SANDIA REPORT

SAND2014-5333492
Unlimited Release
Printed July 2014

Design Optimization Toolkit

Users' Manual

Miguel A. Aguiló Valentín

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

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

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

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

SAND2014-5333492
Unlimited Release
Printed July 2014

Design Optimization Toolkit

Users' Manual

Miguel A. Aguiló Valentín
Computational Solid Mechanics & Structural Dynamics
Albuquerque, New Mexico 87185-0845

Abstract

The Design Optimization Toolkit (DOTk) is a stand-alone C++ software package intended to solve complex design optimization problems. DOTk software package provides a range of solution methods that are suited for gradient/nongradient-based optimization, large scale constrained optimization, and topology optimization. DOTk was design to have a flexible user interface to allow easy access to DOTk solution methods from external engineering software packages. This inherent flexibility makes DOTk barely intrusive to other engineering software packages. As part of this inherent flexibility, DOTk software package provides an easy-to-use MATLAB interface that enables users to call DOTk solution methods directly from the MATLAB command window.

Acknowledgment

Miguel A. Aguiló Valentín thanks the DOE NNSA Advanced Simulation & Computing (ASC) program for their support.

Contents

List of Figures

List of Tables

Nomenclature

$\mathbf{x}$ Primal variable.

λ Dual variable.

$\mathbf{s}$ Trial step.

$\mathbf{u}$ State variable.

$\mathbf{z}$ Control variable.

∇ Differential operator.

$f(\mathbf{x})$ Twice continuously differentiable function.

$\nabla f(\mathbf{x})$ First derivative of $f(\mathbf{x})$, i.e. gradient.

$\nabla^2 f(\mathbf{x})$ Second derivative of $f(\mathbf{x})$, i.e. Hessian.

$g(\mathbf{x})$ General equaltiy constraint.

$h(\mathbf{x})$ General inequality constraint.

$\mathbb{R}^n$ Set of n-dimensional real vectors.

$\|\cdot\|$ Euclidean norm. For $\mathbf{x} \in \mathbb{R}^n$, $\|\mathbf{x}\| = (\sum_{i=1}^{n} \mathbf{x}_i^2)^{1/2}$.

Δ_k Trust-region at k-th iteration.

$f_{\mathbf{u}}$ Derivative of f with respect to state $\mathbf{u}$.

$f_{\mathbf{z}}$ Derivative of f with respect to control $\mathbf{z}$.

$f_{\mathbf{uu}}$ Derivative of $f_{\mathbf{u}}$ with respect to state $\mathbf{u}$.

$f_{\mathbf{uz}}$ Derivative of $f_{\mathbf{u}}$ with respect to control $\mathbf{z}$.

$f_{\mathbf{zu}}$ Derivative of $f_{\mathbf{z}}$ with respect to state $\mathbf{u}$.

$f_{\mathbf{zz}}$ Derivative of $f_{\mathbf{z}}$ with respect to control $\mathbf{z}$.

$g_{\mathbf{u}}$ Derivative of g with respect to state $\mathbf{u}$.

$g_{\mathbf{z}}$ Derivative of g with respect to control $\mathbf{z}$.

$g_{\mathbf{u}}^T$ Transpose of derivative of g with respect to state $\mathbf{u}$.

$g_{\mathbf{z}}^T$ Transpose of derivative of g with respect to control $\mathbf{z}$.

$g_{\mathbf{uu}}^T$ Derivative of $g_{\mathbf{u}}^T$ with respect to state $\mathbf{u}$.

$g_{\mathbf{uz}}^T$ Derivative of $g_{\mathbf{u}}^T$ with respect to control $\mathbf{z}$.

$g_{\mathbf{zu}}^T$ Derivative of $g_{\mathbf{z}}^T$ with respect to state $\mathbf{u}$.

$g_{\mathbf{zz}}^T$ Derivative of $g_{\mathbf{z}}^T$ with respect to control $\mathbf{z}$.

$\delta\mathbf{u}$ Search direction along the state space.

$\delta\mathbf{z}$ Search direction along the control space.

$\phi(\mathbf{z})$ Solution operator, $\phi\colon \mathbb{R}^{n_z} \to \mathbb{R}^{n_u}$.

Chapter 1

Quickstart

The Design Optimization Toolkit (DOTk) is a stand-alone C++ software package intended to solve complex design optimization problems. DOTk software package provides a range of solution methods that are suited for gradient/nongradient-based optimization, large scale constrained optimization, and topology/shape optimization. DOTk was design to have a flexible user interface to allow easy access to DOTk solution methods from external engineering software packages. This inherent flexibility makes DOTk barely intrusive to other engineering software packages. As part of this inherent flexibility, DOTk software package provides an easy-to-use MATLAB [7] interface that enables users to call DOTk solution methods directly from MATLAB's command window.

Install

DOTk operates on most system running Linux operating systems. *DOTk has only been developed and heavily tested on Redhat Enterprise Linux operating systems with GNU compilers.* Additional operating systems and/or compiler combinations have not yet been tested.

Users are expected to complete a series of installation steps before enjoying the spectrum of solution methods available in DOTk software package. These steps are described as follows:

1. Download DOTk.
   Users can download binary executables for your Linux operating system. Advanced users can download the source code and customize it to accomodate user specific capabilities. Contact Miguel A. Aguiló Valentín (`maguilo@sandia.gov`).

2. Install DOTk.[1]
   Edit the following Makefile options:

   2.1 C++ Installation

   2.1.1 Open the Makefile located inside DOTk's install directory

[1] Users are expected to install DOTk software package only if access to the source code is available.

2.1.3 Set DOTk install directory,
e.g. `DOTk_INSTALL_DIR = PATH_TO_DOTk_INSTALL_DIRECTORY`

2.1.3 Set GNU compiler,
e.g. `CXX = g++`

2.1.4 Set GNU compiler flags,
e.g. `CXXFLAGS = -O3`

2.1.5 Open a terminal window and run `make dotk` inside DOTk install directory

2.2 Matlab Installation

2.1.1 Open the Makefile located inside DOTk's matlab directory,
e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab`

2.1.2 Set DOTk install directory,
e.g. `DOTk_INSTALL_DIR = PATH_TO_DOTk_INSTALL_DIRECTORY`

2.1.3 Set path to MATLAB-Executable (MEX) compiler,
e.g. `MEX = /usr/local/matlab/8.1/bin/mex`

2.1.4 Set MEX compiler flag,
e.g. `MEX_FLAG = -cxx`

2.1.5 Set GNU compiler,
e.g. `CXX = g++`

2.1.6 Set GNU compiler flags,
e.g. `CXXFLAGS = -O3`

2.1.7 Open a terminal window and run `make all` inside DOTk matlab directory,
e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab`

3. Enjoy DOTk software package.

Chapter 2

Application Programming Interface

DOTk software package was intentionally designed to provide a flexible application programming interface (API) that enables easy access to the library's solution methods from external software packages. This chapter describes the required APIs to properly communicate with DOTk software package. Examples are provided to minimize any potential inconvenience that may arise during the installation process. *Current developement exist to implement a Phyton [13] API.*

Gradient-Based Solution Methods

A general constrianed optimization problem is formulated as follows:

$$\begin{aligned} &\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}) \\ &\text{s.t.} \\ &\quad g(\mathbf{x}) = 0, \\ &\quad h(\mathbf{x}) > 0. \end{aligned} \tag{2.1}$$

DOTk offers a gamut of gradient-based solution methods to solve the optimization problem in Equation 2.1. DOTk users should select the solution method based on the optimization problem type.

The optimization problem can be characterized by the type of constraint and by the linearity or nonlinearity of the objective and constraint functions. An **unconstrained optimization problem** is one that has no constraints. An **equality constrained optimization problem** is one that is required to satisfy all of its constraints exactly, i.e. hard constraints. An **inequality constrained optimization problem** is one that is not required to satisfy all of its constraints exactly, i.e. soft constraints. A **constrained optimization problem** is one that has both equality and inequality constraints. A **bound constrained optimization problem** is one that only has upper and lower bound constraints on the optimization/design variables. The optimization problem can be further characterized as a **linear programming problem** (LP) if the objective and equality constraint functions are linear or as a **nonlinear programming problem** (NLP) if the objective and constraint functions are nonlinear. An optimization problem where the objective function is quadratic

and all of the equality constraints are linear is known as a **quadratic programming problem** (QP). The solution methods available in DOTk are designed to solve these types of optimization problems.

Linear Algebra API

Users are expected to implement four basic algebraic operations to utilize DOTk solution methods. The four basic algebraic operations are the following:

$$\begin{aligned} \texttt{scal(alpha,x)}&: \alpha x \rightarrow x \\ \texttt{axpy(alpha,x,y)}&: \alpha x + y \rightarrow y \\ \texttt{innr(x,y)}&: x^T y \rightarrow \textit{output} \\ \texttt{normF(A)}&: \sqrt{\text{trace}(A^T A)} \rightarrow \textit{output} \end{aligned} \tag{2.2}$$

All algebraic operations in DOTk are done with these four linear algebra subroutines. DOTk users are expected to implement these operations through the linear algebra API. This design gives users the flexibility to use the default linear algebra API or a custom linear algebra API, e.g. BLAS [1]. For instance, users can parallelize all inner product operations in DOTk, which also parallelize all vector norm operations, by just implementing a parallel inner product subroutine through the linear algebra API.

C++ linear algebra API. The default C++ linear algebra API is shown in Figures 2.1 and 2.2. Users are not expected to implement their own linear algebra API to use DOTk solution methods. A defualt linear algebra API is already provided to DOTk users. *However, users should be aware that the default linear algebra API implementation is serial.* Advanced users can implement their own C++ linear algebra API as follows:

1. Create a new C++ header file
2. Create a new child class that inherits from parent class `DOTk_LinearAlgebra`
3. Declare and define the four basic algebraic operations in the new child class

Users are given the flexibility to declare and define the algebraic operations subroutines as preferred.

MATLAB linear algebra API. The default MATLAB linear algebra API is shown in Figure 2.3. Users are not expected to implement their own MATLAB linear algebra API. File `getLinearAlgbera.m` already defines the four basic algebraic operations for DOTk users. Advanced users can implement their own MATLAB linear algebra API. However, this requires comprehensive knowledge of MATALB executable (MEX) environment [7]. A MEX-File provides an interface to non-MATLAB code, e.g. C++ or FORTRAN subroutines. Once compiled, the non-MATLAB code can be invoked from any MATLAB code as a native MATLAB function.

```
#ifndef SERIALLINEARALGEBRA_H_
#define SERIALLINEARALGEBRA_H_

#include "DOTk_LinearAlgebra.H"

class SerialLinearAlgebra : public DOTk_LinearAlgebra
{
public:
    SerialLinearAlgebra(){}
    virtual ~SerialLinearAlgebra(){}

    // Scales a vector by a real constant.
    virtual void scal(const Real alpha, Vector & x);
    // Constant times a vector plus a vector.
    virtual void axpy(const Real alpha, const Vector & x, Vector & y);
    // Returns the inner product of two vectors.
    virtual Real innr(const Vector & x, const Vector & y);
    // Returns Frobenius norm of a matrix
    virtual Real normF(const MultiVector & A);

private:
    SerialLinearAlgebra(const SerialLinearAlgebra&); // unimplemented
    SerialLinearAlgebra& operator=(const SerialLinearAlgebra& rhs)
    {
        // check for self-assignment by comparing the address of the
        // implicit object and the parameter
        if (this == &rhs)
        {
            return (*this);
        }
        // return the existing object
        return (*this);
    }
};

#endif /* SERIALLINEARALGEBRA_H_ */
```

Figure 2.1. C++ linear algebra API child class definition.

```cpp
#include "SerialLinearAlgebra.H"

void SerialLinearAlgebra::scal(const Real alpha, Vector& x)
{
    /* Implement scal here. */
}
void SerialLinearAlgebra::axpy(const Real alpha, const Vector& x, Vector& y)
{
    /* Implement axpy here. */
}
Real SerialLinearAlgebra::innr(const Vector& x, const Vector& y)
{
    /* Implement innr here. */
}
Real SerialLinearAlgebra::normF(const MultiVector& A)
{
    /* Implement normF here. */
}
```

Figure 2.2. C++ linear algebra API child class declaration.

A custom MATLAB linear algebra API can be implemented as follows:

1. Build MEX interface to non-MATLAB code
2. Compile MEX-file
3. Define linear algebra function in file `getLinearAlgbera.m`

```
function [LinearAlgebra] = getLinearAlgebra()
LinearAlgebra.scal=@(alpha,x) alpha*x;
LinearAlgebra.axpy=@(alpha,x,y) alpha*x+y;
LinearAlgebra.innr=@(x,y) x'*y;
LinearAlgebra.normF=@(x) norm(x,'fro');
end
```

Figure 2.3. MATLAB linear algebra API.

Operators API

Althought the gradient-based solution methods available in DOTk can have multiple algorithmic differences, these solution methods can also share many common components. For instance, first-order and second-order derivative information is required to efficiently solve Equation 2.1. This information is used to compute the trial step that enables the update of the new set of optimization/design variables at each iteration. Depending on the function and derivative information available to users as well as the class of optimization problem, users are expected to provide the available function and derivative information through the operators API. The operators API allows DOTk solution methods to access the function and derivative information that is required to update the set of optimization/design variables at each iteration.

Unconstrained optimization problems. The basic function and derivative information required to efficiently solve unconstrained optimization problems are `Fval(z)` : $f(\mathbf{z}) \rightarrow$ `output`, `F`$_z$`(z)` : `F`$_z(\mathbf{z}) \rightarrow$`output`, `F`$_{zz}$`(z)` : `F`$_{zz}(\mathbf{z}) \rightarrow$`output`. The `z` keyword specifies the control variables. The `Fval(z)` keyword specifies the evaluation of the objective function. The `F`$_z$`(z)` keyword specifies the first derivative of the objective function with respect to the control variables. The `F`$_{zz}$`(z)` keyword specifies the derivative of `F`$_z$ with respect to the control variables, i.e. second-order derivative.

Table 2.1. Required operators for unconstrained optimization problems

Information	Operators
None	$f(\mathbf{z})$
1^{st} order	$f(\mathbf{z})$, $f_{\mathbf{z}}(\mathbf{z})$
1^{st} and 2^{nd} order	$f(\mathbf{z})$, $f_{\mathbf{z}}(\mathbf{z})$, $f_{\mathbf{zz}}(\mathbf{z})$

The operators that users are expected to implement to solve unconstrained optimization problems are outline in Table 2.1. Notice that the requirements change depending on the derivative information available to the user. If no derivative information is available to the user, the user is just required to implement the evaluation of the objective function in the operators API. The finite differencing and quasi-Newton approximation techniques available in DOTk can be used to respectively approximate the first-order and second-order derivative information. If first-order derivative information is available to the users, the user is expected to implement both the objective function evaluation and the first-order derivative information through the operators API. The quasi-Newton approximation techniques available in DOTk can be used to approximate the second-order derivative information. Finally, if both first-order and second-order derivative information is available to the user, the user is expected to implement the objective function evaluation and both first-order and second-order derivative information through the operators API.

Constrained optimization problems. The basic functions and derivative information

required to efficiently solve LP or NLP constrained optimization problems are:

$$
\begin{aligned}
\texttt{Fval(u,z)} &: \texttt{F}(\mathbf{u},\mathbf{z}) \rightarrow \texttt{output} \\
\texttt{F\_u(u,z,output)} &: \texttt{F}_{\mathbf{u}}(\mathbf{u},\mathbf{z}) \rightarrow \texttt{output} \\
\texttt{F\_z(u,z,output)} &: \texttt{F}_{\mathbf{z}}(\mathbf{u},\mathbf{z}) \rightarrow \texttt{output} \\
\texttt{F\_uu(u,z,s,output)} &: \texttt{F}_{\mathbf{uu}}(\mathbf{u},\mathbf{z})\mathbf{s} \rightarrow \texttt{output} \\
\texttt{F\_uz(u,z,s,output)} &: \texttt{F}_{\mathbf{uz}}(\mathbf{u},\mathbf{z})\mathbf{s} \rightarrow \texttt{output} \\
\texttt{F\_zz(u,z,s,output)} &: \texttt{F}_{\mathbf{zz}}(\mathbf{u},\mathbf{z})\mathbf{s} \rightarrow \texttt{output} \\
\texttt{F\_zu(u,z,s,output)} &: \texttt{F}_{\mathbf{zu}}(\mathbf{u},\mathbf{z})\mathbf{s} \rightarrow \texttt{output} \\
\texttt{Gval(u,z,output)} &: \texttt{G}(\mathbf{u},\mathbf{z}) \rightarrow \texttt{output} \\
\texttt{G\_u(u,z,du,output)} &: \texttt{G}_{\mathbf{u}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{u} \rightarrow \texttt{output} \\
\texttt{G\_z(u,z,dz,output)} &: \texttt{G}_{\mathbf{z}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{z} \rightarrow \texttt{output} \\
\texttt{adjG\_u(u,z,}\lambda\texttt{,output)} &: (\texttt{adjG}_{\mathbf{u}}(\mathbf{u},\mathbf{z}))^T\lambda \rightarrow \texttt{output} \\
\texttt{adjG\_z(u,z,}\lambda\texttt{,output)} &: (\texttt{adjG}_{\mathbf{z}}(\mathbf{u},\mathbf{z}))^T\lambda \rightarrow \texttt{output} \\
\texttt{adjG\_uu(u,z,}\lambda\texttt{,du,output)} &: (\texttt{adjG}_{\mathbf{uu}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{u})^T\lambda \rightarrow \texttt{output} \\
\texttt{adjG\_uz(u,z,}\lambda\texttt{,dz,output)} &: (\texttt{adjG}_{\mathbf{uz}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{z})^T\lambda \rightarrow \texttt{output} \\
\texttt{adjG\_zz(u,z,}\lambda\texttt{,dz,output)} &: (\texttt{adjG}_{\mathbf{zz}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{z})^T\lambda \rightarrow \texttt{output} \\
\texttt{adjG\_zu(u,z,}\lambda\texttt{,du,output)} &: (\texttt{adjG}_{\mathbf{zu}}(\mathbf{u},\mathbf{z})\mathrm{d}\mathbf{u})^T\lambda \rightarrow \texttt{output}
\end{aligned}
$$

The **u** keyword specifies the state variables. The $\texttt{F}_u$ keyword specifies the first-order derivative of the objective function with respect to the state variables. The $\texttt{F}_{uu}$ keyword specifies the derivative of $\texttt{F}_u$ with respect to the state variables, i.e. second-order derivative. The $\texttt{F}_{uz}$ keyword specifies the derivative of $\texttt{F}_u$ with respect to the control variables. The $\texttt{F}_{zu}$ keyword specifies the derivative of $\texttt{F}_z$ with respect to the state variables. The `Gval` keyword specifies the evalaution of the equality constraint. The $\texttt{G}_u$ keyword specifies the first-order derivative of the equality constraint with respect to the state variables. The The $\texttt{G}_z$ keyword specifies the first-order derivative of the equality constraint with respect to the control variables. The $\texttt{adjG}_u$ keyword specifies the first-order derivative of the adjoint of the equality constraint with respect to the state variables. The $\texttt{adjG}_z$ keyword specifies the first-order derivative of the adjoint of the equality constraint with respect to the control variables. The $\texttt{adjG}_{uu}$ keyword specifies the derivative of $\texttt{adjG}_u$ with respect to the state variables, i.e the second-order derivative of the adjoint of the equality constraint. The $\texttt{adjG}_{uz}$ keyword specifies the derivative of $\texttt{adjG}_u$ with respect to the control variables. The $\texttt{adjG}_{zz}$ keyword specifies the derivative of $\texttt{adjG}_z$ with respect to the control variables. The $\texttt{adjG}_{zu}$ keyword specifies the derivative of $\texttt{adjG}_z$ with respect to the state variables.

The operators that users are expected to implement to solve LP and NLP constrained optimization problems are outline in Table 2.2. Analogous to unconstrained optimization problems, the requirements may change depending on the derivative information avialable to the user. Users can make use of the finite differencing and quasi-Newton techniques to respectively approximate first-order second-order derivative information if necessary. By giving DOTk users the flexibility to define the required function and derivative information

Table 2.2. Required operators to solve LP and NLP constrained optimization problems.

Information	Operators
None	$f(\mathbf{u},\mathbf{z})$, $g(\mathbf{u},\mathbf{z})$
1^{st} order	$f(\mathbf{u},\mathbf{z})$, $f_{\mathbf{u}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{z}}(\mathbf{u},\mathbf{z})$, $g(\mathbf{u},\mathbf{z})$, $g_{\mathbf{u}}(\mathbf{u},\mathbf{z})$, $g_{\mathbf{z}}(\mathbf{u},\mathbf{z})$, $(g_{\mathbf{u}}(\mathbf{u},\mathbf{z}))^T$, $(g_{\mathbf{z}}(\mathbf{u},\mathbf{z}))^T$
1^{st} and 2^{nd} order	$f(\mathbf{u},\mathbf{z})$, $f_{\mathbf{u}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{z}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{uu}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{uz}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{zz}}(\mathbf{u},\mathbf{z})$, $f_{\mathbf{zu}}(\mathbf{u},\mathbf{z})$ $g(\mathbf{u},\mathbf{z})$, $g_{\mathbf{u}}(\mathbf{u},\mathbf{z})$, $g_{\mathbf{z}}(\mathbf{u},\mathbf{z})$, $(g_{\mathbf{u}}(\mathbf{u},\mathbf{z}))^T$, $(g_{\mathbf{z}}(\mathbf{u},\mathbf{z}))^T$ $(g_{\mathbf{uu}}(\mathbf{u},\mathbf{z}))^T$, $(g_{\mathbf{uz}}(\mathbf{u},\mathbf{z}))^T$, $(g_{\mathbf{zz}}(\mathbf{u},\mathbf{z}))^T$, $(g_{\mathbf{zu}}(\mathbf{u},\mathbf{z}))^T$,

through one common API, any gradient-based solution method available in DOTk can be easily used to solve the optimization problem. This added flexibility is one of the unique feature of DOTk software package.

C++ Operators API. Users are expected to implement the operators API to use DOTk solution methods. This implementation strictly depends on the function and derivative information available to the user. *The user is not required to implement derivative information that is not available.*[1] The C++ operators API can be implemented as follows:

1. Create a new C++ header file
2. Create a new child class that inherits from parent class `DOTk_Operators`
3. Declare and define appropriate operator in new child class

A C++ operators API example for both unconstrained and LP/NLP constrained optimization problems are respectively shown in Figures 2.4-2.5 and Figures 2.6-2.7.

MATLAB Operators API. Users are expected to implement the MATLAB operators API to invoke DOTk solution methods from any MATLAB code. The objective function operators must be implemented in File `getObjectiveFunctionOperators.m` and the equality constraint operators must be implemented in File `getEqualityConstraintOperators.m`. The MATLAB operators API for unconstrained and LP/NLP constrained optimization problems are respectively shown in Figures 2.8 and 2.10.

Advanced users can implement MEX-files to interface non-MATLAB code with MATLAB code. The basic steps are outline as follows:

1. Build MEX interface to non-MATLAB code that evaluates operators
2. Compile user-defined MEX-file

[1] The C++ subroutines corresponding to the unavailable derivative informations do not need to be declared and defined in the operators API child class.

```cpp
#ifndef ROSENBROCKOPERATORS_H_
#define ROSENBROCKOPERATORS_H_

#include "DOTk_Operators.H"

class RosenbrockOperators : public DOTk_Operators
{
public:
    RosenbrockOperators();
    virtual ~RosenbrockOperators(){}

    /* ================================ Main Routines ================================= */

    virtual Real Fval(const Vector &z);
    virtual void Fval(const MultiVector &z, Vector &fval);
    virtual void F_z(const Vector &z, Vector &g);
    virtual void F_zz(const Vector& z, const Vector& dz, Vector& H_dz);
private:
    // unimplemented
    RosenbrockOperators(const RosenbrockOperators&);
    RosenbrockOperators operator=(const RosenbrockOperators&);

};

#endif /* ROSENBROCKOPERATORS_H_ */
```

Figure 2.4. C++ operators API class definition example: Unconstrained problems.

3. Define operators function in Files `getObjectiveFunctionOperators.m` and `getEqualityConstraintOperators.m`

Solution Operators API

Partial differential equations (PDE) constrained optimization problems of the form:

$$\begin{aligned} &\underset{(\mathbf{u},\mathbf{z})\in\mathbb{R}^{n_u}\times\mathbb{R}^{n_z}}{\text{minimize}} && f(\mathbf{u},\mathbf{z}) \\ &\text{s.t.} && \\ &&& g(\mathbf{u},\mathbf{z}) = 0. \end{aligned} \tag{2.3}$$

predominate in many engineering applications and are a primary focus of DOTk software package. The formulation in Equation 2.3 is commonly known as the full-space formulation. Now, assuming that a twice differentiable function $\phi\colon \mathbb{R}^{n_z} \to \mathbb{R}^{n_u}$ such that $g(\phi(\mathbf{z}),\mathbf{z}) = 0\ \forall\ \mathbf{z}\in\mathbb{R}^{n_z}$. If $g_{\mathbf{u}}(\mathbf{u},\mathbf{z})$ is invertible, the implicit function theorem guarantees the existence of ϕ [6]. The existance of $\phi(z)$ enables the reformulation of Equation 2.3 as an unconstrained optimization problem of the form:

$$\underset{\mathbf{z}\in\mathbb{R}^{n_z}}{\text{minimize}} \quad f(\phi(\mathbf{z}),\mathbf{z}) \tag{2.4}$$

```cpp
#include "RosenbrockOperators.H"

RosenbrockOperators::RosenbrockOperators() :
        DOTk_Operators::DOTk_Operators()
{
}
Real RosenbrockOperators::Fval(const Vector& z)
{
    // Implement source code here
}
void RosenbrockOperators::F_z(const Vector &z, Vector &g)
{
    // Implement source code here
}
void RosenbrockOperators::F_zz(const Vector& z, const Vector& dz, Vector& H_dz)
{
    // Implement source code here
}
```

Figure 2.5. C++ operators API class declaration example: Unconstrained problems.

This formulation is commonly called the reduce-space formulation.

Users must implement both the linear algebra API and the operators API to properly solve PDE-constrained optimization problems with DOTk. If a reduced-space formulation is used, users are also expected to implement the solution operators API. However, the amount of work depends on the derivative information available to the user. The three possibilities are outline as follows:

1. *No derivative information is available:* User must not implement the solution operators API.

2. *First-order derivative information is available:* Forward and adjoint solves are essential to assemble the true gradient operator. Therefore, the user must implement two solution operators, the forward and adjoint solves required to compute the adjoint field.

3. *Both first-order and second-order derivative information is available:* Two forward and adjoint solves are essential to compute the true gradient operator and the application of the trial step to the true Hessian operator.

A theoretical framework that outlines how these solution operators are derived and used to solve Equation 2.4 is presented in [2].

C++ Solution Operators API. The C++ solution operators API can be implemented as follows:

```cpp
#ifndef FULLSPACETESTOPERATORS_H_
#define FULLSPACETESTOPERATORS_H_

#include "DOTk_Operators.H"

class FullSpaceTestOperators: public DOTk_Operators
{
public:
    FullSpaceTestOperators();
    virtual ~FullSpaceTestOperators(){}

    /* ========================================== Main Interface ========================================== */

    virtual Real Fval(const Vector& u, const Vector& z);
    virtual void Cval(const Vector& u, const Vector& z, Vector& cval);

    virtual void F_u(const Vector& u, const Vector& z, Vector &g);
    virtual void C_u(const Vector& u, const Vector& z, const Vector& dx, Vector& J_dx);
    virtual void adjC_u(const Vector& u, const Vector& z, const Vector& dy, Vector& J_dy);

    virtual void F_uu(const Vector& u, const Vector& z, const Vector& dx, Vector& H_dx);
    virtual void adjC_uu(const Vector& u, const Vector& z, const Vector& lambda, const Vector& du, Vector& adjC_du);

private:
    // unimplemented
    FullSpaceTestOperators(const FullSpaceTestOperators&);
    FullSpaceTestOperators operator=(const FullSpaceTestOperators&);
};

#endif /* FULLSPACETESTOPERATORS_H_ */
```

Figure 2.6. C++ operators API class definition example: LP and NLP problems.

1. Create a new C++ header file
2. Create a new child class that inherits from parent class `DOTk_SolutionOperators`
3. Declare and define appropriate solution operators in new child class

A C++ solution operators API example is shown in Figures 2.11 and 2.12.

MATLAB Solution Operators API. If a reduced-space formulation is used and derivative information is available, users must implement the MATLAB solution operators API to invoke DOTk solution methods from any MATLAB code. The solution operators must be implemented in File `getSolutionOperators.m`. The MATLAB solution operators API is shown in Figure 2.13.

Advanced users can implement MEX-files to interface non-MATLAB code with MATLAB code. The basic steps are outline as follows:

1. Build MEX interface to non-MATLAB code that evaluates solution operators
2. Compile user-defined MEX-file
3. Define solution operators function in Files `getSolutionOperators.m`

```cpp
#include "FullSpaceTestOperators.H"

FullSpaceTestOperators::FullSpaceTestOperators() :
        DOTk_Operators::DOTk_Operators()
{
}
Real FullSpaceTestOperators::Fval(const Vector& u, const Vector& z)
{
    // Implement source code here
}
void FullSpaceTestOperators::Cval(const Vector& u, const Vector& z, Vector& cval)
{
    // Implement source code here
}
void FullSpaceTestOperators::adjC_u(const Vector& u, const Vector& z, const Vector& lambda, Vector& J_dy)
{
    // Implement source code here
}
void FullSpaceTestOperators::C_u(const Vector& u, const Vector& z, const Vector& du, Vector& J_dx)
{
    // Implement source code here
}
void FullSpaceTestOperators::F_u(const Vector& u, const Vector& z, Vector &g)
{
    // Implement source code here
}
void FullSpaceTestOperators::F_uu(const Vector& u, const Vector& z, const Vector& du, Vector& H_du)
{
    // Implement source code here
}
void FullSpaceTestOperators::adjC_uu(const Vector& u, const Vector& z, const Vector& lambda, const Vector& du, Vector& adjC_du)
{
    // Implement source code here
}
```

Figure 2.7. C++ operators API class declaration example: LP and NLP problems.

Preconditioner API

The iterative essence of gradient-based solutions methods can encounter multiple numerical challenges in practical settings. Preconditioners are used to transform challenging optimization problems into problems that are suitable for numerical solution. DOTk provides users with a preconditioner API that allows them to use a preconditioner of their choice or implement the essential operators required to enable a DOTk preconditioner. This is one additional unique feature of DOTk software package.

C++ Preconditioner API. The C++ preconditioner API can be implemented as follows:

1. Create a new C++ header file
2. Create a new child class that inherits from parent class `DOTk_PreconditionerOperators`
3. Declare and define appropriate solution operators in new child class

A C++ preconditioner operators API example for reduced-space and full-sapce PDE-constrained optimization is respectively shown in Figures 2.14-2.15 and Figures 2.16-2.17. *If the user is interested in using a preconditioner of their choice, the user is not expected to implement the*

```
function [ObjectiveFunctionOperators] = getObjectiveFunctionOperators()
ObjectiveFunctionOperators.Fval=@(z) Fval(z);
ObjectiveFunctionOperators.F_u=@(z) F_u(z);
ObjectiveFunctionOperators.F_z=@(z) F_z(z);
ObjectiveFunctionOperators.F_uu=@(z,du) F_uu(z,du);
ObjectiveFunctionOperators.F_uz=@(z,dz) F_uz(z,dz);
ObjectiveFunctionOperators.F_zz=@(z,dz) F_zz(z,dz);
ObjectiveFunctionOperators.F_zu=@(z,du) F_zu(z,du);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [fval] = Fval(z)
% Implement Fval here.
end
function [f_z] = F_z(z)
% Implement F_z here.
end
function  [fzz_x_dz] = F_zz(z,dz)
% Implement F_zz here.
end
```

Figure 2.8. MATLAB objective function operators API for unconstrained problems

preconditioner operators needed to enable DOTk preconditioning strategies. The user is just expected to implement the `userDefinedLeftPrec` or `userDefinedRightPrec` subroutines.

MATLAB Preconditioner API. If users wish to use a DOTk-based or user-defined preconditioner, they are expected to implement the essential operators in the MATLAB preconditioner API. The preconditioner API allows running DOTk solution method to pass a vector into a MATLAB function, apply the vector to the preconditioner operator, and return the application of the vector to the preconditioner operator to the DOTk solution method through the preconditioner API. For reduced-space PDE-constrained problems, the preconditioner operators must be implemented in File `getReducedSpacePrecOperators.m`. For full-space PDE-constrained problems, the precondtioner operators must be implemented in File `getFullSpacePrecOperators.m`. The MATLAB preconditioner API is shown in Figures 2.18 and 2.19.

Advanced users can implement MEX-files to interface non-MATLAB code with MATLAB code. The basic steps are outline as follows:

1. Build MEX interface to non-MATLAB code that evaluates the application of a vector to the preconditioner operator

2. Compile user-defined MEX-file

3. Define appropriate MATLAB functions in File `getReducedSpacePrecOperators.m` or `getFullSpacePrecOperators.m`

If the preconditioner option is USER_DEFINED, users are not expected to implement the essential operators needed to enable DOTk preconditioner strategies.

```
function [ObjectiveFunctionOperators] = getObjectiveFunctionOperators()
ObjectiveFunctionOperators.Fval=@(u,z) Fval(u,z);
ObjectiveFunctionOperators.F_u=@(u,z) F_u(u,z);
ObjectiveFunctionOperators.F_z=@(u,z) F_z(u,z);
ObjectiveFunctionOperators.F_uu=@(u,z,du) F_uu(u,z,du);
ObjectiveFunctionOperators.F_uz=@(u,z,dz) F_uz(u,z,dz);
ObjectiveFunctionOperators.F_zz=@(u,z,dz) F_zz(u,z,dz);
ObjectiveFunctionOperators.F_zu=@(u,z,du) F_zu(u,z,du);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [fval] = Fval(u,z)
% Implement Fval here.
end
function [f_u] = F_u(u,z)
% Implement F_u here.
end
function [f_z] = F_z(u,z)
% Implement F_z here.
end
function  [fuu_x_du] = F_uu(u,z,du)
% Implement F_uu here.
end
function  [fuz_x_dz] = F_uz(u,z,dz)
% Implement F_uz here.
end
function  [fzz_x_dz] = F_zz(u,z,dz)
% Implement F_zz here.
end
function  [fzu_x_du] = F_zu(u,z,du)
% Implement F_zu here.
end
```

Figure 2.9. MATLAB objective function operators API: LP and NLP problems.

```
function [EqualityConstraintOperators] = getEqualityConstraintOperators()
EqualityConstraintOperators.Cval=@(u,z) Cval(u,z);
EqualityConstraintOperators.C_u=@(u,z,du) C_u(u,z,du);
EqualityConstraintOperators.C_z=@(u,z,dz) C_z(u,z,dz);
EqualityConstraintOperators.adjC_u=@(u,z,lambda) adjC_u(u,z,lambda);
EqualityConstraintOperators.adjC_z=@(u,z,lambda) adjC_z(u,z,lambda);
EqualityConstraintOperators.adjC_uu=@(u,z,lambda,du) adjC_uu(u,z,lambda,du)
EqualityConstraintOperators.adjC_uz=@(u,z,lambda,dz) adjC_uz(u,z,lambda,dz)
EqualityConstraintOperators.adjC_zz=@(u,z,lambda,dz) adjC_zz(u,z,lambda,dz)
EqualityConstraintOperators.adjC_zu=@(u,z,lambda,du) adjC_zu(u,z,lambda,du)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [cval] = Cval(u,z) %#ok<*INUSD,*STOUT>
% Implement Cval here.
end
function [Cu] = C_u(u,z,du)
% Implement C_u here.
end
function [Cz] = C_z(u,z,dz)
% Implement C_z here.
end
function [adjCu] = adjC_u(u,z,lambda)
% Implement adjC_u here.
end
function [adjCz] = adjC_z(u,z,lambda)
% Implement adjC_z here.
end
function [adjCuu_x_du] = adjC_uu(u,z,lambda,du)
% Implement adjC_uu here.
end
function [adjCuz_x_dz] = adjC_uz(u,z,lambda,dz)
% Implement adjC_uz here.
end
function [adjCzz_x_dz] = adjC_zz(u,z,lambda,dz)
% Implement adjC_zz here.
end
function [adjCzu_x_du] = adjC_zu(u,z,lambda,du)
% Implement adjC_zu here.
end
```

Figure 2.10. MATLAB equality constraint operators API: LP and NLP problems.

```cpp
#ifndef REDUCEDSPACEFORMULATION_H_
#define REDUCEDSPACEFORMULATION_H_

#include "DOTk_SolutionOperators.H"

class ReducedSpaceFormulation : public DOTk_SolutionOperators
{
public:
    ReducedSpaceFormulation(){};
    virtual ~ReducedSpaceFormulation();

    virtual void solveForwardProblem(const Vector& z, Vector& rhs, Vector& lhs);
    virtual void solveGradAdjointProblem(const Vector& z, Vector& rhs, Vector& lhs);
    virtual void solveHessForwardProblem(const Vector& z, Vector& rhs, Vector& lhs);
    virtual void solveHessAdjointProblem(const Vector& z, Vector& rhs, Vector& lhs);

private:
    ReducedSpaceFormulation(const ReducedSpaceFormulation&); // unimplemented
    ReducedSpaceFormulation& operator=(const ReducedSpaceFormulation& rhs)
    {
        // check for self-assignment by comparing the address of the
        // implicit object and the parameter
        if(this == &rhs)
        {
            return (*this);
        }
        // return the existing object
        return (*this);
    }
};

#endif /* REDUCEDSPACEFORMULATION_H_ */
```

Figure 2.11. C++ solution operators API child class definition.

```cpp
#include "ReducedSpaceFormulation.H"

ReducedSpaceFormulation::ReducedSpaceFormulation() :
   DOTk_SolutionOperators::DOTk_SolutionOperators() {}

void ReducedSpaceFormulation::solveForwardProblem(const Vector& z, Vector& rhs, Vector& lhs)
{
    /* Implement solveForwardProblem here. */
}
void ReducedSpaceFormulation::solveGradAdjointProblem(const Vector& z, Vector& rhs, Vector& lhs)
{
    /* Implement solveGradAdjointProblem here. */
}
void ReducedSpaceFormulation::solveHessForwardProblem(const Vector& z, Vector& rhs, Vector& lhs)
{
    /* Implement solveHessForwardProblem here. */
}
void ReducedSpaceFormulation::solveHessAdjointProblem(const Vector& z, Vector& rhs, Vector& lhs)
{
    /* Implement solveHessAdjointProblem here. */
}
```

Figure 2.12. C++ solution operators API child class declaration.

```
function SolutionOperators = getSolutionOperators()
SolutionOperators.solveForwardProblem=@(z,rhs)solveForwardProblem(z,rhs);
SolutionOperators.solveGradAdjointProblem=...
    @(z,rhs)solveGradAdjointProblem(z,rhs);
SolutionOperators.solveHessForwardProblem=...
    @(z,rhs)solveHessForwardProblem(z,rhs);
SolutionOperators.solveHessAdjointProblem=...
    @(z,rhs)solveHessAdjointProblem(z,rhs);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [lhs] = solveForwardProblem(z,rhs)
% Implement solveGradAdjointProblem here.
end
function [lhs] = solveGradAdjointProblem(z,rhs)
% Implement solveGradAdjointProblem here.
end
function [lhs] = solveHessForwardProblem(z,rhs) %#ok<*STOUT,*INUSD>
% Implement solveHessForwardProblem here.
end
function [lhs] = solveHessAdjointProblem(z,rhs)
% Implement solveHessAdjointProblem here.
end
```

Figure 2.13. MATLAB solution operators API.

```
#ifndef USERDEFINEDPRECOPERATORS_H_
#define USERDEFINEDPRECOPERATORS_H_

#include "DOTk_PreconditionerOperators.H"

class UserDefinedPrecOperators : public DOTk_PreconditionerOperators
{
public:
    UserDefinedPrecOperators();
    virtual ~UserDefinedPrecOperators(){};

    virtual void userDefinedLeftPrec(const Vector & control, const Vector & direction, Vector & output);
    virtual void userDefinedRightPrec(const Vector & control, const Vector & direction, Vector & output);

private:
    // unimplemented functions
    UserDefinedPrecOperators(const UserDefinedPrecOperators&);
    UserDefinedPrecOperators& operator=(const UserDefinedPrecOperators& rhs);
};
```

Figure 2.14. C++ preconditioner API child class definition: Reduced-space PDE-constrained optimization problems

```
#include "UserDefinedPrecOperators.H"

UserDefinedPrecOperators::UserDefinedPrecOperators() :
       DOTk_PreconditionerOperators::DOTk_PreconditionerOperators()
{}

void UserDefinedPrecOperators::userDefinedLeftPrec(const Vector & control, const Vector & direction, Vector & output)
{
    /* Implement userDefinedLeftPrec here. */
}
void UserDefinedPrecOperators::userDefinedRightPrec(const Vector & control, const Vector & direction, Vector & output)
{
    /* Implement userDefinedRightPrec here. */
}
```

Figure 2.15. C++ preconditioner API child class declaration: Reduced-space PDE-constrained optimization problems

```
#ifndef USERDEFINEDPRECOPERATORS_H_
#define USERDEFINEDPRECOPERATORS_H_

#include "DOTk_PreconditionerOperators.H"

class UserDefinedPrecOperators : public DOTk_PreconditionerOperators
{
public:

    UserDefinedPrecOperators();
    virtual ~UserDefinedPrecOperators(){}

    virtual void invJacobian_State(const Vector & state, const Vector & control,
                                   const Vector & direction, Vector & output);
    virtual void adjointInvJacobian_State(const Vector & state, const Vector & control,
                                          const Vector & direction, Vector & output);
    virtual void userDefinedLeftPrec(const Vector & state, const Vector & control,
                                     const Vector & direction, Vector & output);
    virtual void userDefinedRightPrec(const Vector & state, const Vector & control,
                                      const Vector & direction, Vector & output);
    virtual void invJacobian(const Vector & state, const Vector & control,
                             const Vector & direction, Vector & output);
    virtual void adjointInvJacobian(const Vector & state, const Vector & control,
                                    const Vector & direction, Vector & output);
private:
    // unimplemented functions
    UserDefinedPrecOperators(const UserDefinedPrecOperators&);
    UserDefinedPrecOperators& operator=(const UserDefinedPrecOperators& rhs);
};
```

Figure 2.16. C++ preconditioner API child class definition: Full-space PDE-constrained optimization problems

```cpp
#include "UserDefinedPrecOperators.H"

UserDefinedPrecOperators::UserDefinedPrecOperators() :
    DOTk_PreconditionerOperators::DOTk_PreconditionerOperators()
{}

void UserDefinedPrecOperators::invJacobian_State(const Vector & state, const Vector & control,
                                                 const Vector & direction, Vector & output)
{
    /* Implement invJacobian_State here. */
}
void UserDefinedPrecOperators::adjointInvJacobian_State(const Vector & state, const Vector & control,
                                                        const Vector & direction, Vector & output)
{
    /* Implement adjointInvJacobian_State here. */
}
void UserDefinedPrecOperators::userDefinedLeftPrec(const Vector & state, const Vector & control,
                                                   const Vector & direction, Vector & output)
{
    /* Implement userDefinedLeftPrec here. */
}
void UserDefinedPrecOperators::userDefinedRightPrec(const Vector & state, const Vector & control,
                                                    const Vector & direction, Vector & output)
{
    /* Implement userDefinedRightPrec here. */
}
void UserDefinedPrecOperators::invJacobian(const Vector & state, const Vector & control,
                                           const Vector & direction, Vector & output)
{
    /* Implement invJacobian here. */
}
void UserDefinedPrecOperators::adjointInvJacobian(const Vector & state, const Vector & control,
                                                  const Vector & direction, Vector & output)
{
    /* Implement adjointInvJacobian here. */
}

};
```

Figure 2.17. C++ preconditioner API child class declaration: Full-space PDE-constrained optimization problems

```
function [ReducedSpacePrecOperators] = getReducedSpacePrecOperators()
ReducedSpacePrecOperators.userDefinedLeftPrec = ...
    @(control,direction) userDefinedLeftPrec(control,direction);
ReducedSpacePrecOperators.userDefinedRightPrec = ...
    @(control,direction) userDefinedRightPrec(control,direction);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Matrix_times_direction] = ...
    userDefinedLeftPrec(state,control,direction) %#ok<*STOUT,*INUSD>
% Implement userDefinedLeftPrec here.
end
function [Matrix_times_direction] = ...
    userDefinedRightPrec(state,control,direction)
% Implement userDefinedRightPrec here.
end
```

Figure 2.18. MATLAB preconditioner API: Reduced-space PDE-constrained optimization problems.

```
function [FullSpacePrecOperators] = getFullSpacePrecOperators()
FullSpacePrecOperators.userDefinedLeftPrec = ...
    @(state,control,direction) ...
    userDefinedLeftPrec(state,control,direction);
FullSpacePrecOperators.userDefinedRightPrec = ...
    @(state,control,direction) ...
    userDefinedRightPrec(state,control,direction);
FullSpacePrecOperators.invJacobian = ...
    @(state,control,direction) ...
    invJacobian(state,control,direction);
FullSpacePrecOperators.adjointInvJacobian = ...
    @(state,control,direction) ...
    adjointInvJacobian(state,control,direction);
FullSpacePrecOperators.invJacobian_State = ...
    @(state,control,direction) ...
    invJacobian_State(state,control,direction);
FullSpacePrecOperators.adjointInvJacobian_State = ...
    @(state,control,direction) ...
    adjointInvJacobian_State(state,control,direction);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Implementation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Matrix_times_direction] = ...
    userDefinedLeftPrec(state,control,direction) %#ok<*STOUT,*INUSD>
% Implement userDefinedLeftPrec here.
end
function [Matrix_times_direction] = ...
    userDefinedRightPrec(state,control,direction)
% Implement userDefinedRightPrec here.
end
function [Matrix_times_direction] = ...
    invJacobian(state,control,direction)
% Implement invJacobian here.
end
function [Matrix_times_direction] = ...
    adjointInvJacobian(state,control,direction)
% Implement adjointInvJacobian here.
end
function [Matrix_times_direction] = ...
    invJacobian_State(state,control,direction)
% Implement invJacobian_State here.
end
function [Matrix_times_direction] = ...
    adjointInvJacobian_State(state,control,direction)
% Implement adjointInvJacobian_State here.
end
```

Figure 2.19. MATLAB preconditioner API: Full-space PDE-constrained optimization problems.

Chapter 3

Gradient-Based Optimization

Gradient-based solution methods are highly efficient methods that are well suited for problems where the objective function is smooth and derivative information can be provided. The disadvantage of these solution methods is that only local solutions can be guaranteed. However, to promote the convergence from a remote starting point, DOTk offers several line search and trust-region methods. The combination of the gradient-based solution methods and globalization methods available in DOTk results in a vast suit of tools for the solution of multimodal optimization problems. This chapter describes the parameters that are used to properly set the solution and globalization methods available in DOTk.

General Use Parameters

C++ API

General use set functions. The general-use set functions are shown in Table 3.1. The **sol::params::space(·)**[1] data structure stores the optimization problem dimensions, e.g. `sol ::params::space struct(dim(z), dim(u), dim(g), dim(h))`, where `z` specifies the control variables, `u` specifies the state variables, `g` specifies the vector of equality constraints, `h` specifies the vector of inequality constraints, and `dim(·)` specifies the argument dimensions. The **setMaxNumOptimizationItr** function sets the maximum nuber of optimization iterations allow. The **setObjectiveFuncTol** function sets the stopping criterion due to the objective function. The **setGradientTol** function sets the stopping criterion due to the norm of the gradient. The **setTrialStepTol** function sets the stopping criterion due to the norm of the trial step.

[1]This data structure is only compatible for full-space formulation strategies and hence full-space optimization algorithms. A new interface for a full-space formulation strategy is currently under development and the **sol::params::space(·)** data structure will be deprecated. DOTk users will be informed as soon as this change is made effective

Table 3.1. Set functions for general use parameters.

Function	Argument	Default
sol::params::space	*Integer*	dim(z)=dim(u)=dim(g)=dim(h)=0
setMaxNumOptimizationItr	*Real*	5000
setObjectiveFuncTol	*Real*	1e-12
setGradientTol	*Real*	1e-8
setTrialStepTol	*Real*	1e-12

MATLAB API

General use parameters. The general-use parameters are shown in Table 3.2. The **NumControls** keyword specifies the number of control variables, i.e. design variables. The **NumStates** keyword specifies the number of state variables, i.e. response quantities (e.g. displacements). The **NumEqConstraints** keyword specifies the number of equality constraints. The **NumIeqConstraints** keyword specifies the number of inequality constraints. The **MaxOptimizationItr** keyword specifies the maximum number of optimization iterations allow. The **ObjectiveFuncTol** keyword specifies the stopping criterion due to the objective function. The **GradientTol** keyword specifies the stopping criterion due to the norm of the gradient. The **TrialStepTol** keyword specifies the stopping criterion due to the norm of the search direction step.

Table 3.2. Parameters for general-use optimization commands.

Parameter	Argument	Default
NumControls	*Integer*	0
NumStates	*Integer*	0
NumEqConstraints	*Integer*	0
NumIeqConstraints	*Integer*	0
MaxOptimizationItr	*Integer*	5000
ObjectiveFuncTol	*Real*	1e-12
GradientTol	*Real*	1e-8
TrialStepTol	*Real*	1e-12

Nonlinear Conjugate Gradient

Nonlinear conjugate gradient (CG) methods [4] are first-order methods that are suited for the minimization of general nonlinear functions. This algorithm is appealing for small to medium size nonlinear optimization problems since each iteration only requires the evaluation of the objective function and its gradient. Unlike linear conjugate gradient meth-

ods, the convergence properties of nonlinear CG methods are not well understood [8]. The nonlinear CG methods available in DOTk are Fletcher-Reeves (`FLETCHER_REEVES`), Polak-Ribiere (`POLAK_RIBIERE`), Hestenes-Stiefel (`HESTENES_STIEFEL`), Conjugate Descent (`CONJUGATE_DESCENT`), Hager-Zhang (`HAGER_ZHANG`), Dai-Liao (`DAI_LIAO`), Dai-Yuan (`DAI_YUAN`), hybrid Dai-Yuan (`DAI_YUAN_HYBRID`), and Perry-Shanno (`PERRY_SHANNO`). Users are also expected to specify the corresponding line search method commands. However, default values can always be used.

C++ API

Set functions for nonlinear CG solution methods. The set functions for nonlinear CG solution methods are shown in Table 3.3. The **setSearchDirectionMethod** function sets the method used to compute the search direction. Users can use the set functions in Table 3.3 as follows:

```
instance.getSearchDirectionPtr()->setFunction(argument),
```

where `instance` specifies an object of class `DOTk_NonLinearCG` and both `setFunction` and `argument` are any of the options shown in Table 3.3.

Table 3.3. Set functions for nonlinear conjugate gradient solution method.

Function	Argument	Default
setSearchDirectionMethod	*dotk::types::direction_t*	dotk::types:: HAGER_ZHANG

MATLAB API

Parameters for nonlinear CG solution methods. The parameters for nonlinear CG solution methods are shown in Table 3.4. The **SearchDirectionMethod** keyword specifies the method used to compute the search direction. Figure 3.1 shows the source code for function `getSearchDirectionOptions`. This function is defined and declared in file `getOptimizationOptions.m`. This file denotes the MATLAB input file for DOTk. DOTk users should expect to have access to this file as part of the DOTk package.

Table 3.4. Parameters for nonlinear conjugate gradient solution method.

Parameter	Argument	Default
SearchDirectionMethod	*String*	HAGAR_ZHANG

```
function [Options] = getSearchDirectionOptions(Options)
% Search Direction Method
%   1. FLETCHER_REEVES
%   2. POLAK_RIBIERE
%   3. HESTENES_STIEFEL
%   4. CONJUGATE_DESCENT
%   5. HAGER_ZHANG
%   6. DAI_LIAO
%   7. DAI_YUAN
%   8. DAI_YUAN_HYBRID
%   9. PERRY_SHANNO
Options.SearchDirectionMethod = 'POLAK_RIBIERE';
end
```

Figure 3.1. Function `getSearchDirectionOptions` source code.

Quasi-Newton

Quasi-Newton methods are first-order methods that only require the gradient of the objective function at each iteration. This information can be used to construct an approximation to the inverse of the Hessian operator. In practice, quasi-Newton methods have been observed to produce superlinear convergence [8]. In cases when second-order information is not available, quasi-Newton methods are a good choice for solving optimization problems. DOTk gives users the flexibility to specify an user-defined inverse of the Hessian operator by using the keyword `USER_DEFINED_INV_HESS`. The inverse of the Hessian is defined through the operators API described in Chapter 1. Users are also expected to specify the parameters for the line search or trust region method of choice to properly enable quasi-Newton solution methods in DOTk.

C++ API

Set functions for quasi-Newton solution methods. The set functions for quasi-Newton solution methods are shown in Table 3.5. The **setInvHessianOperator** function sets the approximation method used to compute the inverse Hessian operator required to solve for the trial step $\mathbf{s}_k$, i.e.

$$\mathbf{s}_k = (\nabla^2 F(\mathbf{x}_k))^{-1} \nabla F(\mathbf{x}_k). \tag{3.1}$$

`instance.setFunction(argument)`,

where `instance` specifies an instance of class `DOTk_QuasiNewton` and `setFunction` and `argument` are any of the options shown in Table 3.5. Users can use the set functions in Table 3.5 as follows:

Table 3.5. Set function for quasi-Newton solution method.

Function	Argument	Default
setInvHessianPtr	*dotk::types::invhessian_t*	dotk::types::IDENTITY
	Integer	0

The second argument in set function `setInvHessianPtr` denotes the maximum number of gradient information stored to approximate the inverse Hessian operator. In practice, a value between 4 and 10 is typically used.

MATLAB API

Parameters for quasi-Newton solution methods. The parameters for quasi-Newton solution methods are given in Table 3.6. The **InvHessianOperator** keyword specifies the approximation method used to compute the inverse of the Hessian operator. This operator is require to solve Equation 3.1. The **MaxLimitedMemoryStorage** keyword specifies the maximum number of gradient information stored to approximate the inverse of the Hessian operator.

Table 3.6. Parameters for quasi-Newton solution method.

Parameter	Argument	Default
InvHessianOperator	*String*	IDENTITY
MaxLimitedMemoryStorage	*Integer*	0

Line Search Newton Conjugate Gradient

Line search Newton conjugate gradient methods [8], also known as truncated CG methods, are appropriate for large-scale unconstrained optimization problems. PDE-constrained optimization problems of the form shown in Equation 2.3 can also be efficiently solved with line search Newton CG methods. By employing a reduced-space formulation, the optimization problem in Equation 2.3 can be converted into an unconstrained optimization problem of the form shown in Equation 2.4.

Users can apply quasi-Newton techniques to approximate the Hessian operator when second-order derivative information is not available. However, users should keep in mind

that the second-order convergence rate associated with true Newton methods will be affected by the quasi-Newton approximation to the Hessian. Finally, users are expected to set the line search parameters and the method/approach used to compute the gradient and Hessian operators to enabled the line search Newton CG methods in DOTk. If a preconditioning strategy is enabled, users are also expected to specify the type of preconditioner strategy used for the problem.

C++ API

Set functions for line search Newton CG solution methods. The set functions for line search Newton CG solution methods are shown in Table 3.7. The **setMaxNumCGItr** function sets the maximum number of inner CG iterations allow to compute the trial step. The **setCGRelativeTol** keyword specifies the relative tolerance used to compute the inner CG loop stopping criterion. Users can use the set functions in Table 3.7 as follows:

```
instance.setFunction(argument),
```

where `instance` specifies an object of class `DOTk_LineSearchNewtonCG` and both `setFunction` and `argument` are any of the options shown in Table 3.7.

Table 3.7. Set functions for line search Newton conjugate gradient solution method.

Function	Argument	Default
setMaxNumCGItr	*Integer*	200
setCGRelativeTol	*Real*	1e-2

MATLAB API

Parameters for line search Newton CG solution methods. The parameters for line search Newton CG solution methods are shown in Table 3.8. The **MaxNumCGItr** keyword specifies the maximum number of inner CG iterations allow to compute the trial step. The **RelativeTol** keyword specifies the relative tolerance used to compute the inner CG loop stopping criterion.

Trust-Region Newton Conjugate Gradient

The trust-region Newton conjugate gradient method implemented in DOTk is due to Steinhaug [14]. This solution method is well suited for large-scale unconstrained optimization problems as well as PDE-constrained optimization problems solved using a reduced-space

Table 3.8. Parameters for line search Newton conjugate gradient solution method.

Parameter	Argument	Default
MaxNumCGItr	*Integer*	200
RelativeTol	*Real*	1e-2

formulation. In trust-region methods the trial step is computed by solving a trust-region subproblem of the form

$$\begin{aligned} \underset{s\in\mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2}\mathbf{s}^T\nabla^2 f(\mathbf{x}_k)\mathbf{s} + (\nabla f(\mathbf{x}_k))^T\mathbf{s} + f(\mathbf{x}_k) \\ \text{subject to} \quad & \\ & \|\mathbf{s}\| \leq \Delta_k. \end{aligned} \tag{3.2}$$

Users can apply quasi-Newton techniques to approximate the Hessian operator when second-order derivative information is not available. However, this approximation will impact the second-order convergence rate associated with true Newton methods. Users are expected to specify the trust-region method parameters and the method used to compute the gradient and Hessian operators. If a preconditioning strategy is enabled, users are also expected to specify the type of preconditioning strategy used for the problem.

C++ API

Set functions for trust-region Newton CG solution methods. The set functions for trust-region Newton CG solution methods are shown in Table 3.7. Users can use the set functions in Table 3.7 as follows:

```
instance.setFunction(argument),
```

where `instance` specifies an instance of class `DOTk_TrustRegionNewtonCG` and both `setFunction` and `argument` are any of the options shown in Table 3.7.

MATLAB API

Parameters for trust-region Newton CG solution methods. The parameters for trust-region Newton CG solution methods are shown in Table 3.8.

Inexact Trust-Region Sequential Quadratic Programming

The inexact trust-region Sequential Quadratic Programming (SQP) solution method [10] is well suited for the solution of constrained optimization problems such as large-scale PDE-constrained optimization problems [5, 11]. However, this solution method can be used to solve a vast range of constrained optimization problems encounter in many engineering applications. The inexact trust-region SQP solution method computes an approximate solution of the nonlinear programming problem by solving a sequence of quadratic subproblems which are built from a quadratic model of the Lagrangian and a linear Taylor approximation of the constraints. The advantage of this solution method is the efficient handling of the inherent inexactness due to the iterative solves performed to compute the trial step. Users are expected to specity the trust-region method parameters and the method used to compute the gradient and Hessian operators to enable the inexact trust-region SQP solution method. If a preconditioner strategy strategy is enabled, users are also expected to specify the type of preconditioning strategy used for the problem.

C++ API

Set functions for the inexact trust-region SQP solution method. The set functions for the inexact trust-region SQP solution method are shown in Table 3.9. The **setTangentialSubProblemMaxItr** function sets the maximum number of CG iterations allow to compute the effective tangential step. The **sol::solver::krylov_params(·)**[2] data structure sets the maximum number of iterations allow to solve the augmented system problem, restart frequency, and the problem dimensions, e.g. `sol::solver::krylov_params struct(dim(u), dim(g), max_itr, restart_freq)`. The restart feature is inactive by default. The **setConstraintTol** function sets the inexact trust-region SQP stopping criterion for the norm of the constraint. The **setQuasiNormalTol** function sets the stopping criterion in the quasi-normal step computation. The **setTangentialTol** function sets the stopping criterion in the corrected tangential step computation. The **setTangentialSubProblemProjectionTol** function sets the tangential subproblem projected gradient tolerance. This stopping criterion is used in the computation of the effective tangential step. The **setLagrangeMultipliersTol** keyword specifies the stopping criterion in the computation of the lagrange multipliers. The **setLagrangeMultipliersGradTol** function sets an upper bound on the stopping criterion used for the computation of the lagrange multipliers. The **setOrthogonalityTol** function sets stopping criterion used to control the inexactness cumulative effect due to the null-space projections. This stopping criterion is used in the computation of the effective tangential step. The **setTangentialTolReductionFactor** function specifies an internal reduction factor use to reduce the stopping criterion in the computation of the corrected tangential step. The **setTolReductionFactor** function sets a reduction factor used to adjust the stopping criterion for the augmented system solve required to compute the

[2] A new Krylov solver interface is under development and hence the **sol::solver::krylov_params (·)**data structure will be deprecated as soon as the new Krylov solver interface is made available to DOTk users.

quasi-normal step, effective tangential step, corrected tangential step, and lagrange multipliers. The **setActualOverPredictedReductionParam** function sets the stopping criterion used to determine if the trial step gives sufficient reduction in the merit function. The **setQuasiNormalTrustRegionFractionParam** function sets the trust-region reduction parameter in the computation of the quasi-normal step. The **setAllowedEffectiveTangentialOverTrialStepParam** function sets the trust region subpoblem stopping criterion due to the ratio between the effective tangential step and the trial step.

Users can use the set functions in Table 3.9 as follows:

```
instance.setFunction(argument),
```

where `instance` specifies an instance of class `DOTk_InexactSQP` and `setFunction` and `argument` are any of the options shown in Table 3.9.

Table 3.9. Set functions for inexact trust-region sequential quadratic programming solution method.

Function	Argument	Default
setTangentialSubProblemMaxItr	*Integer*	200
setConstraintTol	*Real*	1e-10
setQuasiNormalTol	*Real*	1e-4
setTangentialTol	*Real*	1e-4
setProjectionTol	*Real*	1e-4
setLagrangeMultipliersTol	*Real*	1e-4
setLagrangeMultipliersGradTol	*Real*	1e4
setOrthogonalityTol	*Real*	0.5
setTangentialTolReductionFactor	*Real*	1e-3
setTolReductionFactor	*Real*	1e-1
setActualOverPredictedReductionParam	*Real*	1e-8
setQuasiNormalTrustRegionFractionParam	*Real*	0.8
setAllowedEffectiveTangentialOverTrialStepParam	*Real*	2.
sol::solver::krylov_params	*Integer*	dim(u)=0
	Integer	dim(g)=0
	Integer	max_itr=200
	Integer	restart_freq=-1

MATLAB API

Parameters for the inexact trust-region SQP solution method. The parameters for the inexact trust-region SQP solution method are shown in Table 3.10. The **TangentialSubProblemMaxItr** keyword specifies the maximum number of CG iterations allow to compute the effective tangential step. The **MaxKrylovSolverItr** specifices the maximum number

of Krylov iterations allow to solve the augmented system problem. The **KrylovRestartItr** specifies the restart frequency for the Krylov solver. Recall that the restart feature is inactive by default. The **ConstraintTol** keyword specifies the inexact trust-region SQP stopping criterion for the norm of the constraint. The **QuasiNormalTol** keyword specifies the iterative solver stopping criterion in the quasi-normal step computation. The **TangentialTol** keyword specifies the iterative solver stopping criterion in the corrected tangential step computation. The **ProjectionTol** keyword specifies the projected gradient tolerance. This stopping criterion is used in the effective tangential step computation. The **LagrangeMultipliersTol** keyword specifies the iterative solver stopping criterion in the lagrange multipliers computation. The **LagrangeMultipliersGradTol** keyword specifies an upper bound on the solver stopping criterion used in the computation of the lagrange multipliers. The **OrthogonalityTol** keyword specifies a stopping criterion used to control the cumulative effect of inexactness due to the null-space projections. This stopping criterion is used in the computation of the effective tangential step. The **TangentialTolReductionFactor** keyword specifies a reduction factor used to decrease the solver stopping criterion in the computation of the corrected tangential step. The **TolReductionFactor** keyword specifies a reduction factor used to adjust the stopping criterion for the augmented system solve required to compute the quasi-normal step, effective tangential step, corrected tangential step, and lagrange multipliers. The **ActualOverPredictedReductionParam** keyword specifies the stopping criterion used to determine if the trial step gives sufficient reduction in the merit function. The **QuasiNormalTrustRegionFractionParam** keyword specifies a conservative trust-region reduction parameter used in the computation of the quasi-normal step. The **MaxEffectiveTangentialOverTrialStepParam** keyword specifies the trust-region subpoblem stopping criterion due to the ratio between the effective tangential step and the trial step.

Gradient Calculation

Gradient-based solution methods require first-order derivative information, i.e. gradient, to find an optimum to the optimization problem. There are cases where first-order derivative information is available to the user and is reasonable to expect the user to provide this information to DOTk solution methods. However, there are cases where first-order derivative information is not available to the user and other approaches, such as finite differencing, are needed to solve the optimization problem.

The finite differencing options available in DOTk are forward difference, backward difference, and central difference. DOTk provides users with a flexible interface that enable the parallelization of the finite diferencing techniques available in DOTk.

Table 3.10. Parameters for inexact trust-region sequential quadratic programming solution method.

Parameter	Argument	Default
TangentialSubProblemMaxItr	*Integer*	200
MaxKrylovSolverItr	*Integer*	200
KrylovRestartItr	*Integer*	-1
ConstraintTol	*Real*	1e-10
QuasiNormalTol	*Real*	1e-4
TangentialTol	*Real*	1e-4
ProjectionTol	*Real*	1e-4
LagrangeMultipliersTol	*Real*	1e-4
LagrangeMultipliersGradTol	*Real*	1e4
OrthogonalityTol	*Real*	0.5
TangentialTolReductionFactor	*Real*	1e-3
TolReductionFactor	*Real*	1e-1
ActualOverPredictedReductionParam	*Real*	1e-8
QuasiNormalTrustRegionFractionParam	*Real*	0.8
MaxEffectiveTangentialOverTrialStepParam	*Real*	2.

C++ API

Set functions for gradient computation. The set functions needed to select the method used to compute the gradient operator are shown in Table 3.11. The **setGradientOperator** function sets the method used to compute the gradient operator. The options keyword are the following:

1. `dotk::types::FORWARD_DIFFERENCE_GRAD`,
2. `dotk::types::BACKWARD_DIFFERENCE_GRAD`,
3. `dotk::types::CENTRAL_DIFFERENCE_GRAD`,
4. `dotk::types::USER_DEFINED_GRAD`,
5. `dotk::types::PARALLEL_FORWARD_DIFFERENCE_GRAD`,
6. `dotk::types::PARALLEL_BACKWARD_DIFFERENCE_GRAD`,
7. `dotk::types::PARALLEL_CENTRAL_DIFFERENCE_GRAD`.

The **setFiniteDifferencePerturbationVec** function sets the vector of finite positive perturbations. The dimensions of the vector is equal to the number of optimization/design variables. DOTk users can set the gradient operator as follows:

```
instance.setGradientPtr(dotk::types:gradient_t),
```

where the default gradient operator is set to `dotk::types::USER_DEFINED_GRAD`. Thus, DOTk expects users to provide an user defined gradient operator, i.e. analytical gradient, to solve the optimization problem of interest. DOTk users can set the gradient operator as follows:

`instance.getGradientPtr()->setFunction(argument)`,

where `instance` specifies a DOTk solution method class object, `getGradientPtr()` returns a C++ shared pointer that grants access to all the public functions in class `DOTk_Gradient`, and both `setFunction` and `argument` are any of the options shown in Table 3.11.

Table 3.11. Set functions for gradient computation.

Function	Argument	Default
setFiniteDifferencePerturbationVec	*std::vector<Real>*	1e-6

MATLAB API

Parameters for gradient computation. The parameters needed to select the method used to compute the gradient gradient are shown in Table 3.12. The **GradientOperator** keyword specifies the technique used to compute the gradient operator. The options are `FORWARD_DIFFERENCE_GRAD`, `BACKWARD_DIFFERENCE_GRAD`, `CENTRAL_DIFFERENCE_GRAD`, `USER_DEFINED_GRAD`, `PARALLEL_FORWARD_DIFFERENCE_GRAD`, `PARALLEL_BACKWARD_DIFFERENCE_GRAD`, and `PARALLEL_CENTRAL_DIFFERENCE_GRAD`. The **FiniteDifferencePerturbation** keyword specifies a finite positive perturbation. Figure 3.2 shows the source code for function `getGradientOperatorOptions`. Function `getGradientOperatorOptions` is defined and declared in file `getOptimizationOptions.m`. DOTk users are expected to modified the keywords to set desired gradient operator options.

Table 3.12. Parameters for gradient computation.

Parameter	Argument	Default
GradientOperator	*String*	USER_DEFINED_GRAD
FiniteDifferencePerturbation	*Real*	1e-6

Hessian Calculation

Users are always recommended to provide second-order derivative information, e.g. Hessian, to exploit the fast convergence rates of DOTk solution methods. However, there are occasions where users cannot provide second-order derivative information. For such cases, DOTk provides useful quasi-Newton techniques to approximate the Hessian or inverse of the

```
function [Options] = getGradientOperatorOptions(Options)
% Gradient Operator Options:
%   1. FORWARD_DIFFERENCE_GRAD
%   2. BACKWARD_DIFFERENCE_GRAD
%   3. CENTRAL_DIFFERENCE_GRAD
%   4. USER_DEFINED_GRAD
%   5. PARALLEL_FORWARD_DIFFERENCE_GRAD
%   6. PARALLEL_BACKWARD_DIFFERENCE_GRAD
%   7. PARALLEL_CENTRAL_DIFFERENCE_GRAD
Options.GradientOperator = 'USER_DEFINED_GRAD';
if (~strcmp(Options.GradientOperator,'USER_DEFINED_GRAD'))
    Options.FiniteDifferencePerturbation = [1e-5, 1e-6];
end

end
```

Figure 3.2. Function `getGradientOperatorOptions` source code.

Hessian operators. The quasi-Newton methods available in DOTk are Broyden-Fletcher-Goldfarb-Shanno (`BFGS`), limited-memory BFGS (`LBFGS`), limited-memory Davidon-Fletcher-Powell (`LDFP`), symmetric-rank-one (`SR1`), and limited-memory SR1 (`LSR1`). DOTk also offers an additional capability that allow users to approximate the Hessian operator by applying a forward finite difference (`FORWARD_FD_HESS`) technique.

C++ API

Set functions for quasi-Newton approximation methods. The set functions for the quasi-Newtonapproximantion methods are shown in Table 3.13. The **setHessianPtr(dotk::types::hessian_t, Integer)** is used to set the method to compute the application of a vector to the Hessian operator. The available `hessian_t` options are:

1. `dotk::types::LBFGS_HESS`
2. `dotk::types::LDFP_HESS`
3. `dotk::types::LSR1_HESS`
4. `dotk::types::SR1_HESS`
5. `dotk::types::DFP_HESS`

6. `dotk::types::IDENTITY_HESS`

7. `dotk::types::USER_DEFINED_HESS`

8. `dotk::types::FORWARD_FD_HESS`

The **setInvHessianPtr(dotk::types::invhessian_t, Integer)** function sets the method used to compute the application of a vector to the inverse of the Hessian operator. The available `invhessian_t` options are:

1. `dotk::types::LBFGS_INV_HESS`

2. `dotk::types::LDFP_INV_HESS`

3. `dotk::types::LSR1_INV_HESS`

4. `dotk::types::SR1_INV_HESS`

5. `dotk::types::BFGS_INV_HESS`

6. `dotk::types::IDENTITY_INV_HESS`

7. `dotk::types::USER_DEFINED_INV_HESS`

The second argument in both **setHessianPtr** and **setInvHessianPtr** functions is used to specify the maximum number of gradient information stored to approximate the quasi-Newton Hessian and inverse of the Hessian operators. This value is set to zero by default and hence users are expected to specify this number if a quasi-Newton Hessian approximation method is utilized.

Users can use the set functions in Table 3.13 to set additional options as follows:

`instance.getHessianPtr()->setFunction(argument)`,

`instance.getInvHessianPtr()->setFunction(argument)`,

where `instance` specifies a DOTk solution method class object, `getHessianPtr()` and `getInvHessianPtr()` return a C++ shared, i.e. smart, pointer that grants access to all the public functions of class `DOTk_Hessian` and `DOTk_InvHessian`, respectively. The `setFunction` and `argument` are any of the options shown in Table 3.13.

Table 3.13. Set functions for quasi-Newton approximation methods.

Function	Argument	Default
setMaxLimitedMemoryStorage	*Integer*	0

Parameters for quasi-Newton approximation methods. The parameters for quasi-Newton approximation methods are shown in Table 3.14. The **HessianOperator** keyword specifies the quasi-Newton technique used to approximate the Hessian. The available options are `LBFGS_HESS`, `LDFP_HESS`, `LSR1_HESS`, `SR1_HESS`, `DFP_HESS`, `IDENTITY_HESS`, and `USER_DEFINED_HESS`. The **InvHessianOperator** keyword specifies the quasi-Newton technique used to approximate the inver of the Hessian. The available options are `LBFGS_INV_HESS`, `LDFP_INV_HESS`, `LSR1_INV_HESS`, `SR1_INV_HESS`, `BFGS_INV_HESS`, `IDENTITY_INV_HESS`, `USER_DEFINED_INV_HE` and `FORWARD_FD_HESS`. The **MaxLimitedMemoryStorage** keyword specifies the maximum number of gradient information stored to compute the quasi-Newton approximation of the second-order derivative information. Figures 3.3 and 3.4 respectively show the source code for functions `getHessianOperatorOptions` and `getInvHessianOperatorOptions`. These functions are defined and declared in file `getOptimizationOptions.m`. DOTk users should expect to have access to the MATLAB input file, i.e. `getOptimizationOptions.m`.

Table 3.14. Parameters for quasi-Newton approximation methods.

Parameter	Argument	Default
HessianOperator	*String*	USER_DEFINED_HESS
InvHessianOperator	*String*	IDENTITY_INV_HESS
MaxLimitedMemoryStorage	*Integer*	0

```
function [Options] = getHessianOperatorOptions(Options)
% Hessian Operator Options:
%   1. LBFGS_HESS
%   2. LDFP_HESS
%   3. LSR1_HESS
%   4. SR1_HESS
%   5. DFP_HESS
%   6. IDENTITY_HESS
%   7. USER_DEFINED_HESS
Options.HessianOperator = 'USER_DEFINED_HESS';
end
```

Figure 3.3. Function `getHessianOperatorOptions` source code.

```
function [Options] = getInvHessianOperatorOptions(Options)
% Inverse Hessian Operator Options:
%   1. LBFGS_INV_HESS
%   2. LDFP_INV_HESS
%   3. LSR1_INV_HESS
%   4. SR1_INV_HESS
%   5. BFGS_INV_HESS
%   6. IDENTITY_INV_HESS
%   7. USER_DEFINED_INV_HESS
Options.InvHessianOperator = 'IDENTITY_INV_HESS';
end
```

Figure 3.4. Function `getInvHessianOperatorOptions` source code.

Line Search Methods

Line search methods promote the convergence of gradient-based solution methods from a remote starting point. These methods are used to find the suitable scaled search direction along which the objective function will be optimize. The step size used to scale the search direction is choosen such that certain criteria are satisfied.

C++ API

Set functions for line search methods. The set functions used to defined the line search method parameters are shown in Table 3.15. The **setStepPtr(dotk::types::linesearch_t)** allows users to select the line search method used to compute the optimal step size. The available options are:

1. `dotk::types::BACKTRACKING_ARMIJO`
2. `dotk::types::BACKTRACKING_GOLDSTEIN`
3. `dotk::types::BACKTRACKING_CUBIC_INTRP`
4. `dotk::types::BACKTRACKING_HAGER_ZHANG`
5. `dotk::types::GOLDENSECTION`

DOTk users can set the line search method as follows:

```
instance.setStepPtr(dotk::types::linesearch_t),
```

where `instance` specifies a DOTk solution method class object. The defualt line search method is set to `dotk::types::BACKTRACKING_CUBIC_INTRP`.

DOTk users can set additional line search options by using the line search set functions in Table 3.15 as follows:

```
instance.getLineSearchPtr()->setFunction(argument).
```

Here, `instance` specifies a DOTk solution method class object, `getStepPtr()` returns a C++ shared pointer that grants access to all the public functions in class `DOTk_LineSearch`, and both `setFunction` and `argument` are any of the options shown in Table 3.15. The **setMaxNumIterations** function sets the maxmimum number of line search iterations performed and hence the number of additional objective function evaluations done to compute the optimal step size. The **setStepTolerance** function sets the minimum step size allow. The **setContractionFactor** function sets how much the step size is contracted at each line search iteration.

Table 3.15. Set functions for line search methods.

Function	Argument	Default
setMaxNumIterations	*Integer*	50
setStepTolerance	*Real*	1e-8
setContractionFactor	*Integer*	0.5

MATLAB API

Parameters for line search methods. The parameters for line search methods are shown in Table 3.16. The **LineSearchMethod** keyword specifies the line search method used to compute the step size. The line search methods keywords are `BACKTRACKING_ARMIJO`, `BACTRACKING_GOLDSTEIN`, `BACKTRACKING_CUBIC_INTRP` `BARZILAI_BORWEIN`, and `GOLDENSECTION`. The **MaxLineSearchItr** keyword specifies the maxmimum number of objective function evaluations allow to compute the optimal step size. The **LineSeachStepTol** keyword specifies the minimum step size allow. The **LineSearchContractionFactor** keyword specifies how much the step size is contracted in each line search iteration. Figure 3.5 shows the source code for the `getLineSearchOptions` function in file `getOptimizationOptions.m`. This file is available/provided to DOTk users, i.e. MATLAB input file to DOTk.

Trust-Region Methods

Trust-region methods, similar to line search methods, promote the convergence of gradient-based solution methods from a remote starting point. The basis for trust-region methods is

Table 3.16. Parameters for line search methods.

Parameter	Argument	Default
LineSearchMethod	*String*	BACKTRACKING_CUBIC_INTRP
MaxLineSearchItr	*Integer*	50
LineSeachStepTol	*Real*	1e-8
LineSearchContractionFactor	*Integer*	0.5

```
function [Options] = getLineSearchOptions(Options)
% Line Search Methods:
%   1. BACKTRACKING_ARMIJO
%   2. BACKTRACKING_GOLDSTEIN
%   3. BACKTRACKING_CUBIC_INTRP
%   4. GOLDENSECTION
Options.LineSearchMethod = 'BACKTRACKING_CUBIC_INTRP';
% General Line Search Options
Options.MaxLineSearchItr = 50;
Options.LineSeachStepTol = 1e-8;
Options.ArmijoConstant = 1e-4;
Options.LineSearchContractionFactor = 0.5;
end
```

Figure 3.5. Line search method options.

to define a model that adequately represents the objective function in a trustworthy region around the current iterate. This model is then used to find the search direction that minimizes the model inside this trusted region. If the search direction is not appropriate, the trust region is reduced and a new search direction that minimizes the model is computed.

C++ API

Set functions for trust region methods. The set functions used to defined the trust region method parameters are shown in Table 3.17. The **setTrustRegionStepPtr(dotk::types::trustregion_t)** allows users to select the trust region method used to compute the optimal trust region radius at each iteration. The available trust region methods are:

1. `dotk::types::DOGLEG`
2. `dotk::types::DOGLEG_HYBRID`

3. `dotk::types::DOUBLE_DOGLEG`

4. `dotk::types::DOUBLE_DOGLEG_HYBRID`

5. `dotk::types::STEP_DISABLED`

DOTk users can then set the trust region method as follows:

```
instance.setTrustRegionStepPtr(dotk::types::trustregion_t),
```

where `instance` specifies a DOTk solution method class object. The defualt trust region method is set to `dotk::types::STEP_DISABLED` and hence DOTk users have to specify the trust region method by default.

The set functions used to specify additional options for trust-region methods are shown in Table 3.17. The **setMaxTrustRegionSubProblemIterations** function sets the maximum number of trust-region iterations allow. The **setMaxTrustRegionRadius** function sets the maximum trust-region radius allow. The **setMinTrustRegionRadius** function sets the minimum trust-region radius allow. The **setTrustRegionRadius** function sets the starting trust-region radius. The **setMinimumActualReductionRatioAllowed** function sets the minimum threshold for the ratio between the actual and predicted reduction. The **setContractionParameter** function sets how much the trust-region radius is contracted if the ratio between the actual and predicted reduction is less than a threshold. The **setExpansionParameter** function sets how much the trust-region radius is expanded if the ratio between the actual and predicted reduction is greater than the ratio threshold. Hybrid trust region methods have additional set functions. The **setMaxLineSearchIterations** function sets the maximum number of line search iterations. The **setLineSearchConstant** sets the armijo constant. The **setLineSearchStepTolerance** sets the minimum line search step allowed. The **setLineSearchContractionFactor** sets the line search step contraction parameter.

Users can use the set functions in Table 3.17 as follows:

```
instance.getTrustRegionStepPtr()->setFunction(argument),
```

where `instance` specifies an object of class `DOTk_InexactSQP`, `DOTk_TrustRegionNewtonCG`, or `DOTk_TrustRegionQuasiNewton`. The `getTrustRegionStepPtr` function returns a C++ shared pointer that grants access to all the public functions in the trust region class. The `setFunction` and `argument` are any of the options shown in Table 3.17.

MATLAB API

Parameters for trust-region methods. The **TrustRegionMethod** keyword is used to specify the trust region method apply to solve the problem. The available trust region method options are `TRUST_REGION_DOGLEG`, `TRUST_REGION_DOGLEG_HYBRID`, `TRUST_REGION_`

Table 3.17. Set functions for trust-region methods.

Function	Argument	Default
setMaxTrustRegionSubProblemIterations	*Integer*	50
setMaxTrustRegionRadius	*Real*	1e4
setMinTrustRegionRadius	*Real*	1e-8
setTrustRegionRadius	*Real*	1e4
setMinimumActualReductionRatioAllowed	*Real*	0.25
setContractionParameter	*Real*	0.5
setExpansionParameter	*Real*	2.0
setMaxLineSearchIterations	*Integer*	50
setLineSearchConstant	*Real*	1e-4
setLineSearchStepTolerance	*Real*	1e-8
setLineSearchContractionFactor	*Real*	0.5

`DOUBLE_DOGLEG`, and `TRUST_REGION_DOUBLE_DOGLEG_HYBRID`. The additional trust region parameters are given in Table 3.18. The **MaxTrustRegionItr** keyword specifies the maximum number of trust-region iterations allow. The **MaxTrustRegionRadius** keyword specifies the maximum trust-region radius allow. The **MinTrustRegionRadius** keyword specifies the minimum trust-region radius allow. The **TrustRegionRadius** keyword specifies the starting trust-region radius. The **AllowableMinimumActualReductionRatio** keyword specifies the minimum threshold for the ratio between the actual and predicted reduction. The **TrustRegionRadiusContractionParam** keyword specifies how much the trust-region radius is contracted if the ratio between the actual and predicted reduction is less than a threshold. The **TrustRegionRadiusExpansionParam** keyword specifies how much the trust-region radius is expanded if the ratio between the actual and predicted reduction is greater than the ratio threshold. Figure 3.6 shows the source code for the `getTrustRegionOptions` function in file `getOptimizationOptions.m`. This file is available/provided to DOTk, i.e. MATLAB input file to DOTk.

Table 3.18. Parameters for trust-region methods.

Parameter	Argument	Default
MaxTrustRegionItr	*Integer*	50
MaxTrustRegionRadius	*Real*	1e4
MinTrustRegionRadius	*Real*	1e-8
TrustRegionRadius	*Real*	1e4
AllowableMinimumActualReductionRatio	*Real*	0.25
TrustRegionRadiusContractionParam	*Real*	0.5
TrustRegionRadiusExpansionParam	*Real*	2.0

```
function [Options] = getTrustRegionOptions(Options)
% Trust-Region Methods:
%   1. TRUST_REGION_DOGLEG
%   2. TRUST_REGION_DOGLEG_HYBRID
%   3. TRUST_REGION_DOUBLE_DOGLEG
%   4. TRUST_REGION_DOUBLE_DOGLEG_HYBRID
Options.TrustRegionMethod = 'TRUST_REGION_DOGLEG';
% General trust region options
Options.MaxTrustRegionItr = 30;
Options.TrustRegionRadius = 1e4;
Options.MaxTrustRegionRadius = 1e4;
Options.MinTrustRegionRadius = 1e-6;
Options.TrustRegionRadiusContractionParam = 0.5;
Options.TrustRegionRadiusExpansionParam = 2;
Options.AllowableMinimumActualReductionRatio = 0.15;
end
```

Figure 3.6. Trust region method options.

Preconditioning Strategies

The current version of DOTk provides several preconditioning options for full-space PDE-constrained optimization problems [9]. Users are expected to implement the essential operators to enable these preconditioners. *However, there is ongoing research to explore suitable preconditioning strategies for reduced-space and full-sapce PDE-constrained problems. The ultimate goal is to facilitate multiple preconditioning strategies without users having to provide the essential operators.*

C++ API

Set functions for preconditioning strategies. The set functions used to specify the preconditioning strategies are shown in Table 3.19. The **setLeftPreconditionerType** function sets the left preconditioning strategy, i.e. $P^{-1}(Ax - b) = 0$. The left preconditioning strategies for reduced-space PDE-constrained optimization problems are:

1. `dotk::types::NO_PREC`
2. `dotk::types::USER_DEFINED_PREC`

The left preconditioning strategies for full-space PDE-constrained optimization problems are:

1. `dotk::types::NO_PREC`
2. `dotk::types::USER_DEFINED_PREC`
3. `dotk::types::FULL_SCHUR_PREC`
4. `dotk::types::INCOMPLETE_SCHUR_PREC`

The **setRightPreconditionerType** function sets the right preconditioning strategy, i.e. $AP^{-1}Px = b$. The right preconditioning strategies available in DOTk for both reduced-space and full-space PDE-constrained optimization are:

1. `dotk::types::NO_PREC`
2. `dotk::types::USER_DEFINED_PREC`

Users can set the preconditioning strategy as follows:

1. Define a C++ shared pointer that owns a pointer to a child class of parent class `DOTk_PreconditionerOperators`, see Preconditioner API subsection in Chapter 2, as follows:

   `std::tr1::shared_ptr<ChildClass> sptr(new ChildClass())`,

   where `ChildClass` specifies the child class name and `sptr` is an object of `std::tr1::shared_ptr`.

2. Define a C++ shared pointer that owns a pointer to a DOTk preconditioner API as follows:

   `std::tr1::shared_ptr<PrecAPI> sptr(new PrecAPI(PrecOp, ParamSpace))`,

   where `PrecAPI` is a `DOTk_FullSpacePrecInterface` or `DOTk_ReducedSpacePrecInterface` object, `PrecOp` is the C++ shared pointer defined in Step 1, and `ParamSpace` is a `sol::params::space`[3] data structure, see the General use parameters section in Chapter 3.

3. Use functions in Table 3.19 to set the preconditioning strategy as follows:

   `sptr->setFunction(argument)`,

   where `sptr` is the C++ shared pointer defined in Step 2 and both `setFunction` and `argument` are options defined in Table 3.19.

[3] A new full-space interface is currently under development and hence the current full-space formulation strategy interface will be deprecated as soon as the new interface is available to DOTk users.

Table 3.19. Set functions for preconditioning strategies.

Function	Argument	Default
setLeftPreconditionerType	*dotk::types::preconditioner_t*	NO_PREC
setRightPreconditionerType	*dotk::types::preconditioner_t*	NO_PREC

MATLAB API

Parameters for preconditioning strategies. The parameters to specify a preconditioning strategy are shown in Table 3.20. The **LeftPreconditionerType** keyword specifies the left preconditioning strategy. The available left preconditioning strategies for reduced-space PDE-constrained optimization problems are `NO_PREC` and `USER_DEFINED_PREC`. The available left preconditioning strategies for full-space PDE-constrained optimization problems are `NO_PREC`, `USER_DEFINED_PREC`, `FULL_SCHUR_PREC`, and `INCOMPLETE_SCHUR_PREC`. The **RightPreconditionerType** keyword specifies the right preconditioning strategy. The right preconditioning strategies available in DOTk for both reduced-space and full-space PDE-constrained optimization are `NO_PREC` and `USER_DEFINED_PREC`. Figures 3.7 and 3.8 show the source code for functions `getFullSpaceLeftPrecOptions` and `getFullSpaceRightPrecOptions`. These functions are defined and declared in file `getOptimizationOptions.m`.

Table 3.20. Parameters for preconditioning strategies.

Parameter	Argument	Default
LeftPreconditionerType	*String*	NO_PREC
RightPreconditionerType	*String*	NO_PREC

```
function [Options] = getFullSpaceLeftPrecOptions(Options)
% Preconditioner Types:
%   1. NO_PREC
%   2. USER_DEFINED_PREC
%   3. FULL_SCHUR_PREC
%   4. INCOMPLETE_SCHUR_PREC
Options.LeftPreconditionerType = 'FULL_SCHUR_PREC';
end
```

Figure 3.7. Function `getFullSpaceLeftPrecOptions` source code.

```
function [Options] = getFullSpaceRightPrecOptions(Options)
% Preconditioner Types:
%   1. NO_PREC
%   2. USER_DEFINED
Options.RightPreconditionerType = 'NO_PREC';
end
```

Figure 3.8. Function `getFullSpaceRightPrecOptions` source code.

Chapter 4

Constraint Modeling

Constrained optimization is the process of optimizing an objective function with respect to a set of design variables in the existence of constraints. These problems can be classified as equality-constrained, inequality-constrained, constrained (both equality and inequality constraints are existent), and bound-constrained optimization problems. DOTk offers users several constraint modeling methods to solve these class of optimization problems.

Bound Constraints

Lets consider the constrained optimization problem of the form

$$\min f(x) \text{ s.t. } x \in X,$$

where

1. X is a convex and close space defined as $X = \{x \mid a \leq x \leq b\}$
2. $f : X \to \mathbb{R}$ is continuously differentiable over X.

The solution methods that are often applied to solve unconstrained optimization problems, combined with the appropriate bound constraint modeling techniques, can be easily applied to solve the resulting bound constrained optimization problem.

DOTk offers users several alternative to solve bound-constrained optimization problems. The bound-constraint modeling methods available to DOTk users are *feasible direction*, *minimum reduction on feasible direction*, *projection of scaled direction into feasible set*, and *minimum reduction of projected scaled direction into feasible set*. DOTk users are referred to Bertsekas book [3] for a great theoretical discussion on these methods. *Users are reminded that these bound-constraint modeling methods only apply to bound constraints and thus should not be applied to solve inequality constrained optimization problems. Furthermore, these methods should only be used to solve reduced-space optimization problems.* Interior-point methods, i.e. barrier methods, are the suitable constraint modeling approach to solve general constrained optimization problems. These constraint modeling techniques are currently under development and should be available to DOTk users in the next DOTk release.

C++ API

Set functions for bound constraint modeling methods. The set function used to enable bound constraint optimization are shown in Table 4.1. The **setControlBounds** function sets the n-dimensional arrays of lower and upper bounds. This automatically sets the **ActiveBoundConstraint** flag to true. This flag is set to **false** by default. Users can use the set functions in Table 4.1 as follows:

`instance.setControlBounds(argument)`,

where `instance` specifies a DOTk vector space class object, e.g. `dotk::DOTk_VectorSpace-Reduced vectorSpace`. Thus, the bounds on the control are managed by the `dotk::DOTk_VectorSpaceReduced` class manager. This class is in charge of managing the data structure for all the solution methods used to solve reduced-space optimization problems. The `argument` keyword denotes any of the options shown in Table 4.1.

DOTk users can set the bound constraint modeling method as follow

`instance.setConstraintPtr(dotk::types::constraint_t)`,

where `instance` denotes a class object from any of the reduced-space solution methods available in DOTk. The available `dotk::types::constraint_t` options are

1. `dotk::types::CONSTRAINT_OFF`
2. `dotk::types::BOUND_FEASIBLE_DIRECTION`
3. `dotk::types::BOUND_FEASIBLE_DIRECTION_MIN_REDUCTION`
4. `dotk::types::BOUND_PROJECT_SCALED_DIRECTION`
5. `dotk::types::BOUND_PROJECTION_ALONG_FEASIBLE_DIRECTION`

Bound constraint are disabled by default and hence DOTk users have to enable bound constraints to solve a bound constraint optimization problem.

Table 4.1. Set functions for bound constraint modeling methods

Function	Argument	Default
setControlBounds	*std::vector<Real>*	Inactive
	std::vector<Real>	Inactive

MATLAB API

Parameters for bound constraint modeling methods. The parameters that enable bound constraint optimization are shown in Table 4.2. The **ConstraintMethod** keyword specifies the bound constraint modeling method. The available options are `BOUND_FEASIBLE_DIRECTION`, `BOUND_FEASIBLE_DIRECTION_MIN_REDUCTION`, `BOUND_PROJECT_SCALED_DIRECTION`, `BOUND_PROJECTION_ALONG_FEASIBLE_DIRECTION`. The **StepSize** keyword denotes the step size used to contract the feasible direction. The feasible direction can be scaled at every iteration for both feasible direction and projection bound constraint modeling methods. DOTk users should used a step size of 0.5 for `BOUND_FEASIBLE_DIRECTION`. Other bound constraint modeling techniques usually do not require scaling of the feasible direction. However, DOTk users can opt to scale the feasible direction since faster convergence rates can be obtained[1]. The **LowerBounds** keyword specifies the n-dimensional array of lower bounds. The **UpperBounds** keyword specifies the n-dimensional array of upper bounds. Figure 4.1 shows the source code for the function `getConstraintOptions`. This function is defined and declared in file `getOptimizationOptions.m`.

Table 4.2. Parameters for bound constraint modeling methods

Parameter	Argument	Default
ConstraintMethod	*String*	OFF
StepSize	*Real*	1.0
LowerBounds	*Real*	-1
UpperBounds	*Real*	-1

Interior Point Methods

The current version of DOTk does not support interior point methods [15] for general constraint modeling. Current efforts exists to provide interior point constraint modeling techniques in the next version of DOTk.

[1]DOTk users should understand that this behavior is problem dependent.

```
function [Options] = getConstraintOptions(Options)
% Constraint Modeling Method
%   1. OFF
%   2. BOUND_FEASIBLE_DIRECTION
%   3. BOUND_FEASIBLE_DIRECTION_MIN_REDUCTION
%   4. BOUND_PROJECT_SCALED_DIRECTION
%   5. BOUND_PROJECTION_ALONG_FEASIBLE_DIRECTION
Options.ConstraintMethod = 'BOUND_PROJECT_SCALED_DIRECTION';
if(~strcmp(Options.ConstraintMethod,'OFF'))
    Options.StepSize = 1.0;
    Options.LowerBounds = zeros(Options.NumControls,1);
    Options.UpperBounds = 5*ones(Options.NumControls,1);
end
end
```

Figure 4.1. Function `getConstraintOptions` source code.

Chapter 5

Diagnostic Tools

DOTk softaware package provides a combination of tools to validate the correctness of the formulation and monitor the progression of the optimization problem at each iteration. This chapter describes the diagnostic tools available in DOTk and outlines how to enable these tools in both C++ and MATLAB APIs. *Current efforts are in progress to enable Python [13] post-processing tools to DOTk users.* Finally, examples are provided to illustrate the proper use of these tools.

Text Files

DOTk provides users with a set of data files that allows them to monitor the progression of the optimization problem at each iteration. The progress report is available through the data file `DOTk_SolutionMethodName.out`, where `SolutionMethod` specifies the solution method in use. The five progress report options are `DOTk_NonLinearCG.out`, `DOTk_QuasiNewton.out`, `DOTk_LineSearchNewtonCG.out`, `DOTk_TrustRegionNewtonCG.out`, and `DOTk_InexactSQP.out`. The optimal solution at a given iteration is available through the data file `DOTk_VariableType_solution.out`, where `VariableType` is either state or control, e.g. `DOTk_state_solution.out`. Additional post-processing flexibility is possible when employing DOTk solution methods from the MATLAB API. The MATLAB API relieves users from the tedious process of transfering text data into MATLAB. Examples of progress report output files are shown in Figures 5.1-5.4.

Progress report keywords. The **Iteration** keyword specifies the number of optimization iterations. The **Func-count** keyword specifies the cummulative number of function evaulations at a given iteration. The **F(x)** keyword specifies the value of the objective function. The **norm(G)** keyword specifies the norm of the gradient. The **norm(P)** keyword specifies the norm of the trial step. The **norm(C)** keyword specifies the norm of the constraints. The **norm(N)** keyword specifies the norm of the normal step. The **norm(T)** keyword specifies the norm of the tangential step. The **LineSrch-Step** keyword specifies the line search step size. The **LineSrch-Itr** keyword specifies the number of line search iterations taken to compute the step size. The **TR-Itr** keyword specifies the number of trust-region iterations taken to compute the trial step. The **TR-Radius** keyword specifies the trust-region radius at a given iteration. The **ActualReduc** keyword specifies the decrease in the objective function

Iteration	Func-count	F(x)	norm(G)	norm(P)	LineSrch-Step	LineSrch-Itr
0	1	5.8000e+00	1.2517e+02	*	*	*
1	7	1.0910e-01	1.5033e+01	1.2517e+02	8.6491e-04	6
2	9	1.5271e-02	5.1949e+00	5.1761e-01	2.7187e-01	2
3	11	8.1752e-04	8.9868e-01	2.2667e-01	3.2703e-01	2
4	12	2.5835e-04	6.1961e-01	2.7958e-02	1.0000e+00	1
5	13	1.4392e-04	1.5630e-01	7.1368e-03	1.0000e+00	1
6	14	1.3017e-04	1.5069e-01	1.3023e-03	1.0000e+00	1
7	15	1.7667e-06	5.3610e-02	2.3234e-02	1.0000e+00	1
8	16	2.2101e-08	1.8016e-03	9.7382e-04	1.0000e+00	1
9	17	3.4009e-11	1.5641e-04	3.3080e-04	1.0000e+00	1
10	18	1.0781e-14	4.4842e-06	1.0512e-05	1.0000e+00	1

Figure 5.1. Progress report for nonlinear CG and quasi-Newton solution methods.

Iteration	Func-count	F(x)	norm(G)	norm(P)	TR-Itr	TR-Radius	ActualReduc	PredReduc	Ratio	Krylov-Itr	Krylov-Error	Krylov-Exit
0	1	5.8000e+00	1.2517e+02	*	*	*	*	*	*	*	*	*
1	2	3.8384e-02	3.9982e-01	2.3024e-01	1	1.0000e+04	5.7616e+00	5.7608e+00	1.0001e+00	2	1.8371e-12	Tolerance
2	3	3.8384e-02	3.9982e-01	5.0621e-01	1	5.0000e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	2	2.5000e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	3	1.2500e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	4	6.2500e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	5	3.1250e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	6	1.5625e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	7	7.8125e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	8	3.9062e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	9	1.9531e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	10	9.7656e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	11	4.8828e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	12	2.4414e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	13	1.2207e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	14	6.1035e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	15	3.0518e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	16	6.1035e-01	1.3895e-02	3.2231e-02	4.3110e-01	1	1.6297e-01	TrustRegion
3	19	5.3029e-03	7.2863e-02	5.8007e-03	1	6.1035e-01	1.9187e-02	1.9110e-02	1.0040e+00	1	4.9192e-02	Tolerance
4	20	3.3045e-03	2.5559e+00	1.7953e-01	1	6.1035e-01	1.9984e-03	5.5193e-03	3.6208e-01	2	1.3045e-14	Tolerance
5	21	2.7801e-05	8.4393e-03	2.5593e-03	1	6.1035e-01	3.2767e-03	3.2707e-03	1.0018e+00	1	7.0694e-03	Tolerance
6	22	7.5800e-08	1.2123e-02	1.1659e-02	1	6.1035e-01	2.7725e-05	2.7598e-05	1.0046e+00	2	8.2971e-16	Tolerance
7	23	2.4293e-09	4.4051e-05	1.2104e-05	1	6.1035e-01	7.3371e-08	7.3370e-08	1.0000e+00	1	4.4104e-05	Tolerance
8	24	5.9194e-16	1.0811e-06	1.1029e-04	1	6.1035e-01	2.4293e-09	2.4292e-09	1.0000e+00	2	3.0540e-21	Tolerance

Figure 5.2. Progress report for trust-region Newton CG solution methods.

at a given iteration. The **PredReduc** keyword specifies the decrease of the quadratic model at a given iteration. The **Ratio** keyword specifies the ratio between the actual and the predicted reduction. The **Krylov-Itr** keyword specifies the total number of inner loop iterations, i.e. Krylov iterations, needed to compute the trial step. The **Krylov-Error** keyword specifies the Krylov problem residual norm. The **Krylov-Exit** keyword specifies the Krylov problem convergence criterion. The possible convergence criteria are the following:

1. `NegCurvature:` negative curvature was observed, i.e. indefinite/negative definite Hessian;

2. `Tolerance:` residual tolerance was met;

3. `TrustRegion:` trust-region constraint violated;

4. `MaxItr:` maximum number of allowed Krylov iteration exceeded;

Iteration	Func-count	F(x)	norm(G)	norm(P)	Krylov-Itr	Krylov-Error	Krylov-Exit	LineSrch-Step	LineSrch-Itr
0	1	5.8000e+00	1.2517e+02	*	*	*	*	*	*
1	2	3.8384e-02	3.9982e-01	2.3024e-01	2	1.8371e-12	Tolerance	1.0000e+00	1
2	4	2.4370e-02	1.1326e+00	5.0621e-01	2	2.2935e-13	Tolerance	2.0847e-01	2
3	5	1.9835e-02	6.3128e+00	2.9138e-01	2	2.1941e-13	Tolerance	1.0000e+00	1
4	6	1.1246e-03	4.8783e-02	5.9031e-03	1	1.6562e-02	Tolerance	1.0000e+00	1
5	7	1.3133e-04	5.1162e-01	7.7761e-02	2	1.9677e-14	Tolerance	1.0000e+00	1
6	8	5.9533e-07	7.4425e-04	5.1087e-04	1	7.8312e-04	Tolerance	1.0000e+00	1
7	9	3.5500e-11	2.6421e-04	1.7243e-03	2	1.2905e-16	Tolerance	1.0000e+00	1
8	10	4.6028e-21	6.5470e-11	1.8273e-06	2	3.0946e-17	Tolerance	1.0000e+00	1

Figure 5.3. Progress report for line search Newton CG solution methods.

Iteration	F(x)	norm(C)	norm(G)	TR-Radius	norm(N)	norm(T)	TR-Itr
0	2.0930e-02	1.0237e+00	2.6471e-03	1.0000e+04	0.0000e+00	0.0000e+00	0
1	5.2700e-02	3.1705e-02	3.6245e-03	1.0000e+02	1.5447e-01	4.6758e-02	1
2	5.3920e-02	6.0869e-04	6.2387e-05	1.0000e+02	4.5461e-03	2.3407e-02	1
3	5.3950e-02	1.7112e-07	2.6161e-08	1.0000e+02	9.5475e-05	3.8666e-04	1
4	5.3950e-02	3.0406e-14	3.5849e-15	1.0000e+02	2.7751e-08	1.5887e-07	1

Figure 5.4. Progress report for trust-region inexact sequential quadratic programming solution method.

5. `ZeroCurvature:` zero curvature was observed, i.e. indefinite/negative definite Hessian;

6. `NaNCurvature:` NaN (not a number) curvature was observed, i.e. indefinite/negative definite Hessian.

C++ API

Set functions used to enable the diagnostics progress reports. The set functions used to enable the diagnostic progress reports are shown in Table 5.1. The **setOutputDataDisplayOption** function sets the diagnostic progress report output frequency. The options are:

1. `dotk::types::ITERATION` both the progress report and optimal solution are outputed at each iteration;

2. `dotk::types::FINAL` the progress report and optimal solution are respectively outputed at each iteration and at the end of the optimization problem;

3. `dotk::types::OFF` both the progress report and optimal solution outputs are inactive.

Users can use the set functions in Table 5.1 as follows:

`instance.getIOPtr()->setFunction(argument)`,

where `instance` specifies a DOTk solution method object, `getIOPtr()` returns a C++ shared pointer that grants access to all the public functions in class `DOTk_IOTools`, and both `setFunc- tion` and `argument` are any of the options shown in Table 5.1.

Table 5.1. C++ set function used to enable the diagnostics progress reports.

Function	Argument	Default
setOutputDataDisplayOption	*dotk::types::display_t*	dotk::types::OFF

MATLAB API

Users have access to additional output information when DOTk solution methods are used through the MATALB API. Relevant output data is stored in a MATALB structure array and provided to users. This set up easily enables access to MATLAB post-processing tools. If only control variables are optimized, the objective function, control, gradient, and trial step values at the final optimization iteration are outputed and repectively stored in variables `ObjectiveFunction`, `Control`, `Gradient`, and `TrialStep`. If both state and control variables are optimized, the objective function, state, control, lagrange multipliers, gradient, and trial step values at the final optimization iteration are outputed and respectively stored in variables `ObjectiveFunction`, `State`, `Control`, `LagrangeMultipliers`, `Gradient`, and `TrialStep`.

Parameter to enable the diagnostic progress reports in MATLAB. The parameters to enable the diagnostic progress reports in MATLAB is shown in Table 5.2. The **Display-Option** keyword specifies how frequent the diagnostic progress reports are outputed. The options are `ITERATION`, `FINAL`, `OFF`. Above is the description of each option.

Table 5.2. Parameter to enable the diagnostics progress reports in MATLAB.

Function	Argument	Default
OutputDataDisplayOption	*String*	OFF

Finite Differencing Check

One unique feature of DOTk software package is the finite differencing check tool. This feature is extremely valuable when users can provide derivative information. This tool allows users to corroborate/diagnose the proper derivation and implementation of the first-order and

second-order derivative information used to solve the optimization problem. Instead of verifying the full gradient and Hessian operators, users can verify each of the pieces/components required to compute both the gradient operator and the application of a vector to the Hessian operators. Thus, the finite differencing check tool guarantees users that the derivative information used to solve the optimization problem was properly derived and implemented.

Table 5.3. C++ functions used to enable the finite differencing check tool.

Function	Argument
checkFirstDerivativeUnconstrainedObjectiveFunction	*Vector*
checkSecondDerivativeUnconstrainedObjectiveFunction	*Vector*
checkFirstDerivativeConstrainedObjectiveFunction	*MultiVector* *dotk::types::derivative_t*
checkSecondDerivativeConstrainedObjectiveFunction	*MultiVector* *dotk::types::derivative_t*
checkFirstDerivativeEqualityConstraint	*MultiVector* *dotk::types::derivative_t*
checkAdjointFirstDerivativeEqualityConstraint	*MultiVector* *dotk::types::derivative_t*
checkAdjointSecondDerivativeEqualityConstraint	*MultiVector* *dotk::types::derivative_t*

C++ API

Finite differencing check functions.[1] The finite differencing check functions are shown in Table 5.3. The **checkFirstDerivativeUnconstrainedObjectiveFunction** function checks the first-order derivative of the objective function used for an unconstrained optimization problem. This function implicitly checks the implementation of the objective function. The **checkSecondDerivativeUnconstrainedObjectiveFunction** function checks the second-order derivative of the objective function used for an unconstrained optimization problem. The **checkFirstDerivativeConstrainedObjectiveFunction** checks the first-order derivative of the objective function used for a constrained optimization problem. This function implicitly checks the implementation of the objective function. The **checkSecondDerivativeConstrainedObjectiveFunction** checks the the second-order derivative of the objective function used for a constrained optimization problem. The **checkFirstDerivativeEqualityConstraint** function checks the first-order derivative of the equality constraint. This function implicitly checks the implementation of the equality constraint. The **checkAdjointFirstDerivativeEqualityConstraint** function checks the adjoint of the

[1] A new C++ API is under development. The current API will be deprecated as soon as the new API is available to DOTk users.

first-order derivative of the equality constraint. The **checkAdjointSecondDerivativeEqualityConstraint** function checks the adjoint of the second-order derivative of the equality constraint.

```
#include "SOL_State.H"
#include "SOL_Diagnostics.hpp"
#include "SerialLinearAlgebra.H"
#include "RosenbrockOperators.H"
int main()
{
    const Int num_controls = 2;
    std::tr1::shared_ptr<RosenbrockOperators> op(new RosenbrockOperators());
    std::tr1::shared_ptr<SerialLinearAlgebra> la(new SerialLinearAlgebra());
    sol::params::space space dim(num_controls);
    sol::SOL_Diagnostics<Real> tools(op, la, space_dim);

    std::vector<Real> z(num controls, 2.);
    tools.checkSecondDerivativeUnconstrainedObjectiveFunction(z);

    return (0);
}
```

Figure 5.5. Finite differencing check example in C++.

The *Vector* and *MultiVector* keywords in Table 5.3 represent a `std::vector< TYPE >` and `std::vector< std::vector<TYPE> >`, repsectivel. The `TYPE` keyword specifies a floating-point type. The *dotk::types::derivative_t* keyword specifies the first-order or second-order derivative type. The derivative type options are:

1. `dotk::types::Z` specifies the first-order derivative of a function with respect to the control variables

2. `dotk::types::U` specifies the first-order derivative of a function with respect to the state variables

3. `dotk::types::UU` specifies the second-order derivative of the first-order derivative of a function, where the first-order derivative was taken with respect to the state variables, with respect to the state variables

4. `dotk::types::ZZ` specifies the second-order derivative of the first-order derivative of a function, where the first-order derivative was taken with respect to the control variables, with respect to the control variables

5. `dotk::types::UZ` specifies the second-order derivative of the first-order derivative of a function, where the first-order derivative was taken with respect to the state variables, with respect to the control variables

6. `dotk::types::ZU` specifies the second-order derivative of the first-order derivative of a function, where the first-order derivative was taken with respect to the control variables, with respect to the state variables

```
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 1.0346e-05, Epsilon = 1.0000e+03
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 7.2955e-09, Epsilon = 1.0000e+02
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 1.2733e-11, Epsilon = 1.0000e+01
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 9.0949e-13, Epsilon = 1.0000e+00
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 1.1369e-12, Epsilon = 1.0000e-01
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 2.8649e-11, Epsilon = 1.0000e-02
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 1.7030e-10, Epsilon = 1.0000e-03
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 9.2791e-10, Epsilon = 1.0000e-04
Expected Value = 1.1882e+03, Finite Difference Appx. = 1.1882e+03, Relative Difference = 2.3088e-08, Epsilon = 1.0000e-05
```

Figure 5.6. Finite differencing check progress report.

Users are expected to follow a sequence of steps to implement enable the finite differencing check tools through a C++ API. These steps are described as follows:

1. Define and declare the linear algebra API, see Figures 2.1 and 2.2
2. Define and declare the operators API, see Figures 2.6 and 2.7
3. Include the necessary header files in the main routine: `DOTk_State`, `DOTk_Diagnostics`, `UserDefinedLinearAlgebraAPI`, `UserdDefinedOperatorsAPI`
4. Initialize C++ shared pointers to store both the linear algebra and operators APIs dynamically allocated objects
5. Initialize a `sol::params::space` data structure to define the number of design variables, see Chapter 3
6. Initialize the `DOTk_Diagnostics` object to enable access to all the finite differencing check tools
7. Initialize `std::vector` of control or state variables[2]
8. Call finite differencing check function `functionName` to corroborate the derivation and implementation of the operators

The `UserDefinedLinearAlgebraAPI` and `UserDefinedOperatorsAPI` keywords respectively specify the linear algebra and operators child classes defined by the user, see Chapter 2. The `functionName` keyword specifies any finite differencing check tool shown in Table 5.3. A finite differencing check C++ example is shown in Figure 5.5.

Finite differencing progress report keywords. The finite differencing check progress report is shown Figure 5.6. The **Expected Value** keyword specifies the evaluation of the

[2]The optimization problem class determines if both control and state variables vector are required.

Table 5.4. MATLAB functions to enable the finite differencing check tools.

Function	Argument
mxCheckFirstDerivativeUnconstrainedObjectiveFunction	*Array*
	Structure Array
mxCheckSecondDerivativeUnconstrainedObjectiveFunction	*Array*
	Structure Array
mxCheckFirstDerivativeConstrainedObjectiveFunction	*Structure Array*
	Structure Array
	Character Array
mxCheckSecondDerivativeConstrainedObjectiveFunction	*Structure Array*
	Structure Array
	Character Array
mxCheckFirstDerivativeEqualityConstraint	*Structure Array*
	Structure Array
	Character Array
mxCheckAdjointFirstDerivativeEqualityConstraint	*Structure Array*
	Structure Array
	Character Array
mxCheckAdjointSecondDerivativeEqualityConstraint	*Structure Array*
	Structure Array
	Character Array

user-defined function. The **Finite Difference Appx.** keyword specifies the evaluation of the finite difference approximation to the function. The **Relative Difference** keyword specifies the difference between the evaluation of the user-defined function and the finite difference approximation. The **Epsilon** keyword specifies a positive finite perturbation.

MATLAB API

Functions to enable the finite differencing check tools. The functions to enable the finite differencing check tools are shown in Table 5.4. The **mxCheckFirstDerivativeUnconstrainedObjectiveFunction** function checks the first-order derivative of the objective function used in unconstrained optimization problems. The **mxCheckSecondDerivativeUnconstrainedObjectiveFunction** function checks the second-order derivative of the objective function used in unconstrained optimization problems. The **mxCheckFirstDerivativeConstrainedObjectiveFunction** checks the first-order derivative of the objective function used in constrained optimization problems. The **mxCheckSecondDerivativeConstrainedObjectiveFunction** checks the second-order derivative of the objective function used in constrained optimization problems. The **mxCheckFirstDerivativeEqualityConstraint** function checks the first-order derivative of the equality constraint. The

```
>> addpath /scratch/maguilo/research/code/soul/matlab/exe/
>> z = ones(1,2)*2;
>> [Options] = getSolDiagnosticsInterface('ObjectiveFunction');
>> mxCheckFirstDerivativeUnconstrainedObjectiveFunction(z,Options)
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 2.2605e-06, Epsilon = 1.0000e+03
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 1.3355e-09, Epsilon = 1.0000e+02
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 1.4211e-12, Epsilon = 1.0000e+01
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 5.6843e-14, Epsilon = 1.0000e+00
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 1.8758e-12, Epsilon = 1.0000e-01
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 8.0718e-12, Epsilon = 1.0000e-02
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 2.2220e-10, Epsilon = 1.0000e-03
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 1.4171e-10, Epsilon = 1.0000e-04
Expected Value = 3.2836e+02, Finite Difference Appx. = 3.2836e+02, Relative Difference = 6.9630e-09, Epsilon = 1.0000e-05
```

Figure 5.7. Finite differencing check example in Matlab: Uncosntrained optimization problems.

checkAdjointFirstDerivativeEqualityConstraint function checks the the adjoint of the first-order derivative of the equality constraint. The **mxCheckAdjointSecondDerivativeEqualityConstraint** function checks the adjoint of the second-order derivative of the equality constraint.

Unconstrained optimization problems. Users are expected to complete the following steps to enable the finite differencing check tools in MATLAB[3]:

1. Set path to DOTk-MEX directory, e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab/exe`
2. Define the linear algebra API in file `getLinearAlgebra.m`, see Figure 2.3
3. Define the objective function operators API in file `getObjectiveFunctionOperators.m`, see Figure 2.8
4. Define a MATLAB array `z` that sets the vector of control variables, e.g.

   ```
   z = ones(1,dim(z))*2
   ```

5. Call function `getSolDiagnosticsInterface(operator_type)` to initialize the Options structure array, e.g.

   ```
   [Options] = getSolDiagnosticsInterface('ObjectiveFunction')
   ```

6. Call DOTk-MEX finite differencing check function, e.g.

   ```
   mxCheckFirstDerivativeUnconstrainedObjectiveFunction(z,Options)
   ```

The `operator_type` keyword specifies the class of operator that will be corrobarated with the finite differencing check tool. The options are `ObjectiveFunction` or `EqualityConstraint`.

```
>> addpath /scratch/maguilo/research/code/sol/matlab/exe/
>> Input.Control = [];
>> Input.States = ones(1,5)*2;
>> Input.NumEqConstraints = 3;
>> [Options] = getSolDiagnosticsInterface('EqualityConstraint');
>> mxCheckFirstDerivativeEqualityConstraint(Input,Options,'U')
 SOL::Derivative_Type = U
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 1.1211e-12, Epsilon = 1.0000e+03
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 2.3704e-14, Epsilon = 1.0000e+02
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 7.3081e-16, Epsilon = 1.0000e+01
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 2.0904e-16, Epsilon = 1.0000e+00
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 1.6911e-15, Epsilon = 1.0000e-01
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 2.0175e-14, Epsilon = 1.0000e-02
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 1.7080e-12, Epsilon = 1.0000e-03
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 1.4221e-09, Epsilon = 1.0000e-04
Expected Value = 2.5493e+01, Finite Difference Appx. = 2.5493e+01, Relative Difference = 1.1851e-05, Epsilon = 1.0000e-05
```

Figure 5.8. Finite differencing check example in MATLAB for unconstrained optimization problems.

Finally, `dim(·)` specifies the argument dimension. A finite differencing check example in MATLAB is shown in Figure 5.7.

Constrained optimization problems. The following steps are expected to be completed by users to enable DOTk's finite differencing check tools in MATLAB:

1. Set path to DOTk-MEX directory, e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab/exe`
2. Define the linear algebra API in file `getLinearAlgebra.m`, see Figure 2.3
3. Define the objective function operators API in file `getObjectiveFunctionOperators.m`, see Figure 2.8
4. Define the equality constrained operators API in file `getEqualityConstrainedOperators.m`, see Figure 2.10
5. Define a MATLAB structure array to set the vector of control variables, vector of state variables, and the number of equality constraints[4], for example:

   ```
   Input.Control = ones(1,dim(control))*2
   Input.States = ones(1,dim(state))*2
   Input.NumEqConstraints = 3
   ```

6. Call function `getSolDiagnosticsInterface(operator_type)` to initialize the options structure array[5], for example

[3] All the MATLAB files, i.e. m-Files, should be located inside the same working directory. If the m-Files are scattered in different directories, users are expected to specify the path to these directories in addition to the path of the DOTk-MEX directory.

[4] Users are expected to initialize the `Input` structure array fields as shown herein.

[5] Users should initialize both objective function and equality constraint options structure arrays for constrained optimization problems.

```
[Options] = getSolDiagnosticsInterface('EqualityConstraint')
```

7. Call DOTk-MEX finite differencing check function `functionName(Input, Options, derivative_type)` to check the `operator_type` implementation, for example

```
mxCheckFirstDerivativeEqualityConstraint(Input, Options, 'U')
```

The `functionName` keyword specifies any finite differencing check function shown in Table 5.4 for constrained optimization problems. The `derivative_type` keyword specifies the first-order or second-order derivative of a function. The `derivative_type` keyword is a character-type argument. The options are `U`, `Z`, `UU`, `UZ`, `ZZ`, and `ZU`. Users are encouraged to review the derivative type descriptions discussed in the **C++ API subsection**. A finite differencing check example in MATLAB is shown in Figure 5.8.

Chapter 6

Tutorial

Simple Unconstrained Optimization Problem

Lets consider the following parameter estimation problem:

$$\underset{(\mathbf{z})\in\mathbb{R}^2}{\text{minimize}} \quad f(\mathbf{z}), \tag{6.1}$$

where

$$f(\mathbf{z}) = 100(z_2 - z_1^2)^2 + (1 - z_1)^2. \tag{6.2}$$

This function is widely known in the literature as the Rosenbrock function [12]. Figure 6.1 shows a two and three dimensional plot of the Rosenbrock function Note that this is an unconstrained optimization problem. The unique solution for this problem lies at $\mathbf{z} = (z_1, z_2) = (1, 1)$, where $f(\mathbf{z}) = 0$. The initial guess for this problem was set to $z_1 = z_2 = 1.2$. The progress report for this problem is shown in Figure 6.2. Users can go over the subsequent subsections to understand how to set up and solve the unconstrained optimization problem in Equation 6.1 in both the C++ and MATLAB APIs.

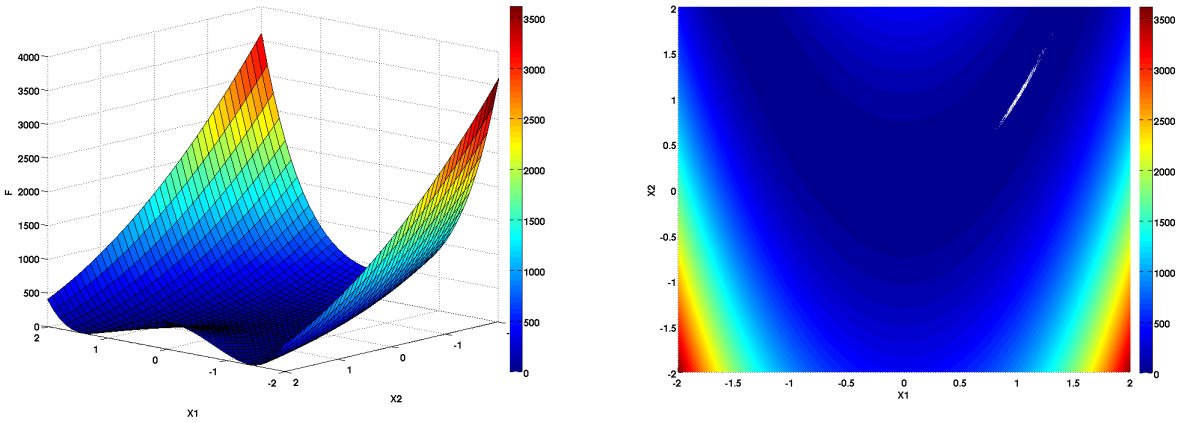

Figure 6.1. Rosenbrock function. Left pane: 3D plot of $F(\mathbf{z})$. Right pane: 2D plot of $F(\mathbf{z})$.

Iteration	Func-count	F(x)	norm(G)	norm(P)	TR-Itr	TR-Radius	ActualReduc	PredReduc	Ratio	Krylov-Itr	Krylov-Error	Krylov-Exit
0	1	5.8000e+00	1.2517e+02	*	*	*	*	*	*	*	*	*
1	2	3.8384e-02	3.9982e-01	2.3024e-01	1	1.0000e+02	5.7616e+00	5.7608e+00	1.0001e+00	2	1.8371e-12	Tolerance
2	3	3.8384e-02	3.9982e-01	5.0621e-01	1	5.0000e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	2	2.5000e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	3	1.2500e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	4	6.2500e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	5	3.1250e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	6	1.5625e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	7	7.8125e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	8	3.9062e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	9	1.9531e-01	-1.4458e-02	3.6265e-02	-3.9867e-01	1	1.6297e-01	TrustRegion
*	*	*	*	*	10	3.9062e-01	2.1182e-02	2.3845e-02	8.8831e-01	1	1.6297e-01	TrustRegion
3	13	4.9812e-03	1.7455e+00	1.4004e-01	1	3.9062e-01	1.2221e-02	9.8105e-03	1.2457e+00	2	6.0996e-14	Tolerance
4	14	8.0668e-04	6.4345e-01	8.1801e-02	1	3.9062e-01	4.1745e-03	3.4340e-03	1.2156e+00	2	1.5092e-13	Tolerance
5	15	4.3272e-05	1.8848e-01	4.4377e-02	1	3.9062e-01	7.6341e-04	6.7483e-04	1.1313e+00	2	8.5783e-14	Tolerance
6	16	2.0445e-07	1.1201e-02	1.0525e-02	1	3.9062e-01	4.3068e-05	4.1250e-05	1.0441e+00	2	8.8517e-14	Tolerance
7	17	5.4503e-12	6.9400e-05	8.3932e-04	1	3.9062e-01	2.0444e-07	2.0375e-07	1.0034e+00	2	1.1206e-15	Tolerance
8	18	3.9193e-21	1.5553e-09	3.9070e-06	1	3.9062e-01	5.4503e-12	5.4502e-12	1.0000e+00	2	6.1624e-18	Tolerance

Figure 6.2. Progress report: Unconstrained optimization example problem.

C++ API

Users are expected to complete a sequence of steps to properly implement the C++ API required to solve the unconstrained optimization problem defined in Equation 6.1. These steps are described as follows:

1. Define and declare the linear algebra API. The C++ linear algebra class definition is shown in Figure 6.3. Figures show the C++ function declaration for each of the linear algebra functions required to perform all the internal DOTk linear algebra operations. The linear algebra operations in this example problem are done in serial. However, users can utilize DOTk default parallel linear algebra class if necessary. Furthermore, recall that users can provide access to their preferred linear algebra library or implement their own through the linear algebra API. Figures 6.4, 6.5, 6.6, and 6.7 show the respective function declaration for the linear algebra operations `scal`, `axpy`, `innr`, and `normF`. These linear algebra functions are needed by the DOTk to perform all the linear algebra operations inside the source code.

2. Define and declare the operators API. The C++ class definition for this example problem is shown in Figure 6.8. Figures 6.9, 6.10, and 6.11 display the C++ function declaration for each of the operators required to solve the unconstrained optimization problem defined in Equation 6.1 with the trust region Newton conjugate gradient algorithm.

3. Declare main routine. This requires including the necessary header files as seen from Figure 6.12.

   (a) Set the number of design variables, i.e. number of controls

   (b) Initialize C++ shared pointers to store the following dynamically allocated objects: linear algebra API, operators API, and the `DOTk_VectorSpaceReduced` instance that stores relevant problem data such as the solution to the problem.

   (c) Initialize DOTk solution method instance, e.g. `dotk::DOTk_TrustRegionNewtonCG`

```
#ifndef SERIALLINEARALGEBRA_H_
#define SERIALLINEARALGEBRA_H_

#include "DOTk_LinearAlgebra.H"

class SerialLinearAlgebra : public DOTk_LinearAlgebra
{
public:
    SerialLinearAlgebra(){}
    virtual ~SerialLinearAlgebra(){}

    // Scales a vector by a real constant.
    virtual void scal(const Real alpha, Vector & x);
    // Constant times a vector plus a vector.
    virtual void axpy(const Real alpha, const Vector & x, Vector & y);
    // Returns the inner product of two vectors.
    virtual Real innr(const Vector & x, const Vector & y);
    // Returns Frobenius norm of a matrix
    virtual Real normF(const MultiVector & A);

private:
    SerialLinearAlgebra(const SerialLinearAlgebra&); // unimplemented
    SerialLinearAlgebra& operator=(const SerialLinearAlgebra& rhs)
    {
        // check for self-assignment by comparing the address of the
        // implicit object and the parameter
        if (this == &rhs)
        {
            return (*this);
        }
        // return the existing object
        return (*this);
    }
};

#endif /* SERIALLINEARALGEBRA_H_ */
```

Figure 6.3. Linear algebra class definition in C++.

(d) Set initial guess, e.g. `setInitialControl(control)`

(e) Set stopping criteria. Here, defualt values are used to solve the problem[1]

(f) Set method used to compute both first-order and second-order derivative information, i.e. gradient and Hessian information

(g) Enable progress report[2]

(h) Call function `getMin` to solve optimization problem[3]

4. If necessary, perform the sequence of steps describe in Chapter 1 to generate DOTk executables

[1]Default values can be used to solve the unconstrained optimization problem defined in Equation 6.1.

[2]Progress report is disabled by default.

[3]The optimal solution will be available to the user through the `DOTk_VectorSpaceReduced` instance, e.g. `vector_space->getConstControl(dotk::types::NEW)[i]`, where `[i]` denotes the i-th element.

```
void SerialLinearAlgebra::scal(const Real alpha, Vector& x)
{
    for(unsigned int i = 0; i < x.size(); ++i)
    {
        x[i] = alpha * x[i];
    }
}
```

Figure 6.4. Vector scale function declaration in C++.

```
void SerialLinearAlgebra::axpy(const Real alpha, const Vector& x, Vector& y)
{
    for(unsigned int i = 0; i < x.size(); ++i)
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```

Figure 6.5. Update vector function declaration in C++.

This sequence of steps is explicitly shown in Figure 6.12. Analogously to the parameters shown in Figure 6.12, users can define additional parameters pertinent to the solution method in used. For instance, users can set the trust region radius contraction parameter as follows `trustRegionNewtonCG.getTrustRegionStepPtr()->setContractionParameter(0.5)`. These parameters and their respective default values are discussed in Chapter 3.

MATLAB API

The MATLAB structure array with the field names and values that correspond to DOTk options are defined in file `getOptimizationOptions.m`. Users are expected to set these options inside `getOptimizationOptions.m`. Users are then required to complete a serie of steps after setting the necessary options for the solution method of choice. These steps will allow users to querry DOTK solution methods directly from MATLAB's command window[4]. These steps are described as follows:

1. Set path to DOTk-MEX install directory, e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab/exe`, in file `driver.m`. Figure 6.13 shows `driver.m` source code [5]

[4]Users may need to compile DOTk MEX executables, see Chapter 1.

[5]The source code, i.e. m-files, for MATLAB routines `driver`, `getOptimizationOptions`, `runDiagnostics`, and `getMin` will be provided to users. Thus, users are not expected to implement these routines. However, users will have the freedom to modify these routines and adapt them to their problem of

```cpp
Real SerialLinearAlgebra::innr(const Vector& x, const Vector& y)
{
    Real innr_x_y = 0.;
    for(unsigned int i = 0; i < x.size(); ++i)
    {
        innr_x_y += x[i] * y[i];
    }
    return (innr_x_y);
}
```

Figure 6.6. Vector inner product function declaration in C++.

```cpp
Real SerialLinearAlgebra::normF(const MultiVector& A)
{
    Real norm_A = 0.;
    for(unsigned int i = 0; i < A.size(); ++i)
    {
        for(unsigned int j = 0; j < A.at(0).size(); ++j)
        {
            norm_A += A[i][j] * A[i][j];
        }
    }
    norm_A = std::sqrt(norm_A);
    return (norm_A);
}
```

Figure 6.7. Frobenius norm function declaration in C++.

2. Define the linear algebra API in m-file `getLinearAlgebra.m`. Figure 6.14 shows the default linear algebra API. The source code, i.e. m-file, for `getLinearAlgebra.m` will be provided to users

3. Define objective function operators API in file `getObjectiveFunctionOperators.m`. Figure 6.15 shows the objective function operators needed to solve the unconstrained optimization problem defined in Equation 6.1

4. Solve unconstrained optimization problem

   (a) Select solution method, e.g. `algorithm_t='TrustRegionNewtonCG'`

   (b) Select problem type/class, e.g. `problem_t='Unconstrained'`

interest if necessary.

```
#ifndef ROSENBROCKOPERATORS_H_
#define ROSENBROCKOPERATORS_H_

#include "DOTk_Operators.H"

class RosenbrockOperators : public DOTk_Operators
{
public:
    RosenbrockOperators();
    virtual ~RosenbrockOperators(){}

    /* ================================ Main Routines ================================ */

    virtual Real Fval(const Vector &z);
    virtual void Fval(const MultiVector &z, Vector &fval);
    virtual void F_z(const Vector &z, Vector &g);
    virtual void F_zz(const Vector& z, const Vector& dz, Vector& H_dz);
private:
    // unimplemented
    RosenbrockOperators(const RosenbrockOperators&);
    RosenbrockOperators operator=(const RosenbrockOperators&);

};

#endif /* ROSENBROCKOPERATORS_H_ */
```

Figure 6.8. C++ class definition.

(c) To enable the finite difference diagnostics tools set derivative check flag to true (`der_flag=true`). However, since the goal is to solve the unconstrained optimization problem defined in Equation 6.1, the derivative check flag is set to false (`der_flag=false`)

(d) Define MATLAB structure array that stores field names and values used to set the number of design variables as follows:[6]

```
Inputs.NumControls = 2
Inputs.NumStates = 0
Inputs.NumEqConstraints = 0
```

[6]Users are expected to define the `Inputs` structure array fields as shown in Step 4.

```
Real RosenbrockOperators::Fval(const Vector& z)
{
    Real fval;
    fval = 100 * pow((z[1] - z[0] * z[0]), 2.)+pow(1 - z[0], 2.);
    return (fval);
}
```

Figure 6.9. C++ function declaration for the objective function declaration.

```
void RosenbrockOperators::F_z(const Vector &z, Vector &g)
{
    g[0] = -400. * (z[1] - pow(z[0], 2.)) * z[0] + 2. * z[0] - 2.;
    g[1] = 200. * (z[1] - pow(z[0], 2.));
}
```

Figure 6.10. C++ function declaration for the first-order derivative of the objective function declaration.

```
void RosenbrockOperators::F_zz(const Vector& z, const Vector& dz, Vector& H_dz)
{
    H_dz[0] = ((2 - 400 * (z[1] - pow(z[0], 2.)) + 800 * pow(z[0], 2.)) * dz[0]) - (400 * z[0] * dz[1]);
    H_dz[1] = (-400. * z[0] * dz[0]) + (200. * dz[1]);
}
```

Figure 6.11. C++ function declaration for the second-order derivative of the objective function declaration.

`Inputs.NumIeqConstraints = 0`

(e) Call function `getOptimizationOptions(algorithm_t,problem_t,Inputs)` to initialize the Options and Operators structure arrays, e.g.

`[Options,Operators] = ...`
`getOptimizationOptions(algorithm_t,problem_t,Inputs).`

Figure 6.16 shows some parts of the source code in `getObjectiveFunctionOperators.m`[7]

(f) Call function `getMin(algorithm_t,Options,Operators)` to solve optimization problem, e.g.

`[Output,TimeData] = getMin(algorithm_t,Options,Operators)`

(g) The Output MATLAB structure array stores relevant problem information. For instance, the solution to the optimization problem is stored in `Output.Control`. The Output MATLAB structure array stores the value of the objective function (`Output.ObjectiveFunction`), optimal solution (`Output.Control`), norm of the gradient (`Output.Gradient`), and norm of the trial step (`Output.TrialStep`) for optimization problems solved with a reduced-space formulation strategy

The implementation of these steps is explicitly shown in Figure 6.13. The `algorithm_t` keyword specifies the solution method used to solve the optimization problem of interest. The options are `NonLinearCG`, `QuasiNewton`, `TrustRegionQuasiNewtonCG`, `LineSearchNewtonCG`, `TrustRegionNewtonCG`, and `InexactSQP`. The `problem_t` keyword specifies the type/class of optimization problem solved based on the type of constraint. The options are `Unconstrained`, `EqualityConstrained`, and `Constrained`.

[7] File `getObjectiveFunctionOperators.m` will be provided to users.

```cpp
#include "SerialLinearAlgebra.H"
#include "RosenbrockOperators.H"

#include "DOTk_VectorSpaceReduced.H"
#include "DOTk_Step.H"
#include "DOTk_TrustRegionStep.H"
#include "DOTk_TrustRegionNewtonCG.H"

int main()
{
    const Int num_controls = 2;
    std::tr1::shared_ptr<SerialLinearAlgebra> linear_algebra(new SerialLinearAlgebra());
    std::tr1::shared_ptr<RosenbrockOperators> operators(new RosenbrockOperators());
    std::tr1::shared_ptr<dotk::DOTk_VectorSpaceReduced>
    vector_space(new dotk::DOTk_VectorSpaceReduced(linear_algebra, num_controls));
    dotk::DOTk_TrustRegionNewtonCG trustRegionNewtonCG(vector_space, operators);

    Vector control(num_controls, 2.);
    vector_space->setInitialControl(control);

    trustRegionNewtonCG.setGradientPtr(dotk::types::USER_DEFINED_GRAD);
    trustRegionNewtonCG.setHessianPtr(dotk::types::USER_DEFINED_HESS);
    trustRegionNewtonCG.setInvHessianPtr(dotk::types::IDENTITY_INV_HESS);
    trustRegionNewtonCG.setTrustRegionStepPtr(dotk::types::TRUST_REGION_DOGLEG);
    trustRegionNewtonCG.enableDiagnosticsOutput(dotk::types::FINAL, true);
    trustRegionNewtonCG.getMin();

    return (0);
}
```

Figure 6.12. Declaration of main routine in C++.

Simple Bound-Constrained Optimization Problem

Lets consider the following parameter estimation problem:

$$\begin{aligned} &\underset{\mathbf{z}\in\mathbb{R}^2}{\text{minimize}} \quad f(\mathbf{z}), \\ &\text{s.t.} \\ &\quad L_1 \leq z_1 \leq U_1 \\ &\quad L_2 \leq z_2 \leq U_2 \end{aligned} \tag{6.3}$$

where $F(\mathbf{z})$ is given by Equation 6.2, L_1 and U_1 respectively sepcify the lower and upper bounds of z_1, and L_1 and U_1 respectively sepcify the lower and upper bounds of z_2. The lower and upper bounds of $z_1 = z_2 = [0, 5]$ in this example problem. The unique solution for this problem lies at $\mathbf{z} = (z_1, z_2) = (1, 1)$, where $f(\mathbf{z}) = 0$. The initial guess for this problem was set to $z_1 = z_2 = 1.2$. The progress report for this problem is shown in Figure 6.17.

The C++ and MATLAB main driver implementations are shown in Figures 6.13 and 6.18. The sequence of steps required to set up the unconstrained optimization problem de-

```
% Driver for Rosenbrock example problem
clc;
clear all;
% Path to executables directory
addpath /scratch/maguilo/dotk/matlab/exe/;
% Introduce the problem
fprintf('\n*** Unconstrained Optimization: Rosenbrock Example Problem ***\n');
% Gradient based optimization algorithm.
% Options: NonLinearCG, QuasiNewton, TrustRegionQuasiNewton,
%          TrustRegionNewtonCG, LineSearchNewtonCG, InexactSQP
algorithm_t = 'TrustRegionNewtonCG';
% Options: Unconstrained, EqualityConstrained, Constrained
problem_t = 'Unconstrained';
% derivative check flag
der_flag = false;
% Set Parameter Space Dimension
Inputs.NumControls = 2;
Inputs.NumStates = 0;
Inputs.NumEqConstraints = 0;
Inputs.NumIeqConstraints = 0;
% Create the DOTk main function interface
[Options, Operators] = getOptimizationOptions(algorithm_t, problem_t, Inputs);
% Run derivative check
if(der_flag == true)
    DiagnosticsToolsInputData.States = randn(Inputs.NumStates,1);
    DiagnosticsToolsInputData.Control = randn(length(Inputs.NumControls,1));
    DiagnosticsToolsInputData.NumEqConstraints = Inputs.NumEqConstraints;
    runDiagnostics(problem_t,DiagnosticsToolsInputData)
    return;
end
fprintf(1,' \n\n');
fprintf(1, ...
    ' Solve Unconstrained Optimization Probelm\n');
% Solve Optimization Problem
[Output,TimeData] = getMin(algorithm_t,Options,Operators);
```

Figure 6.13. Default driver.m source code.

```
function [LinearAlgebra] = getLinearAlgebra()
LinearAlgebra.scal=@(alpha,x) alpha*x;
LinearAlgebra.axpy=@(alpha,x,y) alpha*x+y;
LinearAlgebra.innr=@(x,y) x'*y;
LinearAlgebra.normF=@(x) norm(x,'fro');
end
```

Figure 6.14. Default getLinearAlgebra.m source code.

```
function [ObjectiveFunctionOperators] = getObjectiveFunctionOperators()
ObjectiveFunctionOperators.Fval=@(x) Fval(x);
ObjectiveFunctionOperators.F_u=@(x) F_u(x);
ObjectiveFunctionOperators.F_z=@(x) F_z(x);
ObjectiveFunctionOperators.F_uu=@(x,du) F_uu(x,du);
ObjectiveFunctionOperators.F_uz=@(x,dz) F_uz(x,dz);
ObjectiveFunctionOperators.F_zz=@(x,dz) F_zz(x,dz);
ObjectiveFunctionOperators.F_zu=@(x,du) F_zu(x,du);
end
function [fval] = Fval(x)
fval = 100 * power((x(2) - x(1) * x(1)), 2.)+power(1 - x(1), 2.);
end
function [f_z] = F_z(x,f_z)
    f_z(1) = -400. * (x(2) - power(x(1), 2.)) * x(1) + 2. * x(1) - 2.;
    f_z(2) = 200. * (x(2) - power(x(1), 2.));
end
function  [fzz_x_dz] = F_zz(x, dz)
    fzz_x_dz(1) = ((2 - 400 * (x(2) - power(x(1), 2.)) + 800 * ...
        power(x(1), 2.)) * dz(1)) - (400 * x(1) * dz(2));
    fzz_x_dz(2) = (-400. * x(1) * dz(1)) + (200. * dz(2));
end
```

Figure 6.15. Objective function operators source code.

fined in Equation 6.2 also apply to the bound-constrained optimization problems defined in Eqaution 6.3. The only subtle difference lies in the need to define the lower and upper bound vectors.[8]. For instance, if the C++ main driver is employed, users are required to initialize two standard vectors to set the lower and upper bounds. Users are then required to initialize DOTk lower and upper bounds data structures by calling DOTk function `setControlBounds(lower_bound, upper_bound)`, e.g. `vector_space->setControlBounds (lower_bound, upper_bound)`, see Figure 6.18.

MATLAB users are expected to first set the `Options.BoundConstraints` field to 1 in file `getOptimizationOptions.m` to activate the bound-constraint option. Second, users are expected to define the lower and upper bound vectors in function `getConstraintOptions`. Third, users are expected to select the bound constraint modeling method by setting keyword `ConstraintMethod`. The bound constraint modeling options include `BOUND_FEASIBLE_DIRECTION`, `BOUND_FEASIBLE_DIRECTION_MIN_REDUCTION`, `BOUND_PROJECT_SCALED_DIRECTION`, `BOUND_PROJECTION_ALONG_FEASIBLE_DIRECTION`. Constraint are inactive by default; hence, the `ConstraintMethod` keyword is set to `OFF`. Figure 6.19 shows the source code for function `getConstraintOptions`, which is defined and declared in file `getOptimizationOptions.m`.

[8]Users have the freedom to specify lower and upper bounds for each design variable

```
function [Options, Operators] = ...
    getOptimizationOptions(algorithm_t, problem_t, Inputs)
Options = [];
% Diagnostics Display Option
[Options] = setDiagnosticsDisplayOption(Options);
% Design Variables Information
Options.NumControls = Inputs.NumControls;
Options.NumStates = Inputs.NumStates;
Options.NumEqConstraints = Inputs.NumEqConstraints;
Options.NumIeqConstraints = Inputs.NumIeqConstraints;
% Bound Constraints
Options.BoundConstraints = 0;
% General Optimization Parameters
Options.MaxOptimizationItr = 100;
Options.ObjectiveFuncTol = 1e-14;
Options.GradientTol = 1e-14;
Options.TrialStepTol = 1e-14;
% Maximum Limited Memory Storage
Options.MaxLimitedMemoryStorage = 0;
switch algorithm_t
    case 'NonLinearCG'
        Options = getNonLinearCGOptions(Options);
    case 'QuasiNewton'
        Options = getQuasiNewtonOptions(Options);
    case 'TrustRegionQuasiNewton'
        Options = getTrustRegionQuasiNewtonOptions(Options);
    case 'LineSearchNewtonCG'
        Options = getLineSearchCGOptions(Options);
    case 'TrustRegionNewtonCG'
        Options = getTrustRegionCGOptions(Options);
    case 'InexactSQP'
        [Options] = getInexactSQPOptions(Options, Inputs);
    otherwise
        fprintf(' Invalid DOTk Algorithm. See Users Manual. ');
end
% Set bounds on design variables if necessary
[Options] = getConstraintOptions(Options);
% Set control variables initial guess
Options.Control = 0.5*ones(Inputs.NumControls,1);
% Set linear algebra
Options.LinearAlgebra = getLinearAlgebra;
% Get problem operators
[Operators] = getOperators(problem_t);
end
```

Figure 6.16. DOTk options m-file.

Iteration	Func-count	F(x)	norm(G)	norm(P)	TR-Itr	TR-Radius	ActualReduc	PredReduc	Ratio	Krylov-Itr	Krylov-Error	Krylov-Exit
0	1	5.8000e+00	1.2517e+02	*	*	*	*	*	*	*	*	*
1	2	3.8384e-02	3.9982e-01	2.3024e-01	1	1.0000e+04	5.7616e+00	5.7608e+00	1.0001e+00	2	1.8371e-12	Tolerance
2	3	3.8384e-02	3.9982e-01	5.0621e-01	1	5.0000e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	2	2.5000e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	3	1.2500e+03	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	4	6.2500e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	5	3.1250e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	6	1.5625e+02	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	7	7.8125e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	8	3.9062e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	9	1.9531e+01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	10	9.7656e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	11	4.8828e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	12	2.4414e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	13	1.2207e+00	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	14	6.1035e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	15	3.0518e-01	-1.0700e-01	3.8257e-02	-2.7970e+00	2	2.2935e-13	Tolerance
*	*	*	*	*	16	6.1035e-01	1.3895e-02	3.2231e-02	4.3110e-01	1	1.6297e-01	TrustRegion
3	19	5.3029e-03	7.2863e-02	5.8007e-03	1	6.1035e-01	1.9187e-02	1.9110e-02	1.0040e+00	1	4.9192e-02	Tolerance
4	20	3.3045e-03	2.5559e+00	1.7953e-01	1	6.1035e-01	1.9984e-03	5.5193e-03	3.6208e-01	2	1.3045e-14	Tolerance
5	21	2.7801e-05	8.4393e-03	2.5593e-03	1	6.1035e-01	3.2767e-03	3.2707e-03	1.0018e+00	1	7.0694e-03	Tolerance
6	22	7.5800e-08	1.2123e-02	1.1659e-02	1	6.1035e-01	2.7725e-05	2.7598e-05	1.0046e+00	2	8.2971e-16	Tolerance
7	23	2.4293e-09	4.4051e-05	1.2104e-05	1	6.1035e-01	7.3371e-08	7.3370e-08	1.0000e+00	1	4.4104e-05	Tolerance
8	24	5.9194e-16	1.0811e-06	1.1029e-04	1	6.1035e-01	2.4293e-09	2.4292e-09	1.0000e+00	2	3.0540e-21	Tolerance

Figure 6.17. Progress report: Bound-constrained optimization example problem.

Simple Constrained Optimization Problem

Lets consider the following parameter estimation problem:

$$\begin{aligned} &\underset{(u)\in\mathbb{R}^5}{\text{minimize}} \quad f(\mathbf{u}), \\ &\text{s.t.} \\ &\quad u_1^2 + u_2^2 + u_3^2 + u_4^2 + u_5^2 = 10 \\ &\quad u_2 u_3 - 5\ u_4 u_5 = 0 \\ &\quad u_1^3 + u_2^3 = -1, \end{aligned} \tag{6.4}$$

where

$$f(\mathbf{u}) = f(u_1, u_2, u_3, u_4, u_5) = \exp(u_1 u_2 u_3 u_4 u_5) - 0.5(1 + u_1^3 + u_2^3)^2. \tag{6.5}$$

The unique solution for this problem lies at $\mathbf{u} = (-1.71, 1.59, 1.82. - 0.763, -0.763)$, where $f(\mathbf{u}) = 5.395e^{-2}$. The initial guess for this problem was set to $\mathbf{u} = (-1.8, 1.7, 1.9, -0.8, -0.8)$. The progress report for this problem is shown in Figure 6.20.

C++ API

[9]Users are expected to complete a sequence of steps to implement the C++ main routine that enables DOTk solution methods for constrained optimization problems. These steps are described as follows:

[9]The full-space interface is currently being updated. Thus, the C++ source code display in the following Section will become legacy source code soon. DOTk users will be provided with the new full-space interface as soon as the new interface becomes available.

```cpp
#include "SerialLinearAlgebra.H"
#include "RosenbrockOperators.H"

#include "DOTk_VectorSpaceReduced.H"
#include "DOTk_Step.H"
#include "DOTk_TrustRegionStep.H"
#include "DOTk_TrustRegionNewtonCG.H"

int main()
{
    const Int num_controls = 2;
    std::tr1::shared_ptr<SerialLinearAlgebra> linear_algebra(new SerialLinearAlgebra());
    std::tr1::shared_ptr<RosenbrockOperators> operators(new RosenbrockOperators());
    std::tr1::shared_ptr<dotk::DOTk_VectorSpaceReduced>
    vector_space(new dotk::DOTk_VectorSpaceReduced(linear_algebra, num_controls));
    dotk::DOTk_TrustRegionNewtonCG trustRegionNewtonCG(vector_space, operators);

    Vector control(num_controls, 2.);
    Vector lower_bound(num_controls, 0.);
    Vector upper_bound(num_controls, 5.);
    vector_space->setInitialControl(control);
    vector_space->setControlBounds(lower_bound, upper_bound);

    trustRegionNewtonCG.setGradientPtr(dotk::types::USER_DEFINED_GRAD);
    trustRegionNewtonCG.setHessianPtr(dotk::types::USER_DEFINED_HESS);
    trustRegionNewtonCG.setInvHessianPtr(dotk::types::IDENTITY_INV_HESS);
    trustRegionNewtonCG.setTrustRegionStepPtr(dotk::types::TRUST_REGION_DOGLEG);
    trustRegionNewtonCG.enableDiagnosticsOutput(dotk::types::FINAL, true);
    trustRegionNewtonCG.getMin();

    return (0);
}
```

Figure 6.18. C++ main driver: Bound-consrained optimization example problem.

1. Define and declare the linear algebra API. The linear algebra class is define in the same manner as the linear algebra class define for the unconstrained optimization problem. see Figures 6.3. Thus, the linear algebra operations can be declared as shown in Figures 6.4, 6.5, 6.6, and 6.7. Recall that users can used the linear algebra library of their choice. Thus, users are not required to used DOTk default serial and parallel linear algebra interface. and 2.2

2. Define and declare the operators API. Figure 6.21 shows the operators class declaration for the constrained optimization problem defined in Equation 6.4. Similar to the unconstrained optimization example problem, each of the functions defined in class `FullSpaceTestOperators`[10] needs to be declared by DOTk users. For simplicity, only the function declaration for the objective function is shown in Figure 6.22[11] for this

[10]This class name is specific to the constrained optimization example problem shown herein, see Figure 6.21

[11]Several functions declaration required several lines of source code. Thus, the quality of the figures was

```
function [Options] = getConstraintOptions(Options)
% Constraint Modeling Method
%   1. OFF
%   2. BOUND_FEASIBLE_DIRECTION
%   3. BOUND_FEASIBLE_DIRECTION_MIN_REDUCTION
%   4. BOUND_PROJECT_SCALED_DIRECTION
%   5. BOUND_PROJECTION_ALONG_FEASIBLE_DIRECTION
Options.ConstraintMethod = 'BOUND_PROJECT_SCALED_DIRECTION';
if(~strcmp(Options.ConstraintMethod,'OFF'))
    Options.StepSize = 1.0;
    Options.LowerBounds = zeros(Options.NumControls,1);
    Options.UpperBounds = 5*ones(Options.NumControls,1);
end
end
```

Figure 6.19. Bound constraint modeling options.

Iteration	F(x)	norm(C)	norm(G)	TR-Radius	norm(N)	norm(T)	TR-Itr
0	2.0930e-02	1.0237e+00	2.6471e-03	1.0000e+04	0.0000e+00	0.0000e+00	0
1	5.2700e-02	3.1705e-02	3.6245e-03	1.0000e+04	1.5447e-01	4.6758e-02	1
2	5.3920e-02	6.0869e-04	6.2387e-05	1.0000e+04	4.5461e-03	2.3407e-02	1
3	5.3950e-02	1.7112e-07	2.6161e-08	1.0000e+04	9.5475e-05	3.8666e-04	1
4	5.3950e-02	3.0406e-14	3.5849e-15	1.0000e+04	2.7751e-08	1.5887e-07	1

Figure 6.20. Progress report: Constrained optimization example problem.

example problem

3. Declare main routine. This requires including the necessary header files as seen from Figure 6.23.

   (a) Initialize a `sol::params::space` data structure to define the number of design variables, see Chapter 3

   (b) Initialize DOTk vector space, e.g. `sol::state::reduced_space`, and set initial guess

   (c) Initialize a `sol::solver::krylov_params` data structure to define the dimensions of the augmented problem

   (d) Initialize C++ shared pointers to store the following dynamically allocated objects: linear algebra API, operators API, preconditioner operators API, `DOTk_ConstrainedInterface`, `DOTk_FullSpacePrecInterface`, `DOTk_GMRES`[12], and `DOTk_GMRES_Interface`.

not optimal for display purposes.

[12]The `GMRES` and `Conjugate_Direction` solvers are available in DOTk. The `GMRES` solver is recommended.

```
#ifndef FULLSPACETESTOPERATORS_H_
#define FULLSPACETESTOPERATORS_H_

#include "DOTk_Operators.H"

class FullSpaceTestOperators: public DOTk_Operators
{
public:
    FullSpaceTestOperators();
    virtual ~FullSpaceTestOperators(){}

    /* ========================================== Main Interface ========================================== */

    virtual Real Fval(const Vector& u, const Vector& z);
    virtual void Cval(const Vector& u, const Vector& z, Vector& cval);

    virtual void F_u(const Vector& u, const Vector& z, Vector &g);
    virtual void C_u(const Vector& u, const Vector& z, const Vector& dx, Vector& J_dx);
    virtual void adjC_u(const Vector& u, const Vector& z, const Vector& dy, Vector& J_dy);

    virtual void F_uu(const Vector& u, const Vector& z, const Vector& dx, Vector& H_dx);
    virtual void adjC_uu(const Vector& u, const Vector& z, const Vector& lambda, const Vector& du, Vector& adjC_du);

private:
    // unimplemented
    FullSpaceTestOperators(const FullSpaceTestOperators&);
    FullSpaceTestOperators operator=(const FullSpaceTestOperators&);
};

#endif /* FULLSPACETESTOPERATORS_H_ */
```

Figure 6.21. Bound constraint modeling options.

(e) Initialize DOTk solution method, e.g. `sol::DOTk_InexactSQP`

(f) Set stopping criteria or solve problem with default values

(g) Set second order information (Hessian) type[13]

(h) Enable diagnostics tools, diagnostics are disabled by default

(i) Set initial guess

(j) Call function `getMin` to solve the constrained optimization problem

The implementation of these steps is explicitly shown in Figure 6.23. Analogously to the parameters shown in Figure 6.23, users can define additional parameters pertinent to the solution method in used. The parameters and their respective default values are discussed in Chapter 3.

The `sol::solver::krylov_params(·)` data structure stores information relevant to the iterative solver. For instance, `sol::solver::krylov_params struct(dim(x), dim(g), itr, restart_itr)`, where `x` specifies the vector of primal variables, `g` specifies the vector of equality constraints, `itr` specifies the maximum number of iterations allow, and `restart_itr` specifies the solver's restart frequency, and and `dim(·)` specifies the argument dimension.

The `DOTk_FullSpacePrecInterface`, `DOTk_ConstrainedInterface`, and `DOTk_GMRES_Interface` were respectively created to:

[13]The first-order information is assumed to be provided by the user,i.e. `USER_DEFINED_GRAD`.

```cpp
Real FullSpaceTestOperators::Fval(const Vector& u, const Vector& z)
{
    // J(X) = exp(x1 * x2 * x3 * x4 * x5) - 0.5 * (1 + x1^3 + x2^3)^2
    const Real a = exp(u[0]*u[1]*u[2]*u[3]*u[4]);
    const Real b = 1. + pow(u[0], 3.) + pow(u[1], 3.);
    const Real c = 0.5 * pow(b, 2.);
    Real fval = a - c;
    return (fval);
}
```

Figure 6.22. Objective function declaration in C++.

1. allow users to provide user-defined preconditioners;

2. free users from having to implement the operations required to assemble the gradient operator and the application of a vector to the Hessian operator;

3. and to allow users to provide an user-defined solver to solve the augmented system problem.

MATLAB API

Similar to the previous example problem, the MATLAB structure array with the field names and values that correspond to the DOTk solution method in used is defined in File `getOptimizationOptions.m`. Users are expected to complete a sequence of steps to enable DOTk solution methods through the MATLAB API. These steps are described as follows:

1. Set path to DOTk-MEX directory, e.g. `PATH_TO_DOTk_INSTALL_DIRECTORY/matlab/exe` in file `driver.m`. Figure 6.13 shows `driver.m` source code

2. Define the linear algebra API in m-file `getLinearAlgebra.m`. Figure 6.14 shows `getLinearAlgebra.m` source code.

3. Define the objective function operators API in file `getObjectiveFunctionOperators.m`. Figure 6.24 shows the objective function operators needed to solve the constrained optimization problem defined in Equation 6.4[14]

4. Define the equality constrained operators API in file `getEqualityConstrainedOperators.m`. Figure 6.25 shows a simple example on how to define and declare the equality

```cpp
#include "SOL_State.H"
#include "SOL_GMRES.H"
#include "SOL_IOTools.H"
#include "SerialLinearAlgebra.H"
#include "FullSpaceTestOperators.H"
#include "FullSpacePrecOperators.H"
#include "SOL_ConstrainedInterface.H"
#include "SOL_FullSpacePrecInterface.H"
#include "SOL_AugmentedSystem.H"
#include "SOL_Hessian.H"
#include "SOL_GMRES_Interface.H"
#include "SOL_InexactSQP.H"

int main()
{
    const Int num_states = 5;
    const Int num_controls = 0;
    const Int num_eq_constraints = 3;
    const Int num_iq_constraints = 0;
    const Int max_num_itr = 200;
    const Int num_rstrt_itr = 200;
    // problem set up
    sol::params::space space_dim(num_controls, num_states, num_eq_constraints, num_iq_constraints);
    sol::state::full_space state(space_dim);
    std::tr1::shared_ptr<SerialLinearAlgebra> la(new SerialLinearAlgebra());
    std::tr1::shared_ptr<FullSpaceTestOperators> op(new FullSpaceTestOperators());
    std::tr1::shared_ptr<SOL_ConstrainedInterface> it( new SOL_ConstrainedInterface(op, space_dim) );
    // initialize Krylov solver
    std::tr1::shared_ptr<FullSpacePrecOperators> prec_op(new FullSpacePrecOperators());
    std::tr1::shared_ptr<SOL_FullSpacePrecInterface> prec_it(new SOL_FullSpacePrecInterface(prec_op, space_dim));
    sol::solver::krylov_params krylov_params(num_states, num_eq_constraints, max_num_itr, num_rstrt_itr);
    std::tr1::shared_ptr<SOL_GMRES> solver( new SOL_GMRES(it, la, prec_it, krylov_params) );
    std::tr1::shared_ptr<SOL_GMRES_Interface> solver_interface( new SOL_GMRES_Interface(solver) );
    // optimization problem
    sol::SOL_InexactSQP opt(it, la, solver_interface, space_dim);
    opt.getHessianPtr()->setHessianOperator(sol::sol_types::USER_DEFINED_HESS);
    opt.getIOPtr()->setOutputDataDisplayOption(sol::sol_types::FINAL);
    // set initial guess
    state.Primal.front()[0] = -1.8;
    state.Primal.front()[1] = 1.7;
    state.Primal.front()[2] = 1.9;
    state.Primal.front()[3] = -0.8;
    state.Primal.front()[4] = -0.8;
    opt.getMin(state);

    return (0);
}
```

Figure 6.23. C++ main driver: Constrained optimization example problem.

constrained operators for this example problem[15]

5. Solve constrained optimization problem

   (a) Select solution method, e.g. `algorithm_t='InexactSQP'`

   (b) Select problem type/class, e.g. `problem_t='Constrained'`

   (c) Disable finite difference diagnostics tools

   (d) Define MATLAB structure array that stores the field names and values that are used to set the number of design variables as follows:

[14] Only the objective function declaration is shown in Figure 6.24 for simplicity. Other objective function operators are declared in similar manner.

[15] Only the equality constraint function declaration is shown in Figure 6.25 for simplicity. The other equality constraint operators, e.g. the required derivative operators, are declared in similar manner.

```
function [ObjectiveFunctionOperators] = getObjectiveFunctionOperators()
ObjectiveFunctionOperators.Fval=@(u,z) Fval(u,z);
ObjectiveFunctionOperators.F_u=@(u,z) F_u(u,z);
ObjectiveFunctionOperators.F_z=@(u,z) F_z(u,z);
ObjectiveFunctionOperators.F_uu=@(u,z,du) F_uu(u,z,du);
ObjectiveFunctionOperators.F_uz=@(u,z,dz) F_uz(u,z,dz);
ObjectiveFunctionOperators.F_zz=@(u,z,dz) F_zz(u,z,dz);
ObjectiveFunctionOperators.F_zu=@(u,z,du) F_zu(u,z,du);
end
function [fval] = Fval(u,z)
    % J(X) = exp(x1 * x2 * x3 * x4 * x5) - 0.5 * (1 + x1^3 + x2^3)^2
    a = exp(u(1)*u(2)*u(3)*u(4)*u(5));
    b = 1. + power(u(1), 3.) + power(u(2), 3.);
    c = 0.5 * power(b, 2.);
    fval = a - c;
end
```

Figure 6.24. Objective function operator in MATLAB.

```
Inputs.NumControls = 0
Inputs.NumStates = 5
Inputs.NumEqConstraints = 3
Inputs.NumIeqConstraints = 0
```

(e) Call function `getOptimizationOptions(algorithm_t,problem_t,Inputs)` to initialize both the Options and Operators structure arrays as follows:

```
[Options,Operators] = ...
getOptimizationOptions(algorithm_t,problem_t,Inputs)
```

There are additional input options that advanced users can adjust for the inexact trust region SQP algorithm. Users can set these options in function `getInexactSQPOptions`, which is defined and declared in file `getOptimizationOptions.m`. Figure 6.26 shows function `getInexactSQPOptions` source code

(f) Call function `getMin(algorithm_t,Options,Operators)` to solve optimization problem, e.g.

```
[Output,TimeData] = getMin(algorithm_t,Options,Operators).
```

See Figure 6.13[16]

(g) The Output MATLAB structure array stores relevant problem information. For instance, the optimal state is stored in `Output.State`. The Output MATLAB structure array stores the value of the objective function (`Output.ObjectiveFunction`), optimal state (`Output.State`), optimal control (`Output.Control`),

[16]The same driver used to solve the unconstrained optimization problem can be employed to solve the constrained optimization problem.

```
function [EqualityConstraintOperators] = getEqualityConstraintOperators()
EqualityConstraintOperators.Cval=@(u,z) Cval(u,z);
EqualityConstraintOperators.C_u=@(u,z,du) C_u(u,z,du);
EqualityConstraintOperators.C_z=@(u,z,dz) C_z(u,z,dz);
EqualityConstraintOperators.adjC_u=@(u,z,lambda) adjC_u(u,z,lambda);
EqualityConstraintOperators.adjC_z=@(u,z,lambda) adjC_z(u,z,lambda);
EqualityConstraintOperators.adjC_uu=@(u,z,lambda,du) adjC_uu(u,z,lambda,du);
EqualityConstraintOperators.adjC_uz=@(u,z,lambda,dz) adjC_uz(u,z,lambda,dz);
EqualityConstraintOperators.adjC_zz=@(u,z,lambda,dz) adjC_zz(u,z,lambda,dz);
EqualityConstraintOperators.adjC_zu=@(u,z,lambda,du) adjC_zu(u,z,lambda,du);
EqualityConstraintOperators.setRestoreOperators = ...
    @(restore) setRestoreOperators(restore);
end
function [cval] = Cval(u,z)
    % C1 = x1^2 + x2^2 + x3^2 + x4^2 + x5^2 - 10
    cval(1) = power(u(1), 2.) + power(u(2), 2.) + power(u(3), 2.) + ...
        power(u(4), 2.) + power(u(5), 2.) - 10;
    % C2 = x2 * x3 - 5. * x4 * x5
    cval(2) =  u(2) * u(3) - 5. * u(4) * u(5);
    % C3 = x1^3 + x2^3 + 1.
    cval(3) = power(u(1), 3.) + power(u(2), 3.) + 1.;
end
```

Figure 6.25. Equality constraint operator in MATLAB.

optimal lagrange multipliers (`Output.LagrangeMultipliers`), norm of the gradient (`Output.Gradient`), and norm of the trial step (`Output.TrialStep`) for optimization problems solved with a full-space formulation strategy, e.g.

```
[Output, TimeData] = mxInexactSQP(Options,Operators)
```

```
function [Options] = getInexactSQPOptions(Options, Inputs)
Options.DOTk_Algorithm = 6;
Options.TangentialSubProblemMaxItr = 200;
Options.MaxKrylovSolverItr = 10;
Options.KrylovRestartItr = Options.MaxKrylovSolverItr;
Options.ConstraintTol = 1e-10;
Options.QuasiNormalTol = 1e-4;
Options.TangentialTol = 1e-4;
Options.TangentialTolReductionFactor = 1e-3;
Options.TolReductionFactor = 1e-1;
Options.ProjectionTol = 1e-4;
Options.LagrangeMultipliersTol = 1e-4;
Options.LagrangeMultipliersGradTol = 1e4;
Options.OrthogonalityTol = 0.5;
Options.ActualOverPredictedReductionParam = 1e-8;
Options.QuasiNormalTrustRegionFractionParam = 0.8;
Options.MaxEffectiveTangentialOverTrialStepParam = 2;
% Other Options
Options = getTrustRegionOptions(Options);
Options = getHessianOperatorOptions(Options);
Options = getGradientOperatorOptions(Options);
Options = getFullSpaceRightPrecOptions(Options);
Options = getFullSpaceLeftPrecOptions(Options);
% Set state variables initial guess and corresponding output variable
Options.State = ones(Inputs.NumStates,1);
% Set lagrange multipliers initial guess and corresponding output variable
Options.LagrangeMultipliers = zeros(Inputs.NumEqConstraints,1);
end
```

Figure 6.26. Source code for getInexactSQPOptions function.

References

[1] An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[2] M. A. Aguiló and D. Ridzal. Optimality conditions for the numerical solution of optimization problems with PDE constraints: Theoretical framework and applications to parameter identification and optimal control problems. SAND2014-2464, Sandia National Laboratories, P.O. Box 5800, MS 0845, Albuquerque, NM, 2014.

[3] D.P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[4] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 1964.

[5] M. Heinkenschloss and D. Ridzal. An Inexact Trust-Region SQP Method with Applications to PDE-Constrained Optimization. In *Numerical Mathematics and Advanced Applications: Proceedings of ENUMATH 2007*, pages 613–620. Springer Verlag, 2008.

[6] S. G. Krantz and H. R. Parks. *The Implicit Function Theorem: History, Theory, and Applications*. Birkhauser, 2002.

[7] MATLAB 8.0 and Statistics Toolbox 8.1, The MathWorks, Inc., Natick, Massachusetts, United States.

[8] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

[9] T. Rees, H. Dollar, and A. Wathen. Optimal solvers for pde-constrained optimization. *SIAM Journal on Scientific Computing*, 32(1):271–298, 2010.

[10] D. Ridzal. *Trust–Region SQP Methods with Inexact Linear System Solves for Large–Scale Optimization*. PhD thesis, Department of Computational and Applied Mathematics, Rice University, Houston, TX, April 2006.

[11] D. Ridzal, M. A. Aguiló, and M. Heinkenschloss. Numerical study of a matrix-free trust-region SQP method for equality constrained optimization. SAND2011-9346, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-0845, 2011.

[12] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 33(3):175–184, 1960.

[13] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[14] T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, 20(3):626–637, 1983.

[15] S.J. Wright. *Primal-Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, 1997.

DISTRIBUTION:

1	MS 0346	D. Gregory Tipton, 1523
1	MS 0557	Todd W. Simmermacher, 1523
1	MS 0557	Sharlotte L. Bolyard Kramer, 1528
1	MS 0812	Nicole L. Breivik, 1524
1	MS 0812	Kevin N. Long, 1524
1	MS 0812	William M. Scherzinger, 1524
1	MS 0815	Justine E. Johannes, 1500
1	MS 0828	Ryan B. Bond, 1541
1	MS 0828	Walter R. Witkowski, 1544
1	MS 0828	Angel Urbina, 1544
1	MS 0828	George E. Orient, 1544
1	MS 0828	John R. Red-Horse, 1544
1	MS 0828	Amalia R. Black, 1544
1	MS 0828	Brian Carnes, 1544
1	MS 0828	Michael Ross, 1523
1	MS 0840	Huei Eliot Fang, 1524
1	MS 0840	Stephen W. Attaway, 1555
1	MS 0840	Greg T. Sharp, 1555
1	MS 0845	David E. Womble, 1540
1	MS 0845	Kyran D. Mish, 1542
1	MS 0845	Joseph Jung, 1542
1	MS 0845	Kendall H. Pierson, 1542
1	MS 0885	Mark F. Smith, 1830
1	MS 0897	Teddy D. Blacker, 1543
1	MS 0897	Brett W. Clark, 1543
1	MS 0958	Bradley Howell Jared, 1832
1	MS 1209	Steve N. Kempka, 5940
1	MS 1323	Thomas E. Voth, 1443
		,
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories