Contents

1 Module `Fpu` : Access to low level floating point functions.

This module depends on chcw.c. IT ONLY WORKS FOR INTEL PROCESSORS.

Almost all low level functions are implemented using the x87 functions and x87 rounding modes. There are unfortunately a few problems to understand. The x87 is supposed to be able to return a nearest value and a upper and a lower bound for each elementary operation it can perform. This is not always true. Some functions such as cos(), sin() or tan() are not properly implemented everywhere.

For example, for the angle a= 1.570 796 326 794 896 557 998 981 734 272 092 580 795 288 085 937 5 the following values are computed for cos(a), by (1) the MPFI library (with 128 bits precision), (2) the x87 in low mode, (3) the x87 in nearest mode (default value for the C and Ocaml library on 32 bits linux), (4) the x87 in high mode, (5) the SSE2 implementation (default value for the C and Ocaml library on 64 bits linux):

(1) 6.123 233 995 736 765 886 130 329 661 375 001 464 640 377 798 836e-17
(2) 6.123 031 769 111 885 058 461 925 285 082 049 859 451 216 355 021e-17
(3) 6.123 031 769 111 886 291 057 089 692 912 995 815 277 099 609 375e-17
(4) 6.123 031 769 111 886 291 057 089 692 912 995 815 277 099 609 375e-17
(5) 6.123 233 995 736 766 035 868 820 147 291 983 023 128 460 623 387e-17

The upper bound (4) computed by the x87 is clearly incorrect, as it is lower than the correct value computed by the MPFI library.

The value computed by the SSE2 (5) is much more precise than the one computed by the x87. Unfortunately, there is no way to get an upper and lower bound value, and we are thus stuck with the x87 for computing these (sometimes incorrect) bounds.

The problem here is that the value computed by the standard, C-lib (or ocaml) cos function doesn't always lie in the lower/upper bound interval returned by the x87 functions, and this can be a very serious problem when executing Branch and Bound algorithms which expect the mid-value to be inside the lower/upper interval.

We solved the problem by rewritting the trigonometric functions in order to make them both consistant and correct. We used the following property: when -pi/4≤a≤pi/4 the rounding in 64 bits of the 80 bits low/std/high value returned by the x87 are correct. Moreover, when 0<a<2**53 then (a mod (2Pi_low)) and (a mod (2Pi_high)) are in the same quadrant. Last, (a mod Pi/2_High) ≤ (a mod Pi/2) ≤ (a mod Pi/2_Low). With this implementation, the lower and upper bounds are properly set and they are always lower (resp. higher) than the value computed by the standard cos functions on 32 and 64 bits architecture. This rewritting has been done in assembly language and is quite efficient.

Keep in mind that values returned by the standard (C-lib or Ocaml) cos(), sin() or tan() functions are still different on 32 and 64 bits architecture. If you want to have a program which behaves exactly in the same way on both architectures, you can use the `Fpu` module `fcos`, `fsin` or `ftan` functions which always return the same values on all architectures, or even use the `Fpu_rename` or `Fpu_rename_all` modules to transparently rename the floating point functions.

The functions are quite efficient (see below). However, they have a serious disadvantage compared to their standard counterparts. When the compiler compiles instruction "a+.b", the code of the operation is inlined, while when it compiles "(fadd a b)", the compiler generates a function call, which is expensive.

Intel Atom 230 Linux 32 bits

- tan speed (10000000 calls):2.380149
- ftan speed (10000000 calls):2.528158
- cos speed (10000000 calls):1.804113
- fcos speed (10000000 calls):2.076129
- sin speed (10000000 calls):1.844116
- fsin speed (10000000 calls):1.972123
- +. speed (10000000 calls):0.604037
- fadd speed (10000000 calls):0.980062
- -. speed (10000000 calls):0.644040
- fsub speed (10000000 calls):0.980061
- *. speed (10000000 calls):0.604038
- fmul speed (10000000 calls):0.980061
- /. speed (10000000 calls):0.992062
- fdiv speed (10000000 calls):1.424089
- ** speed (10000000 calls):15.420964
- pow speed (10000000 calls):4.528283
- mod_float speed (10000000 calls):1.996125
- fmod speed (10000000 calls):1.236077

Intel 980X Linux 64 bits

- tan speed (10000000 calls):0.896056

- ftan speed (10000000 calls):0.472029
- cos speed (10000000 calls):0.520033
- fcos speed (10000000 calls):0.400025
- sin speed (10000000 calls):0.524033
- fsin speed (10000000 calls):0.400025
- +. speed (10000000 calls):0.068005
- fadd speed (10000000 calls):0.124008
- -. speed (10000000 calls):0.068004
- fsub speed (10000000 calls):0.120008
- *. speed (10000000 calls):0.068004
- fmul speed (10000000 calls):0.128008
- /. speed (10000000 calls):0.096006
- fdiv speed (10000000 calls):0.156010
- ** speed (10000000 calls):0.668041
- pow speed (10000000 calls):1.028064
- mod_float speed (10000000 calls):0.224014
- fmod speed (10000000 calls):0.152010

```
val ffloat : int -> float
val ffloat_high : int -> float
val ffloat_low : int -> float
```

float() functions. The float function is exact on 32 bits machine but not on 64 bits machine with ints larger than 53 bits

```
val fadd : float -> float -> float
val fadd_low : float -> float -> float
val fadd_high : float -> float -> float
```

Floating point addition in nearest, low and high mode

```
val fsub : float -> float -> float
val fsub_low : float -> float -> float
val fsub_high : float -> float -> float
```

Floating point substraction in nearest, low and high mode

```
val fmul : float -> float -> float
val fmul_low : float -> float -> float
val fmul_high : float -> float -> float
```

Floating point multiplication in nearest, low and high mode

```
val fdiv : float -> float -> float
val fdiv_low : float -> float -> float
val fdiv_high : float -> float -> float
```

Floating point division in nearest, low and high mode

```
val fmod : float -> float -> float
```

Modulo (result is supposed to be exact)

```
val fsqrt : float -> float
val fsqrt_low : float -> float
val fsqrt_high : float -> float
```

Floating point square root in nearest, low and high mode

```
val fexp : float -> float
val fexp_low : float -> float
val fexp_high : float -> float
```

Floating point exponential in nearest, low and high mode

```
val flog : float -> float
val flog_low : float -> float
val flog_high : float -> float
```

Floating point log in nearest, low and high mode

```
val flog_pow : float -> float -> float
val flog_pow_low : float -> float -> float
val flog_pow_high : float -> float -> float
```

Computes x^y for 0 < x < infinity and neg_infinity < y < infinity

```
val fpow : float -> float -> float
val fpow_low : float -> float -> float
val fpow_high : float -> float -> float
```

Computes x^y expanded to its mathematical limit when it exists

```
val fsin : float -> float
val fsin_low : float -> float
val fsin_high : float -> float
```

Computes sin(x) for x in]-2^63, 2^63[

```
val fcos : float -> float
val fcos_low : float -> float
val fcos_high : float -> float
```

Computes cos(x) for x in]-2^63, 2^63[

```
val ftan : float -> float
val ftan_low : float -> float
val ftan_high : float -> float
```

Computes tan(x) for x in]-2^63, 2^63[

```
val fatan : float -> float -> float
val fatan_low : float -> float -> float
val fatan_high : float -> float -> float
```

fatan x y computes atan2 y x

```
val facos : float -> float
val facos_low : float -> float
val facos_high : float -> float
```

arc-cosine functions

```
val fasin : float -> float
val fasin_low : float -> float
val fasin_high : float -> float
```

arc-sinus functions

```
val fsinh : float -> float
val fsinh_low : float -> float
val fsinh_high : float -> float
```

Computes sinh(x)

```
val fcosh : float -> float
val fcosh_low : float -> float
val fcosh_high : float -> float
```

Computes cosh(x)

```
val ftanh : float -> float
val ftanh_low : float -> float
val ftanh_high : float -> float
```

Computes tanh(x)

`val is_neg : float -> bool`

is_neg x returns if x has its sign bit set (true for -0.)

Below, we have functions for changing the rounding mode. The default mode for rounding is NEAREST.

BE VERY CAREFUL: using these functions unwisely can ruin all your computations. Remember also that on 64 bits machine these functions won't change the behaviour of the SSE instructions.

When setting the rounding mode to UPWARD or DOWNWARD, it is better to set it immediately back to NEAREST. However we have no guarantee on how the compiler will reorder the instructions generated. It is ALWAYS better to write:

let a = set_high(); let res = 1./.3. in set_nearest (); res;;

The above code will NOT work on linux-x64 where many floating point functions are implemented using SSE instructions. These three functions should only be used when there is no other solution, and you really know what tou are doing, and this should never happen. Please use the regular functions of the fpu module for computations. For example prefer:

let a = fdiv_high 1. 3.;;

PS: The Interval module and the fpu module functions correctly set and restore the rounding mode for all interval computations, so you don't really need these functions.

PPS: Please, don't use them...

`val set_low : unit -> unit`

Sets the rounding mod to DOWNWARD (towards minus infinity)

`val set_high : unit -> unit`

Sets the rounding mod to UPWARD (towards infinity)

`val set_nearest : unit -> unit`

Sets the rounding mod to NEAREST (default mode)

2 Module `Fpu_rename_all` : Aliases floating point functions to their "constant" counterparts.

As described in the `Fpu` module documentation, there are problems when mixing some C-lib or ocaml native functions with interval programming on 64 bits machine.

The standard floating point functions results will always lie in the `low; high` interval computed by the Fpu module, but they are slightly different on 32 and 64 bits machines.

Using `Open Fpu_rename_all` at the beginning of your program guarantees that floating computation will give the same results on 32 and 64 bits machines. This is not mandatory but might help.

NB: while most transcendantal function are almost as fast, and sometimes faster than their "standard" ocaml counterparts, +. -. *. and /. are much slower (from 50% to 100% depending on the processor. If you want to rename transcendantal functions but not +. -. *. and /. then use the `Fpu_rename` module.

`val (+.) : float -> float -> float`

Computes x + y

`val (-.) : float -> float -> float`

Computes x - y

`val (/.) : float -> float -> float`

Computes x / y

`val ( *. ) : float -> float -> float`

Computes x * y

`val mod_float : float -> float -> float`

Computes x mod y

`val sqrt : float -> float`

square root function

`val log : float -> float`

log function

`val exp : float -> float`

exp function

`val ( ** ) : float -> float -> float`

Computes x^y

`val cos : float -> float`

Computes cos(x) for x in [-2^63, 2^63]

`val sin : float -> float`

Computes sin(x) for x in [-2^63, 2^63]

`val tan : float -> float`

Computes tan(x) for x in [-2^63, 2^63]

`val asin : float -> float`

arc-sinus function

`val acos : float -> float`

arc-cosine function

`val atan2 : float -> float -> float`

atan2 function

`val atan : float -> float`

arc-tan function

`val cosh : float -> float`

cosh function

`val sinh : float -> float`

sinh function

`val tanh : float -> float`

tanh function

3 Module `Fpu_rename` : Aliases floating point functions to their "constant" counterparts, except for "ordinary functions"

As described in the `Fpu` module documentation, there are problems when mixing some C-lib or ocaml native functions with interval programming on 64 bits machine.

The standard floating point functions results will always lie in the `low; high` interval computed by the Fpu module, but they are slightly different on 32 and 64 bits machines.

Using `Open Fpu_rename` at the beginning of your program guarantees that floating computation will give the same results on 32 and 64 bits machines for all transcendantal functions but not for ordinary arithmetic functions.

NB: while most transcendantal function are almost as fast, and sometimes faster than their "standard" ocaml counterparts, +. -. *. and /. are much slower (from 50% to 100% depending on the processor). If you want to rename also +. -. *. and /. then use the `Fpu_rename_all` module.

`val mod_float : float -> float -> float`

Computes x mod y

`val sqrt : float -> float`

square root function

`val log : float -> float`

log function

`val exp : float -> float`

exp function

`val ( ** ) : float -> float -> float`

Computes x^y

`val cos : float -> float`

Computes cos(x) for x in [-2^63, 2^63]

`val sin : float -> float`

Computes sin(x) for x in [-2^63, 2^63]

`val tan : float -> float`

Computes tan(x) for x in [-2^63, 2^63]

`val asin : float -> float`

arc-sinus function

`val acos : float -> float`

arc-cosine function

`val atan2 : float -> float -> float`

atan2 function

`val atan : float -> float`

arc-tan function

`val cosh : float -> float`

cosh function

`val sinh : float -> float`

sinh function

`val tanh : float -> float`

tanh function

4 Module `Interval` : Interval library in OCAML.

ONLY FOR INTEL PROCESSORS.

All operations use correct rounding.

It is not mandatory, but still wise, to read the documentation of the `Fpu` module

WARNING: even if some functions have been associated with operators, such as the interval addition which is associated with the `+$` operator, the priority order between +, * and functions is not maintained. You HAVE to use parenthesis if you want to be sure that `a +$ b *$ c` is properly computed as `a +$ (b *$ c)`.

This library has been mainly designed to be used in a branch and bound optimization algorithm. So, some choices have been made:

- NaN is never used. We either extend functions by pseudo continuity or raise exceptions. For example, `{low=2.;high=3.} /$ {low=0.;high=2.}` returns `{low=1.;high=Inf}`, while `{low=2.;high=3.} /$ {low=0.;high=0.}` or `{low=0.;high=0.} /$ {low=0.;high=0.}` raise a failure.
- Intervals [+Inf,+Inf] or [-Inf,-Inf] are never used and never returned.

- When using a float in the following operations, it must never be equal to +Inf or -Inf or Nan
- Functions such as `log`, `sqrt`, `acos` or `asin` are restricted to their definition domain but raise an exception rather than returning an empty interval: for example `sqrt_I {low=-4;high=4}` returns `{low=0;high=2}` while `sqrt_I {low=-4;high=-2}` will raise an exception.

Another design choice was to have non mutable elements in interval structure, and to maintain an "ordinary" syntax for operations, such as "let a = b+$c in" thus mapping interval computation formula on airthmetic formula. We could have instead chosen to have mutable elements, and to write for example (add_I_I a b c) to perform "a=b+$c". The first choice is, to our point of view, more elegant and easier to use. The second is more efficient, especially when computing functions with many temporary results, which force the GC to create and destroy lot of intervals when using the implementation we chose. Nothing's perfect.

The library is implemented in x87 assembly mode and is quite efficient (see below).

Intel Atom 230 Linux 32 bits:

- ftan speed (10000000 calls):2.528158
- fcos speed (10000000 calls):2.076129
- fsin speed (10000000 calls):1.972123
- tan_I speed (10000000 calls):4.416276
- cos_I speed (10000000 calls):4.936308
- sin_I speed (10000000 calls):5.396338
- fadd speed (10000000 calls):0.980062
- fsub speed (10000000 calls):0.980061
- fmul speed (10000000 calls):0.980061
- fdiv speed (10000000 calls):1.424089
- +$ speed (10000000 calls):1.656103
- -$ speed (10000000 calls):1.636103
- *$ speed (10000000 calls):4.568285
- /$ speed (10000000 calls):4.552285

Intel 980X Linux 64 bits:

- ftan speed (10000000 calls):0.472029
- fcos speed (10000000 calls):0.400025
- fsin speed (10000000 calls):0.400025
- tan_I speed (10000000 calls):0.752047

- cos_I speed (10000000 calls):1.036065
- sin_I speed (10000000 calls):1.104069
- fadd speed (10000000 calls):0.124008
- fsub speed (10000000 calls):0.120008
- fmul speed (10000000 calls):0.128008
- fdiv speed (10000000 calls):0.156010
- +$ speed (10000000 calls):0.340021
- -$ speed (10000000 calls):0.332021
- *$ speed (10000000 calls):0.556035
- /$ speed (10000000 calls):0.468029

```
type interval = {
  low : float ;
```
low bound

```
  high : float ;
```
high bound

```
}
```

The interval type. Be careful however when creating intervals. For example, the following code: `let a = {low=1./.3.;high=1./.3.}` creates an interval which does NOT contain the mathematical object 1/3.

If you want to create an interval representing 1/3, you have to write `let a = 1. /.$ {low=3.0;high=3.0}` because rounding will then be properly set

```
val zero_I : interval
```
Neutral element for addition

```
val one_I : interval
```
Neutral element for multiplication

```
val pi_I : interval
```
`pi` with bounds properly rounded

```
val e_I : interval
```
`e` with bounds properly rounded

```
val printf_I :
  (float -> string, unit, string) Pervasives.format ->
  interval -> unit
```

Prints an interval with the same format applied to both endpoints. Formats follow the same specification than the one used for the regular printf function

```
val fprintf_I :
  Pervasives.out_channel ->
  (float -> string, unit, string) Pervasives.format ->
  interval -> unit
```

Prints an interval into an out_channel with the same format applied to both endpoints

```
val sprintf_I :
  (float -> string, unit, string) Pervasives.format ->
  interval -> string
```

Returns a string holding the interval printed with the same format applied to both endpoints

`val float_i : int -> interval`

Returns the interval containing the float conversion of an integer

`val compare_I_f : interval -> float -> int`

`compare_I_f a x` returns 1 if `a.high<x`, 0 if `a.low<=x<=a.high` and -1 if `x<a.low`

`val size_I : interval -> float`

`size_I a` returns `a.high-a.low`

`val sgn_I : interval -> interval`

`sgn_I a` returns `{low=float (compare a.low 0.);high=float (compare a.high 0.)}`

`val truncate_I : interval -> interval`

`truncate_I a` returns `{low=floor a.low;high=ceil a.high}`

`val abs_I : interval -> interval`

`abs_I a` returns `{low=a.low;high=a.high}` if `a.low>=0.`, `{low=-a.high;high=-a.low}` if `a.high<=0.`, and `{low=0.;high=max -a.low a.high}` otherwise

`val union_I_I : interval -> interval -> interval`

`union_I_I a b` returns `{low=min a.low b.low;high=max a.high b.high}`

`val max_I_I : interval -> interval -> interval`

`max_I_I a b` returns `{low=max a.low b.low;high=max a.high b.high}`

`val min_I_I : interval -> interval -> interval`

`min_I_I a b` returns `{low=min a.low b.low;high=min a.high b.high}`

`val (+$) : interval -> interval -> interval`

`a +$ b` returns `{low=a.low+.b.low;high=a.high+.b.high}` `val (+$.) : interval -> float -> interval`

`a +$. x` returns `{low=a.low+.x;high=a.high+.x}`

`val (+.$) : float -> interval -> interval`

`x +.$ a` returns `{low=a.low+.x;high=a.high+.x}`

`val (-$) : interval -> interval -> interval`

`a -$ b` returns `{low=a.low-.b.high;high=a.high-.b.low}`

`val (-$.) : interval -> float -> interval`

`a -$. x` returns `{low=a.low-.x;high=a.high-.x}`

`val (-.$) : float -> interval -> interval`

`x -.$ a` returns `{low=x-.a.low;high=x-.a.high}`

`val (~-$) : interval -> interval`

`~-$ a` returns `{low=-a.high;high=-a.low}`

`val ( *$. ) : interval -> float -> interval`

`a *$. x` multiplies `a` by `x` according to interval arithmetic and returns the proper result. If `x=0.` then `zero_I` is returned

`val ( *.$ ) : float -> interval -> interval`

`x *$. a` multiplies `a` by `x` according to interval arithmetic and returns the proper result. If `x=0.` then `zero_I` is returned

`val ( *$ ) : interval -> interval -> interval`

`a *$ b` multiplies `a` by `b` according to interval arithmetic and returns the proper result. If `a=zero_I` or `b=zero_I` then `zero_I` is returned

`val (/$.) : interval -> float -> interval`

`a /$. x` divides `a` by `x` according to interval arithmetic and returns the proper result. Raise `Failure "/$."` if `x=0.`

`val (/.$) : float -> interval -> interval`

`x /.$ a` divides `x` by `a` according to interval arithmetic and returns the result. Raise `Failure "/.$"` if `a=zero_I`

`val (/$) : interval -> interval -> interval`

`a /$ b` divides the first interval by the second according to interval arithmetic and returns the proper result. Raise `Failure "/$"` if `b=zero_I`

`val mod_I_f : interval -> float -> interval`

`mod_I_f a f` returns `a` mod `f` according to interval arithmetic et ocaml mod_float definition. Raise `Failure "mod_I_f"` if `f=0.`

`val inv_I : interval -> interval`

`inv_I a` returns `1. /.$ a`. Raise `Failure "inv_I"` if `a=zero_I`

`val sqrt_I : interval -> interval`

`sqrt_I a` returns `{low=sqrt a;high=sqrt b}` if `a>=0.`, `{low=0.;high=sqrt b}` if `a<0.<=b`. Raise `Failure "sqrt_I"` if `b<0.`

`val pow_I_i : interval -> int -> interval`

`Pow_I_i a n` with `n` integer returns interval `a` raised to nth power according to interval arithmetic. If `n=0` then `{low=1.;high=1.}` is returned. Raise `Failure "pow_I_f"` if `n<=0` and `a=zero_I`. Computed with exp-log in base2

`val ( **$. ) : interval -> float -> interval`

`a **$. f` returns interval `a` raised to f power according to interval arithmetic. If `f=0.` then `{low=1.;high=1.}` is returned. Raise `Failure "**$."` if `f<=0. and a=zero_I` or if `f is not an integer value and a.high<0.`. Computed with exp-log in base2

`val ( **$ ) : interval -> interval -> interval`

`a **$ b` returns interval `a` raised to `b` power according to interval arithmetic, considering the restriction of x power y to x $\geq$ 0. Raise `Failure "**$"` if `a.high < 0` or `(a.high=0. and b.high<=0.)`

`val ( **.$ ) : float -> interval -> interval`

`x **.$ a` returns float `x` raised to interval `a` power according to interval arithmetic, considering the restiction of x power y to x $\geq$ 0. Raise `Failure "**.$"` if `x < 0` and `a.high <= 0`

`val log_I : interval -> interval`

`log_I a` returns `{low=log a.low; high=log a.high}` if `a.low>0.`, `{low=neg_infinity; high=log a.high}` if `a.low<0<=a.high`. Raise `Failure "log_I"` if `a.high<=0.`

`val exp_I : interval -> interval`

`exp_I a` returns `{low=exp a.high;high=exp b.high}`

`val cos_I : interval -> interval`

`cos_I a` returns the proper extension of cos to arithmetic interval Returns [-1,1] if one of the bounds is greater or lower than +/-2**53

`val sin_I : interval -> interval`

`sin_I a` returns the proper extension of sin to arithmetic interval Returns [-1,1] if one of the bounds is greater or lower than +/-2**53

`val tan_I : interval -> interval`

`tan_I a` returns the proper extension of tan to arithmetic interval Returns [-Inf,Inf] if one of the bounds is greater or lower than +/-2**53 `val acos_I : interval -> interval`

`acos_I a` raise `Failure "acos_I"` if `a.low>1. or a.high<-1.`, else returns `{low=if a.high<1. then acos a.high else 0; high=if a.low>-1. then acos a.low else pi}`. All values are in [0,pi].

`val asin_I : interval -> interval`

`asin_I a` raise `Failure "asin_I"` if `a.low>1. or a.high<-1.` else returns `{low=if a.low>-1. then asin a.low else -pi/2; high=if a.low<1. then asin a.high else pi/2}`. All values are in [-pi/2,pi/2].

`val atan_I : interval -> interval`

`atan_I a` returns `{low=atan a.low;high=atan a.high}`

`val atan2mod_I_I : interval -> interval -> interval`

`atan2mod_I_I y x` returns the proper extension of interval arithmetic to atan2 but with values in [-pi,2 pi] instead of [-pi,pi]. This can happen when y.low<0 and y.high>0 and x.high<0: then the returned interval is `{low=atan2 y.high x.high;high=(atan2 y.low x.high)+2 pi}`. This preserves the best inclusion function possible but is not compatible with the standard definition of atan2

`val atan2_I_I : interval -> interval -> interval`

Same function as above but when y.low<0 and y.high>0 and x.high<0 the returned interval is [-pi,pi]. This does not preserve the best inclusion function but is compatible with the atan2 regular definition

`val cosh_I : interval -> interval`

cosh_I is the proper extension of interval arithmetic to cosh

`val sinh_I : interval -> interval`

sinh_I is the proper extension of interval arithmetic to sinh

`val tanh_I : interval -> interval`

tanh_I is the proper extension of interval arithmetic to tanh

`val size_max_X : interval array -> float`

Computes the size of the largest interval of the interval vector

`val size_mean_X : interval array -> float`

Computes the mean of the size of intervals of the interval vector

```
val printf_X :
  (float -> string, unit, string) Pervasives.format ->
  interval array -> unit
```

Prints an interval vector with the same format applied to all endpoints.

```
val fprintf_X :
  Pervasives.out_channel ->
  (float -> string, unit, string) Pervasives.format ->
  interval array -> unit
```

Prints an interval vector into an out_channel with the same format applied to all endpoints

```
val sprintf_X :
  (float -> string, unit, string) Pervasives.format ->
  interval array -> string
```

Returns a string holding the interval vector printed with the same format applied to all endpoints

```
val print_X : interval array -> unit
```

Deprecated

```
val print_I : interval -> unit
```

Deprecated

```
val size_X : interval array -> float
```

Deprecated

```
val size2_X : interval array -> float
```

Deprecated

```
val (<$.) : interval -> float -> int
```

Deprecated

```
val pow_I_f : interval -> float -> interval
```

Deprecated

```
val pow_I_I : interval -> interval -> interval
```

Deprecated