Zentrum für Technomathematik
Fachbereich 3 – Mathematik und Informatik

Interface Control Document

Dennis Wassel Tim Nikolayzik
Christof Büskens

Report 08–01

Berichte aus der Technomathematik

Report 08–01 März 2008

Interface Control Document

D. Wassel, T. Nikolayzik, C. Büskens

Abstract. Future space missions arising from the demand for branched trajectories for rockets and entry vehicles, weak stability boundary trajectories as well as trajectories for space vehicles with extremely low thrust propulsion are very complex. Discretization of such missions results in the necessity to solve extremely large nonlinear optimization problems with up to more than 300,000 variables and constraints.

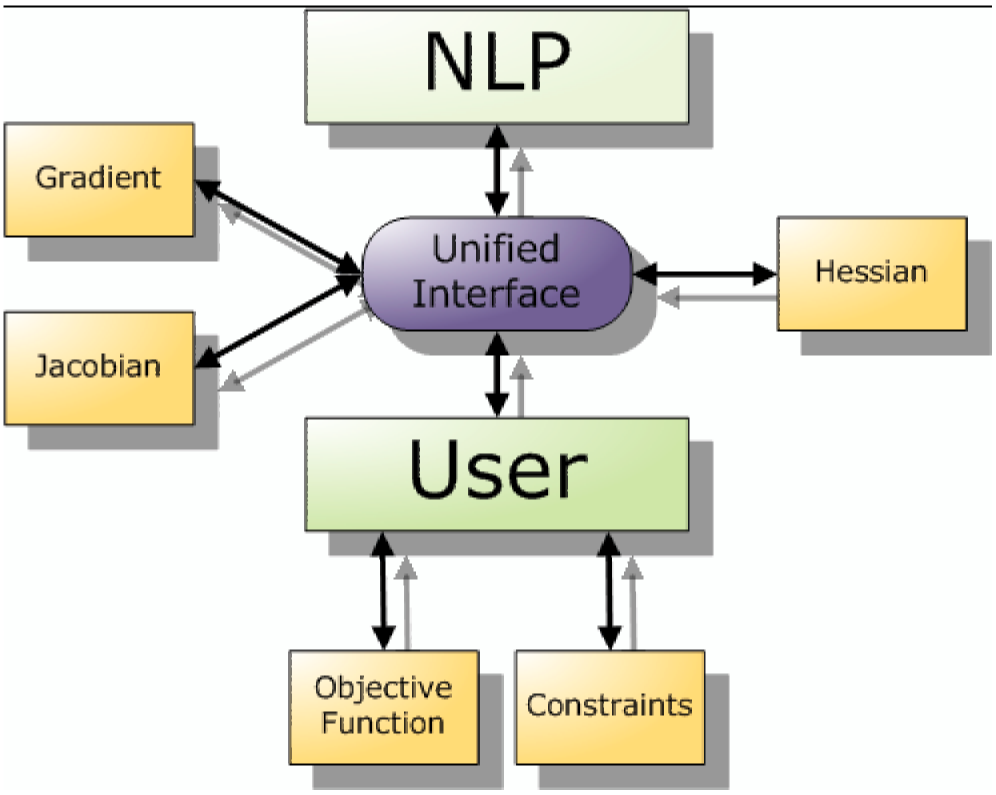
The following paper shows how the interfaces between the different modules of the solver and the interface to the user are defined.

CONTENT

Abbreviations and Terminology

USI	Unified Solver Interface
WMT	Workspace Management Table
CC	Compressed column (matrix storage format)
F	Objective function
G	Constraints
DF	Gradient (vector of first derivatives) of the objective function
DG	Jacobian (matrix of first derivatives) of the constraints
HM	Hessian (matrix of second derivatives) of the Lagrange function
API	Application Programming Interface

Unified Solver Interface



Different solver modules using a Unified Interface

Overview

We propose a Unified Solver Interface (USI) for those solver routines that interface with the user or other modules, to simplify the solver architecture, thereby facilitating long-term maintainability, while increasing internal and external data flow.
The central element of the USI is the definition of four Fortran 95 data structures that encapsulate the user and solver data. These data structures are

- `OptVar` – for optimisation variables, multipliers, constraint values, …
- `Workspace` – for internally needed workspace, counters, …
- `Params` – for all solver parameters (essentially read-only for the solver),
- `Control` – for steering the reverse-communication control flow.

We will use the following naming scheme for instances of the solver data structures:

```
OptVar          opt
Workspace       work
Params          par
Control         cnt
```

Using these data structures enables the developers to harmonise the plethora of different interfaces usually present in big software projects to the single interface

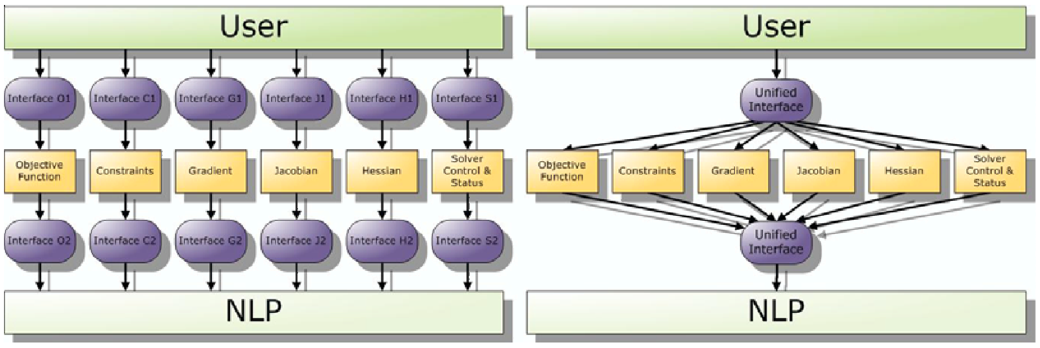
```
Subroutine(opt, work, par, cnt)
```

Rationale

The Unified Solver Interface has a multitude of advantages:

- Uniformity:
  Programmers can rid themselves of the usual hassle of having to keep track of, and checking against, various interfaces.
- Cleanness:
  Preventing bloated and confounding interfaces with lots of arguments to them (this is commonly found in NLP solvers) reduces errors resulting from omissions or mix-ups of arguments. Also, replacing usual conventions such as using parameter arrays by a parameter data structure greatly clarifies usage of the solver (compare `IPARAM(3)` to `Param%MaxIter`, for example).
- Encapsulation:
  Data structures offer a degree of encapsulation that is often useful, e.g. in initialisation before the actual start of the optimisation run, or in internal functionality that is of no concern to the average user (but may readily be inspected and used by advanced users).
- Usability:
  The abovementioned benefits add up to a significant increase in usability by reducing frequent causes of user errors, minimising the amount of detail the user initially has to take care of (which, we believe, is important for encouraging inexperienced users) and keeping the user's own interfacing code concise.
- Maintainability:
  By sharing an interface between all routines, it is exceptionally easy to expand or exchange solver routines, without breaking the API – with routine-specific interfaces, this usually implies the changing every single call in all program modules.
- Modularity:
  Adding new modules, which need additional data and parameters, is simple, since the necessary changes are confined to a single source code file, which defines the data structures and their initialisation and clean-up routines.
- Visibility:
  With the data structures being visible in all routines, every piece of solver or

user data can be accessed everywhere, no matter how “deep” the routine is situated in the calling hierarchy. This prevents the bloating of interfaces towards the top of the calling hierarchy, that is usually found in many pieces of software.



Comparison between traditional and Unified interfaces

Important Fortran modules

The solver WORHP is split into several Fortran `MODULE`s. Here is a list of them, and (some of) the routines and constants they define

`MODULE/Type`	Defines …	Purpose
`std`		Global datatype constants and routines
`Integer const`	`lgc`	Kind constant for logical
`Integer const`	`int`	Kind constant for integer
`Integer const`	`single`	Kind constant for single precision real
`Integer const`	`double`	Kind constant for double precision real
`Logical const`	`true`	.true. value for chosen logical kind
`Logical const`	`false`	.false. value for chosen logical kind
`worhp_data`		Defines the USI data structures
`Derived Type`	`OptVar`	Optimisation variables
`Derived Type`	`Workspace`	Workspace
`Derived Type`	`Params`	Parameters
`Derived Type`	`Control`	Program flow control
`Subroutine`	`StatusMsg`	Print meaningful termination message
`Integer const`	`No_Stage`	(+ 9 more) SQP stage constants
`worhp_core`		Computational routines
`worhp_aux`		Non-computational routines
`Subroutine`	`Init`	Data structure initialisation subroutine
`Subroutine`	`Free`	Data structure deallocation subroutine
`Subroutine`	`InitIWSlice`	Allocate a workspace slice
`Subroutine`	`FreeIWSlice`	Deallocate a workspace slice

worhp		WORHP SQP solver
Subroutine	worhp_full	Full-Feature Interface
Subroutine	worhp_basic	Basic-Feature Interface

worhp.macros.h		Workspace access macros
CPP macro	IWMT_SIZE	Get size of an allocated slice
CPP macro	IWS_SLICE	Access whole workspace slice
CPP macro	IWS_ELEM_1	Get single element of a slice, 1-indexing
CPP macro	IWS_RANGE_1	Get a subslice of a slice, 1-indexing
CPP macro	IWS_INDEX_1	Get physical index of a slice, 1-indexing

Since the central module `worhp` includes all other modules, users will only need to

```
USE worhp
```

and

```
#include "worhp.macros.h"
```

in their Fortran 95 code to get access to all relevant routines. Note that a C preprocessor is needed for translating the workspace access macros into Fortran code; many modern Fortran compilers invoke the preprocessor automatically, if appropriate source file suffixes are used (usually `source.F90`).

Data structures

We provide a global definition of the solver data structures in tabular form here, since they are valid for all USI routines. The table lists those structure members relevant for interfaces, their types and their main purpose.

Future development of the solver will introduce (possibly many) additional data structure members, but no additional data structures. Two examples: The IPFilter parameters will be added to `Params` to enable central access to all solver parameters, and the `OptVar` and `Workspace` type will be updated to enable the user to *optionally* specify additional information about linear and quadratic parts of the objective function, and about linear constraints, to improve the solver performance.

Data type terminology

The following terminology will be used in data type description:

Term	Description
real	Fortran `real(double)` type, i.e. double precision
integer	Fortran `integer` type
index_i	Fortran `integer` type, used as index for the IWMT
index_r	Fortran `integer` type, used as index for the RWMT

counter	Fortran integer type, used as counter
logical	Fortran logical(lgc) type
character	Fortran character type
T(dim)	1D-array of dimension dim of type T
T(dim1,dim2)	2D-array of dimensions (dim1 x dim2) of type T
alloc	Allocatable component, to be allocated by an initialisation routine

OptVar data structure

The OptVar data structure is most relevant to the user, since it contains the main problem data: dimensions, objective function value, constraints and multipliers.

Type OptVar		
N	integer	Number of variables
M	integer	Number of constraints
F	real	Current value of objective function
X	real(N), alloc	Optimisation variables
lambda	real(N), alloc	Box constraint Lagrange multipliers
G	real(M), alloc	Current value of constraints
mu	real(M), alloc	Constraint Lagrange multipliers
L	real(N+M), alloc	Lower bound on X and G
U	real(N+M), alloc	Upper bound on X and G
initialised	logical	Data structure initialisation flag

Workspace data structure

The Workspace data structures contains data relevant to the internal workings of the solver; almost all of them are of no relevance to the average user.

Type Workspace		
DF	CCMIndex(Vector)	Gradient of the objective function
DG	CCMIndex(ComCol)	Jacobian of the constraints
HM	CCMIndex(LowTri)	Hessian of the Lagrange function
ID	CCMIndex(Diagonal)	Identity matrix
Q	CCMIndex	Current QP matrix (HM or ID)
A	CCMIndex(ComCol)	Equality constraints matrix for QP
C	CCMIndex(ComCol)	Inequality constraint matrix for QP
ArmijoAlpha	real	Last Armijo stepsize
dTHd	real	Value of $d^T Hd$
NormDX	real	2-norm of last search direction
NormCon	real	max-norm of constraint violation
NormKKT	real	KKT norm
MeritOldValue	real	Merit function value at $\alpha = 0$
MeritNewValue	real	Merit function value at α
MeritGradient	real	Derivative of Merit function at $\alpha = 0$
rws	real (:), alloc	Real workspace

`iws`	`integer(:), alloc`	Integer workspace
`nrws`	`integer`	Dimension of real workspace
`niws`	`integer`	Dimension of integer workspace
`RWMT`	`integer(100,6)`	Real workspace management table
`IWMT`	`integer(100,6)`	Integer workspace management table
`RWMTnames`	`character*20(100)`	Real workspace slice names
`IWMTnames`	`character*20(100)`	Integer workspace slice names
`newlambda`	`index_r`	New box constraint multipliers
`newmu`	`index_r`	New constraint multipliers
`oldx`	`index_r`	Copy of `X`, used by Armijo
`oldlambda`	`index_r`	Copy of `lambda`, used by Armijo
`oldmu`	`index_r`	Copy of `mu`, used by Armijo
`penalty`	`index_r`	Penalty parameters
`qpeqrhs`	`index_r`	RHS of the QP equality constraints
`qpeqlm`	`index_r`	Multipliers of QP equality constraints
`qpierhs`	`index_r`	RHS of the QP inequality constraints
`qpielm`	`index_r`	Multipliers of QP inequality constraints
`qpdx`	`index_r`	Search direction from QP
`qpiparam`	`index_r`	IPARAM array for QP
`qprparam`	`index_r`	PARAM array for QP
`MajorIter`	`counter`	Main iteration counter
`MinorIter`	`counter`	QP iteration counter
`calls`	`counter`	Reverse Communication call counter
`nEQ`	`integer`	Total numer of equality constraints
`nEQbox`	`integer`	Number of equality constraints on `X`
`nEQgen`	`integer`	Number of equality constraints on `G`
`nIE`	`integer`	Total number of inequality constraints
`nIEbox`	`integer`	Number of inequality constraints on `X`
`nIEgen`	`integer`	Number of inequality constraints on `G`
`nQP`	`integer`	Number of variables for QP
`RelaxCon`	`logical`	“Elastic constraints” in QP
`UseID`	`logical`	Use identity matrix for QP
`KKTok`	`logical`	KKT conditions satisfied
`initialised`	`logical`	Data structure initialisation flag

A central element for workspace partitioning is the concept of the workspace management table: it contains and manages indices of initialised workspace partitions, called “slices”. Allocation and deallocation of workspace slices is done by specialised functions, and access to the slices is provided efficiently by preprocessor macros, so that no direct manipulation of, or access to the management tables is ever necessary (see Workspace management and Internal interfaces for details). For storing the various matrices in compressed-column format, the `CCMIndex` structure is used. It indexes the necessary workspace slices for storing a CC matrix.

`Type CCMIndex`		
`kind`	`integer`	Kind constant
`nnz`	`integer`	Number of nonzero entries

`nrow`	`integer`	Number of rows
`ncol`	`integer`	Number of columns
`val`	`index_r`	CC val array
`row`	`index_i`	CC row array
`col`	`index_i`	CC col array
these values are used to reserve space for dynamic resizing:		
`nnzMax`	`integer`	Max number of nonzero entries
`nrowMax`	`integer`	Max number of rows
`ncolMax`	`integer`	Max number of columns
`nnzMin`	`integer`	Min number of nonzero entries
`nrowMin`	`integer`	Min number of rows
`ncolMin`	`integer`	Min number of columns
`nExt`	`integer`	Resize elements counter against initial dimension

The kind identifier is relevant to routines that take a CC matrix as argument, since some actions operate differently (or not at all) on certain kinds of CC matrices, or can be implemented more efficiently for certain kinds, e.g. matrix-vector multiplication with a diagonal matrix.

`CCMIndex kind constants`	
`ComCol`	General CC matrix without special structure
`LowTri`	Symmetric lower triangle CC matrix, including all diagonal entries
`Diagonal`	Diagonal CC matrix
`Identity`	Identity matrix
`Vector`	Single-column or single-row CC vector
`Struct`	Structure only CC matrix (no value array)

Params data structure

The `Params` data structure encapsulates all solver parameters that are not hard-coded. This data structure will be treated as read-only by the solver routines, since parameters should not usually be changed by the solver; any quantity that needs to be manipulated belongs into the `Workspace` structure instead.

`Type Params`		
`TolOpti`	`real`	Optimality tolerance
`TolFeas`	`real`	Feasibility tolerance
`Infty`	`real`	Value to be treated as infinity
`eps`	`real`	Machine
`ArmijoBeta`	`real`	beta for Armijo rule
`ArmijoSigma`	`real`	sigma for Armijo rule
`ArmijoMinAlpha`	`real`	Minimum Armijo stepsize
`RelaxRho`	`real`	Constraint relaxation penalty increase
`RelaxMaxDelta`	`real`	Max constraint relaxation variable
`RelaxMaxPen`	`real`	Max constraint relaxation penalty
`MaxMajorIter`	`real`	Maximum major iterations
`MaxMinorIter`	`real`	Maximum QP iterations

MaxCalls	real	Maximum number of calls
QPlsTol	real	QP linear solver tolerance
QPipComTol	real	QP IP complementarity tolerance
QPipResTol	real	QP IP residuals tolerance
QPipBarrier	real	QP IP barrier parameter
QPnsnKKT	real	QP NSN KKT tolerance
QPnsnBeta	real	Beta for NSN Armijo stepsize
QPnsnSigma	real	Sigma for NSN Armijo stepsize
QPminAlpha	real	Min alpaha for NSN Armijo stepsize
QPfracBound	real	QP fraction-to-the-boundary
MeritFunction	integer	Selects the merit function
QPmethod	integer	QP solution method (10,20,21,30,31)
QPprint	integer	QP print level
QPitMaxIter	integer	QP iterative solver maxiter
QPitRefMaxIter	integer	QP iterative refinement maxiter
QPits	integer	QP iterative solver selection (3-7)
UserDF	logical	User-supplied obj. function gradient
UserDG	logical	User-supplied constraint Jacobian
UserHM	logical	User-supplied Lagrange Hessian
QPparamCheck	logical	QP parameter checking
QPstrict	logical	QP strict criteria
QPscale	logical	QP automatic scaling
QPgradStep	logical	QP gradient steps
initialised	logical	Data structure initialisation flag

Control data structure

The `Control` data structure holds all necessary facilities to influence the reverse communication program control flow, and to exchange information between the user and the solver.

Type Control		
status	integer	WORHP status flag
LastStage	integer	The last RC stage that was completed
NextStage	integer	The next RC stage to execute
UserAction	logical(12)	User action flags

When using the Full-Feature Interface, the status flag and the user action flags are of interest to the user. The status flag controls the continuation or termination of the reverse communication while-loop in the Full-Feature Interface, and informs the user of the reason for termination. The user action flags are to be polled by the user after every reverse communication step, whether some action needs to be taken, e.g. evaluating the objective function at the current point.

External interfaces

The external interfaces will be described here in a straightforward manner. Arguments are categorised as `type [in]`, `[inout]` or `[out]`. Optional arguments (if any) are enclosed in square brackets in the interface description. If a routine has mutually exclusive optional arguments, or requires at least one of the optional arguments, this is defined in the text body of the documentation.
Some routines are generic in that sense that they may operate on real and integer data in the same way (sorting would be an example of this); those routines will be documented together.

Example:

```
    IHello(message, n, [count])
    RHello(message, x, [count])
```

with arguments

`message`	`character [in]`	Message to display
`n`	`integer [in]`	Number to append to the message
`x`	`real [in]`	Number to append to the message
`count`	`integer [in]`	Repeat count

describes two routines named `IHello` and `RHello`, with a string argument `message`, a numeric argument `n` (or `x`), and an optional argument `count`.

Solver initialisation

To take the burden of initialising the solver data from the user, the subroutine `Init` is supplied for this purpose. Before calling `Init`, the user has to set the following variables to appropriate values:

`opt%N`	Number of variables, > 0
`opt%M`	Number of constraints, ≥ 0
`work%DF%nnz`	Number of nonzero entries of DF, > 0
`work%DG%nnz`	Number of nonzero entries of DG, ≥ 0
`work%HM%nnz`	Number of nonzero entries of HM, ≥ N (HM structure has to include at least the diagonal entries)

The subroutine `Init` adheres to the USI, hence its interface is

```
    Init(opt, work, par, cnt)
```

After `Init` returns, all data structures are initialised, and the solver parameters are set to standard values. In particular, workspace slices for the matrices DF, DG and HM are allocated with the given sizes. Note that the CC matrix structures and values are not initialised by the `Init` subroutine, since there are no sensible “standard values” for sparse matrix structures – this needs to be performed by the user. The future solver development will partly address this issue.

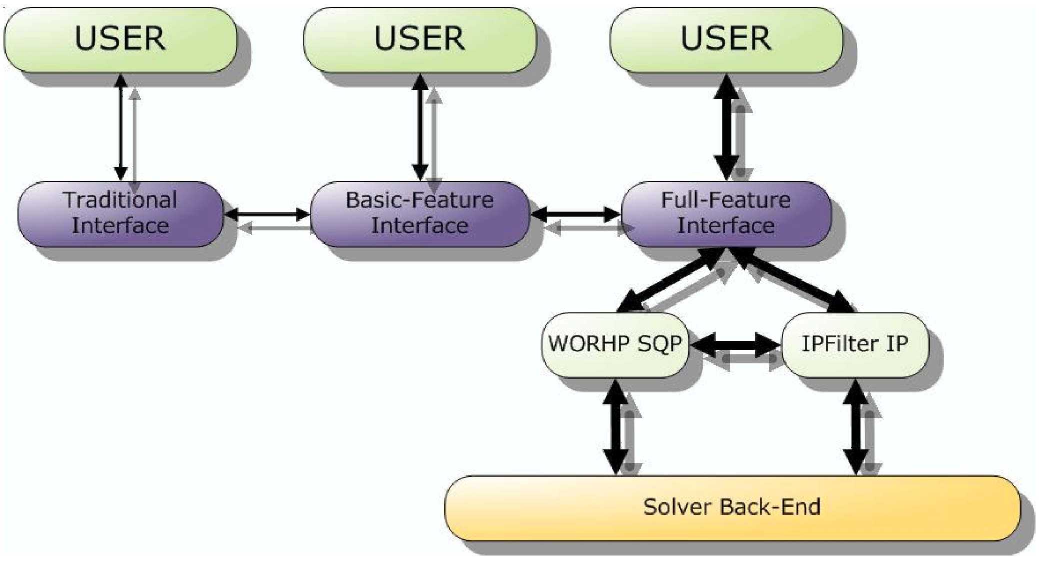
After an optimisation run, and after all required data has been extracted, the user can explicitly deallocate all data structures by calling the `Free` subroutine with the USI

```
    Free(opt, work, par, cnt).
```

The result of any access to the data structures after calling `Free` should be considered as undefined, and will fail completely for all allocatable members.
The deallocated instances may be re-used after another call to `Init` to initialise them to standard values and start another solver run; this will overwrite all previous data, however.

Solver

Solver Interfaces



The *Full-Feature Interface* will serve as the central solver interface, which satisfies all requirements. It adheres to the USI and allows arbitrary user access to data and control flow by making heavy use of reverse communication. Despite its power and transparency, it is simpler and cleaner than any conventional NLP interface.

The solver is called as

```
    worhp_full(opt, work, par, cnt)
```

with the data structures initialised by `Init` and the matrix structures set to appropriate values by the user.

The *Basic-Feature Interface* serves as an equally simple interface, without reverse communication facilities. Like the Full-Feature Interface, It adheres to the USI and thus shares the same solver interface. Since it does not use reverse communication, but the traditional calling mode instead, it requires the user to implement two subroutines `F` and `G`, both of which adhere to the USI as well:

```
    F(opt, work, par, cnt)
    G(opt, work, par, cnt)
```

After the data structures are initialised by `Init` and the matrix structures set to appropriate values by the user, the solver is called once with the Basic-Feature Interface

```
    worhp_basic(opt, work, par, cnt)
```

to start an “unattended” optimisation run.

The solver is furthermore required to provide a *Traditional interface* according to the following excerpt from the requirements document:

```
Solver(start, m, n, ne, nName, nnCon, nnObj, nnJac, 0, 0.0,
     "Method",
     Constraints'Access,
     Objective'Access,
     Double_Array (a),
     Integer_Array (ha),
     Integer_Array (ka),
     Double_Array (bl),
     Double_Array (bu),
     Names,
     Integer_Array (hs),
     Double_Array (x),
     Double_Array (pi),
     Double_Array (rc),
     Inform,
     mincw, miniw, minrw,
     nS, nInf, sInf, Obj, cw, iwu, rwu, cw, iw, iwlen, rw,
     rwlen);
```

with the following parameter definition:

- start: warm / cold start
- m, n, ne, nName, nnCon, nnObj, nnJac: lengths of all arrays
- name of the problem / optimiser
- function pointer for funcon / funobj
- Acol, indA, locA
  define the nonzero elements of the constraint matrix A (4.2), including the Jacobian matrix associated with nonlinear constraints. The nonzeros are stored column-wise. A pair of values Acol(k), indA(k) contains a matrix element and its corresponding row index. The array locA(*) is a set of pointers to the beginning of each column of A within Acol(*) and indA(*). Thus for j = 1 : n, the entries of column j are held in Acol(k : l) and their corresponding row indices are in indA(k : l), where k = locA(j) and l = locA(j + 1) – 1
- Bl, bu: LB&UB of parameters
- Bl, bu: LB&UB of constraints (slacks)
- Names for the parameters or indices
- Hs: Integer Array => 0, unused
- X: start/end values for constraints (slacks)
- start/end values corresponding to cold/warm start

- Pi, costates unused => 0
- Ns: warm start information
- Cu, iu, ru, Cw, iw, rw: workspace related

This interface will be implemented as wrapper of the Basic-Feature Interface.

Using Reverse Communication

In the Full-Feature Interface many flow-control-relevant internal loops are carried out as external reverse communication loops to increase flexibility, transparency and debugging capabilities; the usefulness of this architectural feature has already been demonstrated by the Fortran 95 version of WORHP, which uses no internal flow control loops for the SQP part at all. This means that the user has to construct a loop around the solver to enable it to iterate. Inbetween iterations, the user polls the `Control` data structure to find out which activities have to be performed.

The reverse communication loop is controlled by the status flag `cnt%status` in connection with the two integer constants `terminateSuccess` and `terminateError`. It can be constructed as

```
DO WHILE (cnt%status > terminateError .AND.
          cnt%status < terminateSuccess)
   CALL worhp(opt, work, par, cnt)
END DO
```

or equivalently.
The necessary user actions are encoded in the `cnt%UserAction` logical array, whose indices determine the requested action. These indices are

`iterOutput`	One major iteration is complete, show iteration output
`evalF`	Evaluate F for the current value of `opt%X` and save result to `opt%F`
`evalG`	Evaluate G for the current value of `opt%X` and save result to `opt%G`
`evalDF`	Evaluate DF for the current value of `opt%X` and save result to `RWS_SLICE(work%DF%val)`
`evalDG`	Evaluate DG for the current value of `opt%X` and save result to `RWS_SLICE(work%DG%val)`
`evalHM`	Evaluate HM for the current value of `opt%X` and save result to `RWS_SLICE(work%HM%val)`

If for any of the above indices –say evalF– the expression `cnt%UserAction(evalF)` evaluates to `true`, the user is requested to carry out the corresponding action, in this case to evaluate the objective function for the current value of `opt%X` and save the resulting value to `opt%F`. To prevent superfluous evaluations of functions or derivatives, it is imperative that the user reset any flag after having carried out the requested action, in this case by setting `cnt%UserAction(evalF) = false` (sic!).
This functionality can be constructed as blocks of the following form inside the reverse communication loop:

```
    IF (cnt%UserAction(evalF)) THEN
       ! user code for evaluating F at opt%X
       cnt%UserAction(evalF) = false
    END IF
```

Depending on the amount of information the user can provide, at least the two evaluation blocks for `evalF` and `evalG` have to be provided. Where possible, the user can (and should) provide derivative information and flag this by setting the corresponding parameter; if the user provides DF information, for example, this is flagged by setting `par%UserDF = true`; likewise for DG and HM.

Termination status

After termination of either the Full-Feature Interface or the Basic-Feature Interface, the status flag `cnt%status` holds information about the reason. The status flag may hold the following values

Successful termination	
`OptimalSolution`	Optimal solution found.
`SearchDirectionZero`	Optimal solution found, maybe not to the requested accuracy.
`SearchDirectionSmall`	Optimal solution found, maybe not to the requested accuracy.
`StationaryPointFound`	Stationary point of the Merit Function found.
Unsuccesful termination	
`InitError`	Data structures not initialised properly.
`DataError`	Error in the supplied solver data.
`MaxMajorIter`	Maximum number of major iterations reached.
`MaxCalls`	[RC only] Maximum number of calls reached.
`MinimumStepsize`	Minimum stepsize reached in Armijo rule.
`ProblemInfeasible`	The QP is infeasible.
`QPerror`	The QP could not be solved.

To directly translate the status flag a success/error message, the subroutine

```
    StatusMsg(opt, work, par, cnt)
```

may be called. It checks the `cnt%status` flag and generates meaningful console output.

Workspace access

Automatic workspace partitioning is a major feature of the proposed solver architecture. To hide an unnecessary level of implementation detail from users and developers, functions and preprocessor macros are used to allocate, access and deallocate workspace partitions, called "slices".
We will describe only the slice access macros here, since they are necessary to know for the user, while the allocation and deallocation routines are described in the Internal interfaces section; these are mainly to be used by the developers. Since the macros translate into Fortran 95 array syntax, they cause no performance penalty of their own.

Note: While some of these macros will very probably fail "loudly" when used with an unallocated `index`, some may do so unnoticed, or cause all kinds of errors in other places. It is in the responsibility of the user to ensure that no unallocated `index` is used to access the workspace. Also, there is currently no mechanism to distinguish `IWMT` and `RWMT` indices; it is in the user's responsibility not to mix the two sets of indices, for instance by adhering to appropriate naming conventions.
The solver uses no such naming conventions; all slice indices have meaningful names. Their datatype can either be deduced from their purpose, or looked up in the documentation.

There are two identical sets of macros for integer and real workspace access. Some of the slice access macros are split into two groups: 1-indexing and 0-indexing macros. The 1-indexing macros operate on Fortran-style indices running from 1…n, while 0-indexing macros operate on C-style indices running from 0…n-1. These two groups are distinguished by a trailing `_1` or `_0`. We will document the 1-indexing macros only, since for every `_1` macro, there exists a `_0` version.

To inquire the allocated size of a slice use

```
IWMT_SIZE(index)
RWMT_SIZE(index)
```

To access a whole allocated slice for reading, writing or passing as a function argument, (this type of access should be preferred over the following ones), use

```
IWS_SLICE(index)
RWS_SLICE(index)
```

To access the sub-slice `(i:j)` of an allocated slice, use

```
IWS_RANGE_1(index,i,j)
RWS_RANGE_1(index,i,j)
```

To access an allocated slice element-wise (this is the slowest type of access when used inside a loop), use

```
IWS_ELEM_1(index,i)
RWS_ELEM_1(index,i)
```

To get the direct access through the "physical" `IWS` or `RWS` index of a slice, use

```
    IWS_IDX_1(index)
    RWS_IDX_1(index)
```

[This is to be understood in the following way: If `k = IWS_IDX_1(index)`, then the `j`-th element in 1-indexing of slice `index` is `work%iws(k+j)`]
For the sake of reducing causes of errors, this access mode should only be used if none of the other access modes are feasible or efficient.

Internal interfaces

Reverse Communication Control

Reverse Communication is an integral part of the architecture of WORHP and is carried out internally regardless of the interface used. Each major iteration is internally divided into “stages” which are executed in a very flexible order; each stage, upon completion, decides which stage is to be executed next.

The SQP currently consists of nine stages (plus one no-stage as initial value). These stages are encoded as the following integer constants:

`No_Stage`	No stage, only used as default value in initialisation.
`Init_SQP`	SQP Initialisation stage, only before first major iteration.
`Pre_KKT`	First major iteration stage, sets the user actions for KKT check.
`Check_KKT`	Checks KKT conditions and terminates, if satisfied.
`Create_QP`	Creates the QP and resets some quantities to standard values.
`Solve_QP`	Solves the QP and triggers recovery actions in case of failure.
`Find_Stepsize`	Tries decreasing stepsizes and evaluates the merit function.
`Update_Point`	Updates the current values of opt. variables and multipliers.
`Finalise`	Finalises a major iteration after a stepsize has been found.
`SLP_step`	Prepares to solve the QP with the identity matrix.

The previous and next stage are held in `cnt%LastStage` and `cnt%NextStage`. To simplify stage handling, the subroutine `SetNextStage` sets the next stage to be executed:

```
SetNextStage(cnt, NextStage)
```

with arguments

`cnt`	`Control`	`[inout]`	Control type instance.
`NextStage`	`Integer`	`[in]`	Next stage.

To influence the user actions, a similar routine is supplied. The subroutine `SetUserAction` sets or removes a user action, checking for valid input values.

```
SetUserAction(cnt, [AddAction, RemAction])
```

with arguments

`cnt`	`Control`	`[inout]`	Control type instance
`AddAction`	`Integer`	`[in]`	Add this user action request.
`RemAction`	`Integer`	`[in]`	Remove this user action request.

The subroutine accepts all combinations of its 1 to 3 arguments (in case of 1 argument only, it does nothing, of course). If both optional arguments are present, first the `AddAction` is added, and afterwards the `RemAction` is removed.

Workspace management

To prevent faulty workspace partitioning, which causes errors that can be extremely hard to spot, WORHP supplies automatic workspace partitioning routines. The access to already allocated slices is documented in External interfaces. Every routine that needs a workspace partition has to do so by using the routines described here (or use private allocatable workspace)

To allocate a workspace slice, the subroutines `InitIWSlice` and `InitRWSlice` have to be used. They have tailored non-USI interfaces

```
InitIWSlice(status, work, size, WMTentry, [slice, name])
InitRWSlice(status, work, size, WMTentry, [slice, name])
```

with arguments

`status`	`integer [out]`	Status flag (≠ 0 means error)
`work`	`Workspace [inout]`	Workspace type instance
`size`	`integer [in]`	Size of the slice to allocate
`WMTentry`	`index_i [out]`	`index` of the allocated IWS slice
`WMTentry`	`index_r [out]`	`index` of the allocated RWS slice
`slice`	`pointer [out]`	Pointer to the slice (integer or real)
`name`	`character [in]`	Name of the slice

To ensure that the slice has been allocated successfully and is ready to be used, the status flag must always be checked. Errors may occur when

- The maximum number of slices for the corresponding WMT is reached,
- The requested size of the slice exceeds the overall workspace size,
- There is not enough workspace left to accommodate the requested slice,
- There is no contiguous slice of the requested size available.

The last issue will be fixed by improving the allocation and management routines to prevent workspace fragmentation.

To deallocate a workspace slice, the subroutines `FreeIWSlice` and `FreeRWSlice` have to be used. They have tailored non-USI interfaces

```
FreeIWSlice(status, work, [WMTentry, name])
FreeRWSlice(status, work, [WMTentry, name])
```

with arguments

`status`	`integer [out]`	Status flag (≠ 0 means error)
`work`	`Workspace [inout]`	Workspace type instance
`WMTentry`	`index_i [inout]`	`index` of the IWS slice to free
`WMTentry`	`index_r [inout]`	`index` of the RWS slice to free
`name`	`character [in]`	Name of the slice

Exactly one of the two optional arguments has to be specified. Using `WMTentry` to index a slice is the faster of both possible methods.