rSQP++ : An Object-Oriented Framework for Successive Quadratic Programming

Roscoe A. Bartlett[1] and Lorenz T. Biegler[1]

Department of Chemical Engineering, Carnegie Mellon University

Abstract. An Object-Oriented (OO) framework called rSQP++ is currently being developed for Successive Quadratic Programming (SQP). It is designed to support many different SQP algorithms and to allow for external configuration of specialized application specific linear algebra objects such as matrices and linear solvers. In addition, it is possible for a client to modify the SQP algorithms to meet other specialized needs without having to touch any of the source code within the rSQP++ framework or even having to recompile existing SQP algorithms. Much of this is accomplished through a set of carefully constructed interfaces to various linear algebra objects such as matrices and linear solvers. The initial development of rSQP++ was done in a serial environment and therefore issues related to the use of massively parallel iterative solvers used in PDE constrained optimization have not yet been addressed. In order to more effectively support parallelism, rSQP++ needs the addition and integration of an abstract vector interface to allow more flexibility in vector implementations. Encapsulating vectors away from algorithmic code would allow fully parallel linear algebra, but could also greatly restrict the kinds of operations that need to be performed. The difficulty in developing an abstract vector interface and a proposed design for a remedy are discussed.

1 Introduction

Nonlinear Programming (NLP) is an important tool in many areas of engineering and design. Application areas where large scale NLPs arise include DAE and PDE constrained optimization as well as many others. A standard form for an NLP is

$$\min \; f(x) \tag{1}$$

$$\text{s.t.} \; c(x) = 0 \tag{2}$$

$$x^L \le x \le x^U \tag{3}$$

$$\text{where: } x, x^L, x^U \in \mathbb{R}^n,\; f(x) \in \mathbb{R}^n \to \mathbb{R},\; c(x) \in \mathbb{R}^n \to \mathbb{R}^m$$

A new object-oriented (OO) framework for building Successive Quadratic Programming (SQP) algorithms, called rSQP++, is currently being implemented in C++. The goals for rSQP++ are quite lofty. The rSQP++ framework is being designed to incorporate many different SQP algorithms and to allow external configuration of specialized linear algebra objects such as matrices and linear solvers. Data-structure independence has been recognized as an important feature missing in current optimization software [12]. In addition, it is possible for the client to modify the SQP algorithms to meet other specialized needs without having to touch any of the source code within the rSQP++ framework.

SQP methods are attractive because they generally require fewer iterations and function and gradient evaluations to solve a problem compared to other methods. Another attractive property of SQP methods is that they can be adapted to exploit the structure of an NLP. A variation of SQP, known as Reduced Space SQP (rSQP), works well for NLPs where there are few degrees of freedom and many constraints. 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 3.1).

2 Motivation for developing specialized NLP solvers and next generation optimization software

Much research has been, and continues to be, devoted to finding efficient methods to solve large-scale NLPs. Several methods have been proposed with successful implementations: Generalized Reduced Gradient (GRG) (CONOPT), Successive Linearly Constrained (SLC) (MINOS), Sequential Augmented Lagrangian (SAL) (LANCELOT), and Successive Quadratic Programming (SQP) (SNOPT , SOCS) [12]. More recently, Interior Point (IP) methods have been investigated and show promise (e.g. NITRO, LOQO). Many of these methods and even some implementations have been studied and refined for more than a decade. Many users rely on them through the use of modeling environments like GAMS and AMPL and these solvers are also embedded into many other applications.

With so many different well refined implementations for NLP solvers available, why would anyone want to develop their own? It seems that the overriding motivation for developing specialized optimization software is a desire to better exploit the properties of specialized classes of NLPs. For the most part, all of the aforementioned NLP solvers can only exploit the general sparsity of an NLP and use direct linear solvers that require explicit nonzero entries from the Jacobian and Hessian matrices. These solvers generally can not take advantage of opportunities for problem specific specialized linear algebra or possibly even parallelism. One example of such an application area is DAE constrained optimization of chemical processes. In this type of optimization problem the model, consisting of Differential and Algebraic Equations (DAEs), can be discretized over the time domain using orthogonal collocation on finite elements [4]. The Jacobian of the discretized equations has a special structure that can be exploited to significantly reduce the runtime and storage compared to general solvers (see Section 5). Another example of a specialized application area is PDE constrained optimization using the Finite Element Method (FEM). Large-scale simulation codes have been developed for solving nonlinear PDEs with millions (even billions) of discretized state variables. This has been made possible due to advances in massively parallel iterative linear solvers and multi-processor computers [11]. Most current optimization software can not take advantage of this work.

While the desire to better exploit the properties of specialized NLPs is clear, in practice general NLP solvers are often used instead. The reason for this is simple. These general NLP solvers have been made very reliable due to a lot of development work, testing and refinement on many different optimization problems. The Herculean task of implementing a reliable specialized NLP solver is usually not worth the effort. There are many reasons why developing an NLP solver is so difficult. While implementing the basic computations for generating search directions, in SQP for instance, may be fairly straightforward in some simplified cases, more elaborate means are generally necessary. In addition, sophisticated globalization strategies are needed to achieve convergence from remote starting points. Also, as with any numerical code, special attention needs to be paid to finite precision floating point errors that are amplified by ill conditioning. Another point is that every refined NLP implementation includes many potentially complex heuristics to deal with exception conditions. Even for a conceptually simple algorithm, the tasks of validating and debugging code are terribly difficult.

What is required is a different approach to implementing the next generation of optimization software. Of paramount importance is the need to allow a sophisticated user to specialize the algorithm to exploit “their” special class of NLPs. This might include problem specific data structures, linear solvers and even opportunities for parallelism. This might also include altering the logic of the algorithm. But of course for this software to be useful it needs to include those elements of an NLP solver that are, for the most part, independent of a specific class of NLPs and are so hard to implement well. These include such things as problem independent QP solvers, globalization methods (e.g. line searches, trust regions etc.), quasi-Newton methods as well as many other refinements (e.g. heuristics). Also, if users are going to be allowed to specialize components in the algorithm, the optimization software needs to include optional, but yet relatively efficient, run-time tests to validate the various computations. This includes validating functions and gradients, linear solvers and every other major computation that has clear well defined post conditions. Trying to determine why an algorithm fails to find a desired solution without such built-in testing and validation is an extremely difficult task even for an expert.

3 Successive Quadratic Programming (SQP)

Successive Quadratic Programming (SQP) has proven to be an attractive framework for the development of specialized NLP algorithms. The main feature of an SQP algorithm is the solution of a Quadratic Programming (QP) subproblem at each iteration such as:

$$\min \; g^T d + \tfrac{1}{2} d^T W d \tag{4}$$

$$\text{s.t.} \; A^T d + c = 0 \tag{5}$$

$$x^L - x^k \le d \le x^U - x^k \tag{6}$$

where: $d = x^{k+1} - x^k \in \mathbb{R}^n$ $\quad g = \nabla f(x^k) \in \mathbb{R}^n$
$W = \nabla_x^2 L(x^k, \lambda^k) \in \mathbb{R}^{n \times n}$ $\quad A = \nabla c(x^k) \in \mathbb{R}^{n \times m}$

Globalization methods are used to insure (given a few assumptions are satisfied) the SQP algorithm will converge to a local solution from remote starting points. The major classes of globalization methods include line search and trust region [6]. Because SQP is similar to applying Newton's method to the KKT optimality conditions, it can be shown to be quadratically convergent near the solution of the NLP [5]. One difficulty, however, is that in order to achieve quadratic convergence the exact Hessian of the Lagrangian W is needed which requires exact second order information for $\nabla^2 f(x)$ and $\nabla^2 c_j(x)$, $j = 1 \ldots m$. For many NLP applications this information is not readily available. Also, for large problems (4)–(6) can be extremely expensive, or nearly impossible, to solve directly. These and other difficulties with SQP have motivated the research of large-scale decomposition methods for SQP. One class of these methods is Reduced Space SQP (rSQP).

3.1 Reduced Space Successive Quadratic Programming (rSQP)

In a rSQP method, (4)–(6) is decomposed into two smaller subproblems that, in many cases, are easier to solve. To accomplish this, first a range space / null space decomposition is computed for $A \in \mathbb{R}^{n \times m}$ (assuming it is full rank). This decomposition is defined by a null space matrix Z and range space matrix Y with the following properties:

$$Z \in \mathbb{R}^{n \times (n-m)}, Y \in \mathbb{R}^{n \times m} \text{ s.t. } A^T Z = 0, \begin{bmatrix} Y & Z \end{bmatrix} \text{ nonsingular} \tag{7}$$

By using (7), the search direction d can be broken down into $d = Y p^y + Z p^z$, where $p^y \in \mathbb{R}^m$ and $p^z \in \mathbb{R}^{n-m}$ are the range space and null space steps respectively. By substituting $d = Y p^y + Z p^z$ into (4)–(6) we obtain the range space (8) and null space (9)–(10) subproblems.

Range Space Subproblem	**Null Space (QP) Subproblem**
$p^y = -R^{-1} c^d \quad (8)$	$\min \; (g^r + w)^T p^z + \tfrac{1}{2} (p^z)^T [Z^T W Z] p^z \quad (9)$
	$\text{s.t.} \; b^L \le Z p^z \le b^U \quad (10)$
$R \equiv [A^T Y] \in \mathbb{R}^{m \times m}$	
$g^r \equiv Z^T g \in \mathbb{R}^{n-m}$	$w \equiv Z^T W Y p^y \in \mathbb{R}^{n-m}$
$b^L \equiv x^L - x^k - Y p^y \in \mathbb{R}^n$	$b^U \equiv x^U - x^k - Y p^y \in \mathbb{R}^n$

Using this decomposition, the Lagrange multipliers λ for (5) do not need to be computed in order to compute the steps $d = Yp^y + Zp^z$. However, it is useful to compute these multipliers in the context of some globalization methods and at the solution of the NLP since they represent sensitivities. An expression for computing λ can be derived by applying (7) to $Y^T \nabla L(x, \lambda, \nu)$ to yield $\lambda = -R^{-T} Y^T (g + \nu)$.

There are many details that need to be worked out in order to implement an rSQP algorithm and there are opportunities for a lot of variability. Some significant decisions need to be made: how to compute the range/null space decomposition that defines the matrices Z, Y and R, and how the reduced Hessian $Z^T W Z$ and the cross term w in (9) are calculated (or approximated).

There are several different ways to compute decomposition matrices Z and Y that satisfy (7). Several choices for Z and Y have been investigated that are appropriate for large-scale rSQP [8]. One class of decompostions is based on a variable reduction. In a variable reduction decomposition, the variables are partitioned into dependent (basic) (x^D) and independent (nonbasic) (x^I) sets ($x^T = [(x^D)^T \; (x^I)^T]$) such that the Jacobian of the constraints A^T is partitioned (assuming some permutations) as shown below, where C is a square nonsingular matrix known as the basis matrix. This partitioning is used to define a variable reduction null space matrix Z. Two choices for the range space matrix Y are the coordinate and orthogonal and are shown below.

$$\boxed{\begin{array}{lll} \textbf{Variable Reduction} & \textbf{Coordinate} & \textbf{Orthogonal} \\[6pt] \quad A^T = \begin{bmatrix} C \; N \end{bmatrix} & Z \equiv \begin{bmatrix} -C^{-1}N \\ I \end{bmatrix} & D \equiv -C^{-1}N \\ \text{where:} & Y \equiv \begin{bmatrix} I \\ 0 \end{bmatrix} & Z \equiv \begin{bmatrix} D \\ I \end{bmatrix} \\ \qquad C \in \mathbb{R}^{m \times m} & R = C & Y \equiv \begin{bmatrix} I \\ -D^T \end{bmatrix} \\ \qquad N \in \mathbb{R}^{m \times (n-m)} & & R = C(I + DD^T) \end{array}} \qquad (11)$$

The orthogonal decomposition ($Z^T Y = 0$) is more numerically stable and has other desirable properties in the context of rSQP [8]. However, the amount of dense linear algebra required to compute the factorizations needed to solve for linear systems with R is $O((n-m)^2 m)$ floating point operations (flops) which can dominate the cost of the algorithm for larger $(n-m)$. Therefore, for larger $(n-m)$, the coordinate decomposition ($Z^T Y \neq 0$) is preferred because it is cheaper. The downside is that it is also more susceptible to problems associated with a poor selection of dependent variables and ill conditioning in the basis matrix C that can result in greatly degraded performance.

Another important decision is how the compute the reduced Hessian $B \approx Z^T W Z$. When quasi-Newton is used, limited memory as well as several different dense storage schemes are possible and the best choice depends on the context. In some cases, computing the exact reduced Hessian $B = Z^T W Z$ or using it implicitly in some manner is computationally feasible.

In addition to variations that affect the convergence behavior of the rSQP algorithm, such as range/null 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 (8) can be solved using direct or iterative solvers, and the reduced QP subproblem in (9)–(10) can be solved using a variety of methods (active set vs. interior point) and software [9].

4 An object-oriented approach to SQP (rSQP++)

Most numerical software (optimization, non-linear equation solvers 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. Modern software engineering modeling and development methods, collectively known as Object-Oriented Technology (OOT), can provide powerful abstraction tools for dealing with these types of issues [1], [7]. In addition to abstracting dense linear algebra operations, Object-Oriented Programming (OOP) languages like C++ can be used to abstract any special type of quantity and operation. Also OOT can be used to abstract larger chunks of an algorithm and provide for greater reuse. 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.

There are many types of challenges in trying to build a framework for SQP (as well as for many other methods) that allows for maximal sharing of code, and at the same time is understandable and extendible. Specifically, three types of variability are discussed: (a) Algorithmic variability, (b) implementation variability and (c) NLP specific specializations.

(a) First, we need to come up with a way of modeling and implementing iterative algorithms 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++.

(b) The second type of variability 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 properties of the NLP being solved. One example is whether to represent $D = -C^{-1}N$ in (11) explicitly or implicitly. 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 in what form) or both depends on the choice of QP solver used to solve (9)–(10). Yet another example is allowing different implementations for the QP solver.

(c) A third source of variability is in how to allow users to exploit the special properties of an application area. Abstract interfaces to matrices have been developed that serve as the foundation for facilitating the type of implementation and NLP specific linear algebra variability described above. In addition, these abstract interfaces help manage some of the algorithmic variability such as the choice of different range/null space decompositions.

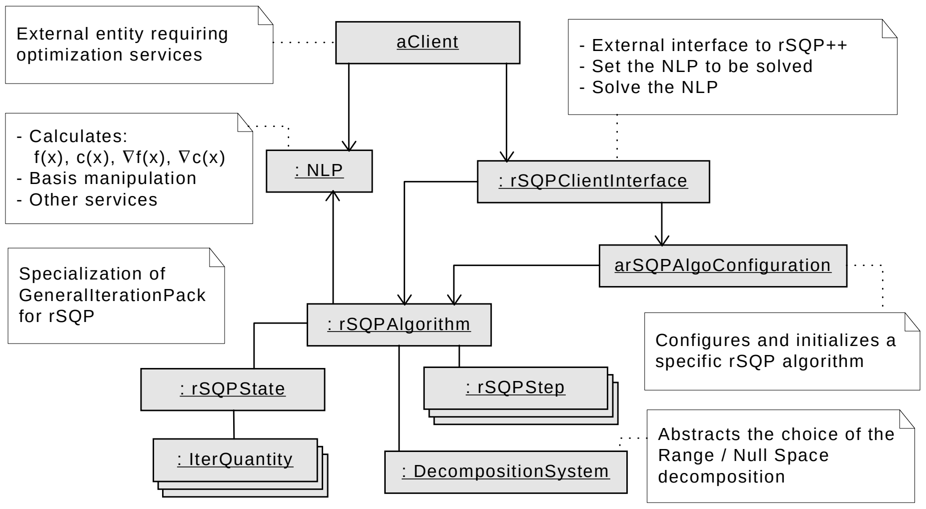

External entity requiring optimization services
aClient
- External interface to rSQP++
- Set the NLP to be solved
- Solve the NLP
- Calculates:
f(x), c(x), ∇f(x), ∇c(x)
- Basis manipulation
- Other services
: NLP
: rSQPClientInterface
arSQPAlgoConfiguration
Specialization of GeneralIterationPack for rSQP
: rSQPAlgorithm
Configures and initializes a specific rSQP algorithm
: rSQPState
: rSQPStep
: IterQuantity
: DecompositionSystem
Abstracts the choice of the Range / Null Space decomposition

Fig. 1. UML Object Diagram: Coarse grained object diagram for an rSQP++ algorithm configured and ready to solve an NLP

Figure 1 shows a coarse grained UML [1] object diagram for a rSQP++ algorithm configured and ready to solve an NLP. At the core is a set of algorithmic objects. The `rSQPAlgorithm` object acts as the center hub for the algorithm and its main job is to fire off a set of steps in sequential order and perform major loops. One or more `rSQPStep` objects perform the actual computations in the algorithm. The `rSQPStep` objects operate on iteration quantity objects (`IterQuantity`) that are stored in the `rSQPState` object. In addition to simple linear execution of an algorithm, more sophisticated control strategies can be performed. This design allows step classes to be shared in many different related algorithms and also provides for modifications of the algorithm by adding, removing and replacing `rSQPStep` and `IterQuantity` objects. In other words, the behavior of the algorithms is not fixed and can be modified at runtime. In this way, users can modify the rSQP algorithms without touching any of the base source code in rSQP++.

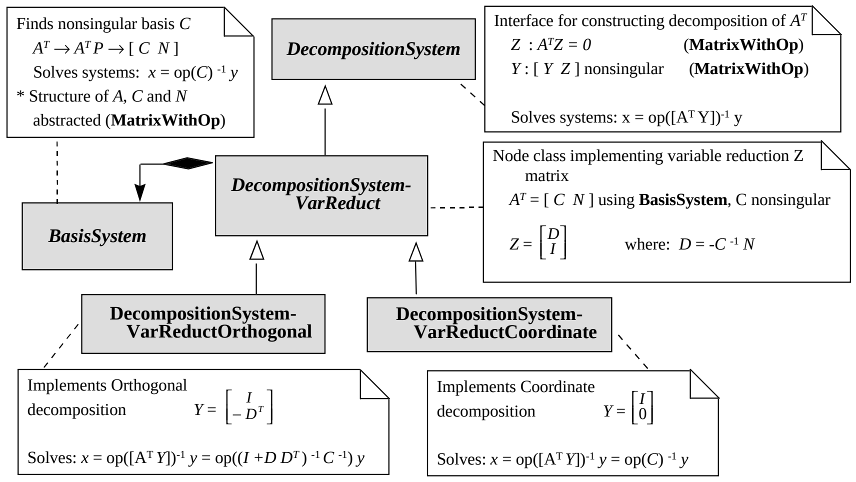

Finds nonsingular basis C
$A^T \rightarrow A^T P \rightarrow [\ C\ \ N\ ]$
Solves systems: $x = \mathrm{op}(C)^{-1}\, y$
* Structure of A, C and N abstracted (**MatrixWithOp**)
DecompositionSystem
Interface for constructing decomposition of A^T
$Z : A^T Z = 0$ (**MatrixWithOp**)
$Y : [\ Y\ \ Z\ ]$ nonsingular (**MatrixWithOp**)
Solves systems: $x = \mathrm{op}([A^T Y])^{-1}\, y$
DecompositionSystem-VarReduct
BasisSystem
Node class implementing variable reduction Z matrix
$A^T = [\ C\ \ N\ ]$ using **BasisSystem**, C nonsingular
$Z = \begin{bmatrix} D \\ I \end{bmatrix}$ where: $D = -C^{-1} N$
DecompositionSystem-VarReductOrthogonal
DecompositionSystem-VarReductCoordinate
Implements Orthogonal decomposition $Y = \begin{bmatrix} I \\ -D^T \end{bmatrix}$
Solves: $x = \mathrm{op}([A^T Y])^{-1}\, y = \mathrm{op}((I + D\, D^T)^{-1} C^{-1})\, y$
Implements Coordinate decomposition $Y = \begin{bmatrix} I \\ 0 \end{bmatrix}$
Solves: $x = \mathrm{op}([A^T Y])^{-1}\, y = \mathrm{op}(C)^{-1}\, y$

Fig. 2. UML Class Diagram: Range/null decomposition system classes

Also shown in Figure 1 are `DecompositionSystem` and `NLP` objects. These objects provide the keys to specializing the linear algebra for a particular NLP. The `DecompositionSystem` interface (Figure 2) abstracts the choice for the range/null space decomposition away from the optimization algorithm. The `DecompositionSystemVarReduct` node subclass is for variable reduction decompositions. A `BasisSystem` object is used to abstract the variable reduction matrices including the basis matrix C. The `DecompositionSystemVar-ReductOrthogonal` and `DecompositionSystemVarReductCoordinate` subclasses implement the orthogonal and coordinate decompositions. The `NLP` interface (Figure 3) is used to abstract the application. The base `NLP` interface provides basic information such as variable bounds and the initial guess and computes $f(x)$ and $c(x)$. The `NLPFirstOrderInfo` specialization is for NLPs that can compute $\nabla f(x)$ and $\nabla c(x)$. The matrix $A = \nabla c(x)$ is represented as an abstract matrix object of type `MatrixWithOp` and can therefore be implemented by any appropriate means. Through this matrix interface, the optimization algorithm can perform only simple operations like matrix vector multiplication $v = op(A)u$. It is only in conjunction with a compatible `BasisSystem` object that the algorithm can perform all the needed computa-

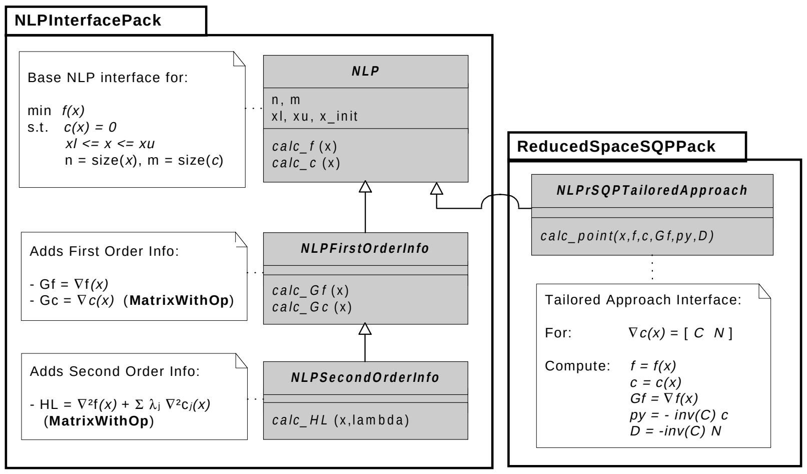


Fig. 3. UML Class Diagram: Interfaces to Nonlinear Programs (NLPs)

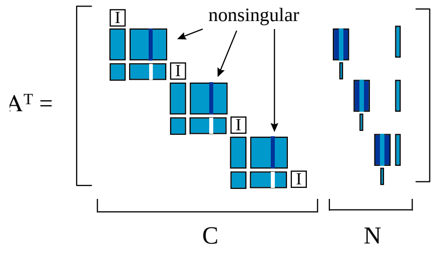
tions. The `NLPSecondOrderInfo` interface is for NLPs that can compute the Hessian of the Lagrangian $\nabla^2 L$ which is also abstracted as a `MatrixWithOp` object. In this way the core rSQP++ code is independent of the specialized data-structures and solvers for an NLP. By configuring the algorithm with `NLP` and `BasisSystem` objects and `MatrixWithOp` objects for $\nabla c(x)$ and possibly $\nabla^2 L$, specialized data structures and linear algebra for an NLP can be accommodated.

The `NLPFirstOrderInfo` interface assumes that matrix vector multiplications with $A = \nabla c(x)$ and its transpose can be performed. The `BasisSystem` interface assumes that linear systems involving the basis matrix C and its transpose can be solved for arbitrary right hand sides. For many applications, these requirements can not be met. For these applications, the `NLPrSQP-TailoredApproach` interface (Figure 3) is defined and it is used by the algorithm to extract the bare minimum information (i.e. $\nabla f(x)$, $p^y = -C^{-1}c$ and $D = -C^{-1}N$). With this information, the coordinate and orthogonal range/null decompositions can both be used. A more detailed introduction to rSQP++ can be found in [2].

5 Tailored linear algebra for specialized NLPs

Figure 4 shows the special structure of a DAE system discretized using orthogonal collocation on finite elements. A specialized implementation of the basis matrix C for this structure called the elemental decomposition [4] was compared to a generic implementation using a general sparse linear solver

(MA28). The Tennessee Eastman challenge problem was used as the example with 80 finite elements for a total NLP size of $n = 43,950$, $n - m = 2,640$ [3]. Figure 4 shows a dramatic reduction of the average CPU time for an rSQP iteration. In addition, storage savings using the elemental decomposition allowed the solution of larger problems. The elemental decomposition was implemented by externally configuring the rSQP++ algorithm with specialized `BasisSystem`, `MatrixWithOp` (for $\nabla c(x)$) and `NLP` objects.



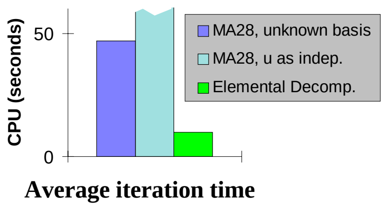


Fig. 4. Tennessee Eastman example: Tailored linear algebra for DAE optimization

A "Tailored Approach" interface (`NLPrSQPTailoredApproach`) has also been implemented for MPSalsa[1] which uses Aztec[2] as the linear algebra library. This interface was used to optimize a CVD reactor model with $m = 31,995$ discretized state variables and $n - m = 1$ decision variables. The rSQP++/MPSalsa implementation converged[3] with the solution in 1.56 hrs while a non-invasive "black box" approach took 22 hrs [10]. In this implementation, multiple processors where also utilized with fair results.

6 Moving from serial to distributed memory: vector interfaces

There are many different types of objects provided by a numerical linear algebra library such as vectors, matrices and linear solvers. We contend that vectors are what primarily "glue" optimization software to the specialized implementations of higher level linear algebra components (e.g. matrices, linear solvers) as well as to other types of objects. For example, in a NLP solver, vector objects represent the estimate of the unknown variables $x^k \in \mathbb{R}^n$, the residual of the nonlinear constraints $c(x^k) \in \mathbb{R}^m$, the gradient of the objective or other functions $\nabla f(x^k) \in \mathbb{R}^n$ as well as many other intermediate

[1] `http://www.cs.sandia.gov/CRF/MPSalsa`

[2] Aztec, `http://www.cs.sandia.gov/CRF/aztec1.html`

[3] Single Intel PIII 500 MHz processor, RedHat Linux 6.1

vectors (i.e. search directions, Lagrange multipliers etc.). Many of these vectors are passed around by the optimization code to various other objects to be used to transform other objects. For instance, vectors are passed to an NLP interface to compute function (scalar $f(x^k)$ and vector $c(x^k)$) and gradient (vector $\nabla f(x^k)$ and matrix $\nabla c(x^k)$) values. Right hand side vectors are passed to linear solvers to compute solution vectors. In addition to having the optimization software just shuffle vector objects around to other objects so they can be operated on or transformed, the algorithm also needs to perform various reduction (e.g. norms, dot products etc.) and transformation (e.g. vector addition, scaling etc.) operations with these vectors. If all of the linear algebra operations are implemented in a shared memory environment, then simple Fortran style arrays for dense and sparse vectors could be used, and the optimization software can trivially and cheaply access any arbitrary vector element. For the most part, utilizing parallelism in a shared memory environment is relatively easy if the optimization software calls standard packages like the BLAS and LAPACK. Multiple processors can be utilized by simply linking the object code to multi-threaded libraries. This is possible partly because the optimization software can communicate with these various implementations using simple serial Fortran style arrays.

The difficulty arises, however, when we move to a distributed memory environment. Now we can no longer deal with simple Fortran style serial arrays and instead have to consider distributed data structures and communication mechanisms. This can be seen in several different packages for distributed memory numerical linear algebra (ScaLAPACK[4], Aztec, PETSc[5] , SuperLU[6] etc.). In such a distributed environment, direct and cheap access to every vector element is no longer a luxury an optimization code can enjoy. Trying to maintain an optimizer for every different distributed linear algebra library and flavor of parallelism (i.e. library based MPI[7], PVM[8] or compiler generated PGHPF[9], MPC++[10]) and also deal with serial vectors as well as other off-the-wall implementations (i.e. out-of-core) is an impossible task. What is needed is a set of abstract vector interfaces that can encapsulate these details away from the optimization algorithm, so that the implementations of the library and algorithm can be varied independently.

Because rSQP++ was developed initially to use serial direct solvers, issues related to the use of massively parallel iterative solvers used in PDE constrained optimization were not addressed. In order to effectively support parallelism, rSQP++ needs the addition and integration of an abstract vector

[4] Scalable LAPACK, `http://www.netlib.org/scalapack/index.html`

[5] Portable Extendable Toolkit for Scientific Computaion, `http://www.mcs.anl.gov/petsc`

[6] Supernodal LU, `http://www.nersc.gov/~xiaoye/SuperLU`

[7] Message Passing Interface, `http://www.mpi-fourm.org/docs/docs.html`

[8] Parallel Virtual Machine, `http://www.epm.ornl.gov/pvm/pvm_home.html`

[9] Portland Group High Performance Fortran, `http://www.pgroup.com`

[10] Massively Parallel C++, `http://pdswww.rwcp.or.jp/mpc++`

interface. Encapsulating vector implementations away from the core rSQP++ algorithmic code would allow for fully parallel linear algebra but could also greatly restrict the kinds of vector operations that could be performed. The primary goal of this discussion is to present a new strategy for implementing abstract vector interfaces that will allow an optimization developer to define and implement arbitrary vector reduction and transformation operations in a way that does not require vectors to reveal how they are implemented (parallel or serial, out-of-core etc.).

Perhaps the primary distinction between vectors and other linear algebra objects is the large number of operations that need to be performed with them. In addition to the 15 operations that are part of the Basic Linear Algebra Subroutines (BLAS), many other types of operations need to be performed as well. For example, in addition to many of the standard BLAS like operations, some of the other vector reduction and transformation operations that the interior-point QP solver OOQP[11] must perform are shown below.

$$y_i \leftarrow \begin{cases} y^{\text{min}} - y_i & \text{if } y_i < y^{\text{min}} \\ y^{\text{max}} - y_i & \text{if } y_i > y^{\text{max}} \\ 0 & \text{if } y^{\text{min}} \leq y_i \leq y^{\text{max}} \end{cases} \quad \text{for } i = 1...n \tag{12}$$

$$\alpha \leftarrow \{\max \alpha \mid x + \alpha d \geq \beta\} \tag{13}$$

Some examples of other non-standard vector reduction operations that may be performed in rSQP++ include the following.

$$\gamma \leftarrow d^{\text{max}} + \frac{1}{\rho} ln \left\{ \frac{1}{n} \sum_{i=1}^{n} exp \left\{ \rho \left((x_i - b_i) - d^{\text{max}} \right) \right\} \right\} \tag{14}$$

$$\gamma \leftarrow max \left\{ \frac{|d_i|}{1 + |x_i|}, i = 1...n \right\} \tag{15}$$

$$\{y_k, k\} \leftarrow max \{y_i, i = 1...n\} \tag{16}$$

Many of these operations are not really vector operations in the mathematical sense. They are more array-type operations that are used to implement other types of computations and are therefore no less useful.

It seems that the most common approach taken by developers of linear algebra libraries has been to provide a large set of primitive vector operations, and then the user can perform more complex operations by stringing together a set of primitives. For most vector operations this is feasible as long as the vector interface includes the needed primitives. For example, the vector reduction operation (15) could be performed with two temporary vectors $u, v \in \mathbb{R}^n$ and five primitive vector operations as $|d_i| \rightarrow u_i$, $|x_i| \rightarrow v_i$, $v_i + 1 \rightarrow v_i$, $u_i / v_i \rightarrow u_i$, $max\{u_i\} \rightarrow \gamma$. Many of the other example vector operations shown above can be performed using primitives. It is

[11] http://www-unix.mcs.anl.gov/~wright/ooqp/

difficult to see how (12) and (13) could be implemented with general primitive vector operations though. Perhaps it is possible but at the very least it would take a number of temporary vectors and several operations. The number of primitive operations that need to be included in a vector interface in order to implement most of the needed vector operations is very large. For example, the HCL[12] vector interface contains more than 50 operations and still can not accommodate some of the above operations. In addition, using these non-overlapping operations can result in a performance bottleneck in a parallel application. For example, in ISIS++[13] the the operation `Dist_Vector::QMR3norm2dot(...)` which performs $\{\alpha,\gamma,\xi,\rho,\epsilon\} \leftarrow \{(x^Tx)^{1/2},(v^Tv)^{1/2},(w^Tw)^{1/2},w^Tv,v^Tt\}$ was added to the base vector class to overcome a bottleneck in the QMR solver[14].

Before a new approach to implementing vector interfaces is presented note that the majority of vector implementations store and manipulate vector elements in continuous chunks (sub-vectors). For instance, most serial vectors are stored and manipulated as a whole 1-D array. For a parallel vector, elements are distributed between a set of processors and operations are performed in parallel on the local sub-vectors. In an out-of-core vector, the elements are stored in a set of files and the elements are read into RAM as sub-vectors in order to participate in vector operations. For efficient implementation in all of these cases, these sub-vectors must be of reasonably large size (i.e. hundreds to thousands of elements) so that the low level numeric operations performed with these elements dominate the runtime cost.

6.1 Vector reduction/transformation operators

A new approach is to allow clients of abstract vector objects to define an operator object, give it to a vector object through an abstract vector interface and then have the vector implementation apply the operator through an abstract operator interface. The idea is to allow a client to create a vector reduction/transformation operator that is equivalent to the following element-wise operators.

$$op(i, v_i^1 \ldots v_i^p, z_i^1 \ldots z_i^q) \to z_i^1 \ldots z_i^q \tag{17}$$

$$op(i, v_i^1 \ldots v_i^p, z_i^1 \ldots z_i^q) \to \beta \tag{18}$$

$$op(\beta^1, \beta^2) \to \beta^2 \tag{19}$$

where $v^1 \ldots v^p \in \mathbb{R}^n$ are a set of p non-mutable input vectors ($p = 0$ allowed), $z^1 \ldots z^q \in \mathbb{R}^n$ are a set of q mutable input/output vectors ($q = 0$ allowed) and β is a reduction target object which may be a simple scalar, a more complex non-scalar (i.e. $\{\alpha, \gamma, \xi, \rho, \epsilon\}$) or NULL. In the most general case

[12] Hilbert Class Library, `http://www.trip.caam.rice.edu/txt/hcldoc/html/index.html`

[13] ISIS++, `http://z.ca.sandia.gov/isis`

[14] Example provided by Ben Allan, `baallan@california.sandia.gov`

the client can define an operator that will simultaneously perform multiple reduction and transformation operations involving a set of vectors. Simpler operations can be formed by setting $p = 0$, $q = 0$ or $\beta = \text{NULL}$. For example, reduction operations over one vector argument such as vector norms ($||v||$) are defined with $p = 1$, $q = 0$ and $\beta = \{\text{scalar}\}$. With this design, all of the standard BLAS operations, all of the example vector operations in (12)–(16) and many many more vector operators can all be expressed. The key to optimal performance is that the vector implementation applies a user defined operator not element-wise as shown above, but instead on an entire set of sub-vectors (for elements $i = a \ldots b$) at once

$$op(a, b, v^1_{a:b} \ldots v^p_{a:b}, z^1_{a:b} \ldots z^q_{a:b}, \beta) \rightarrow z^1_{a:b} \ldots z^q_{a:b}, \beta \tag{20}$$

In this way, as long as the size of the sub-vectors is sufficiently large, the cost of performing a function call to invoke the operator will be insignificant compared to the cost of performing the computations within the operator. In a parallel distributed vector, $op(\ldots)$ is applied to the local sub-vectors on each processor (no communication). The only communication between processors is to reduce the intermediate reduction objects $op(\beta^1, \beta^2) \rightarrow \beta^2$ (unless $\beta = \text{NULL}$ then no communication is required). A prototype implementation called `RTOP` for these operator iterfaces has been developed in C. A paper and a set of html documentation describing this design in much more detail can be found at the web site[15]. Support of more sophisticated client/server runtime configurations and other issues are discussed as well.

This design provides a more feasible means by which unusual vector operations can be efficiently implemented. The developers of vector implementations only need to implement a single `apply(op,...)` method that accepts user defined reduction/transformation operator objects. It is then the optimization algorithm developers' responsibility to implement strange or specialized vector operators, but they can do so in a way that is independent of any particular vector implementation.

7 Summary and future work

Many application areas give rise to large-scale NLPs with specialized properties that can not be exploited by most of the current generation of optimization software. Noted examples include DAE and PDE constrained optimization. The next generation of optimization software needs to allow these specializations but also include those components of successful implementations that are so important for reliability and efficiency. The rSQP++ framework is being developed in an attempt to address these needs in the context of SQP. With rSQP++ the user can externally configure the algorithm with specialized linear algebra objects and can perform other modifications without having to recompile any of the base rSQP++ source code.

[15] `http://dynopt.cheme.cmu.edu/roscoe/RTOp`

In order for rSQP++ to fully exploit parallelism (i.e. PDE constrained optimization), an abstract vector interface must be developed and incorporated. A new design for abstract vector interfaces based on vector reduction/transformation operators would allow developers of optimization algorithms to efficiently implement specialized vector operations, while also allowing great flexibility in the implementation of vectors (i.e. serial, parallel, out-of-core, etc.).

References

1. Booch, G., J. Rumbaugh, and I. Jacobson. "The Unified Modeling Language User Guide." *Addison-Wesley*, New York (1999)
2. Bartlett, R. "An Introduction to rSQP++ : An Object-Oriented Framework for Reduced Space Successive Quadratic Programming" Technical Report, Department of Chemical Engineering, Carniege Mellon University (2000)
3. Bartlett, R. "New Object-Oriented Approaches to SQP for Large-Scale Process Optimization" Parallel and Large-Scale Computing: algorithms and Applications, AICHE Annual Meeting, Los Angeles (2000)
4. Biegler, L., A. Cervantes and A. Waechter. "Advances in Simultaineous Strategies for Dynamic Optimization." CAPD Technical Report B-01-01, Department of Chemical Engineering, Carniege Mellon University (2001)
5. Nocedal, J. and M. Overton. "Projected Hessian Updating Algorithms for Nonlinear Constrained Optimization". *SIAM J. Numer. Anal.*, Philadelphia, **22**, 821 (1985)
6. Nocedal, J. and Stephen Wright. "Numerical Optimization". *Springer*, New York, (1999)
7. Rumbaugh, J. et. al. "Object-Oriented Modeling and Design." *Prentice Hall*, Englewood Cliffs, New Jersey (1991)
8. Schmid, C. and L.T. Biegler. "Acceleration of Reduced Hessian Methods for Large-Scale Nonlinear Programming." *Comp. Chem. Eng.* **17**, 451 (1993)
9. Schmid, C. and L.T. Biegler. "Quadratic Programming Methods for Reduced Hessian SQP." *Comp. Chem. Eng.* **18**, 817 (1994)
10. van Bloemen Waanders, B., A. Salinger, R. Pawlowski, L. Biegler, R. Bartlett "Simultaneous Analysis and Design Optimization of Massively Parallel Simulation Codes using an Object Oriented Framework." Tenth SIAM Conference on Parallel Processing for Scientific Computing (SIAG/SC) (PP01) March (2001)
11. Womble, D.E, S.S. Dosanja, B. Hendrickson, M.A. Heroux, S.J. Plimpton, J.L. Tomkins and D.S. Greenberg. "Massively Parallel Computing: A Sandia Perspective." *Parallel Computing*, **25**, 1853-1876 (1999)
12. Wright, S. "Optimization Software Packages." *ANL/MCS-P8xx-0899*, Mathematics and Computer Science Division, Argonne National Laboratory (1999)