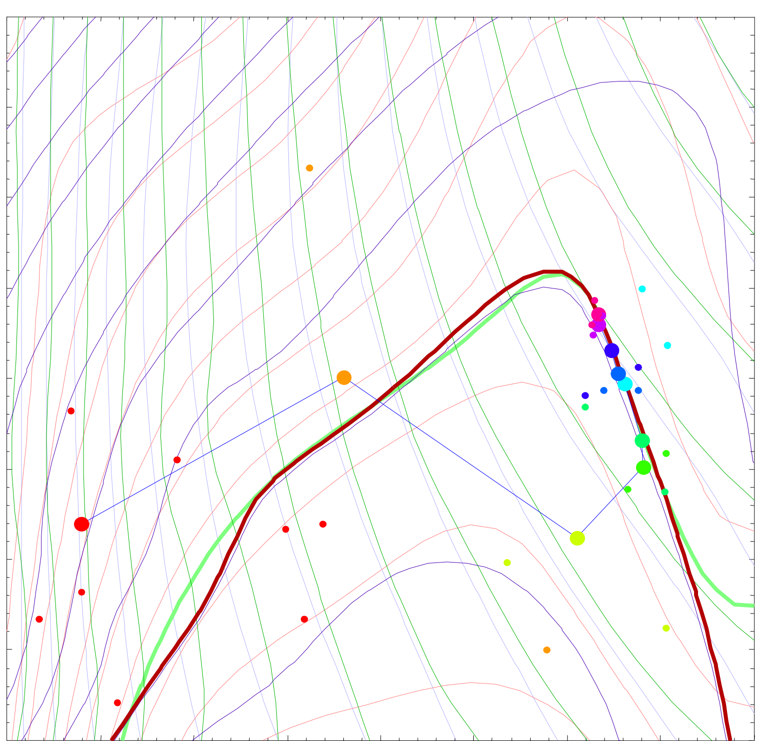
IOptLib User's Manual

Revision 1 (pre-release), 2009.

Igor Grešovnik



Contents:

1 Introduction ... 1

2 Basics ... 4

2.1 Preliminaries ... 4

2.2 Availability and Contents ... 4

2.3 Basic Data Types and Function prototypes ... 4

2.3.1 Standard analysis function ... 5

2.3.2 Tools and templates for implementing analysis functions ... 10

2.3.3 Standard vector function ... 12

2.4 Conversion between standard analysis and standard vector function ... 12

2.4.2 Remarks on double conversion (forth & back) ... 15

2.5 Numerical differentiation of analysis results ... 16

2.5.1 User instructions ... 16

2.5.2 Example for developers: implementation of numerical differentiation ... 20

2.5.3 Extension of the numerical differentiation system ... 23

2.6 Prevention of repeated analysis at the same parameters and analysis counts ... 24

3 Basic Building Blocks ... 27

3.1 Co-ordinate transformations ... 28

3.1.1 Linear transformation of co-ordinates ... 29

3.1.2 Linear (Affine) maps, eigendecomposition and quadratic forms ... 31

3.1.3 Agreements for use of linear (affine) maps ... 34

3.1.4 Implementation of linear and affine maps ... 38

3.1.5 Restricted step constraints ... 44

3.1.6 Restricted region constraints - old implementation ... 47

4 Modification and transformation of optimization problems ... 47

4.1 Combining objectives and constraints defined by different analysis functions ... 48

4.2 Handling of bound constraints ... 51

4.2.1 Combination of discontinuous penalty functions and transformation of co-ordinates ... 53

4.2.2 Implementation remarks on penalty terms and bound constraints ... 58

5 Building Blocks for Successive Approximations ... 66

5.1 Introduction ... 66

5.1.1 Overview of generic utilities from top to bottom ... 67

5.1.2 Basic scheme for use of approximations of analysis response 68
5.1.3 Basic approximation data types 68

5.2 Implementation notes 69
5.2.1 Specific and common auxiliary data structures 69
5.2.2 Approximation data updating functions 69

5.3 Approximation utilities – To implement 71
5.3.1 Things not yet implemented 71
5.3.2 Efficiency issues 71
5.3.3 Weighting functions 81

6 Optimization algorithms 82

7 Testing System 82

7.1 Registering an optimization problem or test case 82

8 Appendix: Common types and related modules 95

8.1 Introduction 95
8.1.1 Comments on ANSI C 95
8.1.2 Followed programming rules 102
8.1.3 Work in multi-thread environment 103

8.2 Vector and Matrix Operations 106
8.2.1 Allocation and access to elements: 106
8.2.2 Reallocation and deletion: 107
8.2.3 Copying and other operations: 108
8.2.4 Binary operations: 109

8.3 Stack Operations 110
8.3.1 Creation, deletion, resizing and copying 111
8.3.2 Element access 114
8.3.3 Other operations 116
8.3.4 Index tables 117

8.4 Error reporting 118

8.5 Other Libraries 119

9 Future plans 119

10 To do (for developers) 120

10.1 Test utilities 120

10.2 Questions to answer 120

10.3 Implementation plans 120
10.3.1 Prevention of successive repetition of analyses 120
10.3.2 Successive approximations building blocks 121
10.3.3 Miscellaneous 122

11 Sandbox ... 125

11.1 Storage of chapters that are in the process of revision ... 126

11.1.1 Agreements for use of linear (affine) maps ... 126

IOptLib is a freeware and comes with no warranty.

1 INTRODUCTION

IOptLib (an abbreviation for Investigative Optimization Library) is an open optimization library designed to sustain development and testing of algorithms for solving practical optimization problems. The library is implemented in *ANSI C* but should be easy to make interfaces for use with other programming languages such as C++, Pascal and Fortran. A priority goal is to develop algorithms suited to problems with computationally expensive and possibly noisy evaluation of the response (i.e. objective and constraint) functions. The library is intended to provide modular building blocks for constructing such algorithms, standardized templates for interfacing tools obtained form other libraries, and testing environment where different performance aspects of algorithms can be readily extensively tested during and after the development stage. Currently most of the efforts are devoted to algorithms based on successive solution of approximated problems obtained by local sampling and approximation of the response functions. Such algorithms have complex designs and involve solution of many sub-problems such as non-linear or quadratic programming problems, matrix algebra, optimal sampling strategies, etc. The intention is therefore to gradually accumulate efficient routines for solving these problems, which will lead to broader serviceability of the library. Any attempt was made to keep open the possibility of starting development of new algorithms or attaching to the existent functionality at any level. The basic library is therefore intended to be distributed as free open source under certain conditions. A couple of algorithms will be available under different negotiable terms since this is necessary to provide the funding for library development, however their building blocks together with a set of quite useful algorithms will be provided with the basic set that is more open what concerns availability.

The original motivation for the library was obtained in optimization of forming processes where evaluation of objective and constraint functions typically involves complex numerical simulation with hundreds of thousands of degrees of freedom, very non-linear and path dependent materials, multi-physics and multi-scale phenomena etc. As result, not only the calculation of the objective and constraint functions takes very long times even on the fastest computers or parallel architectures, but these functions often contain substantial amount of numerical noise. These conditions impose a substantial turn in how algorithm performance is viewed. On one hand the most important measure of algorithm efficiency becomes the number of function evaluations it takes for calculating optimum up to a given accuracy. The CPU time spent by the algorithm becomes somehow less important because function evaluations will normally require incomparably more computational time. Because running optimization procedures will often be just on the limit of affordable, the goal will not always be to find an optimal solution up to a specified accuracy, but rather to achieve significant improvement within an affordable computational time.

The targeted scope of the library is beyond the area of its original motivation. It is intended to provide a pool of algorithms for different problems and facilities for extending this pool. Beside that, interaction with other libraries and use of the library in existing or future software is accounted for as much as possible. A lot of stress is put on defining standard data types and function prototypes used for different purposes, such as evaluation of response functions and their

derivatives or for storing results of such evaluation and their use in building approximations. These standards are defined in such a way that routines for similar tasks from other libraries can be easily incorporated in the system, and functions that are consistent with library standards can be easily exported in standard forms required in other software environments. Wrapping functions and data converters are provided some common cases, and the way how one can create own tools for this is described in this manual. These standards are defined in such a way that routines for similar tasks from other libraries can be easily incorporated in the system, and functions that are consistent with library standards can be easily exported in standard forms required in other software environments. This part of library design is described in Subsection 2.3.1.

The entire Section 2 contains a short overview of the library. This begins with availability information and informative overview of library contents in Section 2.2.

The library comes with a set of basic utilities that are extensively used in implementation of basic building blocks and algorithms. This includes e.g. basic matrix and vector operations and generic implementation of data containers such as stacks. These utilities are well documented in source code, and Subsection 2.3 provides some information for easier navigation. Various sets of building blocks developed for construction of algorithms are described in Section 3 and a short overview of the algorithms provided with the library is given in Section 6.

Many of elementary utilities make use of other free libraries, which are listed in Section 9.5. Contribution of people who designed and implemented these libraries and made them available is gratefully acknowledged.

Comment [a1]: Change this notice when a closed form of the manual is achieved!

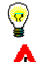
This manual is **incomplete** and has currently some true gaps that could alone render correct usage of the library impossible. However, these gaps should be easily overcome by a **look at the source code**. This is especially true because source code is relatively well documented, in particular each important function has an introductory comment that specifies the meaning of its arguments and what the function does.

Of course, you will sometimes need appropriate tools to search for function definitions and type declarations. Integrated development environments are ideal for such tasks, but file browsing and searching utilities that are nowadays provided by every reasonable operating system will also do the job satisfactory.

Legend of graphic symbols:

- this section / paragraph / text is not yet complete. Since nothing can be considered definite or complete at a library like "IOptLib", this sign will denote portions of this

manual where more content is intended at this very moment, but there was no time to add it.



- Developers' section – these contents will be of more interest for developers of the library than its users. To define the terms – users of the library "IOptLib" will usually be developers of some other software. In this document, term "developers" is used for those who contribute to the library itself, i.e. people who add functionality and make it publicly available, who suggest conceptual changes or who contribute free additional documentation for the benefit of other users and developers.

- Consideration.

- Warning.

2 BASICS

2.1 Preliminaries

Primary subject of this library are tools for solution of nonlinear optimization (non-linear programming or NLP) problems, which can be formulated as

$$\begin{array}{lll} \textit{minimise} & f(\mathbf{x}), & \mathbf{x} \in \mathbb{R}^n \\ \textit{subjected to} & c_i(\mathbf{x}) = 0, & i \in E \\ \textit{and} & c_j(\mathbf{x}) \leq 0, & j \in I \end{array} \qquad (1)$$

x is the vector of optimization (or design) parameters. Function f is called the *objective function* (merit function, fitness function, cost function and other names are also in use) and c_i and c_j are constraint functions that define the *feasible domain*, i.e. the set of admissible points in the design space. E is index set defining the set of equality constraints and I is the set defining *inequality constraints*. Objective and constraint functions are collectively referred to as *response functions* or simply response[1].

Calculation of the objective and constraint functions will be referred to as *direct analysis* or simply *analysis*.

2.2 Availability and Contents

2.3 Basic Data Types and Function prototypes

[1] The term *numerical response* will be used sometimes to emphasize that the response functions are calculated by a numerical simulation. Similarly, the term *numerical analysis* will sometimes be used for analysis.

2.3.1 Standard analysis function

By the term *analysis* we mean calculation of the response (i.e. objective and constraint functions) from (1), which define the optimization problem.

In the source code of optimization programs or libraries, there are usually one or *more analysis functions* for calculating the response. Optimization functions, which implement optimization algorithms for solution of (1), iteratively call the analysis functions to evaluate the response at different parameters, until convergence is achieved. Usually the analysis functions are passed as argument to the optimization functions, therefore each implementation of some optimization algorithm requires a specific *type of analysis functions*[1]. If we want to connect optimization algorithms implemented in some library with analysis functions of incompatible type, we must first implement a suitable interface by defining a wrapping functions, which are of compatible type and call the original analysis functions to evaluate the results.

In order to make interfacing between different libraries and software and development of building blocks as easy as possible, the library makes use of *standard analysis function type*. It is declared as

```
typedef
  int (*analysis_bas_f) (
      vector param,int *calcobj,double **addrobj,
      int *calcconstr,stack *addrconstr,
      int *calcgradobj,vector *addrgradobj,
      int *calcgradconstr,stack *addrgradconstr,void *cd);
```

This definition is enough general and suitable for many special cases, e.g. where constraints can be calculated separately or not from the objective function, or where calculation of response gradients represent considerable or only minimal additional effort with respect to sole calculation of values. Since practically every library user will have to define analysis functions of this type in order to use library functionality, a detailed description is given below. However, *knowledge of all the rules* described below *is not really necessary* since library users can help themselves by some tools prepared to aid defining analysis functions and use existing examples as templates. This is described in subsection 2.3.2.

Table 1: Meaning, types and dimension of arguments of the *standard analysis functions*. (type *analysis_bas_f*). In the table below, integer numbers *numparam* , *numconstraints* and *numobjectives* denote number of parameters, number of constraints and number of objective functions (only 0 or 1 are possible), respectively.

Argument	Meaning	Remarks
vector param	Vector of design parameters.	In general, it must be allocated with correct dimension, i.e. *numparam*.
Flag pointers	Input/output. Define what to evaluate and	Input/output. Pointer to non-zero value means that evaluation is requested, NULL or pointer to 0 means evaluation is not requested.

[1] Type of a function is defined by required types of its arguments and return value.

	inform what has been evaluated.	Output (when evaluation is requested): if evaluation is requested then pointed value is set to 0 if evaluation or return of corresponding results could not be done or if the corresponding response is not defined in the problem corresponding to the analysis function.
int *calcobj	Objective function evaluation.	Requests evaluation of the objective function.
int *calcconstr	Constraint functions evaluation.	Requests evaluation of constraint functions (all in a package).
int *calcgradobj	Evaluation of gradient of the objective function.	Requests evaluation of the gradient of the objective functions.
int *calcgradconstr	Evaluation of gradients of constraint functions.	Requests evaluation of gradients of constraint functions.
Storage addresses	Define address for storage of calculated response	Output. For each type of response there is an argument specifying storage address. Arguments must not be NULL when evaluation of given response is requested (but may be NULL when it is not). Storage is allocated/reallocated by the analysis function when necessary and kept untouched when evaluation of corresponding response is not required. When a given kind of response is requested but it is not defined, the storage would be untouched, but corresponding flag would be set to 0.
double **addrobj	Objective function storage.	**addrobj is set to the value of the objective function. *addrobj is set to NULL when objective function is not defined.
stack *addrconstr	Storage for constraint functions.	Stack holds *numconstr* elements of type double *, which hold values of constraint functions.
vector *addrgradobj	Storage for objective function gradient.	Vector of dimension *numparam*, elementa are components of the objective function gradient.
stack *addrgradconstr	Storage for gradients of constraint functioins.	Stack of *numconstr* elements of type *vector*. Vectors are of dimension *numparam* and hold gradients of individual constraint functions.
Definition data	Additional exchange of information.	Intended for different roles: precise definition of analysis response (e.g. coefficients of quadratic objective functions), may be used for data transfer between the algorithm, analysis and user (state & requests), seamless upgrade of analysis (e.g. non-derivative analysis upgraded by numerical differentiation) etc.
void *cd		Input and/or output, not compulsory. Type and structure of the pointed data is arbitrary, it is interpreted within the analysis function. May be NULL when additional data is not necessary. Caller of the analysis function must know and obey the rules for type and layout of the pointed data, which are defined on the analysis side.
Info mode All flag pointer arguments are NULL	When *calcobj*, *calcconstr*, *calcgradobj* and *calcgradconstr* are all NULL, the analysis function operates in **Info** mode. It does not evaluate anything, but checks all storage address arguments that are different than NULL and allocates or re-allocates the addressed storage if necessary in such a way that all the dimensions of the allocated storage are consistent with the problem defined by the analysis function (e.g. *addrgradconstr* will point to stack with *nconstr* vector elements of dimension *numparam*, provided that there are also constraints in the response).	
Rerutn value (int)		0 if everything is OK, usually a negative error code of the calculation could not be performed correctly.

There are some standard agreements about expected behavior of the standard analysis functions:

The function **returns** an error code, which is 0 if everything is OK, or a negative error code if an error occurs (or at least a non-zero value). Argument `param` defines the vector of parameters

for which the response should be calculated. It is of type `vector`, which is described in section 9.2, together with some basic operations provided for this data type.

Arguments `calcobj`, `calcconstr`, `calcgradobj` and `calcgradconstr` point to flags that define which parts of the response must be calculated and provided via output arguments. They stand for the value of objective function, values of constraint functions, gradient of the objective function and gradients of constraints, respectively. A non-NULL pointer pointing to a non-zero value means that the respective quantity should be calculated while a NULL pointer or a non-NULL pointer pointing to an integer whose value is 0 means that the respective quantity does not need to be calculated at the particular function call. If evaluation of some quantity is requested but it could not be calculated then the function should set the corresponding flag to 0, indicating disability to calculate the particular quantity. This is why the request flags are passed as integer pointers rather than just integers – in this way return information on whether the requested information could actually be provided can be passed back to the caller. The agreement is that if, for some problem, a given quantity is not defined (e.g. constraint functions in the case of unconstrained minimization problem) but is requested with the respect to the state of the corresponding flag (e.g. calcconstraints) then the function should also set that flag to 0.

Each flag pointer argument is followed by the appropriate *address of storage* that must be provided by the caller to store results of evaluation. The *analysis function itself must allocate or reallocate space for storing results whenever necessary*, but there is an error if something should be calculated but the *address* of the appropriate storage is not provided (i.e. the corresponding argument is NULL), and the function should report such errors via the error reporting mechanism (section 9.4).

There is a rule that the *analysis function should not allocate or reallocate any storage it does not actually need* (according to the evaluation requests specified by flag pointers). There are a number of reasons for this, one of them is preventing unnecessary consumption of CPU time and memory resources. Another reason is provision of firm logical rules of function behavior for its callers. For example, when derivative information is not necessary, the caller can call the analysis function with storage address for gradients set to NULL without worrying that this will call exceptions or breakage of program behavior. The rule is thus logical – why should one bother with gradient storage when gradients are not at all requested?

The argument `addrobj` defines the storage address for the objective function value. If the *evaluation of the objective function is not requested* (i.e. the argument `calcobj` is NULL or `*calcobj` is 0) or the objective function is not defined for a given problem then this argument may be NULL, and in any case the analysis function should not do anything with the argument or the data it points to. When *evaluation of the objective function is requested*, `addrobj` must be a valid non-NULL pointer whose value is the address of a pointer to a data unit of type double. The pointer pointed to by `addrobj` may be NULL, however. In this case it is expected that the analysis function will dynamically allocate data storage for data piece of type double and set a pointer pointed to by `addrobj` to the address of the allocated storage. In principle, `addrobj` may also be address of a pointer that points to a static variable of type double, because in this case no allocation or re-allocation would be made. However, it is preferable that `**addrobj` is *dynamically allocated*, in order to prevent troubles with inadvertent implementations of analyses functions that do not strictly obey the standards. The C code that will do the job within the analysis function may look like this:

```
if (calcobj!=NULL) if (*calcobj)  /* evaluation requested */
```

```
{
  if (addrobj!=NULL)
  {
    if (*addrobj==NULL)
      *addrobj=calloc(1,sizeof(**addrobj));
    ... /* 1: perform evaluation */
    **addrobj=... /* 2: store calculated objective function */
  } else
  {
    /* forbidden situation - evaluation requested but storage address not
            provided */
    *calcobj=0;
    … /* launch an error report */
  }
}
```

In practice, most of the described housekeeping operations will be performed by a pre-defined utility function (subsection 2.3.2) and the library user who creates the analysis function will only take care of evaluation and storage steps, denoted by "1:' and "2:" in the above code.

The constraint function values are stored on a `stack` (type described in Section 9.3) pointed to by `addrconstr`, as pointers of type *double* *. Similar rules as for `addrobj` apply, except that stacks are more complex structured data types for which given rules for data access, allocation and re-allocation apply. Gradient of the objective function is stored in a `vector` (Section 9.2) pointed to by `addrgradobj`, and gradients of the constraint functions are stored as pointers of type *vector* on the stack pointed to by `addrgradconstr`.

In accordance with the above described rules, vector `*addrgradobj` is *allocated or re-allocated* by the analysis function *whenever necessary and only when necessary*. This is the case when evaluation of the objective function gradient is requested *and* **gradobj* is either NULL or is allocated but with wrong dimension.

Similar rule holds for `*addrconstr`, except that not only the stack itself is allocated or reallocated when necessary, but this is also valid for its elements, which must be pointers to double. Therefore, if `*addrconstr` points to a stack with more elements than there are constraint functions, the stack dimension will be reduced and redundant elements will be de-allocated . If the stack is not allocated or has smaller number of elements than there are constraint functions, it will be allocated (or re-allocated) together with missing elements. Of course, this will be done only when necessary, i.e. when the evaluation of constraint functions is requested (which means when `calcconstr` points to an non-zero integer *and* constraint functions are actually defined by a given analysis, i.e. the number of constraints is larger than 0).

Similar rules apply for `*addrgradconstr`, except that this stack holds *vector elements*. These have themselves a variable number of elements (dimension), therefore elements are not simply allocated or de-allocated, but also their dimension must be adjusted.

Argument `cd` (whose name may be interpreted as "*client data*"[1]) is additional argument that does not carry standard input or output analysis data, but is intended to *hold additional data that precisely defines the analysis*. It may be NULL when no extra definition data is required and analysis is exactly determined by its implementation in terms of analysis function. This argument is a pointer to the data of indefinite type (void *). It is on the analysis function to interpret the data (and thus assign an internal type to the pointer) and it is on the caller to pass the pointer that points to the data of expected type and structure.

The `cd` argument may be used for more complex data exchange which was not anticipated for standard library utilities or optimization algorithms, therefore it may be used also for arbitrarily exotic extensions of functionality such as for establishing complex communication protocols between optimization algorithms, numerical analyses and end users of the software. Through this concept, it is easy to provide very customized functionality that is intended for special situation, and this can be done in such a way that implementation can still be used in a standard way, without those extra fancy additions. Example of use of the `cd` argument to provide numerical differentiation of the analysis response is provided in Section 2.5.

As a simple ***example***, we may define the analysis function that represents unconstrained minimization problem with quadratic objective function. In order to exactly define the optimization problem, we need additional information, i.e. coefficients of the quadratic function (since the analysis function is intended for *any* quadratic function, not only for some particular function with coefficients known in advance). The analysis function may be designed in such a way that coefficients must be arranged into a vector in a specific order. Therefore, the parameter `cd` may simply be a vector of the appropriate dimension.

Info mode:

When all flag pointer arguments (*calcobj*, *calcconstr*, *calcgradobj* and *calcgradconstr*) are NULL, the analysis functions operates in *info mode*. This means that nothing is evaluated, but the data storage with corresponding address arguments different than NULL is allocated or re-allocated (if necessary) with the appropriate dimensions. This can be used to establish the number of constraints and whether constraints and/or objective function are defined at all for a given problem.

Warnings:

Usage of result storage: When calling the analysis function, the admissible state of the data used for storage of results is relatively free. The analysis function will do the necessary allocation or re-allocation by itself. However, there are some restrictions. Shortly speaking, automatism can only be expected when correct operation is possible. Whenever data pointers addressed by storage address arguments are not NULL, they must point to the data of correct type. For example, `addrgradconstr` (when not NULL) must either point to a NULL pointer or to an allocated `stack` pointer. In the latter case, the stack may have an incorrect number of elements, but all of them must be of type `vector`. Vector elements of that stack may be of incorrect dimensions.

[1] In order to explain the term *client data*, imagine that we have an stand-alone optimization package and an independent software package for direct analyses (which may, for example, include tools for finite element numerical simulation of some process). The analysis package may be implemented as a server that serves requests of optimization package. Optimization package acts as a client to the analysis package and sends requests by calling functions consistent with the standard analysis type. Beside the standard input/output data, the optimization software can pass a pointer *cd*, which is set by the client data in order to pass to the server additional information about what the client wants, i.e. what type of analysis should be performed. Of course, the type and structure of the data passed must be agreed in advance by both software packages.

However, either when the number of stack elements is incorrect (i.e. not equal to the number of constraint functions) or the dimensions of some of its elements are inconsistent (i.e. not equal to the number of optimization parameters), the stack and its elements must be dynamically allocated and their pointers must be the actual access handles. Only in this way eventual re-allocation may be done harmlessly. Once again, storage allocation will usually not done explicitly by the designers of analysis functions, but functions provided by the library will be used instead (subsection 2.3.2).

2.3.2 Tools and templates for implementing analysis functions

For many users it may be a true nuisance to implement analyses functions according to the library standards, while adhering to the above mentioned rules enables a high level of flexibility when using the library or implementing new tools. Therefore, the user can use the provided pre-defined tools that do a large part of the job, and thus concentrate only on implementing the procedures for calculation of the analysis response.

When implementing an analysis function for a given class of optimization problems, it is recommendable to start form simple examples provided in the library source code. The function `testanfunc` that is found in `optbas.c` is provided especially for this purpose. The function defines a simple optimization problem with two design parameters and two constraints, and therefore it features most of what average users will ever need to take care of when implementing such functions. In order to implement a new analysis function, one can copy this template and just replace those parts of code where response is evaluated and stored. For storage of the response, one needs to know some basic things about the vector and stack types, which are outlined in Section 9 (Subsections 9.2 and 9.3).

In order to *allocate the space for storage of results*, check and report *inconsistency in arguments* and ensure proper operation of the analysis function in *info mode* (this is the case when the analysis function is called with all flag pointers NULL, see Sub-section 2.3.1), we should use the function **`prepanfuncbas`**, which is declared as follows:

```
int prepanfuncbas(vector param, int *calcobj, double **addrobj, int
            *calcconstr, stack *addrconstr, int *calcgradobj, vector
            *addrgradobj, int *calcgradconstr,stack *addrgradconstr,void
            *clientdata, int nparam,int nobj,int nconstr,int derobj,int char
            *funcname,char *filename,int fileline);
```

This function is called within of a standard analysis function (of type `analysis_bas_f`, see Sub-section 2.3.1). First group of arguments are the same as for the analysis function (names of these arguments are listed in the above declaration with the same names as in the description of the standard analysis function) and arguments of the analysis function must be passed literally in their place. The next set of arguments define information that is specific for the problem implemented by the analysis function, and these information must be provided and passed within the analysis function:

nparam is the number of parameters of the problem. If the problem is defined for a fixed number of parameters, this number must be passed, otherwise the actual number of parameters (defined as dimension of param) or 0 may be passed.

nobj must be 1 if the objective function is defined for the problem, or 0 if it is not.

nconstr must be the number of constraint functions.

Arguments derobj and derconstr specify whether derivatives (gradients) of the objective functions and constraint functions, respectively, can be calculated by the analysis function. 0 must be passed if the corresponding derivatives can not be calculated, or 1 if they can.

Other arguments provide the necessary information for reporting errors: funcname should be the name of the analysis function in which prepanfuncbas is called, and filename and fileline should be the name of the source file and line number where the function is called (this information is not vital for function operation, but is necessary for correctness of information provided in eventual error reports). The last two arguments are usually provided through pre-defined compiler macros **__FILE__** and **__LINE__** (note double underscores in macro names).

The function ***returns*** *0 if the analysis should proceed* with evaluation of the requested results, *a negative error code* if an error occurred (mainly this would mean inconsistency of input arguments) or *1 if* the analysis function was called in *info mode*.

Schematically , the function is called within the analysis function in the following way:

```
int ret =0;  /* return value of analysis function */
...   /* other declarations */
...   /* eventual auxiliary code to determine parameters of operation, e.g.
            on basis of the client data and/or other arguments – specific for
            the analysis function */
if ( ! ( ret=prepanfuncbas ( ... , /* arguments of the analysis function */
         <nparam>, <nobj>, <nconstr>, <derobj>, <derconstr>, <funcname>,
            __FILE__, __LINE__) )  )
{
  ...  /* Evaluation and storage of results; storage space has already been
            allocated by prepanfuncbas */
}
return ret;
```

Arguments that are replaced by "…" are the arguments of the analysis function in which prepanfuncbas is called and are literally copied from the argument block of that analysis function. Arguments in angle brackets (< >) are prepared within the analysis function before the call, and the last arguments are pre-defined compiler macros that are stated literally (during compile time these macros are replaced by constant values that define the name of the source file and the line number where the macro is called). An example of use can be found in the previously mentioned function testanfunc in optbas.c.

2.3.3 Standard vector function

As equivalent to standard analysis functions, there is a type vec_bas_f for calculating vector response in which all parameter dependent functions are equally treated as components of a vector function. Definition is

```
typedef int (*vec_bas_f) (vector param, int *calcval, vector *valaddr, int
           *calcgrad, matrix *gradaddr, void *clientdata);
```

Similar to standard analysis function, this function takes vector of parameters as first arguments, and then pointers to calculation flags followed by corresponding storage addresses. There is no distinction between different individual function, so values of all of them are stored to a vector addressed by `valaddr`. Gradients of components are stored (if requested and if they can be calculated) *by rows*[1] in the Jacobian matrix pointed to by `gradaddr`. The argument `clientdata` is reserved for additional definition data that specifies how the function is calculated (it can contain coefficients etc.).

2.4 Conversion between standard analysis and standard vector function

The standard analysis function (type `analysis_bas_f`, Section 2.3.1) has been designed to fit well the needs of calculating the response defining optimization problems. One of the design features is that the objective function is distinguished (at least in the level of code design) from constraint functions. This suits well the role in the optimization problems of class (1), but for various analysis tasks objective and constraint functions may be treated equally, since from analysis perspective both objective and constraint functions are just functions defined on the same parameter space. Therefore, for some tasks representation of response by the standard vector function (type `vec_bas_f`, Section 2.3.3) where all response functions are treated equally as components of a single vector function, may be more suitable. For these tasks (numerical differentiation described in Section 2.5.1.22.5 is an example) a seamless conversion between both type of functions is provided.

The basis of conversion is the conversion type `analysis_to_vecfunc_cd` (defined in `optbas.h`), which is a pointer to the structure that contains all the data necessary for conversion and also the auxiliary data for storage of intermediate results. This type is intended for conversion in both directions. Basic data it contains are the function pointer (address of the function to be converted), definition data for that function and eventually the function for de-allocation of the definition data, for either type of function that needs to be converted. Conversion is performed

[1] Sometimes it is beneficial to have the matrix of gradients such that gradients are stored by rows rather than by columns. Conversion between the two forms is done simply by transposition, using e.g. function `mattransp0` from `matrixop.c`.

simply by preparing the conversion data object and calling the appropriate conversion function (with the prepared conversion data as its definition data) instead of the original one.

Declaration of the function that converts from the standard analysis (analysis_bas_f) to standard vector (vec_bas_f) form is

```
int vecfunc_froman(vector param, int *calcval, vector *addrval, int
            *calcgrad, matrix *addrgrad, analysis_to_vecfunc_cd cd);
```

The opposite conversion is performed by the function

```
int anfunc_fromvec(vector param, int *calcobj, double **addrobj, int
            *calcconstr, stack *addrconstr, int *calcgradobj, vector
            *addrgradobj, int *calcgradconstr, stack *addrgradconstr,
            analysis_to_vecfunc_cd cd);
```

Example 1 below shows how conversion can be applied in order to use a fictitious vector function `my_vecfunc` for the definition of the objective function and constraints of the optimization problem, and performing optimization on the problem defined in this way. It is assumed that the vector functions takes a matrix of coefficients as its definition data. In order to use this function as definition of the optimization problem and perform optimization, we need to perform conversion to the standard analysis form. All we need to do is to create the conversion object of the type `analysis_to_vecfunc_cd` and set the address of the vector function (`my_vecfunc` in this case) and pointer to the definition data (matrix *coef*) on the conversion object (function for de-allocation of vector function definition data is set to NULL because the data will be de-allocated independently of the conversion data). After that, we can use the analysis function `anfunc_fromvec` with conversion data as definition data. This function calls the appropriate vector function with its definition data (found on the conversion object) and re-arranges the results of the vector function in the returned data of the (converted) analysis function.

Example 1: Conversion of standard analysis to standard vector function.

```
...
matrix coef=NULL;
analysis_to_vecfunc_cd convertcd=NULL;
...  /* Definition of coefficients for the vector function, etc. */
/* Preparation of conversion data: */
convertcd=new analysis_to_vecfunc_cd();
convertcd->vecfunc=my_vecfunc;    /* original vector function */
convertcd->veccd=coef;    /* vector function definition data */
convertcd->dispveccd=NULL;  /* de-allocate definition data elsewhere */
convertcd->numobj=1;  /* The first component of the original vector function
            is treated as the objective function (and the rest as constraint
            functions) */
...
/* Use of the analysis functioin that has been converted from vector
            function, in optimization: */
optimizebas(..., anfunc_fromvec , convertcd);
...  /* Do something with results */
```

```
/* Clean-up: */
dispmatrix(&coef);
disp analysis_to_vecfunc_cd(&convertcd)
```

2.4.1.1 Preparation of conversion data

Currently there are no special functions for preparation of conversion data. However, preparation is simple enough to be done manually. We just need to create the conversion data object of type `analysis_to_vecfunc_cd` and set the fields that define the function that would be converted to another form (either the standard analysis or vector function). See Example 1 for conversion of vector form to analysis form. The opposite conversion is done similarly, except that we need to set fields `anfunc`, `ancd` and `dispancd` instead of `vecfunc`, `veccd` and `dispveccd`, respectively.

2.4.1.2 Definition of conversion type

Definition of the conversion data type (in `optbas.h`) is as follows:

```
typedef struct _analysis_to_vecfunc_cd {
  int type,id;     /* Type and unique ID */
  int numparam,numobj,numconstr,numval;
  vector param;    /* Vector of parameters */
  int calcobj;     /* beginning of data for analysis function: */
  double *obj;
  int calcconstr;
  stack constr;
  int calcgradobj;
  vector gradobj;
  int calcgradconstr;
  stack gradconstr;
  int anret;     /* end of data for analysis function */
  int calcval;
  vector vecval;  /* beginning of data for vector function */
  int calcgrad;
  matrix vecgrad;
  int vecret;      /* end of data for vector function */
  /* Data for performing analyses: */
  analysis_bas_f anfunc;
  void *ancd;
  void (*dispancd)(void **);
  int anblockgrad;  /* inhibit gradient calculation by anfunc */
  /* Data for evaluating vector function: */
  vec_bas_f vecfunc;
  void *veccd;
  void (*dispveccd)(void **);
  int vecblockgrad;    /* inhibit gradient calculation by vecfunc */
  /* Auxiliary data for additional operations such as line search or
  numerical differentiation: */
  int recordan;
  int recordvec;
  stack anpoints;
```

```
    stack anstore;  /* storage for unused analysis points */
    /* void (*dispanpoint) (void **); */
    stack vecpoints;
    stack vecstore;  /* storage for unused vec. func. points */
    /* Auxiliary points for intermediate results: */
    analysispoint auxanpt;
    vecfuncpoint  auxvecpt;
  } *analysis_to_vecfunc_cd;
```

2.4.2 Remarks on double conversion (forth & back)

The same conversion object of type `analysis_to_vecfunc_cd` can be used for two opposite conversions at the same time (i.e. from the standard vector function to standard analysis function and vice-versa). Undesirable interferences in such scenarios are prevented by using distinct auxiliary data for the two opposite types of conversion.

For example, conversion from standard vector to standard analysis form uses auxiliary data fields `vecret`, `calcval`, `calcgrad`, `vecval` and `vecgrad` for intermediate storage of results of the vector function (field (…)->vecfunc). These results are converted to the form convenient for the standard analysis function before copying them to the output arguments of the conversion function (usually `anfunc_fromvec`). For the intermediate storage, the field `(…)->auxanpt` of type `vecfuncpoint` (designed specially for storing results of vector functions) is used. We could have used the fields anret, calcobj, calcconstr, calcgradobj, calcgradconstr, obj, constr, gradobj and gradconstr as well. However, these fields are also used by functions for the opposite conversion (i.e. from analysis to vector form) for storing results of the converted analysis function, therefore such use could cause undesirable interference. We therefore use different storage on the conversion objects for storage of function results (either of standard analysis or vector function) and for intermediate storage of converted data[1].

This is especially useful when only a temporary conversion to a specific form is necessary in order to perform some operation that is implemented for one form of response functions but not for another. A typical example is numerical differentiation (see Section 2.5). This operation is implemented for standard vector functions (type `vec_bas_f`) while we would sometimes like to use it for numerical differentiation of analysis response calculated by a standard analysis function (type `analysis_bas_f`). Rather than implementing the same operation twice, we can use conversion from analysis to vector function, differentiate the results of the vector function, and convert the function that calculates the numerical gradients of the vector function back to analysis function. In this case, we need the (converted) vector function just for performance of numerical differentiation (in order to calculate gradients that are not provided analytically), while in the final

[1] Actually, intermediate storage for converted data is introduced just for convenience. Alternatively, results of the converted function could be directly copied to the output arguments of the conversion function. With intermediate storage, we can simplify things by treating all special cases (i.e. storage addresses defined or not, calculation of specific response requested or not, etc) separately from conversion between two different forms of data arrangement (i.e. analysis versus vector form).

stage we would like to operate on the analysis function. The above described system enables use of the single conversion data pointer for both ways of conversion (with other words, `anfunc_fromvec` and `vecfunc_froman` will use the same definition data). See Sub-section 2.5.2 on more details how this is implemented.

2.5 Numerical differentiation of analysis results

The role of this chapter is twofold. The first part describes how to use numerical differentiation of a given analysis function, which is implemented in the library. This part is of the sole interest for users of the library.

On the other hand, the second part of the chapter gives a more detailed specification of implementation concepts and is meant as an instructive example of how a given task is implemented in the library. While the first part will be interesting for users of the library, the second part is mainly intended for people who intend to contribute development work to the library or to extend the library for their own needs. These readers can read only the introduction of the first part of the chapter and skip to the second part, unless they will use the numerical differentiation functionality.

2.5.1 User instructions

2.5.1.1 Numerical differentiation: background

Differentiation of the numerical analysis refers to calculation of gradients of all functions that define the analysis, i.e. the objective and constraint functions. In some cases, analysis functions may be able to provide analytical derivatives. Otherwise, we can perform numerical differentiation if we need the derivatives. The simplest approach is to use a finite difference method with the same parameters for each function:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(x_1, ..., x_i + h_i, ..., x_n) - f(\mathbf{x})}{h_i} . \qquad (2)$$

Instead of the forward difference formula (2) the backward or central difference formulas can also be used, which are also simple and direct formulas, except that the central difference requires two additional function evaluations for each derivative instead of one (and is therefore more precise, exact for quadratic functions).

There are also more complex formulas where the number of additional function evaluations is not even known in advance. For example, we can sample function f in a number of points $\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_m$, calculate some kind of interpolation or approximation $\tilde{f}$ of f on basis of the sampled data $f(\mathbf{x}_1)$, $f(\mathbf{x}_2)$, …, $f(\mathbf{x}_m)$, and calculate the approximate derivatives as derivatives of the approximation $\tilde{f}$. Adaptive schemes can be used, which estimate accuracy of the approximation and sample in additional points with automatic adaptation of sampling domain until accuracy is optimal or satisfactory.

In optimization, the response consists of more functions, usually one objective function f and several constraint functions c_i. We can consider the overall response as a *vector function* **g**,

$$\mathbf{g}(\mathbf{x}) = [f(\mathbf{x}), c_1(\mathbf{x}), ..., c_m(\mathbf{x})]. \quad (3)$$

2.5.1.2 Use of numerical differentiation

Let us say we have an analysis function called `my_analysis`, which calculates the objective and eventually the constraint functions but can not calculate their derivatives with respect to parameters. The analysis function must be of the standard form, i.e. of type `analysis_bas_f` (section 2.3.1). We would like to solve the optimization problem defined by this analysis function by using a gradient-based algorithm implemented by the function `grad_optimize` (the name is fictitious and can refer to any gradient based algorithm that is actually implemented).

For generality, we will assume that the analysis function requires additional definition data of type `my_an_def` and that the function `prepare_an_def` is used to prepare this definition data and the data can be de-allocated by the function declared as

```
void disp_an_def (my_an_def *addrdef);
```

In order to perform the optimization, we will define a *new analysis function*, which takes the function `my_analysis` (together with its definition data) for calculating the response and *numerically differentiates the response* whenever required, returning the response and its numerically calculated derivatives (if requested). This is done in the following way:

Example 2: Setting up an analysis that numerically differentiates the originally provided analysis function (and data) and using it for gradient based optimization.

```
my_an_def andeforig=NULL;  /* definition data for original analysis */
analysis_bas_f gradfunc=NULL;  /* new analysis function with original definition of
            response and numerical derivatives */
analysis_to_vecfunc_cd gradcd=NULL;  /* definition data for new analysis */
void (*dispgradcd) (void **)=NULL;  /* function for de-allocation of definition data */
double numderstep;
...
numderstep=1.0e-6;  /* finite difference step used for numerical differentiation */
prepare_an_def(..., &andeforig);  /* set up definition data for the original analysis
            (dynamically allocated) */
/* Prepare the analysis that will add derivative information by numerical differentiation of the
            original analysis: */
```

```
prepanfuncnumgrad(my_analysis, (void *) andeforig, (void (*)(void **))
              disp_an_def,
          numderstep, NULL,   0, 0, 0,
          &gradfunc, &gradcd, &dispgradcd );
/* Perform gradient-based optimization: */
grad_optimize(..., gradfunc, gradcd, ...);
... /* Tread optimization results, etc. */
/* De-allocate auxiliary structures: */
if (dispgradcd!=NULL && gradcd!=NULL)
  dispgradcd((void **) &gradcd);
```

The function **prepanfuncnumgrad** has has been used for preparation of the analysis function for numerical derivation of the originally provided response and its definition data. The first two arguments of this function are the original analysis function and its definition data.

The function chooses the pre-defined analysis function that will perform the analysis with numerically calculated gradients (its address is written to `gradfunc` in the above example) and prepares dynamically allocated definition data for this analysis (its pointer is stored to `gradcd` in the above example). The function also sets the address of the function for de-allocation of the definition data for analysis with numerical gradients, which is used at the end to de-allocate the created definition data.

If the function for de-allocation of the original definition data (in this case) was not specified (i.e. NULL was passed as the corresponding argument to `prepanfuncnumgrad`), then de-allocation of `gradcd` would not de-allocate the definition data. This is useful when we want to further use the original definition data[1].

According to the current implementation, we don't need the `prepanfuncnumgrad` to store the analysis function for numerical differentiation and the function for de-allocation of its definition data, since these are known in advance. The analysis function (its address is stored to `gradfunc`) is **anfunc_fromvec** while the function for de-allocation of the definition data (its address is stored to `dispgradcd`) is **dispanalysis_to_vecfunc_cd**. We could have used these functions directly instead of `gradfunc` and `dispgradcd`, however, provision of function addresses by `prepanfuncnumgrad` enables extensions of the system for numerical differentiation of the analysis response and is therefore safer to use.

The parameter `numderstep` defines the scalar step used for numerical differentiation. The step can be separately provided for each design parameter, which is done by a vector argument following the scalar step (if this argument is NULL then a scalar step is taken).

Declaration of the function that prepares data for numerical differentiation and the meaning of its argument is as follows:

```
analysis_to_vecfunc_cd prepanfuncnumgrad(analysis_bas_f anfunc, void *ancd,
              void (*dispancd) (void**), double step, vector vstep, int backdif,
              int quadratic, int noforcenum, analysis_bas_f *addrfunc,
              analysis_to_vecfunc_cd *addrcd, void (**addrdispcd)(void **) )
```

[1] This may be the case, for example, when the complete job from Example 2 is done within a function and the original analysis function and its definition data are passed from the calling code as arguments of this function. In this case, the caller would create the original definition data and would also have the responsibility to destroy (de-allocate) it when not needed any more.

Function prepanfuncnumgrad creates (allocates) and *prepares the definition data* that will be used for the analysis that undertakes the numerical differentiation of the original response. Pointer to this data is returned by the function and stored to *addrcd if this argument is not NULL (either the address of already allocated data or of a NULL pointer can be passed). After preparation, the definition data contains the address of the original analysis function (provided through argument anfunc) and its definition data (provided through andata, which may be NULL if a definition data is not necessary for the original analysis).

The analysis function that eventually performs numerical differentiation of the original analysis is provided through the output argument addrfunc. The address of the function is stored in the pointer pointed to by this argument, which should be of type analysis_bas_f. The provided function can be used as the original analysis function, except that the provided definition data (the returned pointer) is used and that the function is able to provide gradients of the response by automatic numerical differentiation of the original response whenever necessary.

Since the definition data is dynamically created, it should be de-allocated after use. This should be done by the function whose address is provided through the output argument addrdispcd. De-allocation by using this function also de-allocates the definition data for the original analysis provided that it had been provided by the argument ancd (i.e. if this argument was not NULL when prepanfuncnumgrad was called) and also the appropriate de-allocation function had been provided by the argument dispancd. If that argument is NULL then de-allocation of the created definition data will not affect the definition data for the original function (its de-allocation can be performed elsewhere).

Arguments of the function are described in more detail below.

anfunc is the original analysis function whose response will be numerically differentiated by the provided function.

ancd is the definition data for the original analysis function (may be NULL if the analysis function does not require any particular definition data).

dispancd is an optional argument (it may be NULL) that specifies the function for de-allocation (destruction) of the definition data for original function (defined by the argument ancd). If it is non-NULL then de-allocation of the created definition data (provided through addrcd) will also de-allocate ancd by using this function. If it is NULL, de-allocation of the created definition data will not attempt to de-allocate the original definition data ancd.

step specifies the step length for numerical differentiation (i.e. the amount for which optimization parameters are perturbed when performing numerical differentiation). It should be a positive integer.

Vector of steps vecstep may be specified to define the step length for each parameter separately. If specified then it must have the same dimension as the design space (equivalently, the vector of parameters). Its components have the same meaning as h_i in equation (2). If it is NULL then the scalar step is taken for perturbation in all co-ordinate directions.

The arguments backdif, quadratic, and noforcenum define flags, which define how numerical differentiation is performed. The value 0 can be used for all of these arguments. Their meaning is the following:

backdif: if non-zero then backward difference scheme is performed instead of forward difference.

`quadratic`: if non-zero then a central difference scheme is applied where each derivative is calculated by performing two additional analyses at perturbed parameters.

`noforcenum`: if non-zero then original (analytical) derivatives are used if they can be provided by the original analysis function. This can be used to set up the analysis function which automatically performs numerical differentiation if it is necessary, but uses the originally provided derivatives if their calculation can be performed by the original analysis function.

`addrfunc` (output argument) is the address of the variable (it should be of function type analysis_bas_f) into which the address of the *new analysis function* (that will provide gradients by numerical differentiation) is stored. We can pass NULL for this argument, in this way it is assumed that we know which analysis function is used for numerical differentiation. Currently, this particular function is always **`anfunc_fromvec`**. However it is safer to obtain the function address through output argument `addrfunc` because the system may be extended in the future in such a way that different functions will be used.

`addrcd` (output argument) is the address of a pointer to which the (address of) newly created definition data for the analysis undertaking numerical differentiation is assigned. The analysis is performed by the function whose address is stored to `*addrfunc`, therefore `*addrcd` will point to the definition data that must be used with that function. The data is dynamically allocated, therefore it must be de-allocated when not needed anymore. De-allocation is done by the function whose address is stored in the next argument. The argument `addrcd` may be NULL because the created definition data is also returned by the function.

`addrdispcd` (output argument) is the address of the function pointer to which the (address of) function for de-allocation of the definition data is stored. This function must be called to de-allocate the definition data pointer that is stored to `*addrcd` or returned by `prepanfuncnumgrad`. This argument may be NULL, in which case it is assumed that the caller knows how to de-allocate the created definition data. Currently only one type of the definition data is used and can be de-allocated by the function `dispanalysis_to_vecfunc_cd`, however the system may be extended in the future and it is therefore safer to perform de-allocation by the function provided through the argument `addrdispcd`.

Function ***returns*** the created definition data for the analysis that performs numerical differentiation of the original analysis.

2.5.2 Example for developers: implementation of numerical differentiation

Form (3) where objective and constraint functions are treated on equal terms is somehow more convenient for implementation of numerical differentiation. Basic variants of functions for numerical differentiation of the analysis response are therefore not implemented for functions of type `analysis_bas_f` (section 2.3.1), but for functions of type `vec_bas_f` (Section 2.3.3) representing vector functions. |Numerical differentiation of analysis response is therefore performed in such a way that response is converted to a vector function, which is differentiated numerically, and vector response with its numerical derivatives is converted back to analysis response with numerical derivatives. Conversion between different response is just re-arrangement of data.

Calling scheme is shown in Figure 1. Numerical differentiation is actually performed by the function `vecfuncnumgrad`, which differentiates the response obtained by `vecfunc_froman`. This function just calls the original analysis function and converts its response to vector form before returning it. The function `anfunc_fromvec` re-arranges the numerically differentiated response returned by `vecfuncnumgrad` and returns it. This function can therefore be seamlessly called instead of the original (possibly non-derivative) analysis function to provide response gradients in additional to the response itself. In order to calculate the gradients, the original analysis is calculated several times at perturbed parameters, which is done by `vecfuncnumgrad` that calls the original analysis indirectly through `vecfunc_froman`. This is however not seen by the caller who calls the `anfunc_fromvec` just like any other standard analysis function.

Crucial for the system to work correctly is the analysis definition data for the outer-most level `anfunc_fromvec`, which contains the definition data for all inner levels and also some auxiliary storage which abolishes the need for allocation and de-allocation in subsequent calls[1]. Processing overhead is therefore limited to transcription of data from one form to another (i.e. from analysis to vector and then back to analysis form) and to two additional function calls (for conversions from analysis to vector form and back), which could be avoided if differentiation was performed directly on the analysis response. This is usually negligible as compared to unavoidable additional calls off the original analysis at the perturbed parameters.

It is indicated in Figure 1 which definition data is used in outer and inner level calls. Definition data of type *analysis_to_vecfunc_cd* , which is created and returned by `prepanfuncnumgrad` through the argument `addrcd`[2] (and is denoted *CD* in Figure 1), is used by the outer-most function `anfunc_fromvec`[3]. The same definition data is used by `vecfunc_froman`, which converts analysis response to vector response. The outer-most `anfunc_fromvec` calls the vector function CD->vecfunc (which is set to `vecfuncnumgrad`) with definition data CD->veccd. On the other hand, `vecfunc_froman` calls the analysis function CD->anfunc (which is set to original analysis function whise response should be differentiated) with definition data CD->ancd (which is the original definition data).

CD->veccd is dynamically allocated definition data of type *vecfuncnumgradcd*, designed in particular for numerical differentiation of vector functions (type *vec_bas_f*) and used by `vecfuncnumgrad`. This function calls (for evaluation of non-derivative vector response) CD->veccd->vecfunc, which is set to `vecfunc_froman`, with definition data CD->veccd->veccd, which is set back to CD.

---

[1] Allocation of intermediate result storage is done only in the first call. In subsequent calls, already allocated space is used. This auxiliary space is de-allocated when the definition data for analysis with numerical differentiation is de-allocated.

[2] Function for de-allocation of the created definition data is returned through the argument `addrdispcd`.

[3] Address of this function itself is also provided by `prepanfuncnumgrad` (through the argument `addrfunc`), which makes the system extensible. This function converts vector response (on which numerical differentiation is actually performed) back to analysis response. The caller can just use `dispanalysis_to_vecfunc_cd` for de-allocation (normally, this function will be provided by `prepanfuncnumgrad`). Using the provided function enables extension of the mechanism without affecting the code that makes its use, however it is not foreseen that the type of the definition data could change.

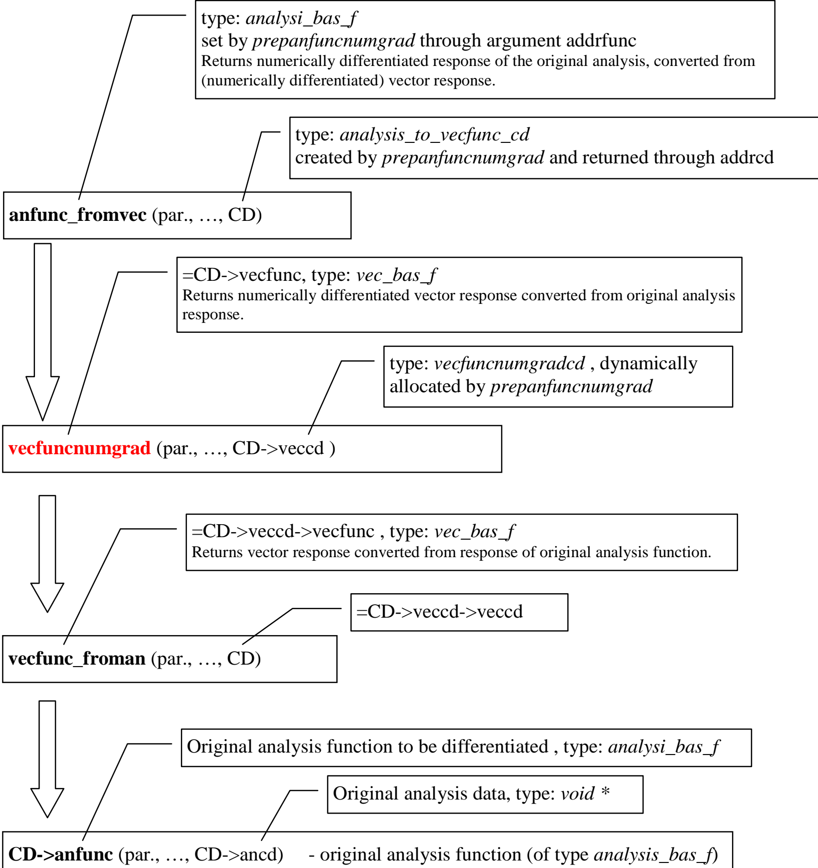


Figure 1: Calling scheme for numerical differentiation of response function calculated by the analysis function.

Below the calling scheme form Figure 1 is depicted in a simpler form:
Calling scheme presented in short:

- **anfunc_fromvec**(…, analysis_to_vecfunc_cd CD)
    - **vecfuncnumgrad** (=CD->vecfunc)(…, vecfuncnumgradcd cdvec (=CD->veccd))
        - **vecfunc_froman** (=cdvec->vecfunc)(…, analysis_to_vecfunc_cd CD (=cdvec->veccd))
            - **anfuncorig** (=CD->anfunc)(…, void * cdorig (=CD->ancd))

2.5.2.1 Additional functionality

The mechanism for numerical differentiation enables some additional functionality. Most of this is enabled by the structure of the type `analysis_to_vecfunc_cd`, which is intended for

conversion between analysis response and vector function response. In order to use this functionality, one must directly access the definition data structure (call it CD, fsay).

For example, setting the flag CD->recordan will cause that **results of individual analyses** will be **recorded** and pushed on the stack CD->anpoints as pointers of type `analysispoint`. These pointers are dynamically allocated and can be popped from the stack for further use (e.g., analysis results at perturbed parameters, which are calculated implicitly at numerical differentiation, can be further used for forming approximated response). Recording (i.e. transcription of analysis response to analysis points on CD->anpoints) is performed by `vecfunc_froman` when CD->recordan is non-zero.

If the caller will only need the recorded analyses temporarily to do some checks something after calling the function, he can move them from CD->anpoints to CD->anstore. In this way, the analysis points will be recycled (used at next analysis recording), which will reduce the need for dynamic allocation (since already allocated analysis points will be used to record results).

2.5.3 Extension of the numerical differentiation system

Most likely, intended extension of the system will include addition of new methods for numerical differentiation. For this, one will need to extend the `vecfuncnumgrad` function, which actually does the job. When extending this function directly, it would probably need new flags on its definition data of the type `vecfuncnumgradcd`, which would tell the function to perform differentiation according to the newly added method. One should therefore extend the body of the function and the definition of the type of its definition data. This is possible if one has the source code of the appropriate modules (in this case `optbas.c`). However, in order to make extension accessible to other users of the library, it should be built into the distribution that is used by these users.

More elegant way is to add a completely new function that would replace `vecfuncnumgradcd` when a new differentiation method would be used, and a new function for preparation of the definition data for the outer-most level analysis function. This function can be based on `prepanfuncnumgrad` and can be its simple extension (e.g. it can simply call `prepanfuncnumgrad` when previously implemented method is requested, or do preparation of definition data itself when the new method is requested). In the case when the newly implemented algorithm is requested, the data preparation function should store address of the new function for numerical differentiation in *CD->vecfunc*, its definition data in *CD->veccd*, and address of the function for de-allocation of CD->veccd in *CD->dispveccd*.

The type of CD->veccd would not necessarily be `vecfuncnumgradcd`, by which we would avoid the necessity to extend this type. If the type would turn sufficient to carry all the data needed for the new algorithm, we can of course keep the old type. In this case we can use other functionality supported by this type, e.g. storing of vector points. If we only need some minor extension of the type (e.g. some additional flags or coefficients), we can use an array of pointers of standardized length where the first pointer points to data of type `vecfuncnumgradcd` and the rest point to additional data. Then we can still use whatever is provided by the original type. Of

course, we must correspondingly implement the de-allocation function, which is stored in *CD->dispveccd*.

When deriving a new definition data preparation function from the original `prepanfuncnumgrad`, we must take care of what to do when the definition data for vector function with numerical differentiation *CD->veccd* is already allocated. If the type is appropriate (i.e. the same as will be used for our definition data) then we can use it and only change the contained data as necessary (e.g. the flags that precisely specify which numerical differentiation algorithm is used). If the type of already allocated CD->veccd is not the same as the one we will use, we must first de-allocate the old data (by using CD->dispveccd) and then create CD->veccd anew. If CD->dispcd is NULL then we don't de-allocate CD->veccd because it will be de-allocated somewhere else[1], otherwise we use this function for de-allocation. Normally, we check equivalence of types by comparing the de-allocation function CD->dispveccd to the one used for de-allocation of our data - if they are the same then we can conclude that the types are the same, too. In this specific case, de-allocation of eventually allocated CD->veccd can be made in any case because de-allocation at this point does not significantly affect the processing time and thus computational efficiency.

2.5.3.1 Additional implementation remarks

Numerical differentiation of analysis results requires double conversion from standard analysis to standard vector function and back to standard analysis function. For both conversions, the same data structure (called *CD* in the above discussion) of the type `analysis_to_vecfunc_cd` is used as definition data. These conversions are performed by `vecfunc_froman` and `anfunc_fromvec`, respectively (Figure 1). The implementation remarks explaining how both conversions can use the same structure are given in Sub-section 2.4.2.

2.6 Prevention of repeated analysis at the same parameters and analysis counts

Some of the provided automatisms may cause the same numerical analysis to be successively performed several times with the same optimization parameters. This is the case fe.g. when the implementation of optimization algorithm requires separate functions for evaluation of each type of response, but the analysis is structured in such a way that the most time consuming part of calculation is the same for all types of response upon which calculation of individual parts of

[1] Such is the agreement for many structures that contain pointers to another possibly dynamically allocated structures. Such a structure contains address of a function for de-allocation of the dynamically allocated structure, and if this address is NULL then this means that the pointer is managed somewhere else (i.e. pointer on the structure is not the main handle of the data), therefore de-allocation should not be made. Such agreement is typical for structures that may contain some additional data of types which are not known in advance. In contrary, for container types that are primarily intended for carrying large amount of data (such as the `stack` type), different agreement may be used, and some function for de-allocation of container types may consider non-specification of function for de-allocation of individual elements just as indication that elements are simple pointers, which can be de-allocated by the standard function `free`.

response only requires a negligible amount extra time[1]. Such redundant repetitions are undesirable especially when the analysis is time consuming.

Such situations can be avoided by **replacing the original analysis function by** the appropriate **wrapping function** that records the last results and prevents repetitive performance of the analysis at the same optimization parameters by simply returning the already calculated response[2]. The function calls the original function when the evaluation of response is actually needed. Let us mention that this mechanism can also be used for defining the analysis function that returns a previously stored response, which may be useful in some cases.

The basic analysis function that prevents redundant calculation is defined in `optbas.c` and is declared as

```
int anfunccountnorepeat(vector param, int *calcobj, double **addrobj, int
            *calcconstr, stack *addrconstr, int *calcgradobj,vector
            *addrgradobj, int *calcgradconstr, stack *addrgradconstr, void
            *cd[4]);
```

The function is of the standard analysis function type `analysis_bas_f`, except that the type of the definition data (the last argument `cd`) is prescribed and must be an array of four pointers. Other arguments must be the same as would be passed to the original function.

`cd[0]` must be the address of the original analysis function, which will be called by `anfunccountnorepeat` whenever necessary (i.e. when the requested response is not known from one of the previous calls).

`cd[1]` must be the definition data for the original analysis.

`cd[2]` must be a pointer to storage of type int. If it is not NULL then the number pointed to by `cd[2]` is used as analysis counter and is incremented each time the original analysis is called within the function.

`cd[3]` must be an address of a variable of type `analysispoint`, and is used to store analysis results. The pointer pointed to by cd[3] may either be NULL or a valid dynamically allocated pointer of type `analysispoint`. If it is NULL then it will be allocated internally within the function. If cd[3] is NULL then the analysis will be performed properly, but the repetition prevention mechanism will not work (i.e. it may happen that calculation with the same parameters is performed several times successively). Normally, a warning message is launched when this happens for the first time[3].

---

[1] This is the case when, for example, all the response is based on the results of a complex numerical analysis, and different types of response such as the objective function and additional constraint functions require only different post-processing of the simulation results, which is much less processing time consuming than the simulation itself.

[2] This mechanism will of course make use of the analysis definition data, which will for the wrapping function contain the address of the original analysis function (which must be called to calculate the response) and its definition data (which will be passed as argument when this function is called) and auxiliary data to keep track of previous calls of the original analysis. This makes sure that the wrapping function can really be used instead of the original at any place, that the mechanism is thread safe and that the wrapping function may be called in nested calls.

[3] This can be prevented by calling `setprintlevelanfuncnorepeat(0)`.

The function `anfunccountnorepeat` stores optimization parameters and results to *(cd[4]) after each call to the original analysis function for calculating the response. At every call, it first checks whether the requested response is stored in *(cd[4]), and if it is, it just copies the response instead of actually calling the original analysis function (whose address is in cd[0] and its definition data in cd[1]). Example 3 shows how to use the mechanism in practice.

Example 3: Use of the wrapping analysis function that prevents successive repetition of analysis defined by `anfunc` and `andata` at the same parameters:

```
analysispoint anpt=NULL;
void *cd[4];
int ancount=0, numparam=3,calcobj=1, calcconstr=1, calcgradobj=1,
            calcgradconstr=1;
double obj=NULL;
vector gradobj=NULL, param=NULL;
stack constr=NULL, gradconstr=NULL;
/* Prepare definition data for wrapping function that will replace the
            original analysis function: */
cd[0]=anfunc;    /* set original analysis function */
cd[1]=andata;    /* set original definition data */
cd[2]=&ancount;
cd[3]=&anpt;
...
param=getvector(numparam);
anfunccountnorepeatsimp(param, &calcobj, &obj, &calcconstr, &constr,
            &calcgradobj, &gradobj, &calcgradconstr, &gradconstr, cd);
...
optimize(...,anfunccountnorepeatsimp, cd);  /* optimization of response
            defined by anfunc and andata, through wrapping analysis function
            that prevents unnecessary repetition */
...
dispanalysispoint(&anpt)    /* don't forget to de-allocate the auxiliary
            storage when not needed any more */
```

Notes:

When successively using the same storage (of type `analysispoint`) for different analyses via definition data of `anfunccountnorepeat`, make sure that all calculation request flags are set to 0 before use with a new analysis. This will enforce actual calculation at the first call and prevent eventual use of response calculated by the previously used analysis in the case that optimization parameters at which the previous analysis was called accidentally mach the parameters at which the new analysis is called. Otherwise, the function is thread safe and suitable for nested calls.

A **simpler variant** of the function exists and is declared as

```
int anfunccountnorepeatsimp(vector param, int *calcobj, double **addrobj, int
            *calcconstr, stack *addrconstr, int *calcgradobj, vector
            *addrgradobj, int *calcgradconstr, stack *addrgradconstr, void
            *cd[3]);
```

The definition data of this function is an array of only three pointers because the storage address for storing the last results is not necessary. Instead, the results are stored in a static variable. This function is still thread safe (because locking is used), but calling it from parallel threads may cause inefficiency because calls from different threads may override the stored data (which may also cause excessive re-allocation when response dimensions are not compatible). Because of this, the function may in some cases not be able to correctly detect that the same analysis had been performed by the previous call in the same thread, and complete calculation will be unnecessarily repeated. This simple variant could not be used in nested calls and would block (because of thread locking) if we attempt to make nested calls. Nested call would occur for example if the `anfunccountnorepeatsimp` was called within the original analysis that is used with this function (and thus called within it).

If we don't need to take care of successive repetitions of the same analysis, we may want to use the analysis that **just counts the number of times the analysis is performed**. Such wrapping analysis is declared as

int anfunccount(vector param, int *calcobj, double **addrobj, int *calcconstr, stack *addrconstr, int *calcgradobj, vector *addrgradobj, int *calcgradconstr, stack *addrgradconstr, void *cd[3]);

Further explanations are not necessary because the pointers in `cd` have the same meaning as with `anfunccountnorepeatsimp` (and the same meaning as the first three pointers in the definition data of `anfunccountnorepeat`).

3 BASIC BUILDING BLOCKS

This Section describes basic building blocks that are commonly used in higher level utilities. These are usually not the most lower level utilities, however they are not meaningful by themselves but are used in yet higher level tools for solving specific problems. A typical example is implementation of affine co-ordinate transformations. In optimization algorithms, these is used at many places: For scaling of variables, for definition of trust region constraints, for deriving multivariate weighting functions from one dimensional forms, etc. It is possible and therefore meaningful to build a common implementation of affine transforms for all of these tasks in a single library module, and this module can then be used as a low lever tool in any of the above mentioned higher level utilities. Still co-ordinate transforms are not the lowest level functionality (i.e. just above the basic language features), because they rely on a whole set of matrix and vector operations. The task of the affine transformation module is to collect all the necessary matrix algebra functionality and pack it in such a way that it can be used in a simple way without referring to individual operations of the underlying implementation. I.e., an affine transformation is represented by a single structure and a corresponding set of operations, which are in the sense of

"define parameters of the transformation", "transform a vector", "inverse transform a vector", "set expected number of inverse transformations", "set symmetric property of transformation matrix", etc. The user does not need to care which methods and functions are applied under given circumstances.

3.1 Co-ordinate transformations

This topic is so common that it seems appropriate to explain implementation details together with some mathematical background[1]. Here it is necessary to accompany the text with some formulas in order to put things precisely. Mathematical terminology in this area completely clear[2], but since use of programming constructs (such as structured type) is often extended beyond its primary scope, one must make clear what is what.

When prescribing some function rule, we deal with vectors of co-ordinates that uniquely define points as elements in the function definition domain, which is a subset of a vector space[3]. Stating point co-ordinates is just a way of labeling points in vector spaces, but this way is not unique. We can always introduce another indexing system by transforming co-ordinates of the space. Co-ordinates of a point in the transformed co-ordinate system are expressed as function of original co-ordinates, e.g.

$$\tilde{\mathbf{x}} = \mathbf{F}(\mathbf{x}), \tag{4}$$

where F should be a continuous one-to-one (bijective) map. Tricky point with co-ordinate transforms is that one must know precisely in which co-ordinate system scalar or vector fields are specified. This will in general differ from case to case and one of the goals of this Section is to make these things absolutely clear.

Where functions are defined depends on the purpose of co-ordinate transforms. One possible purpose is to enable general use of a simple function representation. For example, weighting functions used in moving least squares are in general functions of difference between the point of evaluation and the sampling point that corresponds to a weighting function (see e.g. Sub-section 5.4.2). In most cases, desirable form of weighting functions allows us to introduce new variables in such a way that expressed in new co-ordinates, weighting functions are just functions of the *distance* between the point of evaluation and sampling point (see equations (26) and (27)). This is very useful because for the definition of actual weighting functions, we only need to define a *scalar function* of vector variable (not a vector variable) and parameters of transform (such as **A** and **s**

[1] Usual practice of the library manuals is to elaborate specialized topics in separate documentation while this library is intended to cover only the basic framework. However, in the case of linear transforms it is extremely important to harmonize notation in such a way that the user is always precisely aware of the meaning of individual quantities. This is best achievable without a short mathematical description of the topic with analogous notation used as in the code.

[2] Although in some literature there are differences in use of terms (in particular in engineering books), e.g. the term "linear functions" is often used for something what is otherways called "affine functions".

[3] This discussion is limited to real spaces and thus real co-ordinates.

from (27)). This enables much simpler testing of different forms of weighting functions because replacement of definitions is performed on the level where it is much simpler. Moreover, we can use the same function definition (together with its definition data) for all sampling points while we have different transform parameters (such as **s** and **A**) belonging to different sampling points, which is more meaningful. In this case, definition of functions is bound to transformed co-ordinates while final evaluation (values and gradients) is performed on original co-ordinates, and primary role of co-ordinate transform is in fact function composition.

A different situation occurs when we transform design parameters in order to perform optimization on transformed parameters (e.g to ensure better scaling in the first place, or to enforce bund constraints by sigmoidal parameter transforms). In this case, response function (and eventually their gradients) are defined in the original co-ordinate system, while solution procedures will be performed in the transformed co-ordinates. Moreover, algorithm parameters such as starting guess (and possibly tolerances) are also defined in original co-ordinates. In order to solve the problem, we must transform definition of response surface, their gradients and algorithm parameters to transformed co-ordinates. When results are obtained, they must be transformed to the original co-ordinate system to be readable by the user.

3.1.1 Linear transformation of co-ordinates

Let us denote by x_i and $\tilde{x}_i$ co-ordinates of the same point (or vector) written in two different co-ordinate systems K and $\tilde{K}$, respectively. Co-ordinates of a vector in a given co-ordinate system are coefficients of linear combination of basis vectors that equals this vector. We will denote *basis vectors* of K by $\mathbf{e}_1, \ldots, \mathbf{e}_n$ and basis vectors of $\tilde{K}$ by $\tilde{\mathbf{e}}_1, \ldots, \tilde{\mathbf{e}}_n$. Then we have

$$x_k \, \mathbf{e}_k = \tilde{x}_i \, \tilde{\mathbf{e}}_i \ , \tag{5}$$

where we expressed the same point as linear combination of the two sets of basis vectors. Co-ordinates of a point in a new co-ordinate system are obtained from original co-ordinates by the following linear transformation:

$$\tilde{\mathbf{x}} = \mathbf{A}\,\mathbf{x} = \tilde{\mathbf{A}}^{-1}\,\mathbf{x} \tag{6}$$

or[1] $\tilde{x}_i = a_{ij} x_j$. Inverse transformation matrix $\tilde{\mathbf{A}}$ transforms co-ordinates in $\tilde{K}$ to co-ordinates in K:

$$\mathbf{x} = \tilde{\mathbf{A}}\,\tilde{\mathbf{x}} . \tag{7}$$

[1] We use Einstein's summation agreement: if an index repeats in the same side of equation then this means summation over that index, e.g. $a_{ij}\, x_j = \sum_j a_{ij}\, x_j$

Columns of inverse transformation matrix $\mathbf{A}^{-1}=\tilde{\mathbf{A}}$ *are co-ordinates of basis vectors of new co-ordinate system* $\tilde{K}$ expressed in the original system K:

$$\tilde{\mathbf{e}}_i = \tilde{a}_{ji}\,\mathbf{e}_j \ . \tag{8}$$

To check the above formula, we input (8) into (5): $\tilde{\mathbf{x}}_i\,\tilde{\mathbf{e}}_i = \tilde{x}_i\,\tilde{a}_{ji}\mathbf{e}_j = x_k\mathbf{e}_k$ (where we have summation over i and j according to the summation agreement). Since $\mathbf{e}_i$ are linearly independent basis vectors, coefficients of their linear combinations are unique and we can equate coefficients of the same vectors, which gives $x_j = \tilde{a}_{ji}\tilde{x}_i$ or $\mathbf{x} = \tilde{\mathbf{A}}\,\tilde{\mathbf{x}}$. Since both sets $\{\mathbf{e}_i\}$ and $\{\tilde{\mathbf{e}}_i\}$ are bases of vector space $\mathbb{R}^n$, $\mathbf{A}$ is a full rank $n \times n$ matrix.

If both K and $\tilde{K}$ are *Cartesian co-ordinate systems* with orthogonal and normalized basis vectors then $\mathbf{A}^{-1}$ and hence $\mathbf{A}$ are *orthogonal* matrices,

$$\mathbf{A}\,\mathbf{A}^T = \mathbf{I} \,. \tag{9}$$

This also means that scalar products of any two distinct columns (or rows) of $\mathbf{A}$ is 0 and scalar product of any row (or column) by itself is 1[1]:

$$a_{ki}\,a_{kj} = \delta_{ij}\ ,\ \ a_{ik}\,a_{jk} = \delta_{ij}\ , \tag{10}$$

where $\delta_{ij} = \begin{cases} i; i = j \\ 0; i \neq j \end{cases}$ is the Kronecker symbol. In this case, rows of $\mathbf{A}$ are co-ordinates of corresponding basis vectors $\tilde{\mathbf{e}}_i$ of $\tilde{K}$ in K (since they correspond to columns of $\tilde{\mathbf{A}} = \mathbf{A}^{-1}$). Because in Cartesian systems co-ordinates are scalar products of vector with corresponding basic vectors (i.e. $\tilde{\mathbf{e}}_i = (\tilde{\mathbf{e}}_i \cdot \mathbf{e}_j)\mathbf{e}_j$), we have

$$a_{ij} = \tilde{\mathbf{e}}_i \cdot \mathbf{e}_j \ \text{(Cartesian systems)}. \tag{11}$$

This can be verified as follows: $x_i\mathbf{e}_i = \tilde{x}_j\,\tilde{\mathbf{e}}_j = \tilde{x}_j\left((\tilde{\mathbf{e}}_j \cdot \mathbf{e}_k)\mathbf{e}_k\right)$, after equating coefficients at $\mathbf{e}_j$ we have $x_i = \tilde{x}_j(\tilde{\mathbf{e}}_j \cdot \mathbf{e}_i) = (\mathbf{e}_i \cdot \tilde{\mathbf{e}}_j)x_j$, therefore $\tilde{a}_{ij} = (\mathbf{e}_i \cdot \tilde{\mathbf{e}}_j) = a_{ji}$ (because $\mathbf{A} = \tilde{\mathbf{A}}^{-1} = \tilde{\mathbf{A}}^T$) and finally $a_{ij} = \tilde{\mathbf{e}}_i \cdot \mathbf{e}_j$.

Transformation of linear operators:

[1] This must be true because of the fact that columns of $\mathbf{A}^{-1}$ correspond to co-ordinates of the basis vectors of the second co-ordinate system in the original (cartesian) system, and since basis vectors of both systems are orthogonal and of unit length, the above formula must be valid.

In a given co-ordinate system, a linear operator is represented by a matrix by which column of co-ordinates of original are multiplied in order to obtained co-ordinates of its image:

$$\mathbf{w} = \mathbf{T}\mathbf{u} \;. \tag{12}$$

We are looking for matrix representation of the same linear operator in the co-ordinate system $\tilde{K}$ if co-ordinates are transformed according to (6). We must satisfy

$$\tilde{\mathbf{w}} = \tilde{\mathbf{T}}\tilde{\mathbf{u}} \text{ or } \mathbf{A}\,\mathbf{w} = \tilde{\mathbf{T}}\left(\mathbf{A}\,\mathbf{u}\right).$$

We have $\mathbf{w} = \mathbf{A}^{-1}\tilde{\mathbf{T}}\mathbf{A}\,\mathbf{u}$ and thus from (12) $\mathbf{A}^{-1}\tilde{\mathbf{T}}\mathbf{A} = \mathbf{T}$, and finally

$$\tilde{\mathbf{T}} = \mathbf{A}\,\mathbf{T}\,\mathbf{A}^{-1} \tag{13}$$

3.1.2 Linear (Affine) maps, eigendecomposition and quadratic forms

The most general co-ordinate transformation discussed in this Section is Affine transformation, expressed as

$$\tilde{\mathbf{x}} = \mathbf{A}\left(\mathbf{x} - \mathbf{s}\right), \tag{14}$$

where A is an arbitrary square ($n \times n$) matrix. Without additive term –**s**, this would be a linear transformation. If **A** has full rank, this is a non-degenerate co-ordinate transformation, i.e. transformation is bijective and the dimension of the new co-ordinate system is the same as the dimension of the old one. In this case, there exists inverse of **A**, and we can express original co-ordinates **x** with transformed co-ordinates $\tilde{\mathbf{x}}$.

Of particular importance are cases where the real transformation matrix **A** is *symmetric*. Real symmetric matrices have exactly n eigenvalues (not necessarily unique) and corresponding orthogonal eigenvectors, which satisfy the equation

$$\mathbf{A}\,\mathbf{x}_{\lambda \mathbf{A} i} = \lambda_{\mathbf{A} i}\,\mathbf{x}_{\lambda \mathbf{A} i},\; i = 1, ..., n \;. \tag{15}$$

Eigenvalues $\lambda_{\mathbf{A}i}$ are solutions of the characteristic equation $\det\left(\mathbf{A} - \lambda\,\mathrm{I}\,\right) = 0$ Two eigenvectors that correspond to different eigenvalues are always orthogonal ($\mathbf{x}_i^T\mathbf{x}_j = 0 \forall i \neq j \wedge \lambda_i \neq \lambda_j$). Multiple eigenvalues with multiplicity p are multiple zeros of characteristic equations with p corresponding linearly independent eigenvectors. Any non-trivial linear combination of these vectors is also an eigenvector with the same eigenvalue, and they can be orthogonalized to obtain p orthogonal vectors.

Every symmetric real matrix **A** can be written as

$$\mathbf{A} = \mathbf{U}_{\mathbf{A}}\,\mathbf{D}_{\mathbf{A}}\,\mathbf{U}_{\mathbf{A}}^{T} \tag{16}$$

where $\mathbf{D_A}$ is a diagonal matrix whose diagonal elements are eigenvalues of **A** and $\mathbf{U_A}$ is *orthogonal matrix*[1] whose columns are the corresponding eigenvectors of **A**[2]. The above formula is therefore called eigendecomposition (or spectral decomposition) of **A**.

Matrix **A** is transformed to diagonal form **D** by the so-called *transform of the main axes*,

$$\mathbf{D}_{\mathbf{A}} = \mathbf{U}^{T}\mathbf{A}\,\mathbf{U}. \tag{17}$$

This represent transformation into a new Cartesian co-ordinate system whose basic vectors are eigenvectors of **A**.

A *quadratic form* is a function of the form

$$Q(\mathbf{x}) = \mathbf{x}^{T}\mathbf{Q}\,\mathbf{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} Q_{ij}\, x_i\, x_j \tag{18}$$

where **Q**[3] is a real symmetric matrix. If **Q** is *positive definite* then it can be written as square of another symmetric matrix **A**:

$$\mathbf{Q} = \mathbf{A}^{2} = \mathbf{A}\,\mathbf{A}. \tag{19}$$

We can perform such linear transformation of co-ordinates that Q is expressed as a sum of pure squares in the new co-ordinate system,

$$Q(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}}^{T}\mathbf{K}\,\tilde{\mathbf{x}} = \sum_{i=1}^{r} k_i\, \tilde{x}_i^{2}. \tag{20}$$

This is performed by transformation matrix $\mathbf{U_Q D_{QK}}$ where $\mathbf{U_Q}$ is the matrix whose columns are normalized eigenvectors of **Q** and $\mathbf{D_{QK}}$ is a diagonal matrix with elements

---

[1] A real matrix **U** is orthogonal when $\mathbf{U}^{-1}=\mathbf{U}^{T}$. This means that $\mathbf{U}\mathbf{U}^{T}=\mathbf{U}^{T}\mathbf{U}=\mathbf{I}$, i.e. scalar product of two distinct columns (or rows) are 0 and scalar products of any row or column with itself is 1.

[2] Transformation of the form $\tilde{\mathbf{A}} = \mathbf{G}^{-1}\mathbf{A}\,\mathbf{G}$ where **G** is an invertible matrix is called *similarity transform*. This transform preserves eigenvalues, i.e. $\tilde{\mathbf{A}}$ and $\mathbf{A}$ have the same eigenvalues. If $\mathbf{G}$ is orthogonal then the similarity transform is called *congruent transform*. Beside eigenvalues, this transform also preserves symmetry of the matrix (if $\mathbf{A}$ is symmetric then $\tilde{\mathbf{A}}$ is also symmetric).

[3] If the form Q(**x**)>0 for all **x** then the form is said to be *positive definite* (and so is the matrix **Q**), if it is less than 0 for all **x** it is called *negative definite*, and if it is greater or equal to 0 it is called *positive semi-definite*. Necessary condition for positive definiteness is that all diagonal elements of **Q** are positive. Positive definite matrices have positive eigenvalues.

$$d_{\mathbf{QK}i} = \sqrt{\frac{k_i}{\lambda_{\mathbf{Q}i}}} \ , \tag{21}$$

therefore

$$\begin{aligned} &\mathbf{x} = \mathbf{U}_{\mathbf{Q}} \mathbf{D}_{\mathbf{QK}} \tilde{\mathbf{x}} \ , \\ &\tilde{\mathbf{x}} = \mathbf{D}_{\mathbf{QK}}^{-1} \mathbf{U}_{\mathbf{Q}}^{T} \mathbf{x} \qquad . \\ &\mathbf{K} = \mathbf{D}_{\mathbf{QK}}^{T} \mathbf{U}_{\mathbf{Q}}^{T} \mathbf{Q} \mathbf{U}_{\mathbf{Q}} \mathbf{D}_{\mathbf{QK}} \end{aligned} \tag{22}$$

We see that coefficients k_i can not be just arbitrary, but must have the same sign as the corresponding eigenvalues of **Q** (otherwise there was a negative value under the square root). If we set $\mathbf{D_{QK}}=\mathbf{I}$ then coefficients in (21) are eigenvalues of **Q**, i.e. $k_i = \lambda_{\mathbf{Q}i}$ or $\mathbf{K}=\mathbf{D_Q}$. If **Q** is positive definite then we can perform such linear co-ordinate transformation that $Q(\tilde{\mathbf{x}})=\|\mathbf{x}\|_2$ by setting $d_{\mathbf{QK}i} = \sqrt{1/\lambda_{\mathbf{Q}i}}$.

3.1.2.1 Calculation of gradients

We want to calculate a gradient of a scalar function that is defined on transformed co-ordinates. Note again that $\tilde{\mathbf{x}} = \mathbf{A}(\mathbf{x}-\mathbf{s})$, $\mathbf{x} = \mathbf{A}^{-1}\tilde{\mathbf{x}}+\mathbf{s}$.

If we have a function g defined on transformed co-ordinates and a function h such that

$$h(\mathbf{x}) = g(\tilde{\mathbf{x}}) = g(\mathbf{A}(\mathbf{x}-\mathbf{s})), \tag{23}$$

then gradient of h is

$$\nabla_{\mathbf{x}} h(\mathbf{x}) = \nabla_{\mathbf{x}} g\left(\mathbf{A}\left(\mathbf{x}-\mathbf{s} \right) \right) = \mathbf{A}^T \nabla g\left(\mathbf{A}\left(\mathbf{x}-\mathbf{s} \right) \right). \tag{24}$$

For composition of functions, gradient is

$$\nabla f(g(\mathbf{x})) = f'(g(\mathbf{x}))\nabla g(\mathbf{x}). \tag{25}$$

If we have, for example

$$w(\mathbf{x}) = f\left(\|\mathbf{A}(\mathbf{x}-\mathbf{s})\| \right) \tag{26}$$

then

$$\nabla_{\mathbf{x}} w(\mathbf{x}) = \nabla_{\mathbf{x}} f\left(\|\mathbf{A}(\mathbf{x}-\mathbf{s})\|\right) = f'\left(\|\mathbf{A}(\mathbf{x}-\mathbf{s})\|\right) \mathbf{A}^T \frac{\mathbf{A}(\mathbf{x}-\mathbf{s})}{\|\mathbf{A}(\mathbf{x}-\mathbf{s})\|} \tag{27}$$

because $\nabla_{\mathbf{x}} \|\mathbf{x}\| = \mathbf{x}/\|\mathbf{x}\|$. We can simplify the equation:

$$\nabla_{\mathbf{x}} w(\mathbf{x}) = f'\left(\|\tilde{\mathbf{x}}\|\right) \mathbf{A}^T \frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|}; \quad \tilde{\mathbf{x}} = \mathbf{A}(\mathbf{x}-\mathbf{s}) \tag{28}$$

3.1.3 Agreements for use of linear (affine) maps

Remark: an old version of this chapter has been saved in the Sandbox in Section 12.1.1.

In the IOptLib, linear (in fact affine) transforms are used for several purposes which include:

- sampling of response functions in a given domain, which can be obtained by transforming a unit ball
- definition of a restricted region constraint, where the constraint function that ensures that the solution is included in the unit ball is subjected to co-ordinate transform
- definition of weighting functions, which are obtained by co-ordinate transforms of rotationally symmetric functions scaled for a unit ball

The above mentioned functions and procedures are the most easily defined and performed when the unit ball centered in the co-ordinate origin is the domain of interest. We define the transform **F** such that

$$\mathbf{x} = \mathbf{F}(\tilde{\mathbf{x}}) = \mathbf{A}\tilde{\mathbf{x}} + \mathbf{s}, \tag{29}$$

or

$$\tilde{\mathbf{x}} = \mathbf{F}^{-1}(\mathbf{x}) = \mathbf{A}^{-1}(\mathbf{x}-\mathbf{s}). \tag{30}$$

Affine transform **F** transforms a unit ball centered in the co-ordinate origin to an hyper ellipsoidal region with a center of mass **s** (Figure 2). In optimization methods that utilize successive approximations of the response, such domains are conveniently used for sampling of the response and as restricted region on which the approximated problem is solved in the current iteration, therefore also the sampling weights are defined in such a way that influence of samples on the approximation is significant in the domain of the same shape, centered around the corresponding samples.

The (closed) unit ball is defined as

$$U = \left\{ \mathbf{x} ; \| \mathbf{x} \|_2 \le 1 \right\}. \tag{31}$$

The ellipsoidal domain obtained by transformation of the unit ball by **F** is therefore

$$U_{\mathbf{F}} = \left\{ \mathbf{x} ; \left\| \mathbf{F}^{-1}(\mathbf{x}) \right\|_2 \le 1 \right\}. \tag{32}$$

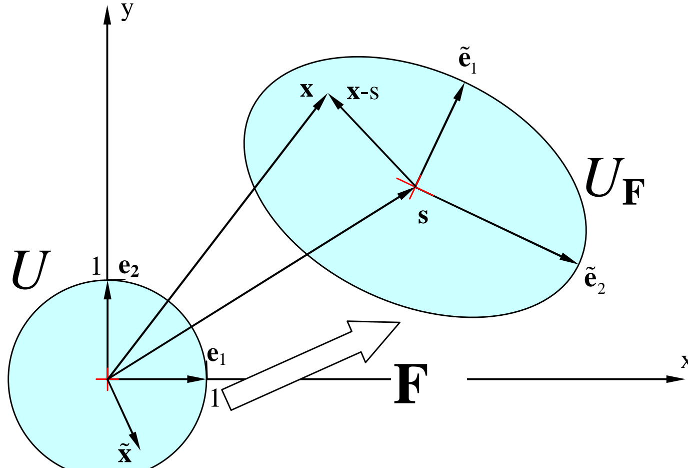


Figure 2: Affine function **F** that maps n unit ball into an ellipsoidal domain centered around **s**.

Sampling is typically done such a way that the specified number m_s of random points with uniform probability density over volume of the unit ball are generated, say $\tilde{\mathbf{x}}_i$. These points are then transformed to $\mathbf{x}_i$ by

$$\mathbf{x}_\mathrm{i} = \mathbf{F}(\tilde{\mathbf{x}}_i). \tag{33}$$

In most cases it is more convenient that the sampling points are uniformly distributed over volume in the unit ball rather than the transformed ellipsoidal domain, which can be very elongated. This is even more obvious when we obtain the samples by solution of the minimal particle potential problem[1]. If the minimal potential problem was used on the ellipsoidal domain that is expressively elongated along one main axis, we would obtain almost uniform distribution along this main axis and a meaningless zigzagging in other directions. When we want to include previously chosen

[1] This ensures that the particles are as far away from each other as possible and they are not concentrated in any part of the sampling domains, which can happen by random sampling.

sampling points $\mathbf{y}_k$ in the minimal particle potential problem (in order to avoid oversampling of parts of the domain), these points are first transformed by inverse transforms into

$$\tilde{\mathbf{y}}_k = \mathbf{F}^{-1}\left(\mathbf{y}_k\right). \tag{34}$$

Then the necessary number m of $\tilde{\mathbf{x}}_i$ are obtained from randomly distributed points in the unit ball (say $\tilde{\mathbf{x}}_i^{(0)}$) from solving the minimal particle potential problem involving also the points $\tilde{\mathbf{y}}_k$. Points $\tilde{\mathbf{x}}_i$ are then transformed to $\mathbf{x}_i$ by $\mathbf{F}$.

Restricted region constraints are defined by transforming independent variables of constraint function that correspond to limiting the domain to the unit ball. For optimization purposes, the unit ball constraint is conveniently defined as

$$\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} = \|\tilde{\mathbf{x}}\|_2^{\ 2} \le 1. \tag{35}$$

The corresponding constraint function is

$$c_U\left(\tilde{\mathbf{x}}\right) = \|\tilde{\mathbf{x}}\|_2^{\ 2} - 1 = \tilde{\mathbf{x}}^T\tilde{\mathbf{x}} - 1. \tag{36}$$

If we want to limit the domain of optimization to the ellipsoidal region obtained from the unit ball by $\mathbf{F}$, we must apply c_U to variables transformed by $\mathbf{F}^{-1}$ because this function transforms the domain of interest to the unit ball (Figure 2). Therefore, the constraint function corresponding to the restricted region constraint is

$$c_r\left(\mathbf{x}\right) = c_U\left(\mathbf{F}^{-1}\left(\mathbf{x}\right)\right). \tag{37}$$

According to (28) and taking into account (36) and (37), gradient of c_r is:

$$\nabla_{\mathbf{x}} c_r\left(\mathbf{x}\right) = \mathbf{A}\,\frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2};\ \tilde{\mathbf{x}} = \mathbf{A}^{-1}\left(\mathbf{x} - \mathbf{x}_i\right)\ . \tag{38}$$

Because sampling is performed inside the ellipsoidal domain obtained by application of $\mathbf{F}$ to the unit ball, it seems reasonable that contours of **weighting functions** corresponding to individual samples will have similar shapes as this domain, but will be centered around the corresponding samples. Therefore we can use a similar idea for weighting functions as for the restricted region constraint function. We define a template weighting function $\mathbf{w}_U\left(\tilde{\mathbf{x}}\right)$ with concentric contours, which decays considerably on the distance 1 from the origin. Actual weighting functions are then obtained by applying the template weighting function to co-ordinates transformed by $\mathbf{F}_i^{-1}$, where $\mathbf{F}_i$ is a function that transforms the unit ball to an ellipsoidal domain centered around the corresponding sampling point. For sampling point $\mathbf{x}_i$ the corresponding function is

$$\mathbf{F}_i(\tilde{\mathbf{x}}) = \mathbf{A}\ \tilde{\mathbf{x}} + \mathbf{x}_i. \tag{39}$$

The weighting function corresponding to the sample $\mathbf{x}_i$ is then

$$w_i(\mathbf{x}) = w_U\left(\mathbf{F}_i^{-1}(\mathbf{x})\right). \tag{40}$$

Because the template weighting function has concentric contours, it can be defined by a function of a single variable $w(x)$, i.e.

$$w_U(\tilde{\mathbf{x}}) = w\left(\left\|\tilde{\mathbf{x}}\right\|_2\right), \tag{41}$$

The weighting function corresponding to the sampling point $\mathbf{x}_i$ is therefore

$$w_i(\mathbf{x}) = w\left(\left\|\mathbf{F}_i^{-1}(\mathbf{x})\right\|_2\right) = w\left(\mathbf{A}^{-1}(\mathbf{x} - \mathbf{r}_i)\right). \tag{42}$$

Function w needs to be defined only for non-negative arguments. We usually require that gradient of $\mathbf{w}_U$ is continuous in the co-ordinate origin, which means that w must have a zero derivative in 0. Commonly used forms for w are Gaussian and reciprocal polynomial (Figure 3):

$$\begin{aligned} w_G(r) &= e^{-r^2}, \\ w_p(r) &= \frac{1}{1+|r|^p},\ p = 2, 3, 4, \ldots \end{aligned} \tag{43}$$

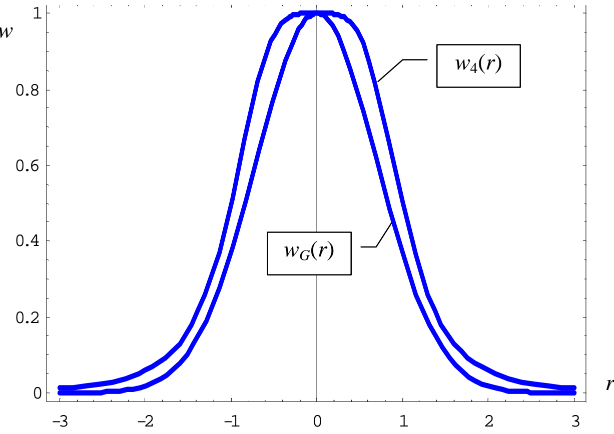


Figure 3: Weighting functions of Gaussian form $w_G(r)$ and reciprocal polynomial form $w_4(r)$.

According to (28) and taking into account (42) gradient of the weighting functions are:

$$\nabla_{\mathbf{x}} w_i(\mathbf{x}) = w'\left(\|\tilde{\mathbf{x}}\|_2\right) \mathbf{A}^{-1} \frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2};\ \tilde{\mathbf{x}} = \mathbf{A}^{-1}(\mathbf{x} - \mathbf{x}_i)\ . \tag{44}$$

3.1.4 Implementation of linear and affine maps

3.1.4.1 Transformation type

Several rules for handling affine (or linear) maps and transformations are implied by the transformation data type and the associated functions. These rules must be followed when implementing lower level function that operate with this data type. The present Sub-section describes the transform types and defines some basic rules.

The user usually uses high level functions and in this way he or she does not interact too much with the basic rules. Most of the users can therefore read only the first part of this section and then skip to the Section 3.1.4.2, which describes the higher leveluse of linear and affine transformations.

The transform data type is defined as

```
typedef struct _lintransfdata {
  int type,id;      /* type and unique object ID */
  int lock;    /* object locking support (to synchronize access in threads) */
  /* Dimensions of the first and the second space, i.e. of inverse
  transformation matrix Ainv: */
  int d1,d2;
  vector shift;    /* translation vector */
  double *a_scal;  /* multiplicative factor for transf. matrix */
  vector a;    /* components od diagonal transformation matrix in the
      domain of transformation (there is no such thing in codomain of
      transformation). If (...)->A is also defined then diag(a) is left
      multiplied when performing transformation. */
  vector ainv;  /* Inverse of a (components are reciprocal comp. of a) */
  matrix A;    /* transformation matrix */
  unsigned long A_flags;    /* flags describing properties of A */
  /* Spectral decomposition: */
  vector A_sd;  /* components are eigenvalues of a */
  matrix A_sU;  /* columns are eigenvectors of A */
  /* QR decomposition: */
  matrix A_Q;   /* orthogonal factor of QR decomposition of A */
```

```
    matrix A_R;   /* upper triangular factor of A */
    /* LDLT and UL decomposition: */
    matrix A_L;   /* lower triangular factor in LDLT decomp. */
    matrix A_U;   /* upper triangular matrix in LU decomp. */
    vector A_D;   /* diagonal factor in LDLT decomp. */

    /* Inverse of A: */
    matrix Ainv;

    matrix Q;   /* coefficients of quadratic form */
    vector q;   /* coefficients of pure quadratic form (Q diagonal) */
    vector dQ;  /* coefficients are eigenvalues of Q */
    matrix UQ;  /* columns are eigenvectors of Q */

    /* Function definition that enables direct use fo transform for
    composition of functions: (consider whether this is a good solution; maybe
    the opposite approach would be better - standardize the form of the
    definition data for functions such that they include the lintear
    transformation structure) */
    int functype;
    void *func;
    void *funcdata;
    void (*dispfuncdata) (void **);
    double *val;    /* used when func==NULL */
    vector gradval; /* used when func==NULL */

    /* Auxiliary vectors & matrixes: */
    vector
      vecaux1,  /* dim. (...)->d1 */
      vecaux2,  /* dim. (...)->d2 */
      vecauxinv1,
      vecauxinv2;
    matrix
      mataux1,
      mataux2,
      matauxinv1,
      matauxinv2;
    /* Auxiliary storage of matrices & vectors: */
    stack matstore;
    stack vecstore;
  } *lintransfdata;
```

The usual rule applies that everything what is on the structure should be dynamically de-allocated together with it (when de-allocation is called, see Section 3.1.4.2), except for the fields for which de-allocation functions are explicitly specified (in this case, a corresponding NULL de-allocation function means that the corresponding pointer will be de-allocated elsewhere).

The structure holds all the data that is directly bound to the maps (divided to *definition and derived data*) as well as auxiliary storage data for functions that use the structure. For *derived data* that is bound to parameter transform (such as factors of various decompositions of the transformation matrix), there is a rule that if these data are allocated then they must also be correctly calculated according to the current definition data. Whenever definition of the map is changed, any allocated derived data must be either re-calculated or de-allocated. In order to safely imply this rule,

only the set of functions that are provided by the module should be used. If new functionality is required, then it should be implemented by using the function from the module or new functions should be implemented within the module that strictly follow the rules.

Some derived quantities (such as factors of decompositions) may be calculated automatically when needed within functions that operate with the map.

The affine map that is represented by the data structure specified above is

$$\mathbf{y}_{(m\times 1)} = \mathbf{A}_{(m\times n)}\,\mathbf{x}_{(n\times 1)} + \mathbf{b}_{(m\times 1)} \;. \tag{45}$$

Dimensions are contained in the following fields of the transformation data structure:

```
d1=n=A->d2
d2=m=A->d1
```

Definition data:

Fields `shift`, `a_scal`, `A` and `a` are considered basic definition fields of the transformation.
However, `A` or `a` may be replaced by its decomposed form (e.g. factors `A_Q` and `A_R`), and `a_inv` of `A_inv` may define the inverse of **A** instead of **A** itself. These are all legal situations that must be gandled by the function working with transformations.

Basic definition fields define **s** (=`shift`) and the *transformation matrix* **A** from equation (14). The following rule for definition of **A** is used:

$$\mathbf{A} = \mathbf{A}'\ (\texttt{a_scal} \mid 1) = (\,\texttt{A} \mid \mathbf{I}\,)\ (\,\mathrm{diag}(\texttt{a}) \mid \mathbf{I}\,)\ \ (\texttt{a_scal} \mid 1)\,. \tag{46}$$

This means that any of the factors `a_scal`, `a` and `A` can be defined or not and if they are not defined then the corresponding unity for multiplication is taken instead. The *transformation matrix* is defined as product of all these factors. It is important that the eventual diagonal factor diag(`a`) (when defined) is right-multiplied with the matrix factor `A`, which means that when a map is applied to a vector, it is first multiplied by the diagonal factor.

The scalar factor provides the possibility of isotropic scaling apart from multiplication by `A`, which can be useful e.g. in restricted step algorithms. The diagonal term allows scaling of each component separately, but in the domain of the map (not in co-domain). This can be useful if we deal with very badly scaled physical quantities (i.e. numbers corresponding to different quantities are several orders of magnitude different) and scaling is used as a tool for pre-conditioning numerical operations that are performed on these quantities. Another benefit of using `a` is boosting efficiency when the transformation matrix is actually diagonal (in this we don't define `A`, and because of this all matrix computations actually fall away).

Of course, actual computations are performed in a more efficient way than it could be concluded from (46). If some factor of this equation is not defined (i.e. it is considered unit for multiplication in the equation) then the corresponding multiplication is skipped.

It is important to remember that[1]

$$\mathbf{A} = c\,\mathbf{A}' \tag{47}$$

where

$$c = (\texttt{a_scal} \mid 1) \tag{48}$$

Derived data: (to be supplemented)

`Ainv` holds inverse of **A**' (i.e. not of actual **A** , but of **A** divided by c , which is a multiplicative factor that is defined by the field `a_scal` or is 1 when this field is NULL). If **A**' (and thus **A**) is diagonal (which is when `a` is specified) then inverse of **A**' is not kept and `Ainv` will be NULL (because it is very simply calculated).

Field `UA` holds an orthogonal matrix whose columns are normalized eigenvectors of **A**' ($\mathbf{U}_\mathbf{A}$ in equation (16)) and dA holds a vector of corresponding eigenvalues of **A**' ($(1/c)\mathbf{D}_\mathbf{A}$). Again, if **A**' is diagonal (`a` is specified) then fields UA and dA will be NULL, since in this case $\mathbf{U}_\mathbf{A}=\mathbf{I}$ and $\mathbf{D}_\mathbf{A}$=diag(`a`).

Auxiliary data:

Auxiliary matrices and vectors are used in order to save time for allocation and de-allocation on account of additional space that remains allocated until it is released (explicitly or implicitly, e.g. when the transformation matrix is de-allocated). For example, when a vector is transformed, it can be multiplied by a matrix that can consist of several factors (equation (46)). Successive multiplications can yield vectors of different lengths, which can not be stored at one location. Therefore, when necessary, the auxiliary matrices and vectors are used for storing immediate results of operations.

Auxiliary matrices and vectors are normally used only by lower level operations and users of the library will not have to deal with them. The rule is that auxiliary matrices or vectors are allocated the first time they are needed. They can then be de-allocated explicitly or implicitly when the whole transformation data structure is de-allocated. Programmers of lower level functions that use auxiliary matrices and vectors must be very careful about the dimensions. Objects of appropriate dimensions must be used (this is not an issue when the domain and co-domain of the map have the same dimensions), otherwise the necessary resizing would annihilate the benefits gained by using these objects. When used in functions dealing with the transformation, dimensions must be checked because it is not guaranteed that they will match (i.e. when the transformation is re-defined in such a way that dimensions of the transformation matrix change, it is not guaranteed

[1] A question is "Why to specify a separate scaling factor while it could be included in **A** in which case special treatment of **A**' is not necessary?" Well, it is always possible not to define the field `a_scal`, in which case the two matrices are equal. Reason for introducing an optional scaling factor is that derived data can be re-used when the transformation matrix is only scaled, which is often beneficial. Such treatment causes some negligible computational overhead (e.g. checking whether additional scaling factor is defined). There is also some implementation overhead, but this should not be a problem because all the basic tools needed to handle transformations are already there and implemented consistently.

that previously allocated auxiliary matrices and vectors are resized or de-allocated). Table 2 lists dimensions of the auxiliary matrices and vectors.

Table 2: Auxiliary storage with dimensions. Note that vectors that do not have the suffix *inv* are defined in the domain of the map and have dimension `d1`, and those with suffix inv are defined in its codomain and have dimension `d2`. Matrices that do not have the suffix *inv* act on (can be left multiplied with) vectors in the domain of the map and thus have dimensions `d2*d1` (the same as the transformation matrix **A**), and matrices with suffix *inv* have dimensions `d1*d2`.

Field	Dimensions	Remarks
Auxiliary vectors and matrices		
`vecaux1`	`d1`	
`vecaux2`	`d1`	
`vecauxinv1`	`d2`	
`vecauxinv2`	`d2`	
`mataux1`	`d2*d1`	
`mataux2`	`d2*d1`	
`matauxinv1`	`d1*d2`	
`matauxinv2`	`d1*d2`	

For other temporary storage, there are stacks of matrices `matstore` and stack of vectors `vecstore`. Matrices and vectors for temporary auxiliary storage should be popped from these stacks and pushed back to them when not needed any more. Derived matrices and vectors are typically stored in these stacks.

Lock:

Field `lock` is intended for locking the object in order to prevent other threads to use it. Functions that use the data on the object (either for reading or modification) should lock it, but should also instruct the called lower level utility functions not to lock the object themselves by the appropriate argument, or call function version that do not perform locking[1]. After locking, the object must always be unlocked (see Sub-section 9.1.3)..

Table 3: Fields of the **`lintransfdata`** type.

Field	Meaning	Remarks

[1] Usually, utility functions that deal with objects of type `lintransfdata` have an extra argument that indicates whether the object should be locked before access to its data or not. Some functions can not have this extra argument because the function form is prescribed. In such cases, there are usually two versions of a function, one that performs locking and one that does not.

Definition data		
a_scal	c , eq. (47)	c=1 if NULL
a	**A'**=A diag(a)	
A	**A'** if a==NULL	If a!=NULL, a is used
Derived data		
Ainv	$\mathbf{A}'^{-1} = c\,\mathbf{A}^{-1}$	
UA	$\mathbf{U_A} = \mathbf{U_{A'}}$ (16)	
dA	$(1/c)\,\mathbf{D_A} = \mathbf{D_{A'}}$=diag(dA) (16)	

3.1.4.2 Management utilities

This section describes allocation and de-allocation of transformation objects, definition of the transformation parameters, enforcement of calculation of auxiliary data such as inverse transformation matrix or factors of decomposed transformation matrix, etc.

3.1.4.3 Mathematical operations

3.1.5 Restricted step constraints

3.1.5.1 Introduction

We will try to express the restricted region constraints in such a way that in some other co-ordinates it is reduced to the unit ball constraint (57). With other words,

$$c(\mathbf{x}) = c_u(\mathbf{F}(\mathbf{x})) = \|\mathbf{F}(\mathbf{x})\|_2^2 - 1 \le 0 . \qquad (49)$$

Here **F** represents a co-ordinate transformation, in particular we are interested in Affine transformations of the form (14)

$$\tilde{\mathbf{x}} = \mathbf{F}(\mathbf{x}) = \mathbf{A}(\mathbf{x} - \mathbf{s}) \qquad (50)$$

with inverse transformation

$$\mathbf{x} = \mathbf{F}^{-1}(\tilde{\mathbf{x}}) = \mathbf{A}^{-1}(\tilde{\mathbf{x}}) + \mathbf{s} . \qquad (51)$$

The feasible region constraint is then

$$\|\mathbf{A}(\mathbf{r} - \mathbf{s})\|_2^2 \le 1 \qquad (52)$$

Feasible region of the constraint (49) with **F** of the form (14) is shown in Figure 4. In the transformed co-ordinates, the constraint transforms to unit ball constraint. Therefore, **F** must transform the feasible region to unit ball, and $\mathbf{F}^{-1}$ transforms an unit ball to the feasible region.

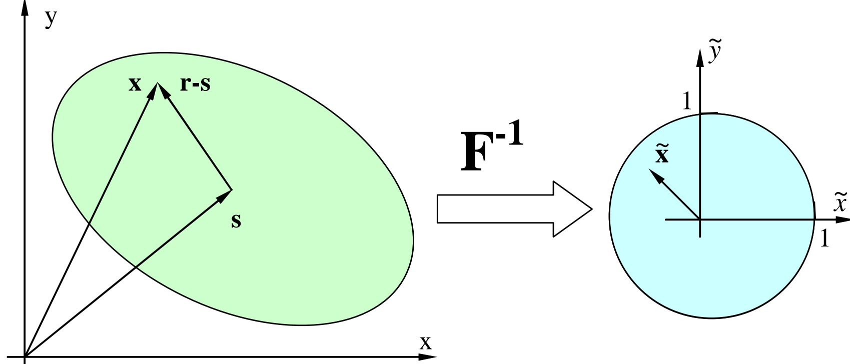


Figure 4: Feasible region of the restricted step constraints in the original (left-hand side) and transformed (right-hand side) co-ordinate system.

Some examples of definition of the feasible region:

Now let us say that the restricted region (i.e. feasible region of the restricted step constraint) is a circular region with radius *r* centered around **s**. If we want **F** to transform the restricted region to the unit ball then **A** must be of the form

$$\mathbf{A} = \frac{1}{r}\mathbf{I}. \tag{53}$$

If the restricted region is ellipsoid with main axes parallel to the co-ordinate axis and with half-axes $\{r_1, r_2, ..., r_n\}$ then **A** has the form

$$\mathbf{A} = diag\left[\frac{1}{d_1}, \frac{1}{d_2}, ..., \frac{1}{d_n}\right]^T = \begin{bmatrix} \frac{1}{d_1} & 0 & ... & 0 \\ 0 & \frac{1}{d_2} & ... & 0 \\ ... & & ... & \\ 0 & ... & 0 & \frac{1}{d_n} \end{bmatrix} \tag{54}$$

Further, let us say that **Q** is a *symmetric positive definite matrix* and restricted region is defined as a set of point for which a quadratic form of the form (18), but centered in **s**, is less of equal to 1:

$$Q(\mathbf{x}-\mathbf{s}) = (\mathbf{x}-\mathbf{s})^T \mathbf{Q}(\mathbf{x}-\mathbf{s}) \le 1 \tag{55}$$

If we write **Q**=$\mathbf{A}^T\mathbf{A}$ then we have

$$Q(\mathbf{x}-\mathbf{s}) = (\mathbf{A}(\mathbf{x}-s))^T \mathbf{A}(\mathbf{x}-s) = \|\mathbf{A}(\mathbf{x}-s)\|_2^2 \le 1,$$

which is precisely (52). Since **Q** is symmetric and positive definite, **A** is also symmetric, therefore constraint (55) corresponds to the constraint (52) if

$$\mathbf{Q} = \mathbf{A}^2. \tag{56}$$

Form (55) may be very useful since the constraint is bound to the value of quadratic form, which may be e.g. the quadratic term of a quadratic approximation of some function and to which it is sensible to tie the restricted step constrain. Obviously, a problem appears when **Q** is not positive definite, since in this case the feasible domain stretches infinitely long in some cases. What we will usually do is to make eigentransform of **Q**, change the sign of eventual negative eigenvalues and set a lower bound for small eigenvalues. In this way, a new **Q** is performed that is positive definite and has eigenvalues bounded below. It may be sensible, for example, to accept the smaller eigenvalues as a given portion of the maximal eigenvalue. If parameters are very badly scaled then we might

perform pre-scaling. This will enable more sensible determination of the largest (by absolute value) eigenvalues[1].

3.1.5.2 Implementation

3.1.5.3 Unit ball constraint

A special case of the restricted region constraint is the unit ball constraint. This constraint requires that parameter vector must be contained in a zero centered unit ball, i.e. $\|\mathbf{r}\| \leq 1$ or formally,

[1] The ratio between individual eigenvalues of response Hessians will change over the design space, but by bad scaling we can promote expressive dominance of eigenvalues whose eigenvectors are parallel to over-scaled co-ordinate directions.

$$c_1(\mathbf{r}) = c_u(\mathbf{r}) = \|\mathbf{r}\| - 1 \le 0 \,. \qquad (57)$$

This constraint is implemented by the analysis function `unitballconstr`, which defines a single constraint and no objective functions.

3.1.6 Restricted region constraints - old implementation

The domain constraint is implemented by the following analysis function:

```
int restrictedstepconstr_olf(vector param,int *calcobj,double **addrobj,
    int *calcconstr,stack *addrconstr,
    int *calcgradobj,vector *addrgradobj,
    int *calcgradconstr,stack *addrgradconstr,void **cd);
```

The function is of a standard type for analysis function, i.e. *analysis_bas_f* . Parameter **cd** is a pointer to the data that contains the parameters defining the restricted region. It must be a table of at least 5 pointers with the following meaning:

cd[0]: **s** (type *vector*)
cd[1]: *r* (type *double* *)
cd[2]: **d** (type *vector*)
cd[3]: **A**
cd[3]: Flags (type *int* *), default 0.

4 MODIFICATION AND TRANSFORMATION OF OPTIMIZATION PROBLEMS

This section describes utilities for setting up definition of an optimization problem by modification, combination or both of one or several other definitions. It is understood that definition of an optimization problem consists of definition of its objective and constraint problems, which is in the IOptLib done by defining a standard analysis function and eventually its definition data (Section 2.3.1).

In IOptLib, tools for several common ways of definition of optimization problems by combination and modification of other definitions are implemented. This is of particular importance for construction of complex approximation based algorithms on which much focus of the library is put in its initial stage of development, however it is also very useful for construction of other classes of algorithms such as penalty algorithms, all kinds of restricted step algorithms, and several other algorithms such as the NLP Simplex algorithm[1].

4.1 Combining objectives and constraints defined by different analysis functions

The following function calculates response functions as combination of response functions defined by several analysis functions and their data:

```
int combinedanalysis(vector param,int *calcobj,double **addrobj,     int
     *calcconstr,stack *addrconstr, int *calcgradobj,vector *addrgradobj,
     int *calcgradconstr,stack *addrgradconstr, stack cd);
```

The function is of standard analysis function type `an_bas_f` , only that the last argument is a stack containing definitions of individual analysis and the corresponding parametric data. This mechanism enables defining new problem on the basis of simpler problem, e.g. one can combine the objective function one problem, defined by its analysis function and data, with the constraints of another problem.

Elements of the stack `cd` are pointers to data structures, which are of type `ancombstruct` defines as follows:

```
typedef struct _ancomb {
    int type,id;      /* type and unique object ID */
    int flags;        /* flags defining how response functions are combined */
    double factor,
           shift,    /* shift and scaling factor of values */
      constrfactor,  /* weight. factor for constraints (if 0 then factor is
             taken) */
      constrshift;   /* shift for constraint functions */
    int nparam,nobj,nconstr;
    analysis point anpt;    /* storage of analysis results */
    analysis_bas_f anfunc;  /* analysis function */
    void *andata;  /* analysis definition data */
    vector auxvec;  /* auxiliary vector */
} *ancomb;
```

There are tools for creating the definition data (argument `cd`) for the combined analysis function `combinedanalysis`) and are described in the Subsection 4.1.1.1 below.

Field `flags` is an or-ed combination of individual basic flags that define how the response of an individual analysis is combined to form the overall response. The basic flags are also defined as macros in `optbas.h`:

- `ANCOMB_SUMOBJ` – objective function of this analysis, shifted by *shift* and then multiplied by *factor*, is summed to the objective function of the combined analysis (formed anew if necessary)
- `ANCOMB_SUMCONSTR` – constraint functions of this analysis, shifted by *constrshift* and then multiplied by *constfractor*, are summed to all constraints of the combined analysis (formed anew if necessary)
- `ANCOMB_SUMOBJTOCONSTR` – objective function, shifted by *shift* and then multiplied by *factor*, is summed to all constraint functions of the combined analysis

`ANCOMB_SUMCONSTRTOOBJ` – constraint functions of this analysis, shifted by *constrshift* and then multiplied by *constrfactor*, are summed to the objective function of the combined analysis (formed anew if necessary)
`ANCOMB_ADDCONSTR` – constraint functions of this analysis, shifted by *constrshift* and then multiplied by *constrfactor*, are added as new constraints to the combined analysis (formed anew if necessary)
`ANCOMB_ADDOBJTOCONSTR` – objective function of this analysis, shifted by *shift* and then multiplied by *factor*, is added as a constraint function to the response of the combined analysis.

The following is not implemented yet:
`ANCOMB_PENALTYSQR – Not implemented yet. Expected behavior:` A square function of constraint functions of this analysis, multiplied by *factor* and shrinked by shift (such that shift equals 1) are added to the objective function of the combined analysis.

shifted by *shift*, are added up to the corresponding constraints of the combined analysis, e.g. (formed anew if necessary)
`ANCOMB_PENALTYSQRHALF - Not implemented yet.`
`ANCOMB_SUMPENALTYADAPT - Not implemented yet.`
`ANCOMB_SUMPENALTYEXP - Not implemented yet.`

Combination of response functions is an important mechanism, used e.g. in the following situations:

- To add constraints which restrict the step size in the restricted step method, when solving the restricted approximate sub-problem
- To make a weighted sum of objectives
- To form constraints from the objectives, e.g. in the minimal potential energy problem of charged particles, we can specify that the domain constraint is defined by some specific contour of the Rosenbrock problem.

The `combinedanalysis` function first runs all individual analyses and checks the corresponding flags in order to calculate the number of constraints and whether the objective function is defined or not. This is then used in the allocation of space for the result of the combined analysis, which is performed by the prepanfuncbas function (Sub-section 2.3.2).

Finally the results of individual analyses are combined in order to calculate the results of the overall combined analysis. First, the objective function and its gradient (if applicable according to whether the combined objective function is defined and according to the flags) are initialized to zero. Next, all results (either from objectives or constraints) of individual analyses are picked that define the constraints of the combined problem, and stored at appropriate locations. Finally, the values that should be summed to constraint or objective functions are picked and added to the current values.

Remarks

The mechanism of combination of analyses is implemented in a very general way. Usually only simple combinations will be used, e.g. combination of response defined by one analysis function with constraints defined by another one (typical example for this is addition of restricted region constraints to the approximated analysis). The purpose of generality is to implement all possible combinations once for always and in one place in order to reduce complexity for users of the library. The price paid for that is complex implementation and an inevitable fact that some uncommon combinations will not be tested for a long time. This makes the possibility of hidden

errors high. When using uncommon combinations, please test the functionality of combined analysis before using it, and in the case of discovered errors inform the developers of the library. Operation of the ***analysis combination system can be tested*** in advance by the **`testancomb`** function. One should see the source code of the function for details.

Addition of penalty terms in place of constraints *is not yet implemented*, therefore the flags ANCOMB_PENALTYSQR, ANCOMB_PENALTYSQRHALF, ANCOMB_PENALTYADAPT, and ANCOMB_PENALTYEXP may not be used. ***It is not yet decided whether the addition of penalty terms will be supported by the mechanism for combining analyses or not***. Currently it is still possible that this will be handled by a separate mechanism.

For individual analysis, it is *illegal to set the flags that would imply calculation of given response functions* of that analysis *if these functions are not defined*. For example, if the objective function is not defined for a given individual analysis, the flags ANCOMB_SUMOBJ, ANCOMB_SUMOBJTOCONSTR, and ANCOMB_ADDOBJTOCONSTR may not be set for this analysis.

A specific response of an individual analysis can have more than one role. For example, an objective function can be added to the objective function of the combined response and ad the same time represent a constraint in the combined response. Although admissible, such use is not encouraged. If we really need such combined response, it is advisable to include that particular analysis twice, each time by defining another role for the objective function of the response. When doing this, one must take care about specifying whether the analysis definition data is de-allocated together with the stack defining the combined data, in order to de-allocate the same definition data only once.

4.1.1.1 Preparation of the combined analysis

```
stack combcd=NULL;
addancomb(&combcd,…)
```

De-allocation of the stack:

```
dispstackallspec(&combst,(void (*) (void **)) dispancomb);
```

4.1.1.2 Open questions for the mechanism of combined analyses

The first open question for combining analyses is how to treat those kinds of individual response that are not defined for some analysis function, but are assumed by the flags. For example, one of the analyses involved in the evaluation of combined response does not define the objective function, but its flags request that the objective function of this analysis is added to the objective function of the combined problem. Currently this situation will generate an error that will be reported by the function `combinedanalysis` . On one hand this imposes less freedom (e.g. the user of the system must take care about which response is defined for which problem). On the other

hand, this is more strict what regards detection of unintended situations (reflected as errors or other kind of exceptional situations). Currently, we in favor of a stricter system, i.e. flags that are related to calculation of a given type of response (i.e. objective or constraint functions) may be set for a given individual analysis only if this kind of response is actually defined for that analysis, and the opposite is regarded an error.

Another open question is whether to include addition of penalty terms to objective function in the mechanism of combined analysis. One argument to do this is to make the mechanism of combining problem definition very general and versatile. The first argument against this is that the complexity of implementation is increased in this way, but for this specific argument the limit of when the things become too complex to justify the benefits of generality is very intuitive and would probably be defined differently by different problems. A better founded argument against this functionality lies in the penalty business itself. The mechanism for combining response is somehow too weak for the implementation of what we need for construction of any kind of penalty functions. This is because the types of penalty functions must be defined in advance with this kind of mechanism. For true flexibility when defining the penalty function, one should have the ability of arbitrary definition of penalty terms as the function of penalty parameters and constraint functions. Therefore, currently it is more accepted opinion that *construction of penalty functions should not rely on the mechanism of combining different response*, but should be implemented specially for this purpose.

In order to recapitulate: Mechanism of combining analyses and constraints should be used only for simple combinations of objective functions and constraints of individual analyses into response of the combined analyses. Flags that define how response of individual analysis is combined should be consistent with what can actually be calculated by a specific analysis. More complex forms of combination (such as e.g. construction of penalty functions where penalty terms can be arbitrary function of constraints and penalty parameters) must be implemented separately.

4.2 Handling of bound constraints

Sometimes we want to separately specify bound constraints in optimization problems of form (1). In this case, in addition to constraints defined by constraint functions $c_i(\mathbf{x})$ and $c_j(\mathbf{x})$ from equation (1), we have *bound constraints* of the form

$$l_k \le x_k \le r_k \ , \ \ k = 1, ..., n \ . \qquad (58)$$

In the above equation, l_k specify the lower bounds and r_k specify the upper bounds on optimization parameters, and they are arranged in vectors **l** and **k**. In some cases, bounds will be defined only for particular parameters, for some of which only minimal (l_k) or only maximal (r_k)[1] value is defined. For the sake of convenience in implementation of computational procedures, we will use in such

[1] In this notation, letter l is used as "left" and r as "right".

cases the formula (58) as if both bounds are defined, and will set $l_k = -\infty$ or $r_k = \infty$ for those bounds that are not defined[1].

There are various possible reasons for stating bound constraints separately. One reason is just for convenience, since it is easier to define just the vectors of lower and upper bounds than to explicitly state constraint functions that would imply respect of the bounds at problem solution. Another reason is that some algorithms can treat simple bound constraints much more efficiently than other types of (generally nonlinear) constraints, and it is sometimes easy to guarantee within an algorithm that bound constraints are never violated in any point where the response is evaluated.

The Investigative Optimization Library provides the following utilities for handling bound constraints:

- Implicit addition of constraint functions that imply bound constraints
- Addition of penalty terms related to bound constraints
- Transformation of optimization parameters in such a way that bound constraints are satisfied

There is a special data structure of type `boundconstrdata` designed to support the related operations. Each individual utility can be used to build higher level functionality. In addition, a modified analysis function is provided, which modifies the original analysis in such a way that any combination of the above operations is performed, according to specifications on the bound constraint structure. This analysis function will often be used in optimization algorithms that will include separate handling of bound constraints and will provide a high level tool to algorithm designer, which can be utilized for easy and automatic use of pre-implemented utilities for bound constraint handling.

Utilities for handling bound constraints are described in more details below. Example application is the nonlinear constrained simplex algorithm with ability of handling bound constraints, which is described in [1][2].

[1] In computer implementation, infinity will be replaced by some large number. Typically the number above which values are considered infinity will be specified together with lower and upper bounds, or some default value will be assumed such as 10^{20}.
[2] This algorithm also describes application of C^0 exact penalty functions, which can be used in direct search methods such as the simplex algorithm.

4.2.1 Combination of discontinuous penalty functions and transformation of co-ordinates [1]

4.2.1.1 Discontinuous exact penalty functions

When only comparison of the objective function at different parameters is performed by the optimization algorithm (such as e.g. the Nelder-Mead Simplex method for unconstrained nonlinear minimization), the method may under some circumstanced still work in the case of diccontinuity of the objective function. We can add jump discontinuities to the objective function, and this does not affect the efficiency of the method.

Let us denote $f_m(\mathbf{x})$ the modified objective function with added jump discontinuities. As long as

$$\begin{aligned} f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow f_m(\mathbf{x}_1) < f_m(\mathbf{x}_2) \\ f(\mathbf{x}_1) = f(\mathbf{x}_2) &\Rightarrow f_m(\mathbf{x}_1) = f_m(\mathbf{x}_2) \end{aligned}, \qquad (59)$$

the minimum of f_m is the same as the minimum of f. The above relation is valid if we define the modified objective function in the following way (Figure 5):

$$f_m(\mathbf{x}) = \begin{cases} f(\mathbf{x}); \mathbf{x} < c \\ f(\mathbf{x}) + h; \mathbf{x} \geq c \end{cases} \quad ; h > 0 \ . \qquad (60)$$

This function is obtained from f by adding to it a positive constant within the following domain:

$$\Omega^+ = \{\mathbf{x} | f(\mathbf{x}) \geq c\}. \qquad (61)$$

Edge of this domain $\partial \Omega^+$ is the level hypersurface (isosurface in 3D, isoline in 2D) of f.

[1] **Remark: this section needs cleaning in order to match the standards of this document regarding clearness and conciseness.**

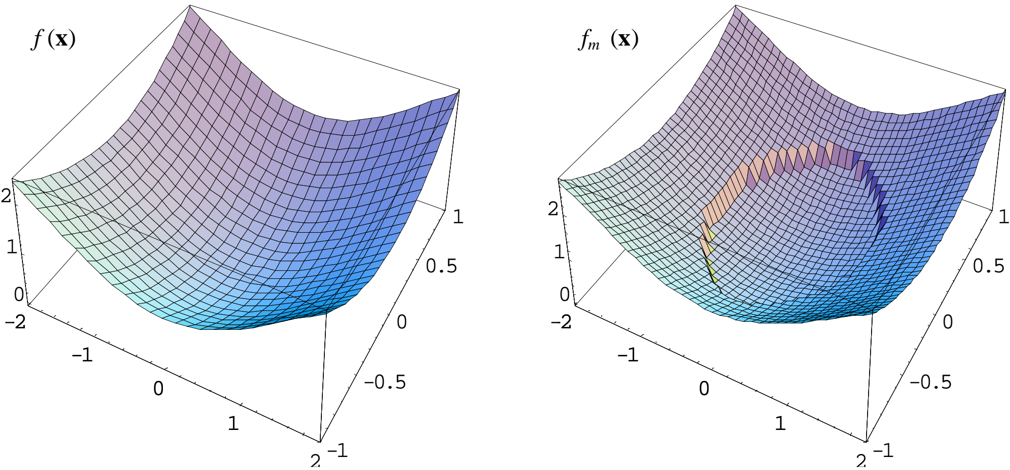


Figure 5: Original and modified objective function.

The fact that modification (60) that introduces a jump discontinuity does not change the performance of the algorithm indicates that the method could be efficiently modified for solution of constraint problems by forming a discontinuous exact penalty function. **Exact penalty function** is obtained by adding a penalty term such that a *minimum of the obtained penalty function corresponds to the constraint minimum of the original problem*. Solution of the original problem can then be obtained by finding a minimum of the penalty function.

When only inequality constraints are present, the penalty function can be formed by addition of penalty terms for each constraint in the following way:

$$f_p\left(\mathbf{x};\mathbf{p}_p\right)=f\left(\mathbf{x}\right)+\sum_{i\in I}h_p\left(c_i\left(\mathbf{x}\right);\mathbf{p}_p\right), \qquad (62)$$

where the penalty terms can be defined for example as

$$h_p\left(c_i,\{k,h\}\right)=\begin{cases}0;\,c_i\le 0\\ h+k\,c_i;\,c_i>0\end{cases};\,h\ge 0\wedge k\ge 0\;. \qquad (63)$$

Non-negative penalty parameter k and h must be large enough if we want that f_p represents an exact penalty function. In the sequel, we define more precisely the conditions that the penalty function is exact penalty function.

We usually require

$$h_p\left(0;\mathbf{p}_p\right)=0\,. \qquad (64)$$

It is clear that in the infeasible region where $\forall i \in I$, $c_i(\mathbf{x}) > 0$, the derivative of h with respect to the violated constraint must be positive, i.e.

$$c > 0 \Rightarrow \frac{\partial h_p(c; \mathbf{p}_p)}{\partial c} > 0 \ . \tag{65}$$

However, this is not a sufficient condition that the penalty function has a local minimum in the solution of the constrained problem. The derivative must be large enough in order to compensate for eventual falling of the objective function as the constraint function grows. What one needs to achieve is that in the infeasible region, the dot product of the gradient of the penalty function with the gradient of any constraint function belonging to a violated constraint, is positive.

The sufficient condition that the penalty function is exact (i.e. it has a local minimum in the solution of the original constrained optimization problem) is the following: There must exist a neighborhood ε of the solution $\mathbf{x}^*$ such that in each point of the neighborhood, the gradient of the penalty function has positive dot product with gradients of all constraint functions which are greater than zero (i.e. belong to violated constraints) in that point. In this way, we can find a neighborhood of $\mathbf{x}^*$ such that a descent path exists from any point in this neighborhood to $\mathbf{x}^*$. The condition can be expressed in the following way:

$$\begin{aligned} &\forall \mathbf{x} \in \varepsilon, \forall i \in I, \\ &c_i(\mathbf{x}) > 0 \Rightarrow \left\langle \nabla_{\mathbf{x}} f_p(\mathbf{x}; \mathbf{p}_p), \nabla c_i(\mathbf{x}) \right\rangle > 0 \ . \end{aligned} \tag{66}$$

The above equation says that the directional gradient of the penalty function must be positive in the direction of the gradient of any violated constraint. From (62) we have

$$\nabla_{\mathbf{x}} f_p(\mathbf{x}; \mathbf{p}_p) = \nabla f(\mathbf{x}) + \sum_{i \in I} \left. \frac{\partial h_p(c; \mathbf{p})}{\partial c} \right|_{c = (c_i(\mathbf{x}))} \nabla c_i(\mathbf{x}) \tag{67}$$

Equation (66) defines the condition that the penalty function has a local minimum that corresponds to the solution of the original constraint optimization problem. From the algorithmic point of view this is not sufficient. We want to ensure that minimization algorithm applied to the penalty function will actually yield the local minimum that corresponds to a local solution of the unconstrained problem (since the penalty function can have several local minima or can even be unbounded below). In our case we will apply the unconstraint Nelder-Mead simplex algorithm, but the same reasoning applies to application of other algorithms. It is intuitively obvious that if the region ε on which (66) holds is larger, the applied minimization algorithm will converge to the solution of the original problem from a larger region. Running the algorithm from a starting point that is far from the region where (66) holds will more likely cause it to diverge (in the case that the penalty function is unbounded below) or converge to a local minimum that is not a solution of the original problem.

The best is if the condition (66) holds everywhere. Considering equations (66) and (67), in order to achieve that, the function $h_p(c;\ldots)$ must grow sufficiently fast with its c. In this way, the second term in (67) can compensate for eventual negative projection of the gradient of the objective function on the gradients of violated constraints. However, making $h_p(c;\ldots)$ grow too fast close to c=0 would introduce ill-conditioning in the minimization of the penalty function. We must therefore look for a suitable compromise, which is not trivial in some cases.

While addition of discontinuous term of the form (60) does not affect the performance of the Nelder-Mead simplex method, addition of penalty terms of the form (62) can significantly reduce its efficiency. This is because the penalty terms limit the space where the simplex moves, and the simplex makes more rejected trials when hitting sharp growth of the penalty function at constraint boundaries.

A disadvantage of the penalty function generated by h_p of the form (63) is that it is difficult to fulfill the condition (66) on a large sub-domains of the infeasible domain in the cases where the objective function falls progressively or when the constraint functions grow regressively with the distance from the zero level hyper-surfaces of constraint functions. This can be alleviated by making h_p grow progressively with increasing positive argument my adding exponential or higher order monomial terms, e.g.

$$h_p\left(c_i,\{k,h\}\right)=\begin{cases}0;\, c_i\le 0\\ h+k\left(c_i+\left(\frac{c_i}{4}\right)^2+\left(\frac{c_i}{8}\right)^3+\left(\frac{c_i}{16}\right)^4+\exp\left(\frac{c_i}{64}\right)\right);\, c_i>0\end{cases};\, h\ge 0\wedge k\ge 0\ . \quad (68)$$

Increasing denominators take care that higher order terms contribute significantly only when the constraint functions are large enough, which makes minimization of the penalty function less ill conditioned. However, this is not so important when the Nelder-Mead simplex method is used for minimization of the penalty function, because this method only makes comparisons of function values and does not make use of higher order function information.

4.2.1.2 Strict respect of bound constraints by parameter transformation

This section describes how violation of bound constraints can be prevented during minimization by the simplex method. This is done by a new analysis function, which shits parameter components that violate bound constraints on interval limits, calculates the objective and constraint functions in new points, and adds a penalty term that depends on how much the constraints were violated.

This procedure should be significantly changed for algorithm that uses function approximations to increase the speed. This is because the procedure introduces discontinuities in the derivatives at constraint bounds.

Let us say that we are solving the problem (1) with only inequality constraints and with additional *bound constraints* on the parameter vector:

$$\forall k, l_k \leq x_k \leq r_k \ . \tag{69}$$

In many cases, the bound constraints are defined only for particular parameters, for some of which only minimal (l_k) or only maximal (r_k)[1] value is defined. For the sake of convenience in implementation of computational procedures, we will use the formula (58) as if both bounds are defined, and will set $l_k = -\infty$ and $r_k = \infty$ for those cases where the bounds are not defined.

Let us say that a direct analysis is called at parameters $\mathbf{x}=\{x_1, x_2, \ldots, x_n\}$ where some of the bound constraints are violated. We actually run the analysis at modified parameters $\tilde{\mathbf{x}}$, which are obtained by correction of actual parameters (at which the analysis is requested) in such a way that which are defined in such a way that bound constraints are satisfied:

$$\forall k, \tilde{x}_k = \begin{cases} x_k \ ; l_k \leq x_k \leq r_k \\ l_k \ ; x_k < l_k \\ r_k \ ; x_k > r_k \end{cases} \tag{70}$$

We then modify the value of the objective function in the following way:

$$\tilde{f}(\mathbf{x}) = f(\tilde{\mathbf{x}}) + \sum_{i=1}^{n} h_{k_l}(\mathbf{x}) + h_{k_r}(\mathbf{x}) \ , \tag{71}$$

where

$$h_{k_l}(\mathbf{x}) = \begin{cases} h_p\left(l_k - x_k ; \mathbf{p}_p\right); l_k > -\infty \\ 0 ; otherwise \end{cases}$$
$$h_{k_r}(\mathbf{x}) = \begin{cases} h_p\left(x_k - r_k ; \mathbf{p}_p\right); r_k < \infty \\ 0 ; otherwise \end{cases} \tag{72}$$

and k_p is a function for generation of penalty terms of a convenient form such as (63) or (68). Constraint functions are not modified and are simply set to the values of constraint functions at $\tilde{\mathbf{x}}$:

$$\forall i \in I, \tilde{c}_i(\mathbf{x}) = c_i(\tilde{\mathbf{x}}) \ . \tag{73}$$

Expression (72) is addition of penalty terms as in (62) ad (63), where the following constraint functions are assigned to bound constraints:

[1] In this notation, letter l is used as "left" and r as "right".

$$\begin{aligned} l_k > -\infty &\Rightarrow c_{k_l}(\mathbf{x}) = l_k - x_k \\ r_k < \infty &\Rightarrow c_{k_r}(\mathbf{x}) = x_k - r_k \end{aligned} . \qquad (74)$$

Penalty terms have the following contributions to the gradient of the objective function:

$$\begin{aligned} \nabla h_{k_l}(\mathbf{x}) &= -\left.\frac{\partial h_p(t;\mathbf{p}_p)}{\partial t}\right|_{t=(l_k - x_k)} \mathbf{e}_k \\ \nabla h_{k_r}(\mathbf{x}) &= \left.\frac{\partial h_p(t;\mathbf{p}_p)}{\partial t}\right|_{t=(x_k - r_k)} \mathbf{e}_k \end{aligned} , \qquad (75)$$

where $\mathbf{e}_k$ is the co-ordinate vector k (component k equals 1, others equal 0).

4.2.2 Implementation remarks on penalty terms and bound constraints

This Section discusses some details relevant for implementation of penalty terms and bound constraints in the *IOptLib* (**Investigative Optimization Library**). It is meant for developers and advanced users of the library because a good knowledge of the library is necessary to understand the section.

We consider modification of the original analysis function according to (71). In principle, the implementation of the modified analysis is quite simple: we form a new analysis function that takes the parameters, calculates the sum of penalty terms according to parameters and bund constraints, modifies the parameters, runs the analysis function at the modified parameters, adds the calculated objective function to the sum of penalty terms to form the modified objective function, takes the calculated constraint functions and returns the results. All the operations could be performed in place, i.e. without allocation of additional space for auxiliary variables.

The scheme is a bit more complicated if one would like to preserve information that is not returned by the modified analysis function, e.g. the modified parameters at which the original analysis function is performed, or the value of the objective function at the modified parameters. In the modified Nelder-Mead algorithm, for example, this information is sometimes desired for checking algorithm progress or for post-processing and analyzing the acquired results. In this case, additional storage is necessary to keep the additional information.

There may be different possibilities with respect to what information should be kept, and modification of analysis defined by (71) can be combined by other modifications such as adaptive penalty functions. Different ways of handling the storage of additional data (together with the appropriate data types) should be implemented in order to optimize the speed and memory usage,

but this would increase the complexity of code and its maintenance costs. In *IOptLib* a compromise solution is achieved by using some standard data types and related functionality. In particular, the type `analysispoint` is utilized that is intended for storage of analysis results. Because of dynamically allocated storage for thing such optimization parameters and values of objective and constraint functions, the amount of additional memory necessary to support comfortable standard uses is not large. Manipulation of additional storage is relatively simple because standard functionality designed around `analysispoint` the type can is used. This functionality can be easily extended in line with the standards when necessary. Beside some additional storage, the cost for using standard data types and procedures is also some additional data transcriptions (e.g. the values of constraint functions are transcribed from the nested (inner) *analysispoint* structure to the outer one).

A scheme for performing the modified analysis function is shown in Figure 6. The structure of data that is passed to the modified function is also shown in the figure.

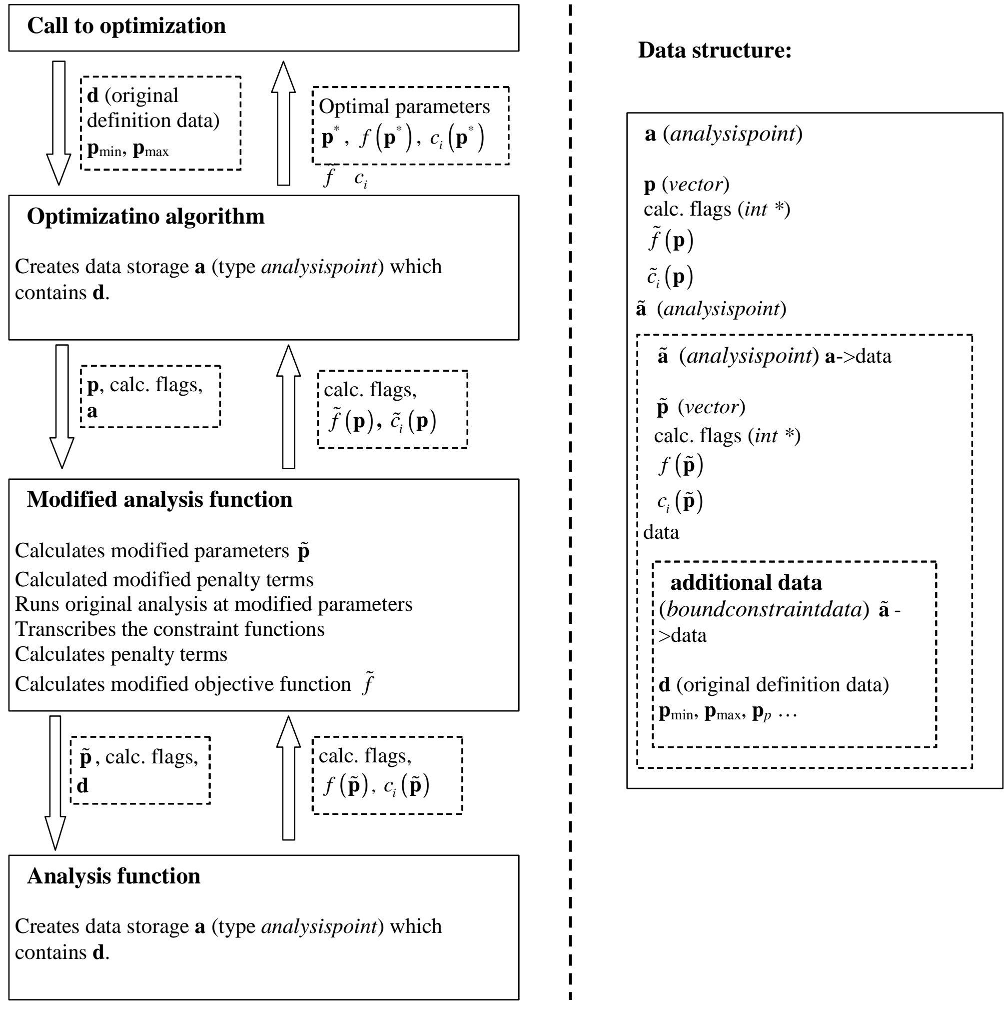


Figure 6: Scheme for handling bound constraints and penalty terms in algorithms.

4.2.2.1 Example: How to prepare bound constraint data and analysis data in an algorithm

This example shows how to prepare the bound constraint data and data for the analysis function in an optimization algorithm that handles bound constraints in a specific way. For most

algorithms where handling of bound constraints would be suitable, the example code can be transferred by slight modifications. For example, different amounts of data can be passed to the algorithm through function arguments, in which way the behavior of the algorithm can be adjusted in more flexible way.

Example 4: Preparation of modified analysis function that handles bound constraints, and its data.

```
...
/* Definitions of variables (some of these can in fact be passed as arguments
            of the function containing this code): */
vector lowbound,upbound;
double bignum;
int numparam,numobj,numconstr;
boundconstrdata bc=NULL;
real_bas_f penfunc=real_bas_f_zero_lin, constrfunc=real_bas_f_lin;
double kpen=1.0, kconstr=1.0;
void (*dispconstrdata) (void **), (*disppendata) (void **);

analysis_bas_f anfunc, anfuncorig;
void *ancd, *ancdorig;
analysispoint anptbc=NULL;
...
...
/* Praparation of bound constraint data: */
prepboundconstrdata(lowbound,upbound,bignum,numparam,numobj,numconstr,&bc);
bc->penfunc=penfunc;
setboundconstrpenpar(bc,0,0,(void *) &kpen,disppendata);
if (penfunc!=NULL)
  bc->transfparam=1;
bc->constrfunc=constrfunc;
setboundconstrconstrpar(bc,0,0,(void *) &kconstr,dispconstrdata);
...
...
/* Preparation of modified analysis function that handles the parameter
            bounds: */
if (lowbound!=NULL || upbound!=NULL)
{
  prepandata_boundconstr(anfuncorig,ancdorig,NULL,bc,&anptbc);
  anfunc=anfunc_mod_boundconstr;
  ancd=anptbcpen;
} else
{
  anfunc=anfuncorig;
  ancd=ancdorig;
}
...
...
/* Use of the analysis function anfunc and its definition data ancd for
            response evaluation within the algorithm… */
...
...
/* Cleaning part: */
boundconstrdata (&bc);
dispanalysispoint (&anptbc);
```

In the first part of the above example, the basic variables used are declared. Some of these variables will usually be passed as arguments of the function that implements a particular optimization algorithm where bound constraints are handled by a modified analysis function, however this is not shown for simplicity. Declarations are only shown in order to present the data types used.

In the second part, the bound constraint data is prepared. The data structure of type *boundconstraintdata* will contain data that defines the bounds (i.e. the vectors *lowbound* and *upbound* and the number *bignum*), as well as data that specifies how to handle bound constraints. It also contains space for auxiliary storage for the operations (such as storage of constraint functions generated out of bounds), which is used by operations that are automatically performed in the modified analysis functions.

The third part contains a typical preparation of the modified analysis function that will handle bound constraints. Instructions for handling the bound constraints are on *bc* that has been prepared before, and the definition data for the modified analysis function is prepared according to the scheme in Figure 6 (right-hand side), using the existing *bc*. The definition data will be a pointer *anptbc* of type *analysispoint*, whose *data* field contains another pointer of type *analysispoint*, and the *data* field of this structure (i.e. *anptbc->data->data*; note the necessary data casts, because *data* is of type void *) will contain the pointer to bound constraint definition data *bc*.

Preparation of bound constraint data:

In this stage only the data that is related to bound constraints and their handling is prepared. The part where the modified analysis is defined can follow immediately, but these parts can also be separated. An example of preparation of bound constraint data can be found in IOptLib function `NLPSimpboundbas()` of the module `optsimp.c`.

In the line

```
prepboundconstrdata(lowbound,upbound,bignum,numparam,numobj,numconstr,&bc);
```

the bound constraint data structure *bc* is allocated (if not already allocated) and initialized, and vector of lower and upper bounds (*lowbound* and *upbound*) are copied to the structure together with *bignum*, which specify the large absolute value above which bounds are considered unspecified. Also problem related data that are necessary for correct performance of procedures (number of parameters *numparam*, number of objective functions *numobj* and number of constraint functions *numconstr* of the original problem) are set.

The following code segment specifies whether the modified analysis function will add penalty terms corresponding to bound constraints to the objective function:

```
bc->penfunc=penfunc;
setboundconstrpenpar(bc,0,0,(void *) &kpen,disppendata);
if (penfunc!=NULL)
  bc->transfparam=1;
```

In the first line we set the penalty generating function, which defines the form of the added penalty terms. Penalty generating function can be passed as argument to the optimization algorithm, but more often a particular form (such as `real_bas_f_zero_lin()`) will be prescribed by the algorithm (and possibly only some coefficients will be passed through arguments). If this function is NULL then penalty terms will not be added.

The second line sets the definition data for the penalty generating function. In this case, the same data is set for all bound constraints (because the second and the third argument are 0). In general, data can be specified for each component separately, and also separately for lower and upper bounds (the second argument in this case specifies the corresponding component; if the third argument is non-zero then data is set for lower bound). In the above case, the penalty generating function used is `real_bas_f_zero_lin` (assigned in definition of the variable *penfunc*), which requires a single non-negative coefficient in form of a pointer to a number of type `double` as definition data. The address of the variable *kpen* is therefore passed as definition data for penalty generating function.

At the end, we specify that transformation of parameters must also be performed by the modified analysis function, such that bound constraints will always be satisfied at parameters at which the original analysis function *anfuncorig* will be called. This is used e.g. in the nonlinear constraint simplex algorithm with bound constraint handling, but should not be in general used in gradient based algorithms[1].

Finally, instructions for conversion of bounds on parameter values to usual constraints are specified:

```
bc->constrfunc=constrfunc;
setboundconstrconstrpar(bc,0,0,(void *) &kconstr,dispconstrdata);
```

This is similar to instructions for addition of penalty terms. If constraint generating function *constrfunc* is NULL then bounds will not be converted to normal constraints. Otherwise, a separate constraint will be added in the modified analysis function for each bound, represented by a suitable constraint function. Similar to penalty terms, constraint function is evaluated by application of `bc->constrfunc` to the difference between parameter component and the corresponding bound (with sign defined such that the difference is positive when bound is violated). In the above example, linear function ($f(x)=x$) is used as constraint generating function. Its implementation `real_bas_f_zero_lin` is assigned in definition of the variable *constrfunc*. In the second line, definition data for this function (which must again be a pointer to a single coefficient of type `double`) is set on *bc* (again, the same data is used for all bounds).

Preparation of modified analysis function:

In the example, original analysis function with original definition data is used if bounds are not defined. This is done in the second part of the if branch:

```
anfunc=anfuncorig;
ancd=ancdorig;
```

If bounds are specified, then the modified analysis function will be used, which performs the requested additional operations according to the bound constraints (defined by the lower bound vector *lowbound*, upper bound vector *upbound* and parameter *bignum*):

```
prepandata_boundconstr(anfuncorig,ancdorig,NULL,bc,&anptbc);
anfunc=anfunc_mod_boundconstr;
ancd=anptbcpen;
```

---

[1] Because it gives raise to non-differentiable objective function of the modified problem (still continuous if the penalty generating function is continuous). Transformation of parameters in simplex algorithm ensures that the solution of modified problem exactly corresponds to the solution of the original problem with added bound constraints, even if the penalty function is not exact.

In the first line, data structure *anptbc* that is used as definition data for the modified analysis is prepared in the form required by the function `anfunc_mod_boundconstr()`. Function arguments are the original analysis function (representing the problem that is solved, except for bound constraints), its definition data, the function for de-allocation of definition data (which is NULL, since original definition data is prepared in the calling environment and should also be de-allocated there if dynamically allocated), the bound constraint data structure *bc*.

In the second and third line, the modified analysis function and its definition data are set. Analysis function is set to `anfunc_mod_boundconstr()`, which is pre-defined in IOptLib and automatically performs all operations for handling bound constraints, according to instructions on the bound constraint data structure *bc*. The structure *anptbc* of type `analaysispoint`, which has been prepared in the first line, will be used as definition data for this modified analysis function. Its structure is depicted in the right-hand side of Figure 6 (which schematically shows function of `anfunc_mod_boundconstr()`).

It is appropriate to mention at this point that it is advisable to launch an error or warning message if lower or upper bounds are defined, but neither the penalty generating nor the constraint generating function is specified (because in this case the bounds will be ignored if only handled implicitly by the modified analysis function; bounds can still be explicitly handled by the algorithm itself).

Cleaning part:

At the end, de-allocation of dynamically allocated data must be performed:

```
boundconstrdata (&bc);
dispanalysispoint (&anptbc);
```

In the above example, the bound constraint data *bc* was allocated by `prepboundconstrdata()` and the definition data for modified analysis *anptbc* has been allocated by `prepandata_boundconstr()`. Although *bc* is also put on *anptbc*, there is an agreement that bound constraint data is never de-allocated together with the definition data for the modified analysis function `anfunc_mod_boundconstr()`.

Note that in the initialization part, *bc* and *anptbc* must be set to NULL. On the contrary, they would contain an undefined address while not allocated, which would result in very unpleasant errors in functions for their initialization.

Let be emphasized again that `anfunc_mod_boundconstr()` uses its definition data (that must be of type `analysispoint`) not only for definition of its behavior, but also for storing analysis results. The modified response (together with optimization parameters at which the function is called) is stored directly in definition data *anptbc* before returned through output arguments. In addition, the results of the original analysis function are stored on *anptbc->data*, which must also be a pointer of type `analysispoint`, together with optimization parameters at which the original analysis function is called. These parameters may be transformed in order to satisfy bound constraints (transformation is performed if *bc->transfparam* is non-zero). Only *anptbc->data->data* alone actually acts as definition data for the modified analysis function. It is set to *bc* by `prepandata_boundconstr()`. Note that it is agreement that the bound constraint data (in this case *bc*) is never de-allocated together with the definition data for modified analysis (in this case *anptbc*, of type `analysispoint`), therefore it must be de-allocated separately.

4.2.2.2 Basic tools for handling bound constraints

4.2.2.3 Addition of penalty terms

4.2.2.4 Conversion of bound constraints to ordinary constraints

4.2.2.5 Modified analysis function

5 BUILDING BLOCKS FOR SUCCESSIVE APPROXIMATIONS

5.1 Introduction

Optimization algorithms employing successive approximation of response functions are one of the main targets of IOptLib. This chapter first describes basic instrumentation for building and using response approximations. The second part is devoted to description of implemented algorithms that make use of response approximation.

We need to mention that many different approximation methods can be used in optimization algorithms. IOptLib is intended to provide ready to use tools for a number of important classes of approximations as well as to allow extension with new classes. Therefore, there will be a fair level of abstraction in the top-most level functions and data types. This introduction is intended to provide the description of the approximation systems starting from the top-most levels.

There are several common or lower level utilities that are also used as part of the instrumentation for successive approximations, but are described in another sections because their more general and basic nature. In such cases, references to the appropriate sections are made.

Overview of intended functionality:

In order to give a feeling of a large diversity of tools that should be supported, let us give a brief overview of what we would like to support in the near future. *Low order polynomial approximations* such as linear or quadratic are considered basic approximation types. These approximations have *local character*. They are determined by set of *constant coefficients* of a fixed set of *basis functions*, which is used over the whole design space (i.e. coefficients are evaluated once for all). The effective range of these approximations is limited by the domain in which the approximated function can be adequately (i.e. with small enough error) approximated by the corresponding polynomial. Coefficients are most often calculated by the *least squares method*, which minimizes the weighted sum of squares of discrepancies between the approximation and original function over the sampling points. Beside a linear combination of a set of basis functions,

the coefficients can also occur in non-linear form, but still be calculated by the same least squares minimization procedure (non-linear least squares approximations).

Different kinds of approximations are designed to overcome the local character of approximations with constant coefficients, e.g. *kriging* approximation or *moving least squares* approximation. The moving least squares approximation is derived from corresponding constant coefficient least squares approximation, where coefficients are not constants in the design space and must be calculated in each point by the usual least squares procedure. This is achieved by non-constant weights, which are usually functions of the distance between the point of evaluation and the sampling point to which the weight corresponds. Moving least squares give rise to a whole set of possible definitions of weighting functions.

Apart from using different types of approximations, we must be able to adapt the approximation procedures to different kinds of analysis response. Most fundamentally, analysis can be or can not be able to provide *gradients* of the response, and one must provide efficient means of constructing approximations according to the current situation regarding this. One can always calculate gradients numerically, but as concerns approximation, it is usually far more efficient (and numerically stable) to just sample non-derivative response in more points and use all the sampled information in building approximations than to first perform additional evaluations to calculate derivatives, and then used derivative information in building approximations.

Another thing, which is more related to the nature of particular optimization problems, is that response functions can have very different properties e.g. with respect to effective range of low order approximations, which may be crucial for determination of sampling region and step restriction in restricted step methods. Sometimes we may assume that the objective function will be more problematic from this respect, and therefore automatic adjustment of algorithm parameters can be based on check performed only on the objective function and not on constraint functions. Sometimes this would not be true, which would reflect on how algorithms should be constructed. The underlying approximation utilities should be designed in such a way that all different situations can be handled.

Remarks on terminology:

A wide variety of terms related to this field are used in literature, not always in uniform manner. A widely used expression "*multi-point approximations*" is sometimes designated to stress that response approximations are generated on basic of evaluation (sampling) of the response in many points in the parameter space, in contrary with e.g. Taylor expansion, which is obtained by evaluating the response and its derivatives in a single point. The term "*response surface methods*" is sometimes used generally for optimization (or other analysis) methods that make use of approximations based on sampling response in a set of points. We find this term less appropriate because the response surface could adequately refer to the graph of the actual response function rather than its local approximation. Therefore we prefer the term "*response approximation methods*".

5.1.1 Overview of generic utilities from top to bottom

5.1.1.1 Hierarchical (top to bottom) arrangement that enables horizontal interactions

On top of the diversity outlined above (but not separated from this), a number of purely implementation issues arise. For example, one question is whether approximation of analysis response should be fundamentally implemented on top of standard vector functions (Section 2.3.3) such as numerical differentiation (Section 2.5.1.2, 2.5) or on top of standard analysis functions (Section 2.3.1).

It is decided that approximation utilities will be unified on a lower level, i.e. on the level of a single scalar response function. Generic utilities for building approximations of scalar functions of vector variable must therefore be provided and can be used very generally for different purposes. However, in the case where several scalar components of response are interconnected in some way (which is the case with different optimization response functions), it is inevitable that implementation must take into account these connections for the sake of efficiency.

One of the common points is that optimization and constraint functions will typically be sampled in same points in the parameter space. When e.g. the same weights are used in the usual or moving least squares, approximation coefficients will be calculated by solving linear systems of equations where system matrices will be the same for all response functions. For the sake of efficiency, the system matrix should be evaluated (assembled) and stored only once, and for the solution of the corresponding systems of equations, the decomposition stage can be performed only once for all the response functions (which will in this case produce different right-hand sides). Another example when sharing of resources for approximation of different response functions is sensible is weighting functions (and possibly their definition data) used for calculating weight assigned to sampling points. Although approximations of individual scalar responses are treated separately, it must be possible that these approximations share common resources, which are managed by higher levels (e.g. generic approximation function for analysis or vector response). This is exactly the way how things are implemented, as will be made evident in the explanations of individual subjects. Separate treatment, on the other hand, enables development of approximation utilities independently of higher level algorithmic architecture.

5.1.2 Basic scheme for use of approximations of analysis response

5.1.3 Basic approximation data types

5.2 Implementation notes

5.2.1 Specific and common auxiliary data structures

Different input data, intermediate results and final approximation data are stored on auxiliary approximation data structures (type `auxapproxdata`). Data stored on this structure have uniform structure, regardless of the type of (single or collective) approximated functions. Input data Whose structure depends on the type of the approximated functions (e.g. vector function, analysis response function, etc.) are on the basic approximation structure of type `funcapproxdata`.

Since some of the input or intermediate data defining approximations can be common for all the collectively approximated functions, there is a common auxiliary data structure for carrying approximation data common to all functions, and there are specific structures for each individual approximated function. Most of the data can reside either on the specific or on the common data structure. Individual types of data (which sometimes refers to groups of data) are treated individually, which enables complete flexibility of defining which data is common for all collectively approximated functions and which is not. When for some function a given kind of data is defined both on common and individual data structure, the individual structure priority is taken into account and the individual data is used. Intermediate results are stored on the individual structure if at least any of the input data necessary to produce that output is provided on the individual structure, and it is stored on the common structure if all the input data come from the common structure. Such arrangement enables a lot of flexibility in saving memory when things can be treated commonly for all functions (e.g. weights or sampling points) and defining things individually when desirable.

5.2.2 Approximation data updating functions

5.2.2.1 Updating weights

Function `approxupdateweights` makes updates the weight information if not yet updated. It takes pointers to the common and specific auxiliary structures as arguments (therefore it does not need to determine itself which is the specific structure – this must be done in the calling environment). One of the common and auxiliary structures may be NULL (to allow, for example, approximating a single (scalar) function).

Input data fields on the `auxapproxdata` structure are `weightfunc` with its definition data `weightdata` (these define the weighting function, normally with possibility of analytical gradient calculation) and `multweightst`. The second structure defines eventual multiplicative

factors by which weights of individual samples are multiplied[1], which enables defining a-priori importance of the samples (e.g. for filtering reasons). If weighting function is not defined then this structure itself defines actual weights[2].

Basic output weighting data fields are `weightst` (weights corresponding to samples) and `gradweightst` (gradients of these weights with respect to co-ordinates of point of evaluation). Corresponding update flags `up_weight` and `up_gradweight` define whether these data are updated or not (i.e. they need to be re-calculated from input data before they are used).

Beside that, there are auxiliary structures `vecweight` and `matgradweight`, in which weights and their gradients are assembled in vector (matrix) form required by some functions for calculating approximation. If these forms are required then transcription is done every time weighting data is needed. This is done by the weighting data update function even if the weighting data is updated, in order to ensure that information is up-to-date in every situation (see below).

It is permissible that one kind of weight input data is defined on a common structure and another on the specific function. In this case, resulting data is defined on the specific function. Only if all input weighting data is defined on the common structure then the results will also be defined on the common structure.

Function for updating weighting data is declared as

```
int approxparamtomat(funcapproxdata fa, auxapproxdata common, auxapproxdata
          spec, auxapproxdata *addractive)
```

The first argument is the pointer to (collective) approximation data structure that can represent approximations of multiple functions. This argument is optional and is used predominantly for identification reasons in error reports. Pointers to the common and specific auxiliary approximation data structures follow, of which at least one must be non-NULL. The last parameter is address where the pointer to the active auxiliary data structure is stored (i.e. the structure where resulting weighting data is stored). This parameter is also optional and is usually used when the caller would like to know which is the auxiliary data structure where results are stored (i.e. whether this is specific or common structure).

5.2.2.1.1 Directly setting the weights:

There is also a possibility that <u>weights corresponding to samples</u> (and eventually their gradients with respect to evaluation point co-ordinates) are <u>provided directly (externally set)</u>, i.e. the resulting fields `weightst` and eventually `gradweightst` are specified on the appropriate auxiliary approximation structure. In this case, the update flags (fields `up_weight` and `up_gradweight`) must be set, because otherwise updating utilities would try to overwrite the externally set resulting fields. Just because of the possibility of directly providing resulting weighting data, when weights are also required in vector form[3] (and their gradients in matrix form),

[1] If not specified, factor 1 is assumed for all samples; if partially specified, factor 0 is assumed for those samples for which factors are not specified.
[2] Which are considered constant, with gradients 0, i.e. they are not applicable e.g. for the moving least squares method.
[3] I.e. not only as pointers to type double on a stack.

the function for updating weighting data will do transcription to vector (matrix) form even if the update flags are set.

5.3 Approximation utilities – To implement

5.3.1 Things not yet implemented

Linear least squares with general basis functions (i.e. we either specify functions and data for evaluation of basis functions, or we specify values of basis functions in the sampling points (i.e. they are evaluated externally).

5.3.2 Efficiency issues

For linear least squares approximations (linear & quadratic polynomials, also moving least squares), implement functions that **do not solve for coefficients**, but **only assemble the system matrix and right-hand side vector**!! Use of these functions should replace straight functions which assemble and solve the equations at the same time.

This will improve the efficiency in the cases when several equations have the same system matrix but different right-hand sides, because decomposition can be made only once for all right hand sides leading to different coefficients. For example, usually there will be the same system matrix for all response functions (if more than one, i.e. if there are objective and constraint functions).

5.4 Lower level utilities for approximations

5.4.1 Basis functions for WLS and MLS approximations

Linear weighted least squares and the moving least squares approximations are based on a set of basis functions[5]. The first is simply a linear combination of basis functions with constant coefficients,

$$y(\mathbf{x};\mathbf{a}) = a_1 f_1(\mathbf{x}) + a_2 f_2(\mathbf{x}) + \ldots + a_n f_n(\mathbf{x}) = \sum_{j=1}^{n} a_j f_j(\mathbf{x}), \tag{76}$$

and the latter is a combination of basis functions with non-constant coefficients:

$$y(\mathbf{x}) = \sum_{j=1}^{n} a_j(\mathbf{x}) f_j(\mathbf{x}). \tag{77}$$

Approximation utilities of IOptLib utilize an uniform function form for calculation of a particular basis function in a particular parameter point $\mathbf{x}$, and eventually its derivatives. The function prototype is as follows:

```
int basis_f_general (int which,vector param,int *addrcalcval,double
            **addrval,
               int *addrcalcgrad,vector *addrgrad, void *clientdata);
```

Arguments of the function have the following meaning:

- `which` (input arg.) specifies which of the basis functions is to be evaluated
- `param` (input arg.) is the vector of independent variables at which this particular function is evaluated
- `addrcalcval` (input/output arg.) is a pointer to an integer that specifies whether the function value should be calculated and returned (non-zero) or not (zero or a NULL pointer). If non-zero and the function value could not be properly evaluated then the integer pointed to by this argument is set to -1.
- `addrval` (output arg.) is the address of a pointer to double (i.e. a variable of type `scalar`) where function value is stored if calculated. If calculation of function value is requested then this address must be non-NULL, but the pointer at the address can be NULL (in this case the pointer is allocated).

- `addrcalcgrad` (input/output arg.) is a pointer to an integer that specifies whether the gradient of the particular basis function should be calculated and returned (non-zero) or not (zero or a NULL pointer). If it points to a non-zero and the function gradient could not be properly evaluated then the integer pointed to by this argument is set to -1 by the function.
- `addrgrad` (output arg.) is the address of a vector where function gradient is stored if calculated. If calculation of function gradient is requested then it must be non-NULL. In this case, the vector at the address can be NULL or of inconsistent dimensions, in which case it will be allocated or re-allocated by the function.
- `clientdata` is a pointer to eventual additional data that precisely defines the basis functions. For example, for polynomial basis functions this would define products of which variables and in which powers constitute particular basis functions, which enables e.g. the same function to cover linear and quadratic basis with different orders of basis functions. This argument can be NULL for the functions that do not require additional definition data (e.g. a particular function for quadratic basis with agreed order of basis functions, where all basis functions are precisely known for space of arbitrary dimension – which is known form the parameter vector). Otherwise, the type of the data must be consistent with what the function expects.
- Function returns 0 if everything is OK or a negative error code if an error occurs.

5.4.1.1 Arbitrary polynomial basis functions

The function basis_f_pol is provided for arbitrary polynomial functions. Its type is equivalent to the type described in the beginning of Section 5.4.1 and is declared as

```
int basis_f_pol (int which,vector param,int *addrcalcval,double **addrval,
                 int *addrcalcgrad,vector *addrgrad, void *clientdata);
```

The ***definition data*** for this function must be a stack (type `stack`) that contains an index table (type `indtab`) for each basis function. The number of elements of the stack must therefore be equivalent to the number of basis functions. Basis functions are monomials, more specifically products of arbitrary numbers (possibly 0) of variables. Examples of basis functions with corresponding indices are:

1 – {} (in this case the index table can be NULL, or it can be allocated but containing no indices)

- $x_3 \cdot x_4$ – {3,4}
- $x_5^2 = x_5 \cdot x_5$ – {5,5}
- x_2 – {2}

User of the function must compose the definition data before calling the function, for which purpose pre-defined utilities for particular standard basis can be used. In particular, IOptLib provides utilities for linear and quadratic bases where basis functions are sorted in a particular order (Section 5.4.1.2).

5.4.1.2 Linear and quadratic basis functions in standard order

IOptLib specifically provides some utilities for two particular bases – linear and quadratic – where basis functions are sorted in an agreed order. The library provides e.g. utilities for setting up the appropriate definition data for these sets of basis functions for the function `basis_f_pol` described in Section 5.4.1.1, and utilities for re-arrangements of coefficients into constant, linear and (eventually) quadratic term. For other operations such as the calculation of a single basis function and eventually its derivative, or calculation of a linear combination of basis functions (which is in the case of ordinary weighted least squares equivalent to calculation function approximation in a specific task), more general functions from Sections 5.4.1 and 5.4.1.1 are used.

Standard linear basis in ^{n} consists of the following basis functions (in the same order):

$$1, x_1, x_2, ..., x_n . \quad (78)$$

Standard quadratic basis in ^{n} consists of the following basis functions (in the same order):

$$\begin{array}{l} 1, \\ x_1, x_2, ..., x_n, \\ x_1^2, x_2^2, x_3^2, ..., x_n^2, \\ x_1 \cdot x_2, x_1 \cdot x_3, ..., x_1 \cdot x_n \\ x_2 \cdot x_3, x_2 \cdot x_4, ..., x_2 \cdot x_n, \\ ..., \\ x_{n-1} \cdot x_n \end{array} . \quad (79)$$

This order is taken because lower order terms are sometimes given more importance (sometimes higher order terms are switched on only in the final stages of computation for better precision), and in the case of quadratic basis privileged treatment of pure quadratic terms is easier in this way.

For composition of the definition data for basis functions of linear and quadratic bases, respectively, the following two functions are used:

```
setup_linear_basis_data (int dim, stack *addrst);
setup_quad_basis_data (int dim, stack *addrst);
```

For both functions, argument `dim` specifies the dimension of space, and argument `addrst` is the address of the stack on which basis function specifications are put as index tables (type `indtab`). Functions automatically perfume all the necessary re-allocation. The definition data are de-alllocated by the function `disp_pol_basis_data`. The function `dispstackallspec` can also be used:

Example 5: Setup, use and de-allocation of definition data for standard linear and quadratic bases.

```
stack deflin=NULL,defbas=NULL;  /* Do not forget to initialize the stacks to
            NULL */
int dimension=5;

...
setup_linear_basis_data (dimension, &deflin);
setup_quad_basis_data (int dim, &defquad);

/* Use of the definition data ... */
...

/* Deallocation of the definition data: */
disp_pol_basis_data(&deflin);  /* by use of specific functio nfor this
            purpose - safer way */
dispstackallspec(&defquad, (void (*) (void **)) dispindtab);  /* By use of
            functins for de-allocation of stacks and index tables */
```

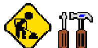
5.4.1.3 Planning types and utilities 

Table 4 shows data used for evaluation of approximations of a set of functions $g_i(\mathbf{x})$, for ordinary weighted least squares and moving least squares. If something is considered a function then this is shown in the left-most column with independent variable in round brackets. Dependencies relevant for evaluation procedure are shown in round brackets in the rest two columns. Also, indices show relations (index i denotes different functions and index k different sampling points, and index l denotes different basis function). For example, if a given quantity has index i this means that it is calculated differently for each function $g_i(\mathbf{x})$, but it does not mean that $g_i(\mathbf{x})$ or g_i occurs explicitly in evaluation formula.

Considerations:

One of the problems is that the basic ***data structures (approximation objects) must be appropriate for different kinds of approximations*** (e.g. weighted least squares & moving least squares). The problem can be solved in such a way that each approximation has its own set of utilities (e.g. "update the structure when sampling points are changed") and corresponding data (some of the data may be shared, but only if there are the same rules for updating). Then, there are generic utilities, which perform the utilities for all types of approximations that will eventually use the same data structure. But **another possibility** is

Some data are shared across different utilities and some data are shared across different functions to be approximated.

- **How to know which data are updated**??? (how to keep information on whether data has been updated?) This can not be done through NULL/non-NULL. (By flags for each data, taking into account specially local and global aux. approx. data?)

Would it be possible to treat e.g. weighting functions common and sampling specific, or vice versa? – Maybe yes, but only for all functions specific or for all functions common - no use of making things different for individual functions. Maybe this can be allowed only for weighting functions??? – what about accounting for taking into account a different number of samples for different functions (e.g. objective/constraint)?

Suggestions:

For each kind of approximation (even with minimal differences), have completely separate data (or client data).

Unify data structure carrying approximations of all different functions with common sampling, then distinguish (e.g. between vector and analysis functions) by functions that handle mapping of vector data (sampled) to individual data. **Argument:** this will mean only one version of different kinds of *updating* functions for each kind of approximation (otherwise combinatorial).

Global structure will contain a stack of local auxiliary approximation data and auxiliary global approximation data.

Auxiliary approximation data structures (local and global) should carry pointers to a global structure (is this necessary??) Unified data structure for a group of functions should

All calculations AND updates will be performed through the unified structure for several approximations! For a single local approximation, there will simply be no global approximation!!!

Provide the "Updated information" by flags!

On the unified global structure, **update function pointers** should be only for updating input data (to reduce their number), e.g. *updatesampling*, *updateweights*, etc. Mapping of global data to local should be done by different functions (because this is bound to the type of the single- or multi-valued function such as scalar, vector, or analysis), and these functions will in general use the updating function after mapping is done. Updating functions will in general only set updated flags to 0, and then calculation functions will do all necessary checking and re-evaluation for the data, because these functions are different for different kind of functions.

Function pointers for evaluation of a single approximated function should also reside on the unified global structure.

Common and specific data:

Auxiliary data (the data that is only used in calculation but does not represent intermediate results) should be stored on the common data structure.

For results and intermediate data (i.e. data that is calculated or derived from other data) and for input (independent) data, the decision on whether the data on the common structure or data on the structure corresponding to individual function is used, is made in the following way. If all the data from which given data is derived is common then the data is also common, otherwise the data is specific. For input data that is not dependent on other data, the rule is that if the data is allocated on the structure corresponding to a specific function then the data is considered specific, otherwise the data is considered common.

Warning: treatment of common data as specific will lead in worse efficiency, because operations that could be performed only once for all the functions will be repeated for each individual function.

Remarks to suggestions:

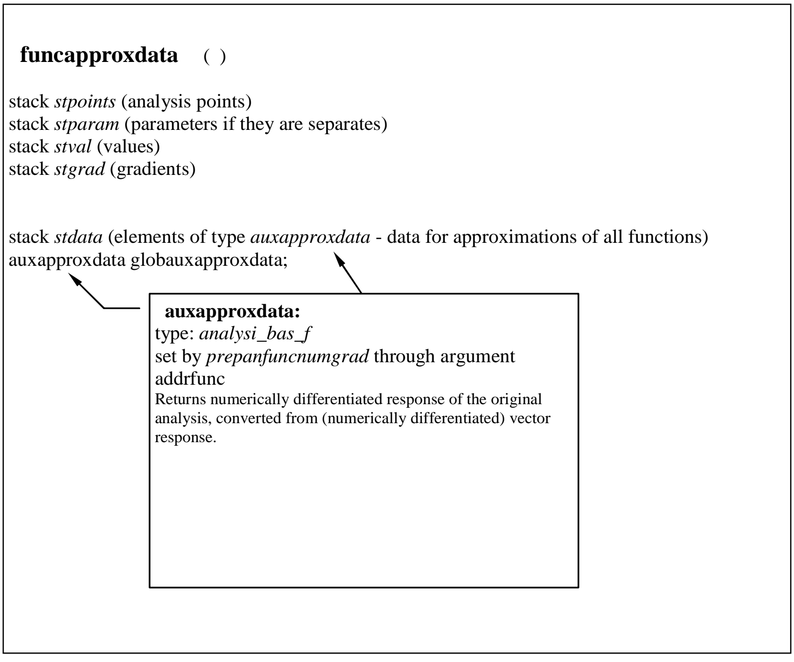


Figure 7: Approximation data structure.

5.4.1.3.1 Different types of data:

Common data shared by all approximations (input data):

$\mathbf{x}$ (point of evaluation)

$\mathbf{r}_k$ (sampling points)

g_{ik} (function i)

Intermediate data (auxiliary data)

Final data (results)

$y_{\mathbf{a}i}(\mathbf{x})$

$\mathbf{a}_i$ / $\mathbf{a}_i(\mathbf{x})$ – Coefficients of approximation, different meaning for different kinds of approximations

Table 4: Data for ordinary weighted least squares and moving least squares approximation with dependencies. Index *i* denotes different functions and index *k* different sampling points, and index *l* denotes different basis functions (only for linear approximation).

Data	**Least squares / fields**	**Moving least squares**
$\mathbf{x}$ (point of evaluation)	The same function coefficients. Approximation value changes.	Re-evaluate function coefficients, weight change. Left-hand side the same for all functions if weighting functions are the same.
$\mathbf{r}_k$ (sampling points)	Different coefficients. Left-hand side the same for all functions.	Different coefficients. Left-hand side the same for all functions.
$y_l(\mathbf{x})$ (basis functions)[1]	y_l or y_{il}	y_l or y_{il}
Basis bunctions may be different for different functions g_i. For some tools, basis functions are nowhere explicitly stated because they are just assumed (e.g. linear, quadratic, etc.).		
y_{lk} (basis functions)	$y_{lk}(\mathbf{r}_k)$ or $y_{ilk}(\mathbf{r}_k)$ (may be different for different functions, this is not common)	$y_{lk}(\mathbf{r}_k)$ or
$w_k(\mathbf{x})$ (weighting functions of sampling points)	/	$w_k(\mathbf{x})$ $(\mathbf{r}_k)$ or $w_{ik}(\mathbf{x})$
Weighting functions may be different for functions g_i, but this is not common.		
w_k =(weights)	$w_k(\mathbf{r}_k, \mathbf{A}, \mathbf{s})$ or $w_{ik}(\mathbf{r}_k, \mathbf{A}, \mathbf{s})$	$w_k(\mathbf{r}_k,\mathbf{x}\ \mathbf{A})$ or $w_{ik}(\mathbf{r}_k,\mathbf{x}\ \mathbf{A})$
Weights may be different for different functions g_i, but this is not common.		
$g_i(\mathbf{x})$ (function i)[2]		
g_{ik} (function values in sampling points)	g_{ik} $(\mathbf{r}_k)$	g_{ik} $(\mathbf{r}_k)$
$\mathbf{a}_i$ / $\mathbf{a}_i(\mathbf{x})$ (approximation coefficients for $g_i(\mathbf{x})$)	$\mathbf{a}_i(\mathbf{C}_i,\mathbf{d}_i)=\mathbf{a}_i(w_k,y_l, g_{ik})=$	
$\mathbf{C}$ / $\mathbf{C}(\mathbf{x})$ (system matrix for calculating coefficients $\mathbf{a}_i$)	$\mathbf{C}(w_k,y_{lk})=\mathbf{C}(\mathbf{r}_k)$ or $\mathbf{C}_i$ $(w_{ik},y_{ilk})=\mathbf{C}_i$ $(\mathbf{r}_k)$	$\mathbf{C}(w_k(\mathbf{x}),y_{lk})=\mathbf{C}(\mathbf{x},\mathbf{r}_k)$ or $\mathbf{C}_i$ $(w_{ik}(\mathbf{x}),y_{ilk})=\mathbf{C}_i(\mathbf{x},\mathbf{r}_k)$
$\mathbf{d}_i$ / $\mathbf{d}_i(\mathbf{x})$ (right-hand side of system of equations for $\mathbf{a}_i$)	$\mathbf{d}_i$ $(w_k, g_{ik}, y_{lk})=\mathbf{d}_i$ $(\mathbf{r}_k)$	$\mathbf{d}_i$ $(w_k(\mathbf{x}), g_{ik}, y_{lk})=\mathbf{d}_i$ $(\mathbf{x},\mathbf{r}_k)$
$\mathbf{LU}_\mathbf{C}$ / $\mathbf{LU}_{\mathbf{C}i}$ (decomposition of $\mathbf{C}$)	$\mathbf{LU}_\mathbf{C}$ $(\mathbf{C})=\mathbf{LU}_\mathbf{C}(w_{ik},y_{ilk})=\mathbf{LU}_\mathbf{C}(\mathbf{r}_k)$ or $\mathbf{LU}_{\mathbf{C}i}(w_{ik},y_{ilk})=\mathbf{LU}_{\mathbf{C}i}(\mathbf{r}_k)$	
$y_{\mathbf{a}i}(\mathbf{x})$ (approximation of function i - the final result)	$y_{\mathbf{a}i}(\mathbf{x})$ $(\mathbf{C}_i,\mathbf{d}_i)=$	$y_{\mathbf{a}i}(\mathbf{x})$ $(\mathbf{C}_i,\mathbf{d}_i)=$
$\mathbf{A}$, $\mathbf{s}$ (approximation region, defining the weighting functions)		

[1] Usually denoted by $f_l(\mathbf{x})$.
[2] Usually denoted by $f(\mathbf{x})$ without an index, here we use different notations because there are several functions.

5.4.2 Weighting functions

Weighting functions are usually scalar functions that have the maximal value in the center of approximation and fall with the distance to the center. In the most general form, we have

$$w(\mathbf{x}) = f\left(\left\| \mathbf{A}\left(\mathbf{x}-\mathbf{s}\right) \right\| \right) \tag{80}$$

We have

$$\nabla w(\mathbf{x}) = f'\left(\left\| \mathbf{A}\left(\mathbf{x}-\mathbf{s}\right) \right\| \right) \frac{\mathbf{A}^T \mathbf{A}\left(\mathbf{x}-\mathbf{s}\right)}{\left\| \mathbf{A}\left(\mathbf{x}-\mathbf{s}\right) \right\|} . \tag{81}$$

We have taken into account

$$\begin{aligned} &\nabla f\left(g(\mathbf{x})\right) = f'\left(g(\mathbf{x})\right) \nabla g(\mathbf{x}), \ \nabla \left\| \mathbf{x} \right\| = \frac{\mathbf{x}}{\left\| \mathbf{x} \right\|}, \\ &\nabla f\left(\mathbf{A}(\mathbf{x}-\mathbf{s})\right) = \mathbf{A}^T \nabla f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{A}(\mathbf{x}-\mathbf{s})} \end{aligned} . \tag{82}$$

If **A** is a symmetric real matrix then we can write **A**=**USU**^{T}, where **D** is a diagonal matrix whose elements are eigenvalues of **A**, and **U** is orthogonal matrix whose columns are corresponding normed eigenvectors.

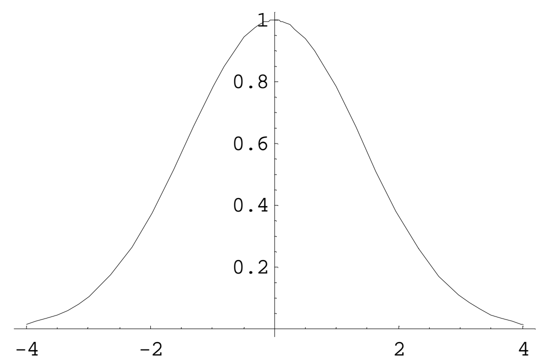


Figure 8: A possible choice for a one-dimensional weighting function $w(t)=w(||\mathbf{A}(\mathbf{x}-\mathbf{s})||)$.

6 OPTIMIZATION ALGORITHMS

7 TESTING SYSTEM

The IOptLib has an extensive system for testing of algorithms and analysis functions[1]. Core of this system is implemented in the module `opttest.c`. In principle, the testing system can be thought as consisting of test examples and the **test driver**. The driver implements general utilities that are independent of specific response, such as addition of noise, counting and recording analyses, automatic numerical differentiation, calculation of optima by robust built-in methods, efficiency statistics, etc.

7.1 Registering an optimization problem or test case

In order to use the functionality of the test driver on a particular optimization problem, the problem must first be **registered in the testing system**. In the simplest way, the problem can be registered by calling the `regoptprob` function with the following declaration:

```
int regoptprob(char *name, analysis_bas_f anfunc, void *andata, void
            (*dispandata)(void **));
```

The registration function `regoptprob` returns an unique identification number, which is assigned to the problem when it is registered. From this point on, the problem will usually be

[1] For analysis functions, one may for example test the consistency of the provided gradients of the response. Some other functionality is also planned such as automatic testing of response smoothness.

referred to by this number[1]. Arguments of the function are a descriptive name[2], original analysis function, its definition data and the function for de-allocation of this definition data (if this function is NULL then the definition data will not be de-allocated when the problem is unregistered).

The testing system is not used exclusively for testing. Some other modules may use it for other purposes because of the easy use of its generally applicable functionality[3].

You can take a look at the function `testopttest` at the end of `opttest.c` in order to get some basic ideas about how the testing system is used. This function was implemented for testing functionality of the module as one goes along. It is expected that the testing system will be extended drastically in the future, therefore checking the source code of the module and especially the contents of the function `testopttest` may be performed in order to get more accurate and updated information than can be found in the manuals. Hopefully, what is currently found about the testing system in the manuals should remain valid in the future, it is only incomplete.

8 APPENDIX: FORMULAE FOR WLS AND MLS

This chapter lists some basic formulas for the weighting least squares and the moving least squares approximations. A separate report on these methods is in preparation, and this aims at serving only as a quick reference for some portions of this manual, in particular the Section 5: Building Blocks for Successive Approximations .

8.1 WLS approximation

Calculation of coefficients:

[1] There are also a number of predefined analysis function, which require an integer pointer as the definition data, and the pointer to the identification number must be passed in its place. These analysis functions locate the original analysis function and definition data through that pointer, and use them for calculation of the response (an example is `analyseoptprob`). After registration, the original analysis function can always be replaced by `analyseoptprob` with pointer to the assigned problem ID as definition data.

[2] This is optional (it can be NULL) and is used by function that print information about the problem.

[3] This is partially due to the fact that problems can be simply referred to through an integer ID after they are installed.

$$\mathbf{C}\mathbf{a} = \mathbf{d}\,, \tag{83}$$

$$C_{ij} = \sum_{k=1}^{Nv} w_k{}^2 f_i(\mathbf{x}_k)\, f_j(\mathbf{x}_k) \tag{84}$$

$$d_i = \sum_{k=1}^{Nv} w_k{}^2 f_i(\mathbf{x}_k)\; y_k \tag{85}$$

Calculation of value and gradient of approximation:

$$\tilde{f}(\mathbf{x};\mathbf{a}) = \sum_{j=1}^{Nb} a_j f_j(\mathbf{x})\,, \tag{86}$$

$$\frac{\partial\, \tilde{f}(\mathbf{x};\mathbf{a})}{\partial\, x_i} = \sum_{j=1}^{Nb} a_j\, \frac{\partial\, f_j(\mathbf{x})}{\partial\, x_i}\,, \tag{87}$$

8.1.1 WLS with value & gradient information:

$$C_{ij} = \sum_{k=1}^{Nv}\left(w_k{}^2 f_i(\mathbf{x}_k)\, f_j(\mathbf{x}_k)\right) + \sum_{k_g=1}^{N_g}\sum_{t=1}^{N}\left(w_{k_g t}{}^2\, \frac{\partial f_i}{\partial x_t}\left(\mathbf{x}_{k_g}\right) \frac{\partial f_j}{\partial x_t}\left(\mathbf{x}_{k_g}\right)\right) \tag{88}$$

$$d_i = \sum_{k=1}^{N_v}\left(w_k{}^2 f_i(\mathbf{x}_k)\; y_k\right) + \sum_{k_g=1}^{N_g}\sum_{t=1}^{N}\left(w_{k_g t}{}^2\, \frac{\partial f_i}{\partial x_t}\left(\mathbf{x}_{k_g}\right) g_{k\,t}\right) \tag{89}$$

w_{kl} is weight assigned to component *l* of the function gradient in the point *k*.

8.2 MLS approximation

$$\begin{aligned} &\mathbf{C}(\mathbf{x})\;\; \mathbf{a}(\mathbf{x}) = \mathbf{d}(\mathbf{x}) \\ &\mathbf{a}(\mathbf{x}) = \left[a_1(\mathbf{x}),\, a_2(\mathbf{x}),\; ...,\, a_n(\mathbf{x})\; \right]^T \end{aligned}\;, \tag{90}$$

$$C_{i\,j}(\mathbf{x}) = \sum_{k=1}^{Nb} w_k(\mathbf{x})^2\; f_i(\mathbf{x}_k)\, f_j(\mathbf{x}_k) \tag{91}$$

$$d_i(\mathbf{x}) = \sum_{k=1}^{Nb} w_k(\mathbf{x})^2 f_i(\mathbf{x}_k)\, y_k \,. \tag{92}$$

Calculation of values:

$$\tilde{f}(\mathbf{x}) = \sum_{i=1}^{Nb} a_i(\mathbf{x}) f_i(\mathbf{x}) \,. \tag{93}$$

8.2.1 Calculation of derivatives of the MLS approximation:

$$\frac{\partial y(\mathbf{x};\mathbf{a}(\mathbf{x}))}{\partial x_l} = \sum_{j=1}^{Nb}\left(a_j(\mathbf{x}) \frac{\partial f_j(\mathbf{x})}{\partial x_l} + \frac{\partial a_j(\mathbf{x})}{\partial x_l} f_j(\mathbf{x}) \right) . \tag{94}$$

Coefficients $\mathbf{a}(\mathbf{x})$ in (94) are obtained by solution of the system (90) by taking into account (91) and (92).

Calculation of $\partial \mathbf{a} / \partial x_l$:

$$\mathbf{C}(\mathbf{x}) \frac{\partial \mathbf{a}(\mathbf{x})}{\partial x_l} = \frac{\partial \mathbf{d}(\mathbf{x})}{\partial x_l} - \frac{\partial \mathbf{C}(\mathbf{x})}{\partial x_l} \mathbf{a}(\mathbf{x}) = \mathbf{q}^{(l)}(\mathbf{x}) \;. \tag{95}$$

$$\begin{aligned} \frac{\partial C_{ij}(\mathbf{x})}{\partial x_l} = \sum_{k=1}^{Nv} \Bigg(& 2 w_k(\mathbf{x}) \frac{\partial w_k(\mathbf{x})}{\partial x_l} f_i(\mathbf{x}_k) f_j(\mathbf{x}_k) + \\ & w_k(\mathbf{x})^2 \frac{\partial f_i(\mathbf{x}_k)}{\partial x_l} f_j(\mathbf{x}_k) + w_k(\mathbf{x})^2 f_i(\mathbf{x}_k) \frac{\partial f_j(\mathbf{x}_k)}{\partial x_l} \Bigg) \end{aligned} \tag{96}$$

$$\frac{\partial d_i(\mathbf{x})}{\partial x_l} = \sum_{k=1}^{Nv}\left(\left(2 w_k(\mathbf{x}) \frac{\partial w_k(\mathbf{x})}{\partial x_l} f_i(\mathbf{x}_k) + w_k(\mathbf{x})^2 \frac{\partial f_i(\mathbf{x}_k)}{\partial x_l} \right) y_k \right) \tag{97}$$

Sequence of calculation:

1. Assembly **C**, **d** according to (88) and (89).
2. Decompose **C**, with eventual regularization if necessary (note that previous values of **a** may be necessary for this).
3. Solve for **a**.
4. Assembly $d\mathbf{C}/dx_l$, $d\mathbf{d}/dx_l$ according to (96) and (97) All these (for each l) are assembled simultaneously, which reduse repeated calculation of gradients of w and f,

but requires a large additional storage. It is recommended tha the storage is on the parent structure (in this way it is shared by all functions).

5. Calculate $\mathbf{q}^{(l)}$ and solve for for $d\mathbf{a}/dx_l$. It is again recommended that $\mathbf{q}$ is common for all functions.

Remarks:

The most problematic is simultaneous storage of all $d\mathbf{C}/dx_l$ for each l. This could be avoided by calculation of $\mathbf{q}_l$ for each l separately, using the same storage for $d\mathbf{C}/dx_l$ and for $d\mathbf{d}/dx_l$. However, this would then require many (n) repeated calculations of w, f_i and their gradients. For now, we stick with simultaneous assembly of all derivatives of the system matrix and the right-hand side vector.

8.2.2 Second order derivatives

$$\frac{\partial^2 y\left(\mathbf{x};\mathbf{a}(\mathbf{x})\right)}{\partial x_l \partial x_m} = \sum_{j=1}^{Nb}\left(\frac{\partial^2 a_j(\mathbf{x})}{\partial x_l \partial x_m} f_j(\mathbf{x}) + \frac{\partial a_j(\mathbf{x})}{\partial x_l}\frac{\partial f_j(\mathbf{x})}{\partial x_m} + \frac{\partial a_j(\mathbf{x})}{\partial x_m}\frac{\partial f_j(\mathbf{x})}{\partial x_l} + a_j(\mathbf{x})\frac{\partial^2 f_j(\mathbf{x})}{\partial x_l \partial x_m}\right) . \tag{98}$$

In the above equation, coefficients $\mathbf{a}(\mathbf{x})$ are obtained by solution of the system (90) by taking into account (91) and (92). Derivatives of the coefficients, ${\partial \mathbf{a}}/{\partial x_l}$, are calculated by solution of the system of equations (95), taking into account (96) and (97).

Calculation of $\frac{\partial^2 \mathbf{a}}{\partial x_l \partial x_m}$:

Derivation of (95) yields:

$$\mathbf{C}(\mathbf{x})\frac{\partial^2 \mathbf{a}(\mathbf{x})}{\partial x_l \partial x_m} = \frac{\partial^2 \mathbf{d}(\mathbf{x})}{\partial x_l \partial x_m} - \frac{\partial^2 \mathbf{C}(\mathbf{x})}{\partial x_l \partial x_m}\mathbf{a}(\mathbf{x}) - \frac{\partial \mathbf{C}(\mathbf{x})}{\partial x_l}\frac{\partial \mathbf{a}(\mathbf{x})}{\partial x_m} - \frac{\partial \mathbf{C}(\mathbf{x})}{\partial x_m}\frac{\partial \mathbf{a}(\mathbf{x})}{\partial x_l} . \tag{99}$$

The second order derivatives of coefficients are again obtained by solution of a system equations with the same system matrix as in the equation calculation for coefficients, and with different right-hand side. The right-hand side is composed of terms from equations (91), (92), (96), (97), and the coefficients **a** and their derivatives, obtained by the solution of equations (90) and (95). The remaining terms are second order derivatives of the system matrix **C** and the right-hand side for calculation of derivatives, vector **d**.

By differentiation of (96), we have

$$\frac{\partial^2 C_{ij}(\mathbf{x})}{\partial x_l \partial x_m} =$$
$$\sum_{k=1}^{Nv}\Bigg(2\frac{\partial w_k(\mathbf{x})}{\partial x_m}\frac{\partial w_k(\mathbf{x})}{\partial x_l} f_i(\mathbf{x}_k) f_j(\mathbf{x}_k) + 2 w_k(\mathbf{x})\frac{\partial^2 w_k(\mathbf{x})}{\partial x_l \partial x_m} f_i(\mathbf{x}_k) f_j(\mathbf{x}_k) +$$
$$2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_l}\frac{\partial f_i(\mathbf{x}_k)}{\partial x_m} f_j(\mathbf{x}_k) + 2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_l} f_i(\mathbf{x}_k)\frac{\partial f_j(\mathbf{x}_k)}{\partial x_m} + \tag{100}$$
$$2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_m}\frac{\partial f_i(\mathbf{x}_k)}{\partial x_l} f_j(\mathbf{x}_k) + w_k(\mathbf{x})^2\frac{\partial^2 f_i(\mathbf{x}_k)}{\partial x_l \partial x_m} f_j(\mathbf{x}_k) + w_k(\mathbf{x})^2\frac{\partial f_i(\mathbf{x}_k)}{\partial x_l}\frac{\partial f_j(\mathbf{x}_k)}{\partial x_m} +$$
$$2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_m} f_i(\mathbf{x}_k)\frac{\partial f_j(\mathbf{x}_k)}{\partial x_l} + w_k(\mathbf{x})^2\frac{\partial f_i(\mathbf{x}_k)}{\partial x_m}\frac{\partial f_j(\mathbf{x}_k)}{\partial x_l} + w_k(\mathbf{x})^2 f_i(\mathbf{x}_k)\frac{\partial^2 f_j(\mathbf{x}_k)}{\partial x_l \partial x_m}\Bigg)$$

and by differentiation of (97), we have

$$\frac{\partial^2 d_i(\mathbf{x})}{\partial x_l \partial x_m} =$$
$$\sum_{k=1}^{Nv}\Bigg(\bigg(2\frac{\partial w_k(\mathbf{x})}{\partial x_m}\frac{\partial w_k(\mathbf{x})}{\partial x_l} f_i(\mathbf{x}_k) + 2 w_k(\mathbf{x})\frac{\partial^2 w_k(\mathbf{x})}{\partial x_l \partial x_m} f_i(\mathbf{x}_k) + 2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_l}\frac{\partial f_i(\mathbf{x}_k)}{\partial x_m} + \tag{101}$$
$$2 w_k(\mathbf{x})\frac{\partial w_k(\mathbf{x})}{\partial x_m}\frac{\partial f_i(\mathbf{x}_k)}{\partial x_l} + w_k(\mathbf{x})^2\frac{\partial^2 f_i(\mathbf{x}_k)}{\partial x_l \partial x_m}\bigg) y_k\Bigg)$$

8.2.3 MLS with values and gradients

$$C_{ij} = \sum_{k=1}^{Nv}\left(w_k(\mathbf{x})^2 f_i(\mathbf{x}_k) f_j(\mathbf{x}_k)\right) + \sum_{k_g=1}^{Ng}\sum_{t=1}^{N}\left(w_{k_g t}(\mathbf{x})^2\frac{\partial f_i}{\partial x_t}(\mathbf{x}_{k_g})\frac{\partial f_j}{\partial x_t}(\mathbf{x}_{k_g})\right) \tag{102}$$

$$d_i = \sum_{k=1}^{N_v}\left(w_k(\mathbf{x})^2 f_i(\mathbf{x}_k)\, y_k\right) + \sum_{k_g=1}^{Ng}\sum_{t=1}^{N}\left(w_{k_g t}(\mathbf{x})^2\frac{\partial f_i}{\partial x_t}(\mathbf{x}_{k_g}) g_{k_g t}\right) \tag{103}$$

First order derivatives of the approximation:

$$\frac{\partial \tilde{f}(\mathbf{x};\mathbf{a}(\mathbf{x}))}{\partial x_l} = \sum_{j=1}^{Nb}\left(a_j(\mathbf{x})\frac{\partial f_j(\mathbf{x})}{\partial x_l} + \frac{\partial a_j(\mathbf{x})}{\partial x_l} f_j(\mathbf{x})\right). \tag{104}$$

$$\mathbf{C}(\mathbf{x})\frac{\partial\,\mathbf{a}(\mathbf{x})}{\partial\,x_l}=\frac{\partial\,\mathbf{d}(\mathbf{x})}{\partial\,x_l}-\frac{\partial\,\mathbf{C}(\mathbf{x})}{\partial\,x_l}\mathbf{a}(\mathbf{x})\ . \tag{105}$$

$$\begin{aligned}\frac{\partial\,C_{ij}}{\partial\,x_m}=&\sum_{k=1}^{Nv}\left(2w_k(\mathbf{x})\frac{\partial\,w_k(\mathbf{x})}{\partial\,x_m}f_i(\mathbf{x}_k)f_j(\mathbf{x}_k)+\right.\\&\left.w_k(\mathbf{x})^2\frac{\partial\,f_i(\mathbf{x}_k)}{\partial\,x_m}f_j(\mathbf{x}_k)+w_k(\mathbf{x})^2 f_i(\mathbf{x}_k)\frac{\partial\,f_j(\mathbf{x}_k)}{\partial\,x_m}\right)+\\&\sum_{k_g=1}^{N_g}\sum_{t=1}^{N}\left(2w_{k_g t}(\mathbf{x})\frac{\partial\,w_k(\mathbf{x})}{\partial\,x_m}\frac{d\,f_i}{d\,x_t}\left(\mathbf{x}_{k_g}\right)\frac{d\,f_j}{d\,x_t}\left(\mathbf{x}_{k_g}\right)+\right.\\&\left.w_{k_g t}(\mathbf{x})^2\frac{\partial^2 f_i}{\partial\,x_t\,\partial\,x_m}\left(\mathbf{x}_{k_g}\right)\frac{\partial\,f_j}{\partial\,x_t}\left(\mathbf{x}_{k_g}\right)+w_{k_g t}(\mathbf{x})^2\frac{\partial\,f_i}{\partial\,x_t}\left(\mathbf{x}_{k_g}\right)\frac{\partial^2 f_j}{\partial\,x_t\,\partial\,x_m}\left(\mathbf{x}_{k_g}\right)\right)\end{aligned} \tag{106}$$

$$\begin{aligned}\frac{\partial\,d_i}{\partial\,x_m}=&\sum_{k=1}^{N_v}\left(\left(2w_k(\mathbf{x})\frac{\partial\,w_k(\mathbf{x})}{\partial\,x_m}f_i(\mathbf{x}_k)\,y_k+w_k(\mathbf{x})^2\frac{\partial\,f_i(\mathbf{x}_k)}{\partial\,x_m}\right)y_k\right)+\\&\sum_{k_g=1}^{N_g}\sum_{t=1}^{N}\left(\left(2w_{k_g t}(\mathbf{x})\frac{\partial\,w_{k_g t}(\mathbf{x})}{\partial\,x_m}\frac{\partial\,f_i}{\partial\,x_t}\left(\mathbf{x}_{k_g}\right)+w_{k_g t}(\mathbf{x})^2\frac{\partial^2 f_i}{\partial\,x_t\,\partial\,x_m}\left(\mathbf{x}_{k_g}\right)\right)g_{k\,t}\right)\end{aligned} \tag{107}$$

8.3 Implementation remarks

8.3.1 Use of linear (affine) transformations in approximation based optimization algorithms

Affine transformations are typically used for sampling, definition of restricted step constraint, and for definition of weighting functions. Usually, linear transformations for different purposes will have the same transformation matrix for various purposes, eventually different by a scalar factor. Therefore, the same decomposition and eventually inverse matrix can be used for all tasks involved in approximation based algorithms.

Beside the different scalar factor in transformation matrix, there may be differences in the shift vector in affine transformations (defining a center of a transformed region).

A general affine transformation (Figure 2) is defined by:

$$\mathbf{x}=\mathbf{F}(\tilde{\mathbf{x}})=\mathbf{A}\,\tilde{\mathbf{x}}+\mathbf{s}\,, \tag{108}$$

and its inverse transformation is (Figure 10)

$$\tilde{\mathbf{x}} = \mathbf{F}^{-1}(\mathbf{x}) = \mathbf{A}^{-1}(\mathbf{x} - \mathbf{s}) . \tag{109}$$

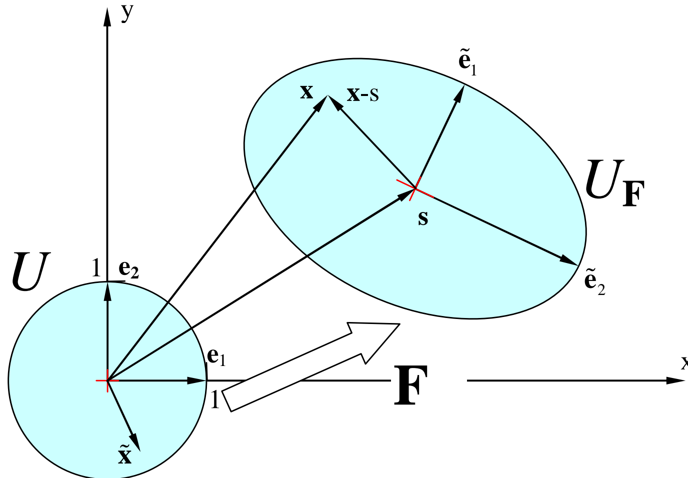


Figure 9: Affine function **F** that maps n unit ball into an ellipsoidal domain centered around **s**.

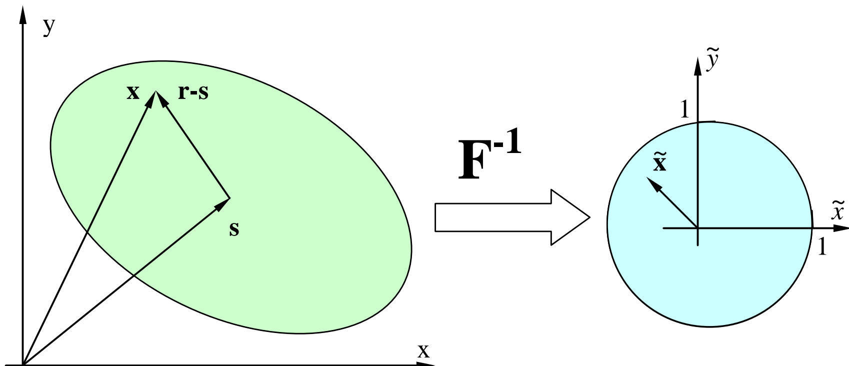


Figure 10: Performing inverse affine transform to change some ellipsoidal region in the space into a unit ball.

8.3.1.1 Gradient of a function of transformed parameters

When we have some function $\tilde{f}(\tilde{\mathbf{x}})$ defined on the domain of the affine transformation **F** and we derive from this function a function that acts on the codomain of transformation **F**, such that

$$f(\mathbf{x}) = \tilde{f}\left(\mathbf{F}^{-1}(\mathbf{x})\right); \tilde{\mathbf{x}} = \mathbf{F}^{-1}(\mathbf{x}) = \mathbf{A}^{-1}(\mathbf{x}-\mathbf{s}) \ , \tag{110}$$

then, according to (134), the gradient of f is[1]

$$\nabla_x f(\mathbf{x}) = \mathbf{A}^{-1T} \nabla_{\tilde{x}} \tilde{f}(\tilde{\mathbf{x}}) = \mathbf{A}^{-1T} \nabla_{\tilde{x}} \tilde{f}\left(\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right) \ . \tag{111}$$

This situation is the most common one for using functions of affine transformed co-ordinates. Usually, we define the transformation **F** such that we perform the operation of interest in the codomain of **F**, such that its domain represents a kind of *reference domain* (or region or space) and its codomain represents the *physical domain*. The reference domain usually serves for definition of templates for specific operations (such as sampling) or to define some template functions (such as weighting functions).

This is particularly useful when the operations act on a restricted domain or when functions of interest have some characteristic domain of interest (such as domain when the function is non-zero or has a significant value, as is the case with weighting functions). In such cases, we can define template operations and functions uniformly for some special domain of interest, and derive actual operations or functions that we need in the physical space by affine transformation of co-ordinates.

Affine transformations can be used for such purpose when it is easy to define operations or functions on a unit ball, and when the limited region of interest is bounded by a hyper-ellipsoid with an arbitrary center. Indeed this if the most general case that we need in approximation-based optimization for sampling, definition of weighting functions, and definition of the restricted step constraint.

If we have, in the contrary, a function $\tilde{f}(\tilde{\mathbf{x}})$ defined on the domain of transformation **F** and we derive from this function the equivalent function on the domain of **F**, such that

$$\tilde{f}(\tilde{\mathbf{x}}) = f\left(\mathbf{F}(\tilde{\mathbf{x}})\right); \mathbf{x} = \mathbf{F}(\tilde{\mathbf{x}}) = \mathbf{A}\tilde{\mathbf{x}} + \mathbf{s} \ , \tag{112}$$

then gradient of $\tilde{f}$ is

$$\nabla_{\tilde{x}} \tilde{f}(\tilde{\mathbf{x}}) = \mathbf{A}^T \nabla_x f(\mathbf{x}) = \mathbf{A}^T \nabla_x f(\mathbf{A}\mathbf{x} + \mathbf{s}) \ . \tag{113}$$

8.3.1.2 Implementation of affine transformations in IOptLib

[1] Note that domain and codomain of **F** can have different dimensions, but usually the dimensions will be the same. In the case of different dimensions, the transformation is not invertible, which significantly limits its use.

Implementation of the affine transformations in the IOptLib accounts for efficient treatment of special cases where the transformation matrix is simple scaling or a diagonal matrix. It also allows for multiplication of some matrix by a diagonal matrix or a scalar factor. The matrix A can be written as

$$\mathbf{A} = c_1\, c\, \mathbf{D}\,\tilde{\mathbf{A}}\ ;\ \mathbf{D} = diag(\mathbf{d}) \tag{114}$$

Multiplication with diagonal matrix:
In general, multiplication with a diagonal matrix is not commutative:

$$[\mathbf{AD}]_{ij}=a_{ij}d_{jj},\ [\mathbf{DA}]_{ij}=a_{ij}d_{ii}$$

Efen if **A** is symmetric, its product with a diagonal matrix **D** is not commutative. In this case $(DA)^T=AD$.

8.3.1.3 Sampling

The simplest sampling is uniformly distributed random sampling. On an arbitrary ellipsoidal domain, we usually perform sampling that is uniformly distributed on its inverse image – the unit ball. Sampling points are therefore specified on a unit ball and then transformed to the actual sampling region by affine transform:

$$\mathbf{x}_i = \mathbf{F}(\tilde{\mathbf{x}}_i)\,. \tag{115}$$

Sometimes we try to improve sampling by solving a specific minimal particle potential problem (in order to maximize distances between sampling points). Usually a number of existent (static) points $\bar{\mathbf{x}}_l$ are considered in such a problem beside the new points we want to position in an optimal way. In this case, we first inverse transform $\bar{\mathbf{x}}_l$, solve the minimal potential problem for new sampling points on an unit ball, and transform the calculated sampling points to the actual sampling region:

$$\begin{aligned}
&\tilde{\bar{\mathbf{x}}}_l = \mathbf{F}^{-1}(\bar{\mathbf{x}}_l)\ \forall\, l \\
&\{\tilde{\mathbf{x}}_k\} = \arg\min_{\{\tilde{\mathbf{x}}_k\}}\Big(\mathbf{P}\big(\{\tilde{\bar{\mathbf{x}}}_l\},\{\tilde{\mathbf{x}}_k\}\big)\Big) \\
&\mathbf{x}_k = \mathbf{F}(\mathbf{x}_k)\ \forall\, k
\end{aligned} \tag{116}$$

8.3.1.4 Weighting functions & calculation of weights

Typically, weighting functions in a multidimensional space are derived from one dimensional functions of an argument that is a norm of the inverse transformed vector. In such a way, we obtain functions whose iso-surfaces are ellipsoids.

$$w(\mathbf{x}) = f\left(\left\|\mathbf{F}^{-1}(\mathbf{x})\right\|_2\right) = f\left(\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|_2\right) \tag{117}$$

$$\begin{aligned}\nabla w(\mathbf{x}) =& = f'\left(\left\|\mathbf{F}^{-1}(\mathbf{x})\right\|_2\right)\nabla\left(\left\|\mathbf{F}^{-1}(\mathbf{x})\right\|_2\right) = \\ & f'\left(\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|\right)\frac{\mathbf{A}^{-1T}\,\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})}{\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|}\end{aligned}\;. \tag{118}$$

$$w(\mathbf{x}) = f\left(\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|\right) \tag{119}$$

We have

$$\nabla w(\mathbf{x}) = f'\left(\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|\right)\frac{\mathbf{A}^{-1T}\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})}{\left\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right\|} = f'\left(\left\|\tilde{\mathbf{x}}\right\|\right)\frac{\mathbf{A}^{-1T}\,\tilde{\mathbf{x}}}{\left\|\tilde{\mathbf{x}}\right\|_2}\;. \tag{120}$$

We have taken into account

$$\begin{aligned}&\nabla f\left(g(\mathbf{x})\right) = f'\left(g(\mathbf{x})\right)\nabla g(\mathbf{x}),\;\; \nabla\left\|\mathbf{x}\right\|_2 = \frac{\mathbf{x}}{\left\|\mathbf{x}\right\|_2},\\ &\nabla f\left(\mathbf{A}(\mathbf{x}-\mathbf{s})\right) = \mathbf{A}^T\,\nabla f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{A}(\mathbf{x}-\mathbf{s})}\end{aligned}\;. \tag{121}$$

If **A** is a symmetric real matrix then we can write **A**=**USU**^{T}, where **D** is a diagonal matrix whose elements are eigenvalues of **A**, and **U** is orthogonal matrix whose columns are corresponding normalized eigenvectors.

Another way to derive expression (120) is to consider the weighting function in the reference co-ordinate system, i.e.

$$w_0(\tilde{\mathbf{x}}) = f\left(\left\|\tilde{\mathbf{x}}\right\|_2\right) \tag{122}$$

and derive the actual weighting function by transformation of co-ordinates, i.e.

$$w(\mathbf{x}) = w_0\left(\mathbf{F}^{-1}\mathbf{x}\right) = w_0\left(\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right) \tag{123}$$

Then we have, according to (134):

$$\nabla_x\, w(\mathbf{x}) = \mathbf{A}^{-1T}\,\nabla_{\tilde{x}}\, w_0\left(\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right) \tag{124}$$

According to (140),

$$\nabla_{\tilde{x}} w_0(\tilde{\mathbf{x}}) = \left.\frac{d f(t)}{d t}\right|_{t=\|\tilde{\mathbf{x}}\|_2} \frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2} \tag{125}$$

and then

$$\nabla_x w(\mathbf{x}) = \mathbf{A}^{-1T} \left.\frac{d f}{d t}\right|_{t=\|\tilde{\mathbf{x}}\|_2} \frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2} = \left.\frac{d f}{d t}\right|_{t=\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\|_2} \frac{\mathbf{A}^{-1T}\left(\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\right)}{\|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\|_2} . \tag{126}$$

8.3.1.5 Restricted step constraint

Restricted step constraint is defined as

$$\|\mathbf{F}^{-1}(\mathbf{x})\|_2^2 = \|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\|_2^2 \leq 1 , \tag{127}$$

therefore the corresponding constraint function is

$$c_r(\mathbf{x}) = \|\mathbf{F}^{-1}(\mathbf{x})\|_2^2 - 1 = \|\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s})\|_2^2 - 1 . \tag{128}$$

Gradient of the constraint function is therefore

$$\nabla c_r(\mathbf{x}) = 2\mathbf{A}^{-1T}\mathbf{A}^{-1}(\mathbf{x}-\mathbf{s}) \tag{129}$$

$$\nabla \|\mathbf{A}\mathbf{x}\|_2^2 = 2\mathbf{A}^T\mathbf{A}\mathbf{x} . \tag{130}$$

8.4 Formulas for function gradients

Jacobian matrix of a vector function $\mathbf{F}: {}^m \rightarrow {}^n$, :

$$\mathbf{F}: \ {}^m \to \ {}^n ; \mathbf{F} = \left(F_1, F_2, ..., F_n\right)$$
$$\mathbf{J}\left(\mathbf{F}(\mathbf{x})\right) = \begin{bmatrix} \nabla F_1(\mathbf{x}) \\ \nabla F_2(\mathbf{x}) \\ \cdots \\ \nabla F_m(\mathbf{x}) \end{bmatrix}; \ \left[\ \mathbf{J}\left(\mathbf{F}(\mathbf{x})\right)\right]_{ij} = \frac{\partial F_i(\mathbf{x})}{\partial x_j}, \ i = 1, ..., n, \ j = 1, ..., m \ . \tag{131}$$

Gradient of composition of scalar functions:

$$\nabla f\left(g(\mathbf{x})\right) = f'\left(g(\mathbf{x})\right) \nabla g(\mathbf{x}) \ . \tag{132}$$

Gradient of composition of scalar and vector function:

$$\nabla f\left(\mathbf{g}(\mathbf{x})\right) = \left(D\mathbf{g}(\mathbf{x})\right)^T \nabla f(\mathbf{t})\Big|_{\mathbf{t}=\mathbf{g}(\mathbf{x})} \ , \tag{133}$$

where

$$D\left(\mathbf{g}(\mathbf{x})\right) = J\left(\mathbf{g}(\mathbf{x})\right)$$

is the Jacobian matrix of g.

Gradient of a linearly transformed function:

$$\nabla_x f\left(\mathbf{A}\mathbf{x}\right) = \mathbf{A}^T \nabla f(\mathbf{t})\Big|_{\mathbf{t}=\mathbf{A}\mathbf{x}} \tag{134}$$

Gradient of norm of a vector is

$$\nabla \left\|\mathbf{x}\right\|_2 = \frac{\mathbf{x}}{\left\|\mathbf{x}\right\|_2}, \ \mathbf{x} \neq 0 \tag{135}$$

and from this it follows

$$\nabla \left\|\mathbf{A}\mathbf{x}\right\|_2 = \mathbf{A}^T \frac{\mathbf{A}\mathbf{x}}{\left\|\mathbf{A}\mathbf{x}\right\|_2}, \ \mathbf{x} \neq 0 \tag{136}$$

and then, taking into account (132),

$$\nabla f\left(\left\|\mathbf{A}\mathbf{x}\right\|_2\right) = f'\left(\left\|\mathbf{A}\mathbf{x}\right\|_2\right) \mathbf{A}^T \frac{\mathbf{A}\mathbf{x}}{\left\|\mathbf{A}\mathbf{x}\right\|_2}, \ \mathbf{x} \neq 0 \tag{137}$$

Gradient of a square norm is

$$\nabla \|\mathbf{x}\|_2^2 = 2\mathbf{x} \tag{138}$$

and therefore

$$\nabla \|\mathbf{A}\,\mathbf{x}\|_2^2 = 2\mathbf{A}^T \mathbf{A}\,\mathbf{x}\ . \tag{139}$$

From (135) and (132) we also have

$$\nabla f\left(\|\mathbf{x}\|_2\right) = \left.\frac{d\,f(t)}{d\,t}\right|_{t=\|\mathbf{x}\|_2} \nabla \|\mathbf{x}\|_2 = \left.\frac{d\,f(t)}{d\,t}\right|_{t=\|\mathbf{x}\|_2} \frac{\mathbf{x}}{\|\mathbf{x}\|_2} \tag{140}$$

9 APPENDIX: COMMON TYPES AND RELATED MODULES

9.1 Introduction

9.1.1 Comments on ANSI C

ANSI C is a highly portable, plain, basic and logical high level programming language. It does not provide or enforce many artificial constructs above the machine level (there is no explicit object oriented programming or OOP support, while one can still keep programming style that is close to these concepts), but still provides what is necessary for high level programming. This includes control flow constructs necessary for *structured programming*, *dynamic memory allocation* and definition *compound data types* for combining heterogeneous data in arbitrary arranged packages and easy addressing, passing and access of individual parts or data conglomerates as whole. Through function pointers that can be kept in static variables and function arguments, data and functions may be treated in more unified way (such as it is common in OOP).

Since C does not enforce any particular programming style and it provides relatively low level but still highly human-readable access to hardware capabilities, the code can be made either very efficient or with interfaces exhibiting high levels of abstraction or encapsulation. Since C is very commonly used (free and commercial compilers are available on almost all platforms), there are also well established procedures for connecting C code with software coded in other programming languages (although this may be very platform dependent). With other words, programming a library in C lets a lot of freedom and generality, and this is the main reason for choice of this language. If it turns very beneficial to have a library with the same functionality as "IOptLib" in another programming language, this should be most entirely solved by building an

interface library in that language. This would enable uninterrupted development of the library and continuation of its integrity.

One of the most typical and for many people most acute feature of programming in C is that one must explicitly handle *dynamic memory allocation* as one goes along. Unless in very simple cases, this also means that memory must be explicitly released when not used any more (there is no implicit allocation and built-in garbage collection). This requires some additional programming discipline. Lack of discipline can lead to serious troubles such as memory leaks (software accumulates allocated memory without de-allocating (releasing) it when not needed any more, and can finally spend all available resources, which leads to crash of the program or the whole system), or memory access errors. The latter typically occur when data storage that is attempted to be used has not been allocated before or not enough space has been allocated for the data, or when we try to access (read or write) memory that has already been de-allocated (released). Another common memory handling error is that one sets by mistake two pointers to point to the same memory location, but they should logically represent different data. Storing one piece of data therefore unexpectedly changes another.

9.1.1.1 About pointers in C

How to deal with pointers in C is the matter of knowing the programming language. It is not intention of this document to teach programming or explain syntax rules of some particular language. However, understanding pointers is so important for using the library, and many people using other programming languages are so unfamiliar with these concepts that it may be appropriate to include a brief recapitulation of the subject.

There is nothing mysterious about pointers – they are just pieces of data of particular kind that carry some information, exactly the same as numbers. They can be stored in variables just as other pieces of data. Difference is that the information they carry is somehow more abstract and related to the machine architecture rather than to the model and procedures we want to build with our code. *Pointers contain memory addresses*, which can be used to access other kinds of data that resides in memory during program execution. Basically, all data that are manipulated by the executed program are contained in memory at some time, only that referencing the corresponding memory locations is usually performed indirectly, not through addressing memory but through names of variables. On the machine level, data access still reduces to addressing memory locations and transferring pieces of memory contents, since variables are just an artificial construct to aid programming.

One of the main reasons why to deal with pointers is to enable dynamic allocation of memory. With this concept, we don't need to know in advance how many data will be treated and how much memory will be needed to perform a given set of operations. At any point in the code, we can ask the operating system to *allocate* an additional piece of memory of a specified size for our use, use this peace of memory through its address, and later (when the memory is not needed any more) tell the system to *de-allocate (release)* this piece of memory, so that it can be used for other purposes[1]. For example, we need to statistically analyze data read from disk, but don't know in advance how many data there will be. Normally, we never operate with particular memory addresses (because this task is taken over by the operating system), but we are still aware that the

[1] Some languages enable this implicitly, without the need to operate with something thought of as memory addresses.

value (content) of the pointer represents the address (i.e. position in the memory) where a given piece of data is located.

Computer memory can be imagined as a contiguous and homogeneous block of equal storing units[1], therefore different addresses (locations) are equivalent. However, similarly as programming language distinguish between data units of different types[2] in order to assign them human understandable meaning, different types of pointers may be declared according to the type of data they are intended to point to. Let us consider the following code:

Example 6:

```
int a=1, b=2, c;
int *ptr1=NULL, *ptr2=NULL;
ptr1=&a;
prt2=&b;
*ptr1=44;
c=*ptr2;
printf("Value of a: %i.\n",a);
printf("Value of c: %i.\n",c);
```

Execution of the above code generates the following output:

```
Value of a: 44.
Value of c: 2.
```

Integer variables `a`, `b` and `c` are defined in the first line, and `a` and `b` are initialized to 1 and 2, respectively.

In the second line, two pointers to integer data are defined, namely *ptr1* and *ptr2*. This is done by use of the * *(de-reference) operator*, which means "take the value the variable points to". The declaration `int *ptr1` should be read as »define a variable `ptr1` such taht `*ptr1` (i.e. the value that ptr1 points to) is of type `int` (which denotes a signed integer in C)«. Both pointers are initialized to `NULL`, which is a pre-defined value for an *undefined address* in C. If the value of some pointer is NULL, then one knows that it does not point to a valid location in the memeory, and may not refer to what that pointer points to.

In the sequel, addresses of variables `a` and `b` are assigned to pointer variables `ptr1` and `ptr2`, respectively. The *address operator &* is used to obtain the storage address, and the statement `ptr1=&a;` means "get the address of variable `a` and store it in the variable `ptr1`". After assignment, poiter variables hold the addresses of the portions of memory where the variables `a` and `b` are stored. The actual addresses depends on the compiler, linker, operating system and are not important for the programmer. It only matters what data is addressed by the pointers.

The statement `*ptr1=44;` stores the number 44 at the *location pointed to by* `ptr1`. Since the type of ptr1 is »pointer to int«, the value is treated as signed integer (which affects the bit representation and length of data piece that is written to the memory). And since the variable `a` is kept at the location pointed to by `ptr1` (because of the assignement `ptr1=&a;`), this assignment also sets the value of `a` to 44. The *dereference operator* * is used to refer to what a pointer points to.

[1] Byte is usually taken as the basic unit on the level of programming language.
[2] While on the machine level, all information is uniformly represented in as a binary sequences of different lengths. On this level, meaning is assigned to the data only through operations used to manipulate the information (e.g. by integer addition of two integer numbers).

Let us mention again that the declared type of the pointer defines how the dereferenced storage is treated. In this case it is treated as data of type `int`, which affects the length of the data and the bit representation of the value.

The statement `c=*ptr2` takes the value pointed to by `ptr2` and assigns it to the variable `c`. Since ptr2 points to the location where variable `b` is stored, and the value of this variable is 2, 2 ia assigned to `c`.

Since the point where the address of a variable is assigned to a pointer, referencing variable or referencing what pointer points to is the same thing. Therefore, whatever is assigned to a variable will affect the value referenced through the pointer and vice versa.

The above example with corresponding comments does not show the true reasons why pointers are useful. There are two basic reasons for that. The first one is dynamic allocation, and the second one is the possibility of passing (e.g. through function arguments) arbitrarily large conglomerates of heterogeneous data by passing a single pointer. Let us explain dynamic memory allocation first.

Dynamic allocation of arrays:

Consider the following code:

Example 7:

```
void printsumseries (int n)
{
  int i, j, *tab=NULL;
  if (n<1)
    return;  /* function does nothing if n<1 */
  tab=malloc(n*sizeof(tab));   /* allocation of table */
  tab[0]=1;  /* initialization of the first element */
  for (i=1; i<n; ++i)
  {
    tab[i]=0;  /* initialization of element i+1 */
    for (j=0; j<i; ++j)
      tab[i]=tab[i]+(i+1)*(j+1)*tab[j];  /* add term j+1 of the sum */
  }
  printf("The first %i elements of the series:\n");
  for (i=0; i<n; ++i)
    printf("%i: %i \n", i+1, tab[i]);
  free(tab);
  tab=NULL;
}

...
printsumseries(4);
```

In the above code, a function is defined for calculation and printing of the first `n` elements of the series defined by

$$s_1 = 1; \quad s_i = \sum_{j=1}^{i-1} (i+j) * s_j, \; i = 1, ..., n \; .$$

The number of terms to be printed is passed as an argument of the function. We can see that for calculation of any term of the series, we need to have all previous terms, therefore we need a table of numbers for storing these terms. Since we don't know in advance how many terms we will need to calculate and print, we also don't know the length of the auxiliary table necessary to store the evaluated terms of the series. We will solve this by dynamically allocating the space for the table each time the function is called.

We define the auxiliary table `tab` as a pointer to `int`. Then we allocate the space for `n` integers (exactly as many as we need) and set tab to the address of the beginning of dynamically allocated space. This is done by the statement `tab=malloc(n*sizeof(*tab));`. The standard function `malloc` instructs the operating system to *allocate a contiguous memory block of a specific length* (specified as an argument that defines the length *in number of bytes*) for the program use. The function *returns the address of the allocated memory* (which we assign to the pointer `tab`). In order to allocate just enough space necessary for storing `n` integers, the operator `sizeof` is used that returns the *size occupied by a data unit* of a given data type (which is system dependent). In order to determine the size of a single data unit, the `sizeof` operator takes as argument either the *name of the type or reference to a variable* of a given type for which the occupied storage size is requested. We used the latter, i.e. we pass `*tab` as argument, therefore the operator returns the size of a data unit pointed to by the pointer tab (which is the size of type `int`). The advantage of referring a variable instead of a type is that if we later decide to change the type of `tab` e.g. to `long *` (integer type `long` requires more storage space than `int` on some systems) then the size will still be correctly calculated without changing the code (if we stated the type, we should change the argument to `sizeof` according to new declaration).

After allocation of a memory block for the table (array) of `n` integers, calculation of the series terms is performed. Each calculated term is stored in the allocated array. Individual numbers (element of the array) are referenced through the pointer that points (holds the address of) to the beginning of the array, since pointers can automatically represent arrays in C. Reference to elements consists of the pointer name followed by index in square brackets. *Elements of arrays are counted from 0* rather than from 1, therefore `tab[1]` refers to the second integer element of the array.

Figure 11 schematically shows the situation in memory after allocation of the table and calculation of the first four terms of the series. Memory is viewed as a contiguous block of bytes, which are denoted as small squares. It is assumed that type `int` is four bytes long, therefore the allocated block of memory (shaded in yellow) is 16 bytes long. We can imagine this block divided to smaller blocks of 4 bytes, each of which will store one integer number. The address of the beginning of the allocated block is assigned to the pointer variable `tab`, which is stored in memory at some other location (unrelated to the location of the allocated memory block). We thus say that `tab` points to the allocated memory block. Since `tab` has been declared as pointer to `int`, we can use it to address successive integer elements of the array. In the figure, reference to the individual integer members of the array (the allocated block) are denoted, together with the calculated values (terms of the series) that are stored in these elements.

After calculation, elements of the series that had been stored in the array are printed out. After this, the array is not needed any more and is *released (de-allocated)* by the standard function `free`. This function instructs the system that a given memory block, which had been dynamically allocated before, is not needed any more. The system releases the allocated memory, which can then be re-used for other purposes (e.g. it may be a constituent part of another dynamically allocated

block of memory reserved at another point of program execution). A *valid pointer to the allocated memory* block must be passed to `free`. This function *may not be called* with a pointer argument that is not the address of dynamically allocated memory, and it may not be called again with the address of the same memory block that has already been released.

As a good programming practice, we set the pointer `tab` to NULL after de-allocation of the memory block it points to[1]. This habit is good for preventing attempts to access the memory that had been released or attempt to release this memory again (which would result to disastrous errors)[2]. A pre-defined value NULL is used to indicate that pointer does not point to a valid location. When we use a pointer in a portion of code that is much isolated from the parts where this pointer is otherwise manipulated (e.g. in some function to which the pointer is passed as argument), we can check the validity of the pointer by checking whether it is NULL. This strategy will only work well if we will strictly set all pointers that are not assigned valid addresses to NULL, which includes points at which the allocated memory pointed by pointers is released.

After execution of the last line of code, where the function is called with argument 4 (number of terms to be printed), the following output is generated:

```
The first 4 elements of the series:
1: 1
2: 3
3: 19
4: 156
```

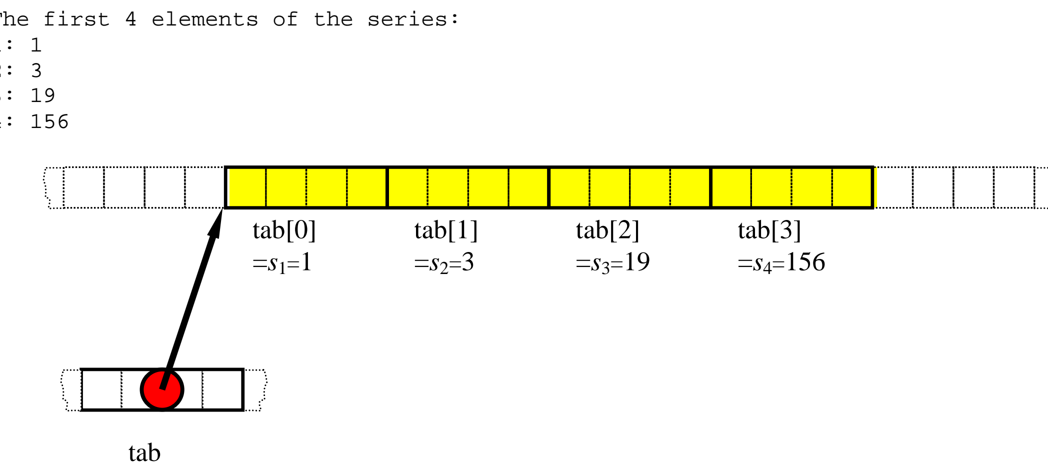


Figure 11: Memory scheme after allocation of space for an array of 4 integer numbers and calculation of the first four elements of the series (Example 7).

Notes on pointer arithmetic and addressing arrays:

[1] Although in this place such caution is not really necessary, since the memory is de-allocated just before the end of the scope of the variable that points to it and it is pretty sure that we will not do anything with this pointer after de-allocation. At least theoretically, we could extend the function definition by addition of some instructions at the end of the function. In this case, setting the pointer to NULL could turn useful, since we could check the pointer and prevent access to the memory that has been de-allocated.

[2] After calling free, the pointer that is passed as argument does not change and therefore still points to the same location. However, the memory at that location is released and it is no longer valid for the program to access that memory.

As mentioned, pointers with specified type can automatically be interpreted as reference points of arrays of elements of a given type. It is considered that such pointers point to blocks of memory containing a number of contiguously arranged elements of a given type. For example, `tab[i]` in the above place refers to the element *i*+1 of the array of integers, thought to begin at the place pointed to by `tab`. Addition of 1 follows from the fact that elements of arrays are counted from 0.

The first element of the array pointed to by `tab` can be equally well referred to by using the *de-reference operator* *, i.e. as `*tab`, or by using the indexing operator *[]* , i.e. as `tab[0]`. On basis of this, *pointer arithmetic* or addition and subtraction of integers is defined for pointers in such a way that addition of 1 is identical to increment of the pointed address by the size of the pointed data type. Therefore, the following two references both refer to identical piece of data stored in memory[1]:

```
tab[i]
*(tab+i)
```

When referencing arrays through pointers, we must take care that we don't reference the elements whose indices exceed actual array bounds. E.g., if address of a variable `a` of type `int` is assigned to the pointer `tab`, we may not refer to the second element of the array pointed to by tab, i.e. `tab[i]`, since only a single integer is stored at a location where the pointer points. Similarly, we may not refer to `tab[n]` in the function defined in Example 7, because the size of the allocated memory whose starting address was assign to tab is only sufficient for `n` elements. The compiler does not control whether we refer to elements out of the array bounds because there is now way to establish that. We can, however, declare pointer variables that are intended to point to arrays of fixed length, which is done as follows:

```
int itab[3];
```

By the above code, a pointer to an array of three integers `itab` is defined. The space for array is allocated right at the definition point because the size is fixed and known in advance. In this case, we can not refer to an element that exceeds the bounds (this would generate a compiler error, e.g. `itab[3]` or `itab[10]`) or assign some other address to the array (e.g., statements `itab=&I;` or `itab=tab;` are illegal). We can only assign new values to elements of such table or get their values, e.g.

```
itab[1]=tab[1];
tab[2]=itab[1];
```

Pointers to compound data units:

In C, we can combine arbitrary data units of different types into conglomerates called *structures*. The point is in creating objects that can represent complex things and referring to these objects as single units, which highly simplifies manipulation and putting things together.

Pointers play two roles in this concept. One is the ability of dynamic extension of data conglomerates (e.g. dynamically allocated arrays with variable size, and some pieces of data may at

[1] This also explains why array elements are counted from 0.

different stages either be allocated and carry useful information or not). The other role is passing information to different execution levels (e.g. to functions, which do a specific job and either use complex data or generate complex results) by passing (i.e. actually copying) between these levels only small data pieces (pointers to the structured data) rather than complete data. A simple demonstration of both roles is given by the definition of `vector` and `matrix` types (Section 9.2).

The void * type:

ANSI C defines the void data type (meaning unspecified, none or any), which can only be used as an imaginary type of return values of functions that don't return anything[1] or to declare pointers for which the type of the data they point to is not defined. Such pointers can be used to point to any type of data, and are useful e.g. for defining container objects for carrying different types of data (such as stacks, see section 9.3).

9.1.2 Followed programming rules

We will not explain any details about memory handling in C. We consider this a pre-requisite to use of the library, and users can consult any book on C for this purpose. However, we would like to mention some rules of good practice that help keeping programming discipline and the rules that mainly apply for this library.

Sometimes quite complex data organization is necessary in order to keep things general as well as efficient and easy to use. For the task of memory organization, appropriate compound types are defined. Usually these will be structured types, with pointers to them declared as separate types. Pointers will mainly be used rather than structures themselves. Data structures may be highly complex and nested several levels (one data type may contain pointers to other data types, which again contain pointers to other data types, etc. In order to manage complexity, for each structure type there will be basic operations defined, which in particular includes storage allocation and de-allocation.

Allocation and de-allocation should always be made by the provided functions. This means that exact definition of a data type may change (i.e. a compound data type may be extended), but this will not affect correct memory handling because the narrow set of basic operations will be updated almost simultaneously.

It is a common rule that every dynamically allocated piece of data must have a unique basic handle, i.e. a pointer to the data through which it is allocated or de-allocated. Other auxiliary pointers may point to the same data, but this will be merely used in oder to assist access to individual parts of the data in the case of nested pointers and when type casting is necessary.

As a good programming practice, all pointers that are intended for basic handles of dynamically allocated data should be initialized to NULL. Whenever such pointer is nod NULL, it is considered that it points to allocated data and can therefore be de-allocated when the data is not needed any more. It is a good practice that the basic handle is set to NULL immediately after de-allocation. This rule can only be skipped when de-allocation is made right before the end of a scope

[1] In some language such functions are referred to as procedures (which do something but do not return a value), in contrast with functions, which do something and return some value.

of the pointer variable, which will therefore not be accessible shortly after de-allocation of the pointed data.

Functions for allocation and de-allocation of complex nested types must be implemented hierarchically in such a way that the above rule is followed strictly. De-allocation functions should take address of the data pointer rather than the pointer itself as an argument. After de-allocation (usually by using the standard `free()` function, the pointer will also be set to NULL. If the data to be de-allocated contains pointers to other dynamically allocated data structures, these will be de-allocated first.

Some data types defined in this library (such as `stack`, see Subsection 9.3) act as containers that can hold pointers to different types of data. Objects of such types may be used in both ways – to hold auxiliary pointers (e.g. to aid operations such as sorting) or to hold many basic handles of data of the same type. Therefore, two ways of de-allocation are supported for such data objects. One way is de-allocation of merely the container itself, without de-allocating its elements (because their basic handles are somewhere else). Another way is de-allocation of the contained elements followed by de-allocation of the container object. Since elements of such container objects are represented as pointers of indefinite type (*void* *), for the second method we need to provide a function that de-allocates individual elements.

9.1.3 Work in multi-thread environment

A process[1] can have several parallel execution threads. These threads share the same process data, but they execute in a parallel manner, which is in the same way as distinct processes. The system alternately assigns chunks of processing time to individual threads of the same process in a similar way than to other processes that run simultaneously at a specific moment. Running a process in parallel threads have many advantages. For example, when a given thread of numerical simulation is performing calculations, some other thread can simultaneously perform processing of already calculated results and can provide their graphical representation to the user[2].

The main problem in execution in multi-threaded regime is *synchronization of data access*. For example, as the simulation thread proceeds, it may delete the results older than the past three iterations. If the thread for graphical representation attempted to access these results after the simulation thread has deleted them, an memory access error would appear.

As there are many different examples of almost inevitable use of parallel threads, there are also very different ways of *sharing data* between multiple threads. However, we can define some general rules for multi-thread environment. The first rule is that only one thread may owe the main

[1] A process is an image of a program in memory that is made by the system when executing the program, together with the corresponding data. In multi-tasking environments many processes are executing in parallel, and the same program (an executable file on the disk or other storage) can be carried out by several processes in the same time (e.g. several identical simulations with different input data can be run at the same time).

[2] This function could be implemented serially, e.g. by inserting chunks of code that handle user requests and graphical representation between the code that performs calculation. However, this would be much more complicated (and in more complex systems, practically impossible) to implement. Execution in parallel threads enables different tasks to be implemented independently while they are still executed in parallel within the same process and therefore have the ability to access the same data.

handle of any dynamically allocated object, therefore de-allocation of that object can only be performed in the main thread. The next rule is that write access to the data must be serialized to prevent unexpected change of the data by another thread when one thread is using the data.

One measure to insure proper synchronization is ***locking*** of the data when it is used, which prevents simultaneous access of the locked data by parallel threads. Locking is performed by integer locks whose state defines whether the corresponding data can be accessed or not. Locking and unlocking is performed by macros `m_threadunlock` and `m_threadlock` (defined in `sysint.h`; see Example 8). Both macros take the lock as argument. The first macro waits until the lock is released and then sets the lock (locks the data). When a lock is set, an attempt to lock it from another thread will block until the lock is released by the thread that had set the lock. The second macro releases the lock that is set by the first macro. It is the responsibility of the user that ***macros are strictly called in pairs lock/unlock*** (with code that deals with locked data put between the matching calls) and that the ***locking is never performed successively within the same thread unless unlocking is performed between*** (this would cause blocking of the thread forever). The most *common error is that locking is repeated in a nested call*, i.e. it is called in a function that is called by the function that performs locking (or in an arbitrarily deeper level). A condition for data locking to work is that it is used strictly and consistently[1].

Macro `m_threadlock` that sets the lock will block execution until the lock that it is setting is released (if no other thread has locked the same lock then the lock is set and execution of the thread that called the macro is continued immediately). After the lock is set, eventual calls to this macro would block until the lock is released by the call to `m_threadunlock`. If several threads attempt to set the lock at approximately the same time, only one of them will succeed immediately while all others will block (wait) until this thread releases the lock (between that, the thread that set the lock would usually perform some tasks on the data that is related to the lock). After the lock is released, another thread will be succeed to set the lock (but only one, again), while other threads will continua to block until that thread releases the lock, and so forth. Locking therefore provides means of serializing otherwise parallel execution of code at given critical moments when shared data needs to be accessed.

Example 8: locking of data for synchronizing parallel threads.

```
int lock=0;  /* initialization to 0 is obligatory */
void *data;
...  /* prepare data */
m_threadlock(lock);  /* set the lock */
```

[1] This means that all functions that may eventually use the same data during their execution, ensure by locking that the data can not be accessed during execution of critical function code by other threads that could eventually modify the data. All functions that modify the data must lock it before doing that, which ensures blocking until the lock that is eventually set by another threads is released. This means that either the data is modified only after the tasks performed in another threads that would eventually use the data are finished and release the lock, or it is modified before that (i.e. function in other threads block until the modification of data is finished). Locking mechanism does not itself prevents access to data by parallel threads, it only works if other threads use the same lock for checking and claming access rights for the data.

```
...  /* Do something with data; because the lock is set, other threads that
            would try to set the lock, would block until the lock is released
            */
m_threadunlock(lock);  /* release the lock, now some other thread that has
            eventually set the lock will continue execution */
```

Different handling rules for locking data:

More complex systems that handle a large number of related tasks usually have some static data that define the state of the system and are handled by system utilities[1]. In order to prevent unsynchronized access, data may be locked on several levels. Sometimes there is a lock for the whole systems and several special locks for particular smaller parts systems. In order to prevent conflicts by nested access to the same lock, it must be exactly defined which functions may set which locks. The main lock for the whole system can be only accessed by basic utility functions that are provided in the main module of the system.

Locking individual data objects may be performed through the locks that are part of the object (i.e. fields of the structured data types, typically named `lock`). In order to prevent nested locking conflicts, there may be separate locks for groups of different kinds of tasks[2] that can be performed on a given type of data objects. Sometimes it is agreed that locks are not set by the lower level functions, but must strictly be set by higher level functions.

Probably the best practice is using *functions with twofold locking operation*. We can provide utilities that work with data objects of a given type in such a way that the caller can indicate through an appropriate function argument whether the utility function should lock the object or not. Then these functions are called in such a way that they don't lock the object if the lock has already been set by the calling code, and such that they lock it elsewhere. Use of twofold locking operation is illustrated by Example 9.

Example 9: Using twofold locking mechanism on linear transformation data object.

```
lintransfdata ld;
ld=newlintransfdata();
...  /* prepare the data, install it in the system */
...
m_threadlock(ld->lock);  /* lock the object */
...  /* use the data object */
my_func(ld,0);  /* call a function that performs operations on data object,
            indicate by the last argument that the function should not lock
            the object because it is already locked */
...
m_threadunlock(ld->lock);  /* unlock the object */
...
displintransfdata(&ld);  /* when not needed any more, de-allocate the data */

...

/* Definition of function my_func: */
void myfunc(lintransfdata ld, int dolock)
{
```

[1] An example is registration system of optimization problems, Section 7.1.
[2] Don't be mislead by "groups". There may be only one lock and two groups of utilities, those that may lock the data and those that may not.

```
    if (dolock)
      m_threadlock(ld->lock);  /* lock the object if this is instructed by the
              corresponding argument */
    ...  /* perform the task (use data on ld) */
    if (dolock)
      m_threadunlock(ld->lock);  /* unlock the object ld if it has been locked
              within this function */
  }
```

9.2 Vector and Matrix Operations

Vector and matrix types are declared in `vec.h` and `mat.h` in the following way:

```
typedef struct _vector {
    int d;      /* dimension (num. of comp.) */
    double * v; /* table of elements, STARTS WITH 1! */
} *vector;

typedef struct _matrix {
    int d1,d2;    /* dimensions (num. of rows and num. of columns) */
    double ** m;  /* table of pointers to lines, COUNT FROM 1! */
    double *comp; /* pointer to components, 0 offset; NEVER ACCESS DIRECTLY!
            */
} *matrix;
```

Only matrix and vector elements may be set directly. Dimensions should always be set by the appropriate library functions for allocation or re-allocation (resize) of vectors and matrix objects. Dimensions and components my however be obtained (read) directly.

Basic operations such as memory handling are defined in the header files `vec.h` and `mat.h`. Vector components are accessed through the field `(…)->v`, which is an array of elements of type *double*. Matrix components are accessed through the field `(…)->m`, which is an *array of pointers* to arrays of elements of type *double*. When a vector or a matrix of given dimension is created (allocated) the storage for components is allocated simultaneously. Array pointers are decremented by one after allocation, therefore ***elements are counted from 1*** (not from 0 as it is common in C).

9.2.1 Allocation and access to elements:

Let us have the following code:

```
matrix A=NULL;
vector b=NULL;
A=getmatrix(5,5);
b=getvector(5);
```

Then `b->v[4]` refers to the fourth element of vector b and `A->m[2][3]` refers to the element of matrix A in the second row and the third column. Attempt to accessing `b->v[0]` or `A->m[6][1]` would be an error (because elements are counted from 1 and because A has been allocated with only 5 rows). Functions *getmatrix* and *getvector* were called for matrix and vector allocation. Both functions require dimension(s) as argument(s) and return pointers to dynamically allocated storage that can hold a matrix or a vector with specified dimensions:

```
vector getvector(int dim);
matrix getmatrix(int dim1,int dim2);
```

Most of the derived data types defined in the library have similar functions for creation of objects of these types. Majority of these functions allocate the data that can be allocated according to specified information and return object pointers, which must be assigned to chosen basic handles. Basic handles can be explicitly defined variables or elements of another complex types such as `stack` (Subsection 9.3).

We can set elements directly, e.g.

```
A->v[2][3]=10.5;
b->v[1]=0.23;
```

9.2.2 Reallocation and deletion:

We may not set dimensions directly, e.g. the statement "`A->d1=6`"is a hard mistake because it changes matrix dimension without re-allocating storage for its elements. Resizing of a matrix can be done e.g. by

```
resizematrix ( &A, 4, 6 );
```

By this call, matrix A is resized to hold 4 by 6 elements. Address of matrix basic handle (pointer to the data through which allocation is made) must be provided as the first argument, followed by the first (number of rows) and the second dimension (number of columns). Values of original elements should be stored before the resizing if we don't want to lose them. Re-allocation can be done in a longer way, by first deleting (de-allocating) the matrix and then allocating it with different dimensions:

```
dispmatrix ( &A );
A=getmatrix(4,6);
```

Resizing operations are defined for many data types intended to hold a variable number of related elements. The first argument is usually object address (address of basic handle must be provided, because the pointer itself may be changed, and the rest of the arguments define the new size (or dimensions) of the object. For matrices, size is defined by two dimensions, number of rows and number of columns, respectively.

De-allocation (deletion) operations are defined for almost all derived data types. The functions usually have suffix "*disp*" (that stands for *dispose*) followed by the name of the corresponding type (matrix in this case). Usually the only argument is the address of the object pointer. The deletion operations release all the dynamically memory occupied by the object (by nested deletion calls, if necessary), and *set object pointer to NULL.*

9.2.3 Copying and other operations:

Another basic operation defined for many derived data types is copying. We can create copies of matrix A in one of the following ways:

```
matrix C1=NULL, C2=NULL, M;
matrix A=NULL;
A=getmatrix(5,5);
A->m[1][1]=1.1; ... /* Assign components of A */
C2=getmatrix(2,50);
C1=copymatrix(A, NULL);  /* case 1 */
M=copymatrix(A, &C2);  /* case 2 */
```

The function `copymatrix` is declared as

```
matrix copymatrix(matrix m1, matrix *m2);
```

In the above code, new matrix pointers C1, C2, and M are defined. The first two pointers are intended for use as basic matrix handles and are therefore initialized to NULL, while M is intended just as auxiliary pointer, which will point to one of the copies of A, whose basic access handle will be assigned to C2.

The `copymatrix` function requires two arguments, the matrix to be copied (represented by a pointer of type `matrix`) and the address of the matrix which the original is copied to. In any case, function returns the pointer of the copy. If the second argument is NULL (case 1 in the above code) then a new matrix is dynamically allocated and its pointer is returned. In this case, the returned pointer must be assigned to some variable because it is the only pointer to the dynamically allocated matrix where the original of the copy is stored. If the second argument is not NULL (case 2 in the above code) then copy is stored at the location pointed to by this argument. If the second argument points to a NULL matrix then the matrix is allocated with the appropriate dimensions and then components of the first matrix are copied to the newly created one. If the matrix is already allocated, then consistency of dimensions are checked first. If necessary, the matrix pointed to by the second argument is re-allocated to have consistent dimensions, and then elements are copied. This is the case in the above example where a copy of matrix A is stored in matrix C2, which has been allocated with different dimensions than A. After the operation, C2 will point to a matrix with the same dimensions as A, holding its copy.

When the second argument is different than NULL, the returned pointer of the copy may be ignored, since the matrix is copied to the location specified by this argument. Sometimes it is still useful to store this pointer, e.g. to make the access to matrix elements easier (this is useful e.g. when the address specified by the second argument refers to a field of some object of a complex type, possibly nested in other objects).

Let us mention that the second argument may be address of the first arguments. Such call does not have any beneficial effect, but this possibility may be used at other unary or binary operations in order to save memory.

9.2.4 Binary operations:

Copying can be considered a simple *unary operation* on a data object (operand): a result of the operation performed on the *operand* (which is in this case a copy of the original with identical contents) is created (evaluated) and stored at the prescribed location. Many other operations are defined on vectors and matrices.

Most commonly used are *binary operations*, where an operation is performed on two operands. A simple example is matrix summation, which is implemented by the function `matsum0` that is declared in `matrixop.h` as follows:

```
matrix matsum0(matrix m1,matrix m2,matrix *m3);
```

The function `matsum0` is used in quite a similar manner as `copymatrix`. The code below may serve as an example:

```
Int dim1=5,dim2=7;
matrix A=NULL, B=NULL, S1=NULL, S2=NULL, S3=NULL, M;
A=getmatrix(dim1,dim2);  B=getmatrix(dim1,dim2);
A->m[1][1]=1.1; ... /* Set contents of A and B */
S2=getmatrix(1,1);
S1=matsum0(A, B, NULL);  /* case 1 */
matsum0(A, B, &S2);  /* case 2 */
M= matsum0(A, B, &B);  /* case 3 */
```

The first two arguments of `matsum0` are operands that are added together, in this case matrices A and B, which were allocated with consistent dimensions as it is necessary for summation. The third argument is the address of the matrix where the result is stored, but this argument may be NULL (unspecified). After performing summation, the function returns the matrix where the result is stored.

In case 1, the storage address is not specified, therefore the function creates a new dynamically allocated matrix, stores the result of summation in this matrix and returns it. In the above case, the result is assigned to a matrix variable S1. S1 should not be the basic handle or the only pointer to an allocated matrix because in this case that matrix would be lost by assignment (i.e. "hanging in space", causing a memory leak because there would be no handle to the matrix and no way to de-allocate it).

In case 2, the result of summation is stored to S2, which already held an allocated matrix. Because dimensions of S2 were different than the dimensions of the result of operation (which is in the case of summation equal to the dimensions of operands), the function first re-allocates the matrix S2 and then stores the result to S2. As usual, S2 is returned, but the returned value (matrix pointer) is not used in this case.

In case 3, the third argument is address of the second operand B, therefore the result (i.e. the sum of A and B) is stored back to B. The returned result (i.e. matrix B) is assigned to M, which is in

this case done only for demonstration. In the case of summation, the result can be stored in one of the operands without any side effects. The operation can be done in place – each pair of components is first added together and then stored, and the overwritten components are not needed any more. The situation is different e.g. in the case of multiplication where components of each matrix are used several times. Storing one component of the result would therefore change information needed for operation. Many vector and matrix binary operations are implemented in such a way that the necessary temporary storage is automatically allocated to perform the operation correctly when the result should be stored into one of the operands. However, allocation or de-allocation of the auxiliary storage may significantly affect the efficiency, therefore such situations should be avoided.

Equivalent operations as those described in this Section for matrices are also defined for vectors.

9.3 Stack Operations

Type `stack` is defined as a container type for different purposes. In computer terminology, the term stack is used for data structure where new elements may only be added (pushed) at its top (after the last element currently on the stack) and picked (popped) from the top. In this library, the `stack` type serves many different purposes (although push/pop operations are also implemented) and is in general used as a *table with variable number of elements*. *Elements may be pointers of any type* (e.g. vectors, matrices, pointers to double, pointers to numbers, etc.).

The `stack` type is defined as follows:

```
typedef struct _stack {
    int n,r;    /* number of occupied / allocated places */
    int ex;     /* excess at reallocation */
    void **s;   /* table of pointers, counting STARTS WITH 1 */
} *stack;
```

The type is adapted to pushing new elements at the top of the stack and getting them from the top. Elements can also be removed from or inserted in the middle, but this is not as efficient. The *table of elements is automatically resized* as necessary. If elements are added and the table is full, it is enlarged, but the number of allocated places is by excess (`(…)->ex`) greater than the minimum necessary. This mechanism is implemented for efficiency – in this way resizing is not necessary every time new elements are added. When elements are deleted, the size of the allocated space for the table is automatically reduced when the number of unnecessary spaces becomes twice smaller than the field `(…)->excess`. Therefore, greater excess means on average more efficient operation but also more unnecessary memory allocation, so a large excess will be chosen when it is expected that a stack will hold a lot of element which will be frequently added or taken from the stack.

Elements can be directly *accessed* through the table of elements `(…)->s`. Elements are counted from 1. For example, `st->s[4]` refers to the fourth element of the stack st. We need to take care that we don't attempt to access elements beyond the actual *number of elements* on the stack, which is obtained through `(…)->n`. `(…)->r` is the allocated size of the element table `(…)->s`, i.e. the number of elements (pointers) for which table is allocated. It can be equal or greater than `(…)->n`.

Field `(…)->ex` is usually not used directly, but is used by functions dealing with stacks in order to determine when to reduce space for the element table or how much (excessive) space to allocate when enlarging the table size. However, the field may be set directly (always to a positive number) in order to change the operation mode of the stack and improve efficiency.

9.3.1 Creation, deletion, resizing and copying

Creation:

Stacks can be created by function `newstack`, whose argument is excess (the number of excessive places at resize), which is assigned to the field `(…)->ex` of the created stack:

```
stack newstack(int excess);
```

The above function does not allocate any space for the element table: this is done when the first element is put on the stack (and in this occasion more space than necessary is allocated, namely by `(…)->ex` more elements more). Sometimes it is known in advance how many elements will be added to a stack at a time. In this case it is sensible to allocate element table for that many elements as necessary, which can be done by the function `newstackr`. The first argument is excess (a property of the created stack) while the second argument is the number of elements for which element table is allocated at creation:

```
stack newstackr(int excess,int r);
```

With the function `newstackrn` also the number of elements (…)->n is immediately set to the allocated size of the table, and elements are set to NULL. It is caller's responsibility to actually set the elements after this call.

Deletion:

There are various functions for *deletion* of stacks and their elements. Function `dispstack` *deletes the stack without affecting its elements* and may be used when elements on the stack are not basic handles for the objects they point to (i.e. there exist other pointers through which these elements can be accessed and eventually de-allocated, or pointers are addresses of static variables or fields of structures). The only argument to this function is the address of the stack (i.e. its pointer) to be de-allocated. Stack pointer is set to NULL after de-allocation.

Function `dispstackval` deletes all elements of the stack and sets the number of elements of the stack `(…)->n` to 0. Standard function `free` is used for de-allocation, therefore this function

can only be used when elements on the stack are simple elements. The stack whose elements are deleted is the only argument of the function.

Function `dispstackvalspec` acts similar as the `dispstackval`, except that the function for deletion of an individual element is specified as the second argument. If this argument is NULL then the standard `free` function is used for de-allocation of elements.

Function `dispstackall` de-allocates the stack together with all of its elements. The only argument is address of the stack, and the standard function `free` is used for de-allocation, therefore the function is only suitable when elements of the stack are simple pointers. Stack pointer is set to NULL after de-allocation.

Function `dispstackallspec` acts similar as `dispstackall`, except that a specific function, which is specified as the second argument, is used for de-allocation of stack elements. Declarations of these above mentioned de-allocation functions are as follows:

```
void dispstack(stack *st);
void dispstackval(stack st);
void dispstackvalspec(stack st,void (*disp) (void **));
void dispstackall(stack *st);
void dispstackallspec(stack *st,void (*disp) (void **));
```

When the *function for deletion of elements* is specified, it must usually be cast to the appropriate type (which is a void function whose only argument is a pointer to a void pointer), and the function must be such that this is possible. In this library, most of the functions for de-allocation of compound data objects are defined consistently with this type. Use of this is demonstrated below on the stack of matrix elements:

Example 10:

```
int i,j,dim=4,num=5;
stack st=NULL;
vector aux;  /* auxiliary vector pointer for easier access */
st=newstack(2);  /* allocate the stack */
for (i=1;i<=num;++i)  {
  aux=getvector(dim);   /* create a new vector */
  pushstack(st, aux);  /* add the created vector on the top of the stack */
  for (j=1;j<=aux->d;++j)
    aux->v[j]=(double) 10*i+j;  /* initialization of vector components */
}
... /* Do something with the stack and its elements */
/* When the vectors are not needed any more, we de-allocate them together
            with the containing stack: */
dispstackallspec(&(st), (void (*) (void **)) dispvector );
```

In the example below, a stack is created and four vectors are created and put on the stack. Variable `aux` is used only as an auxiliary pointer (handle) through which the created vectors are accessed when initializing their components.

After vectors are created and put on the stack, they can be manipulated in various ways. Vectors on the stack can be referenced directly through the table of elements `st->s`, but this may be a bit awkward because of the need of type casting (since elements of stack are declared as pointers of undefined type rather than vector pointers). Whenever we need to access some vector on

the stack, it is therefore useful to assign its pointer to an auxiliary variable that is declared vector. The pointer can be accessed directly (e.g. as `st->s[i]` where *i* is the position of vector element we want to address) or by the function `stackel`, which is somehow safer because the number of elements is checked and invalid access prevented:

```
aux=getvector(st,i);  /* get stack element */
if (aux!=NULL) {... /* do with a vector element whatever necessary */ }
```

When not needed any more, the whole table of vectors is de-allocated at once by calling `dispstackallspec`. Function `dispvector` is passed as the second argument to be used for deletion of individual vector elements, and is cast to the appropriate type.

Remark:

Creation of a vector and assignment of its pointer to an auxiliary pointer variable can be done in the same line as push it on the stack. This is something the language syntax enables, and the difference is more or less aesthetic. The latter way is briefer on the level of source code but maybe slightly less clear for sequential thinking:

```
pushstack(st, aux=getvector(dim));  /* create a vector and add it on the
          top of the stack */
```

Resizing:

A stack can be *resized* by the `resizestack` function, declared as follows:

```
void resizestack(stack *addrst,int excess,int n,void (*disp)(void **));
```

Argument `addrst` is the address of the stack to be resized. argument `excess` is the new excess parameter of the stack (assigned to its field `(…)->ex`). If it is smaller than 1 then it is set automatically according to the number of elements. Argument `n` specifies the requested number of elements after the operation. If `n` is less than the current number of elements on the stack then the added elements are set to NULL. If it is smaller then the excessive elements are deleted (de-allocated) by the function `disp`. If de-allocation function is not specified then de-allocation is done by the standard function `free`, but this is correct only in the case when elements are simple pointers (i.e. they don-t point to structures containing pointers to dynamically allocated memory). If we don't want the excessive elements to be de-allocated (e.g. when elements on stack are not basic handles but auxiliary pointers), we must manually set these elements to NULL.

Copying:

A complete stack of elements can be copied to another stack by a single call to `copystackspec`. As usual for copying operations, eventual contents of the target stacks are overwritten. To perform the operation, we need a methods for deletion and copying of individual elements, which are specified by the third and the fourth elements:

```
stack copystackspec(stack st1,stack *st2,void dispel(void **), void *copyel
          (void *,void **) );
```

Otherwise, the function acts in essentially the same manner as e.g. `copymatrix` described in Subsection 9.2.3. Additional arguments are required merely because in the case of copying

matrices, we know exactly what type of objects we deal with, while the type of stack elements is indefinite and we must explicitly prescribe the way how elements are de-allocated or copied.

This complication falls away when we don't want to create new objects that are copies of the current elements of the copied stack, but only want to have another stack containing the same pointers, e.g. to order elements in a different way which is more convenient for searching. The following function is used for this purpose:

```
stack copystack(stack st1,stack *st2);
```

Let us stress again that the target stack will just contain exact values of pointers on the original stack rather than poiners to dynamic copies of stack elements as it is the case with `copystackspec`. Therefore, we may not e.g. de-allocate elements of both stacks, because elements are the same and each pointer may be de-allocated only once. What concerns the stack itself (without the contents), `copystack` acts in a similar manner than `copymatrix` or `copystackspec`.

9.3.2 Element access

Stacks[1] are intended as container objects for any kind of pointer objects, therefore their elements are declared as pointers of indefinite type, i.e. `void *`. Elements of the stacks can be accessed directly (which is sometimes computationally more efficient), but in this case we may need to use *type casting* in order to tell the compiler what type of object we deal with.

Let us refer to **Example 10** and assume that we want to set the second component of the third vector element of the stack `st` to 9.28. We can do this by direct access to the vector element in the following way:

```
((vector) (st->s[3]))->v[3]=9.29;
```

Vector components are addressed through the field `(…)->v` and stack components are addressed through the field `(…)->s`. However, elements on the stack are of type `void *` and without a type cast, operator `->` (combination of dereferencing and field selection) itself would be illegal because what is pointed to by the stack element is simply a memory location that does not have any structure for the compiler. By type casting (stating the type in parentheses before the reference to the object) we provide (in a way enforce) unambiguous information about the structure of the pointed object through its ordered type (`vector` in this case, see Section 9.2 for declaration). The reference to a given element of the array (…)->v is therefore exactly defined, and vector component is set as intended. Type casting is only provision of information for the compiler. It does not give rise to any additional machine operations, therefore there is no reason for efficiency concerns. On the other hand, casting may be dangerous when used without the necessary caution. For example, if we actually had matrices on the stack `st` but would cast them to vectors and assign

[1] Here we mean objects of type `stack` as defined in this library, not stacks as computer term. As explained before, some features of the `stack` type compliy with the common definition of stacks in computer terminology, but its use in the library extends beyond that.

components as above, this would result to a disaster (very likely to program crash, and certainly to unexpected behavior). Compiler will not warn about such improper use of stack elements because there is no way to detect it[1]. It is sole responsibility of the user of a stack to treat its elements correctly, and most importantly, the user must know which is the type of the elements n a stack[2].

Access to stack elements must be done cautiously. One must be sure that the sequential number of accessed element does not exceed the number f elements that actually are on the stack, which can be checked through the field `(…)->n.`, e.g.

```
int which;
...
if (which <= st->n && which>0) {
  aux=st->s[which];
  ... /* Do something with the element */
}
```

The function

Stack elements (pointers) can be obtained by the function `stackel`, which checks the validity of element index and returns a NULL pointer when the index is out of bounds. By use of this function, the `if` statement of the above code would look like this:

```
if (aux=stackel(st,which)) {
  ... /* Do something with the element */
}
```

Sometimes one may want to use the function **`nstack`**, which returns a given element of the stack, counted backwards from the end of the stack (argument 1 means the last element, argument 2 one before the last, etc.)[3].

There are several ways to add elements to the stack or remove them from it. We have already mentioned the *push* and *pop* operations, which are the most efficient and add an element at the end or take the last element from the stack (and return its pointer):

```
void pushstack(stack st, void *el);
void *popstack(stack st);
```

[1] There is maybe a theoretic chance to detect such cases of improper use, but only by tracking the code and analyzing what it does. Compilers don't have such abilities.

[2] In object oriented languages such as C++, it is easier to achieve more control on this through use of template classes. In particular, it is easier to distinguish between stack of vectors, stacks of matrices, etc., and thus prevent e.g. vector operations on matrix elements of a stack. Such control mechanisms can be established explicitly in C, but this would either require some additional effort in cumbersome coding or additional overheads in resources because of additional checks). In the case of stacks, library implementation opts for plain solution that requires some caution when using the functionality.

[3] Functions `stackel` and `nstack` could be joined, e.g. by negative arguments imposing counting backwards. They are implemented separately in order to reduce possibility of mistakes.

Function insstack inserts an element at a prescribed position specified by the last argument. If the position exceeds (by more than 1) the number of elements currently on the stack, the intermediate positions are filled with NULL pointers. Otherwise, elements after and including the specified position are shifted by one place towards the end of the stack. If necessary, the element table is re-allocated to fit the new stack size.

Opposite operation is element removal, which is performed by delstack. The last argument specifies which element to take from the stack. The removed element (pointer) is returned by the function, and all elements after the specified place are shifted one position towards the beginning of the stack. If the specified position is out of bounds then a NULL pointer is returned. Otherwise, the number of elements (field (…)->n) is reduced by one.

An existing element of the stack can be replaced by another element by the function setstack. Function returns the replaced element that was on the specified position before. If the position is larger than the number of elements, stack is enlarged, intermediate positions are filled with NULL pointers and NULL is returned. The described functions are declared as follows:

```
void insstack(stack st,void *el,int place);
void *delstack(stack st,int place);
void * setstack(stack st,void *el,int place);
```

9.3.3 Other operations

There are a number of useful operations defined for stacks such as sorting, searching, and collective operations such as printing of all elements. Because stacks can contain any type of (pointer) elements, these operations rely on specification of element level operations. For example, sorting and searching operations requires specification of *element comparison*, which defines the relation "greater, equal or smaller" between two elements:

```
int findstack(stack st,void *ptr,int from, int to,int cmp(void *p1,void
            *p2));
void qsortstack(stack st,int cmp(void *p1,void *p2));
int findsortstack(stack st,void *ptr,int from, int to, int cmp(void *p1,void
            *p2));
```

Function **findstack** searches for the first element of the stack st that is equal (in the sense defined by the function argument cmp) to the specified element ptr, and returns its position or 0 if the appropriate element could not be found on the stack. The search is performed only among elements on positions starting at from and ending at to, where the value 0 of from means the first and vlue 0 of to means the last argument (in the case of excessed bounds, these arguments are internally corrected).

The function findsortstack operates in a similar manner, except that elements on the stack must be sorted in ascending order with respect to cmp for proper operation. Taking advantage of sorting means much faster operation, which is useful especially when number of elements is very

large. If there are more elements that are equal with respect to cmp, an arbitrary position of one of them is returned[1].

Function qsortstack rearranges elements of a stack in such a way that they are ordered in ascending order with respect to the *comparison function* cmp.

The *comparison function* must be provided by the user to define the relation, which may be defined in different ways for some types of data (e.g. for strings, the comparison may be defined in such a way that small and capital letters are distinguished or not). The agreement is that the function must return -1 if the first argument is smaller than the second, 0 if arguments are equal and 1 if the first argument is greater than the second.

Collective operations are performed on all elements of the stack and include e.g. printing of elements. A function that performs an operation on a single element must be specified. For example, the following function prints to a file complete information about a stack together with contained elements:

```
void fprintstack(FILE *fp,stack st,void (*fprintel) (FILE *fp,void *el));
```

Argument fprintel specifies how to print contents of an individual elements. If elements of the stack are vectors (custom type vector) then the function may be used in the following way:

```
FILE *fp;
stack st;
…   /* set elements of st, open the file fp… */
fprintstack(fp, st, (void (*) (FILE *,void *)) fprintvector);
```

Note the type casting applied to the function fprintvector, which is used to print contents of an individual vector, in order to comply with the requested type of fprintstack.

9.3.4 Index tables

Index tables are constructs which are in a way similar to stacks, but they contain elements of the type int (sign integer type) rather than pointer, i.e. they can carry only integer elements. The reason for definition of a special type for dynamic tables of integers while stack carrying integer pointers could do the same job is that use of special arrays of integers is more efficient, and efficient is usually very crucial when e.g. tables of indices are needed. The type implementing dynamic index tables (tables of integers in general) indtab is defined as follows:

```
typedef struct _indtab {
    int n,  /* num. of elements */
        r,  /* allocated space */
```

[1] If we need e.g. the position of the first of such elements (which is not so often), we can perform a simple addition check after the function call.

```
        ex, /* excess allowed in reallocation */
        *t; /* table of elements, counting STARTS WITH 1 */
} *indtab;
```

Similar operations as for stacks are defined for index tables.

9.4 Error reporting

An extensible mechanism for error reporting is implemented in the library. In its basic variant, the mechanism enables printing of error and warning messages to standard output (usually to the terminal) and to a pre-defined error file during program execution. The mechanism can be arbitrarily extended (e.g. by launching messages in customized windows and message boxes, or by searching for help tips related to message contents and showing them) without changing how the error reports are triggered. The mechanism provides a uniform way for reporting errors throughout a program that uses the library.

The following is an example of reporting an error within a particular function:

```
#include <er.h>
...
errfunc1(1, "testfunction");
sprintf(ers(),"This is a test error message. Unexpected value (%g)
            occurred.\n", 3.33);
sprintf(ers(),"Since this is only a test, the value was set just like
            that.\n");
errfunc2();
```

Three utilities declared in `er.h` were used for generating an error report, namely the macros `errfunc1` and `errfunc2` and the function `ers`. The macros initialize and finalize the error report.

The initialization macro `errfunc1` stores the relevant data for the report, i.e. the error code and the name of the function in which the error report was launched (which must be provided by the caller as arguments of the macro) and the file and line number of the point where the macro is called (this information provided through pre-defined compiler directives and makes location of error in the source code easy). It also performs some other operations, e.g. checks the message buffer and eventually empties it.

Custom error report must be written to a string buffer which is returned by the function ers(). This function takes care that the returned pointer is always at the end of eventual previously written contents (such as in the above case where writing on the buffer was performed twice) and that at minimum a given (pre-defined) amount of space is available in the buffer.

For information how to extend the functionality of error reporting system, see the header file `er.h` and the error reporting module `er.c`.

9.5 Other Libraries

There are many developers around who are doing a wonderful job by developing numerical libraries that can provide useful tools to the global research community. Thanks to these people, research and also a large deal of high tech oriented commercial community can benefit from cheap and still reliable enough tools that significantly alleviate their development. Here we would like to point out that such work has an invaluable impact on scientific and technological development and in this way strongly promotes development of novelties that are constantly improving the quality of life in our society.

Some of these libraries are also used by *IOptLib*:

- *Meschach* – a linear algebra library
- *Mersene twister* random generator
- For research purposes that serve development of this library, the *FSQP* algorithm is used. Since this is a commercial algorithm, it is not available to users of *IOptLib*; for users who would like to have a powerful non-linear programming engine integrated with *IOptLib*, purchase of the library can be arranged with its distributor, and interface modules for *IOptLib* can be provided that arrange integration of *FSQP* with the library.
- *SolvOpt* algorithm for nonlinear programming

The author of this library gratefully acknowledges the contribution of developers of these libraries to the existence of *IOptLib* at its present form.

10 FUTURE PLANS

11 TO DO (FOR DEVELOPERS)

11.1 Test utilities

Check the formula (27) on restricted region constraint which works on basis of unit ball constraint defined on transformed co-ordinates, in such a way that transform matrix is randomly generated. Check also weighting functions derived from generic functions of one variable applied to vector norm in affine transformed co-ordinates in the same way!

Implement test functions from global optimization community (described in dev\optglob)!

11.2 Questions to answer

11.3 Implementation plans

11.3.1 Prevention of successive repetition of analyses

In `anfunccountnorepeat` (moduole `optbas.c`), implement possibility of calculating only the missing part of the response, if one part has already been calculated at given parameters.

Implement an analysis function that stores *N* (an arbitrary number of) last results and uses these results when an analysis is called at parameters for which response is stored.

Proposed implementation: This analysis function would take an array of four pointers (say `cd`) as definition data. The meaning of the first three would in principle be the same as for `anfunccountnorepeat`. The fourths pointer would contain address of stack containing the stored points. The function would check on this stack for a point with the same parameters as the function was called with, put address of this point (if found) to cd[3] (otherwise, put the last point to cd[3]) and at the same time shift other points towards the back, and perform `anfunccountnorepeat` with definition data `cd`.

Usage: this would be very useful where one might want to store a number of points, e.g. for later conversion to penalty function in a simplex method.

11.3.2 Successive approximations building blocks

Opombe:
Dodati analysispoint vsaj eno številko za dodatni utežni faktor (tako ali tako bi rabil dve celi in dve realni dodatni števili – zaradi tabel.

- Funkcije za **začetni vzorec** (recimo). Parametri: $\mathbf{x}_0$, h, $\mathbf{h}_{vec}$. Smotrno: Najprej n+1 vzorcev (lin. Namesto kvadrat. Aproks.)
- Funkcije za **aproksimacijo začetnega vzorca** (morda kar povezano s funkcijo za generacijo vzorca). Biti mora 100%.
- Funkcijo za **generacijo stabilizacijskega vzorca glede na trust region** iz trenutne aproksimacijske funkcije (za kvadratno in linearno funkcijo).
- Funkcijo za generacijo eksplicitne aproksimacije iz implicitne (npr. MLS -> quadratic)
- Funkcijo za definicijo analize z dodatnimi **omejitvami za restricted region** (v optbas.c)
- Funkcijo za **kombinacijo analiz** , da lahko dodamo omejitve glede dosega koraka.

Pri omejitvi koraka je potrebno dodati možnost uporabe afine transformacije z omejitvijo z enotsko kroglo.

Transformacije parametrov: smotrno bi bilo implementirati linearne in omejitvene transformacije!!!

11.3.2.1 Priority

- **Sampling function**
    - Make a simple one to act as an extensible sceleton (e.g. random first, then with affine transform added, and maybe much later with solving minimal potential problem (initial guess with transformed unit ball random, opt. problem with transformed co-ordinates for particele potential *and* restricted region constraint))
- **Approximated analysis functions**: linear / quadratic LS / MLS
    - Implement & incorporate weighting functions
- **Affine transform** of the problem
    - Work out *rules for application of transform* (e.g. whendirect/inverse transform is used) for a) sampling, b) restricted step constraint implementation and c) approximation (if applicable)

- **Checking scheme** – very simple test case, like 2D quadratic objective / linea constraints
- **Correct formulas for sampling region, restricted step constraint and weighting functions** (with gradients of functions of transformed co-ordinates)! Correct formulas are in linapprox.doc in the appendix (double checked).

11.3.3 Miscellaneous

References:

[1] I. Grešovnik, *Simplex Algorithms for Nonlinear Constraint Optimization Problems, revision 0*, technical report, 2007.

[2]

[3]

[4]

[5] Igor Grešovnik: Linear approximation with regularization and moving least squares, revision 1, technical report, 2010.

[6]

[7]

[8]

[9]

[10]

[11]

[12] I. Grešovnik. "*Optimisation Shell Inverse*", electronic document at http://www.c3m.si/inverse/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[13] I. Grešovnik. "*Download page for Inverse*", electronic document at http://www.c3m.si/inverse/download/ .

[14] I. Grešovnik. "*Inverse manuals*", electronic document at http://www.c3m.si/inverse/doc/man/ , maintained by the Centre for Computational Continuum Mechanics, Ljubljana.

[15] I. Grešovnik. "*Quick introduction to optimization shell Inverse*", electronic document at http://www.c3m.si/inverse/doc/other/index.html .

[16]

[17]

[18] I. Grešovnik. "*A General Purpose Computational Shell for Solving Inverse and Optimisation Problems - Applications to Metal Forming Processes*", Ph.D. thesis, available at http://www.c3m.si/inverse/doc/phd/index.html .

[19] R. Fletcher. Practical Methods of Optimization (second edition). John Wiley & Sons, New York, 1996.

[20]

[21]

[22]

[23]

[24]

[25]

[26]

[27] P. Michaleris, D. A. Tortorelli, C. A Vidal, *Tangent Operators and Design Sensitivity Formulations for Transient Non-Linear Coupled Problems with Applications to Elastoplasticity,* Int. Jour. For Numerical Methods in Engineering, vol. 37, pp. 2471-2499, John Wiley & Sons, 1994.

[28]

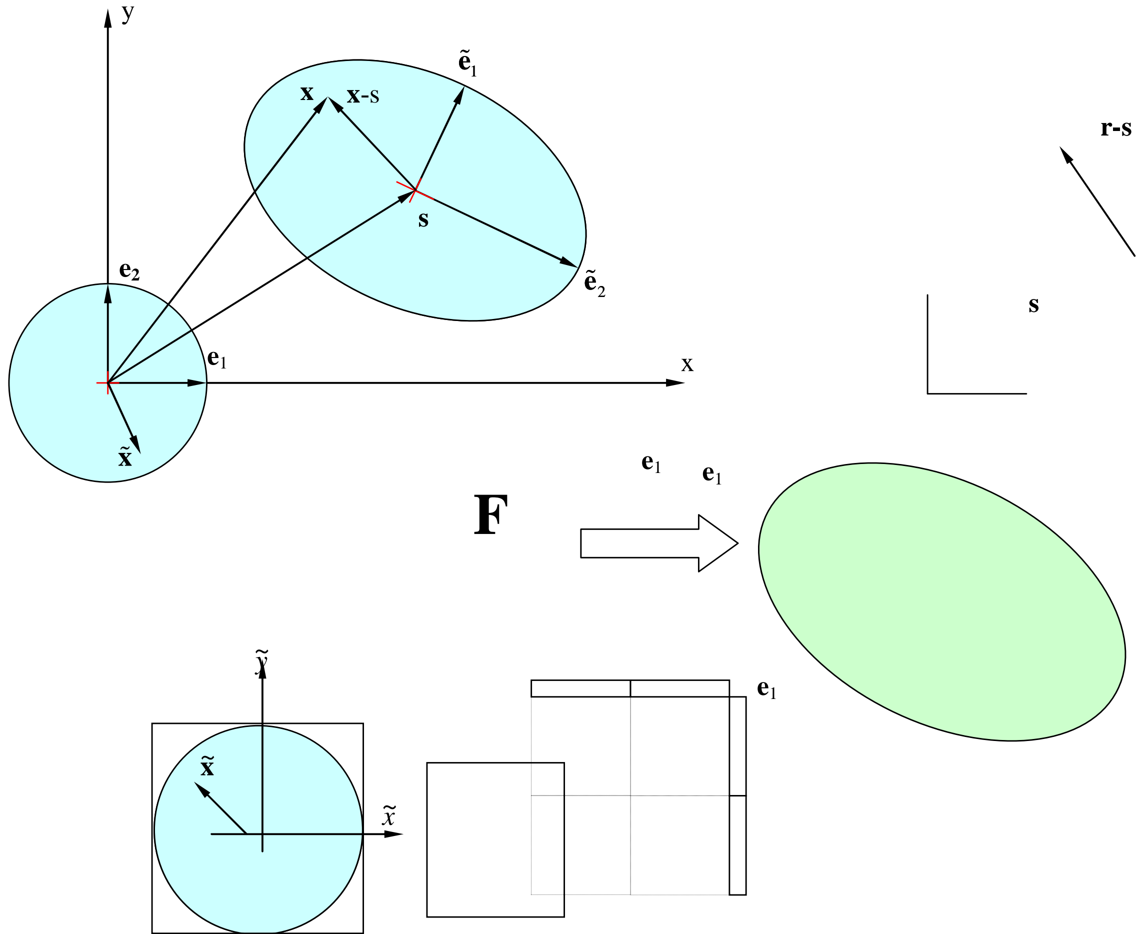
[29]

[30]

[31]

[32]

12 SANDBOX



12.1 Storage of chapters that are in the process of revision

12.1.1 Agreements for use of linear (affine) maps

Remark: this is an old version of **Section 3.1.3**.

In the IOptLib, linear (in fact affine) transforms are used for several purposes which include:

- sampling of response functions in a given domain, which can be obtained by transforming a unit ball
- definition of a restricted region constraint, where the constraint function that ensures that the solution is included in the unit ball is subjected to co-ordinate transform
- definition of weighting functions, which are obtained by co-ordinate transforms of rotationally symmetric functions scaled for a unit ball

The above mentioned functions and procedures are the most easily defined and performed when the unit ball centered in the co-ordinate origin is the domain of interest. We define the transform **F** such that

$$\mathbf{x} = \mathbf{F}\left(\tilde{\mathbf{x}}\right) = \mathbf{A}^{-1}\tilde{\mathbf{x}} + \mathbf{s}, \tag{141}$$

or

$$\tilde{\mathbf{x}} = \mathbf{F}^{-1}\left(\mathbf{x}\right) = \mathbf{A}\left(\mathbf{x} - \mathbf{s}\right). \tag{142}$$

Affine transform **F** transforms a unit ball centered in the co-ordinate origin to an hyper ellipsoidal region with a center of mass **s** (Figure 2). In optimization methods that utilize successive approximations of the response, such domains are conveniently used for sampling of the response and as restricted region on which the approximated problem is solved in the current iteration, therefore also the sampling weights are defined in such a way that influence of samples on the approximation is significant in the domain of the same shape, centered around the corresponding samples.

The (closed) unit ball is defined as

$$U = \left\{ \mathbf{x}; \left\| \mathbf{x} \right\|_2 \leq 1 \right\}. \tag{143}$$

The ellipsoidal domain obtained by transformation of the unit ball by **F** is therefore

$$U_{\mathbf{F}} = \left\{ \mathbf{x}; \left\| \mathbf{F}^{-1}(\mathbf{x}) \right\|_2 \le 1 \right\}. \qquad (144)$$

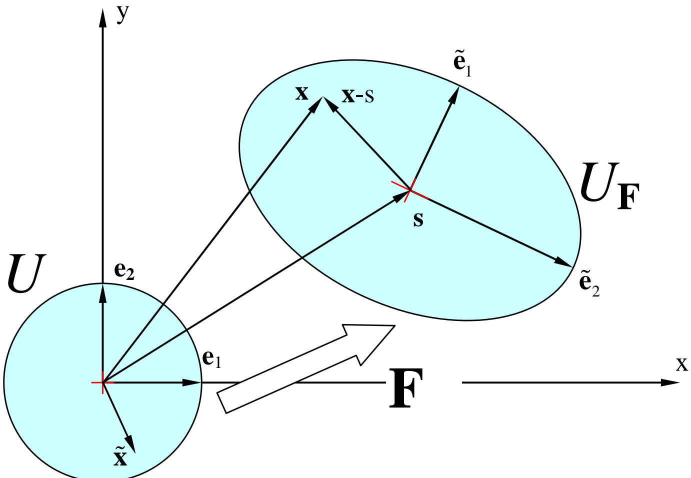


Figure 12: Affine function **F** that maps n unit ball into an ellipsoidal domain centered around **s**.

Sampling is typically done such a way that the specified number m_s of random points with uniform probability density over volume of the unit ball are generated, say $\tilde{\mathbf{x}}_i$. These points are then transformed to $\mathbf{x}_i$ by

$$\mathbf{x}_i = \mathbf{F}(\tilde{\mathbf{x}}_i). \qquad (145)$$

In most cases it is more convenient that the sampling points are uniformly distributed over volume in the unit ball rather than the transformed ellipsoidal domain, which can be very elongated. This is even more obvious when we obtain the samples by solution of the minimal particle potential problem[1]. If the minimal potential problem was used on the ellipsoidal domain that is expressively elongated along one main axis, we would obtain almost uniform distribution along this main axis and a meaningless zigzagging in other directions. When we want to include previously chosen sampling points $\mathbf{y}_k$ in the minimal particle potential problem (in order to avoid oversampling of parts of the domain), these points are first transformed by inverse transforms into

$$\tilde{\mathbf{y}}_k = \mathbf{F}^{-1}(\mathbf{y}_k). \qquad (146)$$

[1] This ensures that the particles are as far away from each other as possible and they are not concentrated in any part of the sampling domains, which can happen by random sampling.

Then the necessary number m of $\tilde{\mathbf{x}}_i$ are obtained from randomly distributed points in the unit ball (say $\tilde{\mathbf{x}}_i^{(0)}$) from solving the minimal particle potential problem involving also the points $\tilde{\mathbf{y}}_k$. Points $\tilde{\mathbf{x}}_i$ are then transformed to $\mathbf{x}_i$ by **F**.

Restricted region constraints are defined by transforming independent variables of constraint function that correspond to limiting the domain to the unit ball. For optimization purposes, the unit ball constraint is conveniently defined as

$$\tilde{\mathbf{x}}^T\tilde{\mathbf{x}} = \|\tilde{\mathbf{x}}\|_2^{\;2} \leq 1 . \tag{147}$$

The corresponding constraint function is

$$c_U(\tilde{\mathbf{x}}) = \|\tilde{\mathbf{x}}\|_2^{\;2} - 1 = \tilde{\mathbf{x}}^T\tilde{\mathbf{x}} - 1 . \tag{148}$$

If we want to limit the domain of optimization to the ellipsoidal region obtained from the unit ball by **F**, we must apply c_U to variables transformed by $\mathbf{F}^{-1}$ because this function transforms the domain of interest to the unit ball (Figure 2). Therefore, the constraint function corresponding to the restricted region constraint is

$$c_r(\mathbf{x}) = c_U\left(\mathbf{F}^{-1}(\mathbf{x})\right). \tag{149}$$

According to (28) and taking into account (36) and (37), gradient of c_r is:

$$\nabla_{\mathbf{x}} c_r(\mathbf{x}) = \mathbf{A}^{-1}\frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2}\; ;\; \tilde{\mathbf{x}} = \mathbf{A}(\mathbf{x} - \mathbf{x}_i) \; . \tag{150}$$

Because sampling is performed inside the ellipsoidal domain obtained by application of **F** to the unit ball, it seems reasonable that contours of **weighting functions** corresponding to individual samples will have similar shapes as this domain, but will be centered around the corresponding samples. Therefore we can use a similar idea for weighting functions as for the restricted region constraint function. We define a template weighting function $\mathbf{w}_U(\tilde{\mathbf{x}})$ with concentric contours, which decays considerably on the distance 1 from the origin. Actual weighting functions are then obtained by applying the template weighting function to co-ordinates transformed by $\mathbf{F}_i^{-1}$, where $\mathbf{F}_i$ is a function that transforms the unit ball to an ellipsoidal domain centered around the corresponding sampling point. For sampling point $\mathbf{x}_i$ the corresponding function is

$$\mathbf{F}_i(\tilde{\mathbf{x}}) = \mathbf{A}^{-1}\tilde{\mathbf{x}} + \mathbf{x}_i . \tag{151}$$

The weighting function corresponding to the sample $\mathbf{x}_i$ is then

$$w_i(\mathbf{x}) = w_U\left(\mathbf{F}_i^{-1}(\mathbf{x})\right). \tag{152}$$

Because the template weighting function has concentric contours, it can be defined by a function of a single variable $w(x)$, i.e.

$$w_U(\tilde{\mathbf{x}}) = w\left(\|\tilde{\mathbf{x}}\|_2\right), \tag{153}$$

The weighting function corresponding to the sampling point $\mathbf{x}_i$ is therefore

$$w_i(\mathbf{x}) = w\left(\left\|\mathbf{F}_i^{-1}(\mathbf{x})\right\|_2\right) = w\left(\mathbf{A}(\mathbf{x}-\mathbf{r}_i)\right). \tag{154}$$

Function w needs to be defined only for non-negative arguments. We usually require that gradient of $\mathbf{w}_U$ is continuous in the co-ordinate origin, which means that w must have a zero derivative in 0. Commonly used forms for w are Gaussian and reciprocal polynomial (Figure 3):

$$\begin{aligned} w_G(r) &= e^{-r^2}, \\ w_p(r) &= \frac{1}{1+|r|^p},\ p = 2, 3, 4, \ldots \end{aligned} \tag{155}$$

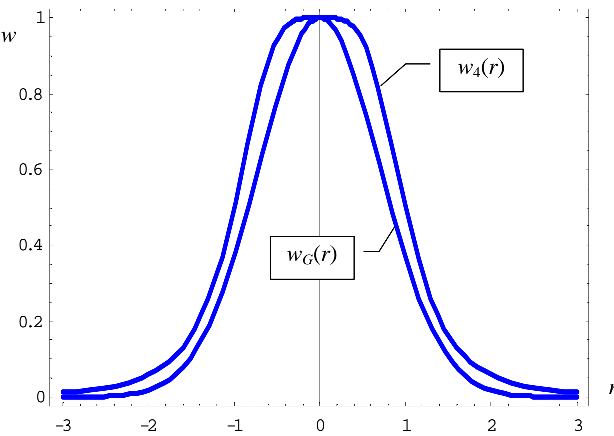


Figure 13: Weighting functions of Gaussian form $w_G(r)$ and reciprocal polynomial form $w_4(r)$.

According to (28) and taking into account (42) gradient of the weighting functions are:

$$\nabla_{\mathbf{x}} w_i(\mathbf{x}) = w'\left(\|\tilde{\mathbf{x}}\|_2\right)\mathbf{A}^{-1}\frac{\tilde{\mathbf{x}}}{\|\tilde{\mathbf{x}}\|_2}\ ;\ \tilde{\mathbf{x}} = \mathbf{A}(\mathbf{x}-\mathbf{x}_i)\ . \tag{156}$$