**SANDIA REPORT**

SAND2009-3969
Unlimited Release
Printed Jine 2009

# Mathematical and High-Level Overview of MOOCHO

# The Multifunctional Object-Oriented arCHitecture for Optimization

Roscoe A. Bartlett

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's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

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

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

| | |
|---|---|
| Telephone: | (800) 553-6847 |
| Facsimile: | (703) 605-6900 |
| E-Mail: | orders@ntis.fedworld.gov |
| Online ordering: | http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online |

SAND2009-3969
Unlimited Release
Printed Jine 2009

# Mathematical and High-Level Overview of MOOCHO

# The Multifunctional Object-Oriented arCHitecture for Optimization

Roscoe A. Bartlett
Department of Optimization and Uncertainty Estimation

Sandia National Laboratories[1], Albuquerque NM 87185 USA,

**Abstract**

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is a object-oriented C++ Trilinos package for solving equality and inequality constrained nonlinear programs (NLPs) using large-scale gradient-based optimization methods. The primary focus of MOOCHO up to this point has been the development of active-set successive quadratic programming (SQP) methods. MOOCHO was initially developed (under the name rSQP++) to support primarily reduced-space SQP (rSQP) but other related types of optimization algorithms can also be developed. Using MOOCHO, it is possible to specialize all of the linear-algebra computations and also modify many other parts of the algorithm externally (without modifying default library source code). One feature of the MOOCHO framework is that it supports completely abstract linear algebra which allows sophisticated implementations on parallel distributed-memory supercomputers but is not tied to any particular linear algebra

[1]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.

library (although adapters to a few linear algebra libraries are available). In addition, MOOCHO contains adapters to support massively parallel simulation-constrained optimization through Thyra Model Evaluator interface. Access to a great deal of linear solver technology in Trilinos is available through the Stratimikos package.

This document provides a high-level overview of MOOCHO that describes the motivation for MOOCHO, the basic mathematical notation used in MOOCHO, the algorithms that MOOCHO implements, and what types of optimization problems are appropriate to be solved by MOOCHO. More detailed documentation on how to install MOOCHO, how to define NLPs, and how to run MOOCHO algorithms is provided in the companion MOOCHO Reference Manual [**?**].

# Acknowledgment

I would like to thank by Ph.D. adviser Dr. Larry Biegler for is help in teaching me optimization and helping to formulate MOOCHO back when it was called rSQP++. I would also like to thank Carl Laird (now Dr. Laird) for helping on many different aspects of the code and the algorithms.

# Contents


0.1 Introduction . . . . . 11

0.2 Mathematical Background . . . . . 12

0.2.1 Nonlinear Program (NLP) Formulation . . . . . 12

0.2.2 Successive Quadratic Programming (SQP) . . . . . 15

0.2.3 Reduced-Space Successive Quadratic Programming (rSQP) . . . . 17

0.2.4 General Inequalities, Slack Variables and Basis Permutations . . . 23

0.3 Basic Software Architecture of MOOCHO . . . . . 26

0.3.1 High-Level Object Diagram for MOOCHO . . . . . 26

0.4 Overview of NLP and Linear-Algebra Interfaces . . . . . 28

0.4.1 Basic Application Requirements for Simulation-Constrained Optimization with MOOCHO . . . . . 28

0.4.2 Overview of `AbstractLinAlgPack`: Interfaces to Linear Algebra Objects . . . . . 30

0.4.3 Overview of `NLPInterfacePack`: Interfaces to Nonlinear Programs . . . . . 35

0.5 Defining Optimization Problems . . . . . 38

0.6 Basic properties of MOOCHO Algorithms . . . . . 38

0.6.1 Solver options . . . . . 39

0.6.2 Algorithm Description and Iteration Output . . . . . 40

0.6.3 Algorithm Summary and Timing . . . . . 41

0.6.4 Algorithm and NLP Testing and Validation . . . . . 41

0.6.5 Algorithm Interruption . . . . . 44

0.7 Summary . . . . . 44

**References** **45**

## Appendix

.1 MOOCHO Equation Summary and Nomenclature Guide . . . . . 45

# List of Figures


1 UML object diagram : Course grained object diagram for MOOCHO . 27

2 UML class diagram : `AbstractLinAlgPack`, abstract interfaces to linear algebra . . . . . 31

3 UML class diagram : `NLPInterfacePack`, abstract interfaces nonlinear programs . . . . . 36

## 0.1 Introduction

MOOCHO is an object-oriented C++ software package which implements gradient-based algorithms for large-scale nonlinear programing. MOOCHO is designed to allow the incorporation of many different algorithms and to allow external configuration of specialized linear-algebra objects such as vectors, matrices and linear solvers (i.e. through Thyra). Data-structure independence has been recognized as an important feature missing in many optimization solvers [**?**].

While the MOOCHO framework can be used to implement many different types of optimization methods (e.g. Generalized Reduced Gradient (GR) [**?**], Augmented Lagrangian (AL) [**?**], Successive Quadratic Programming (SQP) [**?**]) the main focus has been SQP methods. Successive quadratic programming (SQP) and related 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 [**?**]. Another attractive property of SQP methods is that they can be adapted to effectively exploit the structure of the underlying NLP [**?**]. A variation of SQP, known as reduced-space SQP (rSQP), works well for NLPs where there are few degrees of freedom (see Section 0.2.1) and many constraints. Quasi-Newton methods for approximating the reduced Hessian of the Lagrangian are also very efficient for NLPs with few degrees of freedom. Another advantage of rSQP is that a decomposition for the equality constraints can be used which only requires solves with a basis of the Jacobian of the constraints (see Section 0.2.3) and therefore can utilize very specialized application-specific data structures and linear solvers. Therefore, rSQP methods can be tailored to effectively exploit the structure of simulation-constrained optimization problems and can show excellent parallel algorithmic scalability.

There is a distinction to be made between a user of MOOCHO and a developer of MOOCHO, though it may it be narrow one in some cases. Here we define a user as anyone who uses MOOCHO to solve an optimization problem using a pre-written MOOCHO algorithm. A MOOCHO user can vary from someone who uses a pre-developed interface to a modeling environment like AMPL [**?**] to someone who uses MOOCHO to solve a discretized simulation-constrained optimization problem on a massively parallel computer using specialized application-specific data structures and linear solvers [**?**]. While the first type of user does not need to write any C++ code and does not even need to know what C++ is, the latter type of sophisticated user has to write a fair amount of C++ code. There are also many different types of use cases of MOOCHO that lie in between these two extremes.

In the next section (Section 0.2), the basic mathematical structure of SQP methods is presented. This presentation is intended to establish the nomenclature of MOOCHO for users and developers and to describe what algorithms MOOCHO actually implements. The nomenclature that is established is key to being able to understand the output from the MOOCHO algorithms. Appendix .1 contains a summary of this notation. The basic software design of MOOCHO is then described in Section 0.3.

This is followed in Section 0.4 by a basic description of the linear algebra and NLP interfaces in MOOCHO. These interfaces provide the foundation for allowing the types of specialized data structures and linear solvers that an advanced user would use with MOOCHO. Section 0.5 briefly discusses how one can define an NLP for MOOCHO to solve and where to look for more information. Section 0.6 deals with the basics of using MOOCHO to solve optimization problems and describes some of the common properties shared by all MOOCHO algorithms.

## 0.2 Mathematical Background

### 0.2.1 Nonlinear Program (NLP) Formulation

MOOCHO can be used to solve NLPs of the general form:

$$\begin{aligned} \min \quad & f(x) && (1) \\ \text{s.t.} \quad & c(x) = 0 && (2) \\ & x_L \leq x \leq x_U && (3) \end{aligned}$$

where:

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

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 dimension and the same inner product will not always be sufficient to allow for the compatibility of vectors from different vector spaces in the implementation (e.g. coefficients of parallel vectors can have different distributions to processes). Vector spaces become important later when the NLP interfaces and the implementation of MOOCHO is discussed in more detail in Section 0.4.

We assume that $f(x)$ and $c_j(x)$ for $j = 1 \ldots m$ in (1)–(2) are nonlinear functions with at least second-order continuous derivatives. The rSQP algorithms described

later only require first-order information (derivatives) for $f(x)$ and $c_j(x)$. However, these first derivatives can be provided by (directional) finite differences if missing. The simple bound inequality constraints in (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 of the 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, $\delta$ (see the method `max_var_bounds_viol()` in the Doxygen documentation for the ***NLP*** interface) is a relaxation (i.e. wiggle room) that the user can set to allow the optimization algorithm to compute $f(x)$ and $c(x)$ outside the strict variable bounds $x_L \leq x \leq x_U$ in order to compute finite differences and the like. The SQP algorithms in MOOCHO will never evaluate $f(x)$ and $c(x)$ outside the above relaxed variable bounds. This gives users a measure of control in how the optimization algorithms interact with the NLP model.

The Lagrangian function $L(\lambda, \nu_L, \nu_U)$ and the Lagrange multipliers ($\lambda$, $\nu_L$, $\nu_U$) for this NLP are defined by

$$
\begin{aligned}
L(x, \lambda, \nu_L, \nu_U) &= f(x) + \lambda^T c(x) + \nu_L^T (x_L - x) + \nu_U^T (x - x_U) \;\in \mathbf{R}, && (4)\\
\nabla_x L(x, \lambda, \nu) &= \nabla f(x) + \nabla c(x)\lambda + \nu \;\in \mathcal{X}, && (5)\\
\nabla^2_{xx} L(x, \lambda) &= \nabla^2 f(x) + \sum_{j=1}^{m} \lambda_{(j)} \nabla^2 c_j(x) \;\in \mathcal{X}|\mathcal{X}, && (6)
\end{aligned}
$$

where:

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

Above, we use the notation $\lambda_{(j)}$ with the subscript in parentheses to denote the one-based $j^{\text{th}}$ component of the vector $\lambda$ 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 linear-operator space $\mathcal{X}|\mathcal{C}$ where the range and the domain are the vector spaces $\mathcal{X}$ and $\mathcal{C}$ respectively. The returned 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 object $A^T$ defines an adjoint linear operator where $q = A^T p$ maps vectors from $p \in \mathcal{X}$ to $q \in \mathcal{C}$.

Given the definition of the Lagrangian and its derivatives in (4)–(6), the first- and second-order necessary KKT optimality conditions [?] for a solution $(x^*, \lambda^*, \nu_L^*, \nu_U^*)$ to (1)–(3) are given in (7)–(13). There are four different categories of optimality conditions: linear dependence of gradients (7), feasibility (8)–(9), non-negativity of Lagrange multipliers for inequalities (10), complementarity (11)–(12), and curvature (13).

$$\nabla_x L(x^*, \lambda^*, \nu^*) = \nabla f(x^*) + \nabla c(x^*)\lambda^* + \nu^* = 0 \tag{7}$$

$$c(x^*) = 0 \tag{8}$$

$$x_L \leq x^* \leq x_U \tag{9}$$

$$(\nu_L)^*, (\nu_U)^* \geq 0 \tag{10}$$

$$(\nu_L)^*_{(i)}((x_L)_{(i)} - (x^*)_{(i)}) = 0, \quad \text{for } i = 1 \ldots n \tag{11}$$

$$(\nu_U)^*_{(i)}((x^*)_{(i)} - (x_U)_{(i)}) = 0, \quad \text{for } i = 1 \ldots n \tag{12}$$

$$d^T \, \nabla^2_{xx} L(x^*, \lambda^*) \, d \geq 0, \quad \text{for all feasible directions } d \in \mathcal{X}. \tag{13}$$

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

$$d^T \, \nabla^2_{xx} L(x^*, \lambda^*) \, d = 0$$

for $d \neq 0$ in (13).

To solve an NLP, an SQP algorithm must first be supplied an initial guess for the unknown variables $x_0$ and in some cases also initial guesses for the Lagrange multipliers $\lambda_0$ and $\nu_0$. The optimization algorithms implemented in MOOCHO generally require that the initial guess $x_0$ satisfy the variable bounds in (3), and if not, then the elements of $x_0$ can be forced in bounds before starting the algorithm.

### 0.2.2 Successive Quadratic Programming (SQP)

A popular class of methods for solving NLPs is successive quadratic programming (SQP) [**?**]. An SQP method is equivalent, in many cases, to applying Newton's method to solve the optimality conditions represented by (7)–(8). At each Newton iteration $k$ for (7)–(8), the linear subproblem (also known as the KKT system) takes the form

$$\begin{bmatrix} W & A \\ A^T & \end{bmatrix} \begin{bmatrix} d \\ d_\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_x L \\ c \end{bmatrix} \tag{14}$$

where:

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

The Newton matrix in (14) is known as the KKT matrix. By substituting $d_\lambda = \lambda_{k+1} - \lambda_k$ into (14) and moving $\lambda_k$ to the right-hand-side, this linear system becomes equivalent to the optimality conditions of the following QP.

$$\min \quad g^T d + \tfrac{1}{2} d^T W d \tag{15}$$

$$\text{s.t.} \quad A^T d + c = 0 \tag{16}$$

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 to allow for infeasible subproblems, which yields the following QP.

$$\min \quad g^T d + \tfrac{1}{2} d^T W d + M(\eta) \tag{17}$$

$$\text{s.t.} \quad A^T d + (1 - \eta) c = 0 \tag{18}$$

$$x_L - x_k \leq d \leq x_U - x_k \tag{19}$$

$$0 \leq \eta \leq 1 \tag{20}$$

where:

$$M(\eta) \in \mathbb{R} \rightarrow \mathbb{R}.$$

Near the solution of the NLP, the set of optimal active constraints for (17)–(20) will be the same as the optimal active-set for the NLP in (1)–(3) [**?**, Theorem 18.1].

The relaxation of the QP shown in (17)–(20) is only one form of a relaxation but has some essential properties. For example, the solution $\eta = 1$ and $d = 0$ is always feasible by construction. However, the solution $\eta = 1$ and $d = 0$ is of little practical use since it results in zero steps. The penalty function $M(\eta)$ is either linear or quadratic 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 $M(\eta) = \eta\tilde{M}$ or $M(\eta) = (\eta + \frac{1}{2}\eta^2)\tilde{M}$ where $\tilde{M}$ is a large constant often called "big M". The default QP solver in MOOCHO, QPSchur [**?**], is careful not to allow the ill-conditioning associated with $\tilde{M}$ to impact the solution unless it is needed for an infeasible QP.

Once a new estimate of the solution $(x_{k+1}, \lambda_{k+1}, \nu_{k+1})$ is computed, the error in the optimality conditions (7)–(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 (17)–(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. the maximum number of iterations or maximum runtime is exceeded).

The iterates generated from $x_{k+1} = x_k + d$ are generally only guaranteed to converge to a local minimum 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 minimum from remote starting points. One popular class of globalization methods are line search methods. In a line search method, once the step $d$ is computed from the QP subproblem, a line search procedure is used to find a step length $\alpha$ 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 $\ell_1$ (21) where $\mu$ is a penalty parameter that is adjusted to insure descent along the SQP step $x_k + \alpha d$ for $\alpha > 0$.

$$\phi_{\ell_1}(x) = f(x) + \mu||c(x)||_1 \tag{21}$$

An alternative line search based on a "Filter" has also been implemented which generally performs better and does not require the maintenance of a penalty parameter $\mu$.

Other globalization methods such as trust region (using a merit function or the filter) can also be applied to SQP but no trust region method is currently implemented in MOOCHO.

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 [**?**]. 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 using finite differences. Other difficulties with SQP include how to deal with an indefinite projected Hessian. Also, for large problems, the full QP subproblem in (17)–(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. RSQP methods are discussed in more detail in the next section.

### 0.2.3 Reduced-Space Successive Quadratic Programming (rSQP)

In a reduced-space SQP (rSQP) method, the full-space QP subproblem (17)–(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 [**?**, Section 18.3] is computed for some linearly independent set of the linearized equality constraints $A_d \in \mathcal{X}|\mathcal{C}_d$ where $c_d(x) \in \mathcal{C}_d \in \mathbb{R}^r$ are the decomposed equality constraints and $c_u(x) \in \mathcal{C}_u \in \mathbb{R}^{(m-r)}$ are the undecomposed equality constraints and

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

Above, the vector space $\mathcal{C} = \mathcal{C}_d \times \mathcal{C}_u$ denotes a blocked vector space (also known as a product space) 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 linear operator $Z$ and a linear operator $Y$ with the following properties:

$$\begin{aligned} Z &\in \mathcal{X}|\mathcal{Z} \quad \text{s.t. } (A_d)^T Z = 0 \\ Y &\in \mathcal{X}|\mathcal{Y} \quad \text{s.t. } \begin{bmatrix} Y & Z \end{bmatrix} \text{ is nonsingular} \end{aligned} \tag{23}$$

where:

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

It is important to distinguish the vector spaces $\mathcal{Z}$ and $\mathcal{Y}$ from the the linear operators $Z$ and $Y$. The null-space linear operator $Z \in \mathcal{X}|\mathcal{Z}$ is a linear operator that maps vectors from the null-space space $u \in \mathcal{Z}$ to vectors in the space of the unknowns $v = Zu \in \mathcal{X}$. The linear operator $Y \in \mathcal{X}|\mathcal{Y}$ 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 linear operator $Y$ is referred to as the "range-space" linear operator since several popular choices of this linear operator form a basis for the range space of $A_d$. However, note that the linear operator $Y$ need not be a true basis linear operator for the range-space of $A_d$ in order to satisfy the non-singularity property in (23). For this reason, here the linear operator $Y$ will be referred to as the "quasi-range-space" linear operator to make this distinction.

By using (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 (17)–(20) we obtain the quasi-normal (24) and tangential (25)–(27) subproblems. In (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**

$$p_y = -R^{-1}c_d \in \mathcal{Y} \tag{24}$$

where: $R \equiv [(A_d)^T Y] \in \mathcal{C}_d|\mathcal{Y}$ (nonsingular via (23)).

**Tangential (Null-Space) Subproblem (Relaxed)**

$$\begin{aligned} \min \quad & (g^r + \zeta w)^T p_z + \tfrac{1}{2} p_z^T [Z^T W Z] p_z + M(\eta) && (25) \\ \text{s.t.} \quad & U_z p_z + (1-\eta)u = 0 && (26) \\ & b_L \leq Zp_z - (Yp_y)\eta \leq b_U && (27) \end{aligned}$$

where:

$$g^r \equiv Z^T g \in \mathcal{Z}$$
$$w \equiv Z^T W Y p_y \in \mathcal{Z}$$

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

By using this decomposition, the Lagrange multipliers $\lambda_d$ 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 $\mu$ for the merit function [**?**, page 544] or to compute the Lagrangian function. Alternatively, a multiplier free method for computing $\mu$ has been developed and tested with good results [**?**]. 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 [**?**, page 436]. An expression for computing $\lambda_d$ can be derived by applying (23) to $Y^T \nabla L(x, \lambda, \nu) = 0$ to yield

$$
\lambda_d = -R^{-T}\left(Y^T(g+\nu) + U_y^T \lambda_u\right) \in \mathcal{C}_d. \tag{28}
$$

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. There are some significant decisions that need to be made such as 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 (25) are calculated (or approximated).

There are several different ways to compute decomposition matrices $Z$ and $Y$ that satisfy (23) [**?**]. 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 $A_d$ [**?**]. 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. MOOCHO currently does not implement a orthonormal QR decomposition but one can be added at some point if needed. 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 [**?**]. In a variable-reduction decomposition, the variables are partitioned into dependent $x_D$ and independent $x_I$ sets

$$
x_D \in \mathcal{X}_D \tag{29}
$$

$$
x_I \in \mathcal{X}_I \tag{30}
$$

$$x = \begin{bmatrix} x_D \\ x_I \end{bmatrix} \qquad \in \mathcal{X}_D \times \mathcal{X}_I \tag{31}$$

$$\tag{32}$$

where:

$$\begin{aligned} \mathcal{X}_D &\subseteq \mathbb{R}^r \\ \mathcal{X}_I &\subseteq \mathbb{R}^{n-r} \end{aligned}$$

such that the Jacobian of the constraints $A^T$ is partitioned as shown in (33) where $C$ is a square, nonsingular linear operator known as the basis matrix. The variables $x_D$ and $x_I$ are also called the state and design (or controls) variables [**?**] in some contexts or the basic and nonbasic variables [**?**] in other contexts. 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 [**?**] 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 (34) which also determines $U_z$ in (35).

**Variable-Reduction Partitioning**

$$A^T = \begin{bmatrix} (A_d)^T \\ (A_u)^T \end{bmatrix} = \begin{bmatrix} C & N \\ E & F \end{bmatrix} \tag{33}$$

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**

$$Z \equiv \begin{bmatrix} -C^{-1}N \\ I \end{bmatrix} \tag{34}$$

$$U_z = F - E\,C^{-1}N \tag{35}$$

There are many choices for the quasi-range-space matrix $Y$ that satisfy (23). Two relatively computationally inexpensive choices are the coordinate and orthogonal decompositions shown below.

**Coordinate Variable-Reduction Null-Space Decomposition**

$$Y \equiv \begin{bmatrix} I \\ 0 \end{bmatrix} \tag{36}$$

$$R = C \tag{37}$$

$$U_y = E \tag{38}$$

**Orthogonal Variable-Reduction Null-Space Decomposition**

$$Y \equiv \begin{bmatrix} I \\ N^T C^{-T} \end{bmatrix} \tag{39}$$

$$R = C(I + C^{-1} N N^T C^{-T}) \tag{40}$$

$$U_y = E - F N^T C^{-T} \tag{41}$$

The orthogonal decomposition ($Z^T Y = 0$, $Z^T Z \neq I$, $Y^T Y \neq I$) defined in (34)–(35) and (39)–(41) is more numerically stable than the coordinate decomposition defined in (34)–(35) and (36)–(38) and has other desirable properties in the context of rSQP [**?**].

Solves with $R$ in (40) are performed using the Sherman-Morrison-Woodbury formula [**?**] which gives

$$R^{-1} = (I - D\, S^{-1}\, D^T) C^{-1} \tag{42}$$

where $D = -C^{-1} N \in \mathcal{X}_D | \mathcal{X}_I$ and $S = I + D^T D \in \mathcal{X}_I | \mathcal{X}_I$ are explicitly computed, and the symmetric positive definite matrix $S$ is factored using a dense Cholesky method. Therefore, applying $R^{-1}$ only requires a solve with the basis matrix $C$ along with back-solving with the factor of $S$. However, the $n_I$ linear solves needed to form $D = -C^{-1} N$ and the $O((n-r)^2 r)$ dense linear algebra required to compute $D^T D$

can dominate the cost of the algorithm for NLPs with larger numbers of degrees of freedom $(n - r)$.

For larger $(n-r)$ if adjoint solves with $C^T$ are available, the coordinate decomposition ($Z^TY \neq 0$, $Z^TZ \neq I$, $Y^TY \neq I$) defined in (34)–(35) and (36)–(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 and ill-conditioning in the basis matrix $C$ that can result in greatly degraded performance and even failure of an rSQP algorithm. See the MOOCHO option `quasi_range_space_matrix` in Section 0.6.1 for selecting between the orthogonal and the coordinate decompositions.

It is also important to note that MOOCHO can be used to solve non-equality-constrained optimization problems ($m = 0$) and square nonlinear equations ($m = n$). A non-equality-constrained optimization problem is handled by using $Z = I$ and $Y = \{\text{empty}\}$. A square nonlinear problem is handled using $Z = \{\text{empty}\}$ and $Y = I$. MOOCHO configures simpler algorithms in these two cases.

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 the reduced Hessian $B \approx Z^TWZ$ using quasi-Newton methods (e.g. BFGS) [**?**]. 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 (25)) [**?**]. When $(n - r)$ is large, limited-memory quasi-Newton methods can 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 classes of NLPs, 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 [**?**]). See the option `quasi_newton` in Section 0.6.1. Other options include solving for system with the exact reduced Hessian $B = Z^TWZ$ iteratively (using CG for instance) which only requires matrix-vector products with $W$ which can be computed efficiently using automatic differentiation (for instance) in some cases [**?**]. However, MOOCHO currently does not have any such algorithms implemented at this time.

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 (24) can be solved using direct or iterative solvers and the reduced QP subproblem in (25)–(27) can be solved using a variety of methods (active set vs. interior point) and software [**?**].

### 0.2.4 General Inequalities, Slack Variables and Basis Permutations

Up to this point, only simple variable bounds in (3) have been considered and the SQP and rSQP algorithms have been presented in this context. However, the actual underlying NLP may include general inequalities and take the form

$$\begin{aligned} \min \quad & \breve{f}(\breve{x}) && (43)\\ \text{s.t.} \quad & \breve{c}(\breve{x}) = 0 && (44)\\ & \breve{h}_L \leq \breve{h}(\breve{x}) \leq \breve{h}_U && (45)\\ & \breve{x}_L \leq \breve{x} \leq \breve{x}_U && (46) \end{aligned}$$

where:

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

NLPs with general inequalities are converted into the standard form by the addition of slack variables $\breve{s}$ (see (50)). After the addition of the slack variables, the concatenated variables and constraints are then permuted (using permutation matrices $Q_x$ and $Q_c$) according to the current basis selection into the ordering in (1)–(3). The exact mapping from (43)–(46) to (1)–(3) is

$$\begin{aligned} x &= Q_x \begin{bmatrix} \breve{x} \\ \breve{s} \end{bmatrix} && (47)\\ x_L &= Q_x \begin{bmatrix} \breve{x}^L \\ \breve{h}^L \end{bmatrix} && (48)\\ x_U &= Q_x \begin{bmatrix} \breve{x}_u \\ \breve{h}_u \end{bmatrix} && (49)\\ c(x) &= Q_c \begin{bmatrix} \breve{c}(\breve{x}) \\ \breve{h}(\breve{x}) - \breve{s} \end{bmatrix}. && (50) \end{aligned}$$

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 $\nabla c$ takes the form

$$\nabla c = \begin{bmatrix} \nabla \breve{c} & \nabla \breve{h} \\ & -I \end{bmatrix} \tag{51}$$

One question to ask is how the Lagrange multipliers for the original constraints can be extracted from the optimal solution $(x, \lambda, \nu)$ that satisfies the optimality conditions in (7)–(13)? First, consider the linear dependence of gradients optimality condition for the NLP formulation in (43)–(46)

$$\nabla_{\breve{x}} \breve{L}(\breve{x}^*, \breve{\lambda}^*, \breve{\lambda}_I^{\,*}, \breve{\nu}^*) = \nabla \breve{f}(\breve{x}^*) + \nabla \breve{c}(\breve{x}^*)\breve{\lambda}^* + \nabla \breve{h}(\breve{x}^*)\breve{\lambda}_I^{\,*} + \breve{\nu}^* = 0. \tag{52}$$

To see how the Lagrange multiples $\lambda^*$ and $\nu^*$ can be used to compute $\breve{\lambda}^*$, $\breve{\lambda}_I^{\,*}$ and $\breve{\nu}^*$ one simply has to substitute (47) and (50) with $Q_x = I$ and $Q_c = I$, for instance, into (7) and expand as follows

$$\begin{aligned} \nabla_x L(x, \lambda, \nu) &= \nabla f + \nabla c \lambda + \nu \\ &= \begin{bmatrix} \nabla \breve{f} \\ 0 \end{bmatrix} + \begin{bmatrix} \nabla \breve{c} & \nabla \breve{h} \\ & -I \end{bmatrix} \begin{bmatrix} \lambda_{\breve{c}} \\ \lambda_{\breve{h}} \end{bmatrix} + \begin{bmatrix} \nu_{\breve{x}} \\ \nu_{\breve{s}} \end{bmatrix} \\ &= \begin{bmatrix} \nabla \breve{f} + \nabla \breve{c} \lambda_{\breve{c}} + \nabla \breve{h} \lambda_{\breve{h}} + \nu_{\breve{x}} \\ -\lambda_{\breve{h}} + \nu_{\breve{s}} \end{bmatrix}. \end{aligned} \tag{53}$$

By comparing (52) and (53) 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 (53) 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 what one would have expected). Zero multiplier values means that constraints will not impact the optimality conditions or the Hessian of the Lagrangian.

The basis selection shown in (22) and (31) is determined by the permutation matrices $Q_x$ and $Q_c$ and these permutation matrices can be partitioned as

$$Q_x = \begin{bmatrix} Q_{xD} \\ Q_{xI} \end{bmatrix} \tag{54}$$

$$Q_c = \begin{bmatrix} Q_{cD} \\ Q_{cU} \end{bmatrix}. \tag{55}$$

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 column 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 basis selection for the transformed NLP (with $Q_c = I$) given by

$$Q_x = \begin{bmatrix} \breve{Q}_{xD} & \\ & I \\ & \breve{Q}_{xI} \end{bmatrix} \tag{56}$$

$$C = \begin{bmatrix} (\breve{Q}_{xD}\nabla\breve{c})^T & \\ (\breve{Q}_{xD}\nabla\breve{h})^T & -I \end{bmatrix} \tag{57}$$

$$N = \begin{bmatrix} (\breve{Q}_{xI}\nabla\breve{c})^T \\ (\breve{Q}_{xI}\nabla\breve{h})^T \end{bmatrix} \tag{58}$$

could always be used regardless of the properties or implementation of $\nabla\breve{h}$.

Notice that basis matrix in (57) 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

$$D = -\begin{bmatrix} (\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^T \\ (\breve{Q}_{xD}\nabla\breve{h})^T(\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^T - (\breve{Q}_{xI}\nabla\breve{h})^T \end{bmatrix}. \tag{59}$$

Note that if the forward sensitivities $(\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^T$ are computed up front then there is little extra cost in forming this decomposition after the addition of general inequality constraints. The structure of (59) is significant in the context of active-set QP solvers that solve the reduced QP subproblem in (25)–(27) using a variable-reduction null-space decomposition. When an implicit adjoint method is used, a row of $D$ corresponding to a general inequality constraint only has to be computed if the slack for the constraint is at a bound. Also note that the above transformation does not increase the total 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 an active-set rSQP algorithm if the forward/direct sensitivity method is used or if these constraints are not active when using the adjoint method.

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. For general NLPs solved in serial, MOOCHO provides support for general inequality constraints and will automatically add slack variables and perform the needed basis permutations and partitioning.

Most of the steps in an SQP algorithm do not need to know that there are general inequalities in the underlying NLP formulation but some steps may, such as globalization methods and basis selection computations. Therefore, those steps in an SQP

algorithm that need access to this information are allowed more detailed access of the underlying NLP in a limited manner.

Now that the basic mathematical context for MOOCHO is in place, we move on to a description of the basic software infrastructure for MOOCHO.

## 0.3 Basic Software Architecture of MOOCHO

MOOCHO is implemented in C++ using advanced object-oriented software engineering principles. However, using MOOCHO to solve certain types of NLPs does not require any deep knowledge of object-orientation or C++. By copying and modifying example programs it should be possible for a non-C++ expert to implement and solve many different NLPs using MOOCHO. However, solving more advanced NLPs which utilize specialized application-specific data structures and linear solvers does require more detailed knowledge of C++ and some knowledge of object orientation. Although, the included example applications should provide a straightforward road map for getting started with such an application. For simulation-constrained optimization based on parallel Epetra[1]-compatible data structures, using MOOCHO requires almost know deep knowledge of MOOCHO's interfaces at all.

### 0.3.1 High-Level Object Diagram for MOOCHO

There are many different ways to present MOOCHO. Here, we take a top down approach.

Figure 1 shows a high-level object diagram of a MOOCHO 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. Closely associated with the NLP object is a *`BasisSystem`* object. The *`BasisSystem`* object is used to specify and specialize the implementation of the basis matrix $C$. This *`BasisSystem`* object is used by variable-reduction null-space decompositions. 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. By creating a specialized NLP subclass (and the associated linear algebra and *`BasisSystem`* subclasses) an advanced user can fully take over the implementation of all of the major linear-algebra computations in a MOOCHO algorithm. This includes having full freedom to choose the data structures for all of the vectors and the matrices $A$, $C$, $N$, $W$ 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 MOOCHO code has any concept of parallelism. The linear

[1]Epetra is a Trilinos package for distributed-memory vectors and matrices.

**Figure 1.** UML object diagram : Course grained object diagram for MOOCHO

algebra objects associated with the `NLP` and *`BasisSystem`* objects define the foundation for every major computation in a simulation-constrained optimization (SCOPT) algorithm. The exact requirements of the application and the details of the NLP and linear algebra interfaces that satisfy these requirements are discussed in Section 0.4. A complete infrastructure for parallel simulation constrained optimization is supported through the MOOCHO/Thyra adapters described in the MOOCHO Reference Manual [**?**]. Directly interacting with the MOOCHO linear algebra and NLP interfaces is not recommended. Instead, users are strongly encouraged to define there SCOPT NLPs through Thyra.

Once a user has developed *`NLP`* and *`BasisSystem`* classes (i.e. indirectly using `Thyra` and `Thyra::ModelEvaluator`) for their specialized application, an *`NLP`* object can be passed on to a `MoochoSolver` object. The `MoochoSolver` class is a convenient "facade" (see [**?**]) 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 `MoochoSolver` 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 1 also shows the course grained layout of a MOOCHO algorithm. An advanced user can solve even the most complex specialized NLP without needing to understanding how these algorithmic objects work together to implement an optimization algorithm. One only needs to understand the algorithmic framework in order to tinker with the optimization algo-

rithms themselves. Understanding the underlying algorithmic framework is crucial for algorithm developers through.

While MOOCHO offers complete flexibility to solve many different types of specialized NLPs in diverse application areas such as dynamic optimization and control (see [**?**]) and PDEs (see [**?**]) it can also be used to solve more generic NLPs such as are supported by modeling systems like GAMS [**?**] or AMPL [**?**]. For serial NLPs which can compute explicit Jacobian entries for $A$, all that a user needs to do is 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 can use one of a number of different dense or sparse direct linear solvers to implement all of the required functionality. A simple example NLP that derives from `NLPSerialPreprocessExplJac` is described in the MOOCHO Reference Manual [**?**].

## 0.4 Overview of NLP and Linear-Algebra Interfaces

All of the high-level optimization code in MOOCHO is designed to allow arbitrary implementations of the linear-algebra objects. It is the NLP object that defines the basic foundation for all of the linear algebra used by a SCOPT optimization algorithm. The NLP object accomplishes this by exposing a set of abstract linear algebra objects. Before the specifics of the NLP and linear algebra interfaces are described, the specific requirements for SCOPT optimization algorithms are described in Section 0.4.1. This is followed by the descriptions of the linear algebra and NLP interfaces in Sections 0.4.2 and 0.4.3 respectively.

### 0.4.1 Basic Application Requirements for Simulation-Constrained Optimization with MOOCHO

The requirements for large-scale gradient-based SCOPT optimization algorithms implemented in MOOCHO can broken down into three different levels: *direct SCOPT*, *adjoint SCOPT*, and *full-Newton SCOPT*. These three levels represent different levels of intrusiveness and functionality from the underlying application that are used the implement the NLP.

The most basic level of requirements are for *direct SCOPT* methods. This level only requires forward linear solves with the basis matrix for specific right-hand-side vectors. Most applications that utilize an exact Newton-type method for solving the simulation problem can compute the solutions to these linear systems. Both the orthogonal and the coordinate variable-reduction null-space decompositions can be

| Optimization level | Application requirements (additive between levels) |
|---|---|
| *Direct SCOPT* | Evaluation of objective: $x \in \mathcal{X} \rightarrow f \in \mathbb{R}$<br>Evaluation of constraints residual: $x \in \mathcal{X} \rightarrow c \in \mathcal{C}$<br>Evaluation of objective gradient: $x \in \mathcal{X} \rightarrow \nabla f \in \mathcal{X}$<br>Evaluation of direct sensitivity matrix: $D = -C^{-1}N \in \mathcal{X}_D \vert \mathcal{X}_I$<br>Evaluation of Newton step: $p_y = -C^{-1}c(x) \in \mathcal{X}_D$ |
| *Adjoint SCOPT* | Ability to perform mat-vec products:<br>$p = Aq,\ q = A^T q,$ for $q \in \mathcal{C}, p \in \mathcal{X}$<br>Ability to solve linear systems:<br>$p = C^{-1}q,\ q = C^{-T}q,$ for $q \in \mathcal{C}_d, p \in \mathcal{X}_D$ |
| *Full-Newton SCOPT* | Ability to perform mat-vec products:<br>$p = Wq,$ for $q \in \mathcal{X}, p \in \mathcal{X}$ |

**Table 1.** Minimum Application Requirements for levels of invasiveness for Simulation-Constrained Optimization (SCOPT).

implemented with just the quantities $D = -C^{-1}N$ and $p_y = C^{-1}c$. In addition, many different types of globalization methods can also be used (both line search and trust region methods complete with second-order corrections for the constraints).

The next level of requirements is for *adjoint SCOPT* methods. This level requires the ability to perform mat-vec products and linear solves with the non-transposed and transposed basis $C$ and non-basis $N$ matrices with arbitrary vectors. More efficient and robust optimization algorithms can be implemented using this functionality. For example, the ability to solve for transposed systems with the basis matrix $C$ provides the ability to compute estimates for the Lagrange multipliers $\lambda$ and the ability to compute reduced gradients at a cost independent of the number of optimization parameters.

The highest level of requirements is for *full-Newton SCOPT* methods. The minimum requirements for these methods is the ability to compute mat-vec products with an approximation of the Hessian of the Lagrangian $W$. MOOCHO currently does not implement a full-Newton (i.e. full-space) SQP method.

The last set of requirements for SCOPT methods is the requirements on vectors. There is a great diversity of specialized vector or array operations that optimization methods must perform. This difficult set of requirements is handled by a design for vector reduction/transformation operators (RTOp) which is described in [**?**] and mentioned in Section 0.4.2.

Note that this set of requirements satisfies all the requirements of the SCOPT optimization interfaces described in [**?**].

### 0.4.2 Overview of `AbstractLinAlgPack`: Interfaces to Linear Algebra Objects

The linear algebra interfaces described in this section serve two roles. The first role is to abstract the linear algebra objects associated with the NLP interface such as the vector objects for the unknowns $x$, the residual of the constraints $c$ and the gradient of the objective function $\nabla f$; and the matrix objects for the gradients of the constraints $A$, the Hessian of the Lagrangian $W$ and the variable-reduction matrices $C$, $N$, $E$ and $F$. The second role of the linear algebra interfaces is to abstract objects that are specific to the optimization algorithms such as for quasi-Newton approximations for the Hessian and the reduced Hessian of the Lagrangian. The objects from the latter group are obviously dependent on objects from the former group in various ways. This latter role greatly increases the complexity and functionality of these interfaces.

Figure 2 shows a UML class diagram of the basic linear algebra abstractions. The foundation for all the linear algebra is a vector space. A vector space object is represented though an abstract interface called *`VectorSpace`*. A *`VectorSpace`* object

**Figure 2.** UML class diagram : `AbstractLinAlgPack`, abstract interfaces to linear algebra

primarily acts as an "abstract factory" [**?**] 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. Every vector space object is also equipped with an scalar (i.e. inner) product that is used to introduce scaling into the problem as described in [**?**]. Just because two vectors from different vector spaces have the same dimension (and the same inner product) does not automatically imply that the implementations 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") and are therefore not easily compatible in the RTOp sense. This is an important concept to remember.

Vector implementations are abstracted behind interfaces. The vector interface is broken up into two levels: *Vector* and *VectorMutable*. The *Vector* interface is an immutable interface where vector objects can not be changed by the client. The *VectorMutable* interface extends the *Vector* 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 to perform arbitrary reductions and transformations (see the method *apply_op(...)*). The ability to write these types of user-defined operators is critical to the implementation of advanced optimization algorithms [**?**]. A single RTOp application method is the only method that a vector implementation is required to provide (in addition to some trivial methods such as returning the vector space object) 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 subvectors associated with 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 provided by default by every vector implementation through default view subclasses (see the class `VectorMutableSubview`) 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 in a serial setting and the implementation of the compact LBFGS matrix in a distributed parallel setting. 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. the NLP interface).

The basic matrix (i.e. linear operator) interfaces are also shown in Figure 2. The *MatrixOp* interface is for general rectangular matrices. Associated with any *MatrixOp* 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 infinite-dimensional linear operators. 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 *MatrixOp* object then the vectors $y$ and $x$ would have to lie in the column and row spaces respectively in order to perform the matrix-vector product $y = Ax$. Note that despite name, a *MatrixOp* object does not provide any type of efficient access to matrix elements. If explicit matrix elements are required, then the matrix object can support other defined matrix interfaces in order to extract the elements in a sparse (see the interfaces *MatrixExtractSparseElements* and *MatrixConvertToSparse*) or dense (see the interfaces *MatrixOpGetGMS...*) format.

These matrix interfaces go beyond what most other abstract matrix/linear-operator interfaces have attempted. Other abstract linear-operator interfaces only allow the forward applications of $y = Ax$ or the transpose (adjoint) $y = A^T x$ for vector-vector mappings. In addition to this basic functionality, every *MatrixOp* object can pro-

vide arbitrary subviews as *MatrixOp* objects through the *sub_view(...)* methods. These methods have default implementations based on default view subclasses which, fundamentally, is 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 (33) given a Jacobian object `Gc` for $\nabla c$. These matrix interfaces also allow much more general types of linear-algebra operations. The *MatrixOp* interface allows the client to perform level 1, 2 and 3 BLAS operations (see [???] 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 *MatrixOp* object can appear on the left-hand-side. This adds a whole set of issues (i.e. multiple dispatch) 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 *MatrixOp* 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.

Of all of the functionality in the *MatrixOp* interface, the only pure virtual method is the method for the level-2 BLAS operation for matrix-vector multiplication. All other methods have reasonable default implementations based on this one method. Therefore, generating a new concrete *MatrixOp* subclass is usually fairly easy. If the default implementations of some of the other methods are found to be inefficient in important cases, then they can be overridden to provide better, more specialized implementations. This design allows for a pay-as-you-go approach to developing implementations of linear algebra objects and this philosophy applies to all of the linear algebra interfaces in `AbstractLinAlgPack` as well.

Several specializations of the *MatrixOp* interface are also required in order to implement an advanced optimization algorithm. All symmetric matrices are abstracted by the *MatrixSymOp* 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.

The specialization *MatrixOpNonsing* is for nonsingular square matrices that can be used to solve for linear systems. As a result, the level-2 and level-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 *MatrixSymOpNonsing* 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.

Figure 2 shows two other specializations of the `MatrixOp` interface that have not been discussed yet, `MultiVector` and `MultiVectorMutable`. A multi-vector is special kind of matrix where access the rows, columns and/or diagonals may be permitted as *Vector* and *VectorMutable* views. The primary role for a multi-vector object is the creation of tall, thin matrices where each column vector is accessible. It is these types of *VectorMutable* objects that are created by the *create_members(num_vecs)* method on *VectorSpace*. The row space for these types of *MultiVectorMutable MatrixOp* objects are assumed to be small, serial vector spaces in all cases. The ability of a *VectorSpace* object to create *MultiVectorMutable* objects with an arbitrary number of columns implies that every *VectorSpace* object can create other small serial *VectorSpace* objects of arbitrary dimension. In order to directly allow this functionality, the method *small_vec_spc_fcty()* (not shown) returns a factory object for creating these vector spaces. Since *MultiVector* is a type of *MatrixOp*, it can be passed into all of the level-3 BLAS methods on *MatrixOp* and *MatrixOp-Nonsing*. By passing a *MultiVectorMutable* object (from the correct vector space) as the target object for any of these linear algebra operations guarantees that the operation will be supported since it can always be performed, column by column, using the level-2 BLAS methods.

A major part of an rSQP algorithm, based on a variable-reduction null-space decomposition, is the selection of a basis. The fundamental abstraction for this task is the *BasisSystem* interface (as first introduced in Figure 1). The *update_basis()* method takes the rectangular Jacobian `Gc` ($\nabla c$) and returns a *MatrixOpNonsing* object for the basis matrix $C$. This interface assumes that the variables are already sorted according to (31). For many applications, the selection of the basis is known

*a priori* (e.g. simulation-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 an optimization algorithm. The ***BasisSystemPerm*** specialization allows the optimization algorithm 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 (***set_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. The protocol for handling basis changes is somewhat complicated and is beyond the scope of this discussion. Note that the ***BasisSystemPerm*** interface is optional and does not have to be supported by an application.

It is likely that a future version of MOOCHO might use a set of linear algebra interfaces that is directly based on the new Thyra interfaces that are part of Trilinos. If this happens, many of these interfaces will be phased out and/or transitioned to Thyra.

### 0.4.3 Overview of `NLPInterfacePack`: Interfaces to Nonlinear Programs

The hierarchy of NLP interfaces that all MOOCHO optimization algorithms are based on is shown in Figure 3. 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 model.

The base-level NLP interface is called ***NLP*** which 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 ***Vector*** objects `x_init`, `xl` and `xu` respectively. This interface also provides access to ***Permutation*** objects `P_var` and `P_var` for permutation matrices $Q_x$ and $Q_c$, respectively. These matrices are used to permute from the original order of variables and constraints to the current basis selection (see Section 0.2.4).

The ***NLP*** interface allows clients to evaluate just the zero-order quantities $f(x) \in \mathbb{R}$ and $c(x) \in \mathcal{C}$ as scalar and ***VectorMutable*** objects respectively. This is useful since many different steps in an optimization algorithm do not require derivatives for the problem functions. Examples include several different line search and trust region globalization methods (i.e. Filter and exact merit function). Non-gradient-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***

**Figure 3.** UML class diagram : `NLPInterfacePack`, abstract interfaces nonlinear programs

interface can also be used for unconstrained optimization (i.e. $|\mathcal{C}| = m = 0$) or for a system of nonlinear equations (i.e. $|\mathcal{X}| = n = |\mathcal{C}| = m$).

As mentioned in Section 0.2.4, some parts of an optimization algorithm can benefit greatly from knowing about general inequality constraints and become less effective when these constraints are converted to equalities using slack variables. These steps in the optimization algorithm can compute the quantities $\breve{c}(\breve{x})$ and $\breve{h}(\breve{x})$ independently and access the bounds $\breve{h}_L$ and $\breve{h}_U$ through the ***NLP*** interface as well.

The next level of NLP interface is ***NLPObjGrad***. This interface simply adds the ability to compute the gradient of the objective function $\nabla f(x) \in \mathcal{X}$ as a ***VectorMutable*** object `Gf`. For many applications, it is far easier and less expensive to compute derivatives for the objective function than it is for the constraints. That is why this functionality is considered more general than sensitivities for the constraints[2] and is therefore higher in the inheritance hierarchy than interfaces the include derivatives for $\nabla c$.

Derivatives for the constraints $\nabla c$ are broken up into two separate interfaces based on

[2]It turns out that the assumption that getting derivatives of the objective function is easier than getting derivatives for the constraints is not true in general, especially for simulation-constrained optimization problems. Therefore, these interfaces will likely be reworked at some point in the future to better reflect reality.

the requirements for *direct SCOPT* verses *adjoint SCOPT* methods. These interfaces represent the different capabilities of the underlying application code.

For applications that can only satisfy the requirements for *direct SCOPT* there is the ***NLPDirect*** interface. As the name implies, the ***NLPDirect*** 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 ***NLPDirect*** interface.

The ***NLPFirstOrder*** interface is for applications can implement the requirements for *adjoint SCOPT* methods. This NLP interface assumes that the application can, at the very least, form and maintain a ***MatrixOp*** object `Gc` for the gradient of the constraints $\nabla c$. Recall that this implies that mat-vec products with both $\nabla c^T$ <u>and</u> $\nabla c$. Note that operations of the form $p = \nabla c^T q$ can always be approximated using directional finite differences (i.e. $p = \nabla c^T\, q \approx \lim_{\epsilon \to 0}(c(x + \epsilon q) - c(x))/\epsilon)$) but operations of the form $q = \nabla c\, p$ can not. Therefore, this interface can not simply be approximated using finite differences. However, the reverse mode of AD can generally be used to implement products of the form $q = \nabla c\, p$ in an efficient manner without having to actually form the matrix object $\nabla c$ first [**?**]. In order to fully support the requirements for *adjoint SCOPT* methods, a ***NLPFirstOrder*** object must supply a ***BasisSystem*** object that may be specialized for the application's `Gc` matrix object.

The highest level of requirements for *full-Newton SCOPT* methods is satisfied by the ***NLPSecondOrder*** interface. This NLP interface allows the optimization algorithm to compute a ***MatrixSymOp*** 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 (in some cases). Many other possibilities exist but the best approach will be very much application dependent. MOOCHO currently does not support a full-space SQP algorithm and therefore there are currently no facilities for solving linear system with the KKT matrix $W$. However, in the past, this interface has been used within MOOCHO to compute the exact reduced Hessian and use it instead of a quasi-Newton approximation.

Figure 3 shows another NLP interface that has not been discussed yet, ***NLPVarReductPerm***. This interface allows the NLP object and MOOCHO optimization algorithm to collaborate in changing the basis and defining new permutations for the variables and the constraints shown as the permutation matrices `P_var` and `P_var` on the ***NLP*** interface respectively. This interface is considered a mix-in interface that concrete NLP subclasses should only support if changing the basis selection is possible. If an NLP subclass supports the ***NLPVarReductPerm*** and ***NLPFirstOrder*** interfaces, it is also required that the ***BasisSystem*** object exposed by the ***NLPFirstOrder*** interface also support the ***BasisSystemPerm*** interface. A carefully designed collaboration between the ***NLPVarReductPerm*** and ***BasisSystemPerm*** interfaces, which is mediated by a MOOCHO algorithm, makes it possible for the basis selection to change

during the course of an algorithm. This functionality will generally only be supportable by NLPs that provide explicit Jacobian entries and use direct linear solvers. For most specialized applications, the selection of the basis is fixed and unchangeable (e.g. simulation-constrained optimization). Therefore, an NLP subclass does not have to support the *NLPVarReductPerm* or *BasisSystemPerm* interfaces to be used with MOOCHO.

In summary, the *NLP*, *NLPDirect*, *NLPFirstOrder* and *NLPSecondOrder* interfaces represent the four different levels of invasiveness to applications for optimization. The *NLP* interface without equality constraints can used to implement basic NAND (i.e. black-box) optimization algorithms while on the other extreme the *NLPSecond-Order* interface can be used to implement fully-coupled invasive SCOPT methods with access to second derivatives[3].

## 0.5 Defining Optimization Problems

Optimization problems for MOOCHO can be defined in primarily two different ways. First, a general NLP with explicit first-derivative entries can be defined by creating a subclass of `NLPInterfacePack::NLPSerialPreprocessExplJac`. This type of NLP can only be solved using a single process (i.e. no MPI parallelism) and a sparse direct linear solver must be used (i.e. MA28). For this type of NLP, there is not need for the user to partition the variables into dependent variables (i.e. state variables) and independent variables (i.e. optimization parameters).

The second type of NLP that can be solved using MOOCHO are simulation-constrained NLPs where the basis section is known up front. For these types of NLPs, it is recommended that the NLP be specified through the `Thyra::ModelEvaluator` interface (or the `EpetraExt::ModelEvaluator` interface for Epetra-based applications) and this provides access to a significant linear solver capability through Trilinos. These types of NLPs can be solved in single program multiple data (SPMD) mode in parallel on a massively parallel computer. This is the recommended interface for SCOPT applications to adopt and this will most likely drive the development of MOOCHO in the near future.

See the MOOCHO Reference Manual [**?**] for examples of these different types of NLPs.

## 0.6 Basic properties of MOOCHO Algorithms

All MOOCHO algorithms share a few different properties that are described below.

[3]Again, in theory these interfaces can support full-space Newton SQP methods but this has not been implemented in MOOCHO at this point

### 0.6.1 Solver options

Various options can be set in a flexible and user friendly format. Using this format, options are clustered into different "options groups". An example option file containing many off the typical options that a user would set is shown below:

```
begin_options

options_group NLPSolverClientInterface {
    max_iter = 20;
    max_run_time = 2.0; *** In minutes
    opt_tol = 1e-2;
    feas_tol = 1e-7;
*    journal_output_level = PRINT_NOTHING;
*    journal_output_level = PRINT_BASIC_ALGORITHM_INFO;
    journal_output_level = PRINT_ALGORITHM_STEPS;
*    journal_output_level = PRINT_ACTIVE_SET;
*    journal_output_level = PRINT_VECTORS;
*    journal_output_level = PRINT_ITERATION_QUANTITIES;
*    null_space_journal_output_level = DEFAULT;
*    null_space_journal_output_level = PRINT_ACTIVE_SET;
*    null_space_journal_output_level = PRINT_VECTORS;
    null_space_journal_output_level = PRINT_ITERATION_QUANTITIES;
    journal_print_digits = 10;
*    check_results = true;  *** (costly?)
    check_results = false; *** [default]
    calc_conditioning = true;
    calc_matrix_norms = true;  *** (costly?)
    calc_matrix_info_null_space_only = true;  *** (costly?)
}

options_group DecompositionSystemStateStepBuilderStd {
*    null_space_matrix = AUTO;     *** Let the solver decide [default]
    null_space_matrix = EXPLICIT; *** Compute and store D = -inv(C)*N explicitly
*    null_space_matrix = IMPLICIT; *** Perform operations implicitly 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)
}

options_group NLPAlgoConfigMamaJama {
*    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;  *** [default] Use the Filter line search method
}

end_options
```

These and many other options may be included in the `Moocho.opt` file. See the MOOCHO Reference Manual [**?**] for the listing of all of the valid options with some documentation. Note that the mathematical description itself in Section 0.2 is critical in understanding and interpreting these options.

A skeleton for a `Moocho.opt` file can be created using the `generate-opt-file.pl` Perl script [**?**]. This script can be run from the source tree and it also gets installed

in `$TRILINOS_INSTALL_DIR/tools/moocho`.

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

### 0.6.2 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 MOOCHO 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 MOOCHO 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 MOOCHO algorithm can be produced by setting the option `MoochoSolver{print_algo=true}`, where this is shorthand for the `print_algo` option in the `MoochoSolver` options group. With this option set to `true`, the algorithm description is printed to the `MoochoAlgo.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 .1. Each iteration quantity name in the algorithm printout has `'_k'`, `'_kp1'` or `'_km1'` appended to the end of it to designate the iterations $(k)$, $(k+1)$ or $(k-1)$ respectively, for which the quantity was calculated. Much of the difficulty in understanding an algorithm, whether in mathematical notation or implemented in source code, is in 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 `NLPAlgo` object (through functionality in the base class `IterationPack::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 an rSQP algorithm generated by the "MamaJama" configuration[4] is shown in [**?**]. 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.

For a more detailed look at the output files `MoochoAlgo.out` and `MoochoJournal.out` see the MOOCHO Reference Manual [**?**].

### 0.6.3 Algorithm Summary and Timing

In addition to the more detailed information that can be printed to the file `MoochoJournal.out`, summary information about each MOOCHO iteration is printed to the file `MoochoSummary.out`. Also, if the option `MoochoSolver{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 a 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. See the MOOCHO Reference Manual [**?**] for an example of a `MoochoSummary.out` file.

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

### 0.6.4 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 excessive round-off) 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 NLP that the user has written 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

[4] The name MamaJama was used for the very first algorithm configuration class and was meant to be a "do all" configuration class for active-set rSQP algorithms.

very tedious and time consuming.

Fortunately, it is possible to partially 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:

1. Errors in the NLP implementation (e.g. bad derivatives)
2. Errors in the user specialized parts of the optimization algorithm (e.g. a bad specialized ***BasisSystem*** object)
3. Errors in the core optimization code (e.g. errors in mathematics, programming logic, or memory usage)
4. Or, errors in the compiler or runtime environment (e.g. excessive roundoff due to overly aggressive compiler optimizations).

There are many ways to make a mistake in coding the NLP interface. For instance, assuming the user's 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 $\nabla c$ is not consistent with $c(x)$ but only in some regions. The matrix $\nabla c$ 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 $\nabla c$, the entire matrix could be approximately computed by finite differences of course and then compared to the $\nabla c$ 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 derivatives $\nabla f$ and $\nabla c$ by finite differences is an option but it must be explicitly turned on (see the option `NLPFirstDerivTester{fd_testing_method=FD_COMPUTE_ALL}`). As a compromise, by default, directional finite differencing can be used to show that $\nabla c$ is not consistent with $c(x)$, but can not strictly prove that $\nabla c$ is completely correct. This works as follows. The optimization algorithm asks the NLP interface to compute $\nabla c_k$ at a 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 $h \approx 10^{-5}$ (where $h$ can be set by the user through the options in the options group `CalcFiniteDiffProd`). Then the matrix-vector product $t_2 = \nabla c_k v$ would be computed using the $\nabla c_k$ matrix object computed by the NLP interface and the resultant vectors $t_1$ and $t_2$ are then compared. Even if the user did 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 (and truncation errors in the finite-difference computation). Therefore, we need some type of measure of how well $t_1$ and $t_2$ compare. For every such test in MOOCHO there are defined an error tolerance `error_tol` and warning tolerance `warning_tol` that are adjustable by the user. Any relative error greater than `error_tol` will cause the optimization algorithm to be terminated with an error message printed. Any relative error greater than `warning_tol` will 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 `NLPFirstDerivTester` and `CalcFiniteDiffProd` options groups and up to fourth-order central differences are supported (and are the default), which yield very accurate derivatives in many cases. 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 or infeasible, the SQP algorithm will determine this (but the error message produced by the algorithm may not be able to state exactly the cause of the problem).

Every major computation in a SQP 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 ***`MatrixOpNonsing`*** can be checked by computing $q = Ax$ and then comparing $q$ to $b$. Computations can also be validated for the null-space decomposition (see `DecompositionSystemTester`) and QP solver (see `QPSolverRelaxedTester`) 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 MOOCHO, but more is needed for more complete validation. The option `NLPSolverClientInterface{check_results=true}` will turn on all such runtime checks and, with default settings, should be only increase the cost of the algorithm by a constant factor, independent of the size of the problem.

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

### 0.6.5 Algorithm Interruption

All MOOCHO algorithms can be interrupted at any time while the algorithm is running and result in a graceful termination, even for parallel runs with MPI. When running in interactive mode (i.e. the user has access to standard in and standard out in the console) then typing `Ctrl-C` will cause the algorithm to pause and allow the user to enter termination criteria on the standard input stream. Or, a MOOCHO algorithm can be interrupted without access to standard in or standard out (i.e. when running in batch mode) by setting up an interrupt file. This feature allows the client to terminate a MOOCHO algorithm at any time and still result in a graceful exit where the current status of the solution is compiled and returned to the user (through the NLP interface). See the MOOCHO Reference Manual [**?**] for more details.

## 0.7 Summary

MOOCHO currently implements several variates of reduced-space quasi-Newton successive quadratic programming methods. MOOCHO is written in C++ and can be used to address very large-scale optimization problems. MOOCHO is distributed as part of the Trilinos [**?**] collection. Both general serial problems and massively parallel simulation-constrained problems can be addressed with MOOCHO. Parallel simulation-constrained problems can be represented through Thyra and can utilize a great deal of the parallel (and serial) linear solver capability present through Trilinos.

# .1 MOOCHO Equation Summary and Nomenclature Guide

**Standard NLP Formulation**

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

where:

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

**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:

$$\begin{aligned} & \lambda \in \mathcal{C} \\ & \nu \equiv \nu_U - \nu_L \in \mathcal{X} \end{aligned}$$

**Full-Space QP Subproblem**

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

where:

$$\begin{aligned} & d = x_{k+1} - x_k \in \mathcal{X} \\ & g = \nabla f(x_k) \in \mathcal{X} \\ & W = \nabla^2_{xx} L(x_k, \lambda_k) \in \mathcal{X} | \mathcal{X} \\ & M(\eta) \in \mathbb{R} \to \mathbb{R} \\ & A = \nabla c(x_k) \in \mathcal{X} | \mathcal{C} \\ & c = c(x_k) \in \mathcal{C} \end{aligned}$$

**Null-Space Decomposition**

$$\begin{aligned} & Z \in \mathcal{X} | \mathcal{Z} && \text{s.t. } (A_d)^T Z = 0 \\ & Y \in \mathcal{X} | \mathcal{Y} && \text{s.t. } \begin{bmatrix} Y & Z \end{bmatrix} \text{ nonsing} \\ & R \equiv [(A_d)^T Y] \in \mathcal{C}_d | \mathcal{Y} && \text{nonsing} \\ & U_z \equiv [(A_u)^T Z] \in \mathcal{C}_u | \mathcal{Z} \\ & U_y \equiv [(A_u)^T Y] \in \mathcal{C}_u | \mathcal{Y} \\ & d = (1 - \eta) Y p_y + Z p_z \end{aligned}$$

where:

$$\begin{aligned} & p_z \in \mathcal{Z} \subseteq \mathbb{R}^{(n-r)} \\ & p_y \in \mathcal{Y} \subseteq \mathbb{R}^r \end{aligned}$$

**Quasi-Normal (Range-Space) Subproblem**

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

**Tangential (Null-Space) Subproblem (Relaxed)**

$$\begin{array}{lll} & & \text{where:} \\ & & g_{qp} \equiv (g_r + \zeta w) \in \mathcal{Z} \qquad b_L \equiv x_L - x_k - Yp_y \in \mathcal{X} \\ \min & g_{qp}^T p_z + \tfrac{1}{2} p_z^T B p_z + M(\eta) & g_r \equiv Z^T g \in \mathcal{Z} \qquad b_U \equiv x_U - x_k - Yp_y \in \mathcal{X} \\ \text{s.t.} & U_z p_z + (1-\eta)u = 0 & w \equiv Z^T W Y p_y \in \mathcal{Z} \\ & b_L \le Zp_z - (Yp_y)\eta \le b_U & \zeta \in \mathbb{R} \\ & & B \approx Z^T W Z \in \mathcal{Z}|\mathcal{Z} \\ & & U_z \equiv [(A_u)^T Z] \in \mathcal{C}_u|\mathcal{Z} \\ & & U_y \equiv [(A_u)^T Y] \in \mathcal{C}_u|\mathcal{Y} \\ & & u \equiv U_y p_y + c_u \in \mathcal{C}_u \end{array}$$

**Variable-Reduction Decompositions**

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

where:

$$\begin{array}{l} C \in \mathcal{C}_d|\mathcal{X}_D \quad \text{(nonsing)} \\ N \in \mathcal{C}_d|\mathcal{X}_I \\ E \in \mathcal{C}_u|\mathcal{X}_D \\ F \in \mathcal{C}_u|\mathcal{X}_I \end{array}$$

**Coordinate**

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

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

$$\begin{array}{l} R = C \\ U_z = F - EC^{-1}N \\ U_y = E \end{array}$$

**Orthogonal**

$$\begin{array}{l} D \equiv -C^{-1}N \\ Z \equiv \begin{bmatrix} D \\ I \end{bmatrix} \\ Y \equiv \begin{bmatrix} I \\ -D^T \end{bmatrix} \\ R = C(I - DD^T) \\ U_z = F + ED \\ U_y = E - FD^T \end{array}$$

**Orthonormal (QR) Null-Space Decomposition**

$$Q^T A_d = \begin{bmatrix} R^T \\ 0 \end{bmatrix}$$

where:

$$Q = \begin{bmatrix} Y & Z \end{bmatrix} \in \mathcal{X}|(\mathcal{Y} \times \mathcal{Z}) \qquad \text{(nonsingular)}$$

### .1.1 Mathematical Notation Summary and MOOCHO Identifier Mapping

| Mathematical | MOOCHO | 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 $c(x)$ |
| $\mathcal{X} \in \mathbb{R}^n$ | `space_x` | Vector space for $x$ |
| $\mathcal{C} \in \mathbb{R}^m$ | `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 |
| $f(x)\|_x \in \mathbb{R}$ | `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 $c(x)$, $\nabla c = \begin{bmatrix} \nabla c_1 & \dots & \nabla c_m \end{bmatrix}$ |
| ***Lagrangian*** | | |
| $\lambda \in \mathcal{C}$ | `lambda` | Lagrange multipliers for $c(x) = 0$ |
| $\nu \in \mathcal{X}$ | `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 variables, $d = (x_{k+1})^+ - x_k$ |
| $\eta \in \mathbb{R}$ | `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 $c_u = c_{(r+1:m)}$ |
| $\mathcal{C}_d \in \mathbb{R}^r$ | `space_c .sub_space( con_decomp)` | Vector space for decomposed equalities $c_d$ |
| $\mathcal{C}_u \in \mathbb{R}^{(m-r)}$ | `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 |
| $\mathcal{Z} \in \mathbb{R}^{(n-r)}$ | `Z.space_rows()` | Null space. Accessed from the matrix object `Z`. |
| $\mathcal{Y} \in \mathbb{R}^{r}$ | `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` | Nonsingular: Equal to basis $C$ for coordinate decompositions |
| $U_z = [(\nabla c_u)^T Z] \in \mathcal{C}_u\|\mathcal{Z}$ | `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 |
| $\zeta \in \mathbb{R}$ | `zeta` | QP cross term damping parameter (descent for $\phi(x)$) |
| ***Global Convergence*** | | |
| $\alpha \in \mathbb{R}$ | `alpha` | Step length for $x_{k+1} = x_k + \alpha d$ |
| $\mu \in \mathbb{R}$ | `mu` | Penalty parameter used in the merit function $\phi(x)$ |
| $\phi(x) : \mathcal{X} \to \mathbb{R}$ | `merit_func_nlp` | Merit function object that computes $\phi(x)$ |
| $\phi(x)\|_x \in \mathbb{R}$ | `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 |
| $\mathcal{X}_D \in \mathbb{R}^r$ | `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 |
| $C \equiv \nabla_D c_d(x_k)^T$ $\equiv (A^T)_{(1:r,1:r)}$ $\in \mathcal{C}_d \vert \mathcal{X}_D$ | `C` | Nonsingular Jacobian submatrix (basis) for dependent variables $x_D$ and decomposed constraints $c_d(x)$ at $x_k$ |
| $N \equiv \nabla_I c_d(x_k)^T$ $\equiv (A^T)_{(1:r,r+1:n)}$ $\in \mathcal{C}_d \vert \mathcal{X}_I$ | `N` | Jacobian submatrix for independent variables $x_I$ and decomposed constraints $c_d(x)$ at $x_k$ |
| $E \equiv \nabla_D c_u(x_k)^T$ $\equiv (A^T)_{(r+1:m,1:r)}$ $\in \mathcal{C}_u \vert \mathcal{X}_D$ | `E` | Jacobian submatrix for dependent variables $x_D$ and undecomposed constraints $c_u(x)$ at $x_k$ |
| $F \equiv \nabla_I c_u(x_k)^T$ $\equiv (A^T)_{(r+1:m,r+1:n)}$ $\in \mathcal{C}_u \vert \mathcal{X}_I$ | `F` | Jacobian submatrix for independent variables $x_I$ and undecomposed constraints $c_u(x)$ at $x_k$ |

DISTRIBUTION:

| | | |
|---|---|---|
| 2 | MS 9018 | Central Technical Files, 8944 |
| 2 | MS 0899 | Technical Library, 4536 |