RICE UNIVERSITY

Software Design for Simulation Driven Optimization

by

Anthony D Padula

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Houston, Texas

May, 2005

Abstract

Software Design for Simulation Driven Optimization

by

Anthony D Padula

This thesis describes a flexible framework for abstract numerical algorithms which treats algorithms as objects and makes them reusable, composable, and modifiable. These algorithm objects are implemented using the Rice Vector Library (RVL) interface, decoupling the algorithmic code from the details of linear algebra and calculus in Hilbert Space. I made many improvements to the RVL design, including abstract return types for reductions. These improvements allowed me to demonstrate the breadth of this design by incorporating semantically similar objects from other packages which had significant syntatic differences to the RVL objects. By adapting other libraries, I gain access to a variety of tools, including parallel linear algebra implementations. The benefits of the algorithm framework can be seen when abstract numerical algorithms are coupled with parallel simulators without needing to modify either the algorithm or the simulator.

Acknowledgements

This work was supported in part by:

- Exxonmobil Upstream Research Co.
- the Department of Energy under Contract Nos. 74837-001-03 49 and 86192-001-04 49 from the Los Alamos National Laboratory.
- the Rice Terascale Cluster funded by NSF under Grant EIA-0216467, Intel, and HP.

I'd like to thank the following for assistance with this thesis: William W. Symes, Roscoe Bartlett, Bart van Bloemen Waander, Mike Heroux, Sharon Padula, Denis Ridzal, and Hala Dajani.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis concerns the possibility of writing *reusable* computer source code for high–level scientific computation. Reusability allows high–quality implementations of algorithms and data structures to propagate. This thesis deems code reusable across a set of contexts if it can be used *without alteration at the source level* in any of these contexts. LAPACK is an important example of truly reusable scientific code [1]. LAPACK provides generic interfaces to efficient dense linear algebra algorithms. These interfaces involve only arrays of data, array lengths, strides, and the like, which describe the mathematical objects occurring in dense, in–core computational realizations of solution algorithms for linear systems and related problems — namely vectors and matrices identifiable with in–core arrays. This choice of interface design and data structures makes LAPACK reusable across an entire domain of problems.

The problem domain considered in this thesis is the large, simulation–driven *nonlinear program* (NLP)

$$\min f(x) \text{ s.t. } g(x) = 0$$

in which the objective f and the constraint g may be itself the result of extensive computations, such as the numerical solution of a system of partial differential equations. The solution x is a *vector*, and f and g are assumed to be mappings on a subset of a vector space, and appropriately often differentiable. The solution algorithms which scale well to large problems tend to be coordinate–free, that is, they depend only on the behavior of the mathematical objects defining the problem, and not the particular details which implement this behavior. Some examples of such algorithms are the Quasi–Newton methods and Krylov–subspace methods. The algorithms can be expressed entirely using calculus in Hilbert space with no reference to coordinates. Following the nomenclature in [11], I call these abstract numerical algorithms (ANAs).

Although both the problem definitions and solution algorithms can be written abstractly, there must always be concrete data structures implemented underneath the abstraction. A wide range of data structures occur naturally in simulation driven optimization problems — for example gridded data, finite–element bases, and seismic traces. These data structures have a variety of meta–data, such as times, spatial locations, and grid descriptions, which are necessary for the proper implementation of the simulation. Further, the data may be out–of–core or distributed. The LAPACK–style interface is not sufficient for this problem domain, as no high–level programming language has intrinsic types or the ability to express a small set of concrete types which encapsulate all such data structures appearing in simulation–driven optimization.

However, abstract numerical algorithms can't require access to all the details of the data structures, as they are built around mathematical constructs which don't refer to such details. Computational expression of these algorithms requires a set of types that mimic the attributes of the mathematical constructs out of which the algorithms are built. These types should have only those attributes common to all implementations — insofar as possible those of the mathematical counterparts — if they are to represent reusably all such implementations. Such a type, divorced from implementation and described only by its attributes (behavior), is an *abstract type*. The interface to an abstract type is the code expressing these common attributes. When implementations of ANAs only depend on the interfaces to abstract types, then these implementations are reusable across a variety of storage modalities and problem domain data structures. In addition, concrete data structures implementing an abstract type are reusable across all ANAs written in terms of that abstract type.

The Rice Vector Library (RVL) is a set of interfaces defining abstract types usable in NLPs. RVL addresses the abstract type issue more cleanly than other similar packages in the scientific community. I added a number of refinements to the initial version of RVL, expanding the variety of abstract types it encompasses. In addition, I created an entirely independent ALG library for the abstract expression of algorithms, which complements the RVL interfaces for the abstract expression of NLPs. When combined, these two libraries permit the modular construction of abstract algorithms, which results in reusable implementations of NLP algorithms.

I offer two 'proofs' that the reusability goal is met by my design. First, I will demonstrate the reuse of algorithm code written in RVL/ALG across several problems, data structures, and execution environments. The abstract algorithm code is fully functional *without any modification* in both serial and parallel contexts on a variety of data structures. The particular parallel context described in this thesis is a generic fluid dynamics control problem. I created concrete implementations of the RVL interfaces which are dependent on the parallel environment, but the algorithm implementation depends only on the abstract types and so is independent of the execution environment.

Second, I show how semantically similar but syntactically incompatible libraries can be combined using adapter objects in order to leverage the development efforts of other groups. This reuse complements the first proof by incorporating algorithms and data structures I did not create into the abstract framework. Further, I identify the factors that enable or impede such an adaptation, both in the RVL/ALG design and in the other packages.

Agenda

This project is naturally dependent on the choice of programming language, which makes it important to understand the features of a programming language which aid my goals and how these features were developed as the languages evolved. Although FORTRAN 77 is known to create very efficient compiled code, it does not support abstraction. Other interpreted languages provide support for abstraction, but the interpreters are usually slower than compiled programs for performing a given task. C++ retains much of the efficiency of FORTRAN, as it is compiled instead of interpreted, and supports object–oriented programming. Further, using C++ I can program using mixed languages, retaining FORTRAN 77 for the computational core in order to gain efficiency benefits and reuse existing numerical libraries.

I lay out some general design principles which I've identified throughout the course of this project. I have found several tools, including UML diagrams [46] and design patterns [15] which are extremely helpful in designing good interfaces. Part of the difficulty in design is the lack of a concrete objective — 'a good design' is a vague concept. However, designs which are flexible, composable, and reusable have some common features that can be used as indicators for separating good designs from bad ones. For example, a bad design incorporates methods in an abstract base class which may not be appropriate for all realizations of that interface. Children that cannot implement these methods are expected to print error messages or throw exceptions. Such a design removes much of the flexibility of an abstract interface, as we cannot assume that all implementations of the interface provide the same functionality. A good design includes only the methods which are common across all children in the base interface. Specializations of the interface add additional functionality, creating a tree–structured hierarchy of classes.

The RVL project followed an iterative design process which is common to many software packages, growing out of the Hilbert Class Library (HCL) [17]. HCL introduced several important concepts, including the use of vector spaces as explicit objects in code. However, the authors eventually recognized some fundamental design shortcomings in HCL. RVL carried over the successful ideas from HCL, but was a fresh design from the ground up. RVL capitalized on ISO C++ language features to avoid some of the headaches suffered with HCL. Since my involvement in the RVL project, I've made further improvements to broaden the scope of the package to include a large domain of objects and operations.

RVL is not the only abstract interface package for defining NLPs. There are many competing packages, primarily created at the national laboratories — possibly the biggest producers and consumers of large simulation and optimization codes in the United States. Many packages were designed with a particular application or computing environment in mind, limiting their reusability. I present a critical examination of several competing packages, showing that only a few offer the possibility of potentially reusable code.

Even those packages which do not support reusable code for ANAs (in the sense of this thesis), provide many useful tools written using their interfaces which can be reused — such as parallel data structures, finite–elements codes, and optimization algorithms. In order to access these tools, I adapt their interfaces to RVL. Such an adaptation helps to prove the flexibility of the RVL design by demonstrating it as a sort of superset around the other interfaces.

While RVL provides the interfaces representing the abstractions appearing in NLP algorithms, it does not provide interfaces for the algorithms themselves. Therefore, I introduced an independent library (ALG) of abstract interfaces which permit NLP (and many other kinds of) algorithms to be expressed naturally and modularly as objects. Algorithm objects have many advantages over procedural implementations. Algorithm objects can have persistent internal state and can be instantiated and manipulated at runtime. Inheritance and abstraction can be used to define interfaces which are satisfied by several concrete algorithm implementations that behave differently but solve the same problem (e. g. line searches). My design severs the algorithm steps from the stopping criteria. This separation makes both the algorithm steps and the stopping criteria reusable and modular.

The algorithm interfaces are not restricted to optimization and simulation algorithms. The interfaces are general enough to include utility algorithm constructions, like the Master and Slave algorithms for task–level parallelism. The base Master and Slave algorithms are fully implemented except for the data specific methods, which are left for concrete children to implement. My design encapsulates the problem dependent details away from the generic code for the Master and Slave.

Finally, while task–level parallelism can be an extremely efficient and straightforward method for parallelizing some applications, it is not well suited for all problems. Some cases, for example tight couplings in simulation code, necessitate sharing and communication between processes in the parallel environment. However, the details of such parallelism affect only the low–level implementations of linear algebra and data structures. I show that by adapting a parallel linear algebra library, it is possible to wrap a parallel simulation inside a functional, completely obfuscating the parallelism from the solution algorithm. As an example of reuse, I use an unmodified abstract unconstrained minimization algorithm, which previously had only been run in serial, to optimize this parallel functional. I demonstrate that this implementation is scalable by solving a large problem (more than a million nodes) on the Rice Terascale Cluster.

Chapter 2

History and Background

Before discussing software design and coding issues, I will review the development of the language features which I will be using. Object–oriented languages did not simply spring from the forehead of a computer scientist full–formed. The languages have grown slowly over the past several decades, following an almost biological evolution in response to many factors — adapting, growing, and learning from one another. The software packages discussed later grew in a similar manner, albeit over a shorter time–span.

2.1 Early Functional Programming

The earliest programming was done in machine code, consisting of numerical codes for operations and absolute addresses for data and branch targets. The numerical codes made a program very difficult to read. Absolute addressing made a program extremely difficult to maintain, since one inserted line affects all the following branch targets.

The introduction of assembly code added symbolic names to the language, for both code locations and data storage. Although there is not a unified standard assembly code, most such languages have similar features. Each machine instruction is represented with a short mnemonic string. Data is accessed through direct memory addressing or a variable name which represents such an address. Symbolic labels are permitted to indicate branch targets, avoiding the pitfalls of absolute addressing. Assembly code is translated fairly directly into machine code, with some preprocessing to compute numerical addresses for the variable names and tags. Most assembly languages evolved during the early 1950's as an alternative to interpreted languages, which were easier to use than machine code but also extremely slow [42].

Although assembly code is still widely used and is an improvement over straight machine code, it still leaves much to be desired. The instruction codes are often cryptic acronyms, due to both line length restrictions imposed by punchcards as well as the necessity of saving space in the assembler itself, where RAM was extremely limited. Hennessy and Patterson, the standard textbook for computer architecture, uses a more modern assembly language which is fairly readable, but still requires a reader to mentally translate `BNEZ` into "branch not equal zero" [22]. The programmer must mentally manage register and memory usage, which usually results in a far sub–optimal utilization. Also, as most machines only have integer and floating point registers, any work in other data types (complex numbers or long integers for example) must be simulated in software by the programmer.

The development of floating–point operations in hardware was the death of the early interpreted languages and the primary motivation for the development of FORTRAN. The first version, FORTRAN I, was developed around 1956 [5] and included I/O formatting, six character variable names, `IF` statements, `DO` loops, and user–defined subroutines [6]. Later versions of FORTRAN added independent compilation of subroutines, explicit type declarations for variables, a logical `IF` construct, and many other improvements. FORTRAN 77 became the standard version in 1978 and is still widely used today, although more often through old library functions than for writing new code.

FORTRAN allows a total of six data types: integer, real, double precision, complex, logical, and character. It also permits the usage of arrays of these six native data types. The limited data types along with several other important design decisions (such as not allowing recursive subroutine calls) permitted the creation of very good compilers containing many code optimizations, resulting in very fast code. FORTRAN 77 thus became the performance benchmark for later applications. However, despite this desirable high execution speed, FORTRAN 77 is not always easy to program and maintain. It is difficult to modify existing code to reflect a change in data structure. Such a change typically requires modifications at every level of the code to insert extra parameters in function calls and ensure proper initialization, assignment, and use. A change of norms could require finding and modifying every call to the norm function, especially if the new norm involved parameters besides the arrays of data, such as a scaling factor.

Some computer scientists argue for using LISP as a language for scientific computing [20]. LISP is functional, dynamically typed, allows recursion, performs automatic garbage collection, has a symbol type, and is real–time composable to build new programs from old ones. Graham argues that all other languages have spent the last forty years trying to catch up to what LISP got right in the first place. There may be merit in his position, but several facts remain: One, LISP is usually interpreted and thus usually slower than a compiled language, especially on large data sets (some Scheme implementations are compiled). Two, LISP provides inefficient support for array address calculations — typically requiring one function call per element access — even in compiled code. Three, many consider LISP to be practically unreadable, due to both the unusual operator ordering ($(+A\ B)$ instead of $A + B$) and the huge number of parentheses. Finally, most programmers of scientific applications seem to be uncomfortable with LISP, impeding its widespread acceptance.

2.2 Structured Data Types

In 1959, the Department of Defense sponsored meetings at the Pentagon to create a programming language which resembled English and was easy to use (even at the cost of speed) [42]. Out of these meetings came the Common Business Language (CBL), which eventually evolved into COBOL 60. It was the first language to include a high–level macro construct and long descriptive names. More importantly, it provided the first implementation of hierarchical data structures. Although it is extremely easy to read, it likely would not have survived without the DoD mandate for its use, as the early compilers were extremely slow.

COBOL requires detailed description of every variable in a record. Programmers may specify the number of decimal digits as well as the location of an implied decimal point. This flexibility is great when dealing with files, as it bundles this information with the variable, where it belongs, instead of using extra information in input and output statements. If you decide to display more digits of accuracy, only one change is required instead of finding and modifying every location where a particular variable gets printed. This is arguably one of the earliest examples of proper data encapsulation.

ALGOL went through several design iterations [4]. ALGOL 60 was critical, but the version published in 1969 was the most influential on other languages. The aim of its design was orthogonality — "a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of a language" [42]. ALGOL 68 permits users to combine a few primitive data types to define new data types. These types allow users to organize data abstractions to fit particular problems. The language also permits heap–allocated arrays with variable subscript bounds. ALGOL was a beautiful language, but could be extremely difficult to program.

Most imperative languages since 1969 owe many design features to ALGOL 68. Neither PASCAL nor C added new language features, but instead rearranged features in forms suitable for different audiences. PASCAL was designed as a teaching language, and is thus relatively safe to use. For systems programming, C was preceded by CPL, BCPL, and B. However, B is an untyped language, so all data is simply machine words. Kernighan and Ritchie borrowed from ALGOL 68 to design a new version of B, adding many modern language features like user–defined data types [29]. The creators say that C "features economy of expression, modern control flow and data structures, and a rich set of operators". The lack of type checking was both a strength and a weakness. ANSI C (1999) has a more complete type system than the original Kernighan and Ritchie version. This permits wide flexibility to programmers, but can easily be misused to create obfuscated, unmaintainable code.

Structured data types permit the composition of primitive types into new types more suitable for a particular application. In languages without fixed array bounds, one of the most common structured data type is the pair of the array data with an integer describing its length. Further variations include additional integers to describe the starting index and the increment stride. When data is taken from a file, add to the structure the source filename. The application may also need descriptive information about where the data is from and when it was taken. The size and complexity of data structures can grow rapidly. Further, each change in the data structure necessitates changes throughout a program to properly initialize, access, and copy instances of data.

2.3 Object–Orientation

Structures provide a way to associate related bits of data together into a single item. However, programmers often end up writing methods specifically to manipulate a particular structure. One of the most common is a method to copy data from one instance of a structure to another. It seems reasonable to bundle the copy method to the data structure, which prevents name conflicts. Each object has a method named `copy` which implements the same semantic notion, but hides the syntactic difference between copying objects of different types. This is similar to the operations on intrinsic types, where $a + b$ is implemented a different way depending on the type of a and b, but is written the same for many different types.

Dahl and Nygaard were simulation programmers in the 1960's who found existing languages to be ill suited to their needs. In particular, they wanted subprograms which could be restarted at the position where they previously stopped, known as 'coroutines'. They first developed SIMULA I in 1964, but continued to work on it. SIMULA 67 added, among other things, the *class* construct, which bundled data and related routines [12] [37]. Multiple instances of a class are permitted with their own local data, and a class may contain code to be run when an instance is created.

True object–oriented programming, which includes data abstraction, inheritance, and dynamic type binding, was first seen in Smalltalk [19] [32]. The language provides for the simulation of objects which use messages to communicate with each other. Everything in the Smalltalk realm is an object, and each object has a collection of methods available to other objects. Objects are instances of classes, and Smalltalk provides for inheritance among classes. Smalltalk was never widely used, but was very influential among the language design community.

In the mid 1970's, the Department of Defense initiated another round of language evaluation and design. COBOL had been successful, but was not suitable for the embedded systems applications which compose the majority of DoD contracts. Further, there were over 450 different languages in use for defense projects at the time. Out of a long series of design meetings and evaluation phase came Ada. The resulting language contains many interesting and useful features. Ada uses a `package` to encapsulate the specification for data types, data objects, and procedures, thus allowing abstraction. The language permits generic program units, which C++ users would call templates. Ada also has provisions for concurrent execution, intertask communication, and synchronization. An extension in 1995 added support for inheritance, polymorphism, subprogram pointers, and shared data. However, the sheer breadth of features makes the language unwieldy. Usable compilers took four years to build after the initial language design, and still struggled to handle all of Ada's features. Many critics feel that Ada is too large and too complex, making it difficult to write and debug programs [42].

In 1980, Bjarne Stroustrup began modifying the C programming language to incorporate the object–oriented ideas from Smalltalk and SIMULA 67. Over the next decade, the language grew, and by 1990, Release 3.0 of C++ included templates, abstract classes, multiple inheritance, virtual functions, operator overloading, and reference types [42]. However, C++ maintains the imperative nature it took from C. The result is an extremely powerful language that lacks some of the safety provisions that a modern type system would provide. C++ is almost backward compatible with C, which aided in its acceptance. Further, the availability of good, free compliers helps to promote its use. However, the language is very large and complex, and has been criticized as "more of a collection of ideas thrown together than the result of an overall language design plan" [42]. Despite criticisms, C++ has been widely accepted in the scientific community for object–oriented programming due to its power and flexibilty.

Chapter 3

General Design Principles

I've identified good features of RVL and other projects that I could use to guide further developments in the project. These principles are equally applicable to linear algebra interfaces, abstract numerical algorithms, and linear algebra libraries. Many are apt outside of the realm of scientific computing as well. As such, I've described below these design strategies and principles in order to aggregate in one place these generally applicable lessons.

3.1 Beneficial Design Strategies

Many simple principles can greatly improve the efficacy and usability of a design. Some designers consider such issues from the perspective of the expected user, but fail to spend as much consideration on the perspective of developers. Unfortunately, this can make interfacing a package with others much more difficult, as the adapters must implement one interface using another. Having open–source software is necessary for adaptation, but if the internals of a package are poorly designed and poorly documented, having the source is of little benefit.

3.1.1 Documentation and Naming

On any project with more than one developer, or any large project, it is crucial to document all classes and functions when or *before* they are written, in order to ensure that all users and developers have the same semantic notions. The programming language provides tools for describing syntax, but design and documentation are the only tools for describing semantics. The minimal requirements are short, descriptive comments in code. Along with this, a consistent naming convention with descriptive names is also a great benefit. Names should ideally be unique, memorable, and give an idea of what a class or function does. However, long names quickly become cumbersome. At one point, I considered naming a class `TSFCoreNonlin-NonlinearProblemUnconstrainedTakingRVLFunctional`, which is an utterly ridiculous thing to type more than once. In these situations, acronyms and abbreviations are unavoidable. The longer name was replaced with `NPFOUnconstrained` for the parent unconstrained problem in the `TSFCore::Nonlin` package, and `NPRVLFunctional`. You cannot build a constrained problem from a single functional, so this name is still descriptive enough, and is much easier to manage.

One trick to finding good names is to avoid creativity entirely; Use the names that already exist in the common language of mathematics. Mathematical names have often been around for decades, and already survived the Darwinian trials of peer review. Many authors have already taken this advice. However, this leads to a myriad of classes named `Vector`, all of which are slightly different from the others. Modern programming languages provide us with an excellent tool for dealing with this problem — the namespace. A namespace groups together semantically related identifiers, appending the (optional) namespace name to the identifier. Frequently all class names in a package (say RVL) a put into one namespace (usually the package name). When writing code inside the namespace, the namespace name is implicit and may be omitted (so the `Vector` class may be called either `Vector` or `RVL::Vector`. Outside the namespace, all identifiers are *only* accessible by using the namespace name and identifier together (`RVL::Vector`) unless an explicit `using` declaration is made:

```
using RVL::Space;
using TSFCore;
```

Here, the first line allows the programmer to omit the `RVL::` when using the `Space` class later in the program. The second line efficitively includes *all* identifiers in the TSFCore namespace, allowing the programmer to omit any further use of `TSFCore::`

Namespaces allow the use of common (for instance mathematical) names for different types, in different places. The namespace name can be omitted in many situations, but used to distinguish between objects with the same or similar names. Namespaces become critical when working with multiple packages. They help users and developers keep straight which package's objects are being used at any given moment, and prevent name conflicts. Namespaces are simply a standardized method provided by the programming language which replaces the convention of attaching a package name with an underscore, like `Epetra_Vector`. Instead, the classes `TSFCore::Vector` and `RVL::Vector` can be used in the same program without name conflicts.

Taking mathematical names has more benefits. For example, scientific programmers have somewhat standard ideas of what a vector is, what it should do, and with which other objects it might interact. This serves as a guide to both developers and users. If a developer is considering adding functionality to a `Vector` class which falls outside a standard definition of a vector (the standards vary somewhat between disciplines), then they should require a much stronger justification than convenience for this additional requirement/functionality.

Document Everything.

3.1.2 UML and Design Patterns

The Unified Modeling Language (UML) is described in Fowler [46] as "the (mainly graphical) notation that methods use to express designs." UML is a standard way

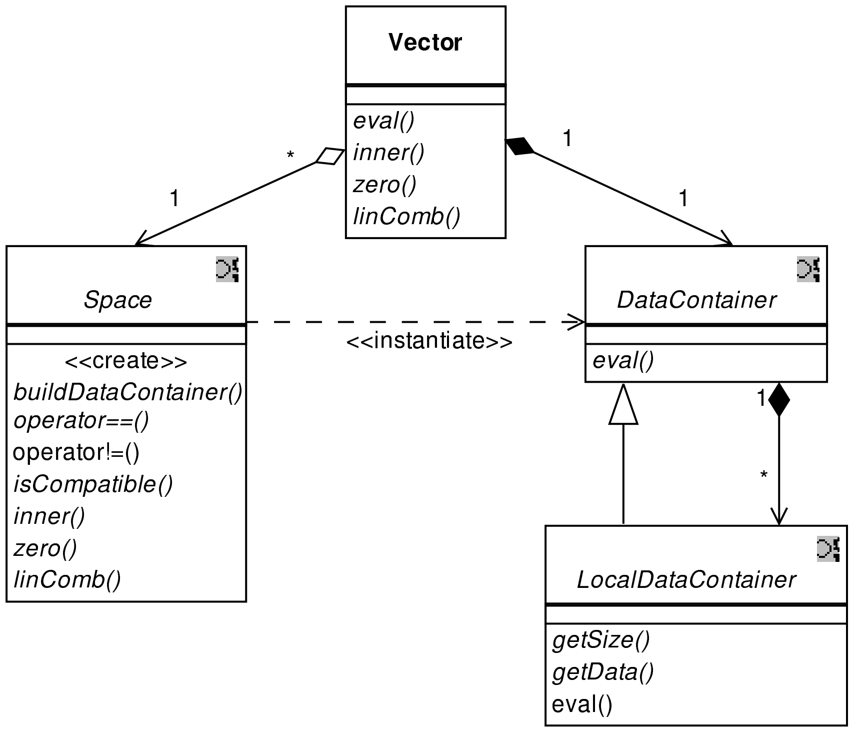

Figure 3.1: The RVL Vector Structure

for diagramming interactions and relationships between objects, types, and packages. UML assigns one meaning to each symbol, so that the same symbols have identical meaning on all diagrams. In my experience, the use of UML as a design tool works very well. Further, UML works best when used *before* coding a design. UML is part of the design stage, not the documentation stage of a project, although the diagrams are useful as part of documentation. Frequently, I found that a design written out can sound good, but when diagrammed, be a horrible mess. Most of the figures in this thesis use UML.

Figure 3.1, reproduced here from Chapter 4, uses UML to show the relationships between four objects, one box per object. The top of the box contains the object name while the bottom shows some of the object methods. Arrows with an open head, such as the one from `LocalDataContainer` to `DataContainer` indicate inheritance, in this case a 'is a' relationship. The arrows with solid diamonds at the base indicate ownership, while the empty diamonds indicate a reference. Thus, `Vector` owns a `DataContainer` and references a `Space` which it does not own. Numbers on arrows show the number of objects participating in each relationship. Any number of `Vector` objects can reference the same space, but each `Vector` owns exactly one `DataContainer`, and each `DataContainer` can only belong to one `Vector`.

This modeling language is easily coupled with a standard set of designs. The book Design Patterns [15] contains a set of patterns which, while not exhaustive, covers the most commonly used patterns in many applications. This book, and others like it, provides us with a new vocabulary for software design. Further, the names selected for patterns are chosen to be descriptive. Thus, I can reasonably make statements like "I used an adapter on the visitor in RVL, FunctionObject, to enable it to act like the RTOp visitor in TSFCore," and have an expectation that the reader understands my meaning, whether they have read the Design Patterns book or not. A *visitor* is an object which visits other objects and does something to them during a visit. An *adapter* serves to adapt one interface to meet the specifications of another interface. The book further provides suggestions on which patterns should be used in conjunction, when to choose a given pattern over another, and what language features to use in implementing a pattern. To aid readers, I will *emphasize* design pattern names when they are mentioned in this document.

3.1.3 Reuse

The recognition of recurring patterns which commonly arise in object–oriented programming leads us to a final piece of general advice: whenever reasonable, use existing ideas, designs, and code. When you create a new vocabulary, you risk alienating potential users and losing your audience. Old code is not always better, but it has been refined in a crucible of use and abuse, which helps to filter out the bugs and inefficiencies. Make new designs in the expectation that they could be used for many years and many different projects. Avoid over–specializing, as it reduces the possibility of reuse.

3.2 Object–Orientation is your friend

Object–orientation is critical in interfacing modern software libraries. Procedural programms are easy to write and compile, but difficult to maintain, modify, or use outside the original scope of the package. LAPACK and the BLAS are widely used libraries for low–level linear algebra. Programmers supply arrays of contiguous data which represent vectors and matrices and use these packages to perform linear algebra operations on data very efficiently. The data must be in contiguous arrays on a single processor in the order that the packages expect. Further, as it is written in FORTRAN 77, only the built–in types are available, and separate but nearly identical implementations must be kept for each data type (single, double, complex, and complex double) or program transformation scripts used to change types. As these libraries are the numerical kernel at the heart of many programs, it is critical that they be extremely efficient. At this level, the different scalar types necessitate different memory access patterns to achieve peak efficiency.

However, for high–level code, there are many details which can be glossed over, as they do not affect the implementation so drastically. For example, an algorithm for sorting lists of data can have the same implementation regardless of the data type as long as an ordering method and copy method for the data type is provided. In the case of optimization code, the code to implement Newton's Method to find x for which $c(x) = 0$ is the same regardless of the particular operator c. In fact, the same code can also be used to find the local minimum of a functional if we treat the gradient as an operator.

Object–orientation provides new tools which make reuse and modification much simpler for high–level code. An abstract interface (say for the operator c) may have numerous implementations, each suited to a different situation, but which share a many common features speicified by the public interface (an operator can be evaluated at a point and has a linear first derivative at that point). As a further benefit, many object oriented languages provide mechanisms for inheritance, thus allowing programmers to reuse code inherited from the parent and only modify small portions in the child class.

Each abstract interface tends to have a tree of children which inherit from it. For example, consider the `TSFCore::Vector` interface. TSFCore is an abstract linear algebra interface which is part of the Trilinos Solver Framework [10]. It uses namespaces to distinguish its objects from similarly named objects in other packages (I denote membership in a namespace with a double colon `Namespace::ObjectName`). `TSFCore::Vector` defines the interface, and also supplies default implementation for some member functions. The package then inherits from `Vector` some base classes, such as `SerialVectorBase` which reimplement some member functions, but are still abstract. Finally, the leaf classes like `SerialVector` fully implement any remaining abstract functions, depending on the particulars of their storage and access schemes. Any of these leaf classes can fill the role of a `Vector` in algorithmic code.

The use of abstract interfaces is an absolute necessity for utilizing many design patterns in [15]. Both the `RVL::FunctionObject` and the `TSFCore::RTOp` make use of the visitor pattern as a central part of their design. While the mechanics in each situation differ slightly, the rough strategy is to specify a abstract interface for a mathematical function of the form $f : X \to Y$, where X and Y are vector spaces. These objects then visit vectors (or collections of vectors) and perform the mapping. A particular function is created by writing implementations for the abstract member methods which were specified by the interface. One simple example from RVL is the identity map called `RVLCopy`. The main method it implemented in the class is the `operator()` method, which performs the visitation. Apart from some array bound checking and such, the core of the method is

```
int n = x.getSize();
for (int i=0;i<n;i++) {
  x.getData()[i]=y.getData()[i];
}
```

Many visitors are quite similar, and a programmer often only changes the one line inside the for–loop to perform the correct mapping. TSFCore's `RTOp` have more methods to implement, in order to guarantee portability and parallel compatibility, but their kernel of code looks remarkably similar. These similarities will allow an intimate cooperation between the packages, which will be discussed in Chapter 6.

The whole point of object orientation is to improve the efficiency of programmers without hurting the run–time efficiency of the code (see Stroustrup [43] section 24.2.4 for a good discussion of this). Given the cost of hardware compared to the total cost of programmers, it is *always* cheaper to buy slightly more hardware than it is to hire more staff. So even *if* object–orientation has slight run–time costs in virtual function calls and such, it is still very beneficial. Properly designed classes can avoid much of the unnecessary run–time overhead and take advantage of the modern compiler's ability to inline and optimize code.

Code cannot be easy to modify if it is difficult to understand what the original code is doing. Documentation for methods should not only describe the details of parameter lists and preconditions, but also what a method is doing. For the whole abstract interface, documentation should describe the interface, what each member method is for, and how this interface interacts with other interfaces. To aid future users and developers, a designer should provide example implementations of abstract interfaces and ensures that these examples aren't just the trivial cases. A lack of documentation and examples may confuse later implementers and may result in the abandonment of an interface.

3.3 Encapsulation

When used properly, object–orientation promotes encapsulation, grouping of related things and controlling access to them. Some of the earliest attempts at encapsulation came from the language improvements which allowed independent compilation of subprograms. A subprogram encapsulates a sequence of statements, allowing the subprogram call to substitute for these statements elsewhere in code. Each subprogram has a fixed interface (which controls access) composed of a list of parameters and a return type. The subprogram has an expected behavior, and when it returns control to the calling routine, the return values and parameters are supposed to be left in a given state. Independent compilation allows subprograms to be compiled and tested individually, more easily than trying to fix an entire program at once. Further, it makes subprograms reusable, as several applications might call the same library of subroutines. This library can be maintained in one location, instead of having to remember all the locations to which a subroutine was copied and updating each one.

However, there are several severe drawbacks to the early procedural programming approaches. Behavior is entirely semantic, leaving the caller dependent on documentation and trust. One common, insidious error in such code is accidentally modifying one of the inputs which was supposed to only be read. Such an error doesn't throw any warnings and frequently shows no evidence until much later in a program. Sometimes, the only indication of such an error is incorrect results at the end of a program. A similar sort of error is accidental modification of a global value that is supposed to be constant. Programmers often define $PI = 3.1415926$ at the top of a program, and a later assignment due to a missing character, such as `if( PI = x/4.0)`, can ruin results.

Modern language features improve the behavior of subroutines and object methods by providing the caller with slightly more control. Read–only inputs can be declared locally constant, which makes modification much more difficult. Data encapsulation limits access and improves upon passing around raw memory addresses. More importantly, abstract interfaces allow us to describe the public face of objects — both data and methods. The interface does not determine how an implementation must do a task, but only describes the tasks to be done. Properly used, encapsulation permits independent fragments of code to be isolated and pieces which perform the same task in different manners swapped without modification of unrelated parts of code.

3.4 Pitfalls in a Software Design

Much can be learned by examining a variety of software packages and discovering which design features work well, and which can actually reduce the functionality. I will take two perspectives in this discussion. The first is from considering the use and development of a stand alone package. I will also discuss this issue as it relates to the interoperability of packages, i. e. does a piece of software play well with others. The later is addressed in Section 6.5.

3.4.1 Too much can be worse than too little

Packages have an inexorable tendency to grow over time. Thus, the only way to avoid becoming the 800 pound gorilla is to start very small. If all of the "wouldn't it be neat" features are added in the beginning, they are impossible to remove later and maintain backward compatibility. Further, such features often turn out to be less useful than anticipated, or can be easily rolled into a more general feature later.

This lesson applies on many levels. On the interface level, it is crucial to limit the number of member functions which are required in the top–level interface. Further, if a member function is deemed to be necessary sometimes, consider if it is necessary and sensible in *all* implementations of an interface. If it is not, I often find that a hierarchical family of interfaces is more appropriate. The most basic interface at the top of the tree contains only the bare necessities of functionality for such a semantic object. The immediate children then have additional member functions, and often implement some of the abstract functions in the parent using the additional knowledge in the child.

One example of this is in the `RVL::DataContainer` class. `DataContainer` is the abstract interface, and has only the four evaluation functions, two write methods (for error reporting and debugging mostly), and a virtual copy constructor called `clone()`.

```
class DataContainer {
public:
  DataContainer() {}
  DataContainer(const DataContainer & D) {}
  virtual ~DataContainer() {}

  virtual DataContainer * clone() = 0;

  /** Evaluate a unary function object. */
  virtual void eval(FunctionObject & f) = 0;

  /** Evaluate a binary function object, with a second DataContainer
      providing the other argument.*/
  virtual void eval(FunctionObject & f,
    DataContainer & x) = 0;

  /** Evaluate a ternary function object, with a second and third
      DataContainer providing the other arguments.*/
  virtual void eval(FunctionObject & f,
    DataContainer & x,
    DataContainer & y) = 0;

  /** Evaluate a quaternary function object, with a second, third and
```

```
      fourth DataContainer providing the other arguments.*/
  virtual void eval(FunctionObject & f,
    DataContainer & x,
    DataContainer & y,
    DataContainer & z) = 0;

  /** report to exception */
  virtual void write(RVLException & e) = 0;

  /** report to ostream */
  virtual ostream & write(ostream & str) = 0;
};
```

Some children of `DataContainer` are specializations, but still abstract. One such is `ProductDataContainer`, which implements the write and evaluation methods, but has abstract functions for accessing the subcomponents of the product structure. Thus, a `ProductDataContainer` is a `DataContainer` which has a collection of `DataContainer` objects inside it. This is one example of the *composite* pattern in [15].

Other methods might have been included in this interface. The `DataContainer` interface could have owned `getSize()` and `operator[](int i)` member functions instead of the specialized product class. Although the default implementation could return 1 for the size and a reference to `*this` for the access function, this would needlessly clutter the interface. Further, it would confuse naming issues elsewhere, since some other subclasses like `LocalDataContainer` also have a `getSize()` method with an entirely different semantic meaning. The current design forces a user to explicitly state whether they are working with a `ProductDataContainer` before trying to call `getSize()` and avoids confusion with other methods.

3.4.2 Code Obesity

It is easy to get carried away in the thrill of creating new classes and types. This can rapidly lead to packages with hundreds of similar classes. More classes result in increased complexity and increased confusion for users and developers. Further, it makes documentation and maintenance much more difficult.

Part of the trouble can occur when a package experiences geometric growth from adding new features. Instead of a single new object with the new feature, new versions of an entire set of objects are added. With just a couple new features, this enormously increases the number of objects. In this situation, it is important to consider two things:

1. Are the new features truly necessary and are all the possible combinations needed?
2. Can the features be implemented once and then reused as needed?

The *visitor* and *strategy* (a family of interchangeable algorithms) patterns can be a big help in situations where the features are functionality. To limit differences in data types and accessing, use of *templates* and *Mixins* (small interfaces for adding functionality) can be handy. I've found templates especially useful to avoid maintaining different copies of code for different scalar types. In HCL, the precursor to RVL, there were two nearly–identical copies of code, one for single precision and the other for double precision, whereas RVL has only one templated interface which suffices for any scalar type.

3.5 Summary of General Design Principles

These general principles should be taught to every beginning programmer. In an effort to promote awareness, here's the shortened form of this chapter:

- Document all code, preferably before writing, and maintain documentation at the same time as maintaining code.
- Choose descriptive class names, preferably based on the names of familiar mathematical or problem specific objects.
- Choose descriptive method names. Establish and follow a naming convention (e. g. I name all copy method `copy()`).
- Choose descriptive identifiers. Naming a gradient vector `Trogdor` is not descriptive.
- Use namespaces to encapsulate logically related classes and procedures. At the very least, use the package name as a namespace around all code in the package.
- Use a high–level language for high–level ideas.
- Reuse numerical kernels (often written in FORTRAN) to leverage their efficiency
- Reuse the work of experts. They have spent time developing and testing good code.
- Learn the common nomenclature of software designers. UML and Design Patterns are good tools for communication.
- Diagraming a design before coding will often reveal flaws.
- Use Object–orientation for high–level code.
- Encapsulate independent fragments of code as objects, methods, or procedures.
- Visitors permit users to add functionality without modifying data structures.

- Templates are a double–edged sword. They can allow reuse, but can also spread quickly throughout a package. Templated code cannot be compiled into an archive. Use them carefully.

- Compile–time checks are always better than run–time checks.

- Some programmers strictly recommend a debugger for finding bugs. I prefer using try/catch blocks to create a calling sequence whenever an exception is thrown.

- Declare variables, parameters, and return values constant whenever it makes sense to do so (the item is semantically constant). The `const` declaration in C++ prevents some accidents but cannot prevent intentional maliciousness.

- Let the compiler optimize your code. It will almost certainly result in faster code.

- *Never* use global variables.

Chapter 4

RVL

In one form or another, the Rice Vector Library (RVL) has been an ongoing project at Rice for the better part of a decade. The authors, originally William W. Symes and Mark S. Gockenbach, were tired of reimplementing the same basic solution algorithms for new data types and hacking apart old code to accommodate new data types. While large sections of code could be cut and pasted together, it was always a struggle to isolate the fragments which could be left intact from those that required careful modification. Further, such software design practices led to multiple copies of very similar code which had to be maintained independently.

The Hilbert Class Library was their first attempt at developing interfaces to layer between data storage code and algorithmic code [17]. They saw that most of their applications involved objects which could be viewed as living in a Hilbert space or mapping between Hilbert spaces. They used the mathematical concepts that defined a Hilbert space and the vectors in the space to guide the design of their interface.

The HCL project was somewhat successful, but, as frequently happens in software design, the authors noticed some inherent shortcomings of the design. Rather than try to make radical changes to overhaul the library, it seemed more beneficial to start again using the lessons learned from the first attempt. The Rice Vector Library is the successor to HCL. While it is not backward compatible with HCL, it draws many features and ideas from its predecessor.

4.1 HCL

The Hilbert Class Library [17] is a collection of linear algebra interfaces originally constructed to facilitate writing optimization code which was not tied to a particular data structure. The interfaces describe objects for performing linear algebra in Hilbert spaces. The final package became cumbersome as it strove to meet the needs of a variety of applications and was eventually replaced with RVL (see Section 6.2.1.2).

The core classes in the library are `HCL_VectorSpace`, `HCL_Vector`, and `HCL_Functional`. The vector space interface acts like a set as well as a *factory*. As a factory, it can generate vectors in the set, but it can also test for membership and test for equality with other vector spaces, which are not typical of factories. The membership and equality tests are mostly for error checking, but allow detection of some errors before computation is attempted. The equality tests are implemented with overloaded operators, allowing very natural looking tests such as `if ( U != V )`. The `HCL_VectorSpace` provides major benefits as an abstract factory, allowing the dynamic allocation of appropriate workspace inside algorithmic code without the algorithmic coding needing a concrete type of vector.

The `HCL_Vector` is a very large abstract interface. In addition to the expected linear combination and inner product methods, the vector class has a collection of over fifty operations, from simple vector addition to many more complicated operations. A method for accessing individual elements directly is provided, but its use is not recommended due to possible inefficiencies. At the least, use of this direct access would incur a virtual function call per element access. In the case of a out–of–core vector, the method might have to retrieve each element from a file, which would be extremely inefficient.

Due to the large number of virtual class methods, implementing a child of

`HCL_Vector` requires a lot of new code. Several examples of concrete vectors are provided as part of the package, including the basic, in–core `HCL_RnVector` and an out–of–core `SGFVector`. However, it is necessary to maintain both a single and double precision version of all code, and a script is provided to facilitate this by essentially performing a global search and replace on the scalar type (I later asked one author why they didn't use a user–defined scalar type which could be altered in a single location, and it simply didn't occur to them).

The mapping interfaces `HCL_Functional`, `HCL_LinearOp`, `HCL_BiLinearOp`, and `HCL_Op` use the vector and space interfaces to define domains and ranges and describe inputs and outputs. These mappings provide access to first and second derivatives where appropriate. All of these classes can also define an evaluation object, which pairs a copy of the object with a target vector in the domain, acting a bit like a *memento* of the object at a specific point. The evaluation then has an independent life from the original mapping and permits efficient reuse of intermediate quantities used in calculating the values and derivatives. Algorithms may also hold on to an evaluation for later use to avoid recalculating values.

Several concrete *composites* and *facades* can be used to manipulate functionals and operators. An example facade is the `HCL_LeastSquaresFncl` which calculates $\|F(u)\|$ from an operator $F(u)$, implementing one interface (functional) using one or more other interfaces (here operator, vector, and space). The various linear combination $aF(u) + bG(u)$ classes for both operators and functionals are good examples of composites, combining two objects F and G to produce an object of the same type. These are very useful tools, as the chain rule may be applied to calculate derivatives, resulting in fully implemented, concrete composites. *Composites* are also used for data structures, such as the `HCL_GenericProductVector` which implements a vector in a product space and works for any product of existing spaces.

Algorithms implemented in HCL are also defined as objects. Abstract base classes are given for linear solvers, line searches, and unconstrained minimization. Each class has a `Parameters()` method for accessing and changing scalar parameters. The solvers can `Solve()`, the line searches `Search()` and the minimizers `Minimize()`. The concrete implementations of such algorithms demonstrate the capabilities of HCL for implementing algorithms free of the details of data storage.

The authors of the HCL paper provide timing comparisons between their HCL implementations and well–known FORTRAN implementations of Limited–Memory BFGS and Implicitly Restarted Arnoldi algorithms. The results demonstrate that there is very little overhead relative to the total runtime involved in the HCL implementations, despite the virtual function calls. As the problem size increases, this overhead is rapidly swamped by the work done inside the critical loops of the computation.

HCL is object–oriented, but many improvements have been made since its inception, and it was easier to learn from HCL and start from scratch on a new project, rather than trying to remain backward compatible. RVL resembles HCL in the semantics of its objects, but owns a deeper hierarchy and has many improvements over its predecessor.

4.2 RVL 1.0

The design principles discussed in Chapter 3 were applied by Symes, et. al. when creating the Rice Vector Library (RVL) for calculus in Hilbert spaces [38]. It is a hierarchical design in two directions, having three levels of data–storage objects and their factories. Further, every interface is built with an abstract base class as the parent and a tree of children. The leaves of this tree are the implemented children or specializations of children. The *composite* pattern is often used to build product structure into classes. Finally, recent changes have increased the use of mix-ins, additional interfaces which are added through multiple inheritance. These serve to simplify many of the hierarchies and clarify the roles of some classes.

4.2.1 Vectors and Spaces

The central concept in RVL is the pair of `Space` and `Vector`, as seen in Figure 4.1. `Space` defines an interface for Hilbert spaces, containing methods for creation of elements, membership tests, and the necessary linear algebra operations. `Vector` is a concrete class which requires a `Space` to instantiate and then acts as a vector in that space. The *facade* design pattern is an object which provides "a unified interface to a set of interfaces in a subsystem [15]." `Vector` is a *facade*, acting as a unified interface between the `Space` and data storage. This maintains the link to the `Space` and protects the user from dynamically allocated data. Data is accessed indirectly through the `Vector` by using a *visitor*, representing an operation to be performed, called `FunctionObject`.

When constructed, `Vector` assumes ownership of a `DataContainer` (DC) created by the `Space`. The `DataContainer` class is an abstract interface whose sole purpose is to be visited by `FunctionObject`. The `DataContainer` works by owning a collection of `LocalDataContainers`, which are encapsulated contiguous arrays of scalars. `DataContainer` passes each of its `LocalDataContainers` to the `FunctionObject` in turn. The `LocalDataContainer` has `getData()` and `getSize()` methods which `FunctionObject` uses to implement a its functionality. For example, the L_2 inner product on v and w contains the code

```
int n=v.getSize();
Scalar * pv = v.getData();
Scalar * pw = w.getData();
Scalar raw = 0.0;
for (int i=0;i<n;i++) {
  raw += pv[i]*pw[i];
}
ip += scale*raw;
```

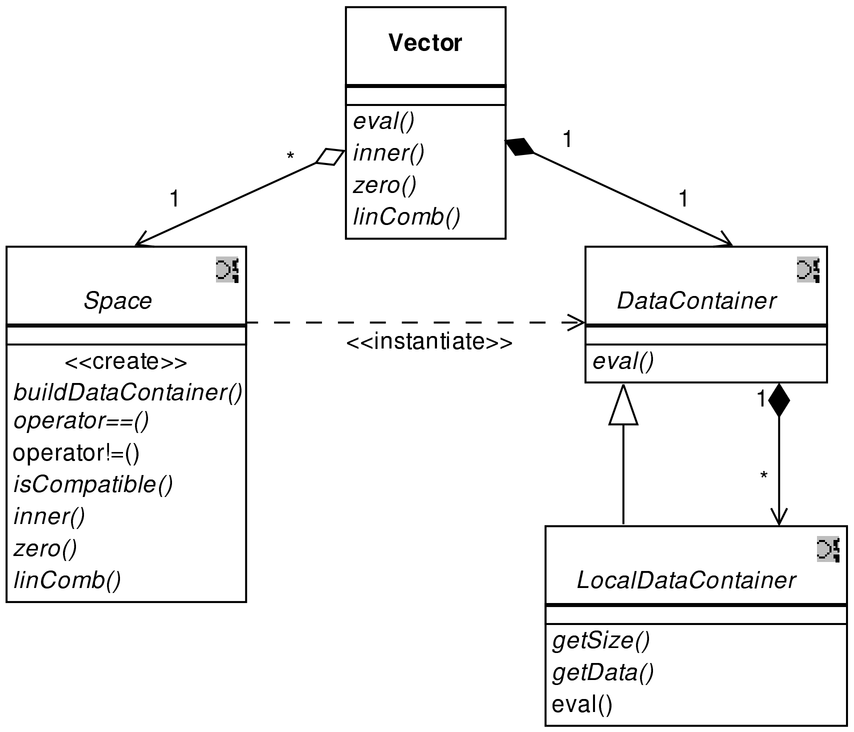

Figure 4.1: The RVL Vector Structure

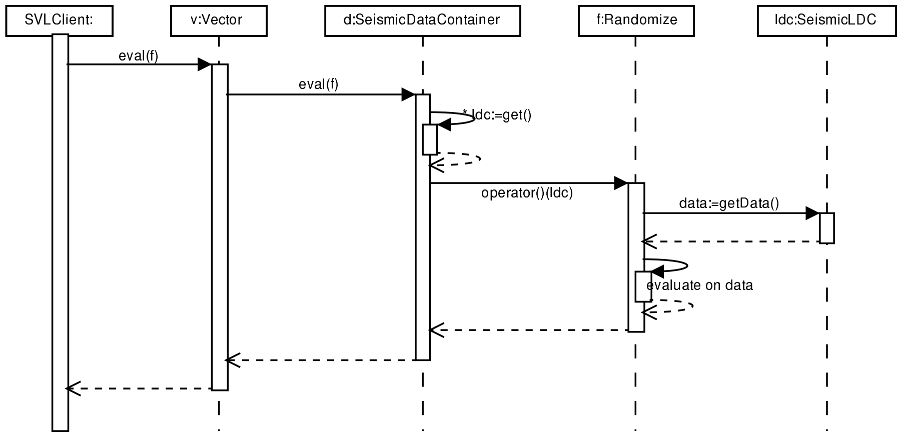

Figure 4.2: Evaluating RVLRandomize on a Vector

Here, the `[]` operators are raw array accesses, and thus cheap. The scalar *ip* is a data member of the function object which is saved for later access. Another example of a visit can be seen in Figure 4.2, which shows the process to randomize a seismic vector.

At first, these relationships can be confusing. All large numerical computations involve arrays of scalars at some level. These arrays are described by the `LocalDataContainer` (LDC) interface. Each array has at least a pointer to the start of the array and an integer number of elements. A specific LDC may have other attributes as well. A `FunctionObject` (FO) performs computations by accessing the elements of one or more LDCs. Specialized function objects may also access the extra attributes of specialized LDCs. The `DataContainer` is a group of one or more LDCs. which is visited by FOs. The FO is invoked in order on each member LDC of the DC. The `Vector` owns exactly one DC and evaluates FOs on it.

The introduction of the `Vector` class as a concrete *facade* instead of an abstract base class is one of the primary contributions of the RVL package. The package completely avoids reference–counting pointers and all that baggage by hiding dynamic allocation of data containers inside `Vector` and explicitly disabling the `new` operator on `Vector`. It is still possible for a determined user to dynamically allocate a `Vector` through subclassing, but the disabling is enough to encourage most algorithm writers to focus on static allocation of workspace and states.

4.2.2 FunctionObject

The base FO class requires very little of its children. FOs must respond to queries about their use of data — whether data is read from or written to. Each FO should have a unique name, which helps to identify them when handling exceptions. The class has two virtual `write()` methods for which default implementations are provided but which may be overloaded by children. The methods describing whether data is read or written are new additions to the class, justified in Section 4.3.1.

The abstract specializations of the base class add the important `operator()` methods which do the actual computational work in a FO. The `Unary, Binary, Ternary, and Quaternary` versions each have an `operator()` method with the appropriate number of parameters. While there is the occasional need for more parameters, I have found it usually more convenient to build a `UnaryFunctionObject` which expects to be called on a series of inputs. I considered crafting a function object which took an arbitrary number of inputs (which I usually refer to as a `NaryFunctionObject`), but I have not found sufficient need or justification to do so.

The `operator()` methods are the main computational methods. For example, the binary function object has the interface which accepts two local data containers as parameters:

```
void operator()(LocalDataContainer<Scalar> & x,
                LocalDataContainer<Scalar> & y)
```

Most exisiting examples of FOs follow a pattern where the first parameter is the result of the computation and all other parameters are inputs. However, as this is not always the case, the `bool readsData(int i)` and `bool writesData(int i)` methods return whether the i^{th} parameter is read or written. Note that, following C++ conventions, the first parameter is $i = 0$. I strongly suggest examining the file "functions.H" to see the implementations of several examples of common function objects.

While there is nothing preventing a FO from having data members and additional methods, the basic interface is meant for data operations between local data containers. However, frequently scientific computations involve operations which reduce one or more data containers to a single piece of data or small data structure. Some of the most basic examples of this are the norms and inner products typical of linear algebra in Hilbert spaces. RVL provides a standard set of interfaces for reductions. Each has a pure virtual method `getResult()` which returns the result of the reduction when called. The implicit assumption is that the reduction type is something which can be passed by value, small enough to be copied without significant overhead. The assumption matches preconceptions of the result of a reduction and does not seem overly restrictive.

4.2.3 Operator and Evaluations

In addition to the above mentioned interfaces, there are several higher–level interfaces which use `Space` and `Vector` to describe common mappings. A `LinearOperator` implements both the forward mapping between the domain and range and the adjoint mapping. An `Operator` also has a domain and range, but in addition to the forward mapping, implements the derivative and adjoint of the derivative. A `Functional` is similar to an `Operator`, but its range is a scalar field and it can evaluate first and second derivatives along with the forward mapping. These mapping classes are usually implemented using `FunctionObjects` and serve to encapsulate a collection of FOs and appropriate spaces together in a convenient bundle. While the base interfaces are not explicitly *facades*, the implementations often are. Only a few methods need to be implemented to create a new functional or operator. In both cases, there are three `apply` methods, a `clone` method, a `domain` method, and two `write` methods. The `Operator` additionally has a `range` method, which is omitted in the `Functional` class since the range is assumed to be the scalar field.

One of the ideas which survives in some form from HCL is the use of evaluation classes, which represent the jet of a mapping at a given point. However, the RVL implementation improves their functionality. For `Functional` and `Operator`, the evaluation is now a concrete class and does *not* have to be reimplemented for each new operator. Instead, the evaluation maintains an independent copy of the mapping with its own internal state along with a reference to the point at which it was evaluated. In the case of `Operator`, the evaluation class provides access to a `LinearOp` which represent the first derivative at the given point. For `Functional`, the gradient is a vector, and an evaluation fx will write the value of the gradient into an input vector $dfdx$ when called `fx.getGradient(dfdx)`. The functional evaluation will also allow access to a linear operator which represents the Hessian at the given point. In both cases, a concrete linear operator is implemented as a friend class to directly access the functionality of the mapping. This provides an object which can be treated as a regular linear operator, but which does not incur any additional storage costs apart from a single reference.

4.2.4 Generic Programming in RVL

A major headache in HCL was maintaining both a single and double precision versions. It was extremely easy for the versions to diverge accidently, then difficult to merge them later. Some authors avoid this problem entirely by only providing one numerical precision [24]. However, this can drive off potential users, as one scalar type is not sufficient for all applications. A crude, but frequently used, solution is the declaration of a local scalar type. This `typedef` can be modified to alter the scalar type for the entire package. However, this solution requires hacking the source files for the library in order to change types and only permits a single type to be used in a given application.

The C++ standard [43] provides a method for generic programming which it calls templates. RVL latched on to this capability eagerly, primarily to avoid the precision headache from HCL. All major interfaces were templated on the scalar type. `DataContainer` and `LocalDataContainer` use the templated scalar for data storage, `Space` and `Vector` use it also to describe the scalar field for linear combination. The other classes in RVL are often forced into templates because they interact with templated types. One other use of templates is to describe the return value of reductions in the function object interfaces. Frequently, this return value matched the scalar data type, and standardized versions like `UnaryFunctionObject-ScalarRedn<Scalar>` were created to describe this common case.

The templated interfaces allow the reuse of the same high–level code on any scalar type. Naturally, different scalar types may require different concrete implementations of some function objects, which is easily accomplished using template specialization. For example, in order to use the BLAS `axpy` procedure to implement linear combinations, four different speicializations of the linear combination ternary function object are needed — one for each of single and double precision real and complex data. Similar template specializations occur whenever procedural numerical kernels are used to implement templated interfaces. However, templates help to isolate the minor modifications (e. g. changing `saxpy` to `daxpy`) in the specializations and avoid maintaining multiple nearly identical copies of code simply to handle different scalar types.

4.3 Modifications to RVL

Since its inception, RVL has been undergoing changes and improvements. Most of these changes are the result of observations while trying to use the package and feedback from other developers [38]. The package described in Section 4.2 is a rough snapshot of the initial ‘release’ of RVL. Below I will detail the modifications to RVL which have occurred since then, along with the reasoning behind each change.

4.3.1 Improved Evaluation Behavior

The following discussion is equally applicable to the `OperatorEvaluation` class, but for brevity I will discuss `FunctionalEvaluation` primarily. One additional change occurred in just the functional evaluation class to make it behave more like the operator evaluation class and help ensure accuracy of results. I modified the `FunctionalEvaluation::getGradient()` method to now return a reference to a vector containing the gradient. This also allows us to avoid some unnecessary gradient calculations, as the evaluation now owns the vector and only recomputes it when needed. However, it is *not* a good idea to save a reference to the gradient, which may also occur when passing a reference to the gradient as a parameter in a constructor, because simply accessing the gradient vector does not ensure that it is updated correctly. The gradient is locked to prevent someone from accidently overwriting it with incorrect values, but the locks cannot prevent users from reading outdated data.

As I and others began writing algorithms, it became apparent that evaluations were useful in some algorithms, but clumsy in others. For iterative algorithms that need to compare the current state and function values to that of a previous state, the evaluations worked quite well:

```
FunctionalEvaluation<Scalar> * fx, * fxprev = NULL;
fx = new FunctionalEvaluation<Scalar>(f,x);
do {
   if( fxprev ) delete fxprev;
   fxprev = fx;
   // update x somehow
   fx = new FunctionalEvaluation<Scalar>(f,x);
} while( some expression not satisfied )
```

Notice the call to create a `new FunctionalEvaluation<Scalar>` inside the loop. This means, for each iteration, the algorithm frees some memory for a vector and a functional and then allocates memory of exactly the same size. Although there is always some cost of copying the `Functional`, it was designed to be lightweight until one of the `apply()` methods is called. Further, creating an evaluation is going to involve at least one assignment to the vector being evaluated, typically an update of the sort $x \leftarrow x + \alpha dx$. However, it is unecessary to delete and allocate a new vector in the same space.

The evaluations work for iterative algorithms resembling the one above, but such use requires dynamic memory management in algorithmic code and wastes time re–allocating the same memory repeatedly. Instead of needing to throw evaluations away and generate new ones for each new point, I could add some sort of `update()` method which would overwrite the current point with new data and clone the functional. This sounds good in principle as it eliminates the need for dynamically allocating evaluations, but still performs a vector copy. Further, there isn't an easy way to allow read–only access to the current point x of an evaluation $f(x)$. Since a `Vector` owns a pointer to a data container, `const` only prevents modifying the value of the pointer, not the values to which it points.

I settled on a better solution. It requires a little more care on the part of a programmer, but is more useful and more efficient. Instead of evaluations owning an independent copy of a vector, they now own a reference to a particular vector. A small amount of hidden mechanism had to be added to the `Vector` class to monitor when its data has changed. The evaluation then notices when a vector has changed since the last call to the functional. If there is a change, the functional is cloned, the old copy discarded, and the evaluation begins afresh. This has the effect of making an evaluation act as $f(x)$, the value of the functional at a given variable x, instead of $f([1,2,3,4,5,\ldots])$ or whatever the value of the variable was at the time of instantiation. The evaluation now acts to encapsulate the vector and functional and ensure synchronization. This permits use of evaluations more like reference parameters. The `getPoint()` method returns a reference to the vector and allows callers to modify the values of this vector if they wish. This avoids the necessity of passing both vectors and functionals to algorithms by instead passing an evaluation as a reference. In algorithms like a line search, I give the line search an evaluation and expect that when the search is done, the evaluation will contain the results of the line search. The earlier design required me to pass vectors back and forth, and the calling algorithm would almost always immediately reevaluate $f(x)$, despite the fact that the search must have already evaluated that point before terminating there. The new design does not use any extra copies of vectors, saving both memory and flops.

As part of this change, I added two new methods to the base `FunctionObject` class. Each FO knows whether it reads or writes any given LDC. I decided this information should be available outside the FO. I added two methods

```
virtual bool readsData(int i) { return true; }
virtual bool writesData(int i) { return true; }
```

to the base class. By default, every FO claims to read and write every LDC. However, concrete subclasses are strongly encouraged to override these methods with more accurate information. This provides several benefits:

1. Cheap monitoring of vectors to avoid unnecessary updating of evaluations.
2. Avoidance of unnecessary reads and writes in out–of–core data structures.
3. Potentially reduce network traffic in some distributed applications.

The drawback to this approach is that a subclass can either make mistakes or lie about its behavior, leading to data errors which are difficult to track down.

4.3.2 Adding const

In C++, an identifier can be labeled as `const`, short for 'constant', which indicates that its value cannot be changed in the current scope. Variables, parameters, and return values may all be labeled as constant to try to prevent accidental modification. One of the major criticisms of the early version of RVL was the complete lack of the use of `const` to ensure data safety at compile time. The original authors' chose to omit `const` due to the fact that `const` may be cast away at any time, and thus does little to prevent maliciousness. Further, in any class with pointers as data members, the language only specifies that the value of the pointer must remain unchanged. The constant nature of the owner does not get passed on to data it owns dynamically. Thus, a `const Vector` does not ensure that the low–level data will not get changed. This makes it frequently easy to follow the letter of the law without any of the spirit of it. It is possible to make a pointer or reference to a constant object, but such an object could *never* be modified, which does not suffice for objects which must be temporarily constant sometimes but non–constant at other times.

Further, when working with other packages (see Section 6.5 for a discussion of the effects of `const` on adaptation), I noticed that `const` propagates very rapidly, especially from a caller to a callee. Use in one place can force use in other places and force changes in otherwise correct code. Class methods must also be made `const`, which forces some data members to be made `mutable` to continue functioning correctly. This is especially true of boolean flags and the data structures for intermediate computations, where memory allocation is often delayed until needed.

When used properly, `const` has some value. Some authors [43] [11] recommend *never* using a non-constant reference parameter. Instead, they suggest always using pointers for parameters whose values may change. Part of the argument is that doing so differentiates the variables. It helps the compiler to catch some mistakes, since the compiler will complain if it is given a non–pointer value where it expects a pointer. However, pointers in general are subject to some difficulties. Procedures must always check if pointer parameters are null before using them to avoid segmentation faults, whereas reference parameters *must* be initialized by the caller before a call can be made. Also, pointers encourage the use of dynamic memory, often where it isn't needed. Pointers have their uses, but should be restricted to situations where they are needed, rather than as a crutch to avoid proper documentation and good software design. It shouldn't be necessary to ponder which parameters may get overwritten and which may not. The design of the function interface should make such questions abundantly clear. Parameters which are `const` won't get overwritten. Further, we try to always make the outputs clear. Usually, they are either the first parameter of a method or the object who's method is invoked. In RVL, the few exceptions to this are not in the public interface.

Asserting that variables and immutable parameters are constant is useful. The available methods on such objects must also be labeled `const`, which helps to differentiate query methods from action methods. Spaces are nearly always constant, since we naturally assume that a given space doesn't change during the course of a program. The methods of the space class are not meant to modify the space, but mediate interactions between members of the space.

Dr. Symes and I made a concerted and careful effort to add `const` where appropriate in RVL. Although our design choices may differ from those of others, we feel we have good reasons for these choices and have considered this issue well enough.

4.3.3 Removing unneeded templates

Generic programming can be a huge benefit to a software package, beyond the obvious use to alter precision of computations. However, it also has some drawbacks. New converts to generic programming often go overboard in templating classes. One major drawback is that templated classes cannot be compiled until they are instantiated. When instantiated, they are then compiled *only* for the particular value of the template parameter. This means that templated classes cannot be put in library archives. They must be included in header files, which leads to longer compilation times.

The other drawback to templates is the complete lack of type information in them. C++ does not permit casting a type to a templated base type unless you know the proper template parameter. For example, given a `DataContainer` pointer, we cannot cast it to a `LocalDataContainer` unless we choose a scalar type. This is particularly a problem when interfacing with other packages which use pointers, especially `void` pointers (see Chapter 6 for discussion of adapting to some of these packages).

In RVL, there were several incidents of templates which had been added unnecessarily. The first was in the abstract `DataContainer` class. This class is a holder for one or more local data containers and a target for visitation by function objects. However, the abstract base class makes *no* mention of local data containers explicitly, much less the scalar type of the data they contained. Further, `DataContainer` is visited by abstract function objects. It does not care whether they are unary, binary, or otherwise. The abstract interface involves only `DataContainer` and `FunctionObject`. The base `FunctionObject` interface also does not need a scalar type. It only has five methods. Two of them take integer inputs and return booleans, one returns a string, and the other two are `write()` methods.

Removing the `Scalar` template from `DataContainer` has some benefits. This process made me realize that `LocalDataContainer` is a `DataContainer`, and arrange the inheritance appropriately. Second, I saw that the scalar type in the local data container and specific function objects is not the same as the scalar type in the `Vector` and `Space` classes. A Vector Space is a set of elements along with a scalar field and a linear combination operator. The scalar template parameter in `Space` is the scalar field for linear combination. The set of elements is the collection of all possible `DataContainers` a `Space` may produce, and does not depend upon the template parameter of the space. Thus, a `Space` may generate data containers which have local data containers with a different type than the one on which the space is templated. If `DataContainer` were templated, I would be forced to choose between adding a second template to the `Space` interface or requiring that the scalar field and storage type of the data container be identical. Instead, the non–templated `DataContainer` better reflects the nature of this class as an element of a set.

Another occurrence of incorrect templates was in the definition of the reduction function objects. It always bothered me that none of the abstract reduction classes, like `UnaryFunctionObjectRedn`, were related to each other except as function objects. `UnaryFunctionObjectRedn` inherits from `UnaryFunctionObject` which inherits from `FunctionObject`. The binary, ternary, and quaternary versions are the same way. All of the reduction specializations add exactly the same methods. Since they are they same methods, why are they declared four times instead of just once? In addition, these interfaces template the type of the reduction value as well as the scalar data type, so the class is a `UnaryFunctionObjectRedn<Scalar, RetScalar>`. When I was adapting RVL to TSFCore and when Hala and I were trying to build a client–server framework [13], we saw the true problem with these templates.

4.3.4 Abstract Return Type

The original designers assumed that reductions would return integers or scalars. This meant the return values could be safely passed around by value at little cost. The only interactions involving the return types were between algorithms, which knew the type they expected to get back, and function objects, which knew the type they were going to return. However, when I began dealing with abstract function objects, I saw that the templated return types could be trouble. It was impossible to cast a `FunctionObject` to a `UnaryFunctionObjectRedn<Scalar, RetType>` without knowing both the scalar type and return value type. There were times when I wanted to get the result of a reduction but did not even know the number of parameters of a particular function object. Of course, it is always possible to find out through trial and error, but this wastes runtime with extra dynamic casts. C++ unfortunately does not permit a switch on a type.

I did not want to revert to the TSFCore solution of using a void pointer for all return types. This is not type safe and feels too much like a hack. My goal all along has been to take advantage of object orientation to improve the software design. The language permits multiple inheritance, so I suggested using a base class `Reduction` which manipulated an abstract base class `RetType`. A `UnaryFunctionObjectReduction` would then be a child of both `UnaryFunctionObject` and `Reduction`.

The `Reduction` class is fully implemented and concrete, which places very little burden on children. It has four methods.

setResult() Reset the result to the initial value.

setResult(RetType &) Set the result to the given value.

getResult() Returns the current value of the result.

createRetType() Returns a pointer to a dynamically allocated instance of the `RetType`. The caller is responsible for deallocating this.

The `Reduction` maintains a reference to a `RetType` object, which it gets from its constructor. Children are responsible for owning the result and passing the reference to the result into the base class constructor.

An `Accumulation` is a commutative reduction. This mixin interface (a small interface that can be appended to a class definition through multiple inheritance) can be used to indicate that a `Reduction` is commutative. The interface only has one pure virtual method, `accumulateResult( RetType &)` , which should accumulate the input return value into an internal buffer. For example, the L^1 norm is an accumulation, and results are accumulated by simple addition. Properly done, the `Accumulation` interface provides an easy way to implement commutative reductions without burdening the function object with worrying about how many local data containers there are. It would also be a way to hide parallel reduction calls outside of the main object code.

The `RetType` is also a very simple interface. Return types are expected to be able to be assigned using an overloaded = operator. The type must be able to allocate duplicates of itself through the `clone()` method. Finally, it is the responsibility of the type to remember a default value and be able to reinitialize itself. This is reasonable, as any class constructor has to initialize the data members to something, and I simply encapsulate this code in a public method which can be called later. These three methods are sufficient to fully implement the methods in the `Reduction` class.

Most frequently, the return type is a simple scalar. I wrote the `ScalarRetType` class to implement such scalar return types. The class is templated on the scalar, and implements the `RetType` methods using a scalar data member. Access to the scalar is through overloaded operators so that it may behave just like a regular scalar. An implicit conversion to the scalar type is also provided. This return type is used to implement scalar reductions which behave like the old templated reductions used to. `UnaryFunctionObjectScalarRedn` owns a `ScalarRetType` and passes it to the `Reduction` constructor. It provides convenience methods `getValue()` and `setValue` which take and return scalars, so a user may ignore the return type class if they wish. Similar implementations are available for binary, ternary, and quaternary function objects. Primarily, I wanted to allow the use of the additional reduction and return type functionality without adding more work and restrictions for the implementer of function objects.

With the new interface, it is possible to write code which manipulates abstract function objects. The return type may be accessed by casting a function object to `Reduction`, without needing to know the scalar type or number of parameters of the function object. Storage can be allocated for a result without knowing the exact return type. Such modifications permit both better client–server code [13] and full adaptation to TSFCore (see Section 6.2).

4.3.5 Generalized Function Objects

When trying to implement some functions, I noticed a severe limitation in the function object interface. The interfaces insist that all local data containers involved be of the same scalar type. For inner products and such, this is not a problem. However, there are cases where multiple data types are necessary. Some examples:

- A binary function object which converts a data container of doubles to one of floats.
- A ternary function object which applies a boolean mask to a local data container and stores the new result in a third.
- A binary function object which computes $\sum_{i=1}^{n} j_i * x_i$ where j is an array of integers and x an array of doubles.

Type conversions often arise when dealing with other linear algebra libraries and old FORTRAN libraries. Further, in order to handle large data sets, it may be necessary to compute in a higher precision when dealing with a small fragment of the data at a time, but then convert into a lower precision to save memory for the entire data set.

The fix for this is to insert an abstract class between the base `FunctionObject` interface and the templated `*naryFunctionObject` interface. In the binary case, the new `GeneralizedBinaryFunctionObject` is

```
template<class Scalar>
class GeneralizedBinaryFunctionObject: public FunctionObject {
public:
  GeneralizedBinaryFunctionObject() {}
  GeneralizedBinaryFunctionObject(
    const GeneralizedBinaryFunctionObject<Scalar> &) {}
  virtual ~GeneralizedBinaryFunctionObject() {}
  /** Evaluation method - virtual */
  virtual void operator()
    (LocalDataContainer<Scalar> &,
     DataContainer &) = 0;
};
```

Notice that the second parameter is a DataContainer instead of a LocalData-Container. The template parameter of the generalized interface only must match the type of the LocalDataContainer parameter.

This addition requires a few modifications in existing RVL library code, but is entirely backwards compatible with existing function objects and data containers. The old *naryFunctionObject classes now inherit from the generalized interface, and implement the inherited method by calling their own virtual method:

```
/** Evaluation method - inherited */
  void operator()
    (LocalDataContainer<Scalar> & x,
     DataContainer & y ) {
    try {
      LocalDataContainer<Scalar> & yt =
        dynamic_cast<LocalDataContainer<Scalar> &>(y);
      (*this)(x,yt);
    } catch(bad_cast) {
      RVLException e;
      e << "Error in BinaryFunctionObject::operator()(LDC, DC) -";
      e << " cannot cast the\n";
      e << "DataContainer parameter to a LocalDataContainer<Scalar>.";
      throw e;
    }
  }
```

In the base `LocalDataContainer` class, small adjustments were needed in the implementation of the `eval()` methods. They now try to cast to the least general type first, and if that cast fails, then cast to the generalized interface instead.

```
 try {
    BinaryFunctionObject<Scalar> & bfo =
      dynamic_cast<BinaryFunctionObject<Scalar> &>(f);
    try {
      LocalDataContainer<Scalar> & lx =
dynamic_cast<LocalDataContainer<Scalar> &>(x);
      bfo(*this,lx);
    }
    catch (bad_cast) {
      ...
    }
  }
  catch (bad_cast) {
    try {
      GeneralizedBinaryFunctionObject<Scalar> & gbfo =
dynamic_cast<GeneralizedBinaryFunctionObject<Scalar> &>(f);
      gbfo(*this, x);
    } catch(bad_cast) {
      ...
    }
  }
```

As we expect most function objects to follow the old, single–type interfaces, this is no additional run–time cost for them, and only one extra cast for the new, mixed–type interfaces. Further, the `eval()` methods are virtual and may be overridden by a concrete LDC which wishes to invoke the generalized interface first.

4.3.6 Streamable

In any numerical software, it is important to be able to read data from somewhere and output results. The RVL designers recognized the need for output in a package, and implemented this through the `write()` methods found in most classes. These methods either take a `ostream` and write to it, or attach their output onto the string in a `RVLException`. Input is provided for data containers through function objects. However, I recognized the need for being able to both output and input objects to something other than the standard streams and exception. This need arises in parallel and component applications, whenever we need to move objects from one machine to another.

The standard library already has I/O streams and file streams, so why not use the stream concept to implement this functionality? A `RVLStream` is expected to behave like a standard I/O stream. Callers should be able to put in and take out standard types — integer, character, floating points types, and strings. Fundamentally, every data structure in C++ is composed of these types. An object then becomes `Streamable` if it knows how to pass its data into and retrieve its data from a stream. We call these actions `Marshall` and `UnMarshall`. An object which is composed of streamable objects implements these methods by marshalling its data members in order and unmarshalling them in the same order. For each object, the implementer must decide an ordering for data as well as which data are necessary. In some cases, the entire state of an object need not be passed (which is partly why I don't simply hack together a brute–force memory copy).

Hala and I developed this idea for use in her thesis [13]. A `RVLStream` was implemented using the socket communications library, and many function objects and data containers were made streamable. Making something `Streamable` is easy

1. Inherit from the `Streamable` parent class

2. Implement `Marshall` to stream each of the data members.

3. Implement `UnMarshall` to grab each data member in the same order you streamed them.

This has not been done for all possible classes, since it was not necessary at the time. It is quick to add this functionality on–demand.

The base stream class has some methods beyond the basic inherent types. It can input and output both a `stl::vector` and an array of basic types. Both methods are templated and implemented in a simple manner, calling the input and output operations of the `RVLStream` for each bit of data. This is naturally not the most efficient way, and children are expected to have their own implementations of these methods to improve upon them. However, I followed our basic principle of implementing whatever possible in the base class and allowing children the choice to overwrite them or not.

Chapter 5

Competitors

There are a wide variety of software packages relevant to the construction and solution of a simulation–driven NLP. I classify these packages into the following taxonomy, expanding on the classification system in [11]:

Linear Algebra Interface Abstract interfaces to linear algebra objects (e. g. TSFCore)

Linear Algebra Library Fully implemented code for linear algebra (e. g. LAPACK [1], TNT [39])

NLP interfaces Abstract descriptions of nonlinear problems (e. g. TSFCore::Nonlin [8])

NLP implementations Implemented problems, often involving simulation code

Abstract Algorithm Interface Abstract definitions of algorithm objects and classes of algorithms (e. g. ALG)

Abstract Numerical Algorithm Coordinate–free implementations of solution methods (e. g. MOOCHO [9])

Calculus Interface Abstract interfaces for calculus objects (e. g. RVL, HCL)

Calculus Library Fully implemented calculus objects (e. g. TSOpt [44], RVLTools)

Some packages have parts which fall into multiple categories, often combining an abstract interface and implementations into a single named package (e. g. Epetra [25], NOX, and LOCA [31]). Each category fills a different role toward the solution of simulation–driven NLPs. Omission of a category as part of the design of a total solution misses opportunities for reuse. I posit that RVL fills a unique, necessary position among its competitors as the sole implementation of a calculus interface, possessing several unique features.

Further, as the RVL/ALG project has grown in an environment full of competitors, the competition encouraged an exchange of design and implementation ideas between the authors of each package, strengthening all of the packages in the process. I will discuss briefly some examples of competing packages and attempt to highlight and evaluate the design decisions in each. This will provide some insight into effective and ineffective design choices and offer evidence of RVL's uniqueness.

5.1 Template Numerical Toolkit

The Template Numerical Toolkit (TNT) is a National Institute of Standards and Technology project implementing templates for arrays and matrices [39]. These objects are simple and easy to use — they resemble the Standard Template Library classes with some additional linear algebra functionality. A number of overloaded operator functions are provided as part of the package, allowing users to write C++ code which greatly resembles MATLAB. The template parameter on the various classes can be filled with any type which has overloaded operators so that it can perform all the tasks of a `float`.

There are three different array types, `Array1D`, `Array2D`, and `Array3D`. The distinction in dimensionality allows multi–dimensional array indexing where appropriate. Otherwise, these classes are very similar, providing an assignment operator, index operator, pointer access, and functions which return the dimensions. There are also `Fortran_Array#D` classes, which use a FORTRAN style storage and access scheme for interfacing with FORTRAN libraries.

TNT used to have a `Matrix` class, but it was deprecated in favor of the `Array2D`. Similarly, the `Vector` class has been replaced by `Array1D`. There used to be an overloaded `*` operator between `Matrix` and `Vector` which performed a matrix–vector multiply. However, this was lost when those classes were deprecated, and there are no longer any interactions between `Array1D` and `Array2D`.

The operators provided can be useful. However, it has been shown that overloaded operators compile into poor programs, due to the necessity of creating temporary storage, and the sometimes vague rules for order of operations. For example, given `Array1D x,y,z`, the result of `x = x/y/z;` is unclear.

The classes in this package provide some slight advantages over the standard template library classes. However, they are not suited for high–level algorithm design, and do not provide any sort of abstract interface. They are concrete classes suited for writing MATLAB– and FORTRAN–like C++ code.

5.2 OOQP

OOQP is a package of quadratic program (QP) solvers in an object–oriented framework [16]. Such a package naturally requires linear algebra objects for vectors and matrices. It defines an abstract interface for these objects, and provides several implementations which meet this interface.

`OOQPVector` is the interface for vectors. An `OOQPVector` has methods for doing linear combinations and dot product, but also has many methods tailored for solving quadratic programs. It has methods to copy the vector to and from an array, which allow a user to implement any operation on the copy then insert the result back into a vector. This is not efficient, but is functional.

The OOQP package has two interfaces for matrices. `GenMatrix` and `SymMatrix` have element access methods for performing computations. These matrix classes are used to construct problem definitions which are solved by the QP Solver. A `Status` class watches the convergence of the solver and determines when to stop.

The linear algebra interfaces are well suited for solving QPs, but not for general use. Thus, it seems best to adapt a general linear algebra interface to the OOQP interfaces in order to use the QP solvers. Any use of the OOQP classes outside of the package would incur unnecessary costs.

5.3 PETSc

The Portable, Extensible Toolkit for Scientific Computing (PETSc) [14] was developed at Argonne National Laboratory to supply the building blocks of parallel scientific applications. Although it claims to be object oriented, it is a far cry from being truly so. In this case, objects are structures which are passed as parameters in function calls. Thus, there are no abstract interfaces for objects. A vector is created by passing a pointer to a `Vec` structure into `VecCreateSeq()` or `VecCreateMPI()`. As PETSc is targeted for parallel applications, even the sequential functions require a MPI communicator, `PETSC_COMM_SELF`. Creating a vector requires a series of calls to allocate the storage, initialize the data, then distribute data as necessary.

Operations on vectors are performed either by a list of standard functions or through the use of `VecGetArray(Vec v, PetscScalar **array)` and `VecRestore-Array(Vec v, PetscScalar **array)`. These functions expose the scalar data pointers without invoking a copy, in order to avoid unnecessary overhead. The exposed data is only the local data for each process. Thus, a user needs to be familiar with MPI to implement any non–diagonal operations. Also, care must be taken in selecting the provided functions to use, as some alternatives require less parallel communication than others.

Matrices, grids, and other objects behave in a similar manner to vector. They are allocated by filling in a pointer using a function call, then initialized and finalized. These objects are used in function calls to perform the calculations for an application. There are many facilities targeted at finite element and gridded applications for solving partial differential equations.

This package represents an older style of object–oriented programming. The interface feels clumsy to a programmer used to classes and generic programming. However, the efficiency of the applications was a foremost concern, and PETSc provides a large suite of solvers and integrators which function in parallel. PETSc requires a intimate knowledge of parallel scientific computing to be used properly, which makes it poorly suited for general use. It may be suited for adaptation to a more user–friendly interface in order to access the efficient parallel tools.

5.4 TSFCore

An example of a linear algebra interface (instead of a linear algebra library) is the TSFCore [10] interface, which is a part of the Trilinos Solver Framework [23]. Ideas have been exchanged between TSFCore and RVL many times, and it is interesting to examine the remaining differences between these two interfaces. Recognizing that TS-FCore is still evolving, the following discussion is based around the version described in [10], which I adapted to RVL in 2003.

TSFCore uses the concept of spaces as vector factories, but does not associate an inner product or linear combination operation with each space, where RVL does. Instead, TSFCore has implementations of an scalar product, linear combination, and other normal operations provided in a library file of TSFCore (Note: Trilinos Release 4.0 adds a scalar product to the TSFCore space class, but still implements the linear combination as a stand–alone procedure rather than a class method). TSFCore is built around the RTOp package [11], which defines a `SubVector` class for contiguous arrays of data and an `RTOp` interface for reduction and transformation operators on subvectors. TSFCore provides an `applyOp()` method in its `Vector` interface, which takes a `RTOp` and applies it to a sequence of `SubVector` objects built from pieces of the vector. This action is similar to the application of a `RVL::FunctionObject` to a `RVL::DataContainer`.

TSFCore has several higher–level interfaces for linear algebra objects. An `OpBase` has a domain space, a range space, and a query method for whether the adjoint is supported. `LinearOp` is a specialization of `Operator` whose application is presumed to be linear, which adds an `apply` method to the interface. TSFCore also contains a `MultiVector` class, conceptually inherited from Epetra, which the designers say is important for efficient implementation of many algorithms. A `MultiVector` is essentially an array of `Vectors` of the same size. A `MultiVector` can be applied in the same manner as a `LinearOp`, or each column can be accessed as a `Vector`.

TSFCore does not include any true parallels to the `Operator` and `Functional` classes in RVL. Instead, the `TSFCore::Nonlin` subpackage includes a nonlinear–problem definition for encapsulating all problems of the form

$$\begin{array}{lcccl} & & c(y,u(l)) & = & 0 \\ gL & \leq & g(y,u(l)) & \leq & gU \\ yL & \leq & y & \leq & yU \\ uL(l) & \leq & u(l) & \leq & uU(l), \text{ for } l = 1\ldots Nu \end{array}$$

which explicitly partitions the variables into state and controls. Both c and g are assumed to be vector–valued. The specialization `NonlinearProblemFirstOrder` adds method for accessing first–derivatives of c and g. While this form is general enough to encompass all NLPs, the pieces of the problem are neither modular nor reusable. The operators c and g are not independent objects. A `newpoint` boolean flag is used in all computational methods to indicate whether the other parameters represent a new y and u, in order to permit some reuse of intermediate data. Finally, it is necessary to use null pointers to indicate the absence of c or g for purely equality constrained or purely inequality constrained problems.

TSFCore uses many modern language features of C++, such as namespaces and templates. This avoids name conflicts and facilitates easy switching between scalar types. There are a few anachronistic features, especially in RTOp, since the precursors of TSFCore were written in C. However, Bartlett asserts that he is phasing out such features and moving toward a fully C++ based interface. This will also allow the replacement of `void *` pointers in some locations with abstract base classes, and should result in a cleaner interface overall.

TSFCore makes strict use of `const` in both parameters, return values, and method declarations. This provides some extra safety in method calls and prevents some mistakes. However, it can result in a few difficulties when trying to adapt to a TSFCore object from one which does not employ `const`. Luckily, I know a trick to get around the restrictions imposed by `const`. If a class has a data member which is a pointer, then as long as a method of the class does not change the *value* of the pointer, the compiler does not complain about modifications and non–const method calls on the object to which the pointer points. If *ptr* is a class data member, then

```
void foo() const {
*ptr = bar; // LEGAL
ptr = NULL; // ILLEGAL
}
```

Thus, a `const` method can call a non–constant method on an non–constant object to which it owns a pointer. The pointer stays constant, even though the object pointed to does not.

5.5 Epetra and Tpetra

Epetra [25] is a linear algebra library, primarily designed for parallel and sparse computations. It is a foundation interface for many of the Trilinos [23] solvers and AztecOO. Additionally, there are implementations of the interface which utilize BLAS and LAPACK routines to perform serial computations.

Epetra provides a very complete list of coding guidelines for developers [24]. Although a few of their recommendations are inconsistent with choices made in RVL, it's still a useful document. Thorough justifications are made for programming practices.

The base `Epetra_Vector` interface allows a vector to be constructed from a block map. Alternately, a vector can be built from a serial array of doubles or taken from a column of a multivector. Methods are provided for overwriting or accumulating elements selected by an index array with data from an array of doubles. A copy of the entire vector can be extracted, a view created, or individual elements accessed. Non–member functions are provided for computing dot–products, norms, linear combinations, and finding a minimum, maximum, or average. The `Epetra_MultiVector` behaves as one would expect, with similar functionality to a `Epetra_Vector`. Interestingly, `Epetra_Vector` is a child of `Epetra_MultiVector`, specializing a multivector with only one column. Thus, their approach considers the multivector a more fundamental object, which is a common viewpoint in the linear solver community.

The basic matrix class is the `Epetra_RowMatrix`, which represents a collection of row vectors. This object can be multiplied by a multivector, sums of absolute values of elements computed, scaled on the right or left by a vector, and the inverse applied to a multivector. Many attributes can be examined, such as norms and number of non-zeroes. Data can be extracted from a row or the diagonal. Implementations of the interface are provided using the compressed row storage (CRS) and variable block row (VBR) formats.

In many ways, the map classes serve as space classes do in other packages. Each map has properties which define how vectors following the map are created and distributed. The only public method besides constructors and the destructor is an assignment operator. Thus, a user can create maps and move them around, but not access the internal data of a map in any manner. A map is needed when creating vectors, multivectors, and other objects, which must be friends in order to use the protected methods of the map. The base map is the `Epetra_Map` which has children `Epetra_LocalMap` and `Epetra_BlockMap`. While this design serves to encapsulate the map data, it seems strange. The friend classes aren't allowed to modify the map data. Why not allow public read–only access to the map data, and avoid the use of `friend` entirely?

Epetra was designed from the beginning as a parallel library. Serial execution is treated as a special case. Almost every class requires a communicator in its constructor. When run in serial, an `Epetra_SerialComm` is provided as a placeholder. Further, all data classes are loaded with parallel specific methods, such as methods for handling both local and global indexing and a `FillComplete` method for optimizing and finalizing storage across processors. This is a sharp contrast to RVL, where parallelism is completely hidden in the base interface.

Epetra is a low–level interface for storing and manipulating data directly. The data storage classes have explicit sizes and access methods, as well as query methods about the layout of data. Operators are between maps, not spaces. There is no interface for functionals or user defined function objects. If a user needs a computational method which is not provided, they can implement it using the overloaded `operator[]`.

Epetra avoids many modern language features that have only recently seen support from compiler writers. Namespaces are replaced with the requirement that all class names begin with `Epetra_` . Templates are not used, and all computations are performed in double precision arithmetic. The `double` declarations are hard coded, and it would require a large effort (or tricky use of scripts) to switch to single precision or complex numbers.

One very intelligent coding guideline is the avoidance of standard library include declarations in header files. Instead, a single "Epetra_ConfigDefs.h" is included when library files are needed. This makes the package more portable, as this is then the only file that needs to be edited when going between a system which does `#include <iostream.h>` instead of `#include <iostream>`.

5.5.1 Future upgrade to Tpetra

The Tpetra project is still in progress, but the goals are simple — extend the functionality of Epetra for double precision real numbers to templated classes. This may seem a simple task, but adding templates is not always easy, especially when dealing with legacy code and parallelism. To my knowledge, the parallel libraries like MPI are not templated. Since C++ doesn't treat types as parameters, the libraries make up their own enumerated values to signify each scalar type, and assign these values descriptive names such as `MPI_Double`. This forces a programmer to use template specialization in order to deal with MPI.

Tpetra has three template parameters which it uses throughout the package:

PacketType Something which can be communicated between nodes.

OrdinalType Used for ordering. Often just integers.

ScalarType values in computations. Often Floating point or complex.

The basic classes in Tpetra are the same as those in Epetra. Most classes require a `Tpetra::Comm` in their constructor. `Tpetra::Vector` behaves generally the way it did in Epetra. However, there are some new additions, such as the `Tpetra::Vector-Space`. This class is a factory for vectors, and it requires a `Tpetra::ElementSpace` or `Tpetra::BlockElementSpace` in its constructor. The element spaces seem to be replacements for the map classes of Epetra, containing methods which can be queried for the various ordinals which describe an arrangement of scalars across processors. The calculation methods for vectors are all built into the `Tpetra::Vector` and concretely implemented around a combination of the BLAS and reduction operations built into the communicator. The `Tpetra::VectorSpace` is also a concrete class.

The sparse matrix class `Tpetra::CisMatrix` nicely encapsulates two methods of sparse matrix storage. It can either do compressed row storage or column storage. Unlike Epetra, this matrix class does not inherit from a base operator, multivector, or matrix class. Its only parents are the general `Tpetra::Object` and `Teuchos::CompObject` classes. Whether this is an intentional design decision or simply an attempt to get a working matrix class without building all of the abstract hierarchy first is unclear from the limited documentation. Apart from all the methods for constructing the matrix and querying it about its properties, there are only three computational methods. The `CisMatrix` can be applied to a `Vector` and store the results in another `Vector`. It can also compute the global one–norm and infinity–norm.

I have high hopes that this package will fill out to fully replace Epetra's functionality for parallel linear algebra.

5.6 Multivector

Several of the packages I've described include a `MultiVector` object. This can be viewed as an array of identical vectors. The authors of these packages often cite efficiency reasons which make such an object necessary. I want to clarify their reasoning and address the apparent lack of such an object in RVL.

Some algorithms need to perform the application of a linear operator to multiple vectors $B \leftarrow A * X$ as well as solving a linear system with multiple right–hand sides $X \leftarrow A^{-1} * B$. While this may resemble a matrix–multiplication, it is not always appropriate to treat X and B as matrices. Instead, they are simply a collection of vectors. In such situations, much speed can be gained by efficient use of caches and eliminating multiple accesses to the same memory address. This is the reason that the

Level 3 BLAS can often achieve near peak efficiency [18]. In order to take advantage of the Level 3 BLAS, the designers of Epetra include an explicit `Multivector` class.

In addition, the multivector methods may be reimplemented in children to gain some added efficiency when performing a parallel reduction. The default implementation simply calls the `Vector::applyOp()` method on each element of the multivector. However, some children avoid performing multiple reductions on scalar return values by treats an array of return values as a reduction type and perform one reduction on the entire array. This greatly reduces the amount of communication in a program. However, such efficiency is not in the abstract multivector interface, but rather a particular implementation of the multivector and reduction operations. Such efficiency also requires a special version of the `applyOp()` method to be reimplemented for each multivector type, which is additional work for the multivector author and not true reuse.

RVL does not have an explicit multivector class. Such an object does not fit in the vision of a `RVL::Vector`. RVL also does not have a matrix class. The RVL linear operator is applied to a single vector at a time. This would seem to preclude us from gaining the efficiencies provided by a multivector.

However, such observations are made from a limited understanding of the package. RVL has no such inadequacies. At all levels of the data hierarchy, product constructions are permitted. Product composites are available to construct objects from components (e. g. `ProductSpace` is built from other `Space` objects). I added Cartesian–power composites which simplified the construction of Cartesian powers of objects. A Cartesian power of a vector type has exactly the abstract properties of a multivector. Absolutely nothing prevents a savvy user from implementing linear operators which recognize such composites and take advantage of their structure or intelligent parallel product data containers which recognize reductions and aggregate parallel reductions to reduce communication. Close examination reveals that the Epetra and TSFCore multivectors are intended to be in–core and contiguous, which precisely describes the `RVL::ProductLocalDataContainer`. There is nothing preventing a `FunctionObject` from checking for product LDCs and altering behavior to handle this special case or a `ProductLocalDataContainer` which checks for reductions, so RVL has not lost any of the potential efficiency of a multivector.

Finally, the `Vector` class in RVL is a higher–level object than the vector classes in other packages. The algorithms `Vector` was designed to address are abstract, and low–level details are meant to be encapsulated inside other classes so that they may be easily changed later. Building such detail into an algorithm ties the implementation to a particular problem or machine and eliminates efficient reuse without modification. The multivector is a low–level efficiency detail which should be kept as an option in low–level code, not a high–level abstraction.

5.7 Is RVL Unique?

Following this critique and comparison of RVL to several other computational packages, it is natural to ask, "Is RVL a unique design which provides something not available in the other packages?" The answer is an unequivocal, "Yes!"

RVL and TSFCore are the only packages known to me which encapsulate operations on data within a visitor pattern in order to make such operations reusable and allow the implementation of new operations without altering the data interface. This feature alone eliminates all the other competitors. RVL then has additional benefits over TSFCore:

- Every `TSFCore::Vector` has a method returning an integer dimension. The original designers of RVL have a very good argument against including an explicit dimension in [41]. Dimensionality is necessary in low–level data types as a control on loops over data. However, high–level data types may not have an explicit finite dimension, especially when we consider adaptive approximations to infinite dimensional objects.

- TSFCore lacks an independent linear combination and scalar product attached to the space, preventing `TSFCore::VectorSpace` from fully expressing the vector space abstraction (although they have recently added a scalar product, linear combination is still missing). When adapting an `RVL::Space` to a `TSFCore::-VectorSpace`, the linear combination and scalar product methods of the `RVL::-Space` cannot be accessed and TSFCore automatically uses the default procedures built in to the package, which are not appropriate for every space. For example, consider the space of all polynomials over the interval $[0, 1]$. There is a well defined inner product $\langle f, g\rangle = \int_0^1 f(x)g(x)dx$. Any given polynomial can be encoded by storing the coefficients in a LDC d so that $f(x) = \sum d[i]x^i$. However, the LDC *must* have a resize method so that storage can be expanded as necessary when doing linear combinations. The sum of a degree n polynomial and a degree m polynomial is a polynomial of degree $\max\{m, n\}$. The linear combination method for such a space must resize the target LDC before adding coefficients. Two loops are necessary to perform the linear combination, as the smaller LDC will run out of coefficients before the larger one.

- TSFCore is restricted to linear algebra objects (vectors and linear operators) and does not provide the interfaces to functionals and operators which RVL has. TSFCore::Nonlin jumps right past functionals and operators into nonlinear problems built around the linear algebra interface. This restricts the reusability of the problem code, as the constraint $c(y, u) = 0$ is not an individual object but a collection of method calls inside the problem object.

- TSFCore permits Cartesian powers of vector in the multivector object, but does not enable Cartesian products, where RVL has both. The TSFCoreExtended package has since added arbitrary Cartesian products.

- TSFCore is built around the use of reference counting pointers, which while not restrictive on computation, imposes additional effort on users to properly initialize and dereference the pointers. However, reference counting pointers are very conventional and widely accepted.

- In Chapter 6, I adapt RVL to TSFCore and vice–versa. With the addition of the base `RVL::Reduction` class and abstract return type, no functionality of TSFCore is lost in such an adapting `TSFCore::Vector` to `RVL::DataContainer`. However, I had to hobble the `RVL::Vector` class in order to adapt to `TSFCore::Vector` (and similarly hobble the RVL spaces), losing access to the linear combination, identity element, and scalar product which belong to the RVL space. TSFCore imposes its own implementation of these methods that may not be appropriate in some situations.

This means that anything which can be done in TSFCore can also be done in RVL, but RVL has capabilities that TSFCore does not, namely dimensionless vectors and spaces with a nonstandard linear combination operation. Further, as discussed in Chapter 6, the TSFCore vector and space classes are semantically similar to the RVL `DataContainer` layer, and the RVL `Vector` and `Space` are higher level abstractions with no true parallel in TSFCore. Finally, TSFCore still takes a very different tack than RVL on functionals and operators, wrapping both inside the `TSFCore::Nonlin::NonlinearProblem`.

It is interesting to note that TSFCore has converged toward RVL with each design iteration. In the beginning, it resembled HCL with the novel reduction–transformation operators. It is now quite similar to RVL, and RVL to it, entirely due to the ongoing exchange of ideas between the packages. Such an exchange is the primary benefit of competition — making both packages stronger in the end.

Chapter 6

Adaptation

I stated earlier that I want to increase the use and reuse of existing code, instead of rewriting new code for each new application. I want to be able to reuse code from other packages as well my own. It is a waste of effort to reimplement existing algorithms and data structures simply to make them fit a new interface. Instead, I want to add small bits of code to make them fit the interface but still function, ideally without modifying the original code. Such small added bits to modify an interface follow the *adapter* pattern [15].

As scientists try to tackle more difficult computational tasks, the complexity of the software increases greatly. Further, the applications often require knowledge from a broader spectrum of disciplines. This can lead to difficulties for the small programming groups typically encountered in Academia and Industry. While such groups could attempt to write all the code from scratch, there is almost always insufficient time to do so. Further, rewriting an existing code to use different data structures is wasteful and likely to introduce errors. An algorithm frequently requires expert knowledge to implement effectively, and without such an expert in a group, it is necessary to consult an expert remotely. Such a remote expert will have their own data–structures and interfaces, and it is difficult to compare the efficiency of different packages in order to find the most suited. With limited resources, it becomes difficult to test code on a variety of different architectures, thus potentially limiting the portability of code.

These difficulties leave us with tough choices. We could stop computing, and emphasize analytical methods. If this is unacceptable, we can purchase sufficient work hours by hiring additional staff and students. Lacking enough money for such an enterprise, we could attempt to form a standards body for numerical data structures and mandate their use to the scientific community. This is likely a Herculean task, which leaves us with one option: If we use good software design that permits easy adaptation between packages, and encourage others to do likewise, then everyone's code becomes more useful. If code is easily adaptable, then it is also easy to adapt and compare similar packages from different sources in order to find the one that best suits a need. This also permits easy testing of packages against a variety of problems to ensure they are stable and produce valid results.

Once data structures grow more complex than simple arrays of contiguous data, object–orientation helps us to avoid many pitfalls of functional code. Without abstract interfaces, we must rewrite every algorithm to accommodate each change in data structure. For n algorithms and m data structures, we get $n*m$ pieces of code to maintain and update. With one common interface to data structures, we instead have $n+m$ pieces of code, each one covering a specific need and which can be maintained independently. Further, we can use inheritance to make incremental changes and establish hierarchies. For example, in RVL, the `RnArray` is an implementation of the `LocalDataContainer` (LDC) interface which represents a simple array of contiguous data. There are children of `RnArray` which add extra data and functionality, like the `SeismicLDC` which carries additional data about time, date, and location in which the data was taken. Some algorithms need this additional data and can access it, but we may still reuse any code that works with `LocalDataContainer` on a `SeismicLDC`.

Many packages have similar needs, but as each programmer has their own style and conventions, we end up with a variety of different classes which implement the

Figure 6.1: 6-pack Grounded Adapter Plugs by Franzus

same idea, but in slightly different fashions. There is frequently semantic overlap between classes in different packages, but there can be large syntactical differences. For example, we often want to encapsulate an array of contiguous data with its size and other relevant information to the application. Here are several implementations:

RVL calls such an object a `LocalDataContainer`

TSFCore uses RTOpPack, which calls them a `SubVector`

TNT has the `Array1D`

C++ STL has a `vector` class

OOQP has a `OOQPVector` which encompasses this concept and some additional functionality.

Theoretically, each of these objects fulfills the same role in the package, and we should

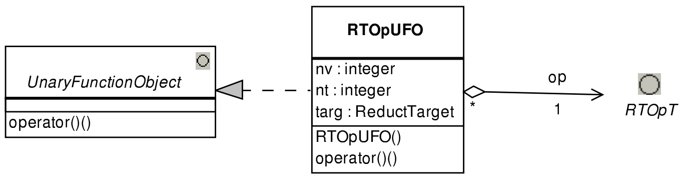

Figure 6.2: An adapter from `RTOpPack::RTOp` to `RVL::FunctionObject`

be able to adapt the interfaces to one another so that a concrete implementation of one could satisfy the interface of another.

The bits of code to adapt one software interface to another are like the extra plugs international travelers need for electric appliances, as shown in Figure 6.1. Each electrical system has its own hidden inner workings behind a standard interface. Plugs manufactured for the United States won't fit the interface for the European power system. Even if the plug fit mechanically, the voltage supplied by each system is different, so appliances will not operate properly. However, the intent of both systems is the same — to supply electrical power on demand. The job of the adapters is to allow a mechanical linkage between different systems (syntax), as well as modifying voltages to match expectations (semantics). Further, the adapters do not need information about how the power system supplies power (implementation) in order to perform their function.

6.1 General Strategy

Since the fundamental ideas in each package are similar, it appears that I could synchronize the interfaces and use the classes interchangeably. For a concrete example, consider combining an RVL–based package with a TSFCore–based package. The Time–Stepping for Optimization (TSOpt) package, which is written using RVL interfaces, automates time–stepping and derivative calculations for PDE systems using the adjoint state method. Thus, I can build an optimal control problem

$J(u) = \|F(u) - y_d\|^2$ where $F(u)$ is an operator which maps the control u to a value of the state y. MOOCHO, an nonlinear optimization package, can solve optimization problems which are defined with the TSFCore interface, but doesn't have a time–stepping tool like TSOpt. In order to combine these packages, I need to be able to use a `RVL::Operator` to build a `TSFCore::Nonlin::NonlinearProblemFirstOrder` and choose some vector class in which data will be stored. Thus, I need *adapters* between the various vector interfaces.

An adapter owns the object which is adapts, and inherits from the interface it is adapting to. An example of this can be seen in Figure 6.2. The adapter implements method calls by forwarding them to the equivalent calls on its data member. When building adapters between packages, it is necessary to start at the lowest level objects and work upward. The high level adapters will use the low level adapters. Suppose I want to adapt an object from package A to handle an interface in package B. For each method call in interface B, I would

1. Wrap the inputs as A objects
2. Call the appropriate A method
3. Wrap the results from this call as B objects. As wrappers usually work by reference, it is not necessary to unwrap inputs which were modified

In the case of RVL and TSFCore (with the subpackage RTOpPack), I first adapt `RTOpPack::SubVector` and `RVL::LocalDataContainer`. This is easily done (and actually unnecessary in one direction due to the `SubVector` special constructor), and I can then use these adapters to adapt `RTOpPack::RTOp` and `RVL::FunctionObject`. These are objects that describe transformations and reductions between the array classes, that is, mappings between vector spaces and from vector spaces onto a set.

The transformation adapters are somewhat more complicated due to philosophical differences, and can not function properly in all cases. `RTOp` only encompasses diagonal operations, that is, ones on the i^{th} element of each array at the same time,

RVL	RVL to TSFCore	TSFCore to RVL	TSFCore
LinearOp	RVLLinearOpAdapter	TSFLinearOpAdapter	LinearOp
Vector	VecTSFCoreVector		
Space	use StandardSpace		
DataContainer	not needed	TSFCoreVectorDC	Vector
FunctionObject	FORTOp	RTOpFO	RTOp
LocalDataContainer	not needed	LocalSubVector	SubVector

Table 6.1: Adapters between RVL and TSFCore.

while `RVL::FunctionObject` is allowed to be more general. `RTOp` treats the result of a reduction as an independent object, and all `RTOp` objects return a result (which is a null pointer in the case of transformations). `FunctionObject` is expected to store the result internally, and return it later when asked. Thus, in RVL, `Reduction` is a mixin, adding the additional interface for retrieving the result. Further, `RTOp` has built–in functionality necessary for MPI–style parallelism, and it is difficult to implement such functionality in an adapter given only the general RVL interface.

The next step up is to implement the adapters between `TSFCore::Vector` and `RVL::DataContainer` as well as `TSFCore::VectorSpace` and `RVL::DataContainer-Factory`. The first two are targets for visitation by `RTOp` and `FunctionObject` and thus need to use the adapters of the visitors. The adapters for `TSFCore::Vector` and `RVL::DataContainer` are easily done, as are the ones for their factories. Finally, RVL has one higher level of objects (`Vector` and `Space`) , but I can build standard versions of these from TSFCore objects using the adapters I've already created and a few extra pieces. Note that it is often possible to demote a class, for example treat an `RVL::Vector` as a `RVL::DataContainer`, and use the lower level adapters, and this trick can be useful.

Both packages have an interface which implements the notion of a linear operator. In both cases, the operator has an apply operation which maps a vector in the domain to a vector in the range. The only difference is that all `RVL::LinearOp` objects are expected to be able to apply the adjoint, whereas `TSFCore::LinearOp` has a query function `opSupported()` which will report whether a particular operator supports forward or adjoint applications. Because they implement a well defined mathematical notion, these interfaces are very similar, which makes them easy to adapt. In the end, I get a hierarchy like that shown in Table 6.1. Any other classes in these packages can be adapted in a similar manner.

I often refer to this project as ATN, which stands for “All Together Now!”. The goal of the project was to demonstrate that these independently created tools could all work together to solve a problem. By reusing the software in each package, I can solve a problem by only creating small adapter classes instead of rewriting the large, complex tools to fit a different interface.

6.2 TSFCore and RVL

One of the primary advantages of object–orientation is the increase in reusability of code. Further, it helps to standardize the interface to data. Both `TSFCore` [10] and `RVL` [38] utilize the idea of transformation operator objects to represent functions and vector containers for storing data. Thus, any `RTOpPack::RTOpT` [11] object can be applied to any `TSFCore::Vector`, and similarly any `RVL::FunctionObject` can be evaluated on any `RVL::Vector`. This separates the functionality of the transformation from the implementation details of data storage, and lets us smoothly reuse transformation operators on new types of vectors.

One potential drawback to an object–oriented approach is that programmers frequently become married to a particular interface. Further, it is not immediately obvious how I might be able to do things like apply an `RTOp` to a `RVL::Vector`. Fortunately, through the iterative process of designing the linear algebra interfaces and the cooperation amongst designers, the core ideas in both packages are quite similar. This allowed me to create adapters which will let an `RVL::Vector` function as a `TSFCore::Vector` and vice versa. Then everyone may program with the interface they are most comfortable with, but still share code and ideas easily.

There are several immediate benefits to such a collaboration. The RVL project has been working on designing out–of–core vectors for use with huge (e. g. Gigabytes of data) vectors which will not fit in RAM. There are several parallel Linear Algebra Packages (LAP) which have already been adapted to meet the TSFCore interface. Both of these lack any similar counterpart on the other side, and it would save everyone time and frustration if they did not have to reimplement such ideas. Further, there are several different Abstract Numerical Algorithms (ANA) written to use each interface, and I would like these algorithms to be able to inter-operate. TSOpt, which is written using the RVL interface, performs time integration of a system of PDEs. It takes a collection of independent objects, such as steps, samplers, and a description of the dynamics of the problem, and creates an operator which maps the control vector to the solution of the state vector. Moocho is a package for optimizing both constrained and unconstrained problem, and has already been interfaced with TSFCore. Finally, FEM is a package for formulating PDE problems, and there are already adapters which allow it to utilize TSFCore vectors and linear operators.

I would like to combine these three packages in order to solve time–dependent optimal control problems with PDE constraints. These problem typically takes the form

$$\begin{array}{lll} \min_{u\in U} & f(y(u,t),u(t),t) & \\ \text{subject to} & c(y(u,t),u(t))=0 & \forall t\in[0,T] \\ & low \leq u(t) \leq up & \forall t\in[0,T] \end{array}$$

where c is a system of PDE's and I call $y \in Y$ the state variables and $u \in U$ the controls. The idea is to use Mathias Heinkenschloss' FEM package to build the finite–element model and formulate the PDE system, then couple that to Bill Symes' TSOpt package [44] to handle the time stepping and adjoint calculations. I can then formulate the problem in a TSFCore::Nonlin interface and use Roscoe Bartlett's MOOCHO [9] package to solve the optimization problem.

The alternative would be to rewrite all three pieces using the opposite linear algebra interface. This would result in two independent packages to maintain, and massive duplication of effort. Further, since I envision many more abstract numerical algorithms written using each LAP, I would like to be able to easily combine future packages as well. Otherwise, each project ends up digging parallel tunnels through the mountain of research, instead of starting at opposite sides and meeting in the middle.

6.2.1 Original Behavior

I should first describe the current behaviors of the packages. This will make clearer the similarities between them, and help the reader to understand the adaptations I made when they are described in Section 6.2.2. RVL has already been discussed in detail in Chapter 4, but I will reiterate a few points to make the comparison between the two packages clear.

6.2.1.1 TSFCore and RTOpPack

First, I should explain about the relationship between TSFCore and RTOpPack. RTOpPack is a smaller package that focuses on the definition of a Reduction/Transformation Operator (RTOp) and its interaction with `SubVector`. A `SubVector` is a view of a chunk of in–core data. This data need not be contiguous, but is required to have a starting point, number of elements, and a fixed stride between elements. This allows it to work on normal, FORTRAN style arrays as well as other views such as picking out one column of a matrix stored in row–major order. The RTOp knows how to take a collection of `SubVector` and `MutableSubVector` (which can be written to as well as read from) and perform its operation on them. If the RTOp is a reduction operation, it stores the result in a `ReductTarget`. The RTOp also can reduce two `ReductTarget` objects into one. Figure 6.3 and Figure 6.4 demonstrate

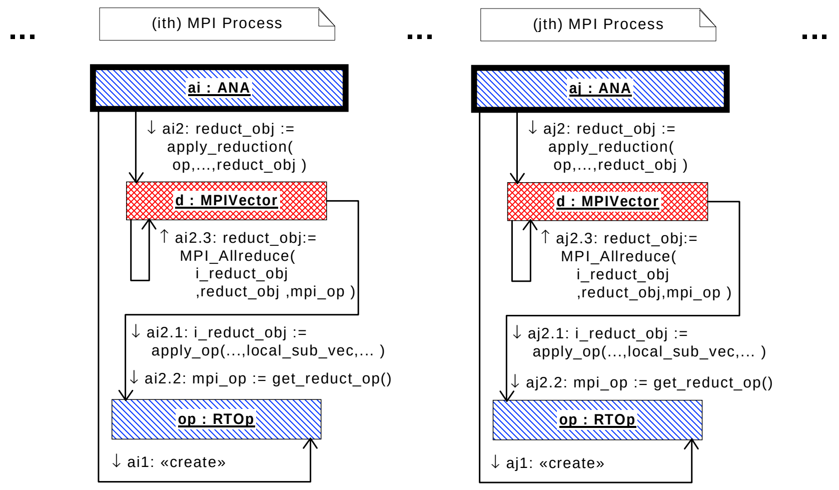
(ith) MPI Process
(jth) MPI Process
ai : ANA
↓ ai2: reduct_obj := apply_reduction(op,...,reduct_obj)
d : MPIVector
↑ ai2.3: reduct_obj:= MPI_Allreduce(i_reduct_obj ,reduct_obj ,mpi_op)
↓ ai2.1: i_reduct_obj := apply_op(...,local_sub_vec,...)
↓ ai2.2: mpi_op := get_reduct_op()
op : RTOp
↓ ai1: «create»
aj : ANA
↓ aj2: reduct_obj := apply_reduction(op,...,reduct_obj)
d : MPIVector
↑ aj2.3: reduct_obj:= MPI_Allreduce(i_reduct_obj ,reduct_obj,mpi_op)
↓ aj2.1: i_reduct_obj := apply_op(...,local_sub_vec,...)
↓ aj2.2: mpi_op := get_reduct_op()
op : RTOp
↓ aj1: «create»

Figure 6.3: Sequence of calls to apply a RTOp to a TSFCore::Vector which uses MPI to communicate between processes [11]

this procedure. These are reproduced (with permission) directly from [11].

TSFCore builds off of this functionality in RTOpPack. It defines `TSFCore::Vector` as an object that knows how to apply an RTOp to itself. The `Vector` iterates through each `SubVector` and applies the RTOp to each of them. It reduces intermediate `ReductTarget` objects into a global one using the `RTOpT::reduce_reduct_objs`. When the RTOp has been applied to all `SubVectors`, then the `Vector` returns the `ReductTarget` to the caller.

This is a slight oversimplification. In the case of some vector types, `apply_op` may require transmitting the RTOp to another process, on which the `SubVector`

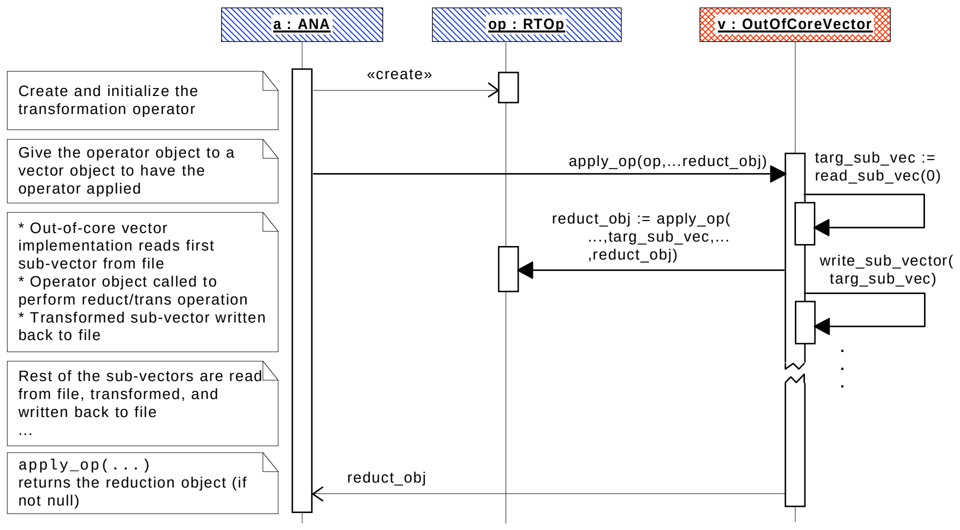
a : ANA
op : RTOp
v : OutOfCoreVector
«create»
Create and initialize the transformation operator
Give the operator object to a vector object to have the operator applied
apply_op(op,...reduct_obj)
targ_sub_vec := read_sub_vec(0)
reduct_obj := apply_op(...,targ_sub_vec,... ,reduct_obj)
* Out-of-core vector implementation reads first sub-vector from file * Operator object called to perform reduct/trans operation * Transformed sub-vector written back to file
write_sub_vector(targ_sub_vec)
Rest of the sub-vectors are read from file, transformed, and written back to file ...
apply_op(...) returns the reduction object (if not null)
reduct_obj

Figure 6.4: Sequence of calls to apply a RTOp to a TSFCore::Vector which accesses out–of–core data [11]

resides, and then doing a global reduction across all processes if necessary. RTOp's are designed to facilitate such migration between processes, and provide methods for loading and extracting their state.

6.2.1.2 RVL

RVL takes a very similar approach. There are four different types of function objects, depending on the number of input vectors they expect: `UnaryFunctionObject`, `BinaryFunctionObject`, `TernaryFunctionObject`, and `QuaternaryFunctionObject`. For each there is an additional subtype for reduction operations, e. g. `UnaryFunction-ObjectReduction`. However, the behaviors of each type are identical apart from the number of inputs, so I will simply refer to a general `FunctionObject` here. `RVL::-Vector` hides its data with an layer of indirection. `Vector` is an aggregation of a `DataContainer` and a `Space`. The `Space` produces data containers and provides the functionality which defines a mathematical Hilbert space. It has member functions to compute a norm, inner product, and linear combinations. It can also generate the zero vector (additive identity) in that space.

`RVL::DataContainer` is a very similar object to a `TSFCore::Vector`. The DC is maintains a collection of one or more `LocalDataContainer` objects, which serve a similar role to `RTOpPack::SubVector`. However, `LocalDataContainer` has a fixed stride of 1 between elements.

I now refer to Figure 6.5, which shows the evaluation of a function object `SVL-Randomize` on a vector containing seismic data. The `RVL::Vector` interface has an `eval()` method, which simply forwards the call to `RVL::DataContainer::eval()`. The particular behavior then depends on the type of data container, but the following description applies to most. The `eval()` method first casts the function object to the correct type (Unary, Binary, Ternary, or Quaternary) depending on the number of inputs. It then contains a loop over all `LocalDataContainers`. For each LDC, it calls the `operator()` method of the function object on the LDC. The function object

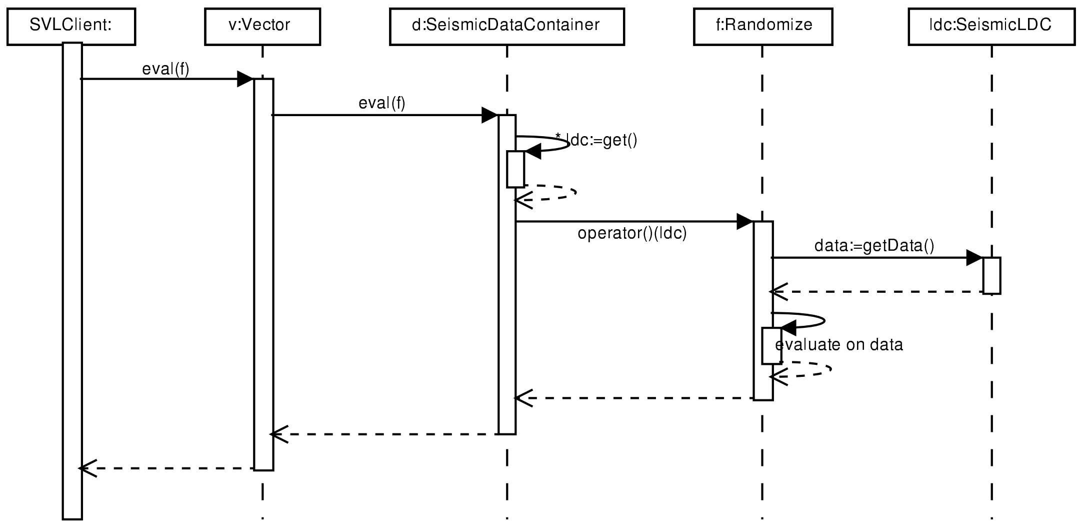

Figure 6.5: Evaluating a RVL::FunctionObject on a RVL::DataContainer

uses the `getData()` and `getSize()` methods of the LDC to perform whatever task it does (in this case, storing a random number in each data element). The function object returns control to the DC, which moves on to the next LDC it owns (if any). Once through all LDCs, the method returns.

Generally, a reduction function object is expected to accumulate results internally. When `getResult` is called, it then converts its internal data to the expected return value. For example, when computing $\|x\|_2$, the function object accumulates $res = \sum x_i^2$ and then does a `return sqrt(res);` when `getResult()` is called. However, such behavior is insufficient when running in parallel and I have an independent instance of the function object on each process.

6.2.1.3 TSOpt

TSOpt [44] is a time–stepping library which was designed for use in opimization with constraints defined by a simulation. At the highest level, it takes a control vector and returns the state vector which satisfies the constraints. To facilitate gradient–based optimization, it can also provide derivatives and adjoint derivatives with respect to the control parameters. The result is an operator

$$S : C \to D$$

where C is the control space, and D is the state space. Objective functions like $J(c,d) = \|F(c) - d\|^2$ are formulated from this operator. The gradients of J are

$$\langle \nabla_c J(c,d), \delta c \rangle = \frac{\partial J}{\partial S}(c,d)\frac{\partial S}{\partial c}(c)\delta c.$$

TSOpt can compute $\frac{\partial S}{\partial c}(c)$ as well as the adjoint of it.

TSOpt is built out of many small pieces, and these pieces are designed to be flexible and reusable. For many problems, a user can simply select appropriate samplers, model builders, and steps off–the–shelf from the available implementations in the library. The `Dynamics` object, which describes the constraint equation

$$F(\frac{du}{dt}, u, c, t) = 0,$$

must be rewritten for each new problem.

TSOpt performs the time–integration by taking a sequence of time steps. It uses a sampler to map the external representation of the control to the internal representation of the control. It initializes the `Clock`, which keeps track of the current time as well as the size of the next time step. Time steps are performed by a `Step` object, which access the `Model` for data and gives this data to the `Dynamics` object when evaluation of the state equations is needed. Derivatives are computed in a similar manner, and adjoint computations utilize a backward stepper, which may be a direct calculation when the equations are time–reversible. In most cases, the Griewank checkpointing scheme [21] works efficiently for the backward stepper when direct calculation is not feasible. This scheme allows users to select a point along a tradeoff curve between memory requirements and computation, but the default settings are well suited for most applications.

The `Dynamics` object also provides support for multistep methods and implicit equations. There is an additional solver method for solving operators of the form

$$(\frac{\partial F}{\partial u'}(u',u,t)+\alpha\frac{\partial F}{\partial u}(u',u,t))x=b$$

As I will discuss later in Section 6.2.4.1, the problem formulation in `TSFCore::Nonlin::TransientNonlinearProblem` is particularly well suited for use by a `Dynamics` object, and provides all of the functionality needed.

6.2.1.4 Moocho

Moocho began life under the name rSQP++. However, since it has expanded to cover algorithms apart from Reduced–Space Sequential Quadratic Programming, it needed a new image. It is designed to solve problems of the form

$$\begin{array}{rrcll} \text{min} & & f(y,u) & & \\ \text{subject to} & & c(y,u) & = & 0 \\ & g_L \leq & g(y,u) & \leq & g_U \\ & u_L \leq & u & \leq & u_U \end{array}$$

using a Simultaneous Analysis and Design (SAND) approach. However, TSOpt doesn't distinguish between state and control variables and it only provides the complete gradient of the operator c. The solution to this is to give Moocho problems which appear to be unconstrained and use an implicit constraint $f(u)=f(y(u),u)$ where $y(u)$ is the solution of $c(y,u)=0$ for a given u.

Moocho is a very large package, but at the top level, the `MoochoSolver` takes a `NLP` object which defines the problem to be solved. I will create the `NLP` using the class `NLPTSFCoreNP`, which is an adapter for a `TSFCore::Nonlin::NonlinearProblem`. See the example in Section 6.2.5.

6.2.1.5 FEM

Generally speaking, FEM takes care of all the details involved in setting up finite–element problems, such as the intricate couting schemes for tracking elements, nodes, and connectivity. A user supplies the problem definition, chooses a basis and a mesh, and out pops the needed matrices and vectors. Of course, no package is quite that simple, but using FEM is far superior to coding this stuff by hand, as it is incredibly easy to make mistakes.

6.2.2 Evaluation on Opposites

Now, I will describe how to utilize a data type from one package on an operator of the opposite package. The general methodology is to wrap the high–level object as something the client recognizes, then inside create a new operator of the type that the internal data object recognizes.

6.2.2.1 Using a FunctionObject on a TSFCore::Vector

All of the necessary objects to perform this task are shown in Figure 6.6. Before describing the sequence of method calls, I should first introduce the new adapters. There are three crucial adapters:

TSFCoreVectorDC This class is a `RVL::DataContainer`. It keeps a pointer to a `TSFCore::Vector`. It can only be created using an existing `TSFCore::Vector`.

FORTOp This class is a `RTOpPack::RTOpT`. It is created to wrap a FunctionObject and provide the `apply_op` interface xexpected of a `RTOpT`.

LocalSubVector This is a `RVL::LocalDataContainer` which simply wraps a `RTOpPack::SubVectorT` and uses its methods to `getSize()` and `getData()` from the `SubVectorT`.

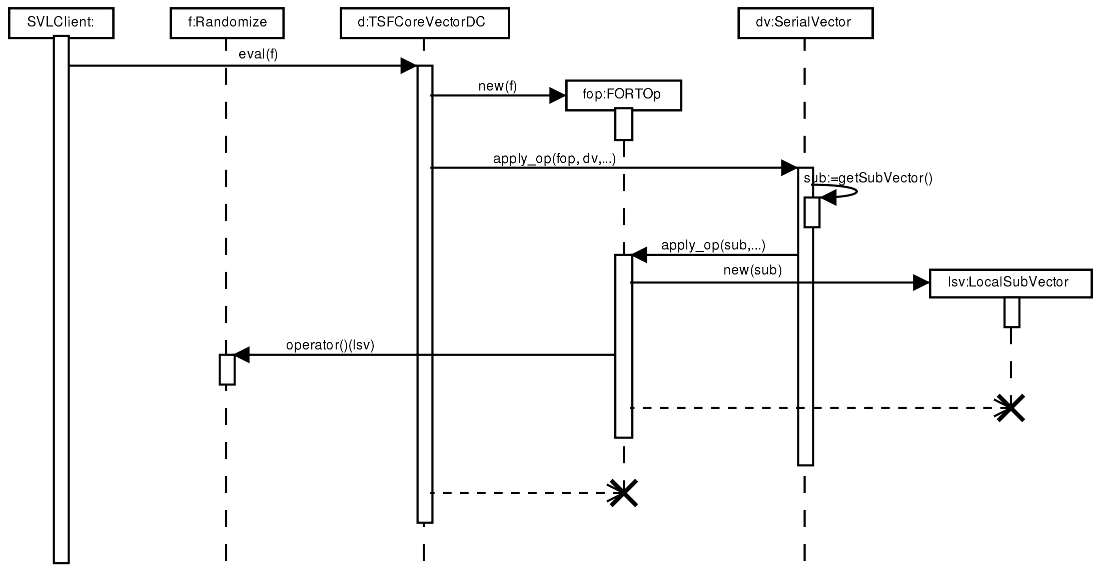

Figure 6.6: Sequence of calls to evaluate a RVL::FunctionObject on a TSFCore::-Vector

It may seem odd to use so many adapters, but it is necessary for getting past the various interfaces.

When the `TSFCoreVectorDC` is given a `FunctionObject` *f* to evaluate, it first builds a new `FORTOp` *op* out of *f*. It then calls the `TSFCore::applyOp()` function on its `TSFCore::Vector` *v* and *op*. This results in a call to `v.apply_op()`. Things proceed as normal, with *v* calling `op.apply_op()` on each of its `SubVectors`.

When `op.apply_op()` is called on a `SubVector` *sub*, it wraps it as a `LocalSub-Vector` *ldc*. This is then passed to `f(ldc)` for the evaluation. *ldc* passes on the `Scalar *` from *sub* so that *f* can work directly with a contiguous array of data.

6.2.2.2 Using a RTOp on a RVL::DataContainer

Now suppose I have a TSFCore client, for example MOOCHO, which owns a collection of RTOps which I would like to use. However, I want to use a `RVL::DataContainer` to handle data storage. How might I accomplish this? In this case, I will need only two new adapters.

DCTSFCoreVector A child of `TSFCore::Vector` equipped to take an `RVL::Data-Container` as input. It will alternatively accept a `RVL::Space` or a `RVL::Data-ContainerFactory`, since both can create the necessary `DataContainer`.

RTOpUFO This is a `RVL::FunctionObject` that wraps a `RTOp`. Note that I must have four of these, one for each number of inputs. In particular, `RTOpUFO` is for an operation with only one input. Further, when creating this object, I must also pass in the number of mutable and immutable inputs, as well as the `ReductTarget`. This extra information is important so that I can correctly call the `RTOp::apply_op` function later.

Luckily, I do not need to implement a new child of `SubVectorT`, since the interface permits construction of a `SubVectorT` from raw data, and I can easily get the raw data from a `LocalDataContainer`. Note that this only involves copying pointers and a couple of integers, so this is not an expensive operation. In fact, all of my adapters never copy data if it can be avoided.

Suppose the TSFCore client owns a `RTOp` *op* and a `DCTSFCoreVector` v. It calls `TSFCore::applyOp(op, v, ...);`, which immediately calls `v.apply_op(op, ...);`. Then v counts the number of input vectors, chooses the appropriate type of `RTOpFO`. Suppose for now that v is the only vector in this operation. Then it will create a `RTOpUFO fop(op, num_vecs, num_targ_vecs, reduct_obj);` Once it has *fop*, it calls `dc.eval(fop);`, where *dc* is the internal `DataContainer`. This causes *dc* to loop through its `LocalDataContainer` members, and on each one, it calls `fop(ldc);`. Finally, *fop* will simply call `getData()` and `getSize()` on the *ldc* and use this information to build a `SubVectorT` *sv*. The original *op* is applied to *sv* as normal.

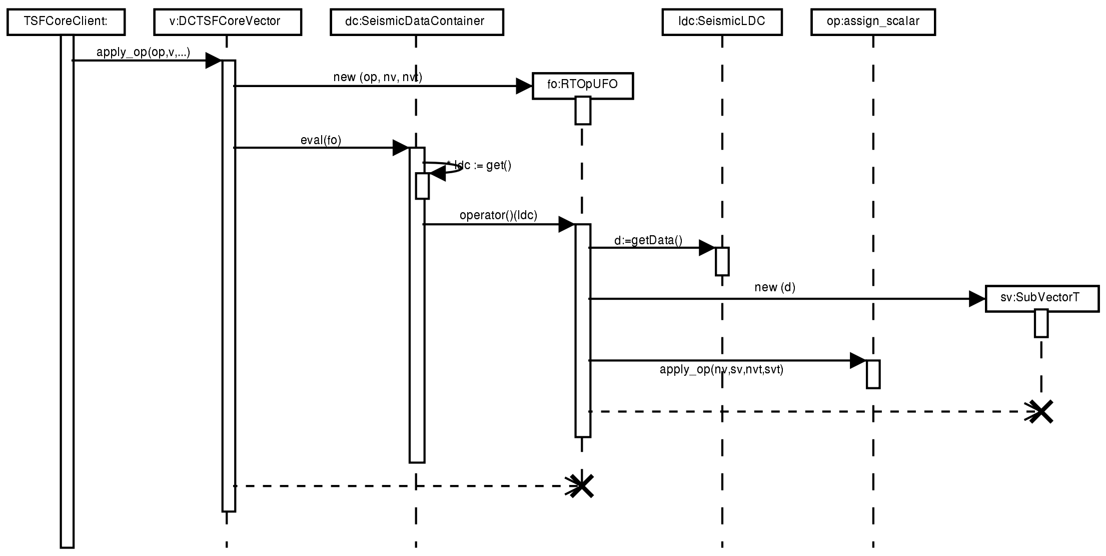

Figure 6.7: Sequence of calls to apply a RTOp to a RVL::DataContainer

6.2.2.3 Difficulties

The type conversions were not necessarily easy. While `SubVector` and `LocalDataContainer` are almost the same concept, some of the other types don't line up as well. A `TSFCore::Vector` can be a `DataContainer`, but without some additions, the opposite is not true. This is because a `DataContainer` has no knowledge of spaces or where it comes from. Similarly, a `TSFCore::Vector` is not a `RVL::Vector` without the addition of the standard functions required by a Hilbert space — namely inner product, linear combination, and a zero element. For simplicity, I attach versions of these functions which are most common in my applications, but this is an assumption and may cause trouble in some cases. Luckily, I mostly do the wrapping in situations where only the `RVL::Vector::eval` function will be called.

Further, the concepts of spaces are slightly different. The `RVL::Space` class has these extra functions I just mentioned. `TSFCore::VectorSpace` more closely aligns with `RVL::DataContainerFactory`. In fact, it is easy to make a `VectorSpace` into a `DataContainerFactory`. This is done in the adapter class `ATN::TSFCoreVectorDCF`.

However, it is somewhat more difficult to make a `RVL::Space` into a `TSFCore::-VectorSpace`. This trouble hinges around the dimensionality of the space. In RVL, a deliberate decision was made to avoid explicit dimensionality at the `Space` level. See the paper [41] for more discussion on this. However, a `TSFCore::VectorSpace` has an public member function `dim()`. They use this to test for uninitialized spaces (dim == 0) as well as to determine compatibility of spaces. If two spaces have the same dimension and utilize the same scalars, then TSFCore says they compatible. RVL takes a tighter notion of compatibility that it closer to the mathematical definition, and leaves the implementation of the compatibility test to each space.

In any case, there is no easy way to determine the size of a RVL space. In order to fully implement the adapter, I had to create a RVL `UnaryFunctionObjectRedn` whose job was to fill in this missing information. Luckily, I can do so without actually touching every element of a vector. I simply need to sum up the size of each `Local-DataContainer`, which has a `getSize()` function that makes this information easily available. So, to compute the dimension of a space, I create an element in that space and then sum up the sizes of each of its `LocalDataContainers`. This unfortunately requires allocating a new vector in space, which makes it an operation best avoided unless necessary.

6.2.3 Higher Level Interoperability

I need to interact with higher level objects in both packages. For example, I might want to apply a `RVL::LinearOp` to a `TSFCore::Vector`. In my large example scenario, MOOCHO asks for a derivative operator from the nonlinear problem definition. This operator is actually built by an RVL package, in this case TSOpt. I can't make TSOpt produce a `TSFCore::LinearOp` without overhauling it, but I can give MOOCHO something that looks like a `TSFCore::LinearOp`.

6.2.3.1 Applying a RVL::LinearOp to TSFCore::Vectors

The RVL linear operator interface only works with `RVL::Vector`. Further, I would like to wrap a `RVL::LinearOp` as a `TSFCore::LinearOp` so that I can encapsulate the `apply()` code as well as be able to pass the linear operator through other TSFCore interfaces (e. g. `NonlinearProblem`). To complete this task, I need three things

LinearOpAdapter is a `TSFCore::LinearOp`. It contains a reference to a `RVL::-LinearOp` and forwards calls to `apply()` after converting `TSFCore::Vector`s to `RVL::Vector`s.

TSFtoRVLVectorBuilder Uses `TSFCoreVectorDC`, mentioned in Section 6.2.2, as well as `TSFCoreVectorDCF` and `StandardSpace` which allow us to convert a `TSFCore::VectorSpace` to a `RVL::Space`.

TSFtoRVLVectorAdapter provides access to the protected `RVL::Vector` constructor so that `TSFtoRVLVectorBuilder` can create a `RVL::Vector` out of the `TSFCoreVectorDC` and `StandardSpace`.

Figure 6.8 shows the sequence for applying a linear operator in this case. I have a TSFCore client which owns at least two `TSFCore::Vector` objects and a linear operator. The operator is actually an RVL object, but is wrapped to mesh with the TSFCore interface. This wrapping is done by `LinearOpAdapter`. When the `LinearOpAdapter::apply` method is called, it takes each `TSFCore::Vector` and creates a `TSFtoRVLVectorBuilder` out of it. The builder creates appropriate `Data-Container`, `DataContainerFactory` (not shown) and `Space` objects for the given vector, then dynamically allocates a `TSFtoRVLVectorAdapter` *vec* from these. It can then return a reference to *vec* as needed, but deallocates *vec* when destroyed. This means that *vec* is not a permanent adapter and is only meant for temporary use. In this case, I can now pass it through the `RVL::LinearOp::apply` interface. From here on the sequence proceeds as for a normal `RVL::LinearOp`.

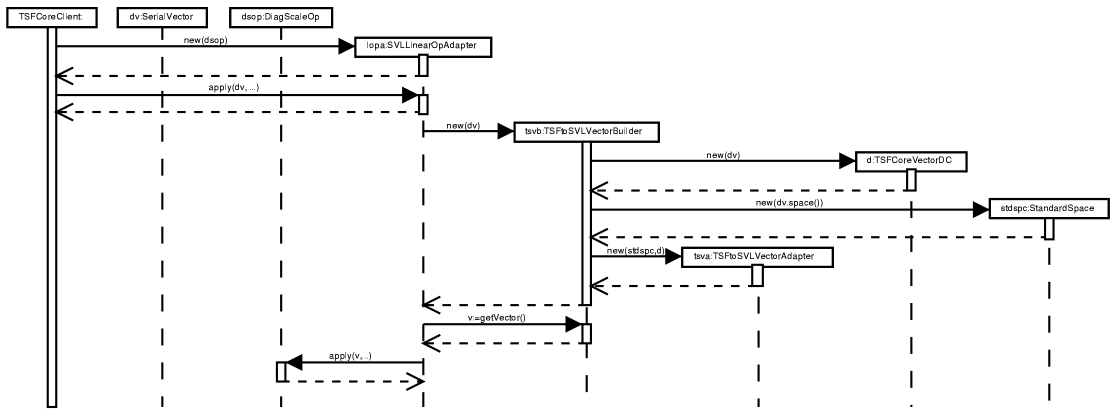

Figure 6.8: Sequence of calls to apply a RVL::LinearOp to a TSFCore::Vector

Note that most linear operators in RVL utilize the `eval()` capabilities of `RVL::-Vector`, so it is quite likely that a sequence similar to that in Figure 6.6 will occur inside. Once this has finished, each method returns and adapters are discarded. This will *not* deallocate the original vector, since the adapter specifically instructs the `RVL::Vector` that it does not own this data. Since this breaks the normal RVL conventions, this behavior is *only* accessible through a protected constructor, thus the need for `TSFtoRVLVectorAdapter`, a child of `RVL::Vector`.

6.2.3.2 Applying a TSFCore::LinearOp to RVL::Vectors

Now, I wish to demonstrate the ability to apply a `TSFCore::LinearOp` to `RVL::-Vector`. In fact, this test will prove more than this, as I can use `TSFCore::-MultiVector` as a linear operator. Thus, I will show that I can in fact interoperate with vectors allocated in different manners. This works because the default `MultiVector` implementation performs a matrix–vector multiply as a series of dot–products. I had to create several new adapters in order to accomplish this task:

TSFLinearOpAdapter This takes a `TSFCore::LinearOp` and wraps it as an `RVL::-LinearOp`. It also uses the `TSFCoreVectorDCF` and `StandardSpace` combination mentioned earlier to handle the type conversion for the domain and range into

`RVL::Space`.

VecTSFCoreVector This is almost identical to the `DCTSFCoreVector` discussed in Section 6.2.2.2. Since I cannot get direct access to the `DataContainer` of a `RVL::Vector`, I must instead wrap that vector and utilize its `eval()` methods.

DCTSFCoreVectorSpace This was mentioned earlier. I've modified it to function given a `RVL::Vector` instead of an `RVL::Space`, since the vector allows us indirect access to the space's methods.

Figure 6.9 diagrams the sequence of call to apply the linear operator. A RVL client is given a `RVL::Vector` *vec* (actually needs a pair of vectors, but I've omitted one for the sake of simplicity) and a `TSFCore::LinearOp` *op* (in my test, a `MultiVector`). It first wraps *op* with a `TSFLinearOpAdapter` and then calls the apply method of the adapter on *vec*. The adapter wraps *vec* with a `VecTSFCoreVector` then calls the apply method of the original op. From here, the normal behavior of a `TSFCore::LinearOp` occurs.

A `RVL::LinearOp` can do both the forward operation and its adjoint. This is not the case with `TSFCore::LinearOp`. Thus, I have built in checks in the `TSF-LinearOpAdapter` which use the `TSFCore::LinearOp::opSupported()` method. If an operation is not supported, and exception is thrown. This matches the expected behavior of a `RVL::LinearOp`.

6.2.3.3 Optimizing an RVL::Functional

I would like to define a `TSFCore::NonlinearProblem` using RVL. The `Functional` class is the natural choice to use, as it provides all of the features needed by `Nonlinear-Problem` and `NonlinearProblemFirstOrder`. A `RVL::Functional` implements the mathematical idea of a functional, namely a map

$$f : X \to Y$$

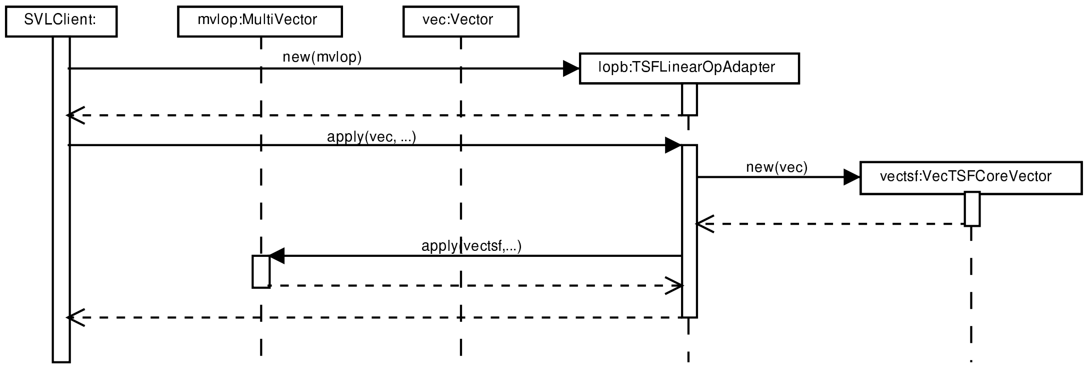

Figure 6.9: Sequence of calls to apply a TSFCore::LinearOp to a RVL::Vector

from the domain X to the range Y. Further, it gives access to the first–derivative of the functional $\frac{\partial f}{\partial x}$.

The class `NPRVLFunctional` is derived from `NPFOUnconstrained`, which is a specialization of `NonlinearProblemFirstOrder`. The `NPRVLFunctional` constructor takes the functional and optional bound constraints on the domain. All member functions of the adapter are either forwarded to the functional following adaptation to a RVL::Vector (see Section 6.2.3.1) or given obvious implementations. For example, `space_g()` simply returns a `TSFCore::VectorSpace` of dimension one.

I gain some efficiency through the use of the `RVL::FunctionalEvaluation` class. When I am asked to calculate quantities at a new point, I create a new evaluation and get the value I need from it. On the other hand, when $newpoint == false$, I can simply consult the previous evaluation without going to the expense of building a new evaluation.

Two adapters are required in order to use Moocho to optimize an `RVL::Functional`. I first build a `NPRVLFunctional` out of the `Functional`, and then build a `NLPTSFCoreNP` out of the `NPRVLFunctional`. This is the nonlinear problem in a form that Moocho recognizes. I can then give the problem to a `MoochoSolver` for minimization.

I have two examples of `NPRVLFunctional` in operation. The first example uses a functional `TOMS566`, which is simply a adapter around the FORTRAN code provided in [34]. This lets me try Moocho on the suite of problems Moré, Garbow, and Hillstrom suggest. Moocho performs as expected, converging to the published minimums in most cases. The few that do not converge are admittedly tricky problems tailored to have extremely poor scaling. The second example solves a least–squares problem using TSOpt [44] to define a ODE constraint of the form $c(y, u) = 0$ and then minimizing $\|y(u) - y^{data}\|$. I will discuss TSOpt more in the following sections.

6.2.4 Putting a TransientNonlinearProblem under TSOpt

Now that I have some interoperability of linear operators and can build a `TSFCore::-NonlinearProblem` out of a `RVL::Functional`, I can focus on using TSFCore to supply the problem definition to TSOpt. This first requires the creation of a `TSFCore::-Nonlin` object which knows about time, and supplies the necessary interface.

6.2.4.1 TransientNonlinearProblem

The new `TransientNonlinearProblem` class is not a child of any current TSFCore class. However, its design is meant to strongly mirror that of `NonlinearProblem`. In fact, these two classes could be siblings if a need was demonstrated, since they have many identical member functions. Both provide access to bounds and spaces for the state variable y and control variable u. They further specify the range of the constraint function c, and methods for setting and retrieving storage for the value of c. In addition to the basic services, `TransientNonlinearProblem` also supplies

y0 initial state as a function of the given control.

dy0 initial perturbation of the state as a function of the given control and control perturbation.

duF final perturbation of the control as a function of the final state and state perturbation, as is needed for the adjoint state method.

calc_c(yp,y,u,t,dt) evaluate the implicit function $c(yp, y, u, t, dt)$ and store the value in c.

There is a natural specialization of `TransientNonlinearProblem` which adds an interface for first–order information, `TransientNonlinearProblemFirstOrder`. This again mimics the `NonlinearProblemFirstOrder` in design. It provides linear operator factories for the derivatives as well as methods for setting and retrieving storage of these derivatives. In this case, there are three derivatives which might be of interest $\frac{\partial c}{\partial y}$, $\frac{\partial c}{\partial yp}$, and $\frac{\partial c}{\partial u}$. There are calculation methods for each which have an identical parameter list to the `calc_c()` method.

It is important to notice that all the calculation methods utilize an optional parameter `newpoint`. When `newpoint == false`, I assume that the other input parameters are identical to those in the previous call to a calculation function, and thus evaluating at the same point. This frequently permits increased efficiencies, and it is important to use this parameter wisely both in creating a `TransientNonlinearProblem` and in calling the calculation methods. In one case, I needed to calculate c, $\frac{\partial c}{\partial y}$, and $\frac{\partial c}{\partial u}$, and therefore used the following sequence of calls

```
prob.calc_c(yp, y, u,t,dt);
prob.calc_DcDy(yp, y, u,t,dt, false);
prob.calc_DcDu(yp, y, u,t,dt, false);
```

6.2.4.2 TransientNPDynamics

The `TSOpt::Dynamics` class is responsible for computing evaluations of a system of partial differential equations. Its function calls require a specific form of calculation, but I can utilize `TransientNonlinearProblemFirstOrder` and `TSFCore::-SerialVector` to perform the necessary work. I chose to use `SerialVector` since the `Dynamics` interface deals with `RVL::LocalDataContainers` but the `Transient-NonlinearProblem` interface only accepts `TSFCore::Vectors`. I can easily re–wrap a LDC u by doing

```
TSFCore::\-SerialVector<Scalar> usv(u.getData(), 1, u.getSize());
```

This results in *usv* accessing the same array of data as u.

The design document for TSOpt [44] includes a thorough description of what the Dynamics methods are supposed to calculate. What follows is simply intended to describe what I did in this subclass, and I advise programmers to consult the TSOpt document for further details.

Given an equation of the form $\frac{dy}{dt} = H(y, u, t)$, the `rhse()` function is supposed to compute

$$y_p = b_0 y_0 + by + aH(y, u, t)$$

However, due to the nature of the `TransientNonlinearProblem` class, I have equations of the form $F(y_p, y, u, t, dt) = 0$. This necessitates computing

$$y_p = by + aF(b_0 y_p / a_0, y, u, t, a)$$

The derivative and adjoint computations follow in a similar manner. I always evaluate F the same way, and omit the addition of the initial terms.

I use the TSFCore functions `assign()`, `Vt_S()` and `Vp_StV()` to carry out the necessary scaling and linear combination of vectors. To ensure accuracy, I am forced to utilize one temporary vector in order to avoid accidently overwriting important data, which can occur when several of the input LDCs point to the same address in memory.

6.2.5 Example Logistics Problem

To prove total interoperability of all the packages, I use a `TSFCore::Nonlin::-TransientNonlinearProblem` to define `TSOpt::Dynamics`. I then build a `TSOpt::-TSOp` and use it to define a LLS functional. Finally, I use Moocho to optimize this functional, as described in Section 6.2.3.3. Figure 6.10 shows the relationships between the objects in this example.

The logistics problem searches for the initial condition $y(0) = u_0$ which gives a desired final result $y(T) = y^{data}$ for some specified T, when y obeys

$$\frac{dy}{dt} = (1 - y^2)$$

The objective is to find $y(T)$ as close as possible to y^{data} in the least–squares sense.

The diagram shows how I build up this application. Given a `TransientNonlinear-Problem`, in this case `LogProbTNLP`, I build a `TransientNPDynamics` from it. I then follow the standard procedures for TSOpt, building a `Stencil` from the `Dynamics` object and a `TSOp` from the `Stencil`. That takes care of the TSOpt portion of the example. I next create a `Functional`, in this case the `LeastSquaresFcnlGN`. The functional is put into a `NPRVLFunctional`, which is a type of `TSFCore::Nonlinear-Problem`. Finally, I build a `NLPInterfacePack::NLP`, and give it to the `MoochoSolver` for optimization. The behavior of the `MoochoSolver` is controlled by the input file "Moocho.opt".

I am struck by the sense that this construction is like a set of nested Russian dolls. I was concerned that there may be much wasted effort involved in such an application. In fact, this is not the case. There are only 3 adapter classes here, and since I have three disjoint packages (RVL, TSFCore, and Moocho), three adapters are the minimum necessary.

TransientNPDynamics Adapts `TSFCore::Nonlin::TransientNonlinearProblem` to `TSOpt::Dynamics`. There is value added here, since I must perform some calculations inside `TransientNPDynamics` in order to produce the correct results. TNLP only gives us F, whereas I need $u_p = b * u + b_0 u_0 + aF$.

NPRVLFunctional Adapts a `RVL::Functional` to a `TSFCore::Nonlin::NPFOUn-constrained`. This is a pure adapter.

NLPTSFCoreNP This lets Moocho optimize any TSFCore problem. This class existed before the ATN project began, and since TSFCore is intended as the standard interface, will be necessary in many projects.

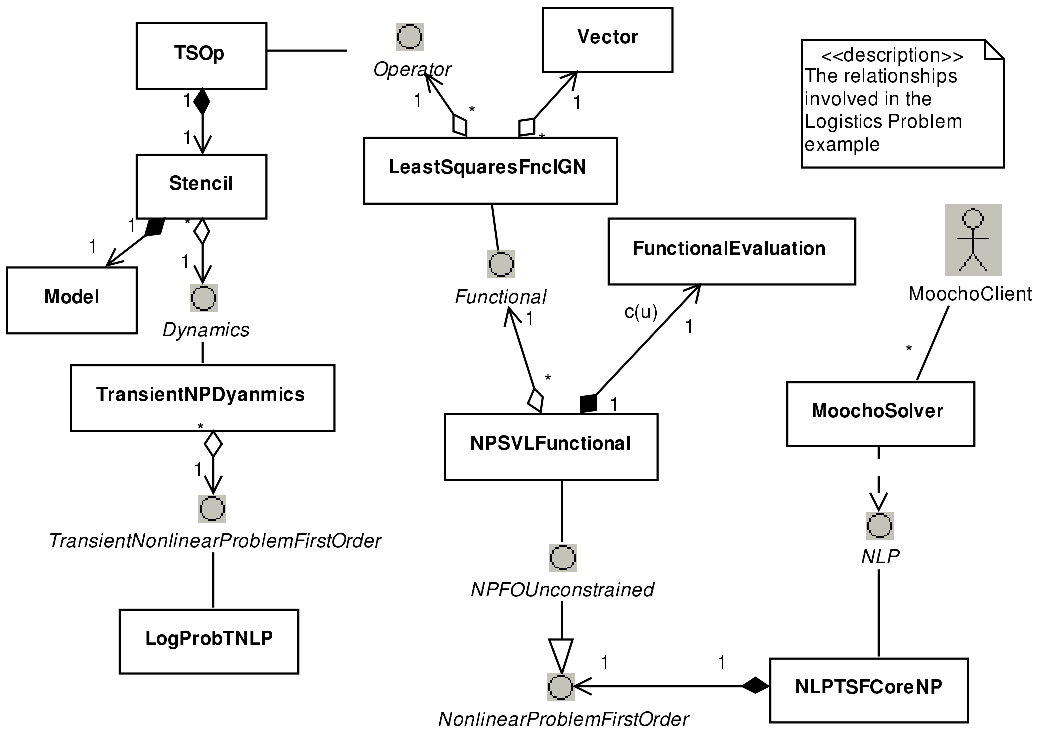

Figure 6.10: Class interaction diagram for the Logistics Problem

All the other classes are abstract interfaces, compositions, or other patterns. Beyond the linear algebra interoperability, the work involved for this logistics example was simply two adapters, both of which should be entirely reuseable due to the use of abstract interfaces on both sides of the adapter. Each adapter implements one interface by calling methods on a second interface, and never deals with concrete types.

Two adapters are all that is needed to connect a time–dependent TSFCore problem into TSOpt into a unconstrained TSFCore problem. This is very little code to gain a huge amount of functionality!

6.3 Epetra and RVL

Epetra was developed by Mike Heroux as a parallel linear algebra library [25]. Although it has been adapted to TSFCore so that it can be incorporated in the Trilinos Solver Framework, I wanted to adapt to it directly instead of going through the TSFCore interface. This would greatly simplify usage by reducing the number of dynamic casts needed to interface with Epetra code as well as demonstrating the flexibility of RVL to handle a library which was designed specifically for parallelism. Also, as Epetra seems to be a simpler interface than TSFCore, a direct adaptation was straightforward.

One limiting factor when dealing with Epetra is the lack of templates. All data in Epetra is double precision data. This means that all the adapters must be concrete classes instead of templates. A templated version named Tpetra is in the works, and modifying the adapters to use Tpetra classes instead should be a relatively easy project later.

A note for readers: the Epetra package uses an underscore naming convention instead of namespaces. Any class names you see with underscores in them are Epetra objects. Any without underscores are either RVL objects or adapters from Epetra objects to RVL objects. The adapters are named using a combination of the Epetra and RVL names.

6.3.1 Adapting Epetra_Vector and Epetra_MultiVector

The basic data storage class in Epetra is the `Epetra_Vector`. On a particular process, each object appears to contain an array of contiguous data. The `Epetra_Vector` has access methods which will expose the data pointer as well as the length of the array. This makes it very easy to treat an `Epetra_Vector` as a `RVL::LocalDataContainer`, since such behavior is all that is required of a LDC.

An `EpetraVectorLDC` can be built from a `Epetra_BlockMap` which describes the allocation of elements on each processor. I also provided a constructor which takes an existing `Epetra_Vector` and makes a copy or view of it. The Epetra interface provides this copy/view functionality. The adapter owns an actual `Epetra_Vector` and the Epetra object can either allocate its own data storage space or reference the space belonging to another object.

The `Epetra_MultiVector` is an array of identical `Epetra_Vector` objects. In RVL terms, this makes it a `ProductDataContainer` because I may treat each element of the array as a LDC. The `EpetraMultiVectorDC` implements the product structure by allocating `EpetraVectorLDCs` as necessary with views of the vectors components of the multivector. The adapter owns a `Epetra_MultiVector` and an array of pointers to `EpetraVectorLDC`. This array of pointers is initialized to null upon construction and filled in as the components are accessed. The adapter's constructor takes a map and an integer size for the array of vectors. Both the map and size are passed directly to the `Epetra_MultiVector` constructor.

As part of some applications involving these adapters, I found that I would like to make a LDC out of a fragment of an `Epetra_Vector`. This was easily accomplished with a slight modifications to the adapters. I affix two additional integer data members `offset` and `trunc` to the adapter class. The `offset` is simply added to the data pointer when the pointer is requested, effectively ignoring the first `offset` data values. The `offset` and `trunc` are both subtracted from the length, causing the LDC to only access elements $offset, offset + 1, \ldots, length - trunc - offset$ instead of elements $1, 2, \ldots, length$. Both `offset` and `trunc` may be set in any of the constructors, but both have default values of 0. When set in a `EpetraMultiVectorDC`, these values are automatically passed on to the owned `EpetraVectorLDC` members.

6.3.1.1 Accessing the Functionality of the Epetra_Vector

Thus far, I've described taking a parallel vector and chopping it into local arrays which I may treat as local data containers. However, at this stage, such classes are no more useful than the basic `RVL::RnArray`. The adaptation is hamstrung unless I can take advantage of the parallel functionality of Epetra.

The `Epetra_Vector` has many computation functions which calculate in parallel the common operations seen in algorithms. Some of these are elements–wise operations like the linear combination `Update()` method. Such operations are trivial to parallelize as long as all participants use the same element map. However, the reductions, like norms and inner products, involve some parallel communication. I can use these methods by writing function objects. Each function object tries to cast its inputs to `EpetraVectorLDCs`, then call the correct `Epetra_Vector` method on the data members of the LDCs. The LDC adapter has a `getVec()` method to allow access to the `Epetra_Vector` it owns as well as having individual methods like `inner()` which call the appropriate Epetra methods. For example, the `EpetraVecInnerProduct` is a binary function object implemented thus:

```
 /** Evaluation method */
    virtual void operator()
      (Local\-Data\-Container<double> & x,
       Local\-Data\-Container<double> & y) {
      try {
EpetraVectorLDC & a = dynamic_cast<EpetraVectorLDC &>(x);
EpetraVectorLDC & b = dynamic_cast<EpetraVectorLDC &>(y);
setValue(a.inner(b));
      } catch (bad_cast) {
RVLException e;
e << "Error in EpetraVecInnerProduct::operator()."
        e << "  Cannot cast input to EpetraVectorLDC\n";
throw e;
      }
    }
```

It is important to note that the Epetra methods *do not* take into account the `offset` and `trunc` values. If an application needs to use nonzero values of these, it will be necessary to write new reductions in the cases where extra zeroes would modify results.

6.3.1.2 Making a Space

Once I have a `DataContainer`, inner product, zero, and linear combination methods, these are easily combined into a `RVL::Space`. The function objects are packaged into a `RVL::LinearAlgebraPackage` and a `EpetraMultiVectorDCFactory` is built to create data containers. The factory takes the same constructor inputs as the multivector adapter.

The space is built using the facade `RVL::StdSpace`. It has a linear algebra package and a data container factory which it references through two protected access methods. The base `StdSpace` class implements all the necessary functionality of a space by calling these methods and using the two objects.

6.3.2 Adapting Linear Operators

Given a `Space` that hides `Epetra_Vectors` at the bottom, I need to be able to perform all of the major linear algebra operations in parallel. I've already described the basic transformations and reduction. What's left is the application of a linear operator.

Some packages choose to treat multivectors as linear operators. However, this is not always efficient for parallel computation. Epetra uses a separate class, the `Epetra_RowMatrix`, for its linear operators. The main method of this class is the `Multiply` method which takes an input vector, an output vector, and a boolean flag to indicate whether to transpose.

Since the `RVL::LinearOperator` class interfaces with `RVL::Vector`, I must write a function object which can visit the adapted local data containers and perform the call to `Multiply`. The `EpetraMatVec` is a `BinaryFunctionObject` which takes the

Epetra_RowMatrix and transpose flag in its constructor. This allows me to reuse the same function object to implement both apply() and applyAdj() methods in the linear operator. The function object's computation method is very simple:

```
virtual void operator()(Local\-Data\-Container<double> & y,
                        Local\-Data\-Container<double> & x) {
   try {
      EpetraVectorLDC & x1 = dynamic_cast<EpetraVectorLDC &>(x);
      EpetraVectorLDC & y1 = dynamic_cast<EpetraVectorLDC &>(y);

      int errcode;
      if( errcode = mat.Multiply(trans, x1.getVec(), y1.getVec()) ) {
         RVLException e;
         e << "Error in EpetraMatVec::operator() - ";
         e << "Epetra_RowMatrix::Multiply returned with error code";
         e << errcode;
         throw e;
      }
   } catch (bad_cast) {
      RVLException e;
      e << "Error in EpetraVecLinComb::operator().  ";
      e << "Cannot cast input to EpetraVectorLDC\n";
      throw e;
   }
}
```

Most of that code is exception handling. The actual work occurs in two dynamic casts and a call to Multiply().

The linear operator stores a reference to the Epetra_RowMatrix. Besides being applied, the operator must provide access to domain and range spaces. These spaces

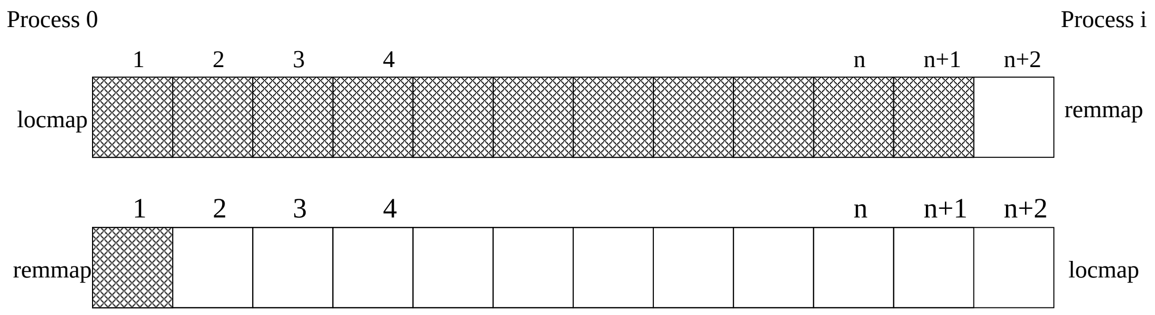

Figure 6.11: Epetra maps used in the data transfer methodology. Filled boxes indicate data on process 0.

are built from the domain and range maps obtained from the row matrix, using the `EpetraMultiVectorSpace` described earlier.

6.3.3 Data transfer using Epetra

Although Epetra was designed for tightly coupled SIMD parallelism, I have found other uses for it. Through clever use of maps and the import/export functions, I can use Epetra to transfer data between MPI processes. This is necessary for one implementation of the Master/Slave paradigm in Chapter 8.

MPI allows users to create communicators from subgroups of a given communicator. After fetching the group for `MPI_COMM_WORLD`, I build a subgroup from process 0 and process i, then build a comm for the new group. This new MPI communicator is used to build an `Epetra_MpiComm`, which can be used to build maps between the two processes.

```
MPI_Group group_world;
MPI_Comm_group(MPI_COMM_WORLD, &group_world);

int procranks[2];
procranks[0] = 0;
procranks[1] = i;
```

```
  MPI_Group_incl(group_world, 2, procranks, &(subgroup));
  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &(subcomm));
  if( rank == 0 || rank == i)
    ecomms = new Epetra_MpiComm(subcomm);
```

After running this code, I have an Epetra communicator between just two processes. This comm is used to build a special pair of maps, as shown in Figure 6.11. Suppose I want to move n doubles from process 0 to process i. Process 0 creates two maps:

```
locmap = new Epetra_Map(n+2, n+1, 0, *(ecomms));
remmap = new Epetra_Map(n+2, 1, 0, *(ecomms));
```

Both maps contain a total of $n + 2$ elements, because Epetra seems to require that each process involved in a map own at least one element for this trick to work. The `locmap` on process 0 has $n + 1$ elements locally, which implies 1 element on process i. The `remmap` on process 0 is the opposite, with only 1 element locally. On process i, I create the same two maps, but I must reverse the order of creation. The map constructor involves global communication across a communicator, so the order is crucial.

The data transfer occurs when I remap data from one map to the other. To move data from process 0, I create a source vector $from$ using the first map — `locmap` on process 0 and `remmap` on process i — and a destination vector *to* using the second map — `remmap` on process 0 and `locmap` on process i Such vectors must be created on both processes. Process 0 fills the data into the $from$ vector and then both processes call the second vector's `Import()` method. On either process, this looks like

```
Epetra_Import mover(to.Map(), from.Map());
to.Import(from, mover, Insert);
```

As long as *to* and $from$ are named correctly, this command is the same, so I encapsulated this code in a function object named `EpetraDataTransfer`. This binary function object casts its parameters to `EpetraVectorLDC` and uses the `getVec()` method to grab the `Epetra_Vector` inside each parameter.

Data transfer proceeds identically when returning data to process 0 except that the maps are switched .

6.3.4 Combination Data Containers

One of the main ideas in RVL was to permit bundling of object meta–data along with array data in such a way that it can be used when needed but is invisible otherwise. Function objects which only touch array data may simply use the base `LocalDataContainer` interface. More specialized function objects, like one that performs interpolation of gridded data, should cast LDCs into a specialized type which has the right meta–data.

If I want to use Epetra to perform data transfer for general data containers, I need a way to handle the extra meta–data. The simplest solution was to pack all the meta–data into an extra `Epetra_Vector` and use the very same data transfer functionality. An application developer simply needs to pack their data into an array of doubles and unpack it on the other side.

I created a general interface for data containers which owned an extra local data container for auxiliary data. The base `AuxComboLDC` owns a reference to a local data container of auxiliary data. This data is often shared between the local pieces of a data container, so I did not want to insist on each LDC owning a copy. The `StdAuxComboLDC` fully implements the functionality of the abstract interface by taking two LDCs in the constructor and treating one as data and the other as auxiliary data. The concrete class can either reference both externally or assume ownership of the actual data when given a pointer instead of a reference.

I wrote two utility function objects to aid interactions with the auxiliary data container. The `AuxRedirectorUFO` redirects the action of a unary function object onto the auxiliary data container instead of the actual data. `CopyToAuxData` is a binary function object which copies the contents of the second parameter onto the auxiliary data of the first. In most cases, more specialized function objects will be needed to serialize the various data members of a local data container. For regularly gridded data, this requires copying a few integers and doubles. For seismic data, it is a more daunting task to serialize all the various data which describes a shot, including time–stamps, location data, and many other bits of information.

6.3.5 Example applications

I wrote several examples which use the Epetra–to–RVL adapters. The Master/Slave task level parallelism is described in Chapter 8. It uses the data transfer methods discussed in Section 6.3.3 to move task data and results between the master and slaves. The parallel finite–element project in Chapter 9 began with code written by Denis Ridzal to assemble the matrices for a finite–element discretization of the advection–diffusion problem. Using the adapters, I built a `RVL::Functional` which computes a quadratic objective function subject to an implicit linear constraint. I've optimized this functional using the LBFGS algorithm described in Section 7.6.5.

6.4 Other Adaptations

The adaptations between RVL, Epetra, and TSFCore were completely successful. In addition to these, I considered adaptation involving HCL and OOQP.

1. HCL is already being used in by the research community. One of the biggest questions asked to the developers of RVL is "I already have all this code written using HCL. Will I be able to continue using it if I switch to RVL, or do I have to redo all the hard work I've put into it so far?" Similar questions arise in any context where a group switches to a seemingly incompatible new interface.

2. As part of solving optimization problems, quadratic programs (QPs) frequently

arise as subproblems inside a solution algorithm. I would like a fast and flexible QP solver which I could take off the shelf and use, without needing to build my own. This has prompted me to consider using OOQP and interfacing with its linear algebra classes

These adaptations were less successful, usually due to limitations in the non–RVL interface. I will detail the problems encountered below. In both cases, a partial adaptation could have been made, but would not have been particularly useful or efficient, so I never actually implemented any adapters.

6.4.1 HCL and RVL

How might I adapt the `RVL::LocalDataContainer` to a `HCL_Vector_d`, the HCL [17] double precision vector class? I first encounter the problem that I may only reasonably adapt `RVL::LocalDataContainer<double>` rather than the templated base type. This is a recurring problem when trying to hook templated packages to non–templated packages. I could make an LDC into a `HCL_Vector_d` by implementing a `RVL::FunctionObject` for every one of the more than 50 methods in `HCL_Vector_d` and then using the correct one when a method is called. However, the reverse could be difficult. Because HCL is written in C++ without using many modern features such as namespaces and templates, it greatly impairs interaction with a package which does. Luckily, the HCL authors were careful to choose unique names with the `HCL_ObjectName` convention as a substitute for namespaces. Further, `HCL_Vector_d` has `Data()` and `Dim()` methods I can use. Thus, it actually exposes a data pointer in addition to the list of standard methods, which allows me to sidestep any problems.

Why wouldn't I match `HCL_Vector_d` with `RVL::Vector` instead of `RVL::Local-DataContainer`, seeing as `HCL_Vector_d` claims to be a vector in Hilbert space and is generated by `HCL_VectorSpace_d`? The answer is that they are in fact incompatible. `RVL::Vector` is fully encapsulated, so an adapter from RVL to HCL could implement the `HCL_Vector_d` methods like `Add()` and `Max()`, but could *never* implement

`Data()`! However, there is a `HCL_Vector_d::Space()` method and `RVL::Vector` is the only class in the RVL hierarchy that knows where it came from. So I could *never* take just a `RVL::DataContainer` or a `RVL::LocalDataContainer` and make it into a `HCL_Vector_d`, without having a `RVL::Space` along with the DC or LDC. I am forced to wave the white flag.

- an adapter from `RVL::Vector` to `HCL_Vector_d` cannot implement a `Data()` method.
- an adapter from `RVL::DataContainer` to `HCL_Vector_d` cannot implement a `Data()` method or a `Space()` method.
- an adapter from `RVL::LocalDataContainer` to `HCL_Vector_d` cannot implement a `Space()` method, and although I could use `LocalDataContainer::-clone()` to fake the factory behavior of a space, there is no way to fake the necessary comparison tests.

Therefore, I could adapt a `HCL_Vector_d` to whatever level of data storage in RVL I wanted, but it is *impossible* to completely adapt an RVL object to HCL. This is entirely due to the mixing of methods and jobs in HCL's vector class.

As a final note, after digging deeply into the HCL documentation, I found that it is perfectly acceptable for the `Data()` function to simply returning an error code. Having methods which might work some of the time is a poor design practice. In this case, it makes it possible to adapt `RVL::Vector` to `HCL_Vector_d` by implementing the other methods and making calls to `Data()` return an error and sacrificing some functionality.

6.4.2 OOQP and RVL or TSFCore

The Object Oriented Software for Quadratic Programming (OOQP) package has vectors which are fully encapsulated. The `OOQPVector` interface has more than 35 pure virtual methods. First, note that OOQP only uses `double` types, so I hit the same template problem I did with HCL. As a result of the strict encapsulation, there is only one apparent way to implement an adapter which makes an `RVL::Vector` or `TSFCore::Vector` into an `OOQPVector`. In the case of RVL, I would need to implement roughly 15 new function objects which would do the various method calls (some of the 35 already exist). The work to adapt TSFCore would be similar, writing new RTOps. However, there are three methods in `OOQPVector` which are troublesome:

- `void copyIntoArray(double v[]) const;`
- `void copyFromArray(double v[]);`
- `void copyFromArray(char v[]);`

These methods require that I be able to serialize any OOQPVector. There are several cases of existing vectors for which this would fail, for example the out–of–core vector in RVL and the parallel vectors in TSFCore. I could probably write visitors which would accumulate data into such arrays, but they are unlikely to function properly in some cases.

In the other direction, I can see only one way in which I might adapt `OOQPVector` into a RVL or TSFCore interface. If I utilize the aforementioned copy to/from array functions, I could take a given `OOQPVector` and when needed copy the data out, transform the data, and copy back to the `OOQPVector`. This would discard most of the functionality given by OOQP, and replace it with an extremely inefficient copying mechanism. I therefore add two extra copy operations per element and lose any benefit of complex data structures in the `OOQPVector`. This demotes an `OOQPVector` to a `RVL::LocalDataContainer` or a `RTOp::SubVector`. Since fully functional LDC and `SubVector` classes exist which provide identical functionality to such an adapted `OOQPVector`, there is no benefit to adaptation here.

6.4.3 OOQP and HCL

Since I've thus far focused on adapting things to RVL and TSFCore, which both utilize exposed data pointers at the base, let's briefly consider adapting two packages which utilize encapsulated arrays as the base type. Could I adapt `OOQPVector` and `HCL_Vector_d`? The design of OOQP was strongly influenced by HCL and they share many common features.

The packages have different lists of 'standard' methods for vectors, which means I can't simply forward all calls. However, most of the functionality is similar, and could be easily implemented using an occasional temporary vector and two or three method calls. In fact, `OOQPVector` has only three methods without such obvious implementations : `gondzioProjection()`, `divideSome()`, and `findBlocking()`. In the other direction, `HCL_Vector_d` has a few difficult methods as well : `Sqrt()`, `Exp()`, `Log()`, `Power()`, and `Greater()`. In both cases, the more complex functions could be implemented using the copy methods on `OOQPVector` or the `HCL_Vector_d::Data()`. Unfortunately, such an implementation would not work efficiently in all cases, and may not be guaranteed to work at all.

This appears to be a fundamental problem with all such designs. Each author has their own list of "absolutely necessary" functions, and not all such functions can be replaced by a sequence of calls to other functions. For example, as long as a design includes a scalar multiply and add, I could compose all different styles of axpy operations. From a component–wise multiply and an invert, I can build a component–wise divide. But, there isn't an easy way to build a component–wise exponentiation. This forces me to resort to the kludges provided for data access, which are frequently inefficient and not always guaranteed to work.

6.5 What was Learned

In order to assist future designs, I consider which features of the various interfaces make adaptation easy and which cause difficulties. I don't find insurmountable obstacles, but there are a few nasty speed bumps.

The first major aid I shall call a common cultural background. Because mathematics is an old discipline, there are many well established and useful concepts for us to build on. These provide a language for communication as well as influencing designs to have similar features. Consider the definition of a Vector Space:

- A set of vectors,
- an associative and commutative vector addition operator,
- an additive identity,
- A scalar field,
- an associative and distributive scalar multiplication operator,
- for a Hilbert space, also need an inner product.

Since the C++ language doesn't have a clear notion of sets, I interpret the idea of a set as being able to produce an element of the set and determine whether a given object is a member of the set. Thus, in many packages, a space is a factory for vectors along with membership tests and some basic operations. This then implies certain behaviors on a vector — it stores data, can be operated on, and knows where it came from. The mathematical definitions provide guidelines for behavior.

The second lesson is critical to all programming — keeping designs simple and organizing classes into atomic pieces, then building structures out of these pieces, produces a flexible design with much reusable code, which is also easier to adapt to other packages. A class with a ton of functionality may be useful to the design at the moment, but such a class is hard to modify and difficult to adapt to other packages with different ideas.

Consider the three common approaches to accessing the data in the contiguous arrays:

A. exposed data pointers, as in `Scalar * LocalDataContainer::getData()`

B. C–style `[]` or FORTRAN–style `()` indexing

C. complete encapsulation, but list of 'standard' methods

Method A is flexible, extensible, and easy to adapt to all other methods, but it breaks the encapsulation and lacks support for ideas like sparsity. Method B is also flexible and extensible, and it is only slightly more difficult to adapt. It still somewhat breaks the encapsulation by providing direct access to data and, when coded poorly, can be extremely inefficient, requiring a virtual function call per element access. Method C provides a well encapsulated data structure and can offer very efficient methods. However, it is very hard to adapt such interfaces, even to other packages using Method C. It is difficult to extend a Method C interface without breaking existing code. Implementing a concrete classes which satisfy this interface can require much work, since it is often necessary to implement every 'standard' method. Algorithms are restricted to using the 'standard' methods, and it is impossible to provide a short list of methods which will cover most algorithms, as the designers of HCL found to their dismay. Frequently, method C classes provide a function for copying into/out-of the container. Such copying allows extensions on the standard methods, but at a cost of increased inefficiency.

When adapting packages which use different access methods, I can always adapt down the list (e. g. from Method A to Method B). It is impossible to adapt up the list efficiently and sometimes completely impossible to do.

Part of the reason it was easy to adapt `RVL::LocalDataContainer` and `RTOp::-SubVector` was that both using Method A of exposing data pointers. Thus, it was easy to implement the `LocalDataContainer::getData()` method by simply returning the pointer from `SubVector::values()`. The other main method is `LocalDataContainer::getSize()`, which is mapped to the `SubVector::subDim()` method.

6.5.1 Roadblocks to Adaptation

A number of design practices and languages features can present roadblocks when trying to adapt packages. Thus far, they can be worked around, but they do present complications. The following is not offered as a critique of these design choices, but as a description of their effect on the adaptability of a package.

6.5.1.1 Const

The C++ language allows programmers to use a `const` declaration in a number of ways. A variable can be declared `const`, meaning its value is set on initialization and can't be changed later. This was initially used to declared things like `const float pi = 3.141592;` but can be used on class objects as well. Further, data members of a class may be declared `const`, and they are not allowed to be modified in any method of the class. Next, a parameter of a function call may be declared `const`, which means that the function must treat this parameter as a constant. This is primarily used to perform a pass–by–reference for efficiency which does not allow changes to the referenced variable. Finally, a method of a class can be made `const`, which guarantees that the method does not change the internal state of the class.

When one package utilizes `const` and another does not, problems arise in adaptation. It is relatively simple to wrap an object with `const` in an interface lacking `const` declarations, as I may always ignore the guarantees and can cast–away `const` when necessary. However, problems arise when I try to use an data member without constant methods inside a constant method or as a constant parameter. Luckily, due to the way C++ treats pointers, I can frequently work around these difficulties. If a class owns a pointer to an object, as long as the pointer's value does not change, anything can be done to the object which it point to. Thus, adapters inherit off one interface and own a pointer to another.

6.5.1.2 Handling Reductions

A reduction is a mapping $f : X \to Z$ from a vector space X to an arbitrary return type Z. The lack of restrictions on the type of Z can be causes some difficulties in designing such a mapping in a programming language. There are several solutions to describing a return type. I could use a `void *` pointer, which can point to anything, and assume that the concrete functions and callers know the proper type of the object. I could template the reduction type in the interface. Finally, I could create an abstract base class for the return type and then inherit from it to implement concrete types.

Each of these choices has some drawbacks. Using `void *` pointers is inherently not type safe. A templated interface in fact creates a set of interfaces indexed by the template parameter, and unless I know the parameter a priori, I can't create a pointer to the base class. This can make some applications difficult, as I could often fully implement an algorithm by calling the virtual methods in the base interface, but such an algorithm would have to become templated as well to be useful. As a concrete example, RTOpPack uses `void *` and RVL used to have templated return types. This made implementation of the `RVL::FunctionObject` to `RTOpPack::RTOp` adapter nearly impossible, since I could not cast a `UnaryFunction-Object` to a `UnaryFunctionObjectRedn<RetType>` without knowing the `RetType`, and there was no way to access such information. This difficulty in addition to similar problems elsewhere lead us to create an abstract base class `RetType` and a base `Reduction` interface with a method

```
RetType & getResult()
```

Further, I can put some useful methods in the base `RetType`, including an assignment operator, clone method, and a reinitialize method. I may then provide useful default implementations of the methods in the `Reduction` interface using the `RetType` methods.

I must consider some drawbacks to the use of an abstract interface as the return type. I can no longer put a call to the `getResult()` method of a reduction f inside an expression. I must instead do

```
ScalarRetType<double> x = f.getResult();
```

where `ScalarRetType` is the implementation of the `RetType` interface for a single scalar. I can then use x just like a normal scalar.

If I change the templated interface of RVL to one based on this abstract base class, it becomes possible to fully implement adapters to and from `RTOp`. It further simplifies other pieces of code, as I can always declare a `RetType *`, allocate a new object of the appropriate `RetType` child from the reduction, and store a result into it.

There is one more difference in the reduction handling of RVL and RTOpPack. In RVL, a `Reduction` is also a `FunctionObject`, and thus when the `Function-Object::eval()` method is called, the object is expected to buffer the results internally. The FO may choose to not form the final result, and instead store an intermediate value. Then, when `Reduction::getResult()` is called, the FO must calculate the final result and return it. In an `RTOp`, every object is treated as a reduction by default with the result as a parameter on the call to `RTOp::apply_op()`. The `RTOp` has a separate method for taking two results and combining them into one in the proper manner. For example, with the usual dot–product, this method simply adds the intermediate results together and returns the sum. The RVL project has since adopted this idea of an `accumulate` function, as such a function proves useful in many contexts. However, in RVL, not every `Reduction` is an `Accumulation`, nor is every `FunctionObject` a `Reduction`.

6.5.1.3 Pervasiveness of Parallelism

Packages frequently make different choices about how to support parallelism and other advanced architectures. These choices can impact the ease with which a package can be adapted. `RTOp` was designed with MPI–style parallelism in mind. Thus, the base interface offers methods specifically for converting internal state into a format which MPI desires, as well as taking advantage of the efficient reduction capability in MPI. RVL chose to accommodate parallelism through subclassing. This makes it fairly easy to adapt `RTOp` into a `FunctionObject` as I may simply ignore all the extra functionality. On the other hand, it is impossible to fully implement an `RTOp` given just a base `FunctionObject` interface. I must instead try to dynamically cast the `FunctionObject` to a subclass which provides needed functionality, and play some tricks to massage that functionality into what I now require.

The RVL group had already worked out a methodology for transmitting objects through a socket using a combination of a `Socket` class and a mixin `Streamable`. A `Streamable` object can read/write its internal state to/from the `Socket`. By creating an implementation of `Socket` called `StateExtractor` that, instead of sending data across a network, simply buffers the data internally, I can then implement the methods required by `RTOp`. In each case, I create a `StateExtractor`, cast the `FunctionObject` to `Streamable`, and use the `StateExtractor` to do what is needed.

6.5.1.4 Different Calling Conventions

When designing a general interface for mappings in vector spaces, I immediately encounter a difficulty — how do you make an interface for an arbitrary number of vectors? I can write methods that take up to n parameters for any fixed n, but I don't want to have to pass around $n-1$ null pointers in the case of a simple unary operation. I could instead write methods which take an array of vector pointers as input, but this forces us to build these arrays every time I wish to call such a function, and of course I also need to pass in the size of the array. RVL takes the former approach, with $n = 4$, and has unary, binary, ternary, and quaternary function objects. RTOpPack takes the later approach, and further partitions the inputs into mutable and immutable vectors. This difference makes for some mildly ugly adapter code, simply for translating two arrays of pointers into the appropriate number of `RVL::Vector`s and vice–versa. Further, out of efficiency considerations, RVL has added query functions which will report whether a function object intends to read from or write to the i^{th} input.

Chapter 7

A Recursive Framework for Flexible Algorithm Design

Notice that most optimization problems can be formulated as minimizing a function f over a subset S of a space X. Further, S is often defined as the intersection of a list of equations, where each equation can be written as $g_i(x) = 0$ or $g_i(x) \geq 0$. Often, assumptions are made on the nature of the functions f and g_i, but these assumptions are made for the sake of the algorithm and are not part of the problem definition. Thus, all that is truly required to define an optimization problem is a method for evaluating functions and a test for membership in a set.

Similarly, optimization algorithms have common structures. The algorithm is given an ordered list of parameters and inputs. An iteration consists of deciding whether the algorithm is finished, and if not update some or all of the state variables. These updates are often the solution of a smaller or simpler problem, or several such problems. Each subproblem exhibits this same recursive structure, until the subproblem becomes small enough to be solved directly, which in most cases means a linear system. This also suggest that while some algorithms are coordinate–free, meaning they look identical on a large class of topological spaces, there eventually must be some algorithms that deal with inputs on an intimate level. A simple example of such a low–level algorithm is the calculation of the inner product. This is often calculated in a loop over the elements of a vector, and thus the algorithm needs to know how to access elements and how many elements there are. Therefore, it is impossible to entirely divorce algorithms and implementation, but it is possible to hide implementation in easily reusable operations.

This leads us to consider whether already existing packages have already accomplished these goals. While each of the previous works contained some good ideas, most packages were written from a pragmatic approach, and fell short. I will examine these projects and learn from the benefits and drawbacks of each. One crucial problem in many of these prior works is the pervasiveness of implementation details in the interface. Ideally, a clean interface should be implementation independent. This allows alteration of the underlying implementation cleanly without having to alter all of the code which uses a package. However, the simple fact that many authors have tried to construct an object oriented system for scientific computing suggests that it is a worthwhile pursuit. It also cautions that it is easy to lose sight of the large goals and allow implementation details to creep in. Finally, I would like to reuse existing code as much as possible, so that I avoid duplicating work, and the prior projects provide a wealth of source material.

A design philosophy must exist before building class objects. I need to identify the desired behaviors of the objects in our package. I need to understand the relationships between the sets of objects, and should solidify these relationships into the class hierarchy. Inheritance provides an vehicle for describing such hierarchies. However, our previous experience with inheritance has shown that it is somewhat limited and is easy to abuse. It is usual to think of a C++ class as a set of related objects, and inheritance from a class describes a subset of the parent set. This is not precisely true. C++ inheritance says that the child class has all the public and protected data members of the parent, as well as all public and protected member functions. The scoping rules allow a child class to reimplement functions and replace data members, meaning that while the child and parent are related, it is not strictly a "subset of" sort of relationship. However, although inheritance can enforce consistent interfaces (syntax), behavior must be enforced by fiat. An abstract base class should have no data members and no function implementations, and therefore there is nothing a child can overwrite. The language does not aid in enforcing consistent behavior (semantics) in children Thus, the abstract base class must document *what* each method in the interface is supposed to do, but leave the details of *how* to a particular implementation.

7.1 Definition and description

Ideally, classes should provide an image of the mathematical world. This would allow easy translation from mathematics into working code. There are several difficulties that need to be worked around. The mathematical world can deal with infinite sums, infinite precision arithmetic, and infinite sets. The computational world is finite, so it at some level requires consideration of stopping criterion and arithmetic error. Computers cannot manipulate entire sets at once, instead dealing with members of sets.

There are several levels of abstraction to consider among algorithms. Some algorithms are coordinate–free, meaning they can be described entirely with high–level operations on vector spaces. This still requires implementation of these operations on a lower level, but these implementations can often be reused. It is impossible to entirely eliminate the details, but if I can isolate them on one level, it should improve my ability to implement and modify algorithms. As an example, consider the Conjugate Gradient method for iteratively solving a symmetric positive definite linear system $Ax = b$ (psuedocode for this algorithm is in Section 7.5). This algorithm employs inner–products, matrix/vector multiplication, and simple vector algebra. Thus, an implementation of this algorithm is independent of the particular Hilbert space which defines the domain. By swapping definitions of vectors and their algebra, the same implementation of the Conjugate Gradient Method can be reused to solve linear problems on any Hilbert space.

The Rice Vector Library (RVL)[41] is written in C++ and has very nice interfaces and implementations of objects including `Space`, `Vector`,`LinearOperator`, and `FunctionObject`. These suffice for encapsulating data and handling most basic operations on data, as well as defining maps between vector spaces. However, this leaves us writing more sophisticated algorithms in the traditional manner, i. e. as a function call with some input parameters and some output parameters. This puts the programmer in a strange, middle ground where the data and many mappings are objects, but the solution algorithms are functions.

If an algorithm is an object as well, then it can:

- have persistent state to store intermediate results and avoid unnecessary computation
- be created, destroyed, and manipulated at runtime.
- exist in multiple copies of the same type of algorithm with separate states, which is not possible with static variables in functions.
- have well defined interfaces, easing reuse and modification.
- take advantage of inheritance.
- be passed as a parameter to other objects. For example, passing in the appropriate linear solver for a given linear operator in order to define an inverse.
- provide easy specification of default settings through the use of default arguments.

These are significant benefits over procedural programming.

7.1.1 What is an Algorithm

Before I can begin to design an `Algorithm` object, I need to define what an algorithm is and what it can do. Here are several definitions of an algorithm:

CS: A Turing machine that stops

Merriam–Webster: A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.

Our Suggestion: An object which can be *run* to perform a task and must stop.

This suggests the following abstract interface:

```
class Algorithm {
public:
  Algorithm() {}
  ~Algorithm() {}

  /** The return value indicates the success
      of the algorithm.
      TRUE: algorithm was run successfully
      FALSE: something went wrong */
  virtual bool run() = 0;
};
```

An important feature of `Algorithm` is that a run might be unsuccessful. While an exception could be thrown for every failure, it would be better to be able to gracefully handle some algorithmic exit conditions and input failures. For example, being given a Vector x not in the domain of a linear operator A makes it impossible to calculate $A * x$. This would cause an exception to be thrown. However, if a Conjugate Gradient algorithm finds a direction of negative curvature (an x for which $x^T Ax < 0$), this is a failure of the inputs (as A is not positive definite) that cannot be detected by compatibility tests in the constructor. Failures like this should be treated as a suggestion to try a more general algorithm, or adjust the parameters of an algorithm. In the example given, the calling algorithm would invoke another iterative solver which can handle indefinite linear operators. This is often how robust optimization algorithms should be built — first try a fast method that only works under some strict assumptions, then if that fails, gradually relax the assumptions until a successful method is found. Another place where this strategy might be useful is with a method which solves to a variable degree of accuracy. If the answer found is unacceptable, fail and try again with a lower tolerance for error. One variant on this is a gridded search. At some point, an algorithm will find the local grid minimum, which means that all neighbors on the grid have higher function values. Thus, the algorithm would fail to find a better solution when next called. This failure is an indication that the grid must be refined if further improvement is desired.

Given the abstract algorithm interface, I can compose new algorithms by piecing together other algorithms:

$$\begin{aligned} Algorithm &\rightarrow ListAlg|LoopAlg \\ ListAlg &\rightarrow AlgorithmAlgorithm \\ LoopAlg &\rightarrow WHILE\ !Terminator\ DO\ Algorithm \end{aligned}$$

ListAlg is an algorithm which contains two algorithms (which might be further lists) and when run executes the algorithms in the list in order. LoopAlg repeatedly runs the inner algorithm until the termination criteria is satisfied. As long as the inner algorithm stops and the terminator will eventually be satisfied, LoopAlg meets the definition of an Algorithm.

When dealing with iterative numerical algorithms, there is an additional component to consider. Numerical algorithms modify some explicit state. For example, many update $x_{k+1} = x_k + \alpha dx_k$. The calling algorithm needs to be able to access the current state x_k outside of a subalgorithm. The caller might also want to set the internal state to a new value to take advantage of information outside of the algorithm (such as another search providing a better result). The subclass `StateAlg` meets this need:

```
template<class T>
class StateAlg: public Algorithm {
public:
  virtual void setState(T & x) = 0;
  virtual T & getState() = 0;
};
```

A StateAlg is simply an algorithm with the addition of access functions to a state, whose type is given by the template parameter T. Notice that StateAlg does not necessarily own a copy of the state as a data member. This is done so that an algorithm may build the state on demand. In many cases, it is cheaper to keep intermediate pieces and only assemble them into an external solution when needed. A simple example of this often occurs in solving a linear system. For efficiency reasons, we may solve a permutation of the system, finding y such that $QAPy = Qb$ instead of finding x such that $Ax = b$. In this case, we store the current guess for y internally, and construct $x = Py$ on demand.

7.1.2 Loop Control

The Merriam–Webster definition of an algorithm mentions that they frequently involve repetition of an operation. When an operation is repeated, there must be some criteria that tells the repetition when to stop, in order to maintain qualification as an algorithm. For example, the following is *not* an algorithm:

```
while(true) {
   //do something
}
```

In fact, the items in many textbooks that claim to be algorithms are in fact *not* algorithms since they start with `for i = 1, 2, ...` as their loop header, thus failing to stop. A `Terminator` is an object that can be queried for a boolean response. For our purposes, we take `true = stop` and `false = continue`. This is so we can build new Terminators using Boolean Algebra.

The abstract interface for `Terminator` is quite simple:

```
class Terminator {
public:
  Terminator() {}
  virtual ~Terminator() {}

  virtual bool query() = 0;
};
```

Skeptical users might say, "Your Terminator interface is practically identical to the Algorithm Interface". This is so. In fact, the crucial difference between what we usually think of as algorithms, and what we think of a stopping criteria is that we expect Terminators *not* to have a publicly accessible state that gets modified when the run method is called. We might make `query()` a `const` method, but this seems too restrictive. The internal state of a Terminator is usually private and inaccessible, so it is unnecessary to make assurances as to whether it might be modified.

Similar to Algorithms, we can build new terminators by combining terminators. There are several `Terminators` which implement the standard logical operations:

AndTerminator Returns stop only if both terminators return stop

OrTerminator Returns stop when either terminator returns stop

NotTerminator Reverses output of terminator

XorTerminator Returns stop when exactly one terminator returns stop

Further, we have implemented a long list of Terminators that might be useful in numerical codes. A few highlights:

CountTerminator Acts like a counter in a for loop. Stop when $count \geq maxcount$.

MinTerminator Useful for watching for a Scalar to fall below a specified tolerance.

IPThresholdTerminator Stop when the inner product of two vectors falls below a tolerance.

NormGradientTerminator Stop when $\|\nabla f(x)\|$ falls below a tolerance.

IterationTableTerminator Combines the function of a CountTerminator with a NormGradientTerminator and adds the functionality of printing a table with the iteration number and current values of $f(x)$ and $\|\nabla f(x)\|$

IOTerminator Prints a message and reads a single character from input. Returns stop if input character is ‘y’ or ‘Y’. Useful for implementing the “eyeball norm” together with an IterationTableTerminator.

Terminators usually take reference data in a constructor, and perform `const` operations on data members to make decisions. Some have side effects, such as incrementing a counter or printing to an output stream. While there are no syntactical restrictions on the behavior of a terminator, we strongly suggest that allowing a terminator to modify external data is a bad idea, as the short–circuit evaluation built in to the boolean operations makes execution of a given terminator unpredictable. As a general guideline, `Algorithms` *should* modify their publicly–accessible state and external data when run, but `Terminators` should *only* modify inaccessible internal state and communicate strictly through the return value of `query()`.

7.2 Abstract Algorithm Design

An abstract algorithm interface is a *Strategy* [15]. It is meant to tie together many related classes which behave different but fulfill the same role. When designing algorithm objects, it is important to approach them differently than functions. While the algorithm still consists of the same basic pieces — a set of inputs and outputs and a piece of code which operates on them — the object nature of algorithms must be considered. Inputs are often taken in the constructor of the algorithm and stored by reference. Outputs can be handled either through reference parameters in the constructor or access methods to internal variables. The `StateAlg` in the previous section is one form of the later approach.

The primary difference when working with algorithm objects is that programmers write *steps*, not entire loops. The constructor for an algorithmic step often contains some initialization code. Typically, this code fills in the intermediate data structures and private data of the algorithm. In the conjugate gradient example discussed earlier, the constructor calls the `restart()` method to initialize $w = A*x$, $r = b-w$, $p = r$, and compute the norm of r. If there is any post–processing which follows the main loop in an algorithm, it either goes into the destructor or into an access method which converts internal representations of the current state into external representations.

Sometimes, we intend to reuse an algorithm repeatedly on different inputs. In this case, providing inputs through the constructor is not always ideal. This is especially true if we intend to pass a specific implementation of an abstract base class to fulfill a role in another algorithm. Consider line searches as an example. The basic interface of a line search is fairly consistent: Given a functional $f(x)$ and a search direction dx, find a step length α so that the trial point $x+\alpha dx$ has a lower function value than the initial point x, thus $f(x+\alpha dx) < f(x)$. There are many different implementations of line searches, including Backtracking [36], Dennis and Schnabel [27], and Moré and Thuente [35], and we would like to be able to use them interchangeably. This motivates me to make an abstract base `LineSearchAlg` which could be used by other algorithms (like a Truncated Newton's Method), allowing the reuse and comparison of the different implementations of line searches without changing the calling algorithm.

The `LineSearchAlg` inherits from `Algorithm` but leaves the `run()` method virtual. It only requires a `Space` in the constructor, as it seems reasonable to limit an instance of a line search to a particular space. There is also an optional `minsteptol` which default to the smallest possible scalar. This tolerance is used as a lower bound on step lengths. The `LineSearchAlg` uses the space to initialize a vector for the starting point x_0 and the starting gradient g_0. It also has two private references to the current dx and functional evaluation $f(x)$, which are initialized to nothing, using the `DynRef` class which behaves like a cross between a reference and a pointer. The base class then has a `set()` method to move the line search to a new starting position:

```
 /** Set up line search problem */
void set(Vector<Scalar> & dx,
         FunctionalEvaluation<Scalar> & fx,
         Scalar step0 = 1.0) {
   try {
      x0.copy(fx.getPoint());
      g0.copy(fx.getGradient());
      dxref.set(dx);
      fxref.set(fx);
      step = step0;
   } catch (RVLException & e) {
      e<<"\ncalled from LineSearchAlg::set\n";
      throw e;
   }
}
```

It copies the starting point and starting gradient, saves references to dx and $f(x)$, and resets the step length.

When run, a line search is expected to modify `fxref.get().getPoint()`, which may only be accessed by calling the base class method `getTrialPoint()` or indirectly through `getFunctionalEvaluation()`. Children are limited in their access in order to prevent them from modifying the `DynRef` objects. A line search is supposed to repeatedly update the trial point until whatever stopping criteria it is using is satisfied. When it stops running, callers may access the results using the same access methods that the child used.

This setup allows us to avoid the need for a factory for line searches. Instead, we pass a single instance of a line search in the proper space to a class which needs one. An example of such is shown in the Limited–Memory BFGS algorithm described in Section 7.6.5.

7.3 Benefits

It requires more up–front effort to use the abstract classes properly than to write monolithic functions which implement the same algorithm. However, taking the initially easy route is shortsighted, as the abstractions provide many benefits.

- Algorithm objects may have persistent state. This makes implementing a single *step* a more reasonable choice, since the intermediate computations frequently involved in a single step may be saved without passing them as parameters.

- It is easy to build composite algorithms using the `ListAlg`, without modifying the component classes. For an example, see the Generalized Pattern Search (GPS) method in subsection 7.6.4 or the extension to the Master–Slave Algorithm in subsection 8.7.1.

- Steps are more re–usable than whole algorithms. Steps can be combined into a composite algorithm or attached to different termination criteria.

- We may use abstraction to define relationships between algorithms and algorithm interfaces. Thus, we could define a general iterative linear solver interface and define a concrete class for the inverse of a linear operator as the pair of a linear operator and a linear solver.
- Inheritance allows reuse of large portions of code. For example, see the Master/Slave code in Chapter 8. The main `run()` methods are implemented exactly once, and children need only fill in the smaller data transfer methods.
- Abstraction lets us write several implementations which solve the same subproblem in different manners, but through the same general interface. Frequently, we can write algorithms which decompose a given problem into subproblems, and use only a general subproblem solver interface. Thus, we can swap subproblem solvers rapidly without modifying the calling algorithm.

7.4 I'll take the high road, and you take the low road

While I've focused on implementing algorithm objects using the RVL classes, the base algorithm interface is not dependent on RVL. The algorithm classes can be used to implement both high and low level algorithms with or without RVL. The original intent was to provide a formal framework for encapsulating the abstract numerical code written at the highest level in RVL using vectors, functionals, and operators. However, I found the basic concepts to be very general. The definition of an algorithm has nothing which restricts it to optimization or linear algebra.

It is entirely feasible to write low–level, array manipulation code in an algorithm object. Doing so could still gain benefits from a consistency of interface and possible reuse through inheritance. For example, it might be useful to write generic server code which would receive packets of information, strip off the first integer, and use it to execute a task from a given list. Both the communications methods and task execution methods can be virtual and implemented in many different fashions. The generic task loop of a server is the same. Such a server might then be subclassed to redirect all tasks to an available machine out of a pool of possible machines, much like the master in a Master/Slave paradigm. The server–side code is isolated from the master–side code, and could use completely different types of communication protocols. A smarter version of the master would queue up requests from the server–side and issue tasks as the slaves requested them. While such a server–master code has not been written, it is entirely possible within the `Algorithm` framework.

7.5 The Conjugate Gradient Method

Consider the well–known Conjugate Gradient method (CG). It is an iterative procedure for approximating the solution of a positive definite linear system $Ax = b$, where positive definite means that $x^T Ax > 0$ for all x . Here's a pseudocode description of the method from Trefethen & Bau's Numerical Linear Algebra, pg 294

$$\begin{aligned}
x_0 = 0,\ \ r_0 &= b,\ p_0 = r_0 \\
for\ n &= 1, 2, 3 \\
\alpha_n &= (r_{n-1}^T r_{n-1})/(p_{n-1}^T A p_{n-1}) \\
x_n &= x_{n-1} + \alpha_n p_{n-1} \\
r_n &= r_{n-1} - \alpha_n A p_{n-1} \\
\beta_n &= (r_n^T r_n)/(r_{n-1}^T r_{n-1}) \\
p_n &= r_n + \beta_n p_{n-1}
\end{aligned}$$

Notice that the code only involves linear combination, inner products, and the application of linear operators. This means that the same abstract `Algorithm` will function on any given `Space`.

We will follow our own recommendation to create a class which implements a single step of CG. First, consider the inputs to CG. It takes a linear operator A and a vector in the range of A for the right hand side b. We allow the user to specify an initial guess, as well as a storage location for the norm of the residual, but can default $x_0 = 0$. Further, CG is clearly a StateAlg with x as the state. So, its constructors are:

```
CGStep( LinearOp<Scalar> & inA, Vector<Scalar> & rhs);
CGStep( Vector<Scalar> & x0, LinearOp<Scalar> & inA,
        Vector<Scalar> & rhs);
CGStep( Vector<Scalar> & x0, LinearOp<Scalar> & inA,
  Vector<Scalar> & rhs, Scalar & rnorm2);
```

Then, the two access functions required for a StateAlg:

```
virtual void setState(Vector<Scalar> & x_);
virtual Vector<Scalar> & getState();
```

Finally, CG can be `run()`

```
bool run();
```

The CG algorithm has several intermediate values which might be useful outside of the algorithm, and provides access functions for these. The residual is $b - Ax$ and the access method returns a reference to the norm of the residual. The curvature is $p^T Ap$, which is positive when A is positive definite.

```
/** get a reference to the norm of the residual */
const Scalar & getResidNorm2() { return rhor; }

/** get a reference to the curvature */
const Scalar & getCurvature() { return curv; }

/** get a reference to the tolerance used on the curvature */
const Scalar & getCurvatureTol() { return rtol; }
```

These are primarily useful in Terminators built later.

To avoid unnecessary computation, the class has several internal data members for storing values between runs of the *CGStep*. Here are all the data members:

```
  Vector<Scalar> xdata; // the state if not provide in the constructor
  Vector<Scalar> & x;   // reference the current iterate
  LinearOp<Scalar> & A; // the linear op we want to invert
  Vector<Scalar> & b;   // the right hand side
  Vector<Scalar> r;     // r = b - A * x
  Vector<Scalar> p;     // the orthogonal part of the residual
  Vector<Scalar> w;     // A * p
  Scalar curv;          // p' * A * p
  Scalar rho;           // r.norm2()
  Scalar & rhor;        // a reference to rho
  bool first;           // true in constructor, false after run
  Scalar tol;           // a tolerance for detecting close to 0
  Scalar rtol;          // tolerance on curv, to prevent overflow.
```

In each constructor, these values are appropriately initialized for the given x_0. Since we also need to re–initialize these values when *setState()* is called, we put the initialization code in a protected function *restart()* and call it in both the constructors and *setState*

```
virtual void restart()
  {
    A.apply(x, w);
    r.linComb(1.0, b, -1.0, w);
    rhor = r.norm2();
    p.copy(r);
    first = true;
  }
```

To finish the implementation, we need to code a single step of CG in the *run()* method.

```
virtual bool run() {
    Scalar alpha, beta;

    A.apply(p,w);
    curv = p.inner(w);

    if (curv > rtol ) {
      alpha = rhor/curv;

      x.linComb(1.0, x, alpha, p);
      r.linComb(1.0, r, -alpha, w);
      beta = rhor;
      rhor = r.norm2();
      rtol = tol*rhor;

      if( beta != 0 )
        beta = rhor/beta;

      p.linComb(1.0, r, beta, p);
    } else {
      if (first)
        x.copy(p);
      return false;
    }
    first = false;
    return true;
  }
```

This closely resembles the pseudocode version of the algorithm. The only major differences are to handle possible floating point problems (e. g. avoiding overflow when dividing by a very small number) and to fail gracefully if a direction of negative curvature is detected when A is not positive definite.

Use of this *CGStep* class is demonstrated in an implementation of a Truncated Newton method. To find a search direction p, *TruncatedNewtonStep* uses *CGStep* to solve the Newton system $\nabla^2 f(x)p = -\nabla f(x)$. Here is the code from the *run()* method of *TruncatedNewtonStep*

```
bool run() {
    Scalar rho, tol, step = 1.0, one = 1.0;
    p_.zero();
    FunctionalEvaluation<Scalar> fe(f_, x_);

    Vector<Scalar> g(f_.getDomain());
    fe.getGradient(g);
    g.negate();

    CGStep<Scalar> cgs(p_, fe.getHessian(), g, rho);

    /* We want tol = min(0.5, sqrt(g.norm())) * g.norm() */
    tol = g.norm();
    if( tol < 0.25 ) {
      tol = sqrt(tol)*tol;
    } else {
      tol = 0.5 * tol;
    }
    tol = tol *tol; // We will compare r.norm2() to tol^2

    MinTerminator<Scalar> t1(rho, tol);
    MinTerminator<Scalar> t2(cgs.getCurvature(),
                             cgs.getCurvatureTol());
    OrTerminator t(t1,t2);

    LoopAlg la(cgs, t);

    la.run();

    /* POSTCONDITION: p_ now contains a descent direction */

    BacktrackingStep<Scalar> bs(step, .5);
    SufficientDecrease<Scalar> sdt(x_, f_, p_, step);

    LoopAlg ls(bs, sdt); /* This will do a backtracking line search */
    ls.run();

    /* POSTCONDITION: step is a valid step length
                      (satisfies sufficient decrease)
    */
    x_.linComb(one, x_, step, p_);

    return true;
  }
```

Notice how we

1. Create a *CGStep*.
2. Build two terminators.
3. Combine them with an *OrTerminator*.
4. Build a *LoopAlg* from the step and the combined terminators.
5. Run the *LoopAlg*.

This sequence of actions is a typical use of a step. A similar sequence of actions performs backtracking along the search direction.

7.6 Other Example Algorithms

Implementation of a variety of optimization algorithms helps to test the `Algorithm` design. In each case, we needed to choose the inputs for the algorithm, the division of the method into smaller algorithms, and the data structures needed in the algorithm.

7.6.1 CGNE

Given a nonsingular matrix $A \in \Re^{m \times n}$ and a vector $b \in \Re^m$, we often want to find $x \in \Re^n$ which minimizes $||Ax - b||^2$. The solution to the linear least–squares problem is also a solution to $A^TAx = A^Tb$. We can use the Conjugate Gradient Normal Equation Error method (CGNE) to iteratively solve this second formulation.

Our implementation follows our usual design. A `CGNEStep` class is a `StateAlg` with a `Vector` state. Each `run()` of this step executes one iteration of CGNE. The constructor for `CGNEStep` takes a `LinearOp` A, and two `Vector` inputs x_0 and b, where x_0 is an initial guess for x. We suggest using $x_0 = 0$ when a better guess is unavailable. The constructor initializes the work vectors and several scalars. Each iteration of the algorithm requires two linear operator applications, three linear combinations, and three inner product calculations.

Since the algorithm usually stops when the norm of the residual $b - Ax_k$ falls below a tolerance, we provide access to a reference of the current norm of the residual. This scalar is needed inside the algorithm, but we can save work by not requiring a terminator to recalculate it. Thus, we recommend using a `MinTerminator` on the norm of the residual, and some sort of counter with a maximum number of iterations set.

7.6.2 ConOptStep

Many of the example algorithms I wrote were continuous optimization algorithms. Once I recognized I was copying much of the code from one to another, it became apparent that a common parent was needed. The `ConOptStep` is a base class for all continuous optimization steps. It has a number of private data members which children may access through methods like `getTrialPoint()`

x0 A copy of the initial starting point for the optimization. Accessed through `getBasePoint()`.

x A copy of the trial point, which is updated as the optimization proceeds. Accessed through `getTrialPoint()`.

fx The functional evaluation at the trial point. Accessed through `getFunctional-Evaluation()`.

scale A scaling factor for the functional. Defaults to the initial functional value. Accessed through `getScale()`.

gradscale A scaling factor for the gradient of the functional. Defaults to the initial norm of the gradient. Accessed through `getGradScale()`.

The class has a `set()` method which takes a vector and a functional evaluation. When this method is used, it switches to referencing these external data values instead of the default internal values. This is done intentionally to allow for coupling between algorithms. We often want to use the same functional evaluation in several places to avoid costly recalculations of a functional value. However, we do not want to force such coupling, as it puts additional burdens on the calling object as well as preventing sometimes useful independence.

The `ConOptStep` is a `StateAlg<Vector<Scalar> >`. It uses the trial point x as its state. When asked for the current state, a constant reference to x is returned. However, due to the internal workings of this class, when a state is set, we copy the input into both x and $x0$ and redirect the references to the internal copy of fx. This essentially resets the algorithm, moving the trial point and base point to the same location.

7.6.3 Line Search

The abstract line search algorithm was already discussed as an example in Section 7.2. The abstract interface is fairly simple, containing the minimum necessities to define a problem — minimize $\phi(\alpha) = f(x + \alpha dx)$. The implementations are significantly more complex, even the straightforward backtracking approach. These line searches get used frequently in other algorithms, such as the Quasi–Newton method in Section 7.6.5, to control the step size once a search direction has been chosen.

There are almost always parameters in such numerical algorithms, and while we often have an idea of good default settings, a good design should allow users these parameters without hacking the source code. There are several ways to do this. One of the simplest is to include all the parameters in the constructor, and use the capability of C++ to define default values for each. However, the parameter list can quickly grow massive and in order to modify a single value at the end of the list, a user must provide values for each parameter before it. The language only allows default values at the end of a parameter list. Another option is to default all values, but provide access methods to set and retrieve them. While effective, this can force a user into making a long sequence of method call to set the parameters, and these settings are not easily saved or copied.

We have found a better solution through the use of a parameter file with key and value pairs. The existing `RVL::Table` will read such pairs from a file or write them to a file. We can subclass the table to implement default values for specific keys. If the table finds a particular key in the file, it uses that value. Otherwise, it will choose the default. For unconstrained minimization, we chose to implement a single `UMinTable` which would hopefully encompass all the parameters typically encountered in this type of optimization. The line search subclasses each take a `Space` and a `UMinTable`, along with an optional `ostream` for output. The space is passed to the base line search class, and the table is queried for pertinent parameter values.

I will not go into the full detail of the line search algorithms. Duplicating the code here would not be helpful, as it is full of sanity checks and output code. After some consideration, we decided not to break these algorithms into steps and terminators. The steps themselves were often trivial, and when not trivial, were too tightly coupled with values used in the terminator. The behavioral changes to functional evaluations (see Section 4.3.1) would have helped to avoid some inefficiencies, but there still seems to be a point at which further code encapsulation is more of a hindrance than a help.

One interesting note — although I do not have the timings to evidence these assertions, we found that in most cases, the more complex line searches, like the Dennis and Schnabel variant [27], did not provide the performance gains we expected. One of the goals in building an abstract base class for line searches was to allow easy swapping of different searches to both compare performance and find the best for a given application. We found it interesting that the backtracking algorithm performed as well or better than the more complex algorithms. We conjecture that part of this behavior can be attributed to it defaulting to a step length of $\alpha = 1.0$ for its initial guess. While a full step is not always sufficient to satisfy the stopping criteria, it often is, meaning that the backtracking algorithm does very little work much of the time.

7.6.4 Generalized Pattern Search

The Generalized Pattern Search (GPS) method is a derivative–free search which uses a grid to guarantee convergence [28] [2]. It permits users to perform any search first, and if that search fails to find simple improvement over the incumbent point, then performs a local search in a pattern. The directions in the pattern must form a positive spanning set of the search space in order for the method to converge to a local optimum.

This algorithm suggests a design of abstract interfaces. Each step in the method is a `LocalSearch`. A `LocalSearch` has an step size, which can be set or retrieved. `LocalSearch` is also a child of `StateAlg`, and the natural choice of state for a derivative free algorithm is an ordered pair of a vector in the search space x and the value of the functional being searched $f(x)$. We call such an ordered pair an `OptPoint`. This allows us to easily compare two `OptPoint` objects without keeping a `FunctionalEvaluation` around or recalculating the function value multiple times. An `OptPoint` is essentially a structure with a constructor and an assignment operator for convenience.

A specialization of `LocalSearch` is a `LocalPoll`. The `LocalPoll` has a pair of pure–virtual methods — `int getNumSearchDir()` which returns the size of the poll set, and `Vector<Scalar> & getSearchDir(int i)` which returns an element of the poll set. It then implements the `run()` method by looping through the poll set until it either finds a better point, or exhausts the poll set.

```
while(!success && (i < n) ) {
  y.linComb(1.0, best.x, Delta, getSearchDir(i));
  FunctionalEvaluation<Scalar> feval(f,y);
  fy = feval.getValue();
  if( fy < best.fx ) {
      success = true;
      best.x.copy(y);
      best.fx = fy;
    }
    i++;
  }
```

A concrete child of `LocalPoll` implements the two pure–virtual methods `int getNumSearchDir()` and `Vector<Scalar> & getSearchDir(int i)`. For example, the `CompassSearch` class takes the search space S as input and has the set of poll directions $\forall e_i, -e_i \in S$, the coordinate directions of the space. Note that such a search is only sensible on spaces with a fixed, finite dimension. Other searches and more complicated spaces might require an update to be performed on the poll between iterations. In particular, the GPS methods for searching in a constrained set would require that the poll directions change when near a boundary.

A `GPSStep` is then a composite object consisting of a `LocalSearch` for the search step and a `LocalPoll` for the poll step. It implements the required `LocalSearch` methods by calling the methods of its children. The `run()` method does

```
 bool run() {
    searchstep.setState(xbest);
    if( searchstep.run() ) { // if search succeeds
      xbest = searchstep.getState(); // update best
      return true;
    } else { // else poll
      pollstep.setState(xbest); // set polling location
      if( pollstep.run() ) { // if poll succeeds
xbest = pollstep.getState(); // update best
return true;
      } else
return false;  // if poll fails, return false
    }
  }
```

This is much like a `CondListAlg`, except that it sets the state of each member before running the algorithm.

A `GPSAlg` is built from a `Functional`, `LocalSearch`, `LocalPoll`, and an initial guess for the `OptPoint`. It builds a `GPSStep` from the search and poll, and terminators from the functional, state, and step size. When the `run()` method is invoked, it checks the terminators then runs the `GPSStep`. If the `GPSStep::run()` returns false, the step size parameter Δ is reduced. Then the terminators are checked again as the loop repeats. The algorithm stops when the `MinTerminator` detects that the step size has fallen below a specified threshold. Note that the initial step size, termination threshold, and fraction by which the step size is reduced are all optional parameters to the `GPSAlg` constructor.

There could be several obvious variations on these classes. The first is a slightly smarter `GPSAlg` which is allowed to increase the step size following successful searches. This has been found in practice to provide faster convergence when the starting location is far from the optimum, and in no way violates the convergence guarantees of the algorithm. Another is the aforementioned variation for linearly constrained optimization. Given a set of linear constraints, the search set needs to include directions tangent to any active constraints (here active means constraints which could be crossed in one step). Also, Audet and Dennis have a new Mesh Adaptive Direct Search method which behaves similar to the GPS method we have implemented, but can handle general constrained optimization problems [3]. Finally, we could replace `GPSAlg` with a multi–start method which owned a number of `GPSStep` objects started at different locations in the search space. This improves the chances of finding the global optimum, and can be easily parallelized since each search is independent.

7.6.5 Quasi–Newton Limited–Memory BFGS

GPS only uses function values, but is known to often require a large number of iterations to converge to a minimum. More information about the shape of the objective function should lead to faster convergence. The Quasi–Newton Limited–Memory BFGS algorithm is a method for unconstrained minimization which uses both function values and gradient information [33]. The Hessian is approximated by a linear operator which is updated at each iteration. This method is useful in situations where the Hessian is either unavailable or expensive to compute at each iteration. The limited–memory version uses less storage than the full BFGS approximation to the Hessian, but with a potential sacrifice in speed of convergence. In practice, it performs well on large problems (i.e. a large number of variables) and can give satisfactory results at a fraction of the storage cost of a full BFGS approximation. This algorithm is often used as a first attempt at optimizing a new unconstrained problem.

The implementation of the LBFGS operator is copied from an earlier version written for RVL. I use the operator as part of a `StateAlg` child which implements the minimization step. The `LBFGSStep` has a single vector as its state, representing the current location in the search space. Each iteration applies the LBFGSOp, which approximates the inverse Hessian, to the negative gradient. This produces a descent direction $d = H * -\nabla f(x)$. We then conduct a line search in this direction from x to find a step. Given a line search ls, LBFGSOp H, and a functional evaluation fx, the pseudocode for the LBFGSStep is:

```
H.apply(fx.getGradient(), dir)
dir.negate()
ls.set(dir, fx)
ls.run()
H.update(ls.getBasePoint(),fx.getPoint(),
         ls.getBaseGradient(), fx.getGradient())
```

This step combined with an appropriate `Terminator` in a `LoopAlg` to produce a minimization algorithm.

The `LBFGSStep` will function correctly with many different line searches. The class takes a `LineSearchAlg` in its constructor, and uses the `set()` method to reposition the line search for a new x and search direction. The line search actually takes care of the update for x as part of the search, which avoids both an extra update of x and a recalculation of the functional value at the new x.

`LBFGSStep` uses an optional parameter file to specify the numerous input parameters. A sensible default value for each parameter is hard coded into the initialization function, so user may set only the parameters for which they desire to use non–default values. This functionality is copied from the earlier implementation of this algorithm, and serves to avoid the need for long lists of inputs to the constructor or numerous class methods for setting parameters. This also allows us to easily modify settings without recompiling the source code.

7.6.6 Sequential Quadratic Programming

The Sequential Quadratic Programming (SQP) algorithm illustrates the idea of constructing a solution from a series of subproblems. It also helps to show that the `Algorithm` interface is not restricted to use in solving unconstrained optimization problems.

Consider the problem

$$\begin{array}{ll} \min & f(x) \\ \text{subject to} & c(x) = 0 \\ & x \in X \\ & f : X \to \Re \\ & c : X \to Y \end{array}$$

where X and Y are Hilbert spaces. This describes a large class of constrained optimization problems. We restrict our attention to equality constrained problems, as the methods for solving them are better understood. Many of the methods for handling inequalities reduce to solving a sequence of equality constrained problems. Of course, this reveals one common structure already — we can often solve problems by instead solving a series of subproblems. It is commonly said that the only problem a computer can solve is the linear system $Ax = b$. One level up from this, Newton's method solves a series of linear systems to find local minima of non–linear functions. Then, using Newton's method repeatedly can find the approximate solution of a PDE. These sorts of nestings are very common in mathematical optimization.

In particular, this nesting is very explicit in Sequential Quadratic Programming methods (SQP). We solve the nonlinear, equality–constrained optimization problem by solving a series of quadratic programming (QP) problems. The SQP method can be outlined as follows:

```
Given $x_0$, $f(x_0)$, $c(x_0)$, $\nabla f(x_0)$, $\nabla c(x_0)$, $H_k \approx \nabla^2 f(x_0)$, $k = 0$

While the termination criteria is not satisfied
    calculate a step direction $\delta x_k$ by solving a QP
    calculate a step length $\alpha_k$ by solving a subproblem
    update $x_{k+1} = x_k + \alpha_k \delta x_k$
    evaluate the functions and gradients at the new $x_{k+1}$
    calculate $H_{k+1}$
    $k \to k + 1$.
end while
```

There is a huge amount of flexibility in how we accomplish many of these steps. This results in a high–level algorithm that is easy to describe in very general terms. It also forces us to ensure that there is enough flexibility in our design. For example, we do not want the interface to be too specific in how the step direction and length are calculated. A bad design would insist on the form of a QP to be solved, while a good design just requires a subalgorithm which produces a step.

Finally, I have ulterior motives in the choice of SQP as a test–algorithm. The local optimization community has many optimization problems where the constraint $c(x) = 0$ is actually a system of partial differential equations. SQP methods may be well suited to solving these problems, and by implementing such methods through my design, I can both test our design and hopefully solve the PDE constrained problems at once.

7.7 Lingering Issues

I've presented a design for an abstract algorithm interface, and several useful concrete children of `Algorithm`. I require stopping criteria for controlling iterative algorithms, and meet this need with the `Terminator` interface along with concrete children for making composite terminators. Finally, as a proof of concept, I describe in detail an implementation of the Conjugate Gradient algorithm for solving linear systems along with the outlines of several implemented optimization algorithms.

There linger some questions about how to use algorithms, especially in the context of optimization, where we often solve a given problem by building a series of subproblems and solving them. A few questions to consider in the future:

1. Who builds the subproblem — the algorithm that needs the subproblem to be solved or the solver?

2. Do algorithms solve problems, or do problems use algorithms to 'get solved'?

3. How do we describe problems, especially in the context of constrained non–linear programming?

4. How do we make algorithms smart enough to choose the right subproblem solver? For example, choosing the right linear solver for a given linear system.

5. Is there a graceful way to maintain the flexibility of templates when we end up interfacing old FORTRAN code so frequently?

6. What general principles should we follow in designing algorithm interfaces for solving a particular class of problems and when implementing particular realizations of these interfaces?

Chapter 8

Master–Slave Parallel Algorithms

Thus far, I've only used the Algorithm framework to implement optimization algorithms. However, the design is quite general and meant to encompass all types of algorithms. Now, I want to illustrate an example which is not connected to optimization. It uses object–orientation to implement the `run()` method of several algorithms around virtual methods which must be filled in by concrete children. This example also illustrates one method for incorporating parallelism into RVL. A completely different method for parallelism will be discussed in Chapter 9.

The Master–Slave approach is a way to parallelize and load balance a large set of independent tasks. The slave processes are given a task by the master, execute the task, then return the results. The Master continues to issue tasks until the supply of tasks is depleted. While this approach is not well suited to all applications, where appropriate it can provide a significant reduction in run–time when compared to a serial implementation of the same application, at the cost of more hardware. This approach works best when the number of tasks is large compared to the number of available slaves and each task involves a large amount of computational work for a small amount of task data.

I will present the basic idea behind this approach, followed by the details of my implementation using the `RVL` and `RVLAlg` interfaces. I will demonstrate its effectiveness with an example application and provide instructions for implementing new master–slave applications.

8.1 Basic Idea

Numerical computations are traditionally parallelized at a low level. Each processor is given a portion of the data for each vector and matrix. The basic linear algebra operations are then parallelized, adding global communication where necessary to exchange information and perform reductions. Frequently, a parallel numerical library is used by algorithm writers to avoid coding parallel operations themselves. This way, many of the headaches of writing parallel code are only suffered once, by an experienced parallel programmer, and algorithm writers can trust that when they do a matrix multiplication all the underlying communication is handled invisibly. There are many examples of parallel numerical libraries, including Epetra [25] and Petsc [14]. These libraries are usually designed to run in a Single–Instruction–Multiple–Data (SIMD) environment.

There are some numerical applications which don't need the tight data coupling provided by such libraries. In these applications, a problem can be subdivided into independent tasks or chunks of data. These tasks can be run in any order, and when completed, a solution to the problem can be synthesized from the results. A simple example of such an application is a brute–force optimization of a functional on a grid G. If the grid has dimensions $d_1, d_2, \ldots, d_n$, then there are $N = \prod_{i=1}^{n} d_i$ different points x_j on the grid. Each functional evaluation $F(x_j)$ is an independent task. A serial implementation would run a loop

```
for j=1:n
   if( F(x_{j}) < best )
      best = F(x_{j});
      bestx = x_{j}
```

```
  end
end
```

However, since each evaluation is independent, the problem can be solved more quickly by partitioning the grid into pieces G_k, and letting a different processor work on each piece. A simple reduction to compare the best solutions on each partition finishes the algorithm.

Such a parallelization works well as long as each partition is the same size, and each processor works at the same rate, and each evaluation $F(x)$ requires the same time. Otherwise, some processors sit idle while waiting for their counterparts to catch up. A simple modification to the procedure ensures that all processors are kept busy as long as work remains to be done. Instead of partitioning the work before starting, all processors pull tasks from the same queue. When one task is finished, the processor retrieves a new one from the queue and continues working. When the queue is empty, all processors do any necessary work to compare results, then stop. This model is known as the *queue of tasks* model. A slight variation on this model is the *Master/Slave model*, where one process acts as the master, assigning tasks to processors and gathering the results. While the queue of tasks model is well suited to a shared memory environment with a small number of processors, the Master/Slave model is more appropriate in a distributed memory environment, where it is extremely difficult to implement a shared task queue.

Here are the pseudocode algorithms for the Master and Slaves:

```
Master while( slaves still working )
           get message from a slave
           if( slave needs new task )
              if( queue is not empty )
                 remove task from queue
                 send task to slave
              else
```

```
          tell slave to quit
        end
      else if( slave has results to send )
        get results from slave
        postprocess results
      end
    end
```

Slave
```
    request new task from master
    while( tasks available )
      get task
      compute task
      contact master about results
      send results
      request new task
    end
```

The Master is responsible for issuing new tasks and gathering and postprocessing results. Slaves repeatedly request new tasks, process the tasks, and transmit results to the Master.

8.2 Abstraction using RVL and RVLAlg

It is possible to divide these algorithms into several smaller steps. The master needs to be able to

1. communicate with the slaves
2. get new tasks
3. detect when there are no more tasks

4. send a task to a slave
5. get results from a slave
6. do something with the results

The slave must know how to

1. communicate with the master
2. receive a new task
3. send results to the master

I can use the infrastructure provided by RVL [41] to standardize the data structures and interfaces. I assume that all tasks involve the application of the same `Operator`. This assumption implies that each task is a `Vector` in the domain of the operator, and each result is a `Vector` in the range. The data transfer functions are defined as pure virtual methods in the abstract base classes, which take a `Vector` reference parameter as input. On the master side, the input also specifies which slave data is being transfered to or from:

```
template<class Scalar>
class DataMaster: public RVLAlg::Algorithm {
protected:
  /** Transmit data to the slave numbered slavenum */
  virtual void sendData( Vector<Scalar> & data, int slavenum) = 0;
  /** Receive result from the slave numbered slavenum */
  virtual void receiveResult( Vector<Scalar> & result,
                              int slavenum) = 0;
  /** Fetch more data from the datasource.
      Returns false when unable to fetch data.
              true when data fetched successfully. */
  virtual bool loadData( Vector<Scalar> & data ) = 0;
  /** Store result to the datasink.  */
  virtual void storeResult( Vector<Scalar> & result) = 0;
...
};

template<class Scalar>
class OpSlave: public RVLAlg::Algorithm {
protected:
  Operator<Scalar> & Op;
  Vector<Scalar> input;
  OperatorEvaluation<Scalar> opeval;
  /*  Get data from the master */
  virtual void fetchData(Vector<Scalar> & input) = 0;
  /* Send results to the master */
  virtual void sendResults(const Vector<Scalar> & output) = 0;
...
};
```

Notice that the master's `loadData()` method returns a boolean value which indicates whether there was data left.

8.3 Isolation of the Communication Layer

These methods allow master and slave to transfer data to each other and give the master a source for data and a sink for results. I still need some way to send small messages between them. Taking a cue from America Online, we define an abstract `Messenger` class which provides this functionality. The master and slave are given a messenger in their constructors and store a reference to it. To make the messenger reusable in other applications, it is templated on a message type. For this application, I've defined

```
enum MSMessages { NewTask, ResultReady, SlaveError, SendingTask,
  TransmitResults, OutOfTasks, MakeYourMaps,
  ErrorInTransmission };
```

which includes all the messages the master and slave need to send. The messenger is then a templated abstract base class:

```
template<class MsgType>
class Messenger {
public:
  Messenger() {}
  ~Messenger() {}

  /** Send a message to a specific destination.
      Return an integer errorcode. */
  virtual int sendMsg( const MsgType & msg, int Destination) = 0;

  /** Receive a message from a specific source.
      Return an integer errorcode. */
  virtual int recvMsg( MsgType & msg, int Source) = 0;

  /** Receive a message from anyone.  Return the rank of the sender.
      If negative, return value reflects an errorcode. */
  virtual int recvMsg( MsgType & msg) = 0;

  /** Return my process rank (or some other unique identifier)
      i.e. the number others would use to send messages to me. */
  virtual int getRank() = 0;
};
```

Sending messages must be directed to a particular destination. Receiving can be general or targeted to a particular sender. The process rank is standardized here, as the ranks are needed to identify destinations. Rank is admittedly an MPI biased word choice, but any unique identification number would suffice. One implementation of this messenger uses MPI and the `MPI_Send` and `MPI_Recv` functions.

8.4 A Variation

While I generally prefer to compose algorithms using the high–level abstract objects in RVL, there are times when the high–level classes can be overkill. The slave algorithm only uses some of the capabilities of the given `Operator`, namely the forward `apply` method and the domain and range. In cases where the operation to be performed by the slave does not require `Space` information, a simplified slave with lesser requirements would suffice. This simplified slave provides the flexibility to use the slaves to perform a low–level task without trying to promote the data and the task to an artificially high level of abstraction. It is quite possible that the data defining a task does not lie in a Hilbert Space and has no reasonable implementation of a norm or inner product. While we could pretend that the data was in Hilbert Space and simply avoid using the unimplemented or incorrect methods, this is a bad design practice. Instead, it is better to implement a new slave algorithm which handles these types of data.

```
template<class Scalar>
class UFOSlave: public RVLAlg::Algorithm {
protected:
  UnaryFunctionObject<Scalar> UFO;
  DataContainer<Scalar> * input;
  Messenger<MSMessages> & m;
  int verbosity;

  UFOSlave();

  virtual void fetchData(DataContainer<Scalar> & input) = 0;
  virtual void sendResults(DataContainer<Scalar> & output) = 0;

public:
  UFOSlave(UnaryFunctionObject<Scalar> & _UFO,
   DataContainerFactory<Scalar> & dcf,
   Messenger<MSMessages> & _m,
   int verb = 0)
    :UFO(_UFO), input(dcf.build()),
     m(_m), verbosity(verb)
  {}
.
.
.
};
```

The `UFOSlave` takes a `UnaryFunctionObject`, a `DataContainerFactory`, and a `Messenger`. It instantiates a `DataContainer` using the factory, fetches data from the master into the DC, then applies the UFO to the DC. The UFO is expected to overwrite the input data with output data and the results are transmitted to the master. As the `DataContainer` can be quite complex, this procedure is sufficient for all cases, although it may require some ingenuity in attaching auxiliary data to the data container. However, it does save us from writing operators with useless adjoint and derivative methods, which was irksome. If it became necessary, it would be an extremely minor modification to create a slave for binary, ternary, or quaternary function objects. However, for these cases, it would have to be standardized which parameter is input and which is output, which reduces the generality of the algorithm. Although the RVL designers often write function objects with the output first and assume all other parameters are input, this is a purely local convention. Recognition of this fact led to the addition of the `readsData()` and `writesData()` methods in a function object, and use of these methods would allow some logic to be added to a slave with multiple data containers.

```
for(int i = 0; i < numParams; i++)
   if(FO.readsData(i))
     fetchData(d[i]);
d[0].eval(FO, d[1], ...);
for(int i = 0; i < numParams; i++)
   if(FO.writesData(i))
     sendResults(d[i]);
```

This requires that the function object is truthful about which data containers are read and written, but lacking a messy *const* scheme, is as much of a guarantee as I may get.

The `OpSlave` and `UFOSlave` give application implementors a choice of whether to invoke task–level parallelism at a high level (operator) or a low level (function object). This broadens the usefulness of the algorithms and reinforces the beauty of the design of the `DataMaster` algorithm. The same master algorithm can satisfy different slaves, because the master's job is simply to issue tasks and store results. Further, there is nothing preventing the same master and slave algorithms from being run in a shared–memory environment. Data transfer becomes a matter of passing pointers and messaging could be as simple as flipping integers in an pair of arrays, resembling a postal system — Put the message in one array, and flip a flag in the other array to indicate a new message.

8.5 Epetra Implementation

A fully implemented master needs a data source and sink and a pair of methods for sending and receiving data from slaves. The data source and sink are methods which take a vector of data as a parameter. The source fills the vector with data, and the sink often writes the results out to a file. In some applications, there is pre– and post–processing of the data, and it occurs in these methods. Typically, these methods are implemented using a `UnaryFunctionObject`.

The methods for sending and receiving data are mirrored on the slave, and are the only virtual methods in the slave class. The implementation of these depends on the type of data being transmitted and the method of transmission. If the data is `Streamable`, the socket code in the *Remote* package [13] would suffice. Applications on parallel clusters of machines could use MPI directly to transfer data. I chose to use MPI indirectly by adapting the Epetra [24] library to RVL, then using the methods of the `Epetra_MultiVector` to transfer data. This adaptation, discussed in Section 6.3, has several benefits:

- Prove the adaptability between RVL and a parallel library, Epetra;
- Avoid rewriting calls to nonblocking MPI functions;
- Gain the parallel linear algebra resources in Epetra;
- Prove that we could use a parallel library designed for Single–Instruction–Multiple–Data (SIMD) use in a Multiple–Instruction–Multiple–Data (MIMD) environment;
- Access the parallel solvers which use Epetra, such as AztecOO [26].

Some drawbacks to this approach became apparent while writing the application:

- Epetra is strictly double–precision, and the templated Tpetra library is still under construction.

- Global parallel communication occurs when building an `Epetra_BlockMap` This means such map construction must be carefully synchronized between processes.
- The map classes use a 0 length as a special flag. This unfortunately eliminates the case where we actually want a map with n elements on one processor and 0 elements on another. Tricks are needed to work around this.

The second drawback requires some care to avoid causing lockups. Notice that the spaces in the slave are defined by the `Operator`. Since the maps are an integral part of the space, the map creation will necessarily precede construction of the slave class, so we cannot hide map construction code in the slave. Map construction may be performed in the master constructor and this is a sensible place for it, as the master requires a set of maps for each slave. An `EpetraSlaveMapCase` class was written to help standardize map creation on the slave. This class was written in conjunction with the constructor for the `EpetraDataMaster` so that maps are created synchronously. Thus, the main program for a Master/Slave application using the Epetra classes looks like

```
Build the Comms between the master and each slave.
if( rank == 0) {// I'm the master
   Build source and sink UFOs
   Build the StdEpetraDataMaster from the UFOs
   master.run();
} else { // I'm a slave
   Build an EpetraSlaveMapCase
   Build an operator using the mapcase.getLocalSpace()
   Build the EpetraOpSlave from the op and data from the mapcase
   slave.run();
}
```

These Epetra Master and Slave classes are reusable on a variety of operators and data sources. It is not crucial that the operator use Epetra, as it is straightforward to wrap an existing operator with a new one which can adapt between the Epetra data types and whatever data type the operator requires.

In situations where using Epetra is impossible or undesirable, I've also implemented a basic, MPI–based version of the Master and Slave algorithms. This involved a pair of data transfer functions objects `MPIDataSend` and `MPIDataRecv`. Both are unary function objects which are used by both the Master and Slave. A bit of template trickery allows me to have a single, templated implementation of both FOs which call a templated function `findMPIDatatype<Scalar>()`. Such a design is far superior to using template specialization on the FOs, as all translation behavior is isolated in the one templated function which can be reused in many MPI applications. At the moment, the FOs both use blocking MPI calls and can only handle the built–in datatypes (int, float, ect...).

Regardless of the particular operator, data transfer functions, and data source and sink, the abstract algorithmic code in the Master and Slave is completely reusable. This code only depends on the abstract RVL interfaces and the abstract `Messenger` interface.

8.6 Example Applications

Some applications are particularly well suited to a Master/Slave framework. These typically have a large number of independent tasks with each task running a variation of the input data. Often, each task takes a large number of cycles and might be difficult to parallelize. Further, the tasks might vary in the number of cycles each requires, which would make a static load balance difficult to compute. The Master/Slave framework is automatically load balanced, with each node sitting idle for no longer than the single longest task — which is a worst case when all nodes finish at once with only one task remaining in the queue. There is no communication between slaves, which implies no need for barriers or other attempts at global parallel synchronization.

The main difficulty in an application lies in deciding what data is necessary to define a task, and packing this data into a `Vector`. Often, we need to attach a piece of auxiliary data to the `Vector` and ensure it gets transfered along with the vector. Examples of auxiliary data are grid specifications, time information, source information, or instructions on the desired output. To aid in the attachment and movement of auxiliary data, the `AuxComboLDC` and `AuxComboDC` were created in the *RVLTools* package. Epetra implementations two interfaces are available as part of *epetraATN*.

The *TSOpt* package [44] performs the time–stepping operations needed to solve a time dependent system of PDEs. In the end, a `TSOp` operator is created which uses samplers to map from the external domain and range to internal storage. I take advantage of the samplers to connect the adapted *Epetra* data classes to other local storage classes, which allows reuse of serial TSOpt code in a Master/Slave environment just by modifying the samplers and the Model slightly. Each task then consists of varying the input to the `TSOp` and recording the resulting output.

A slightly more complicated example uses the seismic simulation code in the *fd* package. Here each task is defined by a parameter file. The tasks require roughly an hour apiece on a workstation. The master runs through a list of parameter files as the data source, while each slave reads a parameter file and runs the simulation specified.

8.7 How to implement a Master–Slave Algorithm

There are only a few tasks which need to be done in order to create a new Master/Slave application.

1. Build a `UnaryFunctionObjectScalarRedn<Scalar,bool>` for the data source.

The `operator()` method of this class must pack either a LDC or an `AuxCombo-LDC` with all of the data to be transmitted to the slave. This function should set the return value to `false` when there is no more data to be sent.

2. Build a `UnaryFunctionObject<Scalar>` for the data sink. This function will be given results returned by the slaves. What is done with these results depends on the application. Note that due to the nature of the algorithm, results will not necessarily be received in the order the problems were issued. Thus, it may be beneficial to tag each input and output so that they may be correctly interpreted.

3. Build either a `UnaryFunctionObject` or an `Operator` which will be run on the slaves. These will be given data in the exact format which it left the data source UFO. In the case of the `UnaryFunctionObject`, output gets packed back into the same buffer, which gets shipped back to the master. In the case of the `Operator`, output goes into a different vector, which is shipped back to the master.

4. Select an existing data transfer method, or write a new one tailored to the application.

8.7.1 Building a Master–Slave configuration into an event loop

There are times when the task–level parallelism suitable for a Master–Slave algorithm arises as a subproblem of a larger application. In this case, the `DataMaster` may be run several times with a different set of tasks each time. However, the `OpSlave` will stop running when it is told there are no more tasks left in the queue. At first, it was tempting to hack the master to not send out the “No More Tasks” message until the Master is destroyed. However, the master would then need to keep track of whether or not each slave has finished the current task in order to stop a `run()`. This approach is unnecessarily complicated. Instead, I should take advantage of the `Algorithm` abstraction and use the `DataMaster` and `OpSlave` as parts of an outer algorithm.

I put the `OpSlave` inside a `LoopAlg` so that it may run more than once. Each run of the `OpSlave` will continue until a set of tasks is finished. I then need a `Terminator` to control the outer loop and stop computation eventually. I reuse the capabilities of the `Messenger` to control the terminator. The terminator stops when it receives a “stop” message, and continues to loop for any other message. I call this a `ListeningTerminator` since it simply listens until it gets a message and then acts on that message. It is crucial that the `ListeningTerminator` use a blocking receive call when it tries to get a message. Otherwise, the slave might begin requesting tasks when the master isn’t running.

On the master side, I use a `ListAlg` to combine the `DataMaster` with a `Broadcast-Alg` algorithm to send the appropriate message to the terminator on the slave side. The `BroadcastAlg` has a very simple behavior. Every time it is run, it transmits a “wake up” message. When it is destroyed, the algorithm destructor transmits the “stop” message. Thus, death of the master causes the slaves to stop running.

The combination of the `ListAlg` with a `BroadcastAlg` and a `DataMaster` sets up the event loop on the master side, and the `LoopAlg` with a `ListeningTerminator` and an `OpSlave` causes the slaves to wait around until woken, then service a set of tasks and go back to sleep.

Chapter 9

Parallel Functional with Implicit FE Constraint

Truly coordinate–free algorithmic code should function correctly in either serial or parallel, and should not require *any* modification. This is possible only because the ANA does not refer explicitly to the details of memory layout or computational implementation. The abstract calculus interfaces provide a layer of indirection between the ANA and the implemented parallel linear algebra libraries.

As proof that the RVL/ALG design produces reusable ANAs, I took a Quasi–Newton algorithm with a LBFGS approximation to the Hessian (the ANA) and applied it to a functional which implements an implicitly constrained advection–diffusion optimal control problem in parallel (the NLP). Prior to this, the ANA had only been used on serial functionals, but it solves the parallel NLP without any modification to the algorithmic code.

Further, I avoid redoing much of the intricate work of writing parallel linear algebra structures by borrowing from other authors and adapting their interfaces to match RVL in order to reuse their implementations. Epetra [25] is a functioning, stable implementation of parallel linear algebra written by professionals who have spent years working on it. It seems only sensible to use such a code rather than try to replicate its facilities from scratch. Further, Epetra has been used by a community of developers already and there are many tools available which utilize its interfaces, like the AztecOO [26] package of parallel linear solvers and preconditioners, which I use to solve the implicit linear constraint.

I will first describe the physics of the advection–diffusion problem and how the continuous constrained problem is discretized and modified into a unconstrained problem. I'll then describe the abstract optimization algorithm I used to solve the unconstrained problem. A `RVL::Functional` serves as the interface between the ANA and the low–level simulation code. After describing this functional, I'll explain how Epetra and AztecOO were used to implement the functional, including the finite–element code constructing the objects describing the discretized system.

9.1 Advection–Diffusion Problem

There are many problems involving the transport of some substance through a fluid media. When the motion of the fluid (advection) dominates the passive diffusion of the substance, such transport can be modeled by the steady–state advection–diffusion equations

$$\begin{aligned} -\epsilon\Delta y(x) + \mathbf{c}(x)\cdot\nabla y(x) + r(x)y(x) &= f(x) + u(x), & x \in \Omega \\ y(x) &= d(x), & x \in \partial\Omega_d \\ \epsilon\tfrac{\partial}{\partial \mathbf{n}}y(x) &= g(x), & x \in \partial\Omega_n \end{aligned}$$

where $\partial\Omega_d \cap \partial\Omega_n = \emptyset$, $\partial\Omega_d \cup \partial\Omega_n = \partial\Omega$, $\mathbf{c}$, d, f, g, r are given operators, $\epsilon < 0$ is a given scalar, and $\mathbf{n}$ denotes the outward unit normal [7]. Here, x is a position in space and $y(x)$ is the concentration of the substance of interest at a given position. The function $\mathbf{c}$ is the velocity field of the underlying fluid, so $\mathbf{c}(x)\cdot\nabla y(x)$ is the advection term. A scaling factor ϵ controls the strength of the diffusion, and the boundary terms restrict either the concentration or the flux at positions on the boundary of the domain. For further discussion of this problem, see Section 8 in [40] and Section 9 in

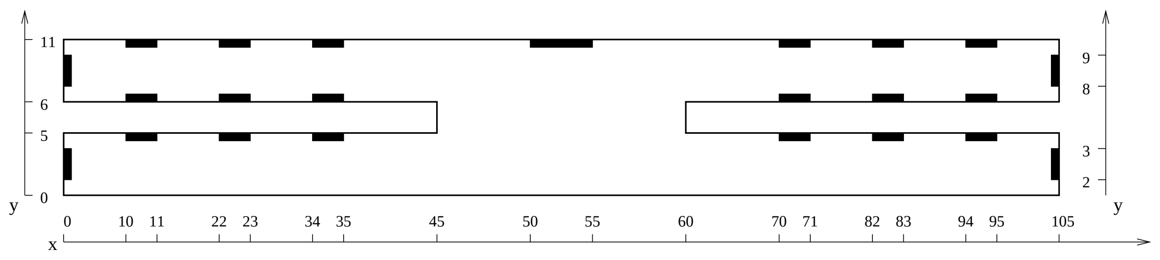

Figure 9.1: Airport model obtained from Sandia National Labs.

[30].

For this demonstration, I chose to use the airport model, shown in Figure 9.1, as the problem domain. The boundary set $\partial\Omega_n$ represents vents in the climate–control system of the airport and I impose a 0 concentration on the rest of the boundary $\partial\Omega_d$. The diffusion term is set $\epsilon = 1e - 4$.

In order to solve an inverse problem to determine an arbitrary source given point-wise measurements of concentrations, I want to minimize

$$\min_{y,u} \frac{1}{2} \int_\Omega (y(x) - \hat{y}(x))^2 dx + \frac{\alpha}{2} \int_\Omega u^2(x) dx$$

for some given data $\hat{y}(x)$, subject to the steady–state advection–diffusion constraint 9.1. Using the implicit–function theorem [36], I can restate this quadratic program as an unconstrained optimization problem

$$\min_{u} \frac{1}{2} \int_\Omega (y(u;x) - \hat{y}(x))^2 dx + \frac{\alpha}{2} \int_\Omega u^2(x) dx$$

where $y(u;x)$ is the solution of 9.1 for a value of u.

I use a finite–element discretization to approximate the solution of this continuous problem. The discretized advection–diffusion constrained problem boils down to

$$\begin{array}{rl} \min_{\mathbf{y},\mathbf{u}} & \frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^T Q (\mathbf{y} - \hat{\mathbf{y}}) + \frac{\alpha}{2}\mathbf{u}^T R \mathbf{u} \\ \text{s.t.} & A\mathbf{y} - B\mathbf{u} - b = 0 \end{array}$$

where

$$\begin{aligned}
(A)_{ij} &= \epsilon \int_\Omega \nabla\phi_j(x)\cdot\nabla\phi_i(x)dx + \int_\Omega \mathbf{c}(x)\cdot\nabla\phi_j(x)\phi_i(x)dx + \int_\Omega r(x)\phi_i(x)\phi_j(x)dx, \\
\mathbf{b}_j &= \int_{\partial\Omega_n} g(x)\phi_j(x)dx + \int_\Omega f(x)\phi_j(x)dx, \\
(B)_{ij} &= \int_\Omega \mu_j(x)\phi_i(x)dx, \\
(Q)_{ij} &= \int_\Omega \phi_j(x)\phi_i(x)dx
\end{aligned}$$

where the linear nodal basis for the state space is $\{\phi_1, \ldots, \phi_M\}$, M nodes in the state space, and the linear nodal basis for the control space is $\{\mu_1, \ldots, \mu_N\}$, N nodes in the control space. R is the discretization of either the L^2 Tikhonov expression or the H^1–seminorm Tikhonov expression

$$(R)_{ij} = \int_\Omega \mu_j(x)\mu_i(x)dx \text{ or } (R)_{ij} = \int_\Omega \nabla\mu_j(x)\cdot\nabla\mu_i(x)dx$$

a regularization term.

Assuming A is nonsingular, so there is a unique feasible point $\mathbf{y}(\mathbf{u})$ for every $\mathbf{u}$, the implicit function theorem let me reformulate the problem as minimizing the functional

$$F(\mathbf{u}) = \frac{1}{2}(A^{-1}(B\mathbf{u}+\mathbf{b}) - \hat{\mathbf{y}})^T Q (A^{-1}(B\mathbf{u}+\mathbf{b}) - \hat{\mathbf{y}}) + \frac{\alpha}{2}\mathbf{u}^T R\mathbf{u}$$

The discretized gradient is then

$$\nabla_{\mathbf{u}} F(\mathbf{u}) = B^T A^{-T} Q A^{-1}(B\mathbf{u}+\mathbf{b}) + \alpha R\mathbf{u}$$

The computation of the discretized functional and gradient only involves the application of linear operators, linear–system solves, and some vector algebra.

9.2 Optimization Algorithm and Functional

To solve the source–inversion problem, I chose to use the Quasi–Newton method with a Limited–Memory BFGS approximation to the Hessian of the functional and a backtracking line search, as described in Section 7.6.5. My implementation of the abstract algorithm, named `UMinLBFGS_BT` takes a `RVL::Functional` F, an initial guess `Vector` u_0, and a `UMinTable`, which is a convenient repository for parameters of unconstrained minimization algorithms along with reasonable default values of these parameters. This algorithm is a *mediator* between the backtracking line search and a `UMinLBFGS` class. The `UMinLBFGS` builds a `LBFGSStep` using the `LineSearchAlg`, functional evaluation, and parameters from the table. This step is given to a `LoopAlg`, composed with a `CountingIterationTable` as the terminator. Both `UMinLBFGS` and `UMinLBFGS_BT` are simply convenience classes which reduce the number of choices a user must make when constructing the solution method from `Algorithm` and `Terminator` objects.

The algorithmic work is done inside the ANA `LBFGSStep`, which is a concrete implementation of the abstract `NewtonStep` interface. The `run()` method of the `NewtonStep` is

```
Vector<Scalar> dir(f.getDomain(), true);
bool cd = calcDir(dir);
bool cs = calcStep(dir);
if (cd && cs) {
  bool upd = update();
  return upd;
}
else {
  return false;
}
```

The `LBFGSStep` implements `calcDir()` and `calcStep()` methods using the `LBFGSOp` approximation to the Hessian and the given line search. The `update()` method calls `LBFGSOp::update()` on the appropriate vectors.

While this design is deeply nested to promote reuse, at the top level, the `main()` driver is quite simple:

```
AdvDiffFunctional f(Comm, argv[1], dalpha); // Build the functional

Vector<T> u(f.getDomain()); // Build the initial guess
RVL::RVLAssignConst<double> init(0.0);
u.eval(init);

string jname="job.bfgs";
RVLUmin::UMinTable<double> tab(jname);
RVLUmin::UMinLBFGS_BT<T> alg(f,x,tab); // Create algorithm

alg.run(); // run the algorithm
```

It builds the functional, then the starting point and the parameter table. These are used to construct the optimization algorithm, which is then run. After the run, u contains the solution.

9.3 Building a Functional

I created the `AdvDiffFunctional` which implements $F(y(u), u)$ and its gradient as described in Section 9.1. This implementation follows the standard pattern I see among concrete `Functional` and `Operator` classes in `RVL`. I wrote two `FunctionObjects` to compute the functional value and gradient. (thus implementing the simulation). They share a reference to a common `AdvDiffProblemHolder`, which builds and holds the various finite–element matrices and vectors involved in the discretized problem (A,B,R,and b). The `AdvDiffFunctional` owns the `AdvDiffProblemHolder` and builds the `FunctionObjects` as they are needed. The functional also stores the computed value of the state $y(u)$ to avoid recomputing it.

This design leaves everything well encapsulated. The calls to the finite–element construction code are all isolated inside the constructor for the `AdvDiffProblem-`

`Holder`. The computations of the functional value and gradient are hidden inside the function objects. The `AdvDiffFunctional` is unaware of parallelism entirely except for passing the necessary `Epetra_Comm` to the problem holder. Thus, to change the matrices or vectors for the finite–element code, I only change the low–level matrix creation code and not the RVL objects. Further, the distribution of data for both y and u among processes is completely controlled by the finite–element code and the mesh files. The functional class is well suited for any implicitly constrained problem, and could be reused with different function objects or a different problem holder. In the future, the functional could be further generalized to satisfy a variety of quadratic objective functions with linear constraints.

All the parallelism is isolated in the `AdvDiffProblemHolder` and the two FOs which implement the simulation. Even these objects only deal indirectly with parallelism through the Epetra linear algebra interface and library (discussed in Section 5.5) and the AztecOO package of linear system solvers (detailed in Section 9.5).

9.4 Denis Ridzal's Trilcode in the Problem Holder

I needed some code to fill in the Epetra matrices and vectors which define the discretized system, namely A, B, and b from the constraint and R from the objective function. Building these objects requires careful work to assemble all the interactions of the finite–elements. This work had already been done by Denis Ridzal in his Trilcode, during a summer visit to Sandia National Labs [7]. The availability of this code and accompanying relatively large problems (0.8 million nodes and 1.6 million elements) were the primary factors motiving the choice of this application as a demonstration of the parallel capabilities of RVL.

I created an `AdvDiffProblemHolder` class to interface with the Trilcode. Apart from the calls to Trilcode, the problem holder class is fairly general. It is simply a structure which contains references to the three Epetra matrices and one Epetra vector, along with an adapter `EpetraMultiVectorSpace` which must be built using map information from the matrices. All the work of the problem holder class is performed in the constructor. It was slightly tempting to define an abstract problem interface and allow the current object to be a concrete implementation, but as there are currently no methods apart from the constructor and destructor, this seemed like overkill.

The Trilcode is separated into several procedures. A `meshreader` function fetches mesh information from a file. This mesh is then given to the assembly function, which fills in the data structures. I had to modify the assembly function slightly, as the C++ implementation didn't provide all the functionality of the MATLAB version (I needed the matrix R). Depending on whether I want to use L^2 type or H^1–seminorm type regularization, R is filled with products of basis functions or dot–products of gradients of basis functions.

The assembly method works by repeatedly calling a local assembly method for each element i. The local method is given the vertices which correspond to that element and calculates the contribution of that element to each of the matrices and vector. These contributions are then added into the correct portion of the data structure. Adjustments are later made for boundary conditions.

9.5 Implement Function Objects using AztecOO

Aztec is a linear solver framework containing a variety of parallel iterative linear solvers and preconditioners [45]. For linear solvers, it offers conjugate gradient, conjugate gradient squared, transpose–free quasi–minimal residual, bi–conjugate gradient with stabilization, and restarted generalized minimal residual methods. The preconditioners include a k–step Jacobi–Neumann series polynomial, a k–step using symmetric Gauss-Seidel, and several variations of non–overlapping domain decomposition preconditioners. The package is aimed at solving large, sparse linear systems in parallel.

The inefficiencies of direct solution methods in both operations and memory use are substantial when solving such systems, making the iterative solvers in Aztec a better choice.

AztecOO is a package of object oriented interfaces to the Aztec solver library [26]. It was designed to work with Epetra data objects as part of the Trilinos Solver Framework. Apart from the new interface, AztecOO adds more sophisticated stopping criteria and a larger choice of preconditioners to the capabilities of Aztec.

The package has one main class named `AztecOO`, which acts as a *mediator* between all of the various components of the package. A problem defined with Epetra objects may be solved using any of the solution methods and preconditioners from Aztec. To solve other problems, the `AztecOO` class allows users to overwrite most of its functionality to permit the use of other preconditioners and solution methods.

Although it is not critical to this application, AztecOO is implemented using only the abstract base interfaces for Epetra. Thus, any linear algebra library which could be adapted to Epetra could be used with AztecOO. For the current project, I use concrete Epetra vectors, adapting them to RVL in order to implement an `RVL::DataContainer` (see Section 6.3).

Inside the function objects used to construct the `AdvDiffFunctional`, AztecOO solves the linear systems which define the implicit advection–diffusion constraint. It is used once to calculate $y(u) = A^{-1}(Bu+b)$ as part of the objective function evaluation. Another solve is needed to calculate $A^{-T}y(u)$ in the gradient calculation.

Using AztecOO is easy. I define an `Epetra_LinearProblem` called $Problem$ from a matrix A, right–hand side vector b, and a solution vector x. Then I create a AztecOO solver, set some of its options, and tell it to iterate:

```
AztecOO Solver(Problem);
Solver.SetAztecOption( AZ_output, AZ_warnings);
Solver.SetAztecOption( AZ_solver, AZ_gmres );
Solver.SetAztecOption( AZ_precond, AZ_Jacobi );
```

```
int res;
res = Solver.Iterate(1000,1e-12);
```

Notice the similarity in design of the `AztecOO` solver algorithm to many of the algorithms in Chapter 7. The algorithm object is created from the minimal required pieces, which themselves are encapsulated. The parameters for the algorithm are set through access methods to achieve the desired behavior. Finally, the algorithm is run, and returns an error code to indicate an algorithmic failure. Here, the stopping criteria is a combination of an iteration count of 1000 and a desired solution tolerance on $\|r\|_2/\|r_0\|_2$, where $r = b - Ax$. However, after delving into source code, this is where the resemblance to my algorithm package ceases, as the `Iterate()` method simply wraps a call to the Aztec procedure

```
AZ_iterate(x_, b_, options_, params_, status_, proc_config_,
           Amat_, Prec_, Scaling_);
```

which performs the actual solve using the given vectors, matrices, and parameters.

9.6 Results

Optimizing the functional on a small, square domain using a four–processor cluster seemed insufficient to fully demonstrate the effectiveness of the design. At that size, it took more time to perform the file I/O than the calculations. I needed a larger problem and more hardware to produce satisfactory 'proof'.

Denis was kind enough to generate a large example using the airport model from Sandia National Labs, shown in Figure 9.1. This model was obtained by simulating air velocities in a two story airport. The two–dimensional problem is roughly 'H' shaped, with a large open well connecting the two floors. There are several air vents, some blowing air into the building and some venting. The velocity data is combined with a finite–element discretization of the building and parameters in the equations are chosen to roughly model the flow of air.

The following results are for a finite–element mesh involving 1654965 elements and 832510 nodes. The mesh is partitioned into 16, 32, and 64 subdomains, and the mesh files are roughly 1 Gigabyte of data total. Results were obtained from the Rice Terascale Cluster, a cluster of over 272 Intel Itanium 2, 64 bit processors, each running at roughly 900 MHz. I chose to use the Myrinet interconnect (Gigabit Ethernet is also available) due to its low latency.

I ran 27 different runs of the code to test all combinations of $16, 32, 64$ processors, $10^{-2}, 10^{-4}, 10^{-8}$ tolerances, and three different preconditioners. The tolerances are given to AztecOO, which will stop GMRES when $\|r\|/\|r_0\| < tol$ or the chosen 1000 iterations have been run. Occasionally, the 10^{-8} tolerance could not be satisfied in 1000 iterations. The three preconditioners were options provided by AztecOO and these descriptions are paraphrased from the user's guide [26]:

- k–step Jacobi, where I used the default $k = 3$.
- Neumann series polynomial of order k, again using the default $k = 3$.
- An additive Schwarz preconditioner, tailored to domain decomposition problems. Each processor approximately solves the local subsystem using Saad's ILUT.

The data in Table 9.1 demonstrates that I get pretty close to the ideal speedup of 2 when I double the number of processors, especially when the solver takes enough iterations to overcome the file I/O and matrix setup costs. Twice as many processors solves the same problem in a roughly half the time. This validates that the code is working correctly in parallel, as this sort of behavior is what I expected. Figure 9.2 shows the runtime decreases as the number of processors vary when using the various preconditioners. Notice that I get the linear speedup with the worse preconditioners, but the Neumann preconditioner performs somewhat better. However, the additive Schwarz preconditioner, which is tailored for domain decomposition problems, performs so well that the communication overhead destroys the expected speedup.

Nodes	Solver Tolerance	Preconditioner	Wall Time
16	1e-2	J	0:21:24
32	1e-2	J	0:12:04
64	1e-2	J	0:07:24
16	1e-4	J	0:96:12
32	1e-4	J	0:49:15
64	1e-4	J	0:25:34
16	1e-8	J	6:52:48
32	1e-8	J	3:28:10
64	1e-8	J	1:42:31
16	1e-2	N	0:11:36
32	1e-2	N	0:07:11
64	1e-2	N	0:05:08
16	1e-4	N	0:45:05
32	1e-4	N	0:23:28
64	1e-4	N	0:13:02
16	1e-8	N	3:12:25
32	1e-8	N	1:35:49
64	1e-8	N	0:47:30
16	1e-2	S	0:02:07
32	1e-2	S	0:02:03
64	1e-2	S	0:02:56
16	1e-4	S	0:03:17
32	1e-4	S	0:02:44
64	1e-4	S	0:03:19
16	1e-8	S	0:07:22
32	1e-8	S	0:04:38
64	1e-8	S	0:04:20

Table 9.1: Wall clock runtimes for Advection–diffusion problem

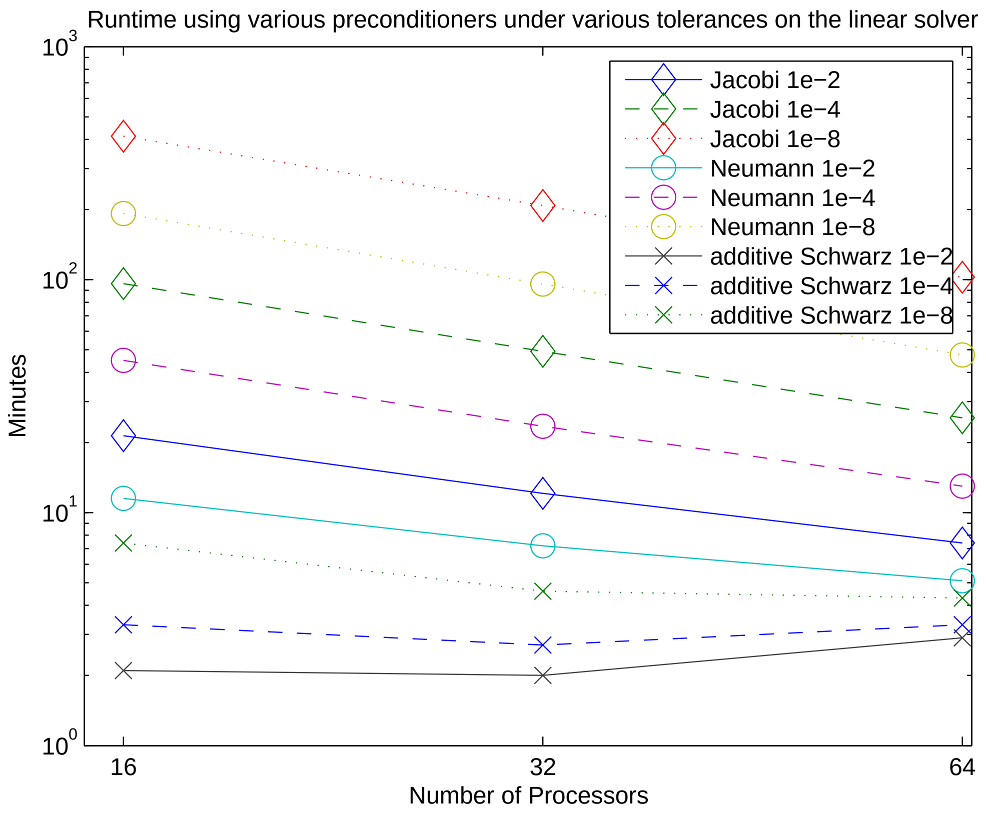

Figure 9.2: Runtimes telling AztecOO to use various preconditioners and requiring GMRES to run to a 10^{-8} tolerance on the residual.

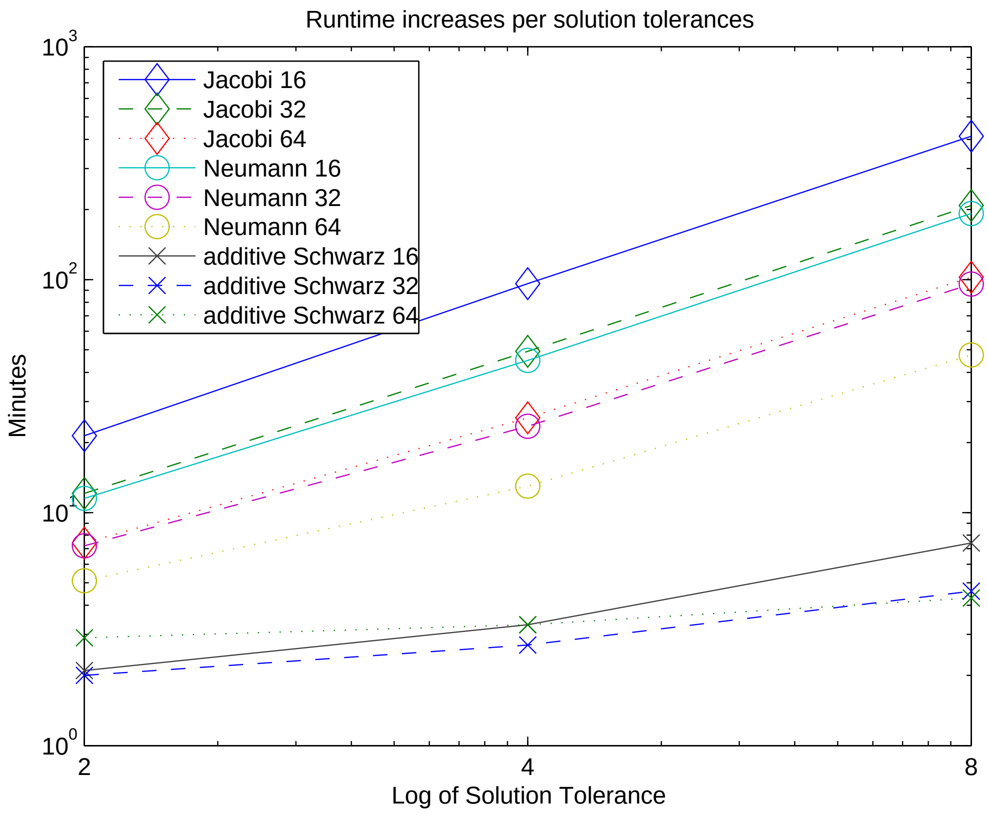

Figure 9.3: Increases in runtime due to tightened residual tolerances required of GMRES

The other Figure 9.3 essentially transposes the data to view the effect of the tighter tolerances on runtimes. Doubling the desired number of digits of accuracy seems to quadruple the workload when using the Jacobi and Neumann preconditioners. The effect is greatly reduced when using the additive Schwarz preconditioner, but it is somewhat tougher to separate the communication overhead and matrix setup costs from the GMRES iterations. Even at 10^{-8}, GMRES takes only roughly 100 iterations to converge when using the additive Schwarz preconditioner.

While these timings are nothing too suprising, they show several things. First, the parallel functional behaves as it should when I vary the preconditioner and tolerance. Second, regardless of the AztecOO settings, the optimization algorithm can use the functional values and gradients it gets to find a local minimum, which in this case must be global since it is a quadratic objective with linear constraints. Third, with a little effort, the RVL package and its extensions can be ported to a large parallel cluster using the Intel compiler, when they were originally designed and tested in serial using the GNU compilers. While this doesn't prove the code is portable, it does increase our hopes, as the Intel compilers are supposed to be close to the C++ standard. All told, these results validate our efforts to combine independently developed abstract numerical algorithms and parallel simulation code into working applications. The adapters are reusable. The functional is reusable for any problems of the form

$$\begin{array}{rl} \min & \frac{1}{2}\|y\|^2 + \frac{\alpha}{2}\langle u, Ru\rangle \\ \text{s.t.} & Ay - Bu - b = 0 \end{array}$$

which are assembled using Epetra. Both the adapters and the functional can serve as guides for interoperating RVL interfaces with simulators.

Chapter 10

Conclusions

The problem domain (NLPs) naturally leads us to implement code objects that mimic the behavior of the conceptual mathematical objects defining the problem — those involved in calculus in Hilbert space. The careful application of design principles to the implementations of such objects results in code which is reusable, composable, and modifiable. The mathematical concepts define object behavior, which is specified in abstract interfaces. The concrete implementations of these interfaces encapsulate the low–level details which come from the particular problem and the programming environment.

By writing algorithm objects to depend only on abstract interfaces to the calculus objects, I have shown that I can exchange the underlying data structures without modifying the algorithms. The exact same LBFGS Quasi–Newton algorithm will optimize functionals on $\Re^n$, grids, and parallel finite–element code. Even better, the underlying implementation of the functional doesn't even have to be written using the same abstract interface, as I can adapt interfaces to each other in order to interoperate code written by different authors in different places, without modifying the base code. I've shown two examples of this, involving RVL, RVLAlg, TSOpt, MOOCHO, TSFCore, Epetra, AztecOO, and other packages. I've discussed adaptation to other interfaces, demonstrating which features aid and which hinder adaptation. As designs improve, adaptation becomes easier and more powerful. It enables authors to write using a comfortable interface with fewer concerns about not being able to use or be used by other packages.

Suddenly, standards become less important. Why mandate a standard interface? Instead, I support an evolutionary struggle between interfaces. Good designs will be used and thrive, and as they do so, will get better. By interacting, good ideas are shared and bad ideas die out. Standards should be replaced every decade or so. Evolving, flexible interfaces continue growing. This is equally true for data structures as it is for algorithms. Spending less time reimplementing old ideas leaves more time for implementing new ones.

Design Lessons

I learned many design and programming lessons while writing this thesis. While the details of each lesson are scattered throughout the thesis, here is a brief summary of the particular lessons relevant to solving simulation driven optimization problems:

- Abstract interfaces allow reuse of code by decoupling interface and implementation.
- Abstract interfaces should be minimalist, and not include *any* method unless every concrete implementation will provide it.
- Specialize an abstract interface to add functionality or identify a subclass — e. g. not every linear operator is invertable. An `inverse()` method belongs in a specialized interface.
- An index method for element access to a data structure can be very inefficient.
- Beware the pervasiveness of parallel details, which are only truly needed in a few places. Communication framework can usually be abstracted to limit dependencies on a particular form of parallelism.

- When adapting data structures, it is possible to efficiently adapt from an exposed pointer to any other data access method. The reverse is not always true.
- Algorithms are objects too.
- Write single algorithm steps instead of entire loops. This permits different stopping criteria to be used with the same step.
- Composites are a good way of reusing code by combining several objects into a new object of the same type.
- Adaptation permits the reuse of tools and implemented code written for one interface with a different interface. This facilitates the creation of applications combining software from many different sources.

Future Work

There is work left to be done. The RVL design only encompasses calculus in Hilbert Spaces. Some people would argue that all real problems involve discrete and categorical data as well as continuous data in Hilbert Space. In order to accommodate such data, abstract parent interfaces to the `Vector` and `Space` classes would be needed which had less required functionality. The low–level `DataContainer` and `FunctionObject` interfaces, with the generalizations for mixed–type operations, are well suited for integer and categorical data without any further modifications.

This extension would permit the implementation of a wider variety of useful algorithms. However, while RVL provides the tools necessary for defining the data objects and operators for nonlinear programs, it does not have any abstractions for the programs themselves. Many other packages have an explicit `Problem` class, containing their idea of an abstract problem. It would be useful to consider what interface can be put on the 'model'. Can a constraint class be built to permit composition of constraints? Even the notion of a constraint is useless without an ordering for vectors.

The ordering introduces further complications of its own. It is also possible that the `Problem` classes are misconceived, and a simpler approach would suffice.

Further 'proof' of the generality of RVL/ALG could be offered. The potential efficiency of implementations in the RVL design is untested and some of this efficiency (or inefficiency) is a compiler issue. However, such efficiency could be easily demonstrated by co-opting the work of other authors through adaptation. Efficiency gains through grouping of parallel reductions and Level 3 BLAS are all at a low–level, and could be encapsulated seamlessly within RVL interfaces.

There are many solution algorithms and data structures still to be implemented in RVL. Denis Ridzal is already at work implementing a SQP method, but there are other linear and nonlinear optimization algorithms which might be incorporated. The hypothetical adaptive grid data structure could also be implemented to demonstrate the arguments against dimensionality in high–level data structures. Since the interfaces are standardized, it might be useful to implement scripts for automating construction of objects like local data containers. Many LDCs only differ in the particular names, types, and amount of meta–data that is encapsulated along with the array data. It would be helpful to have a script that would parse a $(type, name, access)$ list into a working LDC, where *type* is any type, *name* is the chosen variable name, and *access* specifies whether a variable is public, protected, or private as well as read/write accessible.

Apart from algorithms, I have ideas for terminators which would be extremely useful. In particular, while I am certain that such a thing is feasible, I have not implemented a GUI terminator. It would be some work, but with TK/TCL, it is possible to implement a terminator which could display information about functional values, convergence rates, various graphs and plots as well as provide controls for various parameters and solution algorithms. This is an extremely ambitious project, and would require a significant time investment. However, it would be beneficial as both a teaching and research tool.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] Charles Audet and J. E. Dennis Jr. Analysis of generalized pattern searches. Technical Report TR00-07, Rice University, 2000.

[3] Charles Audet and J. E. Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. Technical Report TR04-02, Rice University, 2004. Submitted to SIAM for publication.

[4] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic langauge algol 60. In Peter Naur, editor, *Communications of the ACM*, volume 6, pages 1–17, January 1963.

[5] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, New York, NY, February 1957. Institute of Radio Engineers.

[6] John W. Backus. The history of fortran i, ii, iii. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, New York, NY, 1981.

[7] R. Bartlett, M. Heinkenschloss, D. Ridzal, and B. van Bloemen Waanders. Domain decomposition methods for advection dominated linear–quadratic elliptic optimal controlproblems. Technical report, Sandia National Laboratories, 2005.

[8] Roscoe A. Bartlett. Interfaces extending tsfcore for the development of nonlinear abstract numerical algorithms and interfacing to nonlinear applications. Technical report, Sandia National Laboratories, 2003.

[9] Roscoe A. Bartlett. *MOOCHO : Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide.* Sandia National Labs, 2003.

[10] Roscoe A. Bartlett. A package of light–weight object–oriented abstractions for the development of nonlinear abstract numerical algorithms and interfacing to linear algebra libraries and applications. Technical report, Sandia National Laboratories, 2003.

[11] Roscoe A. Bartlett, Bart G. Van Bloemen Waanders, and Michael A. Heroux. Vector reduction/transformation operators. *ACM Transactions on Mathematical Software*, V(N):1–25, February 2003.

[12] O.J. Dahl and K. Nygaard. *SIMULA 67 Common Base Proposal.* Norwegian Computing Center, Oslo, 1967.

[13] Hala Dajani. Client–server component architecture for scientfic computing. Master's thesis, Rice University, 2003.

[14] Balay et. al. *PETSc Users Manual.* Argonne National Laboratory, August 2003.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object–Oriented Software.* Addison Wesley, 1995.

[16] E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. *ACM Transactions on Mathematical Software*, 29(1):58–81, March 2003.

[17] Mark S. Gockenbach and William W. Symes. An overview of hcl 1.0. *ACM Transactions on Mathematical Software*, I(25):191–212, 1999.

[18] Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[19] Adele Goldberg and David Robson. *Smalltalk–80: The Language and Its Implementation.* Addison–Wesley, Reading, MA, 1983.

[20] Paul Graham. *Hackers and Painters.* O'Reilly, May 2004.

[21] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[22] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., second edition, 1996.

[23] Michael A. Heroux. The trilinos/petra user's guide. Technical report, Sandia National Laboratories, 2001.

[24] Michael A. Heroux. *Epetra Developers Coding Guidelines.* Sandia National Laboratories, December 2003.

[25] Michael A. Heroux. Epetra home page. http://software.sandia.gov/trilinos/-packages/epetra/, 2003.

[26] Michael A. Heroux. *AztecOO User Guide.* Sandia National Laboratories, July 2004.

[27] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations.* Prentice–Hall, 1983.

[28] J. E. Dennis Jr. and Virginia Torczon. Derivative-free pattern search methods for multidisciplinary design problems. In *5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis andOptimization*, pages 922–932. American Institute of Aeronautics and Astronautics, September 1994.

[29] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[30] P. Knabner and L. Angermann. *Numerical Methods for Partial Differential Equations.* Texts in Applied Mathematics, Vol. 44. Springer–Verlag, Berlin, Heidelberg, New York, 2003.

[31] Tamara Kolda, Roger Pawlowski, and Andrew Salinger. Nox and loca home page. http://software.sandia.gov/trilinos/packages/nox/index.html, 2005.

[32] Glenn Krasner, editor. *Smalltalk–80: Bits of History, Words of Advice.* Addison–Wesley, Reading, MA, August 1983.

[33] D. C. LIU and J. NOCEDAL. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989.

[34] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *Transactions on Mathematical Software*, 7(1):17–41, March 1981.

[35] Jorge J. Moré and David J. Thuente. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software*, 20(3):286–307, September 1994.

[36] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer–Verlag New York, Inc., 1999.

[37] Kristen Nygaard and Ole-Johone Dahl. The development of the *simula* language. In *Proceedings of the First ACM SIGPLAN Conference on the History*

of Programming Languages, pages 245–272, New York, NY, January 1978. ACM Press.

[38] Anthony Padula, Shannon D. Scott, and William W. Symes. The standard vector library: A software framework for coupling complex simulation and optimizations. Technical Report TR04-19, Rice University, 2004.

[39] Roldan Pozo. *Template Numerical Toolkit Manual.* National Institute of Standards and Technology, June 2003.

[40] A. Quateroni and A. Valli. *Numerical Approximation of Partial Differential Equations.* Springer, Berlin, Heidelberg, New York, 1994.

[41] Shannon D. Scott and William W. Symes. Towards a standard design for vector classes. Technical Report 11, Rice University, 2001. Trip Annual Report.

[42] Robert W. Sebesta. *Concepts of Programming Languages.* Addison-Wesley Publishing Company, third edition, 1996.

[43] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley Longman, Inc., 2000.

[44] William Symes, Anthony Padula, Hala Dajani, and Eric Dussaud. A time-stepping library for simulation-driven optimization. Technical report, Rice University, 2003.

[45] R.S. Tuminaro, M. Heroux, S. A. Hutchinson, and J.N. Shadid. *Official Aztec User's Guide: Version 2.1.* Sandia National Laboratories, December 1999.

[46] Martin Fowler with Kendall Scott. *UML Distilled: a brief guide to the standard object modeling language.* Addison Wesley Longman, Inc., 2000.