Design Optimization Toolkit
Library for Multidisciplinary Optimization
Surrogate-Based Algorithm Users' Manual

Trust Region, Discrete Empirical Interpolation algorithm for large-scale optimization

Miguel A. Aguiló

aguilo.miguel@gmail.com
November 2016

Abstract

This users' manual presents how to apply the TRROM algorithm for fast large-scale optimization. The proposed optimization framework relies on the trust region method to progressively improve the predictive reliability of parametric reduced-order models during optimization. The trust region framework enables the application of sound metrics to control the error induced by inexact objective and gradient evaluations during optimization while certifying global convergence. To further improve computational efficiency, the proposed framework relies on the matrix-version of the Discrete Empirical Interpolation Method (DEIM) to efficiently reduced nonaffine parametrized systems arising in partial differential equation (PDE) constrained optimization.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

Motivation and Objectives here.

1.2 Contributions

Contributions here.

1.3 Statement of Originality

Statement here.

1.4 Publications

Publications here.

Chapter 2

Linear Algebra Interface

This Chapter describes the core linear algebra interfaces. The optimization algorithm was intentionally designed to enable users to rely on their preferred linear algebra package to perform essential linear algebra operations. This inherent flexibility minimizes software dependencies and gives users the freedom to rely on their custom linear algebra package when using the parametric reduced-order model based optimization algorithm. Examples with Trilinos' linear algebra packages will be provided to illustrate how to implement these core interfaces.

2.1 Vector Interface

An adapter vector container class is needed to defines basic vector container operations that enable common linear algebra operations, data modification, and creation/construction of new vector containers that will utilize the underlying custom vector container operations[1]. The core vector container operations are:

1. `scale(alpha)`: Scale vector by a constant, `this` $= \alpha *$ `this`, where `this` denotes a vector
2. `update(alpha,x,beta)`: Update vector values with scaled values of `x`, `this` = `beta` $*$ `this` + `alpha` $*$ `x`

[1] *Default vector container adapter classes are provided based on C++ standard vector container library and Trilinos Epetra linear algebra package*

3. `dot(x)`: Returns the inner product of two vectors, `output` = $\texttt{this}^{\texttt{T}}\texttt{x}$, where `output` denotes a scalar number

4. `norm()`: Returns the norm of `this` vector, e.g. `output` = $\sqrt{\texttt{this}^{\texttt{T}}\texttt{this}}$

5. `elementWiseMultiplication(x)`: Perform element-wise multiplication between two vectors, `this` = `this` $\diamond$ `x`, where $\diamond$ is used to denote element-wise multiplication

6. `sum()`: Returns the sum of all the elements in the vector, `output` = $\sum_{\texttt{i}=1}^{\texttt{N}} \texttt{this}_{\texttt{i}}$, where `N` denotes the number of elements

7. `modulus()`: Computes the absolute value of each element in the vector, $|\texttt{this}| = \{\texttt{this}$, if `this` ≥ 0; or $-\texttt{this}$, if `this` < 0

8. `max(index)`: Returns the maximum element in a range and its global position index, `output` = max(`index`), where `output` denotes the maximum value and `index` denotes its global position index[2]

9. `min(index)`: Returns the maximum element in a range and its global position index, `output` = min(`index`), where `output` denotes the minimum value and `index` denotes its global position index

10. `fill(alpha)`: Initialize all elements in `this` vector with constant value `alpha`

11. `size()`: Returns the number of local elements in `this` vector

12. `create(length)`: Creates memory for a vector that makes use of the underlying custom linear algebra library, where `length` denotes global length[3]

13. `reference operator[](index)`: Returns a reference to the element at position `index` in the vector container

[2] *We intentionally use the keyword global to distinguish between local data and global data in distributed memory settings. Thus, for serial applications, there is not distinction between global and local data since both denote the same information.*

[3] *The default input argument* ***length*** *is set to 0, where* ***length*** *denotes global length, and thus a vector of global length equal to* ***this*** *vector is created. However, if a non-zero value is provided,* ***length*** $\neq 0$*, memory is created for a vector of global length equal to the non-zero* ***length*** *input parameter.*

14. `const reference operator[](index) const`: Returns a reference to the element at position `index` in the vector container

2.1.1 Example

A brief example is presented to illustrate how to implement a custom vector container class. Specifically, the Trilinos Epetra linear algebra package is used to construct the custom vector class presented herein, which is shortened for simplicity. Readers are encouraged to explore the full adapter vector container class implementation, which is available in *TRROM_EpetraVector.hpp*.

The first step is to implement a custom vector container class that derives from pure virtual parent class `trrom::Vector⟨ScalarType⟩` class, where `ScalarType` denotes a C++ templated typename. Finally, users need to define the mandatory vector container operations presented in Section 2.1. For instance, a custom `Epetra_Vector` container class can be implemented as follows:

```
namespace trrom{
class EpetraVector : public trrom::Vector⟨double⟩{
public:
    explicit EpetraVector(const Epetra_BlockMap & _input) :
        m_Data(new Epetra_Vector(_input)){}
    virtual ~EpetraVector(){}

    int size() const {return (m_Data->MyLength());}
    double norm() const {
        double output = 0.0;
        m_Data->Norm2(&output);
        return (output);
    }
    double & operator[ ](int _input)
        {return (m_Data->operator[ ](_input));}
```

```
    const double & operator[ ](int _input) const
        {return (m_Data->operator[ ](_input));}

    void fill(const double & _input) {m_Data->PutScalar(_input);}
    void scale(constdouble & _input) {m_Data->Scale(_input);}
    void modulus(){m_Data->Abs();}
    void update(constdouble & _alpha,
                const trrom::Vector<double> & _x,
                const double & _beta){
        double this_scalar = 1.0;
        const trrom::EpetraVector & input =
            dynamic_cast<const trrom::EpetraVector &>(_x);
        m_Data->Update(_alpha, *input.m_Data, this_scalar);
    }

    /* implement remaining mandatory vector container operations */

    /* if necessary, implement additional vector container helper functions, e.g. */
    std::tr1::shared_ptr<Epetra_Vector> & data(){return (m_Data);}
    const std::tr1::shared_ptr<Epetra_Vector> & data() const {return (m_Data);}

private:
    std::tr1::shared_ptr<Epetra_Vector> m_Data;
}; /* class EpetraVector */
} /* namespace trrom */
```

2.2 Matrix Interface

An adapter multivector container class is needed to defines basic multivector (e.g. matrix) container operations that enable common linear algebra operations, data modification, and creation/construction of new multivector containers that will utilize the underlying custom multivector container operations[4]. The core multivector container operations are:

1. `scale(alpha)`: Scale multivector by a constant, `This` $= \alpha *$ `This`, where `This` denotes a multivector

2. `update(alpha,X,beta)`: Update multivector values with scaled values of `X`, `This` = `beta` $*$ `This` $+$ `alpha` $*$ `X`

3. `fill(alpha)`: Initialize all elements in `This` multivector with constant value `alpha`

4. `gemv(This_transpose,alpha,x,beta,y)`: General Matrix-Vector multiplication, `y = beta*y + alpha*This`$^{\texttt{This\_transpose}}$`*x`, where `This_transpose` denotes the transpose of `This_transpose` if=true, else no transpose if=false.

5. `gemm(This_transpose,A_transpose,alpha,A,beta,B)`: General Matrix-Matrix multiplication, `B = beta*B + alpha*This`$^{\texttt{This\_transpose}}$`*A`$^{\texttt{A\_transpose}}$, where `This_transpose` and `A_transpose` respectively denotes the transpose of `This` and `A` if=true, else no transpose if=false.

6. `replaceGlobalValue(global_row_index,global_column_index,new_value)`: Replace current value at the specified `(global_row_index,global_column_index)` location with `new_value`

7. `getNumRows()`: Returns the global number of rows of `This` multivector

8. `getNumCols()`: Returns the global number of columns of `This` multivector

9. `insert(x)`: `This` multivector is extended by inserting a new vector `x` after the last vector position, effectively increasing `This` multivector container size by one

[4] *Default multivector container adapter classes are provided based on C++ standard vector container library and Trilinos Epetra linear algebra package*

10. `getVector(global_column_index)`: Vector access function, returns a vector with global position`global_column_index`

11. `create(global_num_rows,global_num_columns)`: Creates memory for a multivector that makes use of the underlying custom multivector operations[5]

2.2.1 Example

A brief example is presented to illustrate how to implement a custom multivector container class. Specifically, the Trilinos Epetra linear algebra package is used to construct a custom multivector class. The example presented herein is shortened for simplicity. Readers are encouraged to explore the full multivector adapter class implementation, which is available in *TRROM_EpetraMultiVector.hpp*.

The first step is to implement a custom multivector container class that derives from the pure virtual parent class `trrom::MultiVector⟨ScalarType⟩`, where `ScalarType` denotes a C++ templated typename. Finally, users need to define each multivector container operations presented in Section 2.2 for the custom multivector container class. For instance, a custom `Epetra_MultiVector` container class can be implemented as follows:

```
namespace trrom{
class EpetraMultiVector : public trrom::MultiVector⟨double⟩{
public:
    EpetraMultiVector(const Epetra_BlockMap & _map,
                      const int & _num_vectors) :
        m_ThisVector(),
        m_Data(new Epetra_MultiVector(_map, _num_vectors)){}
    virtual ~EpetraMultiVector(){}
```

[5] *The default input arguments* ***global_num_rows*** *and* ***global_num_columns*** *are set to 0. Thus, a multivector of* ***global_num_rows*** *and* ***global_num_columns*** *equal to* ***This*** *multivector is created if no values are provided for* ***global_num_rows*** *and* ***global_num_columns****. However, if non-zero values are provided for these input arguments,* $\boldsymbol{global\_num\_rows} \neq 0$ *and* $\boldsymbol{global\_num\_columns} \neq 0$*, memory is created for a multivector of* ***global_num_rows*** *and* ***global_num_columns*** *equal to the non-zero values provided.*

```
int getNumRows() const {return (m_Data->GlobalLength());}
int getNumCols() const {return (m_Data->NumVectors());}
void fill(const double & _input) {m_Data->PutScalar(_input);}
std::tr1::shared_ptr<trrom::Vector<double>> & getVector(int _index){
    m_ThisVector.reset(new trrom::EpetraVector(Epetra_DataAccess::View,
                                               *m_Data,
                                               _index));
    return (m_ThisVector);
}
void replaceGlobalValue(const int & _global_row_index,
                        const int & _vector_index,
                        const int & _value) {
    m_Data->ReplaceGlobalValue(_global_row_index,_vector_index,_value);
}
void gemv(const bool & _this_transpose,
          const double & _this_scalar,
          const trrom::Vector<double> & _input,
          const double & _output_scalar,
          trrom::Vector<double> & _output){
    /* convert bool type to char type */
    std::vector<char> this_transpose = this->transpose(_this_transpose);
    trrom::EpetraVector & output =
        dynamic_cast<trrom::EpetraVector &>(_output);
    const trrom::EpetraVector & input =
        dynamic_cast<const trrom::EpetraVector &>(_input);
    output.data()->Multiply(this_transpose[0],
                            'N',
                            _this_scalar,
```

```
                                        *m_Data,
                                        *input.data(),
                                        _output_scalar);
    }

    /* implement remaining mandatory multivector container operations */

    /* if necessary, implement additional vector container helper functions, e.g. */
    std::tr1::shared_ptr<Epetra_MultiVector> & data(){return (m_Data);}
    const std::tr1::shared_ptr<Epetra_MultiVector> & data() const {return (m_Data);}

private:
    std::tr1::shared_ptr<Epetra_Vector> m_ThisVector;
    std::tr1::shared_ptr<Epetra_MultiVector> m_Data;
}; /* class EpetraMultiVector */
} /* namespace trrom */
```

2.3 Spectral Decomposition Interface

The implementation of an adapter spectral decomposition class is required to enable parametric reduced-order models during optimization. The optimization algorithm will invoke the spectral decomposition solver online to update multiple orthonormal bases from data ensembles (displacement, Lagrange multipliers, etc.). However, if the low rank spectral decomposition solver is enabled, the high fidelity spectral decomposition solver will only be necessary for the first orthonormal basis updates. Subsequent orthonormal basis updates will rely on an in-situ low rank spectral decomposition solver. Thus, the high fidelity singular value decomposition solver will not be required during subsequent orthonormal basis updates. Contrary, if the low rank spectral decomposition solver is disabled, the high fidelity spectral decomposition solver accessed through the adapter spectral decomposition class will be used for all the orthonormal

basis updates during optimization.

The adapter spectral decomposition class needs to provide/implement the following `solve` function:

1. `solve(A,S,U,V)`: Performs singular value decomposition of matrix `A`, where `S` denotes the singular values, `U` denotes the left singular vectors, and `V` denotes the right singular vectors

The algorithm will call `solve` every time an updated set of orthonormal basis is required to improve the predictive reliability of the parametric reduced-order model.

2.3.1 Example

A brief example is presented to illustrate how to implement a custom spectral decomposition class. Specifically, the Trilinos Teuchos serial singular value decomposition solver is used to construct the custom class presented herein, which is shortened for simplicity. Readers are encouraged to explore the full implementation of the adapter spectral decomposition class available in *TRROM_TeuchosSVD.hpp* and *TRROM_TeuchosSVD.cpp*.

The first step is to implement a custom spectral decomposition class that derives from pure virtual parent class `trrom::SpectralDecomposition`. The final step is to implement the `solve` function described in Section 2.3. For instance, a custom `Teuchos` based singular value decomposition class can be implemented as follows:

```
namespace trrom {
class TeuchosSVD : public trrom::SpectralDecomposition {
public:
    TeuchosSVD() : m_LAPACK() {}
    virtual ~TeuchosSVD() {}

    void solve(const std::tr1::shared_ptr<trrom::Matrix> & _A,
```

```
                std::tr1::shared_ptr<trrom::Vector> & _S,
                std::tr1::shared_ptr<trrom::Matrix> & _U,
                std::tr1::shared_ptr<trrom::Matrix> & _V) {
        int nrows = _A->getNumRows();
        int ncols = _A->getNumCols();
        int spectral_dim = std::min(nrows, ncols);

        /* copy data since original data is always overwritten */
        trrom::TeuchosArray<double> A(spectral_dim);
        A.update(1.0, *(_A), 0.0);

        /* resize output data accordingly */
        _S.reset(new trrom::TeuchosArray<double>(spectral_dim));
        _U.reset(new trrom::TeuchosSerialDenseMatrix<double>(nrows, ncols));
        _V.reset(new trrom::TeuchosSerialDenseMatrix<double>(ncols, ncols));

        /* solve singular value problem */
        m_LAPACK->GESVD(/* provide mandatory input parameters */);
    }

private:
    Teuchos::LAPACK<int, double> m_LAPACK;
}; /* class TeuchosSVD */
} /* namespace trrom */
```

2.4 Solver Interface

An adapter solver class is available to enable the application of custom direct or iterative solvers for the solution of the low-fidelity (reduced) model. This interface allows the application of

different solvers for the high- and low-fidelity calculations. This flexibility facilitates the use of a solver that is in agreement with the custom linear algebra specified by the user. However, users can potentially avoid implementing the adapter solver class by enabling the default Trilinos parallel solver.

The adapter solver interface class needs to provide the following functionality:

1. `solve(A,b,x)`: Solves the reduced system of equations, where `A` denotes the reduced matrix, `b` denotes the reduced right hand side vector, and `x` denotes the reduced left hand side vector (solution vector)

The optimization algorithm will query the solver interface during optimization to compute the low-fidelity solution `x`. If the default parallel Trilinos solver is enabled, the algorithm will manage the low-fidelity solve in-situ without having to interact with the application code.

2.4.1 Example

A brief example is presented to illustrate how to implement a custom solver interface class. Specifically, the Trilinos Teuchos serial dense solver solver is used to construct the custom class presented in this section, which is shortened for simplicity. Readers are encouraged to explore the full implementation of the adapter solver interface class available in *TRROM_TeuchosSerialDenseSolver* and *TRROM_TeuchosSerialDenseSolver.cpp*.

The first step is to implement the custom solver interface class that derives from the pure virtual parent class `trrom::SolverInterface`. Finally, the solve function described in Section 2.4 is implemented. For instance, a custom solver interface class based on the Teuchos serial dense solver can be implemented as follows:

```
namespace trrom {
class TeuchosSerialDenseSolver : public trrom::SolverInterface {
public:
```

```
    TeuchosSerialDenseSolver() :
        m_Solver(new Teuchos::SerialDenseSolver<int, double>) {}
    virtual ~TeuchosSerialDenseSolver() {delete m_Solver;}

    void solve(const std::tr1::shared_ptr<trrom::Matrix> & _A,
               const std::tr1::shared_ptr<trrom::Vector> & _b,
               std::tr1::shared_ptr<trrom::Matrix> & _x) {
        int nrows = _A->getNumRows();
        int ncols = _A->getNumCols();

        trrom::TeuchosSerialDenseMatrix matrix(nrows, ncols);
        matrix.update(1., _A, 0.);
        m_Solver->setMatrix(Teuchos::rcp(&(*matrix.data()), false));

        trrom::TeuchosSerialDenseVector rhs(nrows);
        rhs.update(1., _b, 0.);
        trrom::TeuchosSerialDenseVector lhs(ncols);
        m_Solver->setVector(Teuchos::rcp(&(*lhs.data()), false),
                            Teuchos::rcp(&(*rhs.data()), false));

        m_Solver->factorWithEquilibration(true);
        m_Solver->factor();
        m_Solver->solve();

        _lhs.update(1., lhs, 0.);
    }

private:
    Teuchos::SerialDenseSolver<int, double>* m_Solver;
```

```
};  /* class TeuchosSerialDenseSolver */

}  /* namespace trrom */
```

Chapter 3

Optimization Interface

This Chapter describes the core interfaces used by the optimization algorithm to query high- and low-fidelity functional values and derivatives during optimization. This information is required to provide the best set of variables that meet a set of criteria at every iteration. Users are expected to implement these interfaces, without them, the algorithm cannot operate.

A general partial differential equation (PDE) constrained optimization problem is defined as:

$$\begin{aligned} &\min_{(u,z)\in\mathbb{R}^{n_u}\times\mathbb{R}^{n_z}} && f(u,z) \\ &\text{s.t.} && \\ &&& g(u,z) = 0, \\ &&& h(u,z) \leq 0, \end{aligned} \tag{3.1}$$

where u is the state, z is the control variable, f is the objective function, g is the equality constraint (PDE constraint), and h is the inequality constraint. A flexible interface is provided to enable easy communication between the optimization algorithm and a third-party application. These interfaces are discussed in the subsequent Sections.

3.1 Objective Function

3.1.1 Component-based interface

The core objective function operations required by the algorithm for a component-based interface are:

1. `value(u,z,tolerance,flag)`: Evaluates the objective function given a set of state (`u`) and control (`z`) variables. Here, `tolerance` denotes the maximum allowable error due to inexact objective function evaluations, and `flag` is an output parameter that is respectively set to `true` or `false` if the error bound defined by the user is exceeded or not[1]

2. `partialDerivativeControl(u,z,output)`: Computes the partial derivative of the objective function with respect to the control variables, where `output` denotes $\partial \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{z}$

3. `partialDerivativeState(u,z,output)`: Computes the partial derivative of the objective function with respect to the state variables, where `output` denotes $\partial \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}$

4. `partialDerivativeStateState(u,z,du,output)`: Computes the application of direction `du` to the second-order partial derivative of the objective function with respect to the state variables, where `output` denotes $(\partial^2 \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}^2)\mathtt{du}$

5. `partialDerivativeControlControl(u,z,dz,output)`: Computes the application of direction `dz` to the second-order partial derivative of the objective function with respect to the control variables, where `output` denotes $(\partial^2 \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{z}^2)\mathtt{dz}$

6. `partialDerivativeStateControl(u,z,dz,output)`: Computes the application of direction `dz` to the mixed partial derivative of the objective function with respect to the state and control variables, where `output` denotes $(\partial^2 \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}\partial \mathtt{z})\mathtt{dz}$

[1] *The error bound should only be evaluated during low-fidelity evaluations (the parametric reduced-order model is active).*

7. `partialDerivativeControlState(u,z,du,output)`: Computes the application of direction `du` to the mixed partial derivative of the objective function with respect to the control and state variables, where `output` denotes $(\partial^2 \mathtt{f}(\mathtt{u},\mathtt{z})/\partial \mathtt{z} \partial \mathtt{u})\mathtt{du}$

8. `output checkGradientInexactness(u,z,gradient,tolerance)`: Evaluates the error bound on the `gradient` and checks if the allowable error due to an inexact gradient evaluation has been exceeded. The parameter `output` is either set to true or false if the error bound exceeds or satisfies the allowable error `tolerance`[1]

9. `fidelity(flag)`: Informs the third-party application if the parametric reduced-order model is active or inactive, which is relevant in order to avoid unnecessary error bound evaluations. The `flag` is either set to `HIGH_FIDELITY` or `LOW_FIDELITY`, where `HIGH_FIDELITY` denotes that the reduced-order model is inactive and `LOW_FIDELITY` denotes that the reduced order model is active

The optimization algorithm relies on this interface to query relevant objective function information throughout optimization. Thus, the implementation of this interface is mandatory[2].

3.1.2 Standard interface

The core objective function operations required by the algorithm given a standard interface are:

1. `value(u,z,tolerance,flag)`: Evaluates the objective function given a set of state (`u`) and control (`z`) variables. Here, `tolerance` denotes the maximum allowable error due to inexact objective function evaluations, and `flag` is an output parameter that is respectively set to `true` or `false` if the error bound defined by the user is exceeded or not

[2] *Certain derivative operators, specifically the second-order and mixed partial derivatives, might not be defined for a given application. In this case, users are required to declared and defined a* `NULL` *operator (a function that returns or manipulates no data) for these functionalities*

2. `gradient(u,z,λ,g)`[3]: Computes the total derivative of the objective function with respect to the control variables, where λ denotes the Lagrange multipliers `g` denotes `df(u,z)/dz`

3. `hessian(u,z,λ,dz,h_dz)`[4]: Computes the application of direction `dz` to the Hessian operator, where λ denotes the Lagrange multipliers and `h_dz` denotes $(\mathtt{d^2f(u,z)/dz^2})$ `dz`

4. `output checkGradientInexactness(u,z,gradient,tolerance)`: Evaluates the error bound on the `gradient` and checks if the allowable error due to an inexact gradient evaluation has been exceeded. The parameter `output` is either set to true or false if the error bound exceeds or satisfies the allowable error `tolerance`[1]

5. `fidelity(flag)`: Informs the third-party application if the parametric reduced-order model is active or inactive, which is relevant in order to avoid unnecessary error bound evaluations. The `flag` is either set to `HIGH_FIDELITY` or `LOW_FIDELITY`, where `HIGH_FIDELITY` denotes that the reduced-order model is inactive and `LOW_FIDELITY` denotes that the reduced order model is active

3.2 Equality Constraint

3.2.1 Component-based interface

The core equality constraint operations required by the algorithm for a component-based interface are:

1. `solve(z,u,data)`: Solves the following linear system of equations, $\mathtt{u} = \mathtt{A(z)}^{-1}\mathtt{f}$, where `data` denotes a reduced basis data structure of type `trrom:;ReducedBasisData` that the algorithm uses to collect key information during optimization, `u` is the state, `A(z)` is a matrix that depends on the control (`z`) variables, and `f` is the right hand side vector

[3] *Users are responsible for properly computing the Lagrange multipliers and assembling the total derivative.*
[4] *Users are responsible for properly computing the application of direction dz to the Hessian operator.*

2. `applyInverseJacobianState(u,z,b,x)`: Solves the following linear system of equations, $\mathtt{x} = \mathtt{J_u}(\mathtt{u},\mathtt{z})^{-1}\mathtt{b}$, where $\mathtt{J_u}(\mathtt{u},\mathtt{z})$ is the Jacobian with respect to `u` (state variables), `b` is right hand side vector, and `x` is the solution[5,6]

3. `applyAdjointInverseJacobianState(u,z,b,x̂)`: Solves the following linear system of equations, $\hat{\mathtt{x}} = \mathtt{J_u}(\mathtt{u},\mathtt{z})^{-*}\mathtt{b}$, where $*$ denotes the adjoint, $\mathtt{J_u}(\mathtt{u},\mathtt{z})$ is the Jacobian with respect to `u` (state variables), `b` is right hand side vector, and $\hat{\mathtt{x}}$ denotes the solution[7]

4. `partialDerivativeControl(u,z,dp,output)`: Computes the application of direction `dp` to the partial derivative of the equality constraint with respect to the control variables (Jacobian with respect to the control variables), where `output` denotes $\mathtt{J_z}(\mathtt{u},\mathtt{z})\mathtt{dp}$

5. `partialDerivativeState(u,z,dp,output)`: Computes the application of direction `dp` to the partial derivative of the equality constraint with respect to the state variables (Jacobian with respect to the state variables), where `output` denotes $\mathtt{J_u}(\mathtt{u},\mathtt{z})\mathtt{dp}$

6. `adjointPartialDerivativeControl(u,z,λ,output)`: Computes the application of the Lagrange multipliers (λ) to the adjoint of the partial derivative of the equality constraint with respect to the control variables, where `output` denotes $\mathtt{J_z}(\mathtt{u},\mathtt{z})^*\lambda$

7. `adjointPartialDerivativeState(u,z,λ,output)`: Computes the application of the Lagrange multipliers to the adjoint of the partial derivative of the equality constraint with respect to the state variables, where `output` denotes $\mathtt{J_u}(\mathtt{u},\mathtt{z})^*\lambda$

8. `adjointPartialDerivativeStateState(u,z,λ,du,output)`: Computes the application of direction `du` to the adjoint of the second-order partial derivative of the equality constraint with respect to the state variables, where `output` denotes $(\mathtt{J_{uu}}(\mathtt{u},\mathtt{z})^*\lambda)\mathtt{du}$

9. `adjointPartialDerivativeControlControl(u,z,λ,dz,output)`: Computes the application of direction `dz` to the adjoint of the second-order partial derivative of the equality constraint with respect to the control variables, where `output` denotes $(\mathtt{J_{zz}}(\mathtt{u},\mathtt{z})^*\ \lambda)\mathtt{dz}$

[5] *In certain applications (nonlinear elasticity with large deformations) the state variables are necessary to compute the Jacobian operator.*

[6] *This operations is only invoke if analytical Hessian information is provided by the user.*

[7] *This operation is necessary to compute the derivative of the objective function with respect to the control variables (df/dz). Furthermore, if analytical Hessian information is provided, this operation is invoke by the algorithm during the calculation of the application of the trail step to the Hessian operator.*

10. `adjointPartialDerivativeStateControl(u,z,λ,dz,output)`: Computes the application of direction `dz` to the adjoint of the mixed partial derivative of the equality constraint with respect to the state and control variables, where `output` denotes $(\mathtt{J_{uz}}(\mathtt{u},\mathtt{z})^*\lambda)\mathtt{dz}$

11. `adjointPartialDerivativeControlState(u,z,λ,du,output)`: Computes the application of direction `du` to the adjoint of the mixed partial derivative of the equality constraint with respect to the control and state variables, where `output` denotes $(\mathtt{J_{zu}}(\mathtt{u},\mathtt{z})^*\lambda)\mathtt{du}$

The optimization algorithm relies on this interface to query information pertinent to the equality constraint throughout optimization. Thus, the implementation of this interface is mandatory[8]

3.2.2 Standard interface

3.3 Inequality Constraint

3.3.1 Component-based interface

The core inequality constraint operations required by the algorithm for a component-based interface are:

1. `bound()`: Evaluates the inequality constraint bound, `h(u,z) = value(u,z) - bound()` $\leq$ `0`

2. `value(u,z)`: Evaluates the inequality constraint given a set of state (`u`) and control (`z`) variables

3. `partialDerivativeControl(u,z,output)`: Computes the partial derivative of the inequality constraint with respect to the control variables, where `output` denotes $\partial\mathtt{h}(\mathtt{u},\mathtt{z})/\partial\mathtt{z}$

[8] *Certain derivative operators, specifically the second-order and mixed partial derivatives, might not be defined for a given application. Similar to the objective function case, users are still required to declared and defined these functionalities. However, only a* `NULL` *operator is required.*

4. `partialDerivativeState(u,z,output)`: Computes the partial derivative of the inequality constraint with respect to the state variables, where `output` denotes $\partial \mathtt{h}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}$

5. `partialDerivativeStateState(u,z,du,output)`: Computes the application of direction `du` to the second-order partial derivative of the inequality constraint with respect to the state variables, where `output` denotes $(\partial^2 \mathtt{h}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}^2)\mathtt{du}$

6. `partialDerivativeControlControl(u,z,dz,output)`: Computes the application of direction `dz` to the second-order partial derivative of the inequality constraint with respect to the control variables, where `output` denotes $(\partial^2 \mathtt{h}(\mathtt{u},\mathtt{z})/\partial \mathtt{z}^2)\mathtt{dz}$

7. `partialDerivativeStateControl(u,z,dz,output)`: Computes the application of direction `dz` to the mixed partial derivative of the inequality constraint with respect to the state and control variables, where `output` denotes $(\partial^2 \mathtt{h}(\mathtt{u},\mathtt{z})/\partial \mathtt{u}\partial \mathtt{z})\mathtt{dz}$

8. `partialDerivativeControlState(u,z,du,output)`: Computes the application of direction `du` to the mixed partial derivative of the inequality constraint with respect to the control and state variables, where `output` denotes $(\partial^2 \mathtt{h}(\mathtt{u},\mathtt{z})/\partial \mathtt{z}\partial \mathtt{u})\mathtt{du}$

The optimization algorithm relies on this interface to query information pertinent to the inequality constraint throughout optimization. Thus, the implementation of this interface is mandatory[9]

3.3.2 Standard interface

[9] *Certain derivative operators, specifically the second-order and mixed partial derivatives, might not be defined for a given application. Furthermore, for linear inequalities, all the partial derivatives with respect to the state variables are zero. Thus, similar to the objective and equality constraint cases, users are still required to declared and defined these functionalities. However, only a* `NULL` *operator is necessary.*

Chapter 4

Conclusion

4.1 Summary of Thesis Achievements

Summary.

4.2 Applications

Applications.

4.3 Future Work

Future Work.