KONA Documentation

Release 0.1

Dr. Jason Hicken

Jul 19, 2017

Contents

Kona is a library for nonlinear constrained optimization. It was designed primarily for partial-differential-equation (PDE) governed optimization problems; however it is suitable for any (sufficiently smooth) problem where the objective function and/or constraints require the solution of a computational expensive state equation. Kona is also useful for developing new optimization algorithms for PDE-governed optimization as a result of its abstracted vector and matrix implementations.

Contents:

CHAPTER 1

Quick Start Guide

Kona can be installed just like any other Python module.

```
pip install -e .
```

Below is a simple example script that performs gradient-based optimization on a multidimensional Rosenbrock problem using the reduced-space Newton-Krylov (RSNK) algorithm.

```
import kona

# initialize the problem with the design space size
num_design = 2
solver = kona.examples.Rosenbrock(num_design)

# get the optimization algorithm handle -- do not initialize
algorithm = kona.algorithms.UnconstrainedRSNK

# options dictionary -- we only need convergence tolerance for now
optns = {
    'opt_tol' : 1e-12,
}

# initialize the optimization controller
optimizer = kona.Optimizer(solver, algorithm, optns)

# run the optimization
optimizer.solve()

# print solution
print solver.curr_design
```

The above optimization run will produce a `kona_hist.dat` file tracking convergence norms across non-linear iterations.

CHAPTER 2

Installation - macOS

In order to maintain system integrity on macOS, the native Python distribution should **not** be used for development.

Instead, it is highly recommended that you install [Miniconda](https://conda.io/miniconda.html) into your home directory and use this new Python distribution to operate and develop with Kona.

To install Miniconda, first download [the official installation script](https://repo.continuum.io/miniconda/Miniconda2-latest-MacOSX-x86_64.sh). Then run:

```
bash Miniconda2-latest-MacOSX-x86_64.sh
```

Proceed through the dialogue to install Miniconda's Python 2.7 distribution to your home directory. This will also update your *PATH* to make available the newly installed Python executables.

Once installed, grab essential packages using Miniconda's own package manager:

```
conda install numpy scipy nose pip
```

Finally we can install Kona just like any other Python module:

```
pip install -e .
```

CHAPTER 3

Implementation Overview

An important aspect of Kona's implementation is that it makes no assumptions regarding the dimension, type, or parallelization of the state variables. In other words, state variables are assumed to exist in an abstract vector space.

For ease of use, Kona provides a default NumPy implementation for this vector space. For high-performance applications, where variables are typically distributed across multiple processes, the user must implement the storage and linear algebra operations for the vector space. This model allows Kona to be used in a variety of parallel environments, because it remains agnostic to how the user defines, stores, and manipulates the vectors.

However this abstraction only exists for the state-space. For design variables and constraints, it is assumed that the vector spaces never become too large for a single process. Consequently, design and constraint vector spaces are locked into the default NumPy implementation. This allows Kona to exploit various explicit-algebra tools that significantly impove the functionality and efficiency of its optimization algorithms.

Additionally, Kona separates optimization algorithms from the underlying PDE solver such that any new optimization algorithm can be implemented in Kona using Kona's own abstracted vector classes. This allows for rapid development and testing of new algorithms independently from the PDE solver, and guarantees that any solver that has already been integrated with Kona will work correctly underneath any new algorithm that may be added in the future.

CHAPTER 4

History and References

An older version of Kona written in C++ can be found here.

The project now lives on GitHub and has been converted to Python.

A conference paper on Kona can be found *here <http://arc.aiaa.org/doi/abs/10.2514/6.2016-1422)>*.

CHAPTER 5

Acknowledgements

This work is supported by the National Science Foundation under Grant No. 1332819, and the National Aeronautics and Space Administration under Grant No. NNX14AC73A.

CHAPTER 6

API Details

kona.Optimizer

class `kona.`**`Optimizer`**(*solver*, *algorithm*, *optns=None*)

Bases: `object`

This is a top-level optimization controller. It is intended to be the primary means by which a user interacts with Kona.

Variables

- **`_memory`** (`KonaMemory`) – All-knowing Kona memory controller.
- **`_algorithm`** (*`OptimizationAlgorithm`*) – Optimization algorithm object.

Parameters

- **`solver`** (`UserSolver`) –
- **`algorithm`** (*`OptimizationAlgorithm`*) –
- **`optns`** (*`dict, optional`*) –

`set_design_bounds`(*lower=None*, *upper=None*)

Define lower and upper design bounds.

Parameters

- **`lower`** (*`int`*) – Lower bound for design variables.
- **`upper`** (*`int`*) – Upper bound for design variables.

`solve`(*print_opts=False*)

kona.options

exception `kona.options.`**`BadKonaOption`**(*optns*, **keys*)

Bases: `exceptions.Exception`

Special exception class for identifying bad Kona configuration options.

Parameters

- **`optns`** (*`dict`*) – Options dictionary containing the bad configuration.
- **`*keys`** (*`string`*) – Hierarchy of dictionary keys identifying the bad configuration.

`kona.options.`**`get_opt`**(*optns*, *default*, **keys*)

Utility function to make it easier to work with nested options dictionaries.

Parameters

- **`optns`** (*`dict`*) – Nested dictionary.
- **`default`** (*`Unknown`*) – Value to return of the dictionary is empty.
- **`*keys`** (*`string`*) – Keys from which value will be pulled

Returns Dictionary value corresponding to given hierarchy of keys.

Return type Unknown

kona.linalg

Subpackages

kona.linalg.vectors

Submodules

kona.linalg.vectors.common

Subclasses

KonaVector (base class)

class `kona.linalg.vectors.common.`**`KonaVector`**(*memory_obj*, *user_vector=None*)

Bases: `object`

An abstract vector class connected to the Kona memory, containing a common set of algebraic member functions. Allows Kona to operate on data spaces allocated by the user.

Parameters

- **`memory_obj`** (`KonaMemory`) – Pointer to the Kona user memory.
- **`user_vector`** (`BaseVector`) – User defined vector object that contains data and operations on data.

Variables

- **`_memory`** (*`UserMemory`*) – Pointer to the Kona user memory.

- **base** (BaseVector) – User defined vector object that contains data and operations on data.

divide_by (*val*)

Used as the division operator.

Divides the vector by the given scalar value.

Parameters **value** (*float*) – Vector to be added.

equals (*val*)

Used as the assignment operator.

If val is a scalar, all vector elements are set to the scalar value.

If val is a vector, the two vectors are set equal.

Parameters **val** (*float or* KonaVector) – Right hand side term for assignment.

equals_ax_p_by (*a*, *X*, *b*, *Y*)

Performs the scaled summation operation, `a*X + b*Y`, between two vectors, and stores the result in place.

Parameters

- **a** (*float*) – Scalar coefficient for `X`.
- **X** (KonaVector) – Vector for the operation.
- **b** (*float*) – Scalar coefficient for `Y`.
- **Y** (KonaVector) – Vector for the operation.

exp (*vector*)

Performs an element-wise exponential operation on the given vector and stores the result in-place.

Parameters **vector** (KonaVector) – Vector for the operation.

infty

Computes the infinity norm of the vector

Returns Infinity norm.

Return type float

inner (*vector*)

Computes an inner product with another vector.

Returns Inner product.

Return type float

log (*vector*)

Performs an element-wise natural log operation on the given vector and stores the result in-place.

Parameters **vector** (KonaVector) – Vector for the operation.

minus (*vector*)

Used as the subtraction operator.

Subtracts the incoming vector from the current vector in place.

Parameters **vector** (KonaVector) – Vector to be subtracted.

norm2

Computes the L2 (Euclidian) norm of the vector.

Returns L2 norm.

Return type float

`plus`(*vector*)
Used as the addition operator.

Adds the incoming vector to the current vector in place.

Parameters **`vector`** (`KonaVector`) – Vector to be added.

`pow`(*power*)
Performs an element-wise power operation in-place.

Parameters **`power`** (*`float`*) –

`times`(*factor*)
Used as the multiplication operator.

Can multiply both by scalars or element-wise by vectors.

Parameters **`factor`** (*`float or`* `KonaVector`) – Scalar or vector-valued multiplication factor.

DesignVector, StateVector and DualVector

class `kona.linalg.vectors.common.`**`DesignVector`**(*memory_obj*, *user_vector=None*)
Bases: *`kona.linalg.vectors.common.KonaVector`*

Derived from the base abstracted vector. Contains member functions specific to design vectors.

`convert_to_dual`(*dual_vector*)
Copy target state variables from the design vector to the given dual vector.

Parameters **`dual_vector`** (`DualVectorEQ` *`or`* `CompositeDualVector`) – Target for the vector space conversion.

`enforce_bounds`()
Element-wise enforcement of design bounds.

`equals_init_design`()
Sets this vector equal to the initial design point.

`equals_lagrangian_total_gradient`(*at_primal*, *at_state*, *at_dual*, *at_adjoint*, *obj_scale=1.0*, *cnstr_scale=1.0*)
Computes in-place the total derivative of the Lagrangian.

Parameters

- **`at_primal`** (`DesignVector` *`or`* `CompositePrimalVector`) – Current primal point.
- **`at_state`** (`StateVector`) – Current state point.
- **`at_dual`** (`DualVectorEQ`*`,`* `DualVectorINEQ` *`or`* `CompositeDualVector`) – Current lagrange multipliers.
- **`at_adjoint`** (`StateVector`) – Current adjoint variables for the Lagrangian (rhs = -dL/dU)
- **`obj_scale`** (*`float, optional`*) – Scaling for the objective function.
- **`cnstr_scale`** (*`float, optional`*) – Scaling for the constraints.

`equals_objective_partial`(*at_primal*, *at_state*, *scale=1.0*)
Computes in-place the partial derivative of the objective function with respect to design variables.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **scale** (*float, optional*) – Scaling for the objective function.

equals_total_gradient(*at_primal*, *at_state*, *at_adjoint*, *scale=1.0*)
Computes in-place the total derivative of the objective function.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **at_adjoint** (StateVector) – Current adjoint variables.
- **scale** (*float, optional*) – Scaling for the objective function.

restrict_to_design()
Set target state variables to zero, leaving design variables untouched.

Used only for IDF problems.

restrict_to_target()
Set design variables to zero, leaving target state variables untouched.

Used only for IDF problems.

class kona.linalg.vectors.common.**StateVector**(*memory_obj*, *user_vector=None*)
Bases: *kona.linalg.vectors.common.KonaVector*

Derived from the base abstracted vector. Contains member functions specific to state vectors.

equals_constraint_adjoint(*at_primal*, *at_state*, *at_dual*, *state_work*, *scale=1.0*)
Computes in-place the adjoint variables for the constraint terms in the Lagrangian.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **state_work** (StateVector) – Temporary work vector of State type.
- **scale** (*float, optional*) – Scaling for the constraints.

equals_lagrangian_adjoint(*at_kkt*, *at_state*, *state_work*, *obj_scale=1.0*, *cnstr_scale=1.0*)
Computes in-place the adjoint variables for the augmented Lagrangian, linearized at the given KKT vector and state points.

Parameters

- **at_kkt** (ReducedKKTVector) – Current KKT point.
- **at_state** (StateVector) – Current state point.
- **adj_work** (StateVector) – Temporary work vector of State type.
- **state_work** (StateVector) – Temporary work vector of State type.
- **obj_scale** (*float, optional*) – Scaling for the objective function.

- **cnstr_scale** (*float, optional*) – Scaling for the constraints.

equals_objective_adjoint (*at_primal*, *at_state*, *state_work*, *scale=1.0*)
Computes in-place the adjoint variables for the objective function, linearized at the given primal and state points.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **state_work** (StateVector) – Temporary work vector of State type.
- **scale** (*float, optional*) – Scaling for the objective function.

equals_objective_partial (*at_primal*, *at_state*, *scale=1.0*)
Computes in-place the partial derivative of the objective function with respect to state variables.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **scale** (*float, optional*) – Scaling for the objective function.

equals_primal_solution (*at_primal*)
Performs a non-linear system solution at the given primal point and stores the result in-place.

Parameters **at_primal** (DesignVector) – Current primal point.

equals_residual (*at_primal*, *at_state*)
Computes in-place the system residual vector.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.

class kona.linalg.vectors.common.**DualVectorEQ** (*memory_obj*, *user_vector=None*)
Bases: kona.linalg.vectors.common.DualVector

convert_to_design (*primal_vector*)
Copy target state variables from the dual vector into the given design vector.

Parameters **design_vector** (DesignVector) – Source vector for target state variable data.

equals_constraints (*at_primal*, *at_state*, *scale=1.0*)
Evaluate all equality constraints at the given primal and state points, and store the result in-place.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **scale** (*float, optional*) – Scaling for the constraints.

restrict_to_idf()

Set regular dual terms to zero, leaving IDF constraint terms untouched.

Used only for IDF problems.

restrict_to_regular()

Set IDF constraints terms to zero, leaving regular dual terms untouched.

Used only for IDF problems.

class kona.linalg.vectors.common.**DualVectorINEQ**(*memory_obj*, *user_vector=None*)

Bases: kona.linalg.vectors.common.DualVector

equals_constraints(*at_primal*, *at_state*, *scale=1.0*)

Evaluate all in-equality constraints at the given primal and state points, and store the result in-place.

Parameters

- **at_primal** (DesignVector *or* CompositePrimalVector) – Current primal point.
- **at_state** (StateVector) – Current state point.
- **scale** (*float, optional*) – Scaling for the constraints.

kona.linalg.vectors.composite

Subclasses

CompositeVector (base class)

class kona.linalg.vectors.composite.**CompositeVector**(*vectors*)

Bases: object

Base class shell for all composite vectors.

divide_by(*value*)

Used as the division operator.

Divides the vector by the given scalar value.

Parameters value (*float*) – Vector to be added.

equals(*rhs*)

Used as the assignment operator.

If val is a scalar, all vector elements are set to the scalar value.

If val is a vector, the two vectors are set equal.

Parameters rhs (*float or* CompositeVector) – Right hand side term for assignment.

equals_ax_p_by(*a*, *x*, *b*, *y*)

Performs a full a*X + b*Y operation between two vectors, and stores the result in place.

Parameters

- **b** (*a,*) – Coefficients for the operation.
- **y** (*x,*) – Vectors for the operation

exp(*vector*)

Computes the element-wise exponential of the given vector and stores it in place.

Parameters **vector** (CompositeVector) –

infty
Infinity norm of the composite vector.

Returns float

Return type Infinity norm.

inner (*vector*)
Computes an inner product with another vector.

Returns float

Return type Inner product.

log (*vector*)
Computes the element-wise natural log of the given vector and stores it in place.

Parameters **vector** (CompositeVector) –

minus (*vector*)
Used as the subtraction operator.

Subtracts the incoming vector from the current vector in place.

Parameters **vector** (CompositeVector) – Vector to be subtracted.

norm2
Computes the L2 norm of the vector.

Returns float

Return type L2 norm.

plus (*vector*)
Used as the addition operator.

Adds the incoming vector to the current vector in place.

Parameters **vector** (CompositeVector) – Vector to be added.

pow (*power*)
Computes the element-wise power of the in-place vector.

Parameters **power** (*float*) –

times (*factor*)
Used as the multiplication operator.

Can multiply with scalars or element-wise with vectors.

Parameters **factor** (*float or* CompositeVector) – Scalar or vector-valued multiplication factor.

CompositePrimalVector

class kona.linalg.vectors.composite.**CompositePrimalVector** (*primal_vec*, *dual_ineq*)
Bases: *kona.linalg.vectors.composite.CompositeVector*

A composite vector representing a combined design and slack vectors..

Parameters

- **_memory** (KonaMemory) – All-knowing Kona memory manager.

- **`design`** (`DesignVector`) – Design component of the composite vector.
- **`slack`** (`DualVectorINEQ`) – Slack components of the composite vector.

`convert_to_dual`(*dual_vector*)

`equals_init_design`()

`equals_lagrangian_total_gradient`(*at_primal*, *at_state*, *at_dual*, *at_adjoint*, *obj_scale=1.0*, *cnstr_scale=1.0*)

Computes the total primal derivative of the Lagrangian.

In this case, the primal derivative includes the slack derivative.

$$\nabla_{primal}\mathcal{L} = \begin{bmatrix} \nabla_x f(x,u) + \nabla_x c_{eq}(x,u)^T \lambda_{eq} + \nabla_x c_{inq}(x,u)^T \lambda_{ineq} \\ {}^{-1}e - \lambda_{ineq} \end{bmatrix}$$

Parameters

- **`at_primal`** (`CompositePrimalVector`) – The design/slack vector at which the derivative is computed.
- **`at_state`** (`StateVector`) – State variables at which the derivative is computed.
- **`at_dual`** (*`DualVector`*) – Lagrange multipliers at which the derivative is computed.
- **`at_adjoint`** (`StateVector`) – Pre-computed adjoint variables for the Lagrangian.
- **`obj_scale`** (*`float, optional`*) – Scaling for the objective function.
- **`cnstr_scale`** (*`float, optional`*) – Scaling for the constraints.

`init_slack` = **1.0**

`restrict_to_design`()

`restrict_to_target`()

CompositeDualVector

class `kona.linalg.vectors.composite.`**`CompositeDualVector`**(*dual_eq*, *dual_ineq*)

Bases: *`kona.linalg.vectors.composite.CompositeVector`*

A composite vector representing a combined equality and inequality constraints.

Parameters

- **`_memory`** (`KonaMemory`) – All-knowing Kona memory manager.
- **`eq`** (`DualVectorEQ`) – Equality constraints.
- **`ineq`** (`DualVectorINEQ`) – Inequality Constraints

`convert_to_design`(*primal_vector*)

`equals_constraints`(*at_primal*, *at_state*, *scale=1.0*)

Evaluate equality and inequality constraints in-place.

Parameters

- **`at_primal`** (`DesignVector` *`or`* `CompositePrimalVector`) – Primal evaluation point.
- **`at_state`** (`StateVector`) – State evaluation point.
- **`scale`** (*`float, optional`*) – Scaling for the constraints.

restrict_to_idf()

restrict_to_regular()

PrimalDualVector

class kona.linalg.vectors.composite.**PrimalDualVector**(*primal_vec*, *eq_vec=None*, *ineq_vec=None*)

Bases: *kona.linalg.vectors.composite.CompositeVector*

A composite vector made up of primal, dual equality, and dual inequality vectors. :param _memory: All-knowing Kona memory manager. :type _memory: KonaMemory :param primal: Primal component of the composite vector. :type primal: DesignVector :param eq: Dual component corresponding to the equality constraints. :type eq: DualVectorEQ :param ineq: Dual component corresponding to the inequality constraints. :type ineq: DualVectorINEQ

equals_KKT_conditions(*x*, *state*, *adjoint*, *obj_scale=1.0*, *cnstr_scale=1.0*)

Calculates the total derivative of the Lagrangian $\mathcal{L}(x, u) = f(x, u) + \lambda_h^T h(x, u) + \lambda_g^T g(x, u)$ with respect to $\begin{pmatrix} x & \lambda_h & \lambda_g \end{pmatrix}^T$, where h denotes the equality constraints (if any) and g denotes the inequality constraints (if any). Note that these (total) derivatives do not represent the complete set of first-order optimality conditions in the case of inequality constraints. :param x: Evaluate derivatives at this primal-dual point. :type x: PrimalDualVector :param state: Evaluate derivatives at this state point. :type state: StateVector :param adjoint: Evaluate derivatives using this adjoint vector. :type adjoint: StateVector :param obj_scale: Scaling for the objective function. :type obj_scale: float, optional :param cnstr_scale: Scaling for the constraints. :type cnstr_scale: float, optional

equals_homotopy_residual(*dLdx*, *x*, *init*, *mu=1.0*)

Using dLdx=:math:*begin{pmatrix} nabla_x L && h && g end{pmatrix}*, which can be obtained from the method equals_KKT_conditions, as well as the initial values init=:math:*begin{pmatrix} x_0 && h(x_0,u_0) && g(x_0,u_0) end{pmatrix}* and the current point x=:math:*begin{pmatrix} x && lambda_h && lambda_g end{pmatrix}*, this method computes the following nonlinear vector function: .. math:: r(x,lambda_h,lambda_g;mu) = begin{bmatrix} muleft[nabla_x f(x, u) - nabla_x h(x, u)^T lambda_{h} - nabla_x g(x, u)^T lambda_{g}right] + (1 - mu)(x - x_0) \ -mu h(x,u) - (1-mu)lambda_h \ -|g(x,u) - (1-mu)*g_0 - lambda_g|^3 + (g(x,u) - (1-mu)g_0)^3 + lambda_g^3 - (1-mu)hat{g} end{bmatrix} where $h(x, u)$ are the equality constraints, and $g(x, u)$ are the inequality constraints. The vectors λ_h and λ_g are the associated Lagrange multipliers. When mu=1.0, we recover a set of nonlinear algebraic equations equivalent to the first-order optimality conditions. :param dLdx: The total derivative of the Lagranginan with respect to the primal and dual variables. :type dLdx: PrimalDualVector :param x: The current solution vector value corresponding to dLdx. :type x: PrimalDualVector :param init: The initial primal variable, as well as the initial constraint values. :type init: PrimalDualVector :param mu: Homotopy parameter; must be between 0 and 1. :type mu: float

equals_init_guess()

Sets the primal-dual vector to the initial guess, using the initial design.

equals_predictor_rhs(*dLdx*, *x*, *init*, *mu=1.0*)

Using dLdx=:math:*begin{pmatrix} nabla_x L && h && g end{pmatrix}*, which can be obtained from the method equals_KKT_conditions, as well as the initial values init=:math:*begin{pmatrix} x_0 && h(x_0,u_0) && g(x_0,u_0) end{pmatrix}* and the current point x=:math:*begin{pmatrix} x && lambda_h && lambda_g end{pmatrix}*, this method computes the right-hand-side for the homotopy-predictor step, that is .. math:: partial r/partial mu = begin{bmatrix} left[nabla_x f(x, u) - nabla_x h(x, u)^T lambda_{h} - nabla_x g(x, u)^T lambda_{g}right] - (x - x_0) \ -h(x,u) + lambda_h \ -3*(g_0)|g(x,u) - (1-mu)*g_0 - lambda_g|^2 + 3*g_0*(g(x,u) - (1-mu)g_0)^2 + hat{g} end{bmatrix} where $h(x, u)$ are the equality constraints, and $g(x, u)$ are the inequality constraints. The vectors λ_h and λ_g are the associated Lagrange multipliers. :param dLdx: The total derivative of the Lagranginan with respect to the primal and dual variables. :type dLdx: PrimalDualVector :param x: The current solution vector value corresponding to dLdx.

:type x: PrimalDualVector :param init: The initial primal variable, as well as the initial constraint values. :type init: PrimalDualVector :param mu: Homotopy parameter; must be between 0 and 1. :type mu: float

get_base_data(*A*)
Inserts the PrimalDualVector's underlying data into the given array :param A: Array into which data is inserted. :type A: numpy array

get_dual()
Returns 1) eq or ineq if only one is None, 2) a CompositeDualVector if neither is none or 3) None if both are none

get_num_var()
Returns the total number of variables in the PrimalDualVector

get_optimality_and_feasiblity()
Returns the norm of the primal (opt) the dual parts of the vector (feas). If the dual parts of the vector are both None, then feas is returned as zero.

init_dual = 0.0

set_base_data(*A*)
Copies the given array into the PrimalDualVector's underlying data :param A: Array that is copied into the PrimalDualVector. :type A: numpy array

ReducedKKTVector

class kona.linalg.vectors.composite.**ReducedKKTVector**(*primal_vec*, *dual_vec*)
Bases: *kona.linalg.vectors.composite.CompositeVector*

A composite vector representing a combined primal and dual vectors.

Parameters

- **_memory** (KonaMemory) – All-knowing Kona memory manager.
- **_primal** (DesignVector *or* CompositePrimalVector) – Primal component of the composite vector.
- **_dual** (*DualVector*) – Dual components of the composite vector.

equals_KKT_conditions(*x*, *state*, *adjoint*, *barrier=None*, *obj_scale=1.0*, *cnstr_scale=1.0*)
Calculates the total derivative of the Lagrangian $\mathcal{L}(x,u) = f(x,u) + \lambda_{eq}^T c_{eq}(x,u) + \lambda_{ineq}^T (c_{ineq}(x,u) - s)$ with respect to $\begin{pmatrix} x & s & \lambda_{eq} & \lambda_{ineq} \end{pmatrix}^T$. This total derivative represents the Karush-Kuhn-Tucker (KKT) convergence conditions for the optimization problem defined by $\mathcal{L}(x, s, \lambda_{eq}, \lambda_{ineq})$ where the stat variables $u(x)$ are treated as implicit functions of the design.

The full expression of the KKT conditions are:

$$\nabla \mathcal{L} = \begin{bmatrix} \nabla_x f(x,u) + \nabla_x c_{eq}(x,u)^T \lambda_{eq} + \nabla_x c_{inq}(x,u)^T \lambda_{ineq} \\ ^{-1}e - \lambda_{ineq} \\ c_{eq}(x,u) \\ c_{ineq}(x,u) - s \end{bmatrix}$$

Parameters

- **x** (ReducedKKTVector) – Evaluate KKT conditions at this primal-dual point.
- **state** (StateVector) – Evaluate KKT conditions at this state point.
- **adjoint** (StateVector) – Evaluate KKT conditions using this adjoint vector.

- **barrier** (*float, optional*) – Log barrier coefficient for slack variable non-negativity.
- **obj_scale** (*float, optional*) – Scaling for the objective function.
- **cnstr_scale** (*float, optional*) – Scaling for the constraints.

equals_init_guess()

Sets the KKT vector to the initial guess, using the initial design.

init_dual = 0.0

CompositeFactory

class kona.linalg.vectors.composite.**CompositeFactory**(*memory*, *vec_type*)

Bases: object

A factory-like object that generates composite vectors.

It is intended to mimic the function of basic VectorFactory objects.

Parameters

- **memory** (KonaMemory) – All-knowing Kona memory manager.
- **vec_type** (*CompositeVector-like*) – Type of composite vector the factory will produce.

Variables

- **_memory** (KonaMemory) – All-knowing Kona memory manager.
- **_vec_type** (*CompositeVector-like*) – Type of composite vector the factory will produce.
- **_factories** (*list of VectorFactory or* CompositeFactory) – Vector factories used in generating the composite vector of choice.

generate()

request_num_vectors(*count*)

kona.linalg.matrices

Submodules

kona.linalg.matrices.common

Subclasses

KonaMatrix (base class)

class kona.linalg.matrices.common.**KonaMatrix**(*primal=None*, *state=None*, *transposed=False*)

Bases: object

An abstract matrix class connected to Kona memory. This class is used to define a variety of jacobian matrices and other composite objects containing matrix-related methods used in optimization tasks.

Parameters

- **primal** (DesignVector) –
- **state** (StateVector) –
- **transposed** (*boolean, optional*) –

Variables

- **_design** (*PrimalVector*) – Primal vector point for linearization.
- **_state** (StateVector) – State vector point for linearization
- **_transposed** (*boolean*) – Flag to determine if the matrix is transposed

T

Returns the transposed version of the matrix.

Returns **KonaMatrix-like**

Return type Transposed version of the matrix.

linearize (*primal*, *state*)

Store the vector points around which a non-linear matrix should be linearized.

Parameters

- **primal** (DesignVector *or* CompositePrimalVector) –
- **state** (StateVector) –

product (*in_vec*, *out_vec*)

Performs a matrix-vector product at the internally stored linearization.

Parameters

- **in_vec** (KonaVector) –
- **out_vec** (KonaVector) –

Returns **out_vec**

Return type *KonaVector*

Derived Matrices

Residual Jacobians (dRdX, dRdU)

class kona.linalg.matrices.common.**dRdX** (*primal=None*, *state=None*, *transposed=False*)

Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the system residual with respect to primal variables.

product (*in_vec*, *out_vec*)

class kona.linalg.matrices.common.**dRdU** (*primal=None*, *state=None*, *transposed=False*)

Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the system residual with respect to state variables.

precond (*in_vec*, *out_vec*)

product (*in_vec*, *out_vec*)

solve (*rhs_vec*, *solution*, *rel_tol=1e-08*)
Performs a linear solution with the provided right hand side.

If the transposed matrix object is used, and the right hand side vector is `None`, then this function performs an adjoint solution.

Parameters

- **rhs_vec** (`StateVector or None`) – Right hand side vector for solution.
- **rel_tol** (*float*) – Solution tolerance.
- **solution** (`StateVector`) – Vector where the result should be stored.

Returns Convergence flag.

Return type bool

Constraint Jacobians (dCdX, dCdU)

class `kona.linalg.matrices.common.`**dCdX** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Combined partial constraint jacobian matrix that can do both equality and inequality products depending on what input vectors are provided.

product (*in_vec*, *out_vec*)

class `kona.linalg.matrices.common.`**dCdU** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Combined partial constraint jacobian matrix that can do both equality and inequality products depending on what input vectors are provided.

product (*in_vec*, *out_vec*, *state_work=None*)

class `kona.linalg.matrices.common.`**dCEQdX** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the equality constraints with respect to design vars.

product (*in_vec*, *out_vec*)

class `kona.linalg.matrices.common.`**dCEQdU** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the equality constraints with respect to state vars.

product (*in_vec*, *out_vec*)

class `kona.linalg.matrices.common.`**dCINdX** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the inequality constraints with respect to design vars.

product (*in_vec*, *out_vec*)

class `kona.linalg.matrices.common.`**dCINdU** (*primal=None*, *state=None*, *transposed=False*)
Bases: *kona.linalg.matrices.common.KonaMatrix*

Partial jacobian of the inequality constraints with respect to state vars.

product (*in_vec*, *out_vec*)

kona.linalg.matrices.hessian

Submodules

kona.linalg.matrices.hessian.basic

Subclasses

BaseHessian (base class)

class `kona.linalg.matrices.hessian.basic.`**`BaseHessian`**(*vector_factory*, *optns=None*)

Bases: `object`

Abstract matrix object that defines the Hessian of an optimization problem.

Parameters

- **`vector_factory`** (`VectorFactory`) –
- **`optns`** (*`dict, optional`*) –
- **`out_file`** (*`file, optional`*) –

Variables

- **`vec_fac`** (`VectorFactory`) – Generator for arbitrary KonaVector types.
- **`out_file`** (*`file`*) – File stream for data output.

`product`(*in_vec*, *out_vec*)

Applies the Hessian itself to the input vector.

Parameters

- **`in_vec`** (`KonaVector`) – Vector that gets multiplied with the inverse Hessian.
- **`out_vec`** (`KonaVector`) – Vector that stores the result of the operation.

`solve`(*in_vec*, *out_vec*, *rel_tol=1e-15*)

Applies the inverse of the approximate Hessian to the input vector.

Parameters

- **`in_vec`** (`KonaVector`) – Vector that gets multiplied with the inverse Hessian.
- **`out_vec`** (`KonaVector`) – Vector that stores the result of the operation.
- **`rel_tol`** (*`float, optional`*) – Convergence tolerance for the operation.

QuasiNewtonApprox (base class)

class `kona.linalg.matrices.hessian.basic.`**`QuasiNewtonApprox`**(*vector_factory*, *optns={}*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Base class for quasi-Newton approximations of the Hessian

Variables

- **`max_stored`** (*`int`*) – Maximum number of corrections stored.
- **`norm_init`** (*`float`*) – Initial norm of design component of gradient.

- **init_hessian** (KonaVector) – Initial (diagonal) Hessian approximation (stored as a vector).
- **s_list** (*list of KonaVector*) – Difference between subsequent solutions: $s_k = x_{k+1} - x_k$
- **y_list** (*list of KonaVector*) – Difference between subsequent gradients: $y_k = g_{k+1} - g_k$

add_correction (*s_new*, *y_new*)
Adds a new correction to the Hessian approximation.

Parameters

- **s_new** (KonaVector) – Difference between subsequent solutions.
- **y_new** (KonaVector) – Difference between subsequent gradients.

Subclasses

LimitedMemoryBFGS

class kona.linalg.matrices.hessian.**LimitedMemoryBFGS** (*vector_factory*, *optns=None*)
Bases: *kona.linalg.matrices.hessian.basic.QuasiNewtonApprox*

Limited-memory BFGS approximation for the Hessian.

Variables

- **lambda0** (*float*) – ?
- **s_dot_s_list** (*list of float*) – The L2 norm of the step vector.
- **s_dot_y_list** (*list of float*) – Curvature.

add_correction (*s_in*, *y_in*)

product (*in_vec*, *out_vec*)

solve (*u_vec*, *v_vec*, *rel_tol=1e-15*)

LimitedMemorySR1

class kona.linalg.matrices.hessian.**LimitedMemorySR1** (*vector_factory*, *optns=None*)
Bases: *kona.linalg.matrices.hessian.basic.QuasiNewtonApprox*

Limited memory symmetric rank-one update

Variables

- **lambda0** (*float*) – ?
- **threshold** (*float*) – ?

add_correction (*s_in*, *y_in*)
Add the step and change in gradient to the lists storing the history.

product (*u_vec*, *v_vec*)

solve (*u_vec*, *v_vec*, *rel_tol=1e-15*)

ReducedHessian

class `kona.linalg.matrices.hessian.`**`ReducedHessian`**(*vector_factories*, *optns=None*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Reduced-space approximation of the Hessian-vector product using a 2nd order adjoint formulation.

Note: Insert inexact-Hessian paper reference here

Variables

- **`product_fact`** (*`float`*) – Solution tolerance for 2nd order adjoints.
- **`lamb`** (*`float`*) – ???
- **`scale`** (*`float`*) – ???
- **`quasi_newton`** (*`QuasiNewtonApproximation -like`*) – QN Hessian object to be used as preconditioner.

`linearize`(*at_design*, *at_state*, *at_adjoint*, *scale=1.0*)

An abstracted "linearization" method for the matrix.

This method does not actually factor any real matrices. It also does not perform expensive linear or non-linear solves. It is used to update internal vector references and perform basic calculations using only cheap matrix-vector products.

Parameters

- **`at_design`** (`DesignVector`) – Design point at which the product is evaluated.
- **`at_state`** (`StateVector`) – State point at which the product is evaluated.
- **`at_dual`** (*`DualVector`*) – Lagrange multipliers at which the product is evaluated.
- **`at_adjoint`** (`StateVector`) – 1st order adjoint variables at which the product is evaluated.

`product`(*in_vec*, *out_vec*)

Matrix-vector product for the reduced KKT system.

Parameters

- **`in_vec`** (`ReducedKKTVector`) – Vector to be multiplied with the KKT matrix.
- **`out_vec`** (`ReducedKKTVector`) – Result of the operation.

`set_krylov_solver`(*krylov_solver*)

`set_quasi_newton`(*quasi_newton*)

`solve`(*rhs*, *solution*, *rel_tol=None*)

Solve the linear system defined by this matrix using the embedded krylov solver.

Parameters

- **`rhs`** (`DesignVector`) – Right hand side vector for the system.
- **`solution`** (*`PrimalVector`*) – Solution of the system.
- **`rel_tol`** (*`float, optional`*) – Relative tolerance for the krylov solver.

LagrangianHessian

class `kona.linalg.matrices.hessian.`**`LagrangianHessian`**(*vector_factories*, *optns=None*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Matrix object for the Hessian block of the reduced KKT matrix.

Uses the same 2nd order adjoint formulation as ReducedKKTMatrix, but only for the diagonal Hessian block, $\mathsf{W} = \nabla^2_x \mathcal{L}$.

If slack terms are present, it will also perform a product with the slack derivative of the Lagrangian.

`approx`

`linearize`(*X*, *at_state*, *at_adjoint*, *obj_scale=1.0*, *cnstr_scale=1.0*)

`multiply_W`(*in_vec*, *out_vec*)

`multiply_slack`(*in_vec*, *out_vec*)

`precond`(*in_vec*, *out_vec*)

`product`(*in_vec*, *out_vec*)

`set_projector`(*proj_cg*)

`solve`(*rhs*, *solution*, *rel_tol=None*)

TotalConstraintJacobian

class `kona.linalg.matrices.hessian.`**`TotalConstraintJacobian`**(*vector_factories*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Matrix object for the constraint block of the reduced KKT matrix.

Uses the same 2nd order adjoint formulation as ReducedKKTMatrix, but only for the off-diagonal total contraint jacobian blocks, $\mathsf{A} = \nabla_x C$.

Parameters

- **`T`** (`TotalConstraintJacobian`) – Transposed matrix.
- **`approx`** (`TotalConstraintJacobian`) – Approximate/inexact matrix.

`T`

`approx`

`linearize`(*at_design*, *at_state*, *scale=1.0*)

`product`(*in_vec*, *out_vec*)

ReducedKKTMatrix

class `kona.linalg.matrices.hessian.`**`ReducedKKTMatrix`**(*vector_factories*, *optns=None*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Reduced approximation of the KKT matrix using a 2nd order adjoint formulation.

For problems with only equality constraints, the KKT system is given as:

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} & \nabla_x c^T \\ \nabla_x c_{eq} & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x \{ - \lambda^T \nabla_x c \\ -c \end{bmatrix}$$

where $\mathcal{L}$ is the Lagrangian defined as:

$$\mathcal{L}(x, u(x), \lambda) = F(x, u(x)) + \lambda^T c(x, u(x))$$

For problems with inequality constraints, slack variables s are introduced alongside a log-barrier term for non-negativity, such that the Lagrangian $\mathcal{L}$ becomes:

$$\mathcal{L}(x, u(x), \lambda) = F(x, u(x)) + \lambda_{eq}^T c_{eq}(x, u(x)) + \lambda_{ineq}^T \left[c_{ineq}(x, u(x)) - s\right] + \frac{1}{2}\mu \sum_{i=1}^{n_{ineq}} ln(s_i)$$

The inequality constrained KKT system is then defined as:

$$\begin{bmatrix} \nabla_x^2 \mathcal{L} & 0 & \nabla_x c_{eq}^T & \nabla_x c_{ineq}^T \\ 0 & \Sigma & 0 & I \\ \nabla_x c_{eq} & 0 & 0 & 0 \\ \nabla_x c_{ineq} & I & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta s \\ \Delta \lambda_{eq} \\ \Delta \lambda_{ineq} \end{bmatrix} = \begin{bmatrix} -\nabla_x \{ - \lambda_{eq}^T \nabla_x c_{eq} - \lambda_{eq}^T \nabla_x c_{eq} \\ \lambda^T \mu * S^{-1} e \\ -c_{eq} \\ -c_{ineq} + s \end{bmatrix}$$

Note: Currently, Kona does not have any optimization algorithms that support inequality constraints. The slack implementation in this matrix is part of an ongoing development effort to support inequality constraints at a future date.

Variables

- **`product_tol`** (*`float`*) – Tolerance for 2nd order adjoint system solutions.
- **`grad_scale, feas_scale`** (*`scale,`*) – Optimality metric normalization factors.
- **`krylov`** (`KrylovSolver`) – A krylov solver object used to solve the system defined by this matrix.
- ***`dRdU, dCdX, dCdU`*** (`dRdX,`) – Various abstract jacobians used in calculating the mat-vec product.

`linearize` (*at_kkt*, *at_state*, *at_adjoint*, *obj_scale=1.0*, *cnstr_scale=1.0*)

Linearize the KKT matrix at the given KKT, state, adjoint and barrier point. This method does not perform any factorizations or matrix operations.

Parameters

- **`at_kkt`** (`ReducedKKTVector`) – KKT vector at which the product is evaluated
- **`at_state`** (`StateVector`) – State point at which the product is evaluated.
- **`at_adjoint`** (`StateVector`) – 1st order adjoint variables at which the product is evaluated.
- **`obj_scale`** (*`float, optional`*) – Factor by which the objective component of the product is scaled.
- **`cnstr_scale`** (*`float, optional`*) – Factor by which the constraint component of the product is scaled.

`product` (*in_vec*, *out_vec*)

Matrix-vector product for the reduced KKT system.

Parameters

- **`in_vec`** (`ReducedKKTVector`) – Vector to be multiplied with the KKT matrix.
- **`out_vec`** (`ReducedKKTVector`) – Result of the operation.

`set_krylov_solver` (*krylov_solver*)

AugmentedKKTMatrix

class `kona.linalg.matrices.hessian.`**`AugmentedKKTMatrix`**(*vector_factories*, *optns=None*)

Bases: *`kona.linalg.matrices.hessian.basic.BaseHessian`*

Matrix object for the the normal system associated with the reduced KKT system.

The normal system is defined as:

$$\begin{bmatrix} I & 0 & Aeq^T & Ain^T \\ 0 & Sigma & 0 & I \\ Aeq & 0 & 0 & 0 \\ Ain & I & 0 & 0 \end{bmatrix}$$

This matrix is used to solve the normal-step in a composite-step algorithm.

`linearize`(*at_kkt*, *at_state*)

`product`(*in_vec*, *out_vec*)

`solve`(*rhs*, *solution*, *rel_tol=None*)

kona.linalg.solvers

Submodules

kona.linalg.solvers.krylov

Submodules

kona.linalg.solvers.krylov.basic

Subclasses

KrylovSolver (base class)

class `kona.linalg.solvers.krylov.basic.`**`KrylovSolver`**(*vector_factory*, *optns=None*)

Bases: `object`

Base class for all Krylov solvers.

Parameters

- **`vector_factory`** (`VectorFactory`) –
- **`optns`** (*`dict, optional`*) –

Variables

- **`vec_factory`** (`VectorFactory`) – Used to generate abstracted KonaVector objects.
- **`max_iter`** (*`int`*) – Maximum iterations for the CG solve.
- **`rel_tol`** (*`float`*) – Relative residual tolerance for the solution.
- **`check_res`** (*`boolean`*) – Flag for checking the residual after solution is found
- **`out_file`** (*`file`*) – File stream for writing convergence data.

solve (*mat_vec*, *b*, *x*, *precond*)

Solves the Ax=b linear system iteratively.

mat_vec [function] Matrix-vector product for left-hand side matrix A.

b [KonaVector] Right-hand side vector.

x [KonaVector] Solution vector

precond [function] Matrix-vector product for approximate inv(A).

Subclasses

FGMRES

class `kona.linalg.solvers.krylov.`**FGMRES** (*vector_factory*, *optns=None*, *eq_factory=None*, *ineq_factory=None*)

Bases: *kona.linalg.solvers.krylov.basic.KrylovSolver*

Flexible Generalized Minimum RESidual solver.

solve (*mat_vec*, *b*, *x*, *precond*)

STCG

class `kona.linalg.solvers.krylov.`**STCG** (*vector_factory*, *optns=None*, *dual_factory=None*)

Bases: *kona.linalg.solvers.krylov.basic.KrylovSolver*

Steihaug-Toint Conjugate Gradient (STCG) Krylov iterative method

Variables

- **radius** (*float*) – Trust region radius.
- **proj_cg** (*boolean*) –

solve (*mat_vec*, *b*, *x*, *precond*)

FLECS

class `kona.linalg.solvers.krylov.`**FLECS** (*vector_factories*, *optns=None*)

Bases: *kona.linalg.solvers.krylov.basic.KrylovSolver*

FLexible Equality-Constrained Subproblem Krylov iterative solver.

Note: For more information, see *SIAM paper on FLECS <http://epubs.siam.org/doi/10.1137/140994496>*.

Variables

- **primal_factory** (VectorFactory) – Factory for DesignVector objects.
- **dual_factory** (VectorFactory) – Factory for DualVector objects.
- **mu** (*float*) – Quadratic subproblem constraint penalty factor.
- **grad_scale** (*float*) – Scaling coefficient for the primal space.
- **feas_scale** (*float*) – Scaling coefficient for the dual space.

- **lin_depend** (*boolean*) – Flag for Hessian linear dependence.
- **neg_curv** (*boolean*) – Flag for negative curvature in the search direction.
- **trust_active** (*boolean*) – Flag for trust-region detection.

Parameters

- **vector_factories** (*tuple of VectorFactory*) – A pair of vector factories, one for Primal and one for Dual type.
- **optns** (*dict, optional*) – Optiona dictionary

apply_correction (*cnstr*, *step*)

re_solve (*b*, *x*)

solve (*mat_vec*, *b*, *x*, *precond*)

solve_subspace_problems ()

GCROT

class kona.linalg.solvers.krylov.**GCROT** (*vector_factory*, *optns=None*, *eq_factory=None*, *ineq_factory=None*)

Bases: *kona.linalg.solvers.krylov.basic.KrylovSolver*

Generalized Conjugate Residual method with Orthogonalization, Truncated

clear_subspace ()

solve (*mat_vec*, *b*, *x*, *precond*)

kona.linalg.solvers.util

Functions

kona.linalg.solvers.util.**abs_sign** (*x*, *y*)

Returns the value $|x|\text{sign}(y)$; used in GMRES, for example.

Parameters

- **x** (*float*) –
- **y** (*float*) –

Returns float

Return type $|x|\text{sign}(y)$

kona.linalg.solvers.util.**calc_epsilon** (*eval_at_norm*, *mult_by_norm*)

Determines the perturbation parameter for forward-difference based matrix-vector products

Parameters

- **eval_at_norm** (*float*) – the norm of the vector at which the Jacobian-like matrix is evaluated
- **mult_by_norm** (*float*) – the norm of the vector that is being multiplied

Returns float

Return type perturbation parameter

`kona.linalg.solvers.util.`**`eigen_decomp`**(*A*)

Returns the (sorted) eigenvalues and eigenvectors of the symmetric part of a square matrix.

The matrix A is stored in dense format and is not assumed to be exactly symmetric. The eigenvalues are found by calling np.linalg.eig, which is given 0.5*(A^T + A) as the input matrix, not A itself.

Parameters **`A`** (*`2-D numpy.ndarray`*) – matrix stored in dense format; not necessarily symmetric

Returns

- **eig_vals** (*1-D numpy.ndarray*) – the eigenvalues in ascending order
- **eig_vecs** (*2-D numpy.ndarray*) – the eigenvectors sorted appropriated

`kona.linalg.solvers.util.`**`apply_givens`**(*s*, *c*, *h1*, *h2*)

Applies a Givens rotation to a 2-vector

Parameters

- **`s`** (*`float`*) – sine of the Givens rotation angle
- **`c`** (*`float`*) – cosine of the Givens rotation angle
- **`h1`** (*`float`*) – first element of 2x1 vector being transformed
- **`h2`** (*`float`*) – second element of 2x1 vector being transformed

`kona.linalg.solvers.util.`**`generate_givens`**(*dx*, *dy*)

Generates the Givens rotation matrix for a given 2-vector

Based on givens() of SPARSKIT, which is based on p.202 of “Matrix Computations” by Golub and van Loan.

Parameters

- **`dx`** (*`float`*) – element of 2x1 vector being transformed
- **`dy`** (*`float`*) – element of 2x1 vector being set to zero

Returns

- **dx** (*float*) – element of 2x1 vector being transformed
- **dy** (*float*) – element of 2x1 vector being set to zero
- **s** (*float*) – sine of the Givens rotation angle
- **c** (*float*) – cosine of the Givens rotation angle

`kona.linalg.solvers.util.`**`lanczos_tridiag`**(*mat_vec*, *Q*, *Q_init=False*)

Uses the traditional Lanczos algorithm to compute a tridiagonalization.

Since this is based on the Arnoldi’s method, we only require a matrix-vector product for the matrix of interest, and not the full explicit matrix itself.

Parameters

- **`mat_vec`** (*`function`*) – Matrix-vector product for a symmetric matrix.
- **`Q`** (*`List[KonaVector]`*) – Pre-allocated subspace array containing KonaVectors matching the vector-type of the product
- **`Q_init`** (*`boolean`*) – If *True*, start the V-subspace with the Q[0] already stored. If ‘False’, generate a vector of ones in Q[0] to start with.

Returns **T** – Tri-diagonalization of the matrix

Return type array_like

`kona.linalg.solvers.util.`**`lanczos_bidiag`**(*fwd_mat_vec*, *Q*, *q_work*, *rev_mat_vec*, *P*, *p_work*, *Q_init=False*)

Uses the bi-orthogonal Lanczos algorithm to bidiagonalize a matrix.

Since this is based on the Arnoldi's method, we only require a matrix-vector product for the matrix of interest, and not the full explicit matrix itself.

Parameters

- **fwd_mat_vec** (*function*) – Forward matrix-vector product for the matrix of interest
- **Q** (*List [KonaVector]*) – Pre-allocated subspace array containing KonaVectors matching the vector-type of the forward product
- **q_work** (KonaVector) – Work vector matching the KonaVector-type of the forward product
- **rev_mat_vec** (*function*) – Reverse (transpose) matrix-vector product for the matrix of interest
- **P** (*List [KonaVector]*) – Pre-allocated subspace array containing KonaVectors matching the vector-type of the reverse (transpose) product
- **p_work** (KonaVector) – Work vector matching the KonaVector-type of the reverse product
- **Q_init** (*boolean*) – If *True*, start the V-subspace with the Q[0] already stored. If 'False', generate a vector of ones in Q[0] to start with.

Returns **B** – Truncated bi-diagonalization of the matrix

Return type array_like

`kona.linalg.solvers.util.`**`solve_tri`**(*A*, *b*, *lower=False*)

Solve an upper-triangular system $Ux = b$ (lower=False) or lower-triangular system $Lx = b$ (lower=True)

Parameters

- **A** (*2-D numpy.matrix*) – a triangular matrix
- **b** (*1-D numpy.ndarray*) – the right-hand side of the system
- **x** (*1-D numpy.ndarray*) – on exit, the solution
- **lower** (*boolean*) – if True, A stores an lower-triangular matrix; stores an upper-triangular matrix otherwise

`kona.linalg.solvers.util.`**`solve_trust_reduced`**(*H*, *g*, *radius*)

Solves the reduced-space trust-region subproblem (the secular equation)

This assumes the reduced space objective is in the form $g^T x + \frac{1}{2} x^T Hx$. Furthermore, the case $g = 0$ is not handled presently.

Parameters

- **H** (*2-D numpy.matrix*) – reduced-space Hessian
- **g** (*1-D numpy.ndarray*) – gradient in the reduced space
- **radius** (*float*) – trust-region radius

Returns

- **y** (*1-D numpy.ndarray*) – solution to reduced-space trust-region problem
- **lam** (*float*) – Lagrange multiplier value
- **pred** (*float*) – predicted decrease in the objective

`kona.linalg.solvers.util.`**`mod_gram_schmidt`**(*i*, *B*, *C*, *w*, *normalize=False*)

`kona.linalg.solvers.util.`**`mod_GS_normalize`**(*i*, *Hsbg*, *w*)

`kona.linalg.solvers.util.`**`write_header`**(*out_file*, *solver_name*, *res_tol*, *res_init*)

Writes krylov solver data file header text.

Parameters

- **`out_file`** (*`file`*) – File handle for write destination
- **`solver_name`** (*`string`*) – Name of Krylov solver type.
- **`res_tol`** (*`float`*) – Residual tolerance for convergence.
- **`res_init`** (*`float`*) – Initial residual norm.

`kona.linalg.solvers.util.`**`write_history`**(*out_file*, *num_iter*, *res*, *res_init*)

Writes krylov solver data file iteration history.

Parameters

- **`out_file`** (*`file`*) – File handle for write destination
- **`num_iter`** (*`int`*) – Current iteration count.
- **`res`** (*`float`*) – Current residual norm.
- **`res_init`** (*`float`*) – Initial residual norm.

Submodules

kona.linalg.memory

Subclasses

KonaMemory : abstract vector-memory manager

class `kona.linalg.memory.`**`KonaMemory`**(*solver*)

Bases: `object`

All-knowing Big Brother abstraction layer for Kona.

Parameters **`solver`** (`UserSolver`) – A user-defined solver object that implements specific elementary tasks.

Variables

- **`solver`** (`UserSolver`) – A user-defined solver object that implements specific elementary tasks.
- **`primal_factory`** (`VectorFactory`) – Vector generator for primal space.
- **`state_factory`** (`VectorFactory`) – Vector generator for state space.
- **`dual_factory`** (`VectorFactory`) – Vector generatorfor dual space.
- **`precond_count`** (*`int`*) – Counter for tracking optimization cost.
- **`vector_stack`** (*`dict`*) – Memory stack for unused vector data.
- **`rank`** (*`int`*) – Processor rank.

allocate_memory()

Absolute final stage of memory allocation.

Once the number of required vectors are tallied up inside vector factories, this function will manipulate the user-defined solver object to allocate all actual, real memory required for the optimization.

open_file(*filename*)

pop_vector(*vec_type*)

Take an unused user vector object out of the memory stack and serve it to the vector factory.

Parameters **vec_type** (KonaVector) – Vector type to be popped from the stack.

Returns User-defined vector data structure.

Return type *BaseVector*

push_vector(*vec_type*, *user_data*)

Pushes an unused user vector data container into the memory stack so it can be used later in a new vector.

Parameters

- **vec_type** (KonaVector) – Vector type of the memory stack.
- **user_data** (BaseVector) – Unused user vector data container.

VectorFactory : KonaVector generator

class kona.linalg.memory.**VectorFactory**(*memory*, *vec_type=None*)

Bases: object

A factory object used for generating Kona's abstracted vector classes.

This object also tallies up how many vectors of which kind needs to be allocated, based on the memory requirements of each optimization function.

Parameters

- **memory** (KonaMemory) –
- **vec_type** (*PrimalVector or* StateVector *or DualVector*) –

Variables

- **num_vecs** (*int*) – Number of vectors requested from this factory.
- **_memory** (KonaMemory) – All-knowing Kona memory manager.
- **_vec_type** (DesignVector *or* StateVector *or DualVector*) – Kona abstracted vector type associated with this factory

generate()

Generate one abstract KonaVector of this vector factory's defined type.

Returns Abstracted vector type linked to user generated memory.

Return type *KonaVector*

request_num_vectors(*count*)

Put in a request for the factory's vector type, to be used later.

Parameters **count** (*int*) – Number of vectors requested.

Functions

kona.user

Subclasses

BasicVector

class `kona.user.`**`BaseVector`**(*size*, *val=0*)

Bases: `object`

Kona's default data container, implemented on top of NumPy arrays.

Any user defined data container must implement all the methods below.

These vectors are initialized by the user-created *BaseAllocator* object. Therefore, the initialization implementation does not need to exactly follow the example below. The user is free to initialize these vector objects any which way they like, as long as it is in sync with the *BaseAllocator* implementation.

Parameters

- **`size`** (*`int`*) – Size of the 1-D numpy vector contained in this object.
- **`val`** (*`float or array-like, optional`*) – Data value for vector initialization.

Variables **`data`** (*`numpy.array`*) – Numpy vector containing numerical data.

`equals_ax_p_by`(*a*, *x*, *b*, *y*)

Perform the elementwise scaled addition defined below:

$$a\mathbf{x} + b\mathbf{y}$$

The result is saved into this vector.

Parameters

- **`a`** (*`double`*) – Scalar coefficient of `x`.
- **`x`** (`BaseVector`) – Vector to be operated on.
- **`b`** (*`double`*) – Scalar coefficient of `y`.
- **`y`** (`BaseVector`) – Vector to be operated on.

`equals_value`(*value*)

Set all elements of this vector to given scalar value.

Parameters **`value`** (*`float`*) –

`equals_vector`(*vector*)

Set this vector equal to the given vector.

Parameters **`vector`** (`BaseVector`) – Incoming vector for in-place operation.

`exp`(*vector*)

Calculate element-wise exponential operation on the vector.

Parameters **`vector`** (`BaseVector`) – Incoming vector for in-place operation.

`infty`

Infinity norm of the vector.

Returns Infinity norm.

Return type float

inner(*vector*)
Perform an inner product between the given vector and this one.

Parameters **vector** (BaseVector) – Incoming vector for in-place operation.

Returns Result of the operation.

Return type float

log(*vector*)
Calculate element-wise natural log operation on the vector.

Kona will never call this on zero-valued vectors. No special handling of zero values necessary.

Parameters **vector** (BaseVector) – Incoming vector for in-place operation.

plus(*vector*)
Add the given vector to this vector.

Parameters **vector** (BaseVector) – Incoming vector for in-place operation.

pow(*power*)
Calculate element-wise power operation on the vector.

Kona will never call a negative power on zero-valued vectors. No special handling of zero values necessary.

Parameters **power** (*float*) –

times_scalar(*value*)
Multiply all elements of this vector with the given scalar.

Parameters **value** (*float*) –

times_vector(*vector*)
Perform element-wise multiplication between vectors.

Parameters **vector** (BaseVector) – Incoming vector for in-place operation.

UserSolver

class kona.user.**UserSolver**(*num_design*, *num_state=0*, *num_eq=0*, *num_ineq=0*)
Bases: object

Base class for Kona objective functions, designed to be a template for any objective functions intended to be used for kona.Optimize().

This class provides some standard mathematical functionality via NumPy arrays and operators. However, attributes of the derived classes can have different data types. In these cases, the user must redefine the mathematical operation methods for these non-standard data types.

This solver wrapper is not initialized by Kona. The user must initialize it before handing it over to Kona's optimizer. Therefore the intialization implementation details are left up to the user entirely. Below is just an example used by Kona's own test problems.

Parameters

- **num_design** (*int*) – Design space size
- **num_state** (*int, optional*) – State space size.
- **num_eq** (*int, optional*) – Number of equality constraints
- **num_ineq** (*int, optional*) – Number of inequality constraints

Variables

- **`num_design`** (*`int`*) – Size of the design space
- **`num_state`** (*`int`*) – Number of state variables
- **`num_eq`** (*`int`*) – Number of equality constraints
- **`num_ineq`** (*`int`*) – Number of inequality constraints

`allocate_state`(*num_vecs*)
Allocate the requested number of state-space BaseVectors and return them in a plain array.

Parameters **`num_vecs`** (*`int`*) – Number of state vectors requested.

Returns Stack of BaseVectors in the state space

Return type list of BaseVector

`apply_precond`(*at_design*, *at_state*, *in_vec*, *out_vec*)
Apply the preconditioner to the vector at `in_vec` and store the result in `out_vec`. If the preconditioner is nonlinear, evaluate the application using the design and state vectors provided in `at_design` and `at_state`.

Note: If the solver uses `factor_linear_system()`, ignore the (design, state) evaluation point and use the previously factored preconditioner.

Parameters

- **`at_design`** (*`numpy.ndarray`*) – Current design vector.
- **`at_state`** (`BaseVector`) – Current state vector.
- **`in_vec`** (`BaseVector`) – Vector to be operated on.
- **`out_vec`** (`BaseVector`) – Location where user should store the result.

Returns Number of preconditioner calls required for the operation.

Return type int

`apply_precond_T`(*at_design*, *at_state*, *in_vec*, *out_vec*)
Apply the transposed preconditioner to the vector at `in_vec` and store the result in `out_vec`. If the preconditioner is nonlinear, evaluate the application using the design and state vectors provided in `at_design` and `at_state`.

Note: If the solver uses `factor_linear_system()`, ignore the (design, state) evaluation point and use the previously factored preconditioner.

Parameters

- **`at_design`** (*`numpy.ndarray`*) – Current design vector.
- **`at_state`** (`BaseVector`) – Current state vector.
- **`in_vec`** (`BaseVector`) – Vector to be operated on.
- **`out_vec`** (`BaseVector`) – Location where user should store the result.

Returns Number of preconditioner calls required for the operation.

Return type int

current_solution(*num_iter*, *curr_design*, *curr_state*, *curr_adj*, *curr_eq*, *curr_ineq*, *curr_slack*)

Kona will evaluate this method at every outer/nonlinear optimization iteration. It can be used to print out useful information to monitor the process, or to save design points of the intermediate iterations.

The current design vector, current state vector and current adjoint vector have been made available to the user via the arguments.

Parameters

- **num_iter** (*int*) – Current outer/nonlinear iteration number.
- **curr_design** (*numpy.ndarray*) – Current design point.
- **curr_state** (BaseVector) – Current state variables.
- **curr_adj** (BaseVector) – Currently adjoint variables for the Lagrangian.
- **curr_eq** (*numpy.ndarray*) – Current Lagrange multipliers for equality constraints.
- **curr_ineq** (*numpy.ndarray*) – Current Lagrange multipliers for inequality constraints.
- **curr_slack** (*numpy.ndarray*) – Current slack variables associated with inequality constraints.

Returns A string that that Kona will write into its info file.

Return type string, optional

eval_dFdU(*at_design*, *at_state*, *store_here*)

Evaluate the partial of the objective w.r.t. state variable at the design point stored in `at_design` and the state variables stored in `at_state`. Store the result in `store_here`.

Note: If there are no state variables, a zero vector must be stored.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.
- **store_here** (BaseVector) – Location where user should store the result.

eval_dFdX(*at_design*, *at_state*)

Evaluate the partial of the objective w.r.t. design variables at the design point stored in `at_design` and the state variables stored in `at_state`. Store the result in `store_here`.

Note: This method must be implemented for any problem type.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.

Returns Gradient vector.

Return type numpy.ndarray

eval_eq_cnstr(*at_design*, *at_state*)

Evaluate the vector of equality constraints using the given design and state vectors.

The constraints must have the form (c - c_target). In other words, the constraint value should be zero at feasibility.

- For inequality constraints, the constraint value should be greater than zero when feasible.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.

Returns **result** – Array of equality constraints.

Return type numpy.ndarray

eval_ineq_cnstr(*at_design*, *at_state*)

Evaluate the vector of inequality constraints using the given design and state vectors.

The constraints must have the form (c - c_target) > 0. In other words, the constraint value should be greater than zero when feasible.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.

Returns **result** – Array of equality constraints.

Return type numpy.ndarray

eval_obj(*at_design*, *at_state*)

Evaluate the objective function using the design variables stored at `at_design` and the state variables stored at `at_state`.

Note: This method must be implemented for any problem type.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.

Returns Result of the operation. Contains the objective value as the first element, and the number of preconditioner calls used as the second.

Return type tuple

eval_residual(*at_design*, *at_state*, *store_here*)

Evaluate the governing equations (discretized PDE residual) at the design variables stored in `at_design` and the state variables stored in `at_state`. Put the residual vector in `store_here`.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.
- **result** (BaseVector) – Location where user should store the result.

factor_linear_system(*at_design*, *at_state*)

OPTIONAL: Build/factor the dR/dU matrix and its preconditioner at the given design and state vectors, `at_design` and `at_state`. These matrices are then used to perform forward solves, adjoint solves and forward/transpose preconditioner applications.

This routine is only used by matrix-based solvers where matrix factorizations are costly and should be done only once per optimization iteration. The optimization options dictionary must have the `matrix_explicit` key set to `True`.

Note: If the user chooses to leverage this factorization, the (design, state) evaluation points should be ignored for preconditioner application, linear solve, and adjoint solve calls.

Parameters

- **at_design** (*numpy.ndarray*) – Current design vector.
- **at_state** (BaseVector) – Current state vector.

get_rank()

Rank of current process is needed purely for purposes of printing to screen

init_design()

Initialize the first design point. Store the design vector at `store_here`. The optimization will start from this point.

Note: This method must be implemented for any problem type.

Returns Initial design vector.

Return type numpy.ndarray

multiply_dCEQdU(*at_design*, *at_state*, *in_vec*)

Evaluate the matrix-vector product for the state-jacobian of the equality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{eq}(at_design, at_state)}{partial U} in_vec = out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [BaseVector] Vector to be operated on.

numpy.ndarray Result of the product

multiply_dCEQdU_T(*at_design*, *at_state*, *in_vec*, *out_vec*)

Evaluate the transposed matrix-vector product for the state-jacobian of the equality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{eq}(at_design, at_state)}{partial U}^T in_vec =

out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

out_vec [BaseVector] Location where user should store the result.

`multiply_dCEQdX`(*at_design*, *at_state*, *in_vec*)

Evaluate the matrix-vector product for the design-jacobian of the equality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{eq}(at_design, at_state)}{partial X} in_vec = out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

numpy.ndarray Result of the product.

`multiply_dCEQdX_T`(*at_design*, *at_state*, *in_vec*)

Evaluate the transposed matrix-vector product for the design-jacobian of the equality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{eq}(at_design, at_state)}{partial X}^T in_vec =

out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

numpy.ndarray Result of the product.

multiply_dCINdU(*at_design*, *at_state*, *in_vec*)

Evaluate the matrix-vector product for the state-jacobian of the inequality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{eq}(at_design, at_state)}{partial U} in_vec = out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [BaseVector] Vector to be operated on.

numpy.ndarray Result of the product

multiply_dCINdU_T(*at_design*, *at_state*, *in_vec*, *out_vec*)

Evaluate the transposed matrix-vector product for the state-jacobian of the inequality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{ineq}(at_design, at_state)}{partial U}^T in_vec =

out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

out_vec [BaseVector] Location where user should store the result.

multiply_dCINdX(*at_design*, *at_state*, *in_vec*)

Evaluate the matrix-vector product for the design-jacobian of the inequality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{ineq}(at_design, at_state)}{partial X} in_vec = out_vec

---

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

---

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

numpy.ndarray Result of the product.

`multiply_dCINdX_T`(*at_design*, *at_state*, *in_vec*)

Evaluate the transposed matrix-vector product for the design-jacobian of the inequality constraints. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial C_{ineq}(at_design, at_state)}{partial X}^T in_vec =

out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

---

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

numpy.ndarray Result of the product.

`multiply_dRdU`(*at_design*, *at_state*, *in_vec*, *out_vec*)

Evaluate the matrix-vector product for the state-jacobian of the PDE residual. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial R(at_design, at_state)}{partial U} in_vec = out_vec

---

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

---

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [BaseVector] Vector to be operated on.

out_vec [BaseVector] Location where user should store the result.

`multiply_dRdU_T`(*at_design*, *at_state*, *in_vec*, *out_vec*)

Evaluate the transposed matrix-vector product for the state-jacobian of the PDE residual. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial R(at_design, at_state)}{partial U}^T in_vec =

out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [BaseVector] Vector to be operated on.

out_vec [BaseVector] Location where user should store the result.

`multiply_dRdX` (*at_design*, *at_state*, *in_vec*, *out_vec*)

Evaluate the matrix-vector product for the design-jacobian of the PDE residual. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial R(at_design, at_state)}{partial X} in_vec = out_vec

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [numpy.ndarray] Vector to be operated on.

out_vec [BaseVector] Location where user should store the result.

`multiply_dRdX_T` (*at_design*, *at_state*, *in_vec*)

Evaluate the transposed matrix-vector product for the design-jacobian of the PDE residual. The multiplying vector is `in_vec` and the result should be stored in `out_vec`. The product should be evaluated at the given design and state vectors, `at_design` and `at_state` respectively.

rac{partial R(at_design, at_state)}{partial X}^T in_vec =

out_vec

Note: Must always store a result even when it isn't implemented. Use a zero vector of length `self.num_design` for this purpose.

Note: This jacobian is a partial. No total derivatives, gradients or jacobians should be evaluated by any UserSolver implementation.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector] Current state vector.

in_vec [BaseVector] Vector to be operated on.

numpy.ndarray Result of the operation

`solve_adjoint` (*at_design*, *at_state*, *rhs_vec*, *tol*, *result*)

Solve the linear system defined by the transposed state-jacobian of the PDE residual, to the specified absolute tolerance `tol`.

rac{partial R(at_design, at_state)}{partial U}^T result =

rhs_vec

The jacobian should be evaluated at the given (design, state) point, `at_design` and `at_state`.

Store the solution in `result`.

Note: If the solver uses `factor_linear_system()`, ignore the `at_design` evaluation point and use the previously factored preconditioner.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector-line] Current state vector.

rhs_vec [BaseVector] Right hand side vector.

rel_tol [float] Tolerance that the linear system should be solved to.

result [BaseVector] Location where user should store the result.

int Number of preconditioner calls required for the solution.

`solve_linear` (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

Solve the linear system defined by the state-jacobian of the PDE residual, to the specified absolute tolerance `tol`.

rac{partial R(at_design, at_state)}{partial U} result = rhs_vec

The jacobian should be evaluated at the given (design, state) point, `at_design` and `at_state`.

Store the solution in `result`.

Note: If the solver uses `factor_linear_system()`, ignore the `at_design` evaluation point and use the previously factored preconditioner.

at_design [numpy.ndarray] Current design vector.

at_state [BaseVector-line] Current state vector.

rhs_vec [BaseVector] Right hand side vector.

rel_tol [float] Tolerance that the linear system should be solved to.

result [BaseVector] Location where user should store the result.

int Number of preconditioner calls required for the solution.

`solve_nonlinear`(*at_design*, *result*)

Compute the state variables at the given design point, `at_design`. Store the resulting state variables in `result`.

For linear problems, this can be a simple linear system solution:

$$\mathcal{K}(x)\mathbf{u} = \mathbf{F}(x)$$

For nonlinear problems, this can involve Newton iterations:

rac{partial R(x, u_{guess})}{partual u} Delta u =

-R(x, u_{guess})

If the solution fails to converge, the user should return a negative integer in order to help Kona intelligently backtrack in the optimization.

Similarly, in the case of correct convergence, the user is encouraged to return the number of preconditioner calls it took to solve the nonlinear system. Kona uses this information to track the computational cost of the optimization. If the number of preconditioner calls is not available, return a 1 (one).

at_design [numpy.ndarray] Current design vector.

result [BaseVector] Location where user should store the result.

int Number of preconditioner calls required for the solution.

kona.algorithms

Subpackages

kona.algorithms.util

Submodules

kona.algorithms.util.linesearch

Subclasses

LineSearch (base class)

class `kona.algorithms.util.linesearch.`**`LineSearch`**(*optns={}*, *out_file=<open file '<stdout>', mode 'w'>*)

Bases: `object`

Base class for all line search algorithms. Provides rudimentary error-checking functionality, and an interface that should be adhered to when writing new line search functions.

Variables

- **`decr_cond`** (*`float`*) – Sufficient decrease condition.
- **`max_iter`** (*`int`*) – Maximum iterations for the line search.

- **`out_file`** (*`file`*) – File stream for writing data.

Parameters

- **`optns`** (*`dict`*) –
- **`out_file`** (*`file`*) –

`find_step_length`(*merit*)

Find an appropriate step size for the given merit function that leads to the minimum in the search direction.

Parameters **`merit`** (*`MeritFunc-like`*) – Merit function object derived from the base MeritFunc class.

Returns

- **float** (*Step size.*)
- **int** (*Number of iterations taken for the search.*)

BackTracking

class `kona.algorithms.util.linesearch.`**`BackTracking`**(*optns={}, out_file=<open file '<stdout>', mode 'w'>*)

Bases: *`kona.algorithms.util.linesearch.LineSearch`*

Back-tracking line search.

Variables

- **`alpha_init`** (*`float`*) – Initial step size.
- **`alpha_min`** (*`float`*) – Minimum step size.
- **`rdtn_factor`** (*`float`*) – Reduction factor for the step size at each iteration.
- **`p_dot_dfdx`** (*`float`*) – Value of $\langle p, \nabla f \rangle$ at current step.

`find_step_length`(*merit*)

StrongWolfe

class `kona.algorithms.util.linesearch.`**`StrongWolfe`**(*optns={}, out_file=<open file '<stdout>', mode 'w'>*)

Bases: *`kona.algorithms.util.linesearch.LineSearch`*

Strong Wolfe line search.

Variables

- **`alpha_init`** (*`float`*) – Initial step size.
- **`alpha_max`** (*`float`*) – Maximum step size.
- **`curv_cond`** (*`float`*) – Curvature condition to be satisfied.

`find_step_length`(*merit*)

kona.algorithms.util.merit

Subclasses

MeritFunction (base class)

class `kona.algorithms.util.merit.`**`MeritFunction`**(*primal_factory*, *state_factory*, *optns={}*, *out_file=<open file '<stdout>', mode 'w'>*)

Bases: `object`

Base class for all merit functions.

Variables

- **`primal_factory`** (`VectorFactory`) – Generator for new primal vectors.
- **`state_factory`** (`VectorFactory`) – Generator for new state vectors.
- **`out_file`** (*`file`*) – File stream for writing data.
- **`_allocated`** (*`boolean`*) – Flag to track whether merit function memory has been allocated.

Parameters

- **`primal_factory`** (`VectorFactory`) –
- **`state_factory`** (`VectorFactory`) –
- **`optns`** (*`dict`*) –
- **`out_file`** (*`file`*) –

`eval_func`(*alpha*)

Evaluate merit function value at `alpha`

Parameters **`alpha`** (*`float`*) –

Returns float

Return type Value of merit function `alpha`.

`eval_grad`(*alpha*)

Evaluate merit function gradient $\langle p, \nabla f \rangle$ at the given `alpha`

Note: This method can either `pass` or `return 0` for gradient-free merit functions.

Parameters **`alpha`** (*`float`*) –

Returns float

Return type Value of $\langle p, \nabla f \rangle$ at `alpha`.

`reset`(*search_dir*, *x_start*, *u_start*, *p_dot_grad=None*)

Reset the merit function at a new design and state point.

If merit memory is not yet allocated, this function should also do that.

Parameters

- **`search_dir`** (`DesignVector` *or* `CompositePrimalVector`) – Search direction vector in the primal space.
- **`x_start`** (`DesignVector`) – Initial primal vector.
- **`u_start`** (`StateVector`) – State vector corresponding to `x_start`.
- **`p_dot_grad`** (*`float, optional`*) – Value of $\langle p, \nabla f \rangle$ at `x_start`.

ObjectiveMerit

class `kona.algorithms.util.merit.`**`ObjectiveMerit`**(*primal_factory*, *state_factory*, *optns={}*, *out_file=<open file '<stdout>', mode 'w'>*)

Bases: *`kona.algorithms.util.merit.MeritFunction`*

Merit function for line searches applied to the raw objective.

Other, more complicated merit functions can be derived from this.

Variables

- **`last_func_alpha`** (*`float`*) – Last value of alpha for which objective value is evaluated.
- **`last_grad_alpha`** (*`float`*) – Last value of alpha for which objective grad is evaluated.
- **`func_val`** (*`float`*) – Value of the objective at `last_func_alpha`.
- **`p_dot_grad`** (*`float`*) – Value of $\langle p, \nabla f \rangle$ at `last_grad_alpha`.
- **`x_start`** (`DesignVector`) – Initial position of the primal variables, where $\alpha = 0$.
- **`x_trial`** (`DesignVector`) – Trial position of the primal variables at a new alpha.
- **`u_trial`** (`StateVector`) – Trial position of the state variables at a new alpha.
- **`search_dir`** (`DesignVector`) – The search direction vector.
- **`state_work`** (`StateVector`) – Work vector for state operations.
- **`adjoint_work`** (`StateVector`) – Work vector for adjoint operations.
- **`design_work`** (`DesignVector`) – Work vector for primal operations.

`eval_func`(*alpha*)

`eval_grad`(*alpha*)

`reset`(*search_dir*, *x_start*, *u_start*, *p_dot_grad*)

L2QuadraticPenalty

class `kona.algorithms.util.merit.`**`L2QuadraticPenalty`**(*primal_factory*, *state_factory=None*, *eq_factory=None*, *ineq_factory=None*, *optns={}*, *out_file=<open file '<stdout>', mode 'w'>*)

Bases: *`kona.algorithms.util.merit.MeritFunction`*

A merit function with L2 constraint norm pernalty term, used for constrained RSNK problems.

The merit function is defined as:

$$(M)(x, s) = f(x, u(x)) + \frac{1}{2}\mu||c_{eq}(x, u(x))||^2 + \frac{1}{2}\mu||c_{in}(x, u(x)) - s||^2$$

eval_func (*alpha*)

reset (*kkt_start*, *u_start*, *search_dir*, *mu*)

AugmentedLagrangian

class kona.algorithms.util.merit.**AugmentedLagrangian** (*primal_factory*, *state_factory*, *eq_factory=None*, *ineq_factory=None*, *optns={}*, *out_file=<open file '<stdout>', mode 'w'>*)

Bases: *kona.algorithms.util.merit.L2QuadraticPenalty*

An augmented Lagrangian merit function for constrained RSNK problems.

The augmented Lagrangian is defined as:

$$\hat{\mathcal{L}}(x, s) = f(x, u(x)) + \lambda_{eq}^T c_{eq}(x, u(x)) + \lambda_{in}^T \left[c_{in}(x, u(x)) - s\right] + \frac{1}{2}\mu||c_{eq}(x, u(x))||^2 + \frac{1}{2}\mu||c_{in}(x, u(x)) - s||^2$$

Unlike the traditional augmented Lagrangian, the Kona version has the Lagrange multipliers and the slack variables fozen. This is done to make the merit function comparable to the predicted decrease produced by the FLECS solver.

eval_func (*alpha*)

reset (*kkt_start*, *u_start*, *search_dir*, *mu*)

kona.algorithms.base_algorithm

Subclasses

Package contents

Unconstrained Reduced-Space Quasi-Newton

class kona.algorithms.**ReducedSpaceQuasiNewton** (*primal_factory*, *state_factory*, *eq_factory*, *ineq_factory*, *optns=None*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

Unconstrained optimization using quasi-Newton in the reduced space, globalized using either back-tracking or Strong Wolfe line search on the objective as the merit function.

This algorithm can leverage both limited-memory BFGS and limited-memory Symmetric Rank 1 approximations of the Hessian.

Variables

- **factor_matrices** (*bool*) – Boolean flag for matrix-based PDE solvers.
- **min_radius, max_radius** (*radius,*) – Trust radius parameters.
- **mu_init, mu_max, mu_pow, eta** (*mu,*) – Augmented Lagrangian constraint factor parameters.

- **grad_scale, feas_scale** (*scale,*) – Optimality metric normalization factors.
- **approx_hessian** (QuasiNewtonApprox-like) – The quasi-Newton approximation object for the Hessian.
- **globalization** (*string*) – Flag to determine solution globalization type.

solve()

Unconstrained STCG-based RSNK

class kona.algorithms.**UnconstrainedRSNK**(*primal_factory*, *state_factory*, *eq_factory*, *ineq_factory*, *optns=None*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

A reduced-space Newton-Krylov optimization algorithm for PDE-governed unconstrained problems.

This algorithm uses a 2nd order adjoint formulation to compute matrix-vector products with the Reduced Hessian.

The product is then used in a Krylov solver to compute a Newton step.

The step can be globalized using either line-search or trust-region methods. The Krylov solver changes based on the type of globalization selected by the user. Unglobalized problems are solved via FGMRES, while trust-region and line-search methods use Conjugate-Gradient.

Note: Insert inexact-Hessian paper reference here.

Variables

- **factor_matrices** (*bool*) – Boolean flag for matrix-based PDE solvers.
- **iter** (*int*) – Optimization iteration counter.
- **hessian** (*ReducedHessian*) – Matrix object defining the Hessian matrix-vector product.
- ***precond*** (*BaseHessian*-like) – Matrix object defining the approximation to the Hessian inverse.
- **krylov** (*FGMRES* or *STCG*) – A krylov solver object used to solve the system defined by the Hessian.
- **globalization** (*string*) – Flag to determine which type of globalization to use.
- **max_radius** (*radius,*) – Trust radius parameters.
- **line_search** (*BackTracking*) – Back-tracking line search tool.
- **merit_func** (*ObjectiveMerit*) – Simple objective as merit function.

solve()

Unconstrained Predictor-Corrector

class kona.algorithms.**PredictorCorrector**(*primal_factory*, *state_factory*, *eq_factory=None*, *ineq_factory=None*, *optns=None*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

A reduced-space Newton-Krylov algorithm for PDE-governed unconstrained optimization, globalized in a predictor-corrector homotopy path following framework.

This implementation is loosely based on the predictor-corrector method described by **'Brown and Zingg<http://www.sciencedirect.com/science/article/pii/S0021999116301760>'_** for nonlinear computational fluid dynamics problems.

The homotopy map used in this algorithm is given as:

$$mathcal H(x, u) = (1 - \lambda)F(x, u) + \lambda$$

rac{1}{2}(x - x_0)^T(x - x_0)

where x_0 is the initial design point.

factor_matrices [bool] Boolean flag for matrix-based PDE solvers.

lamb, inner_tol, step, nom_dcurv, nom_angl, max_factor, min_factor [float] Homotopy parameters.

scale, grad_scale, feas_scale [float] Optimality metric normalization factors.

hessian [*ReducedHessian*] Matrix object defining the Hessian matrix-vector product.

precond [*BaseHessian*-like] Matrix object defining the approximation to the Hessian inverse.

krylov [*FGMRES*] A krylov solver object used to solve the system defined by the Hessian.

solve()

Equality Constrained Predictor-Corrector

class kona.algorithms.**PredictorCorrectorCnstr**(*primal_factory*, *state_factory*, *eq_factory=None*, *ineq_factory=None*, *optns=None*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

A reduced-space Newton-Krylov algorithm for PDE-governed equality constrained optimization, globalized in a predictor-corrector homotopy path following framework.

This implementation is loosely based on the predictor-corrector method described by **'Brown and Zingg<http://www.sciencedirect.com/science/article/pii/S0021999116301760>'_** for nonlinear computational fluid dynamics problems.

The homotopy map used in this algorithm is given as:

$$mathcal H(x, u) = \mu L(x, u) + (1 - \mu)$$

rac{1}{2} left[

(x - x_0)^T(x - x_0) - (lambda - lambda_0)^T(lambda - lambda_0)

where x_0 is the initial design point and λ_0 is the initial Lagrange multipliers.

factor_matrices [bool] Boolean flag for matrix-based PDE solvers.

mu, inner_tol, step, nom_dcurv, nom_angl, max_factor, min_factor [float] Homotopy parameters.

scale, grad_scale, feas_scale [float] Optimality metric normalization factors.

hessian [*ReducedKKTMatrix*] Matrix object defining the KKT matrix-vector product.

precond [*BaseHessian*-like] Matrix object defining the approximation to the Hessian inverse.

krylov [*FGMRES*] Krylov solver object used to solve the system defined by the KKT matrix-vector product.

solve()

Equality Constrained FLECS-based RSNK

class kona.algorithms.**ConstrainedRSNK**(*primal_factory*, *state_factory*, *eq_factory*, *ineq_factory*, *optns=None*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

A reduced-space Newton-Krylov optimization algorithm for PDE-governed equality constrained problems, globalized with a trust-region approach.

This algorithm uses a 2nd order adjoint formulation of the KKT matrix-vector product, in conjunction with a novel Krylov-method called **'FLECS<http://dx.doi.org/10.1137/140994496>'_** for non-convex saddle point problems.

More information on this reduced-space Newton-Krylov appoach can be found in this paper.

Variables

- **feas_norm0, kkt_norm0** (*grad_norm0,*) – Initial optimality norms.
- **iter** (*int*) – Optimization iteration counter.
- **factor_matrices** (*bool*) – Boolean flag for matrix-based PDE solvers.
- **min_radius, max_radius** (*radius,*) – Trust radius parameters.
- **mu_init, mu_max, mu_pow, eta** (*mu,*) – Augmented Lagrangian constraint factor parameters.
- **grad_scale, feas_scale** (*scale,*) – Optimality metric normalization factors.
- **KKT_matrix** (ReducedKKTVector) – Matrix object defining the KKT matrix-vector product.
- ***precond*** (*BaseHessian*-like) – Matrix object defining the preconditioner to the KKT system.
- **krylov** (*FLECS*) – A krylov solver object used to solve the system defined by this matrix.
- **globalization** (*string*) – Flag to determine solution globalization type.

filter_step(*X*, *state*, *P*, *kkt_rhs*, *kkt_work*, *state_work*, *dual_work*)

solve()

trust_step(*X*, *state*, *adjoint*, *P*, *kkt_rhs*, *state_work*, *dual_work*, *kkt_work*, *kkt_save*)

Equality Constrained Composite-Step RSNK

class kona.algorithms.**CompositeStepRSNK**(*primal_factory*, *state_factory*, *eq_factory*, *ineq_factory*, *optns={}*)

Bases: kona.algorithms.base_algorithm.OptimizationAlgorithm

A reduced-space composite-step optimization algorithm for PDE-governed equality constrained problems, globalized using a trust-region approach.

This implementation is based on the composite-step algorithm proposed by **‘Heinkenschloss and Ridzal<http://epubs.siam.org/doi/abs/10.1137/130921738>‘_**. However, we have omitted the inexactness corrections for simplicity and implemented a 2nd order adjoint approach for producing the necessary matrix-vector products.

Variables

- **`factor_matrices`** (*`bool`*) – Boolean flag for matrix-based PDE solvers.
- **`min_radius, max_radius`** (*`radius,`*) – Trust radius parameters.
- **`mu_max, mu_pow`** (*`mu,`*) – Augmented Lagrangian constraint factor parameters.
- **`normal_KKT`** (*`AugmentedKKTMatrix`*) – Matrix object for the normal step system.
- **`tangent_KKT`** (*`LagrangianHessian`*) – Matrix object for the tangent step system.
- **`globalization`** (*`string`*) – Flag to determine solution globalization type.

`backtracking_step`()

`calc_pred_reduction`()

`eval_merit`(*design*, *state*, *dual*, *cnstr*)

`solve`()

`trust_step`()

Verifier

class `kona.algorithms.`**`Verifier`**(*primal_factory*, *state_factory*, *eq_factory*, *ineq_factory*, *optns=None*)

Bases: `kona.algorithms.base_algorithm.OptimizationAlgorithm`

This is a verification tool that performs finite-difference checks on the provided solver to make sure that the required tasks have been implemented correctly by the user.

Variables

- **`out_stream`** (*`file`*) – File handle for verification output.
- **`factor_matrices`** (*`bool`*) – Boolean flag for matrix-based PDE solvers.
- **`exit_verify`** (*`warnings_flagged,`*) – Flags for terminating verification.
- **`failures`** (*`dict`*) – Dictionary containing verification results.
- **`non_critical, all_tests`** (*`critical,`*) – Lists of dictionary key names for critical, non-critical, and complete verification tests.

`solve`()

kona.examples

Subclasses

Rosenbrock (multidimensional)

class `kona.examples.`**`Rosenbrock`**(*num_design*, *num_state=0*, *num_eq=0*, *num_ineq=0*)

Bases: `kona.user.user_solver.UserSolver`

eval_dFdX(*at_design*, *at_state*)

eval_obj(*at_design*, *at_state*)

init_design()

Simple2x2 (with state variables)

class `kona.examples.`**Simple2x2**
Bases: `kona.user.user_solver.UserSolver`

eval_dFdU(*at_design*, *at_state*, *store_here*)

eval_dFdX(*at_design*, *at_state*)

eval_obj(*at_design*, *at_state*)

eval_residual(*at_design*, *at_state*, *store_here*)

init_design()

multiply_dRdU(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdU_T(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX_T(*at_design*, *at_state*, *in_vec*)

solve_adjoint(*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_linear(*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_nonlinear(*at_design*, *result*)

Spiral

class `kona.examples.spiral.`**SpiralSolver**
Bases: `object`

F

R

alpha

dFdU

dFdX

dRdU

dRdX

linearize(*at_design*, *at_state=None*)

rhs

theta

class `kona.examples.`**Spiral**
Bases: `kona.user.user_solver.UserSolver`

eval_dFdU(*at_design*, *at_state*, *store_here*)

eval_dFdX(*at_design*, *at_state*)

eval_obj(*at_design*, *at_state*)

eval_residual(*at_design*, *at_state*, *store_here*)

init_design()

multiply_dRdU(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdU_T(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX(*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX_T(*at_design*, *at_state*, *in_vec*)

solve_adjoint(*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_linear(*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_nonlinear(*at_design*, *result*)

SphereConstrained

class `kona.examples.`**SphereConstrained**(*init_x=[0.51, 0.52, 0.53]*, *ineq=False*)

Bases: `kona.user.user_solver.UserSolver`

current_solution(*num_iter*, *curr_design*, *curr_state*, *curr_adj*, *curr_eq*, *curr_ineq*, *curr_slack*)

eval_cnstr(*at_design*, *at_state*)

eval_dFdU(*at_design*, *at_state*, *store_here*)

eval_dFdX(*at_design*, *at_state*)

eval_eq_cnstr(*at_design*, *at_state*)

eval_ineq_cnstr(*at_design*, *at_state*)

eval_obj(*at_design*, *at_state*)

eval_residual(*at_design*, *at_state*, *store_here*)

init_design()

multiply_dCEQdX(*at_design*, *at_state*, *in_vec*)

multiply_dCEQdX_T(*at_design*, *at_state*, *in_vec*)

multiply_dCINdX(*at_design*, *at_state*, *in_vec*)

multiply_dCINdX_T(*at_design*, *at_state*, *in_vec*)

multiply_dCdX(*at_design*, *at_state*, *in_vec*)

multiply_dCdX_T(*at_design*, *at_state*, *in_vec*)

ExponentialConstrained

class `kona.examples.`**ExponentialConstrained**(*init_x=[1.0, 1.0]*)

Bases: `kona.user.user_solver.UserSolver`

eval_dFdU(*at_design*, *at_state*, *store_here*)

eval_dFdX(*at_design*, *at_state*)

eval_eq_cnstr(*at_design*, *at_state*)

eval_obj (*at_design*, *at_state*)

eval_residual (*at_design*, *at_state*, *store_here*)

init_design ()

multiply_dCEQdX (*at_design*, *at_state*, *in_vec*)

multiply_dCEQdX_T (*at_design*, *at_state*, *in_vec*)

Constrained2x2

class `kona.examples.`**Constrained2x2**

Bases: `kona.user.user_solver.UserSolver`

eval_dFdU (*at_design*, *at_state*, *store_here*)

eval_dFdX (*at_design*, *at_state*)

eval_eq_cnstr (*at_design*, *at_state*)

eval_obj (*at_design*, *at_state*)

eval_residual (*at_design*, *at_state*, *store_here*)

init_design ()

multiply_dCEQdU (*at_design*, *at_state*, *in_vec*)

multiply_dCEQdU_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dCEQdX (*at_design*, *at_state*, *in_vec*)

multiply_dCEQdX_T (*at_design*, *at_state*, *in_vec*)

multiply_dRdU (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdU_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX_T (*at_design*, *at_state*, *in_vec*)

solve_adjoint (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_linear (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_nonlinear (*at_design*, *result*)

SimpleMDF

class `kona.examples.`**SimpleMDF** (*num_disc=5*, *init_x=5.0*)

Bases: `kona.user.user_solver.UserSolver`

eval_dFdU (*at_design*, *at_state*, *store_here*)

eval_dFdX (*at_design*, *at_state*)

eval_obj (*at_design*, *at_state*)

eval_residual (*at_design*, *at_state*, *store_here*)

init_design ()

multiply_dRdU (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdU_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX_T (*at_design*, *at_state*, *in_vec*)

solve_adjoint (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_linear (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_nonlinear (*at_design*, *result*)

SimpleIDF

class `kona.examples.`**SimpleIDF** (*num_disc=5*, *init_x=5.0*, *approx_inv=True*)

Bases: `kona.user.user_solver.UserSolverIDF`

apply_precond (*at_design*, *at_state*, *in_vec*, *out_vec*)

apply_precond_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

eval_dFdU (*at_design*, *at_state*, *store_here*)

eval_dFdX (*at_design*, *at_state*)

eval_eq_cnstr (*at_design*, *at_state*)

eval_obj (*at_design*, *at_state*)

eval_residual (*at_design*, *at_state*, *store_here*)

init_design ()

multiply_dCEQdU (*at_design*, *at_state*, *in_vec*)

multiply_dCEQdU_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dCEQdX (*at_design*, *at_state*, *in_vec*)

multiply_dCEQdX_T (*at_design*, *at_state*, *in_vec*)

multiply_dRdU (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdU_T (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX (*at_design*, *at_state*, *in_vec*, *out_vec*)

multiply_dRdX_T (*at_design*, *at_state*, *in_vec*)

solve_adjoint (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_linear (*at_design*, *at_state*, *rhs_vec*, *rel_tol*, *result*)

solve_nonlinear (*at_design*, *result*)

CHAPTER 7

Indices and Tables

- genindex
- modindex

CHAPTER 8

Acknowledgements

This work is supported by the National Science Foundation under Grant No. 1332819, and the National Aeronautics and Space Administration under Grant No. NNX14AC73A.

Index