ALCON-S
Version 0.8
User's Guide

Clemens Klein-Robbenhaar
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10
D-10711 Berlin-Wilmersdorf Germany
e-mail: klein-robbenhaar@sc.zib-berlin.de

June 22, 1995

1 Introduction: Purpose of the Algorithm

The software package Alcon-S is intended to solve *nonlinear systems of real equations depending on one real parameter*

$$\begin{aligned} f\colon \mathbb{R}^m \times \mathbb{R} &\rightarrow \mathbb{R}^m \\ f(x,\lambda) &= 0 \end{aligned}$$

The dimension m can be large, that's up to a few thousands. In this case it is assumed that the jacobian matrix

$$f'(x,\lambda) = \left(\frac{\partial f}{\partial x} \, \frac{\partial f}{\partial \lambda} \right)$$

is *sparse*, i.e. it has only very few nonzero entries. Alcon-S actually is limited to the finite dimensional case; it assumes, that m is fixed all over the computation. Therefore it is not possible to compute solutions of infinite dimensional systems with adaptively changing discretizations. Such problems must be computed with a fixed discretization which is adapted to the problem with a priori information; otherwise the results may be not very satisfying.

Method of the Computation Alcon-S computes smooth solution pathes, which do not necessarily have a parametrization over λ. To do this, it uses the well know continuation algorithm, also called pathfollowing or predictor corrector scheme, with a sophisticated steplength control. The program applies a tangential predictor and a Gauß Newton method as corrector. This modification of the usual Newton method treats the parameter like an ordinary variable and solves the arising underdetermined linear systems by the Moore Penrose pseudo inverse. The pseudo inverse is computed by a factorization based on QR or LU factorization for small systems and on a C adaption of the Harwell MA28 sparse solver package in the sparse case. Because for large systems often the parameter plays an exceptional role over the variables which is reflected in the structure of the variables part of the jacobian $\partial f/\partial x$, a block solver scheme which only needs a decomposition of the variables part is also included.

Alcon-S can compute simple singularities along the path. Turning points can be computed by request, although they do not need to be treated with special care by the algorithm. The algorithm to compute turning points is based on an intervall halving algorithm, where cubic hermite interpolation between two points give the next estimate to the turning point.

Simple bifurcations (where shortly speaking only two pathes cross) can also be computed and for the emanating branches the pathfollowing algorithm is restarted. The bifurcations are computed by solving an augmented system suggested by Moore which has the bifurcation point as a local unique solution with an ordinary Newton method. Because bifurcations are destroyed by arbitrary small perturbations, this system treats the unfolded case as well.

The programm is written in ANSI-C in an object oriented programming style, so it is easily extendable or decomposable in several modules. Especially the (Gauß) Newton methods and the methods of linear algebra can be used as standalone modules for other applications.

Because graphic display of the computed solutions is an intersting feature, but the details of the graphic output are heavily machine dependend, Alcon-S does not support any graphic drivers as part of the program. Instead of such a fast becoming antiquated solution some scripts are distributed for the widely spread interactive computation program MATLAB©.

Requirements to the Problem Alcon-S assumes, that the problem has a continuous first derivative, and if bifurcations are computed, additionally a continuous second derivative. The user must supply these derivatives. A numeric derivation is supported; however for the computation of bifurcations, the algorithm is more reliable with analytic derivatives. In this case, at least an analytic computation of the first derivative is highly recommended.

For utilizing the sparsity structure, Alcon-S also need knowledge of the sparsity pattern of the derivatives. It is more useful to support Alcon-S with one (reasonably small) pattern, which is a superset to all sparsity pattern of the derivatives along the path, so the pattern does not change along the solution path.

For starting the continuation, Alcon-S needs an inital guess, which must be a quite good approximation to a solution on the path. Except for an inital steplength estimate, Alcon-S does not need any further parameters to be set for average problems. In case of very hard problems however it supports two securitity parameters, who can be as restrictive as needed.

Further Reading A description of the algorithm (without consideration of sparsity) can be found in the report

> P. Deuflhard; B. Fiedler; P. Kunkel: *Efficient Numerical Pathfollowing beyond Critical Points*, Technical Report 278, Universität Heidelberg, 1984

which discusses an earlier FORTRAN-implementation of this algorithm based on QR factorization. A shorter, but more widely recievable discussion of the same authors is published as

> P. Deuflhard; B. Fiedler; P. Kunkel; *Efficient Numerical Pathfollowing beyond Critical Points*, SIAM J.Numer.Anal. 18:949-987, 1987

The tricky details of the sparse solver MA28 out of the Harwell library are described by the author of the FORTRAN package

I.S. Duff: *MA28 – A Set of FORTRAN Subroutines for Sparse Unsymmetric Linear Equations*, Harwell Report A.E.R.E.-R.8730, 1980

This manual only discusses the usage of the program, not it's working. Details of the algorithm in it's actual implementation and further references can be found in the file `documentation.ps`, referred as "documentation" in the sequel, which is written by the author as his diploma thesis. Because the English idom of the author is somewhat creepy (as Your may have noticed in the meantime), his supervisors suggested him to write this thesis in German language for at least German speaking people may decipher it. The author is very sorry, that no English or other readable version of the documentation exists.

2 How to Install Alcon-S

To install Alcon-S on a UNIX workstation from the `ftp` access of the `eLib`, follow these steps

- Make a new empty subdirectory in Your home directory which name should be `alcon_s` and copy the file `alcon.tar.Z` to this directory. If You use another name or place for the directory, You have to change some environment variable (see below).

- Type

  ```
  uncompress alcon.tar.Z
  tar -xvf alcon.tar
  ```

  The source code now should be unpacked in several subdirectories on Your system. While the compressed `tar` file has a size of 319 kB, the unpacked source codes occupy something around 918 kB.

- The file `makefile.def` contains several global environment variables, which are included by every `Makefile`. You may change these vairables to adapt the installation to Your local system, as there were

Name	Meaning	Default
\$CC	used C-compiler	`gcc`
\$CPPFLAGS	Flags for the compiler	`-g -c -O2`
\$LFLAGS	Flags for the linker	`-g -ldl -lm`
\$BASE	Name of the installation directory	`$HOME/alcon_s`

- After You have changed this variables to Your personal needs, type

  `makeall`

  to start a script, which compiles all submodules in the several subdirectories and links them together. The now completely installed software occupies at 12 MB of memory.

Machine dependent subprograms There is only one machine dependent subroutine in the file `mystd/mystd.c` with name `Seconds`, which returns the actual CPU-time in seconds. If this does not compile and You are not able or willing to rewrite this procedure, replace it by a dummy routine returning a constant floating point value. The only strange behaviour of Alcon-S following from this should be that it states to need 0.0 seconds for each computation that it makes.

If the program compiles, but shows misbehaviour as running, this may be caused by a misfunction of the `isNaN` macro, which is also defined in the `mystd` module. You can use the `test` program in this submodule to test the working of this macro.

3 Running Alcon-S

3.1 Start up

Alcon-S as a standalone application is started by the commandline

```
alcon [problemname [param1 [param2 [ ... ]]]
```

where `problemname` is the name of an installed *problem* of Alcon-S and `param1`, `param2` ... are additional parameter of the problem, to which continuation is *not* applied. A list of preinstalled problems is listed in subsection 3.3 below and section 4 explains, how to install Your own problems. If the problem has more parameters than are specified by the commandline, then default values are used. If the commandline specifies more parameters as there were in the problem the additional parameters are ignored.

If the problem is not installed, Alcon-S asks for another problem name. Type `quit` to leave Alcon-S at this stage.

After reading in the commandline parameters, Alcon-S tries to read default options of the continuation algorithm from the file `./Application/`*problem*`.dat`, if this exists. The options can be changed and stored or reloaded, while Alcon-S is in it's interactive mode. Alcon-S prints a message on the result of the reading of the options.

Bugs: No options can be set with the comandline in the actual implementation.

The parameter values stored in the *problem*`.dat` file override the commandline parameter.

3.2 The interactive Menu Structure

Alcon-S is driven by an interactive menu tree, where one letter commands either swap some switches, call a dialog to change some parameters or branch to a higher or lower menu in the tree. You can identify the actual menu by it's promt. On entering a menu, the whole possibilities of this menu are listed in a tabular. There are a lot menus in Alcon-S, but be not confused – most of the options You will never need. Most interesting are the "jacobian update" switches in the (Gauß) Netwon menus and the "compute/do not compute turning points" and the "scaling" switch in the `!s` "solver-special" menu. For the rest assume, that if some command is not self explaining, it is not important. This section contains a complete list of all commands in all menus of Alcon-S for reference purpose only. Skip it at first reading except for the main menu.

Some commands are applicable in every menu. They are:

`h` :	Help: List all commands
`q` :	Leave this menu
`<ESC>`:	Leave the menu hierarchy to the main menu

The `<ESC>` command is useful, if You get lost in the depth of the hierarchies and don't want to count the number of `q`'s to get back to the main menu.

All the changes that You did apply immediatly. However, some operating systems catch escape keys from the standart input, so with these systems You cannot partipiciate this useful feature. In the file `mystd/mystd.h` the constant `ESC` is defined by it's octal value. You can change this constant to some value reachable by Your keyboard.

3.2.1 The Main Menu

After You start Alcon-S, You come to the main menu. Beside of the always applicable commands You can type:

`n`:	Start a new diagram, forgetting former computed solutions
`f`:	compute an initial solution from the inital guess
`p`:	compute a next point on the path
`b`:	continue the actual solution up to the next singularity of the path
`c`:	complete the whole bifurcation diagram
`x`:	stop following the actual branch and cut off the solution path
`v`:	view the actual computed point
`s`:	show a statistic of the computation
`?`:	branch to the problem menu 3.2.2
`!`:	branch to the solver menu 3.2.4
`w`:	branch to the write menu 3.2.3

You can also type in a number digit by digit. If the number is followed by a `p` or `b` command, then this command is executed this number times; otherwise the number is ignored.

The `c` command also reports the time needed for the completion of the diagram and if some errors has been detected while finishing the diagram. Subsequently it calles the `s` command.

After the `q` command the program leaves the main menu and asks for a new problem name to compute. If You type `quit`, the program exits.

There are two seldom used commands, who are not listed in the main menu. They are:

`k`:	skip the actual branch; continue it later
`l`:	compute the last branch again

The first command is useful, if You want to start in negative λ direction from the inital solution. Alcon-S starts by default in positive direction, but if You type `k` after `f`, the branch in positive λ direction is skipped and the branch in negative λ direction is the actual branch.

Bugs: After `q`uitting the main menu the program forgets all the computation done so far, so a wrong keystroke can destroy much time comsuming computation.

The `l` command was for debugging diagnostics only and is superfluous in the current version. Moreover, it tends to abort the program, especially if it is called in the middle of a branch.

3.2.2 The ? problem Menu

The ? menu handle the needs of the problem, as there are:

`t`:	set the desired accuracy of the computation
`n`:	set a lower bound for the values of the variables and the parameter
`m`:	set an upper bound for the values of the variables and the parameter
`i`:	change the inital guess
`p`:	change the external parameter of the problem
`l`:	load an inital guess from an external file
`s`:	branch to a problem specific menu, if there is one

The contends of the `s` menu is left to the user who has defined the problem (see section 4). The format of the file, from which the `l` command reads, is the one of the *actual graphic point* format made with the `a` command of the `w`rite menu, see below. Consequently it is read from the graphic file as declared in the `w`rite menu in 3.2.3.

From the upper and lower bounds are mostly only the values of the parameter of interest while all bounds of the variables are set normally much too large to influence the computation. Principially these values limit the range of computation to an $m+1$ dimensional quad.

3.2.3 The `w`rite Menu

Before discussing the ! menu with it's many submenus, let's look at the `w`rite menu. There are several commands to dump all computed or special solutions as text or in a format useful for the graphic scripts of MATLAB©, see section 7.

To the `write` menu, there are three associated files:

Name	Usage	Default Filename
Print file	for text containing solution points	`./Application/`*problem*`.net`
Graphic file	for data in a format useful for the graphic scripts	`./graphic/`*problem*`.dat`
Data file	for the default options of Alcon-S for this problem	`./Application/`*problem*`.dat`

The commands for operating on these files are:

`c:`	print the complete computed bifurcation diagram
`f:`	print all solutions except of the free ends of not completly computed branches
`l:`	print all solutions on the last computed branch
`b:`	print bifurcations
`t:`	print turning points
`d:`	print dead ends of branches, where the continuation algorithm has breaked down
`p:`	send the print file to the printer
`g:`	make a graphical two dimensional representation of the whole diagram
`a:`	make a graphical representation of the actual computed point (the one viewed by the `v` command
`v:`	make a graphical representation of the whole diagram in all coordinates
`s:`	save the actual options of Alcon-S as default values to the data file
`r:`	restore the default values from the data file
`o:`	change output parameters and filenames

The `c` to `p` commands apply to the print file; the commands `b` to `d` give only a two dimensional format answer, while the other dump the complete solution. The `g` to `v` commands apply to the graphic file and the `s` and `r` to the data file.

The `o` command calls a small submenu, where beside of the different filenames the output format for the print file and the two coordinates can be choosen. The `g` command assumes normally, that the parameter is taken as the *x-axis* for graphical representation, while one of the variables serves as *y-axis*. It is also possible to choose the norm of the variables as *y-axis*. The `b` to `d` commands also uses this graphical format to produce lines of output:

singularity *y-axis* `at` *x-axis*

However, they never use the norm as *y-axis*.

Bugs: If the size of the problem is changed while Alcon-S runs, which can happen e.g. for some discretizations when the grid is changed, then the `write` menu options are not updated according to this change. Especially the *x-axis* will not be the new index of the parameter, if the dimension of the problem grows.

Do not use the `l` command if You have not finished a branch at the moment because otherwise the program terminates ungracefully.

3.2.4 The `!` solver Menu

The solver menu keeps together the options of the pathfollowing method for solving the problem. There are the commands:

`i`:	inital steplength guess s_{start}
`x`:	storage format of the jacobian matrix
`y`:	storage format of the second derivative
`p`:	branch to the submenu of the Gauß Newton method for pathfollowing
`b`:	branch to the submenu of the Newton method for computing bifurcations
`j`:	branch to the submenu for the decomposition method of the jacobian matrix
`m`:	branch to the submenu for solving the linear equations of the system of Moore
`s`:	branch to the submenu containing several optional switches of Alcon-S

For the inital steplength a default value is computed from the boundary of the parameter to $s_{\text{start}} = 0.001 * (\lambda_{\max} - \lambda_{\min})$.

The storage formats of the jacobian and the second derivative (not the full tensor is needed here, but a special projection, see below in section 4) can be either *full*, where all indices are stored, *sparse*, where only entries contained in a specified pattern are stored, or *eval-only*, where only a procedure for evaluation of Ax and $A^T x$ are supplied. Each problem proposes a special format for each of the derivatives, which seems to be most appropiate. The user can change this default value, but the evaluation method of the problem and the decomposition method of the solver can reject a format, if they cannot handle it. Especially for the format of the jacobian, *eval-only* is always rejected, because this is only useful for iterative methods, while only direct methods are implemented in the current version of Alcon-S. For the second derivative, such a format is supported by the solver, however.

3.2.5 The switches Menu

Let's now look at the several switches of Alcon-S as there are:

`l:`	apply the steplength control strategy
`r:`	set the security factor ρ for steplength estimation
`t:`	compute turning points
`@:`	compute bifurcation points
`s:`	scale the variables and equations by an automatically adapted scaling along the solution path
`i:`	set the scaling modus of the equations
`n:`	use a normalized norm for the variables
`b:`	use a normalized norm for the variables of the system of Moore
`a:`	set the minimal cosinus $\text{tangent}_{\text{min}}$ of the angle betwen two consecutive points
`e:`	apply an additional security check for the tangents to prevent branch jumping at bifurcations
`c:`	choose a more cauticious steplength strategy
`m:`	print messages of the continuation process
`w:`	print warnings on the continuation process

The security factor ρ is used by the steplength control; for this, the algorithm computes after each step an estimate of the optimal steplength s_{opt} from the convergence history. Then is chooses the new steplength as

$$s_{\text{new}} := \rho \cdot s_{\text{opt}}.$$

Obviously the constraints $0.0 < \rho \leq 1.0$ are necessary.
If no steplength control is desired, this factor is obliguous.

If the scaling option is in effect, Alcon-S tries to scale the variables (including the parameter) $y = (x, \lambda)$, such that the scaled variables are at the same order of magnitude. For this it defines a diagonal scaling matrix D at each point that $\bar{y} = Dy$ are the scaled variables. After the computation of a new point $y_{\nu+1}$, the old matrix D_ν is updated like

$$D_{\nu+1} = \text{diag}(d^1_{\nu+1}, \ldots, d^{m+1}_{\nu+1}) := \text{diag}(\,\frac{1}{2}(d^1_\nu + \frac{1}{|y^1_{\nu+1}|}), \ldots, \frac{1}{2}(d^{m+1}_\nu + \frac{1}{|y^{m+1}_{\nu+1}|})\,)$$

with some constrains for the cases $|y^i| \to \infty$ and $|y^i| \to 0$. For the equations there are three possible scaling strategies; taking D^x as the part of the scaling matrix operating on the variables and D^{eq} as the scaling matrix of the equations, one can choose

1. $D^{eq} = Id$, which is the *no scaling* option

2. $D^{eq} = D^{x}$, which is the *same as domain* option

3. $D^{eq} = (D^{x})^{-1}$, which is the *symmetric* option

The symmetric option has it's name from the fact, that with this option in effect the scaled matrix of the variable part of the jacobian $\partial f/\partial x$:

$$\overline{\frac{\partial f}{\partial x}} := D^{eq}\frac{\partial f}{\partial x}D^{x-1}$$

is symmetric iff $\partial f/\partial x$ is symmetric.

If the problem is not too bad scaled from the beginning, the scaling option will impove the performance of the code significantly. On the other hand it may lead to failure for very bad scaled problems. The influence of the choosen scaling of the equations has been neglegible for the test problems so far.

The both normalized norm options replace the standard euclidean norm $\|y\| = (\sum y_i^2)^{1/2}$ by a normalized norm $\|y\|_N = (\sum y_i^2/N)^{1/2}$, where N is the number of coefficients of y. For pathfollowing $N = m+1$ while the system of Moore has twice as much variables, therefore $N = 2m+2$ in this case.

After the computation of a new solution $y_{\nu+1}$ with tangent vector $t_{\nu+1}$ there is made a security check if $t_{\nu+1}$ includes a sufficient angle with the tangent of the last computed point t_ν:

$$\langle t_{\nu+1}, t_\nu\rangle \geq \text{tangent}_{\text{min}}$$

Default value is 0.0; a value lower than -1.0 bypasses this check.

While following a path, the predictor may come near to another path in the neighbourhood of a singularity. Then it is possible that the corrector converges to this path. In such cases the tangent often makes a big turn, while the convergence is pretty fast. The `e` option checks for such inconsistencies between the tangents and the convergence history to prevent path jumping. If there are no bifurcations, this check is superfluous.

The `c` option makes a more restrictive consistency check between tangents and convergence, if no scaling is done. (If scaling is in effect, the more restrictive check is not applicable.) Additional it switches off a check at computing bifurcations, if a former computed bifurcation is so near to the actual point, that the Newton method computing the actual bifurcation seems to be converging to this point. Switching off this check is sometimes necessary in the case of two or more nearby bifurcations.

Bugs: The values of ρ and $\text{tangent}_{\text{min}}$ are not checked for being in valid ranges. Alcon-S is very chatty, if the `m` option is in effect.

Leaving the ! menu (as for the ? menu makes Alcon-S forget all actual solver specific information about a problem actually computed. This is especially true for the pivot strategy of the sparse solver (see below in section 3.2.7).

3.2.6 The Gauß Newton Menus

There are two objects contained with solving nonlinear equations by the Gauß Newton method, the one as the corrector in pathfollowing and the other one as an ordinary Newton method to solve the augmented system of Moore for bifurcations. Both have the same menu structure; beside of the `m` and `w` switches printing messages and warnings there are:

`s`:	scaling method for the Newton process
`u`:	update modus for the jacobian
`g`:	switch global versus local Newton method
`x`:	switch exact versus inexact Newton method
`t`:	set tolerance (superfluous, if used with Alcon-S)

The scaling method can be *none*, in which case canonical norms are used, *standart*, which includes a scaling, that changes while the Newton process in a similar way than the scaling while pathfollowing, with the exception, that the scaling changes at each Newton step and not at each continuation step, and *user-defined*, which means in this case, that the scaling of the continuation process is in effect. The last one is the recommended default for Alcon-S.

The update modus can be one out of three possibilities: *full*, which is the ordinary Newton process making one jacobian evaluation at each Newton step, *simplified*, which meaks only one jacobian evaluation at the beginning of the iteration keeping this jacobian over the whole iteration process, and *Broyden updates* which uses the well known method of Broyden to update the starting jacobian after each iteration step. Every method makes a full jacobian evaluation after the final step for guaranteing that the tangents are accurate.

Normally the Newton methods can be kept local for Alcon-S, what means, that no damping strategy is applied. For the simplified methods it can be reasonable to set the method to global, because then at a damped step the jacobian is automatically reevaluated, possibly leading to convergence instead of termination of the iteration, like the local method would do.

The inexact method leads to an accuracy matching strategy for the linear solvers, because in the first steps normally the linear systems needs not to be solved as exact as possible, but only with an error in the same order as the truncation error of linearization. This option is not useful for pathfollowing, because only direct solvers are implemented in this version of Alcon-S, normally giving accurate solutions after the first step of iterative refinement which is usually performed. For computing bifurcations on the other hand there are several kinds of usefull iterative refinement.

The tolerance option is only useful for the Newton objects used as standalone applications; for Alcon-S, their values are always reset to the tolerance in the ? problem menu when leaving to the main menu.

Bugs: There is also a switch to print convergence vectors; this leads to a segmentation fault if used with rather large ($m > 20$) problems.

3.2.7 The linear Algebra Menus

As for the Gauß Newton objects there are two associated linear algebra objects with their menus. These menus have the options:

`c:`	choose the method of decomposition
`f:`	force the decomposition always to have full rank
`i:`	try iterative refinement to improve the solution
`t:`	set tolerance (superfluous, if used with Alcon-S)
`s:`	branch to a special menu for the actual solver

There are also the ususal `w` and `m` commands.

For pathfollowing beside of choosing between QR decomposition, dense LU decomposition and the sparse solver MA28, one can also switching to the block meta solver, performing a decomposition only on the variables part $\partial f/\partial x$ of the jacobian. Trials to choose the solver for the system of Moore are responsed by a message, that only one solver is available here.

The `f` option forces the solvers to perturb the decomposition to a nearby decomposition of full rank, if the solvers detect rank deficency. This is useful in connection with the block meta solver; at turning points, where $\partial f/\partial x$ becomes rank defect, Govaerts and Pryce have prooven, that one step of iterative refinement makes the solver stable, if the rank deficient matrix $\partial f/\partial x$ is replaced by a nearby matrix of full rank.

The `i` options tries to improve the solution by iterative refinement. This is necessary to set in connection with the full rank option, as explained before. It is also useful in connection with the sparse solver MA28, because this is an easy way to check, if the current pivot strategy is sufficient for a stable decomposition.

The `s` command has different effect depending on the actual solver.

- For the QR decomposition, there is no menu.
- For the dense LU decomposition, one can choose the pivoting strategy. Complete pivoting is necessary, if bifurcations should be computed. Otherwise row pivoting is normally sufficient. Extended row pivoting scans also the following row if some row contains too small entries, which may lead to detection of rank deficency, but is surely weaker than complete pivoting. Because the linear algebra methods can be also used as standalode modules, column pivoting is included, too.
- The sparse solver MA28 has the most switches in it's special submenu; most of them are contained with aspects of the stability of the decomposition or the pivot strategy; after the sparse solver has choosen a pivot strategy for some sparsity pattern, it can use this strategy on any other matrix with the same pattern, which leads to a drastic performance improvement for this decomposition.

`u:`	stability factor for pivot search
`n:`	number of nonzero entries of the matrix to be decomposed
`p:`	overhead place factor of the decomposition
`r:`	switch to remove small entries from the decomposition
`t:`	compute decomposition of transposed
`!:`	use deflation technique to force rank defect
`s:`	compute stability factor in decomposition phase
`b:`	compute largest element in decomposition phase
`o:`	use old pivot strategy, if possible
`x:`	always choose a new pivot strategy, if the rank is decreased
`f:`	choose a new pivot strategy, if the decomposition fails
`a:`	print all decompositions steps (for debugging)

The stability factor u should be between 0.0 and 1.0; the sparse solver tries to minimize the fill-in in each step under the constraint that the modulus of the choosen pivot is always larger than u times the maximal entry in this row. This factor controls the pivot strategy; for a factor of $u = 0.0$ the pivots are choosen on sparsity considerations only, while a factor of $u = 1.0$ leads to ordinary row pivoting.

The place n normally needs not to be changed for use with Alcon-S, because the problem should set this value right. On the other hand, the multiple place needed for the decomposition may exceed $p{\cdot}n$ depending on the nonzero distribution and the pivot strategy. The sparse solver tells the user, when there is not enough space with an estimate of the needed additional space; the user must provide more space by hand.

The `r` switch let the sparse solver remove small entries from the decomposition. This is only useful for special structured matrices, otherwise the result may be desastruous wrong. Always apply iterative refinement and recognize, that the sparse solver can not accept the pivot strategy to following matrices, because they may have pretty different large entries in their nonzero pattern.

The computation of the transposed is useful for overdetermined systems, because this changes the row pivoting strategy to more appropriate column pivoting. It may also be useful to people using columnwise packed storage schemes who may rewrite the interface part of the sparse solver skipping some data resorting routines.

The `!` switch applies inverse power iteration to compute singular vectors to the smallest singular values; these are set to zero by deflation techniques to force a rank defect matrix, which is necessary to compute bifurcations. If this switch is turned off, the smallest pivots are dropped from the decomposition instead, leading to a faster but much less stable method for computing bifurcations.

If no iterative refinement is applied to the solution, the sparse solver estimates the stability of the decomposition by some stability factors. They may be computed in the decomposition phase, which consumes a considerable amount of time, or estimated after the decomposition. Depending on these values the sparse solver decides, if the actual pivot strategy is still useful or should be rejected.

The `o` switch controls, if an old pivoting strategy is applied anyway. If this is false, some of the above and all of the following switches become superfluous. In the context of pathfollowing it is usually useful to keep this flag on. The `x` switch leads to a new pivot strategy while computing bifurcations, because sometimes additional stability requirements are necessary at these points. The `f` option leads to an immediately computation of a new pivot strategy, if the old one failes i.e. because of changing of the nonzero structure. If the sparse solver is used as a standalone module this may be set to false, if the calling program may decide to perform some other action in this case; for use with Alcon-S it is best to keep this true.

- The special menu of the block meta solver is simply the linear algebra menu of the decomposition method of the variables part.
- The special menu for the solver of the system of Moore has some flags for iterative refinement. The solver makes the asumtion, that a perturbation parameter α, measuring the unfolding of the bifurcation, is zero and that the jacobian is rank defect from the beginning of the iteration. Both assumption may be corrected by iterative refinement over the perturbation α (by `a`) or the part of the jacobian neglected by the decomposition (by `j`). One may perform also more expensive iteration over the whole jacobian of the augmented system, overriding the other two options.

  The additional `d` option is usefull if You try to apply Your own solver to Alcon-S. For pathfollowing, You must only supply two methods, one computing the pseudo inverse (ore some other right inverse) and another computing a kernel vector. If You want to apply Your solver to the computation of bifurcations, then You will be confronted with a lot of additional routines. If You turn the `d` option on, then beside of the solve routine (who must compute the pseudo inverse in that case) and the kernel routine (which must also handle more than the one dimensional case, computing orthonormal bases in the multidimensional case) You only need two methods to compute the pseudo inverse of the transposed and a kernel basis of the transposed, and a "NewRank"-method which Alcon-S calles when it starts (or finishes) to compute bifurcations to tell the decomposition to be rank defect (or to be of full rank). Unfortunately this option leads to a possibly less stable solver, especially the `j` switch must be ignored, so keep it off, if You use some of the build in solvers of Alcon-S.

Bugs: In use with the block solver, allocation is sometimes not correct, especially at startup. Just call and leave immediatly the special menus of the solvers to correct the allocation.

If You switch to the block solver, all solver specific parameter are reset to their default values. You must reset them by hand to the desired values.

The sparse solver may recursively reduce the rank of a given matrix in a nearly infinite loop for worse scaled equations. (The loop stops, if the rank reaches zero, of course.) If You want to prevent such a rank decision, You can set the full rank flag to true as a first trial. This leads to a slightly perturbed decomposition, which sometimes can lead to very unsatisfactorily results. In this case, there is a second solution, for which You must change the sources, however.

In the file `LinAlg/MA28.c` there is in the procedure `Estimate2Condition` a constant with name `RANKMAXDEFECT`. If this constant has a nonnegative value, this value will limit the computed rankdefect. Changing this to the value `0` for pathfollowing and `1` if also bifurcations should be computed will stop the sparse solver from computing an arbitrary high rank defect.

The print all option for the sparse solver was only for debugging while developing the sparse solver and is kept for future expansion programming. Turned on with realistic problems it will terminate the program because of too much output.

The special menu of the block solver suggests that no special options for the block solver exists. This is not true, but the only one special option is "hard wired" as a constant, because it seldom needs to be changed. Normally the block solver assumes that the variables part of the jacobian has the same rank as the full jacobian. This leads to a solver scheme similar to those of Govaerts & Pryce (see the documentation). But for computing a special kind of bifurcations (see the biparabel problem of the preinstalled samples in figure 1) it is preferable to assume that the variables part is rank defect and decompose the assumed rank deficient part together with the parameters part. This leads to a solver scheme similar to the "deflated block elimination" of T.F.Chan. The second is more stable but much slower, so the first one is the default. If You want to change this switch, You must change in the source file `LinAlg/block.c` the constant `deflateMode` (at the top of the text just behind the `#includes`) from `false` (first method) to `true` (second method).

3.3 Preinstalled sample Applications

There are several sample applications delivered with Alcon-S, most merely for testing the code after modifications. There is also a more interesting general scheme for elastic rod constructions and a small circuit example. All sample applications are listed in figure 1.

For the stab*constr*, You need a *constr*`.crt` file in the subdirectory `Application/`. These `.crt` files have the following format:

First there are the coordinates of the construction points, where the rods met, enumerated in ascending order:

#no x-coord y-coord z-coord

where the coord's are given in millimeter. Then follows a list of fixed coordinates of points

#point-no. coordinate-no.

with coordinate-no.(x) = 1 etc. Those coordinates may be fixed because i.e. they are the z-coordinate of a point on the bottom.

Name	Description
frame	simple test example (line crosses circle)
biparabel	simple test example – normal form of a (standart speaking) nonsimple bifurcation containing two parabolas crossing in the origin
bruss	linear n-cell reaction diffusion modell
layer	boundary value problem with layer of parameter dependent thickness; solved by simple finite differences
oscillator	invariant manifold of two coupled harmonic oscillators, solved as a PDE by simple finite differences
stab*constr*	elastic rod construction model; computation of deformations under external load
circuits	a flip-flop like bistable circuit and a feedback inverter – computation of operating points by source stepping

Figure 1: Preinstalled Samples

After this a list of connecting rods appends with format

#point-from #point-to sect-area

for each rod connection #point-from with #point-to having the sectional area sect-area (in mm^2).

Finally a list of external loads followes, each line of

#point-no. coordinate-no. load {1/0}

where on point #point-no. in direction of coordinate-no. the load (in N(ewton)) presses, depending on the last entry either constant if this is zero, or linear in λ, as load $\cdot\, \lambda$, if the last entry is one.

Each list is finished by a line full of zeroes. A % at the beginning of a line indicates a comment.

There are two `.crt` files delivered with Alcon-S, the one a small tent and the other one a rather large current supply mast. The coordinates of the second example are still depending on the two ascending angles α and β. The application `mk` computes the height's of the construction levels under the assumption $\beta = 0.6\alpha$ by continuation from the trivial solution $\alpha = 0$ to the correct height of 47.24 m. You can change the relation between α and β as an external parameter to compute different constructions. After reaching the boundary plot the solution with the `wa` command and call the program `makecrt` in the directory `Application/` (UNIX users can call the script `makeCrt` to do this). The programm automatically writes the new `mast.crt` file with the new coordinates.

Bugs: The sample circuits has discontinuous derivatives, which are not smoothed for the inverter. This makes the continuation break down, if one scales up the equations.

4 Defining Your Own Problems for Alcon-S

For defining Your own problem, You must first define Your problem object in its own file, then change a special routine in the Alcon-S sources for the program may initalize Your problem and then recompile the last two modules of the program.

First for defining Your problem change to the subdirectory `Application/`.

There is a start-up file in this directory called `Application.c` beside of the sample applications. (If You want to look at the samples for an already defined problem, look at best first at the `frame.c`.) Copy this file to a file called *Problem*`.c` where *Problem* is the name You want to give to Your problem. (*Problem* should not be `quit` or `uit`.) In this file there is a method `SetUpProblem` at the end of the text which is the constructor for the object `AlconProblem`. Change its name to `SetUp`*Problem*.

This constructor sets up some important values of the object of type `AlconProblem` with global variables. You only need to change these variables (defined just above the constructor). (The definition of the object structure is in the file `connection.h`.)

You must set

`m`	:	dimension of the problem
`Taumin, TauMax`	:	limiting values of the parameter
`Tol`	:	accuracy of the computation
`ProblemName`	:	*Problem*
`NumberofParameter`	:	number of the external parameter
`ParameterString`	:	name of the external parameter

Now You must define Your problem function in the procedure `f`. The prototype of this procedure is:

```
static Bool f(RealVec x, Real tol, Generic data,
RealVec b)
```

where `x = x[1], ...,x[m],x[m+1]=`λ is the vector (`RealVec = double*`), where the function is evaluated, `tol` is the actual tolerance of the function evaluation. The `data` (which is transported through a generic pointer `Generic = void*`) may contain some problem specific data, which can be set by You in the instance variable `problem->fData` in the `SetUp`*Problem* routine. The result should be in the vector `b`, (`b[1], ..., b[m]`). The function result is true if the function evaluation succeds, false otherwise (i.e. because the vector `x` is out of the definition domain).

After this the first derivative must be supplied in the procedure `df` which prototypes to

```
static Bool df(RealVec x, Real tol, Generic data,
LinOp a)
```

The parameter `LinOp a`, who contains the result, is an instance of a structure

```
typedef enum {eEmpty, eFull, eSparse, eEvalOnly} LinOpType;

typedef struct {
        LinOpType type;
        Bool symmetric;
        union {
           RealMat A;
           SparseMat S;
           Bool (*f)(RealVec x, RealVec b, Generic extra, Bool adjoint);
        } d;
        Int m,n;
        Generic extra;
} *LinOp;
```

The `LinOp->type` decides, which of the three variables of the union `LinOp->d` actually contains the information. If it is `eFull`, then the `RealMat =`

`double** A` contains the information in dense format (also here both row and column indices start with `1`). If it is `eSparse`, then the structure `SparseMat S` contains the information, where

```
typedef struct {
   Int nz, nzMax;
   RealVec a;
   IntVec  rowIndex,colIndex;
} *SparseMat;
```

Here each pair `(rowIndex[k], colIndex[k])` defines a nonzero entry whose value is in `a[k]`. Multiple occuring index pairs sum up their values. If any of the indices in a pair is out of range, especially equal to zero, the corresponding value is ignored. The `nzMax` contains the length of the vectors `a, rowIndex, colIndex` and *should be unaltered*, while `nz` containes the last used index and *should be set* appropriate after defining the entries by `df`.

Additional there is the poscibility for the `LinOp->type` to be `eEvalOnly` in which case `LinOp->d` must contain the procedure pointer `LinOp->d.f` such that `LinOp->d.f(x,b, LinOp->extra, false)` computes $b = Ax$ and `LinOp->d.f(x,b, LinOp->extra, true)` computes $b = A^T x$. You do not need to accept this case but return false, because no solver of Alcon-S can accept such `LinOp`.

The value of `symmetric` influences the storage scheme; if the type is `eFull`, then the lower part of `d.A` does not exist in memory, because their entries are the same as in the upper half. If type is `eSparse` entries with `colIndex[k] > rowIndex[k]` are ignored. Do not change this variable, because that may confuse Alcon-S in the `full` case.

By default there is support for numerical evaluation of `df`. Therfore in the `SetUp`*Problem* routine `dfData`, which is given as the argument `data` to `df`, is set to a special `Generic[0...2]` vector, where

`data[0]`:	contains the needed `fData`
`data[1]`:	containes a real array holding the steplength automatically choosen for the computation of the columns of `df` by finite differences
`data[2]`:	contains a coloring of the columns of `df`, where two columns may have the same colors, if they have no two row entries in common

The steplength for the finite differences approximating the derivative are choosen, so that the difference between $f(x)$ and $f(x + he_i)$ is of order of the square root of the machine accuracy. The algorithm to compute these steplength may fail on worse scaled problems, so You may switch this off by setting `data[1]` to `nil`.

The coloring in `data[2]` is used for an effective way to compute the jacobian in the sparse case, because columns with the same color may be computed together with one additional function evaluation only. If You know a coloring, set `(IntVec(dfData[2]))[0]` to the number of colors and `(IntVec(dfData[2]))[i]` to the color of column i. Otherwise set `(IntVec(dfData[2]))[0]=0` and the program will automatically compute some coloring. If You do not want any coloring at all, set `dfData[2]` to `nil`.

Before `df` there are the variables `linType` and `nz` to be set:

`linType`:	recommended type for the jacobian of the problem
`nz` :	upper bound for the number of nonzeroes in the jacobian

The `linType` is the type of the jacobian when the program first starts the problem; this type may be overridden by default values read in from the data file. Depending on the `type`, Alcon-S chooses automatically a QR decomposition for `eFull` LinOp's and the sparse solver for `eSparse` LinOp's.

For small problems, You can set `nz` to 0; Alcon-S will recognize this and change it to the maximal value `m*(m+1)`, if it allocates a sparse LinOp for this problem.

After having done this for the definition of the first derivative, You must do the same for the procedure `cf` with the prototype

```
static Bool cf(RealVec x, RealVec b, Real tol,
Generic data, LinOp c)
```

for defining the second derivative. This routine computes in the `LinOp c`

$$c = (f'^T(x) * b)'$$

or written componentwise:

$$c_{ij} = \sum_{k=1}^{m} \frac{\partial^2 f_k(x)}{\partial x_i \partial x_j} b_k \qquad i, j \in \{1, \ldots, m+1\}$$

where, as usual, $x_{m+1} = \lambda$. Note that this LinOp is symmetric !

If You do not need to compute bifurcations, You can let this procedure always return false, but do not forget to set the variables `ctype` and `nzc` in any case, which have the same meaning as the `linType` and `nz` for the `df` procedure. In contrast to the `df`, here `eEvalOnly` is a quite possible choice for `ctype`.

Bugs: For both `df` and `cf` there is numerical differentiation supported. This is only thought as a start up help. Especially You should not use numerical differentation for both `df` and `cf` at the same time when computing bifurcations.

The next two paragraphs are less interesting; skip if You want.

After defining the derivatives, there are some methods for housekeeping of the data. First there is a `ResetProblem(AlconProblem p, Int n)` which resets the actual problem to the size `n = m+1`. In the file `Application.c` it is assumed, that the special structure of `dfData, cfData` is still the one needed for numerical differentation, so You need to change this possibly. This method is called every time the following menu changes the size `n`; otherwise it will never be called and You may ignore it.

Then there is an empty hull for a menu to set up local variables of the problem. (This is the `?s` "problem special" menu.) You can override this empty menu by just return direct after call, if You do not need a menu.

Now comes a more interesting method: `SetParameter`. Beside of the external parameter it sets the limits of the variables and the parameter. This method is called by the constructor of the problem or explicitly by the `?p-1` command.

Even more important is the `SetInitalGuess`, which sets a guess of an inital solution in the variable `problem->yGuess`. This method is called by the constructor and by the `?i-1` command.

Finally there is a method called `ClearNumDerivates` which delocates any allocated memory in the `fData`, `dfData` and `cfData`. This method is called by the destructor (in `connection.c`).

After You have defined Your problem, You must link it to the rest of Alcon-S. To do this, follow these steps:

- In the file `Application.h` add the prototype of Your constructor by the line

```
extern Bool SetUpProblem(AlconProblem problem);
```

- In the file `connection.c` there is a general constructor called `NewAlconProblem`. In this Method just add the line

  `|| SetUp`*Problem*`(problem)`

  in the first if-clause, where all the other constructors are called if they can construct the problem `name`.

- (For UNIX only) In the `Makefile` add the name *Problem*`.o` to the variable $APPLICATIONS and recompile this local module.

- (For UNIX only) In the main directory ($BASE), add in the `Makefile` the line

  `$(BASE)/Application/`*Problem*`.o \`

  in the variable $OBJS and recompile.

Now You should call Alcon-S to compute Your problem with

`alcon` *Problem* .

Bugs: Contrary to the FORTRAN version of Alcon, the computation is not guaranteed to be in the limits specified by the `SetParameter` method, though it is very unlikely that it exceeds this limits. Your `f` method may check for security against these limits, returning `false` if they are violated. In no case a `df` of `cf` evaluation will be made then, because before every evaluation of a higher derivative a successful function evaluation must be made.

In the current implementation no local variables or options can be saved in the data file *problem*`.dat`.

5 Using Alcon-S as a submodule

Although Alcon-S is a standalone program, it can be turned easily into a subroutine by a slight modification of the sources. How to do this is described in this section; for sake of simplicity it is assumed, that the whole package is installed in a subdirectory of the calling program with name `./alcon_s`.

First You must modify the file `alcon.h`. There is a line

```
#define MAIN
```

which You must change to

```
#undef MAIN
```

Now recompile the main program – the submodules need no recompilation. (If You do this with the `Makefile`, You will get the error: `_main: not defined` or something similar, which You can ignore.)

Now You can call a couple of Alcon-S menu methods from Your program, if You include the header file by

```
#include "./alcon_s/alcon.h"
```

in Your sources before You want to call Alcon-S. It is only necessary to include the standart headers `<stdio.h>` before. In `alcon.h` normally all other headers are automatically included, so You have access to all data structures, if You want to. If You do not want this, i.e. some data types defined by Alcon-S collide with some defined by Your own, just

```
#define _alcon_extern_
```

before You include the header `alcon.h`. The declarations will be automatically excluded.

The methods to which You have access are listed in figure 2. They all return `void`. If You have not defined `_alcon_extern_`, then You may also call other routines or set variables which represent options.

There are four global objects, which are

`gProblem`	contains all data of the problem
`gSolution`	contains all data for the solution algorithm
`gOutput`	filenames and output formats
`gStatistic`	some statistical counters

There is also a procedure `ActualNet()`, which returns the actual net, a data structure, in which the actual bifurcation diagram is stored, and a procedure `ActualEdge()`, which returns the actual edge, where the algorithm does it's computation. The corresponding data structures are defined in the file `Net/Net.h`.

A short description of all files and modules can be found in the `index` file. To each submenue there is a corresponding object, whose public data can be changed by the calling program, if some housekeeping procedures are called afterwards to tell Alcon-S the changes.

Each object has a `Reset`*object* procedure, which resets the internal data after some public variables are changed. It would take to much space to explain each object in detail; look at the *object*`Menu` procedures if You want

Name	Use	Menu
`InitAlcon(void)`	Initalizer for Alcon-S; always call before You call anything else	./.
`InitConnection(char *name)`	Initalizes the problem `name`	./.
`MainEventLoop(void)`	starts the interactive main menu	./.
`NewDiagram(void)`	clears the storage for a new diagram	n
`FirstSolution(void)`	computes starting point	f
`NextPoint(void)`	computes the next point	p
`NextBranch(void)`	computes the next branch	b
`SkipBranch(void)`	skip this branch	k
`KillBranch(void)`	terminate this branch	x
`CompleteDiagram(void)`	complete the diagram	c
`PrintActualPoint(void)`	print the actual point to the standart output	v
`WriteMenu(void)`	start the interactive `write` menu	w
`ReadProc(void)`	read default options from data file	wr
`SaveProc(void)`	save actual options to data file	ws
`PrintDiagram(void)`	print the complete computed bifurcation diagram	wc
`PrintContList(void)`	print all complete solutions	wf
`PrintLastBranch(void)`	print all solutions on the last computed branch	wl
`PrintBifurcations(void)`	print bifurcations	wb
`PrintTurnings(void)`	print turning points	wt
`PrintBoundaries(void)`	print points on boundary of the definition domain	wd
`PrintDeads(void)`	print dead ends of branches	we
`SendFileToPrinter(void)`	send the print file to the printer	wp
`MakeGraphicFile(void)`	make a graphical two dimensional representation	wg
`MakeActualGraphicFile(void)`	make a graphical representation of the actual computed point	wa
`MakeCompleteGraphicFile(void)`	make a graphical representation of the whole diagram	wv
`QuitConnection(void)`	quit the actual problem	q
`QuitAlcon(void)`	quit Alcon-S, delocate all objects	./.

Figure 2: Defined Methods in `alcon.h`

to know how to treat an object, whose variables are changed. Better call the `ResetAlconProblem(gProblem)` after each changing of subobjects on `gProblem` or `gSolution`.

Of special interest is access to the last computed point on the branch. You can do this normally by the following program fragment if the actual point is no singularity (except turning points):

```
Edge e;
RealVec y;
e = ActualEdge();
if (e != nil) {
   if (Regular(e->p1)) {
      y = e->p1->y;
   } else {
      y = e->p0->y;
   }
 }
```

You can look at the `PrintActualPoint` procedure in the file `alcon.c`, if You want to treat the general case.

Bugs: Although You may call `InitConnection` twice with different `names`, only one will be installed at a time; the second overrides the first without delocation its objects correctly. Beside of the considerable loss of space this may confuse Alcon-S.

Alcon-S always assumes per default, that the print file's name is `./Application/`*problem*`.net`, the data file's name `./Application/`*problem*`.dat` and the graphic's file `./graphic/`*problem*`.dat`, where the dot is the directory, from which Alcon-S or the main program is started. If there is no `./Application/` or `./graphic/` directory, Alcon-S cannot find these files; moreover it will abort, if it must write to one of these files. After startup with `InitAlcon` and `InitConnection`, You can find these values as `char*` variables in the global variable `gOutput`:

Name	Variable
Print file	`gOutput->printFile`
Graphic file	`gOutput->graphicFile`
Data file	`gOutput->dataFile`

where You can change them to Your special needs. If You call the `?` problem menu, it will restore the default values on leaving, which is a at now uncorrected software misbehaviour.

If You want to change the defaults from the very beginning, so the programm reads the defaults from the right data file, You must change the sources. In the file `Application/connection.c` there is a procedure `ResetOutput`, where the default values are set.

6 Defining Your Own Solver Method

To define Your own solvers for Alcon-S is somewhat more tricky than defining a problem, but the author hopes, that it may occure less often. Here it is only discussed how to install Your solver for pathfollowing and gives a simplified method for bifurcations. To use Your solver with the block meta solver and the "regular" solver for bifurcations, You should read the documentation to know more about the concept on computing the pseudo inverse derived from a special decomposition scheme to understand the meaning of the extended object.

To give an idea of the working of the objects, which is necessary to understand the following installation steps, it is described, how Alcon-S constructs a solver. Because standart C does not much support object oriented programming there is the special consideration, that the instance variables and the methods of an object are stored in different structures. Each method gets the structure of the variables passed as an argument to know them.

The object responsible for nearly all actions on linear algebra (except for inner products) is called `LinAlg` and is therfore in the directory `LinAlg/`. If Alcon-S constructs an instance of this object, then it first calles an allocation routine, which constructs the structure of the variables, called `LinAlgData`. The program sets some of the variables, especially `type`, which is the type of the `LinAlg` to be constructed and defines the method of decomposition to use. Consequently it is of type `LinAlgType`.

After this preparation it calles the general constructor `NewLinAlg`. This procedure first checks the given `LinAlgData`, then depending on `LinAlgData->type` it calles the specific constructors and finally presets some of the unused methods with dummy routines, for Alcon-S may not abort if it calles them.

The destruction follows just this way back. After a special destructor called by `ClearLinAlg` destroys the `LinAlg` methods, the `LinAlgData` can be delocated or used to construct a new `LinAlg`.

To use the `LinAlg` objects as a standalone module to solve Your own problems, You must go this way, too.

6.1 The Solver for Pathfollowing

Before discussing the several methods to be defined, let us first look at the instance variables. As mentioned before, they are stored in the data structure `LinAlgData`.

The Variables Some of the variables are to be set by the calling program before constructing the object `LinAlg`, others have default values but may be changed before construction, while others contain results of the decomposition and are to be set by the methods after the object is constructed.

If the calling program changes some of the data after the construction of the object (except of `type` which should never be changed directly), it is usually necessary to call the `ResetLinAlg(LinAlg*, LinAlgType, Generic)` routine. The second argument is the new type of the `Linalg` if You want to change it. Always take `nil` as the generic pointer argument, because this pointer is only internally needed.

You can change the contents of the `LinOp` containing the matrix to be decomposed without reseting the object.

variables which are to be set	
`LinOp T`	linear operator of the equations to be solved
`Int m,n`	dimensions of the system of equations
`LinAlgType type`	type of the decomposition
variables which can be set	
`Euclid domainScale` `imageScale`	objects containing the scaling of the equations
`Real eps`	needed accuracy of the computation
`Bool fullRank`	flag for the decomposition not to be rank defect
`Bool iteration`	flag to improve the solution by iterative refinement
`Bool pm, pw`	print messages and warnings
variables to be set by the methods	
`Int p`	rank of the decomposition
`Real test`	a test function for detecting bifurcations
`Real cond`	a condition number estimate
`Real norm`	a norm estimate

Note, that the `LinOp T` has its own dimensions `m,n` (see the definition of the `LinOp` on page 21). If they are larger than the `m,n` of the `LinAlgData`, then only the smaller part of the LinOp, defined by the first `m` rows and `n` columns should be treated by the `LinAlg`.

The variable `eps` has two different meanings. While the decomposition phase it is the accurary of the input data. A rank reduction should be made, if the condition of the matrix grows larger than `1.0/eps`. While the solution phase `eps` is the accuracy of the output, i.e. the accuracy of the solution to be computed.

It is in the responsibility of the methods to use or ignore the flags set by the user. For examble the contends of the `iteration` flag is of no meaning to an iterative solver.

In contrast the variables, which have to be set in the decomposition phase by the methods, must contain reasonable values. Most important is the `test` variable. The pre-installed solvers of Alcon-S always use the determinant as test function for detecting bifurcations. The determinant of a $(m, m+1)$ matrix A with respect to a given (normalized) tangent vector is defined here as

$$\det(A) := (-1)^i \frac{\det(A^i)}{t_i}$$

where t_i is the i-th component of the tangent vector and A^i is the (m, m) matrix constructed from A by omitting the i-th column. In the documentation it is proved, that the definition is independent of i, as long as only $t_i \neq 0$.

The `test` variable must have the property, that it changes sign along a path, when this path crosses a simple bifurcation or when the orientation of the tangent is changed. If You cannot support such a test function, You may return a constant (better nonzero) value, but You must call Alcon-S with the `!s@` flag off. Alcon-S will "detect" bifurcations randomly, but will not try to compute them.

The `cond` and `norm` variables are of lesser interest and may replaced by dummy values without further harm.

The objects `domainScale` and `imageScale` implements the inner products used by Alcon-S. Though they are much simpler, they work similar than the `LinAlg`. You must not worry about their con/destruction, because either Alcon-S or the general constructor of the `LinAlg` cares about. If You decompose a matrix, You may use the help routines for scaling a matrix with given `Euclid`'s which can be found in the `LinAlg/Euclid.h` file. Methods for scaling vectors are supported by the `Euclid` objects.

The Methods After Alcon-S has set the `data` to it's aims, it calles the general constructor `NewLinAlg`, which calles in turn the solver specific constructors. The constructor gets the data as a parameter and returns the complete object `LinAlg`. The constructor must allocate the `LinAlg` and may also allocate some extra space needed for the internal details of the solver. This extra space may be stored in the `Generic data` variable of the `LinAlgData`.

The variable `data` of the `LinAlg` indeed must point to the `LinAlgData` passed to the constructor. This variable is passed to each method as the first argument, if it is called, so the method may know it's own variables. For these methods, the importand ones for pathfollowing are listed below; not initalized methods should be set to `nil`. The general constructor will support dummy procedures for these methods.

Corresponding to the specific constructor, there must be a solver specific destructor, who delocates not only the `LinAlg`, but also the problem specific data stored in `LinAlg->data->data`. It may *not* delocate the `LinAlg->data`, however.

Beside of the `LinAlg` con/destructors, You must define these methods for the `LinAlg` as prototyped (the first argument of the `LinAlgData` is omitted for shortcuts):

Name	Purpose
`Prepare()`	computes the decomposition of the LinOp `data->T`; always called before any solve operations.
`Solve(b,x)`	solves the decomposed system with right hand side `b` and stores the result in `x`
`KernelBasis(t)`	stores a normalized tangent vector in `t[1]`
`NewRank(p)`	changes the preset rank `data->p` to the value of `p`

The `Prepare` method may return false, if any error occures, especially if the computed rank is not the value that `data->p` had before the decomposition. The `Solve` also may return false if some error occures, i.e. the iterative refinement failes.

The `Solve` method must not compute the Moore Penrose pseudo inverse for pathfollowing. Any other right inverse may also applied, but keep the `!se` switch off in this case.

The `NewRank` method is meant to change the `data->p` from it's default value (min(`data->m`, `data->n`)); in the context of pathfollowing this is only used to set the rank back to it's maximal value, if some rank deficency is detected along the path, leading to a smaller `data->p`.

Additionally You can define the following methods, if they are useful to You:

Name	Purpose
`SolverSetUp()`	an interactive menu for setting special options of the solver
`SaveToFile(File)`, `LoadFromFile(File)`	store and load special options to a file
`ResetSolver(Generic*)`	housekeeping routine calles by the `ResetLinAlg` after the `data` is changed
`ForgetHistory()`	forget data possibly stored from previous decompositions which are not useful here
`SaveData(Generic*)` `LoadData(Generic*)`	save and reload data stored from previous decompositions to memory pointed to by the `Generic*`

The error return code `eFileError` of the `SaveToFile` and `LoadFromFile` can be found in the file `mystd/fileio.h`. All other methods return `void`. For the `ResetSolver` method ignore the `Generic*` parameter. because this is only needed for some internally dirty tricks.

The `SaveData` is useful, if the continuation is restarted from the starting point in the other direction and there exist some data (pivot strategies, singular vectors or such) which the solver keeps along the path. A pivot strategy to compute the final boundary point far away from the starting point may not be useful for pathfollowing near the starting point. Normally `ForgetHistory` would be called, if not the pivot strategy could be saved with `SaveData` at the starting point, where it can be reloaded with `LoadData`. The space needed to store the data must be allocated by `SaveData` itself. To avoid memory leaking, this memory should be allocated by only one call to `malloc`.

Note: None of the build in solvers of Alcon-S changes the LinOp stored in `data->T`. Instead of this, the operator is copied to internal memory and this copy is decomposed. The original LinOp is kept for iterative refinement. Your solver may use the LinOp to store the decomposition, but in the case of a sparse operator, either it may not change the sparsity structure or the problem to be solved is responsible to restore the sparsity pattern each time the `df` method is called.

Installation Steps: After the most necessary variables and methods are discussesd in the previous paragraphs, here follows a step by step list to define Your solver *mySolver*:

- Define the con/destructors and the methods of Your solver (which name is assumed to be *mySolver* here) as described above in a new file `mysolver.c`
- prototype the con/destructors in a file `mysolver.h`. (You can also #typedef the internal data structures of the solver for convenience.)
- #typedef `e`*mySolver* in the file `LinAlg.h` as a `enum` case of the type `LinAlgType` anywhere *behind* the `eBlockMat`.
- In the file `LinAlg.c` #include the `"mySolver.h"` and call Your constructor in the procedure `NewLinAlg` inside of the `switch` branch, where the solver specific constructors are called. Also add a call to the destructor in the procedure `ClearLinAlg`.
- If You wish to set Your solver interactively in the LinAlg menu, You must append the name of Your solver to an array of strings containing the names of the preinstalled solvers. This string variable has the name `Methstr` and can be found slightly above the `ChooseMethod` procedure. After this step You must expand the `ChooseMethod` and the according `ChooseList` procedure, which contains the display of all choosable solvers.
- Add the `mySolver.o` file to the `Makefiles` of the following modules:

  ```
  $BASE/LinAlg/
  $BASE/GNMethod/
  $BASE/Application/
  $BASE/Contin/
  $BASE
  ```

  and recompile these modules.

6.2 Expanding the solver for bifurcations

To define a solver for the computation of bifurcations it is normally necessary to define all special methods of the `LinAlg` to implement the decomposition scheme of the documentation, chapter 3. However, there is a shortcut in connection with the deflation (`!msd`) option of the system of Moore. If this flag is on, then a much simpler, but possible less stable solver is applicable effectively by only adding two methods to the `LinAlg`.

The variables do not need additional interest except for the `cond` variable, who must contain some condition estimate having the asyptotic behaviour near a bifurcation, that

$$\mathrm{cond}(f') \sim 1/s$$

where s is the distance to the bifurcation. If this cannot be supplied, a constant positive value should be returned in `cond`.

The methods need more consideration. The `NewRank` now must also handle the case, that the choosen rank can be smaller than the maximal value. The `Prepare` method must compute a decomposition of the new rank `p`, even if the matrix to be decomposed is not rank defect; in this case some parts of the matrix must be neglected to force the rank defect. The `Solve` method now must compute the Moore Penrose pseudo inverse. The `KernelBasis` must return an orthonormal basis of the two dimensional tangent space in `t[1],t[2]` in the rank deficient case. Beside of these additional features of already defined methods, the following new methods must be supplied:

`SolveAd(b,x)`	solves the adjoined equation $A^*x = b$
`KernelAdBasis(t)`	returns a tangential vector of the adjoined matrix in `t[1]` in the rank deficient case

If You set the `Transform`xx methods of Your `LinAlg` to `nil` (or at least one of them), then the solver for the system of Moore will recognize this and compute the bifurcation, as if the `!msd` option would be switched on.

To use Your solver in connection with the block meta solver of Alcon-S, there is no such shortcut, sorry. You will have to read the documentation in this case.

7 Some Remarks about Graphic

Alcon-S does not provide any parts of the program that contains graphical representations of it's results, because those parts would usually be heavy machine dependent. Instead of this some MATLAB© scripts are included in the subdirectory `graphic/`. The scripts, their function and the needed input format provided by Alcon-S are discussed here.

The formats Alcon-S provides three output formats, depending on which kind of information is needed by the graphic. To each one format corresponds exactly one command in the `write` menu. The formats are:

Format	Purpose	Menu
complete	view projections of the complete diagram	`g`
actual	view the actual computed point	`a`
whole	view the whole diagram in all coordinates	`v`

The kind of projection for the complete format is set by the `wo` menu.

The scripts The MATLAB© scripts apply to different formats, as there are

Script	Purpose	Format
linear	plot a display of the whole diagram by linear interpolation	complete
cubic	plot a display of the whole diagram by cubic hermite interpolation	complete
scape	plot a display of the solution, interpreted a 1D-finite difference solution with equisistant nodes	whole
stabbild	plot an actual solution as a rod construction	actual
torus	plot an actual solution as two valued function on a torus	actual
contur	same as torus, but using another perspective	actual
flow	plot an actual solution as a flow diagramm on a torus	actual
poincare	plots a Poincaré cut of the tori along one coordinate axis	actual

The stabbild is only useful for graphic made by stab*constr* problems and the last four are only useful in connection with the oscillator example.

The stabbild also assumes, that there is a *constr*.crt file, which is made by Alcon-S automatically when first running the problem and a *constr*.ptr made by Alcon-S, if You let it run one jacobian evaluation with the evaluation mode set to numerical, which can be done without problems at the inital point normally.

If You write Your own graphic routines, You may want to know the formats in detail; otherwise this section will be of no interest to You.

- The complete format contains lines with ten numbers on each line. While the first line is special, the following lines each represent an edge connecting two solution points. The format of the first line is

  x-coordinate y-coordinate parameter-index

  following seven NaN's which are ignored. The parameter index is $m+1$; normally x-coordinate is the same as $m+1$. If y-coordinate $= 0$ then the norm of the variables was dumped instead of a special coordinate.

  The following lines are of the format

  x1 y1 type1 x2 y2 type2 tx1 ty1 tx2 ty2

  where

x1, y1	x,y-coordinate of the one point
type1	type of the one point
x2, y2	x,y coordinate of the other point
type2	type of the other point
tx1, ty1	tangent at the one point along the edge
tx2, ty2	tangent at the other point along the edge

  The tangents are always oriented in direction of continuation. The type is encoded as an integer number by the following code

0	unused
1	regular point
2	turning point
3	bifurcation
4	boundary
5	dead end
6	only approximative solution
7	inital point

- The actual format is only one line of text, containing first the number of coordinates $m+1$, then the type of the point according to the above listed code followed by the coordinates.
- The complete format contains a list of points, one per line, where the first number on each line is the type followed by the coordinates of the point. No tangential information is given.

8 Copyright Notice and No Warranty Declaration

Any similarity to other programs, living or dead, is more or less coincidental.