# PIPS
## DEVELOPMENT ENVIRONMENT


François IRIGOIN
Pierre JOUVELOT
Rémi TRIOLET
Arnauld LESERVOT
Alexis PLATONOFF
Ronan KERYELL
Fabien COELHO
Béatrice CREUSILLET
Corinne ANCOURT


96/04/23

# Contents

# Introduction

This document aims at presenting PIPS' development environment. It is not linearly organized: PIPS is made of several, sometimes interdependent, components. This paper thus begins with a presentation of PIPS directories. Then the shell environment is described. The next two chapters are devoted to two external tools on which PIPS relies: NewGen and the linear library. Section 5 will then present PIPS' make file policy. Sections 6and 7 are devoted to PIPS libraries and passes. The next section briefly describes some conventions usually respected when developing in PIPS. The last two sections describe the *bug policy* of PIPS and some save and restore information.

This manual is not exhaustive. You can add your own sections and update existing ones if you find missing or erroneous information.

The reader is supposed to be a PIPS user [3], and to have read the reports about NewGen [1, 2]. A good understanding of `pipsmake` mechanisms would also be helpful.

## 1 Shell environment (sh, csh, ksh, tcsh)

Many environment variables are used by PIPS executables and utilities. They mainly describe PIPS directory hierarchy, and compilation options. All these variables are initialized by the script `/projects/Pips/pipsrc.sh` for `sh` and `ksh`, and by `/projects/Pips/pipsrc.csh` for `csh` and `tcsh`. Both scripts are automatically derived from a reference file `pipsrc.ref`.

## 2 PIPS directories

This section describes the directory hierarchy of PIPS. The root directory is `/projects/Pips`. It is owned by the user `pips`.

The sub-directories are described below. Almost at each level, `Makefile`s ensure the coherency of the system, and facilitate the installation of developments into the production hierarchy.

- **Development.**`$PIPS_DEVEDIR`.

  This directory contains the version of PIPS currently being developed.

  - **Passes.**

    This directory contains the sources of the different passes, in several sub-directories named from the passes (`pips`, `tpips`, `wpips`).

  - **Lib.**

    This directory contains the source of the several libraries of PIPS. It is divided into sub-directories named from the libraries. It also contains a `Makefile` to recompile all the development libraries. This `Makefile` is not automatically generated, and it thus not always up-to-date.

  - **Runtime.**

    This directory contains the sources of the libraries used when executing codes generated by PIPS; in fact, there are only two libraries: The first for the HPF compiler `hpfc`, and the second one for the WP65 project.

  - **Scripts.**

    This directory contains the source of the shell scripts which are useful for PIPS, the linear library and NewGen. It is further divided into several directories. In each sub-directory a local `Makefile` performs automatic tasks such as the installation of the sources in the `$PIPS_UTILDIR` directory.

- **Documentation.** `$PIPS_DOCDIR`.

  This directory contains the documentation: Reports for DRET contracts, reports from discussions or analyses, manuals, etc.... But also the description of some internal representations (for instance in the `Newgen` sub-directory), or configuration files, such as `pipsmake-rc.tex`. Some configuration files are automatically derived from others, as well as some documentation (see the `Makefile`).

  – **Newgen.**

    This directory contains the description (in LaTeX) of the internal data structures used in the project, as they are sued in the development and production hierarchies, but not in the release environment. This is due to the lack of modularity induced by NewGen: it is not possible to link without major risks with independent passes of NewGen.

    The local makefile transforms the `*.f.tex` files describing PIPS data structures into NewGen and header files, and exports them to the directory `PIPS_INCLUDEDIR`.

  – **wpips-epips-user-manual.**

    This directory contains the user manual.

- **Production.** `$PIPS_PRODDIR`.

  This directory contains the versions of library and passes temporarily declared as being ready.

  – **Src/Passes.**

    This directory contains the sources of the different passes, in several sub-directories named from the passes (`pips`, `tpips`, `wpips`).

  – **Src/Lib.**

    This directory contains the source of the several libraries of PIPS. It is divided into sub-directories named from the libraries.

  – **Src/Runtime.**

    This directory contains the sources of the libraries used when executing codes generated by PIPS; in fact, there are only two libraries: The first for the HPF compiler `hpfc`, and the second one for the WP65 project.

  – **Bin.** `$PIPS_BINDIR`.

    This directory contains PIPS executables: `pips`, `tpips` and `wpips`.

  – **Lib.** `$PIPS_LIBDIR`.

    This directory contains the libraries (`lib*.a`), and some configuration files, automatically generated from their LaTeX versions in `$PIPS_DOCDIR`: `wpips.rc`, `pipsmake.rc`, and `properties.rc`, .

  – **Include.** `$PIPS_INCLUDEDIR`.

    This directory contains PIPS header files. They are automatically generated from NewGen specifications, from the linear libraries, from PIPS libraries, and from some files in `$PIPS_DOCDIR`.

- **Release.** `$PIPS_RELEDIR`.

  This directory contains the binary executables of a coherent version of PIPS. This version is used for demonstrations and installations.

  *Not used since 1990!*

  - **Config.**

    This directory contains the file to set the environment variables. Some of these variables are used by PIPS or by X or NeWS servers.

  - **Spec.**

    This directory contains the specification files for PIPS internal representation. These files are dynamically read by PIPS passes.

  - **Bin.**

    This directory contains the executables.

  - **Lib.**

    This directory contains`properties.rc`, `pips_help.txt` and `pips.icon`. Maybe we should add `makefile.rc`.

  - **OpenWindow.**

    This directory contains OpenWindows fonts used by PIPS graphical interface.

- **Externals.** `$PIPS_EXTEDIR`.

  This directory contains the executable versions and the header files of libraries developed at CRI and used by PIPS, that is to say the `Newgen` and `Linear` libraries (which are installed here using the `fetch-linear` and `fetch-newgen` libraries), as well as some run-time support files. .

- **Utilities.** `$PIPS_UTILDIR`.

  This directory contains all the tools and scripts which facilitate the management of PIPS and the compilation of `Linear`, `Newgen` and PIPS. Most of them are installed from the `Development/Scripts` sub-directories.

  - `clean-pips`: Removal of unuseful or obsolete files.
  - `make-pips`: Runs `make` in all the libraries and passes of PIPS. The `-d` option recompiles `Development`, while the `-p` option recompiles in `Production` and installs the libraries in `$PIPS_LIBDIR` (this is the default option). The `-n` option, similar to the `-n` option of `make`, shows what is not up-to-date. The `-r` option (for recompile), in combination with `-p`, successively executes `make clean`, `make production-depend` and `make`; this achieves a global recompilation.

    It is necessary to recompute the dependences which are automaticcally generated in each Makefile when PIPS is installed on a new system, because the directory hierarchy might be the same (dependences are almost always expressed unsing absolute paths). Destroying derived files is also necessary when changing the compilation options (for instance when trying `-O3` to achieve better performances, of `-p` to do some profiling).
  - `pips-makemake`: Automatic derivation of library and pass `Makefiles` from local `config.makefiles`.
  - `search-pips`: Search, and possibly removal, of auxilliary files such as `databases`; for instance, destruction of all databases not used since one month:

```
search-pips -w -m -D
```

- `fetch-linear`: put in `Externals` the linear library and the related headers.See also Section 4.
- `fetch-newgen`: put in `Externals` the NewGen library and its related headers. See also Section 3.
- `analyze_libraries`: cf. section Dependance entre Bibliotheques
- `order_libraries`: cf. section Dependance entre Bibliotheques
- `tape-pips`: partial extraction of PIPS files for an external installation or demonstration; the failes are taken in `$PIPS_PRODDIR`; do not forget to install your library before! See also `bar`, `eject`, `fdformat` and `tar`. To use the floppy driver: `/dev/rfd0`.
- `Validate`: Non-regression test; see ?!?;

  The arguments are names of subdirectories in `Pips/Tests/Validation`.

  Tests are organized on a library basis in `Pips/Tests/Validation` while bugs are reported in the same way in `Pips/Tests/Bugs`. Bugs and non-regression tests are characterized by one or more Fortran programs which either core dump (for bugs) or must have a known result (tests).

  Set `PIPSDBM_DEBUG_LEVEL` to 5 to check data structures when they are stored and to 9 to check them when they are delivered; some data structures are not checked, among them, all that contain external types;

  use `accept` to update reference files for validation; can be used with `/usr/ucb/mail: | xx accept` where `xx` is the message number;

  To run it in background, redirect `stderr` on /dev/null. Example:

  ```
  Validate DemoStd 2>/dev/null &
  ```
- `bug-to-validate`: When a bug is corrected, run `bug-to-validate` to move the bug case from `Bugs/xxx` to `Validation/xxx`.

  Do not forget to update `Documentation/pips-bugs.f.tex`.
- `Init, Select, Perform, Display, Pips`: batch interface for PIPS; it is described in ???
- `stf-workspace`: calls Toolpack restructurer `stf` on each Fortran file in a workspace; also directly available as a PIPS transformation.
- `pips-experiment`: runs the same analyses and transformations on each module in a workspace.
- `print-dg-statistics`: exploits some of the statistics generated by the dependence graph computation when a specific property is set to `TRUE` (`RICE_DG_PROVIDE_STATISTICS`).
- there many other things in this directory; volunteers to write their documentation here are welcomed!

- `Experiments`

  This directory is a large piece of disk space to be used temporarily to analyze large programs and to core dump large `pips` processes; no permanent, non-deductible information should be stored there;

A directory **Backup** would also be useful to save a version which has been installed, but is less ready than its developer have thought.

# 3 NewGen

NewGen is a software engineering tool, from which almost all PIPS data structures are generated. It is described in ???. It is considered as an external tool, and is independent from the PIPS project.

However, since this tool still needs some tuning, its internal libraries, `genC.a, hash.a, set.a,...`, are just copied (instead of being linked using `ln` or `ln -s`) in the `$PIPS_EXTEDIR` directory, using the `fetch-newgen` utility each time they are modified. The corresponding header files are also copied in `$PIPS_EXTEDIR` by this command.

NewGen sources are in the directory `$NEWGEN_DIR`. It may sometimes be useful to consult them. Several functionalities are useful when debugging:

- `gen_consistent_p()` checks whether an existing NewGen data structure is (recursively) well formed.
- `gen_defined_p()` checks whether an existing NewGen data structure is (recursively) completely defined.
- `gen_debug` is an external variable to dynamically check the coherence of NewGen data structures.

  Activation:

  ```
  gen_debug |= GEN_DBG_CHECK;
  ```

  Desactivation:

  ```
  gen_debug &= ~GEN_DBG_CHECK;
  ```

  And when all else fails:

  ```
  gen_debug = GEN_TRAV_OBJ
  ```

- To print the type number of a NewGen object under `dbx` or `gdb`:

  ```
  print obj->i
  ```

- To print the name of an entity `e` (an entity is a PIPS data structure, see `ri.f.tex`):

  ```
  print (e+2)->s
  ```

- To print the label of a statement `stmt`:

  ```
  print ((stmt+1)->p+2)->s
  ```

- To print the domain of a NewGen Object from its number (`obj->i`):

  ```
  print Domains[obj->i].name
  ```

In the future, `gdb` macros compatible with C macros will be provided.

# 4 Linear library

The linear library is also independent from PIPS. Its development was funded by the French organization CNRS.

Its root directory is `/projects/C3/Linear/` (`$LINEAR_DIR`). This directory is further divided into two sub-directories:

- `Development.`

  This directory contains the sources of each sub-library, in separated directories named after these sub-libraries. Sources can be modified in this hierarchy. They can be locally compiled using the command `make compile`. To install them in the production hierarchy (see the next item), you must use `make install`.

- `Production.`

  This directory contains the installed sources of the linear library. The subdivision of directories is identical to `Development`, but for an additional directory (`include.dir`), which gathers all the header files from the sub-libraries.

The statute of this library is the same as `libC`, which means that it is considered as an external library. However, since it still requires some tuning, its constituents `arithmetique.a, vecteur.a, ...` are copied (instead of being linked) in the `$PIPS_EXTEDIR` directory. This operation is performed by the `fetch-linear` utility after each installation. This command also copies the corresponding header files in `$PIPS_EXTEDIR`.

# 5 Makefiles

The `make` utility is extensively used to ensure the coherency of PIPS components. However, most `Makefiles` are not written by hand, but are automatically derived from configurations files called `config.makefile`. This automatic derivation, performed by the `pips-makemake` utility, ensures that dependences and installation procedures are respected.

`make depend` must be used regularly to keep file dependencies up-to-date. The local header file corresponding to the current pass or library must have been created before, otherwise `make depend` selects the header file from `$PIPS_INCLUDEDIR`, and the local header will never be created.

`make test`, `make ttest` and `make wtest` generate local versions of PIPS executables (respectively `pips`, `tpips` and `wpips`), to verify that the local library or pass links properly with PIPS, linear and NewGen, and to test the modifications.

`make libxxx.a` just recompiles the local `*.c` files, and creates the local library file `libxxx.a`. This is useful when making a lot of changes to check the syntax; or when making changes in several directories at the same time: in this case, the proper policy is to a) chose one of these directories as the master directory, b) build the `libxxx.a` in the other directories, c) make symbolic links towards them in from the master directory, d) and finally link in this last directory.

A typical example of `config.makefile` file is provided in Figure 1.

# 6 Library internal organization

The source files for the library `b` are in the directory `Development/Lib/b`.

A `config.makefile` must be created in this directory. It can be copied from an existing library, but it must be updated with the names of the local source and object files. Additionnal rules compatible with the `make` syntax can also be added.

Running the command `pips-makemake -l` then automatically derives from the file `config.makefile` the local `Makefile`, which is fully compatible with PIPS standards. Among others this `Makefile` contains an `install` entry to install the library in the production hierarchy.

Great care must be taken when creating the library header file `b.h`. This file includes the local header file `b-local.h` (which contains macros, type definitions,...)

```
...
AR=             $(PIPS_AR)
ARFLAGS=        $(PIPS_ARFLAGS)
CC=             $(PIPS_CC)
CFLAGS=         $(PIPS_CFLAGS)
CPPFLAGS=       $(PIPS_CPPFLAGS)
LD=             $(PIPS_LD)
LDFLAGS=        $(PIPS_LDFLAGS)
LEX=            $(PIPS_LEX)
LFLAGS=         $(PIPS_LFLAGS)
LINT=           $(PIPS_LINT)
LINTFLAGS=      $(PIPS_LINTFLAGS)
YACC=           $(PIPS_YACC)
YFLAGS=         $(PIPS_YFLAGS)
#
# The following macros define your pass.
#
# Name of the target
TARGET=         parallelize
#
# Source, header and object files used to build the target
TARGET_CFILES=  kennedy.c scan.c util.c dependence.c algebre.c \
                divar.c debug.c codegen.c scc.c
TARGET_HEADERS= includes.h
TARGET_OBJECTS= kennedy.o scan.o util.o dependence.o algebre.o \
                divar.o debug.o codegen.o scc.o
#
# List of libraries used to build the target
TARGET_LIBS=    -lprivatize -lusedef -lprettyprint -lsemantics \
                -ltransformer \
                -lcontrol -leffects -lnormalize \
                -lsc -lcontrainte -lvecteur -larithmetique \
                -lri-util -lmisc -lproperties \
                -lgenC /usr/lib/debug/malloc.o \
                -lproperties
```

Figure 1: Example of `config.makefile`.

and the external functions declared in the local source files, which are automatically extracted using `cproto`.

`b.h` must never be directly modified[1]. It is automatically generated when invoking `make header`, or when `b-local.h` has been modified, but not when the source files are modified. This avoids inopportune recompilations when tuning the library, but can lead to coherency problems.

`b.h` is automatically generated when invoking `make install`, before the actual installation in the production hierarchy.

If a main program to test or use the library exists, then it necessarily is a pass (see Section 7), or a main(). This rule is generally not respected, and this induces cycles in case of a global recompilation. (Is this up-to-date?)

## 6.1 Libraries and data structures

PIPS data structures are managed by NewGen. For each data structure declaration file in `Documentation/Newgen`, NewGen derives a header file, which is placed in `$PIPS_INCLUDEDIR`. For instance, the internal representation (i.e. the abstract syntax tree) is called `ri`. It is described in the LaTeX file `ri.f.tex` in **Documentation/Newgen**. It is then automatically transformed into a NewGen file `ri.newgen`. And finally, NewGen produces an internal description file `ri.spec` and a header file `ri.h`, which must be included in each C file using the internal representation.

Thus, it is not possible to build a new library with the name `ri`, because the corresponding header file would be called `ri.h`. The library in which higher order function concerning the `ri` are placed, is then called `ri-util`. It should be the case for all the other sets of data structures specified using NewGen. Such existing libraries are `text-util` and `paf-util`. However, mainly for historical reasons,there are numerous exceptions. Many functions are in fact in the library where they were first necessary. An example is the `syntax` library (the parser).

## 6.2 Tags

Many macros and functions are available from the linear and NewGen libraries, but also in PIPS. Their source code can be retrieved using the *tag* mechanism under `emacs` or `vi`. Tags are regularly recomputed (every hour) by the PIPS utility `make-tags`. This utility uses the environment variable `PIPS_ETAGS` which the local `etags` command, and stores the result in `Pips/Tags/TAGS`. They are made from the development source files of linear, NewGen and PIPS. But anybody can build his/her own tag

## 6.3 Library dependencies

Library dependence cycles must be avoided. Links are not easy to manage when a module $a$ from library $A$ calls a module $b$ form library $B$ which itself calls a module $a'$ from library $A$. This situation also reflects a bad logical organization of libraries. Therefore, PIPS libraries must be organized as a DAG, i.e. they must have a partial order.

Several utilities, such as `analyze_libraries`, `order_libraries`, can be used to determine the dependences between libraries, a total order compatible with the library partial order, and the possible cycles. The results are stored in `/tmp/libs.?` where the question mark stands for several values:

- u: uses, modules used by each library.

[1]A warning should be added at the beginning.

- d: defs, modules defined by each library
- j: join between uses and defs
- o: order between libraries
- etc...

## 6.4 Installation of a new phase (or library)

Some libraries implement functionalities which are accessible by the users via the `pipsmake` mechanism. For a good comprehension of the material contained in this subsection ,the reader is referred to the corresponding documentation.

A PIPS phase *myphase* is implemented as a C function named `myphase` with a single argument, which is the name of the Fortran module to analyze. It returns a boolean value, which indicates whether the computation has performed well. It therefore resembles the following dummy function:

```
bool myphase(char *module_name)
{
  bool good_result_p = TRUE;

  /* some computation */
  ......
  return(good_result_p);
}
```

This phase is usually located in a library named `myphase`, but this is not compulsory. It must set the global variables which are necessary for its execution using `properties`, and acquire the necessary resources by invoking `pipsdbm`. Directly invoking `pipsmake` is forbidden at this level.

Here are now the steps to install a new library named `mylib`:

- Create a new directory `mylib` in `$PIPS_DEVEDIR/Lib` with the fiels `mylib.c` (where the code for the C `mylib()`, which corresponds to the phase to add, generally is) and `main.c` (with the empty function `main()`).
- In the directoty `$PIPS_DEVEDIR/Lib/mylib`:
  - Create a file `config.makefile` (examples can be found in the other libraries).
  - Run the command `pips-makemake` (option -l) to automatically build the local `Makefile` file.
  - Then run the commands `make libmylib.a` and `make install`, to install the library in the `Production` hierarchy. The file `mylib.h` is installed in `$PIPS_INCLUDEDIR`, and `mylib.a` in `$PIPS_LIBDIR`. The library functions are now accessible by the other libraries and passes.

    Notice that the local `Makefile` uses the command `cproto` (in fact the environment variable `PIPS_PROTOIZE`) which is in `$PIPS_UTILDIR`, to automatically build the header file `mylib.h`. To force this file to be build again, you must successively type `touch mylib-local.h` and `make header`.
- In the directory `$PIPS_DEVEDIR/Scripts/env`:

  - Update the file `pipsrc.ref`. It contains the shell environement variables for PIPS. To update it, just add the character string `-lmylib` in the variable `$PIPS_LIBS`. Notice that the order in which the libraries are added is importaant (l'ordre des bibliotheques est important; the new library must be added after the libraries which use its modules, but after the libraries whose modules it uses.
  - Run the `make` command to automaticall build the files `pipsrc.sh` and `pipsrc.csh` respectively for `sh` or `ksh` and `csh` or `tcsh` shells. The command `make install` then installs these scripts in `$PIPS_UTILDIR`.

- In the directory `$PIPS_DOCDIR`:

  - declare the new phase to `pipsmake` by adding the necessary rules in the file `pipsmake-rc.f.tex`. Each rule describes the dependances between the input and output resources of each phase. Just have a look at hte other rules to build your owns. You can also add aliases to obtain nice menus in the `wpips` interface.

    For instance, for a phase *myphase* which reads a resource *res1* andproduces a resource *res2* for the current module, the rule and its alias are:

    ```
    alias myphase 'My Phase'

    myphase > MODULE.res2
            < PROGRAM.entities
            < MODULE.res1
    ```

    Notice that libraries generally use the program entities, that is to say all the objects from the current Fortran program and which have a name. This resource is named `PROGRAM.entities`.
  - New *properties* may be necessary for the new phase. They must be delared in the file `properties-rc.tex`.
  - Run the command `make all` to now derive from `pipsmake-rc.tex` the files `phases.h`, `resources.h` and `builder-map.h` and to copy them into `$PIPS_INCLUDEDIR`; to also derive `pipsmake.rc`, `properties.rc` and `wpips.rc`, and to copy them into `$PIPS_LIBDIR`.

- In the directory `$PIPS_BINSRCDIR`:

  Execute the command `make-pips -r`, to recompile and install updated versions of `pips`, `tpips` and `wpips`.

It can also be necessary to update the shell scripts `Init`, `Build`, `Select`, `Perform`, and `Display` in `$PIPS_DEVEDIR/Scripts/drivers`.

This section did not deal with the case where a new resource is introduced. This is the object of the next section.

## 6.5 Dealing with a new resource

A new resource is declared by its use or its production by one of the rules in the file `pipsmake-rc.tex` (see below). Here are now the steps to follow to deal with a new resource `myres` once the phase `myphase` from the library `mylib` has been declared and installed as shown in the previous section, that is to say that the function

```
bool myphase(char *module_name)
```

is available.

- In the directory $PIPS_DOCDIR:

  - Declare the new resource `myres` by modifying the file `pipsmake-rc.tex`. Each rule describes the dependances between the input and output resources of each phase. Just have a look at hte other rules to build your owns. You can also add aliases to obtain nice menus in the `wpips` interface.

    For instance, for a phase *myphase* which reads a resource `another_res` and produces a resource *my res* for the current module, the rule and its alias are:

    ```
    alias myphase 'My Phase'

    myphase > MODULE.myres
            < PROGRAM.entities
            < MODULE.another_res
    ```

  - Run the command `make all` to now derive from `pipsmake-rc.tex` the files `phases.h`, `resources.h` and `builder-map.h` and to copy them into $PIPS_INCLUDEDIR; to also derive `pipsmake.rc`, `properties.rc` and `wpips.rc` and to copy them into $PIPS_LIBDIR.

- In the directory `$PIPS_DEVEDIR/Lib/pipsdbm`:

  The new resource must be declared to the resource manager `pipsdbm`:

  - Update the file `module.c` by adding the line

    ```
    DBR_MYRES,
    ```

    in the array `load_order[]` which defins the order in which the resources must be unloaded when a module is closed (see the user manual).

  - Update the file `methods_io.c` by adding the following line to the array `method_map[]`:

    ```
    {DBR_MYRES,(chunk *(*)()) undefined\_method,
               undefined\_method,
               undefined\_method,
               (bool (*)()) undefined\_check},
    ```

    The arguments are the reading, writing, freeing and consitency testing functions for the resource. Here they are undefined. But if the resource is a newgen type, the declaration is:

    ```
    {DBR_MYRES, gen_read,
                gen_write,
                gen_free,
                gen_consistent_p},
    ```

  - If the new resource is a file resource (and not a data structure), update the files `disk.c` and `module.c`.

  - Run the commands `make libpipsdbm.a` and `make install` to update `pipsdbm` and install it.

- In the directory $PIPS_BINSRCDIR:

  Execute the command `make-pips -r`, to recompile and install updated versions of `pips`, `tpips` and `wpips`.

**Remark:** It is necessary to recompile PIPS before any test, because changing the header files generated from `pipsmake-rc.tex` can lead to inconsistencies which only appear at run time (the database cannot be created for instance).

### 6.6 Modification or addition of a new NewGen data structure

NewGen data strutures for PIPS are defined in LaTeX files which are in the directory `$PIPS_DOCDIR/Newgen`. These files include both the NewGen specifications, and the comments in LaTeX format. DDL[2] files are automatically generated from the previous LaTeX files. When declaring a new data structure, two cases may arise:

1. The new data strucutre is defined in a *new* file `mysd.f.tex`:

   This file has to be declared in the local `Makefile` (in `$PIPS_DOCDIR/Newgen`) for its further installation in the `Production` hierarchy. It merely consists in adding its name `mysd.f.tex` to the `SOURCE_FILES` list.

   Then, the macro instruction `#include "mysd.h"` must be added to the files

   ```
   $PIPS_DEVEDIR/Lib/top-level/newgen.c
   $PIPS_DEVEDIR/Lib/pipsdbm/methods_io.c
   ```

   It is not compulsory to immediately execute the command `make install` in these directories. But it will be necessary before any global recompilation of PIPS (see below).

2. The new data structure is added to an already existing file `mysd.f.tex`: modifying this file is sufficient, and the above steps are not necessary.

Then, in both cases, the next steps have to be performed:

- In the directory `$PIPS_DOCDIR/Newgen`, execute the command `make` (which is equivalent to `make c_internal_representation`). This command builds the DDL file `mysd.newgen`, and copies it in the directory `$PIPS_INCLUDEDIR`; then in this last directory, it executes the command `newgen -c *.newgen` which builds the C header file corresponding to the DLL file. It also builds the `specs.h` file.
- NewGen globally handles all the data structures defined in the project. Hence any minor modification of one of them, or any addition, potentially leads to a modification of all the interfaces which are generated. It is thus necessary to recompile everything in the `Production` directory. Do not forget tofirst install `$PIPS_DEVEDIR/Lib/top-level` and `$PIPS_DEVEDIR/Lib/pipsdbm` if they have been modified in a previous step. The command `make-pips` achieves the global recompilation of PIPS. It is in the directory `$PIPS_UTILDIR` (Cf Page 4).

### 6.7 Changing the dynamic allocation library

Errors which are the most difficult to find generally are dynamic allocation errors: The allocated space is accesssed after being freed (*dangling pointer*), or a zone larger than the allocated zone is referenced (too long copy of a character string for instance).

[2] Newgen types are called *domains* and are defined using a high level specification language called DDL for *Domain Definition Language*.

A first checking level is available with a set of SUN primitives (`man malloc.h`), by linking the program with `/usr/lib/debug/malloc.o`. The choice of the dynamic allocation library can be made in `pipsrc.ref`, or in the curretn environment, but without any guarantee for the link.

A second level, a lot more efficient, but also much slower, is offerd by a public domain library, `malloclib-pl11`. This library systematically overwrites freed zones with a special pattern, and checks that string operations are valid. It also records in which files allocations and desallocations are performed.

To use it efficiently, all the source files which dynamically allocate or free memory must include `malloc.h` with double quotes (and not < > signs). This is achieved in most cases because NewGen includes `malloc.h`. But some files, such as character strings manipulation files, do not include `GenC.h`: `#include "malloc.h"` must be added to them.

PIPS must then be entirely recompiled (see `make-pips` on Page 4), as well as the libraries in `$PIPS_EXTEDIR`, after renaming (`mv`) `dbmalloc.h` into `malloc.h` in `$PIPS_EXTEDIR` and in the directories containing the sources of the external libraries (NewGen and Linear). Before linking, the environment variable `PIPS_LIBS` must have been modified, either in the current environment, or in `pipsrc.ref`, to include the correct library.

It is recommended to keep an version of PIPS linkeed with a reasonnably fast library to be able to create large databases for the tests. To avoid compiling and linking too many things each time, it is also recommended to modify the shell variables `$PIPS_BINDIR` and `$PIPS_LIBDIR` to keep both versions of PIPS.

### 6.8 Global variables, modifications

Global variables should be avoided, or at least carefully documented to avoid disturbing already existing programs.

If new functionalities are required, it is better to keep the existing module as such, and to create a new one, with another name.

For global variables, initialization, access and reset routines must be provided. Do not forget that there may be several successive requests with `wpips` or `tpips`, whereas tests performs with shell interfaces (for instance `Build`) are much simpler.

## 7 Organization of a PIPS pass

The source files for a pass `p` can be found in the directory `Development/Passes/p`. A pass, as opposed to a library, corresponds to an executable, which can be used by the users.

In this directory, a local `config.makefile` must be created. It must be updated with the names of the source files. Then the local `Makefile` can be automatically derived using `pips-makemake -p`. This `Makefile` contains among others an `install` entry to install the pass in the `Production` hierarchy.

The libraries used to build the pass are in the directories `Production/Libs` and `Externals`. The corresponding header files are in `Production/Include` and `Externals`.

To maximize code reuse, it is recommended to limit code development at this level to functionalities really specific to passes. Everything else should be placed in libraries (for instance in `top-level`).

At the library level, it is also possible to create local executables by invoking `make test`, `make ttest` or `make wtest`.

If a pass and a library are simultaneously developed, `misc` and `parallelize` for instance, one of the library must be chosen for the link, `parallelize` for instance).

It is then necessary to look for `misc.h` and `libmisc.a` in `Development/Lib/misc` instead of `Production/Libs/misc`. This can be done very simply in the config.makefile file by giving the names of the libraries to use:

```
...
# List of libraries used to build the target
TARGET_LIBS=    -lprivatize -lusedef -lprettyprint -lsemantics \
                -ltransformer \
                -lcontrol -leffects -lnormalize \
                -lsc -lcontrainte -lvecteur -larithmetique \
                -lri-util ../../Libs/libmisc.a \
                -lproperties -lgenC /usr/lib/debug/malloc.o \
                -lproperties

$(TARGET): ../Lib/misc/misc.a
```

# 8 Conventions

libraries are named `libXXX.a` where `XXX` is the logical name of the library: `vecteur`, `prettyprint`, `misc`,etc.

It is theoretically unuseful to put libraries in the Makefile dependencies, because header files which are automatically generated are in these dependencies, and are systematically modified each time an installation is performed. However, this does not work if the pass `p` calls the library `a` which needs the library `b`, and if `p` does not directly needs the library `b`: modifications to `b` will not provoke the link of `p`.

Each important library uses an environment variable `XXX_DEBUG_LEVEL` to control debugging messages. This rule hase an exception: `PARSER_DEBUG_LEVEL` corresponds to the `syntax` library .

Level 0 corresponds to the normal behaviour. The highest level is 8. Great care has to be brought to calls to `debug_on()` and `debug_off()`, because successive debug levels are stored in a stack, and it could disturb its coherency. The debug of a function of another library should not unconsciously be activated.

# 9 Bug policy

## 9.1 Bug detection

When a bug has been detected, a small Fortran program (`bug.f` for instance) must be written to reproduce the bug.

If the library `xxx` is responsible for this bug, move bug.f in `Tests/Bugs/xxx/`. This responsibility can be found using `XXX_DEBUG_LEVEL` or a debugging tool.

Then record the new bug in `$PIPS_DOCDIR/pips-bugs.f.tex`.

## 9.2 Bug correction

Go to the directory `/Tests/Bugs/xxx`. In order to use the executable of the library `xxx`, create a symbolic link towards it (`ln -s Development/Lib/xxx/pips` for instance). Here are now the different steps to perform:

- Find the sources responsible for the bugs and correct them (we assume that only one library is responsible for the bug; the case of multiple libraries is presented below).
- Run `make test` in `Development/Lib/xxx` to build an executable.

- Back in `Tests/Bugs/xxx`, run:

  ```
  Init -f bug.f bug
  Display -m -v bug
  ```

  and verify that the bug has been eliminated.

- In `Development/Lib/xxx/`:
  - Launch `Validate Xxx` to run the non regression tests specific to the library `xxx`.
  - Run `Validate` if the changes may affect the results in several phases.
  - If the results are satisfactory, run `make install` to install the new sources in the `Production` hierarchy. Beware that this does not build new executables in `$PIPS_BINDIR`. This can be done with the `make-pips` command, but this is not compulsory, because this command is automatically run each night.
  -

Once these operations have been done, the program `bug.f` and the results obtained for its parallelization must be added to the non-regresson tests in the directory `/Tests/Validation/Xxx`. For that purpose, you can run the command `bug-to-validate bug.f`. But beware that this command is a script shell which provides a mere parallelization of `bug.f`. To transfer all the programs from the directory `Tests/Bugs/xxx` to `/Tests/Validation/Xxx`, you can use the command `dir-to-validate`. Again, this is very convenient when the desired test is the default test (see Section 9.3).

**Remark:** When several libraries are responsible for the bug, say `xxx1`,..., `xxxk`, chose one of the libraries to build the executable by linking with the other libraries. Once the problems are fixed, do not forget to run `make install` in all the libraries.

## 9.3 The `Validation` directories

These directories `/Tests/Validation/Xxx` contain the non-regression tests for each significant library `xxx`, as well as demonstration programs, benchmarks, ....

For each Fortran file `bug.f`, there exists a test file `bug.test`, and a sub-directory `bug.result` in which the results are stored in a `test` file. All these files can be generated by `bug-to-validate bug.f` when dealing with a bug in `Tests/Bugs/xxx`. If this command has been used, the reuslt directory contains the results of a mere parallelization. For more complex results, a specific test file must be written, as well as a script inspired from `bug-to-validate` to install the necessary files in the validation directory.

If the same test file is valid for a whole directory (ex : `Validation/Flint`), a generic file `default_test` can be created in this directory. In this file, the generic names for the program are `tested_file` and `TESTED_FILE`. Here is the `default_test` script of `Validation/Flint`:

```
#!/bin/sh
Init -d -f $PIPSDIR/Tests/Validation/Flint/tested_file.f \
           tested_file 2>/dev/null >/dev/null
Perform -m tested_file flinter 2>/dev/null
cat tested_file.database/TESTED_FILE.flinted
Delete tested_file 2>/dev/null
```

Notice that if there exists a local test for the file (`bug.test`), it will be used first. The priority starts from local files to general ones.

## 9.4 Other validation

The command Validate can be used to measure the impact of a modification by comparing the new behavior of PIPS with the preceding one.

To use it to check the parallelization process, put your test file, say mytest.f which contains the main program MAIN and a subroutine SUB, into one of the directories in ~pips/Pips/Tests/Validation directory. You can also create your own directory there if you want to ensure that a particular aspect of PIPS behaves the correct way.

Once mytest.f is in such a directory, say kludge, you should do the following thing.

```
Validation/kludge: Init -f mytest.f mytest
Validation/kludge: mkdir mytest.result
Validation/kludge: Display -m main > mytest.result/MAIN
Validation/kludge: Display -m sub > mytest.result/SUB
Validation/kludge: Delete mytest
```

Check that the output in the MODULE1, MODULE2, ... files is what you want ... and that's it!

After a while, if you want to check that PIPS still does that it was supposed to do, go into Validation and type

```
Validate kludge
```

If there is any problem, you will get a mail message that tells you want the problem is. You can also type Validate to check everything.

- Validate with no argument validates all the directories which are listed in the file Validation/defaults.
- When a directory of Validation is validated, if for the program foo.f there exists a file foo.test it is executed; otherwise default_test is ran. The output is compared to foo.result/test which must have been created before. The result can be as complex as desired, and not a mere parallelized version.
- tpips scripts can also be used. In this case, do not forget to build a local tpips by invoking make ttest after creating your local pips.

# 10 Miscellaneous

To print a nice version of PIPS output, you can use tgrind:

```
tgrind -lf extr_doall.f
```

It is highly recommended to use the Emacs editor to benefit from the tags which are updated every hour.

# References

[1] Pierre Jouvelot and Rémi Triolet. Newgen : A langage-independent program generator. rapport interne EMP-CRI 191, CRI, École des Mines de Paris, July 1989.

[2] Pierre Jouvelot and Rémi Triolet. *NewGen User Manual.* CRI, École des Mines de Paris, December 1990.

[3] Ronan Keryell. *WPips and EPips User Manual.* CRI, École des Mines de Paris, April 1996. Available from `http://www.cri.ensmp.fr/pips` and as report A/288.

# Index