GAP 4 Package FUtil

Various utilities for use with GAP

0.1.5

August 2017

Frank Lübeck

Frank Lübeck Email: Frank.Luebeck@Math.RWTH-Aachen.De
Homepage: http://www.math.rwth-aachen.de/~Frank.Luebeck
Address: Lehrstuhl D für Mathematik RWTH Aachen Templergraben 64 52062 Aachen Germany

Contents

1 Decimal Approximations of Cylotomic Numbers **3**
1.1 Complex Numbers 3
1.2 Decimal Approximations of Real Numbers 4

2 Interface for Declarations and Method Installations **8**
2.1 Declaration and Installation Functions 9

References **11**

Index **12**

Chapter 1

Decimal Approximations of Cylotomic Numbers

The FUtil package provides some functionality for computing with fixed point approximations of real numbers. The main motivation is the question of deciding if a real cyclotomic number in GAP is positive, see the function `HasPositiveRealPartCyc` (1.2.6).

The idea for this functionalty comes from the deposited contribution `decimal.g` provided by Jean Michel for GAP 3. As an enhancement the FUtil package implements an interval arithmetic with the approximate numbers, such that a comparison function as `TruelyLessDecimal` (1.2.5) can be implemented.

The package also provides the function `ComplexNumber` (1.1.2) which creates objects which have a *real* and *imaginary* part and add and multiply like complex numbers.

1.1 Complex Numbers

1.1.1 IsComplexNumber

▷ `IsComplexNumber(`*c*`)` (Filter)

Returns: `true` or `false`.

▷ `RealPart(`*c*`)` (attribute)

▷ `ImaginaryPart(`*c*`)` (attribute)

Returns: ring elements.

`IsComplexNumber` returns `true` for additive and multiplicative elements *c* which have the attributes `RealPart` and `ImaginaryPart` and are added and multiplied like complex numbers.

The real and imaginary parts themselves can be any ring elements like real cyclotomic numbers, but also decimal approximations of numbers, polynomials or others.

1.1.2 ComplexNumber

▷ `ComplexNumber(`*re*`, `*im*`)` (operation)

▷ `ComplexNumber(`*c*`)` (method)

Returns: an object in `IsComplexNumber` (1.1.1).

In the first general form this operation returns an object `a` such that `IsComplexNumber(a)` is true and *re* and *im* are its `RealPart` (1.1.1) and `ImaginaryPart` (1.1.1), respectively.

For cyclotomic numbers (see IsCyc (**Reference: IsCyc**)) *c* the second mentioned method returns the complex number with the real and imaginary parts as cyclotomic numbers.

For other *c* the one argument form is equivalent to a call with arguments *c* and 0*ca.

Example

```
gap> a := 1/2 * ComplexNumber(Sqrt(2), -Sqrt(2));
1/2*E(8)-1/2*E(8)^3-I*(1/2*E(8)+1/2*E(8)^3)
gap> a^2;
-I
gap> x := Indeterminate(Rationals, "x");;
gap> b := ComplexNumber(1+x,3-x^2);
x+1-I*(x^2+3)
gap> b^2;
-x^4+7*x^2+2*x-8-I*(2*x^3-2*x^2+6*x+6)
gap> c := ComplexNumber(E(7));
1/2*E(7)+1/2*E(7)^6+I*(1/2*E(28)^3-1/2*E(28)^11)
gap> c^7;
1
```

1.2 Decimal Approximations of Real Numbers

1.2.1 IsDecimalApproximation

▷ IsDecimalApproximation(a) (Filter)

Returns: true or false.

▷ Mantissa(a) (attribute)

▷ Precision(a) (attribute)

▷ Epsilon(a) (attribute)

Returns: integers.

IsDecimalApproximation returns true for additive and multiplicative elements a which have attributes Mantissa, Precision and Epsilon. Such an element is interpreted as a real number which has distance at most Epsilon(a) / 10^Precision(a) from the number Mantissa(a) / 10^Precision(a). Arithmetic with such numbers is done via an interval arithmetic.

If a lot of arithmetic is done with such decimal approximations the value of Epsilon can become big, since with each operation worst case estimates for the error are done.

In arithmetic expressions we allow rational numbers as operands, they are implicitly substituted by approximations of the same precision as the other operand.

1.2.2 DecimalApproximation

▷ DecimalApproximation(*c*[, *prec*]) (operation)

Returns: an object in IsDecimalApproximation (1.2.1) or in IsComplexNumber (1.1.1) with real and imaginary part in IsDecimalApproximation (1.2.1).

With this operation real approximations can be created. If the second argument *prec* is given it will be the precision of the result. If it is omitted, then the current value of DefaultDecimalPrecision (which is set to 10 while loading the package) will be used instead.

There are methods for the following types of *c*:

real cyclotomic

returns a decimal approximation, see the comment below for what this means if *c* is not rational.

non-real cyclotomic
the result is in `IsComplexNumber` (1.1.1) and the real and imaginary parts are decimal approximations.

complex number
the result is a complex number with `DecimalApproximation` applied to the real and imaginary parts.

decimal approximation
returns a decimal approximation, maybe with another precision.

Note that in GAP each cyclotomic number has well defined interpretation within the complex numbers, `E(n)` is considered as $e^{2\pi i/n}$. So, the results of `DecimalApproximation` are also well defined for cyclotomic numbers.

Example

```
gap> DefaultDecimalPrecision;
10
gap> a := DecimalApproximation(1/3);
0.333333333
gap> Mantissa(a); Precision(a); Epsilon(a);
3333333333
10
1
gap> 3/4*a;
0.25
gap> # see also 'SqrtDecimalApproximation'
gap> DecimalApproximation(Sqrt(3), 50);
1.73205080756887729352744634150587236694280525381 02
gap> b := DecimalApproximation(E(20));
0.951056516+I*0.309016995
gap> b^5;
-0.000000002+I*1.000000001
gap> DecimalApproximation(last, 3);
I
```

1.2.3 SqrtDecimalApproximation

▷ `SqrtDecimalApproximation(`*r*`, `*prec*`)` (function)

Returns: decimal approximation.

For a positive rational number or a positive real approximation *r* this function computes the square root to precision *prec* via Newton approximation. This can be much faster than via a representation of the square root as cyclotomic number.

Example

```
gap> # use
gap> SqrtDecimalApproximation(2, 50);
1.41421356237309504880168872420969807856967187537 69
gap> # instead of
gap> DecimalApproximation(Sqrt(2), 50);
1.41421356237309504880168872420969807856967187537 69
```

1.2.4 PiDecimalApproximation

▷ `PiDecimalApproximation(prec)` (function)

Returns: decimal approximation of π.

This function computes the decimal approximation of π to precision *prec*. It uses a recursion formula with exponential convergence, see [Koe87].

Example

```
gap> PiDecimalApproximation(707);
3.14159265358979323846264338327950288419716939937510582097\
49445923078164062862089986280348253421170679821480865132 82\
30664709384460955058223172535940812848111745028410270193 85\
21105559644622948954930381964428810975665933446128475648 23\
37867831652712019091456485669234603486104543266482133936 07\
26024914127372458700660631558817488152092096282925409171 53\
64367892590360011330530548820466521384146951941511609433 05\
72703657595919530921861173819326117931051185480744623799 62\
74956735188575272489122793818301194912983367336244065664 30\
86021394946395224737190702179860943702770539217176293176 75\
23846748184676694051320005681271452635608277857713427577 89\
60917363717872146844090122495343014654958537105079227968 92\
589235420199
```

1.2.5 TruelyLessDecimal

▷ `TruelyLessDecimal(a, b)` (function)

Returns: `true`, `false` or `fail`.

The arguments `a` and `b` must be decimal approximations of real numbers. Recall that this means that they are real numbers which are only specified by an interval of rational numbers. This function returns `fail` if the interval overlap. It returns `true` if all numbers in the interval specifying `a` are strictly smaller than all numbers in the interval specifying `b`. Otherwise `false` is returned.

Example

```
gap> a := SqrtDecimalApproximation(2,30);
1.414213562373095048801688724 21
gap> b := SqrtDecimalApproximation(3,20);
1.732050807568877293 5
gap> TruelyLessDecimal(a,b);
true
gap> TruelyLessDecimal(a^2,b^2-1);
fail
```

1.2.6 HasPositiveRealPartCyc

▷ `HasPositiveRealPartCyc(c)` (function)

Returns: `true` or `false`.

The argument *c* must be a cyclotomic number. This function returns true if the real part of this number is greater than zero.

The algorithm is to compute decimal approximations of the number with growing precision until the question can be decided.

(Recall that in GAP cyclotomic numbers are considered with a well defined embedding into the complex numbers, see `DecimalApproximation` (1.2.2).)

Example

```
gap> HasPositiveRealPartCyc(0);
false
gap> HasPositiveRealPartCyc(E(7));
true
gap> HasPositiveRealPartCyc(E(3));
false
```

Chapter 2

Interface for Declarations and Method Installations

The utilities in this Chapter provide an access to GAPs object type - operations - method system which is a bit different from the conventions used in the GAP library. We summarize the main differences.

Families.

It is compulsory in GAP to put each object into a *family*. For example this must be specified with the creation of a *type* of an object, see `FamilyObj` (**Reference: FamilyObj**), `NewFamily` (**Reference: NewFamily**), `NewType` (**Reference: NewType**). But for many objects its familiy seems essentially irrelevant in practice. Here, we define a `DefaultFamily` (2.1.1) and interface functions like `MakeType` (2.1.2) which implicitly include this default family if GAP needs one.

Declaration of operations.

In the GAP library operations are declared with prescribing numbers of arguments and giving constraints on the type of objects for which methods are allowed to install (several declarations of the same operation are possible), see `DeclareOperation` (**Reference: DeclareOperation**). Calls of `InstallMethod` (**Reference: InstallMethod**) for not declared types of object lead to an error.

This is a debugging feature for the method installation. It is irrelevant for the actual method selection and can in fact be circumvented by using `InstallOtherMethod` (**Reference: InstallOtherMethod**). Using the functions `MakeOperation` (2.1.3) and `NewMethod` (2.1.5) one can ignore this debugging feature.

Documentation of methods.

With the installation of a method, see `InstallMethod` (**Reference: InstallMethod**), one can specify a comment string describing for which types of objects a method is applicable. The actual applicability is specified by a filter for each argument - note that some non-simple filters may not be printed in a nice form (try for example `NamesFilter(IsMatrix)`). This together leads for example to difficult to understand information with `ApplicableMethod` (**Reference: ApplicableMethod**).

Our function `NewMethod` (2.1.5) takes strings describing the types of the allowed arguments of a method. These strings are also used for (precise) comments.

Several installations of a method.

Sometimes a method can be used for several types of arguments or one should install a method

several times to get the rankings with respect to several other methods right. This can be done with one call of `NewMethod` (2.1.5) via `or`-conjunctions in the strings describing the argument types.

Compound declarations.
We provide functions that allow to declare several operations, variables, attributes, ..., in one call.

2.1 Declaration and Installation Functions

2.1.1 DefaultFamily

▷ `DefaultFamily` (family)

This is a family used for all new types of objects created with `MakeType` (2.1.2). For many applications you don't need to know about this at all.

2.1.2 MakeType

▷ `MakeType(`*`filt[, data]`*`)` (function)

Returns: an object type.

This creates an object type for which the filters given by *`filt`* are set (including their implied filters). Optionally, a type can also carry an additional arbitrary *`data`* object.

This essentially calls `NewType` (**Reference: NewType**) using `DefaultFamily` (2.1.1) as family.

2.1.3 MakeOperation

▷ `MakeOperation(`*`nam1[, nam2, ...]`*`)` (function)
▷ `MakeAttribute(`*`nam1[, nam2, ...]`*`)` (function)
▷ `MakeGlobalVariable(`*`nam1[, nam2, ...]`*`)` (function)
▷ `MakeGlobalFunction(`*`nam1[, nam2, ...]`*`)` (function)

All arguments must be strings. For each argument *`nam1`*, *`nam2`*, ..., these functions declare one operation, attribute, global variable or global function, respectively, with the given name. In case of operations and attributes, these are declared without constraints on the number or type of arguments of corresponding methods.

Note that the variables with names *`nam1`*, ..., are make read-only by the declaration. This is very useful to avoid accidental overwriting by a user, or for detecting compatibility problems with the GAP library or packages.

2.1.4 MakeProperty

▷ `MakeProperty(`*`nam, impl[, rk]`*`)` (function)

The argument *`nam`* must be a string, *`impl`* must be a filter, and *`rk`* a non-negative integer. This function does several things:

- declares a property with name *`nam`* and rank offset *`rk`*.

- installs *impl* as implied filter of the property.
- installs a setter method that takes only one argument and sets the new property in the argument to true.

Compare with DeclareProperty (**Reference: DeclareProperty**).

2.1.5 NewMethod

▷ NewMethod(*oper, filt, fun[, rk]*) (function)

This function is an interface to the more elaborate function InstallOtherMethod (**Reference: InstallOtherMethod**) in the GAP library.

The argument *oper* must be an operation, *filt* must be a string or list of strings, *fun* a function, and the optional argument *rk* a non-negative integer. Giving a string as *filt* is equivalent to giving a list with this string as single entry.

The strings given in *filt* are split at substrings "or" which are surrounded by whitespace. The remaining substrings each must evaluate (with EvalString (**Reference: EvalString**)) to a filter. The function *fun* is then installed as method for the operation *oper* for all combinations of filters resulting from the splitting and evaluation of the strings in *filt*.

The argument *rk* is used as offset for the rank of the method(s). Its default value, if not given, is 0. One should try to avoid using this optional argument whenever possible. Instead multiple installations of the same method may be more sensible.

References

[Koe87] Max Koecher. *Klassische elementare Analysis*. Birkhäuser Verlag, Basel, 1987. 6

Index

ComplexNumber, 3
    IsCyc, 3

DecimalApproximation, 4
DefaultFamily, 9

Epsilon, 4

HasPositiveRealPartCyc, 6

ImaginaryPart, 3
IsComplexNumber, 3
IsDecimalApproximation, 4

MakeAttribute, 9
MakeGlobalFunction, 9
MakeGlobalVariable, 9
MakeOperation, 9
MakeProperty, 9
MakeType, 9
Mantissa, 4

NewMethod, 10

PiDecimalApproximation, 6
Precision, 4

RealPart, 3

SqrtDecimalApproximation, 5

TruelyLessDecimal, 6