SAND2002-3198
Unlimited Release
October 2002

Large Scale Non-Linear Programming for PDE Constrained Optimization.

Bart van Bloemen Waanders, Roscoe Bartlett, Kevin Long, Paul Boggs, Andrew Salinger
Sandia National Laboratories

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.doe.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/ordering.htm

SAND2002-3198
Unlimited Release
Printed October 2002

Large Scale Non-Linear Programming for PDE Constrained Optimization.

Bart van Bloemen Waanders and Roscoe Bartlett,
Optimization and Ucertainty Quantification Department

Kevin Long and Paul Boggs
Computational Science & Math

Andrew Salinger
Computational Sciences

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Abstract

Three years of large-scale PDE-constrained optimization research and development are summarized in this report. We have developed an optimization framework for 3 levels of SAND optimization and developed a powerful PDE prototyping tool. The optimization algorithms have been interfaced and tested on CVD problems using a chemically reacting fluid flow simulator resulting in an order of magnitude reduction in compute time over a black box method. Sandia's simulation environment is reviewed by characterizing each discipline and identifying a possible target level of optimization. Because SAND algorithms are difficult to test on actual production codes, a symbolic simulator (Sundance) was developed and interfaced with a reduced-space sequential quadratic programming framework (rSQP++) to provide a PDE prototyping environment. The power of Sundance/rSQP++ is demonstrated by applying optimization to a series of different PDE-based problems. In addition, we show the merits of SAND methods by comparing seven levels of optimization for a source-inversion problem using Sundance and rSQP++. Algorithmic results are discussed for hierarchical control methods. The design of an interior point quadratic programming solver is presented.

Acknowledgment

The authors acknowledge the insightful and significant contributions of our collaborators:

Omar Ghattas - Carnegie Mellon University
Larry Biegler - Carnegie Mellon University
Matthias Heinkenschloss - Rice University
George Biros - Courant, New York University
Jon Tolle - University of North Carolina
Anthony Kearsley - National Institute of Standards and Technology

The authors would also like to acknowledge students and post docs who contributed to the project in many ways:

Volkan Akcelik - Carnegie Mellon University
Tony Padula - Rice University
Agata Comas - Rice University
Greg Itle - Carnegie Mellon University
Carl Laird - Carnegie Mellon University

Finally, the authors would like to acknowledge William Hart and Mike Eldred for initiating the project and Roger Pawlowski for his contributions to the CVD optimization problem.

The format of this report is based on information found in [71].

Contents

Figures

Tables

Chapter 1

Introduction

This report presents the results of a three-year research project to investigate algorithms and software for the solution of optimization problems constrained by partial differential equations (PDE). We refer to these problems as PDE-constrained optimization problems, or PDECO. Our emphasis has been on developing algorithms for large-scale problems and the use of parallel computers.

Several examples of PDECO are optimal estimation of material parameters, given experimental data and a physical model; optimal design of a device given a simulator and the definition of an objective; nondestructive detection of defects; and determination of the source of a contaminant, given a flow and dispersion model. All of these problems exhibit large numbers of state and design variables and can be stated in the general form

$$\begin{array}{ll} \underset{y,u}{\text{minimize}} & f(y,u) \\ \text{subject to:} & c(y,u) = 0 \end{array} \qquad (PDECO)$$

where u is the set of parameters to be determined and y is the vector of "state" variables for the PDE system represented by the constraint $c(y,u) = 0$. The objective function measures the discrepancy that we wish to reduce or, in other problems, the design criteria we wish to improve. In this report, we concentrate on equality constraints; our work on inequality constraints is less well developed. We assume that given any value of the parameters, u, we can compute the corresponding state variables y.

Two general approaches for solving such problems are available. The first is to use an existing PDE solver for the constraints to compute y as a function of u and evaluate $f(y(u), u)$. This approach, referred to as the "black-box" approach, is easy to use because it requires no modification to an existing PDE simulator, but restricts the choice of optimization algorithm to those that are slowly convergent for PDECO type problems. Ideally, we would like to use a method that converges quickly to the optimum value, but rapid convergence usually requires the computation of the gradient of the objective function f with respect to u . The computation of this gradient, however, requires the knowledge of the derivative of $y(u)$ with respect to u and this information is not often available from many traditional PDE solvers. Furthermore, it is often extremely difficult, if not impossible as a practical matter, to modify the PDE solver to compute this information. For these and other reasons (detailed in chapter 2) black-box methods are typically restricted to smaller size problems, in particular smaller design spaces.

The second possibility is already suggested by the above discussion, namely, to modify the PDE solver to obtain the needed gradient, sensitivity, and adjoint information. The ability to do this opens up a wide variety of more efficient optimization techniques and provides the tools to address much larger problems. The demonstration of the power of this approach was the major thrust of our work. The conclusion to draw is that PDE and simulation software should be designed with optimization in mind to enable this power to be applied to the many interesting and important SNL problems described below. One facet of our research has been to develop a PDE framework that gives optimization algorithms unprecedented control over the PDE processes.

Black-box methods are also referred to as nested analysis and design (**NAND**) and characterize the majority of current SNL approaches. The ability to interface seamlessly with any simulation code is an obvious key strength of the black-box methods and, coupled with a range of algorithms and frameworks, such DAKOTA [37, 38, 39], have been able to solve complex engineering design problems. As noted above, however, many limitations to this strategy remain, but the continued existence of PDE codes for which gradient information is not available has spurred other research at SNL in pattern search methods to try to improve the efficiency of these solvers for problems where there are no other choices [57].

Optimization methods that are able to obtain gradient, adjoint, and sensitivity information from the PDE solver can often be even more successful by not requiring exact solution of the constraint equations at each iteration. This is especially important in problems where the PDE constraints are nonlinear. In such cases, the constraints are only completely satisfied in the limit as convergence to the optimal parameters is achieved. Thus this strategy is called simultaneous analysis and design (**SAND**) [94] [83]. These methods have great potential for solving large PDECO problems. There are many assumptions associated with the application of **SAND** algorithms to production simulation codes and probably the most obvious disadvantage is the implementation cost necessary to equip PDE solvers with the necessary facilities to compute gradient information. Nevertheless, PDECO may be the only option to address large design spaces.

1.1 State of the Field

To put our work into context, we briefly survey the historical development and current state of algorithms and software for PDECO.

1.1.1 Algorithms and applications

Several areas of research have motivated the development of PDE constrained optimization, including shape optimization in computational fluid dynamics [2] [8] [43] [44] [21] [22] material inversion in geophysics [81] [82] [3], data assimilation in regional weather prediction modeling [124] [74], structural optimization [86] [92] [93], and control of chemical processes [17]. A complete discussion of all the aforementioned disciplines is beyond the scope of the report. Shape optimization in computational fluid dynamics (CFD), however, has arguably made the largest contributions toward direct and adjoint sensitivities, which is one of the important pieces of information needed by **SAND** algorithms, and we therefore provide a brief background of some key developments.

In general, the shape optimization problem for CFD is extremely expensive since the standard solution approach requires the complete solution of computationally expensive flow equations for each optimization iteration. Pironeau first studied derivative-based shape optimization using the adjoint formulation for minimum drag for both Stokes and incompressible Navier-Stokes flow [90]. Jameson applied the adjoint method to shape optimization using the Euler equations [62]. Numerous results have since been published on shape optimization [79] [5] [6] [7] [14] [32], including compressible Navier-Stokes simulations shape optimization of three dimensional wings [75] . Different solution procedures have been attempted to try to improve the convergence of shape optimization algorithms. A "one-shot-method" was introduced early in the 1990's which used multi-grid methods where the optimization and forward problems were solved with different levels of grid fidelity [114]. Typically, the optimization problems were solved on coarser meshes. These methods still required complete convergence of the flow code for each optimization iteration, but could be considered the first attempt toward **SAND** methods. The result was a significant reduction of the overall solution time.

SAND was introduced in the early eighties and nineties [52] [94] [83] and has developed momentum as the state-of-the-art methodology for optimization of large-scale simulation problems. Significant results have been generated for **SAND** methods, in particular for the serial case [2] [61] [8] [43] [114] [12] [67]. Less rapid advances have been made in the area of parallel PDE-constrained optimization. The primary reason for this slow progress is that forward simulation code development has only recently reached a high level of maturity. Combined with the continuing growth in computer capabilities, large-scale PDECO for parallel applications is now an important area of research [44] [20] [21] [22]. Most parallel developments, however, involve specialty simulation codes connected to tailored optimization methods, thereby avoiding some of the interfacing issues that are encountered with legacy production codes. One of our primary goals was to address these interfacing issues, and although interfacing remains problematic, we have made significant progress in our software tools and general understanding of production PDE simulators.

Transient simulation poses yet another level of difficulty to large-scale PDE-constrained optimization. One of the main obstacles is the efficient calculation of sensitivities in a time-stepping scheme for large design spaces. Several approaches can be considered, one of which is to utilize sensitivity calculations for differentiable algebraic equations (DAE) for reduced-gradient calculations. By converting a PDE system to DAEs, various methods, such as multiple shooting, can used to discretize in time [89] [46]. Even though large DAE systems can be solved, these methods are limited to small number of design parameters. For a large number of design variables, adjoint sensitivity in transient simulations have been considered but are not efficient because of the large storage requirements. This results from the need to integrate backward in time to calculate the adjoint vector, which requires storage of the forward problem's solution at every time step [64] [53]. A recent and most promising result from Akcelik et al [3] demonstrated a full space Gauss-Newton method in which they efficiently solved a 2.1 million variable inversion problem using the transient wave equation. Finally, work in the area of time decomposition and control also has produced promising algorithms and results [54].

1.1.2 Software

While progress has been made in developing algorithms for PDECO, the spread of these algorithms to production software has been slow because of the tight coupling required between optimizer and PDE simulation software. Although little work has been done on software frameworks for PDE-constrained optimization and, with the exception of the work presented in this report, virtually no work has been done on object-oriented frameworks for PDECO, several attempts have been made to collect PDECO algorithms in libraries (Veltisto and TRICE) [21] [22] [34]. An encouraging trend is that optimization codes are starting to be written in terms of flexible linear algebra interfaces such as PETSc [9], the Equation Solver Interface (ESI) [103], the Hilbert Class Library (HCL) [51], rSQP++ [10], Trilinos [55] and the Trilinos Solver Framework (TSF) [56]. Similarly on the PDE simulation side, the state of the art is evolving away from codes specialized to a particular discipline and toward general-purpose frameworks such as SIERRA and Nevada [111]. Identifying the additional changes to the design of both optimization software and PDE simulators that will be required for the use of PDECO has been a major focus of this LDRD project.

1.2 Accomplishments of this Project

1.2.1 Classification of PDECO Problems

Large-scale PDE-constrained optimization comes in many forms and the variety of algorithms and interfacing mechanisms presents a complex range of options for a heterogeneous simulation environment such as the one that exists at SNL. To achieve a general approach for **SAND** optimization for a large range of simulation codes is a lofty challenge, because by definition **SAND** methods leverage the linear algebra of the simulation code and therefore each interface needs to be custom designed. This research project addresses these interfacing problems through a variety of software tools and establishes a systematic nomenclature and approach for the consideration of **SAND** optimization. Chapter 2 will introduce a sequence of levels of coupling between PDE solver and optimizer, with Level 0 being the most loosely coupled and Level 6 being the most tightly coupled and potentially yielding the highest performance. Currently, most Sandia applications are capable of Levels 0 and 1 only.

Chapter 2 contains a discussion of the mathematical foundations of PDECO, and a systematic enumeration of the levels at which PDE and optimization codes can be coupled. Briefly, Level 0 is the most loosely coupled black-box algorithm and Level 6 is the most tightly coupled full-space algorithm, which potentially yields the highest performance. Currently, most Sandia PDE applications are capable of Levels 0 and 1 only. In Chapter 3 we discuss the general simulation environment at Sandia and possibilities for PDECO in various disciplines. In Chapter 7 we show performance results for different levels of PDECO.

1.2.2 Software Development

Software is a major challenge in PDECO, and much of our work has been to develop software tools that will aid the exploration of research ideas in PDECO, provide guidance for future development of production-quality PDECO capability, and provide immediate PDECO capability for Sandia problems. These tools have been designed from the start with PDECO and interoperability in mind. We have developed:

- A software framework (rSQP++) for solving reduced-space PDECO problems.
- A software framework (Split/O3D) for solving full-space and inequality-constrained PDECO problems.
- A PDE simulation component system (Sundance) that is capable of providing the additional operations required by the more strongly coupled levels of PDECO.
- An interface between rSQP++ and an existing production PDE code, MPSalsa, which allows **SAND** capability through Level 4.

All of these tools have been implemented in C++, all inter operate via the Trilinos linear algebra components, and all have parallel capability.

1.2.3 Numerical Experiments

We have conducted numerical experiments to evaluate the different levels of PDECO. In Chapter 5, we show results of Level 4 (c.f. Chapter 2) coupling between the rSQP++ optimizer and a Sandia production code, MPSalsa. This resulted in an order of magnitude speedup relative to a Level 1 “black box” method. These experiments have also given us insight into the accuracy required in Jacobian calculations. In Chapter 7, we present a survey of PDECO problems solved using rSQP++ and Sundance. Because of Sundance’s very flexible nature, we have been able to explore all levels of coupling for PDECO; as with MPSalsa, going to Level 4 yields an order of magnitude speedup relative to Level 1, and then going to the highest degree of coupling (at this point, possible only through using Sundance as the PDE solver), Level 6 yields a further order of magnitude speedup beyond Level 4. Figure 1.1 and 1.2 show the results of a numerical experiment solving a source inversion problem constrained by a convection diffusion problem. Large differences in numerical efficiencies can be be observed at each level of optimization.

1.2.4 Hierarchical Control

Although not originally part of the proposal, an interesting class of problems arose that we spent some time considering. In particular, it often occurs in applications that there is more than one objective. The so-called "multi-objective" optimization problem has some special properties when the constraints are PDEs. Suppose, for example, one wants to drill a well into an aquifer to help in preventing contaminants from entering a city water system. The primary objective is to reduce the contaminants in the system below a given threshold. Secondary objectives may include minimizing cost, minimizing the time to completion, and minimizing the amount of water taken out of the aquifer. There is no single way to handle the multi-objective problem, but, building on work done in the area, we were able to show how a formulation, called "hierarchical control," that takes into account an ordering of the objectives can yield solutions that are significantly smoother and thus more useful in many applications. This work is described in Chapter 8 where we demonstrate the effectiveness of our full-space **SAND** approach on a problem with significant inequality constraints.

1.3 Conclusions and Recommendations

With the increasing power of our massively parallel computing platforms and the increasing sophistication of our PDE-based simulations comes an increasing demand for optimization procedures that can exploit this power to improve designs significantly, to control processes better, and to solve complex inversion problems more rapidly. As we have indicated above, traditional, **NAND** approaches to solve PDECO problems are not up to the task and traditional PDE solvers are not designed with optimization in mind and thus are difficult to use with faster methods. The major result of this research project is a demonstration that the potential speedup resulting from modern **SAND** approaches can be achieved. This demonstration was made possible by developing a powerful PDE environment and two advanced optimization codes, which were to nontrivial problems. For SNL to realize fully this potential will require changes in how SNL develops its simulation codes and PDE solvers. The following recommendations address these issues.

1.3.1 Recommendations

1. Because of the large speedup resulting from sensitivities and **SAND** optimization, future simulators and PDE solvers should be designed with optimization in mind and, in particular,

with enhancements that include gradients, sensitivities, and adjoints. These features are difficult to add as an afterthought. Along these same lines, Sierra and Nevada should be extended to include these capabilities.

2. rSQP++ and Split/O3D should be further developed and also interfaced with DAKOTA. DAKOTA is already interfaced with SIERRA, and will eventually be interfaced to Nevada, so the rSQP++/Split/O3D interface to DAKOTA will facilitate PDECO with SIERRA and Nevada.

3. The development of Sundance as a prototyping and rapid development environment for parallel PDECO should be continued. In addition, the possibility of interfacing the Sundance symbolic problem definition capability with Sierra and Nevada should be explored as a path to providing improved PDECO capability to those frameworks.

4. SNL should emphasize the development of frameworks and tools in C++ in a true object-oriented manner. Developers should be encouraged to exploit existing frameworks such as DAKOTA, rSQP++, Trilinos, and TSF and to develop inter-operable components.

5. Incorporate the sensitivity procedures that we have demonstrated with MPSalsa into ongoing projects such as Xyce and Premo.

6. Extend the research and tools begun here to transient problems, inequality constraints, and real-time optimization.

7. Apply PDECO technology to homeland security applications such as improving the response to chemical/biological/radiological attacks on facilities, water distribution networks, and urban facilities.

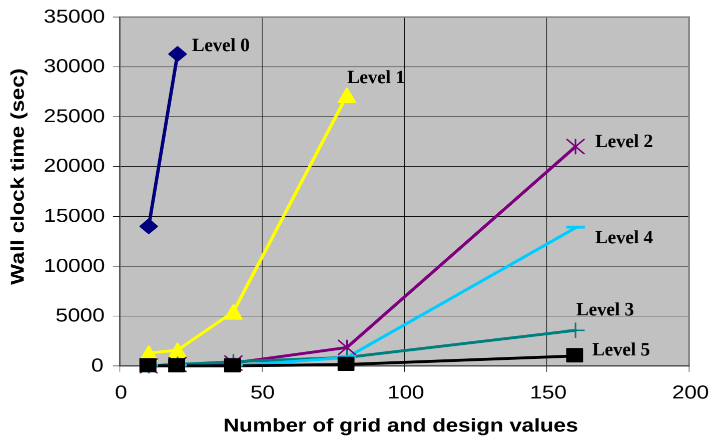


Figure 1.1. Numerical Results for Source Inversion for Convection Diffusion for levels 0-5

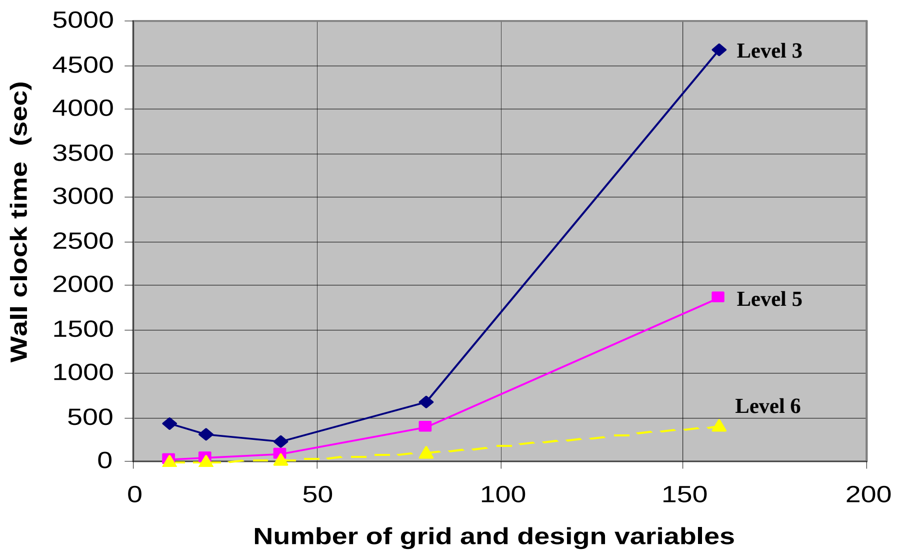


Figure 1.2. Numerical Results for Source Inversion for Convection Diffusion levels 3-6

Chapter 2

Mathematical Overview: Sensitivities and Levels of Optimization

2.1 Overview

An introduction of the appropriate mathematics is presented to emphasize the important linear algebraic components that are necessary for interfacing various levels of optimization methods. This discussion is primarily designed to provide the PDE developer with the fundamental knowledge for considering more efficient ways of solving optimization problems. Optimization methods are classified into two main categories, **NAND** and **SAND**, each of which are further broken down to additional levels. The optimization levels define different interfaces for achieving higher efficiency. The calculation of the derivatives (sensitivities) of “state” variables with respect to “design” variables is a crucial step toward more efficient levels of optimization using so-called reduced-space methods. The incorporation of different types of sensitivities determine which levels of **NAND** and **SAND** optimization are possible.

We consider equality-constrained nonlinear programs (NLPs) of the form

$$\min_{y,u} \quad f(y,u) \tag{2.1.1}$$

$$\text{s.t.} \quad c(y,u) = 0 \tag{2.1.2}$$

where:

$$\begin{aligned} y &\in \mathbf{R}^{n_y} \\ u &\in \mathbf{R}^{n_u} \\ f(y,u) &: \mathbf{R}^{(n_y+n_u)} \rightarrow \mathbf{R} \\ c(y,u) &: \mathbf{R}^{(n_y+n_u)} \rightarrow \mathbf{R}^{n_y}. \end{aligned}$$

Equation (2.1.2) represents a set of nonlinear simulation equations which we refer to as the *constraints*. In this notation, $c(y,u)$ is a vector function where each component $c_j(y,u)$, $j = 1 \ldots n_y$, represents a nonlinear scalar function of the variables y and u. Here, u are often called the design (or control, inversion) variables while y are referred to as the state (or simulation) variables. Note that the number of state y and design u variables are n_y and n_u respectively. A typical simulation code requires that the user specify the design variables u and then the square set of equations $c(y,u) = 0$ is solved for y. In the optimization problem in (2.1.1)–(2.1.2), $f(y,u)$ is a function that we seek to minimize while satisfying the constraints; this function is called the *objective function* or just the *objective*. In an optimization problem, the design variables u are left as unknowns which are determined, along with the states y, in the solution of (2.1.1)–(2.1.2). In some application areas, the partitioning into state and design variables is fixed and known a priori, while in other application areas the selection may be arbitrary.

Here we discuss the issues involved in modifying an existing simulation code or developing a new code that can be used to solve optimization problems efficiently using various levels of gradient-based methods.

The development effort required to implement the needed functionality for a simulation code to be used in a gradient-based optimization algorithm varies depending on the level of optimization method. The goal of this discussion is to be precise about what the requirements are for a simulation code for different levels of intrusive optimization. We define intrusive optimization as methods that require more information from the simulation code and may require more effort to interface. We start with sensitivities for the lower-level optimization methods and then move on to the sensitivities for the more invasive, higher-level methods. This discussion should give the reader some idea what the expected improvements in performance can be by going to higher-level optimization methods. An additional goal of this treatment is to motivate simulation application developers to consider the potential of higher-level optimization methods and to study optimization methods in further detail. References are made to more thorough discussions of specific optimization methods and results from various application areas for the interested reader.

We should also mention that all of the various levels of optimization methods that are discussed here can also handle extra constraints beyond the state constraints shown in (2.1.2). From the standpoint of an application developer, the sensitivity requirements for these extra constraints are the same as for the objective function. In general, the same types of computations that must be performed for the objective function must also be performed for the extra constraints. The handling of these extra constraints is not described here, but is described in the context of reduced-space SQP in Section 4.1.3.

2.2 Implicit State Solution and Constraint Sensitivities

The set of nonlinear equations $c(y,u)=0$ can be solved for y using a variety of methods. Using the solution method it is possible to define an implicit function

$$y = y(u),\ \text{s.t.}\ c(y,u)=0. \tag{2.2.3}$$

The definition in (2.2.3) simply implies that for any reasonable selection of the design variables u, the solution method can compute the states y. Note that evaluating this implicit function requires a complete simulation or "analysis" to be performed by the solution method. The cost of performing the analysis may only be an $O(n_y)$ computation in a best-case-scenario, but for many applications the complexity of the analysis solution is much worse.

In the remainder of this section, we derive the sensitivities of the states y with respect to the designs u as related through the implicit function (2.2.3). We begin with a first-order Taylor expansion of $c(y,u)$ about (y_0,u_0) given by

$$c(y,u) = c(y_0,u_0) + \frac{\partial c}{\partial y}\delta y + \frac{\partial c}{\partial u}\delta u + O(||\delta y||^2) + O(||\delta u||^2) \tag{2.2.4}$$

where:

$\frac{\partial c}{\partial y}$ is a square $\mathbf{R}^{n_y}$-by-$\mathbf{R}^{n_y}$ Jacobian matrix evaluated at (y_0,u_0)
$\frac{\partial c}{\partial u}$ is a rectangular $\mathbf{R}^{n_y}$-by-$\mathbf{R}^{n_u}$ Jacobian matrix evaluated at (y_0,u_0).

In this notation, the Jacobian matrix $\frac{\partial c}{\partial y}$ is defined element-wise as

$$\left(\frac{\partial c}{\partial y}\right)_{(j,l)} = \frac{\partial c_j}{\partial y_l}\text{ , for} j = 1\ldots n_y, l = 1\ldots n_y.$$

If the matrix $\frac{\partial c}{\partial y}$ exists and is nonsingular then the implicit function theorem [80, B.9] states that the implicit function in (2.2.3) exists and is well defined in a neighborhood of a solution (y_0,u_0). In some applications, the matrix $\frac{\partial c}{\partial y}$ is always nonsingular in regions of interest. In other application areas where the selection of state and design variables is arbitrary, the variables are partitioned into states and designs based on the non-singularity of $\frac{\partial c}{\partial y}$. Note that the only requirement for the latter case is for the Jacobian of $c(y,u)$ to be full rank. In any case, we will assume for the remainder of this discussion that, for the given selection of states and designs, the matrix $\frac{\partial c}{\partial y}$ is nonsingular for every point (y,u) considered by an optimization algorithm. The non-singularity of $\frac{\partial c}{\partial y}$ allows us to compute a relationship between changes in y with changes in u. If we require that the residual not change (i.e. $c(y,u)=c(y_0,u_0)$) then for sufficiently small δy and δu the higher order terms can be ignored and (2.2.4) gives

$$\frac{\partial c}{\partial y}\delta y + \frac{\partial c}{\partial u}\delta u = 0. \tag{2.2.5}$$

If $\frac{\partial c}{\partial y}$ is nonsingular then we can solve (2.2.5) for

$$\delta y = -\frac{\partial c}{\partial y}^{-1} \frac{\partial c}{\partial u} \delta u. \tag{2.2.6}$$

The matrix in (2.2.6) represents the sensitivity of y with respect to u (for $c(y,u) =$ *constant*) which defines

$$\frac{\partial y}{\partial u} \equiv -\frac{\partial c}{\partial y}^{-1} \frac{\partial c}{\partial u}. \tag{2.2.7}$$

We refer to the matrix $\frac{\partial y}{\partial u}$ in (2.2.7) as the *direct sensitivity matrix*.

2.3 NAND

Now consider how the above can be used to help solve optimization problems of the form (2.1.1)–(2.1.2). The implicit function $y(u)$ allows the nonlinear elimination of the state variables y and the constraints $c(y,u) = 0$ to form the *reduced objective function*

$$\hat{f}(u) \equiv f(y(u), u). \tag{2.3.8}$$

This nonlinear elimination leaves the following unconstrained optimization problem in the space of the design variables only:

$$\min \hat{f}(u). \tag{2.3.9}$$

The unconstrained optimization problem in (2.3.9) can be solved using a variety of methods. Note that each evaluation of $\hat{f}(u)$ requires the evaluation of $y(u)$ which involves a complete simulation or analysis to solve $c(y,u) = 0$ for y. Therefore, a complete analysis is nested inside of each optimization or design iteration. Optimization approaches of this type are broadly categorized as *nested analysis and design* or **NAND**.

NAND optimization approaches that do not compute gradients will be referred to as **level-0** approaches and, as mentioned in the chapter 1, are generally restricted to search methods. These will not be discussed further here. As we will see below, there are several higher-level approaches that use sensitivities (i.e. derivatives).

Gradient-based optimization methods for (2.3.9) require the computation of the *reduced gradient*

$$\frac{\partial \hat{f}}{\partial u} \in \mathbf{R}^{1 \times n_u}. \tag{2.3.10}$$

There are several relatively fast optimization methods that rely only on the reduced gradient in (2.3.10) such as quasi-Newton methods (i.e. BFGS [85, Chapter 8]). These methods can achieve superlinear rates of convergence when u is of moderate dimension. A general outline for these optimization algorithms is given next in Algorithm 2.3.1.

Algorithm 2.3.1 : *Outline for **NAND** Algorithms for Unconstrained Optimization*

```
1. Initialization: Choose tolerance $\eta \in \mathbf{R}$ and the initial guess $u_0 \in \mathbf{R}^{n_u}$, set $k = 0$
2. Sensitivity computation: Compute the reduced gradient $\frac{\partial \hat{f}}{\partial u}$ at $y = y(u_k), u = u_k$
3. Convergence check: If $||\frac{\partial \hat{f}}{\partial u}|| \leq \eta$ then stop, solution found!
4. Step computation: Compute $\delta u \in \mathbf{R}^{n_u}$ s.t. $\frac{\partial \hat{f}}{\partial u}\delta u < 0$
5. Globalization: Find step length $\alpha$ that ensures progress to the solution
6. Update the estimate of the solution:
   $u_{k+1} = u_k + \alpha\, \delta u$
   $k = k + 1$
   goto step 2
```

A simple choice for the step computation in step 4 of Algorithm 2.3.1 is the steepest descent direction $\delta u = -\frac{\partial \hat{f}}{\partial u}^T$ for which the required descent property holds

$$\frac{\partial \hat{f}}{\partial u}\delta u = -\frac{\partial \hat{f}}{\partial u}\frac{\partial \hat{f}}{\partial u}^T < 0$$

if $\frac{\partial \hat{f}}{\partial u} \neq 0$. Most quasi-Newton methods compute a search direction δu by maintaining a positive-definite matrix B and then computing $\delta u = -B^{-1}\frac{\partial \hat{f}}{\partial u}^T$ (which is also easy to show has the descent property).

The simplest way to compute the reduced gradient is to use finite differences. For example, using one-sided finite differences, each component of the reduced gradient can be approximated as

$$\left(\frac{\partial \hat{f}}{\partial u}\right)_i \approx \frac{f(y(u_k + \epsilon e_i), u_k + \epsilon e_i) - f(y(u_k), u_k)}{\epsilon}, \; i = 1 \ldots n_u. \qquad (2.3.11)$$

NAND optimization approaches that use finite differences as in (2.3.11) will be referred to as **level-1** approaches.

2.3.1 Exact Reduced Gradients

The major drawback of optimization approaches that rely on the finite-difference reduced gradient in (2.3.11) is that n_u analyses are required per optimization iteration and the accuracy of the computed optimal solution is degraded because of the truncation error involved with finite differences.

An alternative approach is to compute the reduced gradient in a more efficient and accurate manner. The exact reduced gradient of $\hat{f}(u) = f(y(u), u)$ is

$$\frac{\partial \hat{f}}{\partial u} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial u} + \frac{\partial f}{\partial u} \tag{2.3.12}$$

where:

$\frac{\partial f}{\partial y}$ is a $\mathbf{R}^{1\times n_y}$ row vector of the gradient w.r.t. y evaluated at (y_k, u_k)
$\frac{\partial f}{\partial u}$ is a $\mathbf{R}^{1\times n_u}$ row vector of the gradient w.r.t. u evaluated at (y_k, u_k)

and $\frac{\partial y}{\partial u}$ is the direct sensitivity matrix defined in (2.2.7). By substituting (2.2.7) into (2.3.12) we obtain

$$\frac{\partial \hat{f}}{\partial u} = -\frac{\partial f}{\partial y}\frac{\partial c}{\partial y}^{-1}\frac{\partial c}{\partial u} + \frac{\partial f}{\partial u}. \tag{2.3.13}$$

The first term in (2.3.13) can be computed in one of two ways. The first approach, called the *direct sensitivity approach*, is to compute the direct sensitivity matrix $\frac{\partial y}{\partial u} = -\frac{\partial c}{\partial y}^{-1}\frac{\partial c}{\partial u}$ first and then compute the product $\frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$. The advantage of this approach is that many simulation codes are already setup to solve for linear systems with $\frac{\partial c}{\partial y}$ since they use a Newton-type method to solve the analysis problem. The disadvantage of the direct sensitivity approach is that to form $\frac{\partial y}{\partial u}$, n_u linear systems must be solved with the Jacobian $\frac{\partial c}{\partial y}$ for each column of $\frac{\partial c}{\partial u}$ as a right-hand side. This is generally a great improvement over the finite-difference reduced gradient in that the solution of a n_u linear systems with $\frac{\partial c}{\partial y}$ is cheaper than a full simulation to evaluate $y(u)$ and the resulting reduced gradient is much more accurate. Optimization algorithms that use this direct sensitivity **NAND** approach will be referred to as **level-2** optimization methods.

The second approach for evaluating (2.3.13), called the *adjoint sensitivity approach*, is to compute

$$\lambda = \frac{\partial c}{\partial y}^{-T}\frac{\partial f}{\partial y}^{T} \in \mathbf{R}^{n_y} \tag{2.3.14}$$

first, followed by the formation of the product $\lambda^T\frac{\partial c}{\partial u}$. The column vector λ is called the vector of *adjoint variables* (or the Lagrange multipliers, see (2.4.16)). The advantage of this approach is that only a single solve with the matrix $\frac{\partial c}{\partial y}^T$ is required to compute the exact reduced gradient. This removes the $O(n_u)$ complexities of the level-1 and level-2 optimization approaches. However, at least one complete analysis is still required per optimization iteration to compute $y = y(u_k)$ in step 2 of Algorithm 2.3.1. The disadvantage of the adjoint sensitivities approach is that simulation codes which solve linear systems with the Newton Jacobian $\frac{\partial c}{\partial y}$ may not be able to solve a linear system efficiently with its transpose. It can be a major undertaking to revise a simulation code to solve with transposed systems, especially if the Jacobian is a parallel object. **NAND** approaches that use adjoint sensitivities will be categorized as **level-3** optimization methods.

2.4 SAND

To this point we have only considered **NAND** optimization approaches that require at least one full simulation problem $c(y,u) = 0$ be solved at every optimization iteration. There are also optimization approaches starting with an initial guess (y_0,u_0) where $c(y_0,u_0) \neq 0$ that will solve the simulation (analysis) problem and the optimization (design) problems simultaneously. These higher-level optimization approaches are referred to as *simultaneous analysis and design* or **SAND**. Many of the **SAND** approaches require the same reduced gradient in (2.3.13). We refer to **SAND** methods that use direct sensitivities as **level-4** methods and those that use adjoint sensitivities as **level-5** methods. In addition to the reduced gradient, level-4 and level-5 **SAND** methods also require that the simulation code (now to be referred to as the application) be able to compute Newton steps of the form

$$\delta y_N = \frac{\partial c}{\partial y}^{-1} c \tag{2.4.15}$$

where $\frac{\partial c}{\partial y}^{-1}$ and c are the Jacobian and the residual of the constraints $c(y,u)$ computed at the current estimate of the solution (y_k,u_k). This is usually not a very difficult extra requirement given the requirements for the reduced gradient. In addition to the requirement that the reduced gradient $\frac{\partial \hat{f}}{\partial u}$ vanishes, **SAND** methods must also be responsible for solving $c(y,u) = 0$ to an acceptable tolerance. The condition that $||c(y,u)||$ (where $||.||$ is some norm) must be reduced below a small tolerance is known as the *feasibility condition*. When we say that an optimization step improves feasibility, we mean that it decreases the *infeasibility* $||c(y,u)||$. In addition to design variables u, **SAND** methods must also explicitly handle the states y as optimization variables. The number of state variables n_y can be very large and this has a significant impact on the methods and implementation approaches that can be used for **SAND** methods. In some applications (e.g. those requiring time-dependent simulations), the amount of storage just needed to store vectors of size n_y can exhaust the RAM of even high-end supercomputers. Algorithm 2.4.1 gives the outline for a basic level-4/level-5 **SAND** method.

Algorithm 2.4.1 : *Outline of a Basic Level-4/Level-5 **SAND** Optimization Algorithm*

1. *Initialization: Choose tolerances $\eta_c, \eta_f \in \mathbf{R}$ and the initial guess $y_0 \in \mathbf{R}^{n_y}$ and $u_0 \in \mathbf{R}^{n_u}$, set $k = 0$*
2. *Sensitivity computation: Compute the reduced gradient $\frac{\partial \hat{f}}{\partial u}$ and the residual c at (y_k,u_k)*
3. *Convergence check: If $||\frac{\partial \hat{f}}{\partial u}|| \leq \eta_f$ and $||c|| \leq \eta_c$ then stop, solution found!*
4. *Step computation:*
   (a) *Feasiblity step: Compute Newton step $\delta y_N = \frac{\partial c}{\partial y}^{-1} c$ at (y_k,u_k)*
   (b) *Optimality step: Compute $\delta u \in \mathbf{R}^{n_u}$ s.t. $\frac{\partial \hat{f}}{\partial u}\delta u < 0$*
5. *Globalization: Find step length α that ensures progress to the solution*

```
6. Update the estimate of the solution:
   $y_{k+1} = y_k + \alpha \left( \delta y_N + \frac{\partial y}{\partial u} \delta u \right)$
   $u_{k+1} = u_k + \alpha \, \delta u$
   $k = k + 1$
   goto step 2
```

Note that Algorithm 2.4.1 has the same basic steps as Algorithm 2.3.1 and that these steps are common to many optimization algorithms. However, the first major difference is that the reduced gradient computed in step 2 is computed at the current estimate of the solution y_k instead of the fully converged solution $y = y(u_k)$ as in Algorithm 2.3.1. Another major difference is the explicit handling of the state variables y and the constraints c. This is seen in the sensitivity computation and the convergence check. The same methods that can be used in a **NAND** algorithm to compute δu, such as steepest descent and quasi-Newton, can also be used in step 4b. While the globalization method used in a **NAND** algorithm may be fairly simple, more sophisticated globalization strategies are needed for **SAND** and these strategies may have to be application dependent. The last major distinction to point out between Algorithm 2.3.1 and 2.4.1 is the update of the state variables y in step 6. It is easy to see that the updated y_{k+1} satisfies the linearized constraints shown in (2.2.4) (with the higher-order terms dropped out and setting $c(y, u) = 0$ and $\delta y = (y_{k+1} - y_k)/\alpha$). Therefore, one iteration of Algorithm 2.4.1 is essentially a Newton iteration for the equations $c(y, u) = 0$ where both y and u are modified. Hence, many **SAND** methods show quadratic rates of local convergence in the constraints (which is common for Newton methods).

What differentiates a level-4 from a level-5 **SAND** method in Algorithm 2.4.1 is how the reduced gradient in step 2 and the update for the states in step 6 are computed. The **SAND** algorithm shown in Algorithm 2.4.1 is essentially equivalent to a reduced-space SQP method that uses a coordinate variable-reduction null-space decomposition (see Section 4.1.3). While there are other examples of level-4 and level-5 **SAND** methods than the one shown in Algorithm 2.4.1, the major types of computations remains the same (i.e. intialization, sensitivity computation, convergence check, step computation and globalization).

It has been shown in many different application areas that level-5 optimization methods can compute a solution for optimization problems of the form in (2.1.1)–(2.1.2) at cost which is a small multiple of the cost of solving a single analysis problem $c(y, u) = 0$ for NLPs with a moderate number of design variables (i.e. $n_u = O(100)$). However, these methods, which use quasi-Newton or similar techniques (for step 4b in Algorithm 2.4.1), generally require more and more optimization iterations to solve an NLP as the number of design variables n_u is increased. The total number of optimization iterations required to reach an acceptable solution tolerance is generally $O((n_u)^{\alpha_{QN}})$ where α_{QN} is some number greater than 0 but generally less than 2.

2.4.1 Full Newton SAND

All of the level-2 through level-5 optimization methods only require first derivatives in the form of the Jacobian matrices $\frac{\partial c}{\partial y}$ and $\frac{\partial c}{\partial u}$ and objective gradients $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial u}$. However, if second derivatives for the constraints and objective function are available, then potentially more efficient higher-level optimization methods are available. Before discussing these higher-level methods and the requirements from application codes we must first present the formal optimality conditions for a solution to (2.1.1)–(2.1.2).

We begin with the definition of an important aggregate function called the *Lagrangian* given by

$$L(y,u,\lambda) \equiv f(y,u) + \lambda^T c(y,u) \tag{2.4.16}$$

where:

$\lambda \in \mathbf{R}^{n_y}$ is the vector of *Lagrange multipliers*.

Given the definition of the Lagrangian, the optimality conditions (also known as the KKT conditions [85]) state that the following are necessary requirements for the solution of (2.1.1)–(2.1.2):

$$\frac{\partial L}{\partial y} = \frac{\partial f}{\partial y} + \lambda^T \frac{\partial c}{\partial y} = 0 \tag{2.4.17}$$

$$\frac{\partial L}{\partial u} = \frac{\partial f}{\partial u} + \lambda^T \frac{\partial c}{\partial u} = 0 \tag{2.4.18}$$

$$\frac{\partial L}{\partial \lambda} = c(y,u) = 0. \tag{2.4.19}$$

All **SAND** methods seek a solution of this set of nonlinear equations. Note that (2.4.17) can be solved for λ and then substituted into (2.4.18), yielding the definition of the reduced gradient in (2.3.13). Therefore, the optimality conditions in (2.4.17) and (2.4.18) are equivalent to $\frac{\partial \hat{f}}{\partial u} = 0$.

The system of nonlinear equations in (2.4.17)–(2.4.19) can be solved using Newton's method which has the following linear subproblem (known as the KKT system)

$$\begin{bmatrix} \frac{\partial^2 L}{\partial y^2} & \frac{\partial^2 L}{\partial y \partial u}^T & \frac{\partial c}{\partial y}^T \\ \frac{\partial^2 L}{\partial y \partial u} & \frac{\partial^2 L}{\partial u^2} & \frac{\partial c}{\partial u}^T \\ \frac{\partial c}{\partial y} & \frac{\partial c}{\partial u} & \end{bmatrix} \begin{bmatrix} \delta y \\ \delta u \\ \delta \lambda \end{bmatrix} = - \begin{bmatrix} \frac{\partial L}{\partial y}^T \\ \frac{\partial L}{\partial u}^T \\ c \end{bmatrix}. \tag{2.4.20}$$

The above Hessians of the Lagrangian function are composites of the following Hessian matrices for the objective and the constraints:

$$\begin{aligned} \frac{\partial^2 f}{\partial y^2} &\in \mathbf{R}^{n_y \times n_y} \\ \frac{\partial^2 f}{\partial y \partial u} &\in \mathbf{R}^{n_y \times n_u} \\ \frac{\partial^2 f}{\partial u^2} &\in \mathbf{R}^{n_u \times n_u} \end{aligned} \tag{2.4.21}$$

$$\left.\begin{array}{l} \dfrac{\partial^2 c_j}{\partial y^2} \in \mathbf{R}^{n_y \times n_y} \\ \dfrac{\partial^2 c_j}{\partial y \partial u} \in \mathbf{R}^{n_y \times n_u} \\ \dfrac{\partial^2 c_j}{\partial u^2} \in \mathbf{R}^{n_u \times n_u} \end{array}\right\} j = 1 \ldots n_y. \tag{2.4.22}$$

Optimization methods that use second derivatives, or approximations to them, will be classified as **level-6** methods. These optimization methods are among the most sophisticated gradient-based methods developed to date and continue to be a topic of active research throughout the scientific community. The general outline for a level-6 **SAND** optimization method is given in Algorithm 2.4.2.

Algorithm 2.4.2 *: Outline of a Level-6* ***SAND*** *Optimization Algorithm*

1. *Initialization: Choose tolerances $\eta_c, \eta_y, \eta_u \in \mathbf{R}$ and the initial guess $y_0 \in \mathbf{R}^{n_y}$, $u_0 \in \mathbf{R}^{n_u}$ and $\lambda_0 \in \mathbf{R}^{n_y}$, set $k = 0$*
2. *Sensitivity computation: Compute $\frac{\partial L}{\partial y}$, $\frac{\partial L}{\partial u}$ $\frac{\partial^2 L}{\partial y^2}$, $\frac{\partial^2 L}{\partial y \partial u}$, $\frac{\partial^2 L}{\partial u^2}$ and c at (y_k, u_k)*
3. *Convergence check: If $||\frac{\partial L}{\partial y}|| \leq \eta_y$, $||\frac{\partial L}{\partial u}|| \leq \eta_u$ and $||c|| \leq \eta_c$ then stop, solution found!*
4. *Step computation: Solve the KKT system in (2.4.20) for $(\delta y, \delta u, \delta\lambda)$*
5. *Globalization: Find step length α that ensures progress to the solution*
6. *Update the estimate of the solution:*
   $y_{k+1} = y_k + \alpha\,\delta y$
   $u_{k+1} = u_k + \alpha\,\delta u$
   $\lambda_{k+1} = \lambda_k + \alpha\,\delta\lambda$
   $k = k + 1$
   goto step 2

Note that a level-6 **SAND** method must also maintain estimates of the Lagrange multipliers in addition to estimates of the states y and the designs u. Level-4 and level-5 **SAND** methods usually do not need an initial guess for λ_0 and do not maintain estimates of λ. The same globalization strategies used in level-4 and level-5 **SAND** methods can be used, unaltered, in a level-6 **SAND** method. In many applications areas, these level-6 optimization methods are quadratically convergent with algorithmic complexities that scale independently of the number of design variables n_u [20]. One of the main disadvantages of this level of invasiveness is that it is difficult for many different types of application codes to generate accurate second derivatives in a reasonably efficient manner. Therefore, there can be a large development overhead and computational expense involved in applying level-6 methods. The KKT system in (2.4.20) is expensive to solve and its solution is a bottleneck in a level-6 **SAND** method. Therefore, the most critical part of a level-6 **SAND** method is how the KKT system in (2.4.20) is solved and there are many different direct and iterative approaches; the best approach is, of course, application dependent.

2.5 Implementation Issues and Summary

In this section, we discuss several issues that relate to the implementation of sensitivities, the overall optimization method complexities/scalabilities, and the interface to optimization methods. First the 7 levels of optimization are summarized below:

Level 0 is a **NAND** nongradient "black box" approach where the optimizer does not require any information from the PDE code other than the objective function value per optimization iteration. This zero level is perfectly suited for simulation problems and codes that are complex and do not calculate exact Jacobians and do not require the investigation of large design spaces. Level 0 may be the only option for PDE codes where the complexity of the physics precludes the calculation of analytic derivatives and where standard approximations are poor. The interfacing cost is minimal, because most black-box methods can communicate through the file system.

Level 1 is a **NAND** gradient-based "black box" approach where the optimizer requires that the PDE code compute the objective function value and the gradient per optimization iteration. The gradient is typically calculated using a finite difference method. Level 1 is suited for simulation codes that are complex, but smooth enough to allow reasonable accuracy in the finite difference calculation. Level 1 is also suitable for problems that pose an insurmountable software challenge and/or do not require the investigation of large design spaces. Similar to level 0 approaches, the interfacing effort is minimal.

Level 2 is a **NAND** gradient-based method that uses direct sensitivities from the simulation code. There are a few simulation codes at Sandia that calculate direct sensitivities. Black-box approaches can typically take advantage of these sensitivities to calculate the objective function gradient. The cost of this calculation is more than repaid by the fact that no extra simulations are needed, unlike the use of finite differences. Besides the computational efficiency, direct sensitivities are more accurate, which results in a faster convergence rate and better solutions. The level of effort to develop direct sensitivities is highly dependent on the design of the simulation code. However, it is the obvious first step to improve efficiency and the obvious first step toward **SAND** optimization. As explained in chapter 2, most simulation codes are already designed to solve the linear system and the implementation of direct sensitivities requires solving this system with different right hand sides.

Level 3 is a **NAND** gradient-based method that uses adjoint sensitivities from the simulation code. Black-box approaches can again take advantage of these sensitivities to calculate the reduced gradient of the objective function. There are significant computational savings because it requires only one solution involving the transpose system of the Jacobian of the forward simulation (independent of the number of design variables n_u). Similar to direct sensitivities, the adjoint method produces accurate gradients. The effort to develop direct sensitivities is highly dependent on the design of the simulation code. If the simulation code has access to the Jacobian for the forward simulation and the simulation code solvers can be used on the transpose of the Jacobian, then the implementation is relatively inexpensive and straightforward. The adjoint formulation is a necessary step toward **SAND** optimization methods. Once the adjoint vector can be calculated, a considerable amount of the implementation effort is complete for a **SAND** method.

Level 4 is a **SAND** gradient-based method dependent on direct sensitivities. The implementation effort associated with direct sensitivities is the same as described for level 2. Instead of passing this information to a black-box optimizer, it is passed directly to algorithms closely coupled to the simulation. Additional implementation effort is therefore involved to make use of a closely coupled algorithm. The extent of the effort depends highly on how amenable a code is to coupling with other algorithms.

Level 5 is a **SAND** gradient-based method that is dependent on the "adjoint formulation". These algorithms require the solution of systems involving the transpose of the state Jacobian. This method is similar to level 3, except that for a nonlinear problem it is considerably more efficient.

Level 6 is known as the full-space method [20] and has the most computational potential for very large design spaces. The level of intrusiveness is the highest as a result of having to assemble and solve the full KKT system or the related QP subproblem. A full-space algorithm generally requires the calculation or approximation of second derivatives in the form of Hessian matrices.

It is very important to understand the implications of computing an accurate reduced gradient in (2.3.13) that is used in level-2 through level-5 methods and how this differs from the way that simulation codes are usually implemented. In a simulation code that uses Newton's method to solve $c(y, u_0) = 0$, it is not critical that exact solves with $\frac{\partial c}{\partial y}$ be performed, even near the solution. All that is required is a solution that improves feasibility (i.e. decreases $||c(y, u)||$). Therefore, many advanced simulation codes are designed to compute approximate Jacobians (i.e. operator splitting and other inexact methods) to make the computation of the solutions cheaper. For optimization this is generally unacceptable. Any significant error in the Jacobians will be reflected in the reduced gradient. In other words, inaccurate Jacobian information is reflected in inaccurate solutions to the optimization problem. This also applies to the Jacobian matrix $\frac{\partial c}{\partial u}$. While a simulation code may be designed to use exact Jacobians and to solve linear systems accurately with $\frac{\partial c}{\partial y}$ and may even be able to solve systems involving $\frac{\partial c}{\partial y}^T$ accurately, such a code is certainly not designed to compute efficient sensitivities for the design variables $\frac{\partial c}{\partial u}$. This matrix $\frac{\partial c}{\partial u}$ can be approximated using finite differences, but this will potentially impose an additional $O(n_u)$ cost per optimization iteration, even for the level-3 adjoint sensitivity approach. In addition, this sensitivity matrix must be exact (or as accurate as possible) or the wrong reduced gradient is computed. In some types of applications, the development effort and computational resources required to compute $\frac{\partial c}{\partial u}$ can be quite small, while in other areas computing this matrix can be difficult and/or expensive.

While exact first derivatives are essential for level 2-5 methods, exact second derivatives for level-6 methods are not as critical since second derivatives do not alter the optimality conditions, but only the efficiency of the optimization algorithm. Quasi-Newton approximations, for example, may not be accurate at all, but they have drastically reduced the computational time on many problems [85, Chapter 10]).

In general, going from one level of optimization method to the next, interfacing a simulation code gets more difficult, but the resulting optimization algorithm becomes more efficient. Therefore, the real trade-off usually between different levels of intrusive optimization is that of developer (i.e. human) resources versus computational resources. For applications with fewer design variables, level-5 methods may actually be faster than level-6 methods because of the cost of computing (or approximating) and using second derivatives. For many other applications, reductions in computational time (which may not be very significant) do not justify the sometimes substantial investment in developer resources needed to implement a level-6 method. However, in cases with large numbers of design variables, level-6 methods offer the only hope of being able to solve difficult optimization problems using reasonable amount of computing resources.

Tables 2.1 and 2.2 summarize the various levels of intrusive optimization and the general requirements from simulation codes for **NAND** and **SAND** optimization methods. One of the more significant pieces of information in these tables is the specific requirements from simulation codes to be used with a particular level of intrusive optimization. The application requirements in each table are additive. For example, all of the requirements for level-2 methods are included in the requirements for level-3 methods. However, the availability of a quantity from a lower-level method in a higher-level method does not mean that that quantity will actually be computed. For example, the ability to compute the direct sensitivity matrix $\frac{\partial y}{\partial u}$ in a level-5 method does not mean that this matrix is actually computed. To compute $\frac{\partial y}{\partial u}$ in a level-5 method defeats the whole purpose of the adjoint computation.

Note that the complexity per optimization iteration and the general number of optimization iterations for level-2 methods is the same as for level-4 methods and the same comparison applies for level-3 and level-5 methods. The difference is that the higher-level methods have a smaller constant than the lower-level methods and these constants are not shown in $O(...)$ notation. The ratio of **NAND** verses **SAND** solution times will be application dependent, but there can be an order of magnitude difference or more with many applications for various reasons that we cannot discuss in detail here. In other applications, the differences in performance will be smaller.

The last issue is how the requirements listed in Tables 2.1 and 2.2 can be met by an application code and how this functionality can be used by an optimization algorithm. One of the major complications is that these simulation codes run in a variety of computing environments that range from simple serial single-process programs to massively parallel programs. Furthermore, the way that a linear system is solved may vary greatly among application areas. In some application areas, direct sparse solvers may be preferable (e.g. in chemical process simulation) while massively parallel preconditioned Krylov-subspace iterative solvers (e.g. in many PDE simulators) are the preferred methods. Or the linear adjoint equation in (2.3.14) could be solved using a nonmatrix-based method (e.g. using a time-stepping adjoint solver). Matrix operator invocations can also be performed in a variety of ways using different data structures. In addition, specialized data structures can be used in many application areas and can greatly improve performance. Therefore, a linear algebra interface that is flexible enough to allow for all of this variability is key to successfully being able to interface an advanced simulation code to a general purpose optimization algorithm. The details of one such interface are described in Section 4.2.3.

Optimization level	Application requirements (additive between levels)	Approximate complexity per optimization iteration	Approximate number of optimization iterations
level-0	Evaluation of objective $f(y,u)$, see (2.1.1) Analysis solution $y(u)$, see (2.2.3)	$O(n_y n_u)$	polynomial to exponential in n_u
level-1	Smoothness of $f(y,u)$ and $y(u)$	$O(n_y n_u)$	$O((n_u)^{\alpha_{QN}})$
level-2	Evaluation of direct sensitivity matrix $\frac{\partial y}{\partial u}$, see (2.2.7) Evaluation of objective gradients $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial u}$, see (2.3.12)	$O(n_y n_u)$	$O((n_u)^{\alpha_{QN}})$
level-3	Computation of adjoints $\lambda = \frac{\partial c}{\partial y}^{-T} \frac{\partial f}{\partial y}^{T}$, see (2.3.14)	$O(n_y)$	$O((n_u)^{\alpha_{QN}})$

Table 2.1. Summary of level-0 to level-3 **NAND** optimization methods.

Optimization level	Application requirements (additive between levels)	Approximate complexity per optimization iteration	Approximate number of optimization iterations
level-4	Evaluation of objective $f(y,u)$, see (2.1.1) Evaluation of constraints residual $c(y,u)$, see (2.1.2) Evaluation of direct sensitivity matrix $\frac{\partial y}{\partial u}$, see (2.2.7) Evaluation of objective gradients $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial u}$, see (2.3.12) Evaluation of Newton step $\delta y = \frac{\partial c}{\partial y}^{-1} c(y,u)$, see (2.4.15)	$O(n_y n_u)$	$O((n_u)^{\alpha_{QN}})$
level-5	Action of $\frac{\partial c}{\partial y}^{-1}$ on arbitrary vectors, see (2.4.15) Action of $\frac{\partial c}{\partial y}^{-T}$ on arbitrary vectors, see (2.3.14) Action of $\frac{\partial c}{\partial u}$ on arbitrary vectors Action of $\frac{\partial c}{\partial u}^{T}$ on arbitrary vectors	$O(n_y)$	$O((n_u)^{\alpha_{QN}})$
level-6	Evaluation of (or matrix-vector products with) Hessians $\frac{\partial^2 c_j}{\partial y^2}$, $\frac{\partial^2 c_j}{\partial y \partial u}$, $\frac{\partial^2 c_j}{\partial u^2}$, for $j = 1 \ldots n_y$, see (2.4.22) $\frac{\partial^2 f}{\partial y^2}$, $\frac{\partial^2 f}{\partial y \partial u}$, $\frac{\partial^2 f}{\partial u^2}$, see (2.4.21)	$O(n_y)$	$O(1)$

Table 2.2. Summary of level-4 to level-6 **SAND** optimization methods.

Chapter 3

PDE Environment

3.1 Overview

The engineering community has a critical need to simulate complex physics and, for the last few decades, has developed numerous production simulation codes to address high fidelity problems. Most of these codes have been parallelized and scale to hundreds and some to thousands of processors. This monumental development and parallelization effort has consumed developers for the last ten years with the somewhat unfortunate absence of any capabilities to address SAND optimization, although some codes can produce limited sensitivity information, which as previously discussed is an initial requirement for SAND optimization. The use of NAND methods in combination with large scale PDE simulation codes are limited to order ten design variables for the foreseeable future assuming the current trends in computer hardware growth do not change. PDECO is therefore a critical development strategy for those interested in the combination of large design space and gradient based optimization of large scale complex problems. Using the seven levels of optimization, we review the different simulation disciplines for SNL and attempt to identify appropriate optimization levels.

Before categorizing SNL simulation codes, additional issues regarding simulation codes need to be discussed:

1. **Implicit vs explicit** - The more efficient methods assume that the solution mechanism is implicit and that a Jacobian is formed so that Newton's method can be applied. Explicit codes depend on using solutions from the previous time step and Jacobians are never formed.

2. **Exact or inexact Jacobian** - The theoretical optimality conditions require that the Jacobian is exact. Robustness of the optimization algorithm depends on the accuracy of the gradient calculations. Any use of approximations could significantly affect the solution. Nevertheless, useful solutions have been obtained for many problems with finite-difference, or other approximate, gradients.

3. **Transient vs steady state** - Although methods have been developed for transient PDECO, significant efficiency problems still need to be resolved for the general optimization algorithms.

4. **Continuum or non-continuum** - SAND methods require smooth problems; to date, there is no reasonable way for non-continuum codes to take advantage of SAND based technologies. A classic example of non-continuum methods is the direct simulation Monte Carlo technique [18] [4] [77].

5. **Level of multi-physics** - Coupling different types of physics codes creates difficult issues for the higher level SAND methods. Issues such as explicit solution procedures and operator splitting are major hindrances to SAND methods.

6. **PDE smoothness -** Gradient methods require smooth behavior. Applications involving chemical reactions and state changes typically make use of database information and impose additional non-differentiable functions. Another example that gives rise to nondifferentiabilities is the gain or loss of material during the course of the computations.

3.2 Sandia Applications and Classifications

At Sandia, a large range of complex simulation codes have been developed to address a variety of high fidelity, complex physics problems in the area of structural dynamics, solid mechanics, thermal/radiation transport, computational fluid dynamics, fire, shock physics and electrical simulation. The scope of providing large-scale optimization capabilities to this engineering community in an efficient and practical fashion is considerable and continues to be a source for future research.

The following sections discuss general characteristics for each discipline and an attempt is made to identify the potential optimization level.

3.2.1 Structural Dynamics

Finite-element structural-dynamics simulation capabilities have been developed that are able to perform static analysis, direct implicit transient analysis, eigenvalue analysis, modal superposition-based frequency response, and transient response. Nonlinear capabilities are currently being developed. Shape optimization problems are the ultimate design target for structural dynamics and at Sandia there are a multitude of structural design challenges. The structural integrity of electronic packages for re-entry vehicles is an example of an important design problem. Although the number of design parameters are on the order of a hundred, as more sophistication to the structural design is added, the desire to investigate larger design spaces will increase.

Static analysis with nonlinear material behavior is another aspect of structural dynamics that can benefit from a SAND formulation. So-called inversion techniques to find the most likely materials in a medium is a potential area of interest that could lead to large number of design variables. However, the ultimate goal for structural dynamics is shape optimization where SAND methods can have a significant impact. Developing efficient optimization methods for transient problems remain a significant research challenge.

3.2.2 Solid Mechanics

Nonlinear solid mechanics is fundamental for investigating manufacturing and geomechanical issues. Finite-element codes have been developed that can handle large deformations, temperature dependency, and quasi-static mechanics problems in three dimensions. A material model for elastic and isothermal elastic-plastic behavior with combined kinematic and isotropic hardening is available. An eight node Lagrangian uniform-strain element is employed with hourglass stiffness to control the zero-energy modes. Highly nonlinear effects include material nonlinearities, geometric nonlinearities due to large rotations, large strains, and surfaces that slide relative to each other. Element birth and death algorithms are available to handle manufacturing situations where material is either added or removed, such as soldering and milling. Contact between surfaces can also be modeled with or without friction, which allows for simulating many difficult processes, such as connector insertion.

In addition to manufacturing examples, these codes are used to model geological systems subject to a variety of stresses. The Yuca Mountain nuclear storage facility is an example where the maximum safety margins for stresses need to be calculated as a function of various deformations to the storage facility and as a function of various loads onto the facility.

Although significant optimization issues exist in solid mechanics in addition to large design spaces, there a number of issues that prevents consideration of intrusive methods. Perhaps the most obvious impediment to SAND methods is the fact that solid mechanics codes do not for a Jacobian and use an explicit pseudo time stepping scheme to converge to a solution. Non differential quantities as a result of severe material deformation also poses a problem. There may be some possibilities for calculating direct sensitivities for a subset of problems, but presumably this would require restructuring the typical solid mechanics code. Certainly, birth/death algorithms are not differentiable and would require a complete new approach.

3.2.3 Thermal

Thermal simulation capabilities handle analysis of systems in which the transport of thermal energy occurs primarily through a conduction process. This nonlinear, finite element, multi-dimensional capability has been extended to handle solid phase chemical reactions and radiation transfer. A steady-state, nonlinear thermal problem without a chemical reaction is well suited for any SAND level optimization scheme. However, the usual difficulties are associated with the multi-coupled physics and transient analysis.

3.2.4 Computational Fluid Dynamics

3.2.4.1 Compressible Fluid Flow

Compressible fluid mechanics codes are needed to simulate accurately the aerodynamics for subsonic, transonic and supersonic flight. Many configurations and flight situations cannot be adequately tested because of high Mach numbers, high Reynolds numbers, and enthalpy conditions. Aerodynamic simulations calculate pressures, shear stress fields, and forces and moments exerted on a structure by the surrounding compressible fluids. If the assumptions for a rigid body fail, the structural response of the system needs to be included. This is often an explicit coupling and therefore a difficult issue for SAND optimization. However, there are numerous design problems in compressible fluid flow, such as steady-state Euler-based, shape optimization that can take advantage of any level of optimization method. A potential problem with compressible fluid flow problems is that the preferred solution mechanism is either matrix-free or pseudo time-stepping with multi-grid methods. The Jacobian is not formed and sensitivities cannot be easily calculated.

As a result of this LDRD project, development of an adjoint formulation is underway for Sandia's new compressible fluid flow code. The goal is to initially conduct shape optimization with the steady state Euler equations. An adjoint formulation for the Roe scheme has been developed and a forward Newton based solution is forthcoming.

3.2.4.2 Direct Simulation Monte Carlo

Computational fluid flow dynamics locally refines the simulation mesh in an attempt to resolve small-scale phenomena. However, hydrodynamic formulations break down as the grid spacing approaches the molecular scale. Direct Simulation Monte Carlo (DSMC) methods [18, 4, 77] are used as an alternative to continuum formulations. In DSMC, the state of the system is given by the position and velocities of particles, but the process decouples the movement from collisions and chemistry. First of all the particles are moved within a time step along a grid independently of each other. At the end of the time step the particles are sampled in each grid cell to determine collision behavior and species distributions using probabilistic techniques. At SNL, DSMC has been used to simulate low-density applications with Knudsen numbers on the order of 0.2 subjected to electromagnetic fields. Numerous other examples in the literature can be found [1, 107].

Clearly the lack of a continuum prevents the use of standard SAND methods and an entire simulation needs to be solved for any aspects of an optimization algorithm to occur. Sensitivity information will also be difficult to acquire by means other than the use of finite differences.

At SNL there are large design codes that predict the affects of certain geometries on the behavior of rarefied gases. These high-fidelity problems are computationally intensive; applying shape optimization, even with a small number of design parameters, requires an enormous amount of computational resources.

3.2.4.3 Incompressible Fluid Flow

Several Navier-Stokes codes have been developed to solve a number of complex design problems. We describe one such code in Chapter 5 for a chemical vapor deposition reactor problem. The general Navier-Stokes CFD simulator is well suited to take advantage of SAND optimization methods. Even though several Navier-Stokes codes have been extended to include chemistry, turbulence, moving interfaces, and elasto-viscoplastic materials, great care has been taken to include capabilities to form a full and exact Jacobian. These codes are complicated, however, and a level 6 interface may require a complete revision. Level-5 optimization is possible since the Jacobian in accessible and the solution of systems using its transpose is available.

3.2.5 Fire

The fire environment simulation software development project is directed at providing simulations for both open large-scale pool fires and building enclosure fires. One class of codes includes turbulence, buoyantly driven incompressible flow, heat transfer, mass transfer, combustion, soot formation, and absorption coefficient modeling. Another class of codes represent the participating-media thermal radiation mechanics. These fire codes rank as some of the more complex codes and are mostly developed with explicit solution methods to couple multi-physics, include approximations for different physics processes, use inexact Newton methods, and accommodate the loss of material.

Theoretically, an implicit coupling of the different physics could make a fire simulation a candidate for higher levels of optimization. The complexity of such a simulation suggests complex design problems and compute intensive simulations. However, the most problematic issue associated with fire simulation is the loss of material. As in solid mechanics, these algorithms are not differentiable. Even assuming loss of material is not an issue , the current explicit coupling still prevents the use of levels 3 or higher. Level 2 methods could be considered but would require cross sensitivities to accommodate the many different physics components. The calculation of cross sensitivities for multiple physics components is an active area of research.

3.2.6 Shock Physics

Shock Physics is handled through a family of codes that model complex multi-dimensional, multi-material problems that are characterized by large deformations and/or strong shocks. The solution strategy consists of a two-step, second-order accurate Eulerian algorithm to solve the mass, momentum, and energy conservation equations. Models exist for computing material strength, fracture, porosity, and high-explosive detonation and initiation. The problems that can be analyzed include penetration and perforation, compression, high explosive detonation and initiation phenomena, and hypervelocity impact. Strong shock simulations require sophisticated and accurate models of the thermodynamic behavior of materials. Phase changes, nonlinear behavior, and fractures are important to predict behavior accurately. Equation-of-state packages are used to predict phase changes.

More recently, Lagrangian solid mechanics capabilities were developed to include arbitrary mesh connectivity, superior artificial viscosity, and improved material models. Problems can be solved using Lagrangian, Eulerian, or an arbitrary Lagrangian-Eulerian (ALE) mesh that is based on a linear finite-element formulation and may have arbitrary connectivity among the elements.

Many issues need to be addressed to implement any intrusive optimization algorithm for Shock physics codes, including transient analysis, non-smooth behavior, explicit solution procedures, material addition and deletion mechanisms. Similar issues exist as in fire simulation.

3.2.7 Electrical Simulation

A substantial number of electrical simulations are conducted at Sandia and a common problem is to match experimental data from a network of circuits to these simulations. Capabilities to solve very large circuit problems are currently being developed. This effort will support analysis of circuit phenomena at a variety of abstraction levels, including device-level, analog signals, digital signals, and mixed signals. Although electrical simulation should be smooth, old device models have been known to use limiter processes that are non-differentiable. Typically, large-scale electrical simulation consists of millions of devices each of which can host at least one design parameter. Therefore, electrical simulation is a reasonably good SAND optimization candidate provided the device model issues can be resolved and also provided optimization methods to handle transient models efficiently can be developed. Algorithms to handle transient processes are available, but they are memory and storage intensive since they require a large number of design variables and large number of time steps.

The solution approach generates nonlinear systems of DAEs and uses Newton's method to solve the resulting nonlinear equations. Thus Xyce generates a Jacobian similar to those required by PDE-based simulations and theoretically adjoint sensitivities can be calculated. Similarly to compressible fluid, additional sensitivity development is underway to develop higher optimization levels capabilities.

3.2.8 Geophysics

Geophysics has long been the source of large inversion problems that are solved to identify materials and related properties and to detect targets. Each of these problems deals with large number of design/inversion parameters. They are often solved in the frequency domain thereby avoiding the issues related to transient phenomena. Considerable research has been conducted at Sandia to solve inversion problems and, although the solution procedures are not entirely along the same lines described in this report, these codes do use Gauss-Newton methods and conjugate gradient solvers [81] [82].

Seismic inversion, structural inversion, and source inversion are all important problems that are amenable to the highest level SAND methods. In fact, the state-of-the-art SAND methods have been applied to a seismic inversion problem where 2.1 million inversion parameters were used for a transient simulation [3]. These problems are implicit, they use exact Jacobians, and can be solved in either steady-state or transient mode. In addition, they are single physics and their solutions are smooth.

3.2.9 Observations & Strategies

Several conclusions have been drawn from our review of the Sandia PDE environment:

1. A wide range of physics are simulated by a variety of methods incorporating both linear and nonlinear solvers. An increasing number of complex design, control, and inversion problems, involving a large number of design/control/inversion parameters demand efficient optimization methods.
2. Most of the critical Sandia simulation codes run in parallel and thus new optimization algorithms need to be designed with large-scale parallelism in mind.
3. The predominant programming language is C++; we strongly support the continued development of frameworks, algorithms, and tools in C++.
4. High-fidelity, multi-physics simulations are crucial to solve Sandia's science and engineering problems. The initial step to create a multi-physics capability is to use explicit solvers. However, as discussed above, this creates difficulties for a SAND optimization method. Thus the use of implicit methods needs to be explored.
5. Transient simulation dominates the problem space at Sandia and SAND optimization methods for transient problems need to be investigated.
6. Individual forward simulators are being consolidated into two principal frameworks, namely, SIERRA and Nevada. Optimization methods and interfaces need to be considered as part of these frameworks.

Although implementing PDECO requires a custom design and an individual approach to each simulation code, it has been our goal to develop methods, algorithms, and frameworks that can be leveraged in other PDE simulation codes. Assuming that sensitivity information is available from the simulation codes and the simulation code conforms to the SAND assumptions, we have developed a framework called rSQP++ that can be interfaced with most codes. The strength of this state-of-art object oriented code is that algorithms can be modified very quickly to adapt to the needs of the optimization problem. In addition, we have interfaced this code to a PDE prototyping code (Sundance) so that algorithms can be easily tested for a range of PDE systems. The next few chapters are dedicated to describing the rSQP++ framework, Sundance, and a full-space SQP method that relies on solving quadratic programs.

Chapter 4

rSQP++ Framework

Described herein is a new object-oriented (OO) framework for building successive quadratic programming Algorithms, called rSQP++, currently implemented in C++. The goals for rSQP++ are quite lofty. The rSQP++ framework is designed to incorporate many different SQP algorithms and to allow external configuration of specialized linear algebra objects such as vectors, matrices and linear solvers. Data-structure independence has been recognized as an important feature missing in current optimization software [123]. In addition, it is possible for an advanced user to modify the SQP algorithms to meet other specialized needs without having to touch any of the default source code within the rSQP++ framework.

Successive quadratic programming (SQP) methods are attractive mainly because they generally require the fewest number of function and gradient evaluations to solve a problem as compared to other optimization methods [105]. Another attractive property of SQP methods is that the structure of the underlying NLP can be exploited more effectively than other methods [118]. A variation of SQP, known as reduced-space SQP (rSQP), works well for NLPs where there are few degrees of freedom (dof) (see Section 4.1.1) and many constraints. Quasi-Newton methods for approximating the reduced Hessian of the Lagrangian are also very efficient for NLPs with few dof. Another advantage of rSQP is that the decomposition used for the equality constraints only requires solves with a basis of the Jacobian (and possibly its transpose) of the constraints (see Section 4.1.3).

4.1 Mathematical Background for SQP

4.1.1 Nonlinear Program (NLP) Formulation

The SQP algorithms implemented with rSQP++ solve NLPs in the standard form:

$$\begin{aligned} \min \quad & f(x) && (4.1.1)\\ \text{s.t.} \quad & c(x) = 0 && (4.1.2)\\ & x_L \leq x \leq x_U && (4.1.3) \end{aligned}$$

where:

$$\begin{aligned} & x, x_L, x_U \in \mathcal{X} \\ & f(x) : \mathcal{X} \rightarrow \mathbf{R} \\ & c(x) : \mathcal{X} \rightarrow \mathcal{C} \\ & \mathcal{X} \subseteq \mathbf{R}^n \\ & \mathcal{C} \subseteq \mathbf{R}^m. \end{aligned}$$

Above, we have been very careful to define vector spaces for the relevant vectors and nonlinear operators. In general, only vectors from the same vector space are compatible and can participate in linear algebra operations. Mathematically, the only requirement for the compatibility of real-valued vector spaces should be that the dimensions match up and that the same inner products are used. However, having the same dimensionality will not be sufficient to allow the compatibility of vectors from different vector spaces in the implementation. The vector spaces become very important later when the NLP interfaces and the implementation of rSQP++ is discussed in more detail (see Section 4.2.3.2).

We assume that the operators $f(x)$ and $c_j(x)$ for $j = 1 \ldots m$ in (4.1.1)–(4.1.2) are nonlinear functions with at least second-order continuous derivatives. The rSQP algorithms described later only require first-order derivative information for $f(x)$ and $c_j(x)$ in the form of a vector $\nabla f(x)$ and a matrix $\nabla c(x)$ respectively. The inequality constraints in (4.1.3) may have lower bounds equal to $-\infty$ and/or upper bounds equal to $+\infty$. The absences of some of these bounds can be exploited by many SQP algorithms.

It is very desirable for the functions $f(x)$ and $c(x)$ to at least be defined (i.e. no NaN or Inf return values) everywhere in the set defined by the relaxed variable bounds $x_L - \delta \leq x \leq x_U + \delta$. Here, δ (see the method `max_var_bounds_viol()` in the *NLP* interface in Section 4.2.3.2) is a relaxation (i.e. wiggle room) that the user can set to allow the optimization algorithm to compute $f(x)$, $c(x)$ and $h(x)$ outside the strict variable bounds $x_L \leq x \leq x_U$ in order to compute finite differences and the like. The SQP algorithms will never evaluate $f(x)$ and $c(x)$ outside this relaxed region. This is an imporant issue to consider when developing the model for the NLP.

The Lagrangian function $L(\lambda, \nu_L, \nu_U)$ (and the Lagrange multipliers (λ, ν_L, ν_U)) and its gradient and Hessian for this NLP are

$$L(x, \lambda, \nu_L, \nu_U) \;=\; \left\{ f(x) + \lambda^T c(x) + \nu_L^T (x_L - x) + \nu_U^T (x - x_U) \right\} \in \mathbf{R} \tag{4.1.4}$$

$$\nabla_x L(x, \lambda, \nu) = \{ \nabla f(x) + \nabla c(x) \lambda + \nu \} \in \mathcal{X} \tag{4.1.5}$$

$$\nabla^2_{xx} L(x, \lambda) = \left\{ \nabla^2 f(x) + \sum_{j=1}^{m} \lambda_{(j)} \nabla^2 c_j(x) \right\} \in \mathcal{X}|\mathcal{X} \tag{4.1.6}$$

where:

$$\begin{aligned}
&\nabla f(x) : \mathcal{X} \rightarrow \mathcal{X} \\
&\nabla c(x) = \left[\begin{array}{cccc} \nabla c_1(x) & \nabla c_2(x) & \ldots & \nabla c_m(x) \end{array} \right] : \mathcal{X} \rightarrow \mathcal{X}|\mathcal{C} \\
&\nabla^2 c_j(x) : \mathcal{X} \rightarrow \mathcal{X}|\mathcal{X} \text{ , for } j = 1 \ldots m \\
&\lambda \in \mathcal{C} \\
&\nu \equiv \nu_U - \nu_L \in \mathcal{X}.
\end{aligned}$$

Above, we use the notation $\lambda_{(j)}$ with the subscript in parentheses to denote the j^{th} component of the vector and to differentiate this from a simple math accent. Also, $\nabla c(x) : \mathcal{X} \rightarrow \mathcal{X}|\mathcal{C}$ is used to denote a nonlinear operator (the gradient of the equality constraints $\nabla c(x)$ in this case) that maps from the vector space $\mathcal{X}$ to a matrix space $\mathcal{X}|\mathcal{C}$ where the columns and rows in this matrix space lie in the vector spaces $\mathcal{X}$ and $\mathcal{C}$ respectively. The returned matrix object $A = \nabla c \in \mathcal{X}|\mathcal{C}$ defines a linear operator where $q = Ap$ maps vectors from $p \in \mathcal{C}$ to $q \in \mathcal{X}$. The transposed matrix object A^T defines a linear operator where $q = A^T p$ maps vectors from $p \in \mathcal{X}$ to $q \in \mathcal{C}$.

Note how the vector and matrix spaces in the above expressions match up. For example, the vectors and matrices in (4.1.5) can be replaced by their vector and matrix spaces as

$$\{ \nabla f(x) + \nabla c(x) \lambda + \nu \} \Rightarrow \{ \mathcal{X} + (\mathcal{X}|\mathcal{C})\mathcal{C} + \mathcal{X} \} \Rightarrow \mathcal{X}.$$

The compatibility of vectors and matrices in linear algebra operations is determined by the compatibility of the associated vector spaces. At all times, we must know to which vector or matrix space a linear algebra quantity belongs.

Given the definition of the Lagrangian and its derivatives in (4.1.4)–(4.1.6), the first- and second-order necessary KKT optimality conditions [80] for a solution $(x^*, \lambda^*, \nu_L^*, \nu_U^*)$ to (4.1.1)–(4.1.3) are given in (4.1.7)–(4.1.13). There are four different categories of optimality conditions shown here: linear dependence of gradients (4.1.7), feasibility (4.1.8)–(4.1.9), non-negativity of lagrange multipliers for inequalities (4.1.10), complementarity (4.1.11)–(4.1.12), and curvature (4.1.13).

$$\nabla_x L(x^*, \lambda^*, \nu^*) = \nabla f(x^*) + \nabla c(x^*)\lambda^* + \nu^* = 0 \tag{4.1.7}$$

$$c(x^*) = 0 \tag{4.1.8}$$

$$x_L \leq x^* \leq x_U \tag{4.1.9}$$

$$(\nu_L)^*, (\nu_U)^* \geq 0 \tag{4.1.10}$$

$$(\nu_L)^*_{(i)}((x_L)_{(i)} - (x^*)_{(i)}) = 0, \quad \text{for } i = 1 \ldots n \tag{4.1.11}$$

$$(\nu_U)^*_{(i)}((x^*)_{(i)} - (x_U)_{(i)}) = 0, \quad \text{for } i = 1 \ldots n \tag{4.1.12}$$

$$d^T \, \nabla^2_{xx} L(x^*, \lambda^*) \, d \geq 0, \quad \text{for all feasible directions } d \, \in \, \mathcal{X}. \tag{4.1.13}$$

Sufficient conditions for optimality require that stronger assumptions be made about the NLP (e.g. constraint qualification on $c(x)$ and perhaps conditions on third-order curvature in case 0 is obtained in (4.1.13)).

To solve a NLP, a SQP algorithm must first be supplied an initial guess for the unknown variables x_0 and in some cases also the Lagrange multipliers λ_0 and ν_0. The optimization algorithms implemented in rSQP++ generally require that x_0 satisfy the variable bounds in (4.1.3), and if not, the elements of x_0 are forced in bounds. The matrix $\nabla c(x)$ is abstracted behind a set of object-oriented interfaces. The rSQP algorithm only needs to perform matrix-vector multiplication with $\nabla c(x)$ and solve for a square, nonsingular basis of $\nabla c(x)$ through a *BasisSystem* interface. The implementation of $\nabla c(x)$ is completely abstracted away from the optimization algorithm. A simpler interface to NLPs has also been developed where the matrix $\nabla c(x)$ is never represented even implicitly (i.e. no matrix-vector products) and only specific quantities are supplied to the rSQP algorithm (see the "Tailored Approach" in [104] and the "direct sensitivity" NLP interface on page 69).

4.1.2 Successive Quadratic Programming (SQP)

A popular class of methods for solving NLPs is successive quadratic programming (SQP) [26]. An SQP method is equivalent, in many cases, to applying Newton's method to solve the optimality conditions represented by (4.1.7)–(4.1.8). At each Newton iteration k for (4.1.7)–(4.1.8), the linear subproblem (also known as the KKT system) takes the form

$$\begin{bmatrix} W & A \\ A^T & \end{bmatrix} \begin{bmatrix} d \\ d_\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_x L \\ c \end{bmatrix} \qquad (4.1.14)$$

where:

$$\begin{aligned} & d = x_{k+1} - x_k \ \in \ \mathcal{X} \\ & d_\lambda = \lambda_{k+1} - \lambda_k \ \in \ \mathcal{C} \\ & W \approx \nabla^2_{xx} L(x_k, \lambda_k) \ \in \ \mathcal{X}|\mathcal{X} \\ & A = \nabla c(x_k) \ \in \ \mathcal{X}|\mathcal{C} \\ & c = c(x_k) \ \in \ \mathcal{C}. \end{aligned}$$

The Newton matrix in (4.1.14) is known as the KKT matrix. By substituting $d_\lambda = \lambda_{k+1} - \lambda_k$ into (4.1.14) and simplifying, this linear system becomes equivalent to the optimality conditions of the following QP

$$\begin{aligned} \text{min} \quad & g^T d + \tfrac{1}{2} d^T W d && (4.1.15) \\ \text{s.t.} \quad & A^T d + c = 0 && (4.1.16) \end{aligned}$$

where:

$$g = \nabla f(x_k) \ \in \ \mathcal{X}.$$

The advantage of the QP formulation over the Newton linear-system formulation is that inequality constraints can be directly added to the QP and a relaxation can be defined which yields the following QP

$$\begin{aligned} \text{min} \quad & g^T d + \tfrac{1}{2} d^T W d + M(\eta) && (4.1.17) \\ \text{s.t.} \quad & A^T d + (1 - \eta) c = 0 && (4.1.18) \\ & x_L - x_k \leq d \leq x_U - x_k && (4.1.19) \\ & 0 \leq \eta \leq 1 && (4.1.20) \end{aligned}$$

where:

$$M(\eta) \in \mathbf{R} \to \mathbf{R}.$$

Near the solution of the NLP, the set of active constraints for (4.1.17)–(4.1.20) will be the same as the optimal active-set for the NLP in (4.1.1)–(4.1.3) [85, Theorem 18.1].

The relaxation of the QP shown in (4.1.17)–(4.1.20) is only one form of a relaxation but has the essential properties. Note that the solution $\eta = 1$ and $d = 0$ is always feasible by construction. The penalty function $M(\eta)$ is either a linear or quadratic term where if $\frac{\partial M(\eta)}{\partial \eta}|_{\eta=0}$ is sufficiently large then an unrelaxed solution (i.e. $\eta = 0$) will be obtained if a feasible region for the original QP exists. For example, the penalty term may take a form such as $M(\eta) = (\tilde{M})\eta$ or $M(\eta) = (\tilde{M})(\eta + {}^1\!/_2\eta^2)$ where $\tilde{M}$ is a large constant often called “big M.”

Once a new estimate of the solution (x_{k+1}, λ_{k+1}, ν_{k+1}) is computed, the error in the optimality conditions (4.1.7)–(4.1.9) is checked. If these KKT errors are within some specified tolerance, the algorithm is terminated with the optimal solution. If the KKT error is too large, the NLP functions and gradients are then computed at the new point x_{k+1} and another QP subproblem (4.1.17)–(4.1.20) is solved which generates another step d and so on. This algorithm is continued until a solution is found or the algorithm runs into trouble (there can be many causes for algorithm failure), or it is prematurely terminated because it is taking too long (i.e. maximum number of iterations or runtime is exceeded).

The iterates generated from $x_{k+1} = x_k + d$ are generally only guaranteed to converge to a local solution to the first-order KKT conditions when close to the solution. Therefore, globalization methods are used to insure (given a few, sometimes strong, assumptions are satisfied) the SQP algorithm will converge to a local solution from remote starting points. One popular class of globalization methods are linesearch methods. In a linesearch method, once the step d is computed from the QP subproblem, a linesearch procedure is used to find a step length α such that $x_{k+1} = x_k + \alpha d$ gives sufficient reduction in the value of a merit function $\phi(x_{k+1}) < \phi(x_k)$. A merit function is used to balance a trade-off between minimizing the objective function $f(x)$ and reducing the error in the constraints $c(x)$. A commonly used merit function is the ℓ_1 defined by

$$\phi_{\ell_1}(x) = f(x) + \mu||c(x)||_1 \tag{4.1.21}$$

where μ is a penalty parameter that is adjusted to insure descent along the SQP step $x_k + \alpha d$ for $\alpha > 0$. An alternative linesearch based on a “filter” has also been implemented which generally performs better and does not require the maintenance of a penalty parameter μ [122] . Other globalization methods such as trust region (using a merit function or the filter) can also be applied to SQP.

Because SQP is essentially equivalent to applying Newton's method to the optimality conditions, it can be shown to be quadratically convergent near the solution of the NLP [84]. It is this fast rate of convergence that makes SQP the method of choice for many applications. However, there are many theoretical and practical details that need to be considered. One difficulty is that in order to achieve quadratic convergence the exact Hessian of the Lagrangian W is needed, which requires exact second-order information $\nabla^2 f(x)$ and $\nabla^2 c_j(x)$, $j = 1 \ldots m$. For many NLP applications, second derivatives are not readily available and it is too expensive and/or inaccurate to compute them numerically. Other difficulties with SQP include how to deal with an indefinite Hessian W. Also, for large problems, the full QP subproblem in (4.1.17)–(4.1.20) can be extremely expensive to solve directly. These and other difficulties have motivated the research of large-scale decomposition methods for SQP. One class of these methods is reduced-space (or reduced-Hessian) SQP, or rSQP for short.

4.1.3 Reduced-Space Successive Quadratic Programming (rSQP)

In a rSQP method, the full-space QP subproblem (4.1.17)–(4.1.20) is decomposed into two smaller subproblems that, in many cases, are easier to solve. To see how this is done, first a null-space decomposition [85, Section 18.3] is computed for some linearly independent set of the linearized equality constraints $A_d \in \mathcal{X}|\mathcal{C}_d$ where $c_d(x) \in \mathcal{C}_d \in \mathbf{R}^{\,r}$ are the decomposed and $c_u(x) \in \mathcal{C}_u \in \mathbf{R}^{\,(m-r)}$ are the undecomposed equality constraints and

$$c(x) = \begin{bmatrix} c_d(x) \\ c_u(x) \end{bmatrix} \in \mathcal{C}_d \times \mathcal{C}_u \implies \nabla c(x_k) = \begin{bmatrix} \nabla c_d(x_k) & \nabla c_u(x_k) \end{bmatrix} = \begin{bmatrix} A_d & A_u \end{bmatrix} \in \mathcal{X}|(\mathcal{C}_d \times \mathcal{C}_u). \tag{4.1.22}$$

Above, the vector space $\mathcal{C} = \mathcal{C}_d \times \mathcal{C}_u$ denotes a concatenated vector space (also known as a product of vector spaces) with a dimension which is the sum of the constituent vector spaces $|\mathcal{C}| = |\mathcal{C}_d| + |\mathcal{C}_u| = r + (m - r) = m$. This decomposition is defined by a null-space matrix Z and a matrix Y with the following properties:

$$\begin{array}{ll} Z \in \mathcal{X}|\mathcal{Z} & \text{s.t. } (A_d)^T Z = 0 \\ Y \in \mathcal{X}|\mathcal{Y} & \text{s.t. } \begin{bmatrix} Y & Z \end{bmatrix} \text{ is nonsingular} \end{array} \tag{4.1.23}$$

where:

$$\begin{aligned}\mathcal{Z} &\subseteq \mathbf{R}^{(n-r)} \\ \mathcal{Y} &\subseteq \mathbf{R}^{r}.\end{aligned}$$

It is important to distinguish the spaces $\mathcal{Z}$ and $\mathcal{Y}$ from the the matrices Z and Y. The null-space matrix $Z \in \mathcal{X}|\mathcal{Z}$ is a linear operator that maps vectors from the space $u \in \mathcal{Z}$ to vectors in the space of the unknowns $v = Zu \in \mathcal{X}$. The matrix $Y \in \mathcal{X}|\mathcal{Y}$ is a linear operator that maps vectors from the space $u \in \mathcal{Y}$ to vectors in the space of the unknowns $v = Yu \in \mathcal{X}$.

In many presentations of reduced-space SQP, the matrix Y is referred to as the “range-space” matrix since several popular choices of this matrix form a basis for the range space of A_d. However, note that the matrix Y need not be a true basis matrix for the range space of A_d in order to satisfy the nonsingularity property in (4.1.23). For this reason, here the matrix Y will be referred to as the “quasi-range-space” matrix to make this distinction.

By using (4.1.23), the search direction d can be broken down into $d = (1-\eta)Yp_y + Zp_z$, where $p_y \in \mathcal{Y}$ and $p_z \in \mathcal{Z}$ are the known as the quasi-normal (or quasi-range space) and tangential (or null space) steps respectively. By substituting $d = (1-\eta)Yp_y + Zp_z$ into (4.1.17)–(4.1.20) we obtain the quasi-normal (4.1.24) and targential (4.1.25)–(4.1.27) subproblems. In (4.1.25), $\zeta \leq 1$ is a damping parameter which can be used to insure descent of the merit function $\phi(x_{k+1} + \alpha d)$.

Quasi-Normal (Quasi-Range-Space) Subproblem

$$p_y = -R^{-1} c_d \in \mathcal{Y} \tag{4.1.24}$$

where: $R \equiv [(A_d)^T Y] \in \mathcal{C}_d|\mathcal{Y}$ (nonsingular via (4.1.23)).

Tangential (Range-Space) Subproblem (Relaxed)

$$\begin{aligned} \text{min} \quad & (g^r + \zeta w)^T p_z + \tfrac{1}{2} p_z^T [Z^T W Z] p_z + M(\eta) && (4.1.25) \\ \text{s.t.} \quad & U_z p_z + (1-\eta) u = 0 && (4.1.26) \\ & b_L \leq Z p_z - (Y p_y)\eta \leq b_U && (4.1.27) \end{aligned}$$

where:

$$
\begin{aligned}
&g^r \equiv Z^T g \,\in\, \mathcal{Z} \\
&w \equiv Z^T W Y p_y \,\in\, \mathcal{Z} \\
&\zeta \,\in\, \mathbf{R} \\
&U_z \equiv [(A_u)^T Z] \,\in\, \mathcal{C}_u | \mathcal{Z} \\
&U_y \equiv [(A_u)^T Y] \,\in\, \mathcal{C}_u | \mathcal{Y} \\
&u \equiv U_y p_y + c_u \,\in\, \mathcal{C}_u \\
&b_L \equiv x_L - x_k - Y p_y \,\in\, \mathcal{X} \\
&b_U \equiv x_U - x_k - Y p_y \,\in\, \mathcal{X}.
\end{aligned}
$$

By using this decomposition, the Lagrange multipliers λ_d for the decomposed equality constraints ($(A_d)^T d + c_d = 0$) do not need to be computed in order to produce steps $d = (1-\eta) Y p_y + Z p_z$. However, these multipliers can be used to determine the penalty parameter μ for the merit function [85, page 544] or to compute the Lagrangian function. Alternatively, a multiplier-free method for computing μ has been developed and tested with good results [104]. In any case, it is useful to compute these multipliers at the solution of the NLP since they give the sensitivity of the objective function to those constraints [80, page 436]. An expression for computing λ_d can be derived by applying (4.1.23) to $Y^T \nabla L(x, \lambda, \nu)$ to yield

$$
\lambda_d = -R^{-T} \left(Y^T (g + \nu) + (U_y)^T \lambda_u \right) \,\in\, \mathcal{C}_d. \tag{4.1.28}
$$

There are many details that need to be worked out in order to implement a rSQP algorithm and there are opportunities for a lot of variability. Some of the more significant decisions that need to be made are: how to compute the null-space decomposition that defines the matrices Z, Y, R, U_z and U_y, and how the reduced Hessian $Z^T W Z$ and the cross term w in (4.1.25) are calculated (or approximated).

There are several different ways to compute decomposition matrices Z and Y that satisfy (4.1.23) [105]. For small-scale rSQP, an orthonormal Z and Y ($Z^T Y = 0$, $Z^T Z = I$, $Y^T Y = I$) can be computed using a QR factorization of A_d [84]. This decomposition gives rise to rSQP algorithms with many desirable properties. However, using a QR factorization when A_d is of very large dimension is prohibitively expensive. Therefore, other choices for Z and Y have been investigated that are more appropriate for large-scale rSQP. Methods that are more computationally tractable are based on a variable-reduction decomposition [105]. In a variable-reduction decomposition, the variables are partitioned into dependent x_D and independent x_I sets

$$x_D \in \mathcal{X}_D \tag{4.1.29}$$

$$x_I \in \mathcal{X}_I \tag{4.1.30}$$

$$x = \begin{bmatrix} x_D \\ x_I \end{bmatrix} \in \mathcal{X}_D \times \mathcal{X}_I \tag{4.1.31}$$

$$\tag{4.1.32}$$

where:

$$\mathcal{X}_D \subseteq \mathbf{R}^r$$
$$\mathcal{X}_I \subseteq \mathbf{R}^{n-r}$$

such that the Jacobian of the constraints A^T is partitioned as shown in (4.1.33) where C is a square, nonsingular matrix known as the basis matrix. The variables x_D and x_I are also called state and design (or controls) variables [20] in some applications or basic and nonbasic variables [78] in others. What is important about this partitioning of variables is that the x_D variables define the selection of the basis matrix C, nothing more. Some types of optimization algorithms give more significance to this partitioning of variables (for example, in MINOS [78] the basic variables are also variables that are not at an active bound) however no extra significance can be attributed here.

This basis selection is used to define a variable-reduction null-space matrix Z in (4.1.34) which also determines U_z in (4.1.35).

Variable-Reduction Partitioning

$$A^T = \begin{bmatrix} (A_d)^T \\ (A_u)^T \end{bmatrix} = \begin{bmatrix} C & N \\ E & F \end{bmatrix} \tag{4.1.33}$$

where:

$$\begin{aligned} C &\in \mathcal{C}_d|\mathcal{X}_D \qquad \text{(nonsingular)} \\ N &\in \mathcal{C}_d|\mathcal{X}_I \\ E &\in \mathcal{C}_u|\mathcal{X}_D \\ F &\in \mathcal{C}_u|\mathcal{X}_I. \end{aligned}$$

Variable-Reduction Null-Space Matrix

$$Z \equiv \begin{bmatrix} -C^{-1}N \\ I \end{bmatrix} \tag{4.1.34}$$

$$U_z = F - E\,C^{-1}N \tag{4.1.35}$$

There are many choices for the quasi-range-space matrix Y that satisfy (4.1.23). Two relatively computationally inexpensive choices are the coordinate and orthogonal decompositions shown below.

Coordinate Variable-Reduction Null-Space Decomposition

$$Y \equiv \begin{bmatrix} I \\ 0 \end{bmatrix} \tag{4.1.36}$$

$$R = C \tag{4.1.37}$$

$$U_y = E \tag{4.1.38}$$

Orthogonal Variable-Reduction Null-Space Decomposition

$$Y \equiv \begin{bmatrix} I \\ N^T C^{-T} \end{bmatrix} \tag{4.1.39}$$

$$R = C(I + C^{-1}NN^TC^{-T}) \tag{4.1.40}$$

$$U_y = E - FN^TC^{-T} \tag{4.1.41}$$

The orthogonal decomposition ($Z^TY = 0$, $Z^TZ \neq I$, $Y^TY \neq I$) defined in (4.1.34)–(4.1.35) and (4.1.39)–(4.1.41) is more numerically stable than the coordinate decomposition and has other desirable properties in the context of rSQP [105]. However, the amount of dense linear algebra required to compute the factorizations needed to solve for linear systems with R in (4.1.40) is $O((n-r)^2 r)$ floating point operations (flops) which can dominate the cost of the algorithm for larger $(n-r)$. Therefore, for larger $(n-r)$, the coordinate decomposition ($Z^TY \neq 0$, $Z^TZ \neq I$, $Y^TY \neq I$) defined in (4.1.34)–(4.1.35) and (4.1.36)–(4.1.38) is preferred because it is cheaper but the downside is that it is also more susceptible to problems associated with a poor selection of dependent variables. Ill-conditioning in the basis matrix C can result with greatly degraded performance and even lead to failure of an rSQP algorithm. See the option `range_space_matrix` in Section 4.3.1.1.

Another important decision is how to compute the reduced Hessian Z^TWZ. For many NLPs, second-derivative information is not available to compute the Hessian of the Lagrangian W directly. In these cases, first-derivative information can be used to approximate $B \approx Z^TWZ$ using quasi-Newton methods (e.g. BFGS) [84]. When $(n - r)$ is small, B is small and cheap to update. Under the proper conditions the resulting quasi-Newton rSQP algorithm has a superlinear rate of local convergence (even using w = 0 in (4.1.25)) [15]. Even when $(n - r)$ is large, limited-memory quasi-Newton methods can still be used, but the price one pays is in only being able to achieve a linear rate of convergence (with a small rate constant hopefully). For some application areas, good approximations of the Hessian W are available and may have specialized properties (i.e. structure) that makes computing the exact reduced Hessian $B = Z^TWZ$ computationally feasible (i.e. see NMPC in [10]). See the options `exact_reduced_hessian` and `quasi_newton` in Section 4.3.1.1.

In addition to variations that affect the convergence behavior of the rSQP algorithm, such as null-space decompositions, approximations used for the reduced Hessian and many different types of merit functions and globalization methods, there are also many different implementation options. For example, linear systems such as (4.1.24) can be solved using direct or iterative solvers and the reduced QP subproblem in (4.1.25)–(4.1.27) can be solved using a variety of methods (active set vs. interior point) and software [106].

Figure 4.1 summarizes five different categories of algorithmic options for a rSQP algorithm, many of which were described above. This set of categories and the options in each category is by no means complete and may other options have been developed and will be developed in the future. In general, any option can be selected independently from each category and form a valid algorithm with unique properties. An exception is that merit functions are not used by the Filter line-search and trust-region globalization methods so it makes no sense to select a merit function when using a Filter method. While some permutations of options are not reasonable (i.e finite-difference w with an exact reduced Hessian B), many permutations are. Just this set of options can produce 480 distinctly different algorithms that may perform very differently on any particular NLP.

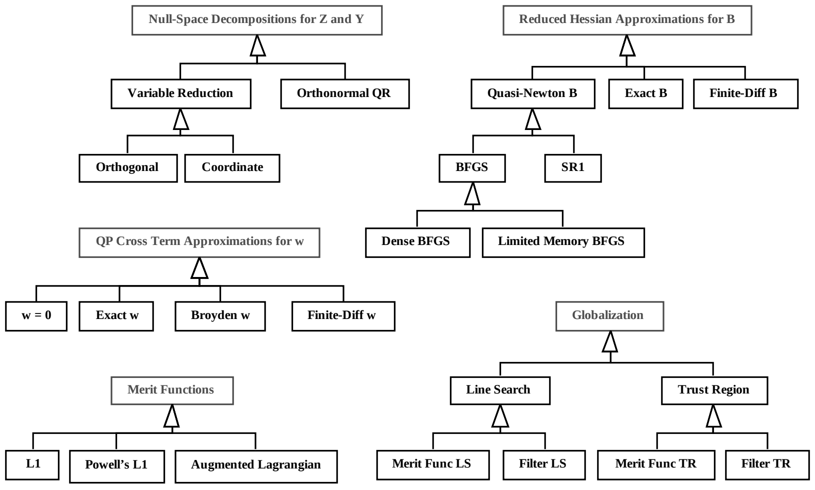


Figure 4.1. UML analysis class diagram : Different algorithmic options for rSQP

4.1.4 General Inequalities and Slack Variables

Up to this point, only simple variable bounds in (4.1.3) have been considered and the SQP/rSQP algorithms have been presented in this context. However, the actual underlying NLP may include general inequalities and take the form

$$
\begin{aligned}
\min \quad & \breve{f}(\breve{x}) && (4.1.42)\\
\text{s.t.} \quad & \breve{c}(\breve{x}) = 0 && (4.1.43)\\
& \breve{h}_L \le \breve{h}(\breve{x}) \le \breve{h}_U && (4.1.44)\\
& \breve{x}_L \le \breve{x} \le \breve{x}_U && (4.1.45)
\end{aligned}
$$

where:

$$
\begin{aligned}
&\breve{x}, \breve{x}_L, \breve{x}_U \ \in \ \breve{\mathcal{X}}\\
&\breve{f}(x) : \ \breve{\mathcal{X}} \rightarrow \mathbf{R}\\
&\breve{c}(x) : \ \breve{\mathcal{X}} \rightarrow \breve{\mathcal{C}}\\
&\breve{h}(x) : \ \breve{\mathcal{X}} \rightarrow \breve{\mathcal{H}}
\end{aligned}
$$

$$
\begin{aligned}
\breve{h}_L, \breve{h}_L &\in \breve{\mathcal{H}} \\
\breve{\mathcal{X}} &\in \mathbf{R}^{\breve{n}} \\
\breve{\mathcal{C}} &\in \mathbf{R}^{\breve{m}} \\
\breve{\mathcal{H}} &\in \mathbf{R}^{\breve{m}_I}.
\end{aligned}
$$

NLPs with general inequalities are converted into the standard form by the addition of slack variables $\breve{s}$ (see (4.1.49)). After the addition of the slack variables, the concatenated variables and constraints are then permuted (using permutation matrices Q_x and Q_c) into the ordering of (4.1.1)–(4.1.3). The exact mapping from (4.1.42)–(4.1.45) to (4.1.1)–(4.1.3) is given below

$$x = Q_x \begin{bmatrix} \breve{x} \\ \breve{s} \end{bmatrix} \tag{4.1.46}$$

$$x_L = Q_x \begin{bmatrix} \breve{x}^L \\ \breve{h}^L \end{bmatrix} \tag{4.1.47}$$

$$x_U = Q_x \begin{bmatrix} \breve{x}_u \\ \breve{h}_u \end{bmatrix} \tag{4.1.48}$$

$$c(x) = Q_c \begin{bmatrix} \breve{c}(\breve{x}) \\ \breve{h}(\breve{x}) - \breve{s} \end{bmatrix} \tag{4.1.49}$$

Here we consider the implications of the above transformation in the context of rSQP algorithms.

Note if $Q_x = I$ and $Q_c = I$ that the matrix ∇c takes the form:

$$\nabla c = \begin{bmatrix} \nabla \breve{c} & \nabla \breve{h} \\ & -I \end{bmatrix} \tag{4.1.50}$$

One question to ask is how the Lagrange multipliers for the original constraints can be extracted from the optimal solution (x, λ, ν) that satisfies the optimality conditions in (4.1.7)–(4.1.13)? First, consider the linear dependence of gradients optimality condition for the NLP formulation in (4.1.42)–(4.1.45)

$$\nabla_{\breve{x}} \breve{L}(\breve{x}^*, \breve{\lambda}^*, \breve{\lambda}_I^{\,*}, \breve{\nu}^*) = \nabla \breve{f}(\breve{x}^*) + \nabla \breve{c}(\breve{x}^*)\breve{\lambda}^* + \nabla \breve{h}(\breve{x}^*)\breve{\lambda}_I^{\,*} + \breve{\nu}^* = 0. \tag{4.1.51}$$

To see how the Lagrange multiples λ^* and ν^* can be used to compute $\breve{\lambda}^*$, $\breve{\lambda}_I{}^*$ and $\breve{\nu}^*$ one simply has to substitute (4.1.46) and (4.1.49) with $Q_x = I$ and $Q_c = I$ into (4.1.7) and expand as follows

$$\begin{aligned}
\nabla_x L(x, \lambda, \nu) &= \nabla f + \nabla c \lambda + \nu \\
&= \begin{bmatrix} \nabla \breve{f} \\ 0 \end{bmatrix} + \begin{bmatrix} \nabla \breve{c} & \nabla \breve{h} \\ & -I \end{bmatrix} \begin{bmatrix} \lambda_{\breve{c}} \\ \lambda_{\breve{h}} \end{bmatrix} + \begin{bmatrix} \nu_{\breve{x}} \\ \nu_{\breve{s}} \end{bmatrix} \\
&= \begin{bmatrix} \nabla \breve{f} + \nabla \breve{c} \lambda_{\breve{c}} + \nabla \breve{h} \lambda_{\breve{h}} + \nu_{\breve{x}} \\ -\lambda_{\breve{h}} + \nu_{\breve{s}} \end{bmatrix}. \qquad (4.1.52)
\end{aligned}$$

By comparing (4.1.51) and (4.1.52) it is clear that the mapping is $\breve{\lambda} = \lambda_{\breve{c}}$, $\breve{\lambda}_I = \lambda_{\breve{h}} = \nu_{\breve{s}}$ and $\breve{\nu} = \nu_{\breve{x}}$. For arbitrary Q_x and Q_c it is also easy to perform the mapping of the solution. What is interesting about (4.1.52) is that it says that for general inequalities $\breve{h}_j(\breve{x})$ that are not active at the solution (i.e. $(\nu_{\breve{s}})_{(j)} = 0$), the Lagrange multiplier for the converted equality constraint $(\lambda_{\breve{h}})_{(j)}$ will be zero. This means that these converted inequalities can be eliminated from the problem and not impact the solution, which is expected. Zero multiplier values means that constraints will not impact the optimality conditions or the Hessian of the Lagrangian.

The basis selection shown in (4.1.22) and (4.1.31) is determined by the permutation matrices Q_x and Q_c and these permutation matrices can be partitioned as follows:

$$Q_x = \begin{bmatrix} Q_{xD} \\ Q_{xI} \end{bmatrix} \qquad (4.1.53)$$

$$Q_c = \begin{bmatrix} Q_{cD} \\ Q_{cU} \end{bmatrix}. \qquad (4.1.54)$$

A valid basis selection can always be determined by simply including all of the slacks $\breve{s}$ in the full basis and then finding a sub-basis for $\nabla \breve{c}$. To show how this can be done, suppose that $\nabla \breve{c}$ is full rank and the permutation matrix $(\breve{Q}_x)^T = \begin{bmatrix} (\breve{Q}_{xD})^T & (\breve{Q}_{xI})^T \end{bmatrix}$ selects a basis $\breve{C} = (\nabla \breve{c})^T (\breve{Q}_{xD})^T$. Then the following basis selection for the transformed NLP (with $Q_c = I$) could always be used regardless of the properties or implementation of $\nabla \breve{h}$

$$Q_x = \begin{bmatrix} \breve{Q}_{xD} & & \\ & & I \\ & \breve{Q}_{xI} & \end{bmatrix} \tag{4.1.55}$$

$$C = \begin{bmatrix} (\breve{Q}_{xD}\nabla\breve{c})^T & \\ (\breve{Q}_{xD}\nabla\breve{h})^T & -I \end{bmatrix} \tag{4.1.56}$$

$$N = \begin{bmatrix} (\breve{Q}_{xI}\nabla\breve{c})^T \\ (\breve{Q}_{xI}\nabla\breve{h})^T \end{bmatrix}. \tag{4.1.57}$$

Notice that basis matrix in (4.1.56) is lower block triangular with non-singular blocks on the diagonal. It is therefore straightforward to solve for linear systems with this basis matrix. In fact, the direct sensitivity matrix $D = -C^{-1}N$ takes the form

$$D = -\begin{bmatrix} (\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^T \\ (\breve{Q}_{xD}\nabla\breve{h})^T(\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^T \quad -(\breve{Q}_{xI}\nabla\breve{h})^T \end{bmatrix}. \tag{4.1.58}$$

The structure of (4.1.58) is significant in the context of active-set QP solvers that solve the reduced QP subproblem in (4.1.25)–(4.1.27) using a variable-reduction null-space decomposition. The rows of D corresponding to general inequality constraints only have to be computed if the slack for the constraint is at a bound. Also note that the above transformation does not increase the number of degrees of freedom of the NLP since $n - m = \breve{n} - \breve{m}$. All of this means that adding general inequalities to a NLP imparts little extra cost for the rSQP algorithm as long as these constraints are not active.

For reasons of stability and algorithm efficiency, it may be desirable to keep at least some of the slack variables out of the basis and this can be accommodated also but is more complex to describe.

Most of the steps in a SQP algorithm do not need to know that there are general inequalities in the underlying NLP formulation but some steps do (i.e. globalization methods and basis selection). Therefore, those steps in a SQP algorithm that need access to this information are allowed to access the underlying NLP in a limited manner (see the Doxygen documentation for the class `NLPInterfacePack::`*`NLP`*).

4.2 Software design of rSQP++

The rSQP++ framework is implemented in C++ using advanced object-oriented software engineering principles. However, to solve certain types of NLPs with rSQP++ does not require any deep knowledge of object-orientation or C++. Example programs can be simple copied and modified.

4.2.1 An Object-Oriented Approach to SQP

4.2.1.1 Motivation for Object-Oriented Methods

Most numerical software (optimization, nonlinear equations etc.) consists of an iterative algorithm that primarily involves simple and common linear algebra operations. Mathematicians use a precise notation for these linear algebra operations when they describe an algorithm. For example, $y = Ax$ denotes matrix-vector multiplication irrespective of the special properties of the matrix A or the vectors y and x. Such elegant and concise abstractions are usually lost, however, when the algorithm is implemented in most programming environments and implementation details such as sparse data structures obscure the conceptual simplicity of the operations being performed. Currently it seems that developers have to choose between easy to use interpretive environments or more traditional compiled languages. Interpretive environments like Matlab© are popular with users since the abstractions they provide are very similar to those used in the mathematical formulation [33]. The problem with interpretive languages like Matlab is that they are not as efficient or as flexible as more general purpose compiled languages. When these algorithms are implemented in a compiled procedural language, like Fortran, the syntax is much more verbose, difficult to read, and prone to coding mistakes. Every data structure is seen in intimate detail and these details can obscure what may be an otherwise simple algorithm. While the level of abstraction provided by environments like Matlab is very useful, more elaborate data structures and operations are needed to handle problems with special structure and computing environments.

Modern software engineering modeling and development methods, collectively known as Object-Oriented Technology (OOT), can provide much more powerful abstraction tools [97], [96]. In addition to abstracting linear algebra operations, Object-Oriented Programming (OOP) languages like C++ can be used to abstract any special type of quantity or operation. Also OOT can be used to abstract larger chunks of an algorithm and provide for greater reuse. While newer versions of Matlab support some aspects of OOT, its proprietary nature and its loose typing are major disadvantages. A newly standardized graphical language for OOT is the Unified Modeling Language (UML) [96]. The UML is used to describe many parts of rSQP++. Appendix F provides a very short overview to the UML.

There are primarily two advantages to using data abstraction: it improves the clarity of the program, and it allows the implementation of the operations to be changed and optimized without affecting the design of the application or even requiring recompilation of much of the code. The concepts of OOT and data abstraction are discussed in more detail later in the context of rSQP.

4.2.1.2 Challenges in Designing Implementations for Numerical SQP Algorithms

There are many types of challenges in trying to build a framework for SQP (as well as for many other numerical areas) that allows for maximal sharing of code, and at the same time is understandable and extensible. Specifically, three types of variability will be discussed.

First, we need to come up with a way of modeling and implementing iterative algorithms, such as SQP, that will allow for steps to be reused between related algorithms and for existing algorithms to be extended. This type of higher-level algorithmic modeling and implementation is needed to make the steps in our rSQP algorithms more independent so that they are easier to maintain and to reuse. A framework called `GeneralIterationPack` has been developed for these types of iterative algorithms and serves as the backbone for rSQP++.

The second type of variability to deal with is in allowing for different implementations of various parts of the rSQP algorithm. There are many examples where different implementation options are possible and the best choice will depend on the general properties (i.e. sizes of n, m, and $n - r$ etc.) of the NLP being solved.

An example is the method used to implement the null-space matrix Z in (4.1.34). One option, referred to as the direct (or explicit) factorization, is to compute $D = -C^{-1}N$ up front. This method requires $(n - r)$ solves with the basis matrix C and also the storage of a dense $r \times (n - r)$ matrix D. Later, however, the tasks of performing matrix-vector products of the form $Z^T g$ and Zp_z, and building the inequality constraints in (4.1.27) are implemented using the precomputed dense matrix D. Therefore, no further solves with the basis matrix C are required. The other option, called the adjoint (or implicit) factorization, is to define Z implicitly and then to compute products like $Z^T g = -N^T(C^{-T}g^y) + g_u$. When there are few active variable bounds (i.e. # active bounds = nact $<< (n - r)$), the adjoint factorization is guaranteed to require fewer solves with C and demand less storage than the direct factorization. However, it is difficult to determine the best choice a priori. See the option `null_space_matrix` in Section 4.3.1.1.

Another example is the implementation of the Quasi-Newton reduced Hessian $B \approx Z^T W Z$.

The choice for whether to store B directly or its factorization (and what form of the factorization) depends on the choice of QP solver used to solve (4.1.25)–(4.1.27). If there are a lot of degrees of freedom ($(n - r)$ is large) then storing and manipulating the dense factors of B will become too expensive and therefore a limited-memory implementation may be preferred. See the option `quasi_newton` in Section 4.3.1.1.

Yet another example of variability in implementation options is in allowing for different implementations of the QP solver as described in Section 4.2.6 (See the option `qp_solver`).

A third source of variability is in how to exploit the special properties of an application area. Issues related to the management of various algorithmic and implementation options are more of a concern to the developers and implementors of the optimization algorithms than to the users of the algorithms. As long as an appropriate interface is available for user to select various options (see Section 4.3.1.1), the underlying complexity is not really their concern. However, what is a concern to advanced users of optimization software is a desire to tailor the numerical linear algebra to the specific properties of their potentially very specialized application area. Data structures, linear solvers and even computing environments (i.e. parallel processing using MPI) can be specialized for many applications. For example, a NLP may have constraints where the basis of the Jacobian C is block diagonal. Therefore, linear systems can be solved by working with the blocks separately and possibly in parallel. Examples of these types of NLPs include Multi-Period Design (MPD) [118] and Parameter Estimation and Data Reconciliation (PEDR) [116]. Another example of a specialized NLP is one where the constraints are comprised of discretized Partial Differential Equations (PDEs). For these types of constraints, iterative solvers have been developed to efficiently solve for linear systems with the basis of the Jacobian C. Abstract interfaces for matrices and linear solvers have been developed that allow the rSQP algorithm to be independent of the implementation of these operations. The abstractions that allow for this variability are described in Section 4.2.3.1.

For some NLPs, the matrix $\nabla c(x_k)$ can not even be formed implicitly (i.e. no matrix-vector products). And, linear systems with the basis of the Jacobian C in (4.1.33) can not be solved with arbitrary right hand sides. Or, solves with C^T are not possible (see [104]). For these types of NLPs, a special "direct sensitivity" interface has been developed (see Section 4.2.3.2). For a "direct sensitivity" NLP, the number of algorithmic and implementation options is greatly constrained and is therefore an example of additional complexity created by the interaction of all three types of variability.

Abstract interfaces to vectors and matrices have been developed and are described in Section 4.2.3.1 that serve as the foundation for facilitating the type of implementation and NLP specific

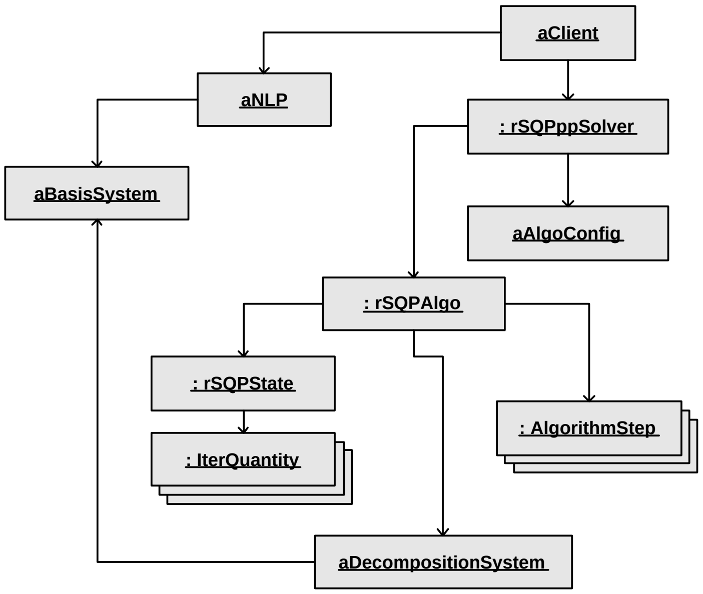


Figure 4.2. UML object diagram : Course grained object diagram for rSQP++

linear algebra variability described above. In addition, these abstract interfaces also help manage some of the algorithmic variability such as the choice of different null-space decompositions.

4.2.2 High-Level Object Diagram for rSQP++

There are many different ways to present rSQP++. Here, we take a top-down approach where we start with the basics and work our way down into more detail. This discussion is designed to help the reader to appreciate how a complex or specialized NLP is solved using rSQP++.

Figure 4.2 shows a high-level object diagram of a rSQP++ application, ready to solve a user-defined NLP. The NLP object `aNLP` is created by the user and defines the functions and gradients for the NLP to be solved (see Section 4.2.3.2). Closely associated with a NLP is a *`BasisSystem`* object. The *`BasisSystem`* object is used to implement the selection of the basis matrix C. This *`BasisSystem`* object is used by a variable-reduction null-space decomposition (see Section 4.2.5). Each NLP object is expected to supply a *`BasisSystem`* object. The NLP and *`BasisSystem`* objects collaborate with the optimization algorithm though a set of abstract linear algebra interfaces (see Section 4.2.3.1). By creating a specialized NLP subclass (and the associated linear algebra and *BasisSystem* subclasses) the implementation of all of the major linear algebra computations can be managed in a rSQP algorithm. This includes having full freedom to choose the data structures for all of the vectors and the matrices A, C, N and how nearly every linear algebra operation is performed. This also includes the ability to use fully transparent parallel linear algebra on a parallel computer even though none of the core rSQP++ code has any concept of parallelism.

Once a user has developed NLP and *BasisSystem* classes for their specialized application, a NLP object can be passed on to a `rSQPppSolver` object. The `rSQPppSolver` class is a convenient "facade" [42] that brings together many different components that are needed to build a complete optimization algorithm in a way that is transparent to the user. The `rSQPppSolver` object will instantiate an optimization algorithm (given a default or a user-defined configuration object) and will then solve the NLP, returning the solution (or partial solution on failure) to the NLP object itself. Figure 4.2 also shows the course grained layout of a rSQP++ algorithm. An advanced user can solve even the most complex specialized NLP without needing to understand how these algorithmic objects work together to implement an optimization algorithm. Understanding the underlying algorithmic framework is only necessary if the optimization algorithms need to be modified. The foundation for the algorithmic framework is discussed in Section 4.2.4. A complete example of a simple but very specialized NLP that overrides all of the linear algebra operations is described in Section 4.4.

While rSQP++ offers complete flexibility to solve many different types of specialized NLPs in diverse application areas such as dynamic optimization and control [16] and PDES [19] it can also be used to solve more generic NLPs such as are supported by modeling systems like GAMS [29] or AMPL [41]. For serial NLPs which can compute explicit Jacobian entries for A, a user needs to to create a subclass of `NLPSerialPreprocessExplJac` and define the problem functions and derivatives. For these types of NLPs, a default *BasisSystem* subclass is already defined which uses a sparse direct linear solver to implement all of the required functionality.

Figure 4.3 shows a UML package diagram of all of the major packages that make up rSQP++. At the very least, each package represents one or more libraries and the package dependencies also show the library dependencies. In many cases, each package is actually a C++ `namespace` (e.g. `namespace AbstractLinAlgPack { ...}`) and selected classes and methods are imported into higher level packages with C++ `using` declarations. The following are very brief descriptions of each package. The packages are described in more detail in Sections 4.2.3–4.2.5 and in Appendix C.

`MemMngPack` contains basic (yet advanced) memory management foundational code such as

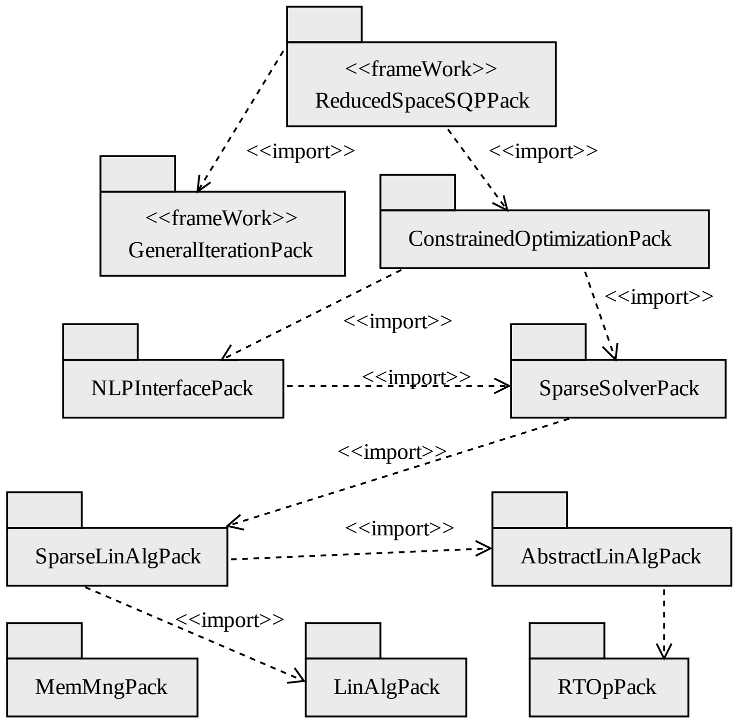


Figure 4.3. UML package diagram : Packages making up rSQP++

smart reference counted pointers and factory interfaces (see Appendix 8.8). These classes provide a consistent memory management style that is flexible and results in robust code. Without this foundation, much of the functionality in rSQP++ would have been very difficult to implement correctly and safely.

`RTOpPack` is comprised of an advanced low-level interface for vector reduction/transformation operators that allows the development of high-level linear algebra interfaces and numerical algorithms (i.e. rSQP++). The basic low-level operator interface is called `RTOp` which allows the development of arbitrary user-defined vector operators. The design of this interface was critical to the development of rSQP++ in a way that allows full exploitation of a parallel computer and specialized application without requiring rSQP++ to have any concept of parallel constructs. The advanced concepts behind the design of `RTOpPack` are described in more detail in [10].

`AbstractLinAlgPack` is a full-featured set of interfaces to linear algebra quantities such as vectors and matrices (or linear operators). A vector interface is the foundation for all numerical applications and provides some of the greatest challenges from an object-oriented design point of view. The vector interface in `AbstractLinAlgPack` is built on the foundation of `RTOpPack` and allows the efficient development of many advanced types of optimization algorithms. There are basic interfaces for general, symmetric and nonsingular matrices. The *`BasisSystem`* interface mentioned above is also include in this package. These linear algebra interfaces are devoid of any concrete implementations and form the foundation for all the linear algebra computations in rSQP++. These interfaces are described in more detail along with the NLP interfaces in Section 4.2.3

`LinAlgPack` contains concrete data types for dense BLAS-compatible linear algebra. Part of this package is a portable C++ interface to a Fortran BLAS library. This package forms the foundation for all dense serial linear algebra data structures and computations that take place in rSQP++.

`SparseLinAlgPack` includes many different implementations of linear algebra interfaces defined in `AbstractLinAlgPack` for serial applications. In addition to a default serial vector class, dense and sparse matrix classes are also provided. Several other important matrix interfaces are also declared that are useful in circumstances where serial linear algebra quantities are mixed with more general (i.e. parallel) linear algebra implementations. The implementations and the interfaces included in this package provide a (nearly) complete linear algebra foundation for the development of any advanced optimization algorithm.

`SparseSolverPack` provides interfaces to direct serial linear solvers, subclasses for several popular implementations (such as several Harwell solvers and SuperLU) and includes a subclass of *`BasisSystem`* that uses one of these direct solvers.

`NLPInterfacePack` defines the basic NLP interfaces that are needed to implement various optimization algorithms (particularly SQP methods). These basic interfaces communicate to an optimization algorithm through linear algebra quantities using the `AbstractLinAlgPack` interface. These basic interfaces are described along with the linear algebra interfaces in Section 4.2.3. This package also contains several NLP node subclasses for common types of NLPs. These subclasses make it very easy to implement a serial NLP.

`ConstrainedOptimizationPack` is a mixed collection of several different types of interfaces and implementations. Some of the major interfaces and implementations defined in this package are for null-space decompositions, QP solvers, merit functions and generic line searches.

`GeneralIterationPack` is a framework for developing iterative algorithms. Any type of iterative algorithm can be developed and there is no specialization for numerics in the package. This framework provides the backbone for all rSQP++ optimization algorithms and is described in more detail in Section 4.2.4

`ReducedSpaceSQPPack` is the highest level package (namespace) in rSQP++. It contains all of the rSQP specific classes and contains the basic infrastructure for building rSQP++ algorithms (such as step classes) as well as other utilities. Also included are the `rSQPppSolver` facade class and two built-in configuration classes for active-set rSQP (`rSQPAlgo_ConfigMamaJama`) and interior-point rSQP (`Algo_ConfigIP`). Basic interaction with a rSQP++ algorithm through a `rSQPppSolver` object is described in the Doxygen documentation starting at

```
RSQPPP_BASE_DOC/ReducedSpaceSQPPack/html/index.html
```

It is not important that the user understand the deatils of all of these packages but some packages are of more interest to an advanced user and these packages are described next. Some of the other packages are described in Appendix C. For details on the installation of rSQP++, see Appendix B.

4.2.3 Overview of NLP and Linear Algebra Interfaces

All of the high-level optimization code in rSQP++ is designed to allow arbitrary implementations of the linear algebra objects. It is the NLP object that defines the basis for all of the linear algebra by exposing a set of abstract "factories" [42] for creating linear algebra objects. Before the specifics of the NLP interfaces are described, the basic linear algebra interfaces are discussed first. These are the interfaces that allow rSQP++ to utilize fully parallel linear algebra in a completely transparent manner.

4.2.3.1 Overview of `AbstractLinAlgPack`: Interfaces to Linear Algebra

Figure 4.4 shows a UML class diagram of the basic linear algebra abstractions. The foundation for all the linear algebra is in vector spaces. A vector space object is represented though an abstract interface called *`VectorSpace`*. A *`VectorSpace`* object primarily acts as an "abstract factory" [42] and creates vectors from the vector space using the *`create_member()`* method. *`VectorSpace`* objects can also be used to check for compatibility using the *`is_compatible()`* method. Every *`VectorSpace`* object has a dimension. Therefore a *`VectorSpace`* object can not be used to represent an infinite-dimensional vector space. This is not a serious limitation since all vectors must have a finite dimension when implemented in a computer. Just because two vectors from different vector spaces have the same dimension does not imply that the implementations

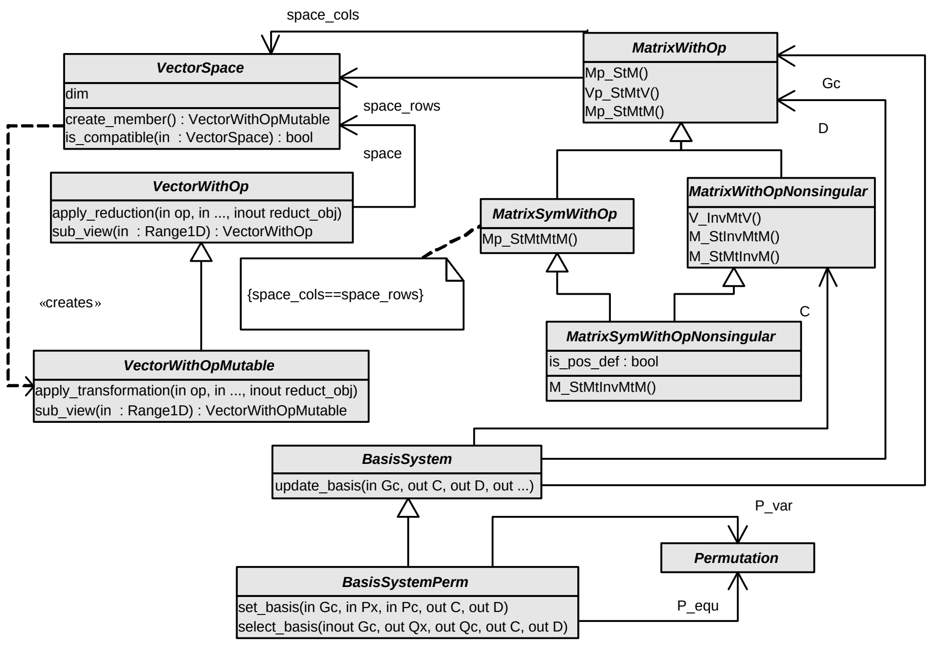

space_cols
VectorSpace
dim
create_member() : VectorWithOpMutable
is_compatible(in : VectorSpace) : bool
space_rows
space
MatrixWithOp
Mp_StM()
Vp_StMtV()
Mp_StMtM()
Gc
D
VectorWithOp
apply_reduction(in op, in ..., inout reduct_obj)
sub_view(in : Range1D) : VectorWithOp
MatrixSymWithOp
Mp_StMtMtM()
MatrixWithOpNonsingular
V_InvMtV()
M_StInvMtM()
M_StMtInvM()
{space_cols==space_rows}
«creates»
C
MatrixSymWithOpNonsingular
is_pos_def : bool
M_StMtInvMtM()
VectorWithOpMutable
apply_transformation(in op, in ..., inout reduct_obj)
sub_view(in : Range1D) : VectorWithOpMutable
BasisSystem
update_basis(in Gc, out C, out D, out ...)
P_var
Permutation
BasisSystemPerm
set_basis(in Gc, in Px, in Pc, out C, out D)
select_basis(inout Gc, out Qx, out Qc, out C, out D)
P_equ

Figure 4.4. UML class diagram : `AbstractLinAlgPack`, abstract interfaces to linear algebra

will be compatible. For example, distributed parallel vectors may have the same global dimension but the vector elements may be distributed to processors differently (we say that they have different "maps"). This is an important concept to remember.

Vector implementations are abstracted behind interfaces. The basic vector interfaces are broken up into two levels: *`VectorWithOp`* and *`VectorWithOpMutable`*. The *`VectorWithOp`* interface is an immutable interface where vector objects can not be changed by the client. The *`VectorWithOpMutable`* interface extends the *`VectorWithOp`* interface in allowing clients to change the elements in the vector. These vector interfaces are very powerful and allow the client to perform many different types of operations. The foundation of all vector functionality is the ability to allow clients to apply user-defined `RTOp` operators which perform arbitrary reductions and transformations (see the methods *`apply_reduction(...)`* and *`apply_transformation(...)`*[1]). The ability to write these types of user-defined operators is critical to the implementation of advanced optimization algorithms. A single operator applica-

[1]Note that both *`apply_reduction(...)`* and *`apply_transformation(...)`* can perform reductions and return reduction objects `reduct_obj`. Assuming that only *`apply_reduction(...)`* can perform a reduction is a common misunderstanding. The differences between these two methods is subtle and the reader should consult the the Doxygen documentation for more details.

tion method is the only method that a vector implementation is required to provide (in addition to some trivial methods such as returning the dimension of the vector) which makes it fairly easy to add a new vector implementation. In addition to allowing clients to apply `RTOp` operators, the other major feature is the ability to create arbitrary subviews of a vector (using the *`sub_view()`* methods) as abstract vector objects. This is an important feature in that it allows the optimization algorithm to access the dependent (i.e. state) and independent (i.e. design) variables separately (in addition to any other arbitrary range of vector elements). Support for subviews is supported by default by every vector implementation through default view classes (see the class `VectorWithOpMutableSubview`) that rely only on the `RTOp` application methods. The last bit of major functionality is the ability of the client to extract an explicit view of a subset of the vector elements. This is needed in a few parts of an optimization algorithm for such tasks as dense quasi-Newton updating of the reduced Hessian and the implementation of the compact LBFGS matrix. Aside from vectors being important in their own right, vectors are also the major type of data that is communicated between higher-level interfaces such as linear operators (i.e. matrices) and function evaluators (i.e. NLP interfaces).

The basic matrix (i.e. linear operator) interfaces are also shown in Figure 4.4. The *`MatrixWithOp`* interface is for general rectangular matrices. Associated with any *`MatrixWithOp`* object is a column space and a row space shown as `space_cols` and `space_rows` respectively in the figure. Since column and row *`VectorSpace`* objects have a finite dimension, this implies that every matrix object also has finite row and column dimensions. Therefore, these matrix interfaces can not be used to represent an infinite-dimensional linear operator. Note that all finite-dimensional linear operators can be represented as a matrix (which is unique) so the distinction between a finite-dimensional matrix and a finite-dimensional linear operator is insignificant. The column and row spaces of a matrix object identify the vector spaces for vectors that are compatible with the columns and rows of the matrix respectively. For example, if the matrix A is represented as a *`MatrixWithOp`* object then the vectors y and x would have to lie in the column and row spaces respectively for the matrix-vector product $y = Ax$.

These matrix interfaces go beyond what most other abstract matrix/linear-operator interfaces have attempted. Other abstract linear-operator interfaces only allow the applications of $y = Ax$ or the transpose (adjoint) $y = A^T x$ for vector-vector mappings. Every *`MatrixWithOp`* object can provide arbitrary subviews as *`MatrixWithOp`* objects through the *`sub_view(...)`* methods. These methods have default implementations based on default view classes which are fundamentally supported by the ability to take arbitrary subview of vectors. This ability to create these subviews is critical in order to access the basis matrices in (4.1.33) given a Jacobian object `Gc` for ∇c. These matrix interfaces also allow much more general types of linear algebra operations. The matrix *`MatrixWithOp`* interface allows the client to perform level 1, 2 and 3 BLAS operations (see Appendix E for a discussion of the convention for naming functions for linear algebra operations)

$$
\begin{aligned}
B &= \alpha\, op(A) + B \\
y &= \alpha\, op(A)\, x + \beta y \\
C &= \alpha\, op(A)\, op(B) + \beta C.
\end{aligned}
$$

One of the significant aspects of these linear algebra operations is that an abstract *`MatrixWithOp`* object can appear on the left-hand-side. This adds a whole set of issues (i.e. multiple dispatch [76, Item 31]) that are not present in other linear algebra interfaces.

The matrix interfaces assume that the matrix operator or the transpose of the matrix operator can be applied. Therefore, a correct *`MatrixWithOp`* implementation must be able to perform the transposed as well as the non-transposed operation. This requirement is important when the NLP interfaces are discussed later.

Several specializations of the *`MatrixWithOp`* interface are also required in order to implement advanced optimization algorithms. All symmetric matrices are abstracted by the *`MatrixSymWithOp`* interface. This interface is required in order for the operation

$$C = \alpha\, op(B)\, op(A)\, op(B^T) + \beta C$$

to be guaranteed to maintain the symmetry of the matrix C. Note that a symmetric matrix requires that the column and row spaces be the same which is shown by the UML constraint {`...`} in Figure 4.4.

The specialization *`MatrixWithOpNonsingular`* is for nonsingular square matrices that can be used to solve for linear systems. As a result, the level 2 and 3 BLAS operations

$$
\begin{aligned}
y &= op(A^{-1})\, x \\
C &= \alpha\, op(A^{-1})\, op(B) \\
C &= \alpha\, op(B)\, op(A^{-1})
\end{aligned}
$$

are supported. The solution of linear systems represented by these operations can be implemented in a number of different ways. A direct factorization followed by back solves or alternatively a preconditioned iterative solver (i.e. GMRES or some other Krylov subspace method) could be used. Or, a more specialized solution process could be employed which is tailored to the special properties of the matrix (i.e. banded matrices).

The last major matrix interface *MatrixSymWithOpNonsingular* is for symmetric nonsingular matrices. This interface allows the implementation of the operation

$$C = \alpha \, op(B) \, op(A^{-1}) \, op(B^T)$$

and guarantees that C will be a symmetric matrix.

A more detailed discussion of these basic linear algebra interfaces can be found in the Doxygen documentation.

A major part of a rSQP algorithm, based on a variable-reduction null-space decomposition (see Section 4.2.5), is the selection of a basis. The fundamental abstraction for this task is *BasisSystem* (as first introduced in Figure 4.2). The *update_basis()* method takes the rectangular Jacobian `Gc` (∇c) and returns a *MatrixWithOpNonsingular* object for the basis matrix C. This interface assumes that the variables are already sorted according to (4.1.31). For many applications, the selection of the basis is known a priori (e.g. PDE-constrained optimization). For other applications, it is not clear what the best basis selection should be. For the latter type of application, the basis selection can be performed on-the-fly and result in one or more different basis selections during the course of a rSQP algorithm. The *BasisSystemPerm* specialization supports this type of dynamic basis selection and allows clients to either ask the basis-system object for a good basis selection (*select_basis()*) or can tell the basis-system object what basis to use (*select_basis()*). The selection of dependent x_D and independent x_I variables and the selection of the decomposed $c_d(x)$ and undecomposed $c_u(x)$ constraints is represented by *Permutation* objects which are passed to and from these interface methods. The protocol for handling basis changes is somewhat complicated and is beyond the scope of this discussion.

4.2.3.2 Overview of `NLPInterfacePack`: Interfaces to Nonlinear Programs

The hierarchy of NLP interfaces that all rSQP++ optimization algorithms are based on is shown in Figure 4.5. These NLP interfaces act primarily as evaluators for the functions and gradients that define the NLP. These interfaces represent the various levels of intrusiveness into an application area.

The base-level NLP interface is called *`NLP`* and defines the nonlinear program. An *`NLP`* object defines the vector spaces for the variables $\mathcal{X}$ and the constraints $\mathcal{C}$ as *`VectorSpace`* objects `space_x` and `space_c` respectively. The *`NLP`* interface allows access to the initial guess of the solution x_0 and the bounds x_L and x_U as *`VectorWithOp`* objects `x_init`, `xl` and `xu` respectively.

The *`NLP`* interface allows clients to evaluate just the zero-order quantities $f(x) \in \mathbf{R}$ and $c(x) \in \mathcal{C}$ as scalar and *`VectorWithOpMutable`* objects respectively. Many different steps in an optimization algorithm do not require sensitivities for the problem functions. Examples include several different line search and trust region globalization methods (i.e. Filter and exact merit function). Nongradient-based optimization methods could also be implemented through this interface but smoothness and continuity of the variables and functions is assumed by default. Note that this interface is the same as a NAND (nested analysis and design) approach if there are no equality constraints (i.e. removed using nonlinear elimination). The *`NLP`* interface can also be used for unconstrained optimization (i.e. $|\mathcal{C}| = m = 0$) or for a system of nonlinear equations (i.e. $|\mathcal{X}| = n = |\mathcal{C}| = m$).

The next level of NLP interface is *`NLPObjGradient`*. This interface simply adds the ability to compute the gradient of the objective function $\nabla f(x) \in \mathcal{X}$ as a *`VectorWithOpMutable`* object `Gf`. For many applications, it is far easier and less expensive to compute sensitivities for the objective function than it is for the constraints. That is why this functionality is considered more general than sensitivities for the constraints and is therefore higher in the inheritance hierarchy than interfaces the include sensitivities for ∇c.

Sensitivities for the constraints ∇c are broken up into two separate interfaces. These interfaces represent the capabilities of the underlying application code. The most general (from the standpoint of the optimization algorithm) interface is *`NLPFirstOrderInfo`*. This NLP interface assumes that the application can, at the very least, form and maintain a *`MatrixWithOp`* object `Gc` for the gradient of the constriants ∇c. Recall that this implies that operations of the form $u = \nabla c^T v$ and $u = \nabla c\, v$ can both be performed with arbitrary vectors. Note that while operations of the form $u = \nabla c^T v$ can be approximated using directional finite differences

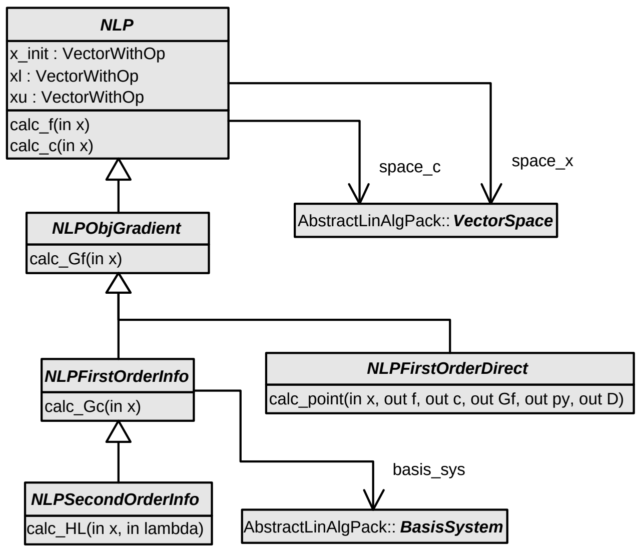


Figure 4.5. UML class diagram : `NLPInterfacePack`, abstract interfaces to nonliner programs

(i.e. $\nabla c^T v = \lim_{\epsilon \rightarrow 0}(c(x + \epsilon v) - c(x))/\epsilon)$), operations of the form $u = \nabla c\, v$ can not, so this interface can not simply be approximated using finite differences. A *`NLPFirstOrderInfo`* object can optionally supply a *`BasisSystem`* object that is specialized for application's `Gc` matrix object. By implementing the *`NLPFirstOrderInfo`* interface (with the associated *`VectorSpace`*, `MatrixWithOp` and *`BasisSystem`* subclasses), the critical linear algebra computations can be performed in a rSQP algorithm. See Section 4.2.5 for a description of how the variable-reduction null-space decompositions use a *`BasisSystem`* object to define all of the required decomposition matrices. An example of a very structured NLP is described in Section 4.4 where all of the linear algebra objects are specialized for the NLP.

For applications that can not satisfy the *`NLPFirstOrderInfo`* interface, there is the *`NLPFirstOrderDirect`* interface. As the name implies, the *`NLPFirstOrderDirect`* interface only requires the direct sensitivity matrix $D = -C^{-1}N$ and the solution to the Newton linear systems $p_y = C^{-1}c$. With usually minor modifications, almost any application code that uses a Newton method for the forward solution can be used to implement the *`NLPFirstOrderDirect`* interface (see Chapter 5 for an example application). Both the orthogonal and the coordinate variable-reduction null-space decompositions can be implemented with just the quantities $D = -C^{-1}N$ and $p_y = C^{-1}c$.

Finally, the most advanced NLP interface defined is *`NLPSecondOrderInfo`*. This NLP interface allows the optimization algorithm to compute a *`MatrixSymWithOp`* matrix object `HL` for the Hessian of the Lagrangian $W = \nabla^2_{xx} L = \nabla^2 f(x) + \sum_{j=1}^{m} \lambda_j \nabla^2 c_j(x)$. How this Hessian matrix object is used can vary greatly. This matrix object can be used to compute the exact reduced Hessian $B = Z^T W Z$ or can be used to form the full KKT matrix. Many other possibilities exist but the best approach will be very much application dependent.

The *`NLP`*, *`NLPFirstOrderDirect`*, *`NLPFirstOrderInfo`* and *`NLPSecondOrderInfo`* interfaces represent four different levels of invasiveness to the application. The *`NLP`* interface without equality constraints can used to implement a basic NAND optimization algorithm while on the other extreme the *`NLPSecondOrderInfo`* interface can be used to implement a fully coupled invasive SAND method with access to second derivatives.

4.2.4 Overview of `GeneralIterationPack` : Framework for General Iterative Algorithms

`GeneralIterationPack` is a framework for building iterative algorithms in C++. This framework is not specific to numerical applications and can be used for any application area where it may be useful. The challenges in building such a framework are in trying to keep the steps and other components in the algorithm as decoupled as possible so that they can be reused in many different related algorithms.

To illustrate the design and the underlying concepts, consider the iterative algorithm shown in Figure 4.6. In such an algorithm, quantities computed in one step are used by one or more other steps. In the example, the iteration quantities are x, p, q, and r. These quantities may represent anything from scalars to vectors or matrices all the way up to arbitrarily complex objects. Such algorithms must be initialized before they can be run as shown in the example. Once some minimum initialization is completed, the algorithm starts to run. The average iteration is executed sequentially from step 1 to step 4 and then loops back to step 1 again with the iteration counter k incremented by one. During some iterations, however, one or more minor loops between steps 2 and 3 may be required. The steps in the algorithm are dependent on the other steps (at least implicitly) through common iteration quantities. For example, steps 2, 3 and 4 all access the iteration quantity q. Steps may also have algorithmic control dependencies required to perform minor loops. In the example, steps 2 and 3 are involved in a minor loop and this suggests some type of dependency between them. The last type of dependency that exists is also between steps and the iteration quantities that are updated or accessed and is related to the storage requirements for iteration quantities. For example, step 1 only requires one storage location for p to update p^k,

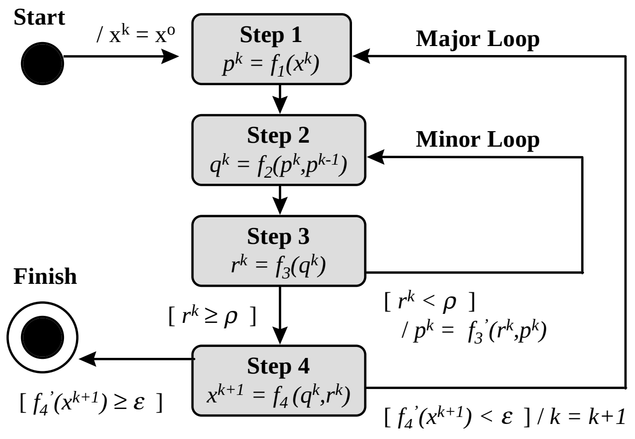


Figure 4.6. **UML Activity Diagram:** Example iterative algorithm

while step 2 requires dual storage for p (p^k and p^{k-1}) in order to update q^k. Suppose step 1 were implemented long before step 2. In this case, it may have been assumed that only one storage location was needed for p. When step 2 is later implemented, will step 1 have to be modified to accommodate the additional storage locations? Many implementation techniques would require the code implementing step 1 to be modified in this case, thereby coupling step 1 to step 2 as well as to the implementation of the iteration quantity p. Finally, there must be some termination criteria for the algorithm. This check for termination occurs after step 4 is completed in the example.

Figure 4.7 shows a UML[2] diagram for the `GeneralIterationPack` framework. At the center of the framework is an `Algorithm` object. Associated with an `Algorithm` object are one or more *`AlgorithmStep`* objects where each is identified by a unique name (`step_name`). Step objects, which are instantiations of subclasses of *`AlgorithmStep`*, implement the steps in the algorithm. Using objects to represent a sub-algorithm is a well known OO design pattern (see the "Strategy" pattern in [42]). Iteration quantities are abstracted behind the *`IterQuantity`* interface and are aggregated into a single `AlgorithmState` object. The `AlgorithmState` object acts as a central repository for these quantities. Individual *`IterQuantity`* objects are identified by a unique name (`iq_name`). Aggregating all of the iteration quantities into one cen-

[2]The UML [96] has a convention for the names of classes and objects which is used in this paper. The names of concrete classes use the font `ConcreteClass`. This is also the font used for objects. An object is an instantiation of a concrete class. Abstract class names, as well as abstract operation names, are in italics such as *`AbstractClass`* and *`AbstractClass::operation(...)`*. While a concrete class may have direct object instantiations, an abstract class (interface) may not (i.e. because these classes always have one or more undefined abstract operation).

tral location helps to remove the data dependencies between Step objects. The Step objects use the *IterQuantityAccess<...>* interface to update and access iteration quantities. The operation *set_k(offset)* is called to update a quantity for the specific iteration `k + offset`, while *get_k(offset)* is used to access a quantity already updated. Such an interface to iteration quantities relieves Step objects from having to know if a quantity requires single or multiple storage. So in the previous mentioned scenario for our example algorithm, when the class for step 2 is implemented after step 1, the class for step 1 would not have to be modified at all or even recompiled. Also, the operation *get_k(offset)* validates that the quantity was indeed updated for the iteration `k + offset`. This feature has been invaluable during the development of rSQP++ in catching mistakes in algorithm logic and/or implementation. Finally, an *AlgorithmTrack* object is used to output intermediate information about the algorithm by examining the `AlgorithmState` object. By creating a subclass of *AlgorithmTrack*, clients can easily monitor the progress of an algorithm. If more sophisticated monitoring and control of an algorithm by the client is required, additional Step objects can be inserted into an already preformed algorithm. In addition, an algorithm can be altered while it is running by adding and removing Step objects, thereby allowing it to be adapted for changing needs.

Figure 4.8 shows an object diagram for the example algorithm in Figure 4.6. In this diagram, the iteration quantities are shown aggregated inside the `AlgorithmState` object where the link qualifier names are given for each quantity. The concrete type of each of these quantities is `IterQuantityAccessContiguous<...>` which provides sequential storage for successive iterations.

This design also allows for distributed algorithmic control. Algorithm control is shared between the `Algorithm` and *AlgorithmStep* objects. The `Algorithm` object is responsible for executing steps sequentially (from Step 1 to Step 4 in our example). *AlgorithmStep* objects are responsible for initialing minor loops through the `Algorithm` object (Step 3 initiates the Minor Loop in the example). Figure 4.9 shows a UML collaboration diagram illustrating how algorithm control is implemented for our example algorithm. The scenario shown is for two major iterations ($k = 0, 1$) where the minor loop is executed once in the first ($k = 0$) iteration.

The details for the interfaces and the collaborations between the objects in this framework are documented in the Doxygen generated documentation starting in the file

```
RSQPPP_BASE_DOC/GeneralIterationPack/html/index.html
```

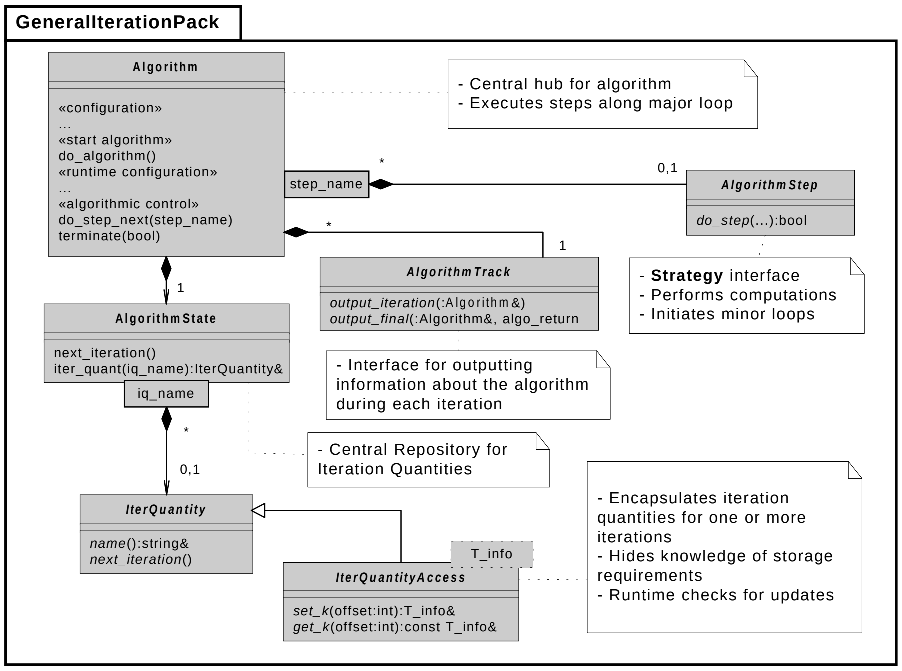

GeneralIterationPack
Algorithm
«configuration»
...
«start algorithm»
do_algorithm()
«runtime configuration»
...
«algorithmic control»
do_step_next(step_name)
terminate(bool)
- Central hub for algorithm
- Executes steps along major loop
step_name
*
0,1
AlgorithmStep
do_step(...):bool
*
1
AlgorithmTrack
output_iteration(:Algorithm&)
output_final(:Algorithm&, algo_return
- Strategy interface
- Performs computations
- Initiates minor loops
1
AlgorithmState
next_iteration()
iter_quant(iq_name):IterQuantity&
iq_name
- Interface for outputting information about the algorithm during each iteration
*
0,1
- Central Repository for Iteration Quantities
IterQuantity
name():string&
next_iteration()
T_info
IterQuantityAccess
set_k(offset:int):T_info&
get_k(offset:int):const T_info&
- Encapsulates iteration quantities for one or more iterations
- Hides knowledge of storage requirements
- Runtime checks for updates

Figure 4.7. UML Class Diagram: `GeneralIterationPack`: An object-oriented framework for building iterative algorithms

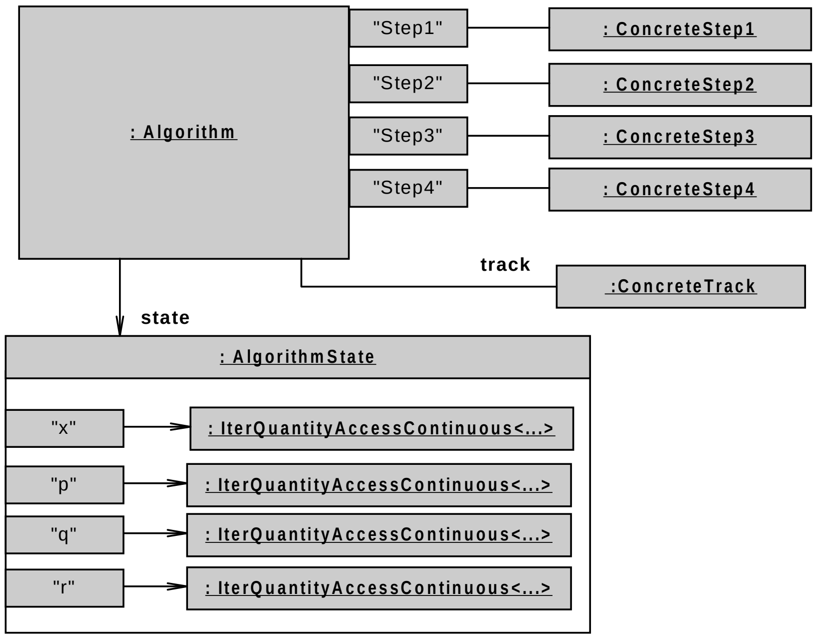


Figure 4.8. UML Object Diagram: Instantiations (objects) of `GeneralIterationPack` classes for the example algorithm in Figure 4.6

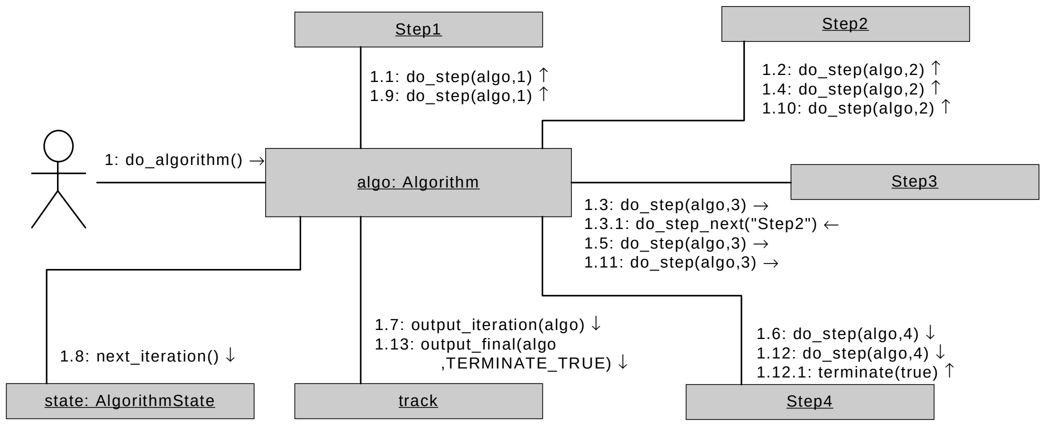


Figure 4.9. UML Collaboration Diagram: Scenario for the example algorithm in Figure 4.6

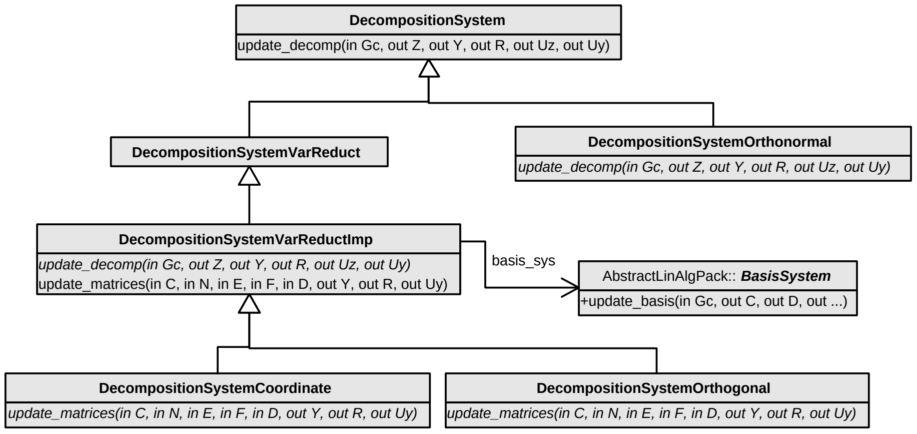

DecompositionSystem
update_decomp(in Gc, out Z, out Y, out R, out Uz, out Uy)
DecompositionSystemVarReduct
DecompositionSystemOrthonormal
update_decomp(in Gc, out Z, out Y, out R, out Uz, out Uy)
DecompositionSystemVarReductImp
update_decomp(in Gc, out Z, out Y, out R, out Uz, out Uy)
update_matrices(in C, in N, in E, in F, in D, out Y, out R, out Uy)
basis_sys
AbstractLinAlgPack:: BasisSystem
+update_basis(in Gc, out C, out D, out ...)
DecompositionSystemCoordinate
update_matrices(in C, in N, in E, in F, in D, out Y, out R, out Uy)
DecompositionSystemOrthogonal
update_matrices(in C, in N, in E, in F, in D, out Y, out R, out Uy)

Figure 4.10. UML Class Diagram: Inheritance hierarchy for null-space decompositions

4.2.5 Overview of Interfaces for Null-Space Decompositions

An important computation in a rSQP algorithm is the null-space decomposition used to project the full-space QP subproblem into the reduced space. In rSQP++, the decomposition matrices Z, Y, U_z and U_y in (4.1.23), (4.1.26)–(4.1.27) are represented by *MatrixWithOp* objects while the nonsingular matrix R in ((4.1.24) is represented by a *MatrixWithOpNonsingular* object. Once these matrix objects are initialized for the current iteration, the rest of the rSQP++ algorithm can be implemented by interacting only with these matrices through the *MatrixWithOp* and *MatrixWithOpNonsingular* interfaces. The basic interface that a rSQP++ algorithm uses to construct the matrices Z, Y, U_z, U_y and R from the matrix A is *DecompositionSystem*. This interface, as well as more specialized interfaces for variable-reduction decompositions, is shown in Figure 4.10.

The *DecompositionSystem* interface has an operation called *update_decomp(...)* which the rSQP++ algorithm calls to update the decomposition matrices. The *DecompositionSystem* interface also exposes a set of factory objects (not shown in the figure) that can create matrix objects for Z, Y, R, Uz and Uy that are compatible with the concrete decomposition-system object.

DecompositionSystemVarReduct is a specialized interface that all variable-reduction decompositions inherit from. *DecompositionSystemVarReductImp* is an implementation node subclass that provides a common implementation that all variable-reduction decompositions can share. This matrix subclass defines the factory objects for Z and Uz. The key to making the variable-reduction decomposition subclasses independent of the special properties of the underlying NLP and linear solver is to use a *BasisSystem* object which takes care of the basis handling. The *BasisSystem* object provides access to the basis matrix C as a *MatrixWithOpNonsingular* object as well as the matrices N, E, and F as *MatrixWithOp* objects. Given a *BasisSystem* object, the *DecompositionSystemVarReductImp* subclass can fully define the null-space matrix Z in (4.1.34) and the projected matrix Uz in (4.1.35). This subclass performs all of the interaction with the *BasisSystem* object to form the basis matrices. However, this subclass can not define the matrix objects for Y, R and Uy since these depend on the definition of the quasi-range-space matrix Y. The computation of these matrix objects are deferred to subclasses through the pure-virtual method *update_matrices(...)*. This method passes the basis matrix objects for C, N, E, F and potentially the direct sensitivity matrix object for D to the subclass which then returns updated matrix objects for Y, R and Uy.

The coordinate decomposition defined in (4.1.36)–(4.1.38) is implemented by the subclass *DecompositionSystemCoordinate*. The implementation of this subclass is very simple as $R = C, U_y = E$.

The orthogonal decomposition defined in (4.1.39)–(4.1.41) is implemented by the subclass *DecompositionSystemOrthogonal*. The implementation if this subclass is more complex because of the more complicated definitions of Y, R, and U_y. See the Doxygen documentation for this subclass for more details on how these matrices are implemented.

The last decomposition system subclass is *DecompositionSystemOrthonormal* which implements a different type of null-space decomposition based on a QR factorization. The linear algebra performed in this class uses dense computations and is therefore only applicable to small serial NLPs.

Since the null-space decomposition is such an important part of a rSQP algorithm it is very important to validate that decomposition matrices Z, Y, R, U_z and U_y obey the correct properties. The test class `DecompositionSystemTester` has been developed for this purpose. The tests performed by this class do not significantly increase the total runtime for the application and can be performed on even the largest and most difficult problems. The tests performed ,of course, catch gross programming and other errors but are also sensitive to ill conditioning in the problem. If any of the tests fail, the overall rSQP algorithm is terminated. This class accepts many different options that control the level of output produced to the `rSQPppJournal.out` file (see the options group `DecompositionSystemTester`).

The decomposition system interfaces and subclasses are part of the package (namespace)

`ConstrainedOptimizationPack` and are documented in the Doxygen collection starting in

`RSQPPP_BASE_DOC/ConstrainedOptimizationPack/html/index.html`

4.2.6 Interfaces to Quadratic Programming Solvers

Another very important numerical computation in a rSQP algorithm is the solution to the reduced-space QP subproblem in (4.1.25)–(4.1.27). In order to decouple the rSQP code away from the QP solver used to solve the QP subproblem, an abstract interface to QP solvers called *`QPSolver-Relaxed`* has been developed. The *`QPSolverRelaxed`* is very general and has seen application in areas other than SQP (such as MPC in [11]). The QP solved by this interface is of the form

$$
\begin{aligned}
\min_{d \in \mathbf{R}^{n_d}} \quad & g^T d + \tfrac{1}{2}\, d^T G d + M(\eta) && (4.2.59)\\
\text{s.t.} \quad & \eta^L \leq \eta && (4.2.60)\\
& d^L \leq d \leq d^U && (4.2.61)\\
& e^L \leq op(E)d - b\eta \leq e^U && (4.2.62)\\
& op(F)d + (1-\eta)f = 0 && (4.2.63)
\end{aligned}
$$

where:

$$
\begin{aligned}
& d, d^L, d^U \ \in \ \mathbf{R}^{n_d}\\
& \eta, \eta^L \ \in \ \mathbf{R}\\
& M(\eta) \ \in \ \mathbf{R} \ \rightarrow \ \mathbf{R}\\
& g \ \in \ \mathbf{R}^{n_d}\\
& G = G^T \ \in \ \mathbf{R}^{n_d \times n_d}\\
& op(E) \ \in \ \mathbf{R}^{m_{in} \times n_d}\\
& e^L, e^U, b \ \in \ \mathbf{R}^{m_{in}}\\
& op(F) \ \in \ \mathbf{R}^{m_{eq} \times n_d}\\
& f \ \in \ \mathbf{R}^{m_{eq}}
\end{aligned}
$$

As shown in (4.2.59)–(4.2.63), a very simple relaxation of the constraints is built into the formulation. The form of this relaxation is biased toward use in a SQP algorithm. The form of the function $M(\eta)$ in the objective (4.2.59) is specified by the subclasses that implement this interface.

An appropriate form of this function for a convex QP solver might be $M(\eta) = (\eta + 1/2\eta^2)\hat{M}$, where $\hat{M}$ is a large constant. For a QP solver capable of handling an indefinite Hessian, $M(\eta) = \eta\hat{M}$, when $\hat{M}$ is a large constant, may be a better choice. No matter how the function $M(\eta)$ is defined, as long as $d(M(\eta))/d(\eta)|_{\eta=\eta^L}$ is sufficiently large, then η will be at its lower bound $\eta = \eta^L$ ($\eta^L = 0$ usually) in (4.2.60) if an unrelaxed feasible region exits for (4.2.61)–(4.2.63).

The method *`QPSolverRelaxed::solve_qp(...)`* is called to pass the arguments defining the QP to the QP solver and to return the solution. If the solution is not found, then a partial solution will be returned and some information as to the status of the returned point will be given (i.e. dual feasible, primal feasible, etc.). The problem vectors g, b, f, d^L, d^U, e^L and e^U are represented as `VectorWithOp` objects. What makes this interface different from other QP interfaces, such as described in [115], is that the defining matrix objects are represented through the abstract interfaces *`MatrixSymWithOp`* for the Hessian G and *`MatrixWithOp`* for the Jacobian matrices E and F. In this way, the client (i.e. the rSQP algorithm in the case of rSQP++) need not know about the special properties of the Hessian or the Jacobian matrices or how the QP is solved.

For some QP solvers that implement the *`QPSolverRelaxed`* interface, such as QPOPT and QPSOL, interaction with the matrices G, E and F through the *`MatrixWithOp`* interface is all that is needed to solve the QP in a reasonably efficient manner (with respect to the specific solver). However, most implementations of the *`QPSolverRelaxed`* interface can not efficiently solve the QP with just the interface provided through *`MatrixSymWithOp`* and *`MatrixWithOp`*. For many of these QP solver subclasses, more specialized matrix interfaces must be supported by matrix objects for G, E and/or F. For example, the subclass for QPKWIK [106] must be able to extract the dense inverse of the Cholesky factor of the Hessian G. In order to do this, the matrix object for G must support the *`MatrixExtractInvCholFactor`* interface. Therefore, to use QPKWIK efficiently, the Hessian G is usually stored and manipulated using the dense inverse of the Cholesky factor. For other QP solvers, other less intrusive matrix interfaces are all that are required. For example, with QPSchur (see [10]) the QP can be efficiently solved if G supports the *`MatrixSymWithOpNonsingular`* interface. Other approaches for solving the QP defined in (4.2.59)–(4.2.63) with QPSchur and the interfaces that the Hessian and Jacobian matrix objects must support are discussed in [10].

In addition to passing in the matrices and vectors that define the QP, the client can also pass in initial guesses for the solution d (primal variables) and the Lagrange multipliers (dual variables) for the simple bound ν, general inequality μ and general equality λ constraints. Given good estimates for the primal and dual variables, an active-set QP solver can find the solution in very few iterations.

At the time of this writing, *`QPSolverRelaxed`* subclasses have been developed for QPOPT

(`QPSolverRelaxedQPOPT`) [47], QPSOL (`QPSolverRelaxedQPSOL`) [45], QPKWIK (`QPSolver-RelaxedQPKWIK`) [106], LOQO (`QPSolverRelaxedLOQO`) [117] and QPSchur (`QPSolver-RelaxedQPSchur`) [10].

The real variability among different types of QPs is in the form of the Hessian G and Jacobian E and F matrices. By defining a single interface for QP solvers, most of the same code that sets up the QP vectors, calls the solver, and interprets the returned solution can be reused for many different QP solver implementations. Using this QP interface makes it relatively easy to swap QP solvers in and out of rSQP++.

Another major advantage to having a single interface to many different QP solvers is that it was possible to implement a testing class called `QPSolverRelaxedTester`. The method `QPSolverRelaxedTester::check_optimality_conditions(...)` checks the optimality conditions of the QP, defined in (4.2.59)–(4.2.63), given the solution (or partial solution) returned from *`QPSolverRelaxed::solve_qp(...)`*. It is critical to stress how important this testing class is and has been for easing the development of new QP solver subclasses and in regression testing existing QP solvers. In addition, this testing method computes the relative errors in the optimality conditions and is useful in determining how much loss of precision has occurred due to round off and ill conditioning. This helps to diagnose when a QP solver may be unstable or when the QP being solved is very ill conditioned. A lot of work has gone into the development of the `QPSolverRelaxedTester` testing class, and this work can be leveraged whenever a new QP solver implementation is created.

4.3 Configurations for rSQP++

An algorithm configuration object, as shown in Figure 4.2, is required to build a valid rSQP++ algorithm and to initialize it before the algorithm is run. This is where a lot of the complexity involved with a rSQP++ algorithm occurs. The individual step objects used to build the algorithm generally are very compact and perform simpler, well defined tasks. Most of these step objects are built to be fairly autonomous with little specific knowledge about other steps. For the most part, Step objects communicate with each other through the iteration quantities that they have in common. Because the individual Step objects are decoupled, they can be used and reused in many related rSQP++ algorithms. However, as is the case with any non-trivial application, the total complexity of the software is as great or greater than the complexity of the algorithm it is implementing. This increase in overall complexity is unavoidable. What has made object-oriented methods successful in so many areas is that this overall complexity is decomposed into manageable chunks that most of us can comprehend. There is a continuous struggle in software modeling and design between more encapsulation to make entities appear simpler on the outside verses less encapsulation with finer-grained objects that are more flexible but are also harder to deal with and understand as a whole. It is our aim to implement algorithms in rSQP++ that strike a reasonable balance between simplicity and flexibility.

Once an algorithm is configured (i.e. Step objects have been added to the `rSQPAlgo` object, and iteration quantity objects have been added to the `rSQPState` object) it is largely self contained. Automatic garbage collection is used extensively in the form of smart reference counted pointers (see the class `ref_count_ptr<...>` in Section 8.8). These smart pointers allow the algorithm to be modified (Step and iteration quantity objects to be added and removed) with minimal danger of causing a memory leak or other memory usage problem often associated with development in C and C++.

A universal rSQP++ solver encapsulation class called `rSQPppSolver` has been developed that hides many of the details of using a configuration object to setup and algorithm and then solve a NLP. This encapsulation class uses an algorithm configuration class called `rSQPAlgo_ConfigMamaJama` (see Section 4.3.1) as the default but other configurations can be used as well. The class `rSQPppSolver` provides simple access to a rSQP++ solver and should be used by even the most advanced user as the entry point to rSQP++.

Doxygen generated documentation for much of what is discussed here begins in the file

`RSQPPP_BASE_DOC/html/index.html`.

It is important to stress what a radical departure from typical algorithmic implementation methods that this design represents. In a typical numerical code that supports several different options, each part of the algorithm is augmented with “if” statements or “select-case” control structures that implement the logic for the different options. Adding a new option to these types of codes requires adding another “else if” or “case” clause. If the code already supports many different options, then the existing “if” or “select-case” logic may be fairly complex and a developer may be fearful (and rightly so) to add a new option without understanding all of the logic in all of the existing “if” or “select-case” control structures. Now consider the design used for rSQP++. All of the complicated logic used to sort out the user-specified options is contained in the configuration object. However, once the configuration object constructs an algorithm, that algorithm is usually much simpler since it does not have to consider all of the possible options and there are far fewer control structures for different algorithmic options. As a result, it is much easier for a developer to reason about what the algorithm does and how to modify it to meet more specialized needs. All of this can be done without having to know very much at all about the ugly configuration object that was used to configure the algorithm.

4.3.1 MamaJama Configurations

There is a rSQP++ class called `rSQPAlgo_ConfigMamaJama` that is used to configure many related reduced-space SQP algorithms. This single configuration class was used during much algorithm development and continues to be modified and enhanced. The name “MamaJama” was used for a complete lack of something more appropriate and is meant to signify that this is a do-all configuration class. In the future, more specialized rSQP++ algorithms will most likely be modifications of the algorithms constructed by objects of this configuration class or initially based on its source code.

4.3.1.1 Solver options

Various options can be set in a flexible and user friendly format (see the class `OptionsFromStream` in Appendix 8.8). Options are clustered into different “options groups”. An example excerpt from an options file is shown in Appendix 8.8. These and many other options may be included in the `rSQPpp.opt` file.

The full set of options that can be used with `rSQPppSolver` and the “MamaJama” configuration is described in the Doxygen documentation starting in the file

```
RSQPPP_BASE_DOC/ReducedSpaceSQPPack/html/rSQPppSolver*.html
```

Documenting rSQP++ is a major task and this issue is discussed in more detail in the next section.

4.3.1.2 Documentation, Algorithm Description and Iteration Output

One of the greatest challenges in developing software of any kind is in maintaining documentation. This is especially a problem with software developed in a research environment. Without good documentation, software can be very difficult to understand and maintain. In addition to the Doxygen generated documentation, which is very effective in describing interfaces and other specifications, there is also a need to document the more dynamic parts of an optimization algorithm. Highly flexible and dynamic software, which rSQP++ is designed to be, can be very hard to understand just by looking at the source code and static documentation.

A problem that often occurs with numerical research codes is that the algorithm described in some paper is not what is actually implemented in the software. This can cause great confusion later on when someone else tries to maintain the code. Some of these discrepancies are only minor implementation issues while others seriously impact the behavior of the algorithm.

Primarily, two features have been implemented to aid in the documentation of a rSQP++ algorithm: the configured algorithm description can be printed out before the algorithm is run, and information is output about a running algorithm.

The first feature is that a printout of a configured rSQP++ algorithm can be produced by setting the option `rSQPppSolver::print_algo = true` in rSQPpp.opt, where this is shorthand for the `print_algo` option in the `rSQPppSolver` options group. With this option set to `true`, the algorithm description is printed to the `rSQPppAlgo.out` file before the algorithm is run. The algorithm is printed using Matlab-like syntax. The identifier names for iteration quantities used in this printout are largely the same as used in the source code. There is a very careful mapping between the names used in the mathematical notation of the SQP algorithm and the identifiers used in the source code and algorithm printout. This mapping for identifiers is given in Appendix A. Each iteration-quantity name in the algorithm printout has `'_k'`, `'_kp1'` or `'_km1'` appended to the end of it to designate the iteration, (k), $(k+1)$ or $(k-1)$ respectively, for which it was calculated. Much of the difficulty in understanding an algorithm, whether in mathematical notation or implemented in source code, is knowing precisely what a quantity represents. By using a careful mapping of names and identifiers, it is much easier to understand and maintain numerical software.

This algorithm printout is put together by the `rSQPAlgo` object (through functionality in the base class `GeneralIterationPack::Algorithm`) as well as the *`AlgorithmStep`* objects. Each step is responsible for printing out its own part of the algorithm. The code for producing this output is included in the same source file as each of the `do_step(...)` functions for each *`AlgorithmStep`* subclass. Therefore, this documentation is decoupled from other steps as much as the implementation code is, and maintaining the documentation is more urgent since it is in the same source file. An example of this printout for a rSQP algorithm is shown in Appendix 8.8. Each Step object is given a name that other steps refer to it by (to initiate minor loops for instance). Also, the name of the concrete subclass which implements each step is included as a guide to help track down the implementations.

Many of the options specified in the input file are shown in the printed algorithm. The user can therefore study the algorithm printout to see what effect some of the options have. For example, the option `rSQPSolverClientInterface::opt_tol` is shown in step 5 ("CheckConvergence") in Appendix 8.8. Some of the options determine the algorithm configuration, which affects what steps are included, how steps are set up and in what order they are included. These option names are not specifically shown in the algorithm printout. For example, the option `rSQPAlgo_ConfigMamaJama::max_dof_quasi_newton_dense` determines when the algorithm configuration will switch from using dense BFGS to using limited-memory BFGS but this identifier name `max_dof_quasi_newton_dense` is not shown anywhere in the listing. However, the configuration object can print out a short log (to the `rSQPppAlgo.out` file) to show the user the logic for how these options impact the configuration of the algorithm.

In addition to this printed algorithm, output can be sent to a journal file `rSQPppJournal.out` while the algorithm is run to display information about each step's computations. The names given to quantities in the journal output are the same as in the algorithm printout. The level of output is determined by the option `rSQPSolverClientInterface::journal_print_level` and the value `PRINT_ALGORITHM_STEPS` is usually the most appropriate and does not produce excessive output. Lower output levels can be set for generating less output for faster execution times while higher output levels can be set to generate lots of information that is useful in debugging or for other purposes. See Appendix 8.8 for an example of this type of printout.

A more detailed look at the output files `rSQPppAlgo.out` and `rSQPppJournal.out` is given in Section 4.5 in the context of a specific example NLP.

4.3.1.3 Algorithm Summary and Timing

In addition to the more detailed information that can be printed to the file `rSQPppJournal.out`, summary information about each rSQP++ iteration is printed to the file `rSQPppSummary.out`. Also, if the option `rSQPppSolver::algo_timing = true` is set, then this file will also get a summary table of the run-times and statistics for each step. These timings are printed out in tabular format giving the time, in seconds, each step consumed for each iteration as well as the sum of the times of all the steps. The bottom of the table gives step statistics: the total time for each step for all the iterations (`total(sec)`), the average step time per iteration (`av(sec)/k`), the minimum step time (`min(sec)`), the maximum step time (`max(sec)`) and the total percentage of time each step consumed (`%total`). See Appendix 8.8 for an example of a `rSQPppSummary.out` file.

This timing information can be used to determine where the bottlenecks are in an algorithm for a particular NLP. Of course for very small NLPs the runtime is dominated by overhead and not numerical computations so the timing of small problems is not terribly interesting.

Less detailed information can also be printed to the console through the `rSQPppSolver` class (see Appendix 8.8).

A more detailed look at the console output and the output file `rSQPppSummary.out` is given in Section 4.5 in the context of a specific example NLP.

4.3.1.4 Algorithm and NLP Testing and Validation

Many computations are performed in order to solve a nonlinear program (NLP) using a numerical optimization method. If there is a significant error (programming bug or round-off errors) in any step of the computation, the numerical algorithm will not be able to solve the NLP, or at least not to a satisfactory tolerance. When a user goes to solve a user-defined NLP and the optimization algorithm fails or the solution found does not seem reasonable, the user is left to wonder what went wrong. Could the NLP be coded incorrectly? Is there a bug in the optimization software that has gone up till now undetected? For any non-trivial NLP or optimization algorithm it is very difficult to diagnose such a problem, especially if the user is not an expert in optimization. Even if the user is an expert, the typical investigative process is still very tedious and time consuming.

Fortunately, it is possible to validate the consistency of the NLP implementation (i.e. gradients are consistent with function evaluations) as well as many of the major steps of the optimization algorithm. Such tests can be implemented in a way that the added cost (runtime and storage) is of only the same order as the computations themselves and therefore are not prohibitively expensive. There are several possible sources for such errors. These sources of errors, from the most likely to the least likely are errors in the NLP implementation and user specialized parts of the optimization algorithm (e.g. a specialized *`BasisSystem`* object), errors in the core optimization code, or even errors in the compilers or runtime environments used.

There are many ways to make a mistake in coding the NLP interface. For instance, assuming the user's underlying NLP model is valid (i.e. continuous and differentiable), the user may have made a mistake in writing the code that computes $f(x)$, $c(x)$, $\nabla f(x)$ and/or $\nabla c(x)$. Suppose the gradient of the constraints matrix ∇c is not calculated in some regions. The matrix ∇c may be used by a generic *`BasisSystem`* object to find and factor the basis matrix C and therefore, the entire algorithm would be affected. To validate ∇c, the entire matrix could be computed by finite differences of course and then compared to the ∇c computed by the NLP interface, but this would be far too expensive in runtime ($O(nm)$) and storage ($O(nm)$) costs for larger NLPs. Computing each individual component of the gradients by finite differences is an option but it must be explicitly turned on (see the option `NLPFirstDerivativesTester::fd_testing_method`). As a compromise, by default, directional finite differencing can be used to show that ∇c is not calculated properly, but can not strictly prove that ∇c is completely correct. This works as follows. The optimization algorithm asks the NLP interface to compute ∇c_k at the point x_k. Then, at the same point x_k, for a random vector v, the matrix-vector product $\nabla c(x_k)v$ is approximated, using central finite differences for instance, as $\nabla c(x_k)v \approx t_1 = (c(x_k + hv) - c(x_k - hv))/2h$ where $h \approx 10^{-5}$. Then the matrix vector product $t_2 = \nabla c_k v$ is computed using the ∇c_k matrix object computed by the NLP interface and the resultant vectors t_1 and t_2 is then compared. Even if the user does an exemplary job of implementing the NLP interface, the computed t_1 and t_2 vectors will not be exactly equal (i.e. $t_1 \neq t_2$) due to unavoidable round-off errors. Therefore, we need some type of measure of how well t_1 and t_2 compare. For every such test in rSQP++ there are defined error (`error_tol`) and warning (`warning_tol`) tolerances that are adjustable by the user but are given reasonable default values. Any relative error greater than `error_tol` will cause the optimization algorithm to be terminated with an error message. Any relative error greater than `warning_tol` will cause a warning message to be printed to the journal file to warn the user of some possible problems. For example, relative errors greater than `warning_tol` = 10^{-12} but smaller than `error_tol` = 10^{-8} may concern us, but the algorithm still may be able to solve the NLP. The finite-difference testing of the NLP interface can be controlled by setting options in the `NLPFirstDerivativesTester` and `CalcFiniteDiffProd` options groups as shown in Appendix 8.8. Testing the NLP's interface at just one point, such as the initial guess x^0, is not sufficient to validate the NLP interface. For example, suppose we have a constraint $c_{10}(x) = x_2^3$ with $\partial c_{10}/\partial x_2 = 3x_2^2$. If the derivative was coded as $\partial c_{10}/\partial x_2 = 3x_2$ by accident, this would appear exactly correct at the points $x_2 = 0$ and $x_2 = 1$ but would not be correct for any other values of x_2. Therefore, it is important to test the NLP interface at every SQP iteration if one really wants to validate the NLP interface. Of course, just because the NLP interface is consistent, does not mean it implements the model the user had in mind, but this is a different matter. If the NLP is unbounded, infeasible or otherwise ill posed, the SQP algorithm will determine this (but the error message produced by the algorithm may not be able to state exactly what the problem is).

Every major computation in a rSQP algorithm can be validated, at least partially, with little extra cost. For example, an interface that is used to solve for a linear system $x = A^{-1}b$ such as the *`MatrixWithOpNonsingular`* can be checked by computing $q = Ax$ and then comparing q to b. The interfaces can also be validated for the null-space decomposition (see `Decomposition-SystemTester` in Section 4.2.5) and QP solver (see `QPSolverRelaxedTester` in Section 4.2.6) objects. Since sophisticated users can come in and replace any of these objects, it is a good idea to be able to test everything that can realistically be tested whenever the correctness of the algorithm is in question or new objects are being integrated and tested. Much of this testing code is already in place in rSQP++, but more is needed for more complete validation.

Such careful testing and validation code can save a lot of debugging time and also help avoid reporting incorrect results which can be embarrassing in an academic research setting or costly in business setting. Testing and validation is no small matter and should be taken seriously, especially in a dynamic environment with lots of variability like rSQP++.

4.3.1.5 Debugging

Whenever software is involved, the need for debugging is unavoidable. When a new user attempts to solve a NLP using rSQP++, the most likely bugs will be in the NLP implementation that the user has to provide. Here, some strategies for debugging are discussed that should help a user to track down bugs associated with the NLP implementation and fix them as quickly as possible. There are many different types of errors that can occur and going into all of these types of errors would require a long discussion. However, below are a few of the more common types of errors that are worth mentioning.

1. Segmentation fault do to runtime memory management error.
2. A linear algebra incompatibility exception is thrown.
3. Gradients of problem functions do not match function values (i.e. finite-difference testing failed).
4. Algorithm prematurely terminated due to some algorithmic error.
5. Unexpected or unreasonable solution is found.

Segmentation faults or thrown exceptions are some of the easiest (or the hardest) bugs to track down. These are almost always caused by some programming error and are not related to the validity of the mathematical formulation for the NLP being implemented. The other errors are harder to track down and are usually caused by a malformed NLP.

The easiest of these errors to track down is when a gradient of the objective or constraints does not match the function value to an acceptable tolerance. It is this type of error that is discussed here. Debugging a large NLP with lots of variables and constraints is generally very difficult. Therefore, serious debugging should be performed on the smallest and simplest example that does not exhibit the correct or expected behavior. For example, the smallest possible mesh size and discretization method should be used for a scalable NLP such as a PDE solver using the finite-element method. Assuming that a problem can be derived that is sufficiently small (i.e. $n, m < 20$) here are the steps to follow in order to diagnose a problem with the NLP formulation. First, the initial point for the NLP needs to be dumped to the file `rSQPppJournal.out` and each component of the gradient has to be checked independently (i.e. component-wise). To do this, set the options `NLPTester::print_all=true` and `NLPFirstDerivativesTester-::fd_testing_method=FD_COMPUTE_ALL`. This will cause the print out of the initial guess x^0 (`xinit`), the bounds x_L (`xl`), x_U (`xu`), the value of the objective $f(x^0)$ (`f`), constraints $c(x^0)$ (`c`), the gradients of the objective $\nabla f(x^0)$ (`Gf`), constraints $\nabla c(x^0)$ (`Gc`) and the relative error in every gradient component. From this information it will be easy to see which component of $\nabla f(x^0)$ or $\nabla c(x^0)$ is causing the problem.

4.4 Examples NLP subclasses

There are several example NLP projects that come with the base distribution of rSQP++. Several of the included example projects implement the following simple NLP

$$\begin{aligned} \text{min} \quad & {}^1\!/\!_2 x^T x && (4.4.64) \\ \text{s.t.} \quad & c_j = x_j(x_{(j+n/2)} - 1) - 10x_{(j+n/2)} = 0, \quad \text{for } j = 1 \ldots n/2. && (4.4.65) \end{aligned}$$

This scalable NLP has $(n - m) = n/2 = m$ degrees of freedom and is referred to as example #2 in [115] and [104]. This NLP has very specialized structure and a valid selection of dependent and independent variables is straightforward to find. Selecting the first m variables as dependent variables gives the following definitions of the basis and nonbasis matrices

$$C = \begin{bmatrix} x_{m+1} - 1 & & & \\ & x_{m+2} - 1 & & \\ & & \ddots & \\ & & & x_{m+m} - 1 \end{bmatrix} \quad (4.4.66)$$

$$N = \begin{bmatrix} x_1 - 10 & & & \\ & x_2 - 10 & & \\ & & \ddots & \\ & & & x_m - 10 \end{bmatrix} \quad (4.4.67)$$

which both happen to be diagonal matrices. Also, the exact Hessian of the Lagrangian W and the reduced Hessian of the Lagrangian B (using a variable-reduction decomposition) take the simple forms

$$W = \begin{bmatrix} I & \Lambda \\ \Lambda^T & I \end{bmatrix} \tag{4.4.68}$$

$$B = N^T C^{-T} C^{-1} N - \Lambda C^{-1} N - N^T C^{-T} \Lambda + I \tag{4.4.69}$$

where Λ is a diagonal matrix with components $(\Lambda)_{(j,j)} = \lambda_{(j)}$ for $j = 1 \ldots m$. Note that the reduced Hessian B in (4.4.69) is also a diagonal matrix.

This NLP and its specific structure are of no practical interest but this NLP is sufficient as a simple example to show how rSQP++ can be used to fully exploit the structure of a class of NLPs from a specialized application area.

Three different implementations of this NLP are described. The first NLP subclass is derived from the generic `NLPSerialPreprocessExplJac` node subclass. This example NLP subclass is included to show how this generic NLP interface subclass can be used and to provide a contrast to the more specialized implementations. The last two NLP subclasses derive directly from the *`NLPFirstOrderInfo`* and *`NLPFirstOrderDirect`* interfaces and demonstrate how to exploit the structure and properties of a NLP.

This first NLP subclass is called `ExampleNLPSerialPreprocessExplJac` and the source code for this project can be found at

`$RSQPPP_BASE_DIR/rSQPpp/examples/ExampleNLPSerialPreprocessExplJac`.

The files `ExampleNLPSerialPreprocessExplJac.h` and `ExampleNLPSerialPreprocess-ExplJac.cpp` contain the declarations and definitions for the NLP subclass and the file `Example-NLPSerialPreprocessExplJacMain.cpp` contains the simple driver program that uses a `rSQPppSolver` object to solve the NLP.

The second two NLP subclasses are called `ExampleNLPFirstOrderInfo` and `Example-NLPFirstOrderDirect`. Both of these subclasses derive from a node subclass `Example-NLPObjGradient` which implements the bulk of the common functionality. The `Example-NLPObjGradient` subclass takes a *`VectorSpace`* object as an argument in its constructor. Using this single *`VectorSpace`* object this entire NLP's implementation can be defined. This vector space is used to define the spaces $\mathcal{X}_D$, $\mathcal{X}_I$ and $\mathcal{C}$ which happen to all be the same for this NLP. A composite vector-space object of type `VectorSpaceCompositeStd` is used for the space $\mathcal{X} = \mathcal{X}_D \times \mathcal{X}_I$. A Specialized `RTOp` operator is used to implement the the constraints residual computation in (4.4.65). Note that the objective in (4.4.64) is simply a dot product for which a default `RTOp` operator already exists.

The NLP subclass `ExampleNLPFirstOrderInfo` derives from *`NLPFirstOrderInfo`* and `ExampleNLPObjGradient`. A specialized *`BasisSystem`* subclass called `ExampleBasisSystem` derives from the standard basis-system subclass `BasisSystemCompositeStd`. The `BasisSystemCompositeStd` subclass implements the *`BasisSystem`* interface for the case where the matrix object `Gc` is simply an aggregate of a *`MatrixWithOpNonsingular`* matrix object for `C` and a *`MatrixWithOp`* matrix object for `N`. For the NLP, the standard matrix subclass `MatrixSymDiagonalStd` is used for the matrices C and N since they are diagonal. The only functionality that the `ExampleBasisSystem` subclass adds is the specialized formation of the direct sensitivity matrix $D = -C^{-1}N$ which is also diagonal for this simple NLP and is also represented using a `MatrixSymDiagonalStd` object. The computation of the diagonal vectors for C and N is also performed by a specialized `RTOp` operator object. The complete source code for this example can be found at

`$RSQPPP_BASE_DIR/rSQPpp/examples/ExampleNLPFirstOrderInfo`.

The last NLP subclass `ExampleNLPFirstOrderDirect` derives from *`NLPFirstOrderDirect`* and `ExampleNLPObjGradient`. This subclass implements the *`calc_point(...)`* method to compute the diagonal direct-sensitivity matrix $D = -C^{-1}N$. Again, this direct-sensitivity matrix is implemented as a `MatrixSymDiagonalStd` object. For complete source code, see the directory

`$RSQPPP_BASE_DIR/rSQPpp/examples/ExampleNLPFirstOrderDirect`.

Since the interfaces *`NLPFirstOrderInfo`* and *`BasisSystem`* can be implemented easily for the NLP in (4.4.64)–(4.4.65) there was really no practical purpose for implementing the *`NLPFirstOrderDirect`* interface since it provides only a subset of the functionality. The only purpose for implementing the `ExampleNLPFirstOrderDirect` subclass was to provide a simple complete example for the *`NLPFirstOrderDirect`* interface.

All of the linear algebra for these NLP subclass is based on a single *`VectorSpace`* object as mentioned above. Therefore, any valid *`VectorSpace`* object can be used along with the vectors it creates. As a result, serial, parallel or other vector implementations can easily be used. These NLP subclasses have been used various serial and parallel vector implementations.

n	$n - m$	N_p	Wall Clock Time (sec)	Scalability
2,000	1,000	1	0.21	1.00
2,000	1,000	2	0.27	2.57
2,000	1,000	4	0.53	10.10
20,000	10,000	1	1.50	1.00
20,000	10,000	2	0.96	1.28
20,000	10,000	4	0.75	2.00
200,000	100,000	1	21.00	1.00
200,000	100,000	2	11.00	1.05
200,000	100,000	4	5.60	1.07
2,000,000	1,000,000	1	190.00	1.00
2,000,000	1,000,000	2	93.00	0.97
2,000,000	1,000,000	4	47.00	0.98

Table 4.1. CPU times and scalability for the example NLP in (4.4.64)–(4.4.65) where N_p is the number of processors and 'Scalability' is the wall-clock CPU time multiplied by the number of processors divided by the CPU time for one processor.

Table 4.1 shows the CPU times and scalabilities for using an example parallel `VectorSpace` class (using MPI) on a distributed-memory Beowulf cluster. The example NLP was run with bad initial guesses and the number of rSQP iterations was cut off at 100 in order to get consistent timings. The rSQP algorithm used a limited-memory BFGS approximation [31] with very good parallel scalability. As a result, all of the linear algebra computations for this simple NLP are all fully scalable. Here we define *scalability* as the ratio of the wall-clock CPU time multiplied by the number of processors divided by the wall-clock time for running the problem on only one processor. Given this definition, perfect scalability is 1.00 which simply means that if we double the number of processors, the best that we can usually hope for is to have the wall-clock time halved. The timings in Table 4.1 are typical for scalable parallel programs. When the amount of computation verses communication is small, the communication tends to dominate which is seen for vectors of size $m = n - m = 1,000$ where there is actually an overall slowdown as more processors are utilized. However, for vectors of size $m = n - m = 10,000$ we see a definite speedup as more processors are added but the scalability is less than perfect. When the size of the vectors are increased to $m = n - m = 100,000$, the algorithm shows almost perfect scalability. Note that 25,000 unknowns per processors (i.e. for $N_p = 4$) is considered small for PDE simulators that use parallel iterative solvers. Finally, for very large vectors of size $m = n - m = 1,000,000$, the timings show better than perfect scalability (i.e. $0.97 < 1.00$) which can also be seen in other parallel programs from time to time (usually do to cache or other hardware issues).

Note that all of the linear algebra operations for this simple example NLP are vector operations which offer the worst computation to communication ratios. Therefore, these results represent the worst-case scenario for rSQP++ with respect to parallel scalability. For more practical applications, the amount of computation per process is much higher and therefore these applications show better overall scalability for smaller problem sizes.

These results show that the rSQP++ framework imparts very little serial overhead and therefore allows for the implementation of very scalable optimization algorithms for application areas where parallelism can be exploited (e.g. PDE constrained optimization). Therefore, the burden is completely on the developers of applications and parallel linear algebra libraries to achieve scalability.

4.5 Detailed Descriptions of Input and Output Files

In this section, a detailed description of the input and output to rSQP++ is given. Here it is assumed that a NLP subclass is developed and a driver program has been written as explained in the examples (see Appendix B for a description of adding a new project to the build system). For this discussion, we will use the included example NLP called `ExampleNLPBanded` which project is located at

`$RSQPPP_BASE_DIR/rSQPpp/examples/ExampleNLPBanded`.

This is a fairly simple NLP that is designed to allow the independent scaling of n and m so that basic serial algorithm scalabilities can be tested. For a more detailed description of this NLP see the Doxygen generated documentation at

`RSQPPP_BASE_DOC/ExampleNLPBanded/html/index.html`.

Before solving this NLP a working directory needs to be created to store the input and output files as follows

```
$ mkdir $RSQPPP_BASE_DIR/tests/ExampleNLPBanded
$ cd $RSQPPP_BASE_DIR/tests/ExampleNLPBanded
```

The next step is to create a symbolic link to the prebuilt executable. Assuming the test suite for the release version was built this link can be created as follows

```
$ ln -s $RSQPPP_BASE_DIR/intermediate/ExampleNLPBanded/
  release/solve_example_nlp .
```

The options file `rSQPpp.opt` needs to be created (using `emacs` for instance) as shown in Appendix 8.8. Note that most of the options are commented out and most of those that are not are at the default values.

Executing NLP creates output to the console and the output files `rSQPppAlgo.out`, `rSQPppSummary.ou` and `rSQPppJournal.out` which are shown in Appendix D.

4.5.1 Output to Console

The console output shown in Appendix 8.8 is generated by a default *`AlgorithmTrack`* object of type `rSQPTrackConsoleStd` which is automatically inserted by the `rSQPppSolver` object. The first thing printed is the size of the NLP where `n = 30400` is the total number of variables, `m = 30000` is the total number of equality constraints and `nz = 599910` is the number of nonzeros in the Jacobian ∇c (`Gc`) for this example. Next, a table containing summary information for each iteration is printed. Each column in this table has the following meaning

- **`k`** : The SQP iteration count. This count starts from zero so the total number of SQP iterations in one plus the final `k`.

- **`f`** : The value of the objective function $f(x)$ at current estimate of the solution x_k

- **`||c||s`** : The scaled residual of the norm of the equality constraints $c(x)$ at current estimate of the solution x_k. The scaling is determined by the convergence check (see step 6 in Appendix 8.8 & 8.8) and this value is actually equal to the iteration quantity `feas_kkt_err` (see the file `rSQPppAlgo.out`). This is the error that is compared to the tolerance `rSQP-SolverClientInterface::feas_tol` in the convergence check. The unscaled constraint norm can be viewed in the more detailed iteration summary table printed in the file `rSQPppSummary.out`.

- **`||rGL||s`** : The scaled norm of the reduced gradient of the Lagrangian $Z^T \nabla_x L$ at current estimate of the solution x_k. The scaling is determined by the convergence check (see step 6 in Appendix 8.8 & 8.8) and this value is actually equal to the iteration quantity `opt_kkt_err` (see the file `rSQPppAlgo.out`). This is the error that is compared to the tolerance `rSQP-SolverClientInterface::opt_tol` in the convergence check. The unscaled norm can be viewed in the more detailed summary table printed in the file `rSQPppSummary-.out`.

- **`QN`** : This field indicates whether a quansi-Newton update of the reduced Hessian was performed or not. The following are the possible values:

  - **`IN`** : Reinitialized (usually to identity I)
  - **`DU`** : A dampened update was performed
  - **`UP`** : An undamped update was performed
  - **`SK`** : The update was skipped on purpose
  - **`IS`** : The update was skipped because it was indefinite

- **`#act`** : Number of active constraints in the QP subproblem. This field only has meaning for an active-set algorithms. For interior-point algorithms, this will just equal the number of bounded variables and does not provide any interesting information.

- **`||Ypy||2`** : The $||.||_2$ norm of the quasi-normal contribution $(Yp_y)_k$. This norm gives a sense of how large the feasibility steps are.

- **`||Zpz||2`** : The $||.||_2$ norm of the tangential contribution $(Zp_z)_k$. This norm gives a sense of how large the optimality steps are.

- **`||d||inf`** : The $||.||_\infty$ norm of the total step $d_k = (Yp_y)_k + (Zp_z)_k$. This norm gives a sense of how large the full SQP steps are in x.

- **`alpha`** : The step length taken along $x_{k+1} = x_k + \alpha d_k$. A step length of $\alpha = 0$ represents a major event in the algorithm such as a line search failure followed by the selection of a new basis or a QP failure followed by a reinitialization of the reduced Hessian. A small number for α indicates that many backtracking line search iterations where required and is an indication that the computed search direction d_k is a poor direction.

After the iteration summary is printed, the CPU time is given in `Total time`. This is the CPU time that is consumed from the time that the `rSQPTrackConsoleStd` object is created up until the time that the final state of the algorithm is reported. Therefore, this CPU time may contain more than just the execution time of the algorithm. For more detailed built-in timings, see the table at the end of the file `rSQPppSummary.out`.

Following the total runtime, the number of function and gradient evaluations is given for the objective and the constraints (i.e. 96 evaluations of $f(x)$ and $c(x)$ and 15 evaluations of $\nabla f(x)$ and $\nabla c(x)$). Note that the reason there is an excessive number of function evaluations is that the options `rSQPppSolver::test_nlp = true` and `rSQPSolverClientInterface::check-_results = true` are being used which results in many finite-difference computations for various tests. The results from some these tests are shown in the file `rSQPppJournal.out` in Appendix 8.8. If these options are set to `false` then the number of function evaluations come down to only 20 for this example NLP.

Below, the major types of output that are written to each output file are discussed. The purpose of this discussion is to familiarize the user with the contents of these files and to give hints of where to look for a certain types of information. Much of the output produced by rSQP++ is omitted from the files included in Appendix 8.8–8.8 for the sake of space.

Before going into the details of each individual file, first a few general comments are made. At the top of every output file is a header that briefly describes the general purpose of the output file. This header is followed by an echo of the options form the `OptionsFromSteam` object. These options include those set in the input file `rSQPpp.opt` or by some other means (e.g. in the executable or on the command line). The purpose of echoing the options in each file is to help record what the setting were that were used to produce the output in the file. Of course the output is also influenced by other factors (e.g. other command-line options, properties of the specific NLP being solved etc.) and therefore these options do not determine the complete behavior of the software.

4.5.2 Output to `rSQpppAlgo.out`

After the initial header and the echoed options

```
********************************************************************
*** Algorithm information output                                 ***
***                                                              ***
*** Below, information about how the the rSQP++ algorithm is     ***
*** setup is given and is followed by detailed printouts of the  ***
*** contents of the algorithm state object (i.e. iteration       ***
*** quantities) and the algorithm description printout           ***
*** (if the option rSQPppSolver::print_algo = true is set).      ***
********************************************************************

*** Echoing input options ...

...
```

the concrete type of the configuration object is printed (in this case `'class ReducedSpaceSQPPack::rSQPAlgo_ConfigMamaJama'`) followed by a header produced by the configuration object it self. The next few lines of output simply traces some of the tasks the configuration object performs. For example, the line

```
Detected that NLP object supports the NLPFirstOrderInfo interface!
```

states that the configuration object has detected that the user's NLP supports the *`NLPFirstOrderInfo`* interface which will determine what type of algorithm will be configured. This detection is performed using the build-in `dynamic_cast<...>` C++ operator.

The next bit of output gives the logic for how the configuration object decides which features to use with the given NLP. For example, the output

```
range_space_matrix == AUTO:
(n-r)^2*r = (400)^2 * 30000 = 505032704 > max_dof_quasi_newton_dense^2 = (500)^2 = 250000
setting range_space_matrix = COORDINATE
```

shows that the $O((n-r)^2 r)$ flops required for the orthogonal variable-reduction null-space decomposition exceeds number of flops for the dense quasi-newton update and therefore the coordinate decomposition will be used. Similar logic is used to determine if dense quasi-Newton or a limited-memory approximation will be used by the algorithm.

Later in the file, the output line

```
Configuring an algorithm for a nonlinear equality constrained NLP ( m > 0 && mI == 0 && num_bounded_x == 0) ...
```

states that an algorithm will be configured for a NLP without any inequality constraints.

This type of output shows how a configuration object can tailor the algorithm it constructs to the specific demands of the NLP being solved. This is a fundamental difference from the way that most numerical software is written. In most numerical software, the code is written with switch statements for every possible option that is supported, making the code hard to develop and understand. In rSQP++, the complexity of supporting a large set of options is first-and-foremost handled by different object configurations. No matter how complex the logic is that is used to setup an algorithm, the resultant configured algorithm becomes a much simpler self-contained entity that is easier to understand.

The remainder of the `rSQPppAlgo.opt` file gives details on the configured algorithm. The first bit of information is a list of step objects that the `rSQPppAlgo` object is configured with along with the names of the concrete classes used to implement the steps. This output begins with

```
*** Algorithm Steps ***

1. "EvalNewPoint"
    (class ReducedSpaceSQPPack::EvalNewPointStd_Step)

...
```

This list of Step objects is followed by a listing of the iteration quantities

```
*** Iteration Quantities ***

...
```

that have been added to the `AlgorithmState` object. These iteration quantities are of more interest to algorithm developers but they also show the list of possible iteration quantities that an advanced user could query in a user-defined `AlgorithmTrack` object that is passed to the `rSQPppSolver` object.

Near the end of the `rSQPppJournal.out` file is a fairly detailed description of the configured algorithm, step-by-step, in a Matlab-like format. The purpose of this algorithm description is to document the major aspects of the algorithm in a way that the user (or algorithm developer) should be able to reason about the implemented algorithm. A short sub-algorithm is output for each step object. Each step object shows all of the iteration quantities that it accesses and updates. For example, the null-space contribution `Zpz_k` is computed first in step {`7. "NullSpaceStep"`} before it can be used to compute the full direction `d_k` in step {`8. "CalcDFromYPYZPZ"`}. This type of information is very helpful in determining what order quantities must be computed in and what the dependencies are.

The last step in the algorithm printout is always the step `"Major Loop"` which is an implicit step that simply states the logic build in to the `Algorithm` class for performing the major loop (i.e. transitioning form k to $k+1$) and in prematurely terminating the algorithm if the maximum number of iterations or the maximum runtime is exceeded.

The very last part of this file contains the following

```
***************************************************************
Warning, the following options groups where not accessed.
An options group may not be accessed if it is not looked for
or if an "optional" options group was looked from and the user
spelled it incorrectly:
```

Here, the name of any option group that was specified in the file `rSQPpp.opt` (or by some other means) that was not read by some object is printed. In this example, all of the specified options groups where read by at least one object during algorithm configuration. The purpose of this printout is to show any options groups that may have been spelled incorrectly or were just not read for some reason. None of the option from any of these printed options groups had any influence on the algorithm what so ever. This information helps a user to avoid the frustrating situation where an option is changed but the algorithm runs unaltered. If there is ever any question as to why an option did not seem to have the desired effect, this output in the the texttttrSQPpp-Algo.out file is the first place to look for an explanation.

4.5.3 Output to `rSQpppSummary.out`

The file `rSQpppSummary.out` is usually the first place to look (after the console output as described above) to investigate the runtime behavior of a configured algorithm.

After the initial header and echoed options are printed the results of the NLP testing (if `rSQPpp-Solver::test_nlp=true`) is given. This is followed by a table where each line is a summary of each iteration. Each column of this table is described in the Doxygen documentation for the track class `rSQPSummaryStd`. The summary table is followed by a printout of the number of function evaluations and the total solution time (just as in the console output).

The `rSQpppSummary.out` file also produces a table (if `rSQPppSolver::print_algo = true`) of the CPU times per step, per iteration. This output begins with the following header and a list of major steps

```
**************************************
*** Algorithm step CPU times (sec) ***

Step names
----------
1) "EvalNewPoint"
...
```

These are the same steps that are printed in the `rSQPppAlgo.out` file.

The table that follows this makes it easy to determine which step objects and which computations are consuming the most CPU time. This type of gross timing is very important in determining where the bottlenecks are occurring and what steps require the most attention for a particular NLP. Note that the information produced in this table supplements traditional profile timings that are produced but tools like `gprof`. For example, the same linear solver may be called in several different steps and the profiler output may make it difficult to determine in what steps most of the solves where being performed. In this example NLP, for the options used, the bulk of the runtime (83.93%) is consumed by the step {`1) "EvalNewPoint"`}. By looking in the `rSQPpp-Algo.out` file, it is easy to see that the major computations in this step is the evaluation of the functions and the gradients of the NLP and the formation of the decomposition matrices. By comparing iterations `k=0` and `k=1` one can see that the runtime drops dramatically from 18.96 seconds to only 2.xxx seconds for subsequent iterations. Therefore, one could quickly infer that the initialization that goes on in this step is quite significant. Further investigation would reveal that the dominate time in this step is consumed by the direct sparse solver (MA28 in this case) and the initial analyze-and-factor used to select the basis is a dominate cost.

4.5.4 Output to `rSQpppJournal.out`

The output file `rSQpppJournal.out` contains detailed, step-by-step, iteration-by-iteration output for a running algorithm. The algorithm description in the output file `rSQPppAlgo.out` is very helpful in understanding the journal output. Depending on the output level for the option `rSQPSolverClientInterface::journal_print_level` set in `rSQPpp.opt` this output can be fairly minimal (i.e. `PRINT_ALGORITHM_STEPS`) or dump everything (i.e. `PRINT-_ITERATION_QUANTITIES`). The output shown in Appendix 8.8 is the output level `PRINT-_ALGORITHM_STEPS` and therefore the amount of output is independent of the NLP size which is usually the most appropriate level (unless debugging). For small NLPs, setting the level to `PRINT_ITERATION_QUANTITIES` usually produces enough output for debugging that opening and debugger is unnecessary in many cases.

After the header and the echoed options are printed, the trace from the initial NLP testing is given (if `rSQPppSolver::test_nlp=true`). The first part of the testing output is the basic tests on the *`VectorSpace`* objects returned from the *`NLP`* interface. More detailed output for these vector-space tests can be produced by setting options in the options group `VectorSpace-Tester` (see the Doxygen documentation). Following the basic tests of the vector-space objects and the vector objects (which are created by the vector-space objects) are finished, other simple tests are performed which basically comprise a unit test for the *`NLP`* interface. Following this simple unit test, the derivative objects `Gf` and `Gc` computed by the *`NLPFirstOrderInfo`* are checked against the functions `f` and `c` using directional finite differencing. This output shows the following relative errors for a single random direction

```
rel_err(Gf'*y,FDGf'*y) = rel_err(6.53040559e+002,6.53040559e+002) = 1.93477565e-011

rel_err(sum(Gc'*y),sum(FDGc'*y)) = rel_err(2.20905038e+008,2.20905038e+008) = 1.37878129e-013
```

This output shows that the finite-difference directional products agree with the analytic directional products by approximately 10 and 12 significant digits for `Gf` and `Gc` respectively. Such a high accuracy for the finite-difference products is a result of the fourth-order four-point finite differencing that is used by default. To set different (and cheaper) finite-differencing strategies see the options group `CalcFiniteDiffProd` (see Appendix 8.8).

After the initial NLP testing completes (successfully), the rSQP algorithm is started with the line

```
*** Starting rSQP iterations ...
```

Most of the output produced for this example NLP is omitted for the sake of space and the output that is included is used to point out several important items.

First note that each step prints out some basic logic and some information for most of the iteration quantities that are computed. For example, `"EvalNewPoint"` prints out the objective function value `f_k` and the infinitely norms of the gradient of the gradient of the objective `Gf_k` and the constraints `c_k`. The number of significant digits printed for floating point numbers in the journal output is controlled by the option `rSQPSolverClientInterface::journal-_print_digits` (which is 6 by default).

Note that if `rSQPSolverClientInterface::check_results=true` that the `"EvalNewPoint"` step will perform finite-difference tests of the NLP gradients for each rSQP iteration. Also note that the results are slightly different than for the initial NLP testing since a different random directional vector is generated. This time the relative error for the gradient of the objective `Gf` is greater than the default warning tolerance of `NLPFirstDerivativesTester::warning_tol=1e-10`. This resulted in the following warning being printed

```
For Gf, there were 1 warning tolerance
violations out of num_fd_directions = 1 computations of FDGf'*y and
the maximum violation was 4.408797e-010 > Gf_waring_tol =
1.000000e-010
```

If the relative error had been greater than `NLPFirstDerivativesTester::error_tol`, then an error message would have been printed and the algorithm would have been terminated. For some difficult ill-conditioned NLPs, the finite-difference tests may fail even though there is not a programming bug. Either the error tolerance can be increased or the tests can be turned of all together in these cases.

The last important detail to point out is the convergence check in step {`5: "CheckConvergence"`}. The output

```
(0) 5: "CheckConvergence"

scale_opt_factor = 1.000000e+000 (scale_opt_error_by = SCALE_BY_ONE)
scale_feas_factor = 1.000000e+000 (scale_feas_error_by = SCALE_BY_ONE)
scale_comp_factor = 1.000000e+000 (scale_comp_error_by = SCALE_BY_ONE)
opt_scale_factor = 1.100000e+001 (scale_opt_error_by_Gf = true)
opt_kkt_err_k    = 1.230623e+002 > opt_tol  = 1.000000e-008
feas_kkt_err_k   = 1.208973e+007 > feas_tol = 1.000000e-010
comp_kkt_err_k   = 0.000000e+000 < comp_tol = 1.000000e-006
step_err         = 0.000000e+000 < step_tol = 1.000000e-002

Have not found the solution yet, have to keep going :-(
```

shows exactly how optimality and feasibility errors are computed and how they are compared to the convergence tolerances `opt_tol` and `feas_tol` that are set in the options group `rSQPSolverClientInterface`. See the step `"CheckConvergence"` in the printed algorithm description in the file `rSQPppAlgo.out` in Appendix 8.8 for the details on how each of these quantities are computed and compare these computed errors to the columns `||c||s` and `||rGL||s` in the console output as shown in Appendix 8.8.

The finial convergence check in iteration `k=13` shows the final KKT errors

```
opt_kkt_err_k    = 3.273859e-012 < opt_tol  = 1.000000e-008
feas_kkt_err_k   = 1.518593e-012 < feas_tol = 1.000000e-010
comp_kkt_err_k   = 0.000000e+000 < comp_tol = 1.000000e-006
step_err         = 0.000000e+000 < step_tol = 1.000000e-002

Jackpot!  Found the solution!!!!!! (k = 13)
```

and then the algorithm is terminated and the optimal solution is communicated to the NLP object.

Chapter 5

MPSalsa/rSQP++ Interface and Results

5.1 Introduction

Our first prototyping project consisted of interfacing a rSQP algorithm to a chemically reacting flow simulator in an attempt to solve an optimization problem for a Chemical Vapor Deposition (CVD) reactor. We selected chemically reacting fluid flow because both the simulation and optimization have very large-scale potential. In addition this problem did not require transient modeling. The initial design problem involved only a single velocity value as the design parameter.

Considering that very little information exists about interfacing rSQP algorithms to large and massively parallel production codes, the primary goal was to identify issues associated with interfacing intrusive algorithms to existing parallel production codes. Our strategy was to start as simple as possible and then consider higher levels of optimization. We therefore started with the direct approach. (level 4). We were not able to completely develop the adjoint interface but we could not conveniently solve the transpose Jacobian matrix within the code.

Small number of design variables have been tested in serial and parallel. For the parallel implementation, rSQP is duplicated on each process and that causes limited scalability. rSQP++ has since then been modified and has demonstrated good scalability as shown in table 4.1.

5.2 CVD Reactor Optimization Problem

The rotating disk reactor is a common configuration for performing Chemical Vapor Deposition (CVD) of thin films, including many important semiconducting materials. The optimization problem formulated in this paper is generated from the work of Sandia researchers attempting to improve the design of the inlet of a rotating disk CVD reactor for use in growing thin films of Gallium Nitride (GaN). GaN is used in blue light emitting diodes and other photonic devices. The quality of the electronic device is highly dependent on the uniformity of the growth rate at different positions in the reactor. We are attempting to use simulations and optimization algorithms to determine if a new reactor, designed with a restricted inlet for reducing the costs of reactant gases, can achieve highly uniform GaN film growth.

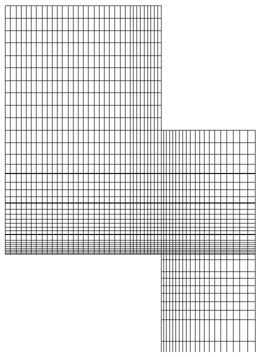
The finite element mesh for the base shape of the reactor is shown in Figure 5.1(a).

This is an axisymmetric (2D) model, where the left side is the axis of symmetry. A mixture of trimethylgallium, ammonia, and hydrogen gases ($Ga(CH_3)_3$, NH_3, and H_2) enter the top of the reactor, flow over the disk, which is heated, and then flow down the annular region out the bottom of the mesh. At the heated disk, the $Ga(CH_3)_3$ and NH_3 react to deposit a GaN film and release three molecules of methane (CH_4). This simplified mechanism has been shown to work well in modeling GaN film uniformities since the growth rate is predominantly transport limited [88]. This mesh depicts a restricted inlet design, where the top of the reactor has a smaller radius than the lower part of the reactor.

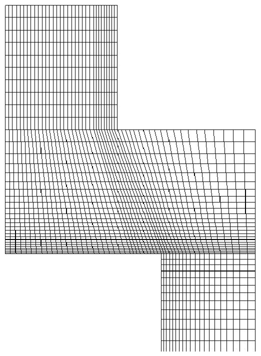
The main parameter used in this paper is the inlet velocity of the gases, V. Two additional parameters in this model define the shape of the inlet, namely the Shoulder Radius and Shoulder Height, which define the position where the mesh transitions from the inlet radius to the larger reactor radius. The mesh is moved algebraically and continuously as a function of these geometric design parameters. Figure 5.1(b) shows how the mesh changes for a decreased shoulder radius, and Figure 5.1(c) shows how the mesh deforms continuously for larger values of the shoulder radius and shoulder height. If the optimum occurs too far away from where the initial mesh is generated, it would be appropriate to remesh the new geometry from scratch.

The objective function measures the uniformity of the growth rate of GaN over the disk. We chose an L^2 norm over an L^{inf} norm so that the objective is continuous and has a continuous derivative. Since the L^2 norm had very small values over a range of parameters, the log was taken. The final form of the objective function is

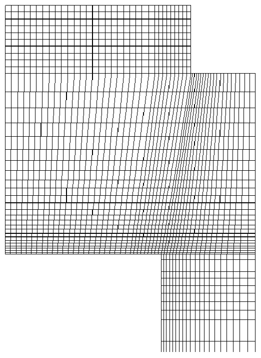
$$\text{Objective Function} = F = log(SD + 10^{-10}) \tag{5.2.1}$$



(a)



(b)



(c)

Figure 5.1. Three different meshes for the restricted inlet design of the rotating disk reactor are shown: (a) the baseline case mesh where the shoulder radius is above the edge of the disk and the height is half of the inlet height; (b) a mesh when the shoulder radius parameter is decreased; (c) a mesh where the shoulder radius and height are both increased above the base case.

where SD is the standard deviation squared and is defined as

$$SD = \frac{1}{N_n} \sum_{i=1}^{N_n} (\frac{g_i - g_{ave}}{g_{ave}})^2. \tag{5.2.2}$$

Here N_n is the number of nodes on the surface, g_i is the growth rate of GaN at node i, and g_{ave} is the average growth rate.

5.3 Numerical Methods

5.3.1 Reacting Flow Simulation

The governing equations and numerical methods summarized in this section have been implemented in the MPSalsa computer code, developed at Sandia National Laboratories. More complete descriptions of the code and capabilities can be found in the following references [108], [100], [109], [101], [88], [40]. The fundamental conservation equations for momentum, heat, and mass transfer are presented for a reacting flow application. The equations for fluid flow consist of the incompressible Navier-Stokes equations for a variable-density fluid and the continuity equation, which express conservation of momentum and total mass. The steady-state momentum equation takes the form:

$$\rho(\mathbf{u} \bullet \nabla)\mathbf{u} - \nabla \bullet \mathbf{T} - \rho\mathbf{g} = 0, \tag{5.3.3}$$

where **u** is the velocity vector, ρ is the mixture density, and **g** is gravity vector. **T** is the stress tensor for a Newtonian fluid:

$$\mathbf{T} = -P\mathbf{I} - \frac{2}{3}\mu(\nabla \bullet \mathbf{u})\mathbf{I} + \mu[\nabla\mathbf{u} + \nabla\mathbf{u}^T] \tag{5.3.4}$$

Here P is the isotropic hydrodynamic pressure, μ is the mixture viscosity, and **I** is the unity tensor. The total mass balance is given by:

$$\nabla \bullet (\rho\mathbf{u}) = 0 \tag{5.3.5}$$

The density depends on the local temperature and composition via the ideal gas law. For non-dilute systems, the multicomponent formulation is used:

$$\rho = \frac{P_o \sum_{j=1}^{N_g} W_j X_j}{RT}, \tag{5.3.6}$$

where P_o is the thermodynamic pressure, R is the gas constant, T is the temperature, X_j is the mole fraction of the j^{th} species, W_j is the molecular weight of the j^{th} species, and N_g is the number of gas-phase species (which is 4 for the model in this paper.

The steady-state energy conservation equation is given as:

$$\rho \hat{C}_p(\mathbf{u} \bullet \nabla)T = \nabla \bullet (\lambda \nabla T) - S, \tag{5.3.7}$$

where $\hat{C}_p$ is the mixture heat capacity and λ is the mixture thermal conductivity. The last term on the right hand side S is the source term due to the heat of reaction, which is negligible under the process conditions in this example problem.

The species mass balance equation is solved for N_g-1 species:

$$\rho(\mathbf{u} \bullet \nabla)Y_k) = \nabla \bullet \mathbf{j}_k + W_k \dot{\omega}_k \qquad \text{for } k = 1, \ldots, N_g\text{-}1, \tag{5.3.8}$$

where Y_j is the mass fraction of the j^{th} species, $\mathbf{j}_k$ is the flux of species k relative to the mass averaged velocity **u** and $\dot{\omega}_k$ is the molar rate of production of species k from gas-phase reactions. A special species equation, which enforces the sum of the mass fractions to equal one, replaces one of the species balances (usually the species with the largest mass fraction):

$$\sum_{k=1}^{N_g} Y_k = 1 \qquad \text{for } k = N_g \tag{5.3.9}$$

The diffusive flux term (Multicomponent Dixon-Lewis Formulation) includes transport due to both concentration gradients and thermal diffusion (Soret effect):

$$\mathbf{j}_k = \rho Y_k \left(\frac{1}{X_k \overline{W}} \sum_{j \neq k}^{N_g} W_j D_{kj} \nabla X_j - \frac{D_k^T}{\rho Y_k} \frac{\nabla T}{T} \right) \tag{5.3.10}$$

Where X_j is the mole fraction of species j, D_{kj} is the ordinary multicomponent diffusion coefficient, and D_k^T is the thermal diffusion coefficient. $\overline{W}$ is the mean molecular weight of the mixture given by:

$$\overline{W} = \sum_{k=1}^{N_g} X_k W_k = \frac{1}{\displaystyle\sum_{k=1}^{N_g} \frac{Y_k}{W_k}} \tag{5.3.11}$$

The conversion between mass (Y_k)and mole (X_k) fractions is:

$$Y_k = \frac{W_k}{\overline{W}} X_k \tag{5.3.12}$$

At the disk surface, surface chemical reactions take place. In general these can be very complicated, but for this model problem the reaction has been shown to be approximated very well by a transport limited model. In this case, the growth rate of GaN on the surface (as well as the consumption of $Ga(CH_3)_3$ and NH_3, and the production of CH_4) is proportional to the concentration of trimethylgallium ($Ga(CH_3)_3$) at the surface.

In general, the numerous physical properties in the above equations are dependent on the local temperature and composition. In the MPSalsa code, we use the Chemkin library and database format to obtain these physical properties. These terms add considerable nonlinearity to the problem.

The above system of 9 coupled PDEs (for unknowns $\mathbf{u}_r$, $\mathbf{u}_z$, $\mathbf{u}_\theta$, P, T, $Y_{Ga(CH_3)_3}$, Y_{CH_4}, Y_{NH_3} and Y_{H_2}) are solved with the MPSalsa code. MPSalsa uses a Galerkin/least-squares finite element method [109] to discretize these equations over the spatial domain. While this code is designed for general unstructured meshes in 2D and 3D, and runs on massively parallel computers, this application is 2D, uses the mesh shown in Figure 5.1(a), and was run on a single processor workstation. The discretized system contains 22000 unknowns.

A fully coupled Newton's method is used to robustly calculate steady-state solutions. While analytic Jacobian entries are supplied for derivatives with respect to the solution variables and the density, derivatives of the other physical properties are only calculated with the numerical Jacobian option. This option uses first order finite differencing on the element level. The resulting linear system at each iteration is solved using the Aztec package of parallel, preconditioned iterative solvers. In this paper, we exclusively used an ILU preconditioner and the GMRES solver with no restarts. On a single processor SGI workstation, a typical matrix formulation required 9 seconds for the inexact analytic Jacobian and 96 seconds to calculate the (nearly) exact finite difference numerical Jacobian. A typical linear solve required 40 seconds.

Parameter continuation methods have been implemented in MPSalsa via the LOCA library [99], [102]. LOCA includes an arclength continuation algorithm for tracking solution branches even when they go around turning points (folds). As will be seen in Section 5.4, this is a powerful tool for uncovering solution multiplicity. In addition, a turning point tracking algorithm has been implemented to directly delineate the region of multiplicity as a function of a second parameter. A complementary tool for performing linearized stability analysis by approximating the few rightmost eigenvalues of the linearized time dependent problem has also been successfully implemented [69], [102], [30].

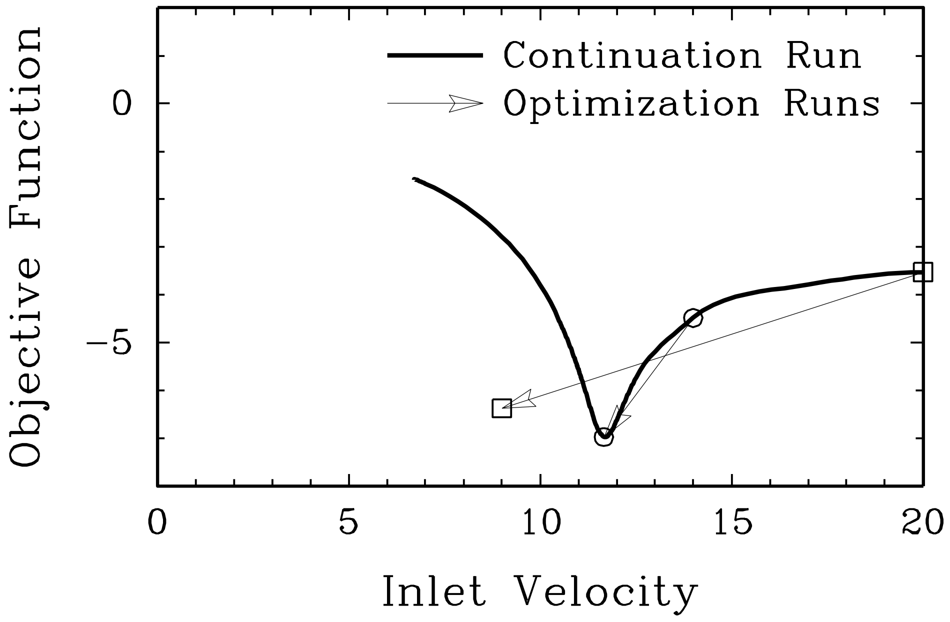


Figure 5.2. Results for a 1 parameter continuation run (bold line), showing the Objective Function as a function of the inlet velocity of the reactant gases. Two results for the rSQP optimizer are shown, where the run starting at $V = 14$ (circle symbols with connecting arrow) converged to the expected local minimum while the run starting at $V = 20$ (square symbols with connecting arrow) converged to a point not seen on the continuation run.

5.4 Results

5.4.1 One Parameter Model

The first results are shown in Figure 5.2 for the one parameter system. Here the inlet velocity V is the design parameter while the Shoulder Radius and Shoulder Height parameters are held fixed at 6.35 and 5.08 as in Figure 5.1(a). Starting at a velocity of $V = 20$ (cm/sec), a simple continuation run down to a velocity of $V = 7$ showed a clear minimum near $V = 11.7$ and Objective Function $F = -6.9$.

Two runs of this problem using the rSQP optimizer were performed. For this run, the exact numerical Jacobian was used, and up to 5 second order correction steps per iteration were allowed. The linear solver tolerance was set at a relative residual reduction of 10^{-8}. When starting at $V = 20$

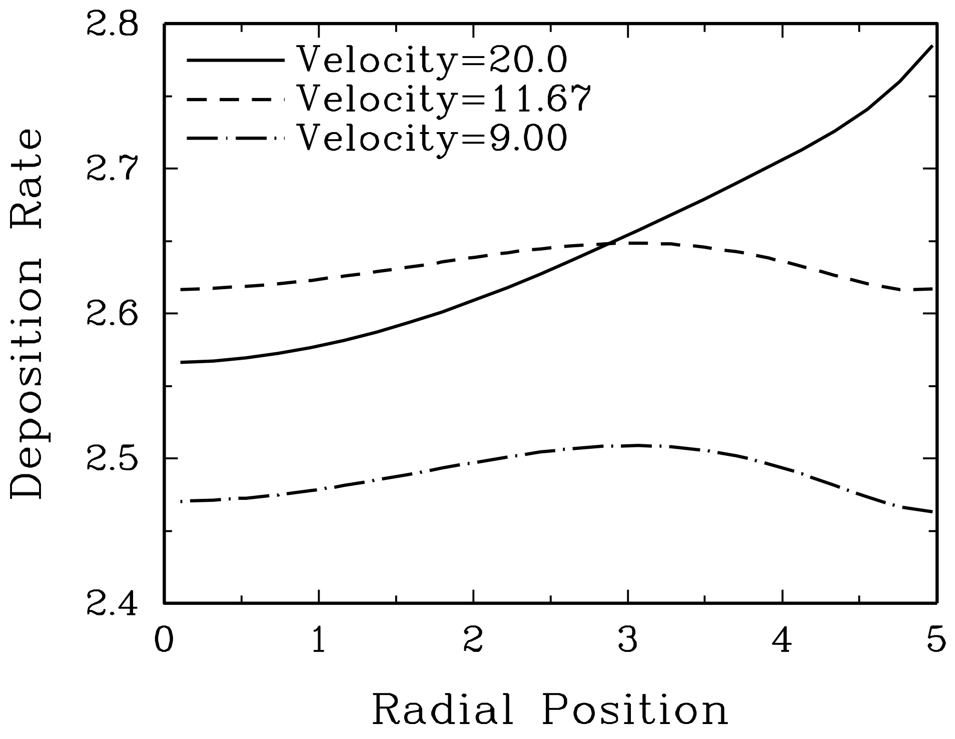


Figure 5.3. Radial profiles of the surface deposition rate at three different solutions: the initial guess at $V = 20$, and the final solutions from the two optimization runs at $V = 11.67$ and $V = 9.00$.

and converged PDE constraints, the optimizer converged in 15 iterations to a point at $V = 9.00$ and $F = -6.36$ (in about 3 hours compute time). However, when starting at $V = 14$ and with a converged steady-state solution, the optimizer reached the minimum at $V = 11.67$ and $F = -6.967$ in 14 iterations. As can be seen in Figure 5.2, the first run does not appear to even be on the solution branch of converged PDE constraints.

Three deposition profiles as a function of radial position are shown in Figure 5.3. The profile at the initial conditions of $V = 20$ has a minimum growth rate at the center and has a 8.5% nonuniformity. The solution found by the optimizer at $V = 11.67$, that also appears to be the minimum from the continuation run, shows a much flatter profile with an internal maximum, and an overall non uniformity of 1.2%. The other solution found by the optimizer at $V = 9.00$ has a very similar shape, a smaller overall growth rate, and a 1.8% nonuniformity. Growth rate nonuniformities in the neighborhood of 1.0% are desirable.

Subsequent parameter continuation and linearized stability analysis calculations revealed that

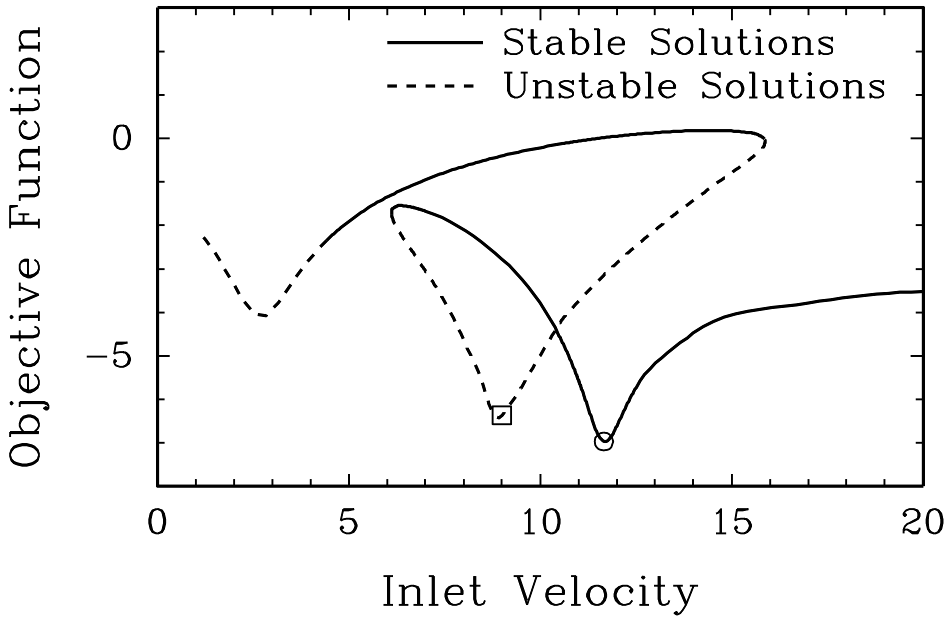


Figure 5.4. Results for a 1 parameter continuation run with arc length continuation and linearized stability analysis are shown. The dashed lines represent unstable solution branches. The symbols show the results of the two optimization runs from Figure 5.2.

this solution is indeed a solution to the PDE constraints, yet a solution that is linearly unstable. The results of an arc length parameter continuation run with linear stability determinations are shown in Figure 5.4. The dashed line indicated physically unstable solutions while the solid lines are locally stable. One can see that there are three local minima in the objective function, only one of which is linearly stable. Over a large range of inlet velocities, $6.11 < V < 15.86$, there are three solutions that exist at the same parameter values. The rSQP optimizer, when started at $V = 20$, jumped into the basin of attraction for a local minimum at $V = 9.00$. The physical basis for the multiplicity is well understood. Recirculation flow cells can develop as a result of the buoyancy force of the heated reactor surface.

5.4.2 Three Parameter Model

The one parameter model showed that it is imperative to be aware of solution multiplicity and unstable solution branches. Continuation runs on the turning points defining the boundaries of multiplicity were performed to see how the region of multiplicity changes as a function of the

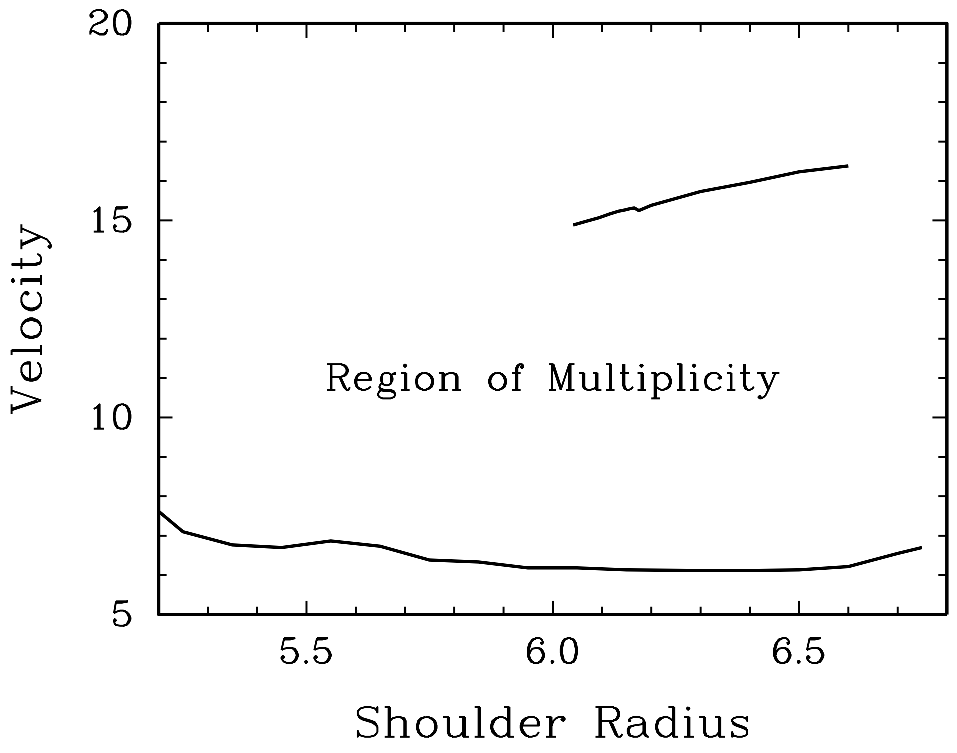


Figure 5.5. Results of turning point continuation runs showing how the region of multiplicity identified in Figure 5.4 changes as a function the geometric Shoulder Radius parameter.

additional geometric parameters. The effect of Shoulder Radius on the multiplicity region is shown in Figure 5.5, and the effect of Shoulder Height on the region of multiplicity is shown in Figure 5.6. The results show that the maximum velocity where multiplicity occurs has a direct dependence on the Shoulder Radius and is relatively insensitive to the Shoulder Height. The minimum velocity where multiplicity occurs is insensitive to the Shoulder Radius but has an inverse dependence on the Shoulder Height.

A single three-parameter optimization run was performed, starting at the same conditions where the one-parameter run that converged to the stable minimum was started: Velocity $= 14.0$, Shoulder Radius $= 6.35$, and the Shoulder Height $= 5.08$. The run was performed with up to 5 second order correction steps per optimization iteration. After 60 iterations, the objective function had been driven down to $F = -6.32$, which is not as low as the $F = -6.967$ achieved in the 1 parameter optimization. Possible reasons for this are that the three-parameter model is converging to a local minimum or that the singularities in the region are causing convergence problems.

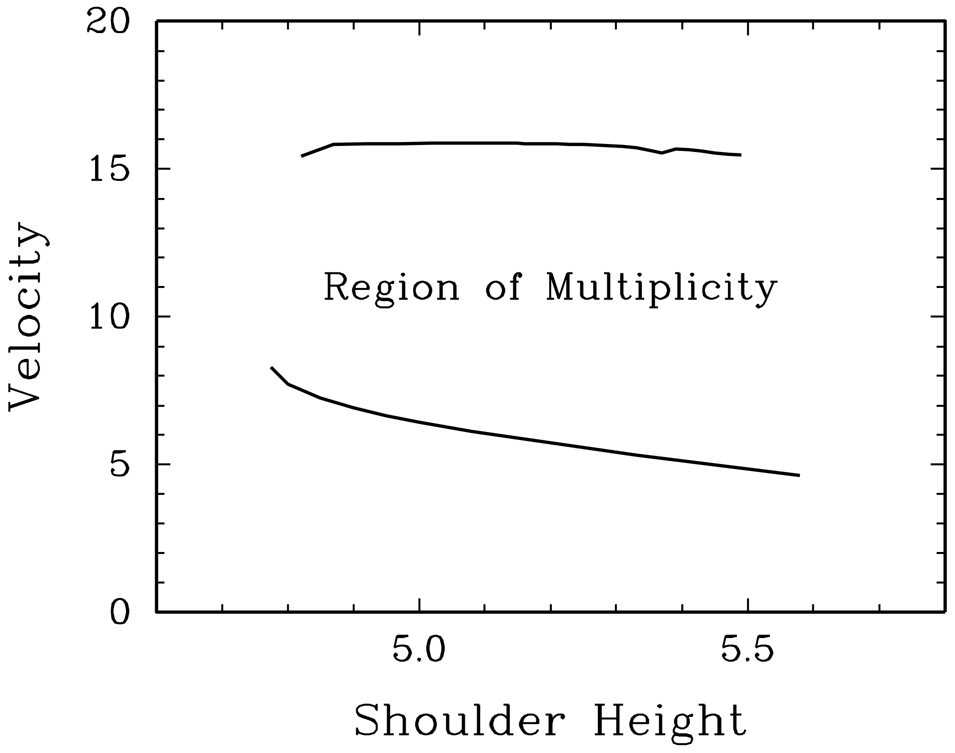


Figure 5.6. Results of turning point continuation runs showing how the region of multiplicity identified in Figure 5.4 changes as a function the geometric Shoulder Height parameter.

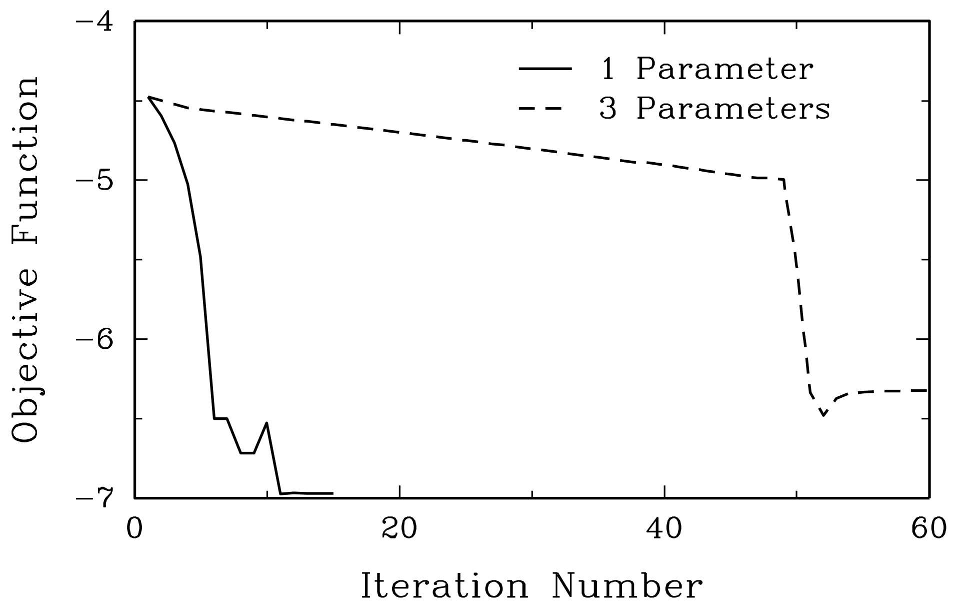


Figure 5.7. A comparison of the 3-parameter optimization run after 60 iterations and the 1-parameter run, started at the same conditions, which converged after 14 iterations.

Future runs will need to be made to fully understand this preliminary result. The result of the three-parameter run is compared to the one-parameter run in Figure 5.7.

5.4.3 Effects of Jacobian Inexactness and Second Order Corrections

To test the effects of inexactness in the Jacobian and Second Order Correction Steps on the convergence of the optimization algorithm, three more runs of the 1-parameter model were performed. These all started at $V = 14$ for comparison with the successful optimization run, which was computed with a full numerical Jacobian and up to 5 second order correction steps per iteration. The results are shown in Figure 5.8.

In the first additional run, the analytic (inexact) Jacobian was used, and the second order corrections were retained. This Jacobian leaves out the derivatives of all the physical properties with respect to the local state (temperature and composition), only including the correct density dependence. The Figure shows that this run converges visibly to the same optimum as the original case, both in iteration 11, though the original case reached the optimum in 14 iterations and the inexact case failed to meet the convergence criterion after 40 iterations. Two more runs were performed

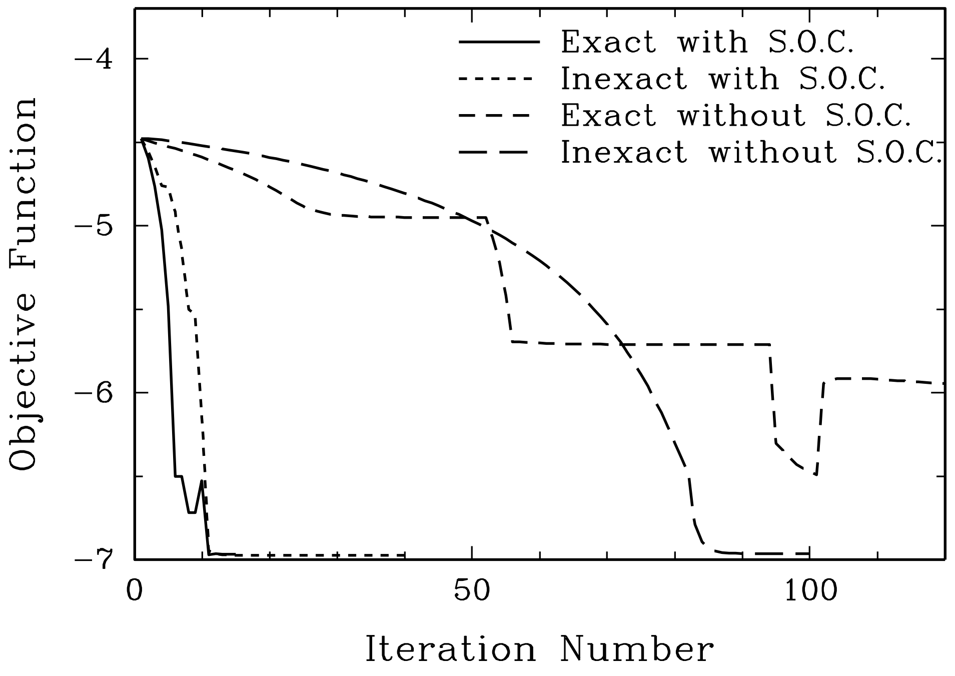


Figure 5.8. A comparison of 4 runs for the 1-parameter model, comparing exact and inexact Jacobians, and with and without second order correction steps (S.O.C.).

where no second order correction steps were allowed. The run with the inexact Jacobian converged visibly to the optimum after 86 iterations though had not converged within the tolerance after 100 iterations. The run with the exact numerical Jacobian without second order corrections had not yet converged to the optimum and was prematurely stopped after 120 iterations, surprisingly performing worse than the run with the inexact Jacobian.

For this problem, MPSalsa required 96 seconds to fill the full numerical Jacobian as compared to only 9 seconds for the analytic Jacobian, while an iterative linear solve required approximately 40 seconds. The runs with second order corrections required, on average, 5 linear solves per iteration, while the runs without second order corrections required exactly 2 linear solves per iteration. Therefore for this problem, the quickest numerical approach for visibly reaching the optimum was using the inexact analytic Jacobian and with the second order correction steps. The runs with the inexact Jacobian did not trigger the convergence tolerance set in the algorithm, and therefore performed many wasted iterations after visibly reaching the optimum. Since there are numerous approximations in the model, particularly with the chemistry mechanisms, the optimum needs only be converged to two digits of accuracy.

5.5 Optimization problem - Source Inversion

The rSQP/MPSalsa code was also used to investigate source inversion problems. Potential application of this problem is chemical/biological/radiological attacks on our nation's infrastructure, such as water distribution systems, large facilities, and urban areas. Given concentration data at several sensor locations within a facility, the goal is to determine the original location and magnitude of the attack subject to Navier Stokes fluid flow. We assume that chemical transport follows diffusive behavior and therefore we use heat as a chemical source, and temperature as chemical concentrations. Even though this application is a real time optimization problem, our initial development efforts were confined to the steady state problem. Two models were investigated, the first was a simple box geometry and the second was a two dimensional model emulating actual airport terminal dimensions and operating conditions.

Figure 5.9 shows the box geometry that was initially used to test our inversion algorithms. The left figure shows the convective steam lines, entering at the top left (Dirichlet condition) and leaving at the bottom right (appropriate outflow conditions). The right figure shows the diffusion behavior as a result of introducing three sources marked on the side of the box with their relative magnitudes.

Prior to conducting the inverse problem, the forward problem was executed to calculate the concentration values at various points in the box geometry, marked with a red “x”. The concentrations at these 25 sensor locations were then used to solve the following optimization problem:

A forward problem was solved using MPSalsa with a 1600 element finite element discretization. This led to 1681 constraints for the discretized Navier Stokes PDE. Three out of 16 fluxes were set nonzero (of magnitudes 1,2, and 5 as seen in the figure) and sensor data was recorded. Then the inverse problem was solved from a trivial initial guess using rSQP/MPSalsa as follows:

minimize:

$$\frac{1}{2}\sum_{i}^{s}\int_{d\Omega}\delta(\mathbf{x}-\mathbf{x}_i)(c-c^*)^2 d\Omega \tag{5.5.13}$$

subject to $c(x,f)=0$ where c represents the Navier Stokes equations (section 5.3.1).

The 16 fluxes converged to the values set in the forward problem in 88 rSQP iterations.

Because of our investment and experience in PDE constrained optimization applied to CVD reactors, this prototype problem was solved within 2 days of first discussing the potential of rSQP/MPSalsa as a counter-terrorism capability.

A more complex geometry and parameter values was tested to emulate the conditions of an airport facility. Figure 5.10 shows a 2D representation of an actual two-story airport terminal. This model represented one sixth of the terminal, which was controlled by a single HVAC system. The model problem used realistic dimensions of a terminal, properties of air, diffusion coefficient for SF_6 (a common tracer for experiments). Flow rates were varied but did approach reasonable conditions.

The problem was formulated the same as the box problem above, except that two of the air flow velocities entering this section of the terminal (from down the hall) were left as unknowns. This meant that the nonlinear Navier-Stokes PDE’s, in addition to the convection-diffusion equation, were part of the constraints. In later runs, a one-equation (Spalart-Almaras) turbulence model was solved in conjunction with these equations. In our first prototype, only three locations along the bottom floor were selected as candidate source locations, leading to a total of 5 design variables. Ten sensor locations were picked (see red X’s in the bottom figure).

A finite element discretization of the PDE’s led to over 200000 algebraic constraints for the 5-parameter optimization problem. One run ran for 2 hours on 64 processors of the Ross CPlant machine and successfully reduced the objective function 3 orders of magnitude from a simple initial guess.

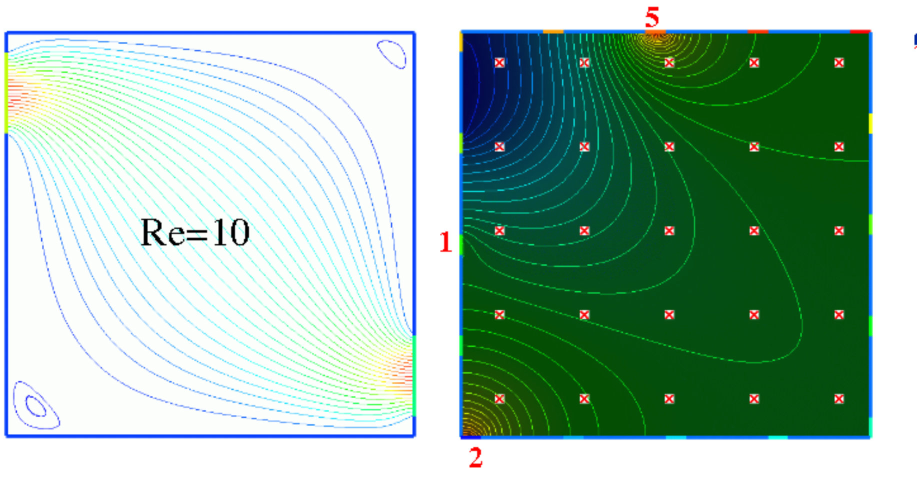


Figure 5.9. Source inversion of convection-diffusion in a box geometry. This was out initial prototype problem for source inversion of chem/bio/rad attack scenarios. The left box shows convective streamlines and the right box shows the diffusive behavior with the red "x" markers denoting sensor locations

Much was learned from this prototype problem. For the optimization problem, this direct sensitivity approach used here could work well up to 20 design variables, but an adjoint sensitivity approach would be preferred to allow for numerous candidate sensor locations. Allowing flow rates as design variables was a big step, since it invoked several coupled nonlinear PDEs as constraints instead of one linear convection-diffusion PDE. Issues that were not faced in this prototype problem are (1) solving the transient problem and (2) dealing with noisy sensor data.

From a modeling standpoint, several areas have been identified where future work would be needed to continue this effort. One is dealing with high Reynolds numbers (turbulence) for air in the large domains. A second is a new interface for choosing potential source locations, since our method of meshing them individually and assigning a side set ID is not adequately flexible or scalable. Another is dealing with agents (such as anthrax particles) that require extensions to the Navier Stokes equations.

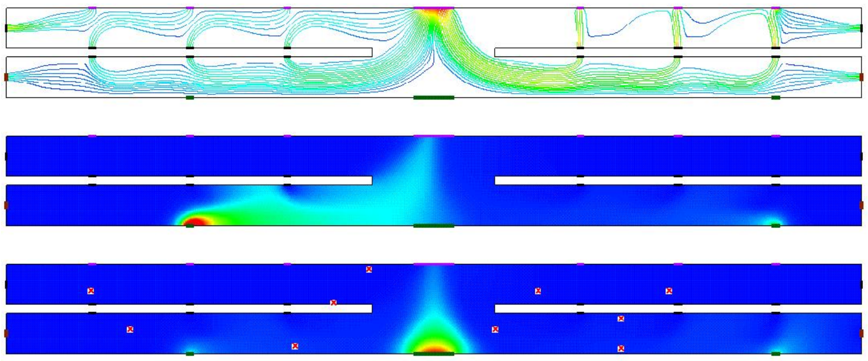


Figure 5.10. Source inversion 2D cross-sectional model of a two-story airport facility. The top figure shows flow streamlines, the middle figure shows concentrations of an agent being released from two locations along the bottom floor, and the third shows the ten sensor locations and concentration profiles from a different source values.

5.6 Conclusions, Stability, Interface & Validation

Solution multiplicity of nonlinear steady-state problems must be recognized and can be diagnosed using stability analysis tools. The technique in this paper of tracking the region of multiplicity is not scalable to larger numbers of design parameters, and is more expensive than the optimization calculations. At a minimum, the stability of the candidate optimum must be checked with a linear stability analysis tool. Concerning inexactness in the Jacobian matrix, and the effect of second order correction steps, we have gathered some evidence. For this run, it appears that inexactness in the Jacobian does not seriously hinder convergence, particularly if second order correction steps are used.

Several conclusions can be drawn from interfacing a rSQP algorithm to a complete fluid flow simulator. Calculating sensitivities is perhaps the single most important modification to a simulation code for PDECO. Once a sensitivity capability exists, the interface to a rSQP algorithm is trivial. As a result of the MPSalsa project, several sensitivity projects have been initiated with new simulation developments. In addition, a research project has been started to investigate methods to handle transient optimization problems efficiently. Another very important conclusion is that conducting algorithmic research with large-scale simulation codes is very difficult. The rSQP algorithms can be tested on small systems, but to validate our algorithms across many PDE-based problems is not practical, especially if that means interfacing with production and cumbersome simulation codes. To address these problems we have developed a symbolic simulation capability and interfaced it with our rSQP algorithms. The next two chapters provide a description of Sundance and Sundance coupled to rSQP++.

Chapter 6

Sundance

Traditional PDE codes solve one of a specific class of PDEs with little hope of obtaining the gradients, adjoints, or Hessians needed for PDECO. Even with modern PDE frameworks such as SIERRA and Nevada, it will require considerable development effort to obtain these quantities. Thus, for optimization with existing PDE codes, one must use the PDE solver as a “black box,” and we are restricted to relatively inefficient **Level-0** or **Level-1** methods.

Since PDE-constrained optimization requires capabilities beyond those available in traditional PDE codes, we have developed a PDE solver system that has been designed from the ground up with large-scale PDE-constrained optimization in mind. This system, called Sundance, accepts a system of coupled PDEs and boundary conditions written in symbolic form that is close to the notation in which a scientist or engineer would normally write them with pencil and paper. Each function or variation appearing in this symbolic description is annotated with a specification of the finite-element basis with which that object will be discretized. This information, along with a mesh, is then used by Sundance to assemble the implied discretized operators. At this point, the user could simply ask Sundance to solve the system, or it could request certain evaluations to be made. These symbolic capabilities make Sundance a powerful rapid prototyping and algorithmic research tool, however, for present purposes the real power of Sundance’s symbolic interface is that the symbolic expressions comprising the PDE and boundary conditions can be differentiated allowing automated derivation of gradients and Hessians as needed in PDECO. We must emphasize that for performance reasons, the high-level objects used for problem specification are not used for numerical calculations. Rather, they are used to marshal a set of internal objects that can be used for efficient calculations.

Sundance has been developed using a component-oriented design. Abstract concepts such as linear solvers, basis functions, quadrature rules, or reordering schemes (to name just a few) are represented in terms of abstract interfaces. A particular realization of such a concept, for instance an Aztec solver, is then implemented as a concrete type and can be plugged into the Sundance system via the interface. This design has two key advantages. First, it makes Sundance highly extensible, since developers can add new components without modifying the core of Sundance. Second, it allows the use of the highest-performance third-party components with Sundance. Sundance does not have built-in meshers, solvers, or visualization capabilities; rather, it uses third-party components for all of those tasks.

In this chapter we will start with an introductory example illustrating basic Sundance syntax. We will then give an overview of the core components of Sundance, with code examples as new capabilities are introduced. Simple examples of the use of Sundance for a linear PDECO problem, a nonlinear PDE, and a transient PDE are given here. Further examples of the use of Sundance in nontrivial, nonlinear PDECO problems are given in Chapter 7. For a comprehensive presentation of Sundance's capabilities and further examples of forward problems, see the Sundance User's Guide [72].

6.1 An introductory example

We begin with a simple example of a forward problem that will show basic Sundance components. Consider the Poisson equation with a unit source

$$\nabla^2 u = 1 \tag{6.1.1}$$

on the rectangle $[0, 0]$ - $[1, 2]$. The sides of the rectangle will be labeled left, right, bottom, top. For boundary conditions, we will choose

- **left** Homogeneous Neumann, $\nabla u \cdot \hat{n} = 0$
- **bottom** Dirichlet, $u = \frac{1}{2}x^2$
- **right** Robin, $u + \nabla \mathbf{u} \cdot \hat{\mathbf{n}} = \frac{3}{2} + \frac{y}{3}$
- **top** Neumann, $\nabla u \cdot \hat{\mathbf{n}} = 1/3$

It is easy to check that the solution is

$$u = \frac{1}{2}x^2 + \frac{1}{3}y. \tag{6.1.2}$$

The solution is in the subspace spanned by second-order Lagrange polynomials, so if we choose that as our basis family we can expect to obtain the exact solution. We can compute the error norm at the end of the calculation as a check that the code is working properly.

This is a simple problem, but it in fact requries most of the components used by Sundance to do more complex problems.

6.1.1 Step-by-step explanation

We start with a step-by-step walkthrough of the code for solving the Poisson problem. When finished, there will be a summary and then the complete Poisson solver code will be listed for reference.

6.1.1.1 Boilerplate

A dull but essential first step is to show the boilerplate C++ common to nearly every Sundance code:

```
#include "Sundance.h"

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(argc, argv);

      /*
       * code body goes here
       */
    }
  catch(exception& e)
    {
       Sundance::handleException(__FILE__, e);
    }
  Sundance::finalize();
}
```

The body of the code – everything else we discuss here – goes in place of the comment `code body goes here`.

6.1.1.2 Getting the mesh

Sundance uses a `Mesh` object to represent a discretization of the problem domain. There are two ways to get a `Mesh` object:

- Create it using Sundance's built-in mesh generation capability. This is limited to meshing very simple domains such as rectangles.
- Read a mesh that has been produced using a third-party mesh generator. The `MeshReader` class provides an interface for reading arbitrary file formats.

For this simple problem, we can use Sundance to generate the mesh.

```
MeshGenerator mesher = new RectangleMesher(0.0, 1.0, nx, 0.0, 2.0, ny);
Mesh mesh = mesher.buildMesh();
```

If you know a little C++ – just enough to be dangerous – you might think it odd that the result of the `new` operator, which returns a pointer, is being assigned to a `MeshGenerator` object which is – apparently – not a pointer. That's not a typo: the `MeshGenerator` object is a **handle** class that stores and manages the pointer to the `RectangleMesher` object. Handle classes are used throughout user-level Sundance code, and among other things relieve you of the need to worry about memory management.

6.1.1.3 Defining coordinate functions

In the Poisson example, the boundary conditions involve functions of the coordinates x and y. We will create objects to represent the coordinate functions x and y.

```
Expr x = new CoordExpr(0);
Expr y = new CoordExpr(1);
```

You have probably guessed that the integer argument to the `CoordExpr` constructor gives the coordinate direction: 0 for x, 1 for y, 2 for z.

The coordinate functions are wrapped in `Expr` handle objects. Class `Expr` is used for all symbolic objects in Sundance. `Expr`s can be operated on with the usual mathematical operators. With our coordinate functions represented as `Expr` objects, we can build complicated functions of position.

6.1.1.4 Defining the cell sets

We've already read a mesh. We need a way to specify *where* on the mesh equations or boundary conditions are to be applied. Sundance uses a `CellSet` object to represent subregions of a geometric domain. A `CellSet` can be any collection of mesh cells, for example a block of maximal cells, a set of boundary edges, or a set of points.

The `CellSet` class has a `subset()` method that can be used as a "filter" that identifies cells that are in a subset defined by the arguments to the `subset` method.

We will apply different boundary conditions on the four sides of the rectangle, so we will want four `CellSet`s, one for each side. We first create a cell set object for the entire boundary,

```
CellSet boundary = new BoundaryCellSet();
```

and then we find the four sides as subsets of the boundary cell set. The four sides of the rectangle can be specified with logical operations on coordinate expressions, as shown in the following code:

```
CellSet left = boundary.subset( x == 0.0 );
CellSet right = boundary.subset( x == 1.0 );
CellSet bottom = boundary.subset( y == 0.0 );
CellSet top = boundary.subset( y == 2.0 );
```

6.1.1.5 Creating a discrete function

We will use discrete functions several places in this problem. A discrete function takes as a constructor argument a vector space object that specifies the mesh, basis, and vector representation to be used in discretizing the function.

The first step is to create a vector space factory object that tells us what kind of vector representation will be used. We'll use Petra vectors, so we create a `PetraVectorType`.

```
TSFVectorType petra = new PetraVectorType();
```

We can now create a `SundanceVectorSpace` containing the mesh, a basis (2nd order Lagrange in this case) and the vector space factory.

```
TSFVectorSpace discreteSpace = new SundanceVectorSpace(mesh, new Lagrange(2), petra);
```

Finally, we can create discrete functions to represent the source term $f = 1.0$ and the expression $\frac{3}{2} + \frac{y}{3}$ that appears in the right BC. Note that there's no particular need to use discrete functions for those terms; we do so here simply to provide an example of constructing a discrete function.

```
Expr f = new DiscreteFunction(discreteSpace, 1.0);
Expr rightBCExpr = new DiscreteFunction(discreteSpace, 1.5 + y/3.0);
```

6.1.1.6 Defining unknown and test functions

We'll use 2nd order piecewise Lagrange interpolation to represent our unknown solution u. With a Galerkin method we define a test function v using the same basis as the unknown. Expressions representing the test and unknown functions are defined easily:

```
Expr v = new TestFunction(new Lagrange(2));
Expr u = new UnknownFunction(new Lagrange(2));
```

6.1.1.7 Creating the gradient operator

The gradient operator is formed by making a `List` containing the partial differentiation operators in the x and y directions.

```
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
```

The gradient thus defined is treated as a vector with respect to the overloaded multiplication operator used to apply the gradient, so that an operation such as `grad*u` expands correctly to {`dx*u, dy*u`}.

6.1.1.8 Writing the weak form

We will use the Galerkin method to construct a weak form. Begin by multiplying Poisson's equation Equation 6.1.1 by a test function v and integrating

$$-\int_\Omega v\nabla^2 u - \int_\Omega vf = 0. \tag{6.1.3}$$

The next step is to integrate by parts, which has the effects of lowering the order of differentiation (and thus relaxing the differentiability requirements on the unknown and test functions) and also making the boundary flux. The resulting weak form is

$$-\int_{\Omega} \nabla v \cdot \nabla u - \int_{\Omega} v f + \int_{\partial\Omega} v \nabla \mathbf{u} \cdot \hat{\mathbf{n}} = 0 \tag{6.1.4}$$

and we will require that this equation hold for any test function v in the space of 2nd order Lagrange interpolants on our mesh. The boundary term gives us a way to apply certain boundary conditions: we can apply the Neumann and Robin BCs by substituting an appropriate value for $\nabla \mathbf{u} \cdot \hat{\mathbf{n}}$ in the boundary term. Referring to the boundary conditions above and our definition of the discrete function `rightBCExpr`, the weak form is written in Sundance as

```
Expr poisson = Integral(-(grad*v)*(grad*u)  - f*v)
                   + Integral(top, v/3.0)
                   + Integral(right, v*(rightBCExpr - u));
```

Notice that the homogeneous BC on the left side does not need to be written explicitly because that boundary term is zero.

6.1.1.9 Writing the essential BCs

The weak form contains the physics in the body of the domain plus the Neumann and Robin boundary conditions. We still need to apply the Dirichlet boundary condition on the bottom edge, which we do with an `EssentialBC` object

```
EssentialBC bc = EssentialBC(bottom, v*(u - 0.5*x*x));
```

The first argument gives the region on which the boundary condition holds, and the second gives an expression that is to be set to zero. Notice that there is a test function in the BC; this identifies the row space on which the BC is to be applied.

6.1.1.10 Creating the linear problem object

A `StaticLinearProblem` object contains everything that is needed to assemble a discrete approximation to our PDE: a mesh, a weak form, boundary conditions, specification of test and unknown functions, and a specification of the low-level matrix and vector representation to be used. All of this information is given to the constructor to create a problem object

```
StaticLinearProblem prob(mesh, poisson, bc, v, u, petra);
```

It may seem unnecessary to provide `v` and `u` as constructor arguments here; after all, the test and unknown functions could be deduced from the weak form. In more complex problems with vector-valued unknowns, however, we will want to specify the order in which the different unknowns and test functions appear, and we may want to group unknowns and test functions into blocks to create a block linear system. Such considerations can make a great difference in the performance of linear solvers for some problems. The test and unknown slots in the linear problem constructor are used to pass information about the function ordering and blocking to the linear problem; these features will be used in subsequent examples.

6.1.1.11 Specifying the solver

A good choice of solver for this problem is BICGSTAB with ILU preconditioning. We'll use level 2 preconditioning, and ask for a convergence tolerance of 10^-14 within 500 iterations.

```
TSFPreconditionerFactory precond = new ILUKPreconditionerFactory(2);
TSFLinearSolver solver = new BICGSTABSolver(precond, 1.e-14, 500);
```

6.1.1.12 Solving the problem

The syntax of Sundance makes the next step look simpler than it really is:

```
Expr soln = prob.solve(solver);
```

What is happening under the hood is that the problem object `prob` builds a stiffness matrix and load vector, feeds that matrix and vector into the linear solver `solver`. If all goes well, a solution vector is returned from the solver, and that solution vector is captured into a discrete function wrapped in the expression object `soln`.

6.1.1.13 Viewing the solution

We next write the solution in a form suitable for viewing by Matlab.

```
FieldWriter writer = new MatlabWriter("heat2D.dat");
writer.writeScalar(mesh, "temperature", soln);
```

6.1.1.14 Checking the error norm

Finally, we compare to the exact solution by computing the error norm. The solution has been returned as a Sundance expression, so we can form an expression for the error

```
Expr exactSoln = 0.5*x*x + y/3.0;
Expr error = exactSoln - soln;
```

and then take the L^2 norm

```
double errorNorm = error.norm();
```

6.1.2 Complete code for the poisson problem

```
#include "Sundance.h"

/** \example heat2D.cpp
 * Solve Poisson’s equation with a unit source term on the
 * rectangle [0,1] x [0, 2] with the following boundary conditions:
 *
 * Left:   Natural, du/dx = 0
 * Bottom: Dirichlet, u= 0.5 xˆ2
 * Right:  Robin, u + du/dx = 3/2 + y/3
 * Top:    Neumann, du/dy = 1/3
 *
 * The solution is u(x,y) = 0.5*xˆ2 + y/3.
 *
 * This problem can be solved exactly in the space of second-order polynomials.
 */

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(argc, argv);

      /* create a simple mesh on the rectangle */
      int nx = 20;
      int ny = 20;
      MeshGenerator mesher = new RectangleMesher(0.0, 1.0, nx, 0.0, 2.0, ny);
      Mesh mesh = mesher.getMesh();

      /* define coordinate functions for x and y coordinates */
      Expr x = new CoordExpr(0);
      Expr y = new CoordExpr(1);

      /* define cells sets for each of the four sides of the rectangle */
      CellSet boundary = new BoundaryCellSet();
      CellSet left = boundary.subset( x == 0.0 );
      CellSet right = boundary.subset( x == 1.0 );
      CellSet bottom = boundary.subset( y == 0.0 );
      CellSet top = boundary.subset( y == 2.0 );

      /* Create a vector space factory, used to
       * specify the low-level linear algebra representation */
      TSFVectorType petra = new PetraVectorType();

      /* create a discrete space on the mesh */
      TSFVectorSpace discreteSpace = new SundanceVectorSpace(mesh, new Lagrange(2), petra);


      /* We’ll use a discrete function to represent the
       * source term, providing a test
       * of our ability to evaluate discrete functions on maximal cells */
      Expr f = new DiscreteFunction(discreteSpace, 1.0);

      /* We’ll use a discrete function to represent the imposed
       * boundary value on the right-hand boundary.  This provides a
       * test of our ability to evaluate discrete functions on
       * lower-dimensional cells. */
      Expr rightBCExpr = new DiscreteFunction(discreteSpace, 1.5 + y/3.0);


      /* create symbolic objects for test and unknown functions */
      Expr v = new TestFunction(new Lagrange(2));
      Expr u = new UnknownFunction(new Lagrange(2));

      /* create symbolic differential operators */
      Expr dx = new Derivative(0);
      Expr dy = new Derivative(1);
      Expr grad = List(dx, dy);

      /* Write symbolic weak equation and Neumann and Robin BCs */
      Expr poisson = Integral(-(grad*v)*(grad*u)  - f*v, new GaussianQuadrature(2))
        + Integral(top, v/3.0) + Integral(right, v*(rightBCExpr - u));


      /* Write essential BCs:
       * Bottom: u=xˆ2
       */
      EssentialBC bc = EssentialBC(bottom, v*(u - 0.5*x*x),
                                   new GaussianQuadrature(4));

      /* Assemble everything into a problem object, with a specification that
       * Petra be used as the low-level linear algebra representation */
      StaticLinearProblem prob(mesh, poisson, bc, v, u, petra);

      /* create a preconditioner and solver */
      TSFPreconditionerFactory precond = new ILUKPreconditionerFactory(1);
      TSFLinearSolver solver = new BICGSTABSolver(precond, 1.e-14, 500);

      /* solve the problem and return the solution as a symbolic object */
      Expr soln = prob.solve(solver);
```

```
      /* write to matlab */
      FieldWriter writer = new MatlabWriter("heat2D.dat");
      writer.writeField(soln);

      /* compare to known solution */
      Expr exactSoln = 0.5*x*x + y/3.0;

      // compute the norm of the error
      double errorNorm = (soln-exactSoln).norm(2);
      double tolerance = 1.0e-9;

      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();
}
```

6.2 A PDE-constrained optimization example

We now show a simple example of how to use Sundance to set up an optimization problem with a PDE constraint. Consider the Poisson equation with source terms parameterized with a design variable α,

$$\nabla^2 u = \sum_k \alpha_k \sin k\pi x. \tag{6.2.5}$$

A simple optimization problem is to choose α such that the state function u is a good fit to a target function $\hat{u}$. This target-fitting problem can be posed as a least-squares problem with objective function

$$f(\alpha) = \frac{1}{2}\int_\Omega (u(\alpha) - \hat{u})^2 + \frac{R}{2}\sum_k \alpha_k^2 \tag{6.2.6}$$

where R sets the control cost. As written, we could solve this problem with a pattern search method in which we solve 6.2.5 for u at each function evaluation. Alternatively, we can let the states become independent variables, but impose equation 6.2.5 as a constraint. In that case, we have a Lagrangian

$$L(\alpha, u, \lambda) = \frac{1}{2}\int_\Omega (u - \hat{u})^2 + \frac{R}{2}\sum_k \alpha_k^2 - \int_\Omega \nabla u \cdot \nabla\lambda - \sum_k \alpha_k \int_\Omega \lambda \sin k\pi x \tag{6.2.7}$$

where λ is a Lagrange multiplier. The necessary condition for solving the optimization problem is that the variations of the Lagrangian with respect to α, u, and λ are all zero.

This example is a quadratic program with an equality constraint, and the solution is obtained with a single linear solve of the KKT system. However, the KKT system is indefinite and is most efficiently solved using a block Schur complement method.

6.2.1 Sundance problem specification

With Sundance, all we need do to pose this problem is to write the Lagrangian using Sundance symbolic objects.

Note that in this problem, the state variable u and Lagrange multiplier λ are unknown functions defined with a finite-element basis. However, the design parameters are unknown "global" parameters, defined independently of the mesh. In Sundance, mesh-based unknowns are `UnknownFunction` expression subtypes and global unknowns are `UnknownParameter` expression subtypes. To optimize performance in parallel, Sundance imposes the restriction that all global unknowns must appear in a separate block from any meshed unknowns; that block is then replicated across processors while blocks containing meshed unknowns are distributed. Matrix blocks mapping between the global unknown space and a meshed unknown space are implemented with multivectors, in which each row (or column, depending on the orientation of the block) is a distributed vector. The specification of the unknowns and block structure for this problem is done with the following Sundance code:

```
Expr u = new UnknownFunction(new Lagrange(2));
Expr v = u.variation();

Expr lambda = new UnknownFunction(new Lagrange(2));
Expr mu = lambda.variation();

Expr alpha1 = new UnknownParameter();
Expr alpha2 = new UnknownParameter();
Expr alpha3 = new UnknownParameter();
Expr alpha = List(alpha1, alpha2, alpha3);
Expr beta = alpha.variation();

TSFVectorType petra = new PetraVectorType();
TSFVectorType dense = new DenseSerialVectorType();

TSFArray<Block> unks = tuple(Block(alpha, dense), Block(u, lambda, petra));

TSFArray<Block> vars = tuple(Block(beta, dense), Block(mu, v, petra));
```

Once the unknowns have been specified, we can write out the objective function and Lagrangian in symbolic form:

```
  Expr objectiveFunction = 0.5*Integral(pow(u-target, 2.0))
        + 0.5*alpha*alpha;
  Expr lagrangian = objectiveFunction - Integral((dx*u)*(dx*lambda))
        - Integral(lambda*forcing);
```

The equation set can be obtained by taking symbolic variations of the Lagrangian.

```
  Expr eqn = lagrangian.variation(List(u, lambda, alpha));
```

We will solve the system using a Schur complement solver, using TSF's block manipulation capabilities. The user-level code to specify a Schur complement solver for a 2 by 2 block system is

```
  TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
  TSFLinearSolver innerSolver = new BICGSTABSolver(1.0e-12, 1000);
  TSFLinearSolver outerSolver = new BICGSTABSolver(1.0e-10, 1000);

  TSFLinearSolver solver = new SchurComplementSolver(innerSolver, outerSolver);
```

Finally, we show complete source code for the PDE-constrained optimization example.

```
#include "Sundance.h"


/**
 *
 */


int main(int argc, void** argv)
{
  try
    {
      Sundance::init(&argc, &argv);

      /*
        Create a mesh object. In this example, we will use a built-in method
        to create a uniform mesh on the unit line. In more realistic problems
        we would use a mesher to create a mesh, and then read the mesh using
        a MeshReader object.
      */
      int n = 10;
      const double pi = 4.0*atan(1.0);
      MeshGenerator mesher = new LineMesher(0.0, pi, n);
      Mesh mesh = mesher.getMesh().getSubmesh();
```

```
/* Define a symbolic object to represent the x coordinate function. */
Expr x = new CoordExpr(0);

Expr psi = List(sin(x), sin(2.0*x), sin(3.0*x));

Expr target = sin(x);

/*
 * Define a cell set that contains all boundary cells
 */
CellSet boundary = new BoundaryCellSet();
/*
 *  Define a cell set that includes all cells at position x=0.
 */
CellSet left = boundary.subset( fabs(x - 0.0) < 1.0e-10 );

/*
 *  Define a cell set that includes all cells at position x=1.
 */
CellSet right = boundary.subset( fabs(x - pi) < 1.0e-10 );



/*
  Define an unknown function and its variation. The constructor
  argument is the basis family with which the function will be
  represented, in this case second-order Lagrange (nodal) polynomials.
*/
Expr u = new UnknownFunction(new Lagrange(2));
Expr v = u.variation();

Expr lambda = new UnknownFunction(new Lagrange(2));
Expr mu = lambda.variation();

Expr alpha1 = new UnknownParameter();
Expr alpha2 = new UnknownParameter();
Expr alpha3 = new UnknownParameter();
Expr alpha = List(alpha1, alpha2, alpha3);
Expr beta = alpha.variation();

TSFVectorType petra = new PetraVectorType();
TSFVectorType dense = new DenseSerialVectorType();

TSFArray<Block> unks = tuple(Block(alpha, dense), Block(u, lambda, petra));

TSFArray<Block> vars = tuple(Block(beta, dense), Block(mu, v, petra));

Expr forcing = alpha * psi;



/*
  Define the differentiation operator of order 1 in direction 0.
*/
Expr dx = new Derivative(0);
Expr objectiveFunction = 0.5*Integral(pow(u-target, 2.0))
  + 0.5*alpha*alpha;

Expr lagrangian = objectiveFunction - Integral((dx*u)*(dx*lambda))
  - Integral(lambda*forcing);

Expr eqn = lagrangian.variation(List(u, lambda, alpha));



/*
  Now specify the boundary conditions on the left and right CellSets.
 */

EssentialBC bc =
  EssentialBC(left, u*mu + v*lambda) && EssentialBC(right, u*mu + v*lambda);




/*
  Create a solver object: stablized biconjugate gradient solver
*/
TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
TSFLinearSolver innerSolver = new BICGSTABSolver(1.0e-12, 1000);
TSFLinearSolver outerSolver = new BICGSTABSolver(1.0e-10, 1000);


TSFLinearSolver solver = new SchurComplementSolver(innerSolver, outerSolver);


/*
  Combine the geometry, the variational form, the BCs, and the solver
  to form a complete problem.
*/
StaticLinearProblem prob(mesh, eqn, bc, vars, unks);
prob.printRowMaps();
mesh.printCells();

/*
  solve the problem, obtaining the solution as a (discrete) Expr object
*/
Expr soln = prob.solve(solver);

/*
  write the solution in a form readable by matlab
*/
FieldWriter writer = new MatlabWriter();
cerr << "u" << endl;
writer.writeField(soln[1][0]);
cerr << "lambda" << endl;
writer.writeField(soln[1][1]);

cerr << soln[0] << endl;
```

```
      /*
        compute the error and represent as a discrete function
      */
      Expr exactSoln = sin(x);

      /*
        compute the norm of the error
      */
      double errorNorm = (soln[1][0] - exactSoln).norm(2);
      double tolerance = 1.0e-10;

      /*
        decide if the error is within tolerance
      */
      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
      Testing::timeStamp(__FILE__, __DATE__, __TIME__);

    }
  catch(exception& e)
    {
      TSFOut::println(e.what());
      Testing::crash(__FILE__);
      Testing::timeStamp(__FILE__, __DATE__, __TIME__);
    }
  Sundance::finalize();
}
```

6.3 Symbolic components

6.3.1 Constant expressions

The simplest type of Expr to create is a constant real-valued Expr, for example:

```
Expr solarMass = 2.0e33; // mass of the Sun in grams
```

Any constant that appears in an expression, for example the constant 2.0 in the expression below,

```
Expr f = 2.0*g;
```

will also be turned into a constant-valued expression. It is important to understand that once created and used in an expression, a constant's value is immutable. If you want to change the constant, you should instead use a `Parameter`.

6.3.2 Parameter expressions

Often you will form a PDE with parameters that will change during the course of a calculation. For example, in a time-marching problem both the time and the timestep can change from step to step. Or, you may want to run a fluid flow simulation at several different values of the Reynolds number. To include in your equation a parameter that is constant in space but can change with time or some other way, you should represent that parameter with a Parameter expression.

```
Expr time = new Parameter(0.0);

for (int i=0; i<10; i++)
{
  cerr << time << ‘‘ ‘‘ << sin(pi*time) << endl;
  // update the time
  time.setValue(time.value() + 0.1);
}
```

The above assumes that the parameter is known. However, in some problems a parameter might be an unknown to be determined in the course of solving a problem; for instance, it could be a design parameter to be determined through optimization. In that case, use an `UnknownParameter`, described in section 6.3.6.

6.3.3 Coordinate expressions

`CoordExpr` is an expression subtype that is hardwired to compute the value of a given coordinate. For example, the following constructs an Expr that represents the coordinate on the zeroth (x) axis:

```
Expr x = new CoordExpr(0); // represents x-coordinate value
```

Such a coordinate expression can be used to define simple position-dependent functions, for example

```
Expr f = sin(x) + 1/4.0*sin(2.0*x) + 1/8.0*sin(3.0*x);
```

6.3.4 Differential operators

The key expression subtype for forming differential operators is the `Derivative` object, representing a partial derivative in a given direction. A `Derivative` is constructed with a single integer argument giving the direction of differentiation, for example,

```
Expr dx = new Derivative(0); // differentiate with respect to 0 coordinate
```

Derivatives are applied using the multiplication (`*`) operator.

Sundance expression objects are programmed to obey the rules of differential calculus. For example,

```
Expr dx = new Derivative(0); // differentiate with respect to 0 coordinate
Expr x = new CoordExpr(0); // represents x-coordinate value
Expr y = new CoordExpr(0); // represents y-coordinate value
Expr f = x*sin(x) + y*x;
Expr df = dx*f;
cout << df << endl;  // prints sin(x) + x*cos(x) + y;
```

Differentiation of discrete functions requires special care, and is discussed in 6.3.7.4

6.3.5 Test and unknown functions

Expression subtypes `TestFunction` and `UnknownFunction` are used to represent test and unknown functions in weak PDEs and boundary conditions. They are constructed with a `BasisFamily` object which specifies the subspace to which solutions and test functions are restricted. For example,

```
Expr T = new UnknownFunction(new Lagrange(1));
Expr varT = new TestFunction(new Lagrange(1));
```

constructs unknown and test functions that live in the space spanned by first-order Lagrange interpolates, i.e., all piecewise linear functions.

6.3.6 Test and unknown parameters

Expression subtypes `TestParameters` and `UnknownParameter` are used to represent test and unknown functions that are independent of space. Their constructors take no arguments. See 6.2.1 for an example of the use of test and unknown parameters.

6.3.7 Discrete functions

Discrete functions represent the value of a field that has been discretized on a space of basis functions. Discrete functions have a number of important uses:

- representing the solution of a finite-element problem
- representing a field for which no analytical expression is available

A discrete function object can be created in a number of ways: by computing the value of an expression on the nodes in a mesh, by reading it from a file, or by "capturing" a solution vector into a discrete function.

6.3.7.1 Creating a scalar-valued discrete function

To create a discrete function, we first need to know the discrete space on which the function will be defined. The construction of this space requires at minimum a mesh, a basis function, and a vector type.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis = new Lagrange(1);
TSFVectorSpace discreteSpace = new SundanceVectorSpace(myMesh, basis, petra);
```

Once you have a discrete space, you can create a discrete function as follows:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x)*sin(y));
```

6.3.7.2 Creating a vector-valued discrete function

Discrete functions representing vector-valued fields have some wrinkles that are important to understand. Consider a discrete function representing a two-component vector field, $\mathbf{u} = (u_x, u_y)$. How is the vector underlying this function stored? One can imagine creating two independent discrete functions

```
Expr ux = new DiscreteFunction(discreteSpace, sin(x)*sin(y));
Expr uy = new DiscreteFunction(discreteSpace, cos(x)*cos(y));
```

and forming a vector-valued expression using the `List` operator,

```
Expr u = List(ux, uy);
```

This is well-defined Sundance code, but it is not usually what you want. A calculation will have improved performance due to cache efficiency if both functions are aggregated into a single vector, with u_x and u_y at each cell listed together. To achieve this aggregation, we need to create a discrete space capable of representing vector-valued functions.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis = new Lagrange(1);
TSFArray<BasisFamily> multiVariableBasis = tuple(basis, basis);
TSFVectorSpace multiVariableDiscreteSpace
  = new SundanceVectorSpace(myMesh, multiVariableBasis, petra);
Expr u = DiscreteFunction::discretize(multiVariableDiscreteSpace,
  List(sin(x)*sin(y), cos(x)*cos(y)));
```

In many problems, it is necessary to use a mixed set of basis functions. For example, in the Taylor-Hood discretization of the incompressible Navier-Stokes equations, the velocity components are represented with 2nd order polynomials and the pressure with 1st order polynomials.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis1 = new Lagrange(1);
BasisFamily basis2 = new Lagrange(2);
TSFArray<BasisFamily> multiVariableBasis = tuple(basis2, basis2, basis1);
TSFVectorSpace multiVariableDiscreteSpace
  = new SundanceVectorSpace(myMesh, multiVariableBasis, petra);
Expr uAndP = DiscreteFunction::discretize(multiVariableDiscreteSpace,
  List(y, x, 0.0));
```

6.3.7.3 Reading a discrete function

Many mesh file formats have the ability to store field data along with the mesh. This field data can be associated with elements or with nodes, depending on the application and the physical meaning of the field. Different mesh file format will index fields in different ways; for example, the Exodus format associates names with fields, while Shewchuk's Triangle format simply lists attributes. Generally, we can look up fields by either a name or by a number indicating the position in an attribute list. Some examples follow:

```
MeshReader reader = new ShewchukMeshReader(‘‘myMesh’’);
Expr temperature = reader.getNodalField(0);
Expr velocity = reader.getNodalField(1, 2, 3)
Expr pressure = reader.getElementalField(4)
```

```
MeshReader reader = new ExodusMeshReader(‘‘myMesh.exo’’);
Expr pressure = reader.getElementalField(‘‘pressure’’);
Expr velocity = reader.getNodalField(‘‘ux’’, ‘‘uy’’, ‘‘uz’’)
Expr temperature = reader.getNodalField(‘‘temperature’’);
```

6.3.7.4 Derivatives of discrete function

Many basis functions used in finite elements calculations are only piecewise differentiable: the function is continuous everywhere and differentiable in the interior of each cell, but the derivative is not defined at boundaries between cells. Such basis functions, and functions represented with them, are said to have C^0 continuity. Since the derivative of such a function will not be continuous at element boundaries, the derivative of a C^0 function is not necessarily C^0. Thus, the derivative of a discrete function defined with a particular discrete space cannot be represented exactly with another discrete function defined with that same space.

For this reason, it is impossible to create directly a discrete function from the derivative of another discrete function. The following will result in a runtime error:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x));
Expr dfdx = new DiscreteFunction(discreteSpace, dx*f);
```

If f is a C^0 function, it is possible to *integrate* derivatives of f. The integral is well-defined since the region on which f is nondifferentiable have no volume. Numerically, it is usually possible to do such integrals because the quadrature points are usually in the interiors of cells. So it’s perfectly sensible, and quite common, to write a weak PDE that includes derivatives of discrete functions.

What is not possible is to obtain pointwise values of the derivative of a discrete function. This is not a common operation during the solution of a PDE, but you may often want to see derivative values during postprocessing and analysis. Because pointwise values are not available, it is impossible to create directly a discrete function from the derivative of another discrete function.

The following will result in a runtime error:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x));
Expr dfdx = new DiscreteFunction(discreteSpace, dx*f);
```

If you really want to look at pointwise derivative values, the best that can be done is to approximate the derivative by projecting into a C^0 space. There are many ways to do this; one of the most common is a least-squares projection, in which you choose coefficients such as to minimize the squared residual.

This is a common enough operation that Sundance has a predefined method for least-squares projection:

```
// f0 is a discrete function
Expr gradF = L2Projection(discreteSpace, List(dx, dy)*f0);
```

Note that since this operation requires the solution of a linear system, it is time-consuming. Again, it usually needs to be done only as a postprocessing step.

Finally, it should be pointed out that the difference between a derivative and its L^2 projection will decrease as the function becomes smoother. For this reason, the L^2 residual of a derivative can be used as an error estimator.

6.3.8 Cell property functions

In some problems, you will need an expression to represent properties of a mesh cell. For example, stabilization methods such as SUPG have terms involving h, the local mesh spacing. In some problems, an explicit expression for a boundary normal is needed.

The local mesh spacing can be obtained using a `CellDiameterExpr`, created as follows.

```
Expr h = new CellDiameterExpr();
```

Similarly, the outward normal of a boundary cell is given by a `CellNormalExpr`, constructed as

```
Expr n = new CellNormalExpr();
```

6.4 Geometric components

6.4.1 Meshes

Sundance can use unstructured meshes in 1, 2, or 3 dimensions. To Sundance, a `Mesh` object is a connected complex of cells. A **zero-cell** is a point. A **maximal** cells is defined with dimension equal to the spatial dimension of the mesh. Each facet of a maximal cell is itself a cell, and so on down to zero cells. Every discrete geometric entity in Sundance is a cell; there is no distinction between “elements”, “edges”, and “nodes”. All are represented by `Cell` objects.

Sundance currently supports the following cell types:

- zero-cells: points
- one-cells: lines
- two-cells: triangles and quadrilaterals (“quads”)
- three-cells: tetrahedra (“tets”) and hexahedra (“bricks” or “hexes”)

The system for representing cells is extensible, so that an advanced user can add additional cell types such as prisms.

Most of the methods of the `Mesh` class are for Sundance’s internal use and will almost never appear at the user level. You will sometimes work with `Cell` objects directly, for instance when probing the value of a function at a point during postprocessing.

6.4.1.1 Mesh I/O

There are almost as many mesh file formats as there are engineers, and it would be foolish to try to build support for file I/O directly into the `Mesh` object. Sundance uses an extensible `MeshReader` class heirarchy to provide an interface for reading from mesh formats. The current version of Sundance supports readers for three mesh formats: a native Sundance text format, Shewchuk’s Triangle format, and Sandia’s Exodus II format. If you want to support some other mesh format you will have to implement your own `MeshReaderBase` subtype.

Using a `MeshReader` is very simple. You create a `MeshReader` object as a handle to an appropriate subtype, and then you call the `readMesh()` method to return a `Mesh` object. The following code reads a mesh in Shewchuk’s Triangle format from files `tBird.1.poly` and `tBird.1.ele`:

```
MeshReader reader = new ShewchukMeshReader("tBird.1");
Mesh mesh = reader.getMesh();
```

Similarly, to write a mesh to Triangle format one does

```
MeshWriter writer = new ShewchukMeshWriter("myMesh");
writer.writeMesh();
```

6.4.1.2 Mesh generator interface

Class `MeshGenerator` provides an interface for mesh generators, and there are implementations for building several simple mesh types. In principle it would be possible to connect a powerful third-party mesh generator to Sundance through the mesh generator interface, but it is generally simpler to have the mesher write the mesh to a file which can be read by a `MeshReader` object.

The most common use of `MeshGenerator` is to build toy meshes for test problems. The three built-in `MeshGenerator` subtypes are

- `LineMesher` meshes a line
- `RectangleMesher` meshes a rectangle with triangles
- `RectanglerQuadMesher` meshes a rectangle with quadrilaterals

6.4.2 Cell sets

A `CellSet` object is used to define a set of cells on which an equation or boundary condition is to be applied. A `CellSet` can be defined independently of any particular mesh; instead of a list of cells, it is a condition or set of condition that can be used to extract a list of cells from a mesh.

6.4.2.1 The set of all maximal cells

The `MaximalCellSet` object identifies all maximal cells in a mesh. The constructor has no arguments:

```
CellSet maxCells = new MaximalCellSet();
```

6.4.2.2 The set of all boundary cells

A `BoundaryCellSet` object identifies all boundary cells of dimension $N - 1$. For example, in a 3D problem a `BoundaryCellSet` will contain all 2D cells on the boundary, but not lines or points that happen to lie on the boundary.

The constructor has no arguments:

```
CellSet boundaryCells = new BoundaryCellSet();
```

6.4.2.3 Defining subsets

Given a cell set, we can use the `subset()` method to define a condition that can extract a subset of the original cell set. The condition can be a mathematical equation or inequality that must be satisfied by any cell to be accepted into the set, or it can be a string label. In “real world” problems the most common condition for defining a cell set will be a label that is associated with the cells by the code that produced the mesh.

```
CellSet boundary = new BoundaryCellSet();
CellSet wall = boundary.subset(‘‘wall’’);
CellSet arc = boundary.subset(x*x + y*y == 1.0 && x < 0.5);
```

6.4.2.4 Logical operations on cell sets

Cell sets can be created by doing set operations – union and intersection – on two or more existing cell sets. Union and intersection are represented by the overloaded addition (`+`) and logical AND (&&) operators.

6.5 Discretization

6.5.1 Basis families

Every unknown field or test function in a Sundance problem must be given a **basis family**.

Currently, the only basis families supported in Sundance are the Lagrange family and the Serendipity family. Lagrange basis functions use Lagrange interpolation about the element’s nodes. Serendipity basis functions are specialized to quadrilateral (“quad”) and hexahedral (“brick”) cells; they require function values on corner and edge nodes only, not on face or center nodes.

6.5.2 Quadrature families

The integrals in a Sundance weak form are done by numerical integration, or quadrature. What is relevant to user-level Sundance code is how one can specify a suite of quadrature rules to be used for a given weak form. Notice that it will not suffice to specify a quadrature rule, because a given term may be integrated on several different cell types. For example, a mesh may contain both quad cells and triangle cells, and the two different cell types will require two different quadrature rules. What is needed is a specification of a *family* of quadrature rules rather than a single rule. The user-level specifier of a family of quadrature rules is the `QuadratureFamily` object. The `buildQuadraturePoints()` method of `QuadratureFamily` returns a set of quadrature points and weights appropriate to a given cell type. The user picks a quadrature family by selecting the appropriate subtype of `QuadratureFamilyBase` and supplying the desired constructor arguments. For example,

```
QuadratureFamily gauss4 = new GaussianQuadrature(4);
```

creates an object that can produce a 4-th order Gaussian quadrature rule for any cell type.

6.5.2.1 Gaussian Quadrature

Gaussian quadrature rules specify both points and weights to give optimal accuracy for all polynomials through a given degree. Gaussian quadrature rules for a line can be derived from the properties of the Legendre polynomials; see any textbook on numerical analysis for a discussion. Gaussian quadrature rules for quadrilaterals and bricks can be formed as “tensor products” of line rules. The development of Gaussian quadrature rules for triangles and tetrahedra is an ongoing research area; an online literature survey through 1998 can be found at Steve Vavasis’ quadrature and cubature page[1]. Symmetric Gaussian quadrature rules through moderate order have been developed for triangles by Dunavant[36] and for tetrahedra by Jinyun[63]. A summary of the quadrature rules that will be generated by Sundance’s `GaussianQuadrature` object is given in the table below.

[1]http://www.cs.cornell.edu/home/vavasis/quad.html

Cell type	**Available orders**	**Reference**	Comments
Line	all	e.g. Hughes[59]	
Triangle	1-12	Dunavant[36]	Orders 3,7, and 11 have negative weights.
Quad	any		Tensor project of two line rules.
Tet	1-6	Jinyun[63]	Order 3 has a negative weight.
Brick	any		Tensor project of three line rules.

6.5.3 Upwinding

Sundance has no built-in upwinding capability, however, it is straightforward to use existing Sundance components to do upwinding via the streamwise upwinding Petrov-Galerkin (SUPG) method.

6.5.4 Specification of row and column space ordering

The order in which equations and unknowns are written can make a difference in the performance of a linear solver, and in keeping with the goal of flexibility, Sundance gives you the ability to specify this ordering. In order to understand how Sundance's ordering specification works, let's look into how Sundance decides unknown and equation numbering.

Given a mesh and a set of unknowns, the Sundance discretization engine will traverse the mesh one maximal cell at a time and find all unknowns associated with that cell and its facets. In a problem with multiple unknowns, say velocity, pressure, and temperature, there can be more than one unknown associated with a cell; if so, the unknowns are assigned in the order that their associated `UnknownFunction` objects are listed in the `StaticLinearProblem` constructor. This scheme gives us two ways to control the unknown ordering:

- **Cell ordering** specifies the order in which cells are encountered as the mesh is traversed.
- **Function ordering** specifies the order in which different functions are listed within a single-cell.

6.5.4.1 Cell ordering

Cell ordering is controlled by giving the linear problem constructor a `CellReorderer` object. Currently, there are two subtypes of `CellReorderer`,

- `RCMCellReorderer` uses the reverse Cuthill-McKee reordering algorithm (e.g., Saad [98]). The RCM algorithm is a modified breadth-first search with desirable behavior during matrix factoring.
- `IdentityCellReorderer` uses the original ordering used by the mesh, i.e., it does no reordering.

The default is `RCMCellReorderer`, and it is a good general choice. You might use `IdentityCellReorder` in cases where your mesh already has a favorable cell ordering, saving the (small) expense of doing an unnecessary reordering.

The cell reordering system is extensible; your favorite reordering algorithm can be added to Sundance by writing a new `CellReorderer` subtype.

The same cell reordering scheme is used for equation numbering (rows) and unknown numbering (columns). Thus, cell reorderings are always symmetric.

6.5.4.2 Function ordering

Function ordering is controlled by the order in which test or unknown functions appear in the linear problem constructor. For example, if `ux`, `uy`, and `p` are unknowns we can order them as: `List(ux, uy, p)`, or as `List(p, ux, uy)` or any of the other permutations. A list with the desired ordering is given to the `StaticLinearProblem` constructor,

```
StaticLinearProblem problem(mesh, eqn, bc, List(vx, vy, q), List(ux,
uy, p), vecType);
```

Notice that the test functions need not have the same ordering as their corresponding unknowns: a nonsymmetric ordering such as

```
StaticLinearProblem problem(mesh, eqn, bc, List(vx, vy, q), List(p,
ux, uy), vecType);
```

is possible.

6.5.5 Block structuring

It is possible to group unknowns and equations into **blocks**, in which case the stiffness matrix becomes a block matrix with each block being an independent object. Sundance's blocking capability makes possible the use of block solvers and preconditioners.

As with function ordering, block structuring is specified by organization of the unknown and test function arguments to the `StaticLinearProblem` constructor.

```
Array<Block> unkBlocks = List(Block(List(U, V), petraType), Block(P, petraType));
```

6.6 Boundary conditions

There are many ways to apply boundary conditions (BCs) in a finite element simulation, and Sundance is designed to be flexible in methods of applying BCs. To begin with, the way a BC gets written depends strongly on the way the weak problem has been formulated; for example, BCs will be written quite differently in least-squares formulations than in Galerkin formulations. For the purposes of user-level Sundance code, the most important classification of boundary conditions is the distinction between BCs that *add into* an expression and BCs that *replace* an expression. BCs that add in to an expression are simply incorporated into an `Integral` object, while replacement-type boundary conditions are specified using `EssentialBC` objects. In Sundance, geometric subdomains are identified using `CellSet` objects. The surface on which a BC is to be applied is specified by passing as an argument the `CellSet` representing that surface.

6.7 Problem manipulation

One of the most powerful features of Sundance is the ability to automate tranformations of problems.

6.7.1 Linearization

It is possible to have Sundance automate the linearization of a nonlinear equation. Automated linearization is restricted to full Newton linearization; alternative linearization schemes such as Oseen must be done by hand.

The `linearization(u, u0)` methods of `Expr` and `EssentialBC` are used to return a new linear expression or BC. Linearization is always done about an initial guess `u0`, which must be a discrete function with the same structure as the unknown argument `u`. The new expression has a new unknown function for the Newton step, or differential, which will have the same structure as the original unknown `u`. Calling `linearization()` on a linear expression simply obtains the same linear expression, but in terms of the Newton step for the original unknown. Note that if *either* the PDE or BC are nonlinear, both must be linearized in order to transform both into equations for the Newton step.

6.7.1.1 Example: Poisson-Boltzmann Equation

For example, the Poisson-Boltzmann equation

$$\int \nabla u \cdot \nabla v + ve^{-u} = 0 \tag{6.7.8}$$

with boundary conditions

$$u(\mathrm{top}) = u_{BC} \tag{6.7.9}$$

can be linearized as follows.

```
Expr eqn = Integral((grad*u)*(grad*v) + exp(-u)*v);
EssentialBC bc = EssentialBC(top, (u - uBC)*v);

Expr linearizedEqn = eqn.linearization(u, u0);
EssentialBC linearizedBC = bc.linearization(u, u0);
```

The resulting expression and BC are equations for the Newton step, accessible as an unknown function through the `differential()` method on the original unknown,

```
Expr du = u.differential();
```

Complete code for the solution of the Poisson-Boltzmann equation (6.7.8) is shown below.

```
#include "Sundance.h"

/** \example inlinePoissonBoltzmann1D.cpp
 * Solve the Poisson-Boltzmann equation \f$\nabla^2 u = e^-u$ on the unit
 * line with boundary conditions:
 * Left: Natural, du/dx=0
 * Right: Dirichlet u = 2 log(cosh(1/sqrt(2)))
 *
 * The solution is 2 log(cosh(x/sqrt(2))).
 *
 * The problem is nonlinear, so we use Newton’s method to iterate
 * towards a solution.
 *
 */

int main(int argc, void** argv)
{
  try
    {

      Sundance::init(&argc, &argv);

      /* create a simple mesh on the unit line */
      double L=1.0;
      int n = 10;
      MeshGenerator mesher = new PartitionedLineMesher(0.0, L, n);
      Mesh mesh = mesher.getMesh();

      /* define an expression representing the x-coordinate function */
      Expr x = new CoordExpr(0);

      /* create a cell set representing the right boundary */
      CellSet boundary = new BoundaryCellSet();
      CellSet right = boundary.subset( x == L );

      /* create a discrete space on the mesh */
      TSFVectorSpace discreteSpace
        = new SundanceVectorSpace(mesh, new Lagrange(2));

      /* create an expression for the initial guess. This will be reused as the
       * starting point for each newton step. Assume u(x)=x as an initial
       * guess, and discretize it.
       */
      Expr u0 = new DiscreteFunction(discreteSpace, x);

      /* create symbolic objects for test and unknown functions. At each newton
       * step we will solve a linearized equation for a step du, so our
       * unknown is du. */
      Expr u = new UnknownFunction(new Lagrange(2), "du");
      Expr v = new TestFunction(new Lagrange(2), "du");

      /* create a differential operator representing the x-derivative. */
      Expr dx = new Derivative(0);

      /* linearized weak equation for the step du */
      Expr nonlinearEqn = Integral((dx*u)*(dx*v) + v*exp(-u));
      Expr linearizedEqn = nonlinearEqn.linearization(u, u0);
      Expr du = u.differential();

      /* Dirichlet boundary condition */
      double uBC = 2.0*log(cosh(L/sqrt(2.0)));
      EssentialBC bc = EssentialBC(right, (u-uBC)*v) ;
      EssentialBC linearizedBC = bc.linearization(u, u0);
      /* linear problem for the step du */
      StaticLinearProblem prob(mesh, linearizedEqn, linearizedBC, v, du);

      /* create linear solver */
      TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
      TSFLinearSolver solver = new BICGSTABSolver(1.0e-12, 1000);


      NewtonLinearization newton(prob, u0, solver);
      Expr soln = newton.solve(NewtonSolver(solver, 8, 1.0e-12, 1.0e-12));

      // compare to exact solution
      Expr exactSoln = 2.0*log(cosh(x/sqrt(2.0)));
      Expr error = new DiscreteFunction(discreteSpace, soln-exactSoln);

      /* write to matlab */
      string filename = "pb1D." + TSF::toString(MPIComm::world().getRank())
        + ".dat";
      FieldWriter writer = new MatlabWriter(filename);
      writer.writeField(soln);

      // compute the norm of the error
      double errorNorm = (exactSoln - soln).norm(2);
      double tolerance = 1.0e-4;
      TSFOut::printf("error = %g\n", errorNorm);

      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();

}
```

6.7.2 Variations

Automated calculation of variations can be useful in a number of ways. For PDEs that can be derived from a variational principle, Sundance's variational capability can be used to derive the PDE. A particularly interesting application of this is to derive a FOSLS discretization from a least-squares functional. Another important use of automated variational calculations is to obtain the first-order necessary conditions for optimality.

An example of the use of the `variation()` method to obtain first-order optimality conditions for a PDE-constrained optimization problem was shown in section 6.2.

6.7.3 Sensitivities: Gradients and Hessians

With some optimization algorithms, we will want to evaluate the gradient or Hessian of an objective function with respect to a field u or parameter α. In Sundance, this is done using the `directSensitivity` method of `Expr`.

6.8 Linear Algebra and Solvers

There are many high-quality numerical linear algebra packages in use, so Sundance is designed to allow third-party linear algebra packages to be imported as plugins. All numerical linear algebra in Sundance is done using the Trilinos Solver Framework (TSF), and the TSF in turn supports plugins of third-party types.

User specification of a linear algebra representation is done by means of a `TSFVectorType` object. This object knows how to build a vector space given a mesh and set of functions, and the vector space in turn knows how to build a vector of the appropriate type.

6.9 Transient problems

Currently, Sundance has no high-level support for transient simulations. However, it is not difficult to code simple timestepping schemes directly in Sundance.

Consider Crank-Nicolson (BE) time discretization for the transient heat equation. If we discretize in time but leave space undiscretized for the moment, the step from u_i to u_{i+1} is given by the PDE

$$u_{i+1} - u_i = \frac{1}{2}\delta t \left[\nabla^2 u_{i+1} + \nabla^2 u_i\right] \tag{6.9.10}$$

plus associated BCs. We can now solve this equation using Sundance, and the solution may be used as the starting value for the next step.

```
#include "Sundance.h"

/**
 * \example timeStepHeat1D.cpp
 *
 * This example shows how to do timestepping in Sundance. We solve the
 * transient heat equation in one dimension using Crank-Nicolson time
 * discretization. The time discretization is done at the symbolic level.
 * Spatial discretization is done via StaticLinearProblem, yielding system
 * matrices and vectors that can be used to march the problem in time.
 *
 * We solve the heat equation u_xx = u_t with boundary conditions
 * u(0)=u(1)=0 and initial conditions u(x,t=0)=sin(pi x). The solution
 * is u(x,t)=exp(-piˆ2 t) sin(pi x).
 */

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(&argc, &argv);

      /* create a simple mesh on the unit line */
      int n = 100;
      MeshGenerator mesher = new LineMesher(0.0, 0.5, n);
      Mesh mesh = mesher.getMesh();

      /* create unknown and variational functions */
      Expr delU = new TestFunction(new Lagrange(1));
      Expr U = new UnknownFunction(new Lagrange(1));

      /* create a differentiation operator */
      Expr dx = new Derivative(0);

      /* the initial conditions will be u0(x,t=0) = sin(pi*x).
       * create a coordinate expression to represent x, then
       * create sin(pi*x), and then project it onto a discrete function. */
      Expr x = new CoordExpr(0);

      double pi = 4.0*atan(1.0);

      TSFVectorSpace discreteSpace
        = new SundanceVectorSpace(mesh, new Lagrange(1));

      Expr u0 = new DiscreteFunction(discreteSpace, sin(pi*x));


      /*
         set up crank-nicolson stepping with timestep = 0.02. The time
         discretization is done at the symbolic level, yielding
         an elliptic problem that we solve repeatedly for the updated
         solution at each time level.
      */

      double deltaT = 0.02;
      Expr cnStep = delU*(U - u0) + deltaT*(dx*delU)*(dx*(U + u0)/2.0);
      Expr eqn = Integral(cnStep);

      /* Define BCs to be zero at both ends */
      CellSet boundary = new BoundaryCellSet();
      CellSet left = boundary.subset( fabs(x - 0.0) < 1.0e-10 );
      CellSet right = boundary.subset( fabs(x - 1.0) < 1.0e-10 );
      EssentialBC bc = EssentialBC(left, delU*U);
      /* create a solver object */
      TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
      TSFLinearSolver solver = new BICGSTABSolver(prec, 1.0e-14, 300);

      /*
         put the time-discretized eqn into a StaticLinearProblem object
         which will do the spatial discretization.
      */
      StaticLinearProblem prob(mesh, eqn, bc, delU, U);

      /*
         Now, loop over timesteps, solving the elliptic problem for u at each
         step. At the end of each step, assign the solution solnU into u0.
         Because Exprs are stored by reference, the updating of u0 propagates
         to the copies of u0 in the equation set and in the
         StaticLinearProblem. The same StaticLinearProblem can be reused
         at all timesteps.
      */
      int nSteps = 100;
      for (int i=0; i<nSteps; i++)
        {
          /* solve the problem */
          Expr soln = prob.solve(solver);
          TSFVector solnVec;
          soln.getVector(solnVec);
          u0.setVector(solnVec);
          /* write the solution at step i to a file */
          char fName[20];
          sprintf(fName, "timeStepHeat%d.dat", i);
          ofstream of(fName);
          FieldWriter writer = new MatlabWriter(fName);
          writer.writeField(u0);
          cerr << "[" << i << "]";
          /* flush the matrix and RHS values */
          prob.flushMatrixValues();
        }
      cerr << endl;

      /* compute the exact solution and the error */
      double tFinal = nSteps * deltaT;
      Expr exactSoln = exp(-pi*pi*tFinal) * sin(pi*x);

      /*
        compute the norm of the error
      */
      double errorNorm = (exactSoln-u0).norm(2);
      double tolerance = 1.0e-4;

      /*
        decide if the error is within tolerance
      */
      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();
}
```

Chapter 7

Sundance Optimization Survey

7.1 Sundance-rSQP++ Interface

Here we describe the basics of a software interface that allows rSQP++ to solve (possibly in parallel) PDE-constrained optimization problems that are modeled using Sundance. One of the requirements for this interface was that it should be as easy as possible (and require as little new code as possible) to prototype a new optimization application. There are several different aspects to a Sundance-rSQP++ application that are logically independent of each other and the software structure reflects this separation of concerns. Before going into the specifics of the software structure, we describe the different independent components that have to be dealt with. A few of these independent components are (1) the statement of the PDE-constrained optimization problem, (2) the linear algebra implementation, and (3) how the optimization problem is solved.

The basic linear algebra implementations used by a Sundance application is determined by an abstract `TSF::`*`TSFVectorType`* object. For example, as shown in Chapter 6, every `Sundance::DiscreteFunction` and `Sundance::StaticLinearProblem` object must have a *`TSFVectorType`* object passed into their constructors. By parameterizing a Sundance application with a *`TSFVectorType`* object and a compatible `TSF::`*`TSFLinearSolver`* object, the specification of the linear algebra implementations is completely determined. The interface `NLPInterfacePack::`*`SundanceProblemFactory`* (or *`SPF`* for short) has been defined to abstract the Sundance PDE-constrained optimization formation. The `SPF` interface allows the development of a Sundance optimization formulation that is independent of the linear algebra implementation and this component is discussed in Section 7.1.2.

Another critical part of the Sundance-rSQP++ interface is the linear algebra interface. Sundance uses the Trilinos Solver Framework (TSF) as its abstract interface to linear algebra in much the same way that rSQP++ uses AbstractLinAlgPack (ALAP for short). Both TSF and ALAP support RTOp operators and have a very similar object model (both are based on HCL [51], but with ALAP to a lesser extent) so it was fairly trivial to develop the “Adapter” [42] subclasses to put ALAP interfaces on TSF linear algebra objects. The details of this ALAP-TSF interface are discussed in Section 7.1.1. The particulars of these basic linear algebra interfaces are not of much concern to individuals that simply want to use Sundance-rSQP++ to prototype PDE-constrained optimization problems. To relieve basic users for the concerns about linear algebra implementations used by Sundance, by a simple concrete C++ class called NLPInterface-Pack::SundanceLinAlgFactory (see Figure 7.2) has been developed that automates the tasks of allowing users to select basic options and of creating compatible *TSFVectorType* and *TSF::TSFLinearSolver* objects that are used by a specific *SPF* object to define the Sundance optimization problem. The example main program in Section 7.1.4 shows how this class is used to specify the linear algebra for a Sundance-rSQP++ optimization problem.

Finally, once the linear algebra implementations and the PDE optimization problem have been defined, the last component to specify is the optimization algorithm. This is where rSQP++ comes in. The primary interface to rSQP++ is through the abstract base class NLPInterfacePack-::*NLPFirstOrderInfo* (see Section 4.2.3.2). The subclass NLPInterfacePack::NLP-Sundance implements this interface for Sundance optimization problems. The details of the NLPSundance subclass are described in Section 7.1.2.

7.1.1 AbstractLinAlgPack-TSF Linear Algebra interface

The Sundance-rSQP++ interface uses the explicit partitioning of variables into states and controls as shown in Chapter 2. In order to implement the Sundance-rSQP++ interface, vector-space objects for the state and control variables, a general matrix object for the sub-Jacobian for the controls $\frac{\partial c}{\partial u}$ or N and a non-singular matrix object for the sub-Jacobian of the states $\frac{\partial c}{\partial y}$ or C are all needed. Figure 7.1 shows a UML class diagram for the adapter subclasses required for the ALAP-TSF interface. These interface classes are collected into a separate project library called Abstract-LinAlgPackTSF. These adapter classes are very straightforward and require little explanation but some simple comments are in order.

The **VectorWithOpMutableTSF** adapter simply forwards RTOp operators through the *apply_reduction(...)* and *apply_transformation(...)* methods on to the aggregate *TSFVectorBase* object (through a TSFVector handle object). That is basically the extent

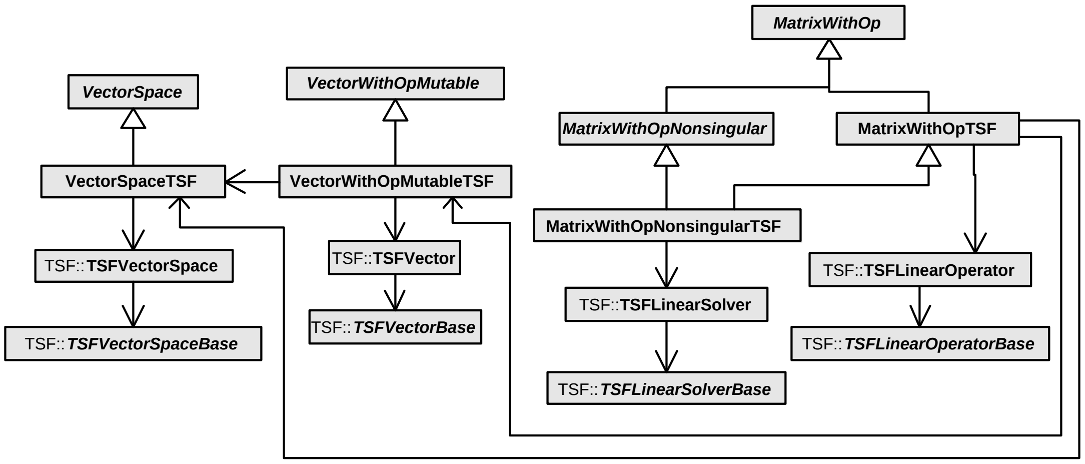


Figure 7.1. UML class diagram : `AbstractLinAlgPackTSF`, Adapter subclasses for `ALAP-TSF` interface

of this subclass. Through these operator methods (which have the same basic implementation) all of the advanced features of the rSQP++ algorithms can be implemented through specialized `RTOp` objects.

The **`VectorSpaceTSF`** adapter uses the *`createMember()`* method of the aggregate *`TSFVectorSpaceBase`* object to implement the `create_member()` method and returns a `VectorWithOpMutableTSF` with an embedded `TSF` vector object.

The **`MatrixWithOpTSF`** adapter simply forwards the vector arguments (after some dynamic casting to get the `TSF` objects) from the *`Vp_StMtV(...)`* method on to the *`TSFLinearOperatorBase`* object though its *`apply(...)`* or *`applyAdjoint(...)`* methods (depending on the value of the transpose argument).

Since the *`TSFLinearOperatorBase`* interface defines the methods *`applyInverse(...)`* and *`applyInverseAdjoint(...)`*, it would seem that the every *`TSFLinearOperatorBase`* object should be able to support the *`MatrixWithOpNonsingular`* interface but this is not the case. Instead, the subclass **`MatrixWithOpNonsingularTSF`** is needed which requires a *`TSFLinearSolverBase`* object to solve for linear systems in the method *`V_InvMtV(...)`*. The inverse methods of on the `TSF` operator object are ignored since there is no guarantee that they will be implemented for a particular linear operator object. This was an important concept that was discovered during the development of the Sundance-rSQP++ interface.

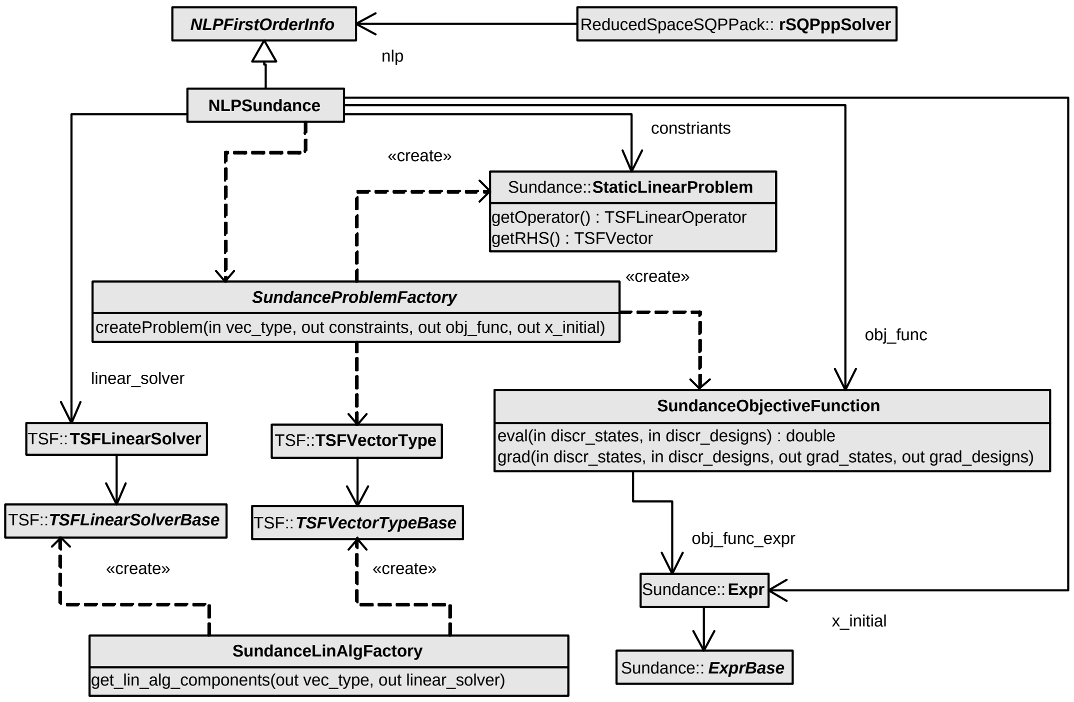

NLPFirstOrderInfo
ReducedSpaceSQPPack:: rSQPppSolver
nlp
NLPSundance
constriants
«create»
Sundance::StaticLinearProblem
getOperator() : TSFLinearOperator
getRHS() : TSFVector
«create»
SundanceProblemFactory
createProblem(in vec_type, out constraints, out obj_func, out x_initial)
obj_func
linear_solver
SundanceObjectiveFunction
eval(in discr_states, in discr_designs) : double
grad(in discr_states, in discr_designs, out grad_states, out grad_designs)
TSF::TSFLinearSolver
TSF::TSFVectorType
TSF::TSFLinearSolverBase
TSF::TSFVectorTypeBase
obj_func_expr
«create»
«create»
Sundance::Expr
x_initial
SundanceLinAlgFactory
get_lin_alg_components(out vec_type, out linear_solver)
Sundance:: ExprBase

Figure 7.2. UML class diagram : Sundance-rSQP++ interface

See the Doxygen documentation for the package `AbstractLinAlgPackTSF` for more details on this `ALAP-TSF` interface.

7.1.2 *`NLPSundance`*: Interface between Sundance PDE-Constrained Optimization Problems and rSQP++

Figure 7.2 shows the basic interfaces and subclasses that make up the Sundance-rSQP++ interface. Users create subclasses of the *`SPF`* interface to implement a new PDE-constrained optimization problem. A *`SPF`* object, along with `TSFVectorType` and `TSFLinearSolver` objects, are passed to the constructor of the NLP subclass `NLPSundance`.

The *`createProblem(...)`* method of the *`SPF`* object is called by the `NLPSundance` object to create `Sundance::StaticLinearProblem` and `SundanceObjectiveFunction` objects. The `StaticLinearProblem` object is used to represent the set of under-determined nonlinear constraints $c(y, u)$ shown in (2.1.2). Associated with a `StaticLinearProblem` object must be a `Sundance::Expr` object (called `x_initial` in the figure), that contains the Sundance discrete functions with the initial guess for the states and controls. These discrete functions must be used to form the variational equations for the PDE constraints. The setup of the `StaticLinearProblem` object must be done in a specific way in order to be used with `NLPSundance` which is shown in the below example program.

The class `SundanceObjectiveFunction` is not a built-in Sundance class but was developed for the Sundance-rSQP++ interface to encapsulate how the objective function and its gradients are computed. The concrete *`SPF`* object creates a `Sundance::Expr` object that represents the objective function (see Section 7.1.4 for an example) and it is this expression object that gets embedded in the `SundanceObjectiveFunction` object that is returned to the `NLPSundance` object.

The `NLPSundance` object extracts `TSFVector` objects from the `Sundance::Expr` object `x_initial` for the initial guess for the states and the controls as

```
x_initital[0].getVector(states_vec);
x_initital[1].getVector(controls_vec);
```

and then builds a `AbstractLinAlgPack::VectorSpaceCompositeStd` object for the concatenation of the spaces of the states and the controls

```
states_spc   = states_vec.getSpace();
controls_spc = controls_vec.getSpace();
```

into a single `AbstractLinAlgPack::`*`VectorSpace`* object. Much of the machinery for handling the mapping from different spaces and vectors for the states y and the controls u to a single space and vector for the variables $x^T = \begin{bmatrix} y^T & u^T \end{bmatrix}$ is implemented by the same `AbstractLinAlgPack::BasisSystemCompositeStd` subclass used by the simple example NLP described in Section 7.1.4. This basis-system subclass also handles the formation of a `AbstractLinAlgPack::`*`MatrixWithOp`* object for the gradient matrix `Gc` (∇c).

7.1.3 PDE Constraints

Here we carefully spell out how the constraints must be modeled using Sundance to form a `StaticLinearProblem` object that is returned from *`SPF::createProblem(...)`*. What is required is that the discrete functions for the solution variables and the unknown functions be blocked into single vectors for the states and controls. For example, if r, s and t are the unknown (i.e. `Sundance::UnknownFunction`) state variables and v, w are the unknown control variables, then these variables must be combined into state y and control u variables as

$$\begin{aligned} y &= \begin{bmatrix} r \\ s \\ t \end{bmatrix} \\ u &= \begin{bmatrix} v \\ w \end{bmatrix}. \end{aligned}$$

This is required so that the `TSFLinearOperator` Jacobian object `Jac` returned from `Jac = StaticLinearProblem::getOperator()` is a block matrix where `C = Jac.getBlock(0,0)` is the basis matrix object for the states while `N = Jac.getBlock(0,1)` is the non-basis matrix object for the controls. See the Doxygen documentation for the class *`SPF`* for details on the assertions for the constraints object.

The example program in Section 7.1.4 shows how this blocking is done in the simple case of single unknown variables for the states and controls. Blocking of multiple variables for states and controls is shown in the 2-D Burger's example.

7.1.4 Example Sundance-rSQP++ Application

In this section, we describe the solution of the following PDE-constrained optimization problem that is modeled by the 1-D Poisson-Boltzman equation

$$\begin{aligned} \text{min} \quad & \tfrac{\beta}{2}\int_{\partial\Omega_r}(u-\hat{u})^2 + \tfrac{\beta-1}{2}\int_{\partial\Omega}(\alpha^2) && && (7.1.1) \\ \text{s.t.} \quad & \nabla^2 u - e^{-u} = 0 && \text{on } \Omega && (7.1.2) \\ & u(x) = \alpha && \text{on } \partial\Omega && (7.1.3) \end{aligned}$$

where $\Omega = [a,b]$, $\hat{u}$ is the target value of the state on the right boundary $\partial\Omega_r$, α is a boundary control function, and β is an objective weighting term that balances the control objective (for

u) with the regularization term (for α). The header file for the *SundanceProblemFactory* subclass for this problem is shown below.

```
// ///////////////////////////////////////////////////////////////////
// SPFPoissonBoltzman1D.h

#ifndef SPF_POISSONBOLTZMAN1D_H
#define SPF_POISSONBOLTZMAN1D_H

#include "SundanceProblemFactory.h"
#include "RTOpPack/include/RTOp_config.h"
#include "NewtonSolver.h"

namespace NLPInterfacePack {


class SPFPoissonBoltzman1D: public SundanceProblemFactory
{
public:


    SPFPoissonBoltzman1D(
        value_type left, value_type right, int n, value_type uRight
        ,value_type uGuess, value_type aGuess, value_type obj_wgt
        );


    void createProblem(
        const TSF::TSFVectorType                                  &vec_type
        ,MemMngPack::ref_count_ptr<Sundance::StaticLinearProblem> *constriants
        ,MemMngPack::ref_count_ptr<SundanceObjectiveFunction>     *obj_func
        ,Expr                                                     *x_initial
        ) const;
    const Mesh& getMesh() const;
    const CellSet& controlsCellSet() const;


private:
    Mesh mesh_;
    Expr coord_x_;
    CellSet boundary_;
    CellSet right_;
    CellSet left_;
    value_type uRight_;
    value_type uGuess_;
    value_type aGuess_;
    value_type obj_wgt_;
};

} // end NLPInterfacePack

#endif
```

Note that the header file `SPFPoissonBoltzman1D.h` is basically just boiler-plate code with the exception of some of the private data members on lines 54–62. The interesting code comes in the source file which is shown below.

```
01 // ///////////////////////////////////////////////////////////////////
02 // SPFPoissonBoltzman1D.cpp
03
04 #include "../include/SPFPoissonBoltzman1D.h"
05 #include "PartitionedLineMesher.h"
06
07 namespace NLPInterfacePack {
08
09 SPFPoissonBoltzman1D::SPFPoissonBoltzman1D(
10     value_type left, value_type right, int n, value_type uRight
11     ,value_type uGuess, value_type aGuess, value_type obj_wgt
12     )
13     :coord_x_( new CoordExpr(0)), uRight_(uRight), uGuess_(uGuess)
14     ,aGuess_(aGuess), obj_wgt_(obj_wgt)
15 {
16 #ifdef RTOp_USE_MPI
17     MeshGenerator mesher = new PartitionedLineMesher(left, right, n);
18     mesh_ = mesher.getMesh();
19 #else
20     MeshGenerator mesher = new LineMesher(left, right, n);
21     mesh_ = mesher.getMesh();
22 #endif
23     // Define cell sets for the boundry and left and right edges
24     boundary_ = new BoundaryCellSet();
25     left_     = boundary_.subset( fabs(coord_x_ - left)  < 1.0e-10 );
26     right_    = boundary_.subset( fabs(coord_x_ - right) < 1.0e-10 );
27 }
28
29 void SPFPoissonBoltzman1D::createProblem(
30     const TSF::TSFVectorType                                  &vec_type
31     ,MemMngPack::ref_count_ptr<Sundance::StaticLinearProblem> *constraints
32     ,MemMngPack::ref_count_ptr<SundanceObjectiveFunction>     *obj_func
33     ,Expr                                                     *x_initial
34     ) const
35 {
36     namespace mmp = MemMngPack;
37     using Sundance::List;
38
39     // Dimension of the finite-element basis functions used
40     const int u_basis_dim = 1, a_basis_dim = 1;
41
42     // Discrete state space on the entire mesh and discrete control space on boundary
43     TSFVectorSpace discreteStateSpace
44         = new SundanceVectorSpace(mesh_, new Lagrange(u_basis_dim), vec_type);
45     TSFVectorSpace discreteControlSpace
46         = new SundanceVectorSpace(mesh_, new Lagrange(a_basis_dim), boundary_, vec_type);
47
48     // Expression for the initial state and controls which are also used for the linearization
49     Expr u0     = new DiscreteFunction(discreteStateSpace,   uGuess_, "u0");
50     Expr alpha0 = new DiscreteFunction(discreteControlSpace, aGuess_, "alpha0");
51
52     // Create the initial point for the optimizer
53     *x_initial = List(u0, alpha0);
54
55     // Test and unknown functions for the state and control
56     Expr u     = new UnknownFunction(new Lagrange(u_basis_dim), "u");
57     Expr alpha = new UnknownFunction(new Lagrange(a_basis_dim), "alpha");
58     Expr v     = u.variation();
59
60     // Nonlinear state equation and boundary conditions
61     Expr dx = new Derivative(0);
62     Expr nonlinearStateEqn = (dx*(u))*(dx*v) + v*exp(-u);
63     EssentialBC nonlinearBC = EssentialBC( boundary_, (u-alpha)*v, new GaussianQuadrature(1) ) ;
64
65     // Integrated linearized state equation and boundary conditions
66     Expr linearizedStateEqn = nonlinearStateEqn.linearization( List(u,alpha), *x_initial );
67     EssentialBC bc = nonlinearBC.linearization( List(u,alpha), *x_initial );
68     Expr eqn = Integral(linearizedStateEqn, new GaussianQuadrature(4));
69
70     // Arrange test and (Newton) unknowns into [state, control] blocks
71     Expr du = u.differential(), dAlpha = alpha.differential();
72     TSFArray<Block> unkBlocks = tuple(Block(du, vec_type), Block(dAlpha, vec_type));
73     TSFArray<Block> varBlocks = tuple(Block(v, vec_type));
74
75     // Create the static linear problem for the step [du, dAlpha]
76     *constraints = mmp::rcp( new StaticLinearProblem(mesh_, eqn, bc, varBlocks, unkBlocks) );
77
78     // Define the objective function.
79     const Expr obj_func_expr
80         = Integral(right_,0.5*obj_wgt_*(u-uRight_)*(u-uRight_)) // Control objective
81         + Integral(boundary_,0.5*(1.0 - obj_wgt_)*alpha*alpha); // Regularization
82     *obj_func = mmp::rcp(
83         new SundanceObjectiveFunction(
84             obj_func_expr, mesh_, u, alpha ) );
85
86 }
   ...
98 } // end NLPInterfacePack
```

Lines 9–27 in this source file contain the constructor which accepts the parameters for the problem and then sets up the mesh and the cell sets for the boundaries of interest. The input parameters for this problem are the domain (`left` and `right`) the number of finite elements (`n`), the target value for the state (`uRight`), the guess for the state and control (`uGuess` and `aGuess`) and the objective function wieght (`obj_wgt`). On lines 17-18 the mesh is set up for parallel execution while lines 20–21 set up for serial execution. The boundary cell sets are specified on lines 24–26.

The most important part of this subclass is of course the implementation of the `create-Problem(...)` method that begins on line 29. The dimensions of the finite-element basis functions are specified at the top of the function on line 40. Next, the `TSF` vector spaces are defined for the states and controls on lines 43–46. Note that the input `vec_type` argument is used as part of the definition for these vector spaces which determines the implementations for the vectors. Also note that the space for the control `alpha` is only defined on the boundary as shown in line 46 and not over the entire domain. This is a very useful feature that allows great flexibility in defining what data can be determined by the optimizer and what data can be specified up front. Given these `TSF` vector space objects, the discrete functions for the states and the controls are defined on lines 49–50 and are supplied with initial guesses. These discrete functions represent the current estimate for the solution and are used as the unknowns in the optimization problem. Later, these discrete functions are used to define the linearized equations. On line 53, the initial guess for the states and controls is packed into an expression `x_initial` which is later returned to the `NLPSundance` object.

Lines 56–63 define the set of nonlinear equations (state equation (7.1.2) on line 62 and the boundary condition (7.1.3) on line 63). What makes this set of equations different from for a standard Sundance problem is that a test function is only defined for the states on line 58 and not for the controls which results in a set of under-determined equations. This set of nonlinear equations must be linearized and this is done using the `linearization(...)` method on lines 66–67. The linearized state equation is then integrated over the entire domain on line 68.

Now that the linearized state equations and boundary conditions have been defined as Sundance expressions, we must tell Sundance to properly block the variables as described in Section 7.1.3. This is done on lines 72–73. Note that the blocked unknown variables are the Newton steps `du` and `dAlpha` and not the original unknowns `u` and `alpha` used to define the nonlinear equations. Finally, the `StaticLinearProblem` object is created on line 76 for the linearized state equation and boundary conditions. This is the constraints object that is returned to the `NLPSundance` object which is used to compute the residual for the nonlinear constraints (which happens to be the negative `TSF` vector returned from the `getRHS()` method) and the Jacobian (which is returned for the `getOperator()` method).

The final part of the `createProblem(...)` method is the definition of the objective function on lines 79–84. First, the expression for the objective function is defined on lines 80-81. Note the domains that the objective terms are integrated over and how they compare to (7.1.1). This expression for the objective is passed into the constructor for a `SundanceObjectiveFunction` object on lines 82–84. Note that this constructor must be given the mesh object and the unknown functions used to define the states and the controls. The constraint object `constraints` and the objective-function object `obj_func` are then returned to the calling `NLPSundance` object along with the initial guess `x_initial`. The discrete functions embedded in the `x_initial` expression object are manipulated by the `NLPSundance` object in order to compute the constraint residual and Jacobian and different iterates.

The last piece of user code to write for this optimization problem is the main driver program. This program is shown in the below source file.

```
// ////////////////////////////////////////////////////////////
// NLPPoissonBoltzman1DMain.cpp

#include <iostream>
#include "../include/SundanceNLPSolver.h"
#include "../include/SPFPoissonBoltzman1D.h"
#include "../include/SundanceLinAlgFactory.h"

int main(int argc, char* argv[] ) {

    namespace NLPIP == NLPInterfacePack;
    using CommandLineProcessorPack::CommandLineProcessor;

    int prog_return; // return code

    // Step 1: Initialize Sundance (i.e. MPI)
    NLPIP::SundanceNLPSolver::init();
    Sundance::init(&argc, (void***)&argv);

    try {

        NLPIP::SundanceNLPSolver       sundance_nlp_solver;
        NLPIP::SundanceLinAlgFactory   lin_alg_fcty;

        // Step 2: Read in input

        double left    = 0.0;
        double right   = 1.0;
        int    n       = 2;
        double uRight  = 2.0*log(cosh(right/sqrt(2.0)));
        double uGuess  = 0.1;
        double aGuess  = 0.2;
        double obj_wgt = 0.99;

        CommandLineProcessor  command_line_processor;

        command_line_processor.set_option( "left",  &left,    "x at the left boundary" );
        command_line_processor.set_option( "right", &right,   "x at the right boundary" );
        command_line_processor.set_option( "n",     &n,       "Number of finite elements" );
        command_line_processor.set_option( "uRight",&uRight,  "Value at the right boundary" );
        command_line_processor.set_option( "uGuess",&uGuess,  "The Guess for u" );
        command_line_processor.set_option( "aGuess",&aGuess,  "The Guess for a" );
        command_line_processor.set_option( "obj_wgt",&obj_wgt,"[0,1] Wieghting for u or a (1.0: all u, 0.0: a
        lin_alg_fcty.setup_command_line_processor( &command_line_processor );
        sundance_nlp_solver.setup_command_line_processor( &command_line_processor );

        CommandLineProcessor::EParseCommandLineReturn
            parse_return = command_line_processor.parse_command_line(argc,argv,&std::cerr);

        if( parse_return != CommandLineProcessor::PARSE_SUCCESSFULL )
            return parse_return;

```

```
 53         // Step 3: create the linear algebra components
 54         TSF::TSFVectorType    vec_type;
 55         TSF::TSFLinearSolver  linear_solver;
 56         lin_alg_fcty.get_lin_alg_components( &vec_type, &linear_solver );
 57 
 58         // Step 4: Create the SundanceProblemFactory
 59         NLPIP::SPFPoissonBoltzman1D probfac(left,right,n,uRight,uGuess,aGuess,obj_wgt);
 60 
 61         // Step 5: Solve the NLP (or the forward problem)
 62         prog_return = sundance_nlp_solver.solve(vec_type, MemMngPack::rcp(&probfac,false), &linear_solver );
 63 
 64     }// end try
 65     catch( const std::exception& except ) {
 66         cerr << "\nCaught as std::exception : " << except.what() << std::endl;
 67         prog_return = -1; //  ToDo: return proper enum!
 68     }
 69 
 70     // Step 6: Finalize Sundance (i.e. MPI)
 71     Sundance::finalize();
 72 
 73     return prog_return;
 74 }
```

As with any Sundance application, `Sundance::init(...)` and `Sundance::finalize()` must be called as shown on lines 18 and 71. The next section of code (lines 27–51 in the `try` block) performs the input of the command-line parameters for the optimization problem that are passed into the constructor for the `SPFPoissonBoltzman1D` object that is created on line 59. Command-line options for the `SundanceLinAlgFactory` object declared on line 23 are inserted into the command-line processor object on line 44 after the application specific options. These options are read from the command-line on line 48. Options are also processed for a `SundanceNLPSolver` object which controls a lot of the default behavior that is independent of the particular problem being solved. To see all of the valid command-line options, the option `--help` can be specified on the command line and will cause the program to print a help message, which for this executable is

```
Usage: ./sundance_nlp_bolt [options]
  options:
  --help                              Prints this help message
  --left                      double  x at the left boundary
                                      (default: --left=0)
  --right                     double  x at the right boundary
                                      (default: --right=1)
  --n                         int     Number of finite elements
                                      (default: --n=2)
  --uRight                    double  Value at the right boundary
                                      (default: --uRight=0.463163)
  --uGuess                    double  The Guess for u
                                      (default: --uGuess=0.1)
  --aGuess                    double  The Guess for a
                                      (default: --aGuess=0.2)
  --obj_wgt                   double  [0,1] Wieghting for u or a (1.0: all u, 0.0: all a)
                                      (default: --obj_wgt=0.99)
  --use-petra                 bool    Determine if Petra (parallel) or serial (LAPACK) linear algebra is used
  --use-serial                        (default: --use-petra)
  --use-aztec                 bool    Determine if Aztec or the default BICGSTAB solver is used
  --use-bicgstab                      (default: --use-bicgstab)
  --ilu_fill                  int     Fill-in factor for ILU
                                      (default: --ilu_fill=1)
  --ilu_overlap               int     Overlap for ILU
                                      (default: --ilu_overlap=1)
  --iter_solve_tol            double  Solve tolerance for iterative solver
                                      (default: --iter_solve_tol=1e-10)
  --iter_solve_maxiter        int     Maximum number of iterations for iterative solver
                                      (default: --iter_solve_maxiter=5000)
  --do-optimization           bool    Determine if optimization or simulation problem is solved
  --do-simulation                     (default: --do-optimization)
  --root-process              int     Index (zero-based) of the root process
                                      (default: --root-process=0)
  --states-guess-file         string  Filename where initial guess for states data is stored (same format as 'sta
                                      (default: --states-guess-file="")
  --controls-guess-file       string  Filename where initial guess for states data is stored (same format as 'con
                                      (default: --controls-guess-file="")
  --states-sol-file           string  Filename where solution for states data is written (flat values only)
                                      (default: --states-sol-file="")
  --controls-sol-file         string  Filename where solution for contorls data is written (flat values only)
                                      (default: --controls-sol-file="")
  --states-matlab-sol-file    string  Filename where solution for states data is written (matlab format)
                                      (default: --states-matlab-sol-file="")
  --controls-matlab-sol-file  string  Filename where solution for contorls data is written (matlab format)
                                      (default: --controls-matlab-sol-file="")
  --max_nl_iter               double  Simulation maximum number of nonlinear iterations
                                      (default: --max_nl_iter=1000)
  --resid_tol                 double  Simulation tolerance (in the ||.||2 norm) for the constraints
                                      (default: --resid_tol=1e-08)
  --compute-gradient          bool    Compute the reduced gradient or not
  --no-compute-gradient               (default: --no-compute-gradient)
  --use-adjoints              bool    Compute the reduced gradient with adjoints or direct sensitivities
  --use-direct                        (default: --use-direct)
```

Once the command-line options are read in, the linear algebra implementations selected by the user on the command-line are created on lines 54–56. By using a `SundanceLinAlgFactory` object, every Sundance optimization problem can automatically support new linear algebra options whenever they are added.

After the linear algebra implementations have been defined and the concrete *`SundanceProblemFactory`* object has been initialized, the rest of the code required to solve the optimization problem is exactly the same for every application. This common code is encapsulated in a helper object of type `SundanceNLPSolver` which is called on line 62. This helper class care of creating a `NLPSundance` object (which in turn calls the `createProblem(...)`

method on the *`SundanceProblemFactory`* object) and then passes this NLP object on to a `rSQPppSolver` object which attempts to solve the optimization problem. A status value (`program_return`) is then is returned from the main driver program to the shell. Any exceptions that are throw (that are not caught elsewhere) will be caught and reported to `std::cerr` on lines 65–68.

7.1.5 The `SundanceNLPSolver` helper class

The `SundanceNLPSolver` class does more than just solve the NLP. It also also allows any *`SundanceProblemFactory`* object to be used to solve the forward simulation problem where the control variables are fixed at the initial guess. The above example Possion-Boltzman program, or any of the other example programs, can be used to solve the forward problem by selecting the command-line argument `--do-simulation`. This will result in the simulation only being performed with the finial objective function value being output to a file called `nle_sol_file.out`. In this mode, the reduced gradient at the converged simulation can also be computed by setting the option `--compute-gradient`. The options `-use-adjoints` and `--use-direct` select the adjoint verses the direct sensitivity methods for computing this reduced gradient respectively. The adjoint method is by far the most efficient.

The ability to do the forward simulation and to compute exact reduced gradients (using both the adjoint and the direct approaches) allows any Sundance/rSQP++ application to also be used in lower-level NAND methods such as described in Chapter 2.

7.2 Example Sundance Optimization Application - Source Inversion of a Convection Diffusion System

Several forward problems from the Sundance test directory have been converted to optimization problems to test the rSQP++ interface. The direct and adjoint interfaces were tested on a heat transfer, Burgers, and a convection diffusion problem. More in depth analyses were conducted using a source inversion problem constrained by convection-diffusion equations. In addition to testing the rSQP++/Sundance interface, the objective of this exercise was to present numerical efficiencies associated with all 7 levels of optimization, and demonstrate this formulation to solve the "chemical/biological attack on a large facility" problem, similar to the work done with MPSalsa. However, in this case we tested the inversion problem with large numbers of inversion parameters.

By specifying a limited number of state values at various points in the domain as targets, a least-squares formulation constrained by a convection-diffusion PDE is used to determine the location of the original source(s) on the boundary. In the case of chemical diffusion, these state values could be concentrations and in the case of heat diffusion these state values could be temperatures. We obtain from a forward simulation 16 "sensor" locations out of 1600 grid points as targets, which are then used in the inversion problem. Since this is an ill-posed problem, a regularization term needs to be added to the objective function. Three obvious options can be considered: the square of f, the square of ∇f and finally the square root of ∇f. Unfortunately as a result of an implementation limitation, the boundary inversion can not make use of gradient based terms for the regularization and therefore the numerical experiments were conducted with the square of f. Our formulation allows locating the source term anywhere on one of the boundaries (i.e. Γ_F):

$$\min_{c,f} \quad \frac{1}{2}\sum_{i}^{s}\int_{\Omega}\delta(\mathbf{x}-\mathbf{x}_i)(c-c^*)^2+\frac{\rho}{2}\int_{\Gamma_F} f^2 \tag{7.2.4}$$

$$\text{s.t.} \quad -k\Delta c+\nabla c\cdot\mathbf{v}=0, \quad \text{in} \quad \Omega \tag{7.2.5}$$

$$\frac{\partial c}{\partial n}=0, \quad \text{on} \quad \Gamma_N \tag{7.2.6}$$

$$c=0, \quad \text{on} \quad \Gamma_D \tag{7.2.7}$$

$$c=f, \quad \text{on} \quad \Gamma_F \tag{7.2.8}$$

where:

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} (-L+y)(L+y) \\ 0 \end{bmatrix}$$
$$\Omega = \{(x,y) : (-L \le x \le +L) \wedge (-L \le y \le +L)\}$$
$$\Gamma_N = \{(x,y) : (-L \le x \le +L) \wedge (y=-L \vee y=+L)\}$$
$$\Gamma_D = \{(x,y) : (x=+L) \wedge (-L \le y \le +L)\}$$
$$\Gamma_F = \{(x,y) : (x=-L) \wedge (-L \le y \le +L)\}$$

where $\delta(\mathbf{x}-\mathbf{x}_i)$ is a delta function that specifies the location of the sensors, c is the vector of calculated state value (concentrations), c^* is the vector of concentration measurements (or targets), ρ is the regularization parameter which is set to 1E-5 for our numerical experiments, f is the source/inversion term, k is the diffusivity constant, and $\mathbf{v}$ is the velocity field. The velocity field is given for this problem and makes (7.2.5) linear in c and therefore no Newton iterations are required to converge to the solution for the forward problem. Figure 7.3 shows the forward simulation on a 40x40 grid (i.e. $n_x = 40$ and $n_y = 40$ finite elements per dimension) for a Gaussian-like source on the left boundary.

Figure 7.4 shows the solution for the inversion problem defined in (7.2.4)–(7.2.8) on a 40x40 grid. The rSQP algorithm was able to successfully solve the problem and recover the entire profile. Small oscillations on the boundary are observed which may be reduced by choosing a different regularization term. Additional experiments were conducted to evaluate different regularization terms and are presented in the next section. The sensor data used for this NLP was taken from a 160x160 forward problem with the same source shown in Figure 7.3.

The source inversion problem was used to demonstrate the numerical efficiencies of the 7 optimization levels by inverting for the boundary source using different grid resolutions. For levels 1-3 we used rSQP++ through the DAKOTA framework and for levels 4 and 5 we used the rSQP++/Sundance interface. Level 6 was not solved for the boundary inversion problem because of implementation limitations. Instead an inversion problem was solved using the full space method where inversion parameters are located within the domain. As the formulation in (7.2.8) suggests, the number of inversion parameters scales with the size of the boundary. The numerical experiment was conducted on a Pentium 4, 2.1 GHz processor and even though Sundance and rSQP++ are parallel capable, the experiments were run serially. Each optimization level was used to complete the inversion for a grid size of 10x10, 20x20, 40x40, 80x80, and a 160x160 grid. The number of inversion parameters matched the size of the grid dimension of a single sided boundary (10, 20,40,80 and 160 inversion parameters). The convergence criteria is controlled by various tolerances, but in our experiment we choose to match objective functions as closely as possible. Table 7.1 shows the objective function values and CPU times for various levels of optimization methods. As expected, the lower-level optimization methods are not able to efficiently drive the objective value down to levels comparable to the higher-level methods.

Figure 7.5 shows graphically the numerical results. Level 0 used a local coordinate pattern search and it is the least efficient algorithm for this problem. These methods are obviously not preferred for smooth and differentiable processes but we have include the results for completeness. Level 1 shows a considerable improvement over level 0 as a result of using gradient information. Direct sensitivities for both NAND and SAND show significant improvements over level 1 because the reduced gradients for level 1 are calculated through finite differences which requires the convergence of a simulation for each inversion parameter. Calculating reduced gradients with direct sensitivities avoids this numerical overhead. Additional seperation between NAND and SAND methods using direct sensitivities can be expected if this had been a non-linear problem.

The adjoint sensitivities are by far the most efficient method to calculate the reduced gradient. There is a significant difference between NAND and SAND because of the simulation overhead that NAND incurs at each optimization iteration. This difference is better observed in Figure 7.6. One would expect that a non-linear simulation problem would incur additional Newton iterations

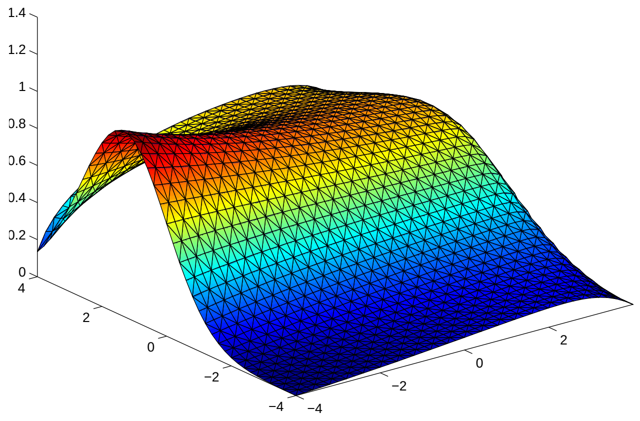


Figure 7.3. Forward simulation for 40x40 boundary source

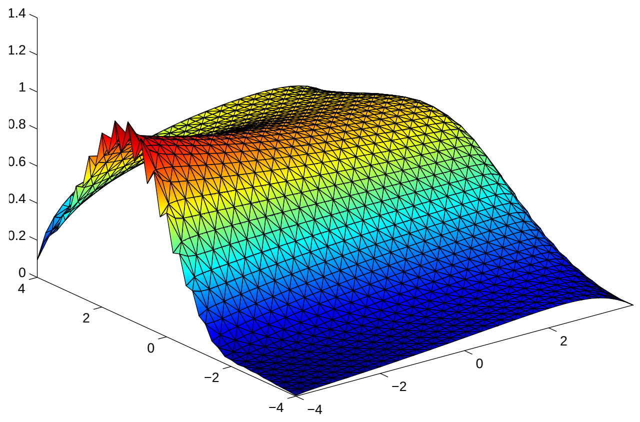


Figure 7.4. Inversion for 40x40 boundary source

Method	$n_x = n_y = 10$	$n_x = n_y = 20$	$n_x = n_y = 40$	$n_x = n_y = 80$	$n_x = n_y = 160$
Sim	0.591	2.119	8.214	32.831	134.396
L-0 Inv	13974.8	31239.3	-	-	-
L-1 Inv	1278.63	1642.32	5385.14	27128.3	-
L-2 Inv	58.5	182.5	293.4	1840.8	22003.2
L-3 Inv	55.1	165.8	465.8	882.8	3620.4
L-4 Inv	9.47	17.32	55.87	835.65	13911
L-5 Inv	8.6	13.0	26.6	151.1	986.5

Method	$n_x = n_y = 10$	$n_x = n_y = 20$	$n_x = n_y = 40$	$n_x = n_y = 80$	$n_x = n_y = 160$
Sim	-	-	-	-	-
L-0 Inv	7.79e-2	5.94e-2	-	-	-
L-1 Inv	9.41e-3	2.52e-5	1.89e-5	1.79e-5	-
L-2 Inv	8.64e-3	1.37e-5	1.70e-5	1.65e-5	1.18e-5
L-3 Inv	8.64e-3	1.37e-5	1.70e-5	1.65e-5	1.18e-5
L-4 Inv	8.61e-3	1.32e-5	1.18e-5	1.18e-5	1.18e-5
L-5 Inv	8.61e-3	1.32e-5	1.18e-5	1.18e-5	1.18e-5

Table 7.1. Summary of CPU times / objective function values for source-inversion on a boundary.

which would add to the NAND expense for each optimization iteration and the gap between levels 3 and 5 would be even greater. Estimated times for level 6 are presented that equals three times the forward simulation cost. This is a conservative estimate considering the full space inversion of a 40x40 grid with 1600 inversion parameters required less than 10 seconds to converge.

The exact value of $\frac{1}{2}\int_{\Gamma_F} f^2$ for the source shown in Figure 7.3 is 1.1788 (to five significant figures). Therefore, for $\rho = 1 \times 10^{-5}$, the minimum value of the objective function in (7.2.4) that can be obtained for a perfect inversion is 1.1788×10^{-5}. The actual objective function value must always be larger than this since the regularization term causes the solution to perturb the sensor matching term in (7.2.4) resulting in an overall elevated objective function. Without the regularization term, the theoretical objective function value should be near zero. This explains why the objective function value for discretization of 40x40 and larger obtain an objective value of 1.18×10^{-5} which is less than one percent off from the perfect inversion value of 1.1788×10^{-5}.

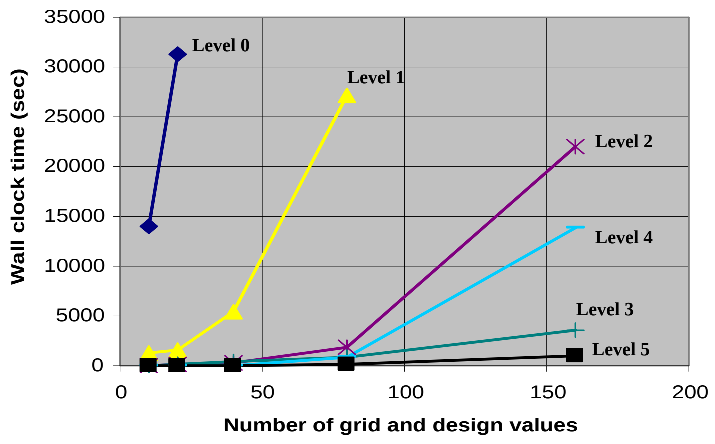


Figure 7.5. Numerical Results for Source Inversion for Convection Diffusion for levels 0-5

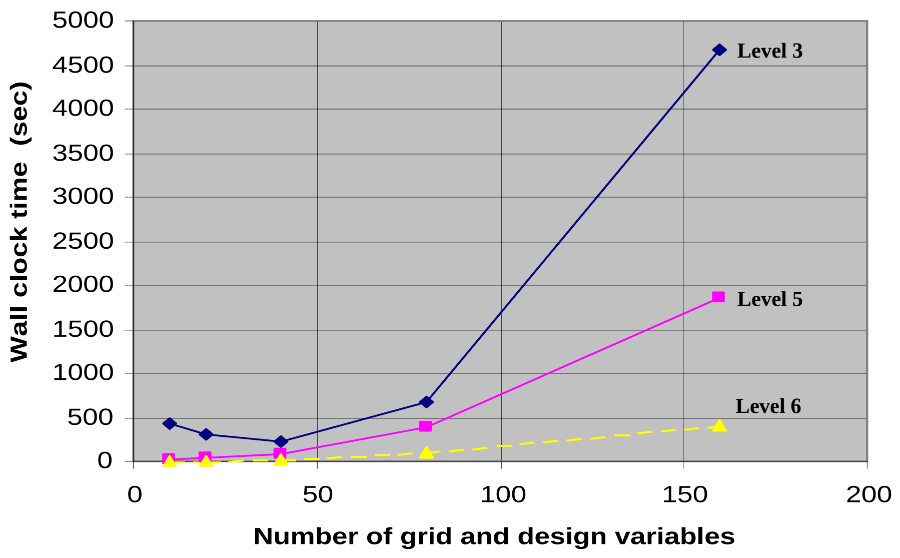


Figure 7.6. Numerical Results for Source Inversion for Convection Diffusion levels 3-6

7.2.1 Inverse problem formulation

We investigate a full space solution methods (level 6) using the source inversion problem where the inversion parameter is located anywhere in the domain. We formulate the problem as follows:

$$
\begin{aligned}
\min_f \frac{1}{2}\sum_i^s \int_\Omega \delta(\mathbf{x}-\mathbf{x}_i)(c-c^*)^2\,d\Omega + \frac{\rho}{2}\int_\Omega p(f)\,d\Omega& \\
\text{subject to:}& \\
-k\Delta c + \nabla c\cdot\mathbf{v} + f = 0, \quad \text{in} \quad \Omega,& \\
\frac{\partial c}{\partial n} = 0, \quad \text{on} \quad \Gamma_N,& \\
c = 0, \quad \text{on} \quad \Gamma_D.& \qquad (7.2.9)
\end{aligned}
$$

The $p(f)$ term in the second part of the objective function is a regularization functional, with ρ as the regularization parameter. If the error was measured throughout the domain, no regularization is needed, the inverse problem can in fact be seen as a matching control problem—which is known to have a unique solution for small Peclet numbers. However, discrete measurements imply multiple solutions, and thus some regularization is necessary. Possible functionals for $p(\cdot)$ are:

$$\int_\Omega f^2\,d\Omega \qquad (7.2.10)$$

$$\int_\Omega \nabla f\cdot\nabla f\,d\Omega \qquad (7.2.11)$$

$$\int_\Omega (\nabla f\cdot\nabla f)^{\frac{1}{2}}\,d\Omega. \qquad (7.2.12)$$

We use the following notation:

$$a(f_1,f_2) := \int_\Omega k\nabla f_1\cdot\nabla f_2\,d\Omega,\ (f_1,f_2) := \int_\Omega f_1 f_2\,d\Omega,\ (f_1,f_2)_\Gamma := \int_\Gamma f_1 f_2\,d\Gamma$$

The Lagrangian functional that corresponds to (7.2.9) is given by:

$$
\begin{aligned}
\mathcal{L}(c,f,\lambda) :=& \frac{1}{2}\sum_i \int_\Omega \delta(\mathbf{x}-\mathbf{x}_i)(c-c^*)^2\,d\Omega + \frac{\rho}{2}(f,f) \\
&+ a(\lambda,c) + (\nabla c\cdot\mathbf{v},\lambda) + (f,\lambda) \qquad (7.2.13)
\end{aligned}
$$

Assuming that the regularization term for the inversion force f is given by (7.2.10) the weak formulation for the optimality conditions (Karush-Kuhn-Tucker conditions) of (7.2.9) is the following:

Find $f, c, \lambda, \in H^1(\Omega)$ such that

$$
\begin{aligned}
a(\psi, c) + (\nabla c \cdot \mathbf{v}, \psi) + (f, \psi) = 0, \forall \psi \in H^1(\Omega) \\
\sum_i (\delta(\mathbf{x} - \mathbf{x}_i)(c - c^*)\psi) + a(\psi, \lambda) + (\nabla \psi \cdot \mathbf{v}, \lambda) = 0, \forall \psi \in H^1(\Omega) \\
\rho(\psi, f) + (\psi, \lambda) = 0,\ \forall \psi \in H^1(\Omega).
\end{aligned}
\tag{7.2.14}
$$

7.2.2 Algorithm

There are several ways to solve the optimality conditions. rSQP++ uses a block-elimination procedure to solve (7.2.14). Given f first solve for c

$$
a(\psi, c) + (\nabla c \cdot \mathbf{v}, \psi) + (f, \psi) = 0, \forall \psi \in H^1(\Omega);
$$

then solve

$$
\sum_i (\delta(\mathbf{x} - \mathbf{x}_i)(c - c^*)\psi) + a(\psi, \lambda) + (\nabla \psi \cdot \mathbf{v}, \lambda) = 0, \forall \psi \in H^1(\Omega)
$$

for λ; and finally solve

$$
\rho(\psi, f) + (\psi, \lambda) = 0,\ \forall \psi \in H^1(\Omega).
$$

to update f. The block-elimination has been used as a preconditioner for (7.2.14). Here we solve the resulting KKT conditions simultaneously:

$$
\begin{bmatrix} \mathbf{W}_{cc} & \mathbf{W}_{cf} & \mathbf{A}_c^T \\ \mathbf{W}_{fc} & \mathbf{W}_{ff} & \mathbf{A}_f^T \\ \mathbf{A}_c & \mathbf{A}_f & \mathbf{0} \end{bmatrix}
\begin{Bmatrix} \mathbf{c} \\ \mathbf{f} \\ \boldsymbol{\lambda} \end{Bmatrix}
= -
\begin{Bmatrix} \mathbf{g}_c + \mathbf{A}_c^T \boldsymbol{\lambda} \\ \mathbf{g}_f + \mathbf{A}_f^T \boldsymbol{\lambda} \\ \mathbf{c} \end{Bmatrix}.
\tag{7.2.15}
$$

7.2.3 Numerical Experiments

Various experiments were conducted using this full space formulation including the evaluation of regularization terms, number of sensors, and number of sources. Both the total variation and Tikhonov regularization were evaluated and for our source selections both were able to recover the original source at similar levels of quality. The total variation regularization however, makes the objective function nonlinear which then requires a Newton method. Because the Tikhonov regularization term makes the objective function linear and thereby requiring no Newton iterations, we used Tikhonov for all of our experiments.

The full space (level 6) method is the most efficient in comparison to levels 0 to 5. Even though we do not have a consistent comparison and if we could generate a boundary inversion problem using full space methods, it would most likely not be as numerically taxing as the full domain inversion problem. The full domain inversion problem converged under 10 seconds whereas level 2, 3, 4, and 5 for the boundary inversion problem for the same size grid (but with smaller number of inversion parameters) converged in 293, 465, 55, and 26 seconds respectively.

Figures 7.7 and 7.8 show results for using different number of sensors.

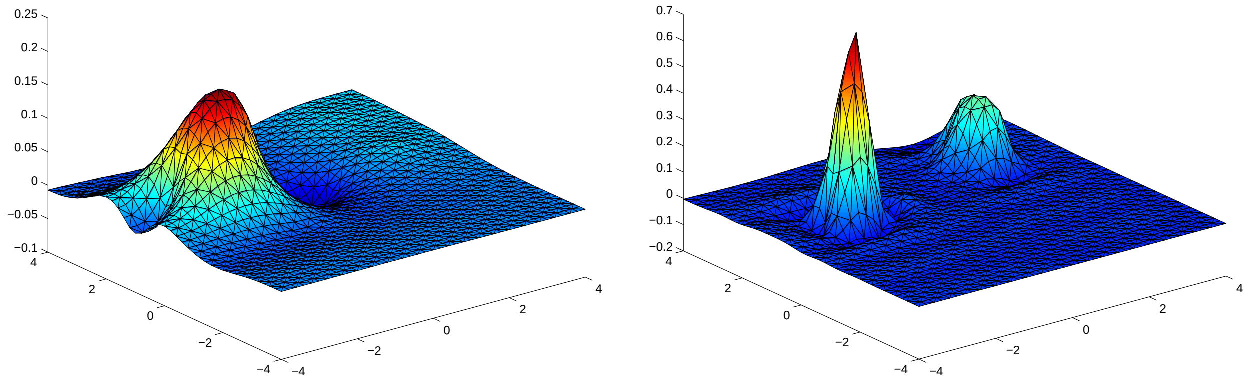


Figure 7.7. Signal Inversion, left 4x4 sensors, right 10 x 10 sensors

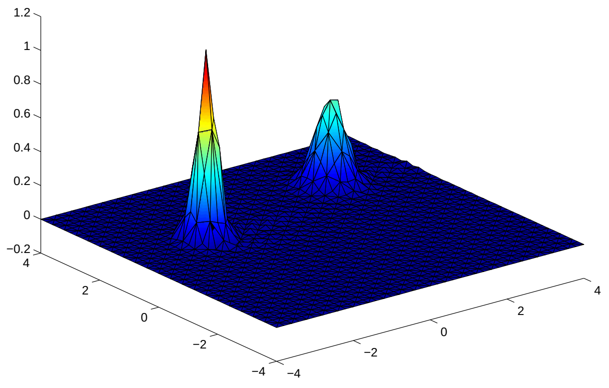


Figure 7.8. Signal Inversion, 40x40 sensor

Chapter 8

Split, O3D and Hierarchical Control

8.1 Overview

Split is a full-space sequential quadratic programming (SQP) algorithm for general large-scale nonlinear programming problems. As noted above, SQP methods proceed by forming at each major step a quadratic programming (QP) approximation to the general problem at the current iterate. The solution of this QP provides a step to adjust the variables and the associated Lagrange multipliers. Thus, an important part of any successful SQP algorithm is a robust QP solver. In this chapter, we concentrate on O3D, our interior-point QP solver that has many advantages for this particular application. We also briefly describe Split. Finally, we discuss our approach to formulating certain control problems where there are multiple objectives, leading to a novel class of hierarchical control problems. Our formulation yields more practical answers than traditional approaches, i.e., our answers tend to be smoother and more robust.

An important part of our research in SQP methods was the design and development of software to implement these methods. Both Split and O3D were implemented to be compatible with Sundance and Trilinos (TSF) and thus be able to take advantage of the unprecented control of the PDE systems that Sundance provides. Earlier in this project, we experimented with a Java implementation and the use of “proxy vectors.” Although these experiments did not work as hoped, we report on this work and the conclusions that we can draw from them.

8.2 O3D

Although quadratic programs arise in independent applications, the primary emphasis in this report is their appearance as a step generator for the solution of general nonlinear programming problems. In this context, there exist numerous features of an algorithm for solving quadratic programs that would be particularly useful, but would not necessarily be of value in a stand-alone solver. This statement is especially true in the large scale case where procedures that approximate the solution can lead to substantial efficiencies. Here, we examine the issues of (approximately) solving large scale quadratic programming problems in the context of the sequential quadratic programming (SQP) algorithm for solving general nonlinear programming problems.

The general nonlinear programming problem can be taken to be of the form

$$\begin{array}{lrcl} \underset{x}{\text{minimize}} & f(x) & & \\ \text{subject to:} & g(x) & \leq & 0 \\ & h(x) & = & 0 \end{array} \qquad (NLP)$$

where $f : \mathcal{R}^n \rightarrow \mathcal{R}^1$, $g : \mathcal{R}^n \rightarrow \mathcal{R}^{m_1}$ and $h : \mathcal{R}^n \rightarrow \mathcal{R}^{m_2}$. At each step of the SQP algorithm a quadratic programming approximation to (NLP) is constructed and its solution is used as a step to improve the current iterate. More specifically we construct a quadratic program of the form

$$\begin{array}{lrcl} \underset{\delta}{\text{minimize}} & c^{\mathsf{t}}\delta + \frac{1}{2}\delta^{\mathsf{t}}Q\delta & & \\ \text{subject to:} & A\delta + b & \leq & 0 \\ & F\delta + d & = & 0 \end{array} \qquad (QP)$$

where $\delta \in \mathcal{R}^n$, $Q \in \mathcal{R}^{n\times n}$, $A \in \mathcal{R}^{m_1\times n}$, $b \in \mathcal{R}^{m_1}$, $F \in \mathcal{R}^{m_2\times n}$, and $d \in \mathcal{R}^{m_2}$.

Let

$$\ell(x,\lambda,\mu) = f(x) + \lambda^{\mathsf{t}}g(x) + \mu^{\mathsf{t}}h(x)$$

be the Lagrangian of (NLP) with multipliers $\lambda \in \mathcal{R}^{m_1}$ and $\mu \in \mathcal{R}^{m_2}$. If the current iterate is x^kthen the correspondence between (NLP) and (QP) is as follows:

$$\begin{array}{rcl} c & = & \nabla f(x^k) \\ A & = & \nabla g(x^k) \\ b & = & g(x^k) \\ F & = & \nabla h(x^k) \\ d & = & h(x^k) \end{array}$$

and Q is a symmetric approximation to the Hessian of the Lagrangian at (x^k, λ^k, μ^k).

The subproblem (QP) is approximately solved to yield the step δ^k and the next iterate is calculated by

$$x^{k+1} = x^k + \alpha\delta^k$$

where α is the steplength To guarantee convergence α must be chosen carefully. Typically a merit function is used to guide this choice. A merit function is a scalar-valued function whose reduction implies progress towards to solution. Thus an important factor in using (QP) as a step generator is to ensure that approximate solutions are such that they are descent directions on the merit function. (See [26] for a more complete discussion of the general issues concerning SQP methods.)

There are numerous issues to consider in solving (QP) and the resolution of these issues depends on whether or not (QP) is to be solved as a stand-alone problem or as a subproblem. First, for a solution to exist, the constraints must be consistent, i.e., there must be at least on point δ such that all of the constraints are satisfied. If the constraints of a stand-alone problem are inconsistent, then it suffices for a solver simply to report this fact back to the user. In the context of SQP, however, it is often the case that subproblems are inconsistent, so a procedure must be devised to use the subproblem to create a descent direction for the merit function. Even if the constraints are consistent, there is no guarantee that a feasible point will be given, so a "phase I" procedure must also be provided.

Another consideration is the nature of Q. If Q is positive definite, then the solution of (QP) is unique. Otherwise, multiple local minima may exist and the question arises of which solution is desired. Furthermore, if Q is indefinite (or negative definite) and the feasible set is not bounded, then there may exist unbounded solutions. When (QP) is a subproblem for SQP, it is reasonable to assume that a solution that is too large will not be of interest since one would expect that (QP) would only be a reasonable approximation for (NLP) in a relatively small region about x^k. From a computational point of view, if Q is indefinite, then directions of negative curvature may be possible to construct and exploit. Finally, Q may be in the form of a quasi-Newton update or a limited memory quasi-Newton update in which case it will represented as a low rank update of a scaled identity matrix.

A quadratic program solver for a stand-alone problem would probably only return the solution x^* and the associated multipliers, along with an indication of which of the inequality constraints are active. This would in turn require procedures to estimate the multipliers and convergence criteria to halt the iteration. For a subproblem solver, there need to be additional features to control the solution process. In particular, as noted above, the length of the step may be important, e.g., in trust-region algorithms, and there it is important to have reasonable estimates of the multipliers when the solver is terminated since these are often used in constructing approximations to the Hessian of the Lagrangian. In addition, several different termination criteria may be needed and, as noted above, a procedure to produce a descent step on the merit function even if the constraints are inconsistent.

Different SQP algorithms can be constructed that favor certain applications and these factors influence the choice of underlying quadratic program solvers. For example, applications with a large number of highly nonlinear inequality constraints should probably be handled differently from applications with only mildly nonlinear constraints. Similarly, the algebraic structure of problems with a large number of equality constraints might receive special consideration. The applications also account for structure in the gradients of the constraints and in the Hessian of the Lagrangian. In large scale problems, this structure often needs to be considered carefully both in the formulation of the problem and in the solution techniques. For example, in the control of partial differential equations one can trade off the size of the problem with the nonlinearity of the problem, i.e., one can sometimes construct a very large, but mildly nonlinear problem or a smaller, but more nonlinear one.

We are primarily concerned with applications in which there is a large number of nonlinear inequality constraints. First, we briefly discuss some features and properties of QP solvers that affect their use in the SQP setting. We then describe the interior-point method, $O3D$, and its properties that show it to be a good candidate for a step generator in an SQP algorithm. Next, we suggest enhancements of $O3D$ that improve the performance and robustness of the basic algorithm. Finally, we discuss its implementation and preliminary numerical results.

In light of the above discussion of the types of quadratic programming problems that arise in (NLP) applications, we briefly review here the issues that should be addressed in designing a QP solver that can be used as a step generator for SQP.

8.2.1 The Constraints

The proper handling of the constraints has a profound effect on many other aspects of the algorithm. In this discusion, we assume that the constraints are consistent; inconsistent constraints are discussed under the heading of early termination below.

First, there is certainly no guarantee that an initial feasible point will be given. Thus a Phase I procedure to compute one must be specified. Typically such a procedure uses a “Big M” method wherein the feasible region is enlarged to enclose the initial approximation to the solution and then shrunk to its original size in the course of the subsequent calculations. If such a method is used, then the size of M, the initial size of the associated "artificial variable," the nature of the enlargement (i.e., should all of the constraints be changed, or just a few), and the procedure to ensure that the artificial variable is reduced, must be specified. The issues in terminating the algorithm while still in Phase I are similar to those in the case of inconsistent constraints and are discussed below.

Although a given initial point may be infeasible, it is possible that it is "close" to the optimal solution. Such a situation is called a "warm start." Interior-point methods have usually not been amenable to taking advantage of warm starts, but significant savings may be possible if such information could be exploited. We intend to investigate this issue further.

8.2.2 Multiplier Estimates

Estimating the multipliers is a particularly difficult problem for primal methods, i.e., primal-dual methods would seem to have an obvious advantage. In the case of nonconvex problems, however, this advantage is not so clear. The issue is to design a method to estimate the multipliers that gives reasonable approximations when far from the optimal solution, and does so at reasonable computational cost. Degenerate constraints, a common occurance in large scale problems, give rise to nonunique multipliers and a lack of strict complimentarity, but degeneracy itself typically does not pose a difficulty for interior-point methods. In primal methods, estimating the multipliers requires a determination of which constraints are active at the solution. This is not an easy task. Our approach for this is to use so-called "Tapia indicators" to estimate the active set, followed by a particulary simple interior-point method on the dual problem. We have implemented this idea and have recorded some excellent results on some highly degenerate problems, including problems with equality constraints where we are guaranteed to have degeneracy.

8.2.3 Early Termination

All of the above considerations are exacerabed by the possibility of early termination of the algorithm. For the solution to be useful in the SQP setting, it must be such that it leads to a descent direction on the merit function. It is often easier to show that the optimal solution has the required descent properties than that an approximate solution does. A major cause of this is that when far from the optimal solution, the determination of the active set and the associated multipliers is especially problematic. This is even more difficult in the case of nonconvex problems. The effect of poor multiplier estimates is to create difficulties in the SQP algorithm which uses these estimates for calculations involving the Lagrangian.

In order to terminate early, the algorithm must have a set of criteria for testing the adequacy of the current approximation. Standard convergence criteria may be adequate for obtaining a highly accurate solution, but may not be particularly good at determining the adequacy of more remote solutions. For example, poor multiplier estimates may may cause a good approximate solution to appear to be much poorer. Criteria for termination may include a simple test to terminate if the length of the solution exceeds a certain length. Such a procedure may allow useful steps in unbounded problems.

8.2.4 Computational Issues

The overriding concern for any QP solver that is to be used as a step generator is that the it be computationally efficient. Most of the work in solving (QP) using an interior-point method is in the solution of the underlying linear system of equations. Using direct factorization methods is quite efficient as long as the linear systems are not too large. If larger problems are to be solved, then iterative methods must be considered. Issues of how to precondition these systems and how accurately they must be solved remain to be addressed. Our implementation using TSF facillitates experimentation with iterative methods and, more importantly, with preconditioners.

8.2.5 Recentering in O3D

Recentering in O3D is an attempt to prevent the early iterates from staying too close to the boundary of the feasible region and thus significantly slowing the algorithm. Complete recentering involves moving the current iterate as far as possible towards the center of the polytope along the level curve corresponding to the current iterate. Because doing this in the full space is prohibitively expensive, we had developed a method based on doing the recentering in a subspace, in the same spirit as $O3D$ itself. Although this procedure had often reduced the number of iterations modestly in early test problems, it did not appear to be effective on the problems arising from PDE constraints. The reasons for this are not entirely clear, and several attempts to improve this procedure were not successful. These attempts included constructing the subspace to be orthogonal to the 3-dimensional subspace generated by the main $O3D$ algorithm and increasing the dimension to four or five.

We finally developed an entirely new approach which has turned out to be much simpler and much more effective. The main idea is move along the Newton recentering direction from the full $O3D$ step until it intersects the level curve corresponding to the current value. Using this idea has proved to be quite effective. The computational cost to form the step is a third that of the subspace recentering algorithm and never more than one iteration is needed as compared to an average of five iterations for the old method. In two early tests, we found unbounded solutions after only a few iterations whereas the old method was still making slow progress after 1000 iterations. More testing here is necessary to tune the entire algorithm.

8.2.6 The O3D Algorithm

$O3D$ is a primal method, implying that it only operates in the primal space and does so by attempting to reduce the objective function. It does this by forming a 3-dimensional approximation to (QP) that can be easily solved. In particular, $O3D$ generates three independent directions at the given feasible point. These directions and the feasible point determine a 3-dimensional affine space. The reduced (QP) is then taken to be the original (QP) restricted to this space. The three-dimensional problem is then solved and the next (strictly feasible) iterate is taken to be 99% of the distance to the boundary in this direction or to the minimum of the quadratic in this direction. Convergence is checked and the procedure repeated as necessary. In discussing the details, we consider only the inequality constraints in (QP); equalities can be included by writing them as two inequalities.

We assume that an initial strictly feasible point δ^0 is given and that the algorithm generates a sequence $\{\delta^k\}$ as follows. Let $\{p_i,\ i = 1, \ldots, 3\}$ be a set of normalized vectors (that depend on δ^k) and set

$$H_k = \gamma_k Q + A^{\mathrm{t}} (D_k)^2 A$$

where D_k is the diagonal matrix whose j^{th} diagonal component is

$$d_j = 1/(A\,\delta^k + b)_j$$

and γ_k is a positive scalar. Denoting by P_k the $n \times 3$ matrix whose columns are the p_j we set

$$\tilde{\delta}^{k+1} = \delta^k + H_k^{-1} P_k\,\zeta.$$

where $\zeta \in \mathcal{R}^3$. Substituting this value of $\tilde{\delta}^{k+1}$ into the (QP) we obtain the 3-dimensional "baby" problem

$$\begin{aligned} &\underset{\zeta}{\text{minimize}} \quad \tilde{c}^{\mathrm{t}}\zeta + \tfrac{1}{2}\zeta^{\mathrm{t}}\tilde{Q}\zeta \\ &\text{subject to: } \tilde{A}\zeta + \tilde{b} \le 0 \end{aligned} \tag{8.2.1}$$

where

$$\begin{array}{rcl} \tilde{c} & = & P_k\, H_k^{-1}(c + Q\,\delta^k), \\ \tilde{b} & = & A\delta^k + b, \\ \tilde{Q} & = & P_k^{\rm t} H_k^{-1} Q\, H_k^{-1} P_k \qquad \text{and} \\ \tilde{A} & = & A\, H_k^{-1} P_k \end{array} \tag{8.2.2}$$

The solution procedure for (8.2.1) is discussed below; here we simply assume that a solution, ζ^k, is at hand. We form the composite direction $s = H_k^{-1} P_k\, \zeta^k$ and compute the step length α^k according to

$$\alpha^k = \arg \underset{\alpha}{\text{minimize}} \quad q(\delta + \alpha s)$$

and

$$\alpha^k = \min\{\alpha^k, .99\}.$$

We now choose the next iterate as

$$\delta^{k+1} = \delta^k + \alpha^k s.$$

The standard convergence tests are as follows: The algorithm is said to have converged on the relative objective function criterion if

$$\frac{|q(\delta^{k+1}) - q(\delta^k)|}{1 + |q(\delta^{k+1})|} \leq \epsilon_{obj} \tag{8.2.3}$$

where ϵ_{obj} is appropriately set, usually around 10^{-8}. The algorithm is said to have converged on the relative step criterion if

$$\max_i \left\{ \frac{|(\delta^{k+1})_i - (\delta^k)_i|}{1 + |(\delta^{k+1})_i|} \right\} \leq \epsilon_{step} \tag{8.2.4}$$

where ϵ_{step} is appropriately set, typically at 10^{-9}. Other convergence criteria are used in conjunction with the procedure to estimate the multipliers as described in the next section.

The key to the effectiveness of $O3D$ is, of course, the choice of the directions, p_i. The arguments and derivations of the directions used in $O3D$ are the same as those given in [27] and [35]. The ideas are based on considering the method of centers [58] and deriving the differential equation that describes the trajectory of the center points for a continuous version of the method of centers. The first direction is therefore the tangent to the trajectory at the given feasible point (also known as the “dual affine direction”) and is given by

$$p_1 = -\left(A^{\rm t} D^2 A + Q/\gamma\right)^{-1} (c + Q\delta) \tag{8.2.5}$$

where γ is interpreted as the residual on the objective function. For reasons given in [27] we take

$$\gamma = \left| \left(\frac{(c + Q\delta)^{\rm t}(c + Q\delta)}{(c + Q\delta)^{\rm t}(A^{\rm t} D e)} \right) \right| .$$

The second direction is the so-called "third-order correction" to p_1 given by

$$p_2 = \left(A^{\mathrm{t}} D^2 A + Q/\gamma\right)^{-1} \sum_{k=1}^{m} A_k^{\mathrm{t}} \left(\frac{\left[A_k p_1\right]^2}{r_k(\delta)^3} \right) \qquad (8.2.6)$$

where A_k is the k^{th} row of A.

The third direction is taken as one of the following. When the current iterate is judged to be "far" from the solution, the direction is an "update" to p_1 based on the first constraint encountered in the direction p_1. Let j be the index of this constraint. Then

$$p_u = \left(A^{\mathrm{t}} D^2 A + Q/\gamma\right)^{-1} A_j^{\mathrm{t}}. \qquad (8.2.7)$$

If the current iterate is judged to be "close" to the solution, then the third direction is based on the Newton recentering direction. This direction consists of a linear combination of the direction p_1 and

$$p_r = \left(A^{\mathrm{t}} D^2 A + Q/\gamma\right)^{-1} A^{\mathrm{t}} D e. \qquad (8.2.8)$$

To complete this description, we need to specify how we decide between the directions p_u and p_r. We judge the iterates to be close if both of the above convergence tests are satisfied with a tolerance of 5×10^{-3}.

Note that all of these directions can be computed by forming only one $n \times n$ matrix. Assume for the moment that this matrix is positive definite, which it will be in the convex case, i.e., when Q is positive definite. In this case p_1 is always a descent direction for the objective function and it is easy to show that the objective function is reduced in the composite direction.

The three-dimensional baby problem (8.2.1) is solved by a simple interior-point method. The point $\zeta = 0$ is always feasible by construction. Our procedure is to compute the direction corresponding to p_1 for the baby problem and to use it alone to determine the next iterate. As above, we use this direction to go 99% of the distance to the boundary or to the minimum of the objective function in that direction. We use the same convergence criteria as above and, as a practical matter, limit the number of iterations. Again, the matrix to be factored in forming p_1 may not be positive definite; we discuss this in the next section.

8.2.7 Implementation and Preliminary Results

All of $O3D$ is implemented using C++ using a style that is in conformance with that of Sundance and Trilinos/TSF. TSF, in particular, provides the ideal framework for the complex vectors and linear operators that are needed by $O3D$. Therefore, the $O3D$ algorithm itself can be written in terms of generic operators and vectors with no concern needed for the underlying complexity.

To be more specific, consider the quadratic program to have only inequality constraints (if there are equality constraints, they can be written as two sets of inequality constraints)

$$\begin{array}{rl} \underset{\delta}{\text{minimize}} & c^{\mathrm{t}}\delta + \frac{1}{2}\delta^{\mathrm{t}}Q\delta \\ \text{subject to:} & A\delta + b \;\leq\; 0. \end{array} \qquad (QP/I)$$

We take A and Q to be TSFLinearOperators and c and b to be TSFVectors, with the only restrictions being those necessary to maintain consistency of the operations, i.e., c and δ must be from the same TSFVectorSpace; b must be in the range space of A; and Q and A must have the same domain. These are all checked by TSF, thus ensuring at compile time that everything is consistent. Aside from these consistency requirements, the operators may have arbitrary complexity. The code to solve the problems can thus be greatly simplified, since the details of the linear algebra can be hidden in this abstraction. We illustrate some of this complexity by describing how a rather complicated problem can be assembled. To clarify the presentation, we us square brackets to indicate a "block" of a matrix. For example,

$$W = [[W_1 \; W_2]]$$

represents a block matrix with one block. This block, in turn, is a (1×2) block matrix. To construct an objective function, we need a linear operator, say Q_{orig}, and a vector, say c_{orig}. These are constructed by the user and used to create an O3DObjective. O3DObjective, in turn, creates the Q matrix and the c vector that $O3D$ will use by "wrapping" these in two levels of blocks as follows:

$$Q = [[Q_{orig}]]$$

and

$$c = [[c_{orig}]] \, .$$

Note that the two levels of blocking will be consistent with how the constraints need to be constructed to accomodate equality constraints, described next. $O3D$ allows a general collection of constraint sets to be used where each set consists of a linear operator, say A_{orig}, and a vector, say b_{orig}. These are passed to construct an instance of O3DConstraint, which wraps them in a block operator. Note that this allows equality constraints to be consistent with inequality constaints, i.e., inequality constraints are of the form

$$A_e = [A_{orig}]$$

and equality constraints are of the form

$$A_i = \begin{bmatrix} A_{orig} \\ -A_{orig} \end{bmatrix},$$

since equalities are written as two inequalities. The final A matrix is then a block with all of the sets, i.e.,

$$A = \begin{bmatrix} [A_1] \\ [A_2] \\ \vdots \\ [A_k] \end{bmatrix}$$

where there are k sets of constraints. This structure allows some constraints to be constructed by Sundance and others to be constructed by other means. This complexity, however, is never seen directly by $O3D$. Thus the code to set up a PDE constraint using Sundance is somewhat complex, but only needs to be done once.

As noted above, the main work in solving a QP using $O3D$ is the formation and solution of the linear systems of equations of the form

$$(A^{\mathrm{t}} D^2 A + Q/\gamma)\, p_i = r.$$

Note that forming this operator is not feasible in many large problems, since the fill-in may lead to a virtually dense matrix. This will certainly be the case in any problem where Q is a quasi-Newton approximation. Thus, to solve such systems, we must consider the use of "matrix-free" iterative methods. This essentially implies that we can form matrix-vector products, but we cannot get access to individual elements of the matrix. TSF allows the easy creation of the operator, while not actually forming it. Thus we can use conjugate gradient or other methods to solve these systems iteratively. Unfortunately, these systems are poorly conditioned, and become more so as the solution is neared, so that preconditioning becomes necessary almost immediately. We are continuing to persue strategies to precondition these systems effectively.

We have, however, been able to solve some example problems that show the effectiveness of the full-space (SAND) approach on these problems. One example problem is as follows:

Consider the rectangular reagon $\Omega = [0, \pi] \times [0, 1]$ and let $\Gamma = \{(x, 0) | 0 \le x \le \pi\}$. The differential equation is given by

$$\begin{aligned} \triangle u(x,y) &= 0 \ \text{ in } \Omega \\ u(x,y) &= 0 \ \text{ on } \partial\Omega \backslash \Gamma \\ u(x,0) &= \sum_{k=1}^{k=N} a_k \sin(kx) \end{aligned}$$

where we wish to choose the parameters a_k to force the solution to match a given target as closely as possible. The particular objective function we choose is

$$f = \frac{1}{2}\int_0^{\pi} (u(x, .5) - \hat{u})^2 dx$$

where

$$\hat{u} = x(\pi - x).$$

We were easily able to solve this problem using Sundance to create all of the operators and vectors on a 20×20 grid. Given the choice of finite element method and using $N = 5$ this resulted in a full-space problem of 1686 variables with 3362 constraints. $O3D$ required only 10 iterations to solve this problem.

8.3 Split

Split, an SQP method designed to work with $O3D$ is described in detail in [24]. Its features include:

- It uses an augmented Lagrangian type of merit function.
- Any number of iterations of $O3D$ on the quadratic programming subproblem yields a descent step on the merit function. Thus $O3D$ and Split are ideally suited for each other.
- A global convergence theory has been developed and published.
- An early implementation has been used to solve many interesting problems.
- It is flexible, allowing control over the use of perturbations and the rate of approaching feasibility. This is important in some applications where we have observed that better answers are obtained by delaying the approach to feasibility.
- Split does not require monotonic decrease in the merit function.

The main advantage of Split and $O3D$ is that they provide a complementary capability to rSQP++. Split/$O3D$ is a full-space method that handles problems with a large number of inequality constraints, but it also allows the use of in-between approaches where some, but not necessarily all of the state equations are optimization variables.

We have now a prototype implementation of Split in C++ using TSF, implemented with the same strategy as $O3D$.We plan to test this in conjunction with $O3D$ and Sundance soon.

8.4 Other Work

In this section we briefly discuss the implementation of $O3D$ in Java and our ideas on the use of proxy vectors. First, we wanted to test the use of Java for implementing a nontrivial numerical algorithm. We knew that there would probably be a significant performance penalty due to the way in which Java is implemented. To overcome this problem, we developed the idea of "proxy vectors" and "proxy operators." The concept is that local objects used by an optimizer on the front-end machine are proxies for remote objects living in a PDE code on a back-end machine. The front-end machine communicates with the back-end machine via sockets. The vector objects on the front-end contain only references to back-end vector objects; the vector objects on the back-end contain actual vector data, possibly distributed over many processors. Method invocation on the front-end results in a message sent to the back-end instructing it to execute the same method using the actual vectors. The same mechanism can be used for proxy operators. If a new vector or operator is needed, it is stored on the back-end with a proxy created on the front-end. If the result of an operation is a scalar, it is retruned to the front-end, but if the result is another vector, then the result stays on the back-end with only an message that the result was completed returned. For example, if the front-end machine requests the norm of a vector, that value will be returned, but if the request is to add two vectors, only a confirmation is returned. Thus all messages between the two machines are short. See [28] for a more complete discussion of this work.

We were able to create a working proxy-vector system, but the communication delays created an unacceptable penalty in the computations. We decided that this approach needed much more work to be successful, but that the results would probably not be worth the effort. Part of the effort would be to create an implementation of TSF in Java, and this does not seem to be a good idea. Thus, we think that there may be a future for proxy linear algebra, it will most likely be from our current C++ implementations.

8.5 Hierarchical Control

Optimal control problems constitute an interesting case of PDE-based optimization problems. There is a rich history of work in this area, beginning with the development of the calculus of variations and work in the control of ordinary differential equations. More recently, researchers have begun to investigate the control of PDEs (see the survey papers [49] and [50]). Instances of these types of problems abound in applications. Thus the development of efficient numerical methods for the solution of these problems has also been the subject of significant recent research.

In this paper we examine a particular instance of optimal control problems where multiple controls seek to force behavior close to multiple "targets" simultaneously. These problems belong to the class of problems called *multicriteria optimization*. There is no unique mathematical formulation of these types of problems; indeed different formulations can generate completely different "optima" solutions. In one formulation the problem is posed as the minimization of a weighted sum of the deviations from the targets with the weights corresponding to an established priority among the targets. (see [68]). Another formulation, sometimes referred to as *goal programming* insists that a set of preferred targets be satisfied to within certain tolerances and the others be reduced as much as possible within these constraints (see [60]). Both of these approaches involve the choice of a set of weights or tolerances for which there may be little theoretical guidance. In the approach that is employed in this paper, called *multilevel optimization,* the problem is modelled as as a set of nested optimization problems in which the solutions of the inner problems are determined using the variables in the outer problems as parameters (see [120]).

Motivated by specific engineering applications, such as those arising in optimal well placement in reservoir engineering, we investigate a means of formulating a class of optimal control problems in which the targets can be partitioned into categories of increasing relative importance. This approach, based on the work of von Stackelberg [121] in an economic context, requires that the deviations from the least important targets, called the "follower" targets, be decreased only after the deviations from the most important targets, called the "leader" targets, satisfy prescribed bounds. This type of optimal control problem has been termed *hierarchical control.* One way of formulating this type of problem is in terms of a nested optimization structure in which, in an "inner minimization", the follower targets are minimized subject to fixed values of certain of the control variables and then an "outer minimization" is performed over the remaining control variables to obtain optimal leader target satisfaction. The resulting accuracy on the follower targets is therefore determined by and is subordinate to the optimization over the leader targets. This type of *bilevel optimization* problem has been the object of a great deal of research (see [120] for an exhaustive bibliography) in finite-dimensional optimization and was given a theoretical grounding in the work of Lions [70] for PDE-constrained control problems where the state equations were a linear hyperbolic system. In this paper, we carry out the analysis for a specific parabolic system and obtain preliminary numerical results that we believe illustrate the promise of this approach.

8.6 Model Formulation

We are concerned with a class of optimal control problems in which there are multiple goals that are to be satisfied, i.e., a multicriteria control problem, and in which the underlying state variables are governed by the parabolic partial differential equation with mixed boundary conditions:

$$\begin{aligned} y_t - \mathbf{A}\, y &= f(x,t) + V(x,t), \quad (x,t) \in Q \\ y(x,0) &= b_0(x), \quad x \in \Omega, \\ y(x,t) &= b_1(x,t), \quad (x,t) \in \Gamma_1 \times (0,T), \\ \frac{dy}{d\eta}(x,t) &= b_2(x,t), \quad (x,t) \in \Gamma_2 \times (0,T), \end{aligned} \tag{8.6.9}$$

where Ω is a bounded open subset of $\mathcal{R}^2$, $T > 0$ is finite, $Q = \Omega \times (0,T)$ and $\Gamma_1 \cup \Gamma_2$ is the boundary of Ω. We assume that the functions in the model are well-behaved, i.e., $f(x,t) \in L^2(0,T;\Omega), b_0(x) \in L^2(\Omega)$, and $b_j(x,t) \in L^2(0,T;\Omega), j = 1,2$. Here $\mathbf{A}$ is a strongly elliptic operator and $V(x,t)$ represents the action of the controls on the system. In particular, we consider the case in which there are k *pointwise controls* $v_1(t), \ldots, v_k(t)$ located respectively at the points $a_1(t), \ldots, a_k(t)$ and that for a given choice of the controls,

$$V(x,t) = \sum_{j=1}^{k} v_j(t)\, \delta(x - a_j(t)).$$

Our goal is to formulate and solve an optimization problem that results in a selection of controls, including both time-dependent magnitudes and locations, that force the solution to the above system at time T to be "close" to a set of targets, $Y_1, \ldots, Y_k$, each $Y_j \in L^2(\Omega)$, while minimizing a cost functional $C(v,a)$. In addition, a set of restrictions on the location of the sites, $a_j(t), j = 1, \ldots, k$, are possible.

Obviously, it is generally impossible to force all of the targets to be satisfied to within some preassigned tolerance (in fact, it is not always possible to satisfy one target exactly). To formulate an optimization problem that can be solved, some priority must be established among the set of targets. A variety of methods have been proposed for carrying out this task. One such formulation is obtained by assigning a set of weights to the targets and minimizing the weighted sum of deviations from the targets. This problem can be expressed in the form

$$\begin{array}{ll} \text{minimize} & C(v,a) + \sum_{j=1}^{k} \frac{\gamma_j}{2} \int_\Omega (\, y(x,T) - Y_j(x)\,)^2\, dx \\ \text{subject to :} & y_t - \mathbf{A}\, y = f(x,t) + V(x,t), \quad (x,t) \in Q \\ & y(x,0) = b_0(x), \quad x \in \Omega \\ & y(x,t) = b_1(x,t), \quad (x,t) \in \Gamma_1 \times (0,T) \\ & \frac{dy}{d\eta}(x,t) = b_2(x,t), \quad (x,t) \in \Gamma_2 \times (0,T), \end{array} \qquad (SD)$$

where the γ_j are the respective weights associated with the different targets. A second approach is to assign acceptable deviations of the state variable from each of the targets and express these tolerances as constraints in the optimization problem. In this case the problem becomes

$$\begin{array}{ll}\text{minimize} & C(v,a)\\ \text{subject to :} & y_t - \mathbf{A}\,y = f(x,t) + V(x,t), \quad (x,t) \in Q\\ & y(x,0) = b_0(x), \quad x \in \Omega\\ & y(x,t) = b_1(x,t), \quad (x,t) \in \Gamma_1 \times (0,T)\\ & \frac{dy}{d\eta}(x,t) = b_2(x,t), \quad (x,t) \in \Gamma_2 \times (0,T)\\ & \int_\Omega (\,y(x,T) - Y_j(x)\,)^2\,dx \le \beta_j, \quad j = 1,\ldots,k.\end{array}$$

In these formulations, additional constraints on the controls could be included. Each of these formulations has certain drawbacks; in the first case a choice of weights is necessary without any *a priori* indication of how this choice will affect the solution; in the latter case it is difficult to specify the small tolerances in such a way as to avoid infeasibilities.

In this paper we follow the work of von Stackelberg (see [121]) and Lions (see [70]) and formulate the problem as a *hierarchical control* problem. This means that we prioritize the goals, i.e., specify a hierarchy of targets. The leading target is taken to be the one of the highest priority and the overriding task of the control problem is to have the state variable approximate this target as accurately as possible at $t = T$. Given this highest priority, the deviation from the target of next highest priority is minimized subject to the satisfaction of this primary goal. Then the deviation from the target of the third highest priority is minimized subject to the condition that the higher targets are satisfactorily approximated, and so on. This hierarchical structure requires a partition of the controls and control locations into corresponding hierarchies. In some problems there may be a natural correspondence but in other cases some flexibility in choosing the controls is available.

For this preliminary study we presume that there is a single leader target, denoted $Y_L(x)$, and a single target of lower priority called the follower target and denoted $Y_F(x)$. We also assume that there are two controls that we arbitrarily partition into leader and follower controls, $(v_L(t), a_L(t))$ and $(v_F(t), a_F(t))$, respectively. Additional follower targets and controls can be added without fundamentally affecting the nature of the model. The control problem we consider is the nested optimization problem, denoted by (OP):

$$\min_{a_L,a_F}\ C(v,a)$$
subject to :
$$(IP2)\quad \begin{cases} \min_{v_L}\ C(v,a) \\ \text{subject to :} \\ \qquad (IP1)\ \begin{cases} \min_{v_F,y}\ C(v,a) + \frac{\gamma_F}{2}\int_\Omega (\,y(x,T) - Y_F(x)\,)^2\,dx \\ \text{subject to :} \\ \quad y_t - \mathbf{A}\,y = f(x,t) + V(x,t), \quad (x,t) \in Q \\ \quad y(x,0) = b_0(x), x \in \Omega \\ \quad y(x,t) = b_1(x,t), (x,t) \in \Gamma_1 \times (0,T), \\ \quad \frac{dy}{d\eta}(x,t) = b_2(x,t), (x,t) \in \Gamma_2 \times (0,T), \end{cases} \\ \int_\Omega (\,y(x,T) - Y_L(x)\,)^2\,dx \le \beta \end{cases}$$

$$g(a) \le 0$$

where γ_F and β are fixed positive constants, $C(v,a)$ represents a general convex cost function depending on the controls, and the last inequalities involving $g : \mathcal{R}^4 \to \mathcal{R}^m$ represent constraints on the locations of the controls. These inequalities may be nonlinear and nonconvex; for example, $a_L(t)$ and $a_F(t)$ might be constrained to be a certain minimal distance apart.

This problem is interpreted in the following manner. The control variables a_L, a_F, and v_L are held fixed and the inner problem (IP1) is solved to determine the optimal choices for v_F and y, thus theoretically determining optimality functions $v_F^*(a_L, a_F, v_L)$ and $y^*(a_L, a_F, v_L)$. It is well known that the problem (IP1) has a unique solution for fixed a_L, a_F, and v_L. Next, these optimality functions are substituted into the objective function and the target constraint for the second inner problem, (IP2). Then this problem is solved with a_L and a_F held fixed determining another optimality function $v_L^*(a_L, a_F)$. Finally, the outer problem, (OP), now having the form

$$\begin{array}{l} \min_{a_L,a_F} C(v_L^*(a_L,a_F), v_F^*(a_L,a_F), v_L^*(a_L,a_F), a_L, a_F) \\ \text{subject to : } g(a_L,a_F) \le 0, \end{array}$$

is solved. Note that the cost function can be thought of as a regularization term in the inner problems, i.e., a term that is used to guarantee the existence of a solution. However, it also has a role as a general objective function to be minimized to the extent possible. In this model, we have optimized the variables (a_L, a_F) outside the optimization with respect to the other control variables and the state variables in order to facilitate the solution of the problem. As noted above, in applications the constraints on these variables can be nonlinear and nonconvex and if included in the inner optimization problems would make these problems difficult to solve and negate the advantages of the hierarchical structure.

The theory underlying the hierarchical control problem defined by the pair of problems (IP1) and (IP2) has been studied by Lions [70], albeit for a different underlying PDE and with *boundary controls*. Lions shows that a solution exists *for every positive* β although in general the target cannot be met exactly ($\beta = 0$); i.e., the problem is approximately controllable (see also Glowinski and Lions [49, 50]). These existence proofs for the solutions to the inner pair of optimization problems given by Lions are not constructive and hence provide no blueprint as to how to obtain numerical solutions. One natural approach is to use a variational method to obtain the optimality conditions for the innermost problem (IP1) and use these equations as constraints when solving (IP2). In the following we establish the optimality conditions for solving (IP1) and then discuss how to approach (IP2).

In order to simplify the notation and make the development more transparent we assume that

$$C(v,a) = \frac{1}{2}\int_0^T (v_L^2(t) + v_F^2(t))\,dt,$$

that $\mathbf{A}$ is the Laplacian operator Δ, and that the boundary conditions are of the Dirichlet type. Extensions to more general parabolic systems are straightforward in concept (but may require significantly more effort to obtain numerical solutions). Thus our PDE has the form

$$\begin{aligned} y_t - \mathbf{\Delta}\, y &= f(x,t) + v_L(t)\,\delta(x - a_L(t)) + v_F(t)\,\delta(x - a_F(t)), \quad (x,t) \in Q && (8.6.10)\\ y(x,0) &= b_0(x), \quad x \in \Omega, && (8.6.11)\\ y(x,t) &= b_1(x,t), \quad (x,t) \in \Gamma \times (0,T), && (8.6.12) \end{aligned}$$

where Γ is the boundary of Ω.

Proposition 1. Let a_L, a_F, and v_L be fixed. If v_F and y are optimal for (IP1) then there exists a dual function $p(x,t) \in L^2(0,T;\overline{\Omega})$ satisfying the PDE

$$\begin{aligned} p_t + \Delta p &= 0, \quad (x,t) \in Q && (8.6.13)\\ p(x,T) &= -\gamma_F\,(y(x,T) - Y_F(x)), \quad x \in \Omega, && (8.6.14)\\ p(x,t) &= 0, \quad (x,t) \in \Gamma \times (0,T), && (8.6.15) \end{aligned}$$

and v_F is given by

$$v_F(t) = p(a_F(t),t). \qquad (8.6.16)$$

Proof: If v_F and y are optimal for (IP1) then the variational equality for the objective function is

$$\int_0^T v_F(t)\, \hat{v_F}(t)\, dt + \gamma_F \int_\Omega (y(x,T) - y_F(x))\, \hat{z}(x,T)\, dx = 0 \tag{8.6.17}$$

for all admissible $\hat{v}_F \in L^2(0,T)$ and $\hat{z} \in L^2(0,T;\Omega)$. $\hat{v}_F$ and $\hat{z}$ are admissible if they satisfy

$$\begin{aligned} \hat{z}_t - \Delta\, \hat{z} &= \hat{v}_F(t)\, \delta(x - a_F(t)), \;\; (x,t) \in Q, && (8.6.18)\\ \hat{z}(x,0) &= 0, \;\; x \in \Omega. && (8.6.19)\\ \hat{z}(x,t) &= 0, \;\; (x,t) \in \Gamma \times (0,T). && (8.6.20) \end{aligned}$$

Multiplying (8.6.18) by $p(x,t)$, integrating over Q, and applying Green's theorem gives

$$\begin{aligned} \int_Q (p_t + \Delta p)\, \hat{z}(t)\, dx\, dt &+ \int_\Omega (p(x,T)\, \hat{z}(x,T) - p(x,0)\, \hat{z}(x,0))\, dx \\ &+ \int_{\Gamma\times(0,T)} (p(x,t)\, \frac{d\hat{z}}{d\eta}(x,t) - \frac{dp}{d\eta}(x,t)\, \hat{z}(x,t))\, dx\, dt \qquad (8.6.21)\\ &= \int_Q \hat{v}_F(t)\, \delta(x - a_F(t))\, p(x,t)\, dx\, dt \end{aligned}$$

where $\frac{d}{d\eta}$ represents the normal derivative. Using (8.6.13)–(8.6.15), and (8.6.18)–(8.6.20) this equation becomes

$$-\gamma_F \int_\Omega (y(x,T) - Y_F(x))\, \hat{z}(x,T)\, dx = \int_0^T \hat{v}_F(t)\, p(a_F(t),t)\, dt. \tag{8.6.22}$$

Equation (8.6.16) follows immediately from this last equation and the Euler equation, (8.6.17).

Using these necessary conditions, the second inner problem (IP2) can now be written

$\min_{v_L,y,p} \int_0^T (v_L^2(t) + p(a_F(t),t)^2)\, dt$
subject to :

$$\begin{aligned} y_t - \boldsymbol{\Delta}\, y &= f(x,t) + v_L(t)\, \delta(x - a_L(t)) \\ &+ p(x,t))\, \delta(x - a_F(t)), \quad (x,t) \in Q \\ y(x,0) &= b_0(x), \quad x \in \Omega, \\ y(x,t) &= b_1(x,t), \quad (x,t) \in \Gamma \times (0,T), \\ p_t + \Delta p &= 0, \quad (x,t) \in Q, \\ p(x,T) &= -\gamma_F\, (y(x,T) - Y_F(x)), \quad x \in \Omega, \\ p(x,t) &= 0, \quad (x,t) \in \Gamma \times (0,T), \\ \int_\Omega (\, y(x,T) - Y_L(x)\,)^2\, dx &\le \beta, \end{aligned}$$

with a_L and a_F fixed.

At this stage there are several possible approaches. One approach would be to incorporate the control variables $a(t)$ directly into the problem (so in effect (IP2) becomes (OP)) and solve the resulting problem. However, this approach severely restricts the numerical methods that we can apply since the state variable occurs in a nonlinear inequality constraint. For example, a reduced variable approach could not be employed. Another approach would be to obtain the optimality conditions for this problem (as was done for (IP1)) and then use these conditions in the formulation of the outer problem. If we take this approach then we are forced to include complementary slackness conditions as part of the necessary conditions which is an added nonlinear difficulty. Both of these methods also suffer from the fact that an *a priori* choice of β is required.

As a result of these complications, we have chosen, following Glowinski and Lions (see [49]) to include the leader target goal as a penalty term in the objective function. That is, we reformulate (IP2) as

$$\begin{array}{l}
\min_{v_L,y,p} \int_0^T (v_L^2(t) + p(a_F(t),t)^2\, dt + \frac{\gamma_L}{2}\int_\Omega (\, y(x,T) - Y_L(x)\,)^2\, dx \\
\text{subject to :} \\
\begin{array}{rcl}
y_t - \boldsymbol{\Delta}\, y &=& f(x,t) + v_L(t)\,\delta(x - a_L(t)) + p(x,t))\,\delta(x - a_F(t)), \quad (x,t) \in Q \\
y(x,0) &=& b_0(x), \quad x \in \Omega, \\
y(x,t) &=& b_1(x,t), \quad (x,t) \in \Gamma \times (0,T), \\
p_t + \Delta p &=& 0, \quad (x,t) \in Q, \\
p(x,T) &=& -a_F\,(y(x,T) - Y_F(x)), \quad x \in \Omega, \\
p(x,t) &=& 0, \quad (x,t) \in \Gamma \times (0,T),
\end{array}
\end{array} \qquad \text{(IP3)}$$

where γ_L is a specified constant. By choosing γ_L sufficiently large we can, in theory, force the deviation from the leader target to be less than β although such a solution will not, in general, be the solution to the original problem (IP2).

We now derive the optimality conditions for this reformulated problem.

Proposition 2. Let a_L and a_F be fixed. If v_L, y, and p are optimal for the problem (IP3) given above, then there exist functions $P(x,t)$ and $Y(x,t)$ in $L^2(0,T,\Omega)$ satisfying

$$\begin{aligned}
Y_t + \Delta Y &= 0, \quad (x,t) \in Q && (8.6.23)\\
Y(x,T) &= -\gamma_F\, P(x,T) - \gamma_L\, (y(x,T) - Y_L(x)), \quad x \in \Omega, && (8.6.24)\\
Y(x,t) &= 0, \quad (x,t) \in \Gamma \times (0,T), && (8.6.25)\\
P_t - \Delta P &= -\delta(x - a_F)\, (p(x,t) - Y(x,t)), \quad (x,t) \in Q && (8.6.26)\\
P(x,0) &= 0, \quad x \in \Omega, && (8.6.27)\\
P(x,t) &= 0, \quad (x,t) \in \Gamma \times (0,T), && (8.6.28)
\end{aligned}$$

and v_L is given by

$$v_L(t) = Y(a_L(t),t), \quad t \in (0,T). \qquad (8.6.29)$$

Proof: If v_L, y, and p are optimal for (IP3) then the variational equation

$$\int_0^T (v_L(t)\, \hat{v}_L(t) + p(a_F(t),t)\, \hat{p}(a_F(t),t))\, dt + \gamma_F \int_\Omega (y(x,T) - Y_L(x))\, \hat{z}(x,T)\, dx = 0 \quad (8.6.30)$$

must be satisfied for every admissible ($\hat{v}_L, \hat{z}, \hat{p}$), i.e., for every ($\hat{v}_L, \hat{z}, \hat{p}$) satisfying

$$\begin{aligned}
\hat{z}_t - \Delta \hat{z} &= \hat{v}_L\, \delta(x - a_L(t)) + \hat{p}(x,t)\, \delta(x - a_F(t)), \ (x,t) \in Q, && (8.6.31)\\
\hat{z}(x,0) &= 0, \ x \in \Omega, && (8.6.32)\\
\hat{z}(x,t) &= 0, \ (x,t) \in \Gamma \times (0,T), && (8.6.33)\\
\hat{p}_t + \Delta \hat{p} &= 0, \ (x,t) \in Q, && (8.6.34)\\
\hat{p}(x,T) &= -\gamma_F\, \hat{z}(x,T), \ x \in \Omega, && (8.6.35)\\
\hat{p}(x,t) &= 0, \quad (x,t) \in \Gamma \times (0,T). && (8.6.36)
\end{aligned}$$

Now multiplying (8.6.31) by $Y(x,t)$ and (8.6.34) by $P(x,t)$, integrating over Q, and again applying Green's theorem, we obtain

$$\begin{aligned}
\int_Q (Y_t + \Delta\, Y)\, \hat{z}(x,t)\, dx\, dt \ &+ \ \int_\Omega (Y(x,T)\, \hat{z}(x,T) - Y(x,0)\, \hat{z}(x,0))\, dx \\
&+ \ \int_{\Gamma \times (0,T)} (Y(x,t)\, \frac{d\hat{z}}{d\eta}(x,t) - \frac{dY}{d\eta}(x,t)\, \hat{z}(x,t))\, dx\, dt \qquad (8.6.37)\\
&= \ \int_Q (\hat{v}_L(x,t)\, \delta(x - a_L(t)) + \hat{p}(a_F(t),t)\, \delta(x - a_F(t))\, Y(x,t)\, dx\, dt
\end{aligned}$$

and

$$\begin{aligned}\int_Q (P_t - \Delta P)\,\hat{p}(x,t)\,dx\,dt \quad &+ \quad \int_\Omega (P(x,T)\,\hat{p}(x,T) - P(x,0)\,\hat{p}(x,0))\,dx \\ &+ \quad \int_{\Gamma\times(0,T)} (P(x,t)\,\frac{d\hat{p}}{d\eta}(x,t) - \frac{dP}{d\eta}(x,t)\,\hat{p}(x,t))\,dx\,dt \\ = 0 \end{aligned} \tag{8.6.38}$$

Using the various PDE's and boundary conditions for the functions in (8.6.37) and (8.6.38) we arrive at

$$\begin{aligned}-\gamma_F \int_\Omega P(x,T)\,\hat{z}(x,T)\,dx \quad &- \quad \gamma_L \int_\Omega (y(x,T) - Y_L(x))\,\hat{z}(x,T)\,dx \\ &= \quad \int_0^T (\hat{v}_L(t)\,Y(a_L(t),t) + \hat{p}(a_F(t),t)\,Y(a_F(t),t))\,dt \end{aligned} \tag{8.6.39}$$

and

$$-\int_0^T (p(a_F(t),t) - Y(a_F(t),t))\,\hat{p}(a_F(t),t)\,dt + \int_\Omega P(x,T)\,\hat{p}(x,T)\,dx = 0. \tag{8.6.40}$$

Using (8.6.35) and rerranging the terms in (8.6.40) yields

$$\gamma_F \int_\Omega P(x,T)\,\hat{z}(x,T)\,dx = \int_0^T \hat{p}(a_F(t),t)\,(p(a_F(t),t) - Y(a_F(t),t))\,dt. \tag{8.6.41}$$

Substituting this last equation into (8.6.39) yields the variational equation (8.6.30).

With this derivation the formulated optimization problem (OP) becomes

$$\begin{array}{l} \min_{a_L,a_F}\ \frac{1}{2}\int_0^T (p(a_F(t),t)^2 + Y(a_L(t),t)^2)\,dt + \frac{\gamma_L}{2}\int_\Omega (y(x,T) - Y_L(x))^2\,dx \\ \text{subject to :} \\ \qquad\qquad \text{equations } (8.6.10) - (8.6.15) \\ \qquad\qquad \text{equations } (8.6.23) - (8.6.29) \\ \qquad\qquad g(a) \le 0. \end{array} \tag{8.6.42}$$

Several additional comments need to be made concerning this formulation. First, the relative sizes of the constants γ_L and γ_F affect how accurately the different targets can be approximated. In order to approximate the leader target as accurately as possible, γ_L must be made large. However, the effect of increasing its size is influenced by the size of γ_F. Thus, as in the first formulation of this section, (SD), with a single objective function incorporating both targets, the magnitudes of γ_F

and γ_L required to achieve the desired target deviations must be determined by experimentation. Our preliminary numerical studies have suggested that if both targets are in the objective function and both constants are large, then there can be difficulties in achieving convergence to the optimal solution. One of the goals of the numerical study described in the next section was to determine how the effect of differing scales of magnitude on the choice of these leader and follower constants affected the optimal solutions in the hierarchical formulation. Secondly, it should be emphasized that in order to provide useful results the optimal control generated by the model must be implementable, e.g., wildly oscillating optimal controls are undesireable. Again our studies to date have indicated that the controls achieved in the hierarchical formulation are better-behaved than those from (SD) for large values of the parameters. Both of these conjectures need further testing and, if possible, theoretical grounding.

Finally, it is clear that this formulation of the problem is fundamentally different from other models. As is well-documented in the finite-dimensional cases of bilevel programming, an optimal solution to a bilevel optimization problem need not be a *Pareto optimal* solution in the sense of multiobjective optimization (see [120]) and there is no reason to assume that this is not the case here. Also, the inclusion of the follower control sites a_F as part of the outer optimization, rather than the inner optimization problem, may seem inconsistent. In formulating the problem in this manner, we were again motivated by an effort to make the problem tractable; complicated (and possibly nonconvex) inequality constraints in the control locations would seriously degrade the ability to express concisely the necessary conditions for the inner problem. All of these points speak to the difficulty in formulating state equations and in solving large scale multicriteria optimization problems. The results presented here represent an initial effort in this direction.

We conclude this section by observing that hierarchical control might profitably be used to formulate a multitude of important scientific applications. For example, in the area of oil reservoir simulation one can formulate optimal well placement problems as hierarchical control problems where desired well productions might form mandatory (or leader) targets while revenue or efficiency based goals are a secondary (follower) targets. Problems in optimal airfoil design can be viewed in a similar way with structural constraints being posed as leader objectives and vorticity minimizing goals being follower targets. Remote manipulator systems, like those employed by space-craft, are required to solve optimal control problems rapidly. In some instances, these systems must accomplish a goal while maintaining prescribed distances from other pieces of machinery. One could formulate a class of hierarchical control in which leader targets include primary objectives and follower targets maintain minimal separation from sensitive machinery whenever possible.

8.7 Numerical Results

In this section we report on some numerical experiments we have run to test some of the issues raised by the formulation of the hierarchical control problem given in the preceding section (also see [25]). The problem addressed is that of the preceding section with the domain Ω taken to be the unit square with the boundary conditions chosen to be zero. Moreover, we have assumed that the control sites are not functions of t but constant. We don't believe that these simplifications prohibit us from making preliminary assessments about the prospects for this type of formulation. In any case, we intend to continue experimention.

We had several goals for these preliminary numerical experiments. First we wanted to determine the possibility of efficiently solving the problem in its hierarchical formulation. Second, we wanted to determine how sensitive the solutions were to different choices of the constants γ_L and γ_F and to compare these results with those obtained by solving the problem with a single objective function containing a weighted sum of the target discrepancies. Finally, we wanted to ascertain if we could solve a problem with nonconvex constraints on the control locations.

We begin by describing the time discretization. Let N_T be the number of time steps desired, so that $\Delta t = \frac{T}{N_T}$. We will denote the estimate of y at the nth time step by y^n where $n = 1 \ldots N_T$. If N_X denotes the number of spatial steps in the x_1 and in the x_2 directions, the spatial step is denoted by h with $h = \frac{1}{N_X}$. The discrete approximation to y is

$$y(n\Delta t, ih, jh) \approx y^n_{i,j}.$$

We follow the *two-step implicit scheme* for parabolic problems as outlined in Glowinski [48]. Accordingly, we define

$$\frac{\partial y}{\partial t}((n+1)\triangle t) \approx \frac{1}{2\Delta t}\left(3y^{n+1}_{i,j} - 4y^n_{i,j} + y^{n-1}_{i,j}\right).$$

Experience with this time discretization has led us to use it on stiff problems when we need to integrate to large values of T. In such cases, the fact that it assures unconditional stability and produces an accuracy to second order in time amply justifies the storage costs.

At each time step we must solve an elliptic problem to obtain $y^{n+1}_{i,j}$. The domain is so simple that we use the very common finite-element triangulation of Ω consisting of bisected squares. The space of polynomials of degree ≤ 1 is used to form a finite dimensional approximation to $L^2(\Omega)$ and $H^1(\Omega)$. More sophisticated schemes are certainly available for both linear and nonlinear parabolic equations. However, for testing optimization formulations of the control problem here, this simple numerical scheme is both adequate and appropriate. For specific applications, more specialized or hybrid discretizations may be called for (see for example [66]).

Two target states, $y_L(x)$ and $y_F(x)$, are used to test the performance of the formulation of the control problem from section 3. While a myriad of test shapes are possible, we choose one specific pair of test shapes that illustrates behavior seen in most of our numerical experiments. The leader target shape is a smooth function with a peak of approximately 1.3 at the point $x_1 = 1/3$, $x_2 = 1/2$ and the follower is a pyramid with a peak of unity at the point $x_1 = \frac{9}{10}, x_2 = \frac{1}{2}$ (see Figures 8.1 and 8.2). Specifically, the target functions are

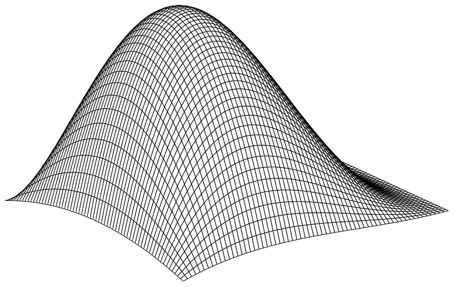


Figure 8.1. The leader target

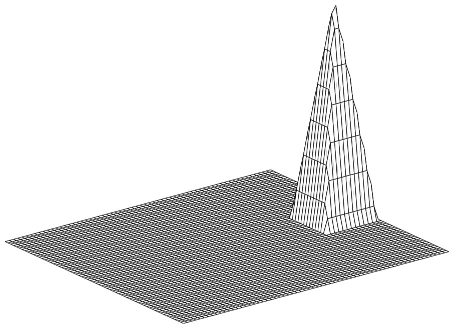


Figure 8.2. The follower target

$$\begin{aligned} y_L(x) &= 35x_1x_2(1-x_2)(1-x_1)^2, \\ y_F(x) &= 2\min\{5x_1-4, 3-5x_2, 5-5x_1, 5x_2-2\}. \end{aligned} \tag{8.7.43}$$

These test problems are similar to those used to study hierarchical control with stationary controls ([13]).

As constraints on the control locations, we required that the parameters a_L and a_F be constrained to be contained inside disjoint balls. The leader location, a_L is constrained to lie within the circle centered at $(\frac{1}{4}, \frac{1}{2})$ where the follower location is constrained to lie within the circle centered at $(\frac{3}{4}, \frac{1}{2})$. Both constraints have radius 0.15 so that the two circles do not intersect (see Figure 8.7).

The optimization problem that arose from our formulation was solved using a sequential quadratic programming (SQP) algorithm. The specifics of the algorithm are contained in [24] and a theoretical analysis that can be found in [23].

The numerical results are summarized in Table 8.1 together with Figures 8.4–8.9. The first two columns of of Table 8.1 give the problem size. The values of γ_L and γ_F are given in the third

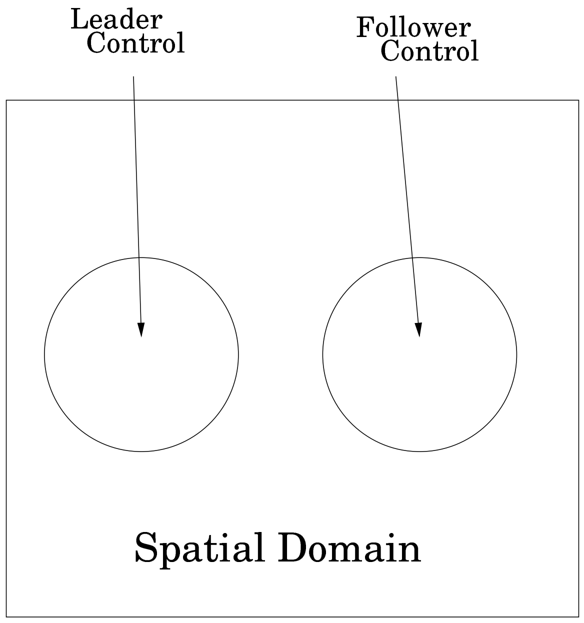


Figure 8.3. Geometric constraints separating the controls

column. The next two columns give the relative discrepancy between state variables and targets in the L^2 norm. The final two columns of the table report on the norm of the controls.

N_X	N_T	(γ_F, γ_L)	$\|y_F - y\|_{L^2}/\|y_F\|_{L^2}$	$\|y_L - y\|_{L^2}/\|y_L\|_{L^2}$	$\|v_F\|$	$\|v_L\|$
64	32	(1.e+3,1.e+3)	2.395302	0.4873116	25.774	3.4065
64	32	(1.e+6,1.e+3)	2.181885	0.4978943	151.05	28.420
64	32	(1.e+3,1.e+6)	2.415231	0.2635630	34.279	448.49
128	32	(1.e+3,1.e+3)	2.371236	0.4732074	28.195	3.5591
128	32	(1.e+6,1.e+3)	2.200413	0.5009123	155.89	29.093
128	32	(1.e+3,1.e+6)	2.418927	0.2701232	34.861	449.12

Table 8.1. Numerical Performance Summary

Our problem formulation worked well with our numerical optimization algorithm. In numerical results not presented here we were able to solve problems with values of γ as large as $1.e16$ and values approaching machine precision. Here we concentrate on the results for more reasonable values of γ. In Figures 8.4, 8.6 and 8.8 the dotted and dashed profiles respectively denote leader and follower target profiles along the line $x_2 = \frac{1}{2}$. The solid lines are the state variable y at terminal time $T = 1$ also along the line $x_2 = \frac{1}{2}$. Clearly both the leader and follower targets

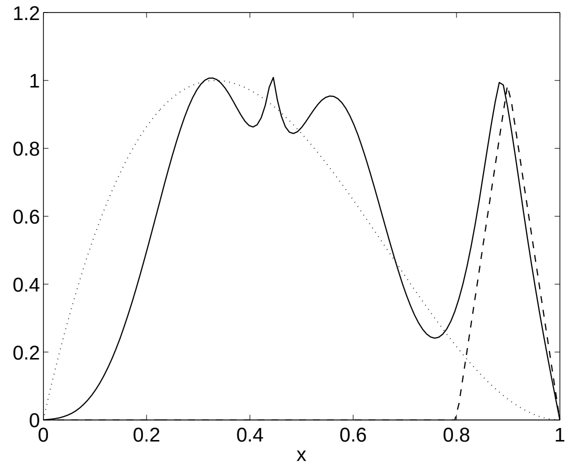


Figure 8.4. The state variables restricted to the line $x_2 = \frac{1}{2}$ with $\gamma_L = \gamma_F = 1.e+3$

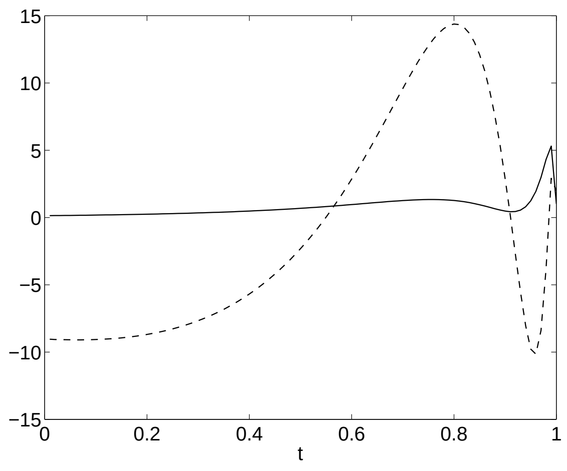


Figure 8.5. The control variables with $\gamma_L = \gamma_F = 1.e+3$

were approximately attained. In Figures 8.5, 8.7 and 8.9 the leader and follower controls, $v_L(t)$ and $v_F(t)$ are shown for $t \in (0,1]$, by solid and dashed lines respectively. In Figure 8.5 the total variations in the two controls are comparable while in Figure 8.9 the value of γ_F is large enough, when compared with γ_L, that the effect of the leader control is greatly diminished. In fact the follower control oscillated so violently that it eclipsed the behavior of the leader control. Finally, it is worth noting that for the numerical examples presented here, the optimal location of both controls was inside the constraint circles.

Our numerical experience illustrated that the difficulty of the SQP algorithm in solving the hierarchical problem tested here increased with the values of the penalty parameters γ_L and γ_F. This fact is not surprising in light of the fact that similar behavior has been observed for the case of hierarchical control of Burgers' Equation ([65]).

8.8 Future Research

The freedom to specify multiple targets is extremely important for many practical problems. The high cost of solving multicriteria optimization problems suggests that there may be instances where hierarchical control problem formulations could yield a computational advantage in an affordable way. We plan to investigate the use of this formulation technique to attack more complicated physical phenomena, including those modeled by nonlinear equations. We anticipate the ideas

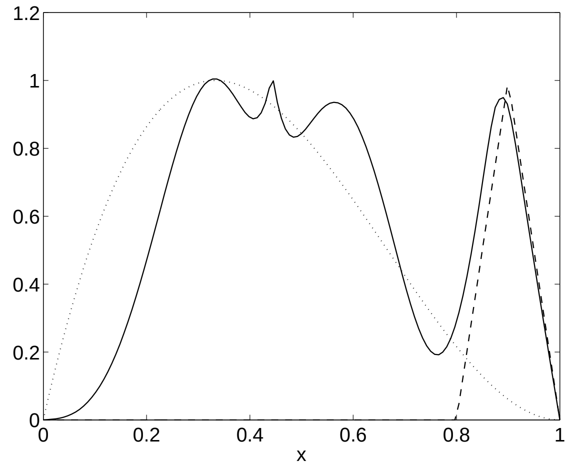


Figure 8.6. The state variables restricted to the line $x_2 = \frac{1}{2}$ with $\gamma_L = 1.e + 6$ and $\gamma_F = 1.e + 3$

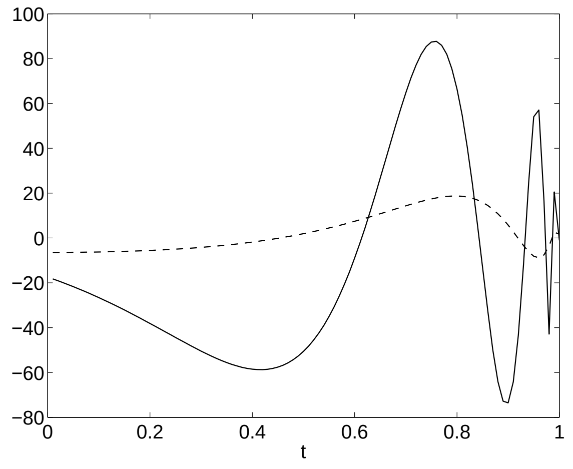


Figure 8.7. The control variables with $\gamma_L = 1.e + 6$ and $\gamma_F = 1.e + 3$

will be fruitful when formulating problems of optimal well placement, contaminant transport, and bioremediation among others.

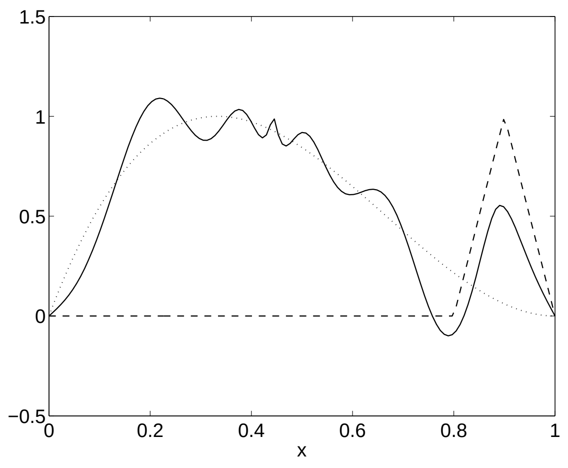


Figure 8.8. The state variables restricted to the line $x_2 = \frac{1}{2}$ with $\gamma_L = 1.e + 3$ and $\gamma_F = 1.e + 6$

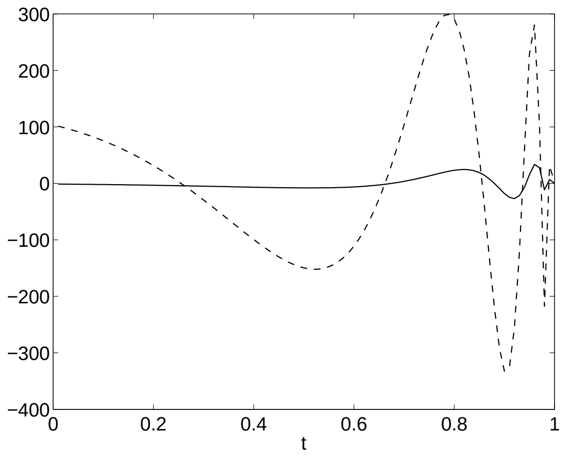


Figure 8.9. The control variables with $\gamma_L = 1.e + 3$ and $\gamma_F = 1.e + 6$

Bibliography

[1] Space station external contamination control requirements. Technical report, NASA/JSC 30426, 1986.

[2] E.M. Cliff A. Shenoy, M. Heinkenschloss. Airfoil design by an all-at-once method. Technical Report 97-15, CAAM Rice University, Department of Computational and Applied Mathematics, 1997.

[3] V. Akcelik, G. Biros, and O. Ghattas. Parallel multiscale gauss-newton-krylov methods for inverse wave propagation. In *Proceedings of the IEEE/ACM SC2002 Conference, Baltimore*. IEEE/ACM, 2002.

[4] F. Alexander and G. Garcia. The direct simulation monte carlo method. *Computers in Physics*, (11):588, 1997.

[5] W. K. Anderson and D. L. Bonhaus. Aerodynamic design on unstructured grids for turbulent flows. Technical Report TM 112867, Langley Research Center, Hampton, Virginia, 1997.

[6] W. K. Anderson and V. Venkatakrishnan. Aerodynamic de-sign optimization on unstructured grids with a continuous adjoint formulation. In *97-0643*. AIAA, 1997.

[7] E. Arian and Salas M.D. Admitting the inadmissible: Adjoint formulation for incomplete cost functionals in aerodynamic optimization. Technical Report 97-69, ICASE, NASA Langley, 1997.

[8] E. Arian and V. N. Vatsa. A preconditioning method for shape optimization governed by the euler equations. Technical Report 98-14, ICASE, NASA Langley, 1998.

[9] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc 2.0. http://www.mcs.anl.gov/petsc.

[10] R. A. Bartlett. *Object Oriented Approaches to Large Scale NonLinear Programming For Process Systems Engineering*. Ph.D Thesis, Chemical Engineering Department, Carnegi Mellon University, Pittsburgh, 2001.

[11] R. A. Bartlett, L. T. Biegler, J. Backstrom, and V. Gopal. Quadratic programming algorithms for larg-scale model predictive controls. *J. Process Control*, 12:775–795, 2002.

[12] A. Battermann and M. Heinkenschloss. Preconditioners for karush-kuhn-tucker matrices arising in the optimal control of distributed systems. In *Optimal Control of Partial Differential Equations, Vorau 1997, Birkhuser Verlag, Basel, Boston, Berlin*, pages 15–32, 1998.

[13] M. Berggren. *Control and Simulation of Advection–Diffusion Problems*. Ph.D Thesis, Computational and Applied Mathematics, Rice University, Houston, Tx., 1995.

[14] M. Berggren. Nunmerical solution of a flow-control problem: Vorticity reduction by dynamical boundary action. *SIAM Jounrnal Scientific Computing*, (Vol 19, No. 3 pp 829-860), 1998.

[15] L. T. Biegler, J. Nocedal, and C. Schmid. A reduced hessian method for large-scale constrained optimization. *SIAM J. Opt.*, 5:314, 1995.

[16] L.T. Biegler, A. Cervantes, and A. Wächter. Advances in simultaineous strategies for dynamic optimization. Technical Report CAPD Technical Report B-01-01, Department of Chemical Engineering, Carniege Mellon University, 2001.

[17] L.T. Biegler, C. Schmidt, and D. Ternet. A multiplier-free, reduced hessian method for process optimization. 1996.

[18] G.A. Bird. *Molecular Dynamics and the Direct Simulation of Gas Flows*. Clarendon, Oxford, 1994.

[19] G. Biros. Parallel newton-krylov algorithms for pde-constrained optimization. *the SCXY Conference Series*, November 1999.

[20] G. Biros and O. Ghattas. Parallel preconditioners for KKT systems arising in optimal control of viscous incompressible flows. In *Proceedings of Parallel CFD '99, Williamsburg, VA, May 23–26, 1999*, Amsterdam, London, New-York, 1999. North Holland. to appear, http://www.cs.cmu.edu/∼oghattas/.

[21] G. Biros and O. Ghattas. Parallel lagrange-newton-krylov-schur methods for pde-constrained optimization. part i: The krylov-schur solver. Technical report, Laboratory for Mechanics, Algorithms, and Computing, Carnegie Mellon University, 2000.

[22] G. Biros and O. Ghattas. Parallel lagrange-newton-krylov-schur methods for pde-constrained optimization. part ii: The lagrange-newton solver, and its application to optimal

control of steady viscous flows. Technical report, Laboratory for Mechanics, Algorithms, and Computing, Carnegie Mellon University, 2000.

[23] P. T. Boggs, A. J. Kearsley, and J. W. Tolle. A global convergence analysis of an algorithm for large scale nonlinearly constrained optimization problem. *SIAM J. Optim.*, 9(4):833–862, 1999.

[24] P. T. Boggs, A. J. Kearsley, and J. W. Tolle. A practical algorithm for general large scale nonlinear optimization problems. *SIAM J. Optim.*, 9(3):755–778, 1999.

[25] P. T. Boggs, A. J. Kearsley, and J. W. Tolle. Hierarchical control of a linear diffusion equation. *in press*, 2002.

[26] P. T. Boggs and J. Tolle. Successive quadratic programming. *Acta Numerica*, 1996.

[27] Paul T. Boggs, Paul D. Domich, and Janet E. Rogers. An interior-point method for general large scale quadratic programming problems. *Annals of Operations Research*, 62:419–437, 1996.

[28] Paul T. Boggs and Kevin R. Long. A software system for pde-constrained optimization problems. In G. DiPillo and A. Murli, editors, *High Performance Algorithms and Software for Nonlinear Optimization*, page in Press, Dordrecht, 2002. Kluwer Academic Publishers.

[29] Brooke et al. *GAMS Release 2.25, Version 92 Language Guide*. GAMS Development Corperation, Washington, DC, 1997.

[30] E. A. Burroughs, L. A. Romero, R. B. Lehoucq, and A. G. Salinger. Large scale eigenvalue calculations for computing the stability of buoyancy driven flows. *Sandia Technical Report*, SAND2001-0113, 2001.

[31] R. H. Byrd, J. Nocedal, and R.B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Math. Prog.*, 63:129–156, 1994.

[32] A. Carle, M. Fagan, and L. L. Green. Preliminary results from the application of automated adjoint code generation to cfl3d. In *AIAA-98-4807*. AIAA, 1998.

[33] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[34] J. E. Dennis, M. Heinkenschloss, and L. N. Vicente. Trust-region interior-point sqp algorithms for a class of nonlinear programming problems. *SIAM Journal on Control and Optimization*, (Volume 36 Number 5), 1998.

[35] Paul D. Domich, Paul T. Boggs, Janet E. Rogers, and Christoph Witzgall. Optimizing over three-dimensional subspaces in an interior-point method for linear programming. *Linear Algebra and its Applications*, 152:315–342, July 1991.

[36] D. A. Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21:1129–1148, 1985.

[37] M.S. Eldred, A.A. Giunta, B.G. van Bloemen Waanders, S.F. Wojtkiewicz, W.E. Hart, and M.P. Alleva. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis. version 3.0 users manual. Technical report SAND2001-3796, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, April 2001.

[38] M.S. Eldred, A.A. Giunta, B.G. van Bloemen Waanders, S.F. Wojtkiewicz, W.E. Hart, and M.P. Alleva. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis. version 3.0 users manual. Technical report SAND2001-3515, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, April 2001.

[39] M.S. Eldred, A.A. Giunta, B.G. van Bloemen Waanders, S.F. Wojtkiewicz, W.E. Hart, and M.P. Alleva. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis. version 3.0 users manual. Technical report SAND2001-3515, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, April 2001.

[40] M.S. Eldred, W.E. Hart, W.J. Bohnhoff, V.J. Romero, S.A. Hutchinson, and A.G. Salinger. Utilizing object-oriented design to build advanced optimization strategies with generic implementation. *Proceedings of the 6th AIAA/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, AIAA-96-4164-CP, Bellevue, WA*, pages 1568–1582, 1996.

[41] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Porogramming*. Scientific Press, 1993.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] O. Ghattas and J. Bark. Optimal control of two- and three-dimensional navier-stokes flows. *Journal of Computational Physics*, (136):231, 1997.

[44] O. Ghattas and C. Orozco. A parallel reduced hessian sqp method for shape optimization. In *Multidisciplinary Design Optimization: State of the Art*. SIAM, 1997.

[45] P. Gill, W. Murry, M. Saunders, and M. Wright. *User's Guide for SOL/QPSOL: A Fortran Package for Quadratic Programming*. Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1983.

[46] P.E. Gill, L.O. Jay, M.W. Leonard, and L.R.and Petzold V. Sharma. An sqp method for the optimal control of large scale dynamical systems. *Jounal Computational Applied Mathematics*, (120 197-213), 2000.

[47] Gill, P., W. Murry and M. Saunders. *User's Guide for QPOPT 1.0: A Fortran Package for Quadratic Programming*. Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1995.

[48] R. Glowinski. *Numerical Methods for Nonlinear Variational Problems*. Springer-Verlag, New York, 1984.

[49] R. Glowinski and J.L. Lions. Exact and approximate controllability for distributed parameter systems i. *Acta Numerica*, pages 269–378, 1994.

[50] R. Glowinski and J.L. Lions. Exact and approximate controllability for distributed parameter systems ii. *Acta Numerica*, pages 159–333, 1995.

[51] M. S. Gockenbach and W. W. Symes. The hilbert class library. http://www.trip.caam.rice.edu/txt/hcldoc/html/index.html.

[52] R.T. Haftka. Simultaneous analysis and design. *AIAA Journal*, 1985.

[53] B. He, O. Ghattas, and J.F. Antaki. Computational strategies for shape optimization of time dependent navier stokes flow. Technical Report CMU-CML-97-102, Carnegie Mellon University, 1997.

[54] M. Heinkenschloss. Time domain decomposition iterative methods for the solution of distributed linear quadratic optimal control problems. Technical Report TR00-31, Rice University, 2000.

[55] M. Heroux. The trilinos project. http://www.cs.sandia.gov/Trilinos/.

[56] M. Heroux, R. Lehoucq, K. Long, and A. Williams. Trilinos solver framework. http://www.cs.sandia.gov/Trilinos/doc/tsf/doc/html/index.html.

[57] P. D. Hough and T. G. Kolda. Asynchronous parallel pattern search for nonlinear optimization. Technical report, Sandia National Laboratories, 2000.

[58] P. Huard. Resolution of mathematical programming with nonlinear constraints by the method of centers. In J. Abadie, editor, *Nonlinear Programming*, pages 209–219, Amsterdam, 1967. North-Holland.

[59] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover, 2000.

[60] J. P. Ignizio. *Goal Programming and Extensions*. Lexington Books, Lexington, Massachusetts, 1976.

[61] A. Iollo, G.Kuruvilla, and S. Ta'asan. Pseudo-time method for optimal shape design using the euler equations. Technical Report 95-59, ICASE, 1995.

[62] A. Jameson. Aerodynamic design via control theory. *journal of Scientific Computing*, (3):233, 1988.

[63] Y. Jinyun. Symmetric gaussian quadrature formulae for tetrahendronal regions. *Computer Methods in Applied Mechanics and Engineering*, 43:349–353, 1984.

[64] R.D. Joslin, M.D. Gunzburger, R.A. Nicolaides, G. Erlebacher, and M.Y. Hussaini. A methodology for the automated optimal control of flows including transitional flows. Technical report, ICASE NASA Langley, 1995.

[65] A. J. Kearsley. The use of optimization techniques in the solution of partial differential equations from science and engineering. Technical Report & P.h.D. Thesis, Rice University, Department of Computational & Applied Mathematics, 1996.

[66] A. J. Kearsley, L. C. Cowsar, R. Glowinski, M. F. Wheeler, and I. Yotov. An optimization approach to multiphase flow. *Jounal of Optimization Theory and Applications*, 111(3):473–488, 2001.

[67] D.E. Keyes, P.D. Hovland, L.C. McInnes, and W. Samyono. Using automatic differentiation for second-order matrix-free methods in pde-constrained optimization. In *Automatic Differentiation of Algorithms: From Simulation to Optimization (G. Corliss et al., eds.), Springer*, pages 35–50, 2000.

[68] J. Koski, H. Eschenauer, and A. Osyczka. *Multicriteria Design Optimization*. Springer - Verlag, Berlin, Germany, 1990.

[69] R. B. Lehoucq and A. G. Salinger. Large-scale eigenvalue calculations for stability analysis of steady flows on massively parallel computers. *International Journal for Numerical Methods in Fluids*, 36:309–327, 2001.

[70] J. L. Lions. Hierarchical control. *Proceedings of the Indian Academy of Sciences (Mathematical Sciences)*, 104(1):295–304, February 1994.

[71] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[72] Kevin R. Long. Solving partial differential equations with sundance. Technical report, Sandia National Laboratories, 2002.

[73] A. Lumsdanie and J. Siek. The matrix template library. http://www.lsc.nd.edu/research/mtl/, 1998.

[74] J.M. Marotzke, R. Giering, K.Q Zhang, D.Stammer, C. Hill, and T. Lee. Construction of the adjoint mit ocean general circulation model and application to atlantic heat transport sensitivity. Technical Report 63, Center for Global Change Science - MIT, 1999. submitted to Journal of Geophyics.

[75] A. Jameson N.A. Pierce L. Martinelli. Optimum aerodynamics design using the navier stokes equations. In *Proceedings of the AIAA-91-0100, 35th Aerospace Science Meeting and Exhibition*, 1997.

[76] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[77] E.P. Munz. Rarified gas dynamics. *Ann. Rev. Fluid Mech.*, (21):387, 1989.

[78] B.A. Murtagh and M.A. Saunders. MINOS 5.4 user's guide. Technical Report Report SOL 83-20R, Department of Operations Research, Stanford University, 1995.

[79] S. K. Nadarajah and A. Jameson J. Alonso. An adjoint method for the calculation of remote sensitivities in supersonic flow. In *AIAA-2002-0261*. AIAA, 2002.

[80] S. Nash and A. Sofer. *Linear and Nonlinear Programming*. McGraw Hill, 1996.

[81] G.A. Newman and D.L. Alumbaugh. 3-d electric magnetic inversion using conjugate gradients. Technical Report SAND97-1296C, Sandia National Laboratories, 1997.

[82] G.A. Newman and D.L. Alumbaugh. Three dimensional massively parallel electromagnectic inversion. *Geophysics Journal International*, (128):345–354, 1997.

[83] P.A. Newman, G.J. WHou, and A.C. Taylor. Observations regarding use of advanced analysis, sensitivity analysis, and design codes in cfd. Technical Report 96-16, NASA ICASE, Institute for Computer Applications in Science and Engineering, 1996.

[84] J. Nocedal and M. Overton. Projected hessian updating algorithms for nonlinear constrained optimization. *SIAM J. Numer. Anal.*, 22:821, 1985.

[85] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, 1999.

[86] C. E. Orozco and O. Ghattas. A reduced sand method for optimal design of nonlinear structures. *International Journal for Numerical Methods in Engineering*, 1997. to appear.

[87] B. Parker. Template composite operators. http://www.gil.com.au/bparker, 1997.

[88] R. P. Pawlowski, C. Theodoropoulos, A. G. Salinger, T. J. Mountziaris, H. K. Moffat, J. N. Shadid, and E. J. Thrush. Fundamental models of the metalorganic vapor-phase epitaxy of galluim nitride and their use in reactor design. *Journal of Crystal Growth*, 221:622–628, 2000.

[89] L. Petzold, J. B. Rosen, P.E. Gill, L.O. Jay, and K. Park. Numerical optimal control of parabolic pdes using dasopt. In *Large Scale Optimization with Applications, Part II: Optimal Design and Control Eds L. Biegler T. Coleman A. Conn F Santosa*. IAM Volumes in Mathematics and Its Applications Vol 1997 pp 288-311, 1997.

[90] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluids Mechanics*, (64), 1974.

[91] R. Pozo. *LAPACK++ v 1.1: High Performance Linear Algebra User's Guide*. NIST, 1996.

[92] U. Ringertz. Optimal design of nonlinear shell structures. Technical Report TN 91-18, The Aeronautical Research Institute of Sweden, 1991.

[93] U. Ringertz. An algorithm for optimization of nonlinear shell structures. *International Journal for Numerical Methods in Engineering*, (38):299–314, 1995.

[94] J. Sobieszczanski-Sobieski R.J. Balling. Optimization of couple systems: A critical overview of approaches. Technical Report 94-100, NASA ICASE Tech. Rep 94-100, Institute for Computer Applications in Science and Engineering, 1994.

[95] S. Roberts et al. Meschach++: Matrix computations in c++. http://www.netlib.org/c/meschach/, 1996.

[96] G. Booch J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[97] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[98] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, MA, 1996.

[99] A. G. Salinger, N.M. Bou-Rabee, E.A. Burroughs, R.B. Lehoucq, R.P. Pawlowski, L.A. Romero, and E.D. Wilkes. LOCA: A library of continuation algorithms - Theroy manual and user's guide. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 2002. SAND2002-0396.

[100] A. G. Salinger, K. D. Devine, G. L. Hennigan, H. K. Moffat, S. A. Hutchinson, and J. N. Shadid. MPSalsa: A finite element computer program for reacting flow problems - part II user's guide. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1996. SAND96-2331.

[101] A. G. Salinger, J. N. Shadid, S. A. Hutchinson, G. L. Hennigan, K. D. Devine, and H. K. Moffat. Analysis of gallium arsenide deposition in a horizontal chemical vapor deposition reactor using massively parallel computations. *Journal of Crystal Growth*, 203:516–533, 1999.

[102] A.G. Salinger, R.B. Lehoucq, and L.A Romero. Stability analysis of large-scale incompressible flow calculations on massively parallel computers. *CFD Journal*, 9(1):529–533, 2001.

[103] Sandia National Labs. ESI: Equation Solver Interface. `http://z.ca.sandia.gove/esi`, 2001.

[104] C. Schmid. *Reduced Hessian Successive Quadratic Programming for Large-Scale Process Optimization*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 1994.

[105] C. Schmid and L. T. Biegler. Acceleration of reduced-hessian methods for large-scale nonlinear programming. *Comp. Chem. Eng.*, 17:451, 1993.

[106] C. Schmid and L. T. Biegler. Quadratic programming methods for reduced hessian sqp. *Comp. Chem. Eng.*, 18:817, 1994.

[107] R.M. Sega and A. Igntiev. A space utlra-vacuum experiment - application to material processing. In *Proceedings of the AIAA/IKI Microgravity Science Symposium*, 1991.

[108] J. N. Shadid, H. K. Moffat, S. A. Hutchinson, G. L. Hennigan, K. D. Devine, and A. G. Salinger. MPSalsa: A finite element computer program for reacting flow problems - Part I theoretical development. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1996. SAND95-2752.

[109] J.N. Shadid. A fully-coupled Newton-Krylov solution method for parallel unstructured finite element fluid flow, heat and mass transport. *IJCFD*, 12:199–211, 1999.

[110] Standish, T.A. *Data Structures, Algorithms & Software Principles in C*. Addison-Wesley, 1994.

[111] J.R. Stewart and H.C. Edwards. The sierra framework for developing advanced parallel mechanics applications. In *Springer Verlag Lecture Notes*.

[112] B. Stroustrup. *The C++ Programming Language, 3rd edition*. Addison-Wesley, New York, 1997.

[113] Sun Microsystems. Java: The pure object oriented language for the web. `http://java.sun.com`.

[114] S. Tasan. One shot methods for optimal control of distributed parameter systems i: Finite dimensional control. Technical Report 91-2, ICASE NASA Langley, 1991.

[115] D. Ternet and L.T. Biegler. *New Approaches to a Reduced Hessian Successive Quadratic Programming Method for Large-Scale Process Optimization*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 1998.

[116] I. B. Tjoa and L. T. Biegler. A reduced successive quadratic programming strategy for errors-in-variables estimation. *Comp. Chem. Eng.*, 16:523, 1992.

[117] Vanderbi, R.J. . An Interior Point code for Quadratic Programming. Technical Report SOR 94-15, Princeton Univeristy, 1994.

[118] Varvarezos, D.K, L.T. Biegler, and I.E. Grossmann. Multiperiod Design Optimization with SQP Decomposition. *Comp. Chem. Eng.*, 18:1087, 1994.

[119] Veldhuizen, T. and E. Gannon. Active Libraries: Rethinking the Roles of Compilers and Libraries. http://oonumerics.org/blitz/papers/, 1998.

[120] L. Vicente and P. Calamai. Bilevel and multilevel programming: A bibliography review. *Journal of Global Optimization*, 6:1–16, 1994.

[121] H. von Stackelberg. *Marktform und Gleichgewicht*. J. Springer, Vienna, 1934.

[122] A. Wachter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, 2002.

[123] S. Wright. Optimization software packages. Technical Report ANL/MCS-P8xx-0899, Mathematics and Computer Science Division, Argonne National Laboratory, 1999.

[124] Z.Wang, K.K Droegemeier, L. White, and I.M. Navon. Application of a new adjoint newton algorithm to the 3-d arps storm scale model using simulated data. 19.

A rSQP++ Equation Summary and Nomenclature Guide

This is a summary of the mathematical expressions in an rSQP algorithm and the quantities in the rSQP++ implementation. This guide provides a precise mapping from mathematical quantities to identifier names used in rSQP++.

Standard NLP Formulation

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & c(x) = 0 \\ & x_L \leq x \leq x_U \end{array}$$

where:

$$\begin{array}{l} x, x_L, x_U \in \mathcal{X} \\ f(x) : \mathcal{X} \rightarrow \mathbf{R} \\ c(x) : \mathcal{X} \rightarrow \mathcal{C} \\ \mathcal{X} \in \mathbf{R}^n \\ \mathcal{C} \in \mathbf{R}^m \end{array}$$

Lagrangian

$$\begin{aligned} L(x, \lambda, \nu_L, \nu_U) = \; & f(x) + \lambda^T c(x) \\ & + (\nu_L)^T (x_L - x) \\ & + (\nu_U)^T (x - x_U) \end{aligned}$$

$$\nabla_x L(x, \lambda, \nu) = \nabla f(x) + \nabla c(x) \lambda + \nu$$

$$\nabla^2_{xx} L(x, \lambda) = \nabla^2 f(x) + \sum_{j=1}^{m} \lambda_j \nabla^2 c_j(x)$$

where:

$$\begin{array}{l} \lambda \in \mathcal{C} \\ \nu \equiv \nu_U - \nu_L \in \mathcal{X} \end{array}$$

Full Space QP Subproblem (Relaxed)

$$\begin{array}{ll} \min & g^T d + {}^1\!/_2 d^T W d + M(\eta) \\ \text{s.t.} & A^T d + (1 - \eta) c = 0 \\ & x_L - x_k \leq d \leq x_U - x_k \end{array}$$

where:

$$\begin{array}{l} d = x_{k+1} - x_k \in \mathcal{X} \\ g = \nabla f(x_k) \in \mathcal{X} \\ W = \nabla^2_{xx} L(x_k, \lambda_k) \in \mathcal{X}|\mathcal{X} \\ M(\eta) \in \mathbf{R} \rightarrow \mathbf{R} \\ A = \nabla c(x_k) \in \mathcal{X}|\mathcal{C} \\ c = c(x_k) \in \mathcal{C} \end{array}$$

Null-Space Decomposition

$$\begin{array}{ll} Z \in \mathcal{X}|\mathcal{Z} & \text{s.t. } (A_d)^T Z = 0 \\ Y \in \mathcal{X}|\mathcal{Y} & \text{s.t. } \left[\begin{array}{cc} Y & Z \end{array} \right] \text{ nonsingular} \\ R \equiv [(A_d)^T Y] \in \mathcal{C}_d|\mathcal{Y} & \text{nonsingular} \\ U_z \equiv [(A_u)^T Z] \in \mathcal{C}_u|\mathcal{Z} & \\ U_y \equiv [(A_u)^T Y] \in \mathcal{C}_u|\mathcal{Y} & \\ d = (1 - \eta) Y p_y + Z p_z & \end{array}$$

where:

$$\begin{array}{l} p_z \in \mathcal{Z} \subseteq \mathbf{R}^{(n-r)} \\ p_y \in \mathcal{Y} \subseteq \mathbf{R}^{r} \end{array}$$

Quasi-Normal (Range-Space) Subproblem

$$p_y = -R^{-1} c_d \in \mathcal{Y}$$

Tangential (Null-Space) Subproblem (Relaxed)

$$\begin{array}{ll} \min & g_{qp}^T p_z + {}^1\!/\!{}_2 p_z^T B p_z + M(\eta) \\ \text{s.t.} & U_z p_z + (1-\eta) u = 0 \\ & b_L \le Z p_z - (Y p_y)\eta \le b_U \end{array}$$

where:

$g_{qp} \equiv (g_r + \zeta w) \ \in \ \mathcal{Z}$
$g_r \equiv Z^T g \ \in \ \mathcal{Z}$
$w \equiv Z^T W Y p_y \ \in \ \mathcal{Z}$
$\zeta \ \in \ \mathbf{R}$
$B \approx Z^T W Z \ \in \ \mathcal{Z}|\mathcal{Z}$
$U_z \equiv [(A_u)^T Z] \ \in \ \mathcal{C}_u|\mathcal{Z}$
$U_y \equiv [(A_u)^T Y] \ \in \ \mathcal{C}_u|\mathcal{Y}$
$u \equiv U_y p_y + c_u \ \in \ \mathcal{C}_u$

$b_L \equiv x_L - x_k - Y p_y \ \in \ \mathcal{X}$
$b_U \equiv x_U - x_k - Y p_y \ \in \ \mathcal{X}$

Variable-Reduction Null-Space Decompositions

$$A^T = \begin{bmatrix} (A_d)^T \\ (A_u)^T \end{bmatrix} = \begin{bmatrix} C & N \\ E & F \end{bmatrix}$$

where:

$C \ \in \ \mathcal{C}_d|\mathcal{X}_D$ (nonsingular)
$N \ \in \ \mathcal{C}_d|\mathcal{X}_I$
$E \ \in \ \mathcal{C}_u|\mathcal{X}_D$
$F \ \in \ \mathcal{C}_u|\mathcal{X}_I$

Coordinate

$$Z \equiv \begin{bmatrix} -C^{-1}N \\ I \end{bmatrix}$$
$$Y \equiv \begin{bmatrix} I \\ 0 \end{bmatrix}$$
$R = C$
$U_z = F - EC^{-1}N$
$U_y = E$

Orthogonal

$D \equiv -C^{-1}N \ \in \ \mathcal{X}_D|\mathcal{X}_I$
$$Z \equiv \begin{bmatrix} D \\ I \end{bmatrix}$$
$$Y \equiv \begin{bmatrix} I \\ -D^T \end{bmatrix}$$
$R = C(I + DD^T)$
$U_z = F + ED$
$U_y = E - FD^T$

Mathematical Notation Summary and rSQP++ Identifier Mapping

Mathematical	rSQP++	Description
Iteration		
$k \in I_+$	`k`	Iteration counter for the SQP algorithm
NLP		
$n \in I_+$	`n`	Number of unknown variables in x
$m \in I_+$	`m`	Number of equality constraints in $c(x)$
$\mathcal{X} \in \mathbf{R}^n$	`space_x`	Vector space for x
$\mathcal{C} \in \mathbf{R}^m$	`space_c`	Vector space for $c(x)$
$x \in \mathcal{X}$	`x`	Unknown variables
$x_L \in \mathcal{X}$	`xl`	Lower bounds for variables
$x_U \in \mathcal{X}$	`xu`	Upper bounds for variables
$f(x)\|_x \in \mathbf{R}$	`f`	Objective function value at x
$g \equiv \nabla f(x) \in \mathcal{X}$	`Gf`	Gradient of the objective function at x
$c(x)\|_x \in \mathcal{C}$	`c`	General equality constraints evaluated at x
$A \equiv \nabla c(x)\|_x \in \mathcal{X}\|\mathcal{C}$	`Gc`	Gradient of $c(x)$ evaluated at x, $\nabla c = \left[\begin{array}{ccc} \nabla c_1 & \ldots & \nabla c_m \end{array} \right]$
Lagrangian		
$\lambda \in \mathcal{C}$	`lambda`	Lagrange multipliers for the general equality constraints
$\nu \in \mathcal{X}$	`nu`	Lagrange multipliers (sparse) for the variable bounds
$\nabla_x L(x_k, \lambda_k, \nu_k) \in \mathcal{X}$	`GL`	Gradient of the Lagrangian
$W \equiv \nabla^2_{xx} L(x_k, \lambda_k) \in \mathcal{X}\|\mathcal{X}$	`HL`	Hessian of the Lagrangian
SQP Step		
$d \in \mathcal{X}$	`d`	Full SQP step for the unknown variables, $d = (x_{k+1})^+ - x_k$
$\eta \in \mathbf{R}$	`eta`	Relaxation variable for QP subproblem
Null-Space Decomposition		
$r \in I_+$	`r`	Number decomposed equality constraints in c_d
$[1:r] \in I_+^2$	`con_decomp`	Range for decomposed equalities $c_d = c_{(1:r)}$
$[r+1:m] \in I_+^2$	`con_undecomp`	Range for undecomposed equalities $c_u = c_{(r+1:m)}$
$\mathcal{C}_d \in \mathbf{R}^r$	`space_c .sub_space( con_decomp)`	Vector space for decomposed equalities c_d
$\mathcal{C}_u \in \mathbf{R}^{(m-r)}$	`space_c .sub_space( con_undecomp)`	Vector space for undecomposed equalities c_u

$c_d = c_{(1:r)} \in \mathcal{C}_d$	`c.sub_view( con_decomp)`	Vector of decomposed equalities
$c_u = c_{(r+1:m)} \in \mathcal{C}_u$	`c.sub_view( con_undecomp)`	Vector of undecomposed equalities
$\mathcal{Z} \in \mathbf{R}^{(n-r)}$	`Z.space_rows()`	Null space. Accessed from the matrix object `Z`.
$\mathcal{Y} \in \mathbf{R}^{r}$	`Y.space_rows()`	Quasi-Range space. Accessed from the matrix object `Y`.
$Z \in \mathcal{X}\|\mathcal{Z}$	`Z`	Null-space matrix for $(\nabla c_d)^T$ $((\nabla c_d)^T Z = 0)$
$Y \in \mathcal{X}\|\mathcal{Y}$	`Y`	Quasi-range-space matrix for $(\nabla c_d)^T$ ($[Y\ Z]$ nonsingular)
$R = [(\nabla c_d)^T Y] \in \mathcal{C}_d\|\mathcal{Y}$	`R`	
$U_z = [(\nabla c_u)^T Z] \in \mathcal{C}_u\|\mathcal{Z}$	`Uz`	
$U_y = [(\nabla c_u)^T Y] \in \mathcal{C}_u\|\mathcal{Y}$	`Uy`	
$p_z \in \mathcal{Z}$	`pz`	Tangential (null-space) step
$Zp_z \in \mathcal{X}$	`Zpz`	Tangential (null-space) contribution to d
$p_y \in \mathcal{Y}$	`py`	Quasi-normal (quasi-range-space) step
$Yp_y \in \mathcal{X}$	`Ypy`	Quasi-norm (quasi-range-space) contribution to d
$g_r = Z^T \nabla f \in \mathcal{Z}$	`rGf`	Reduced gradient of the objective function
$Z_T \nabla L \in \mathcal{Z}$	`rGL`	Reduced gradient of the Lagrangian
$w \approx Z^T W Y p_y \in \mathcal{Z}$	`w`	Reduced QP cross term
$B \approx Z^T W Z \in \mathcal{Z}\|\mathcal{Z}$	`rHL`	Reduced Hessian of the Lagrangian
Reduced QP Subproblem		
$g_{qp} \equiv (g_r + \zeta w) \in \mathcal{Z}$	`qp_grad`	Gradient for the Reduced QP subproblem
$\zeta \in \mathbf{R}$	`zeta`	QP cross term damping parameter (descent for $\phi(x)$)
Global Convergence		
$\alpha \in \mathbf{R}$	`alpha`	Step length for $x_{k+1} = x_k + \alpha d$
$\mu \in \mathbf{R}$	`mu`	Penalty parameter used in the merit function $\phi(x)$
$\phi(x) : \mathcal{X} \rightarrow \mathbf{R}$	`merit_func_nlp`	Merit function object that computes $\phi(x)$
$\phi(x)\|_x \in \mathbf{R}$	`phi`	Value of the merit function $\phi(x)$ at x
Variable Reduction Decomposition		
$[1:r] \in I_+^2$	`var_dep`	Range for dependent variables $x_D = x_{(1:r)}$
$[r+1:n] \in I_+^2$	`var_indep`	Range for independent variables $x_I = x_{(r+1:n)}$
$Q_x \in \mathcal{X}\|\mathcal{X}$	`P_var`	Permuation for the variables for current basis
$Q_c \in \mathcal{C}\|\mathcal{C}$	`P_equ`	Permuation for the constraints for current basis
$\mathcal{X}_D \in \mathbf{R}^r$	`space_x .sub_space( var_dep)`	Vector space for dependent variables x_D

$\mathcal{X}_I \in \mathbf{R}^{(n-r)}$	`space_x .sub_space( var_indep)`	Vector space for independent variables x_I
$x_D \in \mathcal{X}_D$	`x.sub_view( var_dep)`	Vector of dependent variables
$x_I \in \mathcal{X}_I$	`x.sub_view( var_indep)`	Vector of independent variables
$C \equiv \nabla_D c_d(x_k)^T \equiv (A^T)_{(1:r,1:r)} \in \mathcal{C}_d\|\mathcal{X}_D$	`C`	Nonsingular Jacobian submatrix (basis) for dependent variables x_D and decomposed constraints $c_d(x)$ at x_k
$N \equiv \nabla_I c_d(x_k)^T \equiv (A^T)_{(1:r,r+1:n)} \in \mathcal{C}_d\|\mathcal{X}_I$	`N`	Jacobian submatrix for independent variables x_I and decomposed constraints $c_d(x)$ at x_k
$E \equiv \nabla_D c_u(x_k)^T \equiv (A^T)_{(r+1:m,1:r)} \in \mathcal{C}_u\|\mathcal{X}_D$	`E`	Jacobian submatrix for dependent variables x_D and undecomposed constraints $c_u(x)$ at x_k
$F \equiv \nabla_I c_u(x_k)^T \equiv (A^T)_{(r+1:m,r+1:n)} \in \mathcal{C}_u\|\mathcal{X}_I$	`F`	Jacobian submatrix for independent variables x_I and undecomposed constraints $c_u(x)$ at x_k

B Installation of rSQP++

The C++ source code for rSQP++, its supporting packages and a few simple examples are distributed as a single source tree. The build system uses GNU make which is available on Linux, Unix and even Microsoft Windows (using cygwin). The build system is designed primarily for development work and therefore not as easy to install as with installation methods based on GNU automake and autoconf. The distribution comes as a gziped tar file of the name `rSQPpp.tar.gz`. To install the core distribution (assuming a Linux/Unix system), create a base directory and untar the sources. For example, assuming the userid is `joesmith` and the tar file is in Joe's home directory, Joe would perform the following

```
$ mkdir /home/joesmith/rSQPpp.base
$ cd /home/joesmith/rSQPpp.base
$ tar -xzvf /home/joesmith/rSQPpp.tar.gz
```

An environment variable `RSQPPP_BASE_DIR` should then be set to the base directory for rSQP++ as follows (assuming the `bash` shell is being used)

```
$ export RSQPPP_BASE_DIR=/home/joesmith/rSQPpp.base
```

This environment variable (as well as a few others) is used extensively by the build system and the test suite.

The untared source tree should look like the following

```
 $RSQPPP_BASE_DIR/
   |
    -- rSQPpp/
         |
         |-- build
         |
         |-- core
         |   |
         |   |-- AbstractLinAlgPack
         |   |
         |    ...
```

```
|
|-- design
|
|-- doc
|
|-- examples
|    |
|    |-- ExampleNLPBanded
|    |
|     ...
|
|-- design
|
|-- testing
```

For detailed up-to-date information on the installation of rSQP++ for various platforms, see the file

`$RSQPPP_BASE_DIR/rSQPpp/README`

The above `README` file references several other README files that describe the build system, the Doxygen documentation system, the test suite and other topics. The included test suite is fairly extensive and is self checking. The test suite should build and run successfully before any work with rSQP++ attempted. The test suite is also extensible and allows an advanced user to easily add new test modules that can be run with a single command.

Once the proper environment variables are setup the Doxygen generated html pages can be generated. Before the documentation can be generated, the Doxygen configuration files must be setup. To find out how to do this see the file

`$RSQPPP_BASE_DIR/rSQPpp/doc/README.DOCUMENTATION`.

After the configuration files are setup the doxygen documentation can be build using the script

`$RSQPPP_BASE_DIR/rSQPpp/build_doc`.

For most users, however, building the documentation locally is not necessary as prebuild documentation can be found at

`RSQPPP_BASE_DOC/html/index.html`

where `RSQPPP_BASE_DOC`[1] is the URL to the rSQP++ documentation web site.

Although, using Doxygen for your own source code can be very useful in helping to navigate the code.

The simplest way to get starting in solving a custom NLP using rSQP++ is to add a new project to the rSQP++ build system. There is a HowTo file that describes the process of adding a new project to the build system which can be found at

`$RSQPPP_BASE_DIR/rSQPpp/doc/HowTo.NewBuildProject`.

Using the rSQP++ build system is optional as it is possible to simply include the proper Cpp directives in your own build system and then to link to precompiled rSQP++ libraries but this will be much more involved. Using the rSQP++ build system is much easier.

For simpler use of rSQP++, it is possible to use the solvers through one of the prebuilt interfaces to modeling environments like AMPL (see ???). See ??? for a description of some example NLPs for rSQP++.

[1] `RSQPPP_BASE_DOC = http://dynopt.cheme.cmu.edu/roscoe/rSQPpp/doc`

C Descriptions of Individual rSQP++ Packages

`Misc` : Heterogeneous Collection of Utilities

`Misc` is not a package (i.e. C++ `namespace`) at all. Instead, it is just of heterogeneous collection of general programming utilities that really do not belong to any other higher level package exclusively. This package is not shown in Figure 4.3 but all of the other packages depend on components in *Misc*. Most of these utilities fall into one of two categories: memory management and options setting.

There are several C++ classes to aid in memory management. Since C++ allows dynamic memory allocation, does not have garbage collection, and uses pointers to raw memory, memory management is one of the more difficult, if not the most difficult, aspect to using C++. By far, the most important utility class for memory management is `MemMngPack::ref_count_ptr<T>`. This is a templated smart reference counted pointer class modeled after `std::auto_ptr<T>` and the ideas in [76]. The careful and consistent use of objects of this class effectively allow garbage collection in C++. Many other strategies have been proposed for automatic memory management in C++ but the style used by `ref_count_ptr<T>` is the most flexible in many respects. This class forms the foundation for all dynamic memory management in rSQP++ and its lower level packages. The development of this class has been very significant and has allowed things to be done in rSQP++ that would have been nearly impossible to do otherwise.

While `ref_count_ptr<T>` is more than adequate for memory management when all the peers know at least a base class of the objects to be garbage collected, this is not always possible (unlike Java [113], C++ does not have a universal base class called `Object` from which all other classes derive). For example, suppose one peer is given a pointer to a row of a dynamically allocated matrix while another peer is given a pointer to a column of the same matrix. Also, suppose that for the sake of flexibility, these same peers may be given pointers to separately allocated vectors to use. In each case, once each of the peers is finished using the vectors they have been given, it is important that the memory is released so that a memory leak does not occur. In the latter case, once a peer is finished with a vector, the separately allocated buffer of memory should be released. This is done independently of the other peer. However, in the former case, the dynamically allocated matrix should not be released until both of the peers are finished using vectors from this matrix.

So the basic idea here is that a client may be given an object of one type that is dependent on some other dynamically allocated object(s), but does not know how to properly release memory associated with the object once it has finished using it. To allow this type of greater flexibility in memory management, the abstract interface `MemMngPack::`*`ReleaseResource`* has been defined. The use of this class is very simple. A client is given an object `a` of known type `A` to interact with and a pointer `r` to a companion *`ReleaseResource`* object. Once the client is finished using the object `a`, it calls `delete r` and the overridden virtual destructor `r->~Release-Resource()` is called on an object that knows what to delete. A single subclass implementation of the *`ReleaseResource`* interface called `ReleaseResource_ref_count_ptr` has been implemented using the `ref_count_ptr<T>` class. When the overridden virtual function `ReleaseResource_ref_count_ptr::~ReleaseResource_ref_count_ptr()` is called, it calls the destructor on the composite `ref_count_ptr<T>` member `ptr` which calls `delete` on the raw memory to be released. This might seem like much ado about nothing but these two classes have been sufficient for all the (sometimes complex) memory management in rSQP++. This important concept was designed late in the development of rSQP++ but has allowed the creation of some much more flexible software since its adoption.

While the classes `MemMngPack::ref_count_ptr<T>` and `MemMngPack::`*`Release-Resource`* allow the flexible deletion of an object or objects after a client is finished using them, they do not allow the flexible creation of objects. For this purpose, the interface *`MemMngPack-::AbstractFactory<T>`* has been developed which is a universal templated interface for the "factory" pattern [???]. The single virtual method is *`create()`* which returns a `ref_count-_ptr<T>` object containing the allocated object. There is a single subclass

```
namespace MemMngPack {
    template <class T_itfc, class T_impl, class T_PostMod = PostModNothing<T_impl>
        ,class T_Allocator = AllocatorNew<T_impl> >
    class AbstractFactoryStd : public AbsractFactory<T_itfc>;
}
```

which is templated on the interface type `T_itfc` that is represented by the *`Abstract-Factory`* base interface, the concrete implementation type `T_impl`, and also by policy classes that determine how the underlying object is allocated (`T_PostMod`) and how it is modified after allocation (`T_Allocator`). The policy classes have default types which allocate using `new` (`AllocatorNew<T_impl>`) and do no post modification after the initial construction (`Post-ModNothing<T_impl>`). Using these policy classes with the C++ template mechanisms to create different instantiations allows complete flexibility in how objects are allocated and initialized and therefore the `AbstractFactoryStd<...>` subclass is really the only abstract factory subclass needed.

Aside from the type of general dynamic memory management that the C++ operators `new` and delete and the C functions malloc(...) and free(...) were designed for, there is also a need for general workspace that is used during the execution of a C++ function. In Fortran 77, this type of memory must be explicitly passed into a subroutine and clutters the interface. In Fortran 90, this type of memory can be created on-the-fly within a subroutine, but most implementations allocate this memory from the stack and not the heap. The Fortran 90 implementation of automatic workspace has caused problems on several platforms when allocating huge amounts of data. What is needed is a more flexible means to efficiently allocate and release workspace used in a function. For this purpose, the templated class WorkspacePack::Workspace<T> has been designed. Objects of this type can only be allocated on the stack (i.e. operators new and delete have been made private and are not defined as discussed in [76]) and must be given a reference to a WorkspacePack::WorkspaceStore object which is used to obtain a temporary buffer of data. The current implementation of WorkspaceStore allocates a large chunk of memory at once from the operating system and then gives it out as needed. Any memory demands beyond the preallocated amount are handled by new. The WorkspaceStore implementation also keeps statistics that can be used for fine tuning the memory usage later on. Because of the order that C++ creates and destroys automatic objects that are put on the stack, the implementations of WorkspacePack::Workspace<T> and WorkspacePack::WorkspaceStore are very simple and require only $O(1)$ overhead. This is very different from the overhead that can occur from using malloc(...) because of the more complex tasks the operating system has to perform to manage the heap (i.e. regulate fragmentation etc.) as described in [110, Section 8.6]. See the file WorkspacePack.h for more details.

Aggregation and composition are so common and the tasks of writing access functions and data members for C++ classes with aggregate objects are so monotonous that preprocessor macros have been written to automatically insert all the needed declarations. The macro

```
STANDARD_MEMBER_COMPOSITION_MEMBERS(type_name,attribute_name)
```

is used to insert to declarations for a simple member object of a concrete class with value semantics. For example, options such as tolerances (i.e. type_name = double), flags (i.e. type_name = bool) and maximum iteration counts (i.e. type_name = int) can be included in a class interface using this macro. This has relieved the writing of a lot of boiler plate code that had to be written by hand before. However, many objects are polymorphic and do not use value semantics (i.e. those that are instantiations of a subclass). For composition relationships (i.e. memory management obligations assumed) for these types of objects (both polymorphic and non-polymorphic) the macro

```
STANDARD_COMPOSITION_MEMBERS(basetype_name,obj_name)
```

has been defined. This macro inserts the declarations for the member access functions and includes a private data member of type `ref_count_ptr<basetype_name>` to handle the dynamic memory management. For these types of composite associations, when the client object is destroyed, the composite object `obj_name` may also be destroyed (if no other clients are using it) and `ref_count_ptr<T>` takes care of this automatically. For associations that are strictly aggregate (i.e. no ownership of memory is assumed) the macro

```
STANDARD_AGGREGATION_MEMBERS(basetype_name,obj_name)
```

is used. This macro inserts a private data member that is a simple pointer.

Another very useful class is `OptionsFromStreamPack::OptionsFromStream`. This class allows options to be read from a text stream, which is formatted in a very human readable, self documenting manner. Many of the major classes in rSQP++ can accept options in this form. These options can be included in a file or generated in a string within code. Strictly speaking, this is a weakly typed way to specify options but there are a lot of safeguards that make its use more or less bulletproof. For example, see how this text stream is formatted in Section 4.3.1.1. A lot more could be said about how to use the class `OptionsFromStream` from both a user's and developer's point of view, but the interested user can look in the code for examples.

D Samples of Input and Output for rSQP++

Here, portions of the output generated for the example program `ExampleNLPBanded` is given. Lines in the output consisting of three dots

```
...
```

are for parts of the output that have been ommited for the sake of space. This output was generated using the command line

```
$ ./solve_example_nlp --nD=30000 --bw=10 --nI=400 --diag-scal=1e+4 --xo=10.0
```

and the options file shown in Section 8.8. The output to the console is shown in Section 8.8 while excepts from the output files `rSQPppAlgo.out`, `rSQPppSummary.out` and `rSQPpp-Journal.out` are shown in Sections 8.8–8.8.

Note that the content of the output may be different a more current version of rSQP++ than the one used at the time of this writting. However, the general layout of the information will be generally the same.

Input file `rSQPpp.opt`

```
begin_options

options_group rSQPppSolver {
    test_nlp = true; *** (default)
*    test_nlp = false;
    print_algo = true; *** (default)
*    print_algo = false;
    algo_timing = true; *** (default)
*    algo_timing = false;
    configuration = mama_jama;      *** (default)
*    configuration = interior_point;
}

options_group rSQPSolverClientInterface {
*    max_iter = 1000;  *** (default?)
*    max_iter = 3;
*    max_run_time = 1e+10; *** (default?)
*    opt_tol = 1e-6;  *** (default?)
    opt_tol = 1e-8; *** (default=1e-6)
*    feas_tol = 1e-6;  *** (default?)
    feas_tol = 1e-10;  *** (default=1e-6 )
*    step_tol = 1e-2;  *** (default?)
*    journal_output_level = PRINT_NOTHING;              * No output to journal from algorithm
*    journal_output_level = PRINT_BASIC_ALGORITHM_INFO; * O(1) information usually
    journal_output_level = PRINT_ALGORITHM_STEPS;      * O(iter) output to journal     (default)
*    journal_output_level = PRINT_ACTIVE_SET;           * O(iter*nact) output to journal
*    journal_output_level = PRINT_VECTORS;              * O(iter*n) output to journal   (lots!)
*    journal_output_level = PRINT_ITERATION_QUANTITIES; * O(iter*n*m) output to journal (big lots!)
*    journal_print_digits = 6;  *** (default?)
    check_results = true;  *** (costly?)
*    check_results = false; *** (default?)
}

options_group DecompositionSystemStateStepBuilderStd {
    null_space_matrix = AUTO;         *** Let the solver decide (default)
*    null_space_matrix = EXPLICIT;     *** Store D = -inv(C)*N explicitly
*    null_space_matrix = IMPLICIT;     *** Perform operations implicity with C, N
    range_space_matrix = AUTO;        *** Let the algorithm decide dynamically (default)
*    range_space_matrix = COORDINATE;  *** Y = [ I; 0 ] (Cheaper computationally)
*    range_space_matrix = ORTHOGONAL;  *** Y = [ I; -N'*inv(C') ] (more stable)
    max_dof_quasi_newton_dense = 500; *** (default=-1, let the solver decide)
}

options_group rSQPAlgo_ConfigMamaJama {
    quasi_newton = AUTO;   *** Let solver decide dynamically (default)
*    quasi_newton = BFGS;   *** Dense BFGS
*    quasi_newton = LBFGS;  *** Limited memory BFGS
*    line_search_method = AUTO;               *** Let the solver decide dynamically (default)
*    line_search_method = NONE;               *** Take full steps at every iteration
    line_search_method = DIRECT;             *** Use standard Armijo backtracking
*    line_search_method = FILTER;             *** Filter
}

options_group NLPTester {
*    print_all = true;
    print_all = false; *** (default)
}

options_group NLPFirstDerivativesTester {
*    fd_testing_method = FD_COMPUTE_ALL; *** Compute all of the derivatives (O(m))
    fd_testing_method = FD_DIRECTIONAL; *** Only compute along random directions (O(1))
    num_fd_directions = 1;  *** [fd_testing_method == DIRECTIONAL]
    warning_tol   = 1e-10;
    error_tol     = 1e-5;
}

options_group CalcFiniteDiffProd {
*    fd_method_order = FD_ORDER_ONE;          *** Use O(eps) one sided finite differences
*    fd_method_order = FD_ORDER_TWO;          *** Use O(epsˆ2) one sided finite differences
*    fd_method_order = FD_ORDER_TWO_CENTRAL;  *** Use O(epsˆ2) two sided central finite differences
*    fd_method_order = FD_ORDER_TWO_AUTO;     *** Uses FD_ORDER_TWO_CENTRAL or FD_ORDER_TWO
*    fd_method_order = FD_ORDER_FOUR;         *** Use O(epsˆ4) one sided finite differences
    fd_method_order = FD_ORDER_FOUR_CENTRAL; *** Use O(epsˆ4) two sided central finite differences
*    fd_method_order = FD_ORDER_FOUR_AUTO;    *** (default) Uses FD_ORDER_FOUR_CENTRAL or FD_ORDER_FOUR
*    fd_step_select = FD_STEP_ABSOLUTE; *** (default) Use absolute step size fd_step_size
*    fd_step_select = FD_STEP_RELATIVE; *** Use relative step size fd_step_size * ||x||inf
*    fd_step_size = -1.0; *** (default) Let the implementation decide
*    fd_step_size_min = -1.0; *** (default) Let the implementation decide.
*    fd_step_size_f = -1.0; *** (default) Let the implementation decide
*    fd_step_size_c = -1.0; *** (default) Let the implementation decide
*    fd_step_size_h = -1.0; *** (default) Let the implementation decide
}

end_options
```

Console output

The following is output to the console.

```
$ ./solve_example_nlp.rel --nD=30000 --bw=10 --nI=400 --diag-scal=1e+4 --xo=10.0


********************************
*** Start of rSQP Iterations ***
n = 30400, m = 30000, nz = 599910

 k    f         ||c||s    ||rGL||s  QN #act ||Ypy||2 ||Zpz||2 ||d||inf alpha
 ---- --------- --------- --------- -- ---- -------- -------- -------- --------
    0  1.5e+006  1.2e+007  1.2e+002 IN    0   2e+003   4e+005   2e+003    0.001
    1  7.1e+005  1.1e+007        41 SK    0   1e+003   1e+005   6e+002     0.01
    2  7.7e+004  3.7e+006      0.35 SK    0   2e+002   2e+002        6        1
    3  3.2e+004  1.1e+006      0.23 SK    0   1e+002   8e+001        3        1
    4  1.5e+004    3e+005      0.64 SK    0   6e+001   1e+002        6        1
    5  4.4e+003  5.1e+004       2.4 SK    0   8e+001   4e+002        8      0.1
    6  2.5e+003  3.3e+004       0.4 SK    0   3e+001   4e+001        2        1
    7    8e+002  9.6e+003      0.03 SK    0   3e+001        1      0.2        1
    8  4.5e+002    6e+002       0.6 SK    0        1   3e+001        2        1
    9      0.78  1.2e+002     0.014 UP    0        1      0.1     0.01        1

 k    f         ||c||s    ||rGL||s  QN #act ||Ypy||2 ||Zpz||2 ||d||inf alpha
 ---- --------- --------- --------- -- ---- -------- -------- -------- --------
   10     0.012       3.3     0.019 SK    0     0.01      0.2     0.02        1
   11  2.1e-007      0.12  8.7e-005 UP    0   0.0002   0.0006   9e-005        1
   12  3.4e-015  2.1e-005  3.6e-008 UP    0   2e-008   8e-008   4e-008        1
 ---- --------- --------- --------- -- ----
   13  6.3e-024  1.5e-012  3.3e-012  -    -   1e-015        -        -

Total time = 6e+001 sec

Jackpot! You have found the solution!!!!!!

Number of function evaluations:
-------------------------------
f(x)  : 96
c(x)  : 96
Gf(x) : 15
Gc(x) : 15
Solution Found!
```

Output file `rSQPppAlgo.out`

```
********************************************************************
*** Algorithm information output                                 ***
***                                                              ***
*** Below, information about how the the rSQP++ algorithm is     ***
*** setup is given and is followed by detailed printouts of the  ***
*** contents of the algorithm state object (i.e. iteration       ***
*** quantities) and the algorithm description printout           ***
*** (if the option rSQPppSolver::print_algo = true is set).      ***
********************************************************************

*** Echoing input options ...

...

*** Setting up to run rSQP++ on the NLP using a configuration object of type 'class ReducedSpaceSQPPack::rSQPAlgo_ConfigMamaJama' ...

*****************************************************************
*** rSQPAlgo_ConfigMamaJama configuration                     ***
***                                                           ***
*** Here, summary information about how the algorithm is      ***
*** configured is printed so that the user can see how the    ***
*** properties of the NLP and the set options influence       ***
*** how an algorithm is configured.                           ***
*****************************************************************

*** Creating the rSQPAlgo algo object ...

*** Setting the NLP and track objects to the algo object ...

*** Probing the NLP object for supported interfaces ...

Detected that NLP object supports the NLPFirstOrderInfo interface!

range_space_matrix == AUTO:
(n-r)^2*r = (400)^2 * 30000 = 505032704 > max_dof_quasi_newton_dense^2 = (500)^2 = 250000
setting range_space_matrix = COORDINATE

*** Setting option defaults for options not set by the user or determined some other way ...
```

```
null_space_matrix_type == AUTO: Let the algorithm deside as it goes along

*** End setting default options

*** Sorting out some of the options given input options ...

...

quasi_newton == AUTO:
nlp.num_bounded_x() == 0:
n-r = 400 <= max_dof_quasi_newton_dense = 500:
setting quasi_newton == BFGS

...

*** Creating the state object and setting up iteration quantity objects ...

*** Creating and setting the step objects ...

Configuring an algorithm for a nonlinear equality constrained NLP ( m > 0 && mI == 0 && num_bounded_x == 0) ...

*** Algorithm Steps ***

1. "EvalNewPoint"
    (class ReducedSpaceSQPPack::EvalNewPointStd_Step)
2. "RangeSpaceStep"
    (class ReducedSpaceSQPPack::RangeSpaceStepStd_Step)
2.1. "CheckDecompositionFromPy"
    (class ReducedSpaceSQPPack::CheckDecompositionFromPy_Step)
2.2. "CheckDecompositionFromRPy"
    (class ReducedSpaceSQPPack::CheckDecompositionFromRPy_Step)
2.3. "CheckDescentRangeSpaceStep"
    (class ReducedSpaceSQPPack::CheckDescentRangeSpaceStep_Step)
3. "ReducedGradient"
    (class ReducedSpaceSQPPack::ReducedGradientStd_Step)
4. "CalcReducedGradLagrangian"
    (class ReducedSpaceSQPPack::CalcReducedGradLagrangianStd_AddedStep)
5. "CheckConvergence"
    (class ReducedSpaceSQPPack::CheckConvergenceStd_AddedStep)
6.-1. "CheckSkipBFGSUpdate"
    (class ReducedSpaceSQPPack::CheckSkipBFGSUpdateStd_Step)
6. "ReducedHessian"
    (class ReducedSpaceSQPPack::ReducedHessianSecantUpdateStd_Step)
7. "NullSpaceStep"
    (class ReducedSpaceSQPPack::NullSpaceStepWithoutBounds_Step)
8. "CalcDFromYPYZPZ"
    (class ReducedSpaceSQPPack::CalcDFromYPYZPZ_Step)
9.-2. "LineSearchFullStep"
    (class ReducedSpaceSQPPack::LineSearchFullStep_Step)
9.-1. "MeritFunc_PenaltyParamUpdate"
    (class ReducedSpaceSQPPack::MeritFunc_PenaltyParamUpdateMultFree_AddedStep)
9. "LineSearch"
    (class ReducedSpaceSQPPack::LineSearchFailureNewDecompositionSelection_Step)

*** NLP ***
class NLPInterfacePack::ExampleNLPBanded

*** Iteration Quantities ***

...

*** Algorithm Description ***

1. "EvalNewPoint"
    (class ReducedSpaceSQPPack::EvalNewPointStd_Step)
    *** Evaluate the new point and update the range/null decomposition
    if nlp is not initialized then initialize the nlp
    if x is not updated for any k then set x_k = xinit
    if m > 0 and Gc_k is not updated Gc_k = Gc(x_k) <: space_x|space_c
    if mI > 0 Gh_k is not updated Gh_k = Gh(x_k) <: space_x|space_h
    if m > 0 then
      For Gc_k = [ Gc_k(:,equ_decomp), Gc_k(:,equ_undecomp) ] where:
        Gc_k(:,equ_decomp) <: space_x|space_c(equ_decomp) has full column rank r
      Find:
        Z_k  <: space_x|space_null    s.t. Gc_k(:,equ_decomp)' * Z_k = 0
        Y_k  <: space_x|space_range   s.t. [Z_k Y_k] is nonsingular
        R_k  <: space_c(equ_decomp)|space_range
                                      s.t. R_k = Gc_k(:,equ_decomp)' * Y_k
        if m > r : Uz_k <: space_c(equ_undecomp)|space_null
                                      s.t. Uz_k = Gc_k(:,equ_undecomp)' * Z_k
        if m > r : Uy_k <: space_c(equ_undecomp)|space_range
                                      s.t. Uy_k = Gc_k(:,equ_undecomp)' * Y_k
        if mI > 0 : Vz_k <: space_h|space_null
                                      s.t. Vz_k = Gh_k' * Z_k
        if mI > 0 : Vy_k <: space_h|space_range
                                      s.t. Vy_k = Gh_k' * Y_k
      begin update decomposition (class 'class ReducedSpaceSQPPack::DecompositionSystemHandlerVarReductPerm_Strategy')
        *** Updating or selecting a new decomposition using a variable reduction
        *** range/null decomposition object.

        ...

      end update decomposition
      if ( (decomp_sys_testing==DST_TEST)
        or (decomp_sys_testing==DST_DEFAULT and check_results==true)
        ) then
        check properties for Z_k, Y_k, R_k, Uz_k, Uy_k, Vz_k and Vy_k.
      end
    end
    Gf_k = Gf(x_k) <: space_x
    if m > 0 and c_k is not updated c_k = c(x_k) <: space_c
    if mI > 0 and h_k is not updated h_k = h(x_k) <: space_h
    if f_k is not updated f_k = f(x_k) <: REAL
    if ( (fd_deriv_testing==FD_TEST)
      or (fd_deriv_testing==FD_DEFAULT and check_results==true)
      ) then
      check Gc_k (if m > 0), Gh_k (if mI > 0) and Gf_k by finite differences.
    end

2. "RangeSpaceStep"
    (class ReducedSpaceSQPPack::RangeSpaceStepStd_Step)
    *** Calculate the range space step
    py_k = - inv(R_k) * c_k(equ_decomp)
    Ypy_k = Y_k * py_k

2.1. "CheckDecompositionFromPy"
    (class ReducedSpaceSQPPack::CheckDecompositionFromPy_Step)

    ...

2.2. "CheckDecompositionFromRPy"
    (class ReducedSpaceSQPPack::CheckDecompositionFromRPy_Step)
    *** Try to detect when the decomposition is becomming illconditioned

    ...

2.3. "CheckDescentRangeSpaceStep"
    (class ReducedSpaceSQPPack::CheckDescentRangeSpaceStep_Step)
    *** Check for descent in the decomposed equality constraints for the range space step

    ...

3. "ReducedGradient"
    (class ReducedSpaceSQPPack::ReducedGradientStd_Step)
    *** Evaluate the reduced gradient of the objective funciton
    rGf_k = Z_k' * Gf_k

4. "CalcReducedGradLagrangian"
    (class ReducedSpaceSQPPack::CalcReducedGradLagrangianStd_AddedStep)
    *** Evaluate the reduced gradient of the Lagrangian
    if nu_k is updated then
        rGL_k = Z_k' * (Gf_k + nu_k) + GcUP_k' * lambda_k(equ_undecomp)
                + GhUP_k' * lambdaI_k(inequ_undecomp)
    else
        rGL_k = rGf_k + GcUP_k' * lambda_k(equ_undecomp)
                + GhUP_k' * lambdaI_k(inequ_undecomp)
    end

5. "CheckConvergence"
    (class ReducedSpaceSQPPack::CheckConvergenceStd_AddedStep)
    *** Check to see if the KKT error is small enough for convergence
    if scale_(opt|feas|comp)_error_by == SCALE_BY_ONE then
        scale_(opt|feas|comp)_factor = 1.0
    else if scale_(opt|feas|comp)_error_by == SCALE_BY_NORM_2_X then
        scale_(opt|feas|comp)_factor = 1.0 + norm_2(x_k)
    else if scale_(opt|feas|comp)_error_by == SCALE_BY_NORM_INF_X then
        scale_(opt|feas|comp)_factor = 1.0 + norm_inf(x_k)
    end
    if scale_opt_error_by_Gf == true then
        opt_scale_factor = 1.0 + norm_inf(Gf_k)
    else
        opt_scale_factor = 1.0
    end
    opt_err = norm_inf(rGL_k)/opt_scale_factor
    feas_err = norm_inf(c_k)
    comp_err = max(i, nu(i)*(xu(i)-x(i)), -nu(i)*(x(i)-xl(i)))
    opt_kkt_err_k = opt_err/scale_opt_factor
    feas_kkt_err_k = feas_err/scale_feas_factor
    comp_kkt_err_k = feas_err/scale_comp_factor
    if d_k is updated then
        step_err = max( |d_k(i)|/(1+|x_k(i)|), i=1..n )
    else
        step_err = 0
    end
    if opt_kkt_err_k < opt_tol
           and feas_kkt_err_k < feas_tol
           and step_err < step_tol then
       report optimal x_k, lambda_k and nu_k to the nlp
       terminate, the solution has beed found!
    end

6.-1. "CheckSkipBFGSUpdate"
    (class ReducedSpaceSQPPack::CheckSkipBFGSUpdateStd_Step)
    *** Check if we should do the BFGS update

    ...

6. "ReducedHessian"
    (class ReducedSpaceSQPPack::ReducedHessianSecantUpdateStd_Step)
    *** Calculate the reduced hessian of the Lagrangian rHL = Z' * HL * Z

    ...


7. "NullSpaceStep"
    (class ReducedSpaceSQPPack::NullSpaceStepWithoutBounds_Step)
    *** Calculate the null space step by solving an unconstrainted QP
    qp_grad_k = rGf_k + zeta_k * w_k
    solve:
        min     qp_grad_k' * pz_k + 1/2 * pz_k' * rHL_k * pz_k
        pz_k <: R^(n-r)
    Zpz_k = Z_k * pz_k
    nu_k = 0

8. "CalcDFromYPYZPZ"
    (class ReducedSpaceSQPPack::CalcDFromYPYZPZ_Step)
    *** Calculates the search direction d from Ypy and Zpz
    d_k = Ypy_k + Zpz_k

9.-2. "LineSearchFullStep"
    (class ReducedSpaceSQPPack::LineSearchFullStep_Step)
    if alpha_k is not updated then
        alpha_k = 1.0
    end
    x_kp1 = x_k + alpha_k * d_k
    f_kp1 = f(x_kp1)
    c_kp1 = c(x_kp1)

9.-1. "MeritFunc_PenaltyParamUpdate"
    (class ReducedSpaceSQPPack::MeritFunc_PenaltyParamUpdateMultFree_AddedStep)
    *** Update the penalty parameter for the merit function to ensure
    *** a descent direction a directional derivatieve.
    *** phi is a merit function object that uses the penalty parameter mu.

    ...
```

```
9. "LineSearch"
    (class ReducedSpaceSQPPack::LineSearchFailureNewDecompositionSelection_Step)
    do line search step : class ReducedSpaceSQPPack::LineSearchDirect_Step
        *** Preform a line search along the full space search direction d_k.
        Dphi_k = merit_func_nlp_k.deriv()
        if Dphi_k >= 0 then
            throw line_search_failure
        end
        phi_kp1 = merit_func_nlp_k.value(f_kp1,c_kp1,h_kp1,hl,hu)
        phi_k = merit_func_nlp_k.value(f_k,c_k,h_k,hl,hu)
        begin direct line search (where phi = merit_func_nlp_k): "class ConstrainedOptimizationPack::DirectLineSearchArmQuad_Strategy"
            *** start line search using the Armijo cord test and quadratic interpolation of alpha

        ...

        end direct line search
        if maximum number of linesearch iterations are exceeded then
            throw line_search_failure
        end
    end line search step
    if thrown line_search_failure then
      if line search failed at the last iteration also then
        throw line_search_failure
      end
      new decomposition selection : class ReducedSpaceSQPPack::NewDecompositionSelectionStd_Strategy
        if k > max_iter then
          terminate the algorithm
        end
        Select a new basis at current point
        x_kp1 = x_k
        alpha_k = 0
        k=k+1
        goto EvalNewPoint
      end new decomposition selection
    end

10. "Major Loop" :
    if k >= max_iter then
        terminate the algorithm
    elseif run_time() >= max_run_time then
        terminate the algorithm
    else
        k = k + 1
        goto 1
    end


***************************************************************
Warning, the following options groups where not accessed.
An options group may not be accessed if it is not looked for
or if an "optional" options group was looked from and the user
spelled it incorrectly:
```

Output file `rSQPppSummary.out`

```
********************************************************************
*** Algorithm iteration summary output                           ***
***                                                              ***
*** Below, a summary table of the SQP iterations is given as     ***
*** well as a table of the CPU times for each step (if the       ***
*** option rSQPppSolver::algo_timing = true is set).             ***
********************************************************************

*** Echoing input options ...

...

*** Setting up to run rSQP++ on the NLP using a configuration object of type
    'class ReducedSpaceSQPPack::rSQPAlgo_ConfigMamaJama' ...
```

```
test_nlp = true: Testing the NLP!

Testing the supported NLPFirstOrderInfo interface ...

... end testing of nlp


********************************
*** Start of rSQP Iterations ***
n = 30400, m = 30000, nz = 599910

 k      f              ||Gf||inf      ||c||inf       ||rGL||inf     quasi-Newton ...
 ----   ------------   ------------   ------------   ------------   ------------ ...
    0      1.52e+006             10   1.20897e+007        1353.69    initialized ...
    1         713384        11.7743   1.10232e+007        524.977         skiped ...
...
   12   3.37179e-015   3.63585e-008   2.08817e-005   3.63585e-008        updated ...
 ----   ------------   ------------   ------------   ------------   ------------ ...
   13   6.34035e-024   3.27386e-012   1.51859e-012   3.27386e-012              - ...

Number of function evaluations:
-------------------------------
f(x)  : 96
c(x)  : 96
Gf(x) : 15
Gc(x) : 15


************************
**** Solution Found ****

  total time = 61.9129 sec.


**************************************
*** Algorithm step CPU times (sec) ***

Step names
----------
1) "EvalNewPoint"
2) "RangeSpaceStep"
3) "ReducedGradient"
4) "CalcReducedGradLagrangian"
5) "CheckConvergence"
6) "ReducedHessian"
7) "NullSpaceStep"
8) "CalcDFromYPYZPZ"
9) "LineSearch"
10) Iteration total

            steps 1...10 ->

    iter k         1         2         3         4         5         6         7         8         9        10
  --------  --------  --------  --------  --------  --------  --------  --------  --------  --------  --------
         0     18.96    0.2031    0.1093 0.0001131   0.01497    0.2985     2.189  0.009146    0.2678     22.06
         1     2.398    0.1752    0.11248.409e-005  0.002709  0.002098    0.1186  0.006983     0.294     3.111
         2     2.399    0.1757    0.1116 7.99e-005  0.002728  0.002192    0.1183  0.007003   0.04079     2.857
         3     2.421     0.175    0.11348.297e-005  0.002717  0.002115    0.1183  0.006929   0.04122     2.881
         4     2.428     0.172    0.1108 7.99e-005  0.002711   0.02247    0.1181  0.006949     0.041     2.902
         5     2.404    0.1748    0.1115 8.13e-005  0.002742  0.002087    0.1183  0.006936   0.07081     2.892
         6     2.443    0.1714    0.1094 7.99e-005  0.002707  0.002139    0.1156  0.006912    0.0409     2.892
         7     2.397    0.1747    0.1115 7.99e-005  0.002715   0.00211    0.1184  0.006978   0.04138     2.855
         8     2.415    0.1749    0.11158.046e-005  0.002724  0.002148     0.141  0.007056   0.04127     2.896
         9     2.403    0.1752    0.11158.102e-005  0.002751    0.3873    0.1167  0.006928    0.0412     3.244
        10      2.42    0.1715    0.10928.185e-005  0.002711  0.002126    0.1159  0.006915   0.04102     2.869
        11     2.416    0.1747    0.11148.269e-005  0.002704   0.02088     0.118  0.006951   0.04134     2.892
        12     2.402    0.1753    0.11168.018e-005   0.02678   0.02145    0.1156  0.006994   0.04093       2.9
        13     2.404    0.1749    0.11158.185e-005  0.008906         0         0         0         0     2.699
  --------  --------  --------  --------  --------  --------  --------  --------  --------  --------  --------
total(sec)     50.32     2.468     1.557  0.001169   0.08058    0.7677     3.622   0.09268     1.044     59.95
 av(sec)/k     3.594    0.1763    0.11128.351e-005  0.005756   0.05483    0.2587   0.00662   0.07455     4.282
  min(sec)     2.397    0.1714    0.1092 7.99e-005  0.002704         0         0         0         0     2.699
  max(sec)     18.96    0.2031    0.1134 0.0001131   0.02678    0.3873     2.189  0.009146     0.294     22.06
   % total     83.93     4.118     2.596   0.00195    0.1344     1.281     6.041    0.1546     1.741       100
------------------------------
total CPU time = 59.95 sec
```

```
...
```

Output file `rSQPppJournal.out`

```
********************************************************************
*** Algorithm iteration detailed journal output                  ***
***                                                              ***
*** Below, detailed information about the SQP algorithm is given ***
*** while it is running.  The amount of information that is      ***
*** produced can be specified using the option                   ***
*** rSQPSolverClientInterface::journal_output_level (the default ***
*** is PRINT_NOTHING and produces no output)                     ***
********************************************************************

*** Echoing input options ...

...

*** Setting up to run rSQP++ on the NLP using a configuration object of type
    'class ReducedSpaceSQPPack::rSQPAlgo_ConfigMamaJama' ...

test_nlp = true: Testing the NLP!

Testing the supported NLPFirstOrderInfo interface ...

**************************************
*** test_nlp_first_order_info(...) ***
**************************************

Testing the vector spaces ...

Testing nlp->space_x() ...
nlp->space_x() checks out!

Testing nlp->space_c() ...
nlp->space_c() checks out!

**************************************
*** NLPTester::test_interface(...) ***
**************************************

nlp->force_xinit_in_bounds(true)
nlp->initialize(true)

*** Dimensions of the NLP
nlp->n()  = 30400
nlp->m()  = 30000
nlp->mI() = 0

*** Validate the dimensions of the vector spaces
check: nlp->space_x()->dim() = 30400 == nlp->n() = 30400: true
check: nlp->space_c()->dim() = 30000 == nlp->m() = 30000: true
check: nlp->space_h().get() = 00000000 == NULL: true
||nlp->xinit()||inf = 1.00000000e+001

*** Validate that the initial starting point is in bounds ...

check: xl <= x <= xu : true
xinit is in bounds with { max |u| | xl <= x + u <= xu } -> -1.00000000e+050

check: num_bounded(nlp->xl(),nlp->xu()) = 0 == nlp->num_bounded_x() = 0: true

Getting the initial estimates for the Lagrange mutipliers ...

||lambda||inf  = 0.00000000e+000

*** Evaluate the point xo ...

f(xo) = 1.52000000e+006
||c(xo)||inf = 1.20897308e+007
```

```
*** Report this point to the NLP as suboptimal ...

*** Print the number of evaluations ...

nlp->num_f_evals() = 1
nlp->num_c_evals() = 1

Calling nlp->calc_Gc(...) at nlp->xinit() ...

Calling nlp->calc_Gf(...) at nlp->xinit() ...

Comparing products Gf'*y Gc'*y and/or Gh'*y with finite difference values  FDGf'*y, FDGc'*y and/or FDGh'*y for random y's ...

****
**** Random directional vector 1 ( ||y||_1 / n = 5.00741357e-001 )
***

rel_err(Gf'*y,FDGf'*y) = rel_err(6.53040559e+002,6.53040559e+002) = 1.93477565e-011

rel_err(sum(Gc'*y),sum(FDGc'*y)) = rel_err(2.20905038e+008,2.20905038e+008) = 1.37878129e-013

Congradulations!  All of the computed errors were within the specified error tolerance!

... end testing of nlp

************************************
*** rSQPppSolver::solve_nlp()    ***
************************************

*** Starting rSQP iterations ...


(0) 1: "EvalNewPoint"

x is not updated for any k so set x_k = nlp.xinit() ...

||x_k||inf = 1.000000e+001

Updating the decomposition ...

...

Printing the updated iteration quantities ...

f_k         = 1.520000e+006
||Gf_k||inf = 1.000000e+001
||c_k||inf  = 1.208973e+007

*** Checking derivatives by finite differences

Comparing products Gf'*y and/or Gc'*y  with finite-difference values  FDGf'*y and/or FDGc'*y for random y's ...

****
**** Random directional vector 1 ( ||y||_1 / n = 4.995094e-001 )
***

rel_err(Gf'*y,FDGf'*y) = rel_err(1.959355e+002,1.959355e+002) = 4.408797e-010

rel_err(sum(Gc'*y),sum(FDGc'*y)) = rel_err(4.737147e+008,4.737147e+008) = 5.088320e-013

For Gf, there were 1 warning tolerance violations out of num_fd_directions = 1 computations of FDGf'*y
and the maximum violation was 4.408797e-010 > Gf_waring_tol = 1.000000e-010

Congradulations!  All of the computed errors were within the specified error tolerance!

(0) 2: "RangeSpaceStep"

||py||   = 1.000000e+001

||Ypy||2 = 1.732051e+003

(0) 2.1: "CheckDecompositionFromPy"

beta = ||py||/||c|| = 8.271483e-007
```

```
(0) 2.2: "CheckDecompositionFromRPy"

beta = ||(Gc(decomp)'*Y)*py_k + c_k(decomp)||inf / (||c_k(decomp)||inf + small_number)
     = 5.587935e-009 / (1.208973e+007 + 2.225074e-308)
     = 4.622051e-016

(0) 2.3: "CheckDescentRangeSpaceStep"

Gc_k exists; compute descent_c = c_k(equ_decomp)'*Gc_k(:,equ_decomp)'*Ypy_k ...

descent_c = -4.369965e+018

(0) 3: "ReducedGradient"

||rGf||inf = 1.353686e+003

(0) 4: "CalcReducedGradLagrangian"

||rGL_k||inf = 1.353686e+003

(0) 5: "CheckConvergence"

scale_opt_factor = 1.000000e+000 (scale_opt_error_by = SCALE_BY_ONE)
scale_feas_factor = 1.000000e+000 (scale_feas_error_by = SCALE_BY_ONE)
scale_comp_factor = 1.000000e+000 (scale_comp_error_by = SCALE_BY_ONE)
opt_scale_factor = 1.100000e+001 (scale_opt_error_by_Gf = true)
opt_kkt_err_k    = 1.230623e+002 > opt_tol  = 1.000000e-008
feas_kkt_err_k   = 1.208973e+007 > feas_tol = 1.000000e-010
comp_kkt_err_k   = 0.000000e+000 < comp_tol = 1.000000e-006
step_err         = 0.000000e+000 < step_tol = 1.000000e-002

Have not found the solution yet, have to keep going :-(

(0) 6.-1: "CheckSkipBFGSUpdate"

(0) 6: "ReducedHessian"

Basis changed.  Reinitializing rHL_k = eye(n-r) ...

(0) 7: "NullSpaceStep"

||pz_k||inf   = 1.353686e+003
||Zpz_k||2    = 4.271437e+005

(0) 8: "CalcDFromYPYZPZ"

(Ypy_k'*Zpz_k)/(||Ypy_k||2 * ||Zpz_k||2 + eps) = 9.979894e-001
||d||inf = 2.471247e+003

(0) 9.-2: "LineSearchFullStep"

f_k        = 1.520000e+006
||c_k||inf = 1.208973e+007
alpha_k    = 1.000000e+000

||x_kp1||inf   = 2.461247e+003

f_kp1        = 9.123131e+010
||c_kp1||inf = 4.579853e+013

(0) 9.-1: "MeritFunc_PenaltyParamUpdate"

Update the penalty parameter...

Not near solution, allowing reduction in mu ...

mu = 8.286385e-006

(0) 9: "LineSearch"

Begin definition of NLP merit function phi.value(f(x),c(x)):
    *** Define L1 merit funciton (assumes Gc_k'*d_k + c_k = 0):
    phi(f,c) = f + mu_k * norm(c,1)
    Dphi(x_k,d_k) = Gf_k' * d_k - mu * norm(c_k,1)
end definition of the NLP merit funciton
```

```
Dphi_k = -7.389329e+008

Starting Armijo Quadratic interpolation linesearch ...

Dphi_k = -7.38932862e+008
phi_k = 4.52030000e+006

  itr            alpha_k            phi_kp1   phi_kp1-frac_phi
 ----    ----------------    ----------------   ----------------
    0     1.00000000e+000     1.14557884e+013     1.14557839e+013
    1     1.00000000e-001     1.34428112e+010     1.34382983e+010
    2     1.00000000e-002     2.53185818e+007     2.07990207e+007
    3     1.31074052e-003     3.44902601e+006    -1.07117714e+006

alpha_k                = 1.310741e-003
||x_kp1||inf           = 1.177433e+001
f_kp1                  = 7.133842e+005
||c_kp1||inf           = 1.102321e+007
phi_kp1                = 3.449026e+006

(1) 1: "EvalNewPoint"

...

(13) 5: "CheckConvergence"

scale_opt_factor = 1.000000e+000 (scale_opt_error_by = SCALE_BY_ONE)
scale_feas_factor = 1.000000e+000 (scale_feas_error_by = SCALE_BY_ONE)
scale_comp_factor = 1.000000e+000 (scale_comp_error_by = SCALE_BY_ONE)
opt_scale_factor = 1.000000e+000 (scale_opt_error_by_Gf = true)
opt_kkt_err_k    = 3.273859e-012 < opt_tol  = 1.000000e-008
feas_kkt_err_k   = 1.518593e-012 < feas_tol = 1.000000e-010
comp_kkt_err_k   = 0.000000e+000 < comp_tol = 1.000000e-006
step_err         = 0.000000e+000 < step_tol = 1.000000e-002

Jackpot!  Found the solution!!!!!! (k = 13)
```

E A Simple Convention for the Specification of Linear-Algebra Function Prototypes in C++ using Vector and Matrix Objects

A simple convention for the specification of C++ function prototypes for linear algebra operations with vectors and matrices is described. This convention leads to function prototypes that are derived directly from the mathematical expressions themselves (and are therefore easy to remember), allow for highly optimized implementations (through inlining in C++), and do not rely on any sophisticated C++ techniques so that even novice C++ programs can understand and debug through the code.

Introduction

Linear algebra computations such as matrix-vector multiplication and the solution of linear systems serve as the building blocks for numerical algorithms and consume the majority of the runtime of numerical codes. These linear algebra abstractions transcend details such as matrix storage formats (of which there are many) and linear system solver codes (sparse or dense, direct or iterative). Primary linear algebra abstractions include vectors and matrices and the operations that can be performed with them. C++ abstractions for vectors and matrices abound.

Given that convenient vector and matrix abstractions are defined, `Vec` and `Mat` for instance, there is a need to implement BLAS-like linear algebra operations. Given that C++ has operator overloading, it would seem reasonable to implement these operations using a Matlab© like notation. For example, the matrix-vector multiplication $y = y + A^T x$ might be represented in C++ with the statement `y = y + trans(A) * x` (the character `'` can not be used for transpose since it is not a C++ operator). Matlab is seen by many in the numerical computational community to be the ideal for the representation of linear algebra operations using only ASCII characters [33]. The advantages of such an interface are obvious. It is almost the same as standard mathematical notation, which makes it very easy to match the implementation with the operation for the application programmer, and makes the code much easier to understand. The primary disadvantage for this in C++ is that the straightforward implementation requires a lot of overhead because operators are implemented in a binary fashion. For example, for the operation `y = y + trans(A) * x`, a temporary matrix (n^2 overhead) and two temporary vectors ($2n$ overhead) would be created by the compiler. Specifically, the compiler would perform the following operations: `Mat t1 = trans(A); Vec t2 = t1 * x; Vec t3 = y + t2; y = t3;`. Attempts have been made to come up with a strategy in C++ to implement operations like `y = y + trans(A) *` `x` in a way where little overhead is required beyond a direct BLAS call [87]. It is relatively easy to implement these operator functions with only a little constant-time overhead for a small set of linear algebra operations [112, pages 675-677]. However, for more elaborate expressions, a compile time expression parsing method is needed. Some have advocated preprocessing tools, while others have looked at using C++'s template mechanisms [119], [87]. In any case, these methods are complex and not trivial to implement. Also, compilers are very fickle with respect to methods that rely on templates. Perhaps in the future when many C++ compilers implement the ANSI/ISO C++ standard [112], such methods may be more portable and reliable. But for now, such methods are not really appropriate for general application development. Methods based on runtime parsing are also possible but add more of a runtime penalty. Aliasing is also another big problem. For example, suppose we allow users to write expressions like the following:

$$y = x + v + \alpha M^T + \beta y$$

An efficient parser that tries to minimize temporaries will have to scan the entire expression and realize that $y = \beta y$ must be performed first and then no temporaries are needed. A naive parser may perform $y = x$ first and then result in an incorrect evaluation. The problem is that the more efficient the parser the more complicated it is and the harder it will be for inexperienced users to debug through this code.

Without using operator overloading to allow application code to use syntax like `y = y + trans(A) * x`, how can linear algebra operations be implemented efficiently? The simple answer is to use regular functions (member or non-member) inlined to call the BLAS. For example, for the operation $y = y + A^T x$, one might provide a function like `add_to_multiply_transpose(A,x,&y);`. It is trivial to implement such a function to call the BLAS with no overhead if a good inlining C++ compiler is used. The problem with using functions is that it is difficult to come up with good names that users can remember. For example, the above operation has been called `Blas_Mat_Vec_Mult(...)` in LAPACK++ [91], `vm_multadd(...)` in Meschach++ [95], and `mult(...)` in MTL [73]. Even knowing the names of these functions is not enough. You must also know the order the arguments go in and how are they passed.

Convention for specifying function prototypes

Here we consider a convention for constructing C++ function prototypes. The function prototypes are constructed according to this convention where the name of the function and the order of the arguments is easily composed from the mathematical expression itself. To illustrate the convention,

Function Call

$$\begin{array}{ccccccccccc} \texttt{y} & \texttt{+=} & \texttt{alpha} & \texttt{*} & \texttt{A'} & \texttt{*} & \texttt{x} & y & \alpha & A^T & x \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \texttt{V} & \overbrace{\texttt{p\_}} & \texttt{S} & \texttt{t} & \texttt{M} & \texttt{t} & \texttt{V (} & \texttt{\&y,} & \texttt{alpha,} & \overbrace{\texttt{A, trans,}} & \texttt{x)} \end{array} \Longrightarrow \texttt{Vp\_StMtV(\&y,alpha,A,trans,x)}$$

Function Prototype

```
void Vp_StMtV( Vec* vs_lhs, double alpha
  , const Mat& gms_rhs1, BLAS_Cpp::Transp trans_rhs1
  , const Vec& vs_rhs2);
```

Figure E.1. Example of the linear algebra naming convention for $y+=\alpha A^T x$

consider the operation $y = y + \alpha A^T x$. First, rewrite the operation in the form $y+=\alpha A^T x$ (this is well understood by C, C++ and Perl programmers). Next, translate into Matlab-like notation as `y += alpha*A'*x` (except Matlab does not have the operator `+=`). Finally, for `Vec` objects `y` and `x` and a `Mat` object `A`, the function call and its prototype are shown in Figure E.1. The type `BLAS_Cpp::Transp` shown in this function prototype is a simple C++ `enum` with the values `BLAS_Cpp::trans` or `BLAS_Cpp::no_trans`.

Figure E.2 gives a summary of this convention. Given this convention, it is easy to go back and forth between the mathematical notation and the function prototype. For example, consider the following function call and its mathematical expression:

`Mp_StMtM( &C, alpha, A, no_trans, B, trans )`

$\Longrightarrow$

$C+=\alpha A B^T$

One difficulty with this convention is dealing with Level-2 and Level-3 BLAS that have expressions such as:

$$C = \alpha \,\text{op}(A)\, \text{op}(B) + \underbrace{\beta}_{?} C \qquad \text{(xGEMM)}$$

Given $\beta \neq 1$ we can not simply rewrite the above BLAS operation using +=. To deal with this problem, β is moved to the end of the argument list and has a default value of 1.0 as shown below:

$$\texttt{Mp\_StMtM(\&C, alpha, A, trans\_A, B, trans\_B ,} \quad \underbrace{\texttt{beta}}_{\texttt{default to 1.0}} \quad \texttt{)}$$

Operation	**Character (Lower Case)**	
`=`(assignment,equals)	_(underscore)	
`+=`(plus equals)	`p_`	
`+`(addition,plus)	`p`	
`-`(subtraction,minus)	`m`	
`*`(multiplication,times)	`t`	
Operand Type	**Character (Upper Case)**	**Argument(s)**
Scalar	`S`	`double`
Vector	`V`	(rhs) `const Vec&`
		(lhs) `Vec*`
Matrix	`M`	(rhs) `const Mat&, Transp`
		(lhs) `Mat*`

Figure E.2. Naming convention for linear algebra functions in C++

Only exact equivalents to the Level-2 and Level-3 BLAS need be explicitly implemented (i.e. `Vp_StMtV(...)` and `Mp_StMtM(...)`). Functions for simpler expressions can be generated automatically using template functions. As an example, consider the following linear algebra operation and its function call:

$$y = Ax \qquad (\text{xGEMV} \rightarrow y = \alpha \operatorname{op}(A)x + \beta y)$$

$$\Longrightarrow$$

`V_MtV( &y, A, no_trans, x )`

In the above example, the template function `V_MtV(...)` can be inlined to call `Vp_StMtV(...)` which in turn can be inlined to call the BLAS function `DGEMV(...)`. The use of these automatically generated functions makes the application code more readable and also allows for specialization of these simpler operations later if desired. The implementation of the above template function `V_MtV(...)` is trivial and is given below:

```
template<class M_t, class V_t>
inline void V_MtV(V_t* y, const M_t& A, BLAS_Cpp::Transp trans_A, const V_t& x)
{
    Vp_StMtV( y, 1.0, A, no_trans, x, 0.0 );
}
```

Longer expressions such as $y = \alpha A^T x + Bz$ are easily handled using multiple function calls such as:

$y = \alpha A^T x + Bz$
$\Longrightarrow$
```
V_StMtV( &y, alpha, A, trans, x );
Vp_MtV( &y, B, no_trans, z );
```

As stated above, only the base BLAS operations `Vp_StMtV(...)` (e.g. `xGEMV(...)`) and `Mp_StMtM(...)` (e.g. `xGEMM(...)`) must be implemented for the specific vector and matrix types `Vec` and `Mat`. For example, if these are simple encapulations of BLAS compatible serial vectors and matrices (e.g. TNT style) then the call the the BLAS functions can be written as template functions for all serial dense vector and matrix (column oriented) classes. For example:

```
template<class M_t, class V_t>
inline void Vp_StMtV( V_t* y, double alpha, const M_t& A, BLAS_Cpp::Transp trans_A
    , const V_t& x, double beta = 1.0 )
{
    DGEMV( trans_A == no_trans ? ’N’ : ’T’, rows(A), cols(A), alpha
        ,&A(1,1), &A(1,2) - &A(1,1), &x(1), &x(2) - &x(1), beta
        ,&(*y)(1), &(*y)(2) - &(*y)(1) );
}
```

Of course the above function would also have to handle the cases where `rows(A)` and/or `cols(A)` was 1 but the basic idea should be clear. By calling `rows(...)` and `cols(...)` as nonmember functions, they can be overloaded to call the appropriate member functions on the matrix object since there is not standard.

When `Vec` and `Mat` are polymorphic types we can use a trick to implement `Vp_StMtV(...)` and `Mp_StMtM(...)` using member functions. For example:

```
class Vec { ... }
...
class Mat {
public:
    virtual void Vp_StMtV( V_t* y, double alpha, BLAS_Cpp::Transp trans_A
        , const V_t& x, double beta ) const = 0;
    ...
};
...
inline void Vp_StMtV( Vec* y, double alpha, const Mat& A, BLAS_Cpp::Transp trans_A
    , const Vec& x, double beta = 1.0 )
{
    A.Vp_StMtV(y,alpha,trans_A,x,beta);
}
```

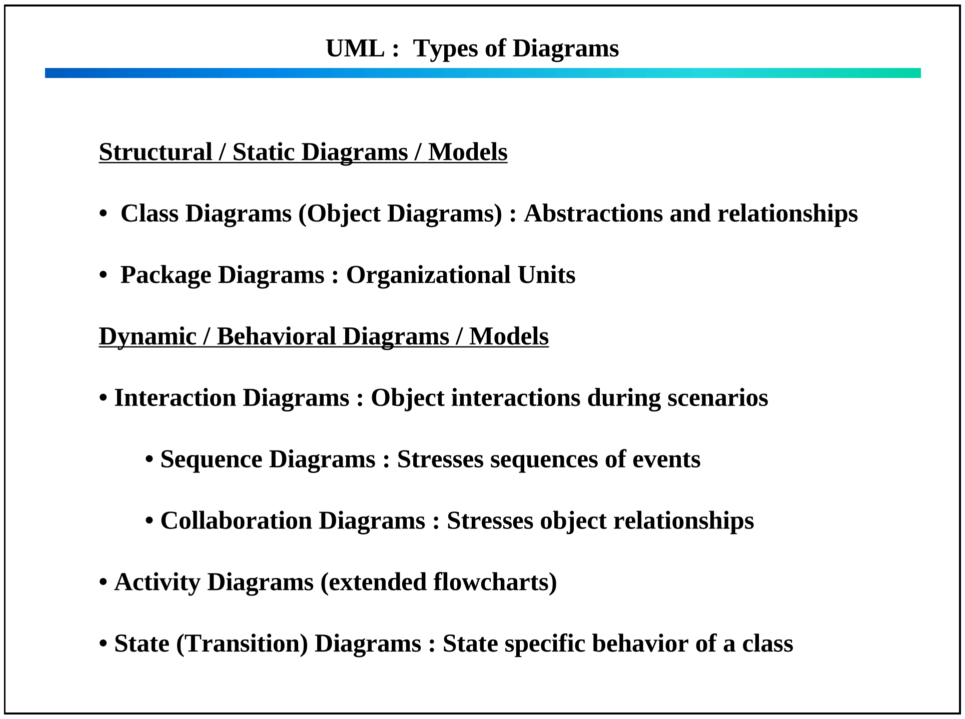
Using these inlined non-member functions there is no extra overhead beyond the inavoidable virtual function calls. In this way there is consistent calling of linear algebra operations irregardless whether the vector and matrix objects are concrete or abstract.

Conclusions

In summary, this convention makes it easy to write out correct calls to linear algebra operations without having to resort to complex operator overloading techniques. After all, the main appeal for operator overloading is to make it easy for users to remember how the call linear algebra operations and to make written code easier to read. The convention described in this paper meets both of these goals and also results in code that is easy for novice C++ developers to understand and debug. Debugging code can easily take longer than writing it in the first place. When concrete abstractions of dense linear algebra types are used, it was shown that these functions do not have to impose any overhead beyond direct BLAS calls if inlining is used. When polymorphic vector and matrix types are used, inlining to call the virtual functions also results in no extra overhead.

F Unified Modeling Language (UML) Quick Reference Guide

The Unified Modeling Language (UML) is a newly standardized graphical language for Object-Oriented modeling (http://www.omg.org).



UML : Types of Diagrams

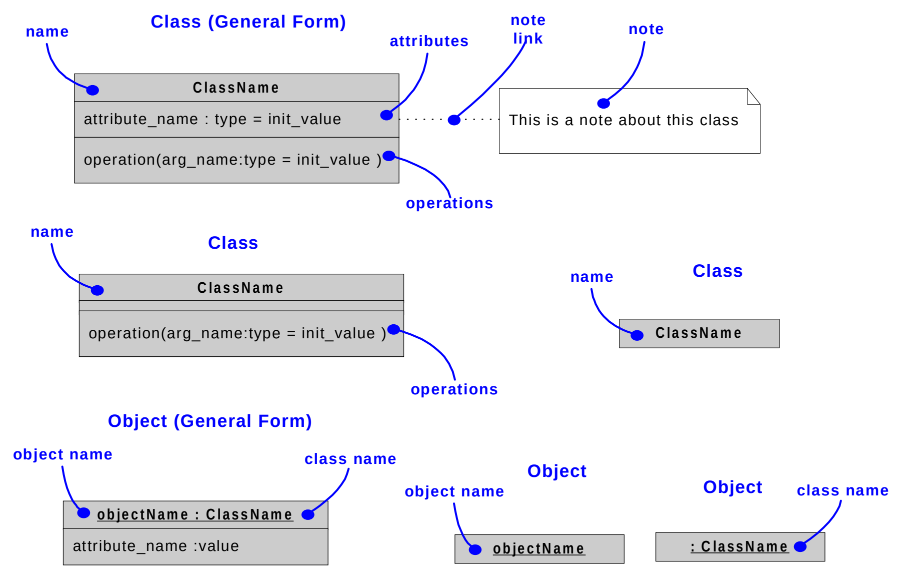
Structural / Static Diagrams / Models

- **Class Diagrams (Object Diagrams) : Abstractions and relationships**
- **Package Diagrams : Organizational Units**

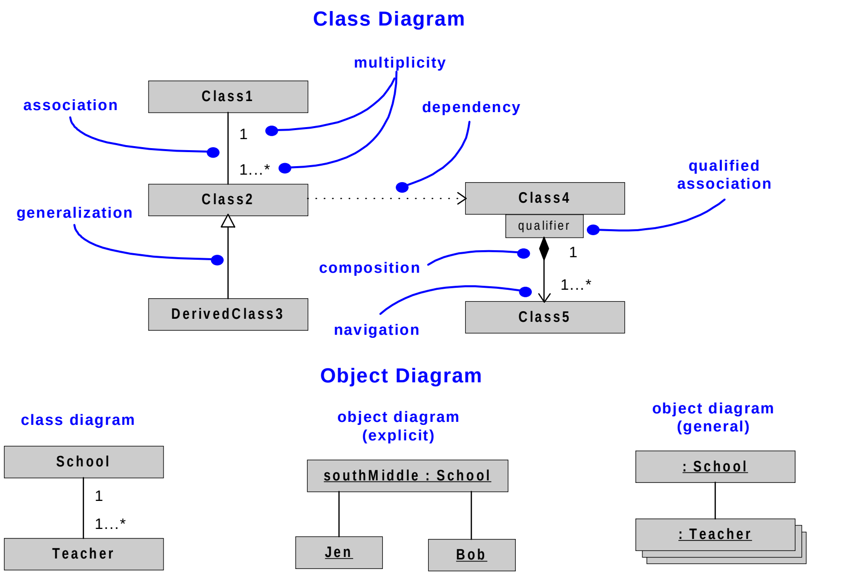
Dynamic / Behavioral Diagrams / Models

- **Interaction Diagrams : Object interactions during scenarios**
  - **Sequence Diagrams : Stresses sequences of events**
  - **Collaboration Diagrams : Stresses object relationships**
- **Activity Diagrams (extended flowcharts)**
- **State (Transition) Diagrams : State specific behavior of a class**

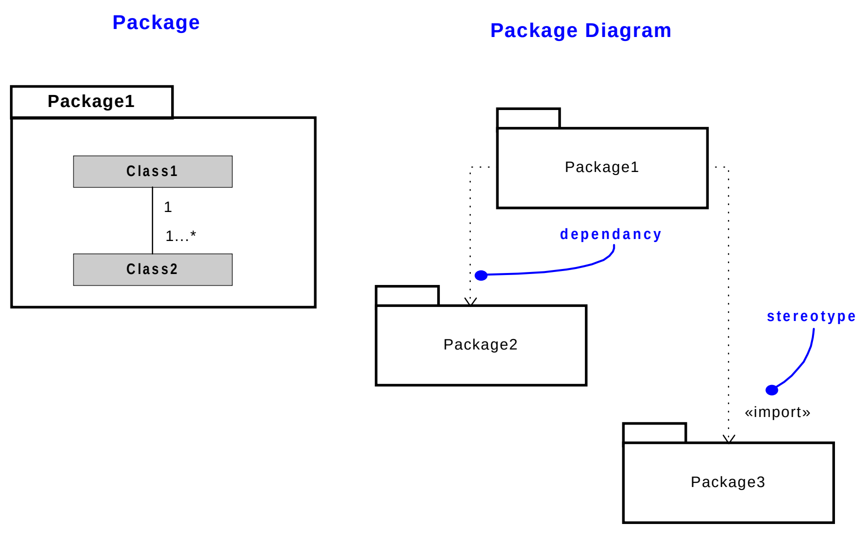
UML Structural Entities : Classes and Objects



UML Structural Diagrams : Class & Object with Relationships

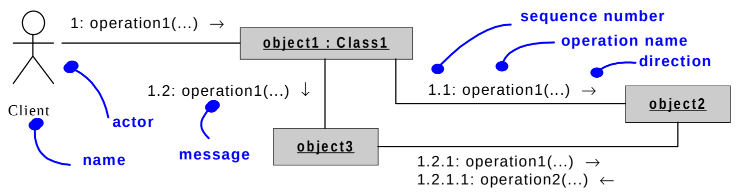


UML Structural Diagrams : Packages Diagrams

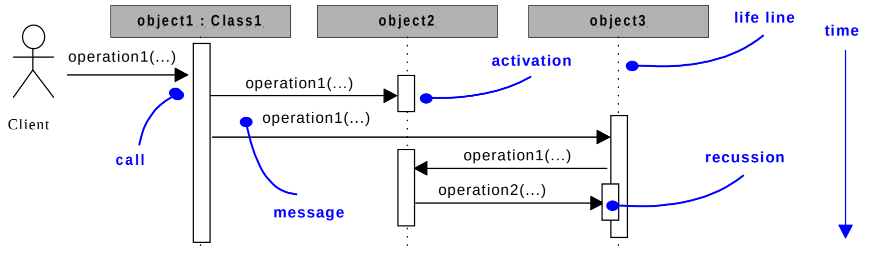


UML Dynamic Diagrams : Interaction Diagrams

Collaboration Diagram



Sequence Diagram



DISTRIBUTION:

1 Omar Ghattas
Carnegie Mellon University 5000 Forbes Ave.Porter Hall, Pittsburgh, PA 15213

1 Larry Biegler
Carnegie Mellon University 5000 Forbes Ave., Pittsburgh, PA 15213

1 George Biros
251 mercer street Courant Institute of Mathematical Sciences New York University New York, NY 10012

1 Matthias Heinkenschloss
Department of Computational and Applied Mathematics - MS 134 Rice University 6100 S Main Street Houston, TX 77005 - 1892

1 Jon Tolle
CB 3180 Smith Building, University of North Carolina, Chapel Hill, NC 27599-3180

1 Tony Kearsley
Mathematical and Computational Science Division National Institute of Standards and Technology Room 375 - North 820 Quince Orchard Road Gaithersburg, MD 20899-0001

1 Roger Ghanem
Dept. of Civil Engineering, John Hopkins University, Baltimore, MD 21218

1 MS 0321
Bill Camp, 9200

1 MS 9003
Kenneth Washington, 8900

1 MS 1110
David Womble, 9214

1 MS 0316
Sudip Dosanjh, 9233

1 MS 9217
Steve Thomas, 8950

1 MS 9217
Paul Boggs, 8950

1 MS 9217
Kevin Long, 8950

1 MS 9217
Patricia Hough, 8950

1 MS 9217
Tamara Kolda, 8950

1 MS 9217
Monica Martinez-Canales, 8950

1 MS 9217
Pamela Williams, 8950

1 MS 0847
Scott Mitchell, 92a11

5 MS 0847
Bart van Bloemen Waanders, 9211

1 MS 1110
Roscoe Bartlett, 9211

1 MS 1110
Andrew Salinger, 9233

1 MS 1110
Roger Pawlowski, 9233

1 MS 1110
John Shadid, 9233

1 MS 0847
Mike Eldred, 9211

1 MS 0819
Tim Trucano, 9211

1 MS 0847
Tony Giunta, 9211

1 MS 1110
Bill Hart, 9214

1 MS 1110
Cindy Phillips, 9214

1 MS 0847
Steven Wojtkiewicz, 9124

1 MS 1110
Mike Heroux, 9214

1 MS 1110
Rich Lehoucq, 9214

1 MS 0316
Curt Ober, 9233

1 MS 0316
Tom Smith, 9233

1 MS 0316
Eric Keiter, 9233

1 MS 0316
Scott Hutchinson, 9233

1 MS 1110
Martin Berggren, 9214

1 MS 0750
Greg Newman, 6116

1 MS 0188
Donna Chavez, 1011

1 MS 0188
LDRD Office, 1011

1 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616

1 MS 0612
Review & Approval Desk, 9612