The Mathemagix packages

Table of contents

Chapter 1
Mmxlight

1.1. Using the Mmx-light interpreter

1.1.1. Terminal interface

The Mmx-light interpreter starts with the command:

```
Shell] mmx-light
```

```
--------------------------------------------------------------
|:*)          Welcome to Mathemagix-light 0.4            (*:|
|------------------------------------------------------------|
|  This software falls under the GNU General Public License  |
|          It comes without any warranty whatsoever           |
|                    www.mathemagix.org                       |
|                     (c) 2001--2010                          |
--------------------------------------------------------------
```

If you use the interpreter in a textual context, several shortcuts can be very useful such as:

- `[Ctr-A]`, `[Ctr-E]` to go to the beginning or end of the instruction line,
- $\uparrow$, $\downarrow$ for the previous and next instruction lines,
- `[tab]` for name completion,
- `[Meta-return]` to add an extra line to the current instruction line when using a text terminal. Note that on several platforms the `[Meta]` key is binded to `[Alt]`. On the Mac OS X terminal, unless the `[Alt]` key has been specifically set so from the terminal preferences menu, one must type `[esc]` and then `[return]`. Within a TeXmacs session, one has to type `[Shift-return]`.

Different modes are available:

- `type_mode? := true;` to print the type information with the result of an evaluation in the interpreter.
- `time_mode? := true;` to print the time needed to evaluate an instruction line when it is bigger than 1 ms.
- `debug_mode? := true;` to print debug information when an error occurs;

- `quiet_mode? := true;` to print only the input command and opt result of an evaluation (no prompt, no banner);

1.1.2. Loading Mathemagix files

A classical way to develop Mathemagix code is to edit files and load them from the interpreter. As an example, we give below the content of a file `step1.mmx` (in `mmxtools/mmx`):

```
f (n: Int): Int == {
  if n < 2 then return n;
  else {
    mmout << n << "\n";
    if n mod 2 = 0 then return f (n quo 2);
    else return f (3*n + 1);
  }
}
```

It can be used from the interpreter as follows:

```
Mmx] include "mmxlight/step1.mmx"
```

```
Mmx] f 5
```

```
5
16
8
4
2
1
```

1.1.3. Using Mmx-light files as scripts

The interpreter can be used directly with a file:

```
mmx-light step1.mmx
```

Mathemagix files can also be used as *scripts*, as shown in this example:

```
#!/usr/bin/env mmx-light
m: Double := 0.0;
for i in 1..#argv do
  m := m + as_double argv[i];
mmout << "Mean: " << m / (#argv-1) << "\n";
```

If this code is saved into a file, say `mean`, it can be used as follows:

```
chmod u+x mean
./mean 3.2 5.6 -1.7

Mean: 2.36666666667
```

provided that the command `mmx-light` is available.

1.1.4. Loading a package

The command to load and use the types and functions exported by an external library is `use`:

```
use "package";
```

Here the dynamic library `libmmxpackage.so` will be searched in the loading path (see variable `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH`).

1.1.5. The environment files

The first time the shell is launched it creates a `.mathemagix` in the home directory. A warning message is printed.

The file `.mathemagix/etc/boot.mmx` is automatically loaded at startup. This is the right place to customize the shell and to load the packages you frequently use. In order to load packages you can proceed as follows:

```
if supports? "numerix" then use "numerix";
if supports? "algebramix" then use "algebramix";
```

Prior to user's boot file a global boot file (usually `/usr/local/etc/mathemagix/boot.mmx`) is loaded. In case you wish to disable both boot files use the option `--noboot` within the `mmx-light` command.

Within an interactive session, ending a line with a ';' actually means finishing with a null instruction. As a consequence this extra ';' prevents from printing the output of the previous instruction.

1.1.6. Help function

Help on functions and types can be obtained with the `help` command:

```
Mmx] help Generator Generic
```

```
Generator (Generic)                                              (Class)

    Available functions

    != : (Generator (Generic), Generator (Generic)) -> Boolean
    = : (Generator (Generic), Generator (Generic)) -> Boolean
```

```
@ : Generator (Generic) -> Generator (Generic)
alias : Generator (Generic) -> Alias (Generator (Generic))
flatten : Generator (Generic) -> Syntactic
```

```
Mmx] help infix +
```

```
+ : (Int, Int) -> Int                                   (Native)
+ : (Double, Double) -> Double                          (Native)
+ : (Syntactic, Syntactic) -> Syntactic                 (Native)
```

1.2. Declaration and control sequences

1.2.1. Variables and constants

1.2.1.1. Identifiers

Identifiers are formed using letters, digits and the special characters `_` and `?`. Identifiers must start with a letter or `_`. Identifiers match the `[a-zA-Z_]+[a-zA-Z0-9_?]*` regular expression.

Examples of identifiers are `x`, `y2`, `my_var` and `test?`.

Some special predefined constants are:

`true`. The boolean constant `true`.

`false`. The boolean constant `false`.

`mmout`. The standard output stream.

`mmerr`. The standard error stream.

Notice that streams can be used with the operators `<<` as in C++:

```
mmout << 3 << "\n";
```

1.2.1.2. Literals

`mmx-light` supports three types of literal constants:

String literals. Strings can be braced into double quotes `" ... "`. Inside such a string a double quote must be blackslashed. In order to avoid blackslashing one can use the stronger string delimiters `/" ... "/`.

```
s1 := "This is a string";
s2 := "Print "foo"";
s2 := /"Print "foo"/;
```

Integer literals. An integer literal is a sequence of digits, possible preceded by a minus sign. It matches the regular expression `[-]?[0-9]+`. Examples: `123456789123456789`, `-123`.

Floating literals. A floating literal is a sequence of digits with a decimal point inside and an optional exponent. It matches the regular expression `[-]?[0-9]+[.][0-9]+[[eE][-]?[0-9]+]?`. Some examples:

```
z := 0.0;
w := -3.14159;
x := 1.11e2007
```

Note that `0.` is not permited, one must write `0.0`.

1.2.1.3. Definition and assignation

To assign a value to a variable, we use the operator `:=`, while constant values are defined by the operator `==`, and cannot be modified hereafter.

```
Mmx] a1 := 1
```

1

```
Mmx] a1_? == 2; a1_?
```

2

```
Mmx] cst: Int == 11111*111111
```

1234554321

```
Mmx] mut: Int := 1010*1234
```

1246340

```
Mmx] cst := - cst
```

```
1:1: exception, expected Alias \/ Tuple
cst := - cst
~~~
```

```
Mmx] mut := - mut
```

−1246340

1.2.2. Operators

Here we list the operators in increasing priority order, together with their internal names:

==	DEFINE	
:=	ASSIGN	
==>	DEFINE_MACRO	
:=>	ASSIGN_MACRO	
+=	PLUS_ASSIGN	
-=	MINUS_ASSIGN	
*=	TIMES_ASSIGN	
/=	OVER_ASSIGN	
<<	LEFT_FLUX	
<<*	LEFT_FLUX_VAR	
<<%	LEFT_FLUX_STR	
>>	RIGHT_FLUX	
=>	IMPLIES	
<=>	EQUIVALENT	
or	SEQOR	
	OR	
xor	XOR	
and	SEQAND	
	AND	
=	EQUAL	
<	LESS	
<=	LEQ	
>	GREATER	
>=	GEQ	
!=	NOT_EQUAL	
!<	NOT_LESS	
!<=	NOT_LEQ	
!>	NOT_GREATER	
!>=	NOT_GEQ	
->	INTO	
:->	MAPSTO	
..	RANGE	
to	RANGEEQ	
+	PLUS	
-	MINUS	
@+	OPLUS	
@-	OMINUS	
*	TIMES	
/	OVER	
@*	OTIMES	
@/	OOVER	
div	DIV	
mod	MOD	
@	COMPOSE	
!	NOT	infix unary
-	MINUS	infix unary
@-	OMINUS	infix unary
++	INC	infix unary
--	DEC	infix unary
#	SIZE	infix unary
^	POWER	
++	INC	postfix unary
--	DEC	postfix unary
!	NOT	postfix unary
'	QUOTE	postfix unary
`	BACKQUOTE	postfix unary
~	TILDA	postfix unary
.	ACCESS	

Whenever writing a complex expression you should carefully consider these priority rules in order to avoid using extra parentheses. For example the expression

```
if ((a*b)+(c)>(d)) then foo ();
```

should be better written

```
if a*b + c > d then foo ();
```

The Mathemagix grammar is specified in the file `mmx-parser.ypp` of the `basix` module.

1.2.3. Control flow

1.2.3.1. Sequence of Instructions

Instructions are separated with `;`.

```
x := 1;
y := 3;
z := x*y;
```

1.2.3.2. Conditionals

The construction is as follows: `if condition then` block1 `else` block2.

condition is any expression that evaluates to a boolean. block1 and block2 are either one instruction or a block of instructions braced into `{...}`. The `else` part is optional.

```
if true then 1;
if a = b then 1 else 2;
if i = 0 then { x := 1; y := 2} else { z := 3; mmerr << "error" }
```

For the sake of readability do not write

```
if a = 0 then { b := 0 }
```

but

```
if a = 0 then b := 0;
```

instead, since `{...}` are not needed for a block with one instruction only. Notice that, according to the priority rules of the operators listed above, the expression

```
if ((a = 0) and (b = 0)) then foo ();
```

should be written

```
if a = 0 and b = 0 then foo ();
```

for the sake of readability.

1.2.3.3. Loops

Loops are constructed as follows: [`for` E1] [`while` E2] [`until` E3] [`step` E4] `do` block.

Here `[...]` means that the expression is optional. block is an instruction or a sequence of instruction delimited by `{...}`. `break` exits the loop, and `continue` goes to the next step of the loop.

```
do mmout << "no end "; // This loop has no end
for i in 1..3 do mmout << i << " ";
i := 0; while i < 0 do { mmout << i << " "; i := i + 1 }
```

1.2.4. Comments

Comments starting with `//` extend to the end of the physical line. Such a comment may appear at the start of a line or following whitespace or code, but not within a string literal. Multi-line comments must be braced into `/{ ... }/` and can be nested.

```
x := 1; // Assign 1 to x.

y := 2;
/{ This
   is multi-line
   comment about y.
}/
```

1.3. FUNCTIONS AND MACROS

1.3.1. Function definition

Any function definition can be preceded by the quantifier `forall(...)`.

A function definition is done as follows:

```
name (arg1: type1, arg2: type2, ...): returned_type == ...
```

MMX-LIGHT also provides a syntax for lambda expressions:

```
lambda (arg1: type1, arg2: type2, ...): returned_type do ...
```

Notice that any of the type specifications can be omitted, in which case the type is assumed to be `Generic`.

```
discr (a, b, c) == b^2 - 4*a*c;

forall (R)
gcd (p: Polynomial(R), q: Polynomial(R)): Polynomial(R) == {...}
```

A macro corresponds to a syntactic definition and can be introduced by using `==>`. Expression macros can also be defined by using `macro` instead of `lambda`. No type is needed:

```
Mmx] square x ==> x*x
```

closure

```
Mmx] square 2.1
```

4.41

```
Mmx] square 2
```

4

```
Mmx] disc (a,b,c) ==> b*b - 4*a*c
```

closure

```
Mmx] disc (1,2,3.001)
```

−8.004

1.3.2. Function call

A function or a macro `foo` is called in the usual way: `foo (arg1, arg2,...)`.

If `foo` is unary then `()` can be omited, but note that `foo a b c` is equivalent to `foo (a (b (c)))`. Function call is always by value.

1.3.3. Functions like methods

The definition of a postfix function is preceded by the keyword `postfix`. The name of the function should start with a '.'.

```
Mmx] postfix .square (i : Int) : Int == i*i;
```

```
Mmx] 3.square
```

9

```
Mmx] postfix .m (a: Int) (b: Int) : Double == 1.0 * a * b;
```

```
Mmx] 3.m 3
```

9

1.3.4. Functional programming

Functions can be treated like other objets. Nested functions are supported by Mmx-light.

```
Mmx] shift (x: Int) (y: Int): Int == x + y;
```

```
Mmx] shift 3
```

closure

```
Mmx] (shift 3) 4
```

7

```
Mmx] map (shift 3, [ 1 to 10 ])
```

$[4, 5, 6, 7, 8, 9, 10, 11, 12, 13]$

1.4. Builtin data types

In this section, we describe the basic types of Mmx-light. For the other types, provided by the extension packages, see their respective documentation.

1.4.1. Generic

The default type of an object is `Generic` and the corresponding variable type is `Alias Generic`:

```
Mmx] a := x
```

x

```
Mmx] type a
```

Alias(Generic)

1.4.2. Boolean

The usual boolean constants are `true` and `false`. The equality and inequality tests are `=` and `!=`. To build boolean expressions, we use the operators `and`, `or` and the negation operator `!`.

```
Mmx] a = a
```

true

```
Mmx] a != a
```

false

```
Mmx] a = b and a != c
```

false

```
Mmx] a = b or a != c
```

true

```
Mmx] !( a = b or a != c)
```

false

1.4.3. Strings

Strings can be braced into double quotes `" ... "`. Inside such a string a double quote must be blackslashed. In order to avoid blackslashing one can use the stronger string delimiters `/" ... "/`.

```
Mmx] s1 := "This is a string"
```

"This is a string"

```
Mmx] s2: String := "Print "foo" "
```

"Print "foo" "

```
Mmx] s2 := /" Print "foo" "/
```

" Print "foo" "

```
Mmx] s3 := s2 >< " and "fii" "
```

" Print "foo" and "fii" "

```
Mmx] s2 << /" and "fuu" "/
```

" Print "foo" and "fuu" "

```
Mmx] #s1
```

16

```
Mmx] search_forwards (s2, "Print", 0)
```

1

```
Mmx] replace (s2, "fuu", "haha")
```

" Print "foo" and "haha" "

1.4.4. Machine integers

An integer literal is a sequence of digits, possibly preceded by a minus sign. It matches the regular expression `[-]?[0-9]+`. Examples: `123456789123456789`, `-123`. The default type for integers is `Int`. It corresponds to machine type `int`. The usual arithmetic operators `+`, `-`, `*` are available, as well as the inplace operators `+=`, `-=`, `*=`.

```
Mmx] a := 2
```

2

```
Mmx] a+3; a-5; a*a
```

4

```
Mmx] a += 1; a *= 2; a -= 3
```

3

```
Mmx] 5 div 2
```

2

```
Mmx] 5 rem 2
```

1

1.4.5. Double

By default, the floating point literals are converted to `Double` types. A floating literal is a sequence of digits with a decimal point inside and an optional exponent. It matches the regular expression `[-]?[0-9]+[.][0-9]+[[eE][-]?[0-9]+]?`. Note that `0.` is not permitted, one must write `0.0`;

```
Mmx] 2.1
```

2.1

```
Mmx] 2.1*3
```

6.3

```
Mmx] 2.3/2-1
```

0.15

1.4.6. Syntactic

The `Syntactic` type can be used to produce document outputs, represented as lisp-type expressions. It is automatically parsed by TeXmacs to display these outputs.

```
Mmx] type_mode?:=true
```

true: Boolean

```
Mmx] 'x
```

x: Literal

```
Mmx] '(f (x, y, z))
```

$f(x, y, z)$: Compound

```
Mmx] '(f (x, y, z)) [2]
```

y: Literal

```
Mmx] $document ("Some ", $with ("color", "red", "red"), " text.";
                "Pythagoras said ", $math ('(a^2 + b^2 = c^2)), ".")
```

Some red text.
Pythagoras said $a^2 + b^2 = c^2$.

1.4.7. Vectors

Vectors are sequences of elements, stored in an array and with a direct access through their index. Their type is parametrized by the type of the elements. The default type is `Vector Generic`. The indices start from 0. The length of a vector is given by the prefix operator `#`. The concatenation of vectors is performed by the operator `><`. The inplace concatenation of vectors is done by the operator `<<`. The classical operations `car`, `cdr`, `cons` are available on vectors.

```
Mmx] v := [1,2,3]
```

$[1, 2, 3]$: Alias(Generic)

```
Mmx] v[0]+v[1]+v[2]
```

6: Int

```
Mmx] #v
```

3: Int

```
Mmx] w := v >< [4,5]
```

$[1, 2, 3, 4, 5]$: Alias(Generic)

```
Mmx] v << [1,2]
```

$[1, 2, 3, 1, 2]$: Alias(Vector(Generic))

```
Mmx] [car v, cdr v, cons (3, v)]
```

$[1, [2, 3, 1, 2], [3, 1, 2, 3, 1, 2]]$: Vector(Generic)

```
Mmx] reverse v
```

$[2, 1, 3, 2, 1]$: Vector(Generic)

```
Mmx] contains?(v,1)
```

true: Boolean

1.4.8. Tuples

Tuples are written inside `(...)`. Elements are separated by `,`, which is associative, so that `(1, (2, 3))` is the same as `(1, 2, 3)`.

```
Mmx] v := (1, 2, 3)
```

$(1, 2, 3)$

```
Mmx] (1, (2, 3))
```

$(1, 2, 3)$

```
Mmx] v:= [1, 2]; (v, ((v)))
```

$([1,2],[1,2])$

Row-tuples rows are separated by `;`, as exemplified with the constructions of matrices (defined in the ALGEBRAMIX package):

```
Mmx] use "algebramix"
```

```
Mmx] (1, 2; 3, 4; 5, 6)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} : \text{Tuple(Generic)}$$

```
Mmx] [1, 2; 3, 4; 5, 6]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} : \text{Matrix(Integer)}$$

Automatic constructions are possible through the `|` resp. `||` notation:

```
Mmx] ( i^2 | i in 1..3 )
```

$\text{iterator}(1,4): \text{Generator(Generic)}$

```
Mmx] [ i * j | i in 1..3, j in 1..4 ]
```

$[1,2,3,2,4,6]: \text{Vector(Integer)}$

```
Mmx] matrix ( i*j | i in 1..5 || j in 1..5 )
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix} : \text{Matrix(Integer)}$$

1.4.9. Iterators

`a to b` means the range [a,b], while `a..b` stands for the half open range [a,b).

```
Mmx] (1 to 4)
```

$\text{iterator}(1,2,3,4): \text{Generator(Generic)}$

```
Mmx] 1 .. 4
```

$\text{iterator}(1,2,3): \text{Generator(Generic)}$

```
Mmx] [1..6]
```

$[1,2,3,4,5]: \text{Vector(Integer)}$

```
Mmx] [i*i | i in 1 to 10]
```

$[1,4,9,16,25,36,49,64,81,100]$: Vector(Integer)

1.4.10. Tables

Tables allow to store the association between keys of one type and values of another type. They are defined by providing a default value. The default type for tables used in the interpreter is `Table(Generic,Generic)`. In the following example, the default value is `1`:

```
Mmx] t := table(1);
```

```
Mmx] t[1]  := -3; t[34] :=  2
```

2

```
Mmx] t[0]
```

1

```
Mmx] contains? (t,2)
```

false

1.4.11. Ports

There is an output stream, which is called `mmout`. It can be used with the operator `<<` to print strings:

```
Mmx] type_mode?:= true;
```

```
Mmx] mmout
```

formatting_port(output_file_port(mmout)): Port

```
Mmx] mmout << "Hello\n";
```

```
Hello
```

```
Mmx] i := 3; mmout << "The square of " << i << " is " << i*i << "\n";
```

```
The square of 3 is 9
```

Output streams can also be defined from files. Here we write a string into the file `toto.txt`, and we load the contents of this file into a string:

```
Mmx] output_file_port ("toto.txt") << "Hi there\n";
```

```
Mmx] load ("toto.txt")
```

"Hi there\n": String

1.4.12. Handling file

Here are some useful commands to read and save data in files. The command to read a MATHEMAGIX file and to evaluate it is `include`. The command to save a `String` in a file is `save`.

The file names in a directory can be recovered by the command `load_directory`. The result is a vector of strings, which corresponds to the name of a file or a subdirectory.

To check if a file or a directory exists, one can use the predicate `readable?`.

```
Mmx] include "example.mmx"
```

```
Mmx] save ("tmp.txt", "A string is stored in the file \n in two lines");
```

```
Mmx] load "tmp.txt"
```

"A string is stored in the file \n in two lines": String

```
Mmx] load_directory "."
```

["..", ".svn", ".", "emacs_mode.en.tm", "index.en.tm"]: Vector(Generic)

1.4.13. Environment

Several functions are available to interact with the environment. To get the value of a variable defined in the environment, one can use `get_env`.

To run a command in this environement, one can use the function `system`:.

```
Mmx] get_env "PWD"
```

/user/mourrain/home/mmx/mmxlight/doc/texmacs

```
Mmx] set_env ("DISPLAY", "arthur:0")
```

0

```
Mmx] system "ls"
```

```
emacs_mode.en.tm
how_to.en.tm
index.en.tm
installation.en.tm
quick_start.en.tm
shell.en.tm
shell_tutorial.en.tm
syntax.en.tm
tmp.txt
toto.txt
```
0: Int

1.5. Classes

1.5.1. Class definition

User classes can be defined within Mmx-light, as shown in the following example:

```
Mmx] type_mode? := true
```

true: Boolean

```
Mmx] class Color == {
       mutable { r: Double; g: Double; b: Double; }
       constructor grey (x: Double) == {
         r == x; g == x; b == x; }
       constructor rgb (r2: Double, g2: Double, b2: Double) == {
         r == r2; g == g2; b == b2; }
     }
```

```
Mmx] rgb (1, 0, 0)
```

rgb$(1,0,0)$: Color

```
Mmx] flatten (c: Color): Syntactic ==
       syntactic ('rgb (as_generic flatten c.r,
                        as_generic flatten c.g,
                        as_generic flatten c.b));
```

```
Mmx] rgb (1, 0, 0)
```

rgb$(1,0,0)$: Color

```
Mmx] mix (c1: Color, c2: Color, a: Double): Color ==
       rgb (a * c1.r + (1-a) * c2.r,
            a * c1.g + (1-a) * c2.g,
            a * c1.b + (1-a) * c2.b);
```

```
Mmx] mix (rgb (1, 0, 0), grey (0.5), 0.5)
```

rgb$(0.75,0.25,0.25)$: Color

1.5.2. Type conversions

The operator `:>` can be used to convert a given type into another, provided that the corresponding function `downgrade` is defined. We show here an example of an explicit conversion from a class `Alpha_color` to the class `Color`:

```
Mmx] class Alpha_color == {
       mutable { c: Color; a: Double; }
       constructor alpha_color (c2: Color, a2: Double) == { c == c2; a ==
     a2; }
     };
```

```
Mmx] alpha_color (rgb(0,0,1), 0.5)
```

object([rgb$(0,0,1)$, 0.5], Alpha_color): Alpha_color

```
Mmx] downgrade (ac: Alpha_color): Color == mix (ac.c, rgb(1,1,1), ac.a);
```

```
Mmx] alpha_color (rgb(0,0,0), 0.6) :> Color
```

rgb$(0.4,0.4,0.4)$: Color

In order to define implicit conversions from a type into another, one can use the function `updgrade`. We complete the example with a convert from `Double` to `Color`:

```
Mmx] upgrade (x: Double): Color == grey x;
```

```
Mmx] mix (1.0, rgb (0, 1, 0), 0.2)
```

$\mathrm{rgb}(0.2, 1, 0.2)$: Color

```
Mmx] mix (0.8, 0.2, 0.4)
```

$\mathrm{rgb}(0.44, 0.44, 0.44)$: Color

```
Mmx] postfix .greyed (c: Color): Color == (c.r + c.g + c.b) / 3;
```

```
Mmx] rgb (1, 0, 0).greyed
```

$\mathrm{rgb}(0.333333333333, 0.333333333333, 0.333333333333)$: Color

1.6. Coding style

- Avoid lines with more than 80 characters. For long instructions, you should better rewrite

  ```
  D := divide((k*(k-i)*(SR[k][i])*(SR[k][k])
              -(i+1)*(SR[k][k-1])*(SR[k][i+1])),delt[k]);
  ```

  into

  ```
  left  == k * (k-i) * SR[k][i] * SR[k][k];
  right == (i+1) * SR[k][k-1] * SR[k][i+1];
  D := divide (left - right, delt[k]);
  ```

- Add spaces around the main operators to ease readability. For instance the expression

  ```
  a:=b[0]*b[2]/c[0]-a[3]*b[0]^2*c[5]+a[6]*a[7]-b[8]^10;
  ```

  should better be written

  ```
  a := b[0]*b[2]/c[0] - a[3]*b[0]^2*c[5] + a[6]*a[7] - b[8]^10;
  ```

- Align operators when useful for readability. For instance write

  ```
  blah == 2;
  boe  == 3;
  ```

- Function and macro names should be meaningful and written in English.

1.7. An Emacs mode for Mathemagix files

To edit Mathemagix code with Emacs, a `mmx` mode can be used. It is available in `mmxlight/emacs/mmx-mode.el` and can be installed as follows:

```
cp mmx/mmxlight/emacs/mmx-mode.el $HOME/.emacs.d
```

The following can be added in the file `.emacs`:

```
(setq load-path
  (append load-path (list (expand-file-name "~/.emacs.d"))))
(setq auto-mode-alist
     (append auto-mode-alist '(("\.mmx" . mmx-mode))))
(autoload 'mmx-mode
     "mmx-mode.el" "Major mode for editing Mathemagix files" t)
```

Once this is done, the `mmx-mode` will be loaded automatically for files with suffix `.mmx`.

Chapter 2
Basix

2.1. Introduction to the Basix package

The `basix` package contains the usual data types, the language environments, the evaluators and the abstract language on which the Mathemagix language is actually based.

Installation notes and the most recent version of this package are to be found from the Mathemagix Internet home page available from www.mathemagix.org.

2.1.1. A simple example of use

Here follows a simple example named `example.cpp`:

```
#include "basix/string.hpp"
#include "basix/list.hpp"

using namespace mmx;

int main () {
  list<string> x ("a", "b", "c");
  list<int> y (1, 2, 3);

  mmout << "x = " << x << "\n";
  mmout << "y = " << y << "\n";
}
```

The following command can be used to compile file:

```
g++ ‘basix-config --cppflags --libs‘ example.cpp -o example
```

As a result you can now run `example` and obtain:

```
["a", "b", "c"]
[1, 2, 3]
```

Hereabove the command `basix-config --cppflags --libs` produces all the necessary include and library paths. More options are described though `basix-config --help`:

```
Usage: basix-config [OPTION]

Available values for OPTION include:

--cppflags    compiler preprocessor flags
--exec-prefix basix install prefix for executables
--help        display this help and exit
--libs        library linking information
--prefix      basix install prefix
--version     output version information
```

2.2. Basic data types

Basix defines a type `nat`, meaning “natural integers”, that corresponds to `unsigned int` within most architecture. Yet no assumption on the size of `nat` is made for the sake of portability.

The standard ouput stream is `mmout`, the standard error stream is `mmerr`, and the standard input is `mmin`.

2.2.1. Strings

Strings are implemented in the class `string` provided by `string.hpp`.

```
string s = "Hello";
mmout << s << "\n";
```

Memory allocation is done in a way so that you can write the following piece of code with essentially no lack of efficiency:

```
// Remove all \ in s
string foo (const string& s) {
  nat i, n= N(s);
  string r;
  for (i=0; i < n; i++)
    if (s[i] != '\') r << s[i];
  return r;
}
```

2.2.2. Lists

Lists are implemented in the template class `list` provided by `list.hpp`. Usual operations are available from `list_sort.hpp`.

```
list<int> x (1, 2, 3);
list<int> y (seq (1, 5));
mmout << "x n= " << x << "\n";
mmout << "y = " << y << "\n";
mmout << "x*y = " << (x*y) << "\n";
mmout << "x==y = " << (x==y) << "\n";
mmout << "x!=y = " << (x!=y) << "\n";
mmout << "x... = " << iterate (x) << "\n";
```

2.2.3. Balanced tree

Balanced tree are implemented in the class `chain` provided by `chain.hpp`. Usage is similar to the one of the lists.

2.2.4. Heaps

Ordered heaps are provided by `heap.hpp`.

2.2.5. Pairs, triples, tuples

Pairs are provided by `pair`, and triples in `triple.hpp`. Tuples are provided by `tuple.hpp`.

2.2.6. Vectors

Arrays are provides by `vector.hpp`.

```
#define C double
vector<C> u = vec (C (5), C (7), C (9));
vector<C> v = vec (C (2), C (3), C (5));
mmout << "u = " << u << "\n";
mmout << "v = " << v << "\n";
v *= u;
```

For efficiency reasons, vectors are indeed parametrized by an abstration layer, that is defined in `vector_naive.hpp`. For instance vectors of `int` with fixed size 10 can be obtained as follows:

```
vector<C, vector_fixed<vector_naive, fixed_value<nat,10> > x (3);
```

2.2.7. Tables

Hash tables are defined in `table.hpp`.

```
pair<int,int> x1 (1, 2), x2 (7, 7);
table<int,int> t (seq (x1, x2));
table<int,int> u (0);
u[1]= 8; u[2]= 7; u[3]= 6; u[4]= 5;
mmout << "t = " << t << "\n";
mmout << "u = " << u << "\n";
reset (u, 3);
```

2.2.8. Iterators

Iterators are defined in `iterator.hpp`.

```
iterator<int> it1;
mmout << "it1 = " << it1 << "\n";
iterator<int> it2 = seq (1, 2, 3);
mmout << "it2 = " << it2 << "\n";
iterator<int> it3 = range_iterator<int> (0, 100);

list<string> l ("a", "b");
iterator<string> it;
for (it = iterate (l); busy (it); ++it)
  mmout << *it << "\n";
```

All non-atomic structures have iterators.

2.3. Basix' Internals

2.3.1. Indirect Data Structures

All Mathemagix classes are declared through the INDIRECT_PROTO and INDIRECT_IMPL macros from a minimal base class representation.

See typical examples of use in `string.hpp` or `list.hpp`.

2.4. Global settings

2.4.1. Memory managment

Memory management is defined in `fast_new.hpp`

2.4.2. Error Management

The macro ERROR must be used to raise a global error.

The macros ASSERT and VERIFY must be used for debugging purposes.

Errors are handled by the "throw/catch" mecanism if exceptions have been enabled at the configuration time.

The function fatal_error quit the execution brutaly.

2.5. Creating TeXmacs documents

TeXmacs documents can be created from the C++ level by using basix/document.hpp.

From the Mathemagix language the interface is provided in `basix/texmacs.mmx`, and can be used as follows:

```
Mmx] include "basix/texmacs.mmx"
```

```
Mmx] $color ("blue", $text "Hello")
```

Hello

```
Mmx] $tm ('block, $tm ('tformat, $tm ('table,
        $tm ('row, $tm ('cell, $text ("x")), $tm ('cell, $text ("y"))),
        $tm ('row, $tm ('cell, $math (1)), $tm ('cell, $math (2))))))
```

x	y
1	2

Chapter 3
Numerix

3.1. Integer and rational numbers

3.1.1. Signed integer numbers

Integer numbers of arbitrarily large size are available through the class `integer` defined in `integer.hpp`. This class is a simple wrapper to the class `mpz_t` of the GMP library.

```
#include<numerix/integer.hpp>
using namespace mmx;
void main () {
  integer a (2), b (3);
  mmout << a * b << "\n";
}
```

3.1.2. Rational numbers

Rational numbers of arbitrarily large size are available through the class `rational` defined in `rational.hpp`. This class is a simple wrapper to the class `mpq_t` of the GMP library.

```
#include<numerix/rational.hpp>
using namespace mmx;
void main () {
  rational a (2), b (3);
  mmout << a / b << "\n";
}
```

3.1.3. Mathemagix interface

```
Mmx] use "numerix";
     type_mode? := true;
```

```
Mmx] 40!
```

815915283247897734345611269596115894272000000000 : Integer

```
Mmx] probable_next_prime 100
```

101 : Integer

```
Mmx] probable_prime? 9973
```

true : Boolean

```
Mmx] gcd (10, 35)
```

5: Integer

```
Mmx] 10 / 23
```

$\frac{10}{23}$: Rational

3.2. Modular numbers

This document describes the use and implementation of the modulars within the `numerix` package.

3.2.1. General modulars and moduli

Moduli are implemented in `modulus.hpp`. An object of type `modulus` is declared as follows:

```
modulus<C,V> p;
```

`V` is an implementation variant. The default variant, named `modulus_naive`, is implemented in `modulus_naive.hpp` and supports a type `C` with an Euclidean division. More variants are described below. Modulars are implemented in `modular.hpp`. An object of type `modular` can be used as follows:

```
modular<modulus<C, V>, W> x, y, z;
modular<modulus<C, V>, W>::set_modulus (p);
x = y * z;
```

The variant `W` is used to define the storage of the modulus. Default storage is `modular_local` that allows the current modulus to be changed at any time. The default modulus is 0. For the `integer` type the default variant is set to `modulus_integer_naive`, so that modular integers can be used as follows:

```
#include<numerix/integer.hpp>
#include<numerix/modular.hpp>
#include<numerix/modulus_integer_naive.hpp>

typedef modulus<integer> M;
M p (9973);
modular<M>::set_modulus (p);
modular<M> a (1), b (3);
c = a * b;
cout << c << "\n";
```

If the modulus is known to be fixed at compilation time then the following declaration is preferable for the sake of efficiency:

```
fixed_value <int, 3> w;
modulus<integer> p (w);
```

If the modulus is not intended to be changed and known at compilation time then the variant `modular_fixed` can be used.

3.2.2. Moduli for hardware integers

From now on a div p and a mod p represent the quotient and the remainder in the long division of a by p respectively. Throughout this section `C` denotes a genuine integer C type. We write n for the bit-size of `C`, and p for a modulus that fits in `C`.

3.2.2.1. Naive implementation

The default variant, `modulus_int_naive<m>`, is defined in `modular_int.hpp`. Each modular product performs one product and one integer division. The parameter `m` is the maximum bit-size of the modulus allowed by the variant. This parameter can be set to n. Best performances are obtained with `m`$=n-1$. Note that we necessarily have `m`$\leqslant n-1$ if `C` is a signed integer type. By convention the modulus 0 actually means 2^m.

3.2.2.2. Moduli with pre-computed inverse

In the variant `modulus_int_preinverse`<m> a suitable inverse of the modulus is pre-computed and stored as an element of type `C`, this yields faster computations when doing several operations with the same modulus. The behavior of the parameter `m` is as in the preceding naive variant. The best performances are attained for when `m`$\leqslant n-2$.

Within the implementation of this variant the following auxiliary quantities are needed:

- a non-negative integer r is defined by $2^{r-1} < p \leq 2^r$,
- an integer s such that $0 \leq s \leq r-1$,
- an integer t such that $t \geq r$ and $s+t-(r-1) \leq n$,
- an integer $q = 2^{s+t}$ div p, that represents the numerical inverse of p to precision $s+t-k$.

By convention we let $r=n$ if p is zero. Note that $q<2^n$, hence fits in `C`.

Modular product.

Let a be an integer such that $0 \leq a < p^2$. The remainder $a \operatorname{mod} p$ can be obtained as follows:

1. Decompose a into a_0 and a_1 such that $a = a_1 2^s + a_0$, with $a_0 < 2^s$, and compute $b = a_1 q \operatorname{div} 2^t$. From $a < p^2$ it follows that $a_1 q < p^2\, q / 2^s \leq p 2^t$. So $a_1 q$ has size at most $r+t$, and we have $b<p$.

2. Compute $c = a - bp$. From the definition of q we have:
$$0 \leq \frac{2^{s+t}}{p} - q < 1 \Longrightarrow 0 \leq a - a_1 q p / 2^t < a_1 p / 2^t + a_0.$$
It follows that $0 \leq c < p + a_1 p / 2^t + a_0$.

3. Let $h = 2^{2r} / 2^{s+t} + 2^s / 2^{r-1}$. From $a_0 < 2^s$ we deduce that $a_1 p / 2^t + a_0 \leq hp$. It thus suffices to substract at most h times p from a to obtain $a \operatorname{mod} p$.

In general we can always take $t = r$ and $s = r - 1$, so that $h = 3$. For when $r + 1 \leq n$, it is better to take $t = r + 1$ and $s = r - 1$ so that $h \leq 2$. Finally, for when $r + 2 \leq n$ one can take $t = r + 3$ and $s = r - 2$ so that $h \leq 1$. These settings are summarized in the following table:

	$r+2\leq n$	$r+1\leq n$	$r\leq n$
s	$r-2$	$r-1$	$r-1$
t	$r+3$	$r+1$	r
h	1	2	3

In these three cases remark that the integer $a_1 q$ has size at most $2n$.

It is clear that the case $r + 2 \leq n$ leads to the fastest algorithm since step 3 reduces to one substraction/comparison that can be implemented as $\min(c, c - p)$ within type C alone. A special attention must be drawn to when $r = 1$, that corresponds to $p = 2$: we must suppose that $s = r - 2 = -1$ is indeed computed within an unsigned int type, hence is sufficiently large to ensure $a_1 = 0$ and $c = a$, hence the correctness of the algorithm.

Numerical inverse.

Let us now turn to the computation of the numerical inverse q of p. The strategy presented here consists in performing one long division in C followed by one Newton iteration. Let $w = 2^{r-1}/p$ be the numerical inverse of p, normalized so that $1/2 \leq w < 1$ holds. Let x_0 denote the initial approximation, and x_1 be its Newton iterate $x_1 = 2x_0 - p x_0^2 / 2^{r-1}$. If $0 \leq w - x_0 < 2^{-s_0}$ for some positive integer s_0 then it is classical that $0 \leq w - x_1 < 2^{-s_1}$ with $s_1 = 2s_0 - 1$.

Under the assumption that $s_0 \leq n/2$, we can compute a suitable x_0 as the floor part of $2^{s_0} w$ by the only use of operations within base type C. In other words we calculate $q_0 = 2^{s_0 + r - 1} \operatorname{div} p = 2^{s_0} x_0$ as follows:

1. If $r \leq s_0$ then q_0 is obtained by means of one division in C.

2. Otherwise $r > s_0$. We decompose p into $p = p_1 2^{r - s_0} + p_0$, with $p_0 < 2^{r - s_0}$. Note that $2^{s_0 - 1} \leq p_1 \leq 2^{s_0}$. Within C perform the long division $2^{2s_0 - 1} = a p_1 + b$. By multiplying both side of the latter equality by $2^{r - s_0}$ we obtain that $2^{s_0 + r - 1} = ap - ap_0 + b 2^{r - s_0}$. From a and b, we easily deduce q_0 since $b 2^{r - s_0} < p$ and $a p_0 < 2^{2s_0 - 1} 2^{r - s_0} / 2^{s_0 - 1} = 2^r$.

The computation of $q_1 = 2^{s_1 + r - 1} \operatorname{div} p$, that is the floor part of $2^{s_1} w$, then continues as follows:

1. From the definition of x_1, one can write: $x_1 2_1^{s_1} = q_0 2^{s_0} - p q_0^2 / 2^r$. We thus compute $q_0 2^{s_0}$ and the ceil part of $p q_0^2 / 2^r$ in order to deduce the floor part c of $x_1 2^{s_1}$. Note that c fits C.

2. From the above inequalities it follows that $0 \leq 2^{s_1} w - c < 2$. We thus obtain $d = 2^{s_1 + r - 1} - cp < 2\text{p}$. If $d \geq p$ then return $q_1 = c + 1$ else return $q_1 = c$.

Finally, q is deduced in this way:

1. Set s_0 as $(s + t - (r - 1)) \operatorname{div} 2$, and compute $q_1 = 2^{s_1 + r - 1} \operatorname{div} p$ as just described.

2. Note that $1 \leq s + t - (r - 1) - s_1 \leq 2$. If $s_1 + 1 = s + t - (r - 1)$ then $q = 2^{s_1 + r} \operatorname{div} p$ otherwise $q = 2^{s_1 + r + 1} \operatorname{div} p$.

Modular product by a constant.

Let b be a modulo p integer that is to be involved in several modular products. Then one can compute $\beta = 2^r b \operatorname{div} p < 2^r$ just once and obtain a speed-up within the products by means of the follows algorithm:

1. Compute $\alpha = a\beta \operatorname{div} 2^r$. It follows that $0 \leq ab - \alpha p < 2p$.

2. Compute $c = ab - \alpha p$. If $c \geq p$ then c=c+1. Return c.

The computation of β can be done as follows from q:

1. Compute $\gamma = qb \operatorname{div} 2^{t+s-r} < 2^r$. Note that $2^r b/p - 2^r b/2^{t+s} - 1 < \gamma \leq 2^r b/p$.

2. Since $0 \leq 2^r b - \gamma p < gp$, with $g = 2$ if $r + 1 \leq n$ and $g = 3$ otherwise, deduce $2^r b \operatorname{div} p$ from γ.

3.2.3. Use from the Mmx-light interpreter

Modular integers are glued to the Mmx-light interpreter. The usual arithmetic operations are available.

```
Mmx] use "numerix";
```

```
Mmx] type_mode? := true;
```

```
Mmx] p: Modulus Integer == 7
```

7 : Modulus(Integer)

```
Mmx] a == 11 mod p
```

4 : Modular(Integer)

```
Mmx] a ^ 101
```

2 : Modular(Integer)

```
Mmx] q == modulus (7 :> Int)
```

7 : Modulus(Int)

```
Mmx] (11 mod q) ^ 100
```

4 : Modular(Int)

3.2.4. Further remarks

- On some processor architectures code to manipulate `short int` can be larger and slower than corresponding code which deals with `int`. This is particularly true on the Intel x86 processors executing 32 bit code. Every instruction which references a `short int` in such code is one byte larger and usually takes extra processor time to execute.

- Do not use `char` but `signed char` or `unsigned char` for the sake of portability.

3.3. Floating point numbers

3.3.1. Floatings

3.3.1.1. C++ interface

Floating point numbers of arbitrarily large size are available through the class `floating` defined in `floating.hpp`. This class is a simple wrapper to the class `mpfr_t` of the MPFR library.

```
#include<numerix/floating.hpp>
using namespace mmx;
void main () {
  floating<> a (2), b (3);
  mmout << a * b << "\n";
}
```

The precision can be set thanks to the global variable `mmx_bit_precision`.

Lower and upper certified approximates can be obtained as follows:

```
typedef rounding_helper<floating<> >::UV Up;
typedef rounding_helper<floating<> >::DV Down;
mmout << Up::sqrt (floating<> (2)) << "\n";
mmout << Down::sqrt (floating<> (2)) << "\n";
```

3.3.1.2. Mathemagix interface

The above floating type is glued to MATHEMAGIX as `Floating`. Default bit-precision can be modified by setting the variable `bit_precision`.

```
Mmx] use "numerix";
     type_mode? := true;
```

```
Mmx] a: Floating == 1.0
```

$1.00000000000000000000:$ Floating

```
Mmx] exp a
```

$2.71828182845904523543:$ Floating

```
Mmx] sin a
```

$0.841470984807896506665:$ Floating

```
Mmx] 1 / 0.0
```

$\infty:$ Floating

```
Mmx] bit_precision := 128
```

$128:$ Int

```
Mmx] exp 1.0
```

$2.7182818284590452353602874713526624977 59$: Floating

3.3.2. Intervals

3.3.2.1. C++ interface

Interval are implemented within the class `interval` defined in `interval.hpp`.

```
#include<numerix/floating.hpp>
#include<numerix/interval.hpp>
using namespace mmx;
typedef interval<floating<> > Interval;
void main () {
  Interval a (2), b (3.0, 3.1);
  mmout << a * b << "\n";
}
```

3.3.2.2. MATHEMAGIX interface

The above interval type is glued to MATHEMAGIX as `Interval`. Classical scientific notation are used for pretty printing intervals.

```
Mmx] a == interval (0.999, 1)
```

1.00: Interval(Floating)

```
Mmx] exp a
```

2.72: Interval(Floating)

```
Mmx] radius a
```

$5.00000000000000118776e-4$: Floating

```
Mmx] lower a
```

0.99899999999999999979: Floating

```
Mmx] upper a
```

1.00000000000000000000: Floating

3.3.3. Complex numbers

3.3.3.1. C++ interface

Complex numbers are available through the class `complex` defined in `complex.hpp`. Over `double` one must include `numerix/complex_double.hpp`.

```
#include<numerix/complex_double.hpp>
using namespace mmx;
typedef complex<double> Complex;
void main () {
  Complex a (2), b (3.0, 0.1);
  mmout << a * b << "\n";
}
```

3.3.3.2. MATHEMAGIX interface

The above complex type is glued to MATHEMAGIX as `Complex`.

```
Mmx] I == complex (0, 1)
```

i: Complex(Rational)

```
Mmx] I * I
```

-1: Complex(Rational)

```
Mmx] 1 / (1 + I)
```

$\frac{1}{2} - \frac{1}{2}\mathrm{i}$: Complex(Rational)

```
Mmx] I == complex (0 :> Floating, 1 :> Floating)
```

$1.00000000000000000000\,\mathrm{i}$: Complex(Floating)

```
Mmx] exp I
```

$0.54030230586813971740 + 0.84147098480789650665\,\mathrm{i}$: Complex(Floating)

3.3.4. Balls

3.3.4.1. C++ interface

Balls are implemented in the class `ball` defined in `ball.hpp`.

```
#include<numerix/ball.hpp>
using namespace mmx;
typedef ball<floating<> > Ball;
void main () {
  Ball a (2), b (3.0, 0.1);
  mmout << a * b << "\n";
}
```

3.3.4.2. MATHEMAGIX interface

The above ball type is glued to MATHEMAGIX as `Ball`.

```
Mmx] a : Ball (Floating, Floating) == ball (3.0, 0.1)
```

$3e0$

```
Mmx] [ center a, radius a ]
```

[3.00000000000000000000, 0.100000000000000000333]

```
Mmx] M (n) ==
       if n = 0 then ball 2.0
       else if n = 1 then ball (-4.0)
       else 111 - 1130 / M (n-1) + 3000 / (M(n-1) * M(n-2));
```

```
Mmx] for n in 1 .. 15 do
       mmout << n << " " << M n << lf;
```

```
1  −4.00000000000000000000
2  18.500000000000000
3  9.37837837837838
4  7.801152737752
5  7.1544144810
6  6.8067847369
7  6.59263277
8  6.449466
9  6.34845
10  6.274
11  6.22
12  6e0
13  0e1
14  0e3
```

3.4. Tangent numbers

First order jet spaces, namely $\mathbb{K}[x]/(x^2)$, are implemented in the class `tangent` defined in `tangent.hpp`. They are compatible with all the floating types.

```
#include<numerix/tangent.hpp>
using namespace mmx;
void main () {
  tangent<double> a (2.0, 3.0);
  mmout << "a= " << base (a)
        << " + " << slope (a) << " * x + O(x^2)\n";
}
```

Chapter 4
Algebramix

4.1. Mechanism for implementation variants

Several important template classes, such as `vector`, `matrix`, `polynomial` and `series` admit a last template parameter `V`, called the *implementation variant* (or the traits class, in C++ terminology). For most Mathemagix template classes, the class itself determines the internal representation of instances. For instance, the `polynomial` class corresponds to dense univariate polynomials, determined by a low level vector of coefficients. The implementation variant is used to specify the actual algorithms which are used for common operations. For instance, the `polynomial_naive` variant uses the naive $O(n^2)$ algorithm for polynomial multiplication, whereas Karatsuba multiplication is used by the variant `polynomial_karatsuba<polynomial_naive>`.

As is suggested by the variant `polynomial_karatsuba<polynomial_naive>`, new variants can be built on top of other variants and existing variants can be customized. Typically, an implementation of dense polynomials provides several features:

- Vectorial routines on polynomials, such as addition.
- Other linear-time routines, such as differentiation.
- Multiplication.
- Division.
- Greatest common divisor.
- Several higher level routines, such as subresultants or multi-point evaluation.

Variants and features are essentially empty classes, which are mainly used as template parameters. Mathemagix provides a standard `implementation` helper class, which allows to associate a concrete implementation to a feature and variant.

For instance, polynomial multiplication corresponds to the feature `polynomial_multiply`. The main multiplication routine for dense polynomials is a static member function

```
implementation<polynomial_multiply,polynomial_naive>::mul
```

of the implementation helper class. This routine directly operates on vectors in memory:

```
template<typename C, typename K> static void
mul (C* dest, const C* s1, const K* s2, nat n1, nat n2) {
  if (n1+n2>0)
    Pol::clear (dest, n1+n2-1);
  while (n2 != 0) {
    Pol::mul_add (dest, s1, *s2, n1);
    dest++; s2++; n2--;
  }
}
```

Here Pol is an alias for implementation<polynomial_linear,polynomial_naive>, hence the implementation of naive multiplication relies on the implementation of naive vectorial and linear-time algorithms on polynomials.

From the high level point of view, the implementation of polynomial multiplication for the type polynomial<C,V> relies on the low level multiplication algorithm determined by the variant V:

```
template<typename C, typename V> polynomial<C,V>
operator * (const polynomial<C,V>& P1, const polynomial<C,V>& P2) {
  typedef implementation<polynomial_multiply,V> Pol;
  nat n1= N(P1), n2= N(P2);
  if (n1 == 0 || n2 == 0) return polynomial<C,V> ();
  nat l= aligned_size<C,V> (n1+n2-1);
  C* r= mmx_new<C> (l);
  Pol::mul (r, seg (P1), seg (P2), n1, n2);
  return polynomial<C,V> (r, n1+n2-1, l);
}
```

A default variant is usually defined in terms of the remaining template parameters. In the case of polynomials, this is accomplished by means of the polynomial_variant_helper template: for any coefficient type C, polynomial_variant_helper<C>::PV yields the default variant for C.

In fact, the template implementation takes three arguments: the feature, the polynomial variant and the variant for the actual implementation. This allows for the customization of a specific feature. More precisely, assume that we are given a feature F and a variant V and that we want to define a new variant customized<V> with a different implementation of the feature F, but unaltered implementations of all other features. This is achieved using a code of the type

```
template<typename V, typename W>
struct implementation<F,V,customized<W> >:
  public implementation<F,V,W>
{
  typedef implementation<F,V,W> Fallback;
  ...
};
```

For the actual implementation of the customized feature, one may rely on the original implementation Fallback specified by the variants V and W. Here we notice that implementation<F,V> is equivalent to implementation<F,V,V>.

We recall that the variant mechanism is only used in order to implement different algorithms for a similar functionality, while presupposing a given internal representation. Some higher level functions on polynomials may still be correct for polynomials with alternative internal representations. In that case, a new template class should be defined for each alternative representation, such as sparse_polynomial. This class should be implemented with care, so as to match the same signature as the usual dense class polynomial. For instance, if we have an operation

```
template<typename C, typename V> polynomial<C,V>
foo (const polynomial<C,V>& p) { ... }
```

which is not specific to dense polynomials, then we should have its sparse counterpart with the same name

```
template<typename C, typename V> sparse_polynomial<C,V>
foo (const sparse_polynomial<C,V>& p) { ... }
```

The high level functionality `bar` can then be implemented by using the class of polynomials itself as a template parameter:

```
template<typename Pol> Pol
bar (const Pol& p) {
  return foo (foo (p));
}
```

4.2. Optimized variants for vectors

4.2.1. Unrolled vectorial operations

In `vector_unrolled.hpp` is implemented a variant for vectorial operations with loop unrolling. Precisely vectorial operations on elements of type `vector<C,vector_unrolled<n,V> >` are unrolled by blocks of size n. For example the piece of code

```
#define vector_unrolled<4, vector_naive> V
vector<signed char, V> v1=..., v2=...;
vector<signed char, V> w= v1 * v2;
```

actually computes the entrywise product of `v1` and `v2` as follows:

```
for (nat i= 0; i <...; i += 4) {
  w1[i  ]= v1[i  ] * v2[i];
  w1[i+1]= v1[i+1] * v2[i+1];
  w1[i+2]= v1[i+2] * v2[i+2];
  w1[i+3]= v1[i+3] * v2[i+3];
}
```

4.2.2. SIMD support

Several architectures support special instructions ot type called SIMD (Single Instruction, Multiple Data) for performing operations on “hardware vectors”. This imposes the memory position of the vectors to be aligned. When allocating memory space for a vector of type `vector<C,V>` of size `n` one thus needs to proceed as follows:

```
nat l= aligned_size<C,V> (n);
C* buf= mmx_new<C> (l);
// fill buf here...
vector<C,V> v (buf, n, l);
```

Note that the memory will be freed once `v` destructed.

In vector_simd.hpp, the variant vector_simd allows one to benefit of the SIMD functionalities. For instance a vector of type vector<unsigned char,vector_simd<8,4> > unrolls blocks of 8 simd sub-vectors, the rest being unrolled by blocks of size 4 with classical instructions.

A default variant of type C is available through the macro Vector_simd_variant(C).

Currently only SSE2 and SSE3 are partially supported – details are to be found in vector_sse.hpp.

4.3. Univariate polynomials

4.3.1. Naive algorithms

Univariate polynomials over C are implemented in the class polynomial<C,V>, defined in polynomial.hpp. The default variant polynomial_naive assumes that C is a field, not necessarily commutative. This provides implementations of the naive algorithms. For instance the product, the division, and the gcd have quadratic costs.

```
#include <numerix/rational.hpp>
#include <algebramix/polynomial.hpp>
...
polynomial<rational> x (C(1), 1); // defines the monomial x
polynomial<rational> p= square (x) + 3 * x + 1;
```

Remark that you cannot use the infix operation ^ for powering, for it is usually reserved to bitwise operations. Instead you can use binary powering with binpow (a, n). For the sake of efficiency, the monomial x^n must be constructed as polynomial<C,V> (C(1), n).

For polynomials over a ring you must add the variant poynomial_ring_naive<V> defined in polynomial_ring_naive.hpp, that essentially contains implementations for the subresultants due to Ducos.

4.3.2. Divide and conquer algorithms

All classical generic divide and conquer algorithms over any field are implemented in polynomial_dicho.hpp. Therein is defined the variant polynomial_dicho<V> that implements the Karatsuba product, division with Newton's inversion algorithm, fast Euclidean sequence, the half-gcd algorithm, Padé approximants, fast multipoint evaluation and interpolation, fast multimod and Chinese remaindering. If you want these algorithms to be applied in $\mathbb{Q}[x]$ you can modify the above piece of code into:

```
#include <algebramix/polynomial.hpp>
#include <algebramix/polynomial_dicho.hpp>
#define polynomial_dicho<polynomial_naive> V
...
polynomial<rational,V> x (C(1), 1); // defines the monomial x
polynomial<rational,V> p= square (x) + 3 * x + 1;
```

For polynomials over a ring you must add the variant polynomial_ring_dicho<V> defined in polynomial_ring_dicho.hpp, that essentially contains the half-subresultant alsgorithm.

4.3.3. Asymptotically fast algorithms

Schönhage & Strassen's, and Cantor & Kaltofen's fast products are implemented in `polynomial_schonhage`. It supports any ring with unity. This provides the `generic` a default variant.

4.3.4. Special types of coefficients

4.3.4.1. Kronecker's subtitution

The Kronecker susbstitution is available for polynomials over polynomials, hardware or long integers, and also with modular coefficients. One must include one of the corresponding files:

- `polynomial_polynomial.hpp`,
- `polynomial_integer.hpp`, `polynomial_modular_integer.hpp`
- `polynomial_int.hpp`, `polynomial_modular_int.hpp`

4.3.4.2. Modulars

If the coefficient ring are modular numbers then a special product is available from `polynomial_modular.hpp`: it first computes the product of the polynomials made of the preimages of the coefficients and reduces the coefficients at the end.

4.3.4.3. Complex numbers

For polynomial over complex numbers you must use `polynomial_complex.hpp`.

4.3.4.4. Quotient fields

If coefficients are in a quotient field of type `quotient`, then one should include `polynomial_quotient.hpp`. A special variant is directly available for rational numbers in `polynomial_rational.hpp`.

4.3.5. Quotient ring

Operations in $\mathbb{K}[x]/p(x)$ are available through the class `modular`. Special types of moduli for polynomials is defined in `modular_polynomial.hpp`.

```
#include "numerix/rational.hpp"
#include "algebramix/modular_polynomial.hpp"
#define C rational
#define Polynomial polynomial<C>
#define Modular modular<modulus<Polynomial> >
...
Polynomial x (C(1), 1);
Polynomial p= square (x) - 2;
Modular::set_modulus (p);
mmout << binpow (Modular (x), 1000) << "\n";
```

4.3.6. MATHEMAGIX interface

```
Mmx] use "algebramix";
     type_mode? := true;
```

```
Mmx] x == polynomial (0, 1)
```

x: Polynomial(Integer)

```
Mmx] (1 + x)^5
```

$x^5+5x^4+10x^3+10x^2+5x+1$: Polynomial(Integer)

```
Mmx] gcd (1 + 2*x + x^2, 1 - x^2)
```

$x+1$: Polynomial(Rational)

```
Mmx] discriminant (1 + x + x^2)
```

3: Rational

```
Mmx] (1 + x)^5 mod 5
```

x^5+1: Polynomial(Modular(Integer))

4.4. POWER SERIES

4.4.1. Introduction

Power series are implemented within the class `series` from `series.hpp`.

```
#include <numerix/rational.hpp>
#include <algebramix/series.hpp>

series<rational> z (rational (1), 1); // variable z
series<rational> f = 1 / (1 - z);
mmout << f[10] << "\n";
```

Series are implemented in a lazy way, which means that the precision has not to be specified in advance. Computations are actually done when a coefficient is actually needed.

The precision used for printing the power series can be set to `n` in the following way:

```
series<C,V>::set_output_order (n);
```

Instead of printing an approximation of the series, it is possible to display the formula it has been built from, as follows:

```
series<C,V>::set_formula_output (true);
```

Withing equality test the precision to be taken into account can be set to `n` via:

```
series<C,V>::set_cancel_order (n);
```

The name of the variable to be printed can be set to `z` with the following command:

```
series<C,V>::set_variable_name ("z");
```

4.4.2. Naive algorithms

Naive algorithms for power series are implemented in `series_naive.hpp`. For instance the product implemented therein has a quadratic cost.

Over numerical types, elementary functions are available from `series_elementary.hpp`.

4.4.3. Relaxed algorithm

The relaxed product for power series is implemented in `series_relaxed.hpp`.

```
#include <numerix/rational.hpp>
#include <algebramix/series.hpp>
#include <algebramix/series_relaxed.hpp>

#define V series_relaxed<series_naive>
series<rational,V> z (rational (1), 1); // variable z
series<rational,V> f = 1 / (1 - z);
mmout << z[10] << "\n";
```

The relaxed product has a softly linear cost.

4.4.4. Recursive series

A series f is said to be recursive if it satisfies an equation $f = \Phi(f)$ that allows to compute the n-th coefficient of f as the n-th coefficient of $\Phi(f)$ with the only knowledge of the $n-1$ first coefficients of f, whenever n is larger than an integer k called the *index*.

For example $f(z) = f(z)^2 + z\, f(z) + 1$, with $f_0 = 1$ can be implemented as follows:

```
template<typename C, typename V>
struct example_series_rep : recursive_series_rep<C,V>
  example_series_rep () {}
  syntactic expression (const syntactic& z) const {
    return apply ("example", z); }
  series<C,V> initialize () {
    series<C,V> f= this->me ();
    this->initial (0)= C(1);
    return square (f) + lshiftz (f, 1) + C(1); }

template<typename C, typename V>
example_series () {
  series_rep<C,V>* rep= new example_series_rep<C,V> ();
  return recursive (series<C,V> (rep)); }
```

4.4.5. Implicit series

Implicit series are implemented in `series_implicit.hpp`.

4.4.6. Vectorial operations

Vectorial and matricial auxilary functions are available from `series_vector.hpp` and `series_matrix.hpp` respectively. They are useful for computing recursive vectors of series.

4.4.7. MATHEMAGIX interface

```
Mmx] use "algebramix";
     type_mode? := true;
```

```
Mmx] z == series (0 :> Rational, 1 :> Rational)
```

$z+O(z^{10})$: Series(Rational)

```
Mmx] f == log (1 - z)
```

$-z-\frac{1}{2}z^2-\frac{1}{3}z^3-\frac{1}{4}z^4-\frac{1}{5}z^5-\frac{1}{6}z^6-\frac{1}{7}z^7-\frac{1}{8}z^8-\frac{1}{9}z^9+O(z^{10})$: Series(Rational)

```
Mmx] f[20]
```

$\frac{-1}{20}$: Rational

4.5. p-ADIC INTEGERS

4.5.1. Introduction

p-adic numbers are implemented as a variant of the series over modular integers: the representation is the same, only the operations differ. All the necessary definitions are gathered in `p_adic.hpp`.

```
#include <numerix/p_adic.hpp>

#define C modular<modulus<int, modulus_int_preinverse<14> > >
#define V series_carry_variant_helper<C>::SV
#define P_adic series<C,V>

C::set_modulus (9973);
P_adic p (C (1), 1); // parameter p
P_adic f = 1 / (1 - p);
mmout << f[10] << "\n";
```

4.5.2. Specific variants

Naive algorithms for p-adic numbers are implemented in `series_carry_naive.hpp`. The relaxed product is available from `series_carry_relaxed.hpp`.

4.5.3. Mathemagix interface

```
Mmx] p == modulus 7
```

7: Modulus(Integer)

```
Mmx] a == p_adic (1 mod p, 1 mod p)
```

$1 + p + O(p^{10})$: P_adic(Modular(Integer))

```
Mmx] b == separable_root (a, 2)
```

$1 + 4\,p + 2\,p^2 + p^3 + 3\,p^4 + 2\,p^5 + 4\,p^6 + 2\,p^7 + 5\,p^8 + O(p^{10})$: P_adic(Modular(Integer))

```
Mmx] b^2
```

$1 + p + O(p^{10})$: P_adic(Modular(Integer))

```
Mmx] p_expansion (100, p)
```

$2\,p^2 + 2$: P_expansion(Modular(Integer))

```
Mmx] p_adic (@p_expansion (100, p))
```

$2 + 2\,p^2 + O(p^{10})$: P_adic(Modular(Integer))

4.6. Quotient fields

4.6.1. C++ interface

Quotient fields are implemented within the class `quotient` from `quotient.hpp`.

```
#include <numerix/rational.hpp>
#include <algebramix/polynomial.hpp>

#define rational C
#define polynomial<C> UP
#define quotient<UP,UP> UF
...
UP p =..., q =...;
UF f (p, q); // Rational function p / q
```

4.6.2. Mathemagix interface

```
Mmx] x == polynomial (0, 1)
```

x: Polynomial(Integer)

```
Mmx] quotient (1 + x, x)
```

$$\frac{x+1}{x}\text{: Quotient(Polynomial(Rational))}$$

4.7. Matrices

The class `matrix<C,V>` implements matrices over `C`. All the available algorithms have a naive implementation in `matrix_naive.hpp`.

```
#include <algebramix/matrix.hpp>
...
matrix<C,V> M (C (0), 3, 2) // 3 x 2 matrix filled with 0
M (0,0)= C(0); M (1,0)= C(1); ...
matrix<C,V> N= M * transpose (M);
mmout << M << "\n";
mmout << det (N) << "\n";
mmout << column_reduced_echelon (M) << "\n";
```

4.7.1. Optimized variants

As for vectors, matrices have a variant for unrolling loops, namely `matrix_unrolled<s,V>`, where `s` represents the number of steps to be unrolled.

Strassen product is implemented in the variant `matrix_strassen<V>`.

Support for SIMD instructions is available through the variant `matrix_simd<n,m,V>`: `n` is the size for unrolling SIMD data and `m` is the one for unrolling the data of the original type. Only SSE2 and SSE3 are supported at the present time.

Multi-threading is possible with the variant `matrix_threads<V>`.

4.7.2. Special types of coefficients

`matrix_double`.hpp contains a recommanded default variant for `double`.

`matrix_int`.hpp contains a recommanded default variant for hardware integers.

`matrix_modular_int`.hpp contains a recommanded default variant for modulars over hardware integers.

`matrix_integer`.hpp contains a recommanded default variant for integers.

4.7.3. Mathemagix interface

```
Mmx] A == [ (i :> Rational) + j | i in 0..5 || j in 0..5 ]
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix}\text{: Matrix(Rational)}$$

```
Mmx] det A
```

0: Rational

```
Mmx] row_echelon A
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} : \text{Matrix}(\text{Rational})$$

```
Mmx] rank A
```

2: Int

```
Mmx] ker A
```

$$\begin{bmatrix} 1 & 2 & 3 \\ -2 & -3 & -4 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} : \text{Matrix}(\text{Rational})$$

4.8. Multi-modular techniques

Naive algorithms for bidirectional multi-modular transformers are implemented in `crt_naive.hpp`.

The following piece of code computes 101 modulo 2, 3, 5, and 7:

```
#include "crt_int.hpp"
...
crt_naive_transformer<int> crter (vec (2, 3, 5, 7));
mmout << direct_crt (101, crter) << "\n";
```

Here `crt_int.hpp` contains specific routines for hardware integers.

4.8.1. Coverings

Let S be a subset of a ring R. A *covering* C of S is a set of coprime elements of R such that any element of S is uniquely charaterized by its residues modulo all the elements of C.

The class `moduli_helper<C,Modulus>` provides the user with a default covering in a specific size whose definition actually depends on R. For example for the integers, the following instructions fill the vector p with a covering for integers of 30 bits.

```
vector<modulus<int> > p;
bool t= moduli_helper<int, modulus<int> >::covering (p, 30);
```

Here the default covering being used is made of the sequence of the prime numbers: 2, 3, 5, 7, 11, etc. The value returned in `t` is `false` if no covering can be made for the specified size. Otherwise `true` is returned.

The sequence used for the covering can be specified as an extra argument:

```
#define Seq fft_prime_sequence_int<10>
vector<modulus<int> > p;
bool t= moduli_helper<int, modulus<int>, Seq>::covering (p, 30);
```

Here are used prime numbers of at most 10 bits. The first ones are well-suited to FFT.

Support for long integers is available in crt_integer.hpp. The following piece of code covers signed integers of 1000 bits with prime hardware integers of at most 20 bits:

```
#include <algebramix/crt_integer.hpp>
#define Seq fft_prime_sequence_int<20>
vector<modulus<int, modulus_int_preinverse<20> > > p;
bool t= moduli_helper<int, modulus<int>, Seq>::covering (p, 1000);
```

4.8.2. Transformers

The divide and conquer algorithm is implemented in `crt_dicho.hpp`:

```
#include <algebramix/crt_integer.hpp>
// let p be a covering obtained as described above
crt_dicho_transformer<integer> crter (p);
```

If the moduli are small, it is often faster to use the naive algorithm. While for large moduli the divide and conquer has soft linear cost. In `crt_blocks.hpp` is implemented a class to stack two different kinds of transformers in the following way:

```
#define Crter_naive crt_naive_transformer<integer, modulus<int> >
#define Crter_dicho crt_dicho_transformer<integer>
#define Crter crt_blocks_transformer<Crter_naive, Crter_dicho, 10>
```

Here 10 is the threshold between the two transformers: each block of 10 moduli is using the naive algorithm, and then the dichotomic algorithm is called with the moduli obtained from the products of those of each block.

Chapter 5

Analyziz

5.1. Univariate polynomials

5.1.1. C++ interface

Analyziz completes Algebramix by providing the user with special variants for polynomials with numerical type coefficients:

- `analyziz/polynomial_double.hpp`,
- `analyziz/polynomial_floating.hpp`,
- `analyziz/polynomial_ball.hpp`.

Root finding is implemented:

- `analyziz/solver_aberth.hpp` contains an implementation of Aberth's method,
- `analyziz/solver_floating.hpp` uses the latter solver with gradual precision doubling,
- `analyziz/solver_ball.hpp` includes certification with ball coefficients.

5.1.2. Mathemagix interface

```
Mmx] use "analyziz"
```

```
Mmx] p: Polynomial Floating == polynomial (-1.0001, 0.0^^4, 1.0)
```

```
1.000000000000000000000000000000000000000 x^5 − 1.0001000000000000000000000000\
00000000003:  Polynomial(Floating)
```

```
Mmx] bit_precision := 128;
     v == roots p
```

```
[−0.80903317406766015764845699419966804250‍59 − 0.587797007527318988514612\
5012659640572021 i, 0.30902317446763615932832932441322177844‍05 − 0.951075536664\
67990966923391945731131135‍80 i, 1.00001999920004799664025533957289252‍8134 −
1.46302386084131197723620294183083642‍8098e − 98 i, 0.309023174467636159328‍3293\
244132217784405 + 0.95107553666467990966923391945731131136‍10 i, −0.80903317406\
76601576484569941996680425059 + 0.58779700752731898851461250126596405720‍21 i]:
Vector(Complex(Floating))
```

```
Mmx] [ eval (p, a) | a in v ]
```

$[-1.17549435082228750796873653722224567781 9e-38, -1.175494350822287507968736537222245677819e-38+5.877471754111437539843682686111228389093e-39\,\mathrm{i}, -1.175494350822287507968736537222245677819e-38-7.315704507899035038550609833263372575725e-98\,\mathrm{i}, -1.469367938527859384960920671527807097273e-39\,\mathrm{i}, -1.175494350822287507968736537222245677819e-38]:$ Vector(Complex(Floating))

```
Mmx] p: Polynomial Ball (Floating, Complex Floating) ==
       polynomial (ball (-1.0001), (ball 0.0)^^4, ball 1.0)
```

$1.000000000000000000000000000000000000\,x^5+(0e-323228496+0e-323228496\,\mathrm{i})\,x^4+(0e-323228496+0e-323228496\,\mathrm{i})\,x^3+(0e-323228496+0e-323228496\,\mathrm{i})\,x^2+(0e-323228496+0e-323228496\,\mathrm{i})\,x-1.000100000000000000000000000000000000:$ Polynomial(Ball(Floating, Complex(Floating)))

```
Mmx] roots p
```

$[-0.8090331740676601576484569941996680 4-0.5877970075273189885146125012659640 57\,\mathrm{i}, 0.309023174467636159328329324413221 78-0.951075536664679909669233919457311 31\,\mathrm{i}, 1.000019999200047996640255339572892528, 0.30902317446763615932832932441322178+0.9510755366646799096692339194573113 11\,\mathrm{i}, -0.80903317406766015764845699419966804+0.58779700752731898851461250126596405 7\,\mathrm{i}]:$ Vector(Ball(Floating, Complex(Floating)))

5.2. Power series

5.2.1. C++ interface

ANALYZIZ completes ALGEBRAMIX by providing the user with special variants for power series with numerical type coefficients:

- `analyziz/series_double.hpp`,
- `analyziz/series_floating.hpp`,
- `analyziz/series_ball.hpp`.

5.2.2. MATHEMAGIX interface

```
Mmx] use "analyziz"
```

```
Mmx] z == series (ball (0 :> Floating), ball (1 :> Floating))
```

$1.000000000000000000000000000000000000\,z+0e-323228496\,z^2+0e-323228496\,z^3+0e-323228496\,z^4+0e-323228496\,z^5+0e-323228496\,z^6+0e-323228496\,z^7+0e-323228496\,z^8+0e-323228496\,z^9+O(z^{10}):$ Series(Ball(Floating, Floating))

```
Mmx] f == log (1 - z)
```

$0e-38-1.000000000000000000000000000000000000\,z-0.50000000000000000000\backslash$ $0000000000000000\,z^2-0.333333333333333333333333333333333333\,z^3-0.25000000000\backslash$ $0000000000000000000000000\,z^4-0.200000000000000000000000000000000000\,z^5-$ $0.166666666666666666666666666666666667\,z^6-0.142857142857142857142857142857\backslash$ $142857\,z^7-0.125000000000000000000000000000000000\,z^8-0.11111111111111111111\backslash$ $1111111111111111\,z^9+O(z^{10})$: Series(Ball(Floating, Floating))

```
Mmx] center f[100]
```

$-0.0100000000000000000000000000000000000001$: Floating

```
Mmx] radius f[100]
```

$4.6079378552233955693302727940582943981 85e-38$: Floating

5.3. MATRICES

5.3.1. C++ interface

ANALYZIZ completes ALGEBRAMIX by providing the user with special variants for matrices with numerical type entries:

- `analyziz/matrix_double.hpp`,
- `analyziz/matrix_floating.hpp`,
- `analyziz/matrix_ball.hpp`.

Computations of eigen values and vectors are implemented in the following files:

- `analyziz/eigen.hpp` contains the generic implementation,
- `analyziz/eigen_floating.hpp` uses the latter solver with gradual precision doubling,
- `analyziz/eigen_ball.hpp` includes certification with ball coefficients.

5.3.2. MATHEMAGIX interface

```
Mmx] use "analyziz";
     type_mode? := true;
```

```
Mmx] rnd () == {
       x == uniform_deviate (0.0, 1.0);
       return ball (x, 0.0000001);
     };
```

```
Mmx] M == [ rnd () | i in 0..3 || j in 0..3 ]
```

$$\begin{bmatrix} 0.601796 & 0.205660 & 0.787142 \\ 0.965621 & 0.989548 & 0.209385 \\ 0.86737 & 0.437911 & 0.142280 \end{bmatrix}: \text{Matrix(Ball(Floating, Floating))}$$

```
Mmx] significant_digits := 5; // print 5 digits only
     eigen_solve M
```

$$\begin{bmatrix} 0.56938-0.0073627\,\mathrm{i} & -0.49790+0.052058\,\mathrm{i} & -0.65354+0.031053\,\mathrm{i} \\ 0.91734-0.011862\,\mathrm{i} & 0.90368-0.094485\,\mathrm{i} & 0.33724-0.016024\,\mathrm{i} \\ 0.56815-0.0073467\,\mathrm{i} & -0.12633+0.013209\,\mathrm{i} & 0.75461-0.035855\,\mathrm{i} \end{bmatrix}:$$
Matrix(Ball(Floating, Complex(Floating)))

```
Mmx] v == eigenvalues M
```

$[0.42825, -0.4132, 1.7186]:$ Vector(Ball(Floating, Complex(Floating)))

```
Mmx] radius v[0]
```

$1.950520130e-6:$ Floating

Chapter 6

Factorix

6.1. Separable factorization

```
Mmx] use "factorix"
```

```
Mmx] help separable_factorization
```

```
separable_factorization : Polynomial (Integer) -> Vector
(Separable_factor (Polynomial (Integer)))                          (Native)
```

```
separable_factorization : Polynomial (Rational) -> Vector
(Separable_factor (Polynomial (Rational)))                         (Native)
```

```
separable_factorization : Polynomial (Modular (Integer)) -> Vector
(Separable_factor (Polynomial (Modular (Integer))))                (Native)
```

```
Mmx] x == polynomial (0 :> Integer, 1 :> Integer);
```

```
Mmx] separable_factorization ((x-1)^10 * (x-2)^2)
```

$(x-2)^2\,(x-1)^{10}$

```
Mmx] v == separable_factorization ((x-1)^10 * (x-2)^2 mod modulus 5)
```

$(x+3)^2\,(x^5+4)^2$

```
Mmx] factor (v[1])
```

$x+4$

```
Mmx] ideg (v[1])
```

5

```
Mmx] mul (v[1])
```

2

```
Mmx] separable? ((x - 1)^10 - 1)
```

true

6.2. Squarefree factorization

```
Mmx] use "factorix"
```

```
Mmx] help squarefree_factorization
```

```
squarefree_factorization : Integer -> Vector (Squarefree_factor
(Integer))                                                     (Native)

squarefree_factorization : Rational -> Vector (Squarefree_factor
(Rational))                                                    (Native)

squarefree_factorization : Polynomial (Integer) -> Vector
(Squarefree_factor (Polynomial (Integer)))                     (Native)

squarefree_factorization : Polynomial (Rational) -> Vector
(Squarefree_factor (Polynomial (Rational)))                    (Native)

squarefree_factorization : Modular (Integer) -> Vector (Squarefree_factor
(Modular (Integer)))                                           (Native)

squarefree_factorization : Polynomial (Modular (Integer)) -> Vector
(Squarefree_factor (Polynomial (Modular (Integer))))           (Native)
```

```
Mmx] x == polynomial (0 :> Integer, 1 :> Integer);
```

```
Mmx] squarefree_factorization (-(x-1)^10 * (x-2)^2)
```

$-(x-2)^2\,(x-1)^{10}$

```
Mmx] squarefree_factorization ((x-1)^10 * (x-2)^2 mod modulus 5)
```

$(x+3)^2\,(x+4)^{10}$

```
Mmx] time_mode? := true;
```

```
Mmx] squarefree_factorization (10006452100200091 * 9 * (x^10-1)^3 *
     (x-2)^2)
```

$35164639\,284560069\,3^2\,(x-2)^2\,(x^{10}-1)^3$

```
Computed in 29365 ms
```

```
Mmx] probable_squarefree_factorization (10006452100200091 * (x^10-1)^3 *
     (x-2)^2)
```

$35164639\,284560069\,(x-2)^2\,(x^{10}-1)^3$

```
Computed in 855 ms
```

```
Mmx] help squarefree?
```

```
squarefree? : Integer -> Boolean                               (Native)
squarefree? : Polynomial (Integer) -> Boolean                  (Native)
squarefree? : Rational -> Boolean                              (Native)
squarefree? : Polynomial (Rational) -> Boolean                 (Native)
squarefree? : Modular (Integer) -> Boolean                     (Native)
squarefree? : Polynomial (Modular (Integer)) -> Boolean        (Native)
```

```
Mmx] squarefree? 35164639
```

true

Chapter 7

Geomsolvex

7.1. Geometric resolution

7.2. Use from the interpreter

```
Mmx] use "geomsolvex"
```

```
Mmx] X == coordinate ('x); Y == coordinate ('y);
```

```
Mmx] x == polynomial_dag (1 :> Rational, X);
     y == polynomial_dag (1 :> Rational, Y);
```

```
Mmx] f1 == x^2 + y^2 - 1;
     f2 == x^2 + x * y - 2;
```

```
Mmx] probable_geometric_solve_reduced_regular ([f1, f2])
```

$$\text{lifting\_fiber}\left(x^2+y^2-1, x^2+x\,y-2, \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right], [0,0], [], x, x^4-\frac{5}{2}x^2+2, 4\,x^3-5\,x, [5\,x^2-8, 3\,x^2-2]\right)$$

```
Mmx] p: Integer == 101;
```

```
Mmx] x == polynomial_dag (1 :> Integer, X);
     y == polynomial_dag (1 :> Integer, Y);
```

```
Mmx] f1 == x^2 + y^2 - 1;
     f2 == x^2 + x * y - 2;
```

```
Mmx] probable_geometric_solve_reduced_regular ([f1 mod p, f2 mod p])
```

$$\text{lifting\_fiber}\left(x^2+y^2-1, x^2+x\,y-2, \left[\begin{array}{cc} 27 & 61 \\ 62 & 17 \end{array}\right], [79,61], [], x, x^4+79\,x^3+23\,x^2+68\,x+36, 1, [x, 95\,x^3+99\,x^2+60\,x+59]\right)$$

For long computations, verbosity can be enabled by setting `geomsolvex_verbose`.

Chapter 8

Continewz

8.1. Analytic functions

This document contains simple examples using analytic functions within Mathemagix.

8.1.1. Use from C++

Basic calculations are shown in `continewz/test/analytic_test.cpp`. The following sample of code shows how to constuct simple analytic functions operating on balls:

```
#include <numerix/ball_complex.hpp>
#include <continewz/analytic_matrix.hpp>
using namespace mmx;

#define R          ball<floating<> >
#define C          ball<complex<floating<> > >
#define Polynomial polynomial<C>
#define Analytic   analytic<C>

int
main () {
  Polynomial P= seq (C (1), C (-2), C (3));
  Analytic f (1);
  Analytic g (P);

  mmout << "f= " << f << lf;
  mmout << "g= " << g << lf;
  mmout << "f + g= " << f + g << lf;
}
```

8.1.2. Use from the interpreter

The following session shows basic operations available for constructing and studing $z \mapsto \log(1 - z)$:

```
Mmx] use "continewz";
     type_mode? := true;
```

```
Mmx] z == analytic (ball 0.0, ball 1.0)
```

$0e-323228496+0e-323228496\,\mathrm{i}+1.0000000000000000\,z+(0e-323228496+0e-323228496\,\mathrm{i})\,z^2+(0e-323228496+0e-323228496\,\mathrm{i})\,z^3+(0e-323228496+0e-323228496\,\mathrm{i})\,z^4+(0e-323228496+0e-323228496\,\mathrm{i})\,z^5+(0e-323228496+0e-323228496\,\mathrm{i})\,z^6+(0e-323228496+0e-323228496\,\mathrm{i})\,z^7+(0e-32322\backslash 8496+0e-323228496\,\mathrm{i})\,z^8+(0e-323228496+0e-323228496\,\mathrm{i})\,z^9+O(z^{10})$: Analytic(Ball(Floating, Floating), Ball(Floating, Complex(Floating)))

```
Mmx] l == log (1 - z)
```

$0e-18+0e-18\,\mathrm{i}-1.0000000000000000\,z-0.50000000000000000\,z^2-0.333333333333\backslash 3333\,z^3-0.25000000000000000\,z^4-0.20000000000000000\,z^5-0.1666666666666667\,z^6-0.1428571428571429\,z^7-0.12500000000000000\,z^8-0.11111111111111111\,z^9+O(z^{10})$: Analytic(Ball(Floating, Floating), Ball(Floating, Complex(Floating)))

```
Mmx] l[20]
```

-0.05000000000000000: Ball(Floating, Complex(Floating))

```
Mmx] radius_bound l
```

0.97418150340835296: Ball(Floating, Floating)

```
Mmx] upper_bound (l, 0.8)
```

1.6585741063681269: Ball(Floating, Floating)

```
Mmx] lower_bound (l, 0.8)
```

$0e-323228496$: Ball(Floating, Floating)

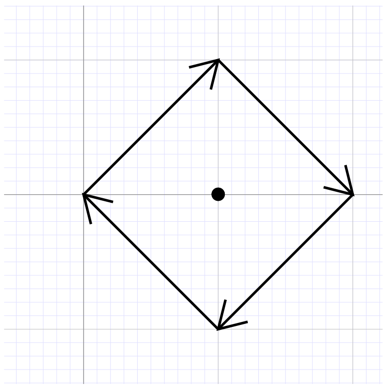
```
Mmx] tail_bound (l, 0.8, 20)
```

0.00330897595339859: Ball(Floating, Floating)

```
Mmx] l # (complex ( 1.0,  1.0)) # (complex ( 1.0, -1.0))
       # (complex (-1.0, -1.0)) # (complex (-1.0,  1.0))
```

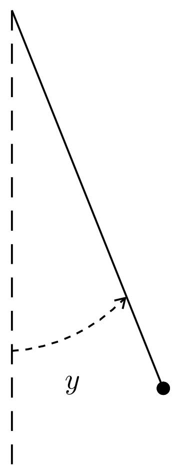
$-6.2831853071796\,\mathrm{i}-1.000000000000000\,z-0.500000000000000\,z^2-0.333333333\backslash 33333\,z^3-0.250000000000000\,z^4-0.200000000000000\,z^5-0.16666666666667\,z^6-0.142857142857143\,z^7-0.125000000000000\,z^8-0.111111111111111\,z^9+O(z^{10})$: Analytic(Ball(Floating, Floating), Ball(Floating, Complex(Floating)))

The latter computation corresponds to the evaluation of $z\mapsto\log(1-z)$ along the following loop around the pole 1:



8.2. Reliable integration of differential equations

This document shows how to use reliable integration for the classical pendulum problem, within the Mathemagix interpreter.



Pendulum vector field

$$\begin{aligned} y' &= y' \\ y'' &= -\sin y \\ (\sin y)' &= y' \cos y \\ (\cos y)' &= -y' \sin y \end{aligned}$$

```
Mmx] use "continewz"
```

```
Mmx] bit_precision := 128; significant_digits := 6;
```

```
Mmx] z == analytic (ball 0.0, ball 1.0);
```

```
Mmx] vector_field (v) == [ v[1], -v[2], v[1]*v[3], -v[1]*v[2] ];
```

```
Mmx] initial_conditions == [ ball 0.0, ball 1.0, ball 0.0, ball 1.0 ];
```

```
Mmx] pendulum == integrate_analytic (vector_field, initial_conditions);
```

```
Mmx] pendulum[0]
```

$0e-323228496+0e-323228496\,\mathrm{i}+1.00000\,z+(0e-323228495+0e-323228495\,\mathrm{i})\,z^2-0.166667\,z^3+(0e-323228495+0e-323228495\,\mathrm{i})\,z^4+0.0166667\,z^5+(0e-323228495+0e-323228495\,\mathrm{i})\,z^6-0.00257937\,z^7+(0e-323228495+0e-323228495\,\mathrm{i})\,z^8+4.43673e-4\,z^9+O(z^{10})$

```
Mmx] significant_digits := 0;
```

```
Mmx] pendulum[0][99]
```

$-3.6721130299715737296645871084399556e-35$

```
Mmx] pendulum[0] (0.0)
```

$0e-323228495+0e-323228495\,\mathrm{i}$

```
Mmx] pendulum[0] (0.1)
```

$0.0998334997425063911976915124459230269$

```
Mmx] radius pendulum[0] (0.2)
```

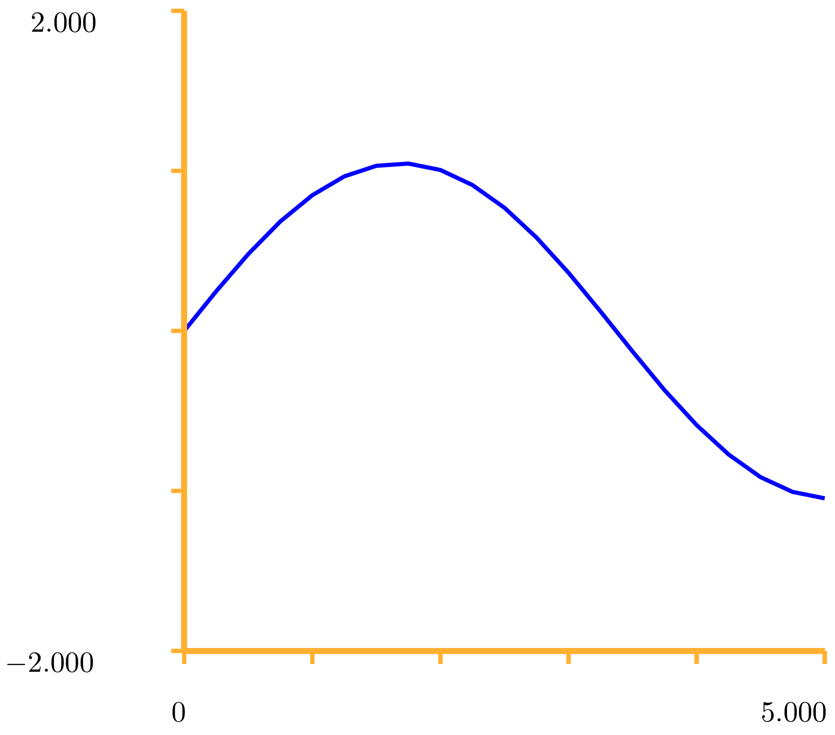
$3.34584113553296018447260516025521065678 7e-38$

```
Mmx] significant_digits := 15;
```

```
Mmx] points == [ (0.25 * t, pendulum[0] (0.25 * t)) || t in 0 to 20 ]
```

$$\begin{bmatrix}
0 & 0e-323228495+0e-323228495\,\mathrm{i} \\
0.250000000000000 & 0.247411953617254 \\
0.500000000000000 & 0.479668179055606 \\
0.750000000000000 & 0.683328616912675 \\
1.00000000000000 & 0.847798681677117 \\
1.25000000000000 & 0.965644417872573 \\
1.50000000000000 & 1.03227880694143 \\
1.75000000000000 & 1.04541037505998 \\
2.00000000000000 & 1.00461455867257 \\
2.25000000000000 & 0.911239711602712 \\
2.50000000000000 & 0.768688968108693 \\
2.75000000000000 & 0.582949063558809 \\
3.00000000000000 & 0.363070877719344 \\
3.25000000000000 & 0.121202209128179 \\
3.50000000000000 & -0.128146241310137 \\
3.75000000000000 & -0.369588550851423 \\
4.00000000000000 & -0.588662784601856 \\
4.25000000000000 & -0.773301810983781 \\
4.50000000000000 & -0.914543989767028 \\
4.75000000000000 & -1.00648255368516 \\
5.00000000000000 & -1.04577846614345
\end{bmatrix}$$

```
Mmx] include "graphix/simple_plot.mmx";
     $draw_diagram ([ [Re center points[i,0],
                      Re center points[i,1]] | i in 0..rows points ])
```



8.3. Homotopy continuation

This document briefly describes how to use numeric homotopy continuation within Mathemagix for solving a polynomial system.

```
Mmx] use "continewz";
     include "continewz/homotopy_solve.mmx"
```

```
Mmx] x1 == coordinate ('x[1]); x2 == coordinate ('x[2]);
```

```
Mmx] f1 == x1 * x2 + x1^2 + x2^2 + x1 - 20
```

$$x_2^2 + x_1 x_2 + x_1^2 + x_1 - 20$$

```
Mmx] f2 == x1 - x2^2 - x1^2 + x2 - 10
```

$$-x_2^2 + x_2 - x_1^2 + x_1 - 10$$

```
Mmx] sys == [ f1, f2 ]
```

$$[x_2^2 + x_1 x_2 + x_1^2 + x_1 - 20, -x_2^2 + x_2 - x_1^2 + x_1 - 10]$$

```
Mmx] sols == homotopy_solve (sys, [ x1, x2 ], 1 :> Ball (Floating,
     Complex Floating))
```

$$\begin{bmatrix} 3.24579224630132 - 3.127341848803361\,\mathrm{i} & 2.886009272444231 + 3.59890931655154\,\mathrm{i} \\ 3.24579224630132 + 3.127341848803361\,\mathrm{i} & 2.886009272444231 - 3.59890931655154\,\mathrm{i} \\ -3.74579224630132 - 5.411641139185416\,\mathrm{i} & -4.386009272444231 + 4.702550221936963\,\mathrm{i} \\ -3.74579224630132 + 5.411641139185416\,\mathrm{i} & -4.386009272444231 - 4.702550221936963\,\mathrm{i} \end{bmatrix}$$

```
Mmx] [ eval (sys [j], [ x1, x2 ], row (sols, 0)) | j in 0..2 || i in 0..2
     ]
```

$$\begin{bmatrix} 0e - 14 + 0e - 14\,\mathrm{i} & 0e - 14 + 0e - 14\,\mathrm{i} \\ 0e - 14 + 0e - 14\,\mathrm{i} & 0e - 14 + 0e - 14\,\mathrm{i} \end{bmatrix}$$

```
Mmx] bit_precision:= 128;
```

```
Mmx] sols[0,0]
```

$$3.24579224630132 - 3.127341848803361\,\mathrm{i}$$

```
Mmx] center sols[0,0]
```

$$3.24579224630131954617 - 3.12734184880336128190\,\mathrm{i}$$

```
Mmx] radius sols[0,0]
```

$$9.979312409837135579177212812263881238814e - 17$$

Chapter 9

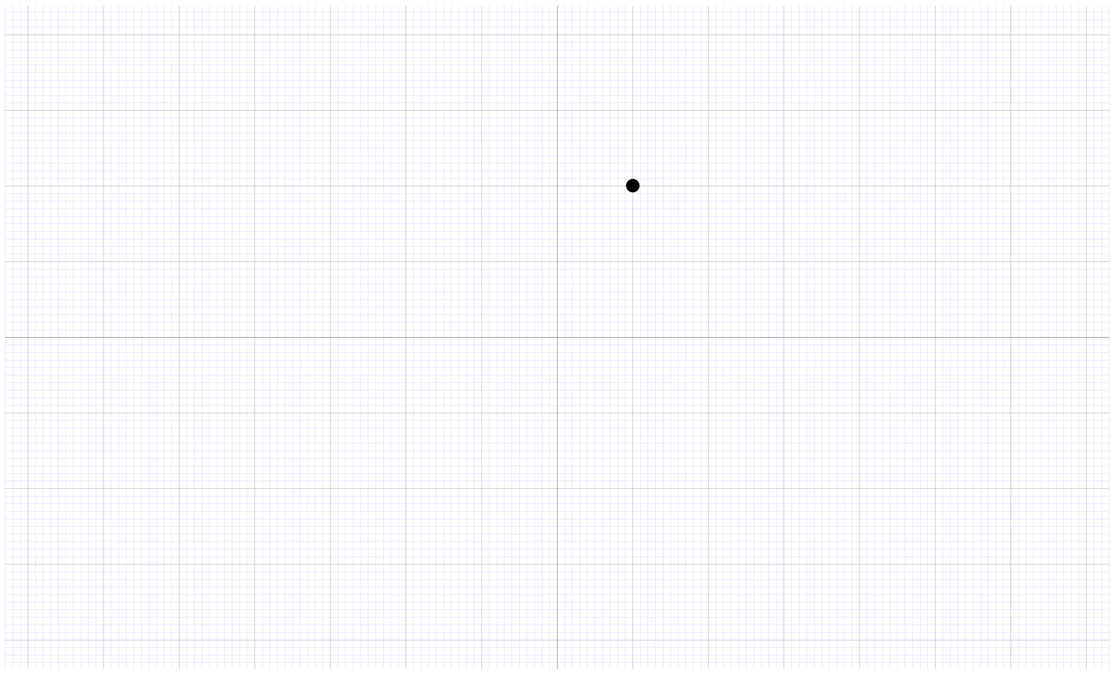
Graphix

9.1. Graphical interface to TeXmacs

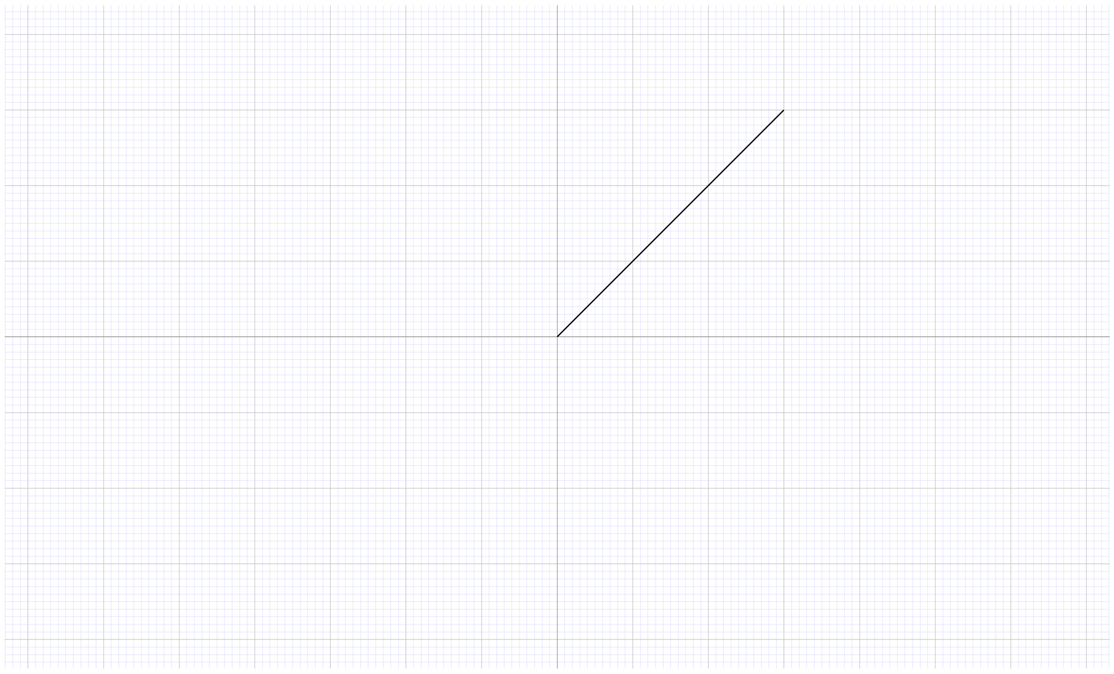
The main graphical objects are defined in `graphix/graphics.mmx`.

```
Mmx] use "graphix"
```

```
Mmx] $point (1, 2)
```



```
Mmx] $line ($point (0, 0), $point (3, 3))
```



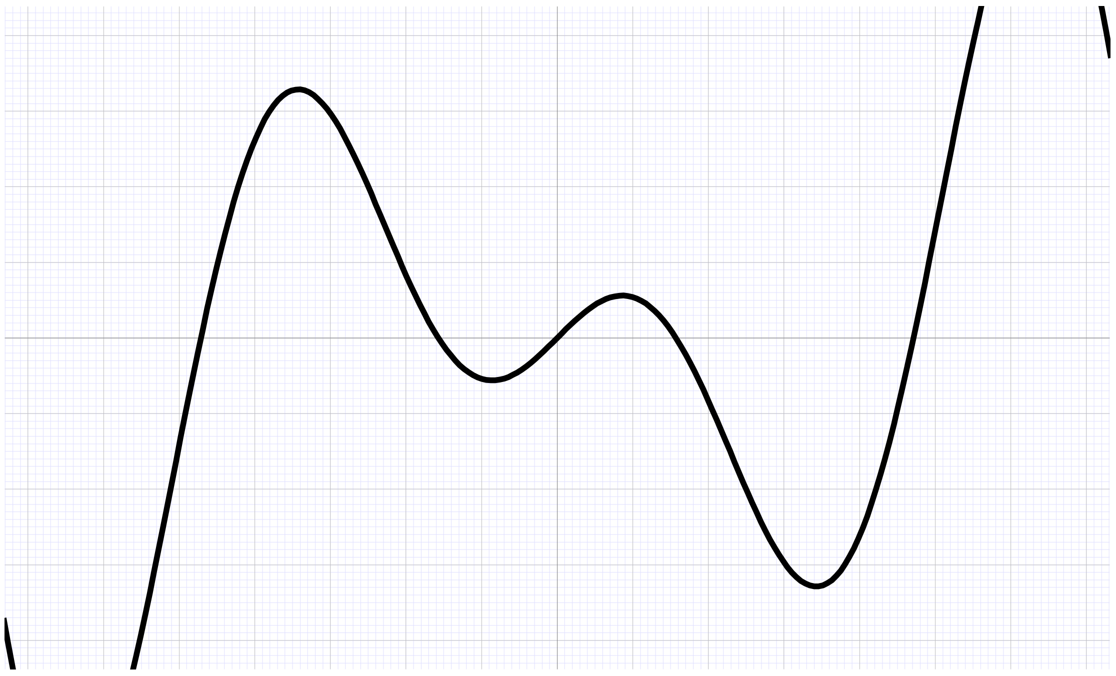
9.2. Graphs of functions

```
Mmx] use "graphix"
```

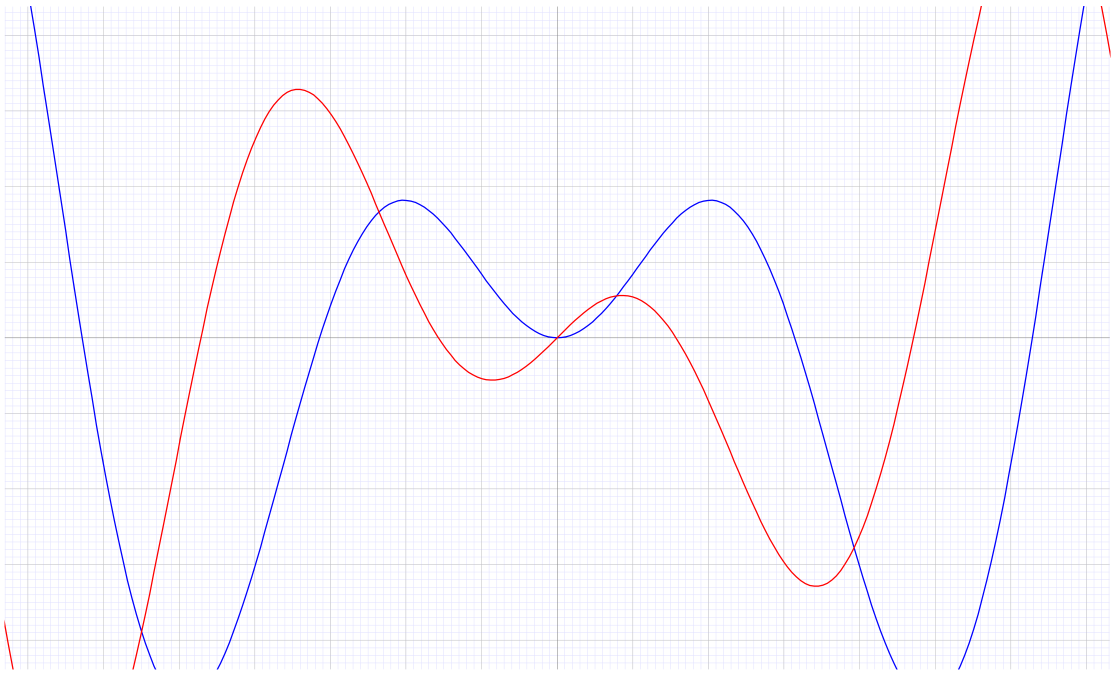
```
Mmx] foo (x) == x * sin x;
```

```
Mmx] bar (x) == x * cos x;
```

```
Mmx] $width ("4ln", $graph (bar))
```



```
Mmx] $group ($color ("blue", $graph (foo)), $color ("red", $graph (bar)))
```



9.3. Animations

Please notice that, at the present time, the following animations are only visible when run within TEXMACS.

9.3.1. Animations that remain in the document

```
Mmx] use "graphix"
```

```
Mmx] $o == $point (0, 0);
```

```
Mmx] $p (t) == $point (2 * cos t, 2 * sin t);
```

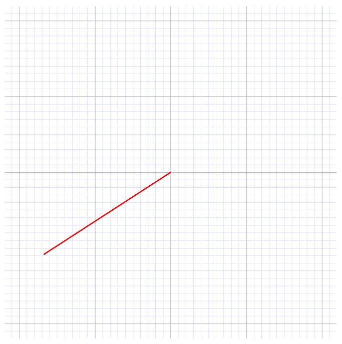
```
Mmx] $red x == $color ("red", x);
```

```
Mmx] $animation ($hold ("0.04sec", $red $line ($o, $p (t/10))) | t in 0
     to 63)
```

9.3.2. Animations that operate on the document

```
Mmx] demo (view: Dynamic, sec: Double): Void == {
       start: Double == time ();
       while time () - start < (1000 * sec) do {
         t: Double == (time () - start) / 1000;
         g: Graphics == $graphics $red $line ($o, $p (t));
         assign (view, as_document
                 $grid (0, 0, 1, $size ("0.3par", "0.3par", g)));
         flush mmout; } };
```

```
Mmx] view == dynamic (as_document $grid (0, 0, 1,
                                  $size ("0.3par", "0.3par", $graphics ())))
```



```
Mmx] demo (view, as_double 10.0);
```

Chapter 10
Mblad

10.1. Download and installation of Blad

Blad is available from

http://www.lifl.fr/~boulier/

In order to work properly with Mathemagix, a version at least 3.9 of Blad compiled with Gmp and Mpfr enabled is necessary. The installation instructions can be found from the latter link.

10.2. Using Bad from Mathemagix

At the present time only a small subset of the Blad functionalities is available.

```
Mmx] use "mblad";
     type_mode? := true;
```

```
Mmx] help bad_rosenfeld_groebner
```

```
bad_rosenfeld_groebner : (Vector (MVPolynomial (Rational)), Vector
(MVPolynomial (Rational)), Vector (Coordinate), Vector (Coordinate)) ->
Vector (Regular_chain (MVPolynomial (Rational)))                    (Native)

bad_rosenfeld_groebner : (Vector (MVPolynomial (Rational)), Vector
(Coordinate), Vector (Coordinate)) -> Vector (Regular_chain (MVPolynomial
(Rational)))                                                        (Native)
```

```
Mmx] x: Coordinate == coordinate ('x)
```

x: Coordinate

```
Mmx] u: Coordinate == coordinate ('u)
```

u: Coordinate

```
Mmx] u_x: Coordinate == derivative (u, x)
```

$\frac{\partial u}{\partial x}$: Coordinate

```
Mmx] eqn: MVPolynomial Rational == u_x^2 - mvpolynomial (4 :> Rational) *
     u
```

$\left(\frac{\partial u}{\partial x}\right)^2 - 4\,u$: MVPolynomial(Rational)

```
Mmx] sols == bad_rosenfeld_groebner ([ eqn ], [ x ], [ u ])
```

$$\left[\left[\left[\left(\frac{\partial u}{\partial x}\right)^2 - 4\,u\right], [\text{differential}, \text{autoreduced}, \text{squarefree}, \text{primitive}]\right], [[u], [\text{differential}, \text{prime}, \text{autoreduced}, \text{squarefree}, \text{primitive}]]\right]: \ \text{Vector}(\text{Regular\_chain}(\text{MVPolynomial}(\text{Rational})))$$

```
Mmx] sols[0].decision_system
```

$$\left[\left(\frac{\partial u}{\partial x}\right)^2 - 4\,u\right]: \ \text{Vector}(\text{MVPolynomial}(\text{Rational}))$$

```
Mmx] sols[0].attrib
```

$$[\text{differential}, \text{autoreduced}, \text{squarefree}, \text{primitive}]: \ \text{Vector}(\text{Generic})$$

Chapter 11

MFGb

11.1. Download and installation of FGb

This document briefly describes how to install FGb.

FGb is a piece of software written by Jean-Charles Faugère, and available from

http://www-salsa.lip6.fr/~jcf/Software/FGb/C API/.

It is not open source, so *please refer to the copyright notice before using* FGb.

11.1.1. Installation for Linux platforms

You need to download the archive `call_FGb*.linux.*.tar.gz` that corresponds to your plateform, and to unarchive it:

```
tar zxvf call_FGb*.linux.*.tar.gz
```

Then you can copy the library `libcallfgb.a` (located in `call_FGb/nv/maple/C/*/`) into a directory of your choice. If copied into `/usr/local/lib` or `/usr/local/lib64`, then no special configuration option is needed for Mathemagix.

11.1.2. Installation for Mac OSX platforms

You need to download the archive `call_FGb4.mac.universal.tar.gz`, and to unarchive it:

```
tar zxvf call_FGb.mac.universal.tar.gz
```

Then you might want to copy all the libraries from `call_FGb/nv/maple/C/darwin/` into a dedicated directory of your choice. For instance, if copied into `/usr/local/lib`, then no special configuration option is needed for Mathemagix.

11.1.3. Configuration

Using `configure`, if the Fgb librairies have not been copied into one of the aforementioned standard location, then you need to specify their path as follows:

```
./configure --with-fgb="fgb_path"
```

Alternatively, for `cmake`, you need to specify the path as follows:

```
cmake ../mmx/mfgb -DFGB_PATH="/fgb_path/lib"
```

11.2. Using FGb within Mathemagix

This document briefly describes how to use FGB within MATHEMAGIX.

11.2.1. Use from C++

Block DRL bases over the rational numbers can be directly computed with `sparse_polynomial` from `multimix`. Examples are available from `mfgb/test/`.

11.2.2. Use from the interpreter

```
Mmx] use "mfgb"
```

```
Mmx] X == coordinate ('x); Y == coordinate ('y);
```

```
Mmx] x == mvpolynomial (1 :> Rational, X);
     y == mvpolynomial (1 :> Rational, Y);
```

```
Mmx] f1 == x^2 + y^2 - 1;
     f2 == x^2 + x * y - 2;
```

```
Mmx] fgb_basis_drldrl ([f1, f2], [X], [Y])
```

$[2\,y^3+x, 2\,y^4+y^2+1]$

```
Mmx] p: Integer == 7
```

7

```
Mmx] x == mvpolynomial (1 :> Integer, X);
     y == mvpolynomial (1 :> Integer, Y);
```

```
Mmx] f1 == x^2 + y^2 - 1;
     f2 == x^2 + x * y - 2;
```

```
Mmx] fgb_basis_drldrl ([f1 mod p, f2 mod p], [X], [Y])
```

$[2\,y^3+x, y^4+4\,y^2+4]$

For long computations, verbosity can be activated with `fgb_verbose? := true`.

Chapter 12
Mpari

12.1. Download and installation of Pari/GP

Pari/GP is available from

http://pari.math.u-bordeaux.fr/

In order to work properly with Mathemagix, a version at least 2.6.0 of Pari/GP compiled with Gmp is necessary. The installation instructions can be found from the latter link.

12.2. Using Pari from Mathemagix

At the present time only a small subset of the Pari functionalities is available.

```
Mmx] use "mpari"
```

```
Mmx] pari_square_root 10000023409823400002340982340000003^2
```

10000023409823400002340982340000003

```
Mmx] pari_factor (2^50-1)
```

$[[3, 1], [11, 1], [31, 1], [251, 1], [601, 1], [1801, 1], [4051, 1]]$

```
Mmx] pari_is_squarefree (2309834082340322)
```

true

```
Mmx] pari_kronecker_symbol (10, 3)
```

1

```
Mmx] x: Polynomial Integer == polynomial (0,1);
```

```
Mmx] v == pari_factor (x^10 - 1)
```

$[[x-1, 1], [x+1, 1], [x^4-x^3+x^2-x+1, 1], [x^4+x^3+x^2+x+1, 1]]$

```
Mmx] q: Polynomial Integer == x^2 + x - 1001
```

$x^2+x-1001$

```
Mmx] pari_nfdisc (q)
```

445

```
Mmx] pari_nfbasis (q)
```

$$\left[1, \frac{1}{3} x - \frac{1}{3}\right]$$

Index