FLIPS Manual
for version 2.0

Written by Mikko Orispää
mikko.orispaa@oulu.fi

FLIPS (Fortran Linear Inverse Problem Solver)

Contents

I. FLIPS Implementation

1 Introduction

FLIPS (Fortran Linear Inverse Problem Solver) is a Fortran 95 module for solving large scale linear inverse problems of the form

$$m = Ax + \varepsilon, \tag{1.1}$$

where m is a N-vector called *the measurement*, A is a $N \times M$-matrix called *the direct theory matrix*, x is a M-vector called *the unknown* and ε is (an optional Gaussian) N-vector called *the error*.

→ See section 2.1

FLIPS solves the problem by transforming it into a simpler form which is then easy and fast to solve. Namely, the equation (1.1) is transformed into the form

$$Y = R \cdot x, \tag{1.2}$$

where R is a $M \times M$ upper triagonal matrix (called *the target matrix*), and Y and x are M-vectors. Vector Y is called *the target vector*. Equation (1.2) is numerically fast to solve using backsubstitution.

→ See chapter 3

Apart from just solving the linear problems, FLIPS is also able to marginalize away and add unknowns to the linear problem without affecting the remaining unknowns or their posteriori covariance.For real-valued problems it is also possible to delete data that has been already fed into the problem.

→ See section 3.2
→ See section 3.3

FLIPS is able to use either the computer memory or binary files to store the problem data. Using binary files is slower than using memory, but makes it possible to solve very large problems. Also the total memory consumption is kept as low as possible. Only the target matrix and vector are stored. The direct theory matrix, measurements and errors can be fed into FLIPS in small chunks or even row-by-row. They are not needed afterwards and can be discarded.

The structure of this manual is the following:

In Chapter 2 the general linear inverse problem is defined and its formal solution is given.

In Chapter 3 the implementation of FLIPS is discussed and the way FLIPS solves the problems is explained.

The second part of the manual consists of the reference manual, in which the FLIPS commands are explained. Also, some simple example programs are given and explained.

The third part of the manual consists of technical notes and instructions of how to compile the FLIPS module and use it in user's own Fortran programs.

2 General Linear Inverse Problem

2.1 Definition

We define here the *real valued* inverse problem. The complex one is defined accordingly.

A general linear inverse problem can be presented as a matrix equation

$$m = A \cdot x + \varepsilon, \tag{2.1}$$

where $m \in \mathbf{R}^M$, $A \in \mathbf{R}^{M \times N}$, $x \in \mathbf{R}^N$ and $\varepsilon \in \mathbf{R}^N$. Here (and from now on) m is called the measurement, A is called the direct theory matrix, ε is called the error and x is called the unknown. The problem is to find either the exact or best possible value for the unknown x.

Depending on the integers M and N and the characteristics of the error ε, there are different possibilities:

If $M = N$ and the values of m and ε are known (in this case we can put $\varepsilon = 0$ without loss of generality), we have a linear system with N equations. If A is invertible, i.e. $\det(A) \neq 0$, the system has an unique solution.

If A is singular, i.e. $\det(A) = 0$, its kernel is non-zero. In this case m either is or is not in the range of A. If $m \in \mathrm{Ran}(A)$, then the system has more than one solution. However, it is possible to find the shortest vector in the solution space. On the other hand, if $m \notin \mathrm{Ran}(A)$, the system does not have an exact solution at all. In this case, it is possible to give "solution" in the least squares sense, i.e., as the vector $\hat{x}$ that minimizes the norm $\|A \cdot \hat{x} - m\|$.

If $M < N$ and the values of m and ε are known (again, we can put $\varepsilon = 0$), we have a linear system with less equations than unknowns. In other words, the system is underdetermined. This problem can be reduced to the previous one by expanding matrix A and vector m with $N - M$ zero rows and elements, respectively. The resulting matrix A will be singular, and the type of the approximate solution depends again if m is in the range of A or not.

If $M > N$ and the values of m and ε ($\varepsilon = 0$) are known, the system is overdetermined, unless A is row-degenerate in such a way that the system reduces to one of the above ones. Overdetermined system has the solution only in the least squares sense.

If m is known and ε is not, but its components are random variables with a known joint distribution function, we have what will be called *a linear stochastic inverse problem*. In this case, the solution is given by the joint posteriori distribution of the unknowns. However, usually it is enough to obtain the *maximum a posteriori estimate (MAP)* and the posteriori (co)variances of the unknowns.

FLIPS is able to solve all above problem types with Gaussian errors. It is able to give the MAP estimates of the unknowns and their posteriori covariance matrix. Note however that solving underdetermined problems with FLIPS requires some extra work from the user part, as some kind of regularization scheme is needed.

2.2 Formal Solution

Without (too much) remorse, we skip most of the mathematical details in what follows, and refer to standard mathematical and stochastic literature[1].

As defined in the last section, the general stochastic linear inverse problem is to find "the best possible" solution to the equation

$$m = A \cdot x + \varepsilon,$$

where matrix A is a known direct theory matrix, and the components of the vectors m, x and ε are random variables. Broadly speaking, this means finding out what kind of information about x can be obtained when the measurement m is fixed and the (statistical) distribution of ε is known.

In the most general sense, the solution is given as the *posteriori density* of x, which is the conditional density of x when the measurement m is fixed. For this, it is necessary to know the distribution function of the error ε. Estimations of error variances can be obtained, for example, by repeating the measurement several times. Then if the mean values of the measurements are used as m, the density of errors ε can be approximated by Gaussian distribution (using the central limit theorem).

If no *a priori* information about x is available, the posteriori density is given by formula

$$\pi(x|m) = C \exp\left\{ -\frac{1}{2} (m - A \cdot x)^T \cdot \Sigma^{-1} \cdot (m - A \cdot x) \right\},$$

where C is a normalization coefficient and Σ is the *error covariance matrix*, defined by

$$\Sigma = \langle \varepsilon \cdot \varepsilon^T \rangle.$$

It is (quite) easy to show that the above posteriori density can be written as

$$\pi(x|m) = C \exp\left\{ -\frac{1}{2} \left[(x - x_0)^T \cdot Q \cdot (x - x_0) + \chi^2 \right] \right\},$$

where

$$\begin{aligned} Q &= A^T \cdot \Sigma^{-1} \cdot A, \\ x_0 &= Q^{-1} \cdot A^T \cdot \Sigma^{-1} \cdot m, \\ \chi^2 &= m^T \cdot \Sigma^{-1} \cdot m - x_0^T \cdot Q \cdot x_0. \end{aligned}$$

The matrix Q above is called the *Fisher information matrix*, and it is (as is Σ) positively definite and symmetric. Hence, the exponential function gets it maximum value $-\chi^2/2$ at point $x = x_0$, so x_0 is the maximum point of the posteriori density. The width of the distribution is determined by the inverse of Q, $Q^{-1} = \Sigma_x$, which is called the *posteriori error covariance matrix*. We can therefore say that the solution to the original inverse problem is given by x_0 and Σ_x. The scalar (or vector) χ^2 is the smallest value (at $x = x_0$) of the quadratic form $(m - A \cdot x)^T \cdot \Sigma^{-1} \cdot (m - A \cdot x)$, and it is called the *residual of the solution*, since it gives some information how well x_0 approximates x.

If, in addition, we have *a priori* information of the unknown x which can be approximated by a Gaussian distribution

$$\pi(x) = C_0 \exp\left\{ -\frac{1}{2} (x - \bar{x}_0)^T \cdot \Sigma_0^{-1} \cdot (x - \bar{x}_0) \right\},$$

[1] See, for example, Kaipio & Somersalo, *Statistical and Computational Inverse Problems*, Springer, 2004.

where $\bar{x}_0$ are the prior values and Σ_0 is the prior covariance matrix of x, they can be added easily to the formal solution formulae, namely,

$$Q = \Sigma_0^{-1} + A^T \cdot \Sigma^{-1} \cdot A,$$
$$x_0 = Q^{-1} \left(\Sigma_0^{-1} \cdot \bar{x}_0 + A^T \cdot \Sigma^{-1} \cdot m \right).$$

3 FLIPS Implementation

3.1 FLIPS algorithm

The formal solution presented in the previous section could be used to solve the linear inverse problem. However, if the matrix A is large, the process involves inversions of large matrices and soon becomes unpractical. FLIPS uses different approach, and utilizes a sequence of elementary rotations to transform the problem into a simpler one.

First step is to "eliminate" the error covariance matrix by transforming the original problem in such a way that the error covariance is transformed into a unit matrix[1]. Note that then it can be discarded from the formal solution formulae. This is done in the following way: Covariance matrix Σ is real symmetric and positive definite, so it can be decomposed using Cholesky decomposition, i.e.,

$$\Sigma = C \cdot C^T,$$

where C is a lower triangular matrix. By setting the first transformation matrix

$$U_0 = C^{-1},$$

the transformed equation will be

$$m_0 = A_0 \cdot x + \varepsilon_0,$$

where $m_0 = U_0 \cdot m$, $A_0 = U_0 \cdot A$ and $\varepsilon_0 = U_0 \cdot \varepsilon$. Then the transformed covariance matrix will be

$$\begin{aligned}
\Sigma_0 &= \langle \varepsilon_0 \cdot \varepsilon_0 \rangle \\
&= \langle C^{-1} \cdot \varepsilon \cdot (C^{-1} \cdot \varepsilon)^T \rangle \\
&= C^{-1} \cdot \langle \varepsilon \cdot \varepsilon^T \rangle \cdot (C^{-1})^T \\
&= C^{-1} \cdot \Sigma \cdot (C^T)^{-1} \\
&= C^{-1} \cdot C \cdot C^T \cdot (C^T)^{-1} \\
&= I,
\end{aligned}$$

where I denotes the unit matrix.

! → Note that the current version of FLIPS can only use diagonal error covariance matrices[2]. In this special case the Cholesky decomposition is simply

$$\Sigma = \Sigma^{1/2} \cdot (\Sigma^{1/2})^T.$$

After this, the new theory matrix A_0 is transformed into a upper triangular one by using so called *Givens rotations*. Givens rotations can be used to zero single

[1] This is also called "the whitening of the noise" as normally distributed noise with unit covariance matrix is usually called "white noise".

[2] The R and MATLAB interfaces for FLIPS can handle also non-diagonal error covariance matrices. Fortran user might want to do this by hand using linear algebra libraries like LAPACK or ATLAS. This will probably change in the future versions of FLIPS.

elements in a matrix. It is carried out by using a transformation matrix of the form

$$\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & i & & j & \end{array} \\ \begin{pmatrix} 1 & \vdots & & \vdots & \\ \cdots & \cos(\phi) & \cdots & \sin(\phi) & \cdots \\ & \vdots & 1 & \vdots & \\ \cdots & -\sin(\phi) & \cdots & \cos(\phi) & \cdots \\ & \vdots & & \vdots & 1 \end{pmatrix} \end{array},$$

where all diagonal elements except elements (i,i) and (j,j) are ones, and all off-diagonal elements except elements (i,j) and (j,i) are zeros. This is a rotation matrix that rotates the i,j-coordinate plane by an angle ϕ. Let us denote this matrix as U_{ji}.

To zero out the (j,i)-element of the theory matrix A (let us denote it by a_{ji}), we apply the rotation U_{ji} with angle ϕ calculated from the formulae

$$\cos(\phi) = \frac{a_{ii}}{\sqrt{a_{ii}^2+a_{ji}^2}} \tag{3.1}$$

$$\sin(\phi) = \frac{a_{ji}}{\sqrt{a_{ii}^2+a_{ji}^2}}. \tag{3.2}$$

Note that since only $\cos(\phi)$ and $\sin(\phi)$ are needed, there is no need to actually solve the angle ϕ from the above equations. After the transformation, the resulting matrix, say A_t,

$$A_t = U_{ij} \cdot A, \tag{3.3}$$

will have a zero as the (j,i)-element. If it happens that $a_{ii} = a_{jj} = 0$, the rotation is just skipped. Note that also the measurement vector m must be transformed using the same rotation matrix U_{ji}.

! → The noteworthy thing about the above transformation is that only the i'th and j'th rows of the matrix A are affected. The other rows will stay intact.

The sequence of the rotations FLIPS performs is the following:

Let us assume that the theory matrix A has N columns, in other words, there are N unknowns in the linear system.

1. The first row of the direct theory matrix A is moved as the first row of the target matrix R.
2. The second row of A is then rotated with the first row of R. After the rotation, the first element of the second row of A is zero. The rotated row is moved as the second row of R.
3. The third row of A is first rotated with the first row of R, which will zero out the first element of the third row. After this the rotated third row is rotated again with the second row of R, which will zero out the second element of the third row. Now the first two elements are zero. It is then moved to the third row of R.
4. The consequent data rows are rotated in the similar manner starting from the first row of R, and they are afterwards moved to the target matrix R.
5. The N^{th} row of A will have only one non-zero element after the rotations, and it will be moved to the last row of R. The target matrix R will now be upper triangular.
6. If there are still more theory matrix rows (i.e. the system is overdetermined), they are rotated with all rows of R. After the rotations, these rows will contain only zeros, and they are discarded.

! → Note that also the measurement vector elements are rotated in the same way as the theory matrix rows resulting what is called the target vector Y.

After all the rows of A are rotated into the target matrix and target vector, we are left with an equation

$$Y = R \cdot x,$$

where Y is a N-dimensional vector and R is a $N \times N$ upper triangular matrix. This equation is is easy to solve using backsubstitution.

! → Note that if there are less equations (rows) than unknowns (columns) in the theory vector A, the resulting R will be degenerate. If one wants to solve underdetermined problems with FLIPS, some kind of regularization is required!

! → Note also, that the problem data can be entered into FLIP row-by-row (or as many rows at the time as required) and that the data is not needed anymore after the rotations are made. Also, FLIPS only stores target vector Y and target matrix R which are always of fixed size (depending only on the number of unknowns).

3.2 Marginalizing unknowns

Another powerful property of FLIPS is the ability to marginalize away any number of unknowns without affecting the remaining unknowns or their posteriori covariance. It is also possible to add new unknowns to a system at any time.

It is fairly easy to show[3] that it is possible to remove rows and columns from the target matrix without affecting the remaining problem as long as they are removed beginning from the upper left corner. For example, if we have a problem with 20 unknowns and we want to marginalize the 5 first unknowns from the problem, we can just discard first 5 rows and columns from the target matrix, and the remaining 15 unkowns together their posteriori covariances are not affected.

It is also possible to marginalize away unknowns which are not at the beginning (index-wise). In this case FLIPS permutates the target matrix columns so that the unknowns to be marginalized are moved to the left. FLIPS then uses resulting matrix (which is no more upper triangular) as a new theory matrix which it rotates again to get an upper triangular target matrix with unknowns to be marginalized in the left. After this, the rows and columns can be removed.

It is also possible to add new unknowns at the end (index-wise) of the target matrix. In this case, FLIPS just adds new columns and rows (containing zeros) to the target matrix.

! → The marginalizing and/or adding unknowns can be done at any time.

3.3 Deletion of data

For real-valued[4] problems FLIPS allows the deletion of already fed-in data rows. This is implemented by what we call *the antirotations*. Antirotations are a modification of Givens rotations and they are defined as follows:

$$\begin{pmatrix} c_a & -s_a \\ -s_a & c_a \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}, \tag{3.4}$$

[3] I will get back to this!

[4] This might be possible also for complex-valued problems. However, it needs further research.

where the antirotation coefficients $c_a \in \mathbb{R}$ and $s_a \in \mathbb{R}$ are determined by the equations

$$\begin{cases} c_a^2 - s_a^2 & = 1, \\ g c_a - f s_a & = 0, \end{cases} \tag{3.5}$$

and $f, g, r \in \mathbb{R}$. For consistency with real Givens rotations, we choose

$$c_a = \frac{f}{\sqrt{f^2 - g^2}}, \tag{3.6}$$

$$s_a = \frac{g}{\sqrt{f^2 - g^2}} \tag{3.7}$$

and

$$r = \sqrt{f^2 - g^2}. \tag{3.8}$$

Note that we have to assume that $|f| > |g|$ in order to $c_a, s_a, r \in \mathbb{R}$. This restriction is, however, always fulfilled if we only try to delete data that have been previously fed in, as seen below.

To show that antirotations delete the data previously fed in, first apply the Givens rotation $G(1,2,\theta)$ to a real vector $F = (f\ g)^T$, $f, g \in \mathbb{R}$, $f, g \neq 0$, i.e.

$$G(1,2,\theta)F = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}, \quad r = \sqrt{f^2 + g^2}. \tag{3.9}$$

If we now want to delete the effect of g in the rotated element r, we apply the antirotation to the vector $G = (r\ g)^T$ (note that always $|r| > |g|$), which yields

$$\begin{pmatrix} c_a & -s_a \\ -s_a & c_a \end{pmatrix} \begin{pmatrix} r \\ g \end{pmatrix} = \begin{pmatrix} \tilde{r} \\ 0 \end{pmatrix}, \tag{3.10}$$

where

$$\tilde{r} = \sqrt{r^2 - g^2} = \sqrt{\left(\sqrt{f^2 + g^2}\right)^2 - g^2} = |f|. \tag{3.11}$$

Losing the sign of f above does not matter, since we want to keep the diagonal of the target matrix R positive.

Antirotations are implemented in FLIPS by making the antirotations between the rows of the target matrix R and the data row to be deleted. This adds another limitation to the use of antirotations: the target matrix must be a full upper-triangular matrix before deleting any data row from it. In other words, the FLIPS problem must be overdetermined before deletion.

3.4 Memory and file storages

By default, FLIPS uses the computer memory to store the data needed (target matrix and vector, for example) to solve the problem. However, it is also possible to use the computer hard disk to store this data, if so required. Using file storage is considerably slower, so it shouldn't be used unless really needed, for example, if the problem does not fit in the computer memory, or there is a need to save the problem data for future reference. Note however, that it is possible to change the data storage method at any time.

! → The binary file storage was originally written using Stream I/O extension of Fortran 2003, which some of the current Fortran 90/95 compilers support. If the used compiler does not support Stream I/O extension, it is advisable to avoid using binary file storage as the alternative file I/O (Direct Access) tends to be extremely slow on most platforms!

→ See Section 8.2

! → The band matrices and common variables are not implemented for file storage problems (for time being)!

3.5 Band matrices

In many problems (like deconvolution, finite-difference method, etc.) the theory matxi A is not full, but more or less band-structured. In these cases, also the resulting target matrix is band-structured. For large problems this would mean of storing a huge number of zeros as FLIPS stores the upper triangular part of the target matrix. However, the user has the option to use band-structured target matrix. If the maximum band of the theory matrix is known, it can be given to the FLIPS when the problem is initialised. If it is not known in advance, the user can give any band size and FLIPS will resize itself if the given size is too small. This resizing is only done on-demand and for large problems it can decrease the speed performance of FLIPS. On the other hand, it can save huge amounts of memory, if the maximum band-size of the theory matrix is relatively small compared to the total number of unknowns.

3.6 Common variables

There are lot of problems, where the theory matrix is "almost" band-structured but there are some unknowns that contribute to many of the single measurements. In other words, the theory matrix is of form

$$A = \begin{pmatrix} B & F \end{pmatrix}, \tag{3.12}$$

where B is a band-structured matrix and F is a full matrix with (relatively) small number of columns. In FLIPS the unknowns corresponding to the columns of the matrix F can be declared as *common variables*. They are handled separately from the unknowns corresponding to the band-structured part B. This saves computer memory as the excess zeros are not stored, and improves the performance as the trivial zeros are not rotated with each other. The number of common variables can be given when the FLIPS problem is initialised. The number of common variables can not be changed directly after the initialisation.

II. Command Reference

4 Global Variables

FLIPS has some global variables, although their number is tried to be kept as small as possible.

FLIPS Global Variables

NUMERICAL PRECISION

`sp` : Integer; holds the single precision kind number

`dp` : Integer; holds the double precision kind number

! → The single and double precision sizes can vary on different platforms. Hence, it is advisable to use above kind parameters when declaring real and complex variables. This way they are assured to be compatible with FLIPS.

MISCELLANEOUS VARIABLES

`defbufsize` : Integer; default buffer size. Currently this is set to 100. *May change or be omitted in the future!*

`verbose` : Logical; default is .FALSE.. This is an experimental flag for printing extra info on standard output when using FLIPS. Used mainly for debugging. *Not fully implemented!*

5 FLIPS Structured Data types

FLIPS module defines four structured data types. Every problem needs its own data type, although they can be reused if wanted. FLIPS data types contain all the information that FLIPS needs to solve the given problem.

The defined data types are:

`flips_s` : Data type for single precision real problem

`flips_d` : Data type for double precision real problem

`flips_c` : Data type for single precision complex problem

`flips_z` : Data type for double precision complex problem

The structure of `flips_s`

Other data types are defined accordingly with obvious changes.

FLAGS Logical variables, mainly used internally

`cplx` : Logical; flag for real/complex (obviously .FALSE.)

`dbl` : Logical; flag for single/double precision (.FALSE.)

`use_files` : Logical; flag for memory/binary file storage

`solexists` : Logical; flag for the existence of the solution

`rinverted` : Logical; flag for the existence of the inverse of target matrix R

`covexists` : Logical; flag for the existence of the posteriori covariance matrix

`fullcov` : Logical; flag for diagonal/full posteriori covariance matrix

PROBLEM SIZE Integer variables keeping track of the problem size

`nrotbuf` : Integer; number of rows in the rotation buffers

`ncols` : Integer; number of unknowns

`nrhs` : Integer; number of columns in the measurement matrix

`nbuf` : Integer; number of rows currently in the rotation buffers

`nrows` : Integer; number of (rotated) rows in the target matrix R

`bw` : Integer; current R matrix bandwidth

`bbw` : Integer; current maximum bandwidth of the rotation buffer

`common` : Integer; number of common variables

MEMORY STORAGE Real arrays used to hold the data when memory storage is used[1]

`rmat` : Real(single), dimension(:); real vector holding the target matrix R stored in row-major format

`cvmat` : Real(single), dimension(:); real vector holding the common variable part of the target matrix (if common variables exist)

[1] Rotation buffers and residual calculation vectors are used also when using binary file storage.

`ymat` : Real(single), dimension(:); real vector holding the target vector Y

`arotbuf` : Real(single), dimension(:); real vector containing the theory matrix rotation buffer

`mrotbuf` : Real(single), dimension(:); real vector containing the measurement data rotation buffer

`invrmat` : Real(single), dimension(:); real vector containing the inverse of R (after it is calculated) in row-major format

`cmat` : Real(single), dimension(:); real vector containing the upper triangular part posteriori covariance matrix *or* the diagonal of it (depending which one is calculated)

`solmat` : Real(single), dimension(:); real vector containing the solution (after it has been calculated)

`residual` : Real(single), dimension(:); real vector containing the residual(s) of the solution (calculated together with the solution)

`tmpres` : Real(single), dimension(:); real vector used for residual calculation (used internally)

BINARY FILE STORAGE These are used only if binary file storage is used

`idnum` : Integer; ID number of the problem. Used for constructing filenames.

`rfile` : Character string; holds the filename of target matrix R

`yfile` : Character string; holds the filename of target vector Y

`cfile` : Character string; holds the filename of posteriori covariance matrix

`irfile` : Character string; holds the filename of the inverse of R

`solfile` : Character string; holds the filename of the solution vector/matrix

`rfileunit` : Integer; holds the unit number of target matrix file

`yfileunit` : Integer; holds the unit number of target vector file

`cfileunit` : Integer; holds the unit number of posteriori covariance file

`irfileunit` : Integer; holds the unit number of inverse of R file

`solfileunit` : Integer; holds the unit number of the solution vector/matrix file

MISC VARIABLES

`fc` : Real(single), scalar; variable holding the (estimated) number of flops (floating point operations) used by current problem. NB: Not reliable anymore!

`zeroth` : Real(single), scalar; zero threshold. Theory matrix elements whose absolute value is less than `zeroth` are treated as zeros.

`rst` : Integer, dimension(*ncols*); Indices of first non-zero elements in R matrix rows.

`rle` : Integer, dimension(*ncols*); Indices of last non-zero elements in R matrix rows.

`bst` : Integer, dimension(*nbuf*); Indices of first non-zero elements in rotation buffer rows.

`ble` : Integer, dimension(*nbuf*); Indices of last non-zero elements in rotation buffer rows.

Usually, these variables are not needed by the user, because FLIPS contains necessary routines to handle them.

! → Not all of the above variables are allocated when the data type is initialised:

- `cmat` and `invrmat` are first allocated when the posteriori covariance is calculated (memory storage).
- Binary files for posteriori covariance matrix and the inverse of the target matrix are created when posteriori covariance matrix is calculated (binary file storage).
- Binary file for the solution is created when the solution is calculated (binary file storage).

6 FLIPS commands

6.1 flips_init

initialises the FLIPS data type.

SYNTAX

```
call flips_init(ftype,ncol,nrhs,bandwidth,common,
                idnum,buffersize,zerothreshold)
```

ARGUMENTS

`ftype`	:	type(flips_<s|d|c|z>); FLIPS data type, uninitialised
`ncol`	:	integer; number of unknowns, i.e. number of columns in the direct theory matrix.
`nrhs`	:	integer; number of alternate measurements, i.e. number of columns in the measurement matrix.
`bandwidth`	:	integer, optional; initial bandwidth of the theory matrix. Default is *ncol*, i.e., the theory matrix is considered full. For bandwidth limited problems setting this argument low can lead to huge improvements in memory consumption. On the other hand, if the bandwidth is set too low, it will be automatically increased when necessary, which can lead to slower performance. Bandwidth argument can only be given for memory storage problems.
`common`	:	Integer, optional; number of common variables. Default is zero.
`idnum`	:	integer, optional; ID number of the problem. If given, binary file storage is used and binary files are identified by the ID number.
`buffersize`	:	integer, optional; number of rows in the rotation buffers. Default value is used, if this is omitted.
`zerothreshold`	:	real, optional. Threshold below which the elements in theory matrix are treated as zeros, i.e. all elements in theory matrix whose absolute value is less than `zerothreshold` are given to FLIPS as zero. Default value is 10^{-10} for both single and double precision.

RULES

Any FLIPS data type must be initialised before any data can be fed in it. FLIPS data type must be uninitialised state before calling flips_init. Otherwise, an error will occur.

→ flips_kill

If FLIPS data type is used before, it must be deallocated before it is initialised again.

FLIPS data type can be of type s (single real), d (double real), c (single complex) or z (double complex).

Bandwidth and common arguments are (at least for now) only implemented for memory storage problems. If they are set for a file storage problem, the

bandwidth will be set to the maximum (*ncol*) and common is set to zero. Also, a warning will be given.

EXAMPLES

1. initialise a single precision real problem with 500 unknowns, no alternate measurements and default rotation buffer size using memory storage. Theory matrix is full (i.e., no bandwidth (or common variables) given):

```
⋮
type(flips_s) ::  prob1
⋮
call flips_init(prob1,500,1)
⋮
```

2. initialise a double precision real problem with 500 unknowns, no alternate measurements and default rotation buffer size using memory storage. Theory matrix is band-limited with maximum (initial) bandwidth = 50:

```
⋮
type(flips_d) ::  prob2
⋮
call flips_init(prob2,500,1,bandwidth=50)
⋮
```

3. initialise a double precision complex problem with 1000 unknowns, 3 alternate measurements and buffersize = 50 using binary file storage (ID 10):

```
⋮
type(flips_z) ::  prob3
⋮
call flips_init(prob3,1000,3,idnum=10,buffersize=50)
⋮
```

6.2 flips_kill

Deallocates FLIPS data type

SYNTAX

```
call flips_kill(ftype,keepfiles)
```

ARGUMENTS

`ftype`	:	type(flips_<s|d|c|z>); FLIPS data type, initialised using flips_init.
`keepfiles`	:	Logical, optional; if .FALSE. and binary file storage is used, the files will be deleted. default is .TRUE., i.e. the binary files are kept. This flag has no function if memory storage is used.

RULES

It is a good practice to deallocate data types and arrays at the end of the program.

FLIPS data type must be deallocated if it is to be used again.

6.3 flips_add

Add data into FLIPS data type

SYNTAX

```
call flips_add(ftype,n,arows,mrows,errors,force_rotations)
```

ARGUMENTS

`ftype` : type(flips_<s|d|c|z>); FLIPS data type, initialised.

`n` : Integer; number of data rows to be fed in.

`arows` : Real/complex, dimension($n*ncols$); n direct theory matrix rows in a vector (row-major) format.

`mrows` : Real/complex, dimension($n*nrhs$); n measurement rows in a vector (row-major) format.

`errors` : Real, dimension(n) or dimension($n*n$) or dimension($n*(n+1)/2$), optional; if vector, measurement error variances, and if matrix, the full covariance matrix. The full covariance matrix can be given either as full matrix in row-major format (dimension($n*n$), or as upper triagonal part in row-major order (dimension($n*(n+1)/2$).

`force_rotations` : Logical, optional, default .FALSE.. By default, the data rows are first stored into rotation buffer and rotated only when the buffer becomes full. If this flag is set to .TRUE., all the rotations for the added data will be made instantly and the rotation buffer will be emptied[1].

RULES

! → Note that measurement (mrows) must be given as a vector even if $n = 1$!

! → Note that it is *not* possible to give distinct errors for alternate measurements (i.e. for different measurement matrix columns).

If the full covariance matrix is used, FLIPS calculates the inverse of the modified Cholesky decomposition of the covariance matrix, and multiplies both `arows` and `mrows` with it. For large problems, this can take a very long time. Therefore, if the covariance matrix is diagonal (i.e. only variance), it is advisable to feed in the covariance data in as standard deviations vector.

If error is the same for all the data rows to be fed in, it can be given as a single scalar.

If error is omitted, FLIPS will use 1.0 as error term.

→ Code examples See section 7 for examples of using flips_add.

[1] This is seldom needed in normal use.

6.4 flips_solve

Solves the FLIPS problem

SYNTAX

```
call flips_solve(ftype,calc_res)
```

ARGUMENTS

`ftype`	:	type(flips_<s|d|c|z>); FLIPS data type, initialised and the data fed in.
`calc_res`	:	Logical, optional; a flag for residual calculation. Default is .TRUE., i.e. the residuals are calculated.

RULES

! → FLIPS does not check if enough data rows are fed into a problem. If less than *ncols* rows are fed in or if the direct theory matrix is degerate, the target matrix will not be invertible. Depending on used Fortran compiler this might give results containing NAN's or INF's, or an error.

If the problem to be solved is underdetermined, some kind of regularization is necessary. See examples in Chapter 7.

→ 7.3

→ 6.9

The solution can be fetched using FLIPS command flips_get.

If residual is calculated, it is stored into *nrhs*-vector `ftype%residual`.

6.5 flips_calc_cov

Calculates the posteriori covariance matrix

SYNTAX

```
call flips_calc_cov(ftype,full)
```

ARGUMENTS

`ftype` : type(flips_<s|d|c|z>); FLIPS data type

`full` : Logical, optional; flag for full posteriori covariance matrix calculation. Default is .FALSE., i.e. only diagonal elements of the covariance matrix are calculated

RULES

Either full covariance matrix or only the diagonal elements can be calculated. If only diagonal elements are calculated, they are stored into *ncol*-vector.

→ 6.9

The posteriori covariance matrix or diagonal elements vector can be fetched using FLIPS command flips_get.

! → Posteriori covariance matrix calculation is very slow when binary file storage is used.

6.6 flips_resize

Marginalizes and/or adds unknowns

SYNTAX

```
call flips_resize(nfob,ofob,newsize,remove,
                  idnum,buffersize)
```

ARGUMENTS

`nfob`	:	type(flips_<s|d|c|z>); uninitialised FLIPS data type.
`ofob`	:	type(flips_<s|d|c|z>); FLIPS data type, of the same type as `nfob`.
`newsize`	:	Integer, optional; number of unknowns in `nfob`.
`remove`	:	Logical, dimension(ofob%ncols), optional; logical mask vector containing .TRUE. at indices which are to be marginalized.
`idnum`	:	Integer, optional; ID number for `nfob` if binary file storage is to be used for `nfob`.
`buffersize`	:	Integer, optional; number of rotation buffer rows for `nfob`

RULES

`nfob` must be in uninitialised state. If it is used before, it must have been deallocated using `flips_kill` before calling `flips_resize`.

If `flips_resize` is called with `newsize` without giving `remove`, then if

- `newsize` < `ofob%ncols`, first `ofob%ncols - newsize` unknowns will be marginalized.
- `newsize` > `ofob%ncols`, `newsize - ofob%ncols` new unknowns will be added to the end.
- `newsize` = `ofob%ncols`, the target matrices R and Y of `ofob` will be copied to `nfob`. Note that it makes more sense to use `flips_copy` in this case!

→ flips_copy

If `flips_resize` is called with `remove` without giving `newsize`, then the unknowns with indices `i` for which `remove(i)==.TRUE.` will be marginalized. The `nfob` will have `ofob%ncols - count(remove)` unknowns.

If `flips_remove` is called with both `newsize` and `remove`, then first those unknowns for which `remove(i)==.TRUE.` will be marginalized and then `newsize - (ofob%ncols - count(remove))` new unknowns will be added to the end, so that `nfob` will have `newsize` unknowns.

EXAMPLES

1. Let `fob1` and `fob2` be two FLIPS data types:

   ```
   type(flips_s) ::  fob1,fob2
   ```

   Let us initialise `fob1` with 100 unknowns and 1 right hand side and feed in the data:

   ```
   call flips_init(fob1,100,1)
   ⋮
   call flips_add(fob1,...)
   ```

Now marginalize first 10 unknowns and put the resulting new problem into `fob2`: now `newsize` will be $100-10=90$, so we can use

```
call flips_resize(fob2,fob1,newsize=90)
```

Another possibility would have been to use `remove` mask vector. Declare `rem_mask` to be 100-dimensional logical vector and set the components that are to be marginalized as `.TRUE.`:

```
logical, dimension(100) ::  rem_mask
⋮
rem_mask = .FALSE.
rem_mask(1:10) = .TRUE.
call flips_resize(fob2,fob1,remove=rem_mask)
```

2. Let `fob1` and `fob2` be FLIPS data types as in the previous example. Let us marginalize unknowns (with indices) 31–40 and add 20 new unknowns. Number of unknowns for `fob2` will then be $100-10+20=110$ (i.e. `newsize=110`). Additionally, let us use binary file storage for `fob2` (with ID number 10):

   ```
   logical, dimension(100) ::  rem_mask
   ⋮
   rem_mask = .FALSE.
   rem_mask(31:40) = .TRUE.
   call flips_resize(fob2,fob1,newsize=110,
                     remove=rem_mask,idnum=10)
   ```

   Now `fob2` will be a FLIPS data type with 110 unknowns, last 20 of which are new, and the problem data is storaged on the disk with ID=10.

6.7 flips_delete

Deletes data rows from FLIPS problem

SYNTAX

```
call flips_delete(ftype,n,arows,mrows,errors)
```

ARGUMENTS

`ftype` : type(flips_<s|d|c|z>); initialised FLIPS data type that has at least $ncols+n$ data rows fed in, i.e., the problem must be over determined before calling `flips_delete`.

`n` : integer; number of rows to be deleted.

`arows` : Real/complex, dimension($n*ncols$); theory matrix rows to be deleted, in row-major order.

`mrows` : Real/complex, dimension($n*nrhs$); measurement matrix rows to be deleted, in row major-order.

`erows` : Real, dimension(n), the measurements error variances to be deleted. Note that no single value error or full error covariance matrices are allowed (at this time)!

RULES

! → It is only safe to delete exactly the same data that has been previously fed in. Trying to delete something else (while possible) might (or probably will) blow things up completely! There is currently no possibility to check this beforehand.

! → Full error covariance matrices as `errors` are not supported at this time.

→ Code examples — See Section 7 for example of using `flips_delete`.

6.8 flips_copy

Copies FLIPS data types

SYNTAX

```
call flips_copy(nfob,ofob,idnum,buffersize)
```

ARGUMENTS

`nfob` : type(flips_<s|d|c|z>); uninitialised FLIPS data type. After execution, it will contain the same data as `ofob`.

`ofob` : type(flips_<s|d|c|z>); same type as `nfob`.

`idnum` : Integer, optional; ID number for `nfob`. If given, `nfob` will use binary file storage.

`buffersize` : Integer, optional; number of rotation buffer rows for `nfob`. If omitted, `nfob` will use the same number as `ofob`.

RULES

FLIPS object `nfob` must be uninitialised. If it is used before, it must have been deallocated using `flips_kill`.

`flips_copy` is handy when user wants to change the storage method from binary file storage to memory storage, or vice versa.

6.9 flips_get

Fetches problem matrices and vectors from FLIPS data type

SYNTAX

```
call flips_get(mtype,ftype,vec/mat)
```

ARGUMENTS

`mtype` : character(len=4); name label of the vector/matrix to be fetched.

`ftype` : type(flips_<s|d|c|z>); FLIPS data type

`vec/mat` : Real/complex matrix or vector; after execution will contain the desired vector or matrix.

RULES

This command can be used to fetch problem data matrices and vectors into a new variable. Matrices can be fetched in matrix or vector form, see below.

The name labels for different vectors/matrices and their sizes are:

`"solu"` : Solution matrix/vector. Size in matrix form is $ncols \times nrhs$ and in vector form (in row-major order) $ncols * nrhs$.

`"cova"` : Posterior covariance matrix (or its diagonal). If only diagonal is calculated, vector size is $ncols$. If full matrix is calculated, the size in the matrix form is $ncols \times ncols$. In the vector form, only the upper triangular part of the symmetric matrix is fetched (in row-major order), so the required size of the vector is $ncols * (ncols+1)/2$.

`"rmat"` : Theory target matrix R. Size in the matrix form is $ncols \times ncols$. In the vector form, only upper triangular part (in row-major order) is fetched, so the size of the vector is $ncols * (ncols+1)/2$.

`"ymat"` : Measurement target matrix/vector Y. Size in the matrix form is $ncols \times nrhs$ and in the vector form (in row-major order) the size is $ncols * nrhs$.

`"invr"` : Inverse of the target matrix R (used in the posteriori covariance calculation). The sizes are the same as in the `'rmat'` case.

7 Examples

7.1 Solving a simple linear system

In the first example, let us solve the following simple (over determined) linear system:

$$m = A \cdot x,$$

where

$$m = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 2 \end{pmatrix} \quad \text{and} \quad A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{pmatrix}.$$

This can be solved using FLIPS as follows:

```
program ex1
  use flips
  implicit none
```

declare variables

```
  type(flips_s) :: h
  real(sp), dimension(3) :: Arow
  real(sp), dimension(1) :: meas
  real(sp), dimension(3) :: solution
```

initialise FLIPS data type

```
  call flips_init(h,3,1)
```

Add data row-by-row

```
  Arow = (/ 1.0, 2.0, 3.0 /)
  meas = 0.0
  call flips_add(h,1,Arow,meas)

  Arow = (/ 2.0, 3.0, 1.0 /)
  meas = 1.0
  call flips_add(h,1,Arow,meas)

  Arow = (/ 3.0, 2.0, 1.0 /)
  meas = 2.0
  call flips_add(h,1,Arow,meas)

  Arow = (/ 1.0, 3.0, 2.0 /)
  meas = 2.0
  call flips_add(h,1,Arow,meas)
```

solve problem

```
  call flips_solve(h)
```

fetch solution

```
  call flips_get("solu",h,solution)

end program ex1
```

7.2 Solving a linear system with errors

Let us add measurement errors to the previous example. In other words, let us solve the problem

$$m = A \cdot x + \varepsilon,$$

where

$$m = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 2 \end{pmatrix}, \qquad A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{pmatrix} \qquad \text{and} \qquad \varepsilon = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{pmatrix}.$$

Note that in this case, the error covariance matrix is

$$\Sigma = \begin{pmatrix} 0.1^2 & 0 & 0 & 0 \\ 0 & 0.2^2 & 0 & 0 \\ 0 & 0 & 0.3^2 & 0 \\ 0 & 0 & 0 & 0.4^2 \end{pmatrix}$$

In the following, above problem is solved by FLIPS using binary file storage and double precision. Moreover, the data is fed into FLIPS two rows at the time:

```
program ex2
  use flips
  implicit none
```

declare variables

```
  type(flips_d) :: h
  real(dp), dimension(6) :: Arows
  real(dp), dimension(2) :: meas, errors
  real(dp), dimension(3) :: solution
  real(dp), dimension(3,3) :: post_covar_matrix
```

initialise FLIPS (with binary file storage)

```
  call flips_init(h,3,1,idnum=1)
```

feed data in

```
  Arows = (/ 1.0, 2.0, 3.0, 2.0, 3.0, 1.0 /)
  meas = (/ 0.0, 1.0 /)
  errors = (/ 0.1, 0.2 /)
  call flips_add(h,2,Arows,meas,errors)

  Arows = (/ 3.0, 2.0, 1.0, 1.0, 3.0, 2.0 /)
  meas = (/ 2.0, 2.0 /)
  errors = (/ 0.3, 0.4 /)
  call flips_add(h,2,Arows,meas,errors)
```

solve and fetch solution

```
  call flips_solve(h)
  call flips_get("solu",h,solution)
```

calculate and fetch the posteriori covariance matrix

```
  call flips_calc_cov(h,full=.TRUE.)
  call flips_get("cova",h,post_covar_matrix)
```

deallocate FLIPS data type

```
  call flips_kill(h)

end program ex2
```

7.3 Solving underdetermined problem (using regularization)

Let us find the shortest vector x that satisfies the underdetermined linear system

$$m = A \cdot x + \varepsilon,$$

where

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 1 & 2 \end{pmatrix}, \qquad m = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and ε is a 2×1 matrix consisting of 1's. Since FLIPS can not directly handle underdetermined systems, we have to use regularization. To this end, we extend the above system by

$$\begin{pmatrix} m \\ 0 \end{pmatrix} = \begin{pmatrix} A \\ I \end{pmatrix} \cdot x + \begin{pmatrix} \varepsilon \\ \varepsilon_r \end{pmatrix},$$

where I is a 3×3 identity matrix, 0 is a 3×1 zero column vector and ε_r is a 3×1 column vector with arbitrary great values, say 10^6. Hence, we replace the original problem with

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 1 \\ 1 \\ 10^6 \\ 10^6 \\ 10^6 \end{pmatrix}.$$

In other words, in addition to the original data which is in the first two rows of the vectors and the matrix, we "allow" every unknown to be statistically zero with a very big standard deviation. This is now "overdetermined" system, so we can use FLIPS.

```
program ex3
  use flips
  implicit none
```

declare variables

```
  type(flips_s) :: h
  integer :: n
  real(sp), dimension(6) :: Adata
  real(sp), dimension(2) :: mdata, edata
  real(sp), dimension(3,3) :: id_mat
  real(sp), dimension(3) :: mreg, ereg, solution, &
                            cov_mat_diag
```

initialise FLIPS

```
  call flips_init(h,3,1)
```

feed in the data

```
  Adata = (/ 1.0, 2.0, 1.0, 3.0, 1.0, 2.0 /)
  mdata = (/ 1.0, 0.0 /)
  edata = 1.0
  call flips_add(h,2,Adata,mdata,edata)
```

construct the regularization matrices and feed them in one row at the time

```
  id_mat = 0.0
  do n = 1,3
     id_mat(n,n) = 1.0
  end do
  mreg = 0.0
  ereg = 1.0E6
  do n = 1,3
     call flips_add(h,1,id_mat(n,:),mreg,ereg)
  end do
```

solve the problem and fetch solution

```
   call flips_solve(h)
   call flips_get("solu",h,solution)
```

calculate the diagonal of the posteriori covariance matrix and fetch it

```
   call flips_calc_cov(h,full=.FALSE.)
   call flips_get("cova",h,cov_mat_diag)
```

deallocate FLIPS data type

```
   call flips_kill(h)
end program ex3
```

7.4 Marginalizing and adding unknowns (using flips_resize)

In this example, we will first create a FLIPS data type with 500 unknowns, add the data (1000 random equations), and then marginalize away first 50 unknowns. After that, we will add 50 new unknowns.

```
program ex4
  use flips
  implicit none
```

declare variables

```
  type(flips_s) :: f1, f2
  real(sp), dimension(500) :: arow
  real(sp), dimension(1000,1) :: meas
  logical, dimension(500) :: remove_mask
  integer :: i
```

initialise FLIPS (f1)

```
  call flips_init(f1,500,1)
```

feed in the data (row-by-row)

```
  call random_number(meas)
  do i = 1,1000
    call random_number(arow)
    call flips_add(f1,1,arow,meas(i,:))
  end do
```

marginalize 50 first unknowns

```
  remove_mask = .FALSE.
  remove_mask(1:50) = .TRUE.
  call flips_resize(f2,f1,remove=remove_mask)
```

Now FLIPS data type `f2` will contain the unknowns 51–500. We could solve both `f1` and `f2` and see, that the solutions for unknowns 51–500 agree.

Next, we reuse `f1`, so we can add 50 new unknowns to `f2`:

deallocate f1

```
  call flips_kill(f1)
```

add 50 unknowns

```
  call flips_resize(f1,f2,newsize=500)
```

FLIPS data type `f1` will now contain 500 unknowns, numbers 451–500 being new. Note that `f1` is not solvable until more data is fed in! If `f2` is not needed anymore, it can be deallocated.

```
.
.
.
end program ex4
```

7.5 Deleting data

In this example, we first feed in all the data, then delete some of it and then solve the problem. After that we solve the same problem without feeding in the before deleted data rows, solve the problem and compare the results.

```
program ex5
  use flips
  implicit none
```

declare variables

```
  type(flips_s) :: h
  real(sp), dimension(200,100) :: theorymat
```

```
                    real(sp), dimension(200) :: meas
                    real(sp), dimension(100) :: sol1, sol2
                    integer :: i

initialise FLIPS    call flips_init(h,100,1,buffersize=100)

construct random data  call random_number(theorymat)
                    call random_number(meas)

feed data in        do i = 1,200
                       call flips_add(h,1,theorymat(i,:),(/ meas(i) /))
                    end do

delete the last 50 data rows  do i = 151,200
                       call flips_delete(h,1,theorymat(i,:), (/ meas(i) /))
                    end do

solve and store solution  call flips_solve(h)

                    sol1 = h%solmat

                    call flips_kill(h)

the same problem without deletion  call flips_init(h,100,1,buffersize=100)
feed in the first 150 rows  do i = 1,150
                       call flips_add(h,1,theorymat(i,:),(/ meas(i) /))
                    end do

solve and compare two solutions  call flips_solve(h)

                    sol2 = h%solmat

                    call flips_kill(h)

the difference should be very small  write(*,*) maxval(sol1 - sol2)

                 end program ex5
```

III. Technical Notes

8 Compiling and installing

8.1 Fortran 2003 extensions

By default, FLIPS uses some F2003 extensions, namely, stream I/O and allocatable components in structured data types. However, not every modern Fortran 95 compiler support these features, see table below:

	Stream I/O	Allocatable components
IBM XLF	Yes	Yes
g95	Yes	Yes
gfortran[1]	Yes	Yes
Absoft Fortran	No[2]	Yes
Intel ifort	Yes[2]	Yes
Portland Group Fortran[3]	No	Yes
Sun Fortran[3]	No	Yes
Pathscale Fortran[3]	No	No

[1] Version 4.2.0 was first to support both extensions.

[2] Stream I/O implemented in version 10.1.
with the proposed F2003 standard.

[3] Tested by J. Vuorinen.

8.2 Compiling FLIPS module

At this time there is a very basic GNU autoconf configure script (made by Juha Vierinen) available. This script tries to find the system's Fortran compiler and automagically modify the file `Make.defs` accordingly to take care of the Fortran 2003 extensions. However, it is advisable to manually check the resulting file `Make.defs` to make sure that the right extensions are being used (and also to check the compiler flags for optimization).

The autoconfiguration is made by giving the command

```
./configure
```

in the FLIPS source root directory.

At this time the allocatable components extension is required and FLIPS can not be compiled without it, but the user can choose how to handle the stream I/O.

! → *Note, however, that the use of binary file storage is advisable only if Stream I/O is available, since the alternative (Direct access file I/O) is very slow in most compilers!*

! → The compiling process needs also GNU M4 macro processor which is freely available (and part of the Mac OS X[1] and most of the Linux distributions).

The `Make.defs` file must have (at least) the following rows (example is for Intel Ifort):

```
FC=ifort
CPP=cpp -P -Wtraditional
CPPFLAGS= -DF2003_STREAM
```

[1] Some Mac OS X versions ship with a buggy M4 macro processor. See the file `INSTALL` in FLIPS source root directory!

```
FFLAGS= -O3 -assume byterecl
M4FLAGS=-DF2003_STREAM
```

The used variables are

`FC`	:	Fortran90 compiler command
`FFLAGS`	:	Fortran compiler options
`CPP`	:	(GNU) C preprocessor command
`CPPFLAGS`	:	`-DF2003_STREAM` (for compilers suporting Stream I/O) *or* `-DDIRECT` (for compilers not supporting the Stream I/O extension)
`M4FLAGS`	:	`-DF2003_STREAM` (for compilers suporting Stream I/O) *or* `-DDIRECT` (for compilers not supporting the Stream I/O extension)

After the autoconfiguration (and possible modifications to the file `Make.defs`), the FLIPS module is build with the command

```
make
```

If everything goes smoothly, the following files will be created:

`libflips.a`	:	Archived library version of the FLIPS module
`flips.o`	:	FLIPS module object file
`flips.mod`	:	(name and extension may vary depending on Fortran compiler) FLIPS module include file

The library file `libflips.a` (and/or `flips.o`) should be copied somewhere in the library search path of the used compiler/linker and the file `flips.mod` somewhere from where the fortran complier searches for the module include files. Alternatively, the file `flips.o` can be used instead of the library file `libflips.a`.

9 Using FLIPS in Fortran programs

Using FLIPS module in one's own Fortran programs is pretty straight forward. See the Examples for how to use FLIPS in Fortran code.

When compiling, the FLIPS module must be linked in the normal way with the program that uses it. Consult the User Guide of the used Fortran compiler for instructions.

10 Interfaces for R and MATLAB

Experimental R and MATLAB interfaces together with *minimal* instructions of how to install and use them can be found in the `interfaces` directory of FLIPS source distribution. These interfaces are not really suitable for larger problems at this point, but they are going to be developed further.

! → We have dropped the support for MATLAB interface for the time being due to the low demand and the high amount of work to maintain it. Last working version of FLIPS with MATLAB interface is 1.5.

11 Misc stuff

11.1 Bugs

FLIPS most probably contains number of bugs, some of which probably big and ugly. If you find that something does not work as it should, please send email describing the problem to

mikko.orispaa@oulu.fi