Rapid Optimization Library

Drew Kouri  Denis Ridzal  Greg von Winckel

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525.

SAND NO. 2017-12025 PE
Outline

Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Overview of ROL

Trilinos package for large-scale optimization. Uses: optimal design, optimal control and inverse problems in engineering applications; mesh optimization; image processing.

- Modern optimization algorithms.
- Maximum HPC hardware utilization.
- Special programming interfaces for simulation-based optimization.
- Optimization under uncertainty.

- Hardened, production-ready algorithms for unconstrained, equality-constrained, inequality-constrained and nonsmooth optimization.
- Novel algorithms for optimization under uncertainty and risk-averse optimization.
- Unique capabilities for optimization-guided inexact and adaptive computations.
- Geared toward maximizing HPC hardware utilization through direct use of application data structures, memory spaces, linear solvers and nonlinear solvers.
- Special interfaces for engineering applications, for streamlined and efficient use.
- Rigorous implementation verification: finite difference and linear algebra checks.
- Hierarchical and custom (user-defined) algorithms and stopping criteria.
Application examples

**Inverse problems in acoustics/elasticity**
- Interface to the Sierra-SD ASC Integrated Code for structural dynamics
  - 1M optimization and state variables

**Estimating basal friction of ice sheets**
- Interface to Trilinos-based Albany
  - 5M optimization, 20M state variables

**Super-resolution imaging**
- GPU image processing using ArrayFire
  - 250K optimization variables, NVIDIA Tesla

**Interface to DGM, a high-order Discontinuous Galerkin code**
- 500K optimization, 2M × 5K state variables
Updates and a roadmap

2017:

A new algorithmic interface: ROLL::OptimizationProblem and ROLL::OptimizationSolver; for ease of use and maintenance.

New capabilities and improved interfaces for optimization under uncertainty and risk-averse optimization.

Expansion of the PDE-OPT Application Development Kit and test suite for PDE-constrained optimization.
Updates and a roadmap

2017:

- A new algorithmic interface: `ROL::OptimizationProblem` and `ROL::OptimizationSolver`; for ease of use and maintenance.
Updates and a roadmap

2017:

- A new algorithmic interface: `ROL::OptimizationProblem` and `ROL::OptimizationSolver`; for ease of use and maintenance.
- New capabilities and improved interfaces for optimization under uncertainty and risk-averse optimization.
Updates and a roadmap

2017:

- A new algorithmic interface: ROL::OptimizationProblem and ROL::OptimizationSolver; for ease of use and maintenance.
- New capabilities and improved interfaces for optimization under uncertainty and risk-averse optimization.
- Expansion of the PDE-OPT Application Development Kit and test suite for PDE-constrained optimization.
Updates and a roadmap

2018:

- Special interface for *transient optimization* through Tempus.
Updates and a roadmap

2018:

- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
Updates and a roadmap

2018:

- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
- Integration of ROL and Dakota, as a Dakota optimization backend.
Updates and a roadmap

2018:
- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
- Integration of ROL and Dakota, as a Dakota optimization backend.
- Integration of ROL and Pyomo, via AMPL Solver Library.
Updates and a roadmap

2018:

- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
- Integration of ROL and Dakota, as a Dakota optimization backend.
- Integration of ROL and Pyomo, via AMPL Solver Library.
- New applications: Optimal design of optical metamaterials, through topology and shape optimization with Maxwell's equations.
2018:

- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
- Integration of ROL and Dakota, as a Dakota optimization backend.
- Integration of ROL and Pyomo, via AMPL Solver Library.
- New applications: Optimal design of optical metamaterials, through topology and shape optimization with Maxwell's equations.
- Python interface (PyROL) and performance improvement through static polymorphism.
Updates and a roadmap

2018:

- Special interface for transient optimization through Tempus.
- Parallel-in-time optimization capabilities and a test suite.
- Integration of ROL and Dakota, as a Dakota optimization backend.
- Integration of ROL and Pyomo, via AMPL Solver Library.
- New applications: Optimal design of optical metamaterials, through topology and shape optimization with Maxwell's equations.
- Python interface (PyROL) and performance improvement through static polymorphism.
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Mathematical abstraction

ROL is used for the numerical solution of optimization problems

\[
\begin{align*}
\text{minimize} \quad & J(x) \\
\text{subject to} \quad & c(x) = 0 \\
& a \leq x \leq b
\end{align*}
\]

where:
- \( J : \mathcal{X} \to \mathbb{R} \) is a Fréchet differentiable functional;
- \( c : \mathcal{X} \to \mathcal{C} \) is a Fréchet differentiable operator;
- \( \mathcal{X} \) and \( \mathcal{C} \) are Banach spaces of functions; and
- \( a \leq x \leq b \) defines pointwise (componentwise) bounds on \( x \).

This abstraction is a valuable guiding principle.
Four basic problem types

**Type-U**

Unconstrained

\[
\min_{x} J(x)
\]

**Type-E**

Equality constrained

\[
\min_{x} J(x) \\
\text{subject to } c(x) = 0
\]

**Type-B**

Bound constrained

\[
\min_{x} J(x) \\
\text{subject to } a \leq x \leq b
\]

**Type-EB**

Equality + Bounds

\[
\min_{x} J(x) \\
\text{subject to } c(x) = 0 \\
a \leq x \leq b
\]

ROL uses slack variables to convert inequality constraints \(c(x) \geq 0\) to Type-EB, i.e., we minimize over \(x\) and slack variables \(s\), where \(c(x) - s = 0, s \geq 0\).
Three API components

Application programming interface

Linear algebra interface
Vector

Functional interface
Objective
BoundConstraint
Constraint
SimOpt
Middleware

Algorithmic interface
Problem/Solver
Algorithm
Step, StatusTest

Methods - Implementation of Step instances
ROL::Vector provides a generic interface for application data structures.

<table>
<thead>
<tr>
<th>Ready-made wrappers</th>
<th>ROL::Vector member functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::vector</td>
<td>dot</td>
</tr>
<tr>
<td>Epetra::MultiVector</td>
<td>set</td>
</tr>
<tr>
<td>Tpetra::MultiVector</td>
<td>plus</td>
</tr>
<tr>
<td>Thyra::VectorBase</td>
<td>norm</td>
</tr>
<tr>
<td>ArrayFire</td>
<td>scale</td>
</tr>
<tr>
<td>ROL::StdVector</td>
<td>clone</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
</tr>
<tr>
<td></td>
<td>applyUnary</td>
</tr>
<tr>
<td></td>
<td>applyBinary</td>
</tr>
<tr>
<td></td>
<td>checkVector</td>
</tr>
</tbody>
</table>

ROL::StdVector encapsulates a Teuchos::RCP to a std::vector which actually contains the data. This type is used in most of ROL's examples and tests.
Core vector operations must be implemented by derived classes.

namespace ROL {

    template<class Real>
    class Vector {
    public:

        virtual void plus( const Vector &x ) = 0; // y ← y + x
        virtual void scale( const Real alpha ) = 0; // y ← αy
        virtual Real dot( const Vector &x ) const = 0; // ⟨y, x⟩
        virtual Real norm() const = 0; // ||y||
        virtual Teuchos::RCP<Vector> clone() const = 0;

    } // class Vector

} // namespace ROL
Optional methods: May want to implement for efficiency.

```cpp
namespace ROL {

template<class Real>
class Vector {
public:

    virtual void axpy( const Real alpha, const Vector &x ); // y ← αx + y
    virtual void zero(); // y ← 0
    virtual Teuchos::RCP<Vector> basis( const int i ) const;
    virtual int dimension() const;
    virtual void set( const Vector &x ); // y ← x
    virtual const Vector & dual() const;

}; // class Vector
} // namespace ROL
```
Elementwise operations → Function evaluations on all vector elements. Needed for algorithms that handle general inequality constraints.

```cpp
namespace ROL {
    template<class Real>
    class Vector {
    public:

        using UF = Elementwise::UnaryFunction<Real>;
        using BF = Elementwise::BinaryFunction<Real>;
        using OP = Elementwise::ReductionOp<Real>;

        virtual void applyUnary(const UF &f); // y ← f(y)
        virtual void applyBinary(const BF &f,
                                  const Vector &x); // y ← f(x,y)

        // Common examples: sum, min, max
        virtual Real reduce(const OP &r) const;
    };
} // namespace ROL
```
Functional interface

**ROL::Objective Methods**

- **value** - $J(x)$
- **gradient** - $g = \nabla J(x)$
- **hessVec** - $Hv = [\nabla^2 J(x)]v$
- **update** - modify member data
- **invHessVec** - $H^{-1}v = [\nabla^2 J(x)]^{-1}v$
- **precond** - approximate $H^{-1}v$
- **dirDeriv** - $\frac{d}{dt}J(x + tv)|_{t=0}$

ROL can use finite differences to approximate derivatives.

For best performance, implement analytic derivatives.

**Tools:** checkGradient, checkHessVec, checkHessSym.

\[ \min_x J(x) \]
\[ \text{subject to } c(x) = 0 \]
\[ a \leq x \leq b \]
Functional interface

**ROL::Constraint Methods**

- value - $c(x)$
- applyJacobian - $[c'(x)]\nu$
- applyAdjointJacobian - $[c'(x)]^*\nu$
- applyAdjointHessian - $[c''(x)](\nu, \cdot)^*u$
- update - modify member data
- applyPreconditioner
- solveAugmentedSystem

ROL can use finite differences to approximate derivatives.

For best performance, implement analytic derivatives.

Tools: checkApplyJacobian, checkApplyAdjointJacobian, checkAdjointConsistencyJacobian, checkApplyAdjointHessian.

$$\min_x J(x)$$
subject to
$$c(x) = 0$$
$$a \leq x \leq b$$
The BoundConstraint class was previously abstract in ROL with implementations needed for each class derived from Vector.

Now, it is a concrete class for any vector that implements ROL's elementwise operations.

```
ROL::Bounds(x_lo, x_up);
```
Algorithmic interface

**ROL::OptimizationProblem**

\[
\min_x \ J(x) \\
\text{s.t.} \quad a \leq x \leq b \\
\quad c\varepsilon(x) = 0 \\
\quad L \leq c\mathcal{I}(x) \leq U
\]

```
ROL::OptimizationProblem(const RCP<Objective<Real>> &obj, 
const RCP<Vector<Real>> &x, 
const RCP<BoundConstraint<Real>> &bnd, 
const std::vector<RCP<Constraint<Real>>> &econ, 
const std::vector<RCP<Vector<Real>>> &emul, 
const std::vector<RCP<Constraint<Real>>> &icon, 
const std::vector<RCP<Vector<Real>>> &imul, 
const std::vector<RCP<BoundConstraint<Real>>> &ibnd);
```

**ROL::OptimizationSolver**

```
ROL::OptimizationSolver(OptimizationProblem<Real> &opt, 
Teuchos::ParameterList &parlist);
int solve(std::ostream &outStream, 
const RCP<StatusTest<Real>> &status = Teuchos::null, 
const bool combineStatus = true);
```

- Also: Fine-grained interface to **ROL:Algorithm** and **ROL:Step**.
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Algorithms - Part 1

**Type-U (unconstrained)**

- Globalization: `ROL::LineSearchStep` and `ROL::TrustRegionStep`.
- Gradient descent, quasi-Newton (limited-memory BFGS, DFP, Barzilai-Borwein), nonlinear CG (9 variants), inexact Newton (including finite difference hessVecs), Newton, with line searches and trust regions.
- Trust-region methods supporting inexact objective functions and inexact gradient evaluations. Enables *adaptive and reduced models*.

**Type-B (bound constrained)**

- Projected gradient and projected Newton methods.
- Primal-dual active set methods.
Type-E (equality constrained)
- Sequential quadratic programming (SQP) with trust regions, supporting inexact linear system solves.
- Augmented Lagrangian methods.

Type-EB (equality + bound constrained)
- Augmented Lagrangian methods.
- Moreau-Yosida regularization.
- Semismooth Newton methods.
- Interior Point SQP methods (primal, primal-dual in development).
Optimization under uncertainty

\[
\minimize_z \mathcal{H}(f(z, \xi)) \\
\minimize_z \mathcal{H}(f(S(z, \xi), z, \xi)) \quad \text{where } S(z, \xi) \text{ solves } c(u, z, \xi) = 0
\]

- Compute controls/designs that are risk-averse or robust to uncertainty in the parameters $\xi$. Here $\mathcal{H}$ is some **hazard functional**.
- Hazard functionals: Conditional value-at-risk (CVaR), Expectation (mean), Mean plus deviation, Mean plus variance, Exponential disutility, Buffered probability of exceedance (bPOE), etc.
- Incorporate sample and adaptive quadrature approaches from uncertainty quantification. Flexible sampling interface through ROL::SampleGenerator and ROL::BatchManager.
- Control inexactness and adaptivity through **trust-region** framework.
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
For the purposes of the tutorial, we designed a special ROL::StdObjective, ROL::StdConstraint, etc., interface that supports the direct use of std::vector, thereby hiding the ROL::Vector abstraction.

This interface can be used in applications if your primary (and only) data structure is std::vector, however ...

It is a terrible idea to copy data from your app's container into an std::vector, and back, just to use this interface.
Rosenbrock: Unconstrained (Type-U)

See rol/tutorial/example_unc.cpp

\[
\min_{x_0, x_1} \left\{ \alpha (x_0^2 - x_1)^2 + (x_0 - 1)^2 \right\}
\]

Here \(\alpha = 100\). Minimum value of 0 is obtained at (1, 1).
Implementing an Objective Class

Rosenbrock function and its derivatives

\[ J(x) = \alpha(x_0^2 - x_1)^2 + (x_0 - 1)^2 \]

\[ \nabla J(x)_0 = 4\alpha(x_0^2 - x_1)x_0 + 2(x_0 - 1) \]

\[ \nabla J(x)_1 = -2\alpha(x_0^2 - x_1) \]

\[ \nabla^2 J(x)v_0 = (12\alpha x_0^2 - 4\alpha x_1 + 2)v_0 - 4\alpha x_0 v_1 \]

\[ \nabla^2 J(x)v_1 = -4\alpha x_0 v_0 + 2\alpha v_1 \]
Implementing an Objective Class

```cpp
template<class Real>
class ObjectiveRosenbrock : public ROL::StdObjective<Real> {
public:
    ObjectiveRosenbrock(void) {}

    Real value(const std::vector<Real> &x, Real &tol) {
        const Real one(1), alpha(100);
        Real val = alpha * std::pow(std::pow(x[0], 2) - x[1], 2)
                   + std::pow(x[0] - one, 2);
        return val;
    }

    void gradient( std::vector<Real> &g, const std::vector<Real> &x, Real &tol ) {
        const Real one(1), two(2), alpha(100);
        g[0] = two*alpha*(std::pow(x[0], 2) - x[1]) * two*x[0] + two*(x[0]-one);
        g[1] = -two*alpha*(std::pow(x[0], 2) - x[1]);
    }

    void hessVec( std::vector<Real> &hv, const std::vector<Real> &v, const std::vector<Real> &x, Real &tol ) {
        const Real two(2), three(3), alpha(100);
        Real h11 = two*two*three*alpha*std::pow(x[0], 2) - two*two*alpha*x[1] + two;
        Real h12 = -two*two*alpha*x[0];
        Real h21 = h12;
        Real h22 = two*alpha;
        hv[0] = h11*v[0] + h12*v[1];
        hv[1] = h21*v[0] + h22*v[1];
    }
}; // class ObjectiveRosenbrock
```
Solving the problem

```cpp
Teuchos::ParameterList parlist;
parlist.sublist("Step").set("Type","Trust Region");
parlist.sublist("Step").sublist("Trust Region").set(
    "Subproblem Solver","Truncated CG");

RCP<std::vector<RealT> > x.rcp = rcp( new std::vector<RealT>(2) );
RCP<ROL::Vector<RealT> > x = rcp( new ROL::StdVector<RealT>(x.rcp) );
(*x.rcp)[0] = static_cast<RealT>(-3);
(*x.rcp)[1] = static_cast<RealT>(-4);

RCP<ROL::Objective<RealT> > obj = rcp( new ObjectiveRosenbrock<RealT>() );

ROL::OptimizationProblem<RealT> problem( obj, x );
problem.check(*outStream);

ROL::OptimizationSolver<RealT> solver( problem, parlist );
solver.solve(*outStream);

*outStream << "x_opt = [" << (*x.rcp)[0] << ", " << (*x.rcp)[1] << "]" << std::endl;
```
Solving the problem

Performing OptimizationProblem diagnostics.
Checking vector operations in optimization vector space X.

Commutativity of addition. Consistency error: >>>>>>>> 0.000000000000e+00
Associativity of addition. Consistency error: >>>>>>>> 4.440892098501e-16

Checking objective function.

<table>
<thead>
<tr>
<th>Step size</th>
<th>grad' * dir</th>
<th>FD approx</th>
<th>abs error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00000000000e-05</td>
<td>-1.28755849134e+01</td>
<td>-1.28754717782e+01</td>
<td>1.13135237813e-04</td>
</tr>
<tr>
<td>1.00000000000e-06</td>
<td>-1.28755849134e+01</td>
<td>-1.28755736046e+01</td>
<td>1.1307794347e-05</td>
</tr>
</tbody>
</table>

Step size norm(Hess * vec) norm(FD approx) norm(abs error)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>1.00000000000e-05</td>
<td>6.23835747001e+02</td>
<td>6.238380884e+02</td>
<td>7.8373039516e-03</td>
</tr>
<tr>
<td>1.00000000000e-06</td>
<td>6.23835747001e+02</td>
<td>6.23836510354e+02</td>
<td>7.83695612072e-04</td>
</tr>
</tbody>
</table>

<w, H(x)v> <v, H(x)w> abs error

-1.59963302477e+01 -1.59963302477e+01 1.24344978758e-14

Truncated CG Trust-Region Solver

<table>
<thead>
<tr>
<th>iter</th>
<th>value</th>
<th>gnorm</th>
<th>snorm</th>
<th>delta</th>
<th>iterCG</th>
<th>flagCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.691600e+04</td>
<td>1.582307e+04</td>
<td>1.269752e+00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4.693141e+03</td>
<td>4.982056e+03</td>
<td>1.269752e+00</td>
<td>3.174380e+00</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>29</td>
<td>3.574815e-06</td>
<td>7.633462e-02</td>
<td>2.940042e-02</td>
<td>6.970662e+00</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>6.533209e-10</td>
<td>1.885039e-04</td>
<td>1.766505e-03</td>
<td>1.742666e+01</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>6.533209e-10</td>
<td>1.885039e-04</td>
<td>1.766505e-03</td>
<td>1.742666e+01</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

x_opt = [1, 1]
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Many simulation-based Type-E problems have the form

\[
\min_{u,z} \quad J(u, z) \quad \text{subject to} \quad c(u, z) = 0
\]

- \(u\) denotes simulation variables (state, basic, \textit{Sim})
- \(z\) denotes optimization variables (controls, parameters, \textit{Opt})

A common Type-U reformulation, by nonlinear elimination is:

\[
\min_{z} \quad J(S(z), z) \quad \text{where} \quad u = S(z) \quad \text{solves} \quad c(u, z) = 0
\]

For these cases, the \textit{SimOpt} interface enables direct use of methods for \textbf{both} unconstrained and constrained problems.
Two formulations

Simulation-based optimization problems assume the form:

\[
\begin{align*}
\text{minimize} & \quad J(u, z) \\
\{u, z\} & \in U \times Z \\
\text{subject to} & \quad c(u, z) = 0
\end{align*}
\]

\(u\) -- state variables; \(z\) -- controls/parameters

Two ways to think about this form:

**REDUCED SPACE**

\[
\begin{align*}
\text{minimize} & \quad \mathcal{J}(z) = J(S(z), z) \\
\text{subject to} & \quad c(S(z), z) = 0
\end{align*}
\]

Here \(u = S(z)\) solves \(c(u, z) = 0\).

**FULL SPACE**

\[
\begin{align*}
\text{minimize} & \quad J(x) \\
\text{subject to} & \quad c(x) = 0
\end{align*}
\]

Here \(x = U \times Z\).
General observations

- The reduced-space form is implicitly constrained. In its simplest form: an unconstrained optimization problem.
- The full-space form is explicitly constrained. In its simplest form: an equality-constrained optimization problem.
- Methods for numerical optimization can be used, however, note ...
- Both forms are posed in function space, where $\mathcal{U}$, $\mathcal{Z}$ and $\mathcal{X}$ are Hilbert spaces or, more generally, Banach spaces.
- There is inherent smoothness in the problem formulations.
- Computer representations must exploit the problem structure:
  - vector space operations, specifically inner products and duality; and
  - derivative operators for objective and constraint functions.
- With some care, we can apply fast (gradient-based, Newton-type) optimization algorithms featuring scalable performance.
Derivatives

minimize \( J(u, z) \)
\[ \{u, z\} \in U \times Z \]

subject to \( c(u, z) = 0 \)

- First derivatives.
  - Objective gradients: \( \nabla_u J(u, z), \nabla_z J(u, z) \).
  - Constraint Jacobians: \( c_u(u, z), c_z(u, z) \).

- Second derivatives.
  - Objective Hessians: \( \nabla_{uu} J(u, z), \nabla_{uz} J(u, z), \nabla_{zu} J(u, z), \nabla_{zz} J(u, z) \).
  - Constraint Hessians: \( c_{uu}(u, z), c_{uz}(u, z), c_{zu}(u, z), c_{zz}(u, z) \).

- It is also useful to consider the Lagrangian functional,
  \( L(u, z, \lambda) = J(u, z) + \langle c(u, z), \lambda \rangle_{C, C^*} \), where \( c : U \times Z \to C \),
  and its first and second derivatives.

- Which operations do we really need?
Derivatives in the reduced space

\[ \minimize_{z \in \mathcal{Z}} \mathcal{J}(z) = J(S(z), z) \]

- Gradient via implicit differentiation of \( c(S(z), z) = 0 \):
  \[
  \nabla \mathcal{J}(z) = S_z(z)^* \nabla_u J(S(z), z) + \nabla_z J(S(z), z) \\
  = -c_z(S(z), z)^* c_u(S(z), z)^* \nabla_u J(S(z), z) + \nabla_z J(S(z), z)
  \]

- Numerical recipe:
  1. For a given \( z \), compute \( u = S(z) \) that solves \( c(u, z) = 0 \).
  2. Solve adjoint equation for \( \lambda = P(z) \): \( c_u(u, z)^* \lambda = -\nabla_u J(u, z) \).
  3. Compute \( \nabla \mathcal{J}(z) = c_z(u, z)^* \lambda + \nabla_z J(u, z) \).

- Note: \( \nabla \mathcal{J}(z) = \nabla_z L(u, z, \lambda) \), treating \( u \) and \( \lambda \) as functions of \( z \); \( u = S(z) \) and \( \lambda = P(z) \).
Derivatives in the reduced space

- **Hessian via implicit function theorem:**

  \[ \nabla^2 J(z) = \nabla_{zu} L(S(z), z, P(z)) S_z(z) + \nabla_{zz} L(S(z), z, P(z)) \\
  + \nabla_z \lambda L(S(z), z, P(z)) P_z(z) \\
  = -c_z(S(z), z)^* c_u(S(z), z)^{-*} \nabla_{uu} L(S(z), z, P(z)) c_u(S(z), z)^{-1} c_z(S(z), z) \\
  - c_z(S(z), z)^* c_u(S(z), z)^{-*} \nabla_{uz} L(S(z), z, P(z)) \\
  - \nabla_{zu} L(S(z), z, P(z)) c_u(s(z), z)^{-1} c_z(S(z), z) + \nabla_{zz} L(S(z), z, P(z)) \n\]

- **Numerical recipe for applying the Hessian to vector \( v \):**

  1. For a given \( z \), compute \( u = S(z) \) that solves \( c(u, z) = 0 \).
  2. Solve adjoint equation for \( \lambda = P(z) \): \( c_u(u, z)^* \lambda = -\nabla_u J(u, z) \).
  3. Solve **linearized state equation:** \( c_u(u, z)w = c_z(u, z)v \).
  4. Solve **adjoint equation:** \( c_u(u, z)^* p = \nabla_{uu} L(u, z, \lambda) w - \nabla_{uz} L(u, z, \lambda) v \).
  5. Compute \( \nabla^2 J(z)v = c_z(u, z)^* p - \nabla_{zu} L(u, z, \lambda) w + \nabla_{zz} L(u, z, \lambda) v \).

- **Note:** Evaluating \( \nabla_{zu} L(u, z, \lambda) w \) includes \( c_{zu}(u, z)(w, \cdot)^* \lambda \). We also need \( c_{zz}(u, z)(v, \cdot)^* \lambda \), \( c_{uz}(u, z)(v, \cdot)^* \lambda \) and \( c_{uu}(u, z)(w, \cdot)^* \lambda \).
Interface requirements for (matrix-free) SimOpt

**REDUCED SPACE**

\[
\minimize_{z \in \mathcal{Z}} \quad J(z) = J(S(z), z)
\]

**FULL SPACE**

\[
\minimize_{\{u,z\} \in \mathcal{U} \times \mathcal{Z}} \quad J(u, z)
\]

subject to \( c(u, z) = 0 \)

- **First derivatives.**
  - Objective gradients: \( \nabla_u J(u, z), \nabla_z J(u, z) \).
  - Action of constraint Jacobians: \( c_u(u, z)v, c_z(u, z)v \).
  - Linearized state and adjoint solves: \( c_u(u, z)^{-1}v, c_u(u, z)^{-*}v \).

- **Second derivatives.**
  - Action of objective Hessians:
    - \( \nabla_{uu} J(u, z)v, \nabla_{uz} J(u, z)v, \nabla_{zu} J(u, z)v, \nabla_{zz} J(u, z)v \).
  - Action of the adjoints of constraint Hessians:
    - \( c_{uu}(u, z)(v, .)^*\lambda, c_{uz}(u, z)(v, .)^*\lambda, c_{zu}(u, z)(v, .)^*\lambda, c_{zz}(u, z)(v, .)^*\lambda \).

- Solution operator \( S(z) = u \), i.e., nonlinear solve.
Other observations

**REduced SPACE**

\[
\begin{align*}
\text{minimize} \quad & J(z) = J(S(z), z) \\
\text{subject to} \quad & c(u, z) = 0
\end{align*}
\]

**FULL SPACE**

\[
\begin{align*}
\text{minimize} \quad & J(u, z) \\
\text{subject to} \quad & c(u, z) = 0
\end{align*}
\]

- The full-space problem may be well-posed even if the solution operator \( S(z) \) is not well-defined, e.g., when the constraint \( c(u, z) = 0 \) has no or multiple solutions \( u \) for a given \( z \).
- Derivative computations for the reduced-space problem involve linear and nonlinear solves, \( c_u(u, z)^{-1}v, c_u(u, z)^{-*}v \) and \( u = S(z) \).
- Derivative computations for the full-space problem require no solves.
- Reduced-space methods eliminate the state variables, \( u \), and maintain constraint feasibility throughout the iteration.
- Full-space methods expose \( u \) and \( z \) as optimization variables, i.e., they do not have to maintain feasibility throughout the iteration.
The SimOpt interface
Middleware for engineering optimization

**ROL::Objective_SimOpt**
- value(u,z)
- gradient_1(g,u,z)
- gradient_2(g,u,z)
- hessVec_11(hv,v,u,z)
- hessVec_12(hv,v,u,z)
- hessVec_21(hv,v,u,z)
- hessVec_22(hv,v,u,z)

**Recall**
- 1 = Sim = u
- 2 = Opt = z

**ROL::Constraint_SimOpt**
- value(u,z)
- applyJacobian_1(jv,v,u,z)
- applyJacobian_2(jv,v,u,z)
- applyInverseJacobian_1(ijv,v,u,z)
- applyAdjointJacobian_1(ajv,v,u,z)
- applyAdjointJacobian_2(ajv,v,u,z)
- applyInverseAdjointJacobian_1(iajv,v,u,z)
- applyAdjointHessian_11(ahwv,w,v,u,z)
- applyAdjointHessian_12(ahwv,w,v,u,z)
- applyAdjointHessian_21(ahwv,w,v,u,z)
- applyAdjointHessian_22(ahwv,w,v,u,z)
- solve(u,z)
SimOpt: Benefits

- Streamlined modular implementation for a very large class of engineering optimization problems.

- Implementation verification through a variety of ROL tests:
  - Finite difference checks with high granularity.
  - Consistency checks for operator inverses and adjoints.

- Access to all optimization methods through a single interface.

- Enables future ROL interfaces for advanced solution checkpointing and restarting, closer integration with time integration libraries, etc.
Overview

Application programming interface

Algorithms

Tutorial 1

Simulation-based optimization

Tutorial 2
Model Rocket Control Problem: Fuel Efficiency
Tsiolkovsky Equation

$$mdv = -v_e dm - mg dt$$
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

Mass, Velocity, Gravity
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - m_f \int_0^t z(s) \, ds \]

Changing mass
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - m_f \int_0^t z(s) \, ds \]

Total Mass
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - m_f \int_0^t z(s) \, ds \]

Fuel Mass
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - m_f \int_{0}^{t} z(s) \, ds \]

Burn rate
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - \frac{m_f}{T} \, t \]

Constant burn rate
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \frac{dv}{dt} = -v_e dm - mg dt \]

\[ m(t) = m_T - \frac{m_f}{T} t \]

Exact velocity

\[ \frac{dh}{dt} = u(t) = v_e \ln \left( \frac{m_T}{m(t)} \right) - gt \]
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \frac{dv}{dt} = -v_e dm - mg dt \]

\[ m(t) = m_T - \frac{m_f}{T} t \]

Exact velocity

\[ \frac{dh}{dt} = u(t) = v_e \ln \left( \frac{m_T}{m(t)} \right) - gt \]

Final altitude \((k = m_f / m_r)\)

\[ h(T) = v_e T \left( 1 + \frac{\ln(1 + k)}{k} \right) - \frac{1}{2} g T^2 \]
Model Rocket Control Problem: Fuel Efficiency

Tsiolkovsky Equation

\[ m \, dv = -v_e \, dm - mg \, dt \]

\[ m(t) = m_T - \frac{m_f}{T} \, t \]

Exact velocity

\[ \frac{dh}{dt} = u(t) = v_e \ln \left( \frac{m_T}{m(t)} \right) - gt \]

Can we find \( z(t) \) that attains the same \( u(T^*) \) with minimal fuel use?
Optimal Fuel Burn Rate

Minimum fuel consumption objective

$$\min f(u, z) = \frac{1}{2} \left( h(T) - \int_0^T u(t) \, dt \right)^2 + \frac{\mu}{2} \int_0^T z^2(t) \, dt$$

Subject to the ODE constraint

$$\dot{u} = v_e \frac{z(t)}{m(t)} - g, \quad u(0) = 0$$

Discretized equations

$$u_k = u_{k-1} - v_e [\ln m_k - \ln m_{k-1}] - d \Delta t, \quad u_0 = 0$$

$$m_k = m_{k-1} - \Delta t z_k, \quad m_0 = m_T$$
#pragma once

#include "ROL_StdVector.hpp"
#include "ROL_Objective_SimOpt.hpp"
#include "ROL_Constraint_SimOpt.hpp"

namespace Rocket {

class Objective : public ROL::Objective_SimOpt<double> {
private:
    using V = ROL::Vector<double>;

    int N;
    double T, dt, mt;
    double htarg, alpha;
    const Teuchos::RCP<const V> w; // Trapezoidal weights

public:

    Objective( int N_, double T_, double mt_, double htarg_,
              double alpha_, const Teuchos::RCP<const V>& w_ ) :
        N(N_), T(T_), dt(T/N), mt(mt_), htarg(htarg_),
        alpha(alpha_), w(w_) {}
    // More functions to follow
};
} // namespace Rocket
Implementing the Objective - Rocket.hpp

double value( const V& u, const V& z, double& tol ) {
    return 0.5*std::pow(htarg-w->dot(u), 2) + 0.5*alpha*dt*z.dot(z);
}

void gradient_1( V& g, const V& u, const V& z, double& tol ) {
    g.set(*w); g.scale(w->dot(u)-htarg);
}

void gradient_2( V& g, const V& u, const V& z, double& tol ) {
    g.set(z); g.scale(alpha*dt);
}
void hessVec_11( V& hv, const V& v, const V& u, const V& z, double& tol ) {
    hv.set(*w); hv.scale(w->dot(v));
}

void hessVec_12( V& hv, const V& v, const V& u, const V& z, double& tol ) {
    hv.zero();
}

void hessVec_21( V& hv, const V& v, const V& u, const V& z, double& tol ) {
    hv.zero();
}

void hessVec_22( V& hv, const V& v, const V& u, const V& z, double& tol ) {
    hv.set(v); hv.scale(alpha*dt);
}
class Constraint : public ROL::Constraint_SimOpt<double> {
private:
    using V = ROL::Vector<double>;

    int N;
    double T, dt, gdt, mt, mf, ve;
    std::vector<double> mass;

public:

    Constraint(int N_, double T_, double mt_,
               double mf_, double ve_, double g_ ) : N(N_), T(T_),
          dt(T/N), gdt(g_*dt), mt(mt_),
          mf(mf_), ve(ve_), mass(N) {
        mass[0] = mt;
    }
    // More functions to follow
};
void update_2( const V& z, bool flag = true, int iter = -1 ) {
  auto& zs = getVector(z);

  mass[0] = mt - dt*zs[0];
  for( int i=1; i<N; ++i )
    mass[i] = mass[i-1] - dt*zs[i];
}

void solve( V& c, V& u, const V& z, double& tol ) {
  auto& us = getVector(u);
  us[0] = -ve*std::log(mass[0]/mt) - gdt;
  for( int i=1; i<N; ++i )
    us[i] = us[i-1] - ve*std::log(mass[i]/mass[i-1]) - gdt;
  value(c,u,z,tol);
}
void value( V& c, const V& u, const V&z, double &tol ) {
    auto& cs = getVector(c); auto& us = getVector(u);
    cs[0] = us[0] + ve*std::log(mass[0]/mt) + gdt;
    for( int i=1; i<N; ++i )
        cs[i] = us[i] - us[i-1] +
            ve*std::log(mass[i]/mass[i-1]) + gdt;
}

void applyJacobian_1( V& jv, const V& v, const V& u,
               const V& z, double& tol ) {
    auto& jvs = getVector(jv); auto& vs = getVector(v);
    jvs[0] = vs[0];
    for( int i=1; i<N; ++i ) jvs[i] = vs[i] - vs[i-1];
}

void applyAdjointJacobian_1( V& ajv, const V& v, const V& u,
               const V& z, double& tol ) {
    auto& ajvs = getVector(ajv); auto& vs = getVector(v);
    ajvs[N-1] = vs[N-1];
    for( int i=N-2; i>=0; --i ) ajvs[i] = vs[i] - vs[i+1];
Implementing the Constraint - Rocket.hpp

```cpp
void applyAdjointJacobian_1(V& ajv, const V& v, const V& u,
                            const V& z, double& tol ) {

    auto& ajvs = getVector(ajv);  auto& vs = getVector(v);
    ajvs[N-1] = vs[N-1];
    for( int i=N-2; i>=0; --i ) ajvs[i] = vs[i] - vs[i+1];
}

void applyInverseJacobian_1(V& ijv, const V& v, const V& u,
                            const V& z, double &tol ) {

    auto& ijvs = getVector(ijv);  auto& vs = getVector(v);
    ijvs[0] = vs[0];
    for( int i=1; i<N; ++i ) ijvs[i] = ijvs[i-1] + vs[i];
}

void applyInverseAdjointJacobian_1(V& ijv, const V& v, const V& u,
                                     const V& z, double &tol ) {

    auto& ijvs = getVector(ijv);  auto& vs = getVector(v);
    ijvs[N-1] = vs[N-1];
    for( int i=N-2; i>=0; --i ) ijvs[i] = ijvs[i+1] + vs[i];
}
```
Implementing the Constraint - Rocket.hpp

```cpp
void applyJacobian_2( V& jv, const V& v, const V& u,
                   const V& z, double& tol ) {
    auto& jvs = getVector(jv);  auto& vs = getVector(v);
    double q{-ve*dt*vs[0]};
    jvs[0] = q/mass[0];
    for( int i=1; i<N; ++i ) {
        jvs[i] = -q/mass[i-1];  q -= ve*dt*vs[i];
        jvs[i] += q/mass[i];
    }
}

void applyAdjointJacobian_2( V& ajv, const V& v, const V& u,
                           const V& z, double& tol ) {
    auto& ajvs = getVector(ajv);  auto& vs = getVector(v);
    ajvs[N-1] = -ve*dt*vs[N-1]/mass[N-1];
    for( int i=N-2; i>=0; --i )
        ajvs[i] = ajvs[i+1]-ve*dt*(vs[i]-vs[i+1])/mass[i];
}
```
Helper functions - Rocket.hpp

Used to access vector elements for the application code

```cpp
std::vector<double>& getVector( ROL::Vector<double>& x ) {
    return *(Teuchos::dyn_cast<ROL::StdVector<double>>(x).getVector());
}

const std::vector<double>& getVector( const ROL::Vector<double>& x ) {
    return *(Teuchos::dyn_cast<const ROL::StdVector<double>>(x).getVector());
}
```
#include "Rocket.hpp"
#include "ROL_OptimizationSolver.hpp"
#include "ROL_Reduced_Objective_SimOpt.hpp"
#include "Teuchos_XMLParameterListHelpers.hpp"
#include <iostream>

int main( int argc, char* argv[] ) {

    using Teuchos::rcp;
    using vector = std::vector<double>;

    auto parlist = rcp( new Teuchos::ParameterList() );
    Teuchos::updateParametersFromXmlFile("Rocket.xml",parlist.ptr());

    int N = parlist->get("Time Steps", 100);
    double T = parlist->get("Final Time", 20.0);
    double g = parlist->get("Gravity Constant", 9.8);
    double mr = parlist->get("Rocket Mass", 20.0);
    double mf = parlist->get("Fuel Mass", 100.0);
    double mu = parlist->get("Mass Penalty", 0.1);
    double ve = parlist->get("Exhaust Velocity", 1000.0);
    double mt = mf + mr; // Total mass
    double dt = T / N; // Time step

    // More to follow
}
// Simulation variable
auto u_rcp = rcp( new vector(N) );
auto u = rcp( new ROL::StdVector<double>(u_rcp) );
auto l = u->dual().clone();

// Optimization variable
auto z_rcp = rcp( new vector(N,mf/T) );
auto z = rcp( new ROL::StdVector<double>(z_rcp) );

// Lagrange multiplier
auto l = u->dual().clone();

// Trapezoidal weights
auto w_rcp = rcp( new vector(N,dt) );
(*w_rcp)[0] *= 0.5; (*w_rcp)[N-1] *= 0.5;
auto w = rcp( new ROL::StdVector<double>(w_rcp) );

// Piecewise constant weights
auto e_rcp = rcp( new vector(N,dt) );
auto e = rcp( new ROL::StdVector<double>(e_rcp) );

auto con = rcp( new Rocket::Constraint( N, T, mt, mf, ve, g ) );
double tol = 1.e-7; // Needed for solve

// Compute solution for constant burn rate
con->update_2(*z);
con->solve(*l, *u, *z, tol);
// u->print(std::cout);
double htarg = w->dot(*u); // Final height

auto obj = rcp( new Rocket::Objective( N, T, mt, htarg, mu, w ) );
auto robj = rcp( new ROL::Reduced_Objective_SimOpt<double>( obj, con, u, z, l ) );

// Full space problem
// auto x = Teuchos::rcp( new ROL::Vector_SimOpt<double>(u,z) );
// ROL::OptimizationProblem<double> opt( obj, x, con, l );
ROL::OptimizationProblem<double> opt( robj, z );
ROL::OptimizationSolver<double> solver( opt, *parlist );
solver.solve( std::cout );
con->update_2(*z);
con->solve(*l, *u, *z, tol);
// u->print(std::cout);
std::cout << "target height = " << htarg <<
" , actual = " << w->dot(*u) << std::endl;
std::cout << "Initial fuel mass = " << mf << std::endl;
std::cout << "Remaining fuel mass = " << mf-e->dot(*z) << std::endl;
Solving the problem

Truncated CG Trust-Region Solver

<table>
<thead>
<tr>
<th>iter</th>
<th>value</th>
<th>gnorm</th>
<th>#fval</th>
<th>#grad</th>
<th>tr_flag</th>
<th>iterCG</th>
<th>flagCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.3333333e+03</td>
<td>8.0000000e+02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.3333333e+03</td>
<td>8.0000000e+02</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5.183948e+03</td>
<td>4.2877777e+04</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5.015379e+03</td>
<td>4.138002e+04</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4.811909e+03</td>
<td>3.575303e+04</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4.700787e+03</td>
<td>2.812899e+04</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

...  

| 18   | 4.401047e+03       | 1.748365e+01    | 19    | 17    | 0       | 4      | 0      |
| 19   | 4.401047e+03       | 3.689599e-04    | 20    | 18    | 0       | 2      | 0      |
| 20   | 4.401047e+03       | 1.141414e-07    | 21    | 19    | 0       | 2      | 0      |
| 21   | 4.401047e+03       | 9.400946e-08    | 22    | 20    | 0       | 4      | 0      |
| 22   | 4.401047e+03       | 2.214017e-08    | 23    | 21    | 0       | 1      | 0      |

target height = 18219.4, actual = 18219.2
Initial fuel mass = 80
Remaining fuel mass = 14.7111
Optimal Burn Rate

![Graph of Optimal Burn Rate](image)

- **Altitude (km)** vs **Time (s)**
- **Burn rate (kg/s)** vs **Time (s)**

**Lines:**
- Blue: constant
- Orange: optimal