

The BMI library (version 6.0)

François Boulier

Table of Contents

1	Introduction	1
1.1	Introduction to the 2023 version	1
2	Configuration and Building	2
	The BMI library copying conditions	2
2.1	Building the development version	2
2.2	Building the BMI library	2
2.3	Building the BMI library for Python/sympy (Linux)	2
2.3.1	A short test	3
2.4	Building the BMI library for Maple	3
2.4.1	Maple related issues	4
2.4.2	BLAD related issues	4
2.4.3	GMP related issues	4
2.4.4	A short test	4
2.4.5	Miscellaneous	5
2.5	Building the BMI library for Sagemath	5
2.5.1	A short test	5
3	Interaction with scientific computing software	7
3.1	The Maple version	7
3.1.1	A trace of execution	7
3.2	The Python/sympy version	8
3.2.1	A trace of execution	8
4	Reference manual	10
4.1	The memory manager	10
4.2	The callback structure	11
4.3	The exported functions	12
4.4	Adding a new exported function	13
4.4.1	Create an exported BMI function	13
4.4.2	Update the table of exported functions	14
4.4.3	Update Makefiles	14
4.4.4	Create an entry point in Balsa	15
4.4.5	Add a method in the Python package	15
4.4.6	Add a test	16

1 Introduction

1.1 Introduction to the 2023 version

The BMI library was initially (2008) designed as the BLAD Maple Interface library. It was the C library at the interface between the Maple `DifferentialAlgebra` package and BLAD. Since then, it has been extended to stand at the interface of the Sagemath `DifferentialAlgebra` package (2012) and the Python/sympy `DifferentialAlgebra` package (2023). The name BMI today stands for BLAD Multiple Interfaces.

The BMI library now is one of the components of the `DifferentialAlgebra` project, hosted on codeberg.org.

Maintaining an interface library between BLAD and regularly changing scientific computing software such as Maple, Sagemath and Python is time consuming and, in 2023, both the Maple and the Sagemath interfaces are not working and should be fixed.

The difficulty in the Maple case, lies in the fact that BLAD is supposed to allocate all its memory in the Maple environment, to use the version of GMP shipped with Maple and to be interrupted when the user presses the stop button. Note that an old version of BLAD and BMI are shipped with Maple and used through the Maple `DifferentialAlgebra` package.

The difficulty in the Sagemath case is that the mechanism which links to Sagemath external libraries changes regularly and the relevant information seems difficult to obtain.

So far, the Python/sympy version seems the easiest to maintain.

2 Configuration and Building

The BMI library copying conditions

The BMI library is *free*; this means that everyone is free to use them and free to redistribute it on a free basis. The library is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this library that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the libraries, that you receive source code or else can get it if you want it, that you can change these libraries or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the BMI library, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the BMI library. If they are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the BMI library are found in the Lesser General Public License version 2.1 that accompanies the source code, see `COPYING`.

2.1 Building the development version

The `git` versioning system does not preserve time stamps. For this reason, if you download the development version of BMI through a `git clone` command, you need the `autotools`. Have a look to `blad/tools/autogen` and `blad/tools/autogen_clean`.

2.2 Building the BMI library

The BMI library (BLAD Multiple Interfaces) rely on the GNU MP (GMP) library. Make sure GMP (the development package) is available on your computer. If it is not, see the GMP web pages at <http://gmplib.org>.

They have an `autoconf/automake/libtool` based configuration system. All the usual `autoconf` configure options are available. Run `./configure --help` for a summary.

2.3 Building the BMI library for Python/sympy (Linux)

Building the library requires Python and a Cython compiler.

Let us assume the `bmi` directory is a subdirectory of `${HOME}/DifferentialAlgebra`. Let us assume that `blad` has been built and installed in `DifferentialAlgebra/blad-install`. It is recommended to build BMI in a subdirectory `DifferentialAlgebra/bmi-build-sympy`.

The `tools` subdirectory of `bmi` contains a few tools including calls to `configure`. Thus, on a Linux machine, from the `DifferentialAlgebra/bmi-build-sympy` directory, you may try:

```
../bmi/tools/bmi_configure_for_sympy
make
make check
make pdf
make install
```

The last command installs BMI in `DifferentialAlgebra/bmi-install-sympy`. The most important subdirectory is `bmi-install-sympy/DifferentialAlgebra`. From this directory, run

```
python3 setup.py build
pip install . --upgrade
```

Then the `DifferentialAlgebra` package gets installed in Python environment. Note that this last step is still experimental and should be redesigned soon: how to deal with compiler options? what about a wheel package?

2.3.1 A short test

```
$ python3
Python 3.11.6 (main, Oct 3 2023, 00:00:00) [GCC 13.2.1 20230728 (Red Hat 13.2.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import *
>>> from DifferentialAlgebra import *
>>> init_printing ()
>>> x = var('x')
>>> y = indexedbase ('y')
>>> R = DifferentialRing (derivations = [x], blocks = [y], notation = 'jet')
>>> ideal = R.RosenfeldGroebner([y[x]**2 - 4*y])
>>> [ C.equations (solved=True) for C in ideal ]
[[ 2      ]
 [[y[x]  = 4 y], [y = 0]]
```

2.4 Building the BMI library for Maple

A version of the GMP library, needed by BLAD and BMI, is distributed with Maple (without any header file, though). At runtime, the BMI library should use this version of GMP.

The BMI library has an `autoconf/automake/libtool` based configuration system. All the usual `autoconf` configure options are available. Run `./configure --help` for a summary. To install the BMI library (by default in `/usr/local`) you need to perform one of the two following steps:

Assume that the command which runs the Maple computer algebra system is `maplecmd` (it should be `maple`) and assume that the BLAD libraries are installed in some directory `bladdir`. Enter the following commands from the directory (usually `bmi`) which contains the source code:

```
MAPLE=maplecmd ./configure --with-blad=bladdir --enable-maple=yes
make
make pdf (optional)
make install
```

2.4.1 Maple related issues

The BMI library is supposed to be used from Maple, version 14 or higher. It used to be possible to use them on Maple 11, 12 and 13 but this feature is not maintained. Moreover, the DifferentialAlgebra package appears, for the first time, in Maple 14.

In order to use the BMI library from Maple, it is necessary to replace the existing (old) library (on Linux systems, it should be called `libbmi.so`) by the new one in the Maple bin directory. The location of the Maple bin directory can be obtained through the following command (here, run on my machine):

```
$ maple
  |\~/|      Maple 14 (X86 64 LINUX)
._|\|      |/_|. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2010
 \ MAPLE /   All rights reserved. Maple is a trademark of
  <---->    Waterloo Maple Inc.
    |      Type ? for help.
> kernelopts(bindir);
      "/usr/local/maple14/bin.X86_64_LINUX"
```

This replacement must be performed manually. The `make install` command does not perform it. Of course, it is strongly suggested to make a copy of the old library `libbmi.so` before replacing it.

Similar considerations hold for the BLAD libraries.

2.4.2 BLAD related issues

Starting from Maple 14, a version of the BLAD libraries are shipped with Maple. However, the header files of the BLAD libraries are not distributed with Maple. You need them to rebuild BMI. It is thus recommended to download the latest version of the BLAD libraries, build and install before building BMI.

2.4.3 GMP related issues

The GMP library is required in order to build the BMI library. There are two difficulties to overcome. First, you should make sure that the linker finds the version of the GMP library that is shipped with Maple. Second, you should have a header file `gmp.h` for that library. This header file is not distributed with Maple.

2.4.4 A short test

After installation of the BMI library in the Maple bin directory, the following commands permit to test, at least, that the linker successfully loads BMI from Maple:

```
$ maple
  |\~/|      Maple 14 (X86 64 LINUX)
._|\|      |/_|. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2010
 \ MAPLE /   All rights reserved. Maple is a trademark of
  <---->    Waterloo Maple Inc.
    |      Type ? for help.
> with (DifferentialAlgebra):
> R := DifferentialRing (derivations = [t], blocks = [u]);
      R := differential_ring

> ideal := RosenfeldGroebner (diff(u(t),t)^2-4*u(t), R);
      ideal := [regular_differential_chain, regular_differential_chain]
```

```
> Equations (ideal, solved);
      /d      \2
[[|-- u(t)|  = 4 u(t)], [u(t) = 0]]
      \dt      /
```

2.4.5 Miscellaneous

If you are using Maple 11, 12 or 13 (say 13), you should pass the parameter `-DMAPLE13` among the `CPPFLAGS`.

You can also enable some more tests in BMI by providing the parameter `-DBMI_MEMCHECK` among the `CPPFLAGS`. Turning it on may help discover some linker issues on MAC OS X and MS-Windows.

On Windows, you should turn on the `-DBMI_BUILDING` parameter among the `CPPFLAGS`.

2.5 Building the BMI library for Sagemath

When built with this setting, a fake Maple API, called BALSA, is compiled. The `sage` directory contains the Cython `DifferentialAlgebra.pxy` file. At make time, some other Cython related files (`bmi_c.pxd` and `bmi_strings.pxy`) are built from the C code. At install time, these three files are copied in the directory given in the `$sagedevelldir` variable, which takes a default value if not set at configure time. A possible call to `configure` could be as follows:

```
./configure --with-blad=bladdir --enable-sage=yes
make
```

Assume now that you have installed Sage on your computer. Moreover, assume that you have run, in your Sage directory:

```
sage -clone blad
```

Then, you may want to configure BMI as follows:

```
sagedevelldir=$sagedir/devel/sage-blad/sage/libs/blad \
./configure --with-blad=bladdir --enable-sage=yes
```

Then, the following command will install the Python/Cython files `DifferentialAlgebra.pxy`, `bmi_c.pxd` and `bmi_strings.pxy` in `sagedevelldir`.

```
make install
```

The `sage` subdirectory of BMI involves a Linux shell script, called `rebuild`, which may help you installing BLAD, BMI in the Sage environment.

2.5.1 A short test

```
$ sage
-----
| Sage Version 4.8, Release Date: 2012-01-20                |
| Type notebook() for the GUI, and license() for information. |
-----
Loading Sage library. Current Mercurial branch is: blad
sage: from sage.libs.blad.DifferentialAlgebra import DifferentialRing
sage: t = var('t')
sage: u = function('u')
sage: R = DifferentialRing (derivations = [t], blocks = [u])
sage: R
differential_ring
```

```
sage: ideal = R.RosenfeldGroebner (diff(u(t),t)^2-4*u(t))
sage: [ C.equations (solved = true) for C in ideal ]
[[D[0](u)(t)^2 == 4*u(t)], [u(t) == 0]]
```

3 Interaction with scientific computing software

For each scientific computing software, there exists a package in the software language plus some C code. Part of this C code is common to all versions: it is located in the `src` and `src/exported` directories.

This C code was initially written for Maple. It contains many callbacks to the Maple API. When writing the Sagemath and the Python/sympy interfaces, it was decided to implement a fake Maple API, called BALSAs, in order to avoid a complete rewriting of the content of `src` and `src/exported`.

The C code of BALSAs is located in `bmi/balsa/bmi_balsa.c`.

3.1 The Maple version

It is located in the `bmi/maple` directory.

The Maple package is called `DifferentialAlgebra0` to avoid conflicts with the `DifferentialAlgebra` package shipped with Maple. Its source code is located in `maple/maplesrc`.

The main entry point is the `bmi_blad_eval` function.

3.1.1 A trace of execution

The `bmi_blad_eval` function (in `src/bmi_blad_eval.c`) is the main entry point between Maple and BMI. The `bmi_blad_eval_python` is the main entry point between Sage and BMI. Here is a basic call to this function, in the Maple case:

```
$ maple
> bmi_entry_point :=
  ExternalCalling:-DefineExternal
    ('bmi_blad_eval',
      ExternalCalling:-ExternalLibraryName("bmi"));

bmi_entry_point := proc()
option
call_external, define_external(bmi_blad_eval, MAPLE, LIB = "libbmi.so");
  call_external(0, 139903675853922, true, false, args)
end proc
```

Here is now the Maple data that is passed to `bmi_blad_eval` in the case of the command `DifferentialRing(derivations = [t], blocks = [u])`.

```
bmi_entry_point
  ("DifferentialRing"("[t]", "[u]", "[]"), "jet", "jet", "0", "0", "small");

      [ 0..337 1-D Array      ]
      [ Data Type: integer[4] ]
      [ Storage: rectangular  ]
      [ Order: C_order        ]
```

The `bmi_blad_eval` receives six parameters. The first one is the command to be performed, in this case `"DifferentialRing"("[t]", "[u]", "[]")`. The five last ones are so-called *options* which permit to tune computations. These options are the input notation, the output notation, the time limit, the memory limit and an indication on the size of the *cells* that BLAD should allocate.

In this case, the result of the command is a Maple `rtable` which contains a BLAD data structure. The `DifferentialRing` Maple function then wraps this `rtable` in a Maple `table` (together with some other information) and returns it. More generally, calls to `bmi_blad_eval` may return any kind of Maple expression.

```
> R := DifferentialRing (derivations = [t], blocks = [u]);
      R := differential_ring

> R[Ranking];
      [ 0..337 1-D Array      ]
      [ Data Type: integer[4] ]
      [ Storage: rectangular  ]
      [ Order: C_order        ]
```

3.2 The Python/sympy version

It is located in the `bmi/sympy` directory.

Its source code is the Cython file `DifferentialAlgebra.pyx`.

To each function of this package there is a corresponding C function, located in the `bmi/balsa/bmi_dapyx.c` file. This function interacts with Balsa to build a fake Maple object that can be processed by the `bmi_blad_eval_python` function, which is a close variant of `bmi_blad_eval`.

3.2.1 A trace of execution

The following trace of execution was obtained in the `bmi-build-sympy/sympy` directory. The C compiler and the linker were called with the `-g` option and no optimization.

Our goal is to follow the execution of the function `indets` of the `DifferentialAlgebra` package. We run `python3` under `gdb`, set a breakpoint at `bmi_dapyx_indets` and carry out an example.

```
$ PYTHONPATH=. gdb --args python3
(gdb) break bmi_dapyx_indets
(gdb) run
>>> from sympy import *
>>> from DifferentialAlgebra import *
>>> x = var('x')
>>> y = indexedbase('y')
>>> R = DifferentialRing (derivations = [x], blocks = [y], notation = 'jet')
>>> R.indets([x, y, y[x]])
Breakpoint 1, bmi_dapyx_indets (eqns=0x7fffe84eeb70 "[x, y, y[x]]",
  DRing=0x5555563125f0, selection=0x7fffe84ede20 "derivatives",
  fullset_or_var=0x7ffff7eea830 <_PyRuntime+11440> "0",
  notin=0x7fffe84eeb40 "jet", notout=0x7fffe84eeb40 "jet", timeout=0,
  memout=0)
```

The `eqns` argument contains the list `[x, y, y[x]]` as a string. The two following arguments contain default values (see the variants of the `indets` function). The four last arguments are `notin` (the input notation), `notout` (the output notation), `timeout` and `memout` (limits on the time and the memory allowed to this computation; here no limits are given). The function `bmi_dapyx_indets` creates a fake Maple object `~F`, reproducing the one that would have been created in Maple. For instance, it calls `MapleListAssign` which is faked in `bmi_balsa.c`. Eventually, it calls the `eval` local function.

```
eval (F=0x555555a82a60, notin=0x555555e63ac0, notout=0x555555c3c1f0,
```

```
timeout=0, memout=0, cellsize=0x7fffe9927998 <small>)
```

In addition to the former arguments, the `cellsize` argument indicates that the computation is expected to require a small amount of memory. The function then call `bmi_blad_eval_python`.

```
bmi_blad_eval_python (kv=0x0, args=0x555555762810)
```

The names of the two arguments are mimicked from the Maple API. The function tunes some settings of the BLAD libraries and calls `bmi_process_command`.

```
bmi_process_command (result=0x7fffffdd600, options=0x7fffffdd620,  
callback=0x7fffe9928650 <mem+40>)
```

This function tunes some further settings of the BLAD libraries and calls the `bai_restart` function of BLAD. It then processes the command within a `BAO_TRY/BAO_CATCH/BAO_ENDTRY` block. Every exception raised in BLAD would thus be eventually caught here and returned gracefully to Python. This function calls `bmi_call_exported` (located in `bmi/src`)

```
bmi_call_exported (name=0x7fffe99220c5 "Indets",  
callback=0x7fffe9928650 <mem+40>)
```

which in turn, calls `bmi_indets` (located in `src/exported`)

```
bmi_indets (callback=0x7fffe9928650 <mem+40>)
```

Using its `callback` argument, `bmi_indets` performs a few queries to Maple (in this case, to BALSAs) in order to recover the relevant information. Eventually, it transforms its result as a string and returns it. Many functions return strings. Functions which build differential rings or ideals return internal BLAD objects.

4 Reference manual

`ALGEB bmi_blad_eval (MKernelVector kv, ALGEB args)` [Function]

The main entry point. The *args* argument points to the six parameters of any call coming from Maple. The `MapleRaiseError` function may be called if an error occurs. This function does not keep any information between two calls, i.e. BLAD is restarted and terminated at each call.

The `bmi_blad_eval` function performs the following operations. First, it initializes the memory manager (`bmi_init_memory`). Second, it processes the options (`bmi_set_options_maple`). Third, it processes the command (`bmi_process_command`). Lastly, it returns the result.

`ALGEB M_DECL bmi_blad_eval_python (MKernelVector kv, ALGEB args)` [Function]

Similar to `bmi_blad_eval`, for Sage.

`static bool bmi_process_command (ALGEB* result, struct bmi_options* options, struct bmi_callback* callback)` [Function]

Called by `bmi_blad_eval`. Processes the command. Restarts BLAD and puts a general exception catching point, to catch any exception raised from BLAD. If an exception is raised by the BLAD parser, manages to build an error message giving the approximate error location, before calling `MapleRaiseError`. Calls the `bmi_callback_exported` function. If no exception is raised, gets the result (an `ALGEB`), terminates BLAD and returns the result.

4.1 The memory manager

It maintains a table of all *cells* allocated by BLAD. The main issue is: these cells are allocated through the Maple `MapleGcAlloc` function. They must be protected from the Maple garbage collector during computations. They must be unprotected at the end of the call, if an error occurs or if the user presses control-C.

The memory manager must also take care to the GMP memory management functions. During BLAD computations, GMP must allocate memory within BLAD cells, using BLAD allocators. Outside BLAD computations, it must allocate memory within Maple memory, using Maple allocators.

`struct bmi_callback* bmi_init_memory (MKernelVector kv)` [Function]

Initializes the memory. It returns a *callback* function which permits the BMI exported functions to call back Maple to get further information.

The `bmi_init_memory` function performs the following operations. First, it clears the global table `mem` in which BLAD cells are stored. Second, it makes GMP allocators point to BLAD allocators (`bmi_init_gmp_allocators_management`). Third, it initializes the global callback structure. Fourth, it makes BLAD cell allocators point to BMI allocators which, eventually, call Maple allocators (`ba0_set_settings_memory_functions`, `MapleAlloc`). Fifth, it makes BLAD interrupt function point to BMI interrupt function which, eventually, calls the Maple interrupt function (`ba0_set_settings_interrupt`, `MapleCheckInterrupt`). Lastly, it returns a pointer to the callback structure. The `mem` data structure is as follows.

```

static struct bmi_memory {
/*
 * areas = a table containing the allocated cells of BLAD.
 *
 * This table and its elements are MAPLE BINARY DAGS, allocated
 * using MapleAlloc and, then, converted to BINARY DAGS by means
 * of BMI_POINTER_TO_BINARY.
 */
    ba0_int_p alloc;
    ba0_int_p size;
    ALGEB areas;
/*
 * For debugging purposes, a counter of calls to MaplePushErrorProc
 * MAPLE 13.
 */
    ba0_int_p maple_sp;

    MKernelVector kv;
    struct bmi_callback callback;
} mem;

```

static void* bmi_malloc (size_t n) [Function]
 This function is called when BLAD allocates a cell. The cell is allocated through MapleAlloc, converted to a Maple BINARY, protected from garbage collection and stored in mem. If needed, mem is resized. The allocated cell is then returned to BLAD.

static void bmi_interrupt (void) [Function]
 This function is regularly called from BLAD to check if the user pressed control-C. It sets the GMP allocators to their Maple values. It calls MapleCheckInterrupt. If this call returns, it sets the GMP allocators to their BLAD values. Otherwise, the following function is called:

static void bmi_error_proc (char* mesg, void* data) [Function]
 This function is called from MapleCheckInterrupt if the user stroke control-C. It gives back to the Maple garbage collector all the BLAD allocated cells which are stored in mem.

4.2 The callback structure

Assume you have a pointer to a Maple table and you want to get the `ordering` entry of that table. Then, store the pointer to this table in the `arg` field using the `bmi_set_callback_maple` function and call the `bmi_table_ordering_op` function. The declaration is as follows.

```

struct bmi_callback {
    MKernelVector kv;
    ALGEB arg;
    ALGEB op;
/* The object the following functions apply to */
/* MAPLE keywords */

```

```

    ALGEB nops;
                                /* indices in MAPLE tables */
    ALGEB ordering;
    ALGEB equations;
    ALGEB notation;
    ALGEB type;
};

void bmi_init_callback (struct bmi_callback* callback,      [Function]
                       MKernelVector kv)
    Initializes callback with kv.

void bmi_set_callback_ALGEB (struct bmi_callback*          [Function]
                             callback, ALGEB arg)
    Stores arg in the field arg of callback. The following commands then apply to arg.

bool bmi_is_string_op (long k, struct bmi_callback*        [Function]
                      callback)
    Returns true if op (k, callback) is a Maple string or a Maple name.

ALGEB bmi_table_ordering_op (long k, struct bmi_callback*  [Function]
                             callback)
    Returns the Ranking entry of op (k, callback), assuming that this last expression
    is a table, involving such an entry.

```

Many similar functions are listed in `bmi_callback.h`.

4.3 The exported functions

The file `bmi_exported.c` involves a table listing the names of the commands and the corresponding BMI exported functions. These exported functions reside in the `src/exported` directory.

```

struct bmi_exported_pair {
    char* name;
    ALGEB (*pointer) (struct bmi_callback*);
};

static struct bmi_exported_pair bmi_exported_table [] = {
    { "AllDerivatives", &bmi_all_derivatives },
    { "Attributes", &bmi_attributes },
    { "BaseFieldGenerators", &bmi_base_field_generators },
    { "BelongsTo", &bmi_belongs_to },
    { "Coeffs", &bmi_coeffs },
    { "DeltaPolynomial", &bmi_delta_polynomial },
    { "DifferentialPrem", &bmi_differential_prem },
    { "DifferentialRing", &bmi_differential_ring },
    ...
    { "Separant", &bmi_separant },
    { "SortByRank", &bmi_sort_by_rank },
    { "Tail", &bmi_tail }
};

ALGEB bmi_call_exported (char* name, struct bmi_callback*  [Function]
                        callback)
    Looks for name in bmi_exported_table and calls the corresponding function.

```

4.4 Adding a new exported function

Let us take the example of the `positive_integer_roots` method of the `RegularDifferentialChain` class of the Python package. It is supposed to eventually call the `bad_positive_integer_roots_mod_regchain` of the BLAD libraries.

Note: these files have actually been removed from BLAD and BMI. This example should thus be changed to another one in a short while.

4.4.1 Create an exported BMI function

Two new files `bmi_positive_integer_roots.c` and `bmi_positive_integer_roots.h` must be created in the `src/exported` directory. Here is the source file:

```
#include "bmi_exported.h"
#include "bmi_mesgerr.h"
#include "bmi_indices.h"
#include "bmi_positive_integer_roots.h"

/*
 * EXPORTED
 * Positive_integer_roots (ratfrac, variable, differential ring)
 *
 * Return a list of positive integers
 */

ALGEB
bmi_positive_integer_roots (
    struct bmi_callback *callback)
{
    struct baz_ratfrac Q;
    struct bav_variable *q;
    struct ba0_tableof_mpz roots;
    char *variable, *ratfrac, *stres;
/*
 * There must be three arguments.
 * The last one must be a table
 */
    if (bmi_nops (callback) != 3)
        BAO_RAISE_EXCEPTION (BMI_ERRNOPS);
    if (!bmi_is_dring_op (3, callback))
        BAO_RAISE_EXCEPTION (BMI_ERRDRNG);
/*
 * The table contains a differential ring
 * Restore these data first
 */
    bmi_set_ordering (3, callback, __FILE__, __LINE__);
/*
 * At this stage, we have a working differential ring
 *
 * Get the variable as a string and use BLAD parsers to convert it
 * into a BLAD variable
 */
    variable = bmi_string_op (2, callback);
    ba0_sscanf2 (variable, "%v", &q);
/*
 * Get the ratfrac as a string and parse it also.
 * The parser to be used is not exactly the same if we are called from
 * Maple or from BALSAs
 */
}
```

```

*/
    ratfrac = bmi_string_op (1, callback);
    baz_init_ratfrac (&Q);
#if ! defined (BMI_BALSA)
    ba0_sscanf2 (ratfrac, "%simplify_expanded_Qz", &Q);
#else
    ba0_sscanf2 (ratfrac, "%Qz", &Q);
#endif
/*
 * Go
 */
    ba0_init_table ((struct ba0_table *) &roots);
    baz_positive_integer_roots_polynom_mpz (&roots, &Q.numer, q);
/*
 * Convert the result as a string (stres)
 */
    stres = ba0_new_printf ("%t[%z]", &roots);
/*
 * Convert stres into a result (res)
 */
    {
        ALGEB res;
        bmi_push_maple_gmp_allocators ();
#if ! defined (BMI_BALSA)
/*
 * res is the Maple object obtained by evaluating the string
 * with Maple interpreter
 */
        res = EvalMapleStatement (callback->kv, stres);
#else
/*
 * res is a BALSA object of type string wrapping stres
 * It will also be evaluated but at the Cython level
 */
        res = bmi_balsa_new_string (stres);
#endif
        bmi_pull_maple_gmp_allocators ();
        return res;
    }
}

```

4.4.2 Update the table of exported functions

This is achieved by modifying `src/bmi_exported.c`. Add one `#include "bmi_positive_integer_roots.h"` statement at the beginning of the file and one line `{"PositiveIntegerRoots", &bmi_positive_integer_roots }` in the array `bmi_exported_table`.

4.4.3 Update Makefiles

The Makefiles to be modified are `sympy/Makefile.am` and `maple/Makefile.am`. One simply needs to add the names of the two new files in the corresponding lists.

4.4.4 Create an entry point in BALSAL

This is achieved by adding one new function `bmi_dapyx_positive_integer_roots` in the file `balsa/bmi_dapyx.c` and its prototype in the file `balsa/bmi_dapyx.h`. This function is the one which is going to be called from the Cython library. Here is its code:

```
/*
 * positive_integer_roots (ratfrac, variable, regchain | differential ring)
 *
 * The three first arguments are the above ones
 * The four last arguments are systematically passed:
 * - input notation
 * - output notation
 * - time limit before time out
 * - memory limit before memory out
 */

ALGEB_string
bmi_dapyx_positive_integer_roots (
    char *ratfrac,
    char *variable,
    ALGEB regchain,
    char *notin,
    char *notout,
    ba0_int_p timeout,
    ba0_int_p memout)
{
    ALGEB F;
    static struct bmi_balsa_object PositiveIntegerRoots =
        { bmi_balsa_string_object, "PositiveIntegerRoots", false, 0 };
    /*
    * F encodes the function call as an ALGEB
    * The function name is:      PositiveIntegerRoots
    * The number of argument is: 3
    */
    F = bmi_balsa_new_function (&PositiveIntegerRoots, 3);
    MapleListAssign (0, F, 1, bmi_balsa_new_string (ratfrac));
    MapleListAssign (0, F, 2, bmi_balsa_new_string (variable));
    MapleListAssign (0, F, 3, regchain);
    /*
    * Let us call the static function eval
    * - The notations may be "undefined" and then need be fixed
    * - Use large cell sizes for BLAD computations
    */
    return (ALGEB_string) eval
        (F,
         fix_notation (notin, regchain),
         fix_notation (notout, regchain), timeout, memout, &large);
}
```

4.4.5 Add a method in the Python package

This is achieved by adding one new method in the file `RegularDifferentialChain.pyx`. Here is the code of the method:

```
# positive_integer_roots
def positive_integer_roots (
    self,
    ratfrac,
    variable,
```

```

        notation = BMI_IX_undefined):
    cdef result
    cdef bytes msgerr, str_notation, str_ratfrac, str_variable
    cdef bmi_c.ALGEB_string L
    str_notation = self.__process_notation (notation)
    str_ratfrac = ascii (ratfrac)
    str_variable = ascii (variable)
    L = bmi_c.bmi_dapyx_positive_integer_roots (
        str_ratfrac, str_variable, self.regchain, self.notation,
        str_notation, 0, 0)
    if bmi_c.bmi_dapyx_is_error (L):
        msgerr = bmi_c.bmi_dapyx_msgerr (L)
        bmi_c.bmi_balsa_clear_ALGEB (L)
        raise RuntimeError (str (msgerr, 'utf-8'))
    result = eval (L.value, globals (), self.evdict)
    bmi_c.bmi_balsa_clear_ALGEB (L)
    return result

```

4.4.6 Add a test

This is achieved by adding one file `test48-py` (over this example) in the `sympy/tests` directory and list `test48.sh` in `sympy/tests/Makefile.am`. Here is the test file called `test48-py`. The test file is mostly needed to test the overall configuration. From a mathematical point of view, the BLAD function is already tested within the BLAD library.

```
# positive_integer_roots
```

```

from sympy import *
from DifferentialAlgebra import *
import unittest

```

```
class TestDifferentialAlgebra(unittest.TestCase):
```

```
    def test_48(self):
```

```
        x,q,a,b = var('x,q,a,b')
```

```
        y1,y2 = indexedbase ('y1,y2')
```

```
        R = DifferentialRing (derivations = [x], blocks = [[y2,y1],[q,a,b]], parameters = [q,a,b], notation = '')
```

```
        P = y1*y2*(q - 27)*(q-1)*(q**2 - 2) + y1**2*(q - 27)*(q-1)**2 + (q - 27)*(q-1)*(q - 3)
```

```
        L = R.positive_integer_roots (P, q)
```

```
        self.assertEqual (L, [1, 27])
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```