Fides Documentation

Release 0.0.2

The Fides developers

Nov 13, 2020

API REFERENCE

Version: 0.0.2
Source code: https://github.com/Fides-dev/fides

CHAPTER

ONE

FIDES PYTHON API

Modules

fides	Fides
fides.minimize	Minimization
fides.trust_region	Trust Region Steps
fides.subproblem	Subproblem Solvers
fides.hessian_approximation	Hessian Update Strategies
fides.logging	Logging
fides.constants	Constants

1.1 Fides

Fides is an interior trust-region reflective optimizer

1.2 Minimization

This module performs the optimization given a step proposal.

Classes Summary

Optimizer(fun, ub, lb[, verbose, options, ...])	Performs optimization

Classes

class fides.minimize.**Optimizer**(*fun*, *ub*, *lb*, *verbose=10*, *options=None*, *funargs=None*, *hessian_update=None*)

Performs optimization

Variables

- **fun** – objective function
- **funargs** – keyword arguments that are passed to the function
- **lb** – lower optimization boundaries

- **ub** – upper optimization boundaries
- **options** – options that configure convergence checks
- **delta** – trust region radius
- **x** – current optimization variables
- **fval** – objective function value at x
- **grad** – objective function gradient at x
- **hess** – objective function Hessian (approximation) at x
- **hessian_update** – object that performs hessian updates
- **starttime** – time at which optimization was started
- **iteration** – current iteration
- **converged** – flag indicating whether optimization has converged

__init__ (*fun*, *ub*, *lb*, *verbose=10*, *options=None*, *funargs=None*, *hessian_update=None*)
Create an optimizer object

Parameters

- **fun** (typing.Callable) – This is the objective function, if no *hessian_update* is provided, this function must return a tuple (fval, grad), otherwise this function must return a tuple (fval, grad, Hessian)
- **ub** (numpy.ndarray) – Upper optimization boundaries. Individual entries can be set to np.inf for respective variable to have no upper bound
- **lb** (numpy.ndarray) – Lower optimization boundaries. Individual entries can be set to -np.inf for respective variable to have no lower bound
- **verbose** (typing.Optional[int]) – Verbosity level, pick from logging.[DEBUG,INFO,WARNING,ERROR]
- **options** (typing.Optional[typing.Dict]) – Options that control termination of optimization. See *minimize* for details.
- **funargs** (typing.Optional[typing.Dict]) – Additional keyword arguments that are to be passed to fun for evaluation
- **hessian_update** (typing.Optional[*fides.hessian_approximation.HessianApproximation*]) – Subclass of fides.hessian_update.HessianApproximation that performs the hessian updates in every iteration.

check_continue()
Checks whether minimization should continue based on convergence, iteration count and remaining computational budget

Return type bool

Returns flag indicating whether minimization should continue

check_convergence (*fval*, *x*, *grad*)
Check whether optimization has converged.

Parameters

- **fval** – updated objective function value
- **x** – updated optimization variables

- **grad** – updated objective function gradient

Return type None

check_finite()
Checks whether objective function value, gradient and Hessian (approximation) have finite values and optimization can continue.

Raises RuntimeError if any of the variables have non-finite entries

check_in_bounds(*x=None*)
Checks whether the current optimization variables are all within the specified boundaries

Raises RuntimeError if any of the variables are not within boundaries

get_affine_scaling()
Computes the vector v and dv, the diagonal of it's Jacobian. For the definition of v, see Definition 2 in [Coleman-Li1994]

Return type typing.Tuple[numpy.ndarray, numpy.ndarray]

Returns v scaling vector dv diagonal of the Jacobian of v wrt x

log_header()
Prints the header for diagnostic information, should complement *Optimizer.log_step()*.

log_step(*accepted*, *steptype*, *normdx*)
Prints diagnostic information about the current step to the log

Parameters

- **accepted** (bool) – flag indicating whether the current step was accepted
- **steptype** (str) – identifier how the current step was computed
- **normdx** (float) – norm of the current step

make_non_degenerate(*eps=2.220446049250313e-14*)
Ensures that x is non-degenerate, this should only be necessary for initial points.

Parameters **eps** – degeneracy threshold

Return type None

minimize(*x0*)
Minimize the objective function the interior trust-region reflective algorithm described by [ColemanLi1994] and [ColemanLi1996] Convergence with respect to function value is achieved when math:|*f_{k+1} - f_k*| < options[*fatol*] - f_k options[*frtol*]. Similarly, convergence with respect to optimization variables is achieved when $||x_{k+1} - x_k||$ < options[*xatol*] - x_k options[*xrtol*]. Convergence with respect to the gradient is achieved when $||g_k||$ < options[*gatol*] or ||*g_k*|| < options[*grtol*] * *f_k*. Other than that, optimization can be terminated when iterations exceed options[*maxiter*] or the elapsed time is expected to exceed options[*maxtime*] on the next iteration.

Parameters **x0** (numpy.ndarray) – initial guess

Returns fval: final function value, x: final optimization variable values, grad: final gradient, hess: final Hessian (approximation)

update_tr_radius(*fval*, *grad*, *step_sx*, *dv*, *qppred*)
Update the trust region radius

Parameters

- **fval** – new function value if step defined by step_sx is taken
- **grad** – new gradient value if step defined by step_sx is taken

- **step_sx** – proposed scaled step
- **dv** – derivative of scaling vector v wrt x
- **qppred** – predicted objective function value according to the quadratic approximation

Return type bool

Returns flag indicating whether the proposed step should be accepted

1.3 Trust Region Steps

This module provides the machinery to compute different trust-region(-reflective) step proposals and select among them based on to their performance according to the quadratic approximation of the objective function

Classes Summary

GradientStep(x, sg, hess, scaling, ...)	This class provides the machinery to compute a gradient step.
Step(x, sg, hess, scaling, g_dscaling, ...)	Base class for the computation of a proposal step
TRStep2D(x, sg, hess, scaling, g_dscaling, ...)	This class provides the machinery to compute an approximate solution of the trust region subproblem according to a 2D subproblem
TRStepFull(x, sg, hess, scaling, g_dscaling, ...)	This class provides the machinery to compute an exact solution of the trust region subproblem.
TRStepReflected(x, sg, hess, scaling, ...)	This class provides the machinery to compute a reflected step based on trust region subproblem solution thathit the boundaries.

Functions Summary

normalize(v)	Inplace normalization of a vector
trust_region_reflective(x, g, hess, scaling, ...)	Compute a step according to the solution of the trust-region subproblem.

Classes

class fides.trust_region.**GradientStep**(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)

This class provides the machinery to compute a gradient step.

__init__(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)

Parameters

- **x** – Reference point
- **sg** – Gradient in rescaled coordinates
- **hess** – Hessian in unscaled coordinates
- **scaling** – Matrix that defines scaling transformation
- **g_dscaling** – Unscaled gradient multiplied by derivative of scaling transformation

- **delta** – Trust region Radius in scaled coordinates
- **theta** – Stepback parameter that controls how close steps are allowed to get to the boundary
- **ub** – Upper boundary
- **lb** – Lower boundary

class fides.trust_region.**Step**(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)

Base class for the computation of a proposal step

Variables

- **x** – current state of optimization variables
- **s** – proposed step
- **sc** – coefficients in the 1D/2D subspace that defines the affine transformed step ss: ss = subspace * sc
- **ss** – affine transformed step: s = scaling * ss
- **og_s** – s without step back
- **og_sc** – st without step back
- **og_ss** – ss without step back
- **sg** – rescaled gradient scaling * g
- **hess** – hessian of the objective function at x
- **g_dscaling** – diag(g) * dscaling
- **delta** – trust region radius in the transformed space defined by scaling matrix
- **theta** – controls step back, fraction of step to take if full step would reach breakpoint
- **lb** – lower boundaries for x
- **ub** – upper boundaries for x
- **minbr** – maximal fraction of step s that can be taken to reach first breakpoint
- **ipt** – index of x that specifies the variable that will hit the breakpoint if a step minbr * s is taken
- **qpval0** – value to the quadratic subproblem at x
- **qpval** – value of the quadratic subproblem for the proposed step
- **shess** – matrix of the full quadratic problem
- **cg** – projection of the g_hat to the subspace
- **chess** – projection of the B to the subspace

__init__(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)

Parameters

- **x** (numpy.ndarray) – Reference point
- **sg** (numpy.ndarray) – Gradient in rescaled coordinates
- **hess** (numpy.ndarray) – Hessian in unscaled coordinates
- **scaling** (scipy.sparse.csc.csc_matrix) – Matrix that defines scaling transformation

- **g_dscaling** (scipy.sparse.csc.csc_matrix) – Unscaled gradient multiplied by derivative of scaling transformation
- **delta** (float) – Trust region Radius in scaled coordinates
- **theta** (float) – Stepback parameter that controls how close steps are allowed to get to the boundary
- **ub** (numpy.ndarray) – Upper boundary
- **lb** (numpy.ndarray) – Lower boundary

calculate()
Calculates step and the expected objective function value according to the quadratic approximation

compute_step()
Compute the step as solution to the trust region subproblem. Special code is used for the special case 1-dimensional subspace case

Return type None

reduce_to_subspace()
This function projects the matrix shess and the vector sg to the subspace

Return type None

step_back()
This function truncates the step based on the distance of the current point to the boundary.

class fides.trust_region.**TRStep2D**(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*, *subspace*)
This class provides the machinery to compute an approximate solution of the trust region subproblem according to a 2D subproblem

__init__(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*, *subspace*)

Parameters **subspace** – Precomputed subspace

class fides.trust_region.**TRStepFull**(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)
This class provides the machinery to compute an exact solution of the trust region subproblem.

__init__(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*)

Parameters

- **x** – Reference point
- **sg** – Gradient in rescaled coordinates
- **hess** – Hessian in unscaled coordinates
- **scaling** – Matrix that defines scaling transformation
- **g_dscaling** – Unscaled gradient multiplied by derivative of scaling transformation
- **delta** – Trust region Radius in scaled coordinates
- **theta** – Stepback parameter that controls how close steps are allowed to get to the boundary
- **ub** – Upper boundary
- **lb** – Lower boundary

class fides.trust_region.**TRStepReflected**(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*, *tr_step*)

This class provides the machinery to compute a reflected step based on trust region subproblem solution thathit the boundaries.

__init__(*x*, *sg*, *hess*, *scaling*, *g_dscaling*, *delta*, *theta*, *ub*, *lb*, *tr_step*)

Parameters **tr_step** – Trust-region step that is reflected

Functions

fides.trust_region.**normalize**(*v*)

Inplace normalization of a vector

Parameters **v** (numpy.ndarray) – vector to be normalized

Return type None

fides.trust_region.**trust_region_reflective**(*x*, *g*, *hess*, *scaling*, *tr_subspace*, *delta*, *dv*, *theta*, *lb*, *ub*, *subspace_dim*)

Compute a step according to the solution of the trust-region subproblem. If step-back is necessary, gradient and reflected trust region step are also evaluated in terms of their performance according to the local quadratic approximation

Parameters

- **x** (numpy.ndarray) – Current values of the optimization variables
- **g** (numpy.ndarray) – Objective function gradient at x
- **hess** (numpy.ndarray) – (Approximate) objective function Hessian at x
- **scaling** (scipy.sparse.csc.csc_matrix) – Scaling transformation according to distance to boundary
- **tr_subspace** (numpy.ndarray) – Precomputed subspace from previous iteration for reuse if proposed step was not accepted
- **delta** (float) – Trust region radius, note that this applies after scaling transformation
- **dv** (numpy.ndarray) – derivative of scaling transformation
- **theta** (float) – parameter regulating stepback
- **lb** (numpy.ndarray) – lower optimization variable boundaries
- **ub** (numpy.ndarray) – upper optimization variable boundaries
- **subspace_dim** (*fides.constants.SubSpaceDim*) – Subspace dimension in which the subproblem will be solved. Larger subspaces require more compute time but can yield higher quality step proposals.

Return type typing.Tuple[numpy.ndarray, numpy.ndarray, float, numpy.ndarray, str]

Returns s: proposed step, ss: rescaled proposed step, qpval: expected function value according to local quadratic approximation, subspace: computed subspace for reuse if proposed step is not accepted, steptype: type of step that was selected for proposal

1.4 Subproblem Solvers

This module provides the machinery to solve 1- and N-dimensional trust-region subproblems.

Functions Summary

dsecular(lam, w, eigvals, eigvecs, delta)	Derivative of the secular equation
dslam(lam, w, eigvals, eigvecs)	Computes the derivative of the solution $s(\lambda)$ with respect to lambda, where s is the ubproblem solution according to
secular(lam, w, eigvals, eigvecs, delta)	Secular equation
slam(lam, w, eigvals, eigvecs)	Computes the solution $s(\lambda)$ as subproblem solution according to
solve_1d_trust_region_subproblem(B, g, s, delta)	Solves the special case of a one-dimensional subproblem
solve_nd_trust_region_subproblem(B, g, delta)	This function exactly solves the n-dimensional subproblem.

Functions

fides.subproblem.**dsecular**(*lam*, *w*, *eigvals*, *eigvecs*, *delta*)

Derivative of the secular equation

$\phi(\lambda) = \frac{1}{||s||} - \frac{1}{\Delta}$

with respect to λ

Parameters

- **lam** (float) – λ
- **w** (numpy.ndarray) – precomputed eigenvector coefficients for -g
- **eigvals** (numpy.ndarray) – precomputed eigenvalues of B
- **eigvecs** (numpy.ndarray) – precomputed eigenvectors of B
- **delta** (float) – trust region radius Δ

Returns $\frac{\partial \phi(\lambda)}{\partial \lambda}$

fides.subproblem.**dslam**(*lam*, *w*, *eigvals*, *eigvecs*)

Computes the derivative of the solution $s(\lambda)$ with respect to lambda, where s is the ubproblem solution according to

$-(B + \lambda I)s = g$

Parameters

- **lam** (float) – λ
- **w** (numpy.ndarray) – precomputed eigenvector coefficients for -g
- **eigvals** (numpy.ndarray) – precomputed eigenvalues of B
- **eigvecs** (numpy.ndarray) – precomputed eigenvectors of B

Returns $\frac{\partial s(\lambda)}{\partial \lambda}$

fides.subproblem.**secular**(*lam*, *w*, *eigvals*, *eigvecs*, *delta*)

Secular equation

$\phi(\lambda) = \frac{1}{||s||} - \frac{1}{\Delta}$

Subproblem solutions are given by the roots of this equation

Parameters

- **lam** (float) – λ
- **w** (numpy.ndarray) – precomputed eigenvector coefficients for -g
- **eigvals** (numpy.ndarray) – precomputed eigenvalues of B
- **eigvecs** (numpy.ndarray) – precomputed eigenvectors of B
- **delta** (float) – trust region radius Δ

Returns $\phi(\lambda)$

fides.subproblem.**slam**(*lam*, *w*, *eigvals*, *eigvecs*)

Computes the solution $s(\lambda)$ as subproblem solution according to

$-(B + \lambda I)s = g$

Parameters

- **lam** (float) – λ
- **w** (numpy.ndarray) – precomputed eigenvector coefficients for -g
- **eigvals** (numpy.ndarray) – precomputed eigenvalues of B
- **eigvecs** (numpy.ndarray) – precomputed eigenvectors of B

Return type numpy.ndarray

Returns $s(\lambda)$

fides.subproblem.**solve_1d_trust_region_subproblem**(*B*, *g*, *s*, *delta*)

Solves the special case of a one-dimensional subproblem

Parameters

- **B** (numpy.ndarray) – Hessian of the quadratic subproblem
- **g** (numpy.ndarray) – Gradient of the quadratic subproblem
- **s** (numpy.ndarray) – Vector defining the one-dimensional search direction
- **delta** (float) – Norm boundary for the solution of the quadratic subproblem

Return type numpy.ndarray

Returns Proposed step-length

fides.subproblem.**solve_nd_trust_region_subproblem**(*B*, *g*, *delta*)

This function exactly solves the n-dimensional subproblem.

$argmin_s\{s^T Bs + s^T g = 0 : ||s|| <= \Delta, s \in \mathbb{R}^n\}$

The solution to is characterized by the equation $-(B + \lambda I)s = g$. If B is positive definite, the solution can be obtained by $\lambda = 0$ if $Bs = -g$ satisfies $||s|| <= \Delta$. If B is indefinite or $Bs = -g$ satisfies $||s|| > \Delta$ and an appropppriate λ has to be identified via 1D rootfinding of the secular equation

$\phi(\lambda) = \frac{1}{||s(\lambda)||} - \frac{1}{\Delta} = 0$

with $s(\lambda)$ computed according to an eigenvalue decomposition of B. The eigenvalue decomposition, although being more expensive than a cholesky decomposition, has the advantage that eigenvectors are invariant to changes in λ and eigenvalues are linear in λ, so factorization only has to be performed once. We perform the linesearch via Newton's algorithm and Brent-Q as fallback. The hard case is treated seperately and serves as general fallback.

Parameters

- **B** (numpy.ndarray) – Hessian of the quadratic subproblem
- **g** (numpy.ndarray) – Gradient of the quadratic subproblem
- **delta** (float) – Norm boundary for the solution of the quadratic subproblem

Return type typing.Tuple[numpy.ndarray, str]

Returns s: Selected step, step_type: Type of solution that was obtained

1.5 Hessian Update Strategies

This module provides various generic Hessian approximation strategies that can be employed when the calculating the exact Hessian or an approximation is computationally too demandind.

Classes Summary

BFGS(dim[, hess_init])	Broyden-Fletcher-Goldfarb-Shanno update strategy.
DFP(dim[, hess_init])	Davidon-Fletcher-Powell update strategy.
HessianApproximation(dim[, hess_init])	Abstract class from which Hessian update strategies should subclass
SR1(dim[, hess_init])	Symmetric Rank 1 update strategy.

Classes

class fides.hessian_approximation.**BFGS**(*dim*, *hess_init=None*)
Broyden-Fletcher-Goldfarb-Shanno update strategy. This is a rank 2 update strategy that always yields positive-semidefinite hessian approximations.

class fides.hessian_approximation.**DFP**(*dim*, *hess_init=None*)
Davidon-Fletcher-Powell update strategy. This is a rank 2 update strategy that always yields positive-semidefinite hessian approximations. It usually does not perform as well as the BFGS strategy, but included for the sake of completeness.

class fides.hessian_approximation.**HessianApproximation**(*dim*, *hess_init=None*)
Abstract class from which Hessian update strategies should subclass

__init__(*dim*, *hess_init=None*)
Initialize self. See help(type(self)) for accurate signature.

class fides.hessian_approximation.**SR1**(*dim*, *hess_init=None*)
Symmetric Rank 1 update strategy. This updating strategy may yield indefinite hessian approximations.

1.6 Logging

This module provides the machinery that is used to display progress of the optimizer as well as debugging information

var logger logging.Logger instance that can be used throughout fides

1.7 Constants

This module provides a central place to define native python enums and constants that are used in multiple other modules

Classes Summary

Options(value)	Defines all the fields that can be specified in Options to `Optimizer`
SubSpaceDim(value)	Defines the possible choices of subspace dimension in which the subproblem will be solved

Classes

class fides.constants.**Options**(*value*)
Defines all the fields that can be specified in Options to `Optimizer`

class fides.constants.**SubSpaceDim**(*value*)
Defines the possible choices of subspace dimension in which the subproblem will be solved

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

B

C

D

F

G

H

L

M

N

O

R

S

T

U