# SID-PSM: A PATTERN SEARCH METHOD GUIDED BY SIMPLEX DERIVATIVES FOR USE IN DERIVATIVE-FREE OPTIMIZATION

A. L. CUSTÓDIO* AND L. N. VICENTE†

**Abstract.** SID-PSM (version 1.2) is a suite of MATLAB [1] functions for numerically solving constrained or unconstrained nonlinear optimization problems, using derivative-free methods. In the general constrained case and for the current version, the derivatives of the functions defining the constraints must be provided. The optimizer uses an implementation of a generalized pattern search method, combining its global convergence properties with the efficiency of the use of quadratic polynomials to enhance the search step and of the use of simplex gradients for guiding the function evaluations of the poll step. An advantage of using SID-PSM, when compared to other pattern search implementations, is the reduction achieved in the total number of function evaluations required. In this document, we will mention the target problems suited for optimization with this code, describe the SID-PSM algorithm, and provide implementation details (including information about the MATLAB functions coded as well as guidelines to obtain, install, and customize the package). The software is freely available for research, educational or commercial use, under a GNU lesser general public license, but we expect that all publications describing work using this software quote the two following references. (1) A. L. Custódio and L. N. Vicente, Using sampling and simplex derivatives in pattern search methods, SIAM Journal on Optimization, 18 (2007) 537–555 (ref. [10]); (2) A. L. Custódio, H. Rocha, and L. N. Vicente, Incorporating minimum Frobenius norm models in direct search, Computational Optimization and Applications 46 (2010), 265–278 (ref. [9]).

**Key words.** derivative-free optimization, optimization software, documentation, generalized pattern search methods, simplex gradient, poll ordering, poisedness, interpolation models, search step, minimum Frobenius norm models

**AMS subject classifications.** 65K05, 90C30, 90C56

## Contents.



*Department of Mathematics, FCT-UNL, Quinta da Torre 2829-516 Caparica, Portugal (alcustodio@fct.unl.pt). Support for this author was provided by Centro de Matemática e Aplicações da Universidade Nova de Lisboa and by FCT under grants POCI/MAT/59442/2004 and PTDC/MAT/64838/2006.

†CMUC, Department of Mathematics, University of Coimbra, 3001-454 Coimbra, Portugal (lnv@mat.uc.pt). Support for this author was provided by FCT under grants POCI/MAT/59442/2004 and PTDC/MAT/64838/2006.

**1. Introduction.** SID-PSM (version 1.2) is a serial MATLAB [1] implementation of a generalized pattern search method (see [4]) based on the algorithmic framework introduced in [10] for incorporating simplex gradient and simplex Hessian information. Recently, quadratic minimum Frobenius norm (MFN) models were incorporated into the code to improve the efficiency of the underlying direct search method (see [9]). SID-PSM is well suited for solving unconstrained and constrained optimization problems, without the use of derivatives of the objective function. In the general constrained case, the first order derivatives of the functions defining the constraints must be provided. Supporting theory for the methods implemented (in terms of global convergence for stationary points) exists except in the nonlinear constrained case.

The software is freely available for research, educational or commercial use, under a GNU lesser general public license, but we expect that all publications describing work using this software quote the references [9, 10].

The type of problems given to SID-PSM must be in the form

$$\min_{x\in\Omega} f(x),$$

where $\Omega$ is either $\mathbb{R}^n$ or the feasible region

$$\Omega = \{x \in \mathbb{R}^n : \ c_i(x) \leq 0, \ i = 1, \ldots, m\}.$$

In the latter case, the user must provide functions to evaluate the gradients $\nabla c_i(x)$, $i = 1, \ldots, m$. Equality constraints, when included in the original problem formulation, could be converted first into double inequalities. However, we note that there are some numerical difficulties associated with this approach. The user is advised not to follow it, but rather to try to get rid of the equalities by eliminating some of the problem variables. SID-PSM does not require any gradient information about the objective function, so one can apply the code to any objective function (in particular to those resulting from running black-box codes or performing physical experiments).

The distinguishing feature between SID-PSM and other derivative-free packages freely available, like NOMAD/NOMADm [2] and APPSPACK [11], is the use of past function evaluations to improve the efficiency of the underlying pattern search method

(version 4.6 of NOMADm [2] incorporates a simplified version of one of the strategies implemented in `SID-PSM`). As reported in [8, 10], the use of simplex derivatives can lead to a significant reduction in the number of function evaluations required, by appropriately ordering the poll vectors. Moreover, the numerical results in [9] show a clear advantage in applying search steps based on MFN models, in the context of direct search methods of directional type.

In `SID-PSM`, previous evaluations of the objective function are used in the computation of simplex derivatives (*e.g.*, simplex gradients or simplex Hessians), which in turn improve the efficiency of a pattern search iteration. The improvement can be achieved in several ways, such as reordering the poll directions according to the angles they make with the negative simplex gradient. The introduction of a search step based on the minimization of a MFN model of the objective function is also considered. Other possible uses for simplex derivatives are a mesh size update based on a sufficient decrease condition, the definition of stopping criteria based on simplex gradients, and the pruning of the poll vectors.

A brief description of the optimization algorithm can be found in Section 2. A more complete description, including some analysis and numerical results, is given in [9, 10]. The next two sections are more suitable for potential users: Section 3 explains how to obtain and install `SID-PSM`, while Section 4 concerns how to run it and interpret the results. A detailed description of the parameters file and of all the other program files is given in Sections 5 and 6, respectively.

**2. The `SID-PSM` algorithm.** In a simplified version, a pattern search method starts with an initial guess solution and performs a sequence of iterations in a rational lattice, trying to improve this initial guess.

At each iteration, two steps are performed: a *search* step and a *poll* step. The *search* step is optional and unnecessary for establishing the convergence properties of the method.

The *poll* step, which will be only performed if the *search* step is unsuccessful in finding a lower objective function value, follows stricter rules since it guarantees the global convergence properties of the method to stationary points (in the lim inf sense, as originally established in [16]). A mesh, or a grid, is considered, defined from a set of directions with appropriate descent properties (called a positive spanning set or a poll set). In the unconstrained case, these directions must positively span $\mathbb{R}^n$, while in the constrained case they need to conform to the geometry of the nearby boundary at the current point, defined by the approximated active constraints gradients (see [13, 14]). A local exploration of a mesh neighborhood of the current iterate is done by testing the points (or feasible points, in the constrained case) corresponding to directions of the poll set $D_k$.

The *search* step is free of any rules as long as only a finite number of (feasible) trial points in the current mesh is considered. If both *search* and *poll* steps are unsuccessful, then the mesh is contracted by reducing the mesh size parameter. In case of success, the mesh size parameter can be maintained or increased, thus possibly expanding the mesh.

During the course of the iterations, a pattern search method generates a number of function evaluations. The corresponding points can be stored together with the objective function values and used to improve the efficiency of the algorithm.

Thus, at the beginning of each iteration of a pattern search method, one can try to identify a subset of these points with some desirable geometrical properties. `SID-PSM` uses the geometrical notion of $\Lambda$-poisedness, introduced in [5] and extended to the

regression and underdetermined cases in [6], as described in [10] (see also the book [7]). Given this subset of points one can compute some form of simplex derivatives. The size and the geometrical properties of the sample set will determine the quality of the corresponding simplex derivatives as an approximation to the exact derivatives of the objective function (see [5, 6, 8]). Simplex derivatives are basically the coefficients of some multivariate polynomial interpolation models associated with the sample set at the current interpolation point. `SID-PSM` allows the computation of determined simplex derivatives as well as underdetermined and overdetermined (regression) forms, depending on the number of points available. Using these simplex derivatives, a direction of potential descent (called a descent indicator) can be computed, at no additional cost in terms of function evaluations.

This descent indicator could be a negative simplex gradient $\iota_k = -\nabla_s f(x_k)$, as defined in Section 4 of [10], or a simplex Newton direction $\iota_k = -H_k^{-1} g_k$, where $g_k$ is a simplex gradient and $H_k$ approximates a simplex Hessian. The type of descent indicator to be computed and used is selected by means of the `shessian` option specified in the `parameters.m` file (negative simplex gradient or simplex Newton direction).

The computed simplex derivatives and the corresponding descent indicator will be used for ordering or pruning the poll set, for updating the mesh size parameter and in the definition of stopping criteria.

The evaluated points can also be used to build a model for the objective function, at the current iterate. `SID-PSM` considers MFN models, whose minimization defines a search step for the algorithm. In this case, the geometry control of the sample set is extremely loose, in an attempt to explore all the information available.

The described procedures incorporate the main features distinguishing `SID-PSM` from other pattern search packages currently available.

A formal description of the algorithm implemented in `SID-PSM` is given in Figure 2.1. The differences between the generalized pattern search method of [4] and this algorithmic framework rely basically in the use of the simplex derivatives and in the definition of the search step. We marked them in italic for a better identification.

**2.1. Ordering.** At each iteration, `SID-PSM` reorders the poll vectors according to a previously established strategy, just before polling starts. In most implementations, this ordering is the one in which the vectors are originally stored, and it is never changed during the course of the iterations. NOMAD/NOMADm [2] implements an option called *dynamic polling*, which consists in bringing into the first column of the poll set $D_{k+1}$ the last poll vector $d_k$ associated with a successful polling iterate. `SID-PSM` allows *dynamic polling* but also *descent indicator polling*.

In this case, `SID-PSM` orders the poll vectors according to the increasing amplitudes of the angles between the descent indicator $\iota_k$ and these poll vectors. Figure 2.2 describes the procedure `order_proc` in the *descent indicator polling* case. In case of failure in computing $\iota_k$ at a given iteration, two different options can be selected for polling: (i) test the vectors in the positive spanning set from the beginning, following the order of the last iteration, or (ii) poll cyclicly in $D_k$, meaning that the first poll vector to be tested is the one immediately after the last one tested in the last iteration.

Variants of these *descent indicator polling* strategies can be considered using a MFN model when it has been built for the search step. The poll directions can be ordered according to the MFN model values computed at the corresponding poll points (with the possible consideration of the two above mentioned strategies in case of failure in building such a model). Alternatively, the gradient and the Hessian matrix of the MFN model can be used to build a descent indicator, which can then be used

`SID-PSM` — **A Pattern Search Method Guided by Simplex Derivatives**

**Initialization**
Choose a feasible point $x_0$ and an initial mesh size parameter $\alpha_0 > 0$. Choose a positive spanning set $D$. If a constrained problem is to be solved then choose $\epsilon = \epsilon_{ini}$ as initial value for considering a constraint as approximated active. Select all constants needed for procedures `[search_proc]`, `[order_proc]`, and `[mesh_proc]`. Set $k = 0$. *Set $X_0 = [x_0]$ to initialize the list of points managed by* `[store_proc]`. *Choose a maximum number $p_{max}$ of points that can be stored. Choose also the minimum $s_{min}$ and the maximum $s_{max}$ number of points involved in any simplex derivatives calculation ($2 \leq s_{min} \leq s_{max}$). Choose $\Lambda > 0$ and $\sigma_{max} \geq 1$, to be used in the $\Lambda$-poised set computation.*

**Identifying a $\Lambda$–poised sample set and computing simplex derivatives**
*Skip this step if there are not enough points, i.e., if $|X_k| < s_{min}$. Set $\Delta_k = \sigma_k \, \alpha_{k-1} \max_{d \in D_{k-1}} \|d\|$, where $\sigma_k \in [1, \sigma_{max}]$. Try to identify a set of points $Y_k$ in $X_k$, within a distance of $\Delta_k$ from $x_k$, with as many points as possible (up to $s_{max}$) and such that $Y_k$ is $\Lambda$–poised, and includes the current iterate $x_k$. If $|Y_k| \geq s_{min}$ then compute some form of simplex derivatives based on $Y_k$ (and from that compute a descent indicator $\iota_k$).*

**Search step**
Call `[search_proc]` to try to compute a feasible mesh point $x$ with $f(x) < f(x_k)$ by evaluating the function only at a finite number of feasible points in the mesh *and calling* `[store_proc]` *each time a point is evaluated.* If such a point is found, then set $x_{k+1} = x$, declare the iteration as successful, and skip the poll step.

**Poll step**
Choose a positive spanning set $D_k \subset D$ that conforms to the geometry of the approximated active constraints at $x_k$. Call `[order_proc]` to order the poll set $P_k = \{x_k + \alpha_k d : d \in D_k\}$. Start evaluating $f$ at the feasible poll points following the order determined *and calling* `[store_proc]` *each time a point is evaluated.* If a feasible poll point $x_k + \alpha_k d_k$ is found such that $f(x_k + \alpha_k d_k) < f(x_k)$, then stop polling, set $x_{k+1} = x_k + \alpha_k d_k$, and declare the iteration as successful. Otherwise declare the iteration as unsuccessful and set $x_{k+1} = x_k$.

**Updating the mesh size parameter and computing new poll directions**
Call `[mesh_proc]` to compute $\alpha_{k+1}$. If a constrained problem is being solved update the tolerance for considering a constraint as approximated active, using the formula $\epsilon = \min(\epsilon_{ini}, 10\,\alpha_{k+1})$. Increment $k$ by one and return to the *simplex derivatives step.*

FIG. 2.1. *Class of pattern search methods implemented in* `SID-PSM`.

for reordering the poll directions.

`SID-PSM` also allows a random order of the positive spanning set. The selection of the order to be used in the poll step is done through the value of `order_option`, in the `parameters.m` file (see Section 5).

**2.2. Search step.** In the current version, `SID-PSM` computes a MFN model once there are more than $n+1$ points for which the objective function has been evaluated.

When there are more than $(n+1)(n+2)/2$ points available for interpolation, two variants are implemented: to consider all the points and compute a regression model or to discard some of them and build a complete determined model (the option is made

| Compute $\cos(\iota_k, d)$ for all $d \in D_k$. Order the columns in $D_k$ according to decreasing values of the corresponding cosines. |
|---|

FIG. 2.2. *Ordering the poll vectors according to the angle distance to the descent indicator.*

through variable `regopt` in `parameters.m` file). In the last case, since the quadratic model has a local purpose, $p\%$ of the necessary points are selected as the ones nearest to the current iterate. The last $(1-p)\%$ are chosen as the ones further away from the current iterate, in an attempt of preserving geometry and diversifying the information used in model computation. By default, $p\% = 80\%$ (in the `quad_Frob.m` file).

The point to be evaluated in the search step of the algorithm results from the minimization of the MFN model in the trust region considered. In this minimization, the value of the trust-region radius is never allowed to be smaller than $10^{-5}$. `SID-PSM` allows the choice between the codes MINPACK2 [15] and the MATLAB function `trust.m`, this last one part of the Optimization toolbox. The option is selected through the value of `trs_solver` in the `parameters.m` file. If no new MFN model is formed at the current iteration, a search step can always be attempted after a first model has been built, using the last model computed (`always` variable set to `1` in `parameters.m` file).

In the constrained case, the MFN models are built using previously evaluated feasible points. If the minimizer of the MFN model within the trust region is infeasible, then it is orthogonally projected onto the feasible region if this is defined by simple bounds, or simply discarded in the presence of more general constraints in which case no search step occurs.

Note that for global convergence purposes the minimizer should be projected onto the mesh. However, this projection is not implemented in `SID-PSM`.

**2.3. Mesh size parameter update.** When `mesh_option` is set to `0` in `SID-PSM` `parameters.m` file, the update of the mesh size parameter is made by expanding the mesh in every successful iteration and contracting it for unsuccessful ones. This strategy can result into an excessive number of later contractions, each one requiring a complete polling, thus leading to an increase in the total number of function evaluations required. A possible strategy to avoid this behavior has been suggested in [12] and consists of expanding the mesh only if two consecutive successful iterates have been computed using the same direction. The user can select this option by setting `mesh_option` to `3` in the `SID-PSM` `parameters.m` file. If mesh size is to be maintained at each successful iteration, then the coefficient `phi` for mesh expansion should be set to `1` in the `parameters.m` file.

In the linear case, when only a simplex gradient $\nabla_s f(x_k)$ is computed, the model $m_k(y) = f(x_k) + \nabla_s f(x_k)^\top (y - x_k)$ can also be used as another alternative to avoid the described behavior, by imposing a sufficient decrease condition on the update of the mesh size parameter $\alpha_k$. In the quadratic case, when a simplex gradient $g_k$ and a diagonal simplex Hessian are computed, the model is replaced by $m_k(y) = f(x_k) + g_k^\top (y - x_k) + (1/2)(y - x_k)^\top H_k (y - x_k)$. We describe such procedures in Figure 2.3, where the sufficient decrease is only applied to successful iterations. This update corresponds to set `mesh_option` to `2`.

`SID-PSM` has also available another variant for updating the mesh size parameter based on a sufficient decrease condition (`mesh_option` set to `1`). This time, when the value of the ratio $\rho_k$ is small ($0 < \rho_k \leq \gamma_1 < 1$), mesh contractions are allowed even in successful iterations. Nevertheless, this option should be used wisely since it can

If the iteration was successful then compute

$$\rho_k \;=\; \frac{f(x_k) - f(x_{k+1})}{m_k(x_k) - m_k(x_{k+1})}.$$

| | | |
|---|---|---|
| If $\rho_k > \gamma_2$ | then | $\alpha_{k+1} = \tau^{j_k^+} \alpha_k$. |
| If $\rho_k \leq \gamma_2$ | then | $\alpha_{k+1} = \alpha_k$. |
| (If `mesh_option` is equal to `1` and $\rho_k \leq \gamma_1$ | then | $\alpha_{k+1} = \tau^{j_k^-} \alpha_k$.) |

If the iteration was unsuccessful, then contract mesh by decreasing the mesh size parameter $\alpha_{k+1} = \tau^{j_k^-} \alpha_k$.

FIG. 2.3. *Updating the mesh size parameter (using sufficient decrease but meeting rational lattice requirements). The constants $\tau$, $\gamma_1$, and $\gamma_2$ must satisfy $\tau \in \mathbb{Q}$, $\tau > 1$, and $0 < \gamma_1 < \gamma_2 < 1$, and should be initialized at iteration $k = 0$ together with $j_{max} \in \mathbb{Z}$, $j_{max} \geq 0$, and $j_{min} \in \mathbb{Z}$, $j_{min} \leq -1$. The exponents satisfy $j_k^+ \in \{0, 1, 2, \ldots, j_{max}\}$ and $j_k^- \in \{j_{min}, \ldots, -1\}$.*

If `stop_grad` is equal to `1` and $\|\Delta_k \nabla_s f(x_k)\|_\infty \leq tol_{grad}$ then stop the algorithm.
If `stop_grad` is equal to `2` and $\Delta_k d^\top \nabla_s f(x_k) \geq -tol_{grad}$, for all $d \in D_k$ then stop the algorithm.

FIG. 2.4. *Stopping criteria based in simplex gradients. The tolerance $tol_{grad} > 0$ is set at iteration $k = 0$.*

lead to premature convergence of the algorithm.

**2.4. Stopping criteria.** The quality of simplex gradients, computed from $\Lambda$-poised sets, as approximations to the exact gradient is analyzed in [5, 6]. Based on such error estimates, one can use an intuitive criterion for stopping a pattern search method based on the size of the simplex gradient (as suggested in [17]). In `SID-PSM`, when the `stop_grad` variable is set to `1` in `parameters.m`, the method is stopped if the norm of a scaled simplex gradient is small (see Figure 2.4).

The analysis in [8] has suggested an alternative criterion to stop the method, by using simplex gradients to compute approximations to the (possible generalized) directional derivatives along the poll directions. If `stop_grad` is set to `2` then the method stops when these directional derivatives are nonnegative (up to a small tolerance). Again see Figure 2.4.

Similarly to other implementations of pattern search, `SID-PSM` has also available a stopping criterion based on the mesh size (option `stop_alfa` in `parameters.m`). The algorithm can either stop if a maximum number of iterations or a maximum number of function evaluations is reached (options `stop_iter` and `stop_fevals` in `parameters.m`, respectively). In these last cases, a maximum number of iterations or of function evaluations should be provided.

Each of these stopping criteria can be applied alone or combined with others.

**2.5. Pruning.** `SID-PSM` incorporates an implementation of a variant of the pruning strategy suggested in [3] for the poll set. This implementation is motivated by the theoretical results derived in [10, Section 6], where it is shown that, for a sufficiently small mesh size parameter, the negative simplex gradient can be considered as an $\epsilon$-approximation (again, see [3]) to the large components of the exact negative gradient.

After reordering the vectors of the poll set according to the descent indicator considered, the resulting matrix can be pruned to the most promising direction or to all the potential descent directions. Since the conditions that guarantee that the descent indicator is an $\epsilon$-approximation are never tested, we recommend a prudent use of the pruning strategies.

**2.6. Storing points for simplex derivatives.** `SID-PSM` maintains a list $X_k$ of points of maximum size $p_{max}$ used for computing the simplex derivatives and the MFN models. It is possible to implement different criteria for deciding whether to store or not a point at which the function has been evaluated. A binary variable (`store_all`) in the `parameters.m` file distinguishes between two simple ways of storing points:

- *store-successful*: the list keeps only the successful iterates $x_{k+1}$ (for which $f(x_{k+1}) < f(x_k)$).
- *store-all*: the list keeps every evaluated point.

In both cases, the points are added sequentially to $X_k$ at the top of the list. Therefore, in `store_succ` the points in $X_k$ are ordered by increasing objective function values. When (and if) $X_k$ has reached its predetermined size $p_{max}$, a point is removed from the bottom of the list before adding a new one. The process prevents the current iterate from being removed from the list (when `store_all` is set to `1` and there occur several consecutive unsuccessful iterations). See Table 2.1 for more information.

| size | shessian | store-successful | store-all |
|---|---|---|---|
| p_max | 0 | $2(n+1)$ | $(n+1)(n+2)$ |
| s_min | 0 | $(n+1)/2$ | $n+1$ |
| s_max | 0 | $n+1$ | $n+1$ |
| p_max | 1 | $4(n+1)$ | $8(n+1)$ |
| s_min | 1 | $n$ | $2n+1$ |
| s_max | 1 | $2n+1$ | $2n+1$ |

TABLE 2.1
*Sizes of the list $X_k$ and of the set $Y_k$.*

**2.7. Cache.** As a pattern search method, `SID-PSM` minimizes the objective function by sampling a pattern of points lying in a rational lattice. As a consequence, the same point can be evaluated more than once. To avoid the occurrence of these kind of situations, a `cache` option is available for use in `SID-PSM` (and can be set in the `parameters.m` file).

Before computing the objective function value at a new point, `SID-PSM` scans a list of points to see whether the current point has been already evaluated. In one of the implemented options, the cache list is equal to $X_k$, the list of points stored for simplex derivatives computation. In another option, a totally independent list is used to store all the previous evaluated points (within a maximum predetermined size). The point comparisons are made using the $\ell_\infty$–norm.

**2.8. Economic options.** Checking $\Lambda$-poisedness can be a computationally expensive task if implemented in a way where points are checked sequentially one by one. The `economic` variable in the `parameters.m` file allows a block $\Lambda$-poisedness checking, before starting the pointwise sequential process. Another possibility is to combine the block checking with the use of the QR decomposition for testing the

Λ-poisedness condition (instead of applying the more expensive SVD decomposition). See Section 6 for more details.

**3. Installing SID-PSM.** SID-PSM (version 1.2) is freely available for research, educational or commercial use under the terms of the GNU lesser general public license. Currently, SID-PSM only runs in MATLAB. The code requires approximately 650Kb of hard disk space.

To get SID-PSM (version 1.2), the user just needs to go to

```
http://www.mat.uc.pt/sid-psm
```

and send an email requesting the code. After saving the file sid_psm_1.2.tar.gz to the disk drive, if the user is running the code on an Unix/Linux platform then the following commands should be executed:

```
gunzip -c sid_psm_1.2.tar.gz | tar xvf -
```

In a Windows platform the user should use Unzip to unpack the compressed file.

A directory named sid_psm_1.2 will be created, either in Windows or Unix/Linux operating systems. This directory must be moved to a working directory in the MATLAB tree. Alternatively, the MATLAB path could be updated accordingly.

If you plan to use the current search option in SID-PSM, based on MFN models, then you need a solver for the solution of the trust-region subproblems. One alternative is to to use the MATLAB function trust.m, part of the Optimization toolbox. Another alternative, which requires a Fortran compiler, is the Fortran subroutine dgqt from MINPACK2 [15] which is provided in the directory sid_psm_1.2.

In the directory sid_psm_1.2 there are the following files:

**Documentation files and MATLAB examples**

| | |
|---|---|
| driver_const.m | A driver for constrained problems. |
| driver_unconst.m | A driver for unconstrained problems. |
| README.txt | A file of basic instructions concerning installation and running issues. |
| sid_psm_manual.pdf | A version of this manuscript. |

**User provided MATLAB files**

| | |
|---|---|
| func_f.m | To compute the value of the objective function. |
| func_const.m | To compute the values of the constraint functions. |
| grad_const.m | To compute the gradients of the constraint functions. |

**Optimizer MATLAB files**

| | |
|---|---|
| `domain.m` | To check the feasibility of a given point. |
| `gen.m` | To compute the positive spanning set to be used by the pattern search method. |
| `grad_act.m` | To determine the approximated active constraints. |
| `lambda_poised.m` | To compute a $\Lambda$-poised set from a given set of points. |
| `match_point.m` | To check if a point has been previously evaluated. |
| `mesh_proc.m` | To update the mesh size parameter. |
| `order_proc.m` | To reorder the columns of a given matrix according to the angles between them and a given vector. |
| `parameters.m` | A file with default values for the parameters to be used by `SID-PSM`. |
| `proj_ort.m` | To compute the orthogonal projection of a given point on a feasible region defined by bounds on the problem variables. |
| `prune_dir.m` | To prune the columns of a given matrix according to the angles between them and a given vector. |
| `quad_Frob.m` | To compute a quadratic interpolation model for the objective function, in the underdetermined case by minimizing the Frobenius norm of the Hessian matrix. |
| `sid_psm.m` | The main program. Applies a pattern search method to the user's problem. |
| `simplex_deriv.m` | To compute simplex derivatives from a given set of points. |

**4. Running `SID-PSM`.** Before running `SID-PSM`, the user should provide the necessary information about the objective function in the file `func_f.m`. Additionally, if there are any constraints, the user must supply the corresponding information in the files `func_const.m` and `grad_const.m` (the latter file can be omitted when only variable bounds are present). Next, at the directory `sid_psm_1.2`, the user should type the following calling sequence:

```
[x_final,f_final,histout] =sid_psm(x_initial,const,output)
```

where `x_final` provides the final iterate computed by the algorithm and `f_final` the corresponding objective function value. The matrix `histout` stores the counter for function evaluations (first column) and the corresponding function values (second column), and is particularly useful to plot the history of the minimization process:

```
plot(histout(:,1),histout(:,2))
```

MATLAB allows `sid_psm` to be called with fewer output arguments, or with none as in:

```
sid_psm(x_initial,const,output)
```

The following data has to be provided:

- `x_initial` (the initial point to start the optimizer);

- `const` (`0` if the problem is unconstrained, `1` if the problem has general constraints; `2` if the constraints are only bounds);
- `output` (`0` for no output on screen and on a file, `1` for a short screen and file output, `2` for a more detailed screen and file output).

For a quick start, or to confirm that the `SID-PSM` installation was completed successfully, the user can run the drivers `driver_unconst.m` or `driver_const.m`.

The first of these m-files applies `SID-PSM` to solve the unconstrained problem

$$\min_{(x_1,x_2)\in\mathbb{R}^2} (x_2 - x_1^2)^2.$$

The user can type the following calling sequence, appropriated for unconstrained minimization (`const = 0`):

```
sid_psm([-1.2;1],0,1)
```

The initial point considered is `[-1.2;1]`. The optimizer uses the default options specified in the file `parameters.m` (see Section 5). By setting `output = 1` in the calling sequence, a short output report, like the one reproduced in Appendix A, is presented both at the screen and in a text file, `sid_psm_report.txt` (stored under the directory `sid_psm_1.0`).

The `driver_const.m` file addresses a constrained version of the above minimization problem

$$\min_{(x_1,x_2)\in\mathbb{R}^2} (x_2 - x_1^2)^2 \text{ s.t. } -2 \leq x_1 \leq 0 \text{ and } x_2 \leq 1.$$

The user could type now the calling sequence for the general constrained case (`const = 1`):

```
sid_psm([-1.2;1],1,2)
```

The main difference in the output, by having set `output = 2` in the calling sequence, is that now are reported the number of function evaluations computed at each iteration, the number of approximated active constraints, and the values of three binary variables identifying $\Lambda$-poisedness success, iteration success, and search step success (see Appendix B). Additionally, the `sid_psm_report.txt` file records the iterates computed during the optimization process. Again, default values for the optimizer parameters are given in `parameters.m` (see Section 5).

If all the problem constraints are bounds, as in the example above, then the user can type the following calling sequence

```
sid_psm([-1.2;1],2,2)
```

where `const = 2`. With this option, at the search step of the algorithm, when the minimizer of the MFN model is not feasible, it is considered its orthogonal projection onto the feasible region. Also, the positive spanning set to be considered for polling is fixed accross the iterations and, based on global convergence considerations, should always include the coordinate directions. Appendix C reports the output obtained for the constrained version of the problem considered.

**5. Parameters MATLAB file.** The algorithmic strategies (including `order_option`, `search_option`, `mesh_option`, `stop_grad`) to be used by the optimizer as well as the values of some constants, parameters, and tolerances which can be user modified

are recorded in the file `parameters.m`. Appendix D gives a detailed description of each one of these variables, of their ranges, and of the default values considered.

**6. Overview of `SID-PSM` MATLAB files.** In this section, we give a detailed description of each one of the program files, in particular of the data that the user should provide in order to solve a problem.

**6.1. User provided files.**

**6.1.1.** `func_f.m`. This function computes the value of the objective function at a point specified by the optimizer. The calling sequence is

```
[f] = func_f(x)
```

**6.1.2.** `func_const.m`. It computes the values of the functions defining the constraints $c_i(x) \leq 0, i = 1, \ldots, m$, at a point provided by the optimizer. The function returns a vector storing the constraints values. The calling sequence is

```
[c_const] = func_const(x)
```

**6.1.3.** `grad_const.m`. This function computes the gradients of the functions defining the constraints $c_i(x) \leq 0,\ i = 1, \ldots, m$, in a columnwise fashion. The calling sequence is

```
[grad_c] = grad_const(x)
```

**6.2. Optimizer files.**

**6.2.1.** `domain.m`. This function checks the feasibility of a given point. The result is a binary variable representing feasibility and a vector storing the values of the functions defining the constraints at the given point. It calls `func_const`.

**6.2.2.** `gen.m`. Function `gen.m` computes a positive spanning set for $\mathbb{R}^n$, when the problem to be solved is unconstrained, when only bound constraints are present, or when there are no approximated active constraints. Otherwise, it computes a set of positive generators for the tangent cone defined by the approximated active constraints gradients. This function also detects any degenerated situation (numerical linear dependence of the constraints gradients).

The input parameters of `gen.m` are the number of variables of the problem, the tolerance for detecting degenerated situations, the number of approximated active constraints, and a matrix storing columnwise the gradients of the functions defining these constraints. Variable `const` allows the distinction between unconstrained and constrained cases.

In the unconstrained case, or when there are no approximated active constraints, `SID-PSM` provides a number of possible positive spanning sets: the minimal positive basis $[-e\ I]$, where $e$ is a vector of ones and $I$ stands for the identity matrix; the maximal positive basis $[I\ -I]$; the positive spanning set $[e\ -e\ I\ -I]$, and a positive basis with angles of uniform amplitude among vectors. The selection of the positive spanning set to be used is made by setting the value of `pss` in `parameters.m`.

For bound-constrained optimization, the positive spanning set must conform to the geometry of the feasible region, and thus only the possibilities $[I\ -I]$ and $[e\ -e\ I\ -I]$, which include the $2n$ coordinate directions, are allowed.

In the general constrained case, when the number of the approximated active constraints does not exceed the problem dimension, a set of positive generators for the

tangent cone defined by the approximated active constraints gradients is computed. The implementation is made using the SVD of the matrix storing columnwise the appropriated gradients, following the algorithm described in [14].

Any degenerated situation resulting from numerical linear dependence of the gradients of the approximated active constraints halts execution of the optimizer `SID-PSM`.

**6.2.3.** `grad_act.m`. Computes the approximated active constraints at a point provided by the optimizer. A constraint $c_i(x) \leq 0$ is said to be approximated active at a point $x$, for a tolerance value $\epsilon$, if $|c_i(x)| \leq \epsilon$.

The input parameters are the tolerance $\epsilon$, the current point $x$, and a vector storing the values of the constraints. The function returns a matrix whose columns are the gradients of the functions defining the approximated active constraints as well as an integer variable representing the number of approximated active constraints. It calls `grad_const`. The tolerance $\epsilon$ is updated as reported in Figure 2.1.

**6.2.4.** `lambda_poised.m`. Given a matrix $X$ storing the points columnwise and a vector $F_{values}$ storing the corresponding function values, the function `lambda_poised` tries to compute a $\Lambda$-poised set, with a number of points between $s_{min}$ and $s_{max}$. The variable `shessian` specifies testing linear $\Lambda$-poisedness (when `shessian = 0`), testing quadratic $\Lambda$-poisedness (when `shessian = 1`) or testing linear $\Lambda$-poisedness when quadratic $\Lambda$-poisedness has failed (when `shessian = 2`).

After verifying that the set has the minimum number of points required, the set $X$ is reduced to the subset $Y$ of points within a distance of $\Delta$ of the current iterate (see Figure 2.1). If this subset does not have the minimum number of points required then `lambda_poised` returns unsuccessful. Otherwise, the $\Lambda$-poised set computation begins.

If `economic` is equal to `1` or to `2`, then block $\Lambda$-poisedness is tested, first for the whole subset $Y$ (or for its first $s_{max}$ columns, when the maximum number of points allowed in the $\Lambda$-poised set is smaller than the total number of columns in the matrix $Y$). If the $\Lambda$-poisedness condition is not satisfied then a new block test is performed, this time considering only the first $s_{min}$ columns of the matrix $Y$.

When block checking is not specified (`economic = 0`) or when the blocks of points considered did not satisfy the $\Lambda$-poisedness condition, a sequential process for $\Lambda$-poised set computation is initialized. One by one, until all the columns of the matrix $Y$ have been tried or the $s_{max}$ number has been achieved, the columns of $Y$ are tested for $\Lambda$-poisedness. At each iteration of this sequential process, a new set is formed by including the next column in the candidate $\Lambda$-poised set. If this new set is not $\Lambda$-poised, then the column is discarded and the set is not updated. When the sequential process ends, it is checked if the final $\Lambda$-poised set $Y$ has the minimum number of points required ($s_{min}$).

The $\Lambda$-poisedness condition is checked using a condition like $\|Y^{-1}\| \leq \Lambda$, where $\Lambda$ is a fixed constant defined in the file `sid_psm.m` through the variable `lambda`. When `economic = 2`, this condition is tested in an approximated manner. A QR decomposition of $Y$ is computed and the diagonal matrix formed by the diagonal elements of the resulting triangular matrix $R$ is used instead of $Y$ when checking the $\Lambda$-poisedness condition.

The function `lambda_poised.m` returns a variable (`poised`) that assumes values `0`, `1`, or `2`, meaning that $\Lambda$-poisedness could not be achieved, or that a linear or quadratic $\Lambda$-poised set was identified, respectively. In these two last cases it also returns the $\Lambda$-poised set $Y$ and the corresponding objective function values.

**6.2.5.** `match_point.m`. This function scans a list of previously evaluated points in order to find a point in the list that matches a given point (for which no function evaluation has yet been computed). The comparison is made within a given tolerance value (`tol_match`). The $\ell_1$–norms of the points in the list are also stored. Points that dist more than `tol_match` (in the $\ell_1$–norm) from the point to be checked are removed from the list. It is only after this first filtering that a rigorous comparison is made, again using the same tolerance value, but this time considering the $\ell_\infty$–norm. The first point in the list satisfying the above mentioned criterion is returned as well as its corresponding function value. A binary variable (`match`) indicates if a match was found.

**6.2.6.** `mesh_proc.m`. Function `mesh_proc.m` updates the mesh size parameter according to the option specified by the variable `mesh_option`, as described in Subsection 2.3. The mesh expansion and contraction coefficients (`phi` and `theta`, respectively, both defined in the file `parameters.m`) should be provided as input to the function, as well as the simplex derivatives when the update is based on a sufficient decrease condition.

**6.2.7.** `order_proc.m`. This function reorders the columns of a matrix, when regarded as vectors, according to the angles that they make with a given vector (by decreasing order of the corresponding cosines). It returns the reordered matrix and a vector storing the cosines of the angles between the columns of the reordered matrix and the vector considered for ordering.

**6.2.8.** `proj_ort.m`. When the feasible region of the problem is defined using only bound type constraints, function `proj_ort` computes the orthogonal projection of a given point onto this feasible region. It call `func_const` and `grad_const`.

**6.2.9.** `prune_dir.m`. This function prunes the columns of a given matrix, again regarded as vectors, to the most promising direction or to all the potential descent directions according to the descent indicator used for ordering (`pruning = 1` or `pruning = 2`, respectively). In the way it is implemented, it requires a previous call to `order_proc.m`. It returns the pruned matrix.

**6.2.10.** `quad_Frob.m`. Given a set of points, and the corresponding objective function values, function `quad_Frob` computes a quadratic interpolation model for the objective function, as described in Section 2.2. It returns the gradient vector, the Hessian matrix of the model, and a variable `quad` which specifies if the model could be computed or if a previous model should be used.

**6.2.11.** `sid_psm.m`. It is the main program file and it applies a pattern search method to a problem, as described in Section 2. The user modifiable parameters are given in the file `parameters.m`, described in Appendix D. Information about the command line and the output can be found in Section 4. The values of the constants used in the computation of the Λ-poised set are given by:

| **Λ-poised set computation** | |
|---|---|
| **Variable** | **Default value** |
| `lambda` | 100 |
| `sigma` | 1 if unsuccess<br>2 if success and mesh size is unchanged<br>4 if success and mesh size is doubled |

The values for the parameters `s_min`, `s_max`, and `p_max` are given in Table 2.1.

The main program calls `domain`, `func_f`, `gen`, `grad_act`, `lambda_poised`, `match_point`, `mesh_proc`, `order_proc`, `proj_ort`, `prune_dir`, `quad_Frob`, and `simplex_deriv`.

**6.2.12.** `simplex_deriv.m`. Given a set of points, this function computes the simplex gradient and, when appropriately specified, it also computes a diagonal form of the simplex Hessian. In the overdetermined case, the system defining the simplex derivatives is solved in the least-squares sense. When the number of points in the $\Lambda$–poised set is not sufficient for defining a determined system it is computed either a minimum norm solution or the closest solution to the previously computed simplex derivatives (variable `min_norm` set to `1` or `0`, respectively, in the file `parameters.m`). A vector is returned where the first $n$ positions store the simplex gradient components, and, if required, the remaining $n$ store the elements of the diagonal approximation to the simplex Hessian matrix.

REFERENCES


[1] *MATLAB®, The MathWorks$^{TM}$.*

[2] M. A. Abramson, *NOMADm version 4.6 User's Guide*, 2008.

[3] M. A. Abramson, C. Audet, and J. E. Dennis Jr., *Generalized pattern searches with derivative information*, Math. Program., 100 (2004), pp. 3–25.

[4] C. Audet and J. E. Dennis Jr., *Analysis of generalized pattern searches*, SIAM J. Optim., 13 (2002), pp. 889–903.

[5] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Geometry of interpolation sets in derivative free optimization*, Math. Program., 111 (2008), pp. 141–172.

[6] ———, *Geometry of sample sets in derivative free optimization: Polynomial regression and underdetermined interpolation*, IMA J. Numer. Anal., 28 (2008), pp. 721–748.

[7] ———, *Introduction to Derivative-Free Optimization*, MPS-SIAM Series on Optimization, SIAM, Philadelphia, 2009.

[8] A. L. Custódio, J. E. Dennis Jr., and L. N. Vicente, *Using simplex gradients of nonsmooth functions in direct search methods*, IMA J. Numer. Anal., 28 (2008), pp. 770–784.

[9] A. L. Custódio, H. Rocha, and L. N. Vicente, *Incorporating minimum Frobenius norm models in direct search*, Comput. Optim. and Appl., 46 (2010), pp. 265–278.

[10] A. L. Custódio and L. N. Vicente, *Using sampling and simplex derivatives in pattern search methods*, SIAM J. Optim., 18 (2007), pp. 537–555.

[11] G. A. Gray and T. G. Kolda, *Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization*, ACM Trans. Math. Software, 32 (2006), pp. 485–507.

[12] P. D. Hough, T. G. Kolda, and V. J. Torczon, *Asynchronous parallel pattern search for nonlinear optimization*, SIAM J. Sci. Comput., 23 (2001), pp. 134–156.

[13] T. G. Kolda, R. M. Lewis, and V. Torczon, *Optimization by direct search: New perspectives on some classical and modern methods*, SIAM Rev., 45 (2003), pp. 385–482.

[14] R. M. Lewis and V. Torczon, *Pattern search methods for linearly constrained minimization*, SIAM J. Optim., 10 (2000), pp. 917–941.

[15] J. J. Moré, D. C. Sorensen, K. E. Hillstrom, and B. S. Garbow, *The MINPACK Project*, in Sources and Development of Mathematical Software, W. J. Cowell, ed., Prentice-Hall, NJ, 1984, pp. 88–111. `http://www.netlib.org/minpack`.

[16] V. Torczon, *On the convergence of pattern search algorithms*, SIAM J. Optim., 7 (1997), pp. 1–25.

[17] P. Tseng, *Fortified-descent simplicial search method: a general approach*, SIAM J. Optim., 10 (1999), pp. 269–288.

## Appendix A. Output when `output = 1`.

Iteration Report:

| iter | f_value | alpha |
|---|---|---|
| 0 | +1.93600000e-001 | +1.20000000e+000 |
| 1 | +1.93600000e-001 | +6.00000000e-001 |
| 2 | +1.59873890e-001 | +6.00000000e-001 |
| 3 | +4.63975073e-003 | +6.00000000e-001 |
| 4 | +4.63975073e-003 | +3.00000000e-001 |
| 5 | +1.03007230e-003 | +3.00000000e-001 |
| 6 | +1.03007230e-003 | +1.50000000e-001 |
| 7 | +1.03007230e-003 | +7.50000000e-002 |
| 8 | +2.53563073e-005 | +7.50000000e-002 |
| 9 | +2.53563073e-005 | +3.75000000e-002 |
| 10 | +1.75977419e-005 | +3.75000000e-002 |
| 11 | +3.87774104e-006 | +3.75000000e-002 |
| 12 | +1.18954131e-007 | +3.75000000e-002 |
| 13 | +1.18954131e-007 | +1.87500000e-002 |
| 14 | +1.18954131e-007 | +9.37500000e-003 |
| 15 | +1.18954131e-007 | +4.68750000e-003 |
| 16 | +9.65872839e-009 | +4.68750000e-003 |
| 17 | +1.25865660e-009 | +4.68750000e-003 |
| 18 | +8.12074274e-010 | +4.68750000e-003 |
| 19 | +8.12074274e-010 | +2.34375000e-003 |
| 20 | +3.09040366e-011 | +2.34375000e-003 |
| 21 | +3.09040366e-011 | +1.17187500e-003 |
| 22 | +1.81841358e-011 | +1.17187500e-003 |
| 23 | +1.81841358e-011 | +5.85937500e-004 |
| 24 | +1.19577452e-012 | +5.85937500e-004 |
| 25 | +1.19577452e-012 | +2.92968750e-004 |
| 26 | +7.54677228e-014 | +2.92968750e-004 |
| 27 | +7.54677228e-014 | +1.46484375e-004 |
| 28 | +4.71550399e-015 | +1.46484375e-004 |
| 29 | +4.71550399e-015 | +7.32421875e-005 |
| 30 | +2.34485272e-016 | +7.32421875e-005 |
| 31 | +3.24955353e-018 | +7.32421875e-005 |
| 32 | +3.24955353e-018 | +3.66210937e-005 |
| 33 | +2.95789087e-023 | +3.66210937e-005 |
| 34 | +2.95789087e-023 | +1.83105469e-005 |
| 35 | +2.95789087e-023 | +9.15527344e-006 |

Final Report:

Elapsed Time = 6.200e-002

| #iter | #isuc | #fevals | final f_value | final alpha |
|---|---|---|---|---|
| 35 | 18 | 143 | +2.95789087e-023 | +9.15527344e-006 |

Minimum Point:
6.95674952e-001
4.83963639e-001

## Appendix B. Output when `output = 2`.

Iteration Report:

| iter | succ | #fevals | f_value | alpha | active | search | poised |
|---|---|---|---|---|---|---|---|
| 0 | – | – | +1.93600000e-001 | +1.20000000e+000 | 1 | – | – |
| 1 | 0 | 2 | +1.93600000e-001 | +6.00000000e-001 | 1 | 0 | 1 |
| 2 | 0 | 3 | +1.93600000e-001 | +3.00000000e-001 | 1 | 0 | 1 |
| 3 | 1 | 1 | +4.81633896e-002 | +3.00000000e-001 | 0 | 1 | 1 |
| 4 | 1 | 2 | +6.48643546e-003 | +3.00000000e-001 | 0 | 0 | 1 |
| 5 | 0 | 7 | +6.48643546e-003 | +1.50000000e-001 | 0 | 0 | 1 |
| 6 | 1 | 7 | +4.82491251e-003 | +1.50000000e-001 | 0 | 0 | 1 |
| 7 | 1 | 1 | +1.90702676e-004 | +1.50000000e-001 | 0 | 1 | 1 |
| 8 | 0 | 7 | +1.90702676e-004 | +7.50000000e-002 | 0 | 0 | 1 |
| 9 | 0 | 7 | +1.90702676e-004 | +3.75000000e-002 | 0 | 0 | 1 |
| 10 | 1 | 1 | +2.98588332e-005 | +3.75000000e-002 | 0 | 1 | 1 |
| 11 | 1 | 1 | +5.47892983e-006 | +3.75000000e-002 | 0 | 1 | 0 |
| 12 | 0 | 7 | +5.47892983e-006 | +1.87500000e-002 | 0 | 0 | 1 |
| 13 | 0 | 7 | +5.47892983e-006 | +9.37500000e-003 | 0 | 0 | 1 |
| 14 | 1 | 1 | +2.82246220e-010 | +9.37500000e-003 | 0 | 1 | 1 |
| 15 | 0 | 7 | +2.82246220e-010 | +4.68750000e-003 | 0 | 0 | 1 |
| 16 | 0 | 7 | +2.82246220e-010 | +2.34375000e-003 | 0 | 0 | 1 |
| 17 | 1 | 1 | +6.40890624e-012 | +2.34375000e-003 | 0 | 1 | 1 |
| 18 | 0 | 7 | +6.40890624e-012 | +1.17187500e-003 | 0 | 0 | 1 |
| 19 | 0 | 7 | +6.40890624e-012 | +5.85937500e-004 | 0 | 0 | 1 |
| 20 | 1 | 1 | +3.25563200e-013 | +5.85937500e-004 | 0 | 1 | 1 |
| 21 | 1 | 1 | +1.50562756e-013 | +5.85937500e-004 | 0 | 1 | 1 |
| 22 | 0 | 7 | +1.50562756e-013 | +2.92968750e-004 | 0 | 0 | 1 |
| 23 | 0 | 7 | +1.50562756e-013 | +1.46484375e-004 | 0 | 0 | 1 |
| 24 | 1 | 1 | +4.67850637e-015 | +1.46484375e-004 | 0 | 1 | 1 |
| 25 | 1 | 1 | +1.64149055e-017 | +1.46484375e-004 | 0 | 1 | 1 |
| 26 | 0 | 7 | +1.64149055e-017 | +7.32421875e-005 | 0 | 0 | 1 |
| 27 | 0 | 7 | +1.64149055e-017 | +3.66210937e-005 | 0 | 0 | 1 |
| 28 | 0 | 7 | +1.64149055e-017 | +1.83105469e-005 | 0 | 0 | 1 |
| 29 | 1 | 1 | +1.64913830e-018 | +1.83105469e-005 | 0 | 1 | 1 |
| 30 | 1 | 1 | +6.61399639e-019 | +1.83105469e-005 | 0 | 1 | 1 |
| 31 | 1 | 1 | +4.48831197e-022 | +1.83105469e-005 | 0 | 1 | 1 |
| 32 | 0 | 7 | +4.48831197e-022 | +9.15527344e-006 | 0 | 0 | 1 |

Final Report:

Elapsed Time = 9.300e-002

| #iter | #isuc | #fevals | final f_value | final alpha |
|---|---|---|---|---|
| 32 | 15 | 133 | +4.48831197e-022 | +9.15527344e-006 |

Minimum Point:
-3.13103514e-001
9.80338104e-002

## Appendix C. Output when `const = 2`.

Iteration Report:

| iter | succ | #fevals | f_value | alpha | active | search | poised |
|---|---|---|---|---|---|---|---|
| 0 | – | – | +1.93600000e-001 | +1.20000000e+000 | 1 | – | – |
| 1 | 0 | 2 | +1.93600000e-001 | +6.00000000e-001 | 1 | 0 | 1 |
| 2 | 0 | 4 | +1.93600000e-001 | +3.00000000e-001 | 1 | 0 | 1 |
| 3 | 1 | 1 | +1.03503391e-002 | +3.00000000e-001 | 0 | 1 | 1 |
| 4 | 0 | 7 | +1.03503391e-002 | +1.50000000e-001 | 0 | 0 | 1 |
| 5 | 1 | 1 | +9.90874758e-003 | +1.50000000e-001 | 0 | 1 | 1 |
| 6 | 1 | 5 | +2.54593991e-003 | +1.50000000e-001 | 0 | 0 | 1 |
| 7 | 1 | 1 | +8.47504437e-006 | +1.50000000e-001 | 0 | 1 | 1 |
| 8 | 0 | 7 | +8.47504437e-006 | +7.50000000e-002 | 0 | 0 | 1 |
| 9 | 1 | 1 | +1.36312586e-006 | +7.50000000e-002 | 0 | 1 | 1 |
| 10 | 0 | 7 | +1.36312586e-006 | +3.75000000e-002 | 0 | 0 | 1 |
| 11 | 0 | 7 | +1.36312586e-006 | +1.87500000e-002 | 0 | 0 | 1 |
| 12 | 1 | 1 | +2.17762931e-008 | +1.87500000e-002 | 0 | 1 | 1 |
| 13 | 0 | 7 | +2.17762931e-008 | +9.37500000e-003 | 0 | 0 | 1 |
| 14 | 1 | 1 | +1.02170381e-008 | +9.37500000e-003 | 0 | 1 | 1 |
| 15 | 0 | 7 | +1.02170381e-008 | +4.68750000e-003 | 0 | 0 | 1 |
| 16 | 0 | 7 | +1.02170381e-008 | +2.34375000e-003 | 0 | 0 | 1 |
| 17 | 1 | 1 | +5.38011513e-010 | +2.34375000e-003 | 0 | 1 | 1 |
| 18 | 0 | 7 | +5.38011513e-010 | +1.17187500e-003 | 0 | 0 | 1 |
| 19 | 1 | 1 | +3.97355868e-013 | +1.17187500e-003 | 0 | 1 | 1 |
| 20 | 0 | 7 | +3.97355868e-013 | +5.85937500e-004 | 0 | 0 | 1 |
| 21 | 0 | 7 | +3.97355868e-013 | +2.92968750e-004 | 0 | 0 | 1 |
| 22 | 1 | 1 | +9.70949566e-014 | +2.92968750e-004 | 0 | 1 | 1 |
| 23 | 1 | 1 | +3.47054359e-014 | +2.92968750e-004 | 0 | 1 | 0 |
| 24 | 0 | 7 | +3.47054359e-014 | +1.46484375e-004 | 0 | 0 | 1 |
| 25 | 0 | 7 | +3.47054359e-014 | +7.32421875e-005 | 0 | 0 | 1 |
| 26 | 1 | 1 | +2.64262021e-017 | +7.32421875e-005 | 0 | 1 | 1 |
| 27 | 0 | 7 | +2.64262021e-017 | +3.66210937e-005 | 0 | 0 | 1 |
| 28 | 0 | 7 | +2.64262021e-017 | +1.83105469e-005 | 0 | 0 | 1 |
| 29 | 1 | 1 | +1.99025520e-017 | +1.83105469e-005 | 0 | 1 | 1 |
| 30 | 1 | 1 | +2.17441192e-018 | +1.83105469e-005 | 0 | 1 | 1 |
| 31 | 1 | 1 | +5.10723718e-019 | +1.83105469e-005 | 0 | 1 | 1 |
| 32 | 1 | 1 | +4.84716643e-019 | +1.83105469e-005 | 0 | 1 | 1 |
| 33 | 1 | 1 | +2.12657486e-019 | +1.83105469e-005 | 0 | 1 | 1 |
| 34 | 1 | 1 | +1.85815446e-019 | +1.83105469e-005 | 0 | 1 | 1 |
| 35 | 0 | 7 | +1.85815446e-019 | +9.15527344e-006 | 0 | 0 | 1 |

Final Report:

Elapsed Time = 7.800e-002

| #iter | #isuc | #fevals | final f_value | final alpha |
|---|---|---|---|---|
| 35 | 18 | 134 | +1.85815446e-019 | +9.15527344e-006 |

Minimum Point:
-4.77341418e-001
2.27854829e-001

**Appendix D. Variables set in file `parameters.m`.**

| Variable | Default value | Range | Description |
|---|---|---|---|
| **Algorithmic strategies** | | | |
| `always` | 1 | 0,1 | 1 if a quadratic model should always be considered at the search step of the algorithm, once it has been computed; 0 otherwise. |
| `cache` | 0 | 0,1,2 | 0 if no point comparisons are performed prior to function evaluation; 1 if all the points where the objective function has been evaluated are used in point comparisons; 2 if the only points used for comparisons are the ones stored for simplex derivatives computation. |
| `economic` | 0 | 0,1,2 | 0 if a sequential process is used when computing a $\Lambda$-poised set; 1 if block $\Lambda$-poisedness is checked before starting the sequential process; 2 if block checking is allowed and additionally the test for the $\Lambda$-poisedness condition is based on the QR decomposition. |
| `mesh_option` | 0 | 0,1,2,3 | 0 if mesh size is increased in every successful iteration; 1 if the mesh update for successful iterations is based on a sufficient decrease condition, allowing mesh contractions; 2 if the mesh update for successful iterations is based on a sufficient decrease condition, but contractions are not allowed; 3 if the mesh size is maintained in successful iterations, except when two consecutive successful iterations are found using the same direction, where the mesh size is increased. |
| `min_norm` | 1 | 0,1 | 1 if a minimum norm solution is computed when solving underdetermined systems corresponding to simplex derivatives calculations; 0 if the closest solution to the previously calculated simplex derivatives is computed in this situation. |

| | | | |
|---|---|---|---|
| `order_option` | 5 | 0 to 9 | 0 if the vectors in the poll set are tested at each iteration in a consecutive order of storage; 1 if the reordering of the poll directions is based on a simplex descent indicator; 2 if dynamic polling is considered, testing first the last successful poll direction; 3 if a random order is considered for the poll set; 4 if a consecutive, cycling order of storage is considered through all the iterations (only available for unconstrained optimization); 5 ordering strategy identical to 4, except when $\Lambda$-poisedness is achieved. In this case the poll directions are ordered according to the simplex descent indicator; 6 if the reordering of the poll directions is based on a model descent indicator; 7 if the reordering of the poll directions is based on the model values; 8 ordering strategy identical to 4, except when a model is available for use. In this case the poll directions are ordered according to the model descent indicator; 9 ordering strategy identical to 4, except when a model is available for use. In this case the poll directions are ordered according to the model values. |
| `pruning` | 0 | 0,1,2 | 0 if no pruning is required; 1 if only the most promising direction is considered for polling; 2 if all the potential descent directions, according to the descent indicator, are used for polling. |
| `pss` | 2 | 0,1,2,3 | 0 for the minimal positive basis $[-e\ I]$; 1 for the maximal positive basis $[I\ -I]$; 2 for the positive spanning set $[e\ -e\ I\ -I]$; 3 for a positive basis with angles of uniform amplitude among vectors. |
| `regopt` | 1 | 0,1 | 1 if, at the search step, a regression model is considered when the number of points in the interpolation set exceeds the number of points required for complete quadratic interpolation; 0 if some of the interpolation points are discarded, in order to only compute determined quadratic interpolation models. |
| `search_option` | 1 | 0,1 | 0 for no search step; 1 for a search step based on a minimum Frobenius norm model. |
| `shessian` | 0 | 0,1,2 | 0 if the simplex derivatives to be computed correspond to a simplex gradient; 1 if the simplex derivatives to be computed correspond to a simplex gradient and a diagonal simplex Hessian; 2 if one first attempts `shessian = 1` and, in case of failure, then tries `shessian = 0`. |

| | | | |
|---|---|---|---|
| `store_all` | 1 | 0,1 | 1 if all function evaluations are stored for simplex derivatives and model computations; 0 if the only points stored with this purpose are the ones corresponding to successful iterations. |
| `trs_solver` | 0 | 0,1 | 1 if `trs` routine, based on `dgqt` solver of MINPACK2 is used for solving the trust-region subproblems; 0 if the solution of the trust-region subproblems is computed using `trust.m` Matlab function. |
| **Stopping criteria** | | | |
| `stop_alfa` | 1 | 0,1 | 1 if the stopping criterion is based on the mesh size parameter; 0 otherwise. |
| `tol_alfa` | $10^{-5}$ | $]0, +\infty[$ | Lowest value allowed for the mesh size parameter. |
| `stop_fevals` | 0 | 0,1 | 1 if the stopping criterion is based on a maximum number of function evaluations; 0 otherwise. |
| `fevals_max` | 1500 | $\mathbb{N}$ | Maximum number of function evaluations allowed. |
| `stop_grad` | 0 | 0,1,2 | 0 if the stopping criterion does not involve any simplex gradients; 1 if the stopping criterion is related to the norm of a scaled simplex gradient; 2 if the stopping criterion is based on the signs of the approximated directional derivatives (computed using simplex gradients). |
| `tol_grad` | $10^{-5}$ | $]0, +\infty[$ | Convergence tolerance when the stopping criterion is based on simplex gradients. |
| `stop_iter` | 0 | 0,1 | 1 if the stopping criterion is based on a maximum number of iterations; 0 otherwise. |
| `iter_max` | 1500 | $\mathbb{N}$ | Maximum number of iterations allowed. |
| **Mesh size parameters** | | | |
| `alfa` | $\max(1, \|x_{initial}\|_\infty)$ | $]0, +\infty[$ | Initial mesh parameter value. (defined as in Moré and Wild). |
| `phi` | 1 | see [4] | Coefficient for mesh expansion. |
| `theta` | 0.5 | see [4] | Coefficient for mesh contraction. |
| **Constrained problems** | | | |
| `epsilon_ini` | $10^{-1}$ | $]0, +\infty[$ | Initial value for considering a constraint as approximated active. |