Numerix

Big integer library

Michel Quercia
`michel.quercia@prepas.org`

version 0.21, March 18, 2005

Contents

Chapter 1

Presentation

`Numerix` is a library implementing arbitrary long signed integers and the usual arithmetic operations between those numbers. Designed for a use with the Objective-Caml language, it is also available with reduced functionalities for the Camllight, C and Pascal languages on 32 or 64 bit Unix-type computers. It is shipped in three different versions:

`Clong`:

> written in standard C. The basic object is a "digit", the length of which is half of a machine word. So, elementary operations giving a two digit result are implemented with ordinary C operations on `long` variables, and this library should be portable to every computer with a binary arithmetic and words of even bit-length not fewer than 32 bits.

`Dlong`:

> also written in C, but a digit is a whole machine word. Operations between digits are handled with the `long long` datatype (a `gcc` feature) which for now restricts this library to 32 bit computers (on the 64 bit computers at the Inria, a `long long` is of the same size as a `long` and there are no double-precision operations).

`Slong` :

> written partly in assembly language for Intel `x86` processors and partly in C, a digit is a 32 bit word. Two different implementations of the `Slong` module are available : one using the standard arithmetic operations available on every processor belonging to the `x86` family, and one using the `SSE2` instruction set available on recent Pentium processors, starting at Pentium-4.

Concerning the speed, `Numerix` compares well to the other multi-precision libraries commonly available, especially `Big_int` (the adaptation of `BigNum` for Camllight/Ocaml) and `GMP`. Below are the computing times for the n first decimal digits of π on a Linux PC with a Pentium-4-2.8Ghz processor and a 512Mb random access memory:

	n	Slong SSE2	Slong x86	Dlong	Clong	GMP 4.1.4	Big_int
Numerix-0.21	10^4	0.01s	0.01s	0.04s	0.06s	0.01s	0.30s
	10^5	0.23s	0.46s	0.99s	1.25s	0.47s	29.75s
	10^6	4.24s	9.23s	18.91s	24.31s	11.33s	2830s
Numerix-0.19	10^4		0.03s	0.08s	0.08s		
	10^5		0.80s	1.92s	2.09s		
	10^6		16.24s	42.32s	47.91s		

The same algorithm is used in the five cases, derived from a series expansion from Ramanujan, and only the big integer implementation differs. The `Slong`, `Clong`, `Dlong` and `GMP` libraries were used with a main program in C, whereas `Big_int` was with a main program in Ocaml. However, the influence of the main language on the running time is negligible for this kind of program for which the main part of the computing time is spent with the operations on several-million-bit-long numbers; the running times are similar when all the libraries are used with an Ocaml main program.

The table above also shows the computing times for the same program on the same computer with the previous `Numerix` version, released in 2002. For the new version the C kernel code was entirely rewritten with more sophisticated algorithms especially for the multiplication and the division of very big integers, resulting in a speedup by a factor of 2 for such numbers. Also, the use of the `SSE2` instruction set for the `Slong` module results in another speedup by a factor of 2 when compared to the standard arithmetic instruction set of Pentium processors. Therefore, the fastest version of `Numerix`-0.21 is 4 times faster than the fastest version of `Numerix`-0.19 for this test.

Chapter 2

Use with Ocaml

Contents

2.1 Interface

`Numerix`-0.21 was developed with Ocaml-3.08. A few tests with Ocaml-3.06, Ocaml-3.07 and Ocaml-3.09-dev have not shown any problem. Thus, `Numerix` is likely to run with any version of Ocaml between 3.06 and 3.09 inclusive. The `Numerix` module contains:

- an abstract description (`Int_type` signature) common to all available big integer implementations;
- the concrete descriptions of the `Clong`, `Dlong`, `Slong`, `GMP` and `Big` submodules compatible with the `Int_type` signature;
- an `Infixes` functor allowing one to use the most common operations on big integers with the infix notation;
- a `Cmp` functor returning an implementation compatible with the `Int_type` signature from two such implementations `A` and `B`, and enabling one to check that a same computation yields identical results with `A` and with `B`;
- a `Count` functor returning a new implementation compatible with the `Int_type` signature from such an implementation `A`, and providing statistics on the number of operations done on big integers as well as on the average and maximum operand sizes;
- a `Rfuns` functor implementing approximation algorithms for the usual mathematical real-valued functions;
- a `Start` functor enabling one to choose at run time through the use of a command line option which big integer implementation to use;
- a timing function.

Below is the public interface from the `numerix.mli` file:

```
                    (* +--------------------------+
                       |  Abstract description    |
                       +--------------------------+ *)

(* rounding mode *)
type round_mode = Floor | Nearest_up | Ceil | Nearest_down

module type Int_type = sig

  type t                      (* integer         *)
  type tref                   (* mutable integer *)
  val name : unit -> string   (* module name     *)
  val zero : t                (* the number 0    *)
  val one  : t                (* the number 1    *)

  (* reference      ------------------------------------------------------ *)
  (*                mode          r       s       a    b      c      res  *)
  val make_ref    :                               t ->                 tref
  val copy_in     :               tref ->         t ->                 unit
  val copy_out    :               tref ->                            t
  val look        :               tref ->                            t

  (* addition       ----------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val add         :                               t -> t   ->        t
  val add_1       :                               t -> int ->        t
  val add_in      :               tref ->         t -> t   ->        unit
  val add_1_in    :               tref ->         t -> int ->        unit

  (* subtraction   ------------------------------------------------------ *)
  (*                mode          r       s       a    b      c      res  *)
  val sub         :                               t -> t   ->        t
  val sub_1       :                               t -> int ->        t
  val sub_in      :               tref ->         t -> t   ->        unit
  val sub_1_in    :               tref ->         t -> int ->        unit

  (* multiplication ----------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val mul         :                               t -> t   ->        t
  val mul_1       :                               t -> int ->        t
  val mul_in      :               tref ->         t -> t   ->        unit
  val mul_1_in    :               tref ->         t -> int ->        unit

  (* division       ----------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val quomod      :                               t -> t   ->        t*t
  val quo         :                               t -> t   ->        t
  val modulo      :                               t -> t   ->        t
  val gquomod     : round_mode ->                 t -> t   ->        t*t
  val gquo        : round_mode ->                 t -> t   ->        t
  val gmod        : round_mode ->                 t -> t   ->        t

  val quomod_in   :               tref -> tref -> t -> t   ->        unit
  val quo_in      :               tref ->         t -> t   ->        unit
  val mod_in      :                       tref -> t -> t   ->        unit
  val gquomod_in  : round_mode -> tref -> tref -> t -> t   ->        unit
  val gquo_in     : round_mode -> tref ->         t -> t   ->        unit
  val gmod_in     : round_mode ->         tref -> t -> t   ->        unit

  val quomod_1    :                               t -> int ->        t*int
  val quo_1       :                               t -> int ->        t
  val mod_1       :                               t -> int ->        int
  val gquomod_1   : round_mode ->                 t -> int ->        t*int
  val gquo_1      : round_mode ->                 t -> int ->        t
  val gmod_1      : round_mode ->                 t -> int ->        int

  val quomod_1_in :               tref ->         t -> int ->        int
  val quo_1_in    :               tref ->         t -> int ->        unit
  val gquomod_1_in: round_mode -> tref ->         t -> int ->        int
  val gquo_1_in   : round_mode -> tref ->         t -> int ->        unit

  (* absolute value ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val abs         :                               t ->               t
  val abs_in      :               tref ->         t ->               unit

  (* opposite       ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val neg         :                               t ->               t
  val neg_in      :               tref ->         t ->               unit

  (* p-th power     ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val sqr         :                               t ->               t
  val pow         :                               t -> int ->        t
  val pow_1       :                             int -> int ->        t
  val powmod      :                               t -> t   -> t ->   t
  val gpowmod     : round_mode ->                 t -> t   -> t ->   t
  val sqr_in      :               tref ->         t ->               unit
  val pow_in      :               tref ->         t -> int ->        unit
  val pow_1_in    :               tref ->       int -> int ->        unit
  val powmod_in   :               tref ->         t -> t   -> t ->   unit
  val gpowmod_in  : round_mode -> tref ->         t -> t   -> t ->   unit

  (* p-th root      ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val sqrt        :                               t ->               t
  val root        :                               t -> int ->        t
  val gsqrt       : round_mode ->                 t ->               t
  val groot       : round_mode ->                 t -> int ->        t
  val sqrt_in     :               tref ->         t ->               unit
  val root_in     :               tref ->         t -> int ->        unit
  val gsqrt_in    : round_mode -> tref ->         t ->               unit
  val groot_in    : round_mode -> tref ->         t -> int ->        unit

  (* factorial      ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val fact        :                             int ->               t
  val fact_in     :               tref ->       int ->               unit

  (* gcd            ---------------------------------------------------- *)
  (*                 d     u     v     p    q     a    b      c      res  *)
  val gcd         :                               t -> t   ->        t
  val gcd_ex      :                               t -> t   ->        t*t*t
  val cfrac       :                               t -> t   ->     t*t*t*t*t
  val gcd_in      : tref->                        t -> t   ->        unit
  val gcd_ex_in   : tref->tref->tref->            t -> t   ->        unit
  val cfrac_in    : tref->tref->tref->tref->tref->t -> t   ->        unit
```

```
(* comparison     ---------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val sgn         :                               t ->               int
val cmp         :                               t -> t    ->       int
val cmp_1       :                               t -> int ->        int
val eq          :                               t -> t    ->       bool
val eq_1        :                               t -> int ->        bool
val neq         :                               t -> t    ->       bool
val neq_1       :                               t -> int ->        bool
val inf         :                               t -> t    ->       bool
val inf_1       :                               t -> int ->        bool
val infeq       :                               t -> t    ->       bool
val infeq_1     :                               t -> int ->        bool
val sup         :                               t -> t    ->       bool
val sup_1       :                               t -> int ->        bool
val supeq       :                               t -> t    ->       bool
val supeq_1     :                               t -> int ->        bool

(* conversion     ---------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val of_int      :                             int ->               t
val of_string   :                          string ->               t
val of_int_in   :               tref ->       int ->               unit
val of_string_in:               tref ->    string ->               unit
val int_of      :                               t ->               int
val string_of   :                               t ->               string
val bstring_of  :                               t ->               string
val hstring_of  :                               t ->               string
val ostring_of  :                               t ->               string

(* random number  ---------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val nrandom     :                             int->                t
val zrandom     :                             int->                t
val nrandom1    :                             int->                t
val zrandom1    :                             int->                t
val nrandom_in  :               tref ->       int->                unit
val zrandom_in  :               tref ->       int->                unit
val nrandom1_in :               tref ->       int->                unit
val zrandom1_in :               tref ->       int->                unit
val random_init :                             int->                unit

(* binary representation  -------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val nbits       :                               t ->               int
val lowbits     :                               t ->               int
val highbits    :                               t ->               int
val nth_word    :                               t -> int ->        int
val nth_bit     :                               t -> int ->        bool
```

```
  (* shift          ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val shl         :                               t -> int ->        t
  val shr         :                               t -> int ->        t
  val split       :                               t -> int ->        t*t
  val join        :                               t -> t   -> int -> t
  val shl_in      :               tref ->         t -> int ->        unit
  val shr_in      :               tref ->         t -> int ->        unit
  val split_in    :               tref -> tref -> t -> int ->        unit
  val join_in     :               tref ->         t -> t   -> int -> unit

  (* display        ---------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val toplevel_print     :                        t ->               unit
  val toplevel_print_tref:        tref ->                            unit

  (* exceptions *)
  exception Error of string

end (* module type Int_type *)

                          (* +-------------------+
                             |   Infix notation  |
                             +-------------------+ *)

module Infixes(E : Int_type) : sig
  open E

  val (  ++  ) :   t -> t   -> t      (* add      *)
  val (  --  ) :   t -> t   -> t      (* sub      *)
  val (  **  ) :   t -> t   -> t      (* mul      *)
  val (  //  ) :   t -> t   -> t      (* div      *)
  val (  %%  ) :   t -> t   -> t      (* modulo   *)
  val (  /%  ) :   t -> t   -> t*t    (* quomod   *)
  val (  <<  ) :   t -> int -> t      (* shl      *)
  val (  >>  ) :   t -> int -> t      (* shr      *)
  val (  ^^  ) :   t -> int -> t      (* pow      *)

  val (  +=  ) : tref -> t -> unit    (* add_in   *)
  val (  -=  ) : tref -> t -> unit    (* sub_in   *)
  val (  *=  ) : tref -> t -> unit    (* mul_in   *)
  val (  /=  ) : tref -> t -> unit    (* quo_in   *)
  val (  %=  ) : tref -> t -> unit    (* mod_in   *)

  val (  +.  ) :   t -> int -> t      (* add_1    *)
  val (  -.  ) :   t -> int -> t      (* sub_1    *)
  val (  *.  ) :   t -> int -> t      (* mul_1    *)
  val (  /.  ) :   t -> int -> t      (* quo_1    *)
  val (  %.  ) :   t -> int -> int    (* mod_1    *)
  val (  /%. ) :   t -> int -> t*int  (* quomod_1 *)
  val (  ^.  ) : int -> int -> t      (* pow_1    *)

  val (  +=. ) : tref -> int -> unit  (* add_1_in *)
  val (  -=. ) : tref -> int -> unit  (* sub_1_in *)
  val (  *=. ) : tref -> int -> unit  (* mul_1_in *)
  val (  /=. ) : tref -> int -> unit  (* quo_1_in *)

  val (   =. ) :   t -> int -> bool   (* eq_1     *)
  val (  <>. ) :   t -> int -> bool   (* neq_1    *)
  val (   <. ) :   t -> int -> bool   (* inf_1    *)
  val (  <=. ) :   t -> int -> bool   (* infeq_1  *)
  val (   >. ) :   t -> int -> bool   (* sup_1    *)
  val (  >=. ) :   t -> int -> bool   (* supeq_1  *)

  val (  ~~  ) : tref -> t            (* look     *)

end (* Infixes functor *)

                        (* +-----------------------+
                           |   Available modules   |
                           +-----------------------+ *)

(* All the following modules implement the Int_type signature.
   A module may be missing on a particular computer when the hardware
   or software available does not permit the compilation of this module.
                                                                      *)

module Big   : Int_type
module Clong : sig ... end   (* concrete descriptions *)
module Dlong : sig ... end   (* conforming to the     *)
module Slong : sig ... end   (* Int_type signature    *)
module Gmp   : sig ... end

(* comparison between two modules *)
module Cmp(A:Int_type)(B:Int_type) : Int_type

(* statistics *)
module Count(A:Int_type) : sig

  type statelt = {
    mutable n:float;  (* number of calls *)
    mutable s:float;  (* sum of sizes    *)
    mutable m:int     (* maximal size    *)
  }

  val cadd  : statelt (* add      sub                            *)
  val cmul  : statelt (* mul      sqr                            *)
  val cquo  : statelt (* quo      modulo       quomod            *)
  val cpow  : statelt (* pow      powmod       fact              *)
  val croot : statelt (* sqrt     root                           *)
  val cgcd  : statelt (* gcd      gcd_ex      cfrac              *)
  val cbin  : statelt (* shr      shl         split      join    *)
                      (* nbits    lowbits     highbits   nth_bit *)
                      (* nth_word random                          *)
  val cmisc : statelt (* abs      neg         make_ref   copy_in *)
                      (* copy_out comparisons conversions         *)

  val clear_stats : unit -> unit (* reset counters *)
  val print_stats : unit -> unit (* print counters *)

  include Int_type

end (* Count functor *)

      (* +----------------------------------------------------+
         |  Approximation of the usual real-valued functions  |
         +----------------------------------------------------+ *)

module Rfuns(E:Int_type) : sig

  exception Error of string

  (* [f a b n] returns x such that |2^n*f(a/b) - x| < 1 *)
  val arccos    : E.t -> E.t -> int -> E.t
  val arccosh   : E.t -> E.t -> int -> E.t
  val arccot    : E.t -> E.t -> int -> E.t
  val arccoth   : E.t -> E.t -> int -> E.t
  val arcsin    : E.t -> E.t -> int -> E.t
  val arcsinh   : E.t -> E.t -> int -> E.t
  val arctan    : E.t -> E.t -> int -> E.t
  val arctanh   : E.t -> E.t -> int -> E.t
  val arg       : E.t -> E.t -> int -> E.t
  val cos       : E.t -> E.t -> int -> E.t
  val cosh      : E.t -> E.t -> int -> E.t
  val cosin     : E.t -> E.t -> int -> E.t*E.t
  val cosinh    : E.t -> E.t -> int -> E.t*E.t
  val cot       : E.t -> E.t -> int -> E.t
  val coth      : E.t -> E.t -> int -> E.t
  val exp       : E.t -> E.t -> int -> E.t
  val ln        : E.t -> E.t -> int -> E.t
  val sin       : E.t -> E.t -> int -> E.t
  val sinh      : E.t -> E.t -> int -> E.t
  val tan       : E.t -> E.t -> int -> E.t
  val tanh      : E.t -> E.t -> int -> E.t

  (* [r_f r a b c] returns the integer approximating c*f(a/b) according
                   to round mode r *)
  val r_arccos  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccosh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccot  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccoth : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arcsin  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arcsinh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arctan  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arctanh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arg     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cos     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cosh    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cosin   : round_mode -> E.t -> E.t -> E.t -> E.t*E.t
  val r_cosinh  : round_mode -> E.t -> E.t -> E.t -> E.t*E.t
  val r_cot     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_coth    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_exp     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_ln      : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_sin     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_sinh    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_tan     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_tanh    : round_mode -> E.t -> E.t -> E.t -> E.t

  (* creation of a r_function *)
  val round : (int -> E.t) -> round_mode -> E.t -> E.t

  (* cache management *)
  val cache_bits  : unit -> int
  val clear_cache : unit -> unit

end (* Rfuns functor *)

                      (* +-----------------------+
                         |  Run-time selection   |
                         +-----------------------+ *)

module type Main_type = sig
  val main : string list -> unit
end

module Start(Main : functor(E:Int_type) -> Main_type) : sig
  val start : unit -> unit
end

                           (* +-----------+
                              |  Timing   |
                              +-----------+ *)

external chrono : string -> unit = "chrono"
```

2.2 The `Int_type` signature

2.2.1 Integers and references

An implementation compatible with the `Int_type` signature provides two datatypes:

- The `t` datatype represents a signed integer. The bit length of such an integer is limited only by the amount of available memory, and in the case of the `Clong`, `Dlong`, `Slong` and `Big` modules, by the maximal size of an Ocaml data (2^{27} bits on a 32 bit computer, 2^{60} bits on a 64 bit computer).

- The `tref` datatype represents a mutable and extensible memory block containing a value of type `t`. This memory block is enlarged on a double the size policy when its current capacity is too short for the data to be stored into. A `tref` memory block in never shrunk.

One creates a reference of type `tref` with the `make_ref` function which makes a physical copy of its argument and returns the pointer to the memory block allocated to the copy. One stores a new integer into a reference of type `tref` with the `xxx_in` functions which do the computation designated by `xxx` and store the result into the `tref` argument given to `xxx_in`. When a `xxx` function computes several results of type `t`, the `xxx_in` associated function receives as additional arguments as many references as there are results to be stored; these arguments must designate distinct memory blocks.

On can retrieve the integer of type `t` stored into a reference of type `tref` with the `copy_out` and `look` functions:

- `copy_out` makes a physical copy of the integer to be retrieved and returns a pointer to this copy. Any subsequent action on the reference is without any effect on the copy returned.

- `look` makes no copy and returns a pointer to the memory block associated with the reference. The integer returned by `look r` is volatile, that is to say that its value may change when a new integer is stored into the `r` reference (the value may also not change if the store results in the reallocation of the memory block).

The user is advised to use `look` only in intermediate computations when he wants to avoid a copy for performance reasons. Read-modify-write operations, for instance `xxx_in r (look r) z`, are handled correctly.

2.2.2 Various versions of an operation

As a general rule an operation between big integers is available in four versions:

- `xxx : t -> t -> t` : computes a result of type `t` from two operands of type `t`.

- `xxx_1 : t -> int -> t` : computes a result of type `t` from an operand of type `t` and an operand of type `int`. `xxx_1` a b is formally equivalent to `xxx_1` a (`of_int` b), but is in general implemented more efficiently, so as to avoid the intermediate result allocation overhead and to reduce the Ocaml garbage collector work.

- `xxx_in : tref -> t -> t -> unit` : computes a result of type `t` from two operands of type `t`, and stores this result into the memory block designated by the reference of type `tref`. `xxx_in` r a b if formally equivalent to `copy_in` r (`xxx` a b), but in general the result is computed directly into the memory block designated by r, so as to avoid the result allocation and the copy overhead, and to reduce the Ocaml garbage collector work.

- `xxx_1_in : tref -> t -> int -> unit` : computes a result of type `t` from two operands of type `t` and `int`, and stores this result into the memory block designated by the operand of type `tref`. `xxx_1_in` r a b is formally equivalent to `copy_in` r (`xxx` a (`of_int` b)), with the same overhead reductions as `xxx_1` and `xxx_in`.

2.2.3 Rounding mode

Operations returning an integer approximation of a real number a (division, square and p-th root) are available in five versions:

`xxx`	*args*	computes $\lfloor a \rfloor$
`gxxx Floor`	*args*	computes $\lfloor a \rfloor$
`gxxx Ceil`	*args*	computes $\lceil a \rceil$
`gxxx Nearest_up`	*args*	computes $\lfloor a + 1/2 \rfloor$
`gxxx Nearest_down`	*args*	computes $\lceil a - 1/2 \rceil$

Note that the `Nearest_up` and `Nearest_down` rounding modes return different results only when $a = k + \frac{1}{2}$ for some integer k: `Nearest_up` returns $k + 1$ whereas `Nearest_down` returns k.

2.2.4 Arithmetic operations

The table below shows the mathematical descriptions of the arithmetic operations implemented in a module compatible with the `Int_type` signature. The letters a, b, c denote values of type `t` or `int`, the letter n denotes an operand of type `int`. When a `xxx` operation returns several results, the `xxx_in` associated operation stores the results into the references received as additional arguments with the same ordering.

operation	arguments			results
`add`	a	b		$a+b$
`sub`	a	b		$a-b$
`mul`	a	b		ab
`quomod`	a	b		$(\lfloor a/b \rfloor, a - \lfloor a/b \rfloor b)$
`quo`	a	b		$\lfloor a/b \rfloor$
`modulo`	a	b		$a - \lfloor a/b \rfloor b$
`abs`	a			$\lvert a \rvert$
`neg`	a			$-a$
`sqr`	a			a^2
`pow`	a	n		a^n
`powmod`	a	b	c	$a^b - \lfloor a^b/c \rfloor c$
`sqrt`	a			$\lfloor \sqrt{a} \rfloor$
`root`	a	n		$\lfloor \sqrt[n]{a} \rfloor$
`fact`	n			$n!$
`gcd`	a	b		d
`gcd_ex`	a	b		(d, u, v)
`cfrac`	a	b		(d, u, v, p, q)

`cfrac` a b returns a (d, u, v, p, q) tuple such that d is the non negative gcd of a and b, $ua - vb = d$, $pu - qv = 1$, $pa = qb$ is the lcm of a and b with the sign of ab. These conditions are sufficient for ensuring the uniqueness of p, q, d when a or b is non null, but the u and v coefficients are not unique and may differ for each of the `Int_type` signature implementations. `gcd_ex` a b returns the (d, u, v) tuple, `gcd` a b returns d. Note that the result ordering has been reversed with respect to `Numerix`-0.19, for compatibility with the C and Pascal interfaces.

2.2.5 Comparisons

`sgn` a returns 1 if $a > 0$, 0 if $a = 0$ and -1 if $a < 0$. `cmp` a b is formally equivalent to `sgn`$(a - b)$, but the subtraction is not really done: a and b are compared bit for bit starting with the most significant ones until the sign of the difference can be determined.

The boolean valued comparison operations are available with the names shown in the `Int_type` signature. The `Clong`, `Dlong`, `Slong` and `GMP` modules also enable one to compare two values of type `t` with the polymorphic infix comparison symbols of Ocaml:

`eq`	`=`	`inf`	`<`	`sup`	`>`
`neq`	`<>`	`infeq`	`<=`	`supeq`	`>=`

The `Big` module does not provide polymorphic comparison operations, therefore one can use only the `eq`, . . . , `supeq` prefix names with this module. Note that the optional patch included in `Numerix`-0.19 to implement polymorphic comparisons for the `Big` module has been removed in `Numerix`-0.21. This patch was returning wrong results when comparing two negative integers and there is no possibility to work around this problem with the current implementation of `Big_int`.

2.2.6 Conversions

`of_int` converts a value of type `int` into the corresponding value of type `t`. `int_of` makes the inverse conversion when the integer to be converted has an absolute value not greater than 2^{30}, otherwise an exception is raised. Note that the limit 2^{30} is independent of the machine word size.

`of_string` s returns the integer of type `t` represented by the s string with respect to the following syntax:

- An optional leading `+` or `-` sign.
- A `0x`, `0X`, `0o`, `0O`, `0b` or `0B` prefix after the optional sign, specifying base 16, 8 or 2. Base 10 is used when there is no such prefix.
- A non empty digit sequence, valid for the base specified, with no space and no underscore. When base 16 is used the letters `a,b,c,d,e,f` and `A,B,C,D,E,F` are accepted.

The conversion of a value of type `t` into a string is done with one of the following functions: `string_of` (base 10), `hstring_of` (base 16), `ostring_of` (base 8), `bstring_of` (base 2). The returned string is compatible with the syntax of `of_string`. This enables one to convert a value a of type `A.t` into the corresponding value b of type `B.t`, `A` and `B` denoting two implementations compatible with the `Int_type` signature, with the instruction:

```
let b = B.of_string(A.hstring_of a)
```

The programmer is advised to use base 16 conversion for this purpose, because it is the conversion that returns the shortest string and its complexity is linear in the bit size of a. Note that this method of conversion does not work when the hexadecimal representation of a exceeds the maximal size of an Ocaml string (that is to say $|a| \geq 16^{2^{24}-4}$ on a 32 bit computer and $|a| \geq 16^{2^{57}-4}$ on a 64 bit computer). In such a case, `A.hstring_of` a returns the `"<very long number>"` string and this string will be rejected by `B.of_string`.

The `toplevel_print` and `toplevel_print_tref` functions convert a value of type `t` or `tref` into its decimal string representation and display the string with the `Format` module printing functions. When the string to be displayed has more than 1000 characters, only the first 200 ones are displayed followed by the number of characters removed and followed by the 200 last characters.

2.2.7 Pseudo-random numbers

The `nrandom`, `nrandom1`, `zrandom` and `zrandom1` functions return n bit pseudo-random integers where n is a non negative argument. The result returned by `nrandom` and `nrandom1` is non negative and not greater than $2^n - 1$. The result returned by `zrandom` and `zrandom1` is not smaller than $-2^n + 1$ and not greater than $2^n - 1$. The result returned by `nrandom1` and `zrandom1` has its n-th bit set, that is to say that its absolute value is not smaller than 2^{n-1}. Note that the `random_bits` function from `Numerix`-0.19 has been replaced by `nrandom` in `Numerix`-0.21.

The pseudo-random generator used depends on the module and the computer. Therefore the results obtained by a program using these functions is not reproducible from one (module,computer) pair to another one. The `random_init` function enables one to initialize the pseudo-random generator of the module and the one of Ocaml from a seed of type `int`. When this seed is null, it is replaced by the date, expressed in seconds, at which the `random_init` function is called. The sequence obtained from a non null seed is reproducible for a given (module,computer) pair. One only has to reinitialize the pseudo-random generator with the same seed.

2.2.8 Access to the binary representation

If a and b denote values of type `t` then:

- `nbits` a returns the number of bits of $|a|$, that is to say 0 when $a = 0$ and $\lceil \log_2 |a| \rceil$ when $a \neq 0$.

- `lowbits` a returns the 31 least significant bits of $|a|$, that is to say $|a| \bmod 2^{31}$.

- `highbits` a returns the 31 most significant bits of $|a|$, that is to say $\lfloor |a|/2^{31-\texttt{nbits}(a)} \rfloor$. Note that when $a \neq 0$ the number returned is considered as a negative number by Ocaml on a 32 bit computer.

- `nth_word` a returns the number formed from the bits of $|a|$ with rank between $16n$ and $16n + 15$, that is to say $\lfloor |a|/2^{16n} \rfloor \bmod 2^{16}$. If $n < 0$ or $n > \texttt{nbits}(a)/16$, the result is null.

- `nth_bit` a returns the n-th bit of $|a|$, that is to say `true` if $\lfloor |a|/2^n \rfloor$ is an odd number, and `false` otherwise. If $n < 0$ or $n > \texttt{nbits}(a)$, the result is `false`.

- `shl` a n returns the number having same sign as a and formed by left shifting $|a|$ by n bits when $n \geq 0$ or right shifting $|a|$ by $-n$ bits when $n < 0$, that is to say $\texttt{sgn}(a)\lfloor 2^n|a| \rfloor$ in both cases.

- `shr` a n returns the number having same sign as a and formed by right shifting $|a|$ by n bits when $n \geq 0$ or left shifting $|a|$ by $-n$ bits when $n < 0$, that is to say $\texttt{sgn}(a)\lfloor |a|/2^n \rfloor$ in both cases.

- `split` a n returns the (q, r) pair such that $|q| = \lfloor |a|/2^n \rfloor$, $|r| = |a| \bmod 2^n$, $qa \geq 0$ and $ra \geq 0$. n must be a non negative integer.

- `join` a b n returns the number $a + 2^n b$, n must be non negative.

2.2.9 Hashing, serialization and de-serialization

The `Clong`, `Dlong`, `Slong` and `GMP` modules come with interfaces with the generic hashing function of Ocaml. The hash key of a big integer from one of these modules is computed from the internal representation of the number, therefore it may depend on the module used. The `Big_int` module comes with a minimal interface with the generic hashing function: only the sign of a number is taken into account when computing the hash key. Therefore, big integers from these five modules can be stored into hash tables using the `Hashtbl.hash` function. Note that the collision ratio will be high when the `Big` module is used.

Concerning serialization and de-serialization, all the five modules come with interfaces with the serialization and de-serialization functions of Ocaml. Therefore the big integers from these modules can be exported or imported with the `output_value` and `input_value` functions and can be converted into and from byte sequences with the functions of the `Marshal` module. Note that the typing must be preserved between the exportation or the conversion into a byte sequence and the importation or the conversion from a byte sequence. In other words, it is impossible to convert a big integer from one module into a big integer from another module with these functions.

2.2.10 Errors

The `Clong`, `Dlong`, `Slong` and `GMP` modules check the validity of the arguments of their functions and raise in case of an invalid argument one of the following `Error` *msg* exceptions:

msg	reason
`integer overflow`	`int_of` a with $\|a\| \geq 2^{30}$
`invalid string`	`of_string` with an invalid string
`multiple result`	`xxx_in` with several identical references
`negative base`	`fact` n with $n < 0$, `sqrt` a with $a < 0$, `root` a n with $a < 0$ and n even
`negative exponent`	`pow` and `powmod` when the exponent is negative `root` when the exponent is not positive
`negative index`	`split`, `join` with $n < 0$
`negative size`	`xrandom`, `xrandom1` with $n < 0$
`number too big`	the result is too big to be stored in an Ocaml value
`division by zero`	`quoxxx`, `modxxx`, `powmod` when the divisor is null

Concerning the `Big` module, an invalid argument may raise an exception at the `Numerix` interface layer, or from within the Ocaml `Big_int` module. In the first case, the exception raised is the appropriate one according to the table above; in the last case, the exception raised is a `Big_int` specific one not listed in the table above.

The C `Numerix` kernel which implements the `Clong`, `Dlong` and `Slong` modules may raise one of the following uncatchable exceptions:

`"Numerix kernel: out of memory"`: a computation cannot be finished because there is not enough available memory.

`"Numerix kernel: number too big"`: a computation cannot be finished because it needs a too big intermediate result.

`"Numerix kernel: xxx"`: The C code detected an internal `Numerix` bug. This should not happen in the user version of `Numerix` because the internal bug checks are deactivated by default. If you encounter such an error, please let me know.

2.3 The functors using the `Int_type` signature

2.3.1 Infix symbols

The `Infixes` functor receives as argument a module compatible with the `Int_type` signature and defines infix equivalents for the most common operations of this module. The infix operations between a reference of type `tref` and a value of type `t` or `int` follow the C syntax. For instance `r -= a` should be read as `sub_in r (look r) a`. Note that the `Infixes` functor from `Numerix`-0.21 is incompatible with the one from `Numerix`-0.19: the later was overloading the usual operations between values of type `int`, resulting in an uneasy use of this functor.

2.3.2 Comparison between two modules

The `Cmp` functor receives as arguments two modules `A` and `B` compatible with the `Int_type` signature and returns a `C` module compatible with this signature. In `C` each operation *op* is done with a call to `A.`*op* and `B.`*op* followed by a semantical comparison of the results. When a comparison fails, that is to say when `A.`*op* and `B.`*op* return semantically different results while their arguments are supposed semantically identical, an exception is raised showing in a textual form the function called and the arguments and results from both modules. This functor was used to debug the modules being developed by comparing them with a reliable module. The use of this functor in other situations is not recommended because doing twice the computations and comparing the results takes a lot of time. Concerning the `gcd_ex`, `gcd_ex_in`, `cfrac` and `cfrac_in` operations, the Bézout coefficients are not compared, those returned by `A` are converted into values of type `B.t` to build results of type `C.t`. Also, the pseudo-random generator of `C` is built from the one of `A` only.

2.3.3 Statistics

The `Count` functor receives as argument a module `A` compatible with the `Int_type` signature and returns a module `B` compatible with this signature in which each operation *op* is done with a call to `A.`*op* and with an update of a counter depending on the operation. The purpose of this functor is to provide statistics on the number of big integer operations done in a program. These operations are merged into eight categories, each category being associated with a different counter:

`cadd`	counts additions and subtractions;
`cmul`	counts multiplications and squares;
`cquo`	counts divisions;
`cpow`	counts exponentiations and factorials;
`croot`	counts square roots and p-th roots;
`cgcd`	counts greatest common divisor and associated operations;
`cbin`	counts operations on the binary representations;
`cmisc`	counts all other operations except `look`, `random_init`, `toplevel_print` and `toplevel_print_tref`.

Each `cxxx` counter has three fields:

`cxxx.n`	number of calls to one of the functions associated with `cxxx`;
`cxxx.s`	sum of the bit sizes of the arguments for all calls;
`cxxx.m`	maximum of the bit sizes of the arguments for all calls.

For each call to a function of `B`, field `n` of the associated counter is increased by one, field `s` is increased by the average bit size of big integer operands and field `m` is updated so as to hold the maximum bit size of an operand for any function associated with this counter. Operands of type `tref`, `int` or `string` are not taken into account in size computations.

One can read the values of the counters and modify them at will, so as to determine how many operations of each category have been done since the last reset. The `clear_stats` function resets all counters and the `print_stats` function displays the statistics relative to each counter (number of calls, average operand bit size, maximum operand bit size).

2.3.4 Approximation of the usual functions

The `Rfuns` functor receives as argument a module `E` compatible with the `Int_type` signature and returns a module implementing approximation algorithms for the usual mathematical functions:

arccos	arccosh	arccot	arccoth	arcsin	arcsinh
arctan	arctanh	cos	cosh	cot	coth
exp	ln	sin	sinh	tan	tanh

Appart from arccot, the mathematical definitions of all the functions above are supposed to be well known and without ambiguity. The arccot function implemented in `Numerix` is mathematically defined by:

$$(\operatorname{arccot} x = \theta) \iff (\cot\theta = x \text{ and } 0 < \theta < \pi).$$

Most mathematical software use another definition with $-\pi/2 < \theta < \pi/2$, but this results in an artificial discontinuity at 0 and I do consider my own definition as better.

f denoting one of the functions above, two interfaces to the approximating algorithm of f are available:

```
f   :               E.t -> E.t -> int -> E.t
r_f : round_mode -> E.t -> E.t -> E.t -> E.t
```

`f` a b n returns an integer x such that $x-1 < 2^n f(a/b) < x+1$, that is to say one of the two numbers $x_1 = \lfloor 2^n f(a/b)\rfloor$ and $x_2 = \lceil 2^n f(a/b)\rceil$. Negative values for n are accepted. The integers a and b may not be both null, and a/b must belong to the domain of f. The quotient $a/0$ is considered as being equal to $+\infty$ or $-\infty$ depending on the sign of a, it is accepted when f has a finite limit at this point. When $x_1 \neq x_2$, one cannot tell which of x_1 or x_2 will be returned: this depends on a and b as well as on the state of the cache used by the algorithm approximating f.

`r_f Floor`	a b c returns $\lfloor cf(a/b)\rfloor$,
`r_f Ceil`	a b c returns $\lceil cf(a/b)\rceil$,
`r_f Nearest_up`	a b c returns $\lfloor cf(a/b) + 1/2\rfloor$,
`r_f Nearest_down`	a b c returns $\lceil cf(a/b) - 1/2\rceil$.

The a and b arguments must obey the same rules as for `f`. The value to be returned is defined in a unique way, therefore it does not depend on the state of the cache used by the algorithm approximating f. Note that the `Nearest_up` and `Nearest_down` rounding modes are equivalent with the functions f available because $cf(a/b)$ cannot be equal to $k+\frac{1}{2}$ for some integer k with these functions.

From a performance viewpoint, one is advised to use the first interface (function `f`), because apart from the `cot` and `tan` functions, the computation of `f` a b n has complexity $O(M(k)\ln k)$ where $M(k)$ denotes the complexity of a multiplication of two integers the product of which fits into k bits, and

$$k = \max(\texttt{nbits}(a), \texttt{nbits}(b), \texttt{nbits}(\lfloor 2^n f(a/b)\rfloor)),$$

whereas the computation of `r_f` r a b c has an unbounded complexity (the algorithm implementing `r_f` consists in computing `f` a b n with increasing values for n until having a result suitable for determining the correct rounding of $cf(a/b)$). The complexities of `cot` and `tan` are unbounded for the same reason: one may have to compute arbitrary precise values of $\cos(a/b)$ and $\sin(a/b)$ when a/b is close to a multiple of $\pi/2$.

The following functions are also available with both interfaces:

arg:	$(\arg(x,y) = \theta) \iff (x+iy = e^{i\theta}\sqrt{x^2+y^2}$ and $-\pi < \theta \leq \pi)$.
cosin:	$\operatorname{cosin} x = (\cos x, \sin x)$,
cosinh:	$\operatorname{cosinh} x = (\cosh x, \sinh x)$,

Formally, `cosin` a b n returns the (`cos` a b n, `sin` a b n) pair, and one is advised to use the `cosin` function rather that to call `cos` and `sin` separately when one wants approximations for the cosine and the sine of a same angle. Similar advices hold for the `r_cosin`, `cosinh` and `r_cosinh` functions.

Concerning the `arg` and `r_arg` functions, their use is to be preferred to the use of `arccos`, `arcsin`, `r_arccos` and `r_arcsin` because these four functions are actually implemented with a call to `arg` or `r_arg` after the computation of a potentially expensive square root. The `arctan`, `r_arctan`, `arccot` and `r_arccot` functions also call `arg` or `r_arg`, but they don't make any preliminary expensive computation, therefore their use is not inefficient.

The iterative precision increase mechanism implemented into the `r_xxx` functions is available for the user with the `round` function: let t be an irrational real number and `f : int -> E.t` a function approximating t such that for all integer n, `f` n returns an integer x such that $x-1 < 2^n t < x+1$. Then `round f` returns a function `r_f : round_mode -> E.t -> E.t` such that `r_f` r c returns the integer approximating ct with respect to the rounding mode r. Note that the computation of `r_f` r c cannot loop, even when t is rational. In the worst case, the computation will end with an error because of insufficient memory or because of a number too big to be computed.

The approximating algorithms implemented in the `Rfuns` functor use a cache memory where the approximations of some frequently used constants are stored. When one of these approximations happens to be insufficient for the current computation, a new approximation with a suitable precision is computed and this new approximation replaces the old one in the cache memory. The constants stored in the cache memory have been chosen so as to be able to retrieve at low cost (a few additions and a shift) the approximations for the following numbers:

$\ln(2)$	$\exp(1)$	$\arctan(1)$
$\ln(3)$	$\exp(-1)$	$\arctan(1/2)$
$\ln(5)$	$\cos(1)$	$\arctan(1/3)$
	$\sin(1)$	$\arctan(1/5)$

The `cache_bits` function returns the sum of the bit sizes of the approximations currently stored in the cache memory: the total size of the cache memory is approximately twice the number returned by `cache_bits`. The `clear_cache` function restores the initial approximations with 100 bit precision, so as to enable the memory manager of Ocaml to reclaim the memory used by the cache.

The use of this cache memory results in a speedup of the computations, but it has the drawback of making not reproducible any computation of the form `f` a b n: the return value may vary depending on the precision with which the constants used by `f` are known. However, the cache management mechanism is designed so as to grant coherence with the past: if a computation `f` a b n returned once a value x then any subsequent computation with the same arguments will return the same value x, even if the precision of the cached constants was increased meanwhile. Of course, the coherence with the past warranty will cease as soon as one resets the cache memory by calling the `clear_cache` function.

Functions from the `Rfuns(E)` module may raise in case of trouble the following `Rfuns(E).Error` *msg* exceptions:

msg	reason
`0/0`	$a = b = 0$
`number too big`	see below
`arcsinh`	
`cos`	
`cosin`	$a/b = \pm\infty$
`sin`	
`tan`	
`arccos`	$\|a/b\| > 1$
`arcsin`	
`arccosh`	$a/b = +\infty$ or $a/b < 1$
`arctanh`	$\|a/b\| \geq 1$
`cot`	$a/b = 0$ or $a/b = \pm\infty$
`coth`	$a/b = 0$
`exp`	$a/b = +\infty$
`ln`	$a/b \leq 0$ or $a/b = +\infty$

When the computation of an intermediate result is impossible because this intermediate result is too big, one of the following exceptions is raised:

- `Rfuns(E).Error "number too big"` : the impossibility was detected by a function from `Rfuns`.

- `E.Error "number too big"` : the impossibility was detected by a function from `E`.

- `"Numerix kernel: number too big"` : the impossibility was detected by the `Numerix` C kernel. In this last case, the exception cannot be caught.

2.3.5 Run-time selection of a module

The `Start` functor enables one to select at run-time which big integer implementation to use. The argument of `Start` is a functor `Main` receiving as argument a big integer implementation compatible with the `Int_type` signature and providing an implementation of the function `main : string list -> unit` which constitutes the entry point of the program.

`Start(Main).start` parses the command line, selects a module `E` compatible with the `Int_type` signature from the `-e xxx` and `-count` options found and then calls `Main(E).main` with the list of the remaining command line parameters as argument. The command line parameter number zero, which generally denotes the program name is included in this list (it wasn't with `Numerix`-0.19).

The `-e xxx` option selects a module among `Clong`, `Dlong`, `Slong`, `GMP`, `Big` where `xxx` is the lowercase name of this module. When several `-e xxx` options are found on the command line, only the last two ones are taken into account and they select the `Cmp(A)(B)` module, `A` being the module designated by the last-but-one option and `B` the module designated by the last option. When no

`-e xxx` option is found on the command line, the module selected is the first one available in the list `Clong`, `Dlong`, `Slong`, `GMP`, `Big`.

The `-count` option selects the `Count(E)` module where `E` is the module selected by the `-e xxx` options.

2.3.6 Timing

`chrono` *msg* prints on the standard output stream the CPU time in seconds since the beginning of the process, the difference with the previous time and the *msg* string. The inclusion of a few calls to `chrono` within a program informs the user of the approximate running times of the various phases in this program. Note that `Numerix`-0.19 used the standard error stream for this purpose; with `Numerix`-0.21 all outputs, including the internal error messages, are done on the standard output stream.

2.4 Use

2.4.1 Compilation

The Ocaml programs using `Numerix` must be compiled with the following commands:

```
ocamlc   options nums.cma  numerix.cma  source files
ocamlopt options nums.cmxa numerix.cmxa source files
```

The `nums.cma`, `nums.cmxa`, `numerix.cma` and `numerix.cmxa` files contain in a compiled form the `Big_int` and `Numerix` libraries. One may have to tell the compilers where to search for these files with the help of a `-I` *path* option.

2.4.2 Example

```
(* file simple.ml: simple demo of Numerix
   compute (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)

open Numerix
module Main(E:Int_type) = struct
  module I = Infixes(E)
  open E
  open I

  let main arglist =

    let n = match arglist with
    | _::"-n"::x::_ -> int_of_string x
    | _             -> 30
    in

    (* d <- 10^n, d2 <- 10^(2n) *)
    let d  = (5 ^. n) << n in
    let d2 = sqr d         in

    (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2))) *)
    let a = gsqrt Nearest_up (d2 *. 200) in
    let b = gsqrt Nearest_up (d2 *. 300) in

    (* r <- round(10^n*(b+a)/(b-a)) *)
    let r = gquo Nearest_up (d**(b++a)) (b--a) in
    Printf.printf "r=%s\n" (string_of r);
    flush stdout

end
let _ = let module S = Start(Main) in S.start()
```

Compilation and execution:

```
> ocamlc -I ~/lib -o simple-byte nums.cma numerix.cma simple.ml
> ./simple-byte -e slong
r=989897948556635619639456814941
> ocamlopt -I ~/lib -o simple-opt nums.cmxa numerix.cmxa simple.ml
> ./simple-opt -e gmp -n 50 -count
r=98989794855663561963945681494117827839318949613133З
   op      count   avg.size   max.size
  add          2        170        171
  mul          4        250        333
  quo          1        253        338
  pow          1          0          0
 root          2        340        341
  gcd          0          -          -
  bin          1        117        117
 misc          1        170        170
>
```

2.4.3 Toplevel

ocamlnumx is a customized Ocaml toplevel linked with the numerix.cma and nums.cma object files. It enables one to use all the Numerix modules, the choice of a big integer implementation being done through an appropriate open directive.

```
> ocamlnumx
ocamlnumx : Ocaml toplevel with big integer libraries
Numerix submodules : Clong Dlong Slong Big Gmp
Numerix version    : 0.21

        Objective Caml version 3.08.0

# open Numerix;;
# module I = Infixes(Slong);; (* output deleted *)
# module R = Rfuns(Slong);;   (* output deleted *)
# open Slong open I open R;;
# let a = r_exp Floor one one (10^.50);;
val a : Numerix.Slong.t = 271828182845904523536028747135266249775724709369995
# #quit;;
>
```

If your Ocaml version supports loadable modules then it is also possible to use the standard Ocaml toplevel by loading manually the `nums.cma` and `numerix.cma` files. Note that in this case it may be necessary to tell `ocaml` where to find the `numerix.cma` file with a `-I` *path* option. Note also that the `toplevel_print` and `toplevel_print_tref` functions must be manually activated with the `#install_ printer` directive.

```
> ocaml -I ~/lib
        Objective Caml version 3.08.0

# #load "nums.cma";;
# #load "numerix.cma";;
# open Numerix;;
# module I = Infixes(Slong);; (* output deleted *)
# module R = Rfuns(Slong);;   (* output deleted *)
# open Slong open I open R;;
# let a = r_exp Floor one one (10^.50);;
val a : Numerix.Slong.t = <abstr>
# #install_printer toplevel_print;;
# a;;
- : Numerix.Slong.t = 271828182845904523536028747135266249775724709369995
# #quit;;
>
```

Chapter 3

Use with Camllight

Contents

The `Numerix` Camllight interface was derived from the Ocaml one by removing or adapting the functionalities specific to the Ocaml language. Please refer to the previous chapter to see the list of available functions, only the differences with the Ocaml version are mentioned here. This interface was successfully tested with Camllight-0.74 and Camllight-0.75.

3.1 Interface

3.1.1 Modules

Camllight has a limited module system and provides neither sub-modules nor functors. However, it is possible to write implementation independent code by using the short functions names, the long ones are inferred by the compiler with the help of `#open` directives in the source file. One only needs to modify these directives (possibly in an automatic way with a preprocessor) and to recompile the source code in order to change the big integer implementation used.

The available modules have the same names as those of Ocaml in lowercase: `clong`, `dlong`, `slong`, `gmp` and `big`. There is no equivalent to the modules built with the Ocaml functors `Cmp`, `Count` and `Rfuns`. The infix notations are available by opening the `infxxx` module where `xxx` is the name of the module implementing big integers.

3.1.2 Functions

The functions described in the `Int_type` Ocaml signature are available with Camllight with only three differences:

- The division without remainder is named `quo` in Ocaml and `div` in Camllight. The reason for this difference is that the `quo` identifier has an infix status in Camllight. The other names derived from `quo`: `quomod`, `quo_1`, `gquo`, etc. are the same as those in Ocaml.

- Accessing the value held by a reference is written `look` or `~~` in Ocaml, whereas it is written `look` or `?` in Camllight. There are two reasons for this difference: the `?` identifier is reserved in Ocaml and the `~~` identifier has a prefix status in Ocaml and an infix one in Camllight.

- The run-time errors raise an `Error` *msg* exception in Ocaml and a `Failure "Numerix kernel:` *msg*`"` exception in Camllight. This is a result of the impossibility in Camllight to raise any exception except `Failure` and `Invalid_argument` from within a C function.

3.2 Use

3.2.1 Compilation

The Caml programs using `Numerix` must be compiled with the following command:

```
camlc -custom options nums.zo numerix.zo source files \
      -lnumerix-caml -lnums -lgmp
```

The `nums.zo` and `numerix.zo` files contain in a compiled form the Caml part of the `Big_int` and `Numerix` libraries. It may be necessary to tell the compiler where to find the `numerix.zo` file with a `-I` *path* option.

The `-lnumerix-caml`, `-lnums` and `-lgmp` options ask the linker to look for the required C primitives in the `libnumerix-caml`, `libnums` and `libgmp` libraries. It may be necessary to tell the linker where to find these libraries with `-ccopt -L`*path* options. If `GMP` is not installed or if its Camllight interface is not included in `Numerix`, Then the `-lgmp` option must be omitted. Similarly, the `nums.zo` and `-lnums` parameters must be omitted if the `big` module is not included in `Numerix`. Note that the `libclnumx` library used with `Numerix`-0.19 has been renamed `libnumerix-caml` in `Numerix`-0.21, the latter name was judged more informative.

3.2.2 Example

```
(* file simple.ml: simple demo of Numerix
   compute (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)
```

```
#open "clong";;
#open "infclong";;

  let main arglist =

    let n = match arglist with
    | _::"-n"::x::_ -> int_of_string x
    | _             -> 30
    in

    (* d <- 10^n, d2 <- 10^(2n) *)
    let d  = (5 ^. n) << n in
    let d2 = sqr d         in

    (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2))) *)
    let a = gsqrt Nearest_up (d2 *. 200) in
    let b = gsqrt Nearest_up (d2 *. 300) in

    (* r <- round(10^n*(b+a)/(b-a)) *)
    let r = gquo Nearest_up (d**(b++a)) (b--a) in
    printf__printf "r=%s\n" (string_of r);
    flush stdout

  in
  main (list_of_vect sys__command_line);;
```

Compilation and execution:

```
> camlc -custom -I ~/lib -o simple nums.zo numerix.zo simple.ml \
        -lnumerix-caml -lnums -lgmp -ccopt -L/home/quercia/lib
> ./simple
r=9898979485566356196394568149411
>
```

Note that the three `libnumerix-caml`, `libnums` and `libgmp` libraries must be given to the linker even if the `clong` module is the only one used, because the other modules are included in `numerix.zo` and contain references to functions from these three libraries.

3.2.3 Toplevel

A customized toplevel is available in Camllight for doing `Numerix` computations:

```
> camllight ~/lib/camlnumx
>       Caml Light version 0.75

camlnumx : Caml toplevel with big integer libraries
Numerix submodules : clong dlong slong big gmp
Numerix version    : 0.21
```

```
##open "slong";;
##open "infslong";;
#fact 30;;
- : t = 265252859812191058636308480000000
#one << 100;;
- : t = 1267650600228229401496703205376
#quit();;
>
```

Chapter 4

Use with C

Contents

4.1 Interface

The `Numerix` C interface was derived from the Ocaml one by adding a simple memory manager in order to cope with the lack of Ocaml GC, and by restricting the interface to the operations implemented in the C kernel of `Numerix`. The main purpose of this interface is to allow a fair comparison between `Numerix` and `GMP` (designed to be used with C), and to allow the compilation and the execution of test programs on computers where Ocaml is not installed. The `Numerix` C interface was successfully tested with the `gcc-3.3.3`, `gcc-2.95.3`, `gcc-2.7.2.3` compilers and the Linux, OpenBSD and Digital Unix operating systems.

4.1.1 Conventions

The three `clong`, `dlong` and `slong` modules are available, as far as the C compiler and the computer hardware allow compilation. The choice of the module to be used is done at compile-time with the help of a `#define use_xxx` directive where `xxx` is the name of the module. This directive can be included in each source file or it can be given to the preprocessor with a `-Duse_xxx` option. Note the following differences with respect to `Numerix`-0.19:

- it is no longer necessary to specify the machine word size;
- there is no default module;
- the header file describing `Numerix`-0.19 was called `c-long_int.h`, the one describing `Numerix`-0.21 is called `numerix.h`.

The `numerix.h` file defines the `xint` datatype representing a big integer and gives prototypes for the functions operating on these big integers. The function names are prefixed with a three character string identifying the module to which they belong: `cx_` for the `clong` module, `dx_` for `dlong` and `sx_` for `slong`. In order to allow the programmer to write big integer implementation independent code, the `numerix.h` file defines a `xx` macro which catenates its argument with the `cx_`, `dx_` or `sx_` prefix depending on which `use_clong`, `use_dlong` or `use_slong` symbol is defined. One will write:

```
xx(add)(&x,a,b);
```

to add `a` and `b` into `x`, this code being transformed by the preprocessor into:

`cx_add(&x,a,b);` or `dx_add(&x,a,b);` or `sx_add(&x,a,b);`

The user is advised to use systematically the `xx` macro rather than to use the expanded identifiers. Doing this this way, he can recompile his program with another big integer implementation by only modifying the `#define use_xxx` directive. Anyway the functions of one module cannot operate on the data of another module and there is no mechanism for distinguishing the `xint` datatype according to a specific module.

As a general rule, a function computing a result `a` of type `xint` is available in two versions differing by their calling convention:

```
xint xx(func)(xint *_a, args)
xint xx(f_func)(args)
```

In both cases, the return value is the computed result `a`. Moreover, if `_a != NULL`, then the result is copied into the memory location designated by `_a`. There are two exceptions to this naming convention: `copy_int` and `copy_string` have as associated functions the `of_int` and `of_string` functions instead of `f_copy_int` and `f_copy_string` for the sake of compatibility with `Numerix`-0.19. A function computing several results `a`, `b`,... of type `xint` is available in only one version:

```
void xx(func)(xint *_a, xint *_b,..., args)
```

The results `a`, `b`,... computed are stored in the memory locations designated by the pointers `_a`, `_b`,... If one of these pointers is `NULL`, the corresponding result is not copied and is not accessible to the caller.

4.1.2 The `numerix.h` file

Below is a part of `numerix.h` giving the prototypes of the public functions:

```
typedef struct {...} *xint;

/*-------------------- creation/destruction */
xint xx(new)();
void xx(free)(xint *_x);

xint xx(copy)   (xint *_b, xint a);
xint xx(f_copy) (xint a);

/*-------------------- addition/subtraction */
xint xx(add)    (xint *_c, xint a, xint b);
xint xx(sub)    (xint *_c, xint a, xint b);
xint xx(add_1)  (xint *_c, xint a, long b);
xint xx(sub_1)  (xint *_c, xint a, long b);

xint xx(f_add)  (xint a, xint b);
xint xx(f_sub)  (xint a, xint b);
xint xx(f_add_1)(xint a, long b);
xint xx(f_sub_1)(xint a, long b);

/*-------------------- multiplication/square */
xint xx(mul)    (xint *_c, xint a, xint b);
xint xx(mul_1)  (xint *_c, xint a, long b);
xint xx(sqr)    (xint *_b, xint a);

xint xx(f_mul)  (xint a, xint b);
xint xx(f_mul_1)(xint a, long b);
xint xx(f_sqr)  (xint a);

/*-------------------- division */
void xx(quomod)   (xint *_c, xint *_d, xint a, xint b);
xint xx(quo)      (xint *_c,           xint a, xint b);
xint xx(mod)      (xint *_d,           xint a, xint b);
long xx(quomod_1) (xint *_c,           xint a, long b);
xint xx(quo_1)    (xint *_c,           xint a, long b);
long xx(mod_1)    (                    xint a, long b);
void xx(gquomod)  (xint *_c, xint *_d, xint a, xint b, long mode);
xint xx(gquo)     (xint *_c,           xint a, xint b, long mode);
xint xx(gmod)     (xint *_d,           xint a, xint b, long mode);
long xx(gquomod_1)(xint *_c,           xint a, long b, long mode);
xint xx(gquo_1)   (xint *_c,           xint a, long b, long mode);
long xx(gmod_1)   (                    xint a, long b, long mode);

xint xx(f_quo)    (xint a, xint b);
xint xx(f_mod)    (xint a, xint b);
xint xx(f_quo_1)  (xint a, long b);
long xx(f_mod_1)  (xint a, long b);
```

```
xint xx(f_gquo)   (xint a, xint b, long mode);
xint xx(f_gmod)   (xint a, xint b, long mode);
xint xx(f_gquo_1) (xint a, long b, long mode);
long xx(f_gmod_1) (xint a, long b, long mode);

/*-------------------- absolute value, opposite */
xint xx(abs)    (xint *_b, xint a);
xint xx(neg)    (xint *_b, xint a);
xint xx(f_abs)  (xint a);
xint xx(f_neg)  (xint a);

/*-------------------- exponentiation */
xint xx(pow)      (xint *_b, xint a, long p);
xint xx(pow_1)    (xint *_b, long a, long p);
xint xx(powmod)   (xint *_d, xint a, xint b, xint c);
xint xx(gpowmod)  (xint *_d, xint a, xint b, xint c, long mode);

xint xx(f_pow)    (xint a, long p);
xint xx(f_pow_1)  (long a, long p);
xint xx(f_powmod) (xint a, xint b, xint c);
xint xx(f_gpowmod)(xint a, xint b, xint c, long mode);

/*-------------------- roots */
xint xx(sqrt)   (xint *_b, xint a);
xint xx(root)   (xint *_b, xint a, long p);
xint xx(gsqrt)  (xint *_b, xint a,         long mode);
xint xx(groot)  (xint *_b, xint a, long p, long mode);

xint xx(f_sqrt) (xint a);
xint xx(f_root) (xint a, long p);
xint xx(f_gsqrt)(xint a,         long mode);
xint xx(f_groot)(xint a, long p, long mode);

/*-------------------- factorial */
xint xx(fact)  (xint *_a, long n);
xint xx(f_fact)(long n);

/*-------------------- Greatest common divisor */
xint xx(gcd)   (xint *_d, xint a, xint b);
void xx(gcd_ex)(xint *_d, xint *_u, xint *_v, xint a, xint b);
void xx(cfrac) (xint *_d, xint *_u, xint *_v, xint *_p, xint *_q, xint a, xint b);
xint xx(f_gcd) (xint a, xint b);

/*-------------------- comparison */
long xx(sgn)    (xint a);
long xx(cmp)    (xint a, xint b);
long xx(cmp_1)  (xint a, long b);

long xx(eq)     (xint a,xint b);
long xx(neq)    (xint a,xint b);
long xx(inf)    (xint a,xint b);
long xx(infeq)  (xint a,xint b);
long xx(sup)    (xint a,xint b);
long xx(supeq)  (xint a,xint b);

long xx(eq_1)   (xint a,long b);
long xx(neq_1)  (xint a,long b);
long xx(inf_1)  (xint a,long b);
long xx(infeq_1)(xint a,long b);
long xx(sup_1)  (xint a,long b);
long xx(supeq_1)(xint a,long b);

/*-------------------- conversion */
xint xx(copy_int)   (xint *_b, long a);
xint xx(of_int)     (long a);
long xx(int_of)     (xint a);
xint xx(copy_string)(xint *_a, char *s);
xint xx(of_string)  (char *s);

char *xx(string_of) (xint a);
char *xx(hstring_of)(xint a);
char *xx(ostring_of)(xint a);
char *xx(bstring_of)(xint a);

/*-------------------- random integers */
void xx(random_init)(long n);
xint xx(nrandom) (xint *_a, long n);
xint xx(zrandom) (xint *_a, long n);
xint xx(nrandom1)(xint *_a, long n);
xint xx(zrandom1)(xint *_a, long n);

xint xx(f_nrandom) (long n);
xint xx(f_zrandom) (long n);
xint xx(f_nrandom1)(long n);
xint xx(f_zrandom1)(long n);

/*-------------------- binary representation */
long xx(nbits)   (xint a);
long xx(lowbits) (xint a);
long xx(highbits)(xint a);
long xx(nth_word)(xint a, long n);
long xx(nth_bit) (xint a, long n);

/*-------------------- shifts */
xint xx(shl)  (xint *_b,          xint a, long n);
xint xx(shr)  (xint *_b,          xint a, long n);
void xx(split)(xint *_b, xint *_c, xint a, long n);
xint xx(join) (xint *_c, xint a,   xint b, long n);

xint xx(f_shl) (xint a,         long n);
xint xx(f_shr) (xint a,         long n);
xint xx(f_join)(xint a, xint b, long n);

/*------------------- timing facility */
void chrono(char *msg);
```

4.1.3 Memory management

A `a` variable of type `xint` is a pointer to a data structure managed by the memory manager included in the C version of `Numerix`. The initialization of `a` is normally done in two steps:

- initialization of the `a` pointer;
- assignment of a value by giving the `&a` address as a result parameter of a computation.

It is possible to merge these two steps into a single one by assigning to `a` the result of type `xint` returned by a computation. Therefore, the following sequences where `a` denotes a variable of type `xint` not initialized and `b`,`c` denote variables of type `xint` initialized having been assigned the values b and c are equivalent: their common effect is to allocate a memory block, to copy into this block the internal representation of the number $b + c$, and to copy the address of the block into `a`.

```
a = xx(new)(); xx(add)(&a,b,c);
a = xx(f_add)(b,c);
a = xx(add)(NULL,b,c);
```

Once the `a` pointer is initialized, the `&a` address can be given as a result parameter to a computation. For instance:

```
xx(mul)(&a,b,c);
```

has for effect to compute the product bc and to copy into `a` the address of the memory block where this product has been stored. It is not necessary for `a` to have been assigned a value prior to this operation. If it is the case, then the memory block containing this value is overwritten with the internal representation of bc if the block is large enough, otherwise a new memory block is allocated to store the result, `a` is modified in order to point to the new block and the old block is reclaimed. The read-modify-write operations where the same variable is given both as an operand and as a result are handled correctly. On the other way, concerning the operations computing several results (`quomod`, `gquomod`, `gcd_ex`, `cfrac` and `split`) one variable cannot be given more than one time as a result. Therefore the following instruction is illegal:

```
xx(quomod)(&a,&a,b,c); /* illegal */
```

The `xx(free)` function enables one to return a memory block to the memory manager when the value stored in this memory block is no longer useful. The instruction:

```
xx(free)(&a);
```

has for effect to free the memory block designated by `a` if there is one and to reinitialize the `a` pointer. After this instruction, the `a` variable is still operational and can be assigned a new value.

4.1.4 Rounding mode

The operations computing an integer approximation of a real number a (division, square root and p-th root) are available in two versions:

```
xx(func) (args)
xx(gfunc)(args, long mode)
```

The `mode` parameter of `xx(gfunc)` specifies in which way the number a is to be rounded:

if `mode` $\& \ 3 = 0$:	compute $\lfloor a \rfloor$;
if `mode` $\& \ 3 = 1$:	compute $\lfloor a + 1/2 \rfloor$;
if `mode` $\& \ 3 = 2$:	compute $\lceil a \rceil$;
if `mode` $\& \ 3 = 3$:	compute $\lceil a - 1/2 \rceil$.

`xx(func)` is equivalent to `xx(gfunc)` with `mode` $= 0$.

4.1.5 Description of the functions

The operations implemented in the C interface of `Numerix` are identical to the ones implemented in the Ocaml interface and described in sections **2.2.4 Arithmetic operations** to **2.2.8 Access to the binary representation**, pages 14 and following, and in section **2.3.6 Timing**, page 24. Below are mentioned the particularities of the C interface.

- When an Ocaml function returns a boolean result, the equivalent C function returns an integer of type `long` the value of which is 0 for `false` and 1 for `true`.

- The C functions converting a big integer into a character string return a pointer to a string allocated on the heap. This string must be released after use by calling the `free` function. Note that `Numerix`-0.19 provided a `xx(free_string)` function for this purpose: this function does not exist anymore in `Numerix`-0.21.

- The `xx(lowbits)` and `xx(highbits)` functions return respectively the 31 least significant bits and the 31 most significant bits of their argument, regardless of the machine word size. Also, the `xx(int_of)` function raises systematically an error when the absolute value of its argument is greater than 2^{30}.

4.2 Use

4.2.1 Compilation

The C programs using `Numerix` must be compiled with the following command:

```
gcc options -Duse_xxx source files -lnumerix-c
```

`-Duse_xxx` specifies which module to use, `clong` or `dlong` or `slong`.

`-lnumerix-c` asks the linker to search in the `libnumerix-c` library the required compiled functions. It may be necessary to tell the linker where to find this library with a `-L`*path* option. Similarly it may be necessary to tell the preprocessor where to find the `numerix.h` header file with a `-I`*path* option. Note that the `libcnumx` library used with `Numerix`-0.19 has been renamed `libnumerix-c` in `Numerix`-0.21, the latter name being judged more informative.

4.2.2 Example

```
/* file simple.c: simple demo of Numerix
   compute  (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "numerix.h"

int main(int argc, char **argv) {

   xint a,b,d,d2,x,y;
   char *s;
   long n;

   /* number of digits */
   if ((argc > 2) && (strcmp(argv[1],"-n") == 0)) n = atol(argv[2]);
   else n = 30;

   /* d <- 10^n, d2 <- 10^(2n) */
   d  = xx(f_pow_1)(5,n); xx(shl)(&d,d,n);
   d2 = xx(f_sqr)(d);

   /* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2)))  */
   a = xx(f_mul_1)(d2,200); xx(gsqrt)(&a,a,1);
   b = xx(f_mul_1)(d2,300); xx(gsqrt)(&b,b,1);

   /* x <- round(10^n*(b+a)/(b-a)) */
   x = xx(f_add)(b,a); xx(mul)(&x,x,d);
   y = xx(f_sub)(b,a);
   xx(gquo)(&x,x,y,1);
   /* print x */
   s = xx(string_of)(x); printf("x=%s\n",s); free(s);

   /* free temporary memory */
   xx(free)(&d); xx(free)(&d2);
   xx(free)(&a); xx(free)(&b);
   xx(free)(&x); xx(free)(&y);

   return(0);
}
```

Compilation and execution:

```
> gcc -O2 -Wall -I/home/quercia/include -Duse_slong \
      -o simple simple.c -lnumerix-c -L/home/quercia/lib
> ./simple -n 20
x=9898979485566356196 42
>
```

Chapter 5

Use with Pascal

Contents

The `Numerix` Pascal interface was derived from the C one and provides the same functionalities. It was developed on a Linux PC with the Free Pascal version 1.0.10 compiler.

5.1 Interface

5.1.1 Units

Three units are defined: `clong`, `dlong` and `slong`. Each unit contains the declaration of the corresponding big integer datatype and the declarations of the associated procedures and functions. The exported identifiers are the same in each unit, this allows one to write unit-independent programs, one only needs to change the `uses` clause in order to change the big integer implementation used.

Below is a part of the `clong.p` file describing the `clong` unit:

```
unit clong;
interface

type xint = ...;

(* creation/destruction *)
function xnew : xint;
```

```
procedure xfree(var x : xint);

procedure copy (var b:xint; a:xint);
function f_copy(a:xint):xint;

(* addition/subtraction *)
procedure add  (var c:xint; a:xint; b:xint   );
procedure sub  (var c:xint; a:xint; b:xint   );
procedure add_1(var c:xint; a:xint; b:longint);
procedure sub_1(var c:xint; a:xint; b:longint);

function f_add  (a:xint; b:xint   ):xint;
function f_sub  (a:xint; b:xint   ):xint;
function f_add_1(a:xint; b:longint):xint;
function f_sub_1(a:xint; b:longint):xint;

(* multiplication *)
procedure mul  (var c:xint; a:xint; b:xint   );
procedure mul_1(var c:xint; a:xint; b:longint);
procedure sqr  (var b:xint; a:xint);

function f_mul  (a:xint; b:xint   ):xint;
function f_mul_1(a:xint; b:longint):xint;
function f_sqr  (a:xint):xint;

(* division *)
procedure quomod   (var c,d:xint; a:xint; b:xint   );
procedure quo      (var c  :xint; a:xint; b:xint   );
procedure modulo   (var d  :xint; a:xint; b:xint   );
procedure quomod_1 (var c  :xint; a:xint; b:longint);
procedure quo_1    (var c  :xint; a:xint; b:longint);
function  mod_1                  (a:xint; b:longint):longint;
procedure gquomod  (var c,d:xint; a:xint; b:xint;    mode:longint);
procedure gquo     (var c  :xint; a:xint; b:xint;    mode:longint);
procedure gmod     (var d  :xint; a:xint; b:xint;    mode:longint);
function  gquomod_1(var c  :xint; a:xint; b:longint; mode:longint):longint;
procedure gquo_1   (var c  :xint; a:xint; b:longint; mode:longint);
function  gmod_1                 (a:xint; b:longint; mode:longint):longint;

function f_quo     (a:xint; b:xint   ):xint;
function f_mod     (a:xint; b:xint   ):xint;
function f_quo_1   (a:xint; b:longint):xint;
function f_mod_1   (a:xint; b:longint):longint;
function f_gquo    (a:xint; b:xint;    mode:longint):xint;
function f_gmod    (a:xint; b:xint;    mode:longint):xint;
function f_gquo_1  (a:xint; b:longint; mode:longint):xint;
function f_gmod_1  (a:xint; b:longint; mode:longint):longint;

(* absolute value/opposite *)
procedure abs  (var b:xint; a:xint);
procedure neg  (var b:xint; a:xint);

function f_abs (a:xint):xint;
function f_neg (a:xint):xint;

(* exponentiation *)
procedure pow    (var b:xint; a:xint;    p:longint);
procedure pow_1  (var b:xint; a:longint; p:longint);
procedure powmod (var d:xint; a:xint;    b:xint; c:xint);
procedure gpowmod(var d:xint; a:xint;    b:xint; c:xint; mode:longint);

function f_pow    (a:xint;    p:longint):xint;
function f_pow_1  (a:longint; p:longint):xint;
function f_powmod (a:xint;    b:xint; c:xint):xint;
function f_gpowmod(a:xint;    b:xint; c:xint; mode:longint):xint;

(* roots *)
procedure sqrt (var b:xint; a:xint);
procedure gsqrt(var b:xint; a:xint;            mode:longint);
procedure root (var b:xint; a:xint; p:longint);
procedure groot(var b:xint; a:xint; p:longint; mode:longint);

function f_sqrt (a:xint                ):xint;
function f_gsqrt(a:xint; mode:  longint):xint;
function f_root (a:xint; p:     longint):xint;
function f_groot(a:xint; p,mode:longint):xint;

(* factorial *)
procedure fact(var a:xint; n:longint);
function f_fact(n:longint):xint;

(* gcd *)
procedure gcd   (var d:xint;         a,b:xint);
procedure gcd_ex(var d,u,v:xint;     a,b:xint);
procedure cfrac (var d,u,v,p,q:xint; a,b:xint);

function f_gcd(a,b:xint):xint;

(* comparison *)
function cmp    (a:xint; b:xint   ):longint;
function cmp_1  (a:xint; b:longint):longint;
function sgn    (a:xint           ):longint;
function eq     (a:xint; b:xint   ):boolean;
function neq    (a:xint; b:xint   ):boolean;
function inf    (a:xint; b:xint   ):boolean;
function infeq  (a:xint; b:xint   ):boolean;
function sup    (a:xint; b:xint   ):boolean;
function supeq  (a:xint; b:xint   ):boolean;
function eq_1   (a:xint; b:longint):boolean;
function neq_1  (a:xint; b:longint):boolean;
function inf_1  (a:xint; b:longint):boolean;
function infeq_1(a:xint; b:longint):boolean;
function sup_1  (a:xint; b:longint):boolean;
function supeq_1(a:xint; b:longint):boolean;

(* conversions *)
procedure copy_int(var b:xint; a:longint);
procedure copy_string(var a:xint; s:pchar);

function of_int(a:longint):xint;
function of_string(s:pchar):xint;

function string_of (a:xint):ansistring;
function hstring_of(a:xint):ansistring;
function ostring_of(a:xint):ansistring;
function bstring_of(a:xint):ansistring;

(* random numbers *)
procedure random_init(n:longint);

procedure nrandom (var a:xint; n:longint);
procedure zrandom (var a:xint; n:longint);
procedure nrandom1(var a:xint; n:longint);
procedure zrandom1(var a:xint; n:longint);

function f_nrandom (n:longint):xint;
function f_zrandom (n:longint):xint;
function f_nrandom1(n:longint):xint;
function f_zrandom1(n:longint):xint;

(* binary representation *)
function int_of  (a:xint           ):longint;
function nbits   (a:xint           ):longint;
function lowbits (a:xint           ):longint;
function highbits(a:xint           ):longint;;
function nth_word(a:xint; n:longint):longint;;
function nth_bit (a:xint; n:longint):boolean;

(* shifts *)
procedure shiftl(var b:xint; a:xint; n:longint);
procedure shiftr(var b:xint; a:xint; n:longint);
procedure split(var b,c:xint; a:xint; n:longint);
procedure join(var c:xint; a:xint; b:xint; n:longint);

function f_shl(a:xint; n:longint):xint;
function f_shr(a:xint; n:longint):xint;
function f_join(a:xint; b:xint; n:longint):xint;

(* timing *)
procedure chrono(msg:pchar);
```

5.1.2 Memory management

A `a` variable of type `xint` is a pointer to a record managed by the memory manager included in the Pascal version of `Numerix`. The initialization of `a` is normally done in two steps:

- initialization of the `a` pointer;
- assignment of a value by giving `a` as a result parameter to a computation.

It is possible to merge these two steps into a single one by assigning to `a` the result of type `xint` returned by a computation. Therefore, the following sequences where `a` denotes a variable of type `xint` not initialized and `b`,`c` denote variables of type `xint` initialized having been assigned the values b and c are equivalent: their common effect is to allocate a memory block, to copy the internal representation of the number $b + c$ into this block, and to copy the address of this block into `a`.

```
a := xnew(); add(a,b,c);
a := f_add(b,c);
```

Once the `a` pointer is initialized, `a` can be given as a result parameter of a computation. For instance:

```
mul(a,b,c);
```

has for effect to compute the product bc and to copy into `a` the address of the memory block where this product has been stored. It is not necessary for `a` to have been assigned a value prior to this operation. If it is the case, then the memory block containing this value is overwritten with the internal representation of bc if the block is large enough, otherwise a new memory block is allocated to store the result, `a` is modified in order to point to the new block and the old block is reclaimed. The read-modify-write operations where the same variable is given both as an operand and as a result are handled correctly. On the other way, concerning the operations computing several results (`quomod`, `gquomod`, `gcd_ex`, `cfrac` and `split`) one variable cannot be given more than one time as a result. Therefore the following instruction is illegal:

```
quomod(a,a,b,c); (* illegal *)
```

The `xfree` procedure enables one to return a memory block to the memory manager when the value stored in this memory block is no longer useful. The instruction:

```
xfree(a);
```

has for effect to free the memory block designated by `a` if there is one and to reinitialize the `a` pointer. After this instruction, the `a` variable is still operational and can be assigned a new value.

5.1.3 Rounding mode

The operations computing an integer approximation of a real number a (division, square root and p-th root) are available in two versions:

```
func (args)
gfunc(args; mode:longint)
```

The `mode` parameter of `gfunc` specifies in which way the number a is to be rounded:

if `mode and` $3 = 0$:	compute $\lfloor a \rfloor$;
if `mode and` $3 = 1$:	compute $\lfloor a + 1/2 \rfloor$;
if `mode and` $3 = 2$:	compute $\lceil a \rceil$;
if `mode and` $3 = 3$:	compute $\lceil a - 1/2 \rceil$.

`func` is equivalent to `gfunc` with `mode` $= 0$.

5.1.4 Description of the functions

The operations implemented in the Pascal interface of `Numerix` are identical to the ones implemented in the Ocaml interface and described in sections **2.2.4 Arithmetic operations** to **2.2.8 Access to the binary representation**, pages 14 and following, and in section **2.3.6 Timing**, page 24. Below are mentioned the particularities of the Pascal interface.

- The `lowbits` and `highbits` functions return respectively the 31 least significant bits and the 31 most significant bits of their argument, regardless of the machine word size. Also, the `xx(int_of)` function raises systematically an error when the absolute value of its argument is greater than 2^{30}.

5.2 Use

5.2.1 Compilation

The Pascal programs using `Numerix` must be compiled with the following command:

`fpc` *options* `-Fu`*ppu_path* `-Fl`*lib_path* *source files*

`-Fu`*ppu_path* designates the directory containing the `clong.ppu`, `clong.o`, `dlong.ppu`, `dlong.o`, `slong.ppu` and `slong.o` compiled files. `-Fl`*lib_path* designates the directory containing the `libnumerix-c` library. These directives can be omitted if these files are stored in directories normally scanned by the Pascal compiler. Note that the `libcnumx` library used with `Numerix`-0.19 has been renamed `libnumerix-c` in `Numerix`-0.21, the latter name being judged more informative.

5.2.2 Example

```
program simple;
```

```
(* file exemples/pascal/simple.p: simple demo of Numerix
   compute  (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) with n digits *)

uses clong;

var a,b,d,d2,x,y:xint;
    n : longint;
    c : word;
begin

   (* number of digits *)
   if (paramcount >= 2) and (paramstr(1) = '-n')
      then val(paramstr(2),n,c)
      else n := 30;

   (* d <- 10^n, d2 <- 10^(2n) *)
   d  := f_pow_1(5,n); shiftl(d,d,n);
   d2 := f_sqr(d);

   (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2)))  *)
   a := f_mul_1(d2,200); gsqrt(a,a,1);
   b := f_mul_1(d2,300); gsqrt(b,b,1);

   (* x <- round(10^n*(b+a)/(b-a)) *)
   x := f_add(b,a); mul(x,x,d);
   y := f_sub(b,a);
   gquo(x,x,y,1);

   (* print x *)
   writeln('x=',string_of(x));

   (* free temporary memory *)
   xfree(d); xfree(d2);
   xfree(a); xfree(b);
   xfree(x); xfree(y);

end.
```

Compilation and execution:

```
> fpc -Fu/home/quercia/lib -Fl/home/quercia/lib simple.p
Free Pascal Compiler version 1.0.10 [2003/06/26] for i386
Copyright (c) 1993-2003 by Florian Klaempfl
Target OS: Linux for i386
Compiling simple.p
Assembling simple
Linking simple
38 Lines compiled, 0.0 sec
> ./simple -n 50
x=989897948556635619639456814941178278393189496131333
>
```

Chapter 6

Installation

Contents

6.1 Downloading

`Numerix` is available at the following URL:

```
http://pauillac.inria.fr/~quercia/cdrom/bibs/numerix.tar.gz
```

You will need the `gcc` C compiler to compile the C and assembly parts of the library, any recent version of `gcc` should fit for that. The library was successfully compiled on a Linux PC with `gcc-3.3.3` as well as on a Dec workstation with `gcc 2.7.2.3` for the `Clong` module only.

For Ocaml you need a not less than `3.06` version and for Camllight a not less than `0.74` version. Ocaml and Camllight are available at the URL:

```
http://pauillac.inria.fr/caml/index-eng.html
```

If you want to include the `Gmp` module in the interfaces for Ocaml and Camllight then you need `GMP` installed on your computer. `GMP` is available at the URL:

```
http://www.swox.com/gmp/
```

The Pascal interface can be compiled only on a Linux PC with the Free Pascal compiler which is available at the URL:

```
http://www.freepascal.org/
```

6.2 Configuration

6.2.1 Automatic configuration

Extract the `numerix.tar.gz` archive in a temporary directory and run the configuration script at the root.

On a Linux PC with the `bash` shell launch:

```
./configure 2>&1 | tee conflog
```

On a Dec workstation with the `csh` shell launch:

```
./configure |& tee conflog
```

This script checks which parts of `Numerix` can be compiled on your computer and creates a `Makefile` file suited for your configuration. The `configure` script accepts the following options:

`--prefix=`*dir*

Set the common root for installation directories:

`INSTALL_LIB` `=` *dir*`/lib`,

`INSTALL_BIN` `=` *dir*`/bin`,

`INSTALL_INCLUDE` `=` *dir*`/include`.

dir must be an absolute path. The default prefix is `$HOME`.

`--libdir=`*dir,* `--bindir=`*dir,* `--includedir=`*dir*

Set one of `INSTALL_LIB`, `INSTALL_BIN` and `INSTALL_INCLUDE` directory regardless of the others. *dir* must be an absolute path.

`--enable-c`, `--disable-c`

Select or un-select the C interface. The default is to select it.

`--enable-caml`, `--disable-caml`

Select or un-select the Camllight interface. The default is to select it when Camllight is found on the computer.

`--enable-ocaml`, `--disable-ocaml`

Select or un-select the Ocaml interface. The default is to select it when Ocaml is found on the computer.

`--enable-pascal`, `--disable-pascal`

> Select or un-select the Pascal interface. The default is to select it when Free Pascal is found on the computer.

`--enable-clong`, `--disable-clong`

> Select or un-select the `Clong` module. The default is to select it.

`--enable-dlong`, `--disable-dlong`

> Select or un-select the `Dlong` module. The default is to select it when the C compiler supports the double precision integer datatype (`long long`) and when this datatype length is twice the length of a `long`.

`--enable-slong`, `--disable-slong`

> Select or un-select the `Slong` module. The default is to select it when the processor is an Intel `x86` one and when the C compiler correctly supports the double precision integer datatype.

`--enable-gmp`, `--disable-gmp`

> Select or un-select the `Gmp` module. The default is to select it when the `GMP` library is found on the computer.

`--enable-caml_bignum`, `--disable-caml_bignum`
`--enable-ocaml_bignum`, `--disable-ocaml_bignum`

> Select or un-select the `Big` module independently for Camllight and for Ocaml. The default is to select it when the corresponding `libnums` library is found on the computer.

`--disable-lang`

> Un-select all the languages not explicitely selected with a `--enable-xxx` option.

`--disable-modules`

> Un-select all the modules not explicitely selected with a `--enable-xxx` option.

`--disable-all`

> Un-select all the module/language combinations not explicitely selected with a `--enable-xxx` option.

`--enable-x86`, `--disable-x86`

> Tell if the processor is `x86` compatible. By default the processor is supposed `x86` compatible if the canonical operating system name contains the `i*86` pattern.

`--enable-sse2`, `--disable-sse2`

> Tell if the processor, which must be `x86` compatible, supports the `SSE2` instruction set. By default, if the processor is `x86` compatible and if the operating system name contains the `Linux` string, the `configure` program reads the `/proc/cpuinfo` file in order to get this information.

`--enable-alloca`, `--disable-alloca`

Tell if the `alloca` temporary allocation function can be used in lieu of `malloc`. The default is to enable the use of `alloca` when `configure` finds a correct interface for this function.

6.2.2 Manual configuration

Normally the `configure` script described in the previous section should create suitable `Makefile`, `kernel/*/makefile` and `kernel/config.h` files. In case of trouble, edit the `Makefile` and `kernel/config.h` files in order to fix the values written by `configure` when they are wrong. After correction, you must re-create the `kernel/*/makefile` auxiliary files in order to take into account the modifications and you must delete the files created during a preceding compilation. To do this, launch:

```
make makefiles
make clean
```

6.2.3 Editing the `Makefile`

Use the values 0 or 1 for boolean parameters (1 = true).

```
PROCESSOR = x86-sse2
```

Specify the processor type: `x86-sse2` for a processor of type `x86` supporting the `SSE2` instruction set, `x86` for a processor of type `x86` not supporting this instruction set, and `generic` for any other processor.

```
MAKE_C_LIB      = 1
MAKE_OCAML_LIB  = 1
MAKE_CAML_LIB   = 1
MAKE_PASCAL_LIB = 1
```

Specify which interfaces you want.

```
USE_CLONG        = 1
USE_DLONG        = 1
USE_SLONG        = 1
USE_GMP          = 1
USE_CAML_BIGNUM  = 1
USE_OCAML_BIGNUM = 1
```

Specify the modules to be compiled: severals modules can be specified. The `Dlong` and `Slong` modules can be compiled only on computers with a `x86` compatible processor. The `Gmp` and `Big` modules can be compiled only if you have `GMP` and `Big_int`.

```
GCC = gcc -O2 -Wall
AR  = ar -rc
RANLIB = ranlib
```

Specify the commands to launch to call the C compiler and the librarian. You can add -Ixxx and -Lxxx directives if the compiler or the linker fail to find some header files or libraries.

```
CAML_LIBDIR = /usr/local/lib/caml-light
CAMLC       = camlc
CAMLLIBR    = camllibr
CAMLMKTOP   = camlmktop
```

Specify the Camllight directory and the commands to launch to call the Camllight compiler, the Camllight archiver and the Camllight toplevel compiler.

```
OCAML_LIBDIR = /usr/local/lib/ocaml
OCAMLC       = ocamlc
OCAMLOPT     = ocamlopt
OCAMLMKTOP   = ocamlmktop
OCAMLMKLIB   = ocamlmklib
```

Specify the Ocaml directory and the commands to launch to call the Ocaml compiler, the Ocaml optimizing compiler, the Ocaml toplevel compiler and the Ocaml library generator.

```
FPC = fpc -v0 -k-lgcc_s
```

Specify the command to launch the Pascal compiler.

```
INSTALL_LIB     = $(HOME)/lib
INSTALL_INCLUDE = $(HOME)/include
INSTALL_BIN     = $(HOME)/bin
```

Specify in which directories the compiled libraries, the header files and the binaries should be installed.

```
C_INSTALL_BIN          = $(INSTALL_BIN)
C_INSTALL_LIB          = $(INSTALL_LIB)
C_INSTALL_INCLUDE      = $(INSTALL_INCLUDE)

CAML_INSTALL_BIN       = $(INSTALL_BIN)
CAML_INSTALL_LIB       = $(INSTALL_LIB)
CAML_INSTALL_INCLUDE   = $(INSTALL_INCLUDE)

OCAML_INSTALL_BIN      = $(INSTALL_BIN)
OCAML_INSTALL_LIB      = $(INSTALL_LIB)
OCAML_INSTALL_INCLUDE  = $(INSTALL_INCLUDE)

PASCAL_INSTALL_BIN     = $(INSTALL_BIN)
PASCAL_INSTALL_LIB     = $(INSTALL_LIB)
PASCAL_INSTALL_INCLUDE = $(INSTALL_INCLUDE)
```

By default the INSTALL_BIN, INSTALL_LIB and INSTALL_INCLUDE directories are used for all languages. You can define a different directory set for each language by modifying the corresponding parameters. Note that the values of OCAML_INSTALL_LIB and CAML_INSTALL_LIB are hard-coded into the ocamlnumx and camlnumx toplevels so that these toplevels can find the numerix.cmi and numerix.zi compiled interfaces by themselves. Therefore, if you want to move these directories, you will need to recompile camlnumx and ocamlnumx.

6.2.4 Editing the kernel/config.h

This file contains internal settings for the C/assembly kernel of Numerix. Normally it is created by the configure script with the help of the informations given or found on the processor and the possibility to use the alloca function. When configure detects wrong informations, use the --enable_xxx and --disable_xxx options described in section 6.2.1 in order to force correct values. If configure fails to write a kernel.config.h file, then copy one of the config/generic.h, config/x86.h or config/x86-sse2.h files onto kernel/config.h, and edit this last file in order to specify the bit length of a machine word and if the alloca function can be used:

```
/* Machine word size */
#define bits_@machine_word_size@

/* Memory allocation strategy  */
@use_alloca@
```

Replace the @machine_word_size@ string with 32 or 64 and the @use_alloca@ string with #define use_alloca or #undef use_alloca.

6.3 Compilation

After the automatic or manual configuration step you can launch the compilation. The targets are:

lib :
: compile the libraries and the interface files;

examples :
: compile the examples;

test :
: execute each example program with the -test option;

install :
: copy the libraries, the header files and the binaries in the directories specified by the INSTALL_xxx variables;

clean :
: delete all compiled files.

On a Linux PC with the bash shell successively launch:

```
make lib      2>&1 | tee liblog
make exemples 2>&1 | tee exlog
make test     2>&1 | tee testlog
```

On a Dec workstation with the `csh` shell successively launch:

```
make lib      |& tee liblog
make exemples |& tee exlog
make test     |& tee testlog
```

There should be neither compile error nor warning. If there are some and if you cannot solve the problem on your own, please send the `conflog`, `liblog`, `exlog` and `testlog` log files to `michel.quercia@prepas.org` for diagnosis. If you have faced some problems that you have been able to fix alone, please let me know so that I may modify the faulty files. The `logs/pentium/xxxlog` files shipped with the distribution contain the compilation logs for a Linux PC, have a look at them in case of trouble.

If the compilation and the tests have been successful, you can install the `Numerix` library with one of the commands:

```
make install 2>&1 | tee inslog
make install |& tee inslog
```

Refer to page 54 for the list of the files to be installed. The files actually installed depend on the modules and languages selected and on the static vs. dynamic choice for the libraries.

Now the installation is finished and you can enjoy the multi-precision programming. The user guide that you are presently reading is available in the `doc/english` subdirectory in PDF and LaTeX formats (files `numerix-eng.pdf` and `numerix-eng.tex`).

6.4 Description of the examples

The `c`,`caml`,`ocaml` and `pascal` sub-directories of the `exemples` directory contain various programs using `Numerix`. To compile these programs launch the command:

```
make examples
```

Concerning the examples in C, Caml and Pascal, a `ex.`*ext* source file is compiled in as many executables as there are available big integer modules for this language. Each executable is named `ex-x` where `x` is the initial letter of the big integer module used. Concerning the examples in Ocaml, a `ex.ml` source file is compiled in two executables: `ex` with the `ocamlc` compiler and `ex-opt` with the `ocamlopt` compiler. The choice of a big integer module is done at run-time with a `-e xxx` option as described in section **2.3.5 Run-time selection of a module**, page 23.

Figure 6.1: list of the `Numerix` files to install

`$(C_INSTALL_LIB)`	`$(CAML_INSTALL_LIB)`	`$(OCAML_INSTALL_LIB)`	`$(PASCAL_INSTALL_LIB)`
libnumerix-c.a	libnumerix-caml.a	libnumerix-ocaml.a dllnumerix-ocaml.so	
	numerix.zo camlnumx big.zi clong.zi dlong.zi gmp.zi slong.zi infbig.zi infclong.zi infdlong.zi infgmp.zi infslong.zi	numerix.a numerix.cma numerix.cmi numerix.cmxa	clong.o clong.ppu dlong.o dlong.ppu slong.o slong.ppu

`$(C_INSTALL_INCLUDE)`	`$(CAML_INSTALL_INCLUDE)`	`$(OCAML_INSTALL_INCLUDE)`	`$(PASCAL_INSTALL_INCLUDE)`
numerix.h	big.ml big.mli clong.ml clong.mli dlong.ml dlong.mli gmp.ml gmp.mli slong.ml slong.mli infbig.ml infbig.mli infclong.ml infclong.mli infdlong.ml infdlong.mli infgmp.ml infgmp.mli infslong.ml infslong.mli	numerix.ml numerix.mli	clong.p dlong.p slong.p

`$(C_INSTALL_BIN)`	`$(CAML_INSTALL_BIN)`	`$(OCAML_INSTALL_BIN)`	`$(PASCAL_INSTALL_BIN)`
		ocamlnumx	

6.4.1 chrono

Speed measurement of the different libraries (C interface only). This program chooses random big integers of sizes n and $2n$ bits and measures the time of various operations between these integers:

`mul`	multiplication n bits by n bits;
`sqr`	square of a n bit integer;
`quomod`	division with remainder $2n$ bits by n bits;
`quo`	division without remainder $2n$ bits by n bits;
`sqrt`	square root of a $2n$ bit integer;
`gcd`	gcd of two n bit integers;
`gcd_ex`	gcd and Bézout coefficients of two n bit integers;
`all`	all the operations above.

Specify on the command line a value for n and which operations to do among `mul`, `sqr`, `quomod`, `quo`, `sqrt`, `gcd` and `gcd_ex`. You can specify a repetition count with the `-r` r option, in this case each operation is repeated r times.

```
> exemples/c/chrono-s -all 1000000 -r 10
    0.03     0.03 début
    0.33     0.30 mul
    0.54     0.21 sqr
    1.31     0.77 quomod
    1.94     0.63 quo
    2.60     0.66 sqrt
    9.81     7.21 gcd
   20.63    10.82 gcd_ex
> exemples/c/chrono-g -all 1000000 -r 10
    0.01     0.01 début
    0.57     0.56 mul
    0.99     0.42 sqr
    3.20     2.21 quomod
    5.40     2.20 quo
    7.06     1.66 sqrt
   60.82    53.76 gcd
  176.33   115.51 gcd_ex
>
```

So on the test computer (Pentium-4-2.8Ghz) with the `Slong` module, the time for multiplying two one million bit numbers is 30 milliseconds, the time for squaring a one million bit number is 21 milliseconds, and so on. The second test shows the corresponding times for the `GMP`-4.1.4 library on the same computer.

6.4.2 digits

Search the smallest power of a number a for which the decimal expansion begins with a given digit sequence (Ocaml interface only). Formally, the program searches a minimal (x, y) pair of natural integers such that $c < a^x/10^y < c+1$ where c is the number designated by the digit sequence. The search is done with n bit approximations of $\ln(a)$, $\ln(10)$ $\ln(c)$ and $\ln(c+1)$ where n is determined from a and c. If the search is unsuccessful or if the solution found cannot be granted minimal then n is doubled and the computation is restarted. The command line parameters are in this order: the base a, the digit sequence c, and the maximum number of trials.

```
> exemples/ocaml/digits 3 1234567890 1
5399108054 2576029200
> exemples/ocaml/digits 3 1234567890 2
2440080224 1164214129 (minimal)
>
```

So $3^{5399108054} \approx 1234567890 \times 10^{2576029200}$, solution found in the first trial, and $3^{2440080224} \approx 1234567890 \times 10^{1164214129}$, solution found in the second trial. The second solution is minimal.

6.4.3 pi

Compute the n first digits of π (C, Caml, Ocaml and Pascal interfaces). This program implements the approximate computation of π described in the `BigNum` reference manual (*The Caml Numbers Reference Manual*, Inria, RT-0141) with a binary summation algorithm. Specify on the command line the number n and the computation options:

`-d` print the steps and the computing time for each step.

`-noprint` do not convert the number into a decimal string.

`-skip` convert the number into a decimal string, but display only the beginning and the end of the string.

`-gcd` reduce the fraction returned by the summation step before computing the quotient (one is advised against this reduction step because it takes longer than the time saved by doing a shorter division).

```
> exemples/caml/pi-s 1000000 -d -skip
    0.00     0.00 module = Slong
    0.05     0.05 puiss-5
    0.41     0.36 sqrt
    3.04     2.63 série lb=6875847
    3.52     0.48 quotient
    4.29     0.77 conversion
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151

>
```

6.4.4 shanks

Compute the modular square root b of a number a modulo an odd prime p. (C, Caml, Ocaml and Pascal interfaces). Specify on the command line the values for a and p with `-p` *value* and `-a` *value* options. If either value is not specified then the corresponding number is chosen at random. In this case, the `-bits` *bits* option specifies the bit size for the random numbers..

```
> exemples/pascal/shanks-s -bits 200
p = 1005766304904354230760358867719456972987081899952626048942177
a = 970580614050603730359753265239590766882437980714585883439039
b = 159719547119039909103138545846153210093959274914166418914496
```

6.4.5 simple

Simple demonstration program (C, Caml, Ocaml and Pascal interfaces). This program shows how to use the various `Numerix` interfaces. It computes the n first digits of $(\sqrt{3}+\sqrt{2})/(\sqrt{3}-\sqrt{2})$.

6.4.6 sqrt-163

Compute $\lfloor 10^n e^{\pi\sqrt{163}} \rfloor$ where n is given on the command line (Ocaml interface only).

```
> exemples/ocaml/sqrt-163-opt 10
262537412640768743.9999999999
```

Note that the result displayed proves that $e^{\pi\sqrt{163}}$ is not an integer: if there was an infinity of `9` after those displayed then the program could not have determined the floor part it was asked for.

6.4.7 cmp, rcheck

These programs are available with the Ocaml interface only. `cmp` makes a sequence of random operations with random integer operands, so as to detect `Numerix` internal bugs. Two big integer modules must be specified on the command line so as to compare the results returned by each module. The other command line options are the following:

`-n` *bits* specify the bit size for the operands;
`-op` *operation* specify one operation to check;
`-r` *count* specify the number of trials to do;
`-s` *seed* seed for the pseudo-random generator.

```
> exemples/ocaml/cmp -n 1000 -r 10000 -e clong -e gmp
Cmp(Clong,Gmp)
i=10000
>
```

10000 operations done without detecting any error.

`rcheck` is is a test program for the real-valued functions of the `Rfuns` functor. The program makes a sequence of computations for each of these functions and prints on the standard output stream `MuPAD` instructions to check the results. The command line options are:

`-bits` p specifies the bit sizes for the a and b operands;
`-n` n specifies the precision for the `xxx` functions of `Rfuns`;
`-c` c specifies the c coefficient for the `r_xxx` functions;
`-niter` i specifies the number of trials to do for each function;
`-seed` s seed for the pseudo-random generator.

```
> exemples/ocaml/rcheck -niter 100 -bits 200 -c 1000000000000 | mupad -P pe

  *----*     MuPAD 2.5.3 -- The Open Computer Algebra System
  /|   /|
 *----* |    Copyright (c)  1997 - 2003  by SciFace Software
 | *--|-*                   All rights reserved.
 |/   |/
 *----*      Licensed to:   Michel Quercia


                             c = 1000000000000

                                   x = 1

     u = 1452379063498458972355530797251267609669641280407182299120931

     v = -1194661026141584247149782554868676781233741566490766043049 2

                                  f = exp

                                 r = ceil
>
```

Only one error was detected: `Numerix` returns the result $x = 1$ for the value of $\lceil c \times \exp(u/v) \rceil$ whereas MuPAD finds another result (not displayed). After verification it turns out that MuPAD was wrong and `Numerix` was right.