The ‘Not-So-Short’ manual for the SAO*i* algorithm

Albert A. Groenwold*

Department of Mechanical Engineering, University of Stellenbosch, Stellenbosch, South Africa.

Version 0.9.0. Released at 12h21 on February 4, 2017.

Abstract

This is the ‘Not-So-Short’ manual for the SAO*i* algorithm. A ‘Quick-Start’ manual is also available. Unfortunately, there is no complete unabridged manual as yet. Until we are able to find the time for writing one, the reader is instead referred to the literature cited herein.

The SAO*i* algorithm is a sequential approximate optimization (SAO) algorithm for bounded or inequality[1] constrained nonlinear programming, based on convex separable approximation functions. The subproblems may be solved in the dual space, or via a diagonalized quadratic programming (QP) approach.

The approximations used are diagonal quadratic in nature. The user may select approximate diagonal curvatures that describe his or her problem best. (For truly separable problems, *exact* second order information may of course be used.) Notwithstanding the diagonal quadratic nature of the approximations used, it is nevertheless possible to accurately and efficiently optimize problems that exhibit strong monotonicities, like those present in structural optimization. This is achieved by constructing the diagonal quadratic approximations to the reciprocal and/or the exponential approximations. Alternatively, the user may select the diagonal quadratic approximation to the approximations present in the well-known method of moving asymptotes (MMA) of Svanberg. (Note that this is philosophically very different to using the MMA algorithm itself.)

What is more: it is in principle possible to construct the diagonal quadratic approximation to any arbitrary (separable) approximation; the user may easily add such an approximation by specifying the approximate diagonal curvatures in a user routine.

The algorithm is in particular aimed at (sparse) large scale ‘simulation-based’ optimization, understood to be optimization problems with computationally demanding numerical simulations or models in the optimization loop. This may make the evaluation and storage of second order information impractical. Typical examples include the optimization of systems or structures modeled using the finite element method (FEM), computational fluid dynamics (CFD) simulations, etc. Having said this, the reciprocal-like behavior present in many a *structural optimization problem* is the main intended application of the algorithm.

The gradients of the constraints may be stored in dense or sparse form.

Note that there are no differences between the dense and sparse implementations, other than the algebra used. In other words, very large scale problems are not treated differently to small scale problems, as is for example done in the BIGDOT algorithm of Vanderplaats. However, since this is work in progress, some options and settings available in the dense setting have unfortunately not yet been ported to the sparse setting.

*Versions earlier than 7.0.0 were documented in collaboration with L.F.P. Etman, *Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands.*

[1]Since Version 0.8.4, equality constraints are also provided for, but this manual has not been updated to reflect this. A look at subdirectory `Examples/equalities/` should suffice to get interested readers started.

For convex problems, global convergence may be enforced using a trust region method based on a nonlinear acceptance filter, conservatism, or so-called 'filtered-conservatism'. (For non-convex problems, this guarantees convergence to at least some local minimum.)

Although the algorithm is primarily aimed at unimodal (structural) optimization problems, multimodal problems may also be optimized. This is effected via a multi-start strategy, combined with a Bayesian acceptance condition. The method works well for problems characterized by global optima for which the region of attraction is comparable or larger than the regions of attraction of other local minima.

The SAO*i* algorithm was developed for objective and constraint that are (at least) once continuously differentiable. However, some of the approximations present in the algorithm rely only on first order gradient information, and not on zeroth order function value information. This makes the algorithm suitable[2] for use in the presence of the type of *non-smooth functions* that arise due to remeshing strategies, etc. In other words, when the optimization problem is based on the solutions of systems of partial differential equations, in combination with variable discretization techniques (e.g. remeshing in spatial domains, and/or variable time stepping in temporal domains), the SAO*i* algorithm represents a suitable algorithmic option.

Note however that general non-smooth problems, e.g. noisy problems, are *not* an intended application of the SAO*i* algorithm. (Possibly contrary to popular belief, remeshing results in 'well-behaved' step discontinuities, and can therefore be addressed by some of the approximations present in the SAO*i* algorithm.)

The latest version of the software is 0.9.0, and it is freely available for academic purposes. Currently, only a FORTRAN version is available. Academic versions lower than 1.0.0 are beta-test versions; feedback will sincerely be appreciated. Note that there are many incomplete and experimental options in the code, not referred to in the documentation. These are mostly for experimentation purposes, including student projects, and should only be used with extreme care.

What is more, we develop and maintain this code after hours. Hence, many bugs and even errors are present in the code; academic versions lower than 1.0.0 should be used in this spirit. A robust, well-coded commercial version is available upon request, but this version unfortunately supports only few of the options and settings available in the academic version.

Finally, versions for parallel processing on CPUs (using openmp and mpi) and GPUs (graphical processor units) also exist, but these are so experimental that we release them only to collaborators.

Keywords: sequential approximate optimization (SAO); quadratic program (QP); simulation-based optimization; structural optimization; large-scale optimization; global convergence; conservative convex separable approximation; trust region; filtered-conservatism; diagonal quadratic approximation; Falk dual, global optimization, non-smooth optimization.

[2]Again, this manual has not been updated to reflect this; a look at subdirectory `Examples/specialized/discont/` should suffice to get interested readers started.

Contents

1 Introduction 5

2 Problem statement 6
2.1 Multimodal problems . . . 6

3 Algorithm structure 7
3.1 User routines . . . 7
3.1.1 Initialize.f . . . 7
3.1.2 Functions.f . . . 7
3.1.3 Gradients.f . . . 7
3.2 Output . . . 7
3.3 Include files . . . 8
3.4 User data . . . 8
3.4.1 Array dimensions . . . 9

4 Examples 9
4.1 Popular unimodal test problems . . . 9
4.2 Popular multimodal test problems . . . 9
4.3 Very large scale optimization (VLSO) . . . 9

5 Comments on the algorithm 9
5.1 The approximations . . . 9
5.1.1 Comments on the quadratically approximated MMA approximations . . . 10
5.1.2 User approximations . . . 11
5.2 Subproblems . . . 11
5.2.1 Subproblems based on duality . . . 11
5.2.2 Subproblems based on QP statements . . . 12
5.2.3 A list of currently available subproblems and solvers . . . 12
5.2.4 Installation of the `Galahad` and `NAG` solvers . . . 13
5.3 On convergence . . . 13
5.3.1 Conservatism . . . 13
5.3.2 Filter-based trust region . . . 13
5.3.3 Filtered conservatism . . . 13
5.3.4 Outstanding . . . 14
5.4 Infeasible subproblems . . . 14
5.4.1 The bounded dual . . . 14
5.4.2 Relaxation . . . 14
5.5 Brief notes on SIMP-like topology optimization problems . . . 15
5.5.1 Continuation strategies . . . 15
5.5.2 Mesh independence filtering . . . 15
5.6 Description of parameters, keywords and identifiers . . . 15

6 Storage schemes 16
6.1 Dense scheme . . . 16
6.2 Pseudo-sparse scheme . . . 16
6.3 Sparse scheme with `nnz` constant and known . . . 16
6.4 Sparse scheme with `nnz` variable and possibly unknown . . . 16

7 On efficiency 17
7.1 Scaling . . . 17
7.2 The approximate higher order curvatures . . . 17
7.2.1 NOTE . . . 17
7.3 Failure of the algorithm, and robustness . . . 17

8 Current activities and future versions **18**

9 User-contributions **18**

10 Availability and conditions of use **18**
10.1 Restrictions . . . 18

11 General **19**
11.1 Bugreports, wishlists and support . . . 19
11.2 Code efficiency . . . 19
11.3 Tuned BLAS . . . 19
11.4 Previous versions and published results . . . 19
11.5 Very brief historical perspective . . . 19
11.6 Trivia . . . 19

A Installation **23**
A.1 Supported platforms and FORTRAN compilers . . . 23
A.1.1 Tested compilers . . . 23
A.1.2 Discontinued compilers . . . 23
A.1.3 Compiler flags . . . 23
A.1.4 The timer command 'etime' in function seconds() . . . 24
A.1.5 The command 'isNaN()' in subroutine testNaN() . . . 24
A.2 Installation and 'make' . . . 24
A.3 Support for `GALAHAD` . . . 24
A.4 Support for `NAG` . . . 25
A.5 Note . . . 25

B A unimodal example: simultaneous shape and sizing design of a 2-bar truss **26**
B.1 Example problem formulation . . . 26
B.2 Running the code without coding gradients . . . 26
B.2.1 User routines . . . 26
B.2.2 Results . . . 27
B.3 Running the code with gradients . . . 28
B.3.1 User routines . . . 28
B.3.2 Results . . . 30
B.4 Running the code with different approximations . . . 31
B.4.1 Results . . . 31
B.5 Running the code with guaranteed convergence . . . 31
B.5.1 Results . . . 31

C A multimodal example: the six-hump camelback problem **32**
C.1 Example problem formulation . . . 32
C.1.1 User routines . . . 32

D Program structure **34**
D.1 Parameters, keywords, variables, identifiers & type declarations . . . 34

E On the diagonal curvatures **39**
E.1 Estimating the principal curvatures . . . 39
E.2 Convexity . . . 40
E.3 Some details on the approximations: Deriving T2:R . . . 40

F Multimodal optimization **42**
F.1 Comment . . . 42
F.2 Proof of the Bayesian stopping criterion . . . 42

1 Introduction

Sequential approximate optimization (SAO) methods are fairly simple algorithmic options for solving nonlinear optimization problems. They are aimed at (very) large scale simulation-based optimization, which require huge computational resources. Typical examples include the optimization of systems or structures modeled using the finite element method (FEM), or computational fluid dynamics (CFD) simulations.

In SAO, the main idea is to sequentially approximate all complex (‘difficult’ or nonlinear) objective and constraint functions present in the real optimization problem by simpler explicit or analytical functions, which (hopefully) represent the true functions well. These ‘replacement’ functions are called approximation functions, or ‘approximations’ for short. When the approximate functions are constructed at some given design point or iterate, the resulting approximate optimization problem is called an ‘approximate sub-optimization problem’, or ‘approximate subproblem’ for short.

The solution of a given approximate subproblem may be obtained using any suitable continuous programming method. Thereafter, the optimal point of the current subproblem becomes the starting point for the next subproblem. This process is repeated until either the sequence of iterates converges, or until some maximum number of iterations have passed.

SAO algorithms depend crucially on not only the formulation of accurate approximation functions, but also (if sometimes to a lesser extent), efficient solvers for the approximate subproblems.

It is at this stage prudent to mention the differences between SAO and sequential quadratic programming (SQP). Sequential quadratic programming methods construct quadratic approximations to the objective function f_0, and linear approximations to the constraints f_j. (The Hessian of the QP problem however is the Hessian of the Lagrangian, and not the Hessian of the objective function f_0, as one might expect after a first casual glance at the problem formulation.)

Without much doubt, if arguably, SQP methods are the state of the art in mathematical optimization. However, they require the evaluation and storage of the Hessian matrix, which is not practical for large-scale simulation-based optimization. (The storage requirements are of $O(n^2)$, with n the number of design variables present. The computational requirements are also considered prohibitive. Finally, many simulation packages are ‘black-boxes’, i.e. the function values are available, and sometimes the derivatives, but Hessian information is almost never available[3].)

SAO methods instead use intermediate variables to introduce desirable nonlinearities into the approximations; it is in part hoped that this will alleviate the shortcomings that arise from neglecting the interaction between the variables in the first place (although the intermediate variable approach can in general of course not be expected to yield comparable accuracy to using full Hessian information). The approximations used in SAO are typically more accurate than the linear Taylor expansion, since additional ‘known’ nonlinearities are exploited to make the response ‘more linear’, using some clever approach, which may be heuristic. However, as said, the approximations used in SAO seldom are as accurate as a complete quadratic Taylor expansion, simply because exact second order information is not used.

In principle, many different intermediate variables may be used in SAO. In *structural optimization*, (separable) reciprocal-like intermediate variables are widely used, due to the very importance of reciprocal relationships in structural modeling[4].

Often, SAO algorithms for simulation-based optimization exploit the advantages of duality, which is not quite as easy a topic as SAO itself. However, if the subproblems are strictly convex and separable (in the primal variables), dual methods may readily be implemented[5].

Incidentally, this constitutes another difference between (popular) SAO implementations and SQP methods: the former almost invariably use a dual statement due to Falk, which effectively makes it impossible for SQP methods to compete with SAO methods if the number of constraints is (far) less than the number of design variables. (Barring the storage requirements of SQP, the contrary may be expected to hold if the number of constraints is (far) higher than the number of design variables.)

[3]The interested reader is reminded of a typical structural optimization problem, solved using the finite element method. Then, the design variables often or invariably depend on the unknown nodal displacements, which may be obtained using the adjoint or direct methods, e.g. see Haftka and Gürdal [1], and many others. This is expensive enough from a computational effort point of view; to attempt to evaluate second order information is unreasonable, if not prohibitive.

[4]For statically determinate structures subject to stress and displacement constraints only, the reciprocal approximation is exact.

[5]The method we present herein provides for a ‘classical’ dual SAO approach, but also a QP-like approach.

Let us however return to SAO methods: the main difficulty in dual SAO methods is to select approximation functions that approximate the true problem as accurately as possible, while satisfying the requirements needed to invoke duality in the first place. Clearly, different approximations may be expected to be optimal for different problems. Accordingly, we provide for a range of possibilities in the SAO*i* algorithm. To facilitate easy inclusion into a dual framework, the approximations used are all diagonal quadratic (and convex and separable). The desired nonlinearities and monotonicities of the approximate functions are then introduced by constructing the quadratic approximation to a number of popular approximations based on various intermediate variables. Examples are the reciprocal approximation, the exponential approximation, and even the approximations used in the popular method of moving asymptotes (MMA) algorithm. However, our approximations are truly quadratic, albeit that they only store diagonal information[6]. For details on the construction of quadratic approximations to the approximations based on arbitrary intermediate variables, see References [2, 3].

2 Problem statement

The SAO*i* algorithm is aimed at unimodal or convex nonlinear inequality-constrained simulation-based optimization problems $\mathrm{P}_{\mathrm{NLP}}$ of the form

$$\begin{aligned} \min\ & f_0(\boldsymbol{x}) \\ \text{subject to}\ & f_j(\boldsymbol{x}) \leq 0, \qquad j = 1, 2, \cdots, n_i, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, 2, \cdots, n, \end{aligned} \tag{1}$$

where $f_0(\boldsymbol{x})$ represents a real-valued scalar objective function, and the $f_j(\boldsymbol{x})$ represent n_i real-valued scalar *inequality* constraint functions[7]. $f_0(\boldsymbol{x})$ and the $f_j(\boldsymbol{x})$ depend on the n real (design) variables $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}^T \in \mathcal{X} \subset \mathcal{R}^n$, hence $\check{x}_i$ and $\hat{x}_i$ respectively indicate lower and upper bounds on variable x_i.

The functions $f_j(\boldsymbol{x})$, $j = 0, 1, 2, \cdots, n_i$ are assumed to be (at least) once continuously differentiable.

2.1 Multimodal problems

The SAO*i* algorithm is aimed at unimodal problems, but smooth multimodal problems may also be optimized. This is effected via a multi-start strategy, combined with a Bayesian acceptance condition due to Snyman and Fatti [4], and related to the procedure proposed by Zielinski [5].

In the acceptance condition, the only assumption made is that the region of the attraction of the global optimum is comparable to, or larger than, the region of attraction of any other local optimum. In many cases, this seems like a reasonable and mild assumption.

The approach is very simple, and merely requires that some desired confidence level q^* is prescribed; the automated multi-start strategy is then terminated when the probability of convergence to the global optimum is larger than, or equal to, q^*. Typical values for q^* are 0.99 through 0.999.

For details, see the examples mentioned in Section 4.2, and the explicit example (the well-known six hump camelback problem) given in Appendix C.

NOTE: it is possible to construct a hybrid optimization algorithm that first initiates a global search pattern before switching to a local refinement phase. This may be done by including a suitable algorithm with global search capability like differential evolution (DE) or particle swarm optimization (PSO), etc.

Doing this is rather easy; the point of insertion of the global algorithm is clearly indicated in the file `split.f`.

[6]To reiterate: this means that the subproblems can be solved using QP methods, *not* that accurate second order information regarding the original problem is used.

[7]It is not required that constraint functions are present. Viz., simple bound constrained problems may also be considered. However, it is *required* that bounds for the primal design variables x_i are given; unbounded problems may of course be considered using artificially large bounds, say 10^{20}.

3 Algorithm structure

Using the SAO*i* algorithm is very simple: it merely requires the modification of the three 'user' subroutines mentioned in the following.

3.1 User routines

The user routines to be modified by the user are:

- `Initialize.f`
- `Functions.f`
- `Gradients.f`

Subroutine `Gradients.f` is only required if the parameter `finite_diff` in the routine `Initialize.f` is set to `false`, else finite differences are resorted to. (It is normally preferable to not compute the gradients using finite differences, if at all possible.)

The three user routines mentioned above are listed in some detail in the appendices, and are briefly discussed in the following.

Optionally, `diaHessUser.f` may be edited if user approximations are given (or when *exact second information is specified for separable problems*), see Section 5.1.2.

3.1.1 Initialize.f

In `Initialize.f` it is only necessary to specify the number of design variables `n`, the number of inequality constraints `ni`, the starting points `x(i)`, and the lower and upper bounds `x_lower(i)` and `x_upper(i)` respectively. Note that the bound *have* to be specified. If no bounds are present, suitably 'large' numbers may be used.

It is possible to specify a few optional parameters. Of these, the most important are `approx_f`, `approx_c`, `force_converge`, and `finite_diff`. The optional parameters[8] are discussed in Appendix D.1.

For additional information, see Appendix B.

3.1.2 Functions.f

In `Functions.f` it is required to specify the objective functions f_0, as well as the constraint functions f_j, $j = 1, 2, \cdots, m$.

For additional information, see Appendix B.

3.1.3 Gradients.f

`Gradients.f` is only needed if the parameter `finite_diff` in `Initialize.f` is set to `.false.`

In this case, it is required to specify the partial derivatives of objective function $\partial f(\boldsymbol{x})/\partial x_i$, $i = 1, 2, \cdots n$, and the partial derivatives of the inequality constraints $\partial f_j(\boldsymbol{x})/\partial x_i$, $j = 1, 2, \cdots n_i$, $i = 1, 2, \cdots n$.

For additional information, see Appendix B.

3.2 Output

A number of output files[9] are created by the SAO*i* algorithm, namely:

[8] The optional parameters are passed via an 'include' file, and are not passed via the subroutine calls.

[9] Users who wish to output additional data from the user routines to file, should note that the files mentioned above are respectively attached to units 8 through 14. In addition, the l-BFGS-b solver used writes output to unit 20. Hence, it is recommended that users start at say unit number 21,

- `Variables.out`

  This file lists the initial point $x_i^{\{0\}}$, the final point $x_i^{\{*\}}$, and the lower and upper bounds $\check{x}_i$ and $\hat{x}_i$ respectively for each component $i = 1, 2, \cdots, n$.

- `History.out`

  This file lists the function values, maximum constraint violations, step size information, and the number of active bounds and constraints as the iterations proceed. This file echoes the data written to the screen.

- `Tolerance-X.out`

  This file lists the achieved tolerance w.r.t. the primal variables $\boldsymbol{x}$.

- `Tolerance-KKT.out`

  This file lists the achieved tolerance w.r.t. the KKT residual.

- `Constraints.out`

  This file lists the final values of the constraints and their associated dual variables.

- `Warnings.out`

  This file lists (non-fatal) warnings issued during execution, which may or may not influence convergence, but also fatal errors. For the warnings, severity 0 implies that convergence is definitely not impaired, whereas severity 10 implies that convergence has almost certainly been impaired.

- `CheckGradients.out`

  This file compares the user specified gradients in `Gradients.f` with their forward finite difference values, if `check_grad = .true.` is specified in `Initialize.f` (the default is `check_grad = .false.`).

3.3 Include files

In subroutine `Initialize.f`, many of the scalar parameters and keywords are not passed in the normal fashion via the subroutine call; instead, they are made available via include files. This is a legacy of `common` blocks in early versions, but not detrimental to functionality at all. The average user may simply specify the required scalar parameters and keywords in subroutine `Initialize.f` without worrying about the specific algorithm structure. The examples in the appendices should suffice in most cases.

If optional data is specified, it is simply required that the file `ctrl_set.f` is included *after* the data is specified; again, see the examples in the appendices.

If optional data already specified is needed in a file, it is required that the file `ctrl_set.f` is included *before* the point where the data is needed.

3.4 User data

The four user arrays `iuser()`, `luser()`, `cuser()` and `ruser()` are available in the three user routines `Initialize.f`, `Functions.f` and `Gradients.f`, and they may be used to pass data around in the algorithm. The data types are:

- `iuser(*)` - integer array
- `luser(*)` - logical (Boolean) array
- `cuser(*)` - character array
- `ruser(*)` - real (double precision) array

The default dimension of all 4 arrays is 5, but this may be increased to an arbitrary number in `size.h`.

since units 15 through 19 are reserved for possible use in future versions of the SAO*i* algorithm.

3.4.1 Array dimensions

The maximum array dimensions are set at 500000; this represents the maximum number of *primal* and *dual* variables n and ni respectively. If this is insufficient, it may be increased in the file `size.h`. (On a typical machine with say 2 GB of memory, far more than 500000 variables can normally be handled, although the actual number that can be handled also depends on the number of constraints declared in `Initialize.f`.)

4 Examples

4.1 Popular unimodal test problems

The distribution includes an Examples subdirectory, which contains a few example problems in further subdirectories. For example, the default example problem for the SAO*i* algorithm resides in `Examples/unimodal/default`. This is a simple problem proposed by Svanberg [6], and it is detailed in Appendix B of this manual.

To run any of the other examples, simply copy all the files in the desired subdirectory to the working directory. (The original example can of course always be restored from `Examples/unimodal/default`.)

Subdirectory `Examples/unimodal/stencil` contains blank 'stencils' or 'macros' to make it really easy for the user to add his or her own problems. The file `Readme.uni.AG` in subdirectory `Examples/unimodal` lists the example problems currently available.

4.2 Popular multimodal test problems

For multimodal test problems, subdirectory `Examples/multimodal` is similar to the above mentioned subdirectory `Examples/unimodal`, except that the former has no `default` subdirectory. There is however an `Examples/multimodal/stencil` subdirectory, to make it easier to construct multimodal test problems.

Finally, please feel free to send any interesting additional problems to `albertg@sun.ac.za` for inclusion in future versions of the distribution.

4.3 Very large scale optimization (VLSO)

Since Version 0.5.7, we have added routines for (sparse) very large scale optimization (VLSO). The routines have been coded rather hurriedly, and are made available due to a few requests. We plan to improve all the routines for VLSO before releasing Version 0.9.0 (and to include many features not yet available in VLSO).

For very large scale optimization (VLSO), subdirectory `Examples/veryLarge` is similar to the above mentioned subdirectory `Examples/unimodal`, except that the former again has no `default` subdirectory. Again, there is an `Examples/veryLarge/stencil` subdirectory.

5 Comments on the algorithm

This section aims to preempt the most likely Q&A's, but is nevertheless highly condensed. For detailed comments on the algorithm, the reader is referred to the cited literature.

5.1 The approximations

While many approximations are possible in SAO, we have restrict ourselves to the diagonal quadratic approximation

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \frac{\partial f^{\{k\}}}{\partial x_i}(x_i - x_i^{\{k\}}) + \frac{1}{2}\sum_{i=1}^{n} c_{2_i}^{\{k\}}(x_i - x_i^{\{k\}})^2, \tag{2}$$

since this allows for a very simple form of the dual [2, 3]. In addition, the formal link between SQP and SAO methods is hereby made [7], albeit only to diagonal QP methods at this stage.

In (2), the only unknowns are the $c_{2_i}^{\{k\}}$, to be determined using some sensible condition or conditions. Without elaboration, the possibilities[10] in the current version of the SAO*i* algorithm are enumerated as follows:

1. A spherical quadratic approximation based on function values, see [3, 8, 9].
2. A spherical quadratic approximation based on error norm of the gradients, see [10].
3. A nonspherical approximation based on the components of the gradients, see [8].
4. The diagonal quadratic Taylor series expansion to the reciprocal approximation, see [2, 3, 7, 11].
5. The diagonal quadratic Taylor series expansion to the exponential approximation, see [2, 3, 11].
6. The diagonal quadratic Taylor series expansion to the MMA approximations of Svanberg, see [2, 3, 12].
7. The diagonal quadratic Taylor series expansion to the CONLIN approximations of Fleury and Braibant, see [13].

For an elaboration of the foregoing, the reader is urged to read Appendix E and the cited literature, but some comments are in order:

- Approximations 1 and 2 are expected to do well for functions that are *water-holding*[11].
- Approximation 3 is a bit more unpredictable: sometimes it performs surprisingly well, other times surprisingly bad.
- Approximations 4 through 6 are expected to do well for functions characterized by significant monotonicities (like those present in many structural optimization problems).
- Approximations 4, 5 and 7 have only been implemented for positive design variables in the current version of the SAO*i* algorithm.
- Our experiments suggest that 7 is ideally suited for topology optimization in the presence of *local stress constraints*, a very difficult problem of high dimensionality.

The options mentioned above may be selected in `Initialize.f`, by setting `approx_f = f` for the objective function f_0, and `approx_c = c` for the constraints f_j, with `f` and `c` the selected number from 1 to 7 from the list above. In the current version, the same approximation is used for all the constraints. (However, also see Section 5.1.2.)

For an example, see Appendix B.

5.1.1 Comments on the quadratically approximated MMA approximations

It should be noted that the diagonal quadratic approximations to the MMA approximations of Svanberg may give very different results to the MMA approximations themselves. Not only do we only construct the second order approximation to the MMA, but the approximation we depart with also differs slightly: the MMA approximations depend on small positive parameters to ensure that the approximations are convex. In our implementation, convexity is enforced via the approximate higher order curvatures $c_{2_i}^{\{k\}}$.

Also, when conservatism is invoked, the conservatism of the MMA approximations is varied by tightening or relaxing the movable asymptotes, while we have again opted to simply manipulate the higher order curvatures $c_{2_i}^{\{k\}}$.

[10]Many other possibilities exist (and have indeed been implemented in the development version of the production code).

[11]With this, we mean functions that appear like say a cup or saucer in 2-D; a typical example being $f = (x_1 - a)^2 + (x_2 - b)^2$. While not obvious from the terminology, this includes the linear approximation.

5.1.2 User approximations

The user may construct his or her own approximation, by setting `approx_f = 100`, and/or `approx_c = 100` in Initialize.f. It is then required that applicable curvatures $c_{2_i}^{\{k\}}$ are coded in `Hessian.f`. This should not be too difficult if the options present are used as examples. (Note that this also provides for the case when different approximations are desired for different constraints.)

5.2 Subproblems

5.2.1 Subproblems based on duality

The current academic version of the SAO*i* algorithm is in part based on the dual of Falk [14]. Since the approximations used are separable, the subproblems can very efficiently be solved in the dual space, and the SAO*i* algorithm is particularly efficient if the number of constraints n_i is (far) less than the number of design variables n.

As an example: in classical minimum compliance topology optimization, there is only a single constraint present, while the number of design variables n can easily run into the millions. It is doubtful if this problem can be solved more efficiently by methods other than those based on the Falk dual. It seems quite reasonable to generalize this statement to problems for which $n_i \ll n$.

It is worth mentioning that the different higher order curvatures $c_{2_i}^{\{k\}}$ mentioned in the foregoing do *not* influence the primal-dual relationships. What is more: it is easy to construct subproblems in which the objective function and the constraints are all based on the exponential approximation, while still allowing for an analytical relationship between the primal and the dual variables. This is not possible with classical implementations of the exponential approximation [15].

The development is as follows: given primal approximate optimization subproblems based on the approximate functions $\tilde{f}_j$, $j = 0, 1, \ldots, m$ given by the (convex) diagonal quadratic approximations (2), the approximate dual subproblem $P_D[k]$ becomes

$$\begin{aligned} &\max_{\boldsymbol{\lambda}} \{\gamma(\boldsymbol{\lambda}) = \tilde{f}_0^{\{k\}}(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{f}_j^{\{k\}}(\boldsymbol{x}(\boldsymbol{\lambda}))\}, \\ &\text{subject to } \lambda_j \geq 0, \qquad j = 1, 2, \ldots, m, \end{aligned} \tag{3}$$

where the notation $\boldsymbol{x}(\boldsymbol{\lambda})$ implies minimization with respect to $\boldsymbol{x}$. The primal-dual relationships between the primal variables x_i, $i = 1, 2, \ldots, n$ and the dual variables λ_j, $j = 1, 2, \ldots, m$, are given by

$$x_i(\boldsymbol{\lambda}) = \begin{cases} \beta_i(\boldsymbol{\lambda}) & \text{if} \quad \check{x}_i^{\{k\}} < \beta_i(\boldsymbol{\lambda}) < \hat{x}_i^{\{k\}}, \\ \check{x}_i^{\{k\}} & \text{if} \quad \beta_i(\boldsymbol{\lambda}) \leq \check{x}_i^{\{k\}}, \\ \hat{x}_i^{\{k\}} & \text{if} \quad \beta_i(\boldsymbol{\lambda}) \geq \hat{x}_i^{\{k\}}, \end{cases} \tag{4}$$

and

$$\beta_i(\boldsymbol{\lambda}) = x_i^{\{k\}} - \left(c_{2i_0}^{\{k\}} + \sum_{j=1}^{m} \lambda_j c_{2i_j}^{\{k\}} \right)^{-1} \left(\frac{\partial f^{\{k\}}}{\partial x_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial f_j^{\{k\}}}{\partial x_i} \right). \tag{5}$$

For details, the reader is referred to our previous efforts.

The dual solvers In the current academic version of the SAO*i* algorithm, there is only one solver available, namely the limited memory bound constrained l-BFGS-b solver [16, 17] developed by Zhu and co-workers.

We have found the l-BFGS-b solver to be highly efficient for solving Falk-like dual formulations, notwithstanding the fact that the second order derivatives resulting from the Falk dual are discontinuous. (These discontinuities result from the modified definition domain of the dual when fixed primal variables become free, and *vice versa.*)

For an important elaboration, also see Section 7.3.

5.2.2 Subproblems based on QP statements

Since (2) is (diagonal) quadratic, it is easy to construct (diagonal) QP subproblems, which may then be solved using any available (sparse) QP solver.

Quadratic program $P_{QP}[k]$ is written as

$$\begin{aligned} \min_{\boldsymbol{s}} \ & \bar{f}_0^{\{k\}})(\boldsymbol{s}) = f_0(\boldsymbol{x}^{\{k\}}) + \nabla f_0^T(\boldsymbol{x}^{\{k\}})\boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T\boldsymbol{Q}^{\{k\}}\boldsymbol{s} \\ \text{subject to } \ & \bar{f}_j^{\{k\}}(\boldsymbol{s}) = f_j(\boldsymbol{x}^{\{k\}}) + \nabla f_j^T(\boldsymbol{x}^{\{k\}})\boldsymbol{s} \leq 0, \\ & \| \boldsymbol{s} \|_\infty \leq \Delta^{\{k\}}, \end{aligned} \tag{6}$$

with $j = 1, 2, \ldots, m$, $\boldsymbol{s} = (\boldsymbol{x} - \boldsymbol{x}^{\{k\}})$ and $\boldsymbol{Q}^{\{k\}}$ the Hessian matrix of the approximate Lagrangian at $\boldsymbol{x}^{\{k\}}$.

Using the diagonal quadratic objective function and constraint function approximations $\bar{f}_j, j = 0, 1, \ldots, m$, the approximate Lagrangian $\mathcal{L}^{\{k\}}$ equals

$$\mathcal{L}^{\{k\}} = \tilde{f}_0(\boldsymbol{x}^{\{k\}}) + \sum_{j=1}^{m} \lambda_j^{\{k\}} \tilde{f}_j(\boldsymbol{x}^{\{k\}}), \tag{7}$$

with

$$Q_{ii}^{\{k\}} = c_{2i_0}^{\{k\}} + \sum_{j=1}^{m} \lambda_j^k c_{2i_j}^{\{k\}}, \tag{8}$$

and $Q_{il} = 0 \ \forall \ i \neq l, i, l = 1, 2, \ldots, n$. The Lagrange multiplier values $\lambda_j^{\{k\}}$ follow from the multiplier estimates due to the solution of the QP subproblem at the previous iterate $\boldsymbol{x}^{\{k-1\}}$. This approach is very similar to the well-known classical SQP method the fundamental difference being the way the Hessian matrix of the Lagrangian function is determined. Instead of using the exact or approximate quasi-Newton Hessian matrices commonly used in classical SQP algorithms, we use only approximate diagonal terms, estimated from suitable intervening variable expressions for the objective function and all the constraints. As with the dual subproblems, we apply convexity conditions to arrive at a strictly convex QP subproblem with a unique minimizer.

The quadratic programming problem requires the determination of the n unknowns x_i, subject to m linear inequality constraints and the trust region bounds. Efficient QP solvers can typically solve problems with very large numbers of design variables n and constraints m. Obviously, it is imperative that the diagonal structure of $\boldsymbol{Q}$ is exploited when the QP subproblems are solved. For details, see Reference [7].

The QP solvers At the moment, we have an interface to the (commercial) `NAG` QP solver `e04nqf` available in the SAO*i* algorithm, but we are including additional (open-source) QP solvers, or solvers that are freely available for academic use, but not commercial use.

In this last class, an interface to the QP solvers `lsqp`, `qpa`, `qpb` and `qpc`, available in `GALAHAD` [18], is available. As said, while `GALAHAD` is not open source, it is freely available to academics. However, users of the SAO*i* algorithm should download `GALAHAD` from the original repositories maintained by the authors of `GALAHAD`. (A word of warning: the `GALAHAD` QP solvers require additional routines from the HSL Mathematical Software Library (formerly the Harwell subroutine library) and downloading the necessary files, and installing them in the correct locations, may require a few minutes of patience.)

The `GALAHAD` solver `lsqp` solves separable quadratic problems, and we have found this solver to be highly efficient, it is as close to tailor made for the SAOi environment as one could wish for. For our purposes, we have found the `qpb` solver to also be very efficient; more so than the `qpa` and `qpc` solvers, which seems to corroborate the findings of the original authors of `GALAHAD`. In fact, use of the `qpa` and `qpc` solvers is not recommended; they are mainly available for experimentation by our students.

5.2.3 A list of currently available subproblems and solvers

Currently, we have the following subproblems and associated solvers available:

- `subsolver` = `1`: Falk-like dual subproblems, solved using the `l-bfgs-b` solver.
- `subsolver` = `20`: the `NAG` `e04nqf` QP solver to solve the equivalent diagonal QP problem
- `subsolver` = `25`: the `GALAHAD` `lsqp` separable QP solver to solve the equivalent diagonal QP problem.
- `subsolver` = `26`: the `GALAHAD` `qpa` QP solver to solve the equivalent diagonal QP problem.
- `subsolver` = `27`: the `GALAHAD` `qpb` QP solver to solve the equivalent diagonal QP problem.
- `subsolver` = `28`: the `GALAHAD` `qpc` QP solver to solve the equivalent diagonal QP problem.

Many more solvers and subproblem statements are under development. Some are included in the code for specific student projects, and should only be used by adventurous folks.

5.2.4 Installation of the `Galahad` and `NAG` solvers

See Appendix A.

5.3 On convergence

5.3.1 Conservatism

Since the approximations used are convex and separable, they may easily be cast in the framework of conservative convex and separable approximations proposed by Svanberg [19]. In addition, since the approximations are diagonal quadratic, their conservatism may be manipulated in a natural way: increasing the approximate diagonal curvatures $c_{2_i}^{\{k\}}$ increases the conservatism of the approximations, and *vice versa*. For details, the reader is referred to Reference [20].

Th process described above is selected by specifying `force_converge` = `1` in `Initialize.f`. (Note however that the default is `force_converge` = `0`, i.e. convergence is not enforced. Instead, the algorithm then relies on the inherent curvatures of the approximations used, to *hopefully* converge.)

Note however that while global convergence is a highly desirable characteristic, it may sometimes imply a considerable increase in computational effort [20, 21].

5.3.2 Filter-based trust region

As an alternative to conservatism, a trust region method based on the filtered acceptance of iterates, inspired by the work of Fletcher, Leyffer and their co-workers [22, 23, 24, 25, 26] is also available. (The filter was originally developed to replace the (problematic) merit function in SLP and SQP algorithms.) The trust region method is selected by selecting `force_converge` = `2` in `Initialize.f`.

5.3.3 Filtered conservatism

Finally, it is also possible to use *filtered conservatism* to enforce convergence and termination. Filtered conservatism aims to combine the salient features of trust region strategies with nonlinear acceptance filters on the one hand, and conservatism on the other. In filtered conservatism methods, the nonlinear acceptance filter is used to decide if an iterate is accepted or rejected. However, if an iterate is rejected, the trust region need *not* be decreased; it may be kept constant. Convergence is then effected by increasing the conservatism of only the unconservative approximations in the (large, constant) trust region, until the iterate becomes acceptable to the filter. Filtered conservatism is automated by selecting `force_converge` = `3` in `Initialize.f`.

In general, it seems that filtered conservatism[12] is the superior strategy [21], i.e. to use `force_converge` = `3` in `Initialize.f`. However, additional validation is required (again, feedback will sincerely be appreciated).

[12]In Section 5.2.2, we elaborate on the fact that the subproblems in the SAO*i* algorithm may be solved using either a dual statement or a QP form.

5.3.4 Outstanding

The routines mentioned in the foregoing have all been coded rather hurriedly, and are in need of revision. In addition, we hope to add (non-monotonic) trust region methods based on classical merit function approaches, etc.

5.4 Infeasible subproblems

5.4.1 The bounded dual

In SAO, it is customary to relax the subproblems, which ensures that an artificial feasible solution exists for each and every subproblem. In the current version of the SAO*i* algorithm, we have not implemented this as the default. Instead, we have opted for the so-called 'bounded dual method, which forces the iterates to feasibility, which we have introduced in Reference [27]. The formulation is extremely simple; we replace approximate dual subproblem $P_D[k]$ in (3) by

$$\begin{aligned} &\max_{\boldsymbol{\lambda}}\{\gamma(\boldsymbol{\lambda}) = \tilde{f}_0^{\{k\}}(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{f}_j^{\{k\}}(\boldsymbol{x}(\boldsymbol{\lambda}))\}, \\ &\text{subject to } 0 \le \lambda_j \le \hat{\lambda}, \qquad j = 1, 2, \ldots, m, \end{aligned} \tag{9}$$

with $\hat{\lambda}$ suitably large and prescribed. Literally, we bound the dual variables $\boldsymbol{\lambda}$ from above. Proof of convergence of the bounded dual method is discussed in References [27, 28], and the method has, among others, been used in Reference [29].

5.4.2 Relaxation

It is possible to use relaxation in the SAO*i* algorithm by specifying a nonzero value for the linear penalty `pen1` in the file `Initialize.f`. Typical values are 10^3, or larger. It is also possible to specify the quadratic penalty `pen2` (θ_2), and an upper bound `ymax` ($\hat{y}$) for the relaxation variables y_j. For the latter two, the defaults should normally suffice.

When relaxation is enforced, we obtain a new inequality-constrained optimization problem $\mathrm{P_{NLP}}$ of the form

$$\begin{aligned} \min\ & f_0(\boldsymbol{x}) + \sum_{j=1}^{n_i}\left(\theta_1 y_j + \frac{1}{2}\theta_2 y_j^2\right) \\ \text{subject to}\ & f_j(\boldsymbol{x}) - y_j \le 0, && j = 1, 2, \cdots, n_i, \\ & \check{x}_i \le x_i \le \hat{x}_i, && i = 1, 2, \cdots, n, \\ & 0 \le y_j \le \hat{y}, && j = 1, 2, \cdots, n_i, \end{aligned} \tag{10}$$

with θ_1 a large 'linear penalty', and $\theta_2 > 0$ a 'quadratic penalty' (θ_2 convexifies the subproblems), while $\hat{y}$ represents an upper bound on the relaxation variables y_j. If a feasible solution to problem $\mathrm{P_{NLP}}$ exists, the algorithm will terminate with $y_j = 0\ \forall\ j$, if θ_1 and $\hat{y}$ are chosen 'sufficiently large'. For details, the reader is referred to the literature by Svanberg on MMA [12, 19, 6].

Currently, relaxation has only been implemented for the dual solvers. (The `NAG` QP solver `e04nqf` is able to handle infeasible problems internally, albeit that this requires changing some settings via calls to the `NAG` routines `e04ntf` and/or `e04nuf`. For details, see the `NAG` documentation.)

Finally: in (10) above, n_i relaxation variables y_j are indicated. Since Version 0.4.4, only a single relaxation variable y is present. This seems to make little difference to the efficiency of relaxation, but has obvious advantages when many constraints, possibly hundreds of thousands, are present.

In dual statements, conservative convex separable approximations (CCSA's) are often used. In quadratic programming, conservatism has for very obvious reasons not been used - while convex QP subproblems are desirable, they are normally not separable. Instead, termination of a sequence of QP subproblems is normally enforced using a trust region approach (as is indeed available in the SAO*i* algorithm). Nevertheless, we have allowed for termination of a QP sequence using conservatism (and filtered-conservatism) in the SAO*i* algorithm; we hope to elaborate on this shortly, but it is clear that this requires *a posteriori* evaluation of the nonlinear objective and constraint functions, augmented by increasing the diagonal curvatures of these unconservative approximations in the approximate (diagonal) Hessian matrix.

5.5 Brief notes on SIMP-like topology optimization problems

In this section, we briefly reflect on topology optimization using SIMP-like relaxation methods. The SIMP or 'solid isotropic material with penalization' method was proposed by Bendsøe [30] and Rozvany and Zhou [31]. The SIMP method in all probability is the most popular method for generating near-discrete optimal topologies. However, the comments made in this section also apply for volumetric penalization methods, like the so-called SINH method (pronounced 'cinch'), proposed by Bruns [32] (and others).

5.5.1 Continuation strategies

In SIMP-like methods, continuation strategies are often used to generate 'good' near-discrete designs. In essence, this entails (slowly) ramping some penalty parameter p from unity to some larger value, say 3, whereafter the penalty parameter p is held constant.

Continuation strategies however complicate the application of a convergence strategy (since convergence can only be enforced once the relaxed problem has become 'stable').

The enforcement of both convergence and a continuation strategy may be implemented using for example the following code fragment in `functions.f` (where the penalty p is denoted `penal`):

```
      if (outerloop.le.20) then
        penal = 1.d0
      else
        penal = penal * 1.02d0
      endif
      penal = min(penal, 3.d0)
!
      force_converge = 0
      if (penal .ge. 3.d0) force_converge = 1
      call reset_conv()
```

Here, `outerloop` represents the outer iteration number - it may be passed via the user array `iuser`. Note that the call to `reset_conv()` is compulsory. `force_converge=2` or `force_converge=3` are also possible. In addition, instead of:

```
      if (penal.ge.3.d0) force_converge = 1
```

one could for example specify:

```
      if (outerloop.ge.85) force_converge = 1
```

etc.

5.5.2 Mesh independence filtering

Note that some implementations of mesh independence filtering may result in solutions that do not satisfy the KKT conditions. When accepting or rejecting these solutions, it should be borne in mind that the actual topology problem is discrete (Boolean) and intractable.

5.6 Description of parameters, keywords and identifiers

Some of the parameters, keywords and identifiers available in the current version of the SAO*i* algorithm are tabulated in Tables 1 through 4 in Appendix D.1. In the tables, red entries indicate options not available in the current version, but planned for future versions. (Many are in fact complete, but have not been tested and/or reported on.) *Blue italicized entries* indicate default values and settings.

6 Storage schemes

The gradients of the constraints may be stored in dense or sparse form. However, not all of the features available in the dense implementation are available in the sparse form, but we are continuously improving this.

In the current version, the main limitations are as follows:

- For the objective function f_0, only `approx_f = 4` is currently supported.
- For the constraint functions f_j, only `approx_c = 4` is currently supported.
- Only the unconditional acceptance of iterates is supported.

An example is included in subdirectory `Examples/veryLarge/vdPlaats1/`.

6.1 Dense scheme

The dense storage structure is invoked by the command `structure=1` in `Initialize.f`. (It is actually the default, so it need not be specified.) In the dense implementation, `nnz`, being the number of non-zero entries in `gc`, is equal to `n*ni` upon entry in `Gradients.f`, and its value should never be changed. The arrays `Acol` and `Aptr` are not used if the dense storage structure is used. What is more, their dimensions are unity.

6.2 Pseudo-sparse scheme

The pseudo-sparse storage structure is invoked by the command `structure=2` in `Initialize.f`. From the users point of view, the pseudo-sparse scheme is identical to the dense scheme, and the gradients are specified in exactly the same manner as for the dense scheme. The only difference is that the algorithm internally feeds a sparse structure to the subproblem solver. So, there is no memory advantage, but potentially a huge reduction in CPU effort (and wall clock time) if the Jacobian is indeed sparse. *A note to developers: the pseudo-sparse storage structure uses the coordinate (COO) storage scheme, and not the compressed sparse row (CSR) storage scheme.*

6.3 Sparse scheme with `nnz` constant and known

If the sparse storage structure is used, this is to be invoked by the command `structure=3` in `Initialize.f`. The default storage scheme in the sparse implementation is the popular and well-known compressed sparse row (CSR) storage scheme.

If the sparse CSR storage scheme is invoked, it is required to specify `nnz` in `Initialize.f`, being the number of non-zero entries in `gc`. In addition, in `Gradients.f`, it is required to specify the integer arrays `Acol` and `Aptr`. Their dimensions are `Acol(nnz)` and `Aptr(ni+1)` respectively. `Aptr(j+1)-Aptr(j)` stores the number of non-zero entries in row `j` of `gc`. `Acol(k)` stores the actual column number of entry `k`. Note that `Aptr(1)=1`, while `Aptr(ni+1)=nnz+1`.

Note that the number of non-zero terms `nnz` has to be *a priori* know, and specified in `Initialize.f`. If `nnz` varies as the iterations progress, `structure=4` should be used, see below.

If the user has data stored in another storage scheme than the default CSR scheme, e.g. the coordinate format (COO) or Ellpack format (ELL), the data should be transformed to the CSR format, but this should be easy. We hope to provide for many different popular storage schemes in future versions. In the mean time, the knowledgeable user may take a look at the file `formats.f` (hacked from the open source SPARSEKIT project by Youcef Saad [33]).

6.4 Sparse scheme with `nnz` variable and possibly unknown

If the number of non-zero terms `nnz` varies as the iterations progress, `structure=4` should be used. In addition, it is required to set `nnz` to the largest number of entries expected. Be sure to select `nnz` adequately large, and to *construct tests* in the user routine `Gradients.f` to ensure that this value is not exceeded, else the results returned by the program may be totally erroneous (if the program does not lead to memory violations in the first place).

7 On efficiency

7.1 Scaling

Papalambros and Wilde [34] (and many others) famously remark that scaling is the single most important, but simplest, reason that can make the difference between success and failure of an optimization algorithm.

The user should therefore do his or her utmost to ensure that not only the design variables are reasonably well scaled, but also the objective function f_0 and the constraint functions f_j. *However, remember that dual algorithms do exactly what the name implies: they solve a dual subproblem, and it is important to scale the constraint functions f_j such that the dual variables λ_j are roughly in the same order of magnitude.* This may sometimes require some numerical experimentation (using small scale versions of the problem being minimized, etc.)

7.2 The approximate higher order curvatures

A further obvious consideration is to select the approximate higher order curvatures $c_{2_i}^{\{k\}}$ in both the approximate objective function and the approximate constraint functions such that they are as representative as possible of the real problem (see Section 5.1). (Note that it is possible to specify different strategies for different constraints, via the 'user' option, see Section 5.1.2.)

7.2.1 NOTE

One other aspect that may be deserved of attention in the context of efficiency, are the tolerances `atol1` and `btol1`, see Section D.1. Very small values for `atol1` and `btol1` may occasionally render the dual subproblems difficult to solve[13]. Values smaller than the default values should be selected when the objective or the constraint functions are linear. *However, only one of* `atol1` *and* `btol1` *may be set equal to zero, to ensure that the primal subproblems are strictly convex, except if problem specific knowledge allows for this. An example of this is when the quadratic approximation to the reciprocal approximation is used for the objective and all the constraint functions. Setting both* `atol1` *and* `btol1` *is probably desired under these conditions, in particular when very large scale optimization (VLSO) is done.*

7.3 Failure of the algorithm, and robustness

No algorithm will never fail. The SAO*i* algorithm relies on the accurate maximization of a sequence of dual subproblems. For this, the limited memory bound constrained l-BFGS-b solver [16, 17] developed by Zhu and co-workers [16, 17] is used in the current academic version of the algorithm, as already mentioned in the foregoing[14]. The l-BFGS-b solver is robust indeed, but it does occasionally fail if the dual subproblems are particularly 'nasty'.

We are working on the development of additional solvers to fall back on when the l-BFGS-b solver fails. Alternatively, primal rather than dual subproblems may be formulated, which may then be solved using any efficient QP solver available to the user, e.g. see [7]. (An interface for the (commercial) `NAG` QP solver `e04nqf` is available in the SAO*i* algorithm.)

Note that the QP approach may be preferable to start of with if $m \gg n$ [7], and in particular when $m^* \gg n$, where m^* indicates the number of active constraints at optimality.

[13]Note that the dual is discontinuous in its second derivatives to start with, even if the real objective and constraint functions are infinitely smooth.

[14]The dual subproblems are concave, but unfortunately, not strictly so. In fact, the dual subproblems are discontinuous in the second derivatives. This should in effect disqualify second order methods (if additional mechanism are not introduced to accommodate the discontinuities). However, the l-BFGS-b solver we use seems to be very effective in maximizing the dual. Vaguely speaking, the fact that only a few memory corrections are used (the default is at little as 8) seems to 'smooth' out the effects of the discontinuities. In our experience, the l-BFGS-b solver requires notably less computational effort to conjugate-gradient methods.

8 Current activities and future versions

In future versions of the SAO*i* algorithm, we hope to extend the functionality. We aim to start with the red entries in Tables 1 and 3 in Appendix D.1.

Other current activities of note include the following:

1. The addition of alternative strategies to those mentioned herein to estimate suitable approximate Hessian terms.
2. The inclusion of a suitable conjugate gradient (CG) solver for (automatic) use as a back-up solver when the l-BFGS-b solver fails.
3. The inclusion of suitable additional quadratic programming (QP) solvers for (automated) use as a back-up solver for when the l-BFGS-b solver fails, but also as *primary solvers* when $m \gg n$.
4. Recoding the convergence mechanisms.
5. Quite a few more.

9 User-contributions

Some subroutines have actually been coded very hurriedly; improvements are most welcome, and will be included in future releases of the algorithm, obviously with due recognition to the contributor.

However, since we are not likely to remove the dependency on `GALAHAD` in future versions, the SAO*i* algorithm is not open-source in the true sense of the word, although it is our intention that it will remain freely available to academic, non-commercial use. Contributors should be aware of this, and realize that their contributions will be subject to the same restrictions.

10 Availability and conditions of use

10.1 Restrictions

1. The academic version of the SAO*i* algorithm is available free of charge upon request from the first author: `albertg@sun.ac.za`.

   Important notice: We have included third party routines in the academic version of our algorithm, which are also subject to restrictions, e.g. the limited memory bound constrained l-BFGS-b solver [16, 17].
2. The SAO*i* algorithm may not be distributed to third parties; all users are required to request their own copy of the program, to help us manage the statistics of the algorithms use. (We hope to make the algorithm available on a web server shortly.)
3. The academic version of the SAO*i* algorithm is made available without any guarantees whatsoever, and it is used at the users' own risk. Legally speaking, there is not even an implied fitness for any particular purpose.
4. Users should cite Reference [20] to refer to the SAO*i* algorithm, and mention that the diagonal approximate Hessian terms are constructed as described in Reference [2]. In addition, users should mention that the sub-problems are solved using the the limited memory bound constrained l-BFGS-b solver [17] (if this is indeed the case). Ditto for the `GALAHAD` and `NAG` solvers, of course.
5. Currently, the source code is only available in FORTRAN.

11 General

11.1 Bugreports, wishlists and support

Please send comments, bugreports and wishlists to the first author: `albertg@sun.ac.za` – doubtless, many an error and bug are still present in the code. The files `Known.Bug.List` and `Wish.List` in subdirectory `Documentation/` summarize the current state, but not even these files are anything close to being complete.

We will gladly offer limited assistance and support to users of the algorithm. However, since the algorithm is made available free of charge to academics and we have full-time obligations other than the development and maintenance of the code, we cannot promise speedy replies to requests for help.

What is more, we will *not* be able to offer advise regarding installation and/or use on platforms other than Linux/UNIX (although we expect that installation on Mac machines should not be too difficult, since these installations are based on Free BSD).

11.2 Code efficiency

In versions prior to version 0.1.5, we have not tried to implement the SAO*i* code in an optimal way. Instead, we focused on implementing the code with clarity in mind, since the code is (also) intended for use in senior undergraduate and postgraduate courses. However, as of Version 0.1.5, some subroutines have been rewritten to exploit the Basic Linear Algebra Subprograms (BLAS), see `http://www.netlib.org/blas/`.

11.3 Tuned BLAS

Users are encouraged to use tuned versions of the BLAS (ATLAS-BLAS) on permanent installations.

11.4 Previous versions and published results

The results obtained with the current version of the SAO*i* algorithm may depend on the compiler and computing platform used. These may in turn also differ from results we have previously published for the SAO*i* algorithm. In part, this is due to differences in implementation, since we continuously try to improve the robustness of the SAO*i* algorithm (to say nothing about removing the inevitable bugs and errors).

11.5 Very brief historical perspective

The SAO*i* algorithm builds on the contributions of many. Firstly, many ideas were borrowed from the CONLIN algorithm initially developed by Fleury and Braibant [13], in turn based on the conservative approximation of Starnes and Haftka [35], as well as the generalization of CONLIN, namely the method of moving asymptotes (MMA) developed by Svanberg [12, 6]. The exponential approximation proposed by Fadel *et al.* [36] also features, albeit in quadratic form. Originally, *spherical* diagonal quadratic approximations were proposed by Snyman and Stander [9]. Conservatism was proposed by Svanberg [19]. Trust region methods are due to the contributions of many in mathematical programming, e.g. see Conn *et al.* [37]. Although uncommon in SAO, we heavily rely on the salient features of a quadratic program (QP) in our algorithm. The filtered acceptance of iterates was proposed by Fletcher, Leyffer and their co-workers [22, 23, 24, 25, 26]. The original form of the dual used is due to Falk [14]. A convex form of the quadratic approximation to the reciprocal approximation was previously used as a heuristic by Duysinx *et al.* [38].

11.6 Trivia

In the acronym 'SAO*i*', the *i* derives from the function `eye`, available in some mathematical manipulation programs. In turn, `eye` derives from the capital letter *I*, often used to indicate that a matrix is diagonal (or 'separable'). The single italicized lower case letter *i* in 'SAO*i*' was simply selected for easy readability; in this form, it also stands for *intermediate* or *intervening* variable, since the approximate curvatures used often derive from linear approximations, written in terms of some suitable intermediate or intervening variable.

References

[1] R.T. Haftka and Z. Gürdal. *Elements of structural optimization*, volume 11 of *Solid Mechanics and its applications*. Kluwer Academic Publishers, Dordrecht, the Netherlands, third edition, 1991.

[2] A.A. Groenwold, L.F.P. Etman, and D.W. Wood. Approximated approximations for SAO. *Struct. Mult. Optim.*, 41:39–56, 2010.

[3] A.A. Groenwold and L.F.P. Etman. On the supremacy of reciprocal-like approximations in SAO - a case for quadratic approximations. In *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*, Lisboa, Portugal, June 2009. Paper 1062.

[4] J.A. Snyman and L.P. Fatti. A multi-start global minimization algorithm with dynamic search trajectories. *J. Optim. Theory Appl.*, 54:121–141, 1987.

[5] R. Zielinsky. A statistical estimate of the structure of multiextremal problems. *Math. Program.*, 21:348–356, 1981.

[6] K. Svanberg. A globally convergent version of MMA without linesearch. In G.I.N. Rozvany and N. Olhoff, editors, *Proc. First World Congress on Structural and Multidisciplinary Optimization*, pages 9–16, Goslar, Germany, 1995.

[7] L.F.P. Etman, A.A. Groenwold, and J.E. Rooda. On diagonal QP subproblems for sequential approximate optimization. In *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*, Lisboa, Portugal, June 2009. Paper 1065.

[8] A.A. Groenwold, L.F.P. Etman, J.A. Snyman, and J.E. Rooda. Incomplete series expansion for function approximation. *Struct. Multidisc. Opt.*, 34:21–40, 2007.

[9] J.A. Snyman and N. Stander. New successive approximation method for optimum structural design. *AIAA J.*, 32:1310–1315, 1994.

[10] D.N. Wilke, S. Kok, and A.A. Groenwold. The application of gradient-only optimization methods for problems discretized using non-constant methods. *Struct. Mult. Optim.*, 40:433–451, 2010.

[11] A.A. Groenwold and L.F.P. Etman. A quadratic approximation for structural topology optimization. *Int. J. Numer. Meth. Eng.*, 82:505–524, 2010.

[12] K. Svanberg. The method of moving asymptotes - a new method for structural optimization. *Int. J. Numer. Meth. Eng.*, 24:359–373, 1987.

[13] C. Fleury and V. Braibant. Structural optimization: a new dual method using mixed variables. *Int. J. Numer. Meth. Eng.*, 23:409–428, 1986.

[14] J.E. Falk. Lagrange multipliers and nonlinear programming. *J. Math. Anal. Appls.*, 19:141–159, 1967.

[15] A.A. Groenwold and L.F.P. Etman. Sequential approximate optimization using dual subproblems based on incomplete series expansions. *Struct. Multidisc. Opt.*, 36:547–570, 2008.

[16] C. Zhu, R.H. Byrd, P. Lu, and J. Nocedal. L-BFGS-B: FORTRAN subroutines for large scale bound constrained optimization. Technical Report NAM-11, Northwestern University, EECS Department, 1994.

[17] R.H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Scient. Comput.*, 16:1190–1208, 1995.

[18] N.I.M. Gould, D. Orban, and Ph.L. Toint. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Trans. Math. Software*, 29:353–372, 2004.

[19] K. Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM J. Optim.*, 12:555–573, 2002.

[20] A.A. Groenwold, D.W. Wood, L.F.P. Etman, and S. Tosserams. Globally convergent optimization algorithm using conservative convex separable diagonal quadratic approximations. *AIAA J.*, 47:2649–2657, 2009.

[21] A.A. Groenwold and L.F.P. Etman. Globally convergent SAO algorithms for large scale simulation-based optimization. In *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*, Lisboa, Portugal, June 2009. Paper 1055.

[22] R. Fletcher and S. Leyffer. Nonlinear programming without a penalty function. *Math. Program.*, 91:239–269, 2002.

[23] R. Fletcher, S. Leyffer, and P.L. Toint. On the global convergence of an SLP-filter algorithm. Technical Report 00/15, Department of Mathematics, University of Namur, Namur, Belgium, 1998.

[24] R. Fletcher, S. Leyffer, and P.L. Toint. On the global convergence of a filter-SQP algorithm. *SIAM J. Optim.*, 13:44–59, 2002.

[25] R. Fletcher, N.I.M. Gould, S. Leyffer, P.L. Toint, and A. Wächter. Global convergence of a trust-region SQP-filter algorithm for general nonlinear programming. *SIAM J. Optim.*, 13:635–659, 2002.

[26] R. Fletcher and S. Leyffer. User manual for filterSQP. Numerical Analysis Report NA\181, Department of Mathematics, University of Dundee, Dundee, Schotland, April 1998.

[27] D.W. Wood, A.A. Groenwold, and L.F.P. Etman. On bounding the dual of Falk to circumvent the requirement of relaxation in globally convergent SAO algorithms. Draft Technical Computing Report SORG/DTCR/04/11, Department of Mechanical and Mechatronic Engineering, University of Stellenbosch, Stellenbosch, South Africa, 2011.

[28] D.W. Wood. *Dual Sequential Approximation Methods in Structural Optimisation*. PhD dissertation (submitted), University of Stellenbosch, Stellenbosch, South Africa, Department of Mechanical and Mechatronic Engineering, 2012.

[29] A.A. Groenwold and L.F.P. Etman. On the conditional acceptance of iterates in SAO algorithms based on convex separable approximations. *Struct. Mult. Optim.*, 42:165–178, 2010.

[30] M.P. Bendsøe. Optimal shape design as a material distribution problem. *Struct. Optim.*, 1:193–202, 1989.

[31] G.I.N. Rozvany and M. Zhou. Applications of COC method in layout optimization. In H. Eschenauer, C. Mattheck, and N. Olhoff, editors, *Proc. Engineering Optimization in Design Processes*, pages 59–70, Berlin, 1991. Springer-Verlag.

[32] T.E. Bruns. A reevaluation of the SIMP method with filtering and an alternative formulation for solid-void topology optimization. *Struct. Multidisc. Optim.*, 30:428–436, 2005.

[33] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Technical report, University of Minnesota, Computer Science Department, June 1994.

[34] P. Y. Papalambros and D. J. Wilde. *Principles of Optimal Design: Modeling and Computation*. Cambridge University Press, New York, NY, USA, second edition, 2000.

[35] J.H. Starnes Jr. and R.T. Haftka. Preliminary design of composite wings for buckling, stress and displacement constraints. *J. Aircraft*, 16:564–570, 1979.

[36] G.M. Fadel, M.F. Riley, and J.M. Barthelemy. Two point exponential approximation method for structural optimization. *Struct. Optim.*, 2:117–124, 1990.

[37] A.R. Conn, N.I.M. Gould, and P.L. Toint. *Trust-region methods*. MPS/SIAM Series on Optimization. SIAM, Philadelphia, 2000.

[38] P. Duysinx, W.H. Zhang, C. Fleury, V.H. Nguyen, and S. Haubruge. A new separable approximation scheme for topological problems and optimization problems characterized by a large number of design variables. In N. Olhoff and G.I.N. Rozvany, editors, *Proc. First World Congress on Structural and Multidisciplinary Optimization*, pages 1–8, Goslar, Germany, 1995.

A Installation

A.1 Supported platforms and FORTRAN compilers

The SAO*i* algorithm is written in FORTRAN-77, 90 and 95. However for the sake of interfacing with external solvers, the installation also assumes that a C-compiler is available (typically, the *GNU gcc* compiler).

A.1.1 Tested compilers

Current versions (later than Version 0.7.0) are aimed at either of the following compilers, which are freely available under Linux:

1. the new Gnu[15] FORTRAN-95 compilers *gfortran-4.3* through *gfortran-4.5*,
2. the independently developed[16] FORTRAN-95 compiler *g95*,

In previous versions, we have also tested the SAO*i* algorithm on Linux platforms using the following compilers:

1. the old Gnu[17] FORTRAN-77 compiler *g77*,
2. the Sun Studio[18] *f95* compiler.

Under MS windows, we have tested previous versions of the SAO*i* algorithm using the following compilers only:

1. the Gnu[19] FORTRAN-95 compiler *gfortran* for windows.

However, the code is FORTRAN-90 compliant, and is expected to port with few problems to other platforms.

A.1.2 Discontinued compilers

Since we have added an interface to the `GALAHAD` QP solvers, it has become very difficult to continue support for the old Gnu-77 compiler *g77*, and we have discontinued this. Knowledgeable users may however do this; limited help is available in `/Utils/f77`. (We have also used the `es` descriptor in some print statements...)

We have also tested earlier versions of the code using the Intel FORTRAN compiler *ifort*, but we have discontinued this as well, since this compiler is not freely available to academics (although it is freely available for non-commercial use). Sadly enough, Intel is not quite as appreciative of academics as the likes of IBM and AMD.

A.1.3 Compiler flags

In the original distribution, only a single compiler flag was set, namely `-O3` (the maximum level of optimization for most compilers). If the `gfortran` compiler is used, it is recommended that the `-march=native` flag is also set[20]. (We here assume that the executable will be rebuilt every time it is installed on a different machine.) For other compilers, some suggestions may be found in the Makefile.

However, note that the `-O3` flag may sometimes result in lack of repeatability (although we have never found this to be detrimental to the extent that the algorithm fails). The flag `-O2` seems to be a conservative and safe alternative, and beneficial to repeatability.

[15] Freely available under the Gnu public license.
[16] Freely available under the Gnu public license.
[17] Freely available under the Gnu public license, but development has been discontinued.
[18] Commercial.
[19] Freely available under the Gnu public license.
[20] In some versions of the distribution, this may be the default.

A.1.4 The timer command 'etime' in function seconds()

It is possible that the timer routine `seconds()` will not work under all operating systems. This may be rectified by commenting out the line

```
seconds=dble(etime(dum))
```

in function `seconds()`, albeit that timekeeping will then not work. Alternatively, the user may replace the line above with a suitable call for his or her platform.

(The double precision function `seconds()` is coded in the file `timer.f`.)

A.1.5 The command 'isNaN()' in subroutine testNaN()

Older compilers may not support the `isNaN()` command; if this is indeed the case, simply comment out the relevant lines in file `testNaN.f`.

A.2 Installation and 'make'

For Unix/Linux, a make utility is available.

Installation on Unix/Linux platforms[21] is as follows:

1. Download the file `SAOi.tar.gz` into a suitable directory.
2. In this directory, unpack this file (using `tar -xvzf SAOi.tar.gz`).
3. Edit the file `Makefile` and uncomment the desired compiler, e.g. change:

   ```
   # F77=gfortran-4.1
   ```

   into:

   ```
     F77=gfortran-4.1
   ```

   to select the *gfortran-4.1* compiler. (The Linux make utility interprets the #-character as the start of a comment.)
4. Type 'make'.
5. Type 'SAO*i*'; this will run the example problem in the current distribution of the algorithm (see Appendix B of this manual).

Finally: `make clean` will delete all temporary runfiles and object files, while `make proper` will also delete module files, and the executable.

A.3 Support for `GALAHAD`

If a valid instance of `GALAHAD` is available, an interface to the `GALAHAD` QP solvers may be created by editing the files `Makefile.galahad32` and/or `Makefile.galahad64` in subdirectory `Utils/galahad/`.

Thereafter, on a 32 bit machine, simply issue the command:

```
> make galahad32
```

On a 64 bit machine, issue the command:

```
> make galahad64
```

This needs to be done only once; thereafter, `make` suffices.

For Linux machines,

[21]For installation on windows platforms, the Makefile will require a little editing...

```
> make galahad
```

suffices; the `uname-m` command is used to detect the machine precision.

A.4 Support for `NAG`

If valid `NAG` libraries are available, the `NAG` QP solvers may be called by editing the file `Makefile.nag` in subdirectory `Utils/nag/`.

Thereafter, issue the command:

```
> make nag
```

Again, this needs to be done only once; thereafter, `make` suffices.

A.5 Note

Note that the instructions used for the automated testing of the SAO*i* algorithm – if at all done by the user – may write instructions to the file `enforce.f`, which will overwrite settings specified in `Initialize.f`.

B A unimodal example: simultaneous shape and sizing design of a 2-bar truss

B.1 Example problem formulation

This is a simple problem proposed by Svanberg [6]. It is an interesting problem in that shape and sizing design are performed simultaneously. The problem may analytically be expressed as

$$\begin{aligned}
&\min_{\boldsymbol{x}} f_0(\boldsymbol{x}) = c_1 x_1 \sqrt{1+x_2^2},\\
\text{subject to } &f_1(\boldsymbol{x}) = c_2\sqrt{1+x_2^2}\left(\frac{8}{x_1}+\frac{1}{x_1x_2}\right) - 1 \le 0,\\
&f_2(\boldsymbol{x}) = c_2\sqrt{1+x_2^2}\left(\frac{8}{x_1}-\frac{1}{x_1x_2}\right) - 1 \le 0,\\
&0.2 \le x_1 \le 4.0,\\
&0.1 \le x_2 \le 1.6,
\end{aligned} \tag{11}$$

with $c_1 = 1.0$ and $c_2 = 0.124$.

B.2 Running the code without coding gradients

We start by running the algorithm without specifying the gradients, for which case the ‘user routines’ follow.

B.2.1 User routines

```
      subroutine SAOi_init (n, ni, ne, x, x_lower, x_upper, ictrl,
     &                      lctrl, rctrl, iuser, luser, cuser, ruser,
     &                      nnz, eqn, lin)
!----------------------------------------------------------------------!
!                                                                      !
!  Initialization of the SAOi algorithm; please see the users manual   !
!  for type declarations and comments                                  !
!                                                                      !
!----------------------------------------------------------------------!
      implicit         none
      include          'ctrl.h'
      logical          eqn(*), lin(*)
      integer          i, j, n, ni, ne, nnz
      double precision x(*), x_lower(*), x_upper(*)
      include          'ctrl_get.inc'
!
      n               =  2        !  the number of design variables
      ni              =  2        !  the number of inequality constraints
!
      x(1)            =  1.5d0    !  the starting point
      x(2)            =  0.5d0
!
      x_lower(1)      =  0.2d0    !  the lower bounds
      x_lower(2)      =  0.1d0
!
      x_upper(1)      =  4.0d0    !  the upper bounds
      x_upper(2)      =  1.6d0
```

```
!
      include          'ctrl_set.inc'
      return
      end subroutine SAOi_init



      subroutine SAOi_funcs (n, ni, ne, x, f, c, iuser, luser, cuser,
     &                       ruser, eqn, lin, ictrl, lctrl, rctrl,
     &                       cctrl)
!----------------------------------------------------------------------!
!                                                                      !
!  Compute the objective function f and the inequality constraint      !
!  functions c(j), j=1,ni                                              !
!                                                                      !
!  Please see the users manual for type declarations and comments      !
!                                                                      !
!----------------------------------------------------------------------!
      implicit         none
      include          'ctrl.h'
      logical          eqn(*), lin(*)
      integer          i, j, n, ni, ne
      double precision f, x(n), c(ni)
      double precision temp0, temp1, temp2, temp3
!
! some often occurring terms
      temp0 = 0.124d0
      temp1 = dsqrt(1.d0 + x(2)**2)
      temp2 = 8.d0/x(1) + 1.d0/(x(1)*x(2))
      temp3 = 8.d0/x(1) - 1.d0/(x(1)*x(2))
!
! the objective function
      f    = x(1)*temp1
!
! the first constraint
      c(1) = temp0*temp1*temp2 - 1.d0
!
! the second constraint
      c(2) = temp0*temp1*temp3 - 1.d0
!
      return
      end subroutine SAOi_funcs
```

NOTE: coding subroutine `GRADIENTS` is not required, since `finite_diff=.true.` in `Initialize.f`. (The file however needs to be present.)

B.2.2 Results

Running the code then results in:

```
----------------------------------------------------------------
   Outer Inner   Function val    Max viol       xnorm       Frel
----------------------------------------------------------------
```

```
      0      0  0.1677052D+01 -0.7576D-01
      1      0  0.1405186D+01  0.1931D+00   0.324D+00  0.102D+00
      2      0  0.1434885D+01  0.5148D-01   0.175D+00 -0.123D-01
      3      0  0.1548319D+01 -0.2303D-01   0.955D-01 -0.466D-01
      4      0  0.1505750D+01  0.5561D-02   0.752D-01  0.167D-01
      5      0  0.1508961D+01 -0.1096D-03   0.346D-01 -0.128D-02
      6      0  0.1508674D+01 -0.1157D-04   0.610D-02  0.115D-03
      7      0  0.1508652D+01  0.3478D-06   0.125D-02  0.899D-05
      8      0  0.1508652D+01  0.3301D-06   0.206D-04  0.188D-06
----------------------------------------------------------------
```

B.3 Running the code with gradients

Next, we specify `finite_diff = .false.` in `Initialize.f` (this is actually the default), and we explicitly give the gradients in `gradients.f`.

B.3.1 User routines

Except for specifying `finite_diff = .false.` in `Initialize.f`, `Initialize.f` and `Functions.f` remain unchanged. `Gradients.f` becomes as follows:

```
      subroutine SAOi_grads (n, x, ni, ne, gf, gc, nnz, Acol, Aptr,
     &                       iuser, luser, cuser, ruser, eqn, lin,
     &                       ictrl, lctrl, rctrl, cctrl)
!----------------------------------------------------------------------!
!                                                                      !
!  Compute the gradients gf(i) of the objective function f, and the    !
!  derivatives gc(k) of the inequality constraint functions c          !
!  w.r.t. the variables x(i)                                           !
!                                                                      !
!  Please see the users manual for type declarations and comments      !
!                                                                      !
!  nnz is the number of non-zero entries in gc. If the default dense   !
!      storage structure is used, nnz = n*ni upon entry, its           !
!      value should not be changed                                     !
!                                                                      !
!  jrow and icol are not used if the dense storage structure is        !
!      used. In addition, their dimensions are unity                   !
!                                                                      !
!  iuser, luser, cuser and ruser are user arrays, which may be used    !
!      at will to pass arbitrary data around between the user          !
!      routines                                                        !
!                                                                      !
!----------------------------------------------------------------------!
      implicit         none
      include          'ctrl.h'
      logical          eqn(*), lin(*)
      integer          i, j, n, ni, ne, nnz, jrow(*), icol(*)
      double precision x(n), gf(n), gc(ni,*)
      double precision tmp0, tmp1, tmp2, tmp3, tmp4
!
! some often occurring terms
      tmp0 = 0.124d0
      tmp1 = dsqrt(1.d0 + x(2)**2)
      tmp2 = 8.d0/x(1) + 1.d0/(x(1)*x(2))
      tmp3 = 8.d0/x(1) - 1.d0/(x(1)*x(2))
      tmp4 = 2.d0*x(2)

! derivatives of the objective function
!-----------------------------------------------------------------------!
      gf(1) = tmp1
      gf(2) = x(1)/(2.d0*tmp1)*tmp4

! derivatives of the inequality constraints
!-----------------------------------------------------------------------!
      gc(1,1) = -tmp0*tmp1*(8.d0/x(1)**2 + 1.d0/(x(1)**2*x(2)))
      gc(1,2) = tmp0/(2.d0*tmp1)*tmp4*tmp2 - tmp0*tmp1/(x(1)*x(2)**2)
      gc(2,1) = -tmp0*tmp1*(8.d0/x(1)**2 - 1.d0/(x(1)**2*x(2)))
      gc(2,2) = tmp0/(2.d0*tmp1)*tmp4*tmp3 + tmp0*tmp1/(x(1)*x(2)**2)
!
      return
      end subroutine SAOi_grads
```

Above, `gc(a,b)` is the derivative of constraint `a` with respect to primal variable `b`. Alternatively, `Gradients.f` may be coded as follows:

```
      subroutine SAOi_grads (n, x, ni, ne, gf, gc, nnz, Acol, Aptr,
     &                       iuser, luser, cuser, ruser, eqn, lin,
     &                       ictrl, lctrl, rctrl, cctrl)
!-----------------------------------------------------------------------!
!                                                                       !
!  Compute the gradients gf(i) of the objective function f, and the    !
!  derivatives gc(k) of the inequality constraint functions c           !
!  w.r.t. the variables x(i)                                            !
!                                                                       !
!  Please see the users manual for type declarations and comments      !
!                                                                       !
!  The storage scheme used for gc is column by column. Hence,           !
!      the derivative of constraint j w.r.t. variable i is              !
!      stored in location k = (i-1)*ni + j, where ni is the             !
!      total number of constraints                                      !
!                                                                       !
!  nnz is the number of non-zero entries in gc. If the dense            !
!      storage structure is used, nnz = n*ni upon entry, its            !
!      value should not be changed                                      !
!                                                                       !
!  jrow and icol are not used if the dense storage structure is         !
!      used. In addition, their dimensions are unity                    !
!                                                                       !
!  iuser, luser, cuser and ruser are user arrays, which may be used    !
!      at will to pass arbitrary data around between the user           !
!      routines                                                         !
!                                                                       !
!-----------------------------------------------------------------------!
      implicit          none
      include           'ctrl.h'
      logical           eqn(*), lin(*)
      integer          i, j, n, ni, ne, nnz, jrow(*), icol(*)
      double precision x(n), gf(n), gc(*)
      double precision temp0, temp1, temp2, temp3, temp4
!
! some often occurring terms
      temp0 = 0.124d0
      temp1 = dsqrt(1.d0 + x(2)**2)
      temp2 = 8.d0/x(1) + 1.d0/(x(1)*x(2))
      temp3 = 8.d0/x(1) - 1.d0/(x(1)*x(2))
      temp4 = 2.d0*x(2)

! derivatives of the objective function
!---------------------------------------------------------------------!
      gf(1) = temp1
      gf(2) = x(1)/(2.d0*temp1)*temp4

! derivative of constraint 1 w.r.t variable 1: k = (1-1)*2 + 1 = 1
!---------------------------------------------------------------------!
      gc(1) = -temp0*temp1*(8.d0/x(1)**2 + 1.d0/(x(1)**2*x(2)))

! derivative of constraint 2 w.r.t variable 1: k = (1-1)*2 + 2 = 2
!---------------------------------------------------------------------!
      gc(2) = -temp0*temp1*(8.d0/x(1)**2 - 1.d0/(x(1)**2*x(2)))

! derivative of constraint 1 w.r.t variable 2: k = (2-1)*2 + 1 = 3
!---------------------------------------------------------------------!
      gc(3) = temp0/(2.d0*temp1)*temp4*temp2
     &      - temp0*temp1/(x(1)*x(2)**2)

! derivative of constraint 2 w.r.t variable 2: k = (2-1)*2 + 2 = 4
!---------------------------------------------------------------------!
      gc(4) = temp0/(2.d0*temp1)*temp4*temp3
     &      + temp0*temp1/(x(1)*x(2)**2)
!
      return
      end subroutine SAOi_grads
```

Above, the column-by-column storage scheme of FORTRAN is exploited.

B.3.2 Results

Running the code this time results in increased accuracy w.r.t. the maximum constraint violation, although the required number of function evaluations remains unchanged (for this problem):

```
----------------------------------------------------------------
   Outer Inner   Function val    Max viol       xnorm       Frel
----------------------------------------------------------------
      0      0  0.1677051D+01 -0.7576D-01
      1      0  0.1405187D+01  0.1931D+00   0.324D+00  0.102D+00
      2      0  0.1434886D+01  0.5148D-01   0.175D+00 -0.123D-01
      3      0  0.1548320D+01 -0.2303D-01   0.955D-01 -0.466D-01
      4      0  0.1505750D+01  0.5561D-02   0.752D-01  0.167D-01
      5      0  0.1508961D+01 -0.1100D-03   0.346D-01 -0.128D-02
      6      0  0.1508674D+01 -0.1192D-04   0.610D-02  0.114D-03
      7      0  0.1508652D+01  0.1494D-07   0.125D-02  0.880D-05
      8      0  0.1508652D+01  0.5722D-11   0.204D-04 -0.854D-08
-----------------------------------------------------------------
```

B.4 Running the code with different approximations

Next, we use a superior approximation (for this problem) by adding `approx_f= 5` and `approx_c= 5` to the file `Initialize.f`.

Elaboration: selecting approximation ‘5’ means that we use the diagonal quadratic Taylor series expansion to the exponential approximation. The exponential approximation in turn may be expected to be very accurate for structural problems that exhibit monotonicities. (The reciprocal approximation, which is exact for statically determinate weight minimization subject to stress and displacement constraints only, is a special case of the exponential approximation.)

B.4.1 Results

Running the code this time results in a lower number of function evaluations:

```
-----------------------------------------------------------------
   Outer Inner   Function val    Max viol       xnorm       Frel
-----------------------------------------------------------------
      0      0  0.1677051D+01 -0.7576D-01
      1      0  0.1429265D+01  0.1050D+00   0.274D+00  0.926D-01
      2      0  0.1514687D+01 -0.1516D-02   0.104D+00 -0.352D-01
      3      0  0.1508950D+01 -0.1372D-03   0.326D-01  0.228D-02
      4      0  0.1508647D+01  0.3892D-05   0.565D-02  0.121D-03
      5      0  0.1508652D+01  0.1166D-08   0.197D-03 -0.229D-05
      6      0  0.1508652D+01 -0.2124D-12   0.145D-04 -0.427D-09
-----------------------------------------------------------------
```

B.5 Running the code with guaranteed convergence

Finally, we enforce convergence by adding `force_converge = 1` to the file `Initialize.f`.

B.5.1 Results

Running the code this time results in higher CPU effort (in that inner loops are generated), but the code would converge, irrespective of the quality of the approximations used:

```
-----------------------------------------------------------------
   Outer Inner   Function val    Max viol       xnorm       Frel
-----------------------------------------------------------------
      1      4  0.1522424D+01 -0.3637D-02   0.179D+00  0.578D-01
      2      0  0.1512300D+01 -0.1194D-02   0.318D-01  0.401D-02
      3      0  0.1508624D+01  0.2939D-04   0.246D-01  0.146D-02
      4      1  0.1508659D+01 -0.2282D-05   0.128D-02 -0.141D-04
      5      0  0.1508652D+01  0.3750D-09   0.100D-02  0.269D-05
      6      0  0.1508652D+01  0.6883D-10   0.936D-04  0.111D-07
-----------------------------------------------------------------
```

The results presented were obtained on an IBM Thinkpad Titanium with a 2.00GHz Intel Pentium M processor, running the Linux 2.6.25.16-0.1-default i686 OS (openSUSE 11.0 (i586)), the gfortran-4.2 compiler, and SAO*i* Version 0.1.3 — November 14, 2008.

C A multimodal example: the six-hump camelback problem

C.1 Example problem formulation

Consider the so-called six-hump camelback problem:

$$\begin{aligned} \min_{\boldsymbol{x}} f_0(\boldsymbol{x}) &= (4 - 2.1x_1^2 + \frac{1}{3}x_1^4)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2, \\ \text{subject to } & -3.0 \leq x_1 \leq 3.0, \\ & -2.0 \leq x_2 \leq 2.0. \end{aligned} \tag{12}$$

C.1.1 User routines

We list only the file `Initialize.f`, the files `Functions.f` and `Gradients.f` are include in the distribution, see subdirectory `Examples/multimodal/camel6`.

```
      subroutine SAOi_init (n, ni, ne, x, x_lower, x_upper, ictrl,
     &                      lctrl, rctrl, iuser, luser, cuser, ruser,
     &                      nnz, eqn, lin)
!----------------------------------------------------------------------!
!                                                                      !
!  Initialization of the SAOi algorithm; please see the users manual   !
!  for type declarations and comments                                  !
!                                                                      !
!----------------------------------------------------------------------!
      implicit         none
      include          'ctrl.h'
      logical          eqn(*), lin(*)
      integer          i, j, n, ni, ne, nnz
      double precision x(*), x_lower(*), x_upper(*)
      include          'ctrl_get.inc'
!
      n                = 2          !  the number of design variables
      ni               = 0          !  the number of inequality constraints
!
      do i=1,n
        x(i)           =-2.d0       !  the starting point
      enddo
!
      x_lower(1)       =-3.D0       !  the bounds
      x_upper(1)       = 3.D0
!
      x_lower(2)       =-2.D0
      x_upper(2)       = 2.D0
!
!
!  specify REQUIRED global optimization parameters
!
!
      multimodal       = .true.     !  force a global search
!
!
!  specify a few OPTIONAL global optimization parameters - often desirable
!
```

```
      force_converge   = 1                        !  enforce convergence
      xtol             = 1.d-6                    !  tighten the tolerance
!
      include          'ctrl_set.inc'
      return
      end subroutine SAOi_init
```

Note that the output created by the SAO*i* algorithm differs dramatically when a global search pattern is invoked by setting `multimodal = .true.`.

D Program structure

D.1 Parameters, keywords, variables, identifiers & type declarations

Keyword/identifier	Type	Possible	Notes	Brief comments	
`approx_f`	int	0	(a)	the objective f_0 is approximated using the best possible fit from a library of suitable approximations	
		1		*the objective f_0 is approximated using the 'standard' spherical quadratic approximation*	*{default}*
		2		the objective f_0 is approximated using the 'error-norm' spherical quadratic approximation	
		3		the objective f_0 is approximated using the non-spherical gradient-condition'	
		4	(b)	the objective f_0 is approximated using the second order Taylor series expansion to the reciprocal function at $\boldsymbol{x}^{\{k\}}$	
		5	(b)	the objective f_0 is approximated using the second order Taylor series expansion to the exponential function at $\boldsymbol{x}^{\{k\}}$	
		6	(c)	the objective f_0 is approximated using the second order Taylor series expansion to the MMA approximations	
		≥ 100	(d)	the objective f_0 is approximated using the second order Taylor series expansion to arbitrary 'user approximation(s)'	
`approx_c`	int	0	(a)	the constraints f_j are approximated using the best possible fit from a library of suitable approximations	
		1		*the constraints f_j are approximated using the 'standard' spherical quadratic approximation*	*{default}*
		2		the constraints f_j are approximated using the 'error-norm' spherical quadratic approximation	
		3		the constraints f_j are approximated using the 'non-spherical gradient-condition'	
		4	(b)	the constraints f_j are approximated using the second order Taylor series expansion to the reciprocal function at $\boldsymbol{x}^{\{k\}}$	
		5	(b)	the constraints f_j are approximated using the second order Taylor series expansion to the exponential function at $\boldsymbol{x}^{\{k\}}$	
		6	(c)	the constraints f_j are approximated using the second order Taylor series expansion to the MMA approximations	
		≥ 100	(d)	the constraints f_j are approximated using the second order Taylor series expansion to arbitrary 'user approximation(s)'	
`force_converge`	int	*0*		*convergence is not enforced*	*{default}*
		1		convergence is enforced via conservatism	
		2		convergence is enforced via a trust region strategy with a nonlinear acceptance filter	
		3		convergence is enforced via filtered conservatism	
`finite_diff`	logical	.true.		the gradients are evaluated using finite differences	
		.false.		*the gradients specified in subroutine* `gradients.f` *are used*	*{default}*
`check_grad`	logical	.true.		check the user specified gradients in subroutine `gradients.f`	
		.false.		*do not check the user specified gradients*	*{default}*

Notes:
(a) Not available in current version.
(b) Recommended for problems which exhibit strong monotonicities, and for positive variables $\boldsymbol{x}$.
(c) Recommended for problems which exhibit strong monotonicities, and for variables $\boldsymbol{x}$ of arbitrary sign.
(d) To be coded by the user in `Hessian.f`. NOTE: it is of course possible to specify different approximations for different constraints here.

Table 1: Some of the most important keywords and identifiers available in the current version. Blue italicized entries represent default values

Keyword/identifier	Type	Possibilities	Notes	Brief comments
`subsolver`	int	*1*		*solve the Falk dual using the l-BFGS-b solver [16, 17]* {*default*}
		20		solve the equivalent QP problem using the `NAG` `e04nqf` solver
		25-28		solve the equivalent QP problem using the `GALAHAD` `lsqp`, `qpa`, `qpb` or `qpc` solvers respectively
`iprint`	integer	< 1		write no output to the screen whatsoever
		1		only output the iterative data to the screen
		2		as for `iprint`$=1$, but also write a summary to the screen
		3		*as for* `iprint`*= 2, but also write a time usage summary to the screen* {*default*}
`multimodal`	logical	.true.		multimodal optimization executed in multistart mode
		.false.		*unimodal or convex optimization* {*default*}
`structure`	integer	1		Dense storage scheme {default}
		2		Pseudo-sparse storage scheme
		3		Sparse storage scheme

Table 2: Some of the most important keywords and identifiers available in the current version (continued). Blue italicized entries represent default values

Variable/parameter	Symbol	Type	Conditions	Notes	Brief comments
n	n	int	$n \geq 1$		number of design variables n
ni	n_i	int	$n_i \geq 0$		number of inequality constraints n_i
ne	n_e	int	$n_e \geq 0$		number of equality constraints n_e
nnz	n_z	int	$n_z \geq 0$		the number of non-zero gradients in gc
x	$\boldsymbol{x}$	dble $[n]$	$\check{x}_i \leq x_i \leq \hat{x}_i \; \forall \, i$		variables x_i, $i = 1, 2, \cdots, n$
x_lower	$\check{\boldsymbol{x}}$	dble $[n]$	$\check{x}_i \leq \hat{x}_i \; \forall \, i$		variable lower bounds $\check{x}_i$, $i = 1, 2, \cdots, n$
x_upper	$\hat{\boldsymbol{x}}$	dble $[n]$	$\hat{x}_i \geq \check{x}_i \; \forall \, i$		variable upper bounds $\hat{x}$, $i = 1, 2, \cdots, n$
f	f_0	dble			objective function $f(\boldsymbol{x})$
c	f_j	dble $[n_i]$			equality and inequality constraints $f_j(\boldsymbol{x})$, $j = 1, 2, \cdots n_i + n_n$
gf	$\boldsymbol{\nabla} f_0$	dble $[n]$			partial derivatives of objective function $\partial f_0(\boldsymbol{x})/\partial x_i$, $i = 1, 2, \cdots n$
gc	$\boldsymbol{\nabla} f_j$	dble $[n_i \times n]$		(a)	partial derivatives of the constraints $\partial f_j(\boldsymbol{x})/\partial x_i$, $j = 1, 2, \cdots n_i + n_e$, $i = 1, 2, \cdots n$
jrow		int $[n_z]$	(b)		pointer array for the sparse storage scheme, see Section 6.3
ncol		int $[n_i]$	(b)		pointer array for the sparse storage scheme, see Section 6.3

Notes:

(a) The storage scheme used for `gc` is column by column. Hence, in the dense storage scheme, the derivative of constraint `j` w.r.t. variable `i` is stored in location `k = (i-1)*ni + j`, where `ni` represents the total number of constraints n_i.

(b) Not used in the dense and pseudo-dense storage schemes.

Table 3: Variables and parameters in the current version

Variable/parameter	Symbol	Type	Conditions	Notes	Brief comments	
`dml_infinity`	δ_∞	dble	$0 < \delta_\infty \leq 1$	(a)	normalized global infinity move limit δ_∞	{*default is* 0.2, *i.e.* 20%}
`xtol`	ϵ_x	dble	$\epsilon_x > 0$		convergence tolerance on $\boldsymbol{x}$ (Euclidian norm)	{*default is* 10^{-4}}
`xtol_inf`	ϵ_{x_∞}	dble	$\epsilon_{x_\infty} > 0$		convergence tolerance on $\boldsymbol{x}$ (infinity norm)	{*default is* 10^{-6}}
`kkt_tol`	ϵ_k	dble	$\epsilon_k \geq 0$		convergence tolerance on KKT residual	{*default is* 10^{-6}}
`ftol`	ϵ_f	dble	—	(b)	convergence tolerance on function value	{*default is* -10^{-12}}
`outermax`	$k_{\max}$	int	$k_{\max} \geq 0$		maximum number of outer iterations	{*default is* 500}
`innermax`	$l_{\max}$	int	$l_{\max} \geq 0$		maximum number of inner iterations	{*default is* 50}
`feaslim`	h_{feas}	dble	$h_{\text{feas}} \geq 0$		printing tolerance for feasibility	{*default is* 10^{-4}}
`deltx`	Δ_x	dble	$\Delta_x > 0$		finite difference interval (only used if `finite_diff` = .true.)	{*default is* 10^{-6}}
`biglam`	$\hat{\lambda}$	dble	$\hat{\lambda} > 0$		upper bound on the dual variables	{*default is* 10^{8}}
`ksubmax`	$m_{\max}$	int	$m_{\max} > 0$		maximum number of subproblem evaluations per iteration	{*default is* 10^{5}}
`tlimit`	$t_{\max}$	dble	$t_{\max} > 0$		maximum time allowed for each subproblem (in seconds)	{*default is* 10^{5}}
`pen1`	θ_1	dble	$\theta_1 \geq 0$	(c)	linear relaxation penalty (relaxation is only invoked once $\theta_1 > 0$)	{*default is* 0.0}
`pen2`	θ_2	dble	$\theta_2 > 0$		quadratic relaxation penalty (only used when $\theta_1 > 0$)	{*default is* 1.0}
`ymax`	$\hat{y}_i$	dble	$\hat{y}_i > 0$		upper bound on the relaxation variables (only used when $\theta_1 > 0$)	{*default is* 10^{5}}
`atol1`	$\check{c}_{2_{i_0}}^{\{k\}}$	dble	$\check{c}_{2_{i_0}}^{\{k\}} > 0$	(d)	minimum objective function curvature	{*default is* 10^{-5}}
`btol1`	$\check{c}_{2_{i_j}}^{\{k\}}$	dble	$\check{c}_{2_{i_j}}^{\{k\}} \geq 0$	(d)	minimum constraint function curvatures	{*default is* 10^{-5}}
`filter_hi`	$\hat{h}$	dble			filter upper bound (i.e. the maximum acceptable constraint violation)	{*default is* 1.0}

Notes:

(a) For variable i, the allowable increment Δ_i is then computed as $\Delta_i = \delta_\infty \times (\hat{x}_i - \check{x}_i)$.

(b) Not recommended in general, and is disabled when $\epsilon_f < 0$ (this is actually the default).

(c) Typical values are 10^3 or larger. However, relaxation is in general not recommended.

(d) For some problems, e.g. problems that are not well scaled, the effect of $\check{c}_{2_{i_0}}^{\{k\}}$ and $\check{c}_{2_{i_j}}^{\{k\}}$ may be (very) important. When changing the $\check{c}_{2_{i_\alpha}}^{\{k\}}$, care should always be taken to ensure that the subproblems remain strictly convex.

Table 4: Optional parameters in the current version. Blue italicized entries represent default values. *Run-of-the-mill problems will normally not require changes to many of these values*

E On the diagonal curvatures

The SAO*i* algorithm uses separable *diagonal quadratic* approximation functions $\tilde{f}_j$ to replace the objective function f_0 and all the constraint functions $f_j, j = 1, 2, \cdots, m$. That is, we consider approximations of the form

$$\tilde{f}_j(\boldsymbol{x}) = f_j(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \left(\frac{\partial f_j}{\partial x_i} \right)^{\{k\}} \left(x_i - x_i^{\{k\}} \right) + \frac{1}{2} \sum_{i=1}^{n} c_{2i_j}^{\{k\}} \left(x_i - x_i^{\{k\}} \right)^2, \tag{13}$$

for $j = 0, 1, 2, \cdots, m$ and with the $c_{2i_j}^{\{k\}}$ suitable higher order approximate diagonal Hessian terms or principal curvatures. It is instructive to rewrite (13) for a given j as

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{x}^{\{k\}}) + \boldsymbol{\nabla}^T f(\boldsymbol{x}^{\{k\}})(\boldsymbol{x} - \boldsymbol{x}^{\{k\}}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{\{k\}})^T \boldsymbol{C}^{\{k\}}(\boldsymbol{x} - \boldsymbol{x}^{\{k\}}), \tag{14}$$

being the familiar second order or quadratic Taylor series expansion of f about $\boldsymbol{x}^{\{k\}}$, if $\boldsymbol{C}^{\{k\}}$ was to represent the Hessian of f at $\boldsymbol{x}^{\{k\}}$. However, in our case, we use an approximate diagonal Hessian matrix, viz., we set $C_{ij}^{\{k\}} = 0 \; \forall \; i \neq j$; the premise being that the Hessian terms $C_{ij}^{\{k\}}$ can either not be calculated efficiently, or demand excessive storage requirements.

E.1 Estimating the principal curvatures

Some 'classical' strategies (using direct variables) for estimating the approximate curvatures $c_{2i_j}^{\{k\}}$ may be found in Reference [8]. Using the word 'estimating' may sometimes be a bit presumptuous — the process of selecting the approximate curvatures at times is more reminiscent of the exploitation of known or suspected problem-specific behavior, in the same spirit as the use of (predominantly reciprocal-like) intervening variables in structural optimization [3].

Nevertheless, let us proceed by suggesting a few possibilities which we have found advantageous, while pointing out that many other possibilities exist, including an intermediate response approach. We will introduce the abbreviation T2:Π, meaning that we construct the quadratic or second order Taylor series expansion to approximation Π, with approximation Π some popular (or other) approximation that is able to capture 'important or dominant' behavior. An example of this would be reciprocal-like behavior in structural optimization.

1. *The quadratic approximation to the reciprocal approximation (T2:R).*

   For T2:R, we select
   $$c_{2i_j}^{\{k\}} = \frac{-2}{x_i^{\{k\}}} \left(\frac{\partial f_j}{\partial x_i} \right)^{\{k\}}. \tag{15}$$
   The methodology of development of (15) is summarized in Section E.3. The development for the approximate curvatures given below is similar; for details, the reader is referred to Reference [2].

2. *The quadratic approximation to the exponential approximation (T2:E).*

   For T2:E, we select
   $$c_{2i_j}^{\{k\}} = \frac{a_i^{\{k\}} - 1}{x_i^{\{k\}}} \left(\frac{\partial f_j}{\partial x_i} \right)^{\{k\}}, \tag{16}$$
   where the $a_i^{\{k\}}$ are the well-known exponents in the exponential approximation; for details, again see Reference [2]. (Note that T2:R is obtained by setting $a_i^{\{k\}} = -1 \; \forall \; i, k$.)

3. *The quadratic approximation to the CONLIN approximation (T2:CONLIN).*

   For T2:CONLIN, we select
   $$c_{2i_j}^{\{k\}} = \begin{cases} \dfrac{-2}{x_i^{\{k\}}} \left(\dfrac{\partial f_j}{\partial x_i} \right)^{\{k\}} & \text{if} \quad \left(\dfrac{\partial f_j}{\partial x_i} \right)^{\{k\}} < 0, \\ 0 & \text{if} \quad \left(\dfrac{\partial f_j}{\partial x_i} \right)^{\{k\}} \geq 0. \end{cases} \tag{17}$$
   For details, again see Reference [2].

4. *The quadratic approximation to the MMA approximation (T2:MMA).*

   For T2:MMA, we select

$$c_{2i_j}^{\{k\}} = \begin{cases} \dfrac{-2}{\left(x_i^{\{k\}} - L_i^{\{k\}}\right)} \left(\dfrac{\partial f_j}{\partial x_i}\right)^{\{k\}} & \text{if} \quad \left(\dfrac{\partial f_j}{\partial x_i}\right)^{\{k\}} < 0, \\ \dfrac{2}{\left(U_i^{\{k\}} - x_i^{\{k\}}\right)} \left(\dfrac{\partial f_j}{\partial x_i}\right)^{\{k\}} & \text{if} \quad \left(\dfrac{\partial f_j}{\partial x_i}\right)^{\{k\}} \geq 0, \end{cases} \tag{18}$$

   where the $L_i^{\{k\}}$ and $U_i^{\{k\}}$ respectively are the lower and upper asymptotes, again, details may be found in Reference [2].

5. *Modifications of T2:R and T2:E (respectively denoted T2:R̄ and T2:Ē).*

   Select

$$c_{2i_j}^{\{k\}} = \frac{1 - a_i^{\{k\}}}{x_i^{\{k\}}} \left|\frac{\partial f_j}{\partial x_i}\right|^{\{k\}}. \tag{19}$$

   The resulting approximations T2:R̄ and T2:Ē are inconsistent when $\partial f_j / \partial x_i > 0$ (for the former, $a_i^{\{k\}} = -1 \; \forall \; i, k$ is of course implied) — they are intended for problems that are multimodal and/or strongly monotonically decreasing.

6. *The classical spherical quadratic approximation (SQ1).*

   For SQ1, we select

$$c_{2i_j}^{\{k\}} = \frac{2[f_j(\boldsymbol{x}^{\{k-1\}}) - f_j(\boldsymbol{x}^{\{k\}}) - \boldsymbol{\nabla}^T f_j(\boldsymbol{x}^{\{k\}})(\boldsymbol{x}^{\{k-1\}} - \boldsymbol{x}^{\{k\}})]}{\|\boldsymbol{x}^{\{k-1\}} - \boldsymbol{x}^{\{k\}}\|_2^2}. \tag{20}$$

   For details, see References [8, 9]. (Note that the order above does not directly relate to the curvatures mentioned in Section 5.1 - the list presented above is a bit more general.)

E.2 Convexity

It is noted that the foregoing curvatures will not always result in convex approximations, required in the dual treatment. This requires additional mechanisms, elaborated upon in References [2, 3, 11]. Essentially, this implies constructing forms like (19).

E.3 Some details on the approximations: Deriving T2:R

Substitution of the reciprocal intervening variables $y_i = x_i^{-1}$, $i = 1, 2, \cdots n$, into a linear Taylor series expansion, yields the reciprocal approximation

$$\tilde{f}_{\text{R}} = f(\boldsymbol{x}^{\{k\}}) + \sum_{i=1}^{n} \left(1 - \frac{x_i^{\{k\}}}{x_i}\right) \left(x_i^{\{k\}}\right) \left(\frac{\partial f}{\partial x_i}\right)^{\{k\}}, \tag{21}$$

e.g. see Haftka and Gürdal [1]. The first and second partial derivatives of (21) respectively are

$$\frac{\partial \tilde{f}_{\text{R}}}{\partial x_i} = \frac{\left(x_i^{\{k\}}\right)^2}{x_i^2} \left(\frac{\partial f}{\partial x_i}\right)^{\{k\}}, \tag{22}$$

and

$$\frac{\partial^2 \tilde{f}_{\text{R}}}{\partial x_i^2} = \frac{-2\left(x_i^{\{k\}}\right)^2}{x_i^3} \left(\frac{\partial f}{\partial x_i}\right)^{\{k\}}. \tag{23}$$

In the current point $\boldsymbol{x}^{\{k\}}$, the first and second partial derivatives of (21) respectively reduce to

$$\frac{\partial \tilde{f}_{\mathrm{R}}}{\partial x_i}(\boldsymbol{x}^{\{k\}}) = \left(\frac{\partial f}{\partial x_i}\right)^{\{k\}} , \tag{24}$$

and

$$\frac{\partial^2 \tilde{f}_{\mathrm{R}}}{\partial x_i^2}(\boldsymbol{x}^{\{k\}}) = \frac{-2}{x_i^{\{k\}}} \left(\frac{\partial f}{\partial x_i}\right)^{\{k\}} . \tag{25}$$

Hence, (13) becomes the quadratic or second order Taylor series expansion to the reciprocal approximation $\tilde{f}_{\mathrm{R}}$ (21), if the curvatures $c_{2i_j}^{\{k\}}$ in (13) are given by (25).

The reader is encouraged to add the quadratic approximation to other approximations to the algorithm (and to share these contributions with other users by including them in the distribution).

F Multimodal optimization

Multistart methods require a termination rule for deciding when to end the sampling, and to then use the current overall minimum function value $\tilde{f}$ as the approximation to the global minimum f^*, i.e.

$$\tilde{f} = \min \left\{ \hat{f}^j ,\ j = 1, 2, \cdots \right\}, \tag{26}$$

where j represents the number of starting points to date, and where the $\hat{f}^j$ are assumed to be feasible local minima, $j = 1, 2, \cdots$.

Define the *region of convergence* of a local minimum $\hat{\boldsymbol{x}}$ as the set of all points $\boldsymbol{x}$ which, when used as starting points for a given algorithm, result in converge to $\hat{\boldsymbol{x}}$. Let R_k denote the region of convergence of local minimum $\hat{\boldsymbol{x}}^k$ and let α_k be the associated probability that a sample point be selected in R_k. The region of convergence and the associated probability for the global minimum $\boldsymbol{x}^*$ are denoted by R^* and α^* respectively. The following basic assumption – which is probably true for many combinations of different algorithms and many functions of practical interest – is now made:

$$\alpha^* \geq \alpha_k \text{ for all local minima } \hat{\boldsymbol{x}}^k. \tag{27}$$

The following theorem may then be proved [4]:

Theorem F.1 *Let r be the number of sample points falling within the region of convergence of the current overall minimum $\tilde{f}$ after $\tilde{n}$ points have been sampled. Then, under assumption* (27) *and a statistically non-informative prior distribution, the probability that $\tilde{f}$ corresponds to f^* may be obtained from*

$$\mathcal{P}_r \left[\tilde{f} = f^* \right] \geq q(\tilde{n}, r) = 1 - \frac{(\tilde{n}+1)!(2\tilde{n}-r)!}{(2\tilde{n}+1)!(\tilde{n}-r)!}, \tag{28}$$

where $\mathcal{P}_r$ is short for 'probability that'.

□

On the basis of Theorem F.1, the adopted *stopping rule* becomes

$$\text{STOP when } \mathcal{P}_r \left[\tilde{f} = f^* \right] \geq q^*, \tag{29}$$

where q^* is some prescribed desired confidence level, typically chosen as 0.99 through 0.999.

F.1 Comment

We have herein used (28) in combination with an SAO algorithm, but (28) may of course also be used in conjunction with SQP algorithms, etc. Our preliminary observations however seem to indicate that approximate methods that do not use exact Hessian information seem to perform better, presumably since the inexact Hessian information allows for some 'hill-climbing' ability, or 'smearing-out' of unfit local minima. For unconstrained or bound constrained problems for example, BFGS (or l-BFGS-b), combined with (28), often does very well indeed.

F.2 Proof of the Bayesian stopping criterion

We present here an outline of the proof of (28), and follow closely the presentation by Snyman and Fatti [4]. Given $\tilde{n}^*$ and α^*, the probability that at least one point, $\tilde{n} \geq 1$, has converged to f^* is

$$\mathcal{P}_r[\tilde{n}^* \geq 1 | \tilde{n}, r] = 1 - (1-\alpha^*)^{\tilde{n}} . \tag{30}$$

In the Bayesian approach, we characterize the uncertainty about the value of α^* by specifying a prior probability distribution for it. This distribution is modified using the sample information (namely, $\tilde{n}$ and r) to form a posterior probability distribution. Let $p_*(\alpha^*|\tilde{n}, r)$ be the posterior probability distribution of α^*. Then,

$$\begin{aligned} \mathcal{P}_r[\tilde{n}^* \geq 1 | \tilde{n}, r] &= \int_0^1 \left[1 - (1-\alpha^*)^{\tilde{n}} \right] p_*(\alpha^*|\tilde{n}, r) d\alpha^* \\ &= 1 - \int_0^1 (1-\alpha^*)^{\tilde{n}} p_*(\alpha^*|\tilde{n}, r) d\alpha^*. \end{aligned} \tag{31}$$

Now, although the r sample points converge to the current overall minimum, we do not know whether this minimum corresponds to the global minimum of f^*. Utilizing (27), and noting that $(1-\alpha)^{\tilde{n}}$ is a decreasing function of α, the replacement of α^* in the above integral by α yields

$$\mathcal{P}_r[\tilde{n}^* \geq 1|\tilde{n}, r] \geq \int_0^1 \left[1-(1-\alpha)^{\tilde{n}}\right] p(\alpha|\tilde{n}, r) d\alpha\ . \tag{32}$$

Now, using Bayes theorem, we obtain

$$p(\alpha|\tilde{n}, r) = \frac{p(r|\alpha, \tilde{n})p(\alpha)}{\displaystyle\int_0^1 p(r|\alpha, \tilde{n})p(\alpha)d\alpha}\ . \tag{33}$$

Since the $\tilde{n}$ points are sampled at random and each point has a probability α of converging to the current overall minimum, r has a binomial distribution with parameters α and $\tilde{n}$. Therefore

$$p(r|\alpha, \tilde{n}) = \binom{\tilde{n}}{r}\alpha^r(1-\alpha)^{\tilde{n}-r}\ . \tag{34}$$

Substituting (34) and (33) into (32) gives:

$$\mathcal{P}_r[\tilde{n}^* \geq 1|\tilde{n}, r] \geq 1 - \frac{\displaystyle\int_0^1 \alpha^r(1-\alpha)^{2\tilde{n}-r}p(\alpha)d\alpha}{\displaystyle\int_0^1 \alpha^r(1-\alpha)^{\tilde{n}-r}p(\alpha)d\alpha}\ . \tag{35}$$

A suitable flexible prior distribution $p(\alpha)$ for α is the beta distribution with parameters a and b. Hence,

$$p(\alpha) = [1/\boldsymbol{\beta}(a,b)]\, \alpha^{a-1}(1-\alpha)^{b-1}, \quad 0 \leq \alpha \leq 1. \tag{36}$$

Using this prior distribution gives

$$\begin{aligned}\mathcal{P}_r[\tilde{n}^* \geq 1|\tilde{n}, r] &\geq 1 - \frac{\Gamma(\tilde{n}+a+b)\,\Gamma(2\tilde{n}-r+b)}{\Gamma(2\tilde{n}+a+b)\,\Gamma(\tilde{n}-r+b)} \\ &= 1 - \frac{(\tilde{n}+a+b-1)!\,(2\tilde{n}-r+b-1)!}{(2\tilde{n}+a+b-1)!\,(\tilde{n}-r+b-1)!}.\end{aligned}$$

Assuming a prior expectation of 1, (viz. $a=b=1$), we obtain

$$\mathcal{P}_r[\tilde{n}^* \geq 1|\tilde{n}, r] = 1 - \frac{(\tilde{n}+1)!\,(2\tilde{n}-r)!}{(2\tilde{n}+1)!\,(\tilde{n}-r)!},$$

which is the required result.