Fides Documentation

Release 0.6.1

The Fides developers

Oct 26, 2021

ABOUT

Version: 0.6.1
Source code: https://github.com/Fides-dev/fides

CHAPTER

ONE

ABOUT FIDES

Fides implements an Interior Trust Region Reflective for boundary costrained optimization problems based on the papers [ColemanLi1994] and [ColemanLi1996]. Accordingly, Fides is named after the Roman goddess of trust and reliability. In contrast to other optimizers, Fides solves the full trust -region subproblem exactly, which can yields higher quality proposal steps, but is computationally more expensive. This makes Fides particularly attractive for optimization problems with objective functions that are computationally expensive to evaluate and the computational cost of solving the trust -region subproblem is negligible.

1.1 Features

- Boundary constrained interior trust-region optimization
- Recursive Reflective and Truncated constraint management
- Full and 2D subproblem solution solvers
- BFGS, DFP and SR1 Hessian Approximations

CHAPTER

TWO

FIDES PYTHON API

Modules

fides	Fides Fides is an interior trust-region reflective optimizer
fides.minimize	Minimization This module performs the optimization given a step proposal.
fides.trust_region	Trust Region Step Evaluation This module provides the machinery to evaluate different trust-region(-reflective) step proposals and select among them based on to their performance according to the quadratic approximation of the objective function
fides.subproblem	Subproblem Solvers This module provides the machinery to solve 1- and N-dimensional trust-region subproblems.
fides.hessian_approximation	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 demanding.
fides.logging	Logging This module provides the machinery that is used to display progress of the optimizer as well as debugging information
fides.constants	Constants This module provides a central place to define native python enums and constants that are used in multiple other modules

2.1 Fides

Fides is an interior trust-region reflective optimizer

2.2 Minimization

This module performs the optimization given a step proposal.

Classes Summary

FunEvaluator(fun, nargout, resfun, funargs)	
Funout(fval, grad, x[, hess, res, sres])	
Optimizer(fun, ub, lb[, verbose, options, ...])	Performs optimization

Classes

class fides.minimize.**FunEvaluator**(*fun*, *nargout*, *resfun*, *funargs*)

__init__(*fun*, *nargout*, *resfun*, *funargs*)

class fides.minimize.**Funout**(*fval*, *grad*, *x*, *hess=None*, *res=None*, *sres=None*)

__init__(*fval*, *grad*, *x*, *hess=None*, *res=None*, *sres=None*)

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

Performs optimization

Variables

- **fevaler** – FunctionEvaluator instance
- **lb** – Lower optimization boundaries
- **ub** – Upper optimization boundaries
- **options** – Options that configure convergence checks
- **delta_iter** – Trust region radius that was used for the current step
- **delta** – Updated trust region radius
- **x** – Current optimization variables
- **fval** – Objective function value at x
- **grad** – Objective function gradient at x
- **x_min** – Optimal optimization variables
- **fval_min** – Objective function value at x_min
- **grad_min** – Objective function gradient at x_min

- **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
- **exitflag** – ExitFlag to indicate reason for termination
- **verbose** – Verbosity level for logging
- **logger** – logger instance

__init__(*fun*, *ub*, *lb*, *verbose=10*, *options=None*, *funargs=None*, *hessian_update=None*, *resfun=False*)
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). If the argument resfun is True, this function must return a tuple (res, sres) instead, where *sres* is the derivative of res.
- **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.
- **resfun** (bool) – Boolean flag indicating whether fun returns function values (False, default) or residuals (True).

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(*step*, *funout*)
Check whether optimization has converged.

Parameters

- **step** (fides.steps.Step) – update to optimization variables
- **funout** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator*

Return type None

check_finite(*funout=None*)
Checks whether objective function value, gradient and Hessian (approximation) have finite values and optimization can continue.

Parameters **funout** (typing.Optional[*fides.minimize.Funout*]) – Function output generated by a *FunEvaluator*

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 its 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*, *step*, *funout*)
Prints diagnostic information about the current step to the log

Parameters

- **accepted** (bool) – flag indicating whether the current step was accepted
- **step** (fides.steps.Step) – proposal step
- **funout** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator*

log_step_initial()
Prints diagnostic information about the initial step to the log

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 using 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[*xtol*] x_k (note that this is checked in transformed coordinates that account for distance to boundaries). 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)

track_minimum(*funout*)
Function that tracks the optimization variables that have minimal function value independent of whether the step is accepted or not.

Parameters **funout** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator* evaluated at new x

Return type None

update(*step*, *funout_new*, *funout*)
Update self according to employed step

Parameters

- **step** (fides.steps.Step) – Employed step
- **funout** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator* for new variables before step is taken
- **funout_new** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator* for new variables after step is taken

Return type None

update_tr_radius(*funout*, *step*, *dv*)
Update the trust region radius

Parameters

- **funout** (*fides.minimize.Funout*) – Function output generated by a *FunEvaluator* for new variables after step is taken
- **step** (fides.steps.Step) – step
- **dv** (numpy.ndarray) – derivative of scaling vector v wrt x

Return type bool

Returns flag indicating whether the proposed step should be accepted

2.3 Trust Region Step Evaluation

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

Functions Summary

trust_region(x, g, hess, scaling, delta, dv, ...)	Compute a step according to the solution of the trust-region subproblem.

Functions

fides.trust_region.**trust_region**(*x*, *g*, *hess*, *scaling*, *delta*, *dv*, *theta*, *lb*, *ub*, *subspace_dim*, *stepback_strategy*, *refine_stepback*, *use_scaled_gradient*, *logger*)
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
- **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.
- **stepback_strategy** (*fides.constants.StepBackStrategy*) – Strategy that is applied when the proposed step exceeds the optimization boundary.
- **refine_stepback** (bool) – If set to True, proposed steps that are computed via the specified stepback_strategy will be refined via optimization.
- **use_scaled_gradient** (bool) – If set to True, the scaled gradient will be added to the set of proposal steps
- **logger** (logging.Logger) – logging.Logger instance to be used for logging

Return type fides.steps.Step

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

2.4 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 subproblem solution according to
get_1d_trust_region_boundary_solution(B, g, ...)	
quadratic_form(Q, p, x)	Computes the quadratic form $x^T Q x + x^T p$
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, ...)	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 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

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

fides.subproblem.**get_1d_trust_region_boundary_solution**(*B*, *g*, *s*, *s0*, *delta*)

fides.subproblem.**quadratic_form**(*Q*, *p*, *x*)
Computes the quadratic form $x^T Q x + x^T p$

Parameters

- **Q** (numpy.ndarray) – Matrix
- **p** (numpy.ndarray) – Vector
- **x** (numpy.ndarray) – Input

Return type float

Returns Value of form

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*, *s0*)

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
- **s0** (numpy.ndarray) – reference point from where search is started, also counts towards norm of step

Return type numpy.ndarray

Returns Proposed step-length

fides.subproblem.**solve_nd_trust_region_subproblem**(*B*, *g*, *delta*, *logger=None*)

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 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 appropriate 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 separately 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
- **logger** (typing.Optional[logging.Logger]) – Logger instance to be used for logging

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

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

2.5 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 demanding.

Classes Summary

BB([init_with_hess])	Broydens "bad" method as introduced in [Broyden 1965](https://doi.org/10.1090%2FS0025-5718-1965-0198670-6).
BFGS([init_with_hess])	Broyden-Fletcher-Goldfarb-Shanno update strategy.
BG([init_with_hess])	Broydens "good" method as introduced in [Broyden 1965](https://doi.org/10.1090%2FS0025-5718-1965-0198670-6).
Broyden(phi[, init_with_hess])	BroydenClass Update scheme as described in [Nocedal & Wright](http://dx.doi.org/10.1007/b98874) Chapter 6.3.
DFP([init_with_hess])	Davidon-Fletcher-Powell update strategy.
FX([happ, hybrid_tol])	
GNSBFGS([hybrid_tol])	
HessianApproximation([init_with_hess])	Abstract class from which Hessian update strategies should subclass
HybridFixed([happ, switch_iteration])	
HybridSwitchApproximation([happ])	
IterativeHessianApproximation([init_with_hess])	Iterative update schemes that only use s and y values for update.
PSB([init_with_hess])	Powell-symmetric-Broyden update strategy as introduced in [Powell 1970](https://doi.org/10.1016/B978-0-12-597050-1.50006-3).
SR1([init_with_hess])	Symmetric Rank 1 update strategy as described in [Nocedal & Wright](http://dx.doi.org/10.1007/b98874) Chapter 6.2.
SSM([update_method])	Structured Secant Method as introduced by [Dennis et al 1989](https://doi.org/10.1007/BF00962795), which is compatible with BFGS, DFP and PSB update schemes.
StructuredApproximation([update_method])	

continues on next page

Table 5 – continued from previous page

TSSM([update_method])	Totally Structured Secant Method as introduced by [Huschens 1994](https://doi.org/10.1137/0804005), which uses a self-adjusting update method for the second order term.

Functions Summary

broyden_class_update(y, s, mat[, phi, v])	Scale free implementation of the broyden class update scheme.

Classes

class fides.hessian_approximation.**BB**(*init_with_hess=False*)

Broydens "bad" method as introduced in [Broyden 1965](https://doi.org/10.1090%2FS0025-5718-1965-0198670-6). This is a rank 1 update strategy that does not preserve symmetry or positive definiteness.

This scheme only works with a function that returns (fval, grad)

class fides.hessian_approximation.**BFGS**(*init_with_hess=False*)

Broyden-Fletcher-Goldfarb-Shanno update strategy. This is a rank 2 update strategy that preserves symmetry and positive-semidefiniteness.

This scheme only works with a function that returns (fval, grad)

__init__(*init_with_hess=False*)

Create a Hessian update strategy instance

Parameters **init_with_hess** (typing.Optional[bool]) – Whether the hybrid update strategy should be initialized according to the user-provided objective function

class fides.hessian_approximation.**BG**(*init_with_hess=False*)

Broydens "good" method as introduced in [Broyden 1965](https://doi.org/10.1090%2FS0025-5718-1965-0198670-6). This is a rank 1 update strategy that does not preserve symmetry or positive definiteness.

This scheme only works with a function that returns (fval, grad)

class fides.hessian_approximation.**Broyden**(*phi*, *init_with_hess=False*)

BroydenClass Update scheme as described in [Nocedal & Wright](http://dx.doi.org/10.1007/b98874) Chapter 6.3. This is a generalization of BFGS/DFP methods where the parameter phi controls the convex combination between the two. This is a rank 2 update strategy that preserves positive-semidefiniteness and symmetry (if $\phi \in [0, 1]$).

This scheme only works with a function that returns (fval, grad)

Parameters **phi** (float) – convex combination parameter interpolating between BFGS (phi==0) and DFP (phi==1).

__init__(*phi*, *init_with_hess=False*)

Create a Hessian update strategy instance

Parameters **init_with_hess** (typing.Optional[bool]) – Whether the hybrid update strategy should be initialized according to the user-provided objective function

class fides.hessian_approximation.**DFP**(*init_with_hess=False*)
Davidon-Fletcher-Powell update strategy. This is a rank 2 update strategy that preserves symmetry and positive-semidefiniteness.

This scheme only works with a function that returns (fval, grad)

__init__(*init_with_hess=False*)
Create a Hessian update strategy instance

Parameters **init_with_hess** (typing.Optional[bool]) – Whether the hybrid update strategy should be initialized according to the user-provided objective function

class fides.hessian_approximation.**FX**(*happ=<fides.hessian_approximation.BFGS object>*, *hybrid_tol=0.2*)

__init__(*happ=<fides.hessian_approximation.BFGS object>*, *hybrid_tol=0.2*)
Hybrid method HY2 as introduced by [Fletcher & Xu 1986](https://doi.org/10.1093/imanum/7.3.371). This approximation scheme employs a dynamic approximation as long as function values satisfy $\frac{f_k - f_{k+1}}{f_k} < \epsilon$ and employs the iterative scheme applied to the last dynamic approximation if not.

This scheme only works with a function that returns (fval, grad, hess)

Parameters **hybrid_tol** (typing.Optional[float]) – switch tolerance ϵ

class fides.hessian_approximation.**GNSBFGS**(*hybrid_tol=1e-06*)

__init__(*hybrid_tol=1e-06*)
Hybrid Gauss-Newton Structured BFGS method as introduced by [Zhou & Chen 2010](https://doi.org/10.1137/090748470), which combines ideas of hybrid switching methods and structured secant methods.

This scheme only works with a function that returns (res, sres)

Parameters **hybrid_tol** (float) – switching tolerance that controls switching between update methods

class fides.hessian_approximation.**HessianApproximation**(*init_with_hess=False*)
Abstract class from which Hessian update strategies should subclass

__init__(*init_with_hess=False*)
Create a Hessian update strategy instance

Parameters **init_with_hess** (typing.Optional[bool]) – Whether the hybrid update strategy should be initialized according to the user-provided objective function

get_mat()
Getter for the Hessian approximation :rtype: numpy.ndarray :return:

init_mat(*dim*, *hess=None*)
Initializes this approximation instance and checks the dimensionality

Parameters

- **dim** (int) – dimension of optimization variables
- **hess** (typing.Optional[numpy.ndarray]) – user provided initialization

set_mat(*mat*)
Getter for the Hessian approximation :return:

class fides.hessian_approximation.**HybridFixed**(*happ=<fides.hessian_approximation.BFGS object>*, *switch_iteration=20*)

__init__(*happ=<fides.hessian_approximation.BFGS object>*, *switch_iteration=20*)
Switch from a dynamic approximation that to the user provided iterative scheme after a fixed number of iterations. The iterative scheme is initialized and updated from the beginning, but only employed after the specified number of iterations.

This scheme only works with a function that returns (fval, grad, hess)

Parameters **switch_iteration** (typing.Optional[int]) – Iteration after which this approximation is used

get_mat()
Getter for the Hessian approximation :rtype: numpy.ndarray :return:

class fides.hessian_approximation.**HybridSwitchApproximation**(*happ=<fides.hessian_approximation.BFGS object>*)

__init__(*happ=<fides.hessian_approximation.BFGS object>*)
Create a Hybrid Hessian update strategy that switches between an iterative approximation and a dynamic approximation

Parameters **happ** (*fides.hessian_approximation.IterativeHessianApproximation*) – Iterative Hessian Approximation

get_mat()
Getter for the Hessian approximation :rtype: numpy.ndarray :return:

init_mat(*dim*, *hess=None*)
Initializes this approximation instance and checks the dimensionality

Parameters

- **dim** (int) – dimension of optimization variables
- **hess** (typing.Optional[numpy.ndarray]) – user provided initialization

class fides.hessian_approximation.**IterativeHessianApproximation**(*init_with_hess=False*)
Iterative update schemes that only use s and y values for update.

class fides.hessian_approximation.**PSB**(*init_with_hess=False*)
Powell-symmetric-Broyden update strategy as introduced in [Powell 1970](https://doi.org/10.1016/B978-0-12-597050-1.50006-3). This is a rank 2 update strategy that preserves symmetry and positive-semidefiniteness.

This scheme only works with a function that returns (fval, grad)

class fides.hessian_approximation.**SR1**(*init_with_hess=False*)
Symmetric Rank 1 update strategy as described in [Nocedal & Wright](http://dx.doi.org/10.1007/b98874) Chapter 6.2. This is a rank 1 update strategy that preserves symmetry but does not preserve positive-semidefiniteness.

This scheme only works with a function that returns (fval, grad)

class fides.hessian_approximation.**SSM**(*update_method='BFGS'*)
Structured Secant Method as introduced by [Dennis et al 1989](https://doi.org/10.1007/BF00962795), which is compatible with BFGS, DFP and PSB update schemes.

This scheme only works with a function that returns (res, sres)

class fides.hessian_approximation.**StructuredApproximation**(*update_method='BFGS'*)

__init__(*update_method='BFGS'*)
This is the base class for structured secant methods (SSM). SSMs approximate the hessian by combining the Gauss-Newton component C(x) and an iteratively updated component that approximates the difference S to the true Hessian.

init_mat(*dim*, *hess=None*)
Initializes this approximation instance and checks the dimensionality

Parameters

- **dim** (int) – dimension of optimization variables
- **hess** (typing.Optional[numpy.ndarray]) – user provided initialization

class fides.hessian_approximation.**TSSM**(*update_method='BFGS'*)
Totally Structured Secant Method as introduced by [Huschens 1994](https://doi.org/10.1137/0804005), which uses a self-adjusting update method for the second order term.

This scheme only works with a function that returns (res, sres)

Functions

fides.hessian_approximation.**broyden_class_update**(*y*, *s*, *mat*, *phi=None*, *v=None*)
Scale free implementation of the broyden class update scheme. This can either be called by using a phi parameter that interpolates between BFGS (phi=0) and DFP (phi=1) or by using the weighting vector v that allows implementation of PSB (v=s), DFP (v=y) and BFGS (V=y+rho*B*s).

Parameters

- **y** – difference in gradient
- **s** – search direction in previous step
- **mat** – current hessian approximation
- **phi** – convex combination parameter. Must not pass this parameter at the same time as v.
- **v** – weighting vector. Must not pass this parameter at the same time as phi.

2.6 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

Functions Summary

create_logger(level)	Creates a logger instance.

Functions

fides.logging.**create_logger**(*level*)
Creates a logger instance. To avoid unnecessary locks during multithreading, different logger instance should be created for every

Parameters **level** (int) – logging level

Return type logging.Logger

Returns logger instance

2.7 Constants

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

Classes Summary

ExitFlag(value)	Defines possible exitflag values for the optimizer to indicate why optimization exited.
Options(value)	Defines all the fields that can be specified in Options to Optimizer
StepBackStrategy(value)	Defines the possible choices of search refinement if proposed step reaches optimization boundary
SubSpaceDim(value)	Defines the possible choices of subspace dimension in which the subproblem will be solved.

Classes

class fides.constants.**ExitFlag**(*value*)
Defines possible exitflag values for the optimizer to indicate why optimization exited. Negative value indicate errors while positive values indicate convergence.

DELTA_TOO_SMALL = -5
Trust Region Radius too small to proceed

DID_NOT_RUN = 0
Optimizer did not run

EXCEEDED_BOUNDARY = -4
Exceeded specified boundaries

FTOL = 1
Converged according to fval difference

GTOL = 3
Converged according to gradient norm

MAXITER = -1
Reached maximum number of allowed iterations

MAXTIME = -2
Expected to reach maximum allowed time in next iteration

NOT_FINITE = -3
Encountered non-finite fval/grad/hess

XTOL = 2
Converged according to x difference

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

DELTA_INIT = 'delta_init'
initial trust region radius

FATOL = 'fatol'
absolute tolerance for convergence based on fval

FRTOL = 'frtol'
relative tolerance for convergence based on fval

GATOL = 'gatol'
absolute tolerance for convergence based on grad

GRTOL = 'grtol'
relative tolerance for convergence based on grad

MAXITER = 'maxiter'
maximum number of allowed iterations

MAXTIME = 'maxtime'
maximum amount of walltime in seconds

STEPBACK_STRAT = 'stepback_strategy'
method to use for stepback

SUBSPACE_DIM = 'subspace_solver'
trust region subproblem subspace

THETA_MAX = 'theta_max'
maximal fraction of step that would hit bounds

XTOL = 'xtol'
tolerance for convergence based on x

class fides.constants.**StepBackStrategy**(*value*)
Defines the possible choices of search refinement if proposed step reaches optimization boundary

MIXED = 'mixed'
mix reflections and truncations

REFLECT = 'reflect'
recursive reflections at boundary

SINGLE_REFLECT = 'reflect_single'
single reflection at boundary

TRUNCATE = 'truncate'
truncate step at boundary and re-solve

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

FULL = 'full'
Full $\mathbb{R}^n$

STEIHAUG = 'scg'
CG subspace via Steihaug's method

TWO = '2D'
Two dimensional Newton/Gradient subspace

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

B

C

D

E

F

G