



C interfaces to GALAHAD DGO

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Fri Mar 18 2022

1 GALAHAD C package dgo	1
1.1 Introduction	1
1.1.1 Purpose	1
1.1.2 Authors	1
1.1.3 Originally released	1
1.1.4 Terminology	1
1.1.5 Method	2
1.1.6 References	2
1.2 Call order	2
1.3 Symmetric matrix storage formats	3
1.3.1 Dense storage format	3
1.3.2 Sparse co-ordinate storage format	3
1.3.3 Sparse row-wise storage format	3
2 File Index	5
2.1 File List	5
3 File Documentation	7
3.1 dgo.h File Reference	7
3.1.1 Data Structure Documentation	8
3.1.1.1 struct dgo_control_type	8
3.1.1.2 struct dgo_time_type	9
3.1.1.3 struct dgo_inform_type	9
3.1.2 Function Documentation	10
3.1.2.1 dgo_initialize()	10
3.1.2.2 dgo_read_specfile()	11
3.1.2.3 dgo_import()	11
3.1.2.4 dgo_reset_control()	12
3.1.2.5 dgo_solve_with_mat()	13
3.1.2.6 dgo_solve_without_mat()	15
3.1.2.7 dgo_solve_reverse_with_mat()	17
3.1.2.8 dgo_solve_reverse_without_mat()	21
3.1.2.9 dgo_information()	24
3.1.2.10 dgo_terminate()	25
4 Example Documentation	27
4.1 dgot.c	27
4.2 dgotf.c	34
Index	43

Chapter 1

GALAHAD C package dgo

1.1 Introduction

1.1.1 Purpose

The dgo package uses a **deterministic partition-and-bound trust-region method to find an approximation to the global minimizer of a differentiable objective function $f(x)$ of n variables x , subject to a finite set of simple bounds $x^l \leq x \leq x^u$ on the variables.** The method offers the choice of direct and iterative solution of the key trust-region subproblems, and is suitable for large problems. First derivatives are required, and if second derivatives can be calculated, they will be exploited—if the product of second derivatives with a vector may be found but not the derivatives themselves, that may also be exploited.

Although there are theoretical guarantees, these may require a large number of evaluations as the dimension and nonconvexity increase. The alternative GALAHAD package bgo may sometimes be preferred.

1.1.2 Authors

J. Fowkes and N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England.

1.1.3 Originally released

July 2021, C interface August 2021.

1.1.4 Terminology

The *gradient* $\nabla_x f(x)$ of $f(x)$ is the vector whose i -th component is $\partial f(x)/\partial x_i$. The *Hessian* $\nabla_{xx} f(x)$ of $f(x)$ is the symmetric matrix whose i, j -th entry is $\partial^2 f(x)/\partial x_i \partial x_j$. The Hessian is *sparse* if a significant and useful proportion of the entries are universally zero.

1.1.5 Method

Starting with the initial box $x^l \leq x \leq x^u$, a sequence of boxes is generated by considering the current set, and partitioning a promising candidate into three equally-sized sub-boxes by splitting along one of the box dimensions. Each partition requires only a pair of new function and derivative evaluations, and these values, together with estimates of Lipschitz constants, makes it possible to remove other boxes from further consideration as soon as they cannot contain a global minimizer. Efficient control of the dictionary of vertices of the sub-boxes is handled using a suitable hashing procedure provided by the GALAHAD package hash; each sub-box is indexed by the concatenated coordinates of a pair of opposite vertices. At various stages, local minimization in a promising sub-box, using the GALAHAD package trb, may be used to improve the best-known upper bound on the global minimizer. If $n = 1$, the specialised GALAHAD univariate global minimization package ugo is called directly.

We reiterate that although there are theoretical guarantees, these may require a large number of evaluations as the dimension and nonconvexity increase. Thus the method should best be viewed as a heuristic to try to find a reasonable approximation of the global minimum.

1.1.6 References

The global minimization method employed is an extension of that due to

Ya. D. Sergeyev and D. E. Kasov (2015), "A deterministic global optimization using smooth diagonal auxiliary functions", Communications in Nonlinear Science and Numerical Simulation, Vol 21, Nos 1-3, pp. 99-111.

but adapted to use 2nd derivatives, while in the special case when $n = 1$, a simplification based on the ideas in

D. Lera and Ya. D. Sergeyev (2013), "Acceleration of univariate global optimization algorithms working with Lipschitz functions and Lipschitz first derivatives" SIAM J. Optimization Vol. 23, No. 1, pp. 508–529

is used instead. The generic bound-constrained trust-region method used for local minimization is described in detail in

A. R. Conn, N. I. M. Gould and Ph. L. Toint (2000), Trust-region methods. SIAM/MPS Series on Optimization.

1.2 Call order

To solve a given problem, functions from the dgo package must be called in the following order:

- [dgo_initialize](#) - provide default control parameters and set up initial data structures
- [dgo_read_specfile](#) (optional) - override control values by reading replacement values from a file
- [dgo_import](#) - set up problem data structures and fixed values
- [dgo_reset_control](#) (optional) - possibly change control parameters if a sequence of problems are being solved
- solve the problem by calling one of
 - [dgo_solve_with_mat](#) - solve using function calls to evaluate function, gradient and Hessian values
 - [dgo_solve_without_mat](#) - solve using function calls to evaluate function and gradient values and Hessian-vector products
 - [dgo_solve_reverse_with_mat](#) - solve returning to the calling program to obtain function, gradient and Hessian values, or
 - [dgo_solve_reverse_without_mat](#) - solve returning to the calling program to obtain function and gradient values and Hessian-vector products
- [dgo_information](#) (optional) - recover information about the solution and solution process
- [dgo_terminate](#) - deallocate data structures

See Section 4.1 for examples of use.

1.3 Symmetric matrix storage formats

The symmetric n by n matrix $H = \nabla_{xx}f$ may be presented and stored in a variety of formats. But crucially symmetry is exploited by only storing values from the lower triangular part (i.e, those entries that lie on or below the leading diagonal).

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.3.1 Dense storage format

The matrix H is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since H is symmetric, only the lower triangular part (that is the part H_{ij} for $0 \leq j \leq i \leq n - 1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array `H_val` will hold the value H_{ij} (and, by symmetry, H_{ji}) for $0 \leq j \leq i \leq n - 1$.

1.3.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry, $0 \leq l \leq ne - 1$, of H , its row index i , column index j and value H_{ij} , $0 \leq j \leq i \leq n - 1$, are stored as the l -th components of the integer arrays `H_row` and `H_col` and real array `H_val`, respectively, while the number of nonzeros is recorded as `H_ne = ne`. Note that only the entries in the lower triangle should be stored.

1.3.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i -th row of H the i -th component of the integer array `H_ptr` holds the position of the first entry in this row, while `H_ptr(n)` holds the total number of entries plus one. The column indices j , $0 \leq j \leq i$, and values H_{ij} of the entries in the i -th row are stored in components $l = \text{H_ptr}(i), \dots, \text{H_ptr}(i+1)-1$ of the integer array `H_col`, and real array `H_val`, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

dgo.h	7
---------------------------------	---

Chapter 3

File Documentation

3.1 dgo.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
#include "trb.h"
#include "ugo.h"
#include "hash.h"
```

Data Structures

- struct [dgo_control_type](#)
- struct [dgo_time_type](#)
- struct [dgo_inform_type](#)

Functions

- void [dgo_initialize](#) (void **data, struct [dgo_control_type](#) *control, int *status)
- void [dgo_read_specfile](#) (struct [dgo_control_type](#) *control, const char specfile[])
- void [dgo_import](#) (struct [dgo_control_type](#) *control, void **data, int *status, int n, const real_wp_ x_l[], const real_wp_ x_u[], const char H_type[], int ne, const int H_row[], const int H_col[], const int H_ptr[])
- void [dgo_reset_control](#) (struct [dgo_control_type](#) *control, void **data, int *status)
- void [dgo_solve_with_mat](#) (void **data, void *userdata, int *status, int n, real_wp_ x[], real_wp_ g[], int ne, int(*eval_f)(int, const real_wp_[], real_wp_ *, const void *), int(*eval_g)(int, const real_wp_[], real_wp_ [], const void *), int(*eval_h)(int, int, const real_wp_[], real_wp_[], const void *), int(*eval_hprod)(int, const real_wp_[], real_wp_[], const real_wp_[], bool, const void *), int(*eval_prec)(int, const real_wp_[], real_wp_ [], const real_wp_[], const void *))
- void [dgo_solve_without_mat](#) (void **data, void *userdata, int *status, int n, real_wp_ x[], real_wp_ g[], int(*eval_f)(int, const real_wp_[], real_wp_ *, const void *), int(*eval_g)(int, const real_wp_[], real_wp_ [], const void *), int(*eval_hprod)(int, const real_wp_[], real_wp_[], const real_wp_[], bool, const void *), int(*eval_shprod)(int, const real_wp_[], int, const int[], const real_wp_[], int *, int[], real_wp_[], bool, const void *), int(*eval_prec)(int, const real_wp_[], real_wp_[], const real_wp_[], const void *))
- void [dgo_solve_reverse_with_mat](#) (void **data, int *status, int *eval_status, int n, real_wp_ x[], real_wp_ f, real_wp_ g[], int ne, real_wp_ H_val[], const real_wp_ u[], real_wp_ v[])
- void [dgo_solve_reverse_without_mat](#) (void **data, int *status, int *eval_status, int n, real_wp_ x[], real_wp_ f, real_wp_ g[], real_wp_ u[], real_wp_ v[], int index_nz_v[], int *nnz_v, const int index_nz_u[], int nnz_u)
- void [dgo_information](#) (void **data, struct [dgo_inform_type](#) *inform, int *status)
- void [dgo_terminate](#) (void **data, struct [dgo_control_type](#) *control, struct [dgo_inform_type](#) *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct dgo_control_type

Examples

[dgot.c](#), and [dgotf.c](#).

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	error	error and warning diagnostics occur on stream error
int	out	general output occurs on stream out
int	print_level	the level of output required. Possible values are: <ul style="list-style-type: none"> • ≤ 0 no output, • 1 a one-line summary for every improvement • 2 a summary of each iteration • ≥ 3 increasingly verbose (debugging) output
int	start_print	any printing will start on this iteration
int	stop_print	any printing will stop on this iteration
int	print_gap	the number of iterations between printing
int	maxit	the maximum number of iterations performed
int	max_evals	the maximum number of function evaluations made
int	dictionary_size	the size of the initial hash dictionary
int	alive_unit	removal of the file alive_file from unit alive_unit terminates execution
char	alive_file[31]	see alive_unit
real_wp_	infinity	any bound larger than infinity in modulus will be regarded as infinite
real_wp_	lipschitz_lower_bound	a small positive constant ($\leq 1e-6$) that ensure that the estimated gradient Lipschitz constant is not too small
real_wp_	lipschitz_reliability	the Lipschitz reliability parameter, the Lipschitz constant used will be a factor lipschitz_reliability times the largest value observed
real_wp_	lipschitz_control	the reliability control parameter, the actual reliability parameter used will be .lipschitz_reliability <ul style="list-style-type: none"> • $\text{MAX}(1, n - 1) * \text{.lipschitz_control} / \text{iteration}$
real_wp_	stop_length	the iteration will stop if the length, delta, of the diagonal in the box with the smallest-found objective function is smaller than .stop_length times that of the original bound box, delta_0
real_wp_	stop_f	the iteration will stop if the gap between the best objective value found and the smallest lower bound is smaller than .stop_f
real_wp_	obj_unbounded	the smallest value the objective function may take before the problem is marked as unbounded
real_wp_	cpu_time_limit	the maximum CPU time allowed (-ve means infinite)
real_wp_	clock_time_limit	the maximum elapsed clock time allowed (-ve means infinite)

Data Fields

bool	hessian_available	is the Hessian matrix of second derivatives available or is access only via matrix-vector products?
bool	prune	should boxes that cannot contain the global minimizer be pruned (i.e., removed from further consideration)?
bool	perform_local_optimization	should approximate minimizers be improved by judicious local minimization?
bool	space_critical	if .space_critical true, every effort will be made to use as little space as possible. This may result in longer computation time
bool	deallocate_error_fatal	if .deallocate_error_fatal is true, any array/pointer deallocation error will terminate execution. Otherwise, computation will continue
char	prefix[31]	all output lines will be prefixed by prefix(2:LEN(TRIM(prefix))-1) where prefix contains the required string enclosed in quotes, e.g. "string" or 'string'
struct hash_control_type	hash_control	control parameters for HASH
struct ugo_control_type	ugo_control	control parameters for UGO
struct trb_control_type	trb_control	control parameters for TRB

3.1.1.2 struct dgo_time_type

Data Fields

real_sp_	total	the total CPU time spent in the package
real_sp_	univariate_global	the CPU time spent performing univariate global optimization
real_sp_	multivariate_local	the CPU time spent performing multivariate local optimization
real_wp_	clock_total	the total clock time spent in the package
real_wp_	clock_univariate_global	the clock time spent performing univariate global optimization
real_wp_	clock_multivariate_local	the clock time spent performing multivariate local optimization

3.1.1.3 struct dgo_inform_type

Examples

[dgot.c](#), and [dgotf.c](#).

Data Fields

int	status	return status. See DGO_solve for details
int	alloc_status	the status of the last attempted allocation/deallocation
char	bad_alloc[81]	the name of the array for which an allocation/deallocation error occurred
int	iter	the total number of iterations performed
int	f_eval	the total number of evaluations of the objection function
int	g_eval	the total number of evaluations of the gradient of the objection function

Data Fields

int	h_eval	the total number of evaluations of the Hessian of the objection function
real_wp_	obj	the value of the objective function at the best estimate of the solution determined by DGO_solve
real_wp_	norm_pg	the norm of the projected gradient of the objective function at the best estimate of the solution determined by DGO_solve
real_wp_	length_ratio	the ratio of the final to the initial box lengths
real_wp_	f_gap	the gap between the best objective value found and the lowest bound
char	why_stop[2]	why did the iteration stop? This wil be 'D' if the box length is small enough, 'F' if the objective gap is small enough, and '' otherwise
struct dgo_time_type	time	timings (see above)
struct hash_inform_type	hash_inform	inform parameters for HASH
struct ugo_inform_type	ugo_inform	inform parameters for UGO
struct trb_inform_type	trb_inform	inform parameters for UGO

3.1.2 Function Documentation

3.1.2.1 dgo_initialize()

```
void dgo_initialize (
    void ** data,
    struct dgo_control_type * control,
    int * status )
```

Set default control values and initialize private data

Parameters

in, out	<i>data</i>	holds private internal data
out	<i>control</i>	is a struct containing control information (see dgo_control_type)
out	<i>status</i>	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): <ul style="list-style-type: none"> • 0. The import was succesful.

Examples

[dgot.c](#), and [dgotf.c](#).

3.1.2.2 dgo_read_specfile()

```
void dgo_read_specfile (
    struct dgo_control_type * control,
    const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

in, out	<i>control</i>	is a struct containing control information (see dgo_control_type)
in	<i>specfile</i>	is a character string containing the name of the specification file

3.1.2.3 dgo_import()

```
void dgo_import (
    struct dgo_control_type * control,
    void ** data,
    int * status,
    int n,
    const real_wp_ x_l[],
    const real_wp_ x_u[],
    const char H_type[],
    int ne,
    const int H_row[],
    const int H_col[],
    const int H_ptr[] )
```

Import problem data into internal storage prior to solution.

Parameters

in	<i>control</i>	is a struct whose members provide control paramters for the remaining prcedures (see dgo_control_type)
in, out	<i>data</i>	holds private internal data
in, out	<i>status</i>	is a scalar variable of type int, that gives the exit status from the package. Possible values are: <ul style="list-style-type: none"> • 1. The import was succesful, and the package is ready for the solve phase • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction $n > 0$ or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated.

Parameters

in	n	is a scalar variable of type int, that holds the number of variables.
in	x_l	is a one-dimensional array of size n and type double, that holds the values x^l of the lower bounds on the optimization variables x . The j -th component of x_l , $j = 0, \dots, n - 1$, contains x_j^l .
in	x_u	is a one-dimensional array of size n and type double, that holds the values x^u of the upper bounds on the optimization variables x . The j -th component of x_u , $j = 0, \dots, n - 1$, contains x_j^u .
in	H_type	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal' or 'absent', the latter if access to the Hessian is via matrix-vector products; lower or upper case variants are allowed.
in	ne	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes.
in	H_row	is a one-dimensional array of size ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL
in	H_col	is a one-dimensional array of size ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL
in	H_ptr	is a one-dimensional array of size $n+1$ and type int, that holds the starting position of each row of the lower triangular part of H , as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL

Examples

[dgot.c](#), and [dgotf.c](#).

3.1.2.4 dgo_reset_control()

```
void dgo_reset_control (
    struct dgo_control_type * control,
    void ** data,
    int * status )
```

Reset control parameters after import if required.

Parameters

in	<i>control</i>	is a struct whose members provide control parameters for the remaining procedures (see dgo_control_type)
in, out	<i>data</i>	holds private internal data
in, out	<i>status</i>	is a scalar variable of type int, that gives the exit status from the package. Possible values are: <ul style="list-style-type: none"> 1. The import was successful, and the package is ready for the solve phase

3.1.2.5 dgo_solve_with_mat()

```

void dgo_solve_with_mat (
    void ** data,
    void * userdata,
    int * status,
    int n,
    real_wp_ x[],
    real_wp_ g[],
    int ne,
    int(*) (int, const real_wp_[], real_wp_ *, const void *) eval_f,
    int(*) (int, const real_wp_[], real_wp_[], const void *) eval_g,
    int(*) (int, int, const real_wp_[], real_wp_[], const void *) eval_h,
    int(*) (int, const real_wp_[], real_wp_[], const real_wp_[], bool, const void *)
eval_hprod,
    int(*) (int, const real_wp_[], real_wp_[], const real_wp_[], const void *) eval_←
prec )

```

Find an approximation to the global minimizer of a given function subject to simple bounds on the variables using a partition-and-bound trust-region method.

This call is for the case where $H = \nabla_{xx}f(x)$ is provided specifically, and all function/derivative information is available by function calls.

Parameters

in, out	<i>data</i>	holds private internal data
in	<i>userdata</i>	is a structure that allows data to be passed into the function and derivative evaluation programs.

Parameters

<i>in, out</i>	<i>status</i>	<p>is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are:</p> <ul style="list-style-type: none"> • 0. The run was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction $n > 0$ or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -40. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
<i>in</i>	<i>n</i>	is a scalar variable of type int, that holds the number of variables
<i>in, out</i>	<i>x</i>	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x , $j = 0, \dots, n-1$, contains x_j .
<i>in, out</i>	<i>g</i>	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g , $j = 0, \dots, n-1$, contains g_j .
<i>in</i>	<i>ne</i>	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix H .
	<i>eval_f</i>	<p>is a user-supplied function that must have the following signature:</p> <pre>int eval_f(int n, const double x[], double *f, const void *userdata)</pre> <p>The value of the objective function $f(x)$ evaluated at $x = x$ must be assigned to f, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into <i>eval_f</i> via the structure <i>userdata</i>.</p>

Parameters

	<i>eval_g</i>	is a user-supplied function that must have the following signature: <code>int eval_g(int n, const double x[], double g[], const void *userdata)</code> The components of the gradient $g = \nabla_x f(x)$ of the objective function evaluated at $x = x$ must be assigned to <code>g</code> , and the function return value set to 0. If the evaluation is impossible at x , return should be set to a nonzero value. Data may be passed into <code>eval_g</code> via the structure <code>userdata</code> .
	<i>eval_h</i>	is a user-supplied function that must have the following signature: <code>int eval_h(int n, int ne, const double x[], double h[], const void *userdata)</code> The nonzeros of the Hessian $H = \nabla_{xx} f(x)$ of the objective function evaluated at $x = x$ must be assigned to <code>h</code> in the same order as presented to <code>dgo_import</code> , and the function return value set to 0. If the evaluation is impossible at x , return should be set to a nonzero value. Data may be passed into <code>eval_h</code> via the structure <code>userdata</code> .
	<i>eval_prec</i>	is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature: <code>int eval_prec(int n, const double x[], double u[], const double v[], const void *userdata)</code> The product $u = P(x)v$ of the user's preconditioner $P(x)$ evaluated at x with the vector $v = v$, the result u must be returned in <code>u</code> , and the function return value set to 0. If the evaluation is impossible at x , return should be set to a nonzero value. Data may be passed into <code>eval_prec</code> via the structure <code>userdata</code> .

Examples

[dgot.c](#), and [dgotf.c](#).

3.1.2.6 dgo_solve_without_mat()

```
void dgo_solve_without_mat (
    void ** data,
    void * userdata,
    int * status,
    int n,
    real_wp_ x[],
    real_wp_ g[],
    int(*) (int, const real_wp_[], real_wp_ *, const void *) eval_f,
    int(*) (int, const real_wp_[], real_wp_[], const void *) eval_g,
    int(*) (int, const real_wp_[], real_wp_[], const real_wp_[], bool, const void *)
eval_hprod,
    int(*) (int, const real_wp_[], int, const int[], const real_wp_[], int *, int[],
real_wp_[], bool, const void *) eval_shprod,
    int(*) (int, const real_wp_[], real_wp_[], const real_wp_[], const void *) eval_←
prec )
```

Find an approximation to the global minimizer of a given function subject to simple bounds on the variables using a partition-and-bound trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, and all function/derivative information is available by function calls.

Parameters

in, out	<i>data</i>	holds private internal data
in	<i>userdata</i>	is a structure that allows data to be passed into the function and derivative evaluation programs.
in, out	<i>status</i>	<p>is a scalar variable of type int, that gives the entry and exit status from the package. On initial entry, status must be set to 1. Possible exit are:</p> <ul style="list-style-type: none"> • 0. The run was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction $n > 0$ or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -40. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
in	<i>n</i>	is a scalar variable of type int, that holds the number of variables
in, out	<i>x</i>	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j-th component of x , $j = 0, \dots, n-1$, contains x_j .
in, out	<i>g</i>	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g , $j = 0, \dots, n-1$, contains g_j .

Parameters

	<i>eval_f</i>	<p>is a user-supplied function that must have the following signature:</p> <pre>int eval_f(int n, const double x[], double *f, const void *userdata)</pre> <p>The value of the objective function $f(x)$ evaluated at $x = x$ must be assigned to f, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into <i>eval_f</i> via the structure <i>userdata</i>.</p>
	<i>eval_g</i>	<p>is a user-supplied function that must have the following signature:</p> <pre>int eval_g(int n, const double x[], double g[], const void *userdata)</pre> <p>The components of the gradient $g = \nabla_x f(x)$ of the objective function evaluated at $x = x$ must be assigned to g, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into <i>eval_g</i> via the structure <i>userdata</i>.</p>
	<i>eval_hprod</i>	<p>is a user-supplied function that must have the following signature:</p> <pre>int eval_hprod(int n, const double x[], double u[], const double v[], bool got_h, const void *userdata)</pre> <p>The sum $u + \nabla_{xx} f(x)v$ of the product of the Hessian $\nabla_{xx} f(x)$ of the objective function evaluated at $x = x$ with the vector $v = v$ and the vector u must be returned in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. The Hessian has already been evaluated or used at x if <i>got_h</i> is true. Data may be passed into <i>eval_hprod</i> via the structure <i>userdata</i>.</p>
	<i>eval_shprod</i>	<p>is a user-supplied function that must have the following signature:</p> <pre>int eval_shprod(int n, const double x[], int nnz_v, const int index_nz_v[], const double v[], int *nnz_u, int index_nz_u[], double u[], bool got_h, const void *userdata)</pre> <p>The product $u = \nabla_{xx} f(x)v$ of the Hessian $\nabla_{xx} f(x)$ of the objective function evaluated at x with the sparse vector $v = v$ must be returned in u, and the function return value set to 0. Only the components $\text{index_nz_v}[0:\text{nnz_v}-1]$ of v are nonzero, and the remaining components may not have been set. On exit, the user must indicate the <i>nnz_u</i> indices of u that are nonzero in $\text{index_nz_u}[0:\text{nnz_u}-1]$, and only these components of u need be set. If the evaluation is impossible at x, return should be set to a nonzero value. The Hessian has already been evaluated or used at x if <i>got_h</i> is true. Data may be passed into <i>eval_prec</i> via the structure <i>userdata</i>.</p>
	<i>eval_prec</i>	<p>is an optional user-supplied function that may be NULL. If non-NULL, it must have the following signature:</p> <pre>int eval_prec(int n, const double x[], double u[], const double v[], const void *userdata)</pre> <p>The product $u = P(x)v$ of the user's preconditioner $P(x)$ evaluated at x with the vector $v = v$, the result u must be returned in u, and the function return value set to 0. If the evaluation is impossible at x, return should be set to a nonzero value. Data may be passed into <i>eval_prec</i> via the structure <i>userdata</i>.</p>

Examples

[dgotf.c](#), and [dgotf.c](#).

3.1.2.7 dgo_solve_reverse_with_mat()

```
void dgo_solve_reverse_with_mat (
    void ** data,
    int * status,
```

```

    int * eval_status,
    int n,
    real_wp_ x[],
    real_wp_ f,
    real_wp_ g[],
    int ne,
    real_wp_ H_val[],
    const real_wp_ u[],
    real_wp_ v[] )

```

Find an approximation to the global minimizer of a given function subject to simple bounds on the variables using a partition-and-bound trust-region method.

This call is for the case where $H = \nabla_{xx}f(x)$ is provided specifically, but function/derivative information is only available by returning to the calling procedure

Parameters

in, out	<i>data</i>	holds private internal data
---------	-------------	-----------------------------

Parameters

<code>in, out</code>	<code>status</code>	<p>is a scalar variable of type <code>int</code>, that gives the entry and exit status from the package.</p> <p>On initial entry, <code>status</code> must be set to 1. Possible exit are:</p> <ul style="list-style-type: none"> • 0. The run was succesful • -1. An allocation error occurred. A message indicating the offending array is written on <code>unit.control.error</code>, and the returned allocation status and a string containing the name of the offending array are held in <code>inform.alloc_status</code> and <code>inform.bad_alloc</code> respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on <code>unit.control.error</code> and the returned allocation status and a string containing the name of the offending array are held in <code>inform.alloc_status</code> and <code>inform.bad_alloc</code> respectively. • -3. The restriction $n > 0$ or requirement that <code>type</code> contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component <code>inform.factor_status</code> • -10. The factorization failed; the return status from the factorization package is given in the component <code>inform.factor_status</code>. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component <code>inform.factor_status</code>. • -16. The problem is so ill-conditioned that further progress is impossible. • -18. Too many iterations have been performed. This may happen if <code>control.maxit</code> is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if <code>control.cpu_time_limit</code> is too small, but may also be symptomatic of a badly scaled problem. • -40. The user has forced termination of solver by removing the file named <code>control.alive_file</code> from unit <code>unit.control.alive_unit</code>.
----------------------	---------------------	---

Parameters

	<i>status</i>	(continued) <ul style="list-style-type: none"> • 2. The user should compute the objective function value $f(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in f, and $eval_status$ should be set to 0. If the user is unable to evaluate $f(x)$— for instance, if the function is undefined at x— the user need not set f, but should then set $eval_status$ to a non-zero value. • 3. The user should compute the gradient of the objective function $\nabla_x f(x)$ at the point x indicated in x and then re-enter the function. The value of the i-th component of the g gradient should be set in $g[i]$, for $i = 0, \dots, n-1$ and $eval_status$ should be set to 0. If the user is unable to evaluate a component of $\nabla_x f(x)$ — for instance if a component of the gradient is undefined at x — the user need not set g, but should then set $eval_status$ to a non-zero value. • 4. The user should compute the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x and then re-enter the function. The value l-th component of the Hessian stored according to the scheme input in the remainder of H should be set in $H_val[l]$, for $l = 0, \dots, ne-1$ and $eval_status$ should be set to 0. If the user is unable to evaluate a component of $\nabla_{xx} f(x)$ — for instance, if a component of the Hessian is undefined at x — the user need not set H_val, but should then set $eval_status$ to a non-zero value. • 5. The user should compute the product $\nabla_{xx} f(x)v$ of the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x with the vector v, add the result to the vector u and then re-enter the function. The vectors u and v are given in u and v respectively, the resulting vector $u + \nabla_{xx} f(x)v$ should be set in u and $eval_status$ should be set to 0. If the user is unable to evaluate the product— for instance, if a component of the Hessian is undefined at x — the user need not alter u, but should then set $eval_status$ to a non-zero value. • 6. The user should compute the product $u = P(x)v$ of their preconditioner $P(x)$ at the point x indicated in x with the vector v and then re-enter the function. The vector v is given in v, the resulting vector $u = P(x)v$ should be set in u and $eval_status$ should be set to 0. If the user is unable to evaluate the product— for instance, if a component of the preconditioner is undefined at x — the user need not set u, but should then set $eval_status$ to a non-zero value. • 23. The user should follow the instructions for 2 and 3 above before returning. • 25. The user should follow the instructions for 2 and 5 above before returning. • 35. The user should follow the instructions for 3 and 5 above before returning. • 235. The user should follow the instructions for 2, 3 and 5 above before returning.
<i>in, out</i>	<i>eval_status</i>	is a scalar variable of type int, that is used to indicate if objective function/gradient/Hessian values can be provided (see above)
<i>in</i>	<i>n</i>	is a scalar variable of type int, that holds the number of variables
<i>in, out</i>	<i>x</i>	is a one-dimensional array of size n and type double, that holds the values x of the optimization variables. The j -th component of x , $j = 0, \dots, n-1$, contains x_j .
<i>in</i>	<i>f</i>	is a scalar variable pointer of type double, that holds the value of the objective function.

Parameters

in, out	<i>g</i>	is a one-dimensional array of size <i>n</i> and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The <i>j</i> -th component of <i>g</i> , $j = 0, \dots, n-1$, contains g_j .
in	<i>ne</i>	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the Hessian matrix <i>H</i> .
in	<i>H_val</i>	is a one-dimensional array of size <i>ne</i> and type double, that holds the values of the entries of the lower triangular part of the Hessian matrix <i>H</i> in any of the available storage schemes.
in	<i>u</i>	is a one-dimensional array of size <i>n</i> and type double, that is used for reverse communication (see above for details)
in, out	<i>v</i>	is a one-dimensional array of size <i>n</i> and type double, that is used for reverse communication (see above for details)

Examples

[dgot.c](#), and [dgottf.c](#).

3.1.2.8 dgo_solve_reverse_without_mat()

```
void dgo_solve_reverse_without_mat (
    void ** data,
    int * status,
    int * eval_status,
    int n,
    real_wp_ x[],
    real_wp_ f,
    real_wp_ g[],
    real_wp_ u[],
    real_wp_ v[],
    int index_nz_v[],
    int * nnz_v,
    const int index_nz_u[],
    int nnz_u )
```

Find an approximation to the global minimizer of a given function subject to simple bounds on the variables using a partition-and-bound trust-region method.

This call is for the case where access to $H = \nabla_{xx} f(x)$ is provided by Hessian-vector products, but function/derivative information is only available by returning to the calling procedure.

Parameters

in, out	<i>data</i>	holds private internal data
---------	-------------	-----------------------------

Parameters

<code>in, out</code>	<code>status</code>	<p>is a scalar variable of type int, that gives the entry and exit status from the package.</p> <p>On initial entry, status must be set to 1. Possible exit are:</p> <ul style="list-style-type: none"> • 0. The run was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restriction $n > 0$ or requirement that type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows', 'diagonal' or 'absent' has been violated. • -7. The objective function appears to be unbounded from below • -9. The analysis phase of the factorization failed; the return status from the factorization package is given in the component inform.factor_status • -10. The factorization failed; the return status from the factorization package is given in the component inform.factor_status. • -11. The solution of a set of linear equations using factors from the factorization package failed; the return status from the factorization package is given in the component inform.factor_status. • -16. The problem is so ill-conditioned that further progress is impossible. • -18. Too many iterations have been performed. This may happen if control.maxit is too small, but may also be symptomatic of a badly scaled problem. • -19. The CPU time limit has been reached. This may happen if control.cpu_time_limit is too small, but may also be symptomatic of a badly scaled problem. • -40. The user has forced termination of solver by removing the file named control.alive_file from unit unit control.alive_unit.
----------------------	---------------------	---

Parameters

	<i>status</i>	<p>(continued)</p> <ul style="list-style-type: none"> 2. The user should compute the objective function value $f(x)$ at the point x indicated in x and then re-enter the function. The required value should be set in f, and <code>eval_status</code> should be set to 0. If the user is unable to evaluate $f(x)$ — for instance, if the function is undefined at x — the user need not set f, but should then set <code>eval_status</code> to a non-zero value. 3. The user should compute the gradient of the objective function $\nabla_x f(x)$ at the point x indicated in x and then re-enter the function. The value of the i-th component of the gradient should be set in $g[i]$, for $i = 0, \dots, n-1$ and <code>eval_status</code> should be set to 0. If the user is unable to evaluate a component of $\nabla_x f(x)$ — for instance if a component of the gradient is undefined at x — the user need not set g, but should then set <code>eval_status</code> to a non-zero value. 5. The user should compute the product $\nabla_{xx} f(x)v$ of the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x with the vector v, add the result to the vector u and then re-enter the function. The vectors u and v are given in u and v respectively, the resulting vector $u + \nabla_{xx} f(x)v$ should be set in u and <code>eval_status</code> should be set to 0. If the user is unable to evaluate the product — for instance, if a component of the Hessian is undefined at x — the user need not alter u, but should then set <code>eval_status</code> to a non-zero value. 6. The user should compute the product $u = P(x)v$ of their preconditioner $P(x)$ at the point x indicated in x with the vector v and then re-enter the function. The vector v is given in v, the resulting vector $u = P(x)v$ should be set in u and <code>eval_status</code> should be set to 0. If the user is unable to evaluate the product — for instance, if a component of the preconditioner is undefined at x — the user need not set u, but should then set <code>eval_status</code> to a non-zero value. 7. The user should compute the product $u = \nabla_{xx} f(x)v$ of the Hessian of the objective function $\nabla_{xx} f(x)$ at the point x indicated in x with the sparse vector $v = v$ and then re-enter the function. The nonzeros of v are stored in <code>v[index_nz_v[0:nnz_v-1]]</code> while the nonzeros of u should be returned in <code>u[index_nz_u[0:nnz_u-1]]</code>; the user must set <code>nnz_u</code> and <code>index_nz_u</code> accordingly, and set <code>eval_status</code> to 0. If the user is unable to evaluate the product — for instance, if a component of the Hessian is undefined at x — the user need not alter u, but should then set <code>eval_status</code> to a non-zero value. 23. The user should follow the instructions for 2 and 3 above before returning. 25. The user should follow the instructions for 2 and 5 above before returning. 35. The user should follow the instructions for 3 and 5 above before returning. 235. The user should follow the instructions for 2, 3 and 5 above before returning.
<i>in, out</i>	<i>eval_status</i>	is a scalar variable of type <code>int</code> , that is used to indicate if objective function/gradient/Hessian values can be provided (see above)
<i>in</i>	<i>n</i>	is a scalar variable of type <code>int</code> , that holds the number of variables
<i>in, out</i>	<i>x</i>	is a one-dimensional array of size n and type <code>double</code> , that holds the values x of the optimization variables. The j -th component of x , $j = 0, \dots, n-1$, contains x_j .

Parameters

in	<i>f</i>	is a scalar variable pointer of type double, that holds the value of the objective function.
in, out	<i>g</i>	is a one-dimensional array of size n and type double, that holds the gradient $g = \nabla_x f(x)$ of the objective function. The j-th component of g, $j = 0, \dots, n-1$, contains g_j .
in, out	<i>u</i>	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=5,6,7 above for details)
in, out	<i>v</i>	is a one-dimensional array of size n and type double, that is used for reverse communication (see status=5,6,7 above for details)
in, out	<i>index_nz↔ _v</i>	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=7 above for details)
in, out	<i>nnz_v</i>	is a scalar variable of type int, that is used for reverse communication (see status=7 above for details)
in	<i>index_nz↔ _u</i>	is a one-dimensional array of size n and type int, that is used for reverse communication (see status=7 above for details)
in	<i>nnz_u</i>	is a scalar variable of type int, that is used for reverse communication (see status=7 above for details). On initial (status=1) entry, nnz_u should be set to an (arbitrary) nonzero value, and nnz_u=0 is recommended

Examples

[dgot.c](#), and [dgotf.c](#).

3.1.2.9 dgo_information()

```
void dgo_information (
    void ** data,
    struct dgo_inform_type * inform,
    int * status )
```

Provides output information

Parameters

in, out	<i>data</i>	holds private internal data
out	<i>inform</i>	is a struct containing output information (see dgo_inform_type)
out	<i>status</i>	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): <ul style="list-style-type: none"> • 0. The values were recorded succesfully

Examples

[dgot.c](#), and [dgotf.c](#).

3.1.2.10 dgo_terminate()

```
void dgo_terminate (
    void ** data,
    struct dgo\_control\_type * control,
    struct dgo\_inform\_type * inform )
```

Deallocate all internal private storage

Parameters

in, out	<i>data</i>	holds private internal data
out	<i>control</i>	is a struct containing control information (see dgo_control_type)
out	<i>inform</i>	is a struct containing output information (see dgo_inform_type)

Examples

[dgot.c](#), and [dgotf.c](#).

Chapter 4

Example Documentation

4.1 dgot.c

This is an example of how to use the package to find an approximation to the global minimum of a given function within a bounded region. A variety of supported Hessian and constraint matrix storage formats are shown.

Notice that C-style indexing is used, and that this is flagged by setting `control.f_indexing` to `false`.

```
/* dgot.c */
/* Full test for the DGO C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "dgo.h"
// Custom userdata struct
struct userdata_type {
    double p;
    double freq;
    double mag;
};
// Function prototypes
int fun( int n, const double x[], double *f, const void * );
int grad( int n, const double x[], double g[], const void * );
int hess( int n, int ne, const double x[], double hval[], const void * );
int hess_dense( int n, int ne, const double x[], double hval[], const void * );
int hessprod( int n, const double x[], double u[], const double v[],
    bool got_h, const void * );
int shessprod( int n, const double x[], int nnz_v, const int index_nz_v[],
    const double v[], int *nnz_u, int index_nz_u[], double u[],
    bool got_h, const void * );
int prec( int n, const double x[], double u[], const double v[], const void * );
int fun_diag( int n, const double x[], double *f, const void * );
int grad_diag( int n, const double x[], double g[], const void * );
int hess_diag( int n, int ne, const double x[], double hval[], const void * );
int hessprod_diag( int n, const double x[], double u[], const double v[],
    bool got_h, const void * );
int shessprod_diag( int n, const double x[], int nnz_v, const int index_nz_v[],
    const double v[], int *nnz_u, int index_nz_u[],
    double u[], bool got_h, const void * );

int main(void) {
    // Derived types
    void *data;
    struct dgo_control_type control;
    struct dgo_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.p = 4.0;
    userdata.freq = 10;
    userdata.mag = 1000;
    // Set problem data
    int n = 3; // dimension
    int ne = 5; // Hessian elements
    double x_l[] = {-10,-10,-10};
    double x_u[] = {0.5,0.5,0.5};
    int H_row[] = {0, 1, 2, 2, 2}; // Hessian H
    int H_col[] = {0, 1, 0, 1, 2}; // NB lower triangle
```

```

int H_ptr[] = {0, 1, 2, 5};    // row pointers
// Set storage
double g[n]; // gradient
char st;
int status;
printf(" C sparse matrix indexing\n\n");
printf(" tests options for all-in-one storage format\n\n");
for(int d=1; d <= 5; d++){
    // Initialize DGO
    dgo_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing
    control.maxit = 2500;
    // control.trb_control.maxit = 100;
    //control.print_level = 1;
    // Start from 0
    double x[] = {0,0,0};
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "coordinate", ne, H_row, H_col, NULL );
            dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                ne, fun, grad, hess, hessprod, prec );
            break;
        case 2: // sparse by rows
            st = 'R';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "sparse_by_rows", ne, NULL, H_col, H_ptr );
            dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                ne, fun, grad, hess, hessprod, prec );
            break;
        case 3: // dense
            st = 'D';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "dense", ne, NULL, NULL, NULL );
            dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                ne, fun, grad, hess_dense, hessprod, prec );
            break;
        case 4: // diagonal
            st = 'I';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "diagonal", ne, NULL, NULL, NULL );
            dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                ne, fun_diag, grad_diag, hess_diag,
                hessprod_diag, prec );
            break;
        case 5: // access by products
            st = 'P';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "absent", ne, NULL, NULL, NULL );
            dgo_solve_without_mat( &data, &userdata, &status, n, x, g,
                fun, grad, hessprod, shessprod, prec );
            break;
    }
    // Record solution information
    dgo_information( &data, &inform, &status );
    if(inform.status == 0){
        printf("%c:%6i evaluations. Optimal objective value = %5.2f"
            " status = %1i\n", st, inform.f_eval, inform.obj, inform.status);
    }else if(inform.status == -18){
        printf("%c:%6i evaluations. Best objective value = %5.2f"
            " status = %1i\n", st, inform.f_eval, inform.obj, inform.status);
    }else{
        printf("%c: DGO_solve exit status = %1i\n", st, inform.status);
    }
    //printf("x: ");
    //for(int i = 0; i < n; i++) printf("%f ", x[i]);
    //printf("\n");
    //printf("gradient: ");
    //for(int i = 0; i < n; i++) printf("%f ", g[i]);
    //printf("\n");
    // Delete internal workspace
    dgo_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int eval_status, nnz_u, nnz_v;
double f = 0.0;
double u[n], v[n];
int index_nz_u[n], index_nz_v[n];
double H_val[ne], H_dense[n*(n+1)/2], H_diag[n];
for(int d=1; d <= 5; d++){
    // Initialize DGO
    dgo_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = false; // C sparse matrix indexing

```



```

control.maxit = 2500;
// control.trb_control.maxit = 100;
//control.print_level = 1;
// Start from 0
double x[] = {0,0,0};
switch(d){
  case 1: // sparse co-ordinate storage
    st = 'C';
    dgo_import( &control, &data, &status, n, x_l, x_u,
               "coordinate", ne, H_row, H_col, NULL );
    while(true){ // reverse-communication loop
      dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                                n, x, f, g, ne, H_val, u, v );
      if(status == 0){ // successful termination
        break;
      }else if(status < 0){ // error exit
        break;
      }else if(status == 2){ // evaluate f
        eval_status = fun( n, x, &f, &userdata );
      }else if(status == 3){ // evaluate g
        eval_status = grad( n, x, g, &userdata );
      }else if(status == 4){ // evaluate H
        eval_status = hess( n, ne, x, H_val, &userdata );
      }else if(status == 5){ // evaluate Hv product
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 6){ // evaluate the product with P
        eval_status = prec( n, x, u, v, &userdata );
      }else if(status == 23){ // evaluate f and g
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
      }else if(status == 25){ // evaluate f and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 35){ // evaluate g and Hv product
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 235){ // evaluate f, g and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else{
        printf(" the value %li of status should not occur\n",
              status );
        break;
      }
    }
    break;
  case 2: // sparse by rows
    st = 'R';
    dgo_import( &control, &data, &status, n, x_l, x_u,
               "sparse_by_rows", ne, NULL, H_col, H_ptr );
    while(true){ // reverse-communication loop
      dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                                n, x, f, g, ne, H_val, u, v );
      if(status == 0){ // successful termination
        break;
      }else if(status < 0){ // error exit
        break;
      }else if(status == 2){ // evaluate f
        eval_status = fun( n, x, &f, &userdata );
      }else if(status == 3){ // evaluate g
        eval_status = grad( n, x, g, &userdata );
      }else if(status == 4){ // evaluate H
        eval_status = hess( n, ne, x, H_val, &userdata );
      }else if(status == 5){ // evaluate Hv product
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 6){ // evaluate the product with P
        eval_status = prec( n, x, u, v, &userdata );
      }else if(status == 23){ // evaluate f and g
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
      }else if(status == 25){ // evaluate f and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 35){ // evaluate g and Hv product
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else if(status == 235){ // evaluate f, g and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
      }else{
        printf(" the value %li of status should not occur\n",
              status );
        break;
      }
    }
  }
}

```

```

        break;
case 3: // dense
    st = 'D';
    dgo_import( &control, &data, &status, n, x_l, x_u,
                "dense", ne, NULL, NULL, NULL );
    while(true){ // reverse-communication loop
        dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                                    n, x, f, g, n*(n+1)/2,
                                    H_dense, u, v );
        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate f
            eval_status = fun( n, x, &f, &userdata );
        }else if(status == 3){ // evaluate g
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 4){ // evaluate H
            eval_status = hess_dense( n, n*(n+1)/2, x, H_dense,
                                      &userdata );
        }else if(status == 5){ // evaluate Hv product
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 6){ // evaluate the product with P
            eval_status = prec( n, x, u, v, &userdata );
        }else if(status == 23){ // evaluate f and g
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 25){ // evaluate f and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 35){ // evaluate g and Hv product
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 235){ // evaluate f, g and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else{
            printf(" the value %li of status should not occur\n",
                   status);
            break;
        }
    }
    break;
case 4: // diagonal
    st = 'I';
    dgo_import( &control, &data, &status, n, x_l, x_u,
                "diagonal", ne, NULL, NULL, NULL );
    while(true){ // reverse-communication loop
        dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                                    n, x, f, g, n, H_diag, u, v );
        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate f
            eval_status = fun_diag( n, x, &f, &userdata );
        }else if(status == 3){ // evaluate g
            eval_status = grad_diag( n, x, g, &userdata );
        }else if(status == 4){ // evaluate H
            eval_status = hess_diag( n, n, x, H_diag, &userdata );
        }else if(status == 5){ // evaluate Hv product
            eval_status = hessprod_diag( n, x, u, v, false,
                                         &userdata );
        }else if(status == 6){ // evaluate the product with P
            eval_status = prec( n, x, u, v, &userdata );
        }else if(status == 23){ // evaluate f and g
            eval_status = fun_diag( n, x, &f, &userdata );
            eval_status = grad_diag( n, x, g, &userdata );
        }else if(status == 25){ // evaluate f and Hv product
            eval_status = fun_diag( n, x, &f, &userdata );
            eval_status = hessprod_diag( n, x, u, v, false,
                                         &userdata );
        }else if(status == 35){ // evaluate g and Hv product
            eval_status = grad_diag( n, x, g, &userdata );
            eval_status = hessprod_diag( n, x, u, v, false,
                                         &userdata );
        }else if(status == 235){ // evaluate f, g and Hv product
            eval_status = fun_diag( n, x, &f, &userdata );
            eval_status = grad_diag( n, x, g, &userdata );
            eval_status = hessprod_diag( n, x, u, v, false,
                                         &userdata );
        }else{
            printf(" the value %li of status should not occur\n",
                   status);
            break;
        }
    }
}

```

```

    }
    break;
case 5: // access by products
    st = 'P';
    dgo_import( &control, &data, &status, n, x_l, x_u,
               "absent", ne, NULL, NULL, NULL );
    nnz_u = 0;
    while(true){ // reverse-communication loop
        dgo_solve_reverse_without_mat( &data, &status, &eval_status,
                                       n, x, f, g, u, v, index_nz_v,
                                       &nnz_v, index_nz_u, nnz_u );

        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate f
            eval_status = fun( n, x, &f, &userdata );
        }else if(status == 3){ // evaluate g
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 5){ // evaluate Hv product
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 6){ // evaluate the product with P
            eval_status = prec( n, x, u, v, &userdata );
        }else if(status == 7){ // evaluate sparse Hess-vect product
            eval_status = shessprod( n, x, nnz_v, index_nz_v, v,
                                     &nnz_u, index_nz_u, u,
                                     false, &userdata );
        }else if(status == 23){ // evaluate f and g
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 25){ // evaluate f and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 35){ // evaluate g and Hv product
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 235){ // evaluate f, g and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else{
            printf(" the value %li of status should not occur\n",
                  status);
            break;
        }
    }
    break;
}
// Record solution information
dgo_information( &data, &inform, &status );
if(inform.status == 0){
    printf("%c:%6i evaluations. Optimal objective value = %5.2f"
          " status = %li\n", st, inform.f_eval, inform.obj, inform.status);
}else if(inform.status == -18){
    printf("%c:%6i evaluations. Best objective value = %5.2f"
          " status = %li\n", st, inform.f_eval, inform.obj, inform.status);
}else{
    printf("%c: DGO_solve exit status = %li\n", st, inform.status);
}
//printf("x: ");
//for(int i = 0; i < n; i++) printf("%f ", x[i]);
//printf("\n");
//printf("gradient: ");
//for(int i = 0; i < n; i++) printf("%f ", g[i]);
//printf("\n");
// Delete internal workspace
dgo_terminate( &data, &control, &inform );
}
}

// Objective function
int fun( int n,
        const double x[],
        double *f,
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    *f = pow(x[0] + x[2] + p, 2) + pow(x[1] + x[2], 2) + mag * cos(freq*x[0])
        + x[0] + x[1] + x[2];
    return 0;
}

// Gradient of the objective
int grad( int n,
        const double x[],
        double g[],
        const void *userdata ){

```

```

    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    g[0] = 2.0 * ( x[0] + x[2] + p ) - mag * freq * sin(freq*x[0]) + 1;
    g[1] = 2.0 * ( x[1] + x[2] ) + 1;
    g[2] = 2.0 * ( x[0] + x[2] + p ) + 2.0 * ( x[1] + x[2] ) + 1;
    return 0;
}
// Hessian of the objective
int hess( int n,
          int ne,
          const double x[],
          double hval[],
          const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    hval[0] = 2.0 - mag * freq * freq * cos(freq*x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 4.0;
    return 0;
}
// Dense Hessian
int hess_dense( int n,
                int ne,
                const double x[],
                double hval[],
                const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    hval[0] = 2.0 - mag * freq * freq * cos(freq*x[0]);
    hval[1] = 0.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 2.0;
    hval[5] = 4.0;
    return 0;
}
// Hessian-vector product
int hessprod( int n,
              const double x[],
              double u[],
              const double v[],
              bool got_h,
              const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    u[0] = u[0] + 2.0 * ( v[0] + v[2] )
           - mag * freq * freq * cos(freq*x[0]) * v[0];
    u[1] = u[1] + 2.0 * ( v[1] + v[2] );
    u[2] = u[2] + 2.0 * ( v[0] + v[1] + 2.0 * v[2] );
    return 0;
}
// Sparse Hessian-vector product
int shessprod( int n,
               const double x[],
               int nnz_v,
               const int index_nz_v[],
               const double v[],
               int *nnz_u,
               int index_nz_u[],
               double u[],
               bool got_h,
               const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    double p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for(int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 0:
                p[0] = p[0] + 2.0 * v[0]
                     - mag * freq * freq * cos(freq*x[0]) * v[0];
                used[0] = true;
                p[2] = p[2] + 2.0 * v[0];
                used[2] = true;
                break;
            case 1:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;

```

```

        p[2] = p[2] + 2.0 * v[1];
        used[2] = true;
        break;
    case 2:
        p[0] = p[0] + 2.0 * v[2];
        used[0] = true;
        p[1] = p[1] + 2.0 * v[2];
        used[1] = true;
        p[2] = p[2] + 4.0 * v[2];
        used[2] = true;
        break;
    }
}
*nnz_u = 0;
for(int j = 0; j < 3; j++){
    if(used[j]){
        u[j] = p[j];
        *nnz_u = *nnz_u + 1;
        index_nz_u[*nnz_u-1] = j;
    }
}
return 0;
}
// Apply preconditioner
int prec( int n,
          const double x[],
          double u[],
          const double v[],
          const void *userdata ){
    u[0] = 0.5 * v[0];
    u[1] = 0.5 * v[1];
    u[2] = 0.25 * v[2];
    return 0;
}
// Objective function
int fun_diag( int n,
              const double x[],
              double *f,
              const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    *f = pow(x[2] + p, 2) + pow(x[1], 2) + mag * cos(freq*x[0])
        + x[0] + x[1] + x[2];
    return 0;
}
// Gradient of the objective
int grad_diag( int n,
               const double x[],
               double g[],
               const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    g[0] = -mag * freq * sin(freq*x[0]) + 1;
    g[1] = 2.0 * x[1] + 1;
    g[2] = 2.0 * ( x[2] + p ) + 1;
    return 0;
}
// Hessian of the objective
int hess_diag( int n,
               int ne,
               const double x[],
               double hval[],
               const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    hval[0] = -mag * freq * freq * cos(freq*x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    return 0;
}
// Hessian-vector product
int hessprod_diag( int n,
                   const double x[],
                   double u[],
                   const double v[],
                   bool got_h,
                   const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    u[0] = u[0] + -mag * freq * freq * cos(freq*x[0]) * v[0];
    u[1] = u[1] + 2.0 * v[1];

```

```

    u[2] = u[2] + 2.0 * v[2];
    return 0;
}
// Sparse Hessian-vector product
int shessprod_diag( int n,
                    const double x[],
                    int nnz_v,
                    const int index_nz_v[],
                    const double v[],
                    int *nnz_u,
                    int index_nz_u[],
                    double u[],
                    bool got_h,
                    const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    double p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for(int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 0:
                p[0] = p[0] - mag * freq * freq * cos(freq*x[0]) * v[0];
                used[0] = true;
                break;
            case 1:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                break;
            case 2:
                p[2] = p[2] + 2.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for(int j = 0; j < 3; j++){
        if(used[j]){
            u[j] = p[j];
            *nnz_u = *nnz_u + 1;
            index_nz_u[*nnz_u-1] = j;
        }
    }
    return 0;
}

```

4.2 dgotf.c

This is the same example, but now fortran-style indexing is used.

```

/* dgot2.c */
/* Full test for the DGO C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "dgo.h"
// Custom userdata struct
struct userdata_type {
    double p;
    double freq;
    double mag;
};
// Function prototypes
int fun( int n, const double x[], double *f, const void * );
int grad( int n, const double x[], double g[], const void * );
int hess( int n, int ne, const double x[], double hval[], const void * );
int hess_dense( int n, int ne, const double x[], double hval[], const void * );
int hessprod( int n, const double x[], double u[], const double v[],
              bool got_h, const void * );
int shessprod( int n, const double x[], int nnz_v, const int index_nz_v[],
               const double v[], int *nnz_u, int index_nz_u[], double u[],
               bool got_h, const void * );
int prec( int n, const double x[], double u[], const double v[], const void * );
int fun_diag( int n, const double x[], double *f, const void * );
int grad_diag( int n, const double x[], double g[], const void * );
int hess_diag( int n, int ne, const double x[], double hval[], const void * );
int hessprod_diag( int n, const double x[], double u[], const double v[],
                  bool got_h, const void * );
int shessprod_diag( int n, const double x[], int nnz_v, const int index_nz_v[],
                   const double v[], int *nnz_u, int index_nz_u[],
                   double u[], bool got_h, const void * );

```

```

int main(void) {
    // Derived types
    void *data;
    struct dgo_control_type control;
    struct dgo_inform_type inform;
    // Set user data
    struct userdata_type userdata;
    userdata.p = 4.0;
    userdata.freq = 10;
    userdata.mag = 1000;
    // Set problem data
    int n = 3; // dimension
    int ne = 5; // Hesssian elements
    double x_l[] = {-10,-10,-10};
    double x_u[] = {0.5,0.5,0.5};
    int H_row[] = {1, 2, 3, 3, 3}; // Hessian H
    int H_col[] = {1, 2, 1, 2, 3}; // NB lower triangle
    int H_ptr[] = {1, 2, 3, 6}; // row pointers
    // Set storage
    double g[n]; // gradient
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" tests options for all-in-one storage format\n\n");
    for(int d=1; d <= 5; d++){
        // Initialize DGO
        dgo_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        control.maxit = 2500;
        // control.trb_control.maxit = 100;
        //control.print_level = 1;
        // Start from 0
        double x[] = {0,0,0};
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                dgo_import( &control, &data, &status, n, x_l, x_u,
                           "coordinate", ne, H_row, H_col, NULL );
                dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                                   ne, fun, grad, hess, hessprod, prec );
                break;
            case 2: // sparse by rows
                st = 'R';
                dgo_import( &control, &data, &status, n, x_l, x_u,
                           "sparse_by_rows", ne, NULL, H_col, H_ptr );
                dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                                   ne, fun, grad, hess, hessprod, prec );
                break;
            case 3: // dense
                st = 'D';
                dgo_import( &control, &data, &status, n, x_l, x_u,
                           "dense", ne, NULL, NULL, NULL );
                dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                                   ne, fun, grad, hess_dense, hessprod, prec );
                break;
            case 4: // diagonal
                st = 'I';
                dgo_import( &control, &data, &status, n, x_l, x_u,
                           "diagonal", ne, NULL, NULL, NULL );
                dgo_solve_with_mat( &data, &userdata, &status, n, x, g,
                                   ne, fun_diag, grad_diag, hess_diag,
                                   hessprod_diag, prec );
                break;
            case 5: // access by products
                st = 'P';
                dgo_import( &control, &data, &status, n, x_l, x_u,
                           "absent", ne, NULL, NULL, NULL );
                dgo_solve_without_mat( &data, &userdata, &status, n, x, g,
                                      fun, grad, hessprod, shessprod, prec );
                break;
        }
        // Record solution information
        dgo_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i evaluations. Optimal objective value = %5.2f"
                  " status = %1i\n", st, inform.f_eval, inform.obj, inform.status);
        }else if(inform.status == -18){
            printf("%c:%6i evaluations. Best objective value = %5.2f"
                  " status = %1i\n", st, inform.f_eval, inform.obj, inform.status);
        }else{
            printf("%c: DGO_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for(int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
    }
}

```

```

    //for(int i = 0; i < n; i++) printf("%f ", g[i]);
    //printf("\n");
    // Delete internal workspace
    dgo_terminate( &data, &control, &inform );
}
printf("\n tests reverse-communication options\n\n");
// reverse-communication input/output
int eval_status, nnz_u, nnz_v;
double f = 0.0;
double u[n], v[n];
int index_nz_u[n], index_nz_v[n];
double H_val[ne], H_dense[n*(n+1)/2], H_diag[n];

for(int d=1; d <= 5; d++){
    // Initialize DGO
    dgo_initialize( &data, &control, &status );
    // Set user-defined control options
    control.f_indexing = true; // Fortran sparse matrix indexing
    control.maxit = 2500;
    // control.trb_control.maxit = 100;
    //control.print_level = 1;
    // Start from 0
    double x[] = {0,0,0};
    switch(d){
        case 1: // sparse co-ordinate storage
            st = 'C';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "coordinate", ne, H_row, H_col, NULL );
            while(true){ // reverse-communication loop
                dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                    n, x, f, g, ne, H_val, u, v );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess( n, ne, x, H_val, &userdata );
                }else if(status == 5){ // evaluate Hv product
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else if(status == 23){ // evaluate f and g
                    eval_status = fun( n, x, &f, &userdata );
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 25){ // evaluate f and Hv product
                    eval_status = fun( n, x, &f, &userdata );
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else if(status == 35){ // evaluate g and Hv product
                    eval_status = grad( n, x, g, &userdata );
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else if(status == 235){ // evaluate f, g and Hv product
                    eval_status = fun( n, x, &f, &userdata );
                    eval_status = grad( n, x, g, &userdata );
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else{
                    printf(" the value %li of status should not occur\n",
                        status );
                    break;
                }
            }
            break;
        case 2: // sparse by rows
            st = 'R';
            dgo_import( &control, &data, &status, n, x_l, x_u,
                "sparse_by_rows", ne, NULL, H_col, H_ptr );
            while(true){ // reverse-communication loop
                dgo_solve_reverse_with_mat( &data, &status, &eval_status,
                    n, x, f, g, ne, H_val, u, v );
                if(status == 0){ // successful termination
                    break;
                }else if(status < 0){ // error exit
                    break;
                }else if(status == 2){ // evaluate f
                    eval_status = fun( n, x, &f, &userdata );
                }else if(status == 3){ // evaluate g
                    eval_status = grad( n, x, g, &userdata );
                }else if(status == 4){ // evaluate H
                    eval_status = hess( n, ne, x, H_val, &userdata );
                }else if(status == 5){ // evaluate Hv product
                    eval_status = hessprod( n, x, u, v, false, &userdata );
                }else if(status == 6){ // evaluate the product with P
                    eval_status = prec( n, x, u, v, &userdata );
                }else if(status == 23){ // evaluate f and g

```



```

        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
    }else if(status == 25){ // evaluate f and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else if(status == 35){ // evaluate g and Hv product
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else if(status == 235){ // evaluate f, g and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else{
        printf(" the value %li of status should not occur\n",
            status);
        break;
    }
}
break;
case 3: // dense
st = 'D';
dgo_import( &control, &data, &status, n, x_l, x_u,
    "dense", ne, NULL, NULL, NULL );
while(true){ // reverse-communication loop
    dgo_solve_reverse_with_mat( &data, &status, &eval_status,
        n, x, f, g, n*(n+1)/2,
        H_dense, u, v );
    if(status == 0){ // successful termination
        break;
    }else if(status < 0){ // error exit
        break;
    }else if(status == 2){ // evaluate f
        eval_status = fun( n, x, &f, &userdata );
    }else if(status == 3){ // evaluate g
        eval_status = grad( n, x, g, &userdata );
    }else if(status == 4){ // evaluate H
        eval_status = hess_dense( n, n*(n+1)/2, x, H_dense,
            &userdata );
    }else if(status == 5){ // evaluate Hv product
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else if(status == 6){ // evaluate the product with P
        eval_status = prec( n, x, u, v, &userdata );
    }else if(status == 23){ // evaluate f and g
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
    }else if(status == 25){ // evaluate f and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else if(status == 35){ // evaluate g and Hv product
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else if(status == 235){ // evaluate f, g and Hv product
        eval_status = fun( n, x, &f, &userdata );
        eval_status = grad( n, x, g, &userdata );
        eval_status = hessprod( n, x, u, v, false, &userdata );
    }else{
        printf(" the value %li of status should not occur\n",
            status);
        break;
    }
}
break;
case 4: // diagonal
st = 'I';
dgo_import( &control, &data, &status, n, x_l, x_u,
    "diagonal", ne, NULL, NULL, NULL );
while(true){ // reverse-communication loop
    dgo_solve_reverse_with_mat( &data, &status, &eval_status,
        n, x, f, g, n, H_diag, u, v );
    if(status == 0){ // successful termination
        break;
    }else if(status < 0){ // error exit
        break;
    }else if(status == 2){ // evaluate f
        eval_status = fun_diag( n, x, &f, &userdata );
    }else if(status == 3){ // evaluate g
        eval_status = grad_diag( n, x, g, &userdata );
    }else if(status == 4){ // evaluate H
        eval_status = hess_diag( n, n, x, H_diag, &userdata );
    }else if(status == 5){ // evaluate Hv product
        eval_status = hessprod_diag( n, x, u, v, false,
            &userdata );
    }else if(status == 6){ // evaluate the product with P
        eval_status = prec( n, x, u, v, &userdata );
    }else if(status == 23){ // evaluate f and g
        eval_status = fun_diag( n, x, &f, &userdata );
        eval_status = grad_diag( n, x, g, &userdata );
    }
}

```

```

    }else if(status == 25){ // evaluate f and Hv product
        eval_status = fun_diag( n, x, &f, &userdata );
        eval_status = hessprod_diag( n, x, u, v, false,
                                     &userdata );
    }else if(status == 35){ // evaluate g and Hv product
        eval_status = grad_diag( n, x, g, &userdata );
        eval_status = hessprod_diag( n, x, u, v, false,
                                     &userdata );
    }else if(status == 235){ // evaluate f, g and Hv product
        eval_status = fun_diag( n, x, &f, &userdata );
        eval_status = grad_diag( n, x, g, &userdata );
        eval_status = hessprod_diag( n, x, u, v, false,
                                     &userdata );
    }else{
        printf(" the value %li of status should not occur\n",
               status);
        break;
    }
}
break;
case 5: // access by products
    st = 'P';
    dgo_import( &control, &data, &status, n, x_l, x_u,
               "absent", ne, NULL, NULL, NULL );
    nnz_u = 0;
    while(true){ // reverse-communication loop
        dgo_solve_reverse_without_mat( &data, &status, &eval_status,
                                     n, x, f, g, u, v, index_nz_v,
                                     &nnz_v, index_nz_u, nnz_u );

        if(status == 0){ // successful termination
            break;
        }else if(status < 0){ // error exit
            break;
        }else if(status == 2){ // evaluate f
            eval_status = fun( n, x, &f, &userdata );
        }else if(status == 3){ // evaluate g
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 5){ // evaluate Hv product
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 6){ // evaluate the product with P
            eval_status = prec( n, x, u, v, &userdata );
        }else if(status == 7){ // evaluate sparse Hess-vect product
            eval_status = shessprod( n, x, nnz_v, index_nz_v, v,
                                   &nnz_u, index_nz_u, u,
                                   false, &userdata );
        }else if(status == 23){ // evaluate f and g
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
        }else if(status == 25){ // evaluate f and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 35){ // evaluate g and Hv product
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else if(status == 235){ // evaluate f, g and Hv product
            eval_status = fun( n, x, &f, &userdata );
            eval_status = grad( n, x, g, &userdata );
            eval_status = hessprod( n, x, u, v, false, &userdata );
        }else{
            printf(" the value %li of status should not occur\n",
                   status);
            break;
        }
    }
    break;
}
// Record solution information
dgo_information( &data, &inform, &status );
if(inform.status == 0){
    printf("%c:%6i evaluations. Optimal objective value = %5.2f"
           " status = %li\n", st, inform.f_eval, inform.obj, inform.status);
}else if(inform.status == -18){
    printf("%c:%6i evaluations. Best objective value = %5.2f"
           " status = %li\n", st, inform.f_eval, inform.obj, inform.status);
}else{
    printf("%c: DGO_solve exit status = %li\n", st, inform.status);
}
//printf("x: ");
//for(int i = 0; i < n; i++) printf("%f ", x[i]);
//printf("\n");
//printf("gradient: ");
//for(int i = 0; i < n; i++) printf("%f ", g[i]);
//printf("\n");
// Delete internal workspace
dgo_terminate( &data, &control, &inform );
}
}

```

```

// Objective function
int fun( int n,
        const double x[],
        double *f,
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    *f = pow(x[0] + x[2] + p, 2) + pow(x[1] + x[2], 2) + mag * cos(freq*x[0])
        + x[0] + x[1] + x[2];
    return 0;
}

// Gradient of the objective
int grad( int n,
        const double x[],
        double g[],
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    g[0] = 2.0 * ( x[0] + x[2] + p ) - mag * freq * sin(freq*x[0]) + 1;
    g[1] = 2.0 * ( x[1] + x[2] ) + 1;
    g[2] = 2.0 * ( x[0] + x[2] + p ) + 2.0 * ( x[1] + x[2] ) + 1;
    return 0;
}

// Hessian of the objective
int hess( int n,
        int ne,
        const double x[],
        double hval[],
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;

    hval[0] = 2.0 - mag * freq * freq * cos(freq*x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 4.0;
    return 0;
}

// Dense Hessian
int hess_dense( int n,
        int ne,
        const double x[],
        double hval[],
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;

    hval[0] = 2.0 - mag * freq * freq * cos(freq*x[0]);
    hval[1] = 0.0;
    hval[2] = 2.0;
    hval[3] = 2.0;
    hval[4] = 2.0;
    hval[5] = 4.0;
    return 0;
}

// Hessian-vector product
int hessprod( int n,
        const double x[],
        double u[],
        const double v[],
        bool got_h,
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;

    u[0] = u[0] + 2.0 * ( v[0] + v[2] )
        - mag * freq * freq * cos(freq*x[0]) * v[0];
    u[1] = u[1] + 2.0 * ( v[1] + v[2] );
    u[2] = u[2] + 2.0 * ( v[0] + v[1] + 2.0 * v[2] );
    return 0;
}

// Sparse Hessian-vector product
int shessprod( int n,
        const double x[],
        int nnz_v,
        const int index_nz_v[],
        const double v[],
        int *nnz_u,
        int index_nz_u[],

```

```

        double u[],
        bool got_h,
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    double p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for(int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 1:
                p[0] = p[0] + 2.0 * v[0]
                    - mag * freq * freq * cos(freq*x[0]) * v[0];
                used[0] = true;
                p[2] = p[2] + 2.0 * v[0];
                used[2] = true;
                break;
            case 2:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                p[2] = p[2] + 2.0 * v[1];
                used[2] = true;
                break;
            case 3:
                p[0] = p[0] + 2.0 * v[2];
                used[0] = true;
                p[1] = p[1] + 2.0 * v[2];
                used[1] = true;
                p[2] = p[2] + 4.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for(int j = 0; j < 3; j++){
        if(used[j]){
            u[j] = p[j];
            *nnz_u = *nnz_u + 1;
            index_nz_u[*nnz_u-1] = j+1;
        }
    }
    return 0;
}
// Apply preconditioner
int prec( int n,
        const double x[],
        double u[],
        const double v[],
        const void *userdata ){
    u[0] = 0.5 * v[0];
    u[1] = 0.5 * v[1];
    u[2] = 0.25 * v[2];
    return 0;
}
// Objective function
int fun_diag( int n,
        const double x[],
        double *f,
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    *f = pow(x[2] + p, 2) + pow(x[1], 2) + mag * cos(freq*x[0])
        + x[0] + x[1] + x[2];
    return 0;
}
// Gradient of the objective
int grad_diag( int n,
        const double x[],
        double g[],
        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double p = myuserdata->p;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    g[0] = -mag * freq * sin(freq*x[0]) + 1;
    g[1] = 2.0 * x[1] + 1;
    g[2] = 2.0 * ( x[2] + p ) + 1;
    return 0;
}
// Hessian of the objective
int hess_diag( int n,
        int ne,
        const double x[],
        double hval[],

```

```

        const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    hval[0] = -mag * freq * freq * cos(freq*x[0]);
    hval[1] = 2.0;
    hval[2] = 2.0;
    return 0;
}
// Hessian-vector product
int hessprod_diag( int n,
                   const double x[],
                   double u[],
                   const double v[],
                   bool got_h,
                   const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    u[0] = u[0] + -mag * freq * freq * cos(freq*x[0]) * v[0];
    u[1] = u[1] + 2.0 * v[1];
    u[2] = u[2] + 2.0 * v[2];
    return 0;
}
// Sparse Hessian-vector product
int shessprod_diag( int n,
                   const double x[],
                   int nnz_v,
                   const int index_nz_v[],
                   const double v[],
                   int *nnz_u,
                   int index_nz_u[],
                   double u[],
                   bool got_h,
                   const void *userdata ){
    struct userdata_type *myuserdata = (struct userdata_type *) userdata;
    double freq = myuserdata->freq;
    double mag = myuserdata->mag;
    double p[] = {0., 0., 0.};
    bool used[] = {false, false, false};
    for(int i = 0; i < nnz_v; i++){
        int j = index_nz_v[i];
        switch(j){
            case 1:
                p[0] = p[0] - mag * freq * freq * cos(freq*x[0]) * v[0];
                used[0] = true;
                break;
            case 2:
                p[1] = p[1] + 2.0 * v[1];
                used[1] = true;
                break;
            case 3:
                p[2] = p[2] + 2.0 * v[2];
                used[2] = true;
                break;
        }
    }
    *nnz_u = 0;
    for(int j = 0; j < 3; j++){
        if(used[j]){
            u[j] = p[j];
            *nnz_u = *nnz_u + 1;
            index_nz_u[*nnz_u-1] = j+1;
        }
    }
    return 0;
}

```


Index

- dgo.h, [7](#)
 - dgo_import, [11](#)
 - dgo_information, [24](#)
 - dgo_initialize, [10](#)
 - dgo_read_specfile, [10](#)
 - dgo_reset_control, [12](#)
 - dgo_solve_reverse_with_mat, [17](#)
 - dgo_solve_reverse_without_mat, [21](#)
 - dgo_solve_with_mat, [13](#)
 - dgo_solve_without_mat, [15](#)
 - dgo_terminate, [24](#)
- dgo_control_type, [8](#)
- dgo_import
 - dgo.h, [11](#)
- dgo_inform_type, [9](#)
- dgo_information
 - dgo.h, [24](#)
- dgo_initialize
 - dgo.h, [10](#)
- dgo_read_specfile
 - dgo.h, [10](#)
- dgo_reset_control
 - dgo.h, [12](#)
- dgo_solve_reverse_with_mat
 - dgo.h, [17](#)
- dgo_solve_reverse_without_mat
 - dgo.h, [21](#)
- dgo_solve_with_mat
 - dgo.h, [13](#)
- dgo_solve_without_mat
 - dgo.h, [15](#)
- dgo_terminate
 - dgo.h, [24](#)
- dgo_time_type, [9](#)