M A C E K 1.0.2

"MAtroids Computed Efficiently" Kit
Macek version 1.0.2, manual version 0.999.
10 February 2003

Petr Hliněný

(Developed with help of Geoff Whittle, Victoria University, and the Marsden Fund of New Zealand.)

Table of Contents

1 Overview of Macek

The *Macek Project* has been developed primarily for math researchers in matroid theory. (If you do not know what matroid theory is, then the package is likely not for you.) This project is intended both to help with usual tiresome matroid routines, and to allow for long exhaustive computations over matroid classes. We suggest potential users to read the book [J.G. Oxley, Matroid Theory, Oxford University Press 1992,1997]. The project web page with recent updates can be found at http://www.mcs.vuw.ac.nz/research/macek/.

So far, the project deals with matroids represented by matrices over finite fields and partial fields. Many common matroids are distributed with the program, and new ones may be easily entered. There are various tools for handling matroids, their matrices, and sets of matroids. One may pivot matrices, delete or contract matroid elements, and generate 3-connected extensions for matroid representations. Structural tests for minors, equivalence, connectivity, branch-width, etc, are also provided in the project. You may read about the theoretical background of the *Macek Project* in [P. Hlineny, Practical Computations with Representable Matroids, manuscript 2002].

However, there are certain limitations to the computing power of Macek — following from the fact that only represented matroids are handled in the program, and from theoretical problems with inequivalent matroid representations over larger fields, etc. So be careful and read the documentation thoroughly.

More functions are planned for the future. . .

1.1 Using Macek

The Macek package can be used on (almost) any computer platform with the GNU C compiler, relevant C libraries and development environment, and the basic set of GNU utilities. All these required programs are free for anybody – see http://www.gnu.org. More about supported platforms is in Section 2.1 [Installing], page 3. (No support for M$ Windows is provided; however, there are no major design reasons why the program would not compile and work there after few adjustments.)

When considering the Macek program, do not expect anything like a graphical user-interface. The program provides only a rich command-line (non-interactive) interface, with some basic internal scripting capabilities. (Given the nature of the program, and also that of matroids, a graphical interface would not be much use anyway.) So forget your mouse and use the keyboard! For more complicated batch-computations, you should call the Macek program within a suitable external scripting language, like a unix shell.

Many of the computations in Macek are exponential, even in their nature, and so they may take quite long time. However, usually matroids on up to about 20 elements can be handled in reasonable computing time on a usual modern PC computer. The memory requirements of the program also grow exponentially for certain computations, but this should not cause serious problems in modern computers with 64MB or more.

1.2 About Version 1.0

The Macek project started during 2001, and the first stable public release was version 0.8 in March 2002. The minor version 0.8.2 is the last one in that branch. The following

development branch 0.9.x have brought many updates and speed-ups to the program, and have resulted in a new stable branch 1.0 in August 2002.

The main improvements in version 1.0 are these:

More information about matroid properties, now also including connectivity. Section 6.2 [Printing], page 26.

Cleaner and faster algorithms are now used for various structural functions, mainly for minor and equivalence testing. Section 6.6 [Structural], page 28.

Better organized canonical check in extension generating, which makes extensions of large matrices much faster. Section 6.7 [Generating], page 30.

Some limited program-flow capabilities for internal scripting in Macek. Section 6.9 [Com-Flow], page 33.

Confusing atomic matrix-entry expressions like $a-1$ in '`-a-1`' are no longer scanned from the input. That means, '`-a-1`' is read really as $-a-1=-(a+1)$. Unfortunately, this may cause some problems when reading matrices written by an earlier version of Macek over partial fields like $NREG$ or $GMEAN$. Section 4.2 [Matrix-Entry], page 12.

2 Quick-Start

This chapter contains installation instructions for the Macek package, and a brief introduction to using the program. It is intended for those who want quickly see the program in action, without reading the long manual first. Consult also the file '`doc/QUICKSTART`' for up-to-date information. We provide examples demonstrating the basic parts and concepts of the program. However, if you want to understand these examples (and the program itself) in depth, then there is no other way than to read the whole manual...

2.1 Installation of Macek

We briefly describe how to install the Macek package on your computer. This description is prepared mainly for skilled computer users. If you have problems following these instructions or obtaining the required GNU programs, better ask your computer specialist. No knowledge of matroids is necessary to install the package.

The Macek package can be used on almost any computer platform with the GNU C (`gcc`) compiler, relevant C libraries and development environment, and the basic set of GNU utilities (at least `gmake`, `flex`). All these required programs are free for anybody – get them from `http://www.gnu.org` or other mirrors. No C++ compiler/development is necessary. In particular, Linux and other unix-clones, with GNU C development and GNU utils installed, would work fine. The list of the current "officially" supported platforms can be found in the file '`doc/SUPPORTED`' of the Macek package.

First create a subdirectory '`macek`', copy the package archive to it, and unpack the archive with tar / gzip. To obtain a current documentation, read the files in the '`doc`' subdirectory, or type '`gmake info`' in the top directory. Then compile the program with '`gmake compile`' in the top directory. You may also run '`gmake all`' and '`gmake xall`' in the '`src`' subdirectory, with the same effect. Keep in mind that you *really have* to use the GNU version of `make` – `gmake`, not the ordinary version.

If the compilation is successful, then the resulting two executables '`macek`' and '`macek.nodebug`' appear in the '`src`' subdirectory. You may also look at the file '`src/Make.local`' which contain local modifications to the project makefiles. Various useful things may be set/modified there, but these require detailed knowledge of your computing platform. When you get into troubles, Section 8.2 [Troubleshooting], page 45 may help...

2.2 Run the Program

Run the `macek` executable from the '`exe`' subdirectory of the package, where it is symlinked from. We also suggest to run the program on a terminal with at least 100 columns, and better with 132 columns. Try first to call '`macek -h`' to see a simple online help. Follow the instructions to obtain more online help.

The presented Macek examples are called from the `bash` shell. However, similar shells like `zsh`, `ksh` should work in the same way. If you want to use `csh` or others, you may need to modify or escape active characters in the commands. (In particular, it looks like '`!`' is active in `tcsh` even when it is quoted. So if you do not find a way around such a syntactical problem, use `bash` as your shell.) Notice that you need to have the current directory in the shell search path, or you have to call the program as `./macek ...`.

The following call

```
bash$ macek -g-2 -pGF3 '!print' U24
```

produces an output similar to this sample:

```
567~        Printing output of the command "!print ((t))   ..[1]":
567~  Printing matrix of the frame [U24]: "the matroid U_2,4 uniform"
   ~   ------------------------------------------------------------
   ~   matrix 0x80fcb48 [U24], r=2, c=2, tr=0, ref=(nil)
   ~             '-1')   '-2')
   ~
   ~    '1')       1       1
   ~    '2')       1       2
   ~   ------------------------------------------------------------
```

The option '`-g-2`' suppresses usual debugging messages during program run. The option '`-pGF3`' selects the finite field $GF(3)$ for the computation. Then there is the program command `!print` followed by the command parameter – the matroid `U24`. (Note the quotas around the command since '`!`' is an active shell character.) The meaning of `!print` command is pretty obvious — it prints the matrix representing the matroid $U24$ over $GF(3)$. Numbers on the left of the output mean current time in seconds (modulo 1000), which may be useful to see in longer computations.

Next, try to run

```
bash$ macek -g+1 -pGF3 '!print' U24
```

with the option '`-g+1`' (or even higher values) instead of '`-g-2`'. In this way, you get some debugging messages that show what the program does. For example, among other output lines, you may see:

```
[emflexsu:frame_doinput_()89  ~926] Calling to scan a list of frames...
[emflexsu:frame_doinput_()99  ~926] Input frame - scanning "!print":
[emflexsu:frame_doinput_()99  ~926] Input frame - scanning "U24":
[emflex.l:    frame_flex()520 ~926] Including from 'U24' (->'U24')
```

The prefix of each debugging line point to the source file, function and line that generated the message, and then follows the message itself. The debugging messages are mainly for those, who want to follow the program computation in the source files, and for catching possible bugs in further development. If you do not want the messages at all, you may run the `macek.nodebug` executable instead. However, note that the latter version also skips all internal consistency checks in the program.

As the reader may have noticed, each command in Macek starts with the character '`!`'. However, for simple command-line use there are simplified shortcuts available. See Section 6.11 [Procedures], page 34. The following are three easy examples.

```
bash$ macek -pGF3 print U24
bash$ macek -pGF3 minor F7- U24
bash$ macek -pGF3 prints F7-
```

2.3 Matrices and Frames

In order to use the Macek program, one needs to input the *matrices representing matroids*. The program works with matrices in the standard reduced form, i.e. without the

leading unit matrix. Some common matroid representations are distributed with this package in the '`exe/Matrices`' subdirectory. You may easily create your own matroid files in a similar fashion, with space-separated entries, line by line. Each matrix line should start with a space. Comment the files by lines starting with '`#` '. Any bracketed math expression may appear as a matrix entry.

It is also possible to give a matrix directly on the command line like these:

```
bash$ macek -pGF5 '!print' ' 1 2; 3 2+2'
bash$ macek -pGF2 '!print' ' 1 1; 0 0 1; 0 1 0 1'
bash$ macek -pGF4 '!print' ' w w^2; w^3'
bash$ macek -pGF4 '!print' ' w w^2; (w^3+w)*(w+1)'
```

Here '`;`' replaces line-ends. Notice that, for example, inputting an entry '`w`' in $GF(3)$ or '`2`' in $GF(2)$ cause an error.

In fact, the basic input entity in the program is called a *frame*; See Chapter 4 [Frames], page 11. One frame usually holds one matrix, but it may also hold frame- *commands* and *options*; See Chapter 6 [Commands], page 25, See Chapter 5 [Options], page 21. All given command-line arguments that do not start with '`-`' are read as frames. These result in a tree-structure of frames, with the first argument as the root.

The tree structure can be printed with a command:

```
bash$ macek -pGF4 '!prtree' U24 '{ U25 U35 F7 }'
...
~820~   Printing the subtree of the frame 0x81552b8 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [U24] m2x2  "the matroid U_2,4 uniform"
    ~       (2.2)fr  [noname-2]   ""
    ~         (3.1)fr  [U25] m2x3  "the matroid U_2,5 uniform"
    ~         (3.2)fr  [U35] m3x2  "the matroid U_3,5 uniform"
    ~         (3.3)fr  [F7] m3x4  "the matroid F_7 Fano"
~820~  ------------------------------------
```

The command `!prtree` (with no matrix) forms the root frame, the next two arguments form its descendant frames, and the included matroids $U25, U35, F7$ form the descendants of the second son of the root.

Another example is the command `!move` that manipulates the frames in the tree (moves, copies, or deletes them). To understand this command better, you need to learn about addressing command parameters Section 4.5 [Addressing], page 15. (Nodes of the tree are addressed by bracketed expressions in the natural way; '`T`' picks a node, '`S`' picks all sons of a node, the lower-case letters '`t`','`s`' also erase the picked nodes in some commands.) Run the next examples, and see the action:

```
bash$ macek '!prtree;!move ((t));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!prtree;!move (()(t));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!prtree;!move ((t)(()(t)));!prtree' W3 '{ W4 R10 R12 }'

bash$ macek '!prtree;!move ((T)) >(()(t));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!prtree;!move ((T)) >((2)(t));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!prtree;!move ((t)) >(()(t));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!prtree;!move ((T)) >(((t)));!prtree' W3 '{ W4 R10 R12 }'
```

```
bash$ macek '!move (()(S)) >(((s)));!prtree' W3 '{ W4 R10 R12 }'
bash$ macek '!move (()(S)) >(((t(t(t)))));!prtree' W3 '{ W4 R10 R12 }'
```

2.4 Examples of Use

In this section, we show several more examples demonstrating the use of some Macek commands. (Recall that you get an online overview of all commands with '`macek -HHc`'.)

One may easily pivot matrices like in the next example:

```
bash$ macek -pReg '!print;!pivot 1 2;!print' R10
```

Matroid elements are deleted or contracted in the following way:

```
bash$ macek -pReg '!print;!delete -3;!print' R10
bash$ macek -pReg '!print;!delete 2;!print' R10
bash$ macek -pReg '!print;!contract 2;!print' R10
bash$ macek -pReg '!print;!contract -5;!print' R10
```

Minor-testing (up to inequivalence of representations!) is demonstrated in the next commands:

```
bash$ macek -pReg '!minor' R12 R10
bash$ macek -pReg '!minor' R12 grK33
```

Extensions or coextensions of represented matroids are generated as follows (all 3-connected, matrix-equivalence factored-out):

```
bash$ macek -pBinary '!extend c;!prtree' W3
bash$ macek -pReg '!extend c;!prtree' W4
bash$ macek -pReg '!extend r;!prtree' R12
bash$ macek -pReg '!extend b;!prtree' R10
```

Some more involved chains of commands are demonstrated in the following examples:

```
bash$ macek -pReg '!deleach;!prtree;!filt-minor;!prtree' R12 grK33
bash$ macek -pReg '!deleach;!prtree;!filx-minor;!prtree' R12 grK33
bash$ macek -pdyadic '!extend r;!prtree;!minor' F7- 'F7-;!dual'
```

3 The Macek Program

First, read the installation instructions in See Section 2.1 [Installing], page 3. After installing the Macek program successfully, you find the executable(s) in the '`src`' subdirectory of the package. However, we suggest to run the executable from the '`exe`' subdirectory of the package, where it is symlinked from.

Since the Macek program has only command-line interface, we suggest to run it within a suitable (comfortable) command-shell, like unix shells `bash`, `zsh`, new `ksh` or similar. If you want to use `csh`-clones, you would probably have to adjust the provided examples. (In particular, it looks like '`!`' is active in `tcsh` even when it is quoted. So if you do not find a way around such a syntactical problem, use `bash` as your shell.) Moreover, to get the program output neatly organized, we suggest to use a terminal of 100 or more (up to 132) characters wide.

The program has two executables — `macek` and `macek.nodebug`. Usually you would run the first one. The second executable, `macek.nodebug`, is, however, faster since it skips most of the internal consistency checks and all debugging messages. So it is suitable for long computations when you are already sure that your script computes the right results correctly.

3.1 Command-line Arguments

We list all command-line options of the Macek program (i.e. all recognized arguments starting with '`-`'):

* `-gN` (or `--debug=N`) Adjust the debugging level in the program by N — how much is printed during program run See Section 3.3 [Messages], page 8. (Not applicable to `macek.nodebug`.)

* `-h` (or `--help`) Print a simple program help.

* `-H[H][pfco]` Print more help on specified topics (`p`artial fields, `f`rames, `c`ommands, `o`ptions), and even more with `-HH`. See Chapter 4 [Frames], page 11.

* `-pF` (or `--pfield==F`) Set the default partial field in the program to F. See Section 3.6 [P-Fields], page 9.

* `-v` (or `--version`) Print the program version.

3.2 Frame Input

All other command-line arguments of the Macek program (that do not start with '`-`') are read as *frames*; they form the program input. A frame is the basic data-entity in the program. See Chapter 4 [Frames], page 11.

As explained later in details, the frames form a tree-structure in the program. The first frame-arguments forms the root of the tree, and all others are its sons. Moreover, some frames may include other subframes that are then stored as their sons, and so on. . .

When giving frames as arguments to the program, do not forget to quote them, as they may contain spaces and active shell characters inside.

3.3 Program Output

There are two categories of messages printed from the Macek program — the regular *output*, and the *debugging messages*. (Debugging messages are used to trace the program execution, and to provide additional profiling information.)

Typical command output in the program looks like the following:

```
705~  Printing the subtree of the frame 0x80e9a38 [noname]:
   ~   (1.1)fr  [noname] ""
   ~     (2.1)fr  [U25] "the matroid U_2,5 uniform"
   ~       (3.1)fr  [U25_r1] "mat #1 row co-exten to 'U25'"
```

For profiling purposes, each output line starts with the current time `~nnn~` in seconds modulo 1000. Then the output itself follows.

Typical debugging message in the program looks like the following:

```
[gener.c:gener_extensions()368 ~520] >>extension #1 of [U25]: 1,w+1,w
```

The starting bracketed information contain the source file name, the function, and the line of the message, and the current time in seconds modulo 1000. These information allow to easily track the program computation in the program source. The next text message then tells you what the program currently does. You may control the amount of printed debugging messages with ⟨undefined⟩ [-g], page ⟨undefined⟩. No debugging messages are printed from the `macek.nodebug` executable.

3.4 Error Reporting

If anything unusual or problematic happens in the Macek program, then an *error message* is issued. Similarly as in debugging messages Section 3.3 [Messages], page 8, an error message first tells you what function in what source file generated this message, and then the text description follows. Fatal errors immediately terminate the program, while the program run continues for other errors. (However, this may cause subsequent induced errors.) Additionally, the programs prints a warning at the end when an error happened during the computation.

A usual error message looks like the next example

```
*** ERROR (by emflexsup.c, in yyincl() 412) in "NOxxx" l.1: ***
  Cannot open include file 'NOxxx'!
```

or

```
*** ERROR: (reported by frameop.c, in frame_getparamfr_r() l.706) ***
  Missing requested subframe of 0x80ea5f8 in '(t|', depth 1.
```

Such messages usually tell you that there was something wrong with the program input or commands. You have to correct the input to get your computation right. . .

Moreover, there is another more serious type of error messages, called *program errors*, which start with '`Program ERROR [bad:-(]:` '. These indicate that something very wrong happened inside the program. Such a program error may happen after a usual error. If a program error is issued for a correct program input, then it indicates a bug in the program. Please report such incidents to the author at http://www.mcs.vuw.ac.nz/research/macek/.

To make the Macek program more reliable, there is number of internal checks implemented at various places in `macek`. (These checks usually watch consistency of data structures, or provide alternative ways of computing the same results.) Uncovered inconsistencies result in program errors. Learn more about them in the program sources. However, such internal checks take time, and so they are often only randomized to make them faster. The provided alternative fast executable `macek.nodebug` skips all these internal checks.

3.5 Program Environment

This section describes how the Macek program interacts with its computing environment; including reading / writing files, search paths, using environment variables, etc. . .

The program input is taken from one input stream that starts with the command-line argument, but then it may include input from arbitrary files. File names follow unix conventions, and they are case-sensitive (of course, if supported by the system). A file name may contain spaces and other strange characters, but then it must be quoted. Be reasonable, and use "normal" file names, though.

If a file name starts with the slash '`/`', then it is taken as an absolute path from the filesystem root. Otherwise, the file is searched in the input search path, which may be obtained by calling '`macek -Hf`'. See Section 3.1 [Command-line], page 7. Similarly, the output search path (different!) is used when writing files.

When reading, the given file name is first tried without added extension, and then the default extension is appended. When writing to a file, the default extension is automatically appended unless the file already has an extension.

To avoid an accidental loss of computing data, the Macek program in some situations automatically saves the whole frame tree to the '`/tmp/`' directory. If the program computation is long (in about minutes), then the frame is saved to the file '`/tmp/macek-out-NN.mck`'. Similarly, if an error happens, then the current frame tree is saved to '`/tmp/macek-err-NN.mck`'. A message is printed after such automatic save.

Handling environment variables will be added later.....

3.6 Partial Fields

The Macek program can work with matrices over so called *partial fields*. A partial field is an extension of a usual field which allows the sum to be a partial operation (i.e. not all results are defined). A typical example are regular (also known as totally-unimodular) matrices in which only the numbers $-1, 0, 1$ are used, and the sums $1 + 1$ or $-1 - 1$ are undefined. You do not need to know about partial fields to use this program, just consider usual finite fields instead.

The program works only with finite partial fields, which means those in which the equation $x = y + 1$ has finitely many solutions. This includes all finite fields. To obtain the list of all partial fields currently implemented in the Macek program, run '`macek -HHp`'. You set the partial field for the program as with ⟨undefined⟩ [-p], page ⟨undefined⟩. If you want to add more partial fields to the program, you have to update the program source and recompile it. (See the file '`src/pfield/pfdef-more.inc`'.)

Each partial field in the program is represented by the generators of the multiplicative subgroup. Specifically, each number is given by a sign (with values 0,-1,1), and a list of integral exponents corresponding to the generators. Multiplication is implemented in the obvious way. Addition is implemented via multiplication and a table of *fundamental elements* — those x for which $x-1$ is defined. If some input expression (sum) is not defined in the partial field, then an error is reported. Division by zero results in a zero value, but no error is reported.

A matrix over a partial field is *proper* if all of its subdeterminants are defined. Then also all subsequent matrix operations in the program will be defined. When reading input, proper matrices are checked by a randomized test (unless this feature is switched off), and possible errors are reported. One may also thoroughly test proper matrices with an explicit command (which is quite slow), Section 6.6 [Structural], page 28. If it still happens that an improper matrix gets into the program, then lots of arithmetic-error messages may be reported during program execution.

One may also change between partial fields during program execution, and to translate matrices from one partial field to another. The list of all supported partial field translations, run '`macek -HHp`'. Read more in Section 6.8 [Diff-fields], page 33.....

Remark. Partial fields naturally (implicitly) appear in matroid-representation theory, like in totally-unimodular or dyadic matrices. They were formalized by Charles Semple and Geoff Whittle in [Partial Fields and Matroid Representation, Advances in Applied Mathematics 17 (1996), 184-208]. We follow their formalization here.

4 Frames – Data Handling

As noted above, the basic data entity in the Macek program is called a *frame*. One frame usually holds a matrix representation (only one), but it may also hold an arbitrary number of frame- *commands* and *options*; Chapter 6 [Commands], page 25, Chapter 5 [Options], page 21. Each frame can be identified by its name Section 5.2 [Naming], page 21.

One frame may also refer to several subframes, called its *sons*. In this way, frames in the program form a rooted tree-structure; the frames are the nodes of the tree. (In general, the structure of frames could form an arbitrary directed graph, but the program allows only one rooted tree as the frame structure.)

4.1 General Input Syntax

Frames are given to the Macek program in a text format. The frame input is organized by lines which may have several different meanings. A line starting with '`#`' or '`%`' is a *comment line*, and it is ignored. A line starting with space(s) or '`=`' is a *matrix line* — its expressions define the entries of the matrix in this frame. For example, let the following be the file '`U24x`':

```
# The uniform matroid U_2,4 represented over GF(3) or GF(5).
  1     1
  1     2
```

A line starting with '`<`' or the keyword '`include`' is an *include line*. Each subsequent word on such a line gives a file-name to be included into the input stream. If the file-names contain special characters, like a space, enclose them into quotas like '`< "long file name.mck"`'. Note that include lines have nothing to do with subframes. We continue the previous example with the file '`U35x`', using an include:

```
# The uniform matroid U_3,5 represented over GF(5).
< U24x
  1     3
```

Lines starting options, commands, or subframes are described in details later. For now, we just give a simple introduction. A *command* is given on a line starting with '`!`' or the keyword '`command`'. The next call shows the command for printing a matrix.

```
bash$ macek -pGF5 '!print' U25
```

An *option* is given on a line starting with '`@`' or the keyword '`option`'. One useful basic option tells the program to consider the previous entered matrix entries in the matrix-transposed way. This example shows an easy way to obtain a file for the matroid $U25$ from $U35$ (cf. the previous paragraph).

```
# The uniform matroid U_2,5 represented over GF(5),
#  obtained by transposing U_3,5.
< U35x
@transpose
```

As already noted, input frames are given to the program as command-line arguments Section 3.2 [Frame-arg], page 7. To make the life easier, there are several shortcuts used for an input text taken from the command-line: Before reading the frame, all occurrences of the character '`;`' on the command-line are replaced by newlines, and all occurrences of '`,`'

are replaced by spaces. This does not apply to quoted strings inside the input. Moreover, any resulting input line that starts with a digit is taken for a matrix line, and any line starting with a letter or one of '`./~`' is taken for an include line. In particular, an input '`U24`' given as an argument to the program means to read the file '`U24`' (the same as the "full syntax" '`< U24`').

Using these shortcuts, one may give the whole input matrix on a line. Moreover, one may combine the matrix input with commands or options, and even with include lines (the matrix is then a concatenation of all input matrix lines):

```
bash$ macek -pGF5 '!print; 1 2; 3 4'
bash$ macek -pGF5 '1,2;!print;3,4'
bash$ macek -pGF5 'U24; 0 2; 3 4 ;!print'
```

In this context we remark that one cannot input a list of matroids like '`U24; U25; U35`' since this would result in concatenation of all three matrices. On the other hand, another shortcut described later allows to simply input a list of matroids as subframes. Section 4.4 [Subframes], page 15.

```
bash$ macek -pGF5 '!prtree;!print ((S))' '{U24 U25 U35}'
```

4.2 Matrices and their Entries

The program works with matrices representing matroids in the standard reduced form, i.e. without the leading unit submatrix. In this way the rows of the matrix correspond to the elements of a selected basis, and the columns to the remaining matroid elements. Before reading on here, it is good to understand (partial) fields; See Section 3.6 [P-Fields], page 9. We provide a more theoretical overview of matrix representations and their (in)equivalence in the next section.

This part provides a detailed description of matrix entries in frames. In general, the expressions on the n-th matrix line of the frame input give the entries of the n-th row of the matrix. (Modulo possible use of the '`@transpose`' option.) On one matrix line, the k-th expression gives the entry in the k-th column. The total number of rows of the resulting matrix is equal to the number of input matrix lines, and the total number of columns is equal to the maximal number of expressions over all matrix lines. Entries that are not directly given (like the rest of a short matrix line) are filled with zeros.

```
bash$ macek '!print' ' 1 0 1; 0 0 0 0 1'
bash$ macek ' 0;!print; 1 0 1; 1 0 0 0 1'
bash$ macek '!print; 0; 1 0 1; 1 0 0 0 1;@transpose'
bash$ macek '!print' ' 1 0 1 1;@transpose; 0 0 1 0 1'
```

An *expression* on the matrix line is a sequence of characters with no spaces. Spaces separate one expression from another. The *atomic expressions* describe generators of the partial field, and depend on its selection Section 3.6 [P-Fields], page 9. Moreover, one may use a special symbol '`o`' for zero. An expression is then built up from atomic expressions using parentheses '`()`' and symbols '`+-`', '`*/`', and '`^`' for arithmetic operations in the natural way.

It is best to illustrate matrix expressions by several examples. Keep in mind that not all expressions are defined in partial fields, so they may result in an error message.

```
bash$ macek -pGF4  '!print' ' 1+w^2 w*(w+w^3) 1+w+w*w'
bash$ macek -pGF5  '!print' ' 1+1+1 2^2+3^3+4^4'
bash$ macek -pNREG '!print' ' (a^4-a^3)^2 ((a-1)^2+a-1)*(a^3-a^2)'
```

Notice that some atomic expressions may look similarly like arithmetic operations. A good example is the generator $a-1$ of the near-regular partial field. In such a case we use brackets '`[a-1]`' for the generator. (This is an important change from the pre-1.0 versions of Macek which scanned '`a-1`' as an atom, creating confusion in expressions like $-a-1$.) However, an input like '`(a-1)^2`' is still correct – this is evaluated as arithmetic subtraction and power. If you are still confused, then use more parentheses.

Since we want to use one input file to represent the same matroid over different partial fields, we need a way to replace "transcendental elements" in the general representation matrix by specific field elements. We also need to ensure that the current partial field has necessary algebraic properties to represent the matroid. See Section 5.3 [Substitution], page 22.

There is an option '`@replace`' that accomplishes the first task. It is used as '`@replace X (a-1)`' to replace all further occurrences of the symbol '`X`' on the matrix input with the expression '`(a-1)`'. (We suggest to use parentheses to avoid confusion after a replacement.) Similarly, an option like '`@repl-PF X (a^2)`' replaces the symbol '`X`' only in one specific partial field `PF`. This pfield-specific replacement has priority. The symbol replacement is fully recursive, and it may be used only in matrices. One may prevent a symbol from a recursive replacement by prefixing it with the underscore '`_`', which is otherwise silently ignored on matrix lines.

Another option '`@require`' checks necessary algebraic properties of the current partial field. Use it as '`@require a+1 [.01]`', where the version with one expression followed by a dot checks whether the expression is defined over the current partial field, and the version with the second parameter as '`0`' or '`1`' check whether the expression value is zero or nonzero.

As an example we show the Fano matroid with a requirement of characteristic 2.

```
@comment "the matroid F_7 Fano"
@require 1+1 0
        1       1       1       0
        1       1       0       1
        1       0       1       1
```

Another example shows use of the symbolic replacement to give a representation of the matroid $U24$ in various finite fields. (Notice that if -1 was substituted for X over $GF(4)$, then the second requirement would fail.)

```
@replace    X    -1
@repl-GF(4) X     w
@require    (X)   1
@require    (X)-1 1
        1       1
        1       (X)
```

The next calls show the use of '`_`' in preventing recursive replacements. (The first call obviously results in an error.)

```
bash$ macek -pGF4 '@replace w w+1;!print' U24
bash$ macek -pGF4 '@replace w _w+1;!print' U24
```

Some common matroid representations are distributed with this package in the '`exe/Matrices`' subdirectory. We do not list them here since more matroids are added frequently. Look at the file listing of '`exe/Matrices/ *`' to see all of them. Each distributed matroid file is commented. You may easily create your own matroid files in a similar fashion. We suggest to enter new matroid representations (if possible) as highly symmetric matrices – see the command '`!selfmap`'. A symmetric matrix may help some algorithms to run faster.

If you create new matroid files that may be interesting and useful to others, please send them to the author.

4.3 Matroid Representations

In general, a matroid representation is a matrix whose columns represent the matroid elements, and usual linear dependency determines the dependent sets. However, it is better to work with a representation in the so called *standard reduced form*, obtained as follows: Choose a basis of the matroid and display it as a (maximal) unit submatrix. Then forget possible remaining zero rows and the columns of the unit submatrix. Finally, the rows of this standard-form matrix correspond to the elements of the selected basis, and the columns correspond to the remaining matroid elements. In this manual, we simply say a *matrix* instead of a "standard-form matrix".

There is no way to give names to the matroid elements – matrix lines, but the lines initially receive number labels as follows: The rows (elements of the selected basis) are numbered from 1 to R, and the columns (remaining elements) are numbered from -1 to $-C$. If a command later changes the matrix in a way that the lines move elsewhere, the labels are moved along with them. (See, for example, in the commands `!dual` or `!pivot`.)

When giving a matrix over a partial field, it is important to ensure that the matrix is proper — that means all subdeterminants are defined over this partial field Section 3.6 [P-Fields], page 9. Only then it is guaranteed that no arithmetic error occurs during program execution, and that the results are correct. See Section 6.6 [Structural], page 28.

Moreover, some commands in the program require connectivity of the given matroid to compute the result correctly. Some commands even need the input matroid to be 3-connected. In such cases you should enter only sufficiently connected matroids to avoid error messages or, even worse, incorrect results. So read carefully the description of commands below.

Another major problem with using the Macek program is caused by an existence of inequivalent representations of matroids. Two matrices (with lines labeled by the matroid elements) are called *strongly equivalent* if one can be transformed to the other one using elementary matrix operations. Two matrices are, on the other hand, called *unlabeled equivalent* if they are strongly equivalent up to an isomorphism of the underlying matroid. (In other words, if they are strongly equivalent after forgetting the line labels.)

For example, the following two quaternary representations of the matroid $U24$ are not strongly equivalent, but they are unlabeled equivalent:

```
1   1          1   1
1   w          1  w+1
```

You have to thoroughly consider the **problems with inequivalence** of representations when dealing with matroid minors, equivalence, or when generating matrix extensions. The Macek program has, so far, no way to find out that two inequivalent matrices actually represent the same matroid. (There are, however, no such problems at all when working with binary or regular matroids only.) Read the next chapters for specific information. See Section 6.6 [Structural], page 28, See Section 6.7 [Generating], page 30.

4.4 Subframes

A *subframe* starts with the keyword '`SUBFRAME`' on a separate line, and ends with the keyword '`EOFRAME`'. All input between these keywords is read into a new son of the current frame. There may be arbitrarily many subframes given, and arbitrarily nested, forming thus a *rooted-tree structure*. One may also use shortcuts '`{`' and '`}`' for start and an end of a subframe.

If the shortcuts '`{`' and '`}`' are used, then more than one (or even all of them) may be written on the same line. Moreover, all other words on such a line are taken as a separate subframe each. For example, the following shortcut produces the next tree of frames.

```
bash$ macek -pGF4 '!prtree;{ U24 {{U24 U25} {U35}} }'
~944~   Printing the subtree of the frame 0x81551f8 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [U24] m2x2  "the matroid U_2,4 uniform"
    ~         (3.1)fr  [U24-1]   ""
    ~           (4.1)fr  [U24] m2x2  "the matroid U_2,4 uniform"
    ~           (4.2)fr  [U25] m2x3  "the matroid U_2,5 uniform"
    ~           (4.3)fr  [U35] m3x2  "the matroid U_3,5 uniform"
```

When reading the program input, the first frame argument on the command line forms the root of the frame tree, and all possible other frame arguments are then arranged as its sons (in addition to its subframes). One may combine the concept of subframes and multiple frame arguments like in the following example.

```
bash$ macek -pGF4 '!prtree' '{ U24 U25 }' U35
~998~   Printing the subtree of the frame 0x8155b28 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [noname-1]   ""
    ~         (3.1)fr  [U24] m2x2  "the matroid U_2,4 uniform"
    ~         (3.2)fr  [U25] m2x3  "the matroid U_2,5 uniform"
    ~       (2.2)fr  [U35] m3x2  "the matroid U_3,5 uniform"
```

4.5 Addressing the Frame-tree

So far, we have shown only default parameter addressing in frame commands. However, one often needs to address arbitrary frames in the subtree, not only the pre-defined ones. Notice that some commands require precisely one frame in the parameter, while most of them accept an arbitrarily long list of input frames. This section shows the syntax of the *parameter addressing*. It is possible to skip this section until you get to the chapter Chapter 6 [Commands], page 25.

In general, nodes in the frame tree are addressed using natural correspondence between rooted trees and *balanced bracketings*. In this interpretation, '`()`' means the current (root)

frame — that one holding the command, and '`(())`' means the first son of the current one. You can see that the frame addressing is relatively rooted at the current frame. There is no way to address frames out of the current subtree. To actually point to a node in the tree, one must give a letter '`t`' or '`T`' in the bracketing. Alternatively, a letter '`s`' or '`S`' picks all sons of the pointed node instead. The difference between the letters '`t,s`' and '`T,S`' is that the lower-case letters request to erase the picked nodes after processing certain commands, while the upper-case letters do not. The current frame is never erased.

We now provide several sample addresses to illustrate the concept. The address '`(T(T))`' picks both the root and its first son. The address '`(()()((T)))`' picks the son of the son of the third son of the root. To save repetitions in the address, one may use numbers; '`((2)((T)))`' is equivalent to the previous example. The address '`((3)(5T))`' picks five sons of the root starting with the fourth one (the first three are skipped). Moreover, you may use, instead of the repetition number, a text '`/name/`' which skips sons up to the one named `name`. If the frame tree does not contain the requested nodes, an error message is printed.

Special rules concern non-positive repetition numbers. If '`(N...)`' is used, and $N <= 0$, then the repetition number actually used in $N + D$, where D is the number of remaining sons of the parent of the current node (inclusive). For example, '`((0T))`' has the same effect as '`(S)`', while '`(()(-1T))`' picks all sons of the root except the first and the last one.

In addition to the previous, you may concatenate more than one address to one with '`+`', like '`((t))+((t))`' picks the same first son twice. To save typing of closing brackets, you may end the address expression simply with '`|`', like '`(((()(T(s|`'. As a special concept, you may write the (whole) address as '`~1`' to pick all resulting frames of the previous command, or '`~N`', $N = 1, 2, ..., 9$ to pick the resulting frames of the N-th previous command. Similarly '`^N`' picks the previous results and allows them for further erasing, depending on the command. (The same thing as '`T,S`' versus '`t,s`'.) Be careful not to erase these frames before using them again.

Some commands have also output parameter address which tells them where to store the resulting frames. The output address starts with '`>`' and continues with the same bracketed expression as described above. However, this time all nonexistent nodes from the address are automatically created (no error is reported). Another exception concerns non-positive repetition numbers; if they are followed by a `t,T,s,S` letter, then they refer to the number of remaining output frames instead to the remaining sons in the tree. For example, '`((-1t))`' stores all but the last output frames as (new) sons of the root.

To learn the concept of parameter addressing well, it is best to play with the command '`!move`' which moves (copies, deletes) nodes across the frame tree. We provide several examples. Make more examples yourself.

```
bash$ macek '!prtree;!move ((T)) >(()(t));!prtree' 'W3;{}{}'

~052~   Printing the subtree of the frame 0x8155048 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spok"
    ~         (3.1)fr  [W3-1]   ""
    ~         (3.2)fr  [W3-2]   ""
~052~   ------------------------------------
~052~   Printing the subtree of the frame 0x8155048 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spok"
    ~         (3.1)fr  [W3-1]   ""
    ~         (3.2)fr  [W3-2]   ""
    ~       (2.2)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spok"
~052~   ------------------------------------


bash$ macek '!prtree;!move ((S)) >(()(s));!prtree' 'W3;{}{}'

~153~   Printing the subtree of the frame 0x81553b8 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spokes"
    ~         (3.1)fr  [W3-1]   ""
    ~         (3.2)fr  [W3-2]   ""
~153~   ------------------------------------
~153~   Printing the subtree of the frame 0x81553b8 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spokes"
    ~         (3.1)fr  [W3-1]   ""
    ~         (3.2)fr  [W3-2]   ""
    ~       (2.2)fr  [W3-0]   ""
    ~         (3.1)fr  [W3-1]   "fr #1 got by '!move ((S))', to '(()"
    ~         (3.2)fr  [W3-2]   "fr #2 got by '!move ((S))', to '(()"
~153~   ------------------------------------


bash$ macek '!prtree;!move ((t));!prtree' 'W3;{}{}'

~298~   Printing the subtree of the frame 0x8155128 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spokes"
    ~         (3.1)fr  [W3-1]   ""
    ~         (3.2)fr  [W3-2]   ""
~298~   ------------------------------------
~298~   Printing the subtree of the frame 0x8155128 [noname]:
    ~     (1.1)fr  [noname]   ""
~298~   ------------------------------------
```

```
bash$ macek '!prtree;!move (T) >((()(t)));!prtree'

~379~   Printing the subtree of the frame 0x8155050 [noname]:
    ~     (1.1)fr  [noname]   ""
~379~   ------------------------------------
~379~   Printing the subtree of the frame 0x8155050 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [noname-0]   ""
    ~         (3.1)fr  [noname-0]   ""
    ~         (3.2)fr  [noname]   "fr #1 got by '!move (T)', to '(("
~379~   ------------------------------------

bash$ macek '!prtree;!move ((T|+((T| >((()((0t|;!prtree' 'W3' ''

~164~   Printing the subtree of the frame 0x8155208 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spokes"
    ~       (2.2)fr  [noname-2]   ""
~164~   ------------------------------------
~164~   Printing the subtree of the frame 0x8155208 [noname]:
    ~     (1.1)fr  [noname]   ""
    ~       (2.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spokes"
    ~       (2.2)fr  [noname-2]   ""
    ~         (3.1)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spok"
    ~         (3.2)fr  [W3] m3x3  "the matroid W_3, wheel of 3 spok"
~164~   ------------------------------------
```

4.6 Macro Substitutions

The Macek program provides also simple text-based macro processing. When you write '`$macro`' or '`${macro}`', then this expression is replaced with the first word (use quotas for longer text with spaces) following the latest option '`@sub-macro`'; Section 5.3 [Substitution], page 22. If no such option is found, then the first word of '`@subd-macro`' is taken. If even this is not found, then the replacement text is empty. To input the character '`$`' itself, use '`\$`' or '`$$`'. Under normal circumstances, the macro processing is not recursive. This is a different concept than '`~1,^1`' shortcuts described in the previous section.

This kind of macro-processing is provided only for command-, option-, and include-lines. For replacements in matrix entries use '`@replace`' described above Section 4.2 [Matrix-Entry], page 12.

Again, it is best to illustrate the use of macros in several examples.

```
bash$ macek '@sub-mac ABCDefgh;!prtext $mac'
395~    ABCDefgh
```

```
bash$ macek '@sub-mac "ABCD efgh .. WXYZ";!prtext $mac'
457~    ABCD efgh .. WXYZ

bash$ macek '@sub-mac TEXT;@sub-mac "X-$mac-$mac-X";!prtext $mac'
519~    X-TEXT-TEXT-X

bash$ macek '@sub-mac TEXT;@sub-mac "X-$mac-$$mac-$mac-X";!prtext $mac'
519~    X-TEXT-$mac-TEXT-X
```

More involved and nonstandard examples are provided here. Be careful when using text macros in this nonstandard way since strange things may happen... (Like an error in the last example.)

```
bash$ macek -pGF4 '@sub-line "U24;{U35}";$line;!prtree'
273~ Printing the subtree of the frame 0x80f6f38 [U24]:
   ~   (1.1)fr 0x80f6f38 [U24] "the matroid U_2,4 uniform"
   ~     (2.1)fr 0x8116610 [U35] "the matroid U_3,5 uniform"

bash$ macek '@sub-line "!prtree";$line' R10
    ~   (1.1)fr 0x80f8918 [noname] ""
    ~     (2.1)fr 0x8103090 [R10] "the matroid R_10"

bash$ macek '@sub-a "$$b";@sub-b "$$a";$a'
... error happens ...
```

5 Frame Options

An *option* starts with the keyword '`option`' or the character '`@`' on a line. The next word is the option name, and option values continue after the name. Various options have various numbers of values, and an arbitrary number of option lines of the same name may appear. Quote the option values if they contain spaces or other special characters.

In general, an option holds arbitrary information that we want to store in the input frame. This information affects the input frame immediately from the line containing the option. Moreover, depending on particular situation, option values may be inherited by all subframes of the frame holding this option.

To learn more, read next about particular option groups recognized by the Macek program. The list of all recognized options is obtained by calling '`macek -HHo`'. (Not all of them may be described in this manual.) If an unknown option or the wrong number of values are given, then an error message is reported.

5.1 Inheritance of Option Values

An option affects the frames it appears in, and, in most cases, it also affects all descendant frames. However, in some situations we want to explicitly request repeating an option in the descendants, like when writing a frame to a file, or when generating new subframes in the program. (This does not concern the `name` and `comment` options which are repeated automatically, see below.)

For all options having names that are listed as the values of some '`@finherit`' option, the last one option instance is copied to every descendant frame when it is written to a file. The option '`@finheritall`' works similarly, but it copies all option instances to the written file.

Learn more about inheritance from the following examples. (View the results in the output file '`sample.mck`'.)

```
bash$ macek '@replace X 1;!writeto sample' '@replace Y 2'
bash$ macek '@replace X 1;!writeto sample' '@finherit replace'
bash$ macek '@replace X 1;@finheritall replace;\
      !writeto sample' '@replace Y 2'
bash$ macek '@replace X 1;@finheritall replace finheritall;\
      !writeto sample' '@replace Y 2'
```

The options '`@extinherit`' and '`@extinheritall`' achieve similar effects when new subframes are generated by extending a matrix; See Section 6.7 [Generating], page 30.

5.2 Naming the Frames

Each frame has its name, and optionally a text comment (about one line of a text). The name for a frame may be explicitly set by giving the option '`@name "frame-name"`'. Similarly, one may give a comment to the frame by '`@comment "some comment..."`'.

The frame comments are there just for user information, while the names are used elsewhere in the program, such as when writing frames to files. See Section 6.3 [FWriting], page 26.

5.3 Options for Substitutions

The Macek program provides two ways of replacing text on the input. The first one, called *substitution*, is used for option/command values, include files, etc. The second one, called *replacement*, is used in matrix entries or in field expressions.

When '`$macro`' appears on the input, then it is substituted with the first value of the last instance of the option '`@sub-macro`'. (Based on the current input scanned so far.) When '`@sub-macro`' option is not found, then '`@subd-macro`' is tried instead.

When an option '`@replace X (expr)`' is given in a frame, then all following appearances of the symbol '`X`' in matrix-entry expressions is replaced with the text '`(expr)`'. We suggest to use capital letters for such symbols. Similarly, an option '`@repl-PF X (expr)`' replaces the symbol '`X`' only when in the specific partial field `PF`. This pfield-specific replacement has priority.

You may prevent a symbol from a recursive replacement '`@replace X (expr+_X)`' by prefixing it with the underscore '`_`', which is otherwise silently ignored on matrix lines. To ensure that the replaced values satisfy your requirements, you may use the option '`@require (expr) [01.]`'. See Section 4.2 [Matrix-Entry], page 12.

5.4 Adding and Erasing Options

All frame options are read and interpreted before command processing starts, but it is still possible to add more options to selected frames during command execution using the command '`!append "input" >dest`'. See Section 6.4 [FReading], page 27. The intention of this command is to append the additional input to the selected frame — so the command '`!append "@option value" >(T)`' adds the given option to the current frame at the time when this command is executed.

It is not possible to delete options that were already scanned to a frame, but special options '`@erase optname`' and '`@eraseall optname`' are provided to suppress the last or all previous occurrences of the selected option(s). This means that the suppressed option will not be interpreted further. (However, the option still remains in the option list of its frame.) You may also use a combination like '`!append "@eraseall optname" >(T)`' to erase the selected option values via a command.

Try to play with the next examples to learn more:

```
bash$ macek '@sub-x AB;@sub-x $x-C' '!prtext $x'
bash$ macek '@sub-x AB;@sub-x $x-C;@erase sub-x' '!prtext $x'
bash$ macek '@sub-x AB;@sub-x $x-C;@eraseall sub-x' '!prtext $x'
bash$ macek '@sub-x AB;@sub-x $x-C' '@erase sub-x;!prtext $x'

bash$ macek '@sub-x A;!append (T) "@sub-x $x-B;!prtext $$x"'
bash$ macek -pGF4 '!extend b;!append (T) \
      "@ext-forbid U25";!extend b >(()(s));!prtree' U24
```

5.5 Options for Generating Extensions

The program provides commands for generating extensions of matrices, See Section 6.7 [Generating], page 30. These commands need to keep additional information about the matrices during the generation process, which is achieved using the various `ext-*` options.

The option '`@ext-bsize R C`' tells that the base minor of generating process occupies the first `R` rows and `C` columns of the matrix. The option '`@ext-signature`' keeps the signature of the elimination sequence of generating process. (The signature tells, in a bit-representation, whether rows 0 or columns 1 were extended at each step of the process.) These options are set during the generating process, and they should never be altered by hand.

The option '`@ext-forbid min1 min2 ...`' lists matroids which are forbidden as minors when generating extensions. Each value of the option represents one forbidden minor, and these values are processed in the same way as the input frames in the program. Notice that this value-processing happens later, when the respective '`!extend`' command is executed. For example, the option '`@ext-forbid F7 "F7;!dual"`' means that the matroid $F7$ and its dual will be excluded in the next extension-generating command.

The option '`@ext-tight min1 min2 ...`' lists matroids which form the set defining a "tight-major" for generated extensions. This option works similarly as `ext-forbid`. The option '`@ext-nofan f`' tells that no fan of length f or longer may appear along an elimination sequence when generating extensions. Notice also the options '`@extinherit`' and '`@extinheritall`' that control inheritance of other options in the generated extensions Section 5.1 [Inheritance], page 21.

5.6 Other Options

The option '`@nopfcheck`' causes the program to skip the partial field test. If the partial field currently used in the program is not total, then not all matrices are valid Section 3.6 [P-Fields], page 9. So a randomized quick test is usually run to uncover most of undefined matrices. However, for a very long input you may want to skip even this quick test.

The option '`@transpose`' immediately transposes the matrix scanned so far. If more matrix lines appear after this option, then they continue the transposed matrix. Usually you would use this option to obtain a dual matroid from an existing one. This option is never inherited or written to files.

```
bash$ macek '!print (S)' 'grK5' 'grK5;@transpose'
```

The option '`@inputpf PF`' immediately switches the input partial fields to new `PF`. That means all subsequent matrices in the subtree of the current frame are read and stored in the new partial field. See Section 6.8 [Diff-fields], page 33. No other frames than the current one and its descendants are affected. Do not mix different partial fields for one matrix.

```
bash$ macek -pGF2 '!print ((T));!pfield GF3;!print (()(T))' \
       U23 '@inputpf GF3;U24'
```

6 Frame Commands

A *command* starts with the keyword '`command`' or the character '`!`' on a line. The next word is the command name, and command parameters continue after the name. For example, we write a frame to a file with '`!writeto filename ((T))`'. It is possible to quote parameters with spaces '`"long parameter text"`'.

When reading the input, commands are scanned and immediately stored into their frames. However, they are executed (in order) later, after the whole input is scanned. If more than one frame of the input tree holds commands, then the frames are processed in the reversed depth-first order. In particular, descendants are processed before the root. Each command can access only the frame it is stored in, and its descendants. (Like if this frame was the root of the whole tree.) Usually, you should give all commands in the root frame.

6.1 Command Overview

When using commands and options in the Macek program, it is important to fully understand the order in which options and commands are scanned and applied, as written above. The later execution of commands, in particular, means that all matrix entries and all options from the input are already known when the command is executed, even if they appear after this command. It does not matter how you mix between commands and options on the input.

Yes, there is an exception to the previous rule – the `@sub-*` options vs. command parameters. Keep in mind that the input macro substitution applies when reading the command, not when executing it. Therefore, all `@sub-*` options must appear before the respective macros are used on the input.

Also notice that some options, like '`@transpose`' or '`@inputpf`', behave more like commands, but they still stay as options with their immediate application. (You may consider using the corresponding commands '`!dual`' or '`!pfield`', respectively.)

Before proceeding further, be sure that you understand about frame-addressing in command parameters; See Section 4.5 [Addressing], page 15.

A command has several or no parameters. (Among them, the possible output address '`>xxx`' has special meaning and position.) If the required parameter is missing, then it is substituted by its default value. Determining the default value is a kind of a magic; it depends on a command context, current frame tree, etc. If you are not sure what the default value in the specific situation is, then read the program output where the parameters are printed, or add the command '`!verbose`'.

run the program with `-g2` and see the substitution made in the debugging output.

The purpose of this "magic" parameter substitution is to save you typing the parameters over and over. In most easy situations you may just use the commands with no parameters at all, and the default values will do what you expect. (Unless your expectations are very unrealistic.)

Another note concerns frame names and comments. Some commands change the frame name to indicate their effect on the frame (matrix). They also may set the frame comment to a description of the command.

To learn more, read next about particular commands recognized by the Macek program. The list of all recognized commands is obtained by calling '`macek -HHc`'. (Not all of them may be described in this manual.)

6.2 Printing Commands

We start with the printing commands in the program. The command '`!pr`' prints a simple description of the given frame(s). The command '`!print`' similarly prints the matrix(ces) of the given frame(s), and '`!prtext`' prints the text given as a parameter.

The command '`!prmore`' prints various additional matroidal information about the given matroid(s). The printed information does not depend on particular representation or the partial field, only on abstract matroid properties. This may be used to better understand the structure, or to compare matroids over different pfields by hand (using a hash-value printed at the end). Currently, numbers of bases are printed out, and small flats and separations are listed; all depending on the current output verbosity by '`!verbose`'.

To display the whole subtree of the given frame(s), use the command '`prtree`'. Each descendant frame (up to certain depth) is printed on a separate line, in the depth-first order.

In general, one may control the amount of information printed out with the commands '`!verbose`' and '`!quiet`', that can be used with a number `+-N`.

Try the following examples:

```
bash$ macek -pGF4 '!pr (s)' U24 U25
bash$ macek -pGF4 '!quiet;!pr (s)' U24 U25
bash$ macek -pGF4 '!prtree' U24 '{U25 U35}'
bash$ macek -pGF4 '!prtree (s)' U24 '{U25 U35}'
bash$ macek -pGF4 '!print ((t)(s))' U24 '{U25 U35}'
bash$ macek -pGF4 '!verbose;!print' U24
bash$ macek -pREG '!prmore' R10
bash$ macek -pGF3 '!verbose 2;!prmore' F7-
```

6.3 Writing to Files

The commands described here are provided for saving (writing) frames to files. One may write either one frame (including its matrix), or the whole subtree of a frame. The format of the output file is a text described in the section Section 4.1 [Frame-syntax], page 11. (The syntax is likely more formal than what you would give to the program on a command line, but the general rules are the same.)

When writing (or reading) files, search paths are used, See Section 3.5 [Environment], page 9. Each file name is automatically appended with the extension '`.mck`' (if it is not given otherwise). If no file name is specifically given to the command, then the name of the frame is used.

The command '`!write (frame)`' writes the frame addressed by `(frame)` to a file named by the frame. More than one frame may be given. The command '`!writetree (frame)`' writes the whole subtree of the frame addressed by `(frame)` to a file named by the frame. The commands '`!writeto fname (frame)`' and '`!writetreeto fname (frame)`' do the same, but they use a name '`fname`' for the saved file. If '`fname`' ends with a slash '`/`', then it is used as a directory prefix for writing, and the frame name is used for the file itself.

Possible commands contained in a frame are never written to a file. On the other hand, all options except `@transpose` are written, and even some options inherited from ancestors may be written if requested Section 5.1 [Inheritance], page 21. The exception is the `@name` option which is not written for the root frame, so that this frame later gets its name from the file name.

Again, we provide few examples:

```
bash$ macek -pGF3 '!writeto sample1 ((T))' 'W3'
bash$ macek -pGF3 '!writetreeto sample2 (T)' '{U24 W3}'
bash$ macek -pREG '!extend c;!write ((S))' grK33
```

6.4 Reading Frames

Under normal circumstances, you do not need any command to read frames, since the input frames are scanned with the input. However, in some cases extra commands are necessary; like if you want to read a frame in another partial field than the current one, or if you want to add more options to frames after scanning the input.

The command '`!read input >(dest)`' reads a frame (sub)tree from the string `input`, and stores the tree as rooted at the position `(dest)` in the current tree. The string `input` is considered similarly as a command-line argument to the program, including use of the line shortcuts as described in [line-shortcuts], page 11.

The command '`!append (fram) input`' reads the given text `input`, and appends it to the given frame `(fram)`. The appending works as if the given text continued the original input stream of the frame, but you must understand that many other things may have already happened from the original input scanning, which may result in unexpected effects. In general, we suggest to use this command only in situations when you want to add more options to some frame during command executions, or if you want to add additional commands to the current root frame. (However, if you add commands to descendant frames, they never get executed again unless `!restart` is used.)

6.5 Manipulating Frames and Matrices

We provide two commands for rearranging the frame tree in the program. The command '`!move (src) >(dest)`' moves (copies, or deletes) the given source frames addressed by `(src)` to the destination positions addressed by `(dest)`. In accordance with the addressing convention Section 4.5 [Addressing], page 15, the source frames are copied if they are selected with '`T`' or '`S`', and they are moved if selected with '`t`' or '`s`'. If the destination parameter is not given, then the selected frames are deleted from the tree. When deleting a frame with descendants, the whole subtree is disposed of. The root frame (of this command) cannot be deleted.

The command '`!flatten (src) >(dest)`' collects all descendants of the frames in `(src)`, and stores them in the positions addressed by `(dest)`. The command '`!mmove (src) >(dest)`' is similar to `!move`, but it moves only the matrix, and no other frame attributes. The command '`setname name (frame)`' sets a new name to the given (one) frame.

Many examples of the '`!move`' command are presented in Section 4.5 [Addressing], page 15. We provide a few more here:

```
bash$ macek '!move ((T))+((T)) >(()(S));!prtree' grK33
bash$ macek '!flatten (T) >(()(s));!prtree' '{grK5 grK33}'
bash$ macek '!flatten (s) >((s));!prtree' '{grK5 grK33}'
```

Besides the tree-manipulating commands described above, we provide a bunch of commands for manipulating matrices in the frames.

The commands '`!dual (mat)`' transposes the matrix(ces) in the given frame(s) `(mat)`. Its effect is similar to the option '`@transpose`' applied after the matrix, but the important difference follows from the fact that options are applied immediately while commands are executed later, Section 5.6 [Other options], page 23.

The command '`!pivot row col (mat)`' pivots the given matrix in `(mat)` on the entry at `row` times `col`. (Rows and columns are numbered in order from 1.) The pivoted entry must be nonzero. Pivoting switches the labels of the pivoted row and column. The resulting matrix replaces the previous one (in the same frame).

The command '`!delete lab (mat)`' deletes the element of the label `lab` from the matroid represented by the given matrix in `(mat)`. Note that, unlike when pivoting, the elements are identified by their *labels*, not by their order! This allows to delete (as a matroid element) not only columns of the matrix, but also rows after (automatic) pivoting. The command '`!contract lab (mat)`' similarly contracts the given element.

The command '`!deleach (mat) >(dest)`' creates a list of new frames with matrices obtained by deleting each one of the elements of the matroid represented by the given matrix in `(mat)`. The resulting new frames are stored according to the output address given in `(dest)`. The command '`!coneach (mat) >(dest)`' works similarly for contractions. The command '`!remeach (mat) >(dest)`' produces all one-element deletions and contractions of the matrix (as the previous two together).

```
bash$ macek -pGF2 '!print (S)' grK5 'grK5;!dual'
bash$ macek -pGF5 '!print;!pivot 2 1;!print' P8
bash$ macek -pREG '!print;!delete 2;!print' grK5
bash$ macek -pGF2 '!coneach;!print ((S));!prtree' W3
bash$ macek -pGF4 '!remeach;!print ((S))' U24
```

6.6 Structural Matroid Functions

We start with the command '`!inpfield (mat)`' which checks whether the given matrix(ces) `(mat)` is proper over the current partial field Section 3.6 [P-Fields], page 9. The result is printed out. This command has no meaning for normal fields.

```
bash$ macek -pREG '@nopfcheck;!print (S);!inpfield (S)' \
        ' 1 1; 1 0' ' 1 1; 1 -1'
```

Another command '`!mhash h-value (mat)`' is used to find matroids in the list `(mat)` which have the given matroid hash-value `h-value`. This is the same hash-value as computed and printed in '`!prmore`' Section 6.2 [Printing], page 26. The value is matroid-invariant, and so it may be used to informally compare distinct matroid representations, even over different partial fields. Non-equality guarantees that the matroids are not isomorphic. The next example works when using matroids hash-value ver 1.0. If the program is upgraded to a higher hash version, you have to adjust these examples first.

```
bash$ macek -pRoot6 '!prmore;!pivot 1 4;!prmore' O7
bash$ macek -pGF3 '!mhash 13068150 (S)' O7 P7
```

Next we describe a collection of minor-structural commands. (You should first understand problems concerning inequivalent matroid representation from Section 4.3 [Representations], page 14.) If you want to see more about the command result (like where the minor is displayed, etc.), use the command `!verbose` before Section 6.2 [Printing], page 26.

The command '`!minor (mat) (min)`' finds out whether the given matroid(s) in `(mat)` has a "minor" in the given list `(min)`. Here by M having a *minor* N we mean that some strongly equivalent matrix representation of M displays a submatrix which is unlabeled equivalent to N. So when asking for a minor N in the matroidal sense, one has to give **all representations** of N up to unlabeled equivalence in the list `(min)`. The minor (if found) can be displayed when '`!verbose`' printing was requested before. In such case a submatrix of an equivalent matrix of `(mat)` that is equal to `(min)` up to scale is printed out. With '`!verbose 2`', all displayed minors are printed out.

The command '`!equiv (mat1) (mat2)`' looks for unlabeled equivalent pairs of matroids in the given lists. (In general, equivalence testing is much faster than minor testing.) The command '`!eqpair (mat)`' is similar to the previous one – it looks for each matroid in the list whether some other matroid further in the list is equivalent to this one.

The command '`!tmajor (mat) (min)`' tests whether the matroid in `(mat)` is a "tight major" of the given list `(min)` of matroids. In theory, a matroid M is a tight major of a family F if no element of M can be both contracted and deleted keeping a minor in F. For our implementation, the same notes as for `!minor` apply here. A warning is printed if `(mat)` itself has no minor in `(min)`.

```
bash$ macek -pREG '!minor' '{W4 R10 R12}' grK33
bash$ macek -pREG '!print;!verbose;!minor' R12 grK33
bash$ macek -pREG '!verbose 2;!minor' grK4 grK3

bash$ macek -pREG '!deleach;!equiv' R10 grK33
bash$ macek -pREG '!eqpair' 'R12;!coneach (T) >((0t))'
bash$ macek -pREG '!verbose;!eqpair' 'R12;!remeach (T) >((0t))'

bash$ macek -pGF2 '!tmajor' '{S8 R12}' '{F7 F7#}'
bash$ macek -pGF2 '!verbose 2;!tmajor' R12 grK33
bash$ macek -pGF2 '!extend bb;!tmajor ((TS))' F7 '{F7 F7#}'
```

The command '`!unique (mat1) (mat2)`' is a special function provided for "clearing" lists of matroids of isomorphic non-equivalent pairs. The idea is that the list `(mat1)` is that one to clean, and the second list `(mat2)` contains alternative representations of the matroids from the first list (in one-to-one correspondence). If these assumptions are not true, then an error is reported. Only those matroids of the first list are accepted that have no equivalent matroid further down in any of the two lists. Efficient use of this function requires a way of finding inequivalent representations of a given matrix — like the $w- > w+1$ automorphism of the field $GF(4)$.

To show an example of the command '`!unique`', we consider $GF(4)$ co-extensions of the matroid $P7$.

```
bash$ macek -pGF4 '!extend r;!prtree;!writetreeto p7ex ((T))' P7
```

This call generates 19 nonequivalent co-extensions of $P7$, and writes them to the file '`p7ex.mck`'. You may look at the generated matroids using '`!prmore`', and you find out from their structural properties that some non-equivalent pairs are likely to be isomorphic. Precisely, you find out these isomorphic pairs with the next call, where '`@replace w _w+1`' applies $w- > w+1$ automorphism to the matrices in '`p7ex.mck`'. Section 4.2 [Matrix-Entry], page 12.

```
bash$ macek -pGF4 '!unique' p7ex '@replace w _w+1;<p7ex'
```

Finally, we are left with several other structural commands. Use '`!bwidth3 (mat)`' to see whether the given 3-connected matroid(s) have branch-width at most 3, or higher. (It is yours responsibility to ensure that the tested matroid really is 3-connected!) Call '`!fan (mat)`' to print the longest fan found in the given connected matroid(s), and '`!hasfan f (mat)`' to see whether the matroid(s) has a fan of length at least `f`, Again, you may request printing the fan with '`!verbose`'.

```
bash$ macek -pREG '!remeach;!bwidth3 ((TS))' R10
bash$ macek -pREG '!extend r;!bwidth3 ((TS))' R12
bash$ macek -pREG '!verbose;!bwidth3 ((TS))' R12
bash$ macek -pGF2 '!fan' '{F7 W3 W4 R10 R12}'
bash$ macek -pGF2 '!verbose;!fan' '{F7 W3 W4 R10 R12}'
```

We provide commands for determining matroid connectivity. The command '`!connectivity (mat)`' prints the connectivity of given matroids (2,3,4,...). The commands '`!isconn (mat) c`' and '`!isconn3 (mat)`' check the required connectivity of given matroids. Notice that, unlike [Oxley], we do not consider the matroid $U23$ to be 3-connected. We define a matroid M to be n-connected, $n > 0$, iff M has at least $2n - 2$ elements, and M has no proper k-separation for $k = 1, ..., n - 1$.

For example, '`!isconn (mat) 4`' is passed by matroids that are at least 4-connected. Another example uses the 3-connectivity filter to prepare correct input to '`!bwidth3`' command.

```
bash$ macek -pREG '!remeach;!connectivity ((TS))' R10
bash$ macek -pREG '!remeach;!connectivity ((TS))' grK5
bash$ macek -pREG '!remeach;!filt-isconn3;!bwidth3 ((TS))' grV8
```

6.7 Generating Extensions

An important function of the Macek program is to generate 3-connected (co)extensions of a matroid over the partial field. All matroids we speak about here **must be 3-connected**. We refer also to the description of options used in the extension generating process Section 5.5 [Extensions], page 22.

The command '`!extend [rcb]+ (mat) >(dest)`' generates 3-connected (co)extensions to the matrix given in `(mat)` according to the first text parameter, and stores them in the given destination position `(dest)`. A letter `r` in the first parameter means to do a row coextension, a letter `c` means to do a column extension, and a letter `b` means to do both of them. (You would mostly use `b` here unless you know really well what you are doing.)

It is possible to combine multiple letters in the first parameter of `!extend`. For example, '`!extend bbb (mat)`' generates three steps of one-element extensions, and all results of each

of the steps are stored. It is also possible to give more than one matrix in the input list `(mat)`. Then the extensions are generated for each of the matrices. However, in such case it is not allowed to use multiple steps.

To extend the given one matroid to a specified size and rank, use the command '`!extendsize r c (mat)`'. This command repeats the extension steps until all extension matrices of dimensions `r` times `c` are produced. Unlike for `!extend`, the intermediate constructed extensions are not stored here.

Try the following few examples:

```
bash$ macek -pGF4 '!extend b;!prtree' F7
bash$ macek -pGF4 '!extend r (S);!prtree' F7 F7#
bash$ macek -pGF2 '!extend bbb;!prtree' F7
bash$ macek -pGF2 '!extendsize 5 5;!prtree' F7
```

Another set of examples shows the effects of additional generating attributes given by the `ext-` options Section 5.5 [Extensions], page 22.

```
bash$ macek -pGF5 '!extend;!prtree' U25
bash$ macek -pGF5 '@ext-forbid "U35";!extend;!prtree' U25
bash$ macek -pGF5 '!extend;@ext-forbid "U25;!dual";!prtree' U25
bash$ macek -pdyadic '!extend;!prtree' F7-
bash$ macek -pdyadic '!extend;@ext-nofan 4;!prtree' F7-
```

When generating extensions with '`@ext-forbid M`', only those extensions not containing an M-minor are created. This construction is equivalent to generating all extensions, and then filtering out those with an M-minor, but the above example is faster. On the other hand, the option '`@ext-nofan f`' is a *sequential* option – it restricts the appearance of the whole generating sequence, not only the resulting matroid. For example, a matroid N having no fan would not be generated with '`@ext-nofan 4`' if all sequences leading to N contain a 4-fan.

Warning!! The above described commands `!extend` and `!extendsize` are very complex in their nature, and one may easily produce "false" results when (s)he does not fully understand all the hidden details of the computation. That is why we provide here the following detailed explanation of the extension-generating algorithm in Macek. The theory behind our extension-generating algorithm is written in the paper [Petr Hlineny: Equivalence-Free Exhaustive Generation of Represented Matroids, submitted 2002]. However, you do not have to worry about most of the details if you are working only within "nice" partial fields with unique matroid representability like binary, regular, or ternary.

We assume that the reader is familiar with matroid representations and their (in)equivalence Section 4.3 [Representations], page 14. A matroid S is called a *stabilizer* for a given partial field if, for any 3-connected (or stable) matroid M with an S-minor, any two representations of M displaying the same subrepresentation of S are strongly equivalent. Moreover, S is called a *strong stabilizer* if every subrepresentation of S extends to whole M.

Consider that we want to generate all matroids having the given matroid (called further the *base minor*) S as a minor, subject to representability over the pfield and to other conditions (attributes). Then, by Seymour's splitter theorem, there is a sequence of single steps (extensions / co-extensions) building a matroid M from the base minor S keeping 3-connectivity; except the case when the base minor is a wheel or a whirl(!). Such a sequence

of single steps, when viewed in the reverse order (i.e. as deletions / contractions), is called the *elimination sequence* for the (resulting) matroid M over the base minor S.

Formally, an elimination sequence consists of the base minor S in the given matrix representation, of the resulting matrix for the matroid M, of the order of lines of M as they are added to S, and of the *signature* of the sequence telling which lines of M are extended and which are coextended. (The sequence signature is taken separately from the order since the order naturally follows from the matrix representation of M, while the signature does not.) The base minor S is always displayed in the matrix representation of M in the upper-left corner.

We call two elimination sequences *equivalent* if their base minors are unlabeled identical, they have the same additional attributes, and their resulting matrices are unlabeled equivalent. To avoid generating equivalent sequences repeatedly, we require the generated elimination sequences to be minimal with respect to the following *canonical order*: We compare two sequences lexicographically first by their signatures (preferring the signature bits corresponding to lines closer to the base minor), and then by their lines as they are added in the sequence (again preferring the lines closer to the base minor).

One important note concerns the use of letters `r`,`c` as modifiers of the '`!extend [rcb]`' command: These letters allow you to choose which extensions (row/column) are done at each step of generating, but they do not modify in any way the canonical order of sequences. In particular, if you specify '`!extend cr`', then you get only(!) those canonical extensions that happen to add a column before adding a row, but not those that add a row before adding a column.

After all, unless you are using "nice" partial fields with unique representability like binary, regular, or ternary; we suggest to generate extensions from strong stabilizers, to guarantee extendability and to limit inequivalent representations. Notice, however, that even when the base minor S is a strong stabilizer for the current partial field, two nonequivalent sequences may produce isomorphic matroids (to M) — this may happen if there are more (labeled) S-minors in M which display inequivalent representations of S.

Since the options `@ext-bsize` and `@ext-signature` describing the elimination sequence are stored with the generated extension matrices, one may continue the generating process in multiple steps while still keeping uniqueness of the generated sequences over the whole universe generated from S. This allows to continue the computation in parallel on many computers – make the first step(s), and then distribute each of the extensions to another computer.

However, never touch the `@ext-bsize` and `@ext-signature` options by hand, or(!) you twist the canonical order and lose extensions. Moreover, never change the matrix between generating steps for the same reason, and do not mix matroids of elimination sequences created in different major versions of the program. Do not even switch between partial fields during generating.

If you want to generate extensions of, say, two base minors $S1, S2$, then you have to manually exclude repetition of those extensions that contain both $S1, S2$. One easy way to achieve this is to generate the extensions of $S2$ with an additional option '`@ext-forbid S1`'. (Consider also the option '`@extinherit[all] ext-forbid`' to inherit the exclusion of $S1$ for the generated extensions.) You may learn more in the practical examples below Chapter 7 [Practical], page 37.

6.8 Working in Different Partial Fields

The Macek program allows to work in different partial fields than the given one by `-pPF` ⟨undefined⟩ [-p], page ⟨undefined⟩. The command '`!pfield NPF`' switches the program to temporarily use a new partial field `NPF`. The new partial field stays in effect until a new call to `!pfield`, or until the current frame execution is finished. Compare this command with the option '`@inputpf PF`', Section 5.6 [Other options], page 23.

However, calling `!pfield` only switches the program's internal arithmetic, but the matrices in the frame tree still remain represented over the original partial field. If you want to work with them in the new partial field, then you must first import them. For that purpose the '`!import transl (mat)`' command is provided. All entries of the matrices in the list `(mat)` are translated to the current partial field using the translation named `transl`. Read about partial-field translations in Section 3.6 [P-Fields], page 9.

After importing a matrix to a different partial field, this matrix may no longer be proper Section 3.6 [P-Fields], page 9. So, unless you are sure that this matrix is indeed proper, you should call the command `!inpfield` to test it.

```
bash$ macek -pGF4 '!print;!pfield GF2;!import Id0;!print' U24
bash$ macek -pNREG '!print;!pfield GF3;!import Nreg-tr;!print' P7
bash$ macek -pNreg '!extend r;!prtree;!pfield GF3;!import \
       Nreg-tr ((S));!equiv ((S)) ((S))' P7
```

6.9 Command-Flow Control

The Macek program provides simple command-flow control described here. The command '`!restart`' restarts command processing in the whole frame tree. (Commands already processed are deleted, so they are not executed again.) This command may be useful, for example, in connection with the command `!append` which adds new code to descendant frames. See Section 6.4 [FReading], page 27.

The command '`!skip n`' causes command processing to skip the next (up to) `n` commands in the current frame. Commands in other frames are not affected. The command '`!exit r`' immediately stops command execution, and returns the value `r` to the calling shell. See below for an example.

The command '`!iflist len [<=>!] (fram)`' is used to test whether the given frame-list `(fram)` contains number of frames comparable to the value `len`. One may use relations `=`, `!=`, `>`, `>=`, `<`, `<=`. If the relation is true, then the next command after `!iflist` is executed, otherwise the next command is skipped. If you want to skip more commands, use `!iflist` in combination with `!skip`.

Try the following simple examples.

```
bash$ macek '!iflist 0 < (S);!print (S)'
bash$ macek '!iflist 0 < (S);!print (S)' R10
bash$ macek '!iflist 0 = (S);!skip 3;!print;!extend;!prtree'
bash$ macek '!iflist 0 = (S);!skip 3;!print;!extend;!prtree' W3
bash$ macek '!iflist 0 = (S);!skip 2;!print;!extend;!prtree'
```

The next example is more involved. See that the command `!extend` is copied to all descendant subframes, and then it is executed in each one of these subframes after `!restart`.

```
bash$ macek '!append (S) "!extend c (T) >((0t))"\
       ;!restart;!prtree' W3 W4 R10
```

Another involved example is the procedure `&splitlist` (Section 6.11 [Procedures], page 34) distributed with the package:

```
#  - use in macek as  '{<list};&splitlist [length] [depth(]'
@subd-param1    10
@subd-param2    "(("
!move ${param2}(${param1}t)| >${param2})(s)|
!iflist ${param1} < ((S))
!append (T) "&splitlist $param1 $param2"
```

This procedure serves for breaking-up the given frame list into small pieces. Notice the command-flow in that procedure; first one small chunk of the list is moved to a new node, then the length of remaining list is tested, and, if longer than the given value, the whole procedure is appended again to the current frame. (The appended commands are automatically executed after `!append`.) An example of use is here:

```
bash$ macek '!prtree' '{<bw3-tern-exc};&splitlist 4'
```

The `!exit` command returns an integer value back to the calling shell. In `bash`, one may retrieve the returned value as follows:

```
bash$ macek -g-2 '!exit 123' ; echo $?
123
```

6.10 Command-output Filtering

Some of the above described commands that usually print a "yes/no"-type answer, may be modified by a prefix to filter the input list of frames. The prefix '`filt-*`' causes the command '`*`' to keep those frames for which the answer is "yes", and to delete the others. (Address the frames with `s` or `t`.) The prefix '`filx-*`' has the exactly opposite meaning.

The prefixes '`rem-*`' and '`rex-*`' suppress both printing and filtering in the command. The only result of such a modified command is the resulting list to be remembered for subsequent '`~N,^N`' parameter addressing, Section 4.5 [Addressing], page 15.

The commands that can be modified by these prefixes include `!minor`, `!tmajor`, `!inpfield`, `!isconn`, `!equiv`, `!hasfan`, `!bwidth3`, `!mhash`. Find the current list of all modifiable commands by calling '`macek -Hc`'.

Here are a few examples that illustrate these concepts:

```
bash$ macek -pREG '!deleach;!coneach;!minor' R12 grK33
bash$ macek -pREG '!deleach;!coneach;!filt-minor;!prtree' R12 grK33
bash$ macek -pREG '!deleach;!coneach;!rem-minor;!pr ~1' R12 grK33
```

6.11 Procedures – Collecting Commands

Often, one needs to execute a whole sequence of commands repeatedly for different parameter values. For this purpose the program provides the concept of *procedures*.

A procedure line starts with the keyword '`procedure`' or the character '`&`'. The procedure is written as '`&proc p1 p2 ...`'. When scanning input, such a procedure call is expanded

into the following actions: Substitutions are created as '`@sub-param1 p1`', '`@sub-param2 p2`', etc. Then the file '`proc`' is included into the place. It is assumed that this file contains a sequence of commands, using the parameter values as `$param1`, `$param2`, etc. The possible output parameter '`>out`' is accessed as `$paramres`. To give default values to the procedure parameters, use '`@subd-param1 p1-default`'.

See examples distributed with the program in '`Procedures/*`'...

To simplify single command-line calls to some mostly used Macek functions, we provide few shortcuts that are implemented as include files. You may use simple calls like the following ones:

```
bash$ macek print R10
bash$ macek -pGF4 print U35
bash$ macek print grK33 grK5
bash$ macek -pREG prints grK5
bash$ macek -pREG conn R10 R12
bash$ macek -pREG minor R10 grK33
bash$ macek -pREG equiv R10 grK33
```

The above shortcuts use the feature of an automatic file-include for command line arguments to Macek. So '`print`' is actually a file containing the print command, and similarly with others. See '`Procedures/shortcut/*`'. User may easily prepare more such shortcuts. However, we suggest to use shortcuts only in those very simple situations like the above examples.

7 Practical Macek Computations

In this section we want to present several practical examples of computations with the Macek program. They are intended both to show you the power of this program in practice, and to demonstrate that it computes correctly some well-known matroidal results. We start with proving that the matroid $R10$ is a splitter in the class of regular matroids (see Seymour's decomposition theorem).

7.1 $R10$ as a Splitter for Regular

```
bash$ macek -pREG '!extend b;!prtree' R10
```

This program call works in the regular partial field. A representation of the matroid $R10$ (distributed with the program) is read from a file, Then the command '`!extend`' is called to get all 3-connected row- and column-extensions of the matrix of $R10$ in regular numbers, using the default parameter address `((T))`. As you may immediately see, no extension is generated. (If there were some, they would be stored to `>(((0t)))`.) Thus, using Seymour's splitter theorem, $R10$ is a splitter for 3-connected regular matroids.

```
bash$ macek -pNREG '!extend b;!prtree' R10
```

Similarly, we show that $R10$ is a splitter for 3-connected near-regular matroids.

7.2 Extending $F7$ in Binary

```
bash$ macek -pGF2 '!extend b;!prtree;!minor' F7 F7#
```

In this case, we generate all binary extensions to the matrix of the Fano matroid $F7$. Then we print the two generated extension in the tree as sons of $F7$, and finally we show that both the extensions have the dual of $F7$ as a minor. Hence $F7$ is a splitter for binary matroids with no $F7*$ minor.

Alternatively, one may achieve the same result with another call that excludes the $F7*$-minor immediately in the generating process. (Notice that commands are executed even inside option parameters.)

```
bash$ macek -pGF2 '@ext-forbid "F7;!dual";!extend b' F7
```

```
bash$ macek -pGF2 '!extend r;!print ((s))' F7
```

We continue in the direction of the previous example. We compute the two binary row coextensions of the Fano matroid $F7$ and print them. See that the first one of them is the affine plane $AG(3,2)$, and the second one is known as $S8$. Both of these matroids are distributed with the program.

```
bash$ macek -pGF2 '!extend c;@ext-forbid "AG32;!dual"' S8
```

The next call shows that there is only one column extension to $S8$ with no $AG(3,2)*$-minor. This extension is known as $P9$.

7.3 Extending $K5$ in Binary

```
bash$ macek -pGF2 '!extend b;!minor' grK5 '{grK33,"grK33;!dual"}'
```

Here we generate all binary 3-connected extensions of the matroid of the graph $K5$, and then check which of them have the matroid of the graph $K33$ or the dual as a minor. We see that 5 out of 6 generated matroids have one of the minors.

```
bash$ macek -pGF2 \
      '@ext-forbid grK33 "grK33;!dual";!extend bbb;!prtree' grK5
```

Notice that this command should be called from one (logical) line of the shell. Now we compute the binary extensions to $K5$ in three steps, adding one element at each step. We immediately exclude those extensions with $K33$ or the dual as a minor. Since no new extension is generated at the third step, this computation proves that there are altogether only two binary 3-connected (row) extension to $K5$ without $K33$ or the dual as a minor. These two matroids are known as $T12$ and $T12/e$.

7.4 Ternary vs. Regular extensions

In this example we show that generating regular extensions of a matroid gives the same results as generating the same extensions over the ternary field with a forbidden $U24$-minor. (The idea behind this is that ternary matroids without $U24$-minor are binary, and hence also regular.)

```
bash$ macek -pREG \
      '!extendsize 6 6;!prtree;!writetreeto ex-reg ((T))' grK33
bash$ macek -pGF3 '@ext-forbid U24;\
       !extendsize 6 6;!prtree;!writetreeto ex-tern ((T))' grK33
```

Again, these commands should be called from one (logical) line of the shell. The resulting lists of each of the extension commands are written to the files '`ex-reg.mck`' and '`ex-tern.mck`'. (Actually, the same list as `ex-tern` can be obtained by generating all ternary extensions, and then filtering out those with $U24$-minors.) Finally we look at the generated lists again, and show that they are the same with the following command. We do not need to switch between the regular and ternary partial fields here since the regular representations may be read everywhere.

```
bash$ macek -pGF3 '!equiv' ex-tern ex-reg
```

7.5 Extending $F7$ in Quaternary

```
bash$ macek -pGF4 '!extend bb;@ext-forbid U25;!minor' F7 U24
```

In this example we show the quaternary 3-connected extensions of the Fano matroid $F7$ without $U25$-minors. Notice that all such generated extension have no $U24$-minor either, and so they are all binary. Since this computation would continue forever, it cannot serve as a rigorous proof, but it suggests that all quaternary extensions of $F7$ without $U25$-minor are binary. (Which is, indeed, known to be true.)

Try the same example with more extension steps, that is like:

```
bash$ macek -pGF4 '!extend bb..b;@ext-forbid U25;!minor' F7 U24
```

7.6 Examining Near-Regular Extensions

In this section we show how to generate near-regular extensions of the matroid $P7$, and how to see the inequivalent ones of the same matroids. Run the following two commands, and watch their outputs carefully.

```
bash$ macek -pNREG '!extend b;!quiet;!prmore' P7
bash$ macek -pGF3 '!extend b;!quiet;!prmore' P7
```

You see that a total of 8 extensions are generated in the near-regular case, but only 4 of them have distinct matroid hash-values. This suggests that there are, in fact, only 4 non-isomorphic extensions. (Which can be verified by other means, try it...) Distinct matroid hash-values always mean non-isomorphic matroids. On the other hand, there are 12 extensions generated in the ternary case, and you may find all those 4 near-regular hash-value classes as distinct matroids there (non-isomorphic since ternary representations are unique).

This example is to show you that nonequivalent representations of the same matroid frequently occur when working over more complex (partial) fields, and a possible way how to handle them. Another interesting point here is that two distinct matroids of the 12 ternary extensions have the same hash-value (this is in version 1.0 hash-values which may change in future!). To see that the matroids are really different, find out that one has two triangles while the other has only one. Run the same script without the suppressing command `!quiet` to see more about structure of these two (and others) matroids. Hence, even if two hash-values are the same, the matroids still may be non-isomorphic!

```
bash$ macek -pGF3 '!extend r;!prmore' P7
bash$ macek -pGF3 '!extend r;!verbose;!prmore' P7
```

7.7 Extending Whirls

In this section we want to demonstrate the fact that wheels and whirls are exceptions in Seymour's splitter theorem which is a base of our extension-generating algorithm, Section 6.7 [Generating], page 30. This is another potential problem, in addition to non-equivalent representations, that must be closely watched when using extension generating functions of Macek.

```
bash$ macek -pGF4 '!extendsize 4 4;!writetreeto xu24 ((T))' U24
bash$ macek -pGF4 '!extendsize 4 4;!writetreeto xwh3 ((T))' Wh3
bash$ macek -pGF4 '!equiv' xwh3 xu24
```

The first call generates 8-element rank-4 extensions of $U24$ – the 2-whirl. The second call generates 8-element rank-4 extensions of $Wh3$ – the 3-whirl. On the third line you may then see that not all matroids generated secondly appear in the first list, despite them all having a $U24$-minor.

7.8 Branch-Width 3

```
bash$ macek -pREG '!bwidth3 ((T));!deleach;!coneach;!bwidth3' R10
```

In this example we show that the regular matroid $R10$ is an excluded minor for matroids of branch-width 3. The first command `!bwidth3 ((T))` verifies that $R10$ itself has branch-width bigger than 3. Then the next two commands construct all one-element deletions and

contractions of $R10$, which are all stored to the default address `>(((0t)))`. Finally, we see that all these deletions and contractions have branch-width 3 in `!bwidth3` (with the default frame address `((S))`).

One *important remark* here is that you cannot perform the same computation with an arbitrary matroid, since the command `!bwidth3` requires a 3-connected matroid on the input. Since the matroid $R10$ is 4-connected, all its one-element deletions and contractions are indeed 3-connected, but this may not be true for other matroids! Then you have to use the command '`!filt-isconn3`' to filter out the matroids which are not 3-connected.

```
bash$ macek -pREG '!remeach;!filt-isconn3;!bwidth3 ((TS))' grV8
```

7.9 Binary Excluded Minors for Branch-Width 3

It is known that if M is an excluded minor for branch-width 3, then M has at most 14 elements [Hall, Oxley, Semple, Whittle]. Moreover, Dharmatilake conjectured that all binary excluded minor for branch-width 3 belong to the following set: $\{grQ3, grO6, grK5, grK5*, grV8, grV8*, R10, ND11, ND14, ND23\}$. The first seven of them are regular matroids. We need to look only at remaining binary non-regular matroids on up to 14 elements. We have proved this conjecture using a Macek script described next.

The whole script, named `bw3bin`, is prepared as a procedure. One call to it performs one step of the whole computation. See below how to call this procedure step by step... The procedure takes one argument which contains a starting list of matroids for this step. The default value for the argument is `"f7."`.

```
# Call this as:  macek -pGF2 '&bw3bin f7..'
#
@subd-param1 "f7."
```

Next the procedure prepares its working subtree. (This is not necessary in general, but we do so here to neatly organize the procedure.) Each node in the subtree is named and commented so that you see its purpose. Keep in mind that this part of the procedure is processed immediately when the file is read from the input.

```
@sub-known-excluded "bw3-bin-exc"
@sub-excluded "(((S)))"
{
@name "bw3bin-w"
@comment "bw3bin working subframe:"
{
<${known-excluded}
}{
@name extens1
@comment "new b-extensions of $input [${param1}]..."
}{
@name e-bwidth4
@comment "those generated with bwidth 4 get here:"
}{
@name e-bwidth4n
@comment "those new excl-minors with bwidth 4 get here:"
```

```
}{
@name e-bwidth3
@comment "those next with bwidth 3 get here:"
}}
@sub-input "(()(S))"
{
<$param1
@comment "this is the starting set of matroids ${param1}:"
}
```

Here the computation part of the procedure starts. We first define macros for easy access to parts of the above subtree. Notice that we later close the parameter addresses that use these macros with '|', so that we do not have to bother with the proper number of brackets that must be closed. Then we decrease output verbosity level, and print the whole starting tree.

```
@sub-generto  "(((4)("
@sub-generall  "(((1)("
@sub-gener4bw  "(((2)("
@sub-gener4n  "(((3)("
!quiet
!prtree (T)
```

This part generates all one-element extensions and coextensions to the starting list of matroids. Then the generated extensions are copied to a storage, and they are as well separated by branch-width into `generto` and `$gener4bw`. Notice the way we separate the generated list — we first mark those frames not having branch-width 3 by the command `!rex-bwidth3`, and then we move the marked frames away. (If we used `!filt-bwidth3`, then those frames not having branch-width 3 would be lost.)

```
!extend b $input  >$generto(0t)|
!move ${generto}S|  >$generall(0t)|
!rex-bwidth3 ${generto}S|
!move ^1  >$gener4bw(0t)|
```

Finally, we store the results into several files. Out of those (co)extensions not having branch-width 3 we filter out ones having minor among already known excluded minors. The remaining ones are returned as possible new excluded minors (which do not actually exist as we see).

```
!writetreeto ${param1}b-all (T)
!writetreeto ${param1}b ${generto}T|
!writetreeto ${param1}b-4 ${gener4bw}T|
!move ${gener4bw}S|  >$gener4n(0t)|
!filx-minor ${gener4n}s| $excluded
!writetreeto ${param1}b-4n ${gener4n}T|
!prtree
```

Here we show how the above procedure is called to get the final result. The whole file defining the procedure should be included in the Macek distribution under the name '`bw3bin`'. You need to first create the starting list of matroids '`f7.`' looking like:

```
{ F7 }
```

Recall that we are searching only through non-regular binary matroids, and hence we may assume, up to duality, that all matroids we need to generate contain an $F7$ minor. We then call:

```
bash$ macek -pGF2 '&bw3bin f7.'
```

The previous command performs one step of the search process. The next step starts with the file '`f7.b`' generated previously. Then the next step starts with '`f7.bb`', etc...

```
bash$ macek -pGF2 '&bw3bin f7.b'
bash$ macek -pGF2 '&bw3bin f7.bb'
bash$ macek -pGF2 '&bw3bin f7.bbb'
bash$ macek -pGF2 '&bw3bin f7.bbbb'
bash$ macek -pGF2 '&bw3bin f7.bbbbb'
bash$ macek -pGF2 '&bw3bin f7.bbbbbb'
```

We have to execute seven steps total to get to matroids on 14 elements. You may easily check in the resulting files that no other excluded minors for branch-width 3 were found.

7.10 Ternary Excluded Minors for Near-Regular

It is known that if M is an excluded minor for near-regular representability, then M has at most 8 elements [Geelen, unpublished]. Moreover, it happens that all four excluded minors for GF(3) representability are also excluded minors for near-regular representability. Thus to find all excluded minors for near-regular representability, one just has to search through ternary non-binary matroids. Similarly as above, all such matroids must contain an extension of the 3-whirl.

We show how to find all 6 such excluded minors using the Macek program. (They are $F7-$, $AG(2,3)-e$, their duals, $P8$, and $AG(2,3)-e$ by $DeltaY$.)

The whole script, named `excnreg`, is prepared as a procedure. Each call executes one step of the whole computation. (There are, actually, only two meaningful steps, since we need to get from 6-element Wh3 to 8 element matroids.) The procedure takes one argument which is then used to construct the filenames of three input lists - starting near-reg and ternary lists, and previous excluded minor list. (Possible second argument should be always '`b`'...) The default value for the argument is `"nre6."`.

```
#
# Call this script as  macek -pNREG '&excnreg nre6.' .
#
@subd-param1 "nre6"
@subd-param2 "b"
@sub-extdesc $param2
@sub-nrein $param1
@sub-nretin $nrein-tern
@sub-nrexcl $nrein-excl
```

This part prepares the working subframes with their names and comments.

```
{
@comment "For extensions in Near-Reg :"
{
@comment "the starting list"
```

```
<$nrein
}{
@comment "the extensions"
}}
{
@comment "For extensions in GF(3) :"
@ext-forbid $nrexcl
{
@comment "the starting list"
# (must be read in ternary later!)
}{
@comment "the extensions"
}}
{
@comment "For filtering minors"
{}{}
}
```

Here we generate all next-step extensions in the near-regular partial field. Then we switch to the ternary field, and generate analogous ternary extensions. The forbidden list for ternary extensions — excluded minors known so far, is given by an option above. Notice that we can read the input ternary list only after switching to the ternary field, otherwise an error would occur. (Other possibility is to consider the option '`@inputpf GF3`' above...) We have to keep the ternary and near-regular lists separately (despite them containing the same matroids), because their extension signatures are different.

```
!quiet
!prtree
!prtext "Now we generate all next-step extensions in Near-reg."
!extend $extdesc (((S))) >((((0t)|

!pfield GF3
!prtext "Now we generate extensions in GF3 with no known excl minors."
!read $nretin >((3)(t))
!move ((3)(s)) >((((0t)|
!extend $extdesc ((((S))) >(((((0t)|
```

Next we look at which of the ternary extensions have no minor among the near-regular extensions. These are the new excluded minors. Resulting lists are written to files for use in the next step.

```
!move ((()(S)| >((2)(((0t)|
!import nreg-tr ((2)((S)|
!rex-minor ((()(()(S)| ((2)((S)|
!move ^1 >((2)((()((0t)|
!read "@comment \"For merging excluded minors :\";$nrexcl" >((3)(t)|
!move ((2)(()(S)| >((3)((0t)|
!prtree

!writetreeto "$nrein$extdesc-exc" ((2)(()(T)|
!writetreeto "$nrein$extdesc-excl" ((3)(T)|
!writetreeto "$nrein$extdesc-tern" ((()(()(T)|
```

```
!pfield Nreg
!writetreeto "$nrein$extdesc" ((()(T)|
!prtext "One step of &excnreg finished. Call next with one more \"b\"."
```

Here we show how the above procedure is called to get the final result. The whole file defining the procedure should be included in the Macek distribution under the name '`excnreg`'. You need to first create the starting lists of matroids '`nre6.`' and '`nre6.-tern`' looking like

```
{ Wh3 }
```

and an empty file '`nre6.-excl`' for collecting the excluded minors. Then you call:

```
bash$ macek -pNREG '&excnreg nre6.'
bash$ macek -pNREG '&excnreg nre6.b'
...
```

8 Remarks

8.1 Program Reliability

In this chapter we discuss correctness and reliability of the results of Macek computations. When developing this project, we made every possible effort to produce a stable and very reliable software tool. Such an effort is, indeed, necessary if we want to use Macek computation results in research papers. However, if your view of computer programs is distorted by bad experiences with products of one unnamed Redmond company, then there are many ways how you can test our program, and ensure that you are getting correct answers.

Almost every algorithm in our program contains thorough internal checks. These include watching data consistency at critical steps, repeating algorithms on modified input data (an equivalent input like a matrix with swapped or pivoted lines, or a dual matrix, etc.), and using alternative or brute-force algorithms for the same question. User may find much more information about the internal checks directly in Macek source files.

To save time, time-consuming internal checks are randomized and executed only occasionally. Moreover, we provide an alternative executable '`macek.nodebug`' which does not include the internal checks, and so it is significantly faster.

Extensive debugging messages generated by the program with '`-g3`' show you exactly what the program does, and they may be easily followed in the program source files (which contain also detailed descriptions of algorithms). Since randomized algorithms are used in the program, the course of program computation may vary from one run to another, but the final results should, of course, stay the same. User may also test correctness of Macek computations by comparing the output with known theoretical results. Several such examples are presented in Chapter 7 [Practical], page 37.

Another way of checking the program output is to compare it for different versions of the program. For example, from Macek version 0.8 many structural algorithms were replaced by new ones that are cleaner and faster. In particular, this means that comparing structural answers from version 0.8.2 with the same answers from version 1.0.2 provides a nontrivial proof of correctness. Find out more about this topic by reading the corresponding source files.

Lastly, we want to note that we have designed our program to be an advanced tool for skilled users. (Here we mean mainly skilled in matroid theory.) The Macek program is not idiot-proof. So keep in mind that "*garbage in, garbage out*". Watch out carefully whether the program is really computing the tasks you expect it to. And finally, (RTFM!) read this whole manual very carefully before starting with complicated computations.

8.2 Troubleshooting

The troubleshooting section is not written yet. If you have troubles with the Macek program, let me know so that I may include some solution tips here. . .

`http://www.mcs.vuw.ac.nz/research/macek/`.

8.3 Adding Functions to Macek

This chapter of the Macek manual is provided for those who are not satisfied with the current functionality of this program; who want more functions in it, and who are able to contribute to Macek development. There are lots more things you can do if you really want!

According to the GPL license covering Macek, you may obtain and modify any source file of the program. However, we suggest you follow the guidelines provided next, so that your additions to Macek will be compatible with the future development. And, if you write a nice piece of code for Macek, let me know so that I can arrange it within the master distribution. http://www.mcs.vuw.ac.nz/research/macek/.

Before playing with new functions for Macek, read the relevant source files of the distribution since they contain a lot of technical description and comments. Then prepare you code separately in a separate directory. (You may use the provided subdirectory '`src/addons`'...)

There are three areas in which it is useful to add you code directly to the existing Macek code, and special (empty so far) files '`*.inc`' are provided there for you: additional partial field definitions and translations '`pfield/pfdef-more.inc,pftran-more.inc`', additional command handles and option descriptions '`frame/frcoms-more.inc,fropts-more.inc`', and extra control rules for extension generating '`gener/gener-more.inc`'. Read the comments in these files, and follow samples there.

Good luck with development!

8.4 Acknowledgements

The author acknowledges support from the Victoria University of Wellington, and from the Marsden Fund of New Zealand (a grant to Geoff Whittle).

The author also thanks Geoff Whittle for stimulating discussions about the nature and goals of this program, mainly in the early stages of development, and Steven Archer for testing the program and poolishing this manual.

Index

Here we list the concept index for this manual. (Go to the next index if you look for program commands / options.)

P

Q

R

S

T

V

W

Here we list the index of all frame- commands and options described in this manual.

T

U

V

W