Floating-point numbers

Version 1.0.3

15/02/2023

Laurent Bartholdi

Integration of mpfr, mpfi, mpc, fplll and cxsc in GAP

Laurent Bartholdi Email: laurent.bartholdi@gmail.com
Homepage: https://www.math.uni-sb.de/ag/bartholdi/

Address: FR Mathematik
D-66041 Saarbrücken
Germany

Abstract

This document describes the package Float, which implements in GAP arbitrary-precision floating-point numbers.

For comments or questions on Float please contact the author.

Copyright

Acknowledgements

Part of this work was supported by the "Swiss National Fund for Scientific Research (SNF)", the "German National Science Foundation (DFG)", and the Courant Research Centre "Higher Order Structures" of the University of Göttingen.

Contents

Chapter 1

Licensing

Chapter 2

Float package

2.1 A sample run

The extended floating-point capabilities of GAP are installed by loading the package via `LoadPackage("float");` and selecting new floating-point handlers via `SetFloats(MPFR)`, `SetFloats(MPFI)`, `SetFloats(MPC)` or`SetFloats(CXSC)`, depending on whether high-precision real, interval or complex arithmetic are desired, or whether a fast package containing all four real/complex element/interval arithmetic is desired:

Example

```
gap> LoadPackage("float");
Loading FLOAT 0.7.0 ...
true
gap> SetFloats(MPFR); # floating-point
gap> x := 4*Atan(1.0);
.314159e1
gap> Sin(x);
.169569e-30
gap> SetFloats(MPFR,1000); # 1000 bits
gap> x := 4*Atan(1.0);
.314159e1
gap> Sin(x);
.125154e-300
gap> String(x,300);
".314159265358979323846264338327950288419716939937510582097494459230781640628\
620899862803482534211706798214808651328230664709384460955058223172535940812848\
111745028410270193852110555964462294895493038196442881097566593344612847564823\
378678316527120190914564856692346034861045432664821339360726024914127e1"
gap>
gap> SetFloats(MPFI); # intervals
gap> x := 4*Atan(1.0);
.314159e1(99)
gap> AbsoluteDiameter(x); Sup(x); Inf(x);
.100441e-29
.314159e1
.314159e1
gap> Sin(x);
-.140815e-29(97)
gap> 0.0 in last;
```

```
true
gap> 1.0; # exact representation
.1e1(inf)
gap> IncreaseInterval(last,0.001); # now only 8 significant bits
.1e1(8)
gap> IncreaseInterval(last,-0.002); # now becomes empty
\emptyset
gap> r2 := Sqrt(2.0);
.141421e1(99)
gap> MinimalPolynomial(Rationals,r2);
-2*x_1^2+1
gap> Cyc(r2);
E(8)-E(8)^3
gap>
gap> SetFloats(MPC); # complex numbers
gap> z := 5.0-1.0i;
.5e1-.1e1i
gap> (1+1.0i)*last^4*(239+1.0i);
.228488e6
gap> Exp(6.2835i);
.1e1+.314693e-3i
```

Chapter 3

Polynomials

3.1 The Floats pseudo-field

Polynomials with floating-point coefficients may be manipulated in GAP; though they behave, in subtle ways, quite differently than polynomials over rings. A "pseudo-field" of floating-point numbers is available to create them using the standard GAP syntax.

3.1.1 FLOAT_PSEUDOFIELD

▷ FLOAT_PSEUDOFIELD (global variable)

The "pseudo-field" of floating-point numbers, containing all floating-point numbers in the current implementation.

Note that it is not really a field, e.g. because addition of floating-point numbers is not associative. It is mainly used to create indeterminates, as in the following example:

Example

```
gap> x := Indeterminate(FLOAT_PSEUDOFIELD,"x");
x
gap> 2*x^2+3;
2.0*x^2+3.0
gap> Value(last,10);
203.0
```

3.2 Roots of polynomials

The Jenkins-Traub algorithm has been implemented, in arbitrary precision for MPFR and MPC.

Furthermore, CXSC can provide complex enclosures for the roots of a complex polynomial.

3.3 Finding integer relations

The PSLQ algorithm has been implemented by Steve A. Linton, as an external contribution to Float. This algorithm receives as input a vector of floats x and a required precision ε, and seeks an integer vector v such that $|x \cdot v| < \varepsilon$. The implementation follows quite closely the original article [BB01].

3.3.1 PSLQ

▷ PSLQ(*x*, *epsilon*[, *gamma*]) (function)

▷ PSLQ_MP(*x*, *epsilon*[, *gamma*[, *beta*]]) (function)

Returns: An integer vector v with $|x \cdot v| < \varepsilon$.

The PSLQ algorithm by Bailey and Broadhurst (see [BB01]) searches for an integer relation between the entries in x.

β and γ are algorithm tuning parameters, and default to $4/10$ and $2/\sqrt{(}3)$ respectively.

The second form implements the "Multi-pair" variant of the algorithm, which is better suited to parallelization.

Example

```
gap> PSLQ([1.0,(1+Sqrt(5.0))/2],1.e-2);
[ 55, -34 ] # Fibonacci numbers
gap> RootsFloat([1,-4,2]*1.0);
[ 0.292893, 1.70711 ] # roots of 2x^2-4x+1
gap> PSLQ(List([0..2],i->last[1]^i),1.e-7);
[ 1, -4, 2 ] # a degree-2 polynomial fitting well
```

3.4 LLL lattice reduction

A faster implementation of the LLL lattice reduction algorithm has also been implemented. It is accessible via the commands `FPLLLReducedBasis(m)` and `FPLLLShortestVector(m)`.

Chapter 4

Implemented packages

4.1 MPFR

4.1.1 IsMPFRFloat

▷ `IsMPFRFloat` (filter)
▷ `TYPE_MPFR` (global variable)

The category of floating-point numbers.
Note that they are treated as commutative and scalar, but are not necessarily associative.

4.2 MPFI

4.2.1 IsMPFIFloat

▷ `IsMPFIFloat` (filter)
▷ `TYPE_MPFI` (global variable)

The category of intervals of floating-point numbers.
Note that they are treated as commutative and scalar, but are not necessarily associative.

4.3 MPC

4.3.1 IsMPCFloat

▷ `IsMPCFloat` (filter)
▷ `TYPE_MPC` (global variable)

The category of intervals of floating-point numbers.
Note that they are treated as commutative and scalar, but are not necessarily associative.

4.4 CXSC

4.4.1 IsCXSCReal

▷ IsCXSCReal (filter)
▷ IsCXSCComplex (filter)
▷ IsCXSCInterval (filter)
▷ IsCXSCBox (filter)
▷ TYPE_CXSC_RP (global variable)
▷ TYPE_CXSC_CP (global variable)
▷ TYPE_CXSC_RI (global variable)
▷ TYPE_CXSC_CI (global variable)

The category of floating-point numbers.

Note that they are treated as commutative and scalar, but are not necessarily associative.

4.5 FPLLL

4.5.1 FPLLLReducedBasis

▷ FPLLLReducedBasis(*m*) (operation)

Returns: A matrix spanning the same lattice as *m*.

This function implements the LLL (Lenstra-Lenstra-Lovász) lattice reduction algorithm via the external library fplll.

The result is guaranteed to be optimal up to 1%.

4.5.2 FPLLLShortestVector

▷ FPLLLShortestVector(*m*) (operation)

Returns: A short vector in the lattice spanned by *m*.

This function implements the LLL (Lenstra-Lenstra-Lovász) lattice reduction algorithm via the external library fplll, and then computes a short vector in this lattice.

The result is guaranteed to be optimal up to 1%.

References

[BB01] D. H. Bailey and D. J. Broadhurst. Parallel integer relation detection: techniques and applications. *Math. Comp.*, 70(236):1719–1736 (electronic), 2001. 7, 8

Index