpha \tag{10.22}$$

$$\eta\left((1 + r)2^\alpha - 1\right) < r \tag{10.23}$$

$$\eta < \frac{1}{2^\alpha + \frac{2^\alpha - 1}{r}}. \tag{10.24}$$

For our second order method with a reduction factor $r = \frac{1}{2}$, we require that $\eta < \frac{1}{10}$. This V-cycle error reduction can achieved for tuned multigrid iteration (Trottenberg, Oosterlee, and Schüller 2001).

Let us try and verify this remarkable property of FMG experimentally. We can use SNES `ex5` if we turn off the nonlinearity by using $\lambda = 0.0$. We can limit ourselves to a single Newton iterate, `-snes_max_it 1`, and thus standard V-cycle MG with GMRES

```
./ex5 -mms 1 -par 0.0 -da_refine 3 -snes_type newtonls -snes_max_it 1 -ksp_rtol 1e-10
  -pc_type mg -snes_monitor_short -ksp_monitor_short
```

gives

```
0 SNES Function norm 0.0287773
  0 KSP Residual norm 0.793727
  1 KSP Residual norm 0.00047526
  2 KSP Residual norm 4.18007e-06
  3 KSP Residual norm 1.1668e-07
  4 KSP Residual norm 3.25952e-09
  5 KSP Residual norm 7.274e-11
1 SNES Function norm 2.251e-10
N: 625 error l2 1.21529e-13 inf 9.53484e-12
```

and it changes little if we refine six more times

```
0 SNES Function norm 0.000455131
  0 KSP Residual norm 50.6842
  1 KSP Residual norm 0.00618427
  2 KSP Residual norm 9.87833e-07
  3 KSP Residual norm 2.99517e-09
1 SNES Function norm 2.83358e-09
N: 2362369 error l2 1.28677e-15 inf 7.68693e-12
```

If instead we run with FMG,

```
./ex5 -mms 1 -par 0.0 -da_refine 3 -snes_type newtonls -snes_max_it 1 -ksp_rtol 1e-10
  -pc_type mg -pc_mg_type full -snes_monitor_short -ksp_monitor_short
```

we do not seem to see the convergence acceleration

```
0 SNES Function norm 0.0287773
  0 KSP Residual norm 0.799687
  1 KSP Residual norm 6.95292e-05
  2 KSP Residual norm 1.50836e-06
  3 KSP Residual norm 2.62524e-08
  4 KSP Residual norm 6.184e-10
  5 KSP Residual norm 1.275e-11
1 SNES Function norm 3.757e-11
N: 625 error l2 2.1428e-14 inf 1.80611e-12
```

although its a little more apparent as we refine,

```
0 SNES Function norm 0.000455131
  0 KSP Residual norm 51.2
  1 KSP Residual norm 2.92416e-06
  2 KSP Residual norm 3.76404e-09
1 SNES Function norm 8.50096e-09
N: 2362369 error l2 1.70304e-15 inf 6.22476e-11
```

In order to untangle this, we need to consider the discretization error inherent in our approximation, since the proof above for FMG is predicated on the solve reaching discretization error rather than some arbitrary residual target. We can do this by limiting the linear solve to a single iterate and looking at the convergence. We cannot use `-ksp_max_it 1` since that causes a failure of the linear iteration, whereas `-ksp_rtol 1e-1` accomplishes the same objective. The script below compares standard GMG with V-cycles to the FMG, using a slightly stronger smoother (5 iterates of Chebyshev/SOR) in order to guarantee the sufficient decrease in error for each V-cycle.

```
#! /usr/bin/env python
import argparse
import subprocess
import numpy as np

parser = argparse.ArgumentParser(
  description = 'CAAM 519 FMG',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
args = parser.parse_args()

sizesA = []
sizesB = []
errorA = []
errorB = []
for k in range(args.kmax):
  options = ['-snes_type', 'newtonls', '-snes_max_it', '1', '-da_refine', str(k),
             '-par', '0.0', '-ksp_atol', '1e-1', '-mms', '1',
             '-pc_type', 'mg', '-pc_mg_type', 'multiplicative',
             '-mg_levels_ksp_max_it', '5']
  cmd = './ex5 '+' '.join(options)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  sizesA.append(int(out[1]))
  errorA.append(float(out[4]))
for k in range(args.kmax):
  options = ['-snes_type', 'newtonls', '-snes_max_it', '1', '-da_refine', str(k),
             '-par', '0.0', '-ksp_atol', '1e-1', '-mms', '1',
             '-pc_type', 'mg', '-pc_mg_type', 'full',
             '-mg_levels_ksp_max_it', '5']
  cmd = './ex5 '+' '.join(options)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  sizesB.append(int(out[1]))
  errorB.append(float(out[4]))
SA = np.array(sizesA)
SB = np.array(sizesB)

from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
loglog(SA, errorA, 'r', SA, 5e-6 * SA ** -0.5, 'r--',
       SB, errorB, 'g', SB, 5e-6 * SB ** -1., 'g--')
```

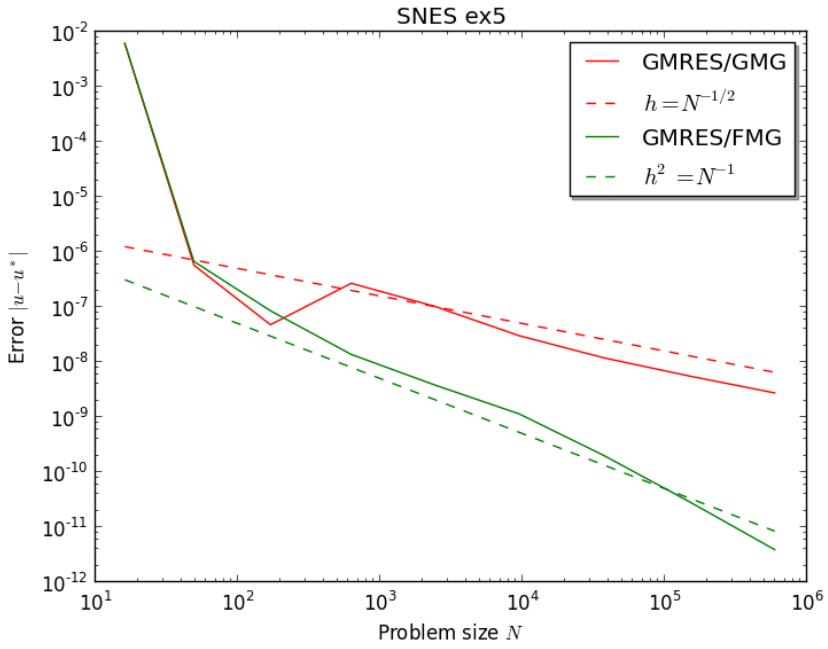


Figure 10.2: Comparison of a single iterate of V-cycle GMG and FMG for the Poisson equation.

```
title('SNES ex5')
xlabel('Problem size $N$')
ylabel('Error $\|u - u^*\|$')
legend(['GMRES/GMG', '$h = N^{-1/2}$', 'GMRES/FMG', '$h^2 = N^{-1}$'],
       'upper right', shadow = True)
if args.save:
  savefig('fmg.png')
else:
  show()
```

We can see in Figure 10.2 that for our simple FD discretization, which has a truncation error in $\mathcal{O}(h^2)$, FMG produces a solution which decreases with the discretization error in just a single iteration. The pure V-cycle implementation gets to only $\mathcal{O}(h)$ error in one iterate.

10.3.3 FAS Multigrid

FAS stands for Full Approximation Storage, and was originally proposed by Achi Brandt as an alternative to defect correction multigrid which computes the entire coarse solution directly rather than a correction to the fine solution.

One of the main reasons for this is it allows the generalization of multigrid to nonlinear equations. Starting with the original fine grid equation

$$F_f(u_f) = b_f, \tag{10.25}$$

we solve a coarse problem

$$F_c(u_c) = b_c, \tag{10.26}$$

and then correct the fine solution using

$$u_f \leftarrow u_f + P\left(u_c - Ru_f\right). \tag{10.27}$$

The question is what should be use for the rhs b_c? In defect correction multigrid, this is the residual restricted to the coarse grid. Here we will use

$$b_c = Rb_f + F_c(Ru_f) - RF_f(u_f). \tag{10.28}$$

If we suppose our equation is linear, meaning $F = A$, then we have

$$\begin{aligned} F_c(u_c) &= Rb_f + F_c(Ru_f) - RF_f(u_f) && (10.29)\\ A_c u_c &= Rb_f + A_c Ru_f - RA_f u_f && (10.30)\\ A_c\left(u_c - Ru_f\right) &= R\left(b_f - A_f u_f\right) && (10.31)\\ A_c \delta u_c &= Rr_f && (10.32) \end{aligned}$$

so that defect correction multigrid and FAS are mathematically equivalent. Note that the manipulation in Eq. 10.31 fails for a nonlinear operator F, which is why FAS is used as the starting point for a multilevel nonlinear solver.

One important property of the coarse grid equation for FAS is that it is satisfied by the restriction of the true fine solution u_f^*,

$$\begin{aligned} F_c(Ru_f^*) &= Rb_f + F_c(Ru_f^*) - RF_f(u_f^*) && (10.33)\\ R\left(F_f(u_f^*) - b_f\right) &= 0 && (10.34)\\ 0 &= 0. && (10.35) \end{aligned}$$

This means that once the fine solution is obtained, the coarse problem will not introduce any change via the update in Eq. 10.27. This is a guiding principle for derivation of coarse grid equation in more sophisticated settings, such as optimization problems.

Superlinear Convergence We can repeat our argument above for the convergence of FMG when using FAS. However, this time we will allow the possibility of superlinear convergence. If we have q-convergence of order greater than

one, we would have

$$\eta E_a^q < rCh^\alpha \tag{10.36}$$
$$\eta \left(E - E_d\right)^q < rCh^\alpha \tag{10.37}$$
$$\eta \left((1+r)C(2h)^\alpha - Ch^\alpha\right)^q < rCh^\alpha \tag{10.38}$$
$$\eta \left((1+r)2^\alpha - 1\right)^q < rC^{1-q}h^{\alpha(1-q)} \tag{10.39}$$
$$\eta < C^{1-q}h^{\alpha(1-q)}\frac{1}{2^\alpha + \frac{2^\alpha - 1}{r}}. \tag{10.40}$$

We can interpret this as saying that η can become progressively bigger as we refine the grid, but this rapidly become of no use. It seems that superlinear algebraic convergence does not help the solve because the algebraic progress outstrips the accuracy of the discretization at the subsequent level, since the discretization error decreases only linearly. We might instead allow the size of the fine grid to increate at a superlinear rate to offset this disparity. Suppose that we have solved the nonlinear problem on a coarse grid of resolution $2h$, and then interpolate to a fine grid of resolution h^q,

$$\eta E_a^q < rCh^{q\alpha} \tag{10.41}$$
$$\eta \left(E - E_d\right)^q < rCh^{q\alpha} \tag{10.42}$$
$$\eta \left((1+r)C(2h)^\alpha - Ch^\alpha\right)^q < rCh^{q\alpha} \tag{10.43}$$
$$\eta \left((1+r)2^\alpha - 1\right)^q < rC^{1-q} \tag{10.44}$$
$$\eta < C^{1-q}\frac{1}{2^\alpha + \frac{2^\alpha - 1}{r}}. \tag{10.45}$$

and thus for $\alpha = 2$ and $r = \frac{1}{2}$, we require that $\eta < \frac{1}{10C^{q-1}}$.

10.3.4 Why does Multigrid fail?

Reductively, there are two failure modes for MG: failure of the smoother to remove high frequency error components and failure of the coarse correction to remove low frequency error components. A strategy for examining these two cases is presented in (Diskin, Thomas, and Mineck 2005). It will be helpful for us to enumerate a more precise list of problems that can arise.

Inaccurate coarse operator Helmholtz example. Use renormalization approach.

Ineffective smoother Helmholtz and MHD examples. Use artifical diffusion/filtering and defect correction. This could also apply to PDE-constrained optimization.

Inaccurate prolongation PDE-constrained optimization example. Use a solve instead.

Smoother misses isolated modes Find an example. Use a Krylov accelerator.

Smoother fails for multiple intervals Stokes example. Use a Schur-complement/distributive smoother.

10.3.5 Algebraic Multigrid

Agglomeration variants of algebraic multigrid (AMG) create subspaces using constant basis functions over sets of cells, which can optionally be smoothed by application of the operator itself. The original system is then projected into this subpsace using Galerkin projection, $P^T AP$. After projection, we can think of the operator as approximating its symbol as the effect of boundary conditions is diluted or removed. Thus, it is important to capture the nullspace of the symbol of the operator in the coarse basis. Since AMG is so often used to solve the Laplacian, most implementations automatically provide the constant vector as a member of the coarse space. We will demonstrate that other systems require additional (near) nullspace vectors.

The balance of momentum for linear elasticity in an isotropic, homogeneous medium can be written

$$\nabla \cdot \sigma = \vec{f} \tag{10.46}$$

$$\nabla \cdot (\lambda \mathrm{Tr}(\varepsilon) I + 2\mu\varepsilon) = \tag{10.47}$$

and is implemented in PETSc SNES ex17. The energy in this system, in the absence of body force $\vec{f}$, is invariant to translations and rotations. Thus, the operator above annihilates the infinitesimal generators of these transformations. We can see the effect of these symmetries by looking at a solve in ex17. We will discretize using Q_1 finite elements on a quadrilateral grid for the unit square. We begin with the default GMRES/ILU(0) solver for a series of regularly refined meshes:

```
> ./ex17 -sol_type elas_quad -simplex 0 -displacement_petscspace_order 1 -dm_refine 5 -snes_converged_reason -ksp_rtol 1e-
  Linear solve converged due to CONVERGED_RTOL iterations 77
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
> ./ex17 -sol_type elas_quad -simplex 0 -displacement_petscspace_order 1 -dm_refine 6 -snes_converged_reason -ksp_rtol 1e-
  Linear solve converged due to CONVERGED_RTOL iterations 167
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
> ./ex17 -sol_type elas_quad -simplex 0 -displacement_petscspace_order 1 -dm_refine 7 -snes_converged_reason -ksp_rtol 1e-
  Linear solve converged due to CONVERGED_RTOL iterations 411
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
```

We can see that the number of iterates increases drastically as the problem size is increased. If we use GAMG, the number of iterates is greatly reduced, but still grows somewhat as the problem size is increased

```
knepley/pylith $:/PETSc3/petsc/petsc-pylith/src/snes/examples/tutorials$ ./ex17 -sol_type elas_quad -simplex 0 -displaceme
  Linear solve converged due to CONVERGED_RTOL iterations 10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
knepley/pylith $:/PETSc3/petsc/petsc-pylith/src/snes/examples/tutorials$ ./ex17 -sol_type elas_quad -simplex 0 -displaceme
  Linear solve converged due to CONVERGED_RTOL iterations 11
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
knepley/pylith $:/PETSc3/petsc/petsc-pylith/src/snes/examples/tutorials$ ./ex17 -sol_type elas_quad -simplex 0 -displaceme
```

```
  Linear solve converged due to CONVERGED_RTOL iterations 13
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
knepley/pylith $:/PETSc3/petsc/petsc-pylith/src/snes/examples/tutorials$ ./ex17 -sol_type elas_quad -simplex 0 -displacement_petscspace_
  Linear solve converged due to CONVERGED_RTOL iterations 15
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
knepley/pylith $:/PETSc3/petsc/petsc-pylith/src/snes/examples/tutorials$ ./ex17 -sol_type elas_quad -simplex 0 -displacement_petscspace_
  Linear solve converged due to CONVERGED_RTOL iterations 16
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
```

References

Bunch, James R and John E Hopcroft (1974). "Triangular factorization and inversion by fast matrix multiplication". In: *Mathematics of Computation* 28.125, pp. 231–236. DOI: 10.2307/2005828.

Saad, Yousef (2003). *Iterative Methods for Sparse Linear Systems*. 2nd. SIAM. DOI: 10.1016/S1570-579X(01)80025-2.

Trottenberg, U., C.W. Oosterlee, and A. Schüller (2001). *Multigrid*. Academic Press. ISBN: 012701070X.

Diskin, Boris, James L. Thomas, and Raymond E. Mineck (2005). "On Quantitative Analysis Methods for Multigrid Solutions". In: *SIAM Journal on Scientific Computing* 27.1, pp. 108–129. ISSN: 1064-8275. DOI: 10.1137/030601521.

Chapter 11

Nonlinear Solvers

$$\|e_{k+1}\| \leq C\|e_k\|^q \tag{11.1}$$

11.1 Single Step Solvers

11.1.1 What is the Picard Iteration?

If we are given the fixed-point problem,

$$x = K(x), \tag{11.2}$$

then the simplest iteration towards solution would be to just keep applying the operator K,

$$x_{n+1} = K(x_n). \tag{11.3}$$

This iterative method is also called nonlinear Richardson iteration, Picard iteration, or the method of successive substitution. It is convergent if the operator K is *contractive* (Ortega and Rheinboldt 1987; Kelley 1995). However, if we begin with a system of nonlinear equations,

$$\mathcal{F}(x) = 0, \tag{11.4}$$

there is some ambiguity about the definition of this iteration. We could use the trivial formulation

$$x = x - \mathcal{F}(x) \equiv K(x), \tag{11.5}$$

but this may not be the best choice. For example, suppose that the problem consists of a small nonlinear perturbation $\mathcal{M}$ to a linear operator A,

$$\mathcal{F}(x) \equiv Ax + \mathcal{M}(x) - b = 0. \tag{11.6}$$

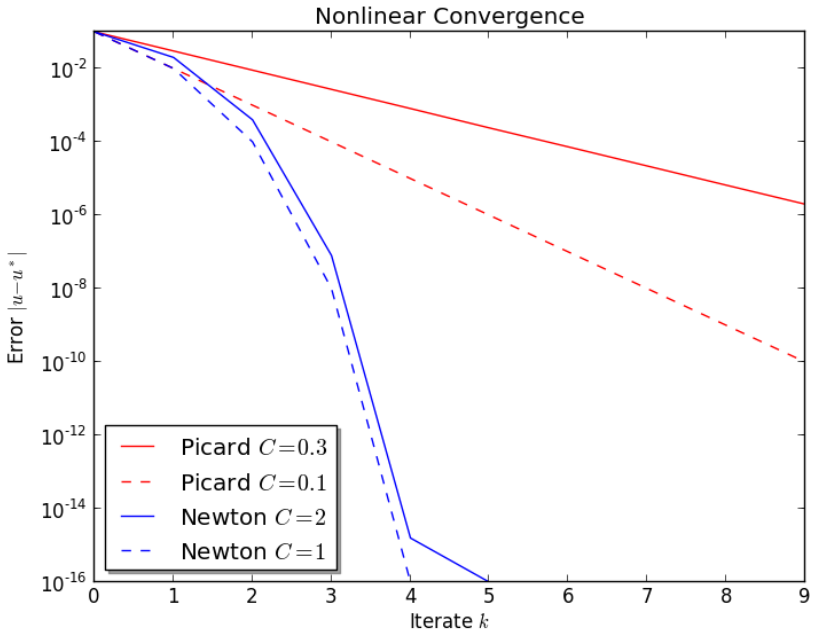


Figure 11.1: Comparison of different convergence rates

Then we can define an iteration

$$x = A^{-1}\left(b - \mathcal{M}(x)\right) \equiv K(x), \tag{11.7}$$

which is much more likely to converge, and can sometimes be seen as a sort of inexact Newton method.

11.2 Multistep Solvers

11.3 Solver Composition

(Brune et al. 2015)

References

Ortega, James M and Werner C Rheinboldt (1987). *Iterative solution of nonlinear equations in several variables*. Vol. 30. Society for Industrial and Applied Mathematics.

Kelley, C. T. (1995). *Iterative Methods for Linear and Nonlinear Equations*. Philadelphia: SIAM.

Brune, Peter R., Matthew G. Knepley, Barry F. Smith, and Xuemin Tu (2015). "Composing Scalable Nonlinear Algebraic Solvers". In: *SIAM Review* 57.4. http://www.mcs.anl.gov/papers/P2010-0112.pdf, pp. 535–565. DOI: 10.1137/130936725. URL: http://www.mcs.anl.gov/papers/P2010-0112.pdf.

Chapter 12

Problem Solutions

The best organized and most challenging problem sets I have ever encountered came from the classic texts *Concrete Mathematics* (Graham, Knuth, and Patashnik 1989) and *Classical Electrodynamics* (Jackson 1962). I will endeavor to be as complete, insightful, and bedeviling as those authors.

12.1 Programming Basics

Problem I.1 Following the online directions, install the latest release of PETSc.

Solution I.1 On my Mac Air, I configure using

```
./config/configure.py --PETSC_ARCH=arch-master-debug
  --download-chaco --download-ctetgen --download-exodusii
  --download-hdf5 --download-metis --download-mpich
  --download-netcdf --download-p4est --download-parmetis
  --download-pragmatic --download-triangle
  --with-cc="/Users/knepley/MacSoftware/bin/ccache␣gcc␣-Qunused-arguments"
  --with-cxx="/Users/knepley/MacSoftware/bin/ccache␣g++␣-Qunused-arguments"
  --with-fc="/Users/knepley/MacSoftware/bin/ccache␣gfortran"
  --with-shared-libraries
```

and then `make all`. After installation, you can test your PETSc using

```
  make check
```

Problem I.2 Clone my sample repository of PETSc code onto your local machine, https://bitbucket.org/knepley/simplepetscexample.

Solution I.2 We can clone the repository using SSH

```
git clone git@bitbucket.org:knepley/simplepetscexample.git tutorial_code
```

or if port 22 is not open, you can use HTTPS

```
git clone https://knepley@bitbucket.org/knepley/simplepetscexample.git tutorial_code
```

assuming you have put your SSH key on Bitbucket. Otherwise, you should remove the `username@` from the URL.

Problem I.3 Create a repository on GitHub and checkin you LaTeX paragraph.

Solution I.3 You can create a Git repository on Bitbucket by following these steps, and definitely setup SSH. Once your essay is written, you can schedule it for commit using

```
git add scicompEssay.tex
```

commit it

```
git commit -m "Draft␣essay"
```

and push to Bitbucket

```
git push
```

This can also be accomplished using a GUI tool, such as SourceTree from Atlassian which has many associated tutorials.

Problem I.4 Write a makefile that compiles the code in the sample repository and commit it.

Solution I.4 A possible makefile is shown below, and we will discuss the elements in detail.

```
CFLAGS =
CPPFLAGS =

bin/ex5: src/ex5.o src/myStuff.o src/myStuff2.o
	-@${MKDIR} bin
	${CLINKER} -o $@ $^ ${PETSC_LIB}
	${DSYMUTIL} $@
	${RM} $^

clean::
	${RM} bin/ex5
	${RM} -r bin/ex5.dSYM

include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
```

The `CFLAGS` and `CPPFLAGS` variables can be used to feed additional compiler options to the build. Note that the executable `bin/ex5` depends on only the object files `src/ex5.o, src/myStuff.o, src/myStuff2.o`. These are built automatically using rules imported from `${PETSC_DIR}/lib/petsc/conf/rules`. The commands shell `MKDIR`, `CLINKER`, `DSYMUTIL`, and `RM` are defined in `${PETSC_DIR}/lib/petsc/conf/variables` and allow the makefile to be portable across operating systems and architectures. Note also that we have use *automatic variables* in this makefile, which are defined here. The `$@` refers to the target name, here `bin/ex5`, while the `$^` is a list of all the prerequisites, here `src/ex5.o, src/myStuff.o, src/myStuff2.o`. Writing rules this way make them less error-prone and more robust to change. We can submit the makefile using

```
git add makefile
git commit -m "Added makefile"
git push
```

Problem I.5 The simple Python script below runs the sample `ex5` for a range of problem sizes and plots the timing.

```
#! /usr/bin/env python
import os

sizes = []
times = []
for k in range(5):
  Nx = 10 * 2**k
  modname = 'perf%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname]
  os.system('./bin/ex5 '+' '.join(options))
  perfmod = __import__(modname)
  sizes.append(Nx ** 2)
  times.append(perfmod.Stages['Main Stage']['SNESSolve'][0]['time'])
print zip(sizes, times)

from pylab import legend, plot, loglog, show, title, xlabel, ylabel
plot(sizes, times)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Time (s)')
show()

loglog(sizes, times)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Time (s)')
show()
```

Notice that the logging information is output in a Python module named `perf1.py` for $k = 1$. Each time we output a module, it must have a different name since Python caches module contents by name.

Modify this Python script to report the linear solver time (`KSPSolve`) instead of the nonlinear solve time (`SNESSolve`), and plot it for the GMRES/ILU (`-ksp_type gmres -pc_type ilu`) and GMRES/GAMG (`-ksp_type gmres -pc_type gamg`) solvers on the same graph. For extra credit, look at the performance as the number of processes increases.

Solution I.5 Here is my script which compares the performance of two different solvers as a function of problem size:

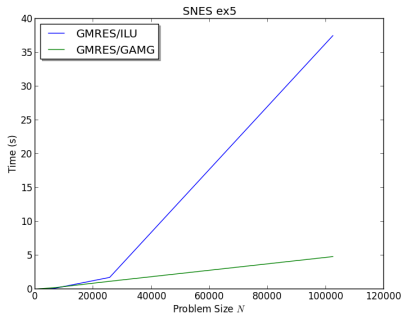
```
#! /usr/bin/env python
import argparse
import os

parser = argparse.ArgumentParser(
  description = 'CAAM 519 Homework I.4',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
parser.add_argument('--debug', action='store_true', default=False,
                    help='Turn on debugging')
args = parser.parse_args()

sizes = []
timesA = []
timesB = []
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfA%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname]
  cmd = './bin/ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  sizes.append(Nx ** 2)
  timesA.append(perfmod.Stages['Main Stage']['KSPSolve'][0]['time'])
print zip(sizes, timesA)
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfB%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname, '-pc_type', 'gamg']
  cmd = './bin/ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  timesB.append(perfmod.Stages['Main Stage']['KSPSolve'][0]['time'])
print zip(sizes, timesB)

from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
plot(sizes, timesA, sizes, timesB)
```

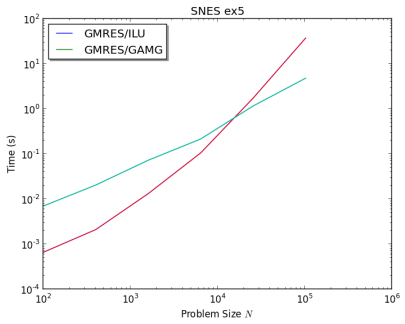
```
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Linear Solver Time (s)')
legend(['GMRES/ILU', 'GMRES/GAMG'], 'upper left', shadow = True)
if args.save:
  savefig('hw1_4_linear.png')
else:
  show()

loglog(sizes, timesA, sizes, timesB)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Linear Solver Time (s)')
legend(['GMRES/ILU', 'GMRES/GAMG'], 'upper left', shadow = True)
if args.save:
  savefig('hw1_4_log.png')
else:
  show()
```

Running it using

```
./bin/prob1_4.py --kmax=6 --save
```

produces the graphs below. Note that GMRES/ILU is initially faster than GMRES/GAMG, but eventually is much more expensive because it is not linear in the problem size. Note that when run with debugging enabled, GAMG is always slower than ILU due to the great number of internal checks made and necessity of optimized code for good performance.





Problem I.6 Modify the script from Problem 5 to report the assembly time (`SNESFunctionEval` and `SNESJacobianEval`), of both the residual and Jacobian, instead of the nonlinear solve time (`SNESSolve`), and plot it for the GMRES/ILU and GMRES/GAMG solvers on the same graph.

Solution I.6 Here is my script which compares the performance of two different solvers as a function of problem size:

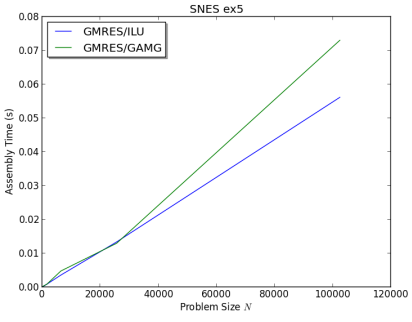
```
#! /usr/bin/env python
import argparse
import os

parser = argparse.ArgumentParser(
  description = 'CAAM 519 Homework I.5',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
parser.add_argument('--debug', action='store_true', default=False,
                    help='Turn on debugging')
args = parser.parse_args()

sizes = []
timesA = []
timesB = []
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfA%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname]
  cmd = './bin/ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  sizes.append(Nx ** 2)
  timesA.append(perfmod.Stages['Main Stage']['SNESFunctionEval'][0]['time'] +
                perfmod.Stages['Main Stage']['SNESJacobianEval'][0]['time'])
print zip(sizes, timesA)
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfB%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname, '-pc_type', 'gamg']
  cmd = './bin/ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  timesB.append(perfmod.Stages['Main Stage']['SNESFunctionEval'][0]['time'] +
                perfmod.Stages['Main Stage']['SNESJacobianEval'][0]['time'])
print zip(sizes, timesB)
```

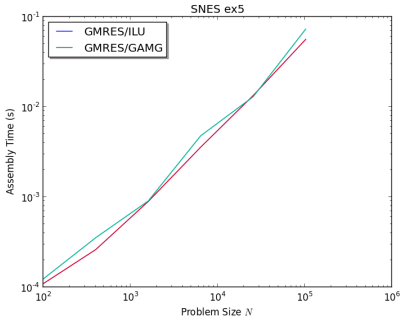
```
from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
plot(sizes, timesA, sizes, timesB)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Assembly Time (s)')
legend(['GMRES/ILU', 'GMRES/GAMG'], 'upper left', shadow = True)
if args.save:
  savefig('hw1_5_linear.png')
else:
  show()

loglog(sizes, timesA, sizes, timesB)
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Assembly Time (s)')
legend(['GMRES/ILU', 'GMRES/GAMG'], 'upper left', shadow = True)
if args.save:
  savefig('hw1_5_log.png')
else:
  show()
```

Running it using

```
./bin/prob1_5.py --kmax=6 --save
```

produces the graphs below. Note that since the number of Newton iterations is insensitive to the linear solver used, the assembly time is very similar, and probably only varies due to machine noise.





12.2 Finding and Relating Information

Problem II.1 It is quite likely that no matter what profession you choose to pursue after this course, expository writing will form a large part of your workload. Please write a paragraph or two describing what you hope to learn from this course, suggestions for upcoming units, or broader thoughts on scientific

computing and its progress as a discipline. Typeset your work in LaTeX and include at least one citation using BibTeX.

Solution II.1 The problem of transmission of computational knowledge is often framed as a problem *reproducibility* (Stodden et al. 2010). The goal is that any computation performed for the paper can be run and checked by the reviewer/reader. However, this is often unrealistic, given specialized hardware, large problem sizes, non-portable libraries, and just plain awful code from the researchers. While I believe it should be a requirement that any code employed in a paper should be provided with the paper itself, for the reasons propounded in the excellent article by LeVeque (LeVeque 2013), reproducibility is today impractical. So how do we move forward?

In the situation where only a handful of researchers have the specialized skills and resources to evaluate pieces of computational research, it is crucial that they inform the process. It is many times impossible to locate or enlist the help of such researchers for the traditional peer review process. A possible antidote would be the introduction of *open review* for computational publications. This would allow moderated comment on a paper, after publication, by all members of the community. This would help to provide validation of the results, put the work in context, compare it to other efforts, and help young people understand the algorithmic and programmatic decision making process.

The open review process also restores the natural role of a scientific journal, namely selection of important results. Free services such as arXiv enable efficient dissemintation of scientific work for no cost to the author, removing the need for journal publication. Journal editors and reviewers could then focus on selecting the most important work and highlighting it for readers.

Problem II.2 Create a PDF from your essay source and submit it by email with the subject *Essay I*.

Solution II.2 The best way to create PDF from LaTeX is to use `pdflatex`,

```
pdflatex essay.tex
bibtex essay
pdflatex essay.tex
pdflatex essay.tex
```

where the repetition is necessay to assure that the metadata stored in auxiliary files is consistent. This process can be handled in an elegant way by using the `latexmk` program,

```
latexmk -pdf essay.tex
```

If you rely on TeX source or BibTeX files in other locations, you can use

```
TEXINPUTS=${TEXINPUTS}:/path/to/tex BIBINPUTS=${BIBINPUTS}:/path/to/bib
  latexmk -pdf essay.tex
```

Problem II.3 Using any internet resources available, answer the following questions, providing proper citation for the information you provide:

1. Give the generating function for the sequence $1, 1, 2, 2, 3, 5, 5, 7, 10, 15, 15, 20, 27, 37, \ldots$.
2. Give an asymptotic expansion for the Gamma function $\Gamma(z)$ as $z \to \infty$ with error term.
3. On Ubuntu systems, why can you get the error `ImportError: No module named _md5` when using `import hashlib` in Python?
4. Does the popular nonlinear solver deserve to be called the *Newton-Raphson* method? Why or why not?
5. Who is Leonid Kantorovich?
6. How do I solve the semiconductor equations?

Solution II.3

1. The On-line Encyclopedia of Integer Sequences calls this Aitken's array, also called the Bell triangle or the Peirce triangle. It is the number of equivalence relations $a(n, k)$ on $\{0, ..., n\}$ such that k is not equivalent to n, $k + 1$ is not equivalent to n, ..., $n - 1$ is not equivalent to n. The double-exponential generating function is given by

$$\sum_{n,k} a(n-k,k)\frac{x^n y^k}{n!k!} = \exp(e^{x+y} - 1 + x), \tag{12.1}$$

but it appears that the general term can also be expressed as

$$a(n,k) = \sum_{i=0}^{k} \binom{k}{i} \sum_{j=0}^{n-k+i} \left\{ {n-k+i \atop j} \right\}. \tag{12.2}$$

2. The Digital Library of Mathematical Functions from NIST has an asymptotic expansion for the Γ-function,

$$\Gamma(z) \sim e^{-z} z^z \sqrt{\frac{2\pi}{z}} \sum_{k=0}^{\infty} \frac{g_k}{z^k}. \tag{12.3}$$

If we consider the truncated expansion with error term R,

$$\Gamma(z) \sim e^{-z} z^z \sqrt{\frac{2\pi}{z}} \left(\sum_{k=0}^{K-1} \frac{g_k}{z^k} + R_K(z) \right), \tag{12.4}$$

we have the bound,

$$|R_K(z)| \leq \frac{(1 + \zeta(K))\Gamma(K)}{2(2\pi)^{K+1}|z|^K} \left(1 + \min\left(\sec(\mathrm{ph} z), 2\sqrt{K}\right)\right). \tag{12.5}$$

3. A question on Stack Overflow tells us that `libssl-dev` must be installed before configuring and building your Python interpreter, in order to have the md5 module function correctly.

4. According to Wikipedia, Newton's method was first published in 1685 by John Wallis, but this method was purely algebraic, only worked for polynomials and was not structured as an iterative update. In 1690, Joseph Raphson published a simplified description which generated successive approximations. However, it was not until 1740 that Thomas Simpson published a version of Newton's method based on calculus and intended for general nonlinear equations. Thus, it appears there might be more justification for calling it the Newton-Simpson method.

5. An excellent article on the life and work of Leonid Kantorovich can be found at the Russian Virtual Computer Museum. Although most people are familiar with the contributions of Wiener and von Neumann to the development of computing, few people today seem to recall the contributions of Kantorovich. Not only did he pioneer approximate methods of functional analysis and linear programming, but he was one of the first mathematical users of automatic computing. Continuing the applied mathematics tradition in St. Petersburg (Leningrad) he developed early parallel programming techniques to compensate for the speed of his machines, finishing tabulation of Bessel function more quickly than Americans using the more powerful MARC and EINIAC. He famously delivered a talk entitled *Functional Analysis and Computational Mathematics*, with S.L. Sobolev and L.A. Lyusternik, at the Third All-Union Mathematical Congress in 1956, which sounds modern even today.

6. The drift-diffusion model for transport of electrons and holes in a semiconductor is often called the *semiconductor equations*. NanoHub has a good description of their solution methods. The Gummel method is very popular, since it decouples the Poisson equation for potential from the continuity equations for electrons and holes. In general, Gummel's method seems to be preferred at low bias because of its lower cost per iteration. At medium and high bias the coupling between equations becomes stronger, and the convergence rate deteriorates, and Newton's method becomes more competitive, as in this paper. However, since Gummel's method has a fast initial error reduction, many authors couple the two procedures, using Newton's method after several Gummel's iterations. This may be a good candidate for nonlinear preconditioning algorithms.

Problem II.4 Using any internet resources available, answer the following questions, providing proper citation for the information you provide:

1. What relation generates the sequence $8, 12, 16, 24, 32, 36, 48, 96, 128, 160, 192, 288, 768, \ldots$?
2. Give an asymptotic expansion for the complete elliptic integral $E(k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$ as $k \to 1$.

3. In Linux, if you receive a linker error with the text "`relocation R_X86_64_32S against symbol` ...", what has happened?

4. Has Hilbert's 13th Problem been solved? If so, who solved it and when.

5. Who invented the Python language? What language did this person work on prior to Python?

6. If I am simulating an incompressible flow, what discretization would be "mass conservative" for these equations?

Solution II.4

1. The On-line Encyclopedia of Integer Sequences tells us that this sequence consists of numbers n for which $n = \varphi(x)\varphi(y)$, where $n = x + y$ and $\varphi(x)$ is the Euler totient function of x.

2. The Digital Library of Mathematical Functions from NIST has an convergent series for $E(k)$ as $k \to 1$

$$E(k) = 1 + \frac{1}{2}\sum_{m=0}^{\infty} \frac{\left(\frac{1}{2}\right)_m \left(\frac{3}{2}\right)_m}{(2)_m m!} k'^{2m+2} \left(\ln\frac{1}{k'} + \psi(1+m) - \psi\left(\frac{1}{2}+m\right) - \frac{1}{(2m+1)(2m+2)}\right) \tag{12.6}$$

   where $\psi(x)$ is the digamma function, $(\cdot)_m$ is Pochhammer's Symbol, and $|k'| < 1$.

3. A good explanation of this error is given here. In this case, objects compiled as *position independent code* are being linked with objects which were not. On 32-bit Linux versions, this works just fine, but the 64-bit versions are more strict when linkers and throw an error. The solution is to give the appropriate compiler flag, e.g. `-fPIC`, and as the webpost points out the error could be coming from a library in the link.

4. In his 13th Problem, Hilbert asks for the solution of an equation of seventh degree

$$x^7 + ax^3 + bx^2 + cx + 1 = 0 \tag{12.7}$$

   which can be considered a function of three variables a, b, and c. Could we express this function as the composition of a finite number of two-variable functions? If we consider the allowable solution functions to come from the space of continuous functions, then the affirmative answer was given in 1957 by Vladimir Arnold, then a nineteen year old student of Andrey Kolmogorov. However, if we consider the solution to be an algebraic function, then the problem is still open.

5. Guido van Rossum developed the Python language. Before that, he had worked on ABC, which he discusses in his PyCon 2016 talk.

6. The finite volume method can be mass convservative to machine precision, since the continuity equation can be written as a conservation law.

Problem II.5 Make a contribution to Wikipedia and send the link to your edit.

Solution II.5 I helped explain non-conforming finite element spaces on Wikipedia.

12.3 PETSc Introduction

Problem III.1 Consider the fixed point problem

$$x = Gx \qquad x \in \mathcal{B} \tag{12.8}$$

where $\mathcal{B}$ is some Banach space. It is very common to solve these problems using an iterative method

$$x_{i+1} = \mathcal{M}\left(x_i, \ldots, x_{i-m}\right) \tag{12.9}$$

where x_{i+1} is the next approximate solution, $\{x_i, \ldots, x_{i-m}\}$ are previous approximate solutions, and $\mathcal{M}$ is some function defining the method. In (Anderson 1965), Anderson proposed an iterative method for systems of nonlinear equations, in which the next approximate solution x_{i+1} is chosen to satisfy a minimzation problem involving k prior solutions. However, we will restrict ourselves to the case of *no* prior solutions, or what is called *simple mixing* (Fang and Saad 2009),

$$x_{i+1} = x_i + \beta f_i, \tag{12.10}$$

where f_i is the residual vector at the ith iterate. In ex5 from Problem 2, implement a simple mixing solver using the `SNESSHELL` type in PETSc. Compare the convergence of simple mixing to Newton's method for the default initial guess. Plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the runtime. The y-axis shows the precision of the result, and here we will use the problem size as a proxy for the precision, an acceptable approach for this well-conditioned problem.

Solution III.1 We begin by implementing the simple mixing iteration, shown below. We check the initial residual and report it for the monitors, and then iterate until convergence.

```
#undef __FUNCT__
#define __FUNCT__ "SimpleMixing"
PetscErrorCode SimpleMixing(SNES snes, Vec x) {
  Vec r;
  PetscScalar *beta;
  PetscReal atol, rtol, r0, res;
  PetscInt maxits, n;
  PetscErrorCode ierr;

  PetscFunctionBeginUser;
  ierr = SNESShellGetContext(snes, (void **) &beta);CHKERRQ(ierr);
  ierr = SNESGetFunction(snes, &r, NULL, NULL);CHKERRQ(ierr);
  ierr = SNESGetTolerances(snes, &atol, &rtol, NULL, &maxits, NULL);CHKERRQ(ierr);
  /* Check Initial Residual */
  ierr = SNESComputeFunction(snes, x, r);CHKERRQ(ierr);
  ierr = VecNorm(r, NORM_2, &r0);CHKERRQ(ierr);
  ierr = SNESSetIterationNumber(snes, 0);CHKERRQ(ierr);
  ierr = SNESSetFunctionNorm(snes, r0);CHKERRQ(ierr);
  ierr = SNESMonitor(snes, 0, r0);CHKERRQ(ierr);
  if (r0 < atol) {
    ierr = SNESSetConvergedReason(snes, SNES_CONVERGED_FNORM_ABS);CHKERRQ(ierr);
    PetscFunctionReturn(0);
  }
  for (n = 0; n < maxits; ++n) {
    /* Calculate new approximate solution x_{i+1} = x_i + beta f_i */
    ierr = VecAXPY(x, *beta, r);CHKERRQ(ierr);
    ierr = SNESComputeFunction(snes, x, r);CHKERRQ(ierr);
    ierr = VecNorm(r, NORM_2, &r0);CHKERRQ(ierr);
    /* Check Convergence */
    ierr = SNESSetIterationNumber(snes, n+1);CHKERRQ(ierr);
    ierr = SNESSetFunctionNorm(snes, res);CHKERRQ(ierr);
    ierr = SNESMonitor(snes, n+1, res);CHKERRQ(ierr);
    if (res < atol) {ierr = SNESSetConvergedReason(snes, SNES_CONVERGED_FNORM_ABS);CHKERRQ(ierr);}
    if (res/r0 < rtol) {ierr = SNESSetConvergedReason(snes, SNES_CONVERGED_FNORM_RELATIVE);CHKERRQ(ierr);}
  }
  if (n == maxits) {ierr = SNESSetConvergedReason(snes, SNES_DIVERGED_MAX_IT);CHKERRQ(ierr);}
  PetscFunctionReturn(0);
}
```

I declare a new variable in `main()`,

```
  PetscScalar beta = 1.0;
```

and then after line 121 in ex5.c I add,

```
  ierr = SNESSetType(snes, SNESSHELL);CHKERRQ(ierr);
  ierr = SNESShellSetSolve(snes, SimpleMixing);CHKERRQ(ierr);
```

```
  ierr = SNESShellSetContext(snes, &beta);CHKERRQ(ierr);
  ierr = PetscOptionsGetScalar(NULL, "-beta", &beta, NULL);CHKERRQ(ierr);
```

We can compare the convergence behavior to Newton. For the small initial problem with 16 unknowns, the Newton solver,

```
./ex5 -snes_monitor -snes_converged_reason -snes_type newtonls
```

takes many fewer iterates

```
  0 SNES Function norm 2.075636426232e-01
  1 SNES Function norm 1.489679343548e-02
  2 SNES Function norm 1.139674015480e-04
  3 SNES Function norm 6.924167017530e-09
  4 SNES Function norm 4.440892098501e-16
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 4
```

than the Anderson solver

```
./ex5 -snes_monitor -snes_converged_reason
```

which takes 13 iterates

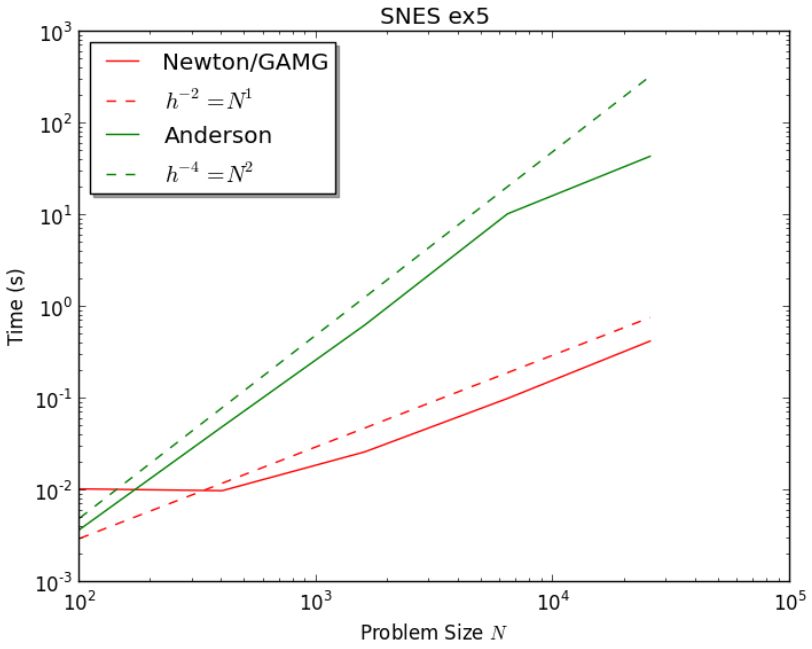
```
  0 SNES Function norm 2.075636426232e-01
  1 SNES Function norm 3.160865516063e-02
  2 SNES Function norm 7.040770859724e-03
  3 SNES Function norm 1.651820286652e-03
  4 SNES Function norm 3.919656779638e-04
  5 SNES Function norm 9.325862697018e-05
  6 SNES Function norm 2.220260987729e-05
  7 SNES Function norm 5.286694642948e-06
  8 SNES Function norm 1.258867274423e-06
  9 SNES Function norm 2.997639181324e-07
 10 SNES Function norm 7.138051039135e-08
 11 SNES Function norm 1.699730844962e-08
 12 SNES Function norm 4.047442381250e-09
 13 SNES Function norm 9.637872722124e-10
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 13
```

However, the Anderson solver does much less work, so we should really examine the work/precision tradeoff. To do this, we will use the runtime as our definition of work, and the problem size as a proxy for the precision, an acceptable approach for this well-conditioned problem.

```
#! /usr/bin/env python
import argparse
import os
import numpy as np

parser = argparse.ArgumentParser(
  description = 'CAAM 519 Homework III.1',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
parser.add_argument('--debug', action='store_true', default=False,
```

```
                    help='Turn on debugging')
args = parser.parse_args()

sizes = []
timesA = []
timesB = []
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfA%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname, '-snes_type', 'newtonls', '-pc_type', 'gamg']
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  sizes.append(Nx ** 2)
  timesA.append(perfmod.Stages['Main Stage']['SNESSolve'][0]['time'])
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfB%d' % k
  options = ['-da_grid_x', str(Nx), '-da_grid_y', str(Nx), '-log_view',
             ':%s.py:ascii_info_detail' % modname, '-snes_max_it', str(100000), '-beta', '-0.1']
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  os.system(cmd)
  perfmod = __import__(modname)
  timesB.append(perfmod.Stages['Main Stage']['SNESSolve'][0]['time'])
N = np.array(sizes)

from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
loglog(N, timesA, 'r', N, 3e-5 * N ** 1., 'r--', N, timesB, 'g', N, 5e-7 * N ** 2, 'g--')
title('SNES ex5')
xlabel('Problem Size $N$')
ylabel('Time (s)')
legend(['Newton/GAMG', '$h^{-2} = N^1$', 'Anderson', '$h^{-4} = N^2$'], 'upper left', shadow = True)
if args.save:
  savefig('hw3_1_log.png')
else:
  show()
```

First we note that for larger problems, convergence can only be achieved with a reduced β (we use $\beta = -0.1$ for our tests). We can also see that simple Anderson mixing performs better than Newton only for the very smallest problem sizes, and that the convergence deteriorates significantly with problem size.



Problem III.2 The QR decomposition is a representation of a matrix A in terms of an orthogonal matrix Q and an upper triangular matrix R,

$$A = QR. \tag{12.11}$$

It is described in detail on Wikipedia, and also in many textbooks (Trefethen and Bau, III 1997). Implement the QR decomposition in PETSc for arbitrary matrix dimension using the Gram-Schmdit process. In your code, include a test of the routine for some matrix A which reports $||A - QR||$.

Extra Credit: Implement QR using Householder reflectors or Givens rotations.

Extra Extra Credit: Implement the TSQR Algorithm.

Solution III.2 The Gram-Schmidt process (Stewart 2011) orthogonalizes a set of vectors by subtracting off the piece of an initial vector parallel to each of the subsequent vectors. For example, suppose we have vectors v_0 and v_1. then the vector

$$u_1 = v_1 - \frac{v_1 \cdot v_0}{v_0 \cdot v_0} v_0 \tag{12.12}$$

is orthogonal to v_0, as can be verified by explicit calculation

$$v_0 \cdot u_1 = v_0 \cdot v_1 - \frac{v_1 \cdot v_0}{v_0 \cdot v_0} v_0 \cdot v_0 = 0. \tag{12.13}$$

If we normalize the vectors u_k, then we have an orthonormal system e_k defined by

$$u_k = v_k - \sum_{j=0}^{k-1} \frac{v_k \cdot u_j}{u_j \cdot u_j} u_j \qquad e_k = \frac{u_k}{||u_k||} \tag{12.14}$$

which we can use to express our original vectors

$$v_k = \sum_{j=0}^{k} (v_k \cdot e_j) e_j = \sum_{j=0}^{k} \frac{v_k \cdot u_j}{||u_j||} e_j. \tag{12.15}$$

This can be written in matrix form $V = QR$, where

$$Q = [e_0, \ldots, e_n] \qquad R = \begin{pmatrix} v_0 \cdot e_0 & v_1 \cdot e_0 & v_2 \cdot e_0 & \ldots \\ 0 & v_1 \cdot e_1 & v_2 \cdot e_1 & \ldots \\ 0 & 0 & v_2 \cdot e_2 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{12.16}$$

In PETSc, the easiest way to code this for parallel computing is to have R replicated on each process, but the columns of Q distributed across processes. The code below implements QR in this fashion.

```
static char help[] = "Naive QR implementation\n";

#include <petsc.h>

#undef __FUNCT__
#define __FUNCT__ "ComputeQR"
PetscErrorCode ComputeQR(PetscInt n, Vec V[], Vec *Q[], Mat *R)
{
  PetscScalar *r;
  PetscReal norm;
  PetscInt k, j;
  PetscErrorCode ierr;

  PetscFunctionBeginUser;
  if (n <= 0) SETERRQ1(PETSC_COMM_SELF, PETSC_ERR_ARG_OUTOFRANGE,
    "The number of vectors to orthogonalize %D must be positive", n);
  ierr = VecDuplicateVecs(V[0], n, Q);CHKERRQ(ierr);
  ierr = MatCreateSeqDense(PETSC_COMM_SELF, n, n, NULL, R);CHKERRQ(ierr);
  ierr = MatDenseGetArray(*R, &r);CHKERRQ(ierr);
  for (k = 0; k < n; ++k) {
```

```
    ierr = VecCopy(V[k], (*Q)[k]);CHKERRQ(ierr);
    for (j = 0; j < k; ++j) {
      /* R_{jk} = v_k \cdot e_j */
      ierr = VecDot(V[k], (*Q)[j], &r[k*n+j]);CHKERRQ(ierr);
      ierr = VecAXPY((*Q)[k], -r[k*n+j], (*Q)[j]);CHKERRQ(ierr);
    }
    ierr = VecNorm((*Q)[k], NORM_2, &norm);CHKERRQ(ierr);
    ierr = VecScale((*Q)[k], 1.0/norm);CHKERRQ(ierr);
    ierr = VecViewFromOptions((*Q)[k], NULL, "-Q_view");CHKERRQ(ierr);
    ierr = VecDot(V[k], (*Q)[k], &r[k*n+k]);CHKERRQ(ierr);
  }
  ierr = MatDenseRestoreArray(*R, &r);CHKERRQ(ierr);
  ierr = MatViewFromOptions(*R, NULL, "-R_view");CHKERRQ(ierr);
  PetscFunctionReturn(0);
}

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc, char **argv)
{
  Vec *V, *Q, tmp;
  Mat R;
  const PetscInt n = 3;
  PetscScalar *r;
  PetscInt k;
  PetscMPIInt rank;
  PetscErrorCode ierr;

  ierr = PetscInitialize(&argc, &argv, NULL, help);CHKERRQ(ierr);
  ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
  /* Create initial vectors */
  ierr = VecCreate(PETSC_COMM_WORLD, &tmp);CHKERRQ(ierr);
  ierr = VecSetFromOptions(tmp);CHKERRQ(ierr);
  ierr = VecSetSizes(tmp, PETSC_DETERMINE, n);CHKERRQ(ierr);
  ierr = VecDuplicateVecs(tmp, n, &V);CHKERRQ(ierr);
  if (!rank) {
    ierr = VecSetValue(V[0], 0, 12.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[0], 1, 6.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[0], 2, -4.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[1], 0, -51.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[1], 1, 167.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[1], 2, 24.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[2], 0, 4.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[2], 1, -68.0, INSERT_VALUES);CHKERRQ(ierr);
    ierr = VecSetValue(V[2], 2, -41.0, INSERT_VALUES);CHKERRQ(ierr);
  }
```

```
  /* QR factor */
  ierr = ComputeQR(n, V, &Q, &R);CHKERRQ(ierr);
  /* Check factors */
  ierr = MatDenseGetArray(R, &r);CHKERRQ(ierr);
  for (k = 0; k < n; ++k) {
    PetscReal error;

    ierr = VecSet(tmp, 0.0);CHKERRQ(ierr);
    ierr = VecMAXPY(tmp, k+1, &r[k*n], Q);CHKERRQ(ierr);
    ierr = VecAXPY(tmp, -1.0, V[k]);CHKERRQ(ierr);
    ierr = VecNorm(tmp, NORM_2, &error);CHKERRQ(ierr);
    if (PetscAbsReal(error) > 1.0e-10) SETERRQ3(PETSC_COMM_WORLD, PETSC_ERR_PLIB,
      "||QR_%D - V_%D|| = %g\n", k, k, error);
  }
  ierr = MatDenseRestoreArray(R, &r);CHKERRQ(ierr);
  /* Cleanup */
  ierr = VecDestroy(&tmp);CHKERRQ(ierr);
  ierr = VecDestroyVecs(n, &V);CHKERRQ(ierr);
  ierr = VecDestroyVecs(n, &Q);CHKERRQ(ierr);
  ierr = MatDestroy(&R);CHKERRQ(ierr);
  ierr = PetscFinalize();
  return ierr;
}
```

It tests the routine on the following factorization

$$\begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix} = \begin{pmatrix} \frac{6}{7} & -\frac{69}{175} & -\frac{58}{175} \\ \frac{3}{7} & \frac{158}{175} & \frac{6}{175} \\ -\frac{2}{7} & \frac{6}{35} & -\frac{33}{35} \end{pmatrix} \begin{pmatrix} 14 & 21 & -14 \\ 0 & 175 & -70 \\ 0 & 0 & 35 \end{pmatrix} \tag{12.17}$$

and checks the solution by explicit multiplication.

12.4 Parallelism

12.5 Data Layout and Discretization I

Problem V.1 The schemes detailed above are designed for the situation in which we fix the value of a single unknown. However, it is easy to imagine that we would like to fix the value of a combination of unknowns. For example, suppose we are solving a problem for fluid flow in a cavity where the flow obeys the Euler equations, so that the constraint specifies that the fluid has no velocity normal to the boundary. If the boundary aligns with the coordinates axes, we can simply constrain that component of velocity. Now let our boundary be inclined at the 45° angle (see Fig. **??**) so that now our constraint equation

becomes

$$\begin{aligned} \vec{u} \cdot \hat{n} &= 0 \\ \begin{pmatrix} u \\ v \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} &= 0 \\ u + v &= 0. \end{aligned}$$

Design a system for enforcing this boundary condition in both cases above, namely eliminating constrained unknowns and replacing redundant equations.

Solution V.1 When eliminating constrained unknowns, it makes sense to define the global basis as one consisting of only unconstrained dofs. For this problem, we can redfine the global unknowns by rotating the original space

$$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u + v \\ v - u \end{pmatrix}$$

which we recognize as the rotation matrix for $\theta = 45°$. We can use the same local assembly procedure, but when we map local unknowns to global unknowns, instead of simply identifying local and global variables, we admit a linear map,

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \ddots \end{pmatrix} \begin{pmatrix} u_0 \\ v_0 \\ \vdots \\ u_k \\ v_k \\ \vdots \end{pmatrix}$$

where u_k, v_k come from a constrained vertex. For more general discretizations, we define a linear transformation M which maps local unknowns to unconstrained global unknowns.

If instead we would like to work in the original basis, then we can replace the equation for v_k by

$$u_k + v_k = 0$$

instead of just fixing the value of v_k.

12.6 Simple Finite Differences

Problem VI.1

Part I Carry out the MMS procedure for a modified Bratu equation which incorporates an inhomogeneous coefficient,

$$-\nabla \cdot \left(\tanh \left(x - \frac{1}{2} \right) \nabla u \right) - \lambda e^u = 0. \tag{12.18}$$

Use the same exact solutions as shown in the text,

$$u^* = x(1-x)y(1-y) \qquad u^* = \sin(\pi x)\sin(\pi y). \tag{12.19}$$

Create convergence graphs for the solutions in both the ℓ_2 and ℓ_∞ norms, as in the text. This problem becomes very hard to solve as the mesh size is increased, the opposite of the behavior we saw with the original equation. For example, if we use our script from before

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls \
    -da_grid_x 17 -da_grid_y 17 -da_refine $i \
    -pc_type mg -pc_mg_levels 3 -pc_mg_galerkin \
    -mg_levels_ksp_norm_type unpreconditioned -mg_levels_ksp_chebyshev_esteig 0.5,1.2 \
    -mg_levels_pc_type sor -pc_mg_type full -mms 3 \
  -snes_monitor -snes_converged_reason -ksp_converged_reason
done
```

we cannot even converge the first system.

```
  0 SNES Function norm 4.865204926677e-01
  Linear solve converged due to CONVERGED_RTOL iterations 3
  1 SNES Function norm 5.168062931528e-02
  Linear solve converged due to CONVERGED_RTOL iterations 15
  2 SNES Function norm 4.184362140608e-02
  Linear solve converged due to CONVERGED_RTOL iterations 23
  3 SNES Function norm 3.866392229628e-02
  Linear solve converged due to CONVERGED_RTOL iterations 25
  4 SNES Function norm 3.606888512126e-02
  Linear solve converged due to CONVERGED_RTOL iterations 30
  5 SNES Function norm 3.410310612571e-02
  Linear solve converged due to CONVERGED_RTOL iterations 58
  6 SNES Function norm 3.294064502174e-02
  Linear solve converged due to CONVERGED_RTOL iterations 57
  7 SNES Function norm 3.291316232478e-02
  Linear solve converged due to CONVERGED_RTOL iterations 54
Nonlinear solve did not converge due to DIVERGED_LINE_SEARCH iterations 7
```

Since we have control over the grid refinement, one common technique is to solve a smaller problem and use this as the initial guess for the larger problem, which is called *grid sequencing*. PETSc provides grid sequencing automatically using the option `-snes_grid_sequence`, however we will have to alter our error checking code to extract the final solution and grid by adding

```
  ierr = SNESGetSolution(snes, &y);CHKERRQ(ierr);
  ierr = SNESGetDM(snes, &dm);CHKERRQ(ierr);
```

Starting from a smaller grid,

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls -snes_grid_sequence $i -da_refine 1 \
    -ksp_rtol 1e-9 -pc_type mg -pc_mg_levels 3 -pc_mg_galerkin \
    -mg_levels_ksp_norm_type unpreconditioned -mg_levels_ksp_chebyshev_esteig 0.5,1.2 \
    -mg_levels_pc_type sor -pc_mg_type full -mms 3 \
    -snes_monitor -snes_converged_reason -ksp_converged_reason
done
```

we can converge the first five systems,

```
            0 SNES Function norm 6.805338772655e-01
            Linear solve converged due to CONVERGED_RTOL iterations 4
            1 SNES Function norm 5.154734539031e-01
            Linear solve converged due to CONVERGED_RTOL iterations 4
            2 SNES Function norm 5.899227765372e-02
            Linear solve converged due to CONVERGED_RTOL iterations 2
            3 SNES Function norm 4.111593886420e-03
            Linear solve converged due to CONVERGED_RTOL iterations 3
            4 SNES Function norm 2.234476660521e-05
            Linear solve converged due to CONVERGED_RTOL iterations 3
            5 SNES Function norm 1.179499962334e-09
          Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 5
          0 SNES Function norm 9.350819446678e-03
          Linear solve converged due to CONVERGED_RTOL iterations 4
          1 SNES Function norm 8.975872639470e-07
          Linear solve converged due to CONVERGED_RTOL iterations 4
          2 SNES Function norm 1.581449936603e-12
        Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
        0 SNES Function norm 4.820528651869e-03
        Linear solve converged due to CONVERGED_RTOL iterations 5
        1 SNES Function norm 2.839396976835e-08
        Linear solve converged due to CONVERGED_RTOL iterations 4
        2 SNES Function norm 3.513911836431e-15
      Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
      0 SNES Function norm 2.461494407946e-03
      Linear solve converged due to CONVERGED_RTOL iterations 10
      1 SNES Function norm 8.896591851778e-10
      Linear solve converged due to CONVERGED_RTOL iterations 10
      2 SNES Function norm 9.442247176540e-17
    Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
    0 SNES Function norm 1.244744690374e-03
    Linear solve converged due to CONVERGED_RTOL iterations 2807
    1 SNES Function norm 2.782787902882e-11
    Linear solve converged due to CONVERGED_RTOL iterations 2556
    2 SNES Function norm 1.850505579361e-16
  Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
  0 SNES Function norm 6.259834963960e-04
  Linear solve converged due to CONVERGED_RTOL iterations 7049
  1 SNES Function norm 8.694365205916e-13
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
0 SNES Function norm 3.139064083951e-04
Linear solve did not converge due to DIVERGED_ITS iterations 10000
Nonlinear solve did not converge due to DIVERGED_LINEAR_SOLVE iterations 0
```

which reveals that the problem is becoming quite ill-conditioned and our geometric multigrid solver cannot cope with the variation in coefficient. For these smaller serial problems, LU can be effective for checking convergence

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls -snes_grid_sequence $i \
    -da_refine 1 -ksp_rtol 1e-9 \
    -pc_type lu -mms 3 \
    -snes_converged_reason
done%$
```

and we see that if the linear systems can be solved, the nonlinear equation is effectively preconditioned with grid sequencing.

```
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 169 error l2 2.20735e-06 inf 7.9559e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 625 error l2 3.081e-07 inf 2.42794e-05
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 2401 error l2 4.0427e-08 inf 8.29001e-06
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 2
N: 9409 error l2 5.16863e-09 inf 2.62871e-06
```

```
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
N: 37249 error l2 6.53061e-10 inf 7.96102e-07
Nonlinear solve converged due to CONVERGED_FNORM_RELATIVE iterations 1
N: 148225 error l2 8.20737e-11 inf 2.33813e-07
```

Part II Update the flop counting in the residual and Jacobian evaluation to account for the additional flops in the new equation. Using the total flop count output for each run, plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the flops executed. The y-axis shows the precision of the result, and here we will use the error. Make this plot comparing two different solvers: SNES Newton using LU with grid sequencing, and SNES Newton using GMRES/GMG with grid sequencing.

Solution VI.1 For the new equation, we must derive a finite difference approximation of the operator. Previously, we had reasoned in the following way,

$$u_{xx}(x_i, y_i) \approx \frac{u_x(x_{i+1/2}) - u_x(x_{i-1/2})}{h} \tag{12.20}$$

$$\approx \frac{\frac{u(x_{i+1})-u(x_i)}{h} - \frac{u(x_i)-u(x_{i-1})}{h}}{h} \tag{12.21}$$

$$\approx \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{h^2}. \tag{12.22}$$

Thus now we have

$$\left(\frac{\partial}{\partial_x}\tanh\left(x-\frac{1}{2}\right)u_x\right)(x_i, y_i) \tag{12.23}$$

$$\approx \frac{\tanh\left(x_{i+1/2}-\frac{1}{2}\right)u_x(x_{i+1/2}) - \tanh\left(x_{i-1/2}-\frac{1}{2}\right)u_x(x_{i-1/2})}{h} \tag{12.24}$$

$$\approx \frac{\tanh\left(x_{i+1/2}-\frac{1}{2}\right)\frac{u(x_{i+1})-u(x_i)}{h} - \tanh\left(x_{i-1/2}-\frac{1}{2}\right)\frac{u(x_i)-u(x_{i-1})}{h}}{h} \tag{12.25}$$

$$\approx \frac{\tanh\left(x_{i+1/2}-\frac{1}{2}\right)u(x_{i+1})}{h^2} + \frac{\tanh\left(x_{i-1/2}-\frac{1}{2}\right)u(x_{i-1})}{h^2} \tag{12.26}$$

$$- \frac{\left(\tanh\left(x_{i+1/2}-\frac{1}{2}\right) + \tanh\left(x_{i-1/2}-\frac{1}{2}\right)\right)u(x_i)}{h^2}, \tag{12.27}$$

and an identical expression applies in the y-direction. For the first exact solution, we have

$$-\nabla\cdot\left(\tanh\left(x-\frac{1}{2}\right)\nabla u^*\right) - \lambda e^{u^*} \tag{12.28}$$

$$= -\nabla\cdot\tanh\left(x-\frac{1}{2}\right)\begin{pmatrix}(1-2x)y(1-y)\\ x(1-x)(1-2y)\end{pmatrix} - \lambda e^{x(1-x)y(1-y)} \tag{12.29}$$

$$= -(1-2x)y(1-y)\mathrm{sech}^2\left(x-\frac{1}{2}\right) + 2y(1-y)\tanh\left(x-\frac{1}{2}\right)$$
$$+ 2x(1-x)\tanh\left(x-\frac{1}{2}\right) - \lambda e^{x(1-x)y(1-y)}, \tag{12.30}$$

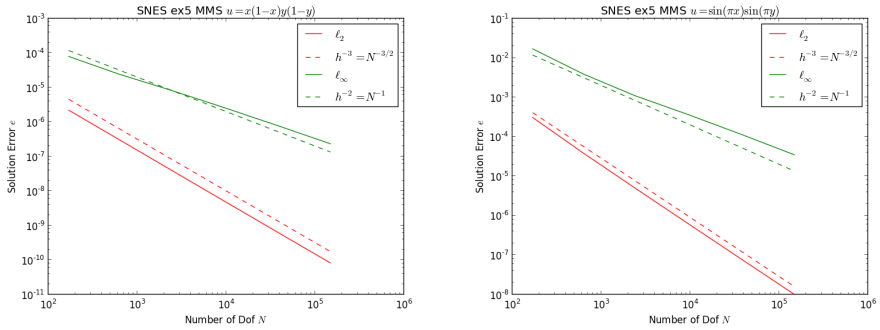


Figure 12.1: Mesh convergence of solutions to the modified Bratu equation $-\nabla \tanh\left(x-\frac{1}{2}\right)\nabla u - \lambda e^u = 0$.

and for the second

$$-\nabla\cdot\left(\tanh\left(x-\frac{1}{2}\right)\nabla u^*\right) - \lambda e^{u^*} \tag{12.31}$$

$$= -\nabla\cdot\tanh\left(x-\frac{1}{2}\right)\begin{pmatrix}\pi\cos(\pi x)\sin(\pi y)\\ \pi\sin(\pi x)\cos(\pi y)\end{pmatrix} - \lambda e^{\sin(\pi x)\sin(\pi y)} \tag{12.32}$$

$$= -\pi\cos(\pi x)\sin(\pi y)\operatorname{sech}^2\left(x-\frac{1}{2}\right) + \pi^2\sin(\pi x)\sin(\pi y)\tanh\left(x-\frac{1}{2}\right)$$
$$+\pi^2\sin(\pi x)\sin(\pi y)\tanh\left(x-\frac{1}{2}\right) - \lambda e^{\sin(\pi x)\sin(\pi y)}, \tag{12.33}$$

$$= -\pi\cos(\pi x)\sin(\pi y)\operatorname{sech}^2\left(x-\frac{1}{2}\right)$$
$$+2\pi^2\sin(\pi x)\sin(\pi y)\tanh\left(x-\frac{1}{2}\right) - \lambda e^{\sin(\pi x)\sin(\pi y)}. \tag{12.34}$$

Note that after our modification to the equation, the differential operator is still linear in u, even if it is nonlinear in x. Thus our Jacobian is just this operator itself. We introduce residual and Jacobian callbacks for the new model, which allows us to evaluate our error for a series of sizes, using for example a script like

```
for i in `seq 1 6`;
do
  ./ex5 -snes_type newtonls -snes_grid_sequence $i \
    -ksp_rtol 1e-9 \
    -da_refine 1 -mms 4 \
    -pc_type lu
done
```

which produces the plots in Figure 12.1. We note that the ℓ_∞ convergence is not maintained, which may have something to do with the extreme ill-conditioning of the Jacobian near the solution.

We can modify the `PetscLogFlops()` arguments in `FormFunctionLocalMMS3()`, `FormFunctionLocalMMS4()`, and `FormJacobianLocalMMS3()` in order to account for the extra flops used to compute the heterogeneous coefficient. Then using the script below, we can compute a work=precision diagram for two different solver configurations.

```
#! /usr/bin/env python
import argparse
import subprocess
import numpy as np

parser = argparse.ArgumentParser(
  description = 'CAAM 519 Homework IV.1',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
parser.add_argument('--debug', action='store_true', default=False,
                    help='Turn on debugging')
args = parser.parse_args()

errorA = []
flopsA = []
errorB = []
flopsB = []
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfA%d' % k
  options = ['-snes_type', 'newtonls', '-snes_grid_sequence', str(k), '-da_refine', '1',
             '-ksp_rtol', '1e-9', '-pc_type', 'lu', '-mms', '3',
               '-log_view', ':%s.py:ascii_info_detail' % modname]
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  errorA.append(float(out[4]))
  perfmod = __import__(modname)
  flopsA.append(perfmod.Stages['Main Stage']['summary'][0]['flops'])
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfB%d' % k
  options = ['-snes_type', 'newtonls', '-snes_grid_sequence', str(k), '-da_refine', '1',
             '-snes_max_linear_solve_fail', '10', '-ksp_rtol', '1e-9', '-pc_type', 'mg',
             '-pc_mg_levels', '3', '-pc_mg_galerkin',
```

```
            '-mg_levels_ksp_norm_type', 'unpreconditioned',
            '-mg_levels_ksp_chebyshev_esteig', '0.5,1.2',
            '-mg_levels_pc_type', 'sor', '-pc_mg_type', 'full', '-mms', '3',
            '-log_view', ':%s.py:ascii_info_detail' % modname]
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  errorB.append(float(out[4]))
  perfmod = __import__(modname)
  flopsB.append(perfmod.Stages['Main Stage']['summary'][0]['flops'])
FA = np.array(flopsA)
FB = np.array(flopsB)

from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
loglog(flopsA, errorA, 'r', FA, 0.5 * FA ** -1., 'r--',
       flopsB, errorB, 'g', FB, 5.0 * FB ** -1., 'g--')
title('SNES ex5')
xlabel('Flops executed $F$')
ylabel('Error $\|u - u^*\|$')
legend(['Newton/LU', '$h^2 = N^{-1}$', 'Newton GMRES/GMG', '$h^{-2} = N^{-1}$'],
       'upper right', shadow = True)
if args.save:
  savefig('hw4_1_log3.png')
else:
  show()
```

Looking at Figure 12.2, we see that geometric multigrid (GMG) never becomes practical for any problem size which can fit in the memory of my laptop.

Problem VI.2

Part I Carry out the MMS procedure for an equation similar to Lane-Emden Equation by modifying the Bratu example,

$$-\Delta u - u^{\lambda} = 0, \tag{12.35}$$

where $\lambda = 1, 2, 5$. Use the same exact solutions as shown in the text,

$$u^* = x(1-x)y(1-y) \qquad u^* = \sin(\pi x)\sin(\pi y). \tag{12.36}$$

Create convergence graphs for the solutions in both the ℓ_2 and ℓ_∞ norms, as in the text.

Part II Update the flop counting in the residual and Jacobian evaluation to account for the additional flops in the new equation. Using the total flop count

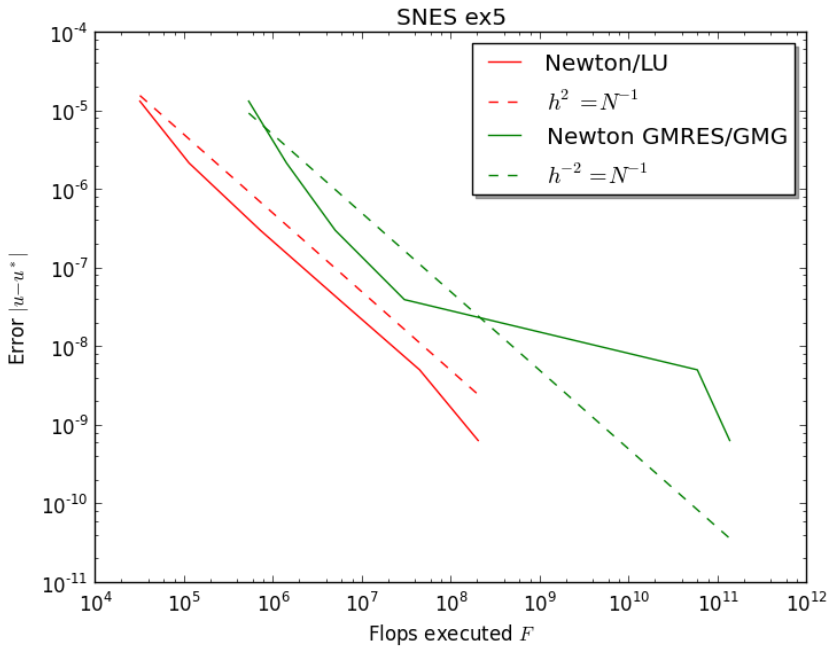


Figure 12.2: A work precision diagram for the modified Bratu equation $-\nabla \tanh\left(x - \frac{1}{2}\right) \nabla u - \lambda e^u = 0$ using two different solver configurations.

output for each run, plot a *work-precision* diagram for this solve. The x-axis should show the total work, and as a proxy we will use the flops executed. The y-axis shows the precision of the result, and here we will use the error. Make this plot comparing two different solvers: SNES Newton using LU, and SNES Newton using GMRES/GMG.

Extra Cedit Use a higher order approximation of the derivative, predict the enhanced convergence, and demonstrate it with a convergence graph.

Solution VI.2

Part I If we insert our first MMS solution, $u^* = x(1-x)y(1-y)$, into the equation, we get

$$2x(1-x) + 2y(1-y) - x^\lambda(1-x)^\lambda y^\lambda (1-y)^\lambda,$$

and similarly for the other solution

$$2\pi^2 \sin(\pi x)\sin(\pi y) - \sin^\lambda(\pi x)\sin^\lambda(\pi y).$$

We can evaluate the mesh convergence of our method using the Python script below,

```
#!/usr/bin/env python
import matplotlib.pyplot as plt
import os
import subprocess
import numpy as np
from pylab import annotate, legend, plot, loglog, show, title, xlabel, ylabel, figure, savefig

mmsNames = [5,6]
for l in [1.0, 3.0, 5.0]:
  for mms in mmsNames:
    print("MMS: ", mms)
    sizes = []
    errorsl2 = []
    errorslinf = []
    for k in range(4):
      Nx = 10*2**k
      modname = 'perf%d' % k
      options = ['-snes_type', 'newtonls', '-par', str(l), '-mms', str(mms),
        '-da_grid_x', str(Nx), '-da_grid_y', str(Nx),
        '-log_view', ':%s.py:ascii_info_detail' % modname]
      output = subprocess.check_output('./ex5 ' + ' '.join(options), shell=True)
      if output.find('l2') == -1: raise RuntimeError("l2 norm Not found")
      data = output.strip().split(' ')
      errorsl2.append(float(data[4]))
```

```
      errorslinf.append(float(data[6]))
      #perfmod = __import__(modname)
      if not float(data[1]) == Nx**2: raise RuntimeError("Size mismatch found")
      sizes.append(float(data[1]))
    # Linear Fit
    print zip(sizes, errorsl2)
    sizes = np.array(sizes)
    x = np.log10(np.array(sizes))
    yl2 = np.log10(np.array(errorsl2))
    ylinf = np.log10(np.array(errorslinf))
    X = np.hstack((np.ones((x.shape[0],1)),x.reshape((x.shape[0],1))))
    beta = np.dot(np.linalg.pinv(np.dot(X.transpose(),X)),X.transpose())
    beta = np.dot(beta, yl2.reshape((yl2.shape[0],1)))
    interceptl2 = beta[0][0]
    slopel2 = beta[1][0]
    beta = np.dot(np.linalg.pinv(np.dot(X.transpose(),X)),X.transpose())
    beta = np.dot(beta, ylinf.reshape((ylinf.shape[0],1)))
    interceptlinf = beta[0][0]
    slopelinf = beta[1][0]
    # Plot
    loglog(sizes, errorsl2, sizes, 0.9*sizes**-1.5, sizes, errorslinf, sizes, 0.9*sizes**-1)
    title('SNES ex5 $\lambda$ '+str(l)+' MMS '+str(mms))
    xlabel('Number of Dof $N$')
    ylabel('Solution Error $e$')
    legend(['$\ell_2$', '$h^{-3} = N^{-3/2}$', '$\ell_\infty$', '$h^{-2} = N^{-1}$'])
    annotate('fit slope %.2f' % slopel2, (sizes[len(sizes)/2], errorsl2[len(sizes)/2]),
      (sizes[len(sizes)/2-1], errorsl2[len(sizes)/2]))
    annotate('fit slope %.2f' % slopelinf, (sizes[len(sizes)/2], errorslinf[len(sizes)/2]),
      (sizes[len(sizes)/2-1], errorslinf[len(sizes)/2]))
    savefig(os.path.join('figures', 'fd_2_'+str(mms)+'_l'+str(int(l))+'.png'))
    show()
```

which produces the plots in Figure 12.3. Clearly the first MMS solution suffers from supercovnergence, but we can see that the second shows the expected convergence behavior.

Part II We change the `PetscLogFlops()` calls to reflect the additional flops from the new term, where we count exponentiation and tracendental functions as a single flop since it is quite complex to do a better job. The script below generates a work-precision test for the case $\lambda = 5$ using the second MMS solution for the LU and GMG solvers

```
#! /usr/bin/env python
import argparse
import subprocess, os
import numpy as np
```

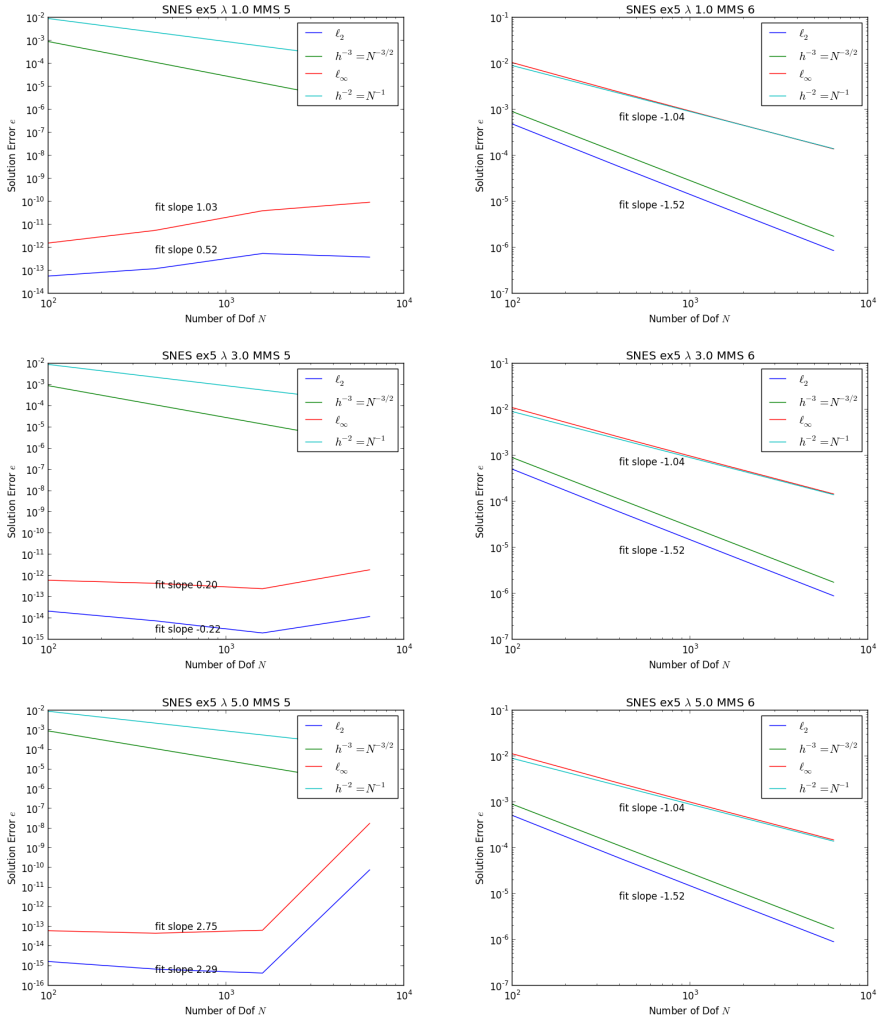


Figure 12.3: Mesh convergence of solutions to the modified Bratu equation $-\Delta u - u^\lambda = 0$.

```
parser = argparse.ArgumentParser(
  description = 'CAAM 519 Chapter FD Problem 2',
  epilog = 'For more information, visit http://www.mcs.anl.gov/petsc',
  formatter_class = argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--kmax', type=int, default=5,
                    help='The number of doublings to test')
parser.add_argument('--save', action='store_true', default=False,
                    help='Save the figures')
parser.add_argument('--debug', action='store_true', default=False,
                    help='Turn on debugging')
args = parser.parse_args()

errors = {'LU': [], 'GMG': []}
flops = {'LU': [], 'GMG': []}
l = 5.0
mms = 6
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfA%d' % k
  options = ['-snes_type', 'newtonls', 'par', str(l), '-da_refine', str(k+1),
             '-ksp_rtol', '1e-9', '-pc_type', 'lu', '-mms', str(mms),
             '-log_view', ':%s.py:ascii_info_detail' % modname]
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  errors['LU'].append(float(out[4]))
  perfmod = __import__(modname)
  flops['LU'].append(perfmod.Stages['Main Stage']['summary'][0]['flops'])
for k in range(args.kmax):
  Nx = 10 * 2**k
  modname = 'perfB%d' % k
  options = ['-snes_type', 'newtonls', 'par', str(l), '-da_refine', str(k+1),
             '-snes_max_linear_solve_fail', '10', '-ksp_rtol', '1e-9',
             '-pc_type', 'mg', '-pc_mg_levels', '3', '-pc_mg_galerkin',
             '-mg_levels_ksp_norm_type', 'unpreconditioned',
             '-mg_levels_pc_type', 'sor', '-pc_mg_type', 'full', '-mms', str(mms),
             '-log_view', ':%s.py:ascii_info_detail' % modname]
  cmd = './ex5 '+' '.join(options)
  if args.debug: print(cmd)
  out = subprocess.check_output(['./ex5']+options).split(' ')
  # This is l_2, out[6] is l_infty
  errors['GMG'].append(float(out[4]))
  perfmod = __import__(modname)
  flops['GMG'].append(perfmod.Stages['Main Stage']['summary'][0]['flops'])
```

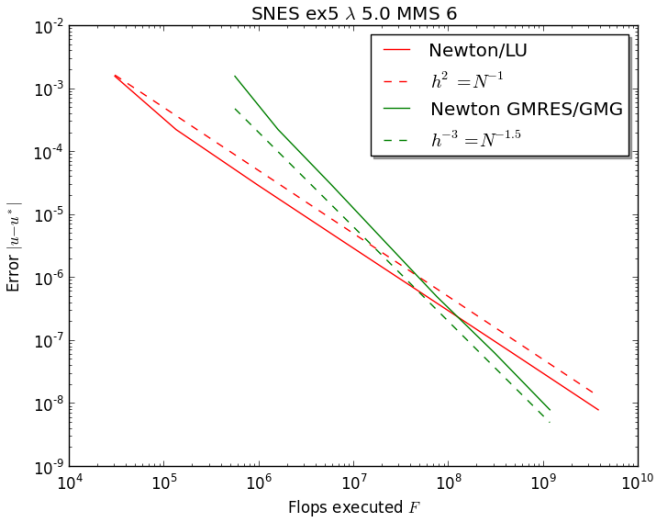


Figure 12.4: Work-precision comparison of LU and GMG linear solvers inside of Newton for the modified Bratu equation $-\Delta u - u^{\lambda} = 0$.

```
FA = np.array(flops['LU'])
FB = np.array(flops['GMG'])
if args.debug: print FA, FB

from pylab import legend,plot,loglog,show,title,xlabel,ylabel,savefig
loglog(flops['LU'], errors['LU'], 'r', FA, 50*FA ** -1., 'r--',
       flops['GMG'], errors['GMG'], 'g', FB, 200000*FB ** -1.5, 'g--')
title('SNES ex5 $\lambda$ '+str(l)+' MMS '+str(mms))
xlabel('Flops executed $F$')
ylabel('Error $\|u - u^*\|$')
legend(['Newton/LU', '$h^2 = N^{-1}$',
        'Newton GMRES/GMG', '$h^{-3} = N^{-1.5}$'], 'upper right', shadow = True)
if args.save:
  savefig(os.path.join('figures', 'fd_2_wp.png'))
else:
  show()
```

and we run it using

```
prob4_2_wp.py --kmax=7 --save
```

to build Figure 12.4. We can see that GAMG is an $\mathcal{O}(N)$ method, whereas the sparse direct solver LU is performing at close to $\mathcal{O}(N^{1.5})$.

12.7 Data Layout and Discretization II

12.8 Simple Finite Elements

12.9 Performance Modeling

Problem IX.1 The following pseudocode is a naive implementation of a dense matrix-vector multiplication (assuming matrix dimension of $N \times N$, vector dimension of N).

C

```
for (i=0; i < N; ++i) {
  double sum = 0.0;
  for (j=0; j < N; ++j) {
    sum += a[i*N+j]*b[j];
  }
  c[i] = sum;
}
```

Fortran

```
for(i=1,N)
  sum = 0
  for(j=1,N)
    sum = sum + a(i,j)*b(j)

  c(i) = sum
```

Based on the above code, answer the following questions:

1. Count the total number of floating point operations, in terms of N. The unit will be a FLOP, abbreviated with F.
2. Count the total number of bytes transferred to/from memory if each floating point number is 8 bytes, abbreviated with B, in terms of N.
3. Compute the arithmetic intensity, meaning the ratio of floating point operations to total bytes transferred, and approximate for large N. (This will give you a value in F/B, where F is a FLOP and B is a byte).
4. If a processor flop rate is 2 GF/s, and memory bandwidth is 8 GB/s, is the program flop rate limited, or memory bandwidth limited?
5. What fraction of peak performance do you estimate can be obtained?

Solution IX.1

1. Each entry of the matrix is multiplied by some element of `b`. Thus there are N^2 multiplies. Each multiplication is added into the `sum` variable. Thus there are N^2 additions. Thus there are $2N^2$ flops.
2. Each element of the matrix is retrieved once. Thus there are N^2 matrix transfers. An element of the vector `b` is transferred for each matrix element. Thus there are N^2 input vector transfers. Each element of the output `c` is written back to memory. Thus there are N output vector transfers. Thus there are $8(2N^2 + N)$ bytes transferred.

   We have not indicated a cache was present for this problem. Suppose that you had a cache. If the cache is smaller than $8(2N)$ bytes, you would get

no reuse, and likely this is the case unless it can hold $8(3N)$ bytes since many caches have a Least Recently Used (LRU) policy for eviction. If it is larger than this, the `b` vector can remain resident as rows of the matrix are loaded, and we would have $8(N^2 + 2N)$ bytes transferred which coincides with the perfect cache model.

3. The arithmetic intensity is the ratio of floating point operations to total bytes transferred

$$\frac{2N^2}{8(2N^2 + N)} = \frac{1}{8(1 + \frac{1}{2N})} \approx \frac{1}{8} \tag{12.37}$$

4. In order to run at peak, the processor would require 2 GF/s $\times$ 8 B/F = 16 GB/s, but it only has 8 GB/s of memory bandwidth available. Thus it is a bandwidth limited computation on this architecture.

5. The process is capable of 8 GB/s, so the computation can run at a rate of 8 GB/s$\frac{1}{8}$F/B = 1 GF/s, or 50% of theoretical peak for this processor.

Problem IX.2 Consider the Gram-Schmidt Orthogonalization process. Starting with a set of vectors $\{v_i\}$, create a set of orthonormal vectors $\{n_i\}$.

$$n_1 = \frac{v_1}{||v_1||} \tag{12.38}$$

$$n_2 = \frac{w_2}{||w_2||} \text{ where } w_2 = v_2 - (n_1 \cdot v_2)n_1 \tag{12.39}$$

$$n_k = \frac{w_k}{||w_k||} \text{ where } w_k = v_k - \sum_{j<k}(n_j \cdot v_k)n_j \tag{12.40}$$

What is

1. the balance factor β for this algorithm?

2. the bandwidth required to run at peak (b_{req}) on your computer?

3. the maximum achievable flop rate ($r_{\max}$) on your computer?

Extra Credit Can this algorithm be improved?

Solution IX.2 In order to make the problem precise, lets first write representative PETSc code for this process

```
for (i = 0; i <= K; ++i) {
  for (j = 0; j < i; ++j) {
    VecDot(v[i], n[j], &proj[j]);
  }
  for (j = 0; j < i; ++j) {
    VecAXPY(v[i], -proj[j], n[j]);
```

```
  }
  VecNorm(n[i], NORM_2, &norm);
  VecScale(n[i], 1.0/PetscSqrtReal(norm));
}
```

For vectors of length N and B_r-byte reals (usually 8), the operations used in the the Gram-Schmidt process have the following computational cost:

- vector norm $||v||$ uses $2N-1$ F,
- normalizing a vector uses $3N-1$ F,
- vector dot product uses $2N-1$ F,
- vector subtraction and scaling use N F.

To produce n_1, we just normalize the vector, so we read in v_1 for the dot product, read it again to scale, and write back n_1, giving

$$\beta_1 = \frac{3N-1}{3NB_r} \approx \frac{3}{3B_r}\text{Ky} = \frac{1}{8}\text{Ky}. \tag{12.41}$$

In order to calculate n_2, we read v_2 and n_1 for the dot product, and again for the subtraction and write w_2, then read w_2 for normalization, and write n_2,

$$\beta_2 = \frac{(3N-1)+(4N-1)}{(3+5)NB_r} \approx \frac{7}{64}\text{Ky}. \tag{12.42}$$

For the kth vector, we have normalization and $(k-1)$ subtractions, dot products, and scales,

$$\beta_k = \frac{(3N-1)+(k-1)(4N-1)}{(5(k-1)+3)NB_r} \approx \frac{4k-1}{(5k-2)8}\text{Ky}, \tag{12.43}$$

and the arithmetic intensity for the entire process, as k becomes large, is very nearly

$$\beta = \frac{1}{10}\text{Ky}. \tag{12.44}$$

My old Mac Air laptop has $r_{\text{peak}} = 1700\text{MF/s}$ and $r_{\text{peak}} = 1122\text{MB/s}$, so that the bandwidth required to run at peak

$$b_{\text{req}} = \frac{1700\text{MF/s}}{\frac{1}{10}\text{Ky}} = 17,000\text{MB/s}, \tag{12.45}$$

which obviously exceeds the capacity of the machine. The maximum achievable flop rate is in fact

$$r_{\max} = 1122\text{MB/s} \cdot \frac{1}{10}\text{Ky} = 112\text{MF/s}, \tag{12.46}$$

which is 6.7% of peak performance.

Extra Credit Suppose instead that we perform all operations with vector v_k at once, namely that we calculate k dot products in a single operation (`VecMDot`) and k subtractions in a single operation (`VecMAXPY`).

```
for (i = 0; i <= K; ++i) {
  VecMDot(v[i], i, n, proj);
  VecMAXPY(v[i], i, -proj, n);
  VecNorm(n[i], NORM_2, &norm);
  VecScale(n[i], 1.0/PetscSqrtReal(norm));
}
```

This would mean that for each round, we would load v_k and k vectors for the dot products, similarly for the subtractions also writing the answer, and after each round, we normalize w_k. The total memory traffic is therefore

$$N \sum_{k=1}^{K} 2k + 5 = 2N \frac{K(K+1)}{2} + 5KN = K(K+6)N. \tag{12.47}$$

The total flops done is the same

$$\sum_{k=1}^{K} (3N-1) + (k-1)(4N-1) = (3N-1)K + \frac{1}{2}K(K-1)(4N-1), \tag{12.48}$$

so that the total arithemtic intensity is

$$\beta = \frac{(3N-1)K + \frac{1}{2}K(K-1)(4N-1)}{K(K+6)NB_r}, \tag{12.49}$$

$$\approx \frac{2K+1}{8(K+6)}. \tag{12.50}$$

We see that for large K, we can approach $\lim_{K\to\infty} \beta = \frac{1}{4}\text{Ky}$ so that we increase performance by more than a factor of 2. This could also happen by calculating in single precision, where $B_r = 4$.

Problem IX.3 Change the performance model for sparse matrix-vector multiplication (SpMV) so that the loads from memory are uncached. How does the dependence on row occupancy change?

Solution IX.3 The common first assumption for SpMV analysis is that matrix data is loaded only once, that vector data is also loaded only once, and that latency to access vector data is negligible. These assumptions are undermined if you permute the matrix using a randomized ordering. The matrix data will still be accessed only once and contiguously, but for large enough matrices the vector data will almost always generate a cache miss (high latency) and the vector entry will be evicted before it is used again. Moreover, typically nothing else on the cache line will be used.

Our model for the memory must be modified for this case, so that we communicate

$$4(m + nz) + 8(mV + nz(V + 1))\text{B} \tag{12.51}$$

which gives an arithmetic intensity

$$\beta = \frac{2nzV}{4(m + nz) + 8(mV + (V + 1)nz)}\text{Ky} = \frac{1}{\left(4 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V} + 4}\text{Ky}, \tag{12.52}$$

as compared to the expression given a perfect cache,

$$\beta_{\text{cache}} = \frac{1}{\left(8 + \frac{2}{V}\right)\frac{m}{nz} + \frac{6}{V}}\text{Ky}. \tag{12.53}$$

The new term is comparable to the matrix term for $V = 1$, and much larger as the number of vectors grows, and also dominates the other vector terms. Thus it will control the performance of multivector multiply. In the limit of a small cache, you get very poor temporal locality for the vector entries when using any matrix ordering, so that the last term will again appear and dominate performance.

Problem IX.4 Run the STREAMS benchmark on your personal computer and graph the results as a function of core count. What do the results tell you about the architecture of your machine?

Solution IX.4 You can run the benchmark automatically using PETSc,

```
make NP=<max cores> streams
```

which on my laptop produced Figure 12.5. We can see that a single core saturates the memory bandwidth of this machine so that the second core does not do any good for bandwidth constrained computations. However, as discussed on StackOverflow, the association of cores to memory may be suboptimal, causing interference or saturating one memory channel at the expense of others. We can use the `MPI_BINDING` variable to tell PETSc what to do,

```
make NP=<max cores> MPI_BINDING="--bind-to␣socket" streams
```

but since my machine has a single socket, the results are unchanged.

References

Graham, Ronald L, Donald E Knuth, and Oren Patashnik (1989). *Concrete Mathematics*. Addison-Wesley.

Jackson, John David (1962). *Classical Electrodynamics*. John Wiley & Sons.

Stodden, V. et al. (2010). "Reproducible Research: addressing the need for data and code sharing in computational science". In: *Computing in Science and Engineering* 12.5, pp. 8–13. URL: http://www.bepress.com/cgi/viewcontent.cgi?article=1034%5C&context=sagmb.

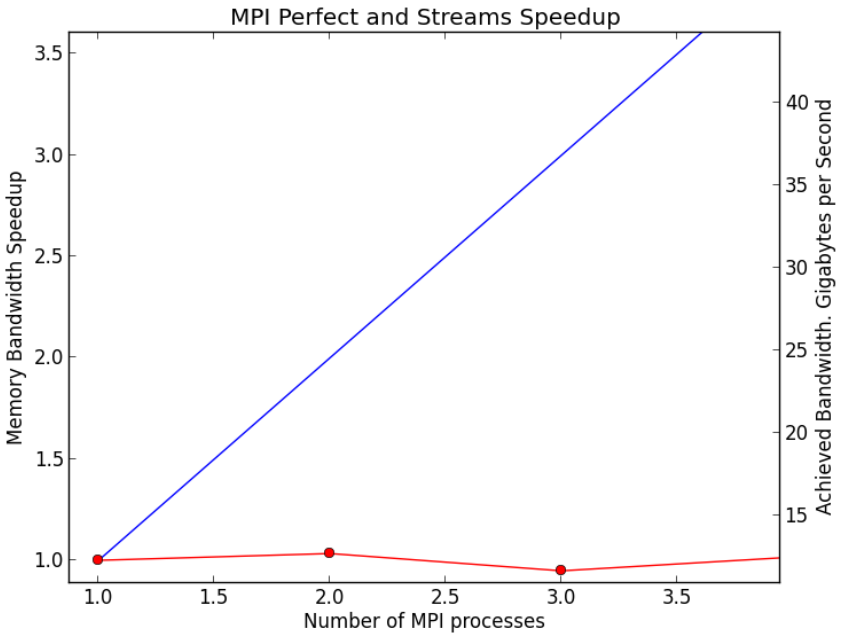


Figure 12.5: STREAMS benchmark run for a Mac Air laptop with an Intel Core i5 processor.

LeVeque, Randall J. (Apr. 2013). "Top Ten Reasons to Not Share Your Code (and why you should anyway)". In: *SIAM News* 46.3. URL: http://faculty.washington.edu/rjl/pubs/topten/.

Anderson, Donald G (1965). "Iterative procedures for nonlinear integral equations". In: *Journal of the ACM (JACM)* 12.4, pp. 547–560.

Fang, Haw-ren and Yousef Saad (2009). "Two classes of multisecant methods for nonlinear acceleration". In: *Numerical Linear Algebra with Applications* 16.3, pp. 197–221. DOI: 10.1002/nla.617.

Trefethen, Lloyd N. and David Bau, III (1997). *Numerical Linear Algebra*. Philadelphia, PA: Society for Industrial and Applied Mathematics, pp. xii + 361. ISBN: 0-89871-361-7.

Stewart, G. W. (2011). *Fredholm, Hilbert, Schmidt: Three Fundamental Papers on Integral Equations*. Translated with commentary by G. W. Stewart. URL: http://www.cs.umd.edu/~stewart/FHS.pdf.