Pure-Rational - Rational number library for the Pure programming language

Author: Rob Hubbard
Author: Albert Graef <Dr.Graef@t-online.de>
Author: Jiri Spitz <jiri.spitz@bluetone.cz>
Date: 2010-03-18

Contents

This module contains a Pure port of Q+Q Rob Hubbard's rational number library for the Q programming language (see <http://q-lang.sourceforge.net/addons.html>). This package contains `rational.pure`, a collection of utility functions for rational numbers, and `rat_interval.pure`, a module for doing interval arithmetic needed by `rational.pure`. These modules are designed to work with the `math.pure` module (part of the standard Pure library), which contains the definition of the `rational` type and implements the basic rational arithmetic.

1 Copying

2 Installation

Run `make install` (as root) to install it in the Pure library directory. This requires GNU make, and of course you need to have Pure installed. Pure has to be built with support for the GNU Multiprecision Library (GMP) to use rational numbers.

`make install` tries to guess your Pure installation directory. If it gets this wrong, you can install using `make install prefix=/usr` which sets the installation prefix. Please see the Makefile for details.

3 Introduction

3.1 The Rational Module

The module defines a type `rational` for Albert Graef's Pure programming language (<http://code.google.com/p/pure-lang/>). The module is compatible with Pure version 0.43 (onwards).

3.2 The Files and the Default Prelude

The implementation of the `rational` type and associated utilities is distributed across various files.

3.2.1 `Math.pure` and Other Files

The file `math.pure` defines the type, its constructors and 'deconstructors' and basic arithmetical and mathematical operators and functions. This is part of the standard Pure library. A few definitions associated with `rationals` are defined in the appropriate file in the default prelude. For example: the type tests are contained in `primitives.pure`.

It is also possible to create rational complex numbers (in addition to 'double' complex numbers and integral or Gaussian complex numbers). That is `rational` plays nicely with `complex`, as provided by Albert Graef in the `math.pure` module. This is discussed further in Rational Complex Numbers.

3.2.2 `Rational.pure`

Additional 'Rational utilities', not included in the `math.pure` module, are defined in `rational.pure`. The functions include further arithmetical and mathematical operators and functions, continued fraction support, approximation routines and string formatting and evaluation.

The Rational utilities include some 'rational complex number' functions.

3.2.3 `Rat_interval.pure`

Amongst the Rational utilities are some functions that return a rational interval. The file `rat_interval.pure` is a partial implementation of interval arithmetic, and is not included in the default prelude. Intervals are discussed further in Intervals.

3.3 Notation

Throughout this document, the parameters q, q0, q1,. . . usually denote `rationals` ($\in \mathbf{Q}$), parameters z,. . . usually denote integers ($\in \mathbf{Z}$; `integers`), r,. . . usually denote real numbers ($\in \mathbf{R}$; `reals`), c,. . . usually denote complex numbers ($\in \mathbf{C}$), n,. . . usually denote parameters of any numeric type, v ,. . . usually denote parameters of any interval type, and x,. . . usually denote parameters of any type.

The `reals` are not just the `doubles`, but include `rationals` and `integers`. The term 'rational' usually refers to a rational number $\in \mathbf{Q} \supset \mathbf{Z}$, or an expression of type `rational` or `integer`.

3.4 Acknowledgements

Thank you to Dr Albert Graef for helpful feedback on the Q language and for answering my many questions. Albert performed the organisational work, splitting my original main source file for partial inclusion in the default prelude, and provided some of the 'architectural' functions required for smooth and consistent integration with his Q system.

Thanks to various members of the Q users mailing list (<https://lists.sourceforge.net/lists/listinfo/q-langusers>) for the feedback and suggestions.

4 The `rational` Type

4.1 Constructors

`rationals` are constructed with the function rational.

rational (z1, z2): given a pair of integers (z1, z2), this returns the `rational` equivalent to the fraction z1/z2. This is the inverse (up to equivalence) of num_den (see 'Deconstructors').

Example 1 Constructing a fraction:

```
> rational (44, 14);
22%7
>
```

rational z: given an integer z, this returns the `rational` equivalent to the `integer` z.

Example 2 Converting from an `integer`:

```
> rational 3;
3%1
>
```

(%) n1 n2: is a `rational`-aware division function, which may be used as a constructor (for `integers` n1 and n2). This is described in More on Division.

4.2 'Deconstructors'

A rational number is in simplest form if the numerator and denominator are coprime (i.e. do not have a factor in common) and the denominator is positive (and, specifically, non-zero). Sometimes the term 'irreducible' is used for a rational in simplest form. This is a property of the representation of the rational number and not of the number itself.

num q: given a `rational` or `integer` q, returns the '(signed) simplest numerator', i.e. the numerator of the normalised form of q.

den q: given a `rational` or `integer` q, returns the '(positive) simplest denominator', i.e. the denominator of the normalised form of q.

num_den q: given a `rational` or `integer` q, returns a pair (n, d) containing the (signed) simplest numerator n and the (positive) simplest denominator d. This is the inverse (up to equivalence) of rational as defined on integer pairs (see Constructors).

Example 3 Using num_den to obtain a representation in simplest form:

```
> let q = (44%(-14));
> num q;
-22
> den q;
7
> num_den q;
(-22,7)
> num_den 3;
(3,1)
> num_den (-3);
(-3,1)
>
```

Together, num and den are a pair of 'decomposition' operators, and num_den is also a decomposition operator. There are others (see Decomposition). The integer and fraction function (see Integer and Fraction Parts) may be used in conjunction with num_den_gauss to decompose a `rational` into integer, numerator and denominator parts.

4.3 Type and Value Tests

The functions rationalp and ratvalp and other rational variants are new for `rationals` and the standard functions exactp and inexactp are extended for `rationals`.

A value is 'exact', or of an exact type, if it is of a type that is able to represent the values returned by arithmetical operations exactly; in a sense, it is 'closed' under arithmetical operations. Otherwise, a value is 'inexact'. Inexact types are able to store some values only approximately.

`double` is not an exact type. The results of some operations on some values that are stored exactly, can't be stored exactly. (Furthermore, `doubles` are intended to represent real numbers; no irrational number ($\in \mathbf{R} \setminus \mathbf{Q}$) can be stored exactly as a `double`; even some rational ($\in \mathbf{Q}$) numbers are not stored exactly.)

`rational` is an exact type. All rational numbers (subject to available resources, of course) are stored exactly. The results of the arithmetical operations on `rationals` are `rationals` represented exactly. Beware that the standard `intvalp` and ratvalp may return 1 even if the value is of `double` type. However, these functions may be combined with exactp.

exactp x: returns whether x has an exact value.

inexactp x: returns whether x has an inexact value.

rationalp x: returns whether x is of type `rational`.

ratvalp x: returns whether x has a rational value.

Example 4 Rational value tests:

```
> let l = [9, 9%1, 9%2, 4.5, sqrt 2, 1+i, inf, nan];
> map exactp l;
[1,1,1,0,0,1,0,0]
> map inexactp l
[0,0,0,1,1,0,1,1]
> map rationalp l;
[0,1,1,0,0,0,0,0]
> map ratvalp l;
[1,1,1,1,1,0,0,0]
> map (\x -> (exactp x && ratvalp x)) l // "has exact rational value"
[1,1,1,0,0,0,0,0]
> map intvalp l // for comparison
[1,1,0,0,0,0,0,0]
> map (\x -> (exactp x && intvalp x)) l // "has exact integer value"
[1,1,0,0,0,0,0,0]
>
```

See Rational Complex Numbers for details about rational complex numbers, and Rational Complex Type and Value Tests for details of their type and value tests.

4.4 Internal Representation

It is not appropriate to give details of the internal representation of rational numbers as implemented by this module. That is subject to change.

The module makes no *guarantee* as to whether the internal representation is in simplest form, how the sign is handled, nor what components are stored (integer part, sign, numerator and denominator of fraction part,. . .).

The representation is private, therefore you can not operate on rationals except through the public interface. However, it is useful to be able to understand responses given by the module when debugging.

See, for example, 'Deconstructors' for details of a public interface function providing a representation with guaranteed properties.

5 Arithmetic

5.1 Operators

The standard arithmetic operators (+), (−) and (*) are overloaded to have at least one `rational` operand. If both operands are `rational` then the result is `rational`. If one operand is `integer`, then the result is `rational`. If one operand is `double`, then the result is `double`. The operators (/) and (%) are overloaded for division on at least one `rational` operand. The value returned by (/) is always `inexact` (in the sense of Type and Value Tests). The value returned by (%) is `exact` (if it exists). The standard function `pow` is overloaded to have a `rational` left operand. If `pow` is passed `integer` operands where the right operand is negative, then a `rational` is returned. The right operand should be an `integer`; negative values are permitted (because $q^{-z} = 1/q^z$). It is not overloaded to also have a `rational` right operand because such values are not generally rational (e.g. $q^{1/n} = \sqrt[n]{q}$). The standard arithmetic operator (^) is also overloaded, but produces a `double` value (as always).

The values of `pow 0 0` and `0^0` (with `integer` or `rational` zeroes) are left undefined.

Example 5 Arithmetic:

```
> 5%7 + 2%3;
29%21
> str_mixed ans;
"1+8/21"
> 1 + 2%3;
5%3
> ans + 1.0;
2.66666666666667
> 3%8 - 1%3;
1%24
> (11%10) ^ 3;
1.331
> pow (11%10) 3;
1331%1000
> pow 3 5;
243
> pow 3 (-5);
1%243
>
```

(See the function str_mixed .)

Beware that (/) on `integers` will not produce a `rational` result.

Example 6 Division:

```
> 44/14;
3.14285714285714
> 44%14;
22%7
> str_mixed ans;
```

```
"3+1/7"
```

(See the function str_mixed .)

5.2 More on Division

There is a `rational`-aware divide operator on the numeric types:

n1 % n2: returns the quotient ($\in \mathbf{Q}$) of n1 and n2. If n1 and n2 are `rational` or `integer` then the result is `rational`. This operator has the precedence of division (/).

Example 7 Using % like a constructor:

```
> 44 % 14;
22%7
> 2 + 3%8; // "2 3/8"
19%8
> str_mixed ans;
"2+3/8"
>
```

(See the function str_mixed .)

reciprocal n: returns the reciprocal of n: 1/n.

Example 8 Reciprocal:

```
> reciprocal (22%7);
7%22
>
```

The following division functions are parameterised by a rounding mode roundfun. The available rounding modes are described in Rounding to Integer.

divide roundfun n d: for `rationals` n and d returns a pair (q, r) of 'quotient' and 'remainder' where q is an `integer` and r is a `rational` such that $|r| < |d|$ (or better) and $n = q * d + r$. Further conditions may hold, depending on the chosen rounding mode roundfun (see Rounding to Integer). If roundfun = floor then $0 \leq r < d$. If roundfun = ceil then $-d < r \leq 0$. If roundfun = trunc then $|r| < |d|$ and sgn r $\in \{0, \text{sgn } d\}$. If roundfun = round, roundfun = round_zero_bias or roundfun = round_unbiased then $|r| \leq d/2$.

quotient roundfun nN d: returns just the quotient as produced by divide roundfun n d.

modulus roundfun n d: returns just the remainder as produced by divide roundfun n d.

q1 div q2: (overload of the built-in div) q1 and q2 may be `rational` or `integer`. Returns an `integer`.

q1 mod q2: (overload of the built-in mod) q1 and q2 may be `rational` or `integer`. Returns a `rational`. If q = q1 div q2 and r = q1 mod q2 then $q1 = q * q2 + q$, $q \in \mathbf{Z}$, $|r| < |q2|$ and sgn r $\in \{0, \text{sgn } q2\}$.

5.3 Relations — Equality and Inequality Tests

The standard arithmetic operators (==), (~=), (<), (<=), (>), (>=) are overloaded to have at least one `rational` operand. The other operand may be `rational`, `integer` or `double`.

Example 9 Inequality:

```
> 3%8 < 1%3;
0
>
```

5.4 Comparison Function

cmp n1 n2: is the 'comparison' (or 'compare') function, and returns sgn (n1 − n2); that is, it returns −1 if n1 < n2, 0 if n1 = n2, and +1 if n1 > n2.

Example 10 Compare:

```
> cmp (3%8) (1%3);
1
>
```

6 Mathematical Functions

Most mathematical functions, including the elementary functions (sin, $\sin^{-1}$, sinh, $\sinh^{-1}$, cos,. . . , exp, ln,. . .), are not closed on the set of rational numbers. That is, most mathematical functions do not yield a rational number in general when applied to a rational number. Therefore the elementary functions are not defined for `rationals`. To apply these functions, first apply a cast to `double`, or compose the function with a cast.

6.1 Absolute Value and Sign

The standard `abs` and `sgn` functions are overloaded for `rationals`.

abs q: returns absolute value, or magnitude, |q| of q; abs q = |q| = q × sgn q (see below).

sgn q: returns the sign of q as an `integer`; returns −1 if q < 0, 0 if q = 0, +1 if q > 0.

Together, these functions satisfy the property ∀q • (sgn q) * (abs q) = q (i.e. ∀q • (sgn q) * |q| = q). Thus these provide a pair of 'decomposition' operators; there are others (see Decomposition).

6.2 Greatest Common Divisor (GCD) and Least Common Multiple (LCM)

The standard functions `gcd` and `lcm` are overloaded for `rationals`, and mixtures of `integer` and `rational`.

gcd n1 n2: The GCD is also known as the Highest Common Factor (HCF). The GCD of rationals q1 and q2 is the largest (therefore positive) rational f such that f divides into both q1 and q2 exactly, i.e. an integral number of times. This is not defined for n1 and n2 both zero. For integral q1 and q2, this definition coincides with the usual definition of GCD for integers.

Example 11 With two `rationals`:

```
> let a = 7%12;
> let b = 21%32;
> let f = gcd a b;
> f;
7%96
> a % f;
8%1
> b % f;
9%1
>
```

Example 12 With a `rational` and an `integer`:

```
> let f = gcd (6%5) 4;
> f;
2%5
> (6%5) % f;
3%1
> 4 % f;
10%1
>
```

Example 13 With integral `rationals` and with `integers`:

```
> gcd (rational 18) (rational 24);
6%1
> gcd 18 24;
6
>
```

Example 14 The behaviour with negative numbers:

```
> gcd (rational (-18)) (rational 24);
6%1
> gcd (rational 18) (rational (-24));
6%1
> gcd (rational (-18)) (rational (-24));
6%1
>
```

lcm n1 n2: The LCM of rationals q1 and q2 is the smallest positive rational m such that both q1 and q2 divide m exactly. This is not defined for n1 and n2 both zero. For integral q1 and q2, this definition coincides with the usual definition of LCM for integers.

Example 15 With two `rationals`:

```
> let a = 7%12;
> let bB = 21%32;
> let m = lcm a b;
> m;
21%4
> m % a;
9%1
> m % b;
8%1
>
```

Example 16 With a `rational` and an `integer`:

```
> let m = lcm (6%5) 4;
> m;
12%1
> m % (6%5);
10%1
>
```

Example 17 The behaviour with negative numbers:

```
> lcm (rational (-18)) (rational 24);
72%1
> lcm (rational 18) (rational (-24));
72%1
> lcm (rational (-18)) (rational (-24));
72%1
>
```

Together, the GCD and LCM have the following property when applied to two numbers: (gcd q1 q2) * (lcm q1 q2) = |q1 * q2|.

6.3 Extrema (Minima and Maxima)

The standard `min` and `max` functions work with `rational` values.

Example 18 Maximum:

```
> max (3%8) (1%3);
3%8
>
```

7 Special `rational` Functions

7.1 Complexity

The 'complexity' (or 'complicatedness') of a `rational` is a measure of the greatness of its simplest (positive) denominator.

The complexity of a number is not itself made available, but various functions and operators are provided to allow complexities to be compared. Generally, it does not make sense to operate directly on complexity values.

The complexity functions in this section may be applied to `integers` (the least complex), `rationals`, or `reals` (`doubles`; the most complex).

Functions concerning 'complexity' are named with 'cplx', whereas functions concerning 'complex numbers' (see Rational Complex Numbers) are named with 'comp'.

7.1.1 Complexity Relations

n1 eq_cplx n2: "[is] equally complex [to]" — returns 1 if n1 and n2 are equally complex; returns 0 otherwise. Equal complexity is not the same a equality; n1 and n2 are equally complex if their simplest denominators are equal. Equal complexity forms an equivalence relation on `rationals`.

Example 19 Complexity equality test:

```
> (1%3) eq_cplx (100%3);
1
> (1%4) eq_cplx (1%5);
0
> (3%3) eq_cplx (1%3); // LHS is not in simplest form
0
>
```

:n1 not_eq_cplx n2“ ”not equally complex“ — returns 0 if n1 and n2 are equally complex; returns 1 otherwise.

:n1 less_cplx n2” “[is] less complex [than]” (or “simpler”) — returns 1 if n1 is strictly less complex than n2; returns 0 otherwise. This forms a partial strict ordering on `rationals`.

Example 20 Complexity inequality test:

```
> (1%3) less_cplx (100%3);
0
> (1%4) less_cplx (1%5);
1
> (3%3) less_cplx (1%3); // LHS is not in simplest form
1
>
```

n1 less_eq_cplx n2: “less or equally complex” (or “not more complex”) — returns 1 if n1 is less complex than or equally complex to n2; returns 0 otherwise. This forms a partial non-strict ordering on `rationals`.

n1 more_cplx n2: “[is] more complex [than]” — returns 1 if n1 is strictly more complex than n2; returns 0 otherwise. This forms a partial strict ordering on `rationals`.

n1 more_eq_cplx n2: “more or equally complex” (or “not less complex”) — returns 1 if n1 is more complex than or equally complex to n2; returns 0 otherwise. This forms a partial non-strict ordering on `rationals`.

7.1.2 Complexity Comparison Function

cmp_complexity n1 n2: is the ’complexity comparison’ function, and returns the sign of the difference in complexity; that is, it returns -1 if n1 is less complex than n2, 0 if n1 and n2 are equally complex (but not necessarily equal), and $+1$ if n1 is more complex than n2.

Example 21 Complexity comparison:

```
> cmp_complexity (1%3) (100%3);
0
> cmp_complexity (1%4) (1%5);
-1
> cmp_complexity (3%3) (1%3); // LHS is not in simplest form
-1
>
```

7.1.3 Complexity Extrema

least_cplx n1 n2: returns the least complex of n1 and n2; if they're equally complex, n1 is returned.

Example 22 Complexity selection:

```
> least_cplx (100%3) (1%3);
100%3
> least_cplx (1%5) (1%4);
1%4
> least_cplx (1%3) (3%3); // second argument not in simplest form
1%1
>
```

most_cplx n1 n2: returns the most complex of n1 and n2; if they're equally complex, n1 is returned.

7.1.4 Other Complexity Functions

complexity_rel n1 op n2: returns "complexity-of n1" compared by operator op to the "complexity-of n2". This is equivalent to prefix complexity rel op n1 n2 (below), but is the more readable form.

Example 23 Complexity relations:

```
> complexity_rel (1%3) (==) (100%3);
1
> complexity_rel (1%4) (<=) (1%5);
1
> complexity_rel (1%4) (>) (1%5);
0
>
```

prefix_complexity_rel op n1 n2: returns the same as complexity_rel n1 op n2, but this form is more convenient for currying.

7.2 Mediants and Farey Sequences

mediant q1 q2: returns the canonical mediant of the rationals q1 and q2, a form of (nonarithmetic) average on rationals. The mediant of the representations n1/d1 = q1 and n2/d2 = q2, where d1 and d2 must be positive, is defined as (n1 + n2)/(d1 + d2). A mediant of the rationals q1 and q2 is a mediant of some representation of each of q1 and q2. That is, the mediant is dependent upon the representations and therefore is not well-defined as a function on pairs of rationals. The canonical mediant always assumes the simplest representation, and therefore is well-defined as a function on pairs of rationals.

By the phrase "the mediant" (as opposed to just "a mediant") we always mean "the canonical mediant".

If q1 < q2, then any mediant q is always such that q1 < q < q2.

The (canonical) mediant has some special properties. If q1 and q2 are integers, then the mediant is the arithmetic mean. If q1 and q2 are unit fractions (reciprocals of integers), then the mediant is the harmonic mean. The mediant of q and 1/q is ±1, (which happens to be a geometric mean with the correct sign, although this is a somewhat uninteresting and degenerate case).

Example 24 Mediants:

```
> mediant (1%4) (3%10);
2%7
> mediant 3 7; // both ‘‘integers‘‘
5%1
> mediant 3 8; // both ‘‘integers‘‘ again
11%2
> mediant (1%3) (1%7); // both unit fractions
1%5
> mediant (1%3) (1%8); // both unit fractions again
2%11
> mediant (-10) (-1%10);
-1%1
>
```

farey -k: for an `integer` k, farey returns the ordered list containing the order-k Farey sequence, which is the ordered list of all rational numbers between 0 and 1 inclusive with (simplest) denominator at most k.

Example 25 A Farey sequence:

```
> map str_mixed (farey 6);
["0","1/6","1/5","1/4","1/3","2/5","1/2","3/5","2/3","3/4","4/5","5/6","1"]
>
```

(See the function str_mixed .)

Farey sequences and mediants are closely related. Three rationals q1 < q2 < q3 are consecutive members of a Farey sequence if and only if q2 is the mediant of q1 and q3. If rationals q1 = n1/d1 < q2 = n2/d2 are consecutive members of a Farey sequence, then n2d1 − n1d2 = 1.

7.3 `rational` Type Simplification

rat_simplify q: returns q with `rationals` simplified to `integers`, if possible.

Example 26 `rational` type simplification:

```
> let l = [9, 9%1, 9%2, 4.5, 9%1+i, 9%2+i];
l;
[9,9%1,9%2,4.5,9%1+:1,9%2+:1]
> map rat_simplify l;
[9,9,9%2,4.5,9+:1,9%2+:1]
>
```

See Rational Complex Numbers for details about rational complex numbers, and Rational Complex Type Simplification for details of their type simplification.

8 Q → Z — Rounding

8.1 Rounding to Integer

Some of these are new functions, and some are overloads of standard functions. The behaviour of the overloads is consistent with that of the standard functions.

floor q: (overload of standard function) returns q rounded downwards, i.e. towards −1, to an `integer`, usually denoted bQc.

ceil q: (overload of standard function) returns q rounded upwards, i.e. towards +1, to an `integer`, usually denoted dQe.

trunc q: (overload of standard function) returns q truncated, i.e. rounded towards 0, to an `integer`.

round q: (overload of standard function) returns q 'rounded off', i.e. rounded to the nearest `integer`, with 'half-integers' (values that are an integer plus a half) rounded away from zero.

round_zero_bias q: (new function) returns q 'rounded off', i.e. rounded to the nearest `integer`, but with 'half-integers' rounded towards zero.

round_unbiased q: (new function) returns q rounded to the nearest `integer`, with 'half-integers' rounded to the nearest even `integer`.

Example 27 Illustration of the different rounding modes:

```
> let l = while (<= 3) (+(1%2)) (- rational 3)
> map double l; // (just to show the values in a familiar format)
[-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0]
> map floor l;
[-3,-3,-2,-2,-1,-1,0,0,1,1,2,2,3]
> map ceil l;
[-3,-2,-2,-1,-1,0,0,1,1,2,2,3,3]
> map trunc l;
[-3,-2,-2,-1,-1,0,0,0,1,1,2,2,3]
> map round l;
[-3,-3,-2,-2,-1,-1,0,1,1,2,2,3,3]
> map round_zero_bias l;
[-3,-2,-2,-1,-1,0,0,0,1,1,2,2,3]
> map round_unbiased l;
[-3,-2,-2,-2,-1,0,0,0,1,2,2,2,3]
>
```

(See the function double.)

8.2 Integer and Fraction Parts

integer_and_fraction roundfun q: returns a pair (z, f) where z is the 'integer part' as an `integer`, f is the 'fraction part' as a `rational`, where the rounding operations are performed using rounding mode roundfun (see Rounding to Integer).

Example 28 Integer and fraction parts with the different rounding modes:

```
> let nc = -22%7;
> integer_and_fraction floor nc;
(-4,6%7)
> integer_and_fraction trunc nc;
(-3,-1%7)
> integer_and_fraction round nc;
(-3,-1%7)
>
```

It is always the case that z and f have the property that q = z + f. However, the remaining properties depend upon the choice of roundfun. Thus this provides a 'decomposition' operator; there are others (see Decomposition). If roundfun = floor then $0 \leq f < 1$. If Round = ceil then $-1 < f \leq 0$. If roundfun = trunc then $|f| < 1$ and sgn f $\in \{0, \text{sgn } q\}$. If roundfun = round, roundfun = round_zero_bias or roundfun = round_unbiased then $|f| \leq 1/2$.

fraction roundfun q: returns just the 'fraction part' as a `rational`, where the rounding operations are performed using roundfun. The corresponding function 'integer' is not provided, as integer roundfun q would be just roundfun q. The integer and fraction function (probably with trunc or floor rounding mode) may be used in conjunction with num_den (see 'Deconstructors') to decompose a `rational` into integer, numerator and denominator parts.

int q: overloads the built-in int and returns the 'integer part' of q consistent with the built-in.

frac q: overloads the built-in frac and returns the 'fraction part' of q consistent with the built-in.

Example 29 Standard integer and fraction parts:

```
> let nc = -22%7;
> int nc;
-3.0
> frac nc;
-0.142857142857143
>
```

9 Rounding to Multiples

round_to_multiple roundfun multOf q: returns q rounded to an integer multiple of a non-zero value multOf, using roundfun as the rounding mode (see Rounding to Integer). Note that it is the multiple that is rounded in the prescribed way, and not the final result, which may make a difference in the case that multOf is negative. If that is not the desired behaviour, pass this function the absolute value of multOf rather than multOf. Similar comments apply to the following functions.

floor_multiple multOf q: returns q rounded to a downwards integer multiple of multOf.

ceil_multiple multOf q: returns q rounded to an upwards integer multiple of multOf.

trunc_multiple multOf q: returns q rounded towards zero to an integer multiple of multOf.

round_multiple multOf q: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded away from 0.

round_multiple_zero_bias multOf q: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded towards 0.

round_multiple_unbiased multOf q: returns q rounded towards the nearest integer multiple of multOf, with half-integer multiples rounded to an even multiple.

Example 30 Round to multiple:

```
> let l = [34.9, 35, 35%1, 35.0, 35.1];
> map double l; // (just to show the values in a familiar format)
[34.9,35.0,35.0,35.0,35.1]
> map (floor_multiple 10) l;
[30,30,30,30,30]
> map (ceil_multiple 10) l;
[40,40,40,40,40]
> map (trunc_multiple 10) l;
[30,30,30,30,30]
> map (round_multiple 10) l;
[30,40,40,40,40]
> map (round_multiple_zero_bias 10) l;
[30,30,30,30,40]
> map (round_multiple_unbiased 10) l;
[30,40,40,40,40]
```

(See the function double.)

The round multiple functions may be used to find a fixed denominator approximation of a number. (The simplest denominator may actually be a proper factor of the chosen value.) To approximate for a bounded (rather than particular) denominator, use rational approx max den instead (see Best Approximation with Bounded Denominator).

Example 31 Finding the nearest q = n/d value to $1/e \approx 0.368$ where d = 1000 (actually, where d|1000):

```
> let co_E = exp (-1);
co_E;
0.367879441171442
> round_multiple (1%1000) co_E;
46%125
> 1000 * ans;
368%1
>
```

Example 32 Finding the nearest q = n/d value to 1/[U+0278] ≈ 0.618 where $d = 3^5 = 243$ (actually, where d|243):

```
> let co_Phi = (sqrt 5 - 1) / 2;
> round_multiple (1%243) co_Phi;
50%81
>
```

Other methods for obtaining a `rational` approximation of a number are described in R → Q — Approximation.

10 Q → R — Conversion / Casting

double q: (overload of built-in) returns a `double` having a value as close as possible to q. (Overflow, underflow and loss of accuracy are potential problems. `rationals` that are too absolutely large or too absolutely small may overflow or underflow; some `rationals` can not be represented exactly as a `double`.)

11 R → Q — Approximation

This section describes functions that approximate a number (usually a `double`) by a `rational`. See Rounding to Multiples for approximation of a number by a `rational` with a fixed denominator. See Numeral String → Q — Approximation for approximation by a `rational` of a string representation of a real number.

11.1 Intervals

Some of the approximation functions return an `interval`. The file `rat_interval.pure` is a basic implementation of interval arithmetic, and is not included in the default prelude. It is not intended to provide a complete implementation of interval arithmetic. The notions of 'open' and 'closed' intervals are not distinguished. Infinite and half-infinite intervals are not specifically provided. Some operations and functions may be missing. The most likely functions to be used are simply the 'deconstructors'; see Interval Constructors and 'Deconstructors'.

11.1.1 `interval` Constructors and 'Deconstructors'

Intervals are constructed with the function interval.

interval (n1, n2): given a pair of numbers (z1 <= z2), this returns the `interval` z1..z2. This is the inverse of `lo_up`.

Example 33 Constructing an interval:

```
> let v = interval (3, 8);
> v;
interval::Ivl 3 8
>
```

lower v: returns the infimum (roughly, minimum) of v.

upper v: returns the supremum (roughly, maximum) of v.

lo_up v: returns a pair (l, u) containing the lower l and upper u extrema of the interval v. This is the inverse of interval as defined on number pairs.

Example 34 Deconstructing an interval:

```
> lower v;
3
> upper v;
8
> lo_up v;
(3,8)
>
```

11.1.2 `interval` Type Tests

exactp v: returns whether an interval v has exact extrema.

inexactp v: returns whether an interval v has an inexact extremum.

intervalp x: returns whether x is of type `interval`.

interval_valp x: returns whether x has an interval value.

ratinterval_valp x: returns whether x has an interval value with rational extrema.

intinterval_valp x: returns whether x has an interval value with integral extrema.

Example 35 `interval` value tests:

```
> let l = [interval(0,1), interval(0,1%1), interval(0,3%2), inter-
val(0,1.5)];
> map exactp l;
[1,1,1,0]
> map inexactp l
[0,0,0,1]
> map intervalp l;
[1,1,1,1]
> map interval_valp l;
[1,1,1,1,]
> map ratinterval_valp l;
[1,1,1,1]
> map intinterval_valp l;
[1,1,0,0]
```

11.1.3 `interval` Arithmetic Operators and Relations

The standard arithmetic operators (+), (−), (*), (/) and (%) are overloaded for `intervals`. The divide operators (/) and (%) do not produce a result if the right operand is an interval containing 0. **Example 36** Some intervals:

```
> let a = interval (11, 19);
> let b = interval (16, 24);
> let c = interval (21, 29);
> let d = interval (23, 27);
>
```

Example 37 `interval` arithmetic:

```
> let p = interval (0, 1);
> let s = interval (-1, 1);
> a + b;
interval::Ivl 27 43
> a - b;
interval::Ivl -13 3
> a * b;
interval::Ivl 176 456
> p * 2;
interval::Ivl 0 2
> (-2) * p;
interval::Ivl -2 0
> -c;
interval::Ivl -29 -21
> s * a;
interval::Ivl -19 19
> a % 2;
interval::Ivl (11%2) (19%2)
> a / 2;
```

```
interval::Ivl 5.5 9.5
> reciprocal a;
interval::Ivl (1%19) (1%11)
> 2 % a;
interval::Ivl (2%19) (2%11)
> a % b
interval::Ivl 11%24 19%16
> a % a; // notice that the intervals are mutually independent here
interval::Ivl (11%19) (19%11)
>
```

There are also some relations defined for `intervals`. The standard relations (==) and (˜=) are overloaded.

However, rather than overloading (<), (<=), (>), (>=), which could be used for either ordering or containment with some ambiguity, the module defines `(before)`, `(within)`, and so on. 'Strictness' refers to the properties at the end-points.

v1 before v2: returns whether v1 is entirely before v2.

v1 strictly_before v2: returns whether v1 is strictly entirely before v2.

v1 after v2: returns whether v1 is entirely after v2.

v1 strictly_after v2: returns whether v1 is strictly entirely after v2.

v1 within v2: returns whether v1 is entirely within v2; i.e. whether v1 is subinterval of v2.

v1 strictly_within v2: returns whether v1 is strictly entirely within v2; i.e. whether v1 is proper subinterval of v2.

v1 without v2: returns whether v1 entirely contains v2; i.e. whether v1 is superinterval of v2. 'Without' is used in the sense of outside or around.

v1 strictly_without v2: returns whether v1 strictly entirely contains v2; i.e. whether v1 is proper superinterval of v2.

v1 disjoint v2: returns whether v1 and v2 are entirely disjoint.

v strictly_disjoint v2: returns whether v1 and v2 are entirely strictly disjoint.

Example 38 `interval` relations:

```
> a == b;
0
> a == a;
1
> a before b;
0
> a before c;
1
> c before a;
0
> a disjoint b;
0
> c disjoint a;
```

```
1
> a within b;
0
> a within c;
0
> d within c;
1
> c within d;
0
> a strictly_within a;
0
> a within a;
1
>
```

(The symbols a through d were defined in Example 36.)
These may also be used with a simple (`real`) value, and in particular to test membership.
Example 39 Membership:

```
> 10 within a;
0
> 11 within a;
1
> 11.0 within a;
1
> 12 within a;
1
> 12.0 within a;
1
> 10 strictly_within a;
0
> 11 strictly_within a;
0
> (11%1) strictly_within a;
0
> 12 strictly_within a;
1
> (12%1) strictly_within a;
1
>
```

(The symbol a was defined in Example 36.)

11.1.4 `interval` Maths

Some standard functions are overloaded for intervals; some new functions are provided.

abs v: returns the interval representing the range of (x) as x varies over v.

Example 40 Absolute interval:

```
> abs (interval (1, 5));
interval::Ivl 1 5
> abs (interval (-1, 5));
interval::Ivl 0 5
```

```
> abs (interval (-5, -1));
interval::Ivl 1 5
>
```

sgn v: returns the interval representing the range of sgn(x) as x varies over v.

#v: returns the length of an interval.

Example 41 Absolute interval:

```
> #d;
4
```

(The symbol d was defined in Example 36.)

11.2 Least `complex` Approximation within

rational_approx_epsilon r: Find the least complex (see Complexity Extrema) `rational` approximation to r (usually a `double`) that is -close. That is find the q with the smallest possible denominator such that such that $|q - r| \leq$. (> 0.)

Example 42 `rational` approximation to $\approx 3.142 \approx 22/7$:

```
> rational_approx_epsilon .01 pi;
22%7
> abs (ans - pi);
0.00126448926734968
>
```

Example 43 The golden ratio `[U+0278]` $= (1 + \sqrt{5})\ /\ 2 \approx 1.618$:

```
> let phi = (1 + sqrt 5) / 2;
> rational_approx_epsilon .001 phi;
55%34
> abs (ans - phi);
0.000386929926365465
>
```

rational_approxs_epsilon r: Produce a list of ever better `rational` approximations to r (usually a `double`) that is eventually -close. (> 0.)

Example 44 `rational` approximations to :

```
> rational_approxs_epsilon .0001 pi;
[3%1,25%8,47%15,69%22,91%29,113%36,135%43,157%50,179%57,201%64,223%71,245%78,
267%85,289%92,311%99,333%106]
>
```

Example 45 `rational` approximations to the golden ratio `[U+0278]`; these approximations are always reverse consecutive Fibonacci numbers (from f1: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . .):

```
> rational_approxs_epsilon .0001 phi;
[1%1,3%2,8%5,21%13,55%34,144%89]
>
```

(The symbol phi was defined in Example 43.)

rational_interval_epsilon r: Find the least complex (see Complexity Extrema) `rational` interval containing r (usually a `double`) that is -small. That is find the least complex (see Complexity Extrema) q1 $\leq$ q2 such that r $\in$ [q1, q2] and q2 $-$ q1 $\leq$. ($>$ 0.)

Example 46 `rational` interval surrounding :

```
> let i_Pi = rational_interval_epsilon .01 pi;
> i_Pi;
interval::Ivl (47%15) (22%7)
> double (lower i_Pi); pi; double (upper i_Pi);
3.13333333333333
3.14159265358979
3.14285714285714
>
```

(The functions lower and upper are described in Interval Constructors and 'Deconstructors'.)
Example 47 `rational` interval surrounding the golden ratio `[U+0278]`:

```
> rational_interval_epsilon .001 phi
interval::Ivl (55%34) (89%55)
> #ans;
1%1870
```

(The symbol Phi was defined in Example 43. The function # (intervals) is described in Interval Maths.)

11.3 Best Approximation with Bounded Denominator

rational_approx_max_den maxDen r: Find the closest `rational` approximation to r (usually a `double`) that has a denominator no greater than maxDen. (maxDen $>$ 0).

Example 48 `rational` approximation to :

```
> rational_approx_max_den 10 pi;
22%7
>
```

Example 49 `rational` approximation to the golden ratio `[U+0278]`:

```
> rational_approx_max_den 1000 phi;
1597%987
>
```

(The symbol phi was defined in Example 43.)

rational_approxs_max_den maxDen r: Produce a list of ever better `rational` approximations to r (usually a `double`) while the denominator is bounded by maxDen (maxDen $>$ 0).

Example 50 `rational` approximations to :

```
> rational_approxs_max_den 100 pi;
[3%1,25%8,47%15,69%22,91%29,113%36,135%43,157%50,179%57,201%64,223%71,245%78,
267%85,289%92,311%99]
>
```

Example 51 `rational` approximations to the golden ratio `[U+0278]`:

```
> rational_approxs_max_den 100 phi;
[1%1,3%2,8%5,21%13,55%34,144%89]
>
```

(The symbol phi was defined in Example 43.)

rational_interval_max_den maxDen r: Find the smallest `rational` interval containing r (usually a `double`) that has endpoints with denominators no greater than maxDen (maxDen > 0).

Example 52 `rational` interval surrounding :

```
> let i_Pi = rational_interval_max_den 100 pi ; i_Pi
interval::Ivl (311%99) (22%7)
> double (lower i_Pi); pi; double (upper i_Pi);
3.14141414141414
3.14159265358979
3.14285714285714
>
```

Example 53 `rational` interval surrounding the golden ratio `[U+0278]`:

```
> rational_interval_max_den 1000 phi
interval::Ivl (987%610) (1597%987)
>
```

(The symbol phi was defined in Example 43.)

To approximate for a particular (rather than bounded) denominator, use round to multiple instead (see Rounding to Multiples).

12 Decomposition

There is more than one way to 'decompose' a rational number into its 'components'. It might be split into an integer and a fraction part — see Integer and Fraction Parts; or sign and absolute value — see Absolute Value and Sign; or numerator and denominator — see 'Deconstructors'.

13 Continued Fractions

13.1 Introduction

In "pure-rational", a continued fraction $a_0 + (1 / (a_1 + (1 / (a_2 + \cdots + 1 / a_n))))$ where $\forall i > 0 \bullet a_i \neq 0$, is represented by $[a_0, a_1, a_2, \ldots, a_n]$.

A 'simple' continued fraction is one in which $\forall i \bullet a_i \in \mathbf{Z}$ and $\forall i > 0 \bullet a_i > 0$.

Simple continued fractions for rationals are not quite unique since $[a_0, a_1, \ldots, a_n, 1] = [a_0, a_1, \ldots, a_{n+1}]$. We will refer to these as the 'non-standard' and 'standard' forms, respectively. The following functions return the standard form.

13.2 Generating Continued Fractions

13.2.1 Exact

continued_fraction q: Find 'the' (exact) continued fraction of a rational (including, trivially, integer) value q.

Example 54 The rational 1234/1001:

```
> continued_fraction (1234%1001);
[1,4,3,2,1,1,1,8]
> evaluate_continued_fraction ans;
1234%1001
```

13.2.2 Inexact

continued_fraction_max_terms n r: Find up to n initial terms of continued fraction of the value r with the 'remainder', if any, in the final element. (If continued_fraction_max_terms n r returns a list of length n or less, then the result is exact.)

Example 55 First 5 terms of the continued fraction for the golden ratio `[U+0278]`:

```
> continued_fraction_max_terms 5 phi;
  [1,1,1,1,1,1.61803398874989]
> evaluate_continued_fraction ans;
1.61803398874989
>
```

(The symbol phi was defined in Example 43.)

continued_fraction_epsilon r: Find enough of the initial terms of a continued fraction to within of the value r with the 'remainder', if any, in the final element.

Example 56 First few terms of the value $\sqrt{2}$:

```
> continued_fraction_epsilon .001 (sqrt 2);
[1,2,2,2,2,2.41421356237241]
> map double (convergents ans);
[1.0,1.5,1.4,1.41666666666667,1.41379310344828,1.41421356237309]
>
```

13.3 Evaluating Continued Fractions

evaluate_continued_fraction aa: Fold a continued fraction aa into the value it represents. This function is not limited to simple continued fractions. (Exact simple continued fractions are folded into a rational.)

Example 57 The continued fraction [1, 2, 3, 4] and the non-standard form [4, 3, 2, 1]:

```
> evaluate_continued_fraction [1,2,3,4];
43%30
> continued_fraction ans;
[1,2,3,4]
> evaluate_continued_fraction [4,3,2,1];
43%10
> continued_fraction ans;
[4,3,3]
>
```

13.3.1 Convergents

convergents aa: Calculate the convergents of the continued fraction aa. This function is not limited to simple continued fractions.

Example 58 Convergents of a continued fraction approximation of the value $\sqrt{2}$:

```
> continued_fraction_max_terms 5 (sqrt 2);
[1,2,2,2,2,2.41421356237241]
> convergents ans;
[1%1,3%2,7%5,17%12,41%29,1.41421356237309]
>
```

14 `rational complex` Numbers

Pure together with `rational.pure` provide various types of number, including `integers` (**Z**), `doubles` (**R**, roughly), `complex` numbers (**C**) and Gaussian integers (**Z**[i]), `rationals` (**Q**) and rational complex numbers (**Q**[i]).

Functions concerning 'complex numbers' are named with 'comp', whereas functions concerning 'complexity' (see Complexity) are named with 'cplx'.

14.1 `rational complex` Constructors and 'Deconstructors'

`complex` numbers can have rational parts.

Example 59 Forming a `rational complex`:

```
> 1 +: 1 * (1%2);
1%1+:1%2
> ans * ans;
3%4+:1%1
>
```

And rational numbers can be given complex parts.

Example 60 `complex rationals` and complicated `rationals`:

```
> (1 +: 2) % (3 +: 4);
11%25+:2%25
> ans * (3 +: 4);
1%1+:2%1
> ((4%1) * (0 +: 1)) % 2;
0%1+:2%1
> ((4%1) * (0 +: 1)) % (1%2);
0%1+:8%1
> ((4%1) * (0 +: 1)) % (1 + (1%2) * (0 +: 1));
8%5+:16%5
> ans * (1+(1%2) * (0 +: 1));
0%1+:4%1
> ((4%1) * (0 +: 1)) / (1 + (1%2) * (0 +: 1));
1.6+:3.2
>
```

The various parts of a complex rational may be deconstructed using combinations of num and den and the standard functions re and im.

Thus, taking real and imaginary parts first, a rational complex number may be considered to be (x_n / x_d) + (y_n / y_d) * i with x_n, x_d, y_n, $y_d \in \mathbf{Z}$.

A rational complex number may also be decomposed into its 'numerator' and 'denominator', where these are both integral complex numbers, or 'Gaussian integers', and the denominatoris a minimal choice in some sense.

One way to do this is so that the denominator is the minimum positive integer. The denominator is a complex number with zero imaginary part.

Thus, taking numerator and denominator parts first, a rational complex number may be considered to be $(\mathrm{n}_x + \mathrm{n}_y * \mathrm{i}) / (\mathrm{d} + 0 * \mathrm{i})$ with n_x, n_y, $\mathrm{d} \in \mathbf{Z}$.

Another way to do this is so that the denominator is a Gaussian integer with minimal absolute value. Thus, taking numerator and denominator parts first, a rational complex number may be considered to be $(\mathrm{n}_x + \mathrm{n}_y * \mathrm{i}) / (\mathrm{d}_x + \mathrm{d}_y * \mathrm{i})$ with n_x, n_y, d_x, $\mathrm{d}_y \in \mathbf{Z}$.

The d_x, d_y are not unique, but can be chosen such that $\mathrm{d}_x > 0$ and either $|\mathrm{d}_y| < \mathrm{d}_x$ or $\mathrm{d}_y = \mathrm{d}_x > 0$.

num_den_nat c: given a `complex rational` or `integer` c, returns a pair (n, d) containing an integral complex (Gaussian integral) numerator n, and the smallest natural (i.e. positive integral real) complex denominator d, i.e. a complex number where $\Re(\mathrm{d}) \in \mathbf{Z}$, $\Re(\mathrm{d}) > 0$, $\Im(\mathrm{d}) = 0$; i.e. the numerator and denominator of one 'normalised' form of c.

This is an inverse (up to equivalence) of rational as defined on integral complex pairs (see Constructors).

num_den_gauss c: given a `complex rational` or `integer` c, returns a pair (n, d) containing an integral complex (Gaussian integral) numerator n, and an absolutely smallest integral complex denominator d chosen s.t. $\Re(\mathrm{d}),=(\mathrm{d}) \in \mathbf{Z}$, $\Re(\mathrm{d}) > 0$, and either $|\Re(\mathrm{d})| < \Im(\mathrm{d})$ or $\Re(\mathrm{d}) = \Im(\mathrm{d}) > 0$; i.e. the numerator and denominator of another 'normalised' form of c.

This is an inverse (up to equivalence) of rational as defined on integral complex pairs (see Constructors).

num_den c: synonymous with num_den_gauss.

This is an inverse (up to equivalence) of rational as defined on integer pairs (see Constructors).

num c: given a `complex rational` or `integer` c, returns just the numerator of the normalised form of c given by num_den c.

den c: given a `complex rational` or `integer` c, returns just the denominator of the normalised form of c given by num_den c.

Example 61 `rational` complex number deconstruction:

```
> let cq = (1+2*i)%(3+3*i);
cq;
1%2+:1%6
> (re cq, im cq);
(1%2,1%6)
> (num . re) cq;
1
> (den . re) cq;
2
> (num . im) cq;
1
> (den . im) cq;
6
> let (n_nat,d_nat) = num_den_nat cq;
```

```
> (n_nat, d_nat);
(3+:1,6+:0)
> n_nat % d_nat;
1%2+:1%6
> abs d_nat;
6.0
> let (n, d) = num_den_gauss cq;
(n, d);
(1+:2,3+:3)
> let (n,d) = num_den cq;
(n, d);
(1+:2,3+:3)
> n % d
1%2+:1%6
> abs d
4.24264068711928
> (re . num) cq;
1
> (im . num) cq;
2
> (re . den) cq; //always > 0
3
> (im . den) cq; //always <= (re.den)
3
>
```

14.2 `rational complex` Type and Value Tests

Beware that `intcompvalp` and `ratcompvapl` may return 1 even if the value is of `complex` type with `double` parts. However, these functions may be combined with `exactp`.

complexp x: standard function; returns whether x is of `complex` type.

compvalp x: standard function; returns whether x has a `complex` value ($\in \mathbf{C} = \mathbf{R}[i]$).

ratcompvalp x: returns whether x has a rational complex value ($\in \mathbf{Q}[i]$).

intcompvalp x: returns whether x has an integral complex value ($\in \mathbf{Z}[i]$), i.e. a Gaussian integer value.

Example 62 `rational` complex number value tests:

```
> let l = [9, 9%1, 9%2, 4.5, sqrt 2, 1+:1, 1%2+:1, 0.5+:1, inf, nan];
> map exactp l;
[1,1,1,0,0,1,1,0,0,0]
> map inexactp l;
[0,0,0,1,1,0,0,1,1,1]
> map complexp l;
[0,0,0,0,0,1,1,1,0,0]
> map compvalp l;
[1,1,1,1,1,1,1,1,1,1]
> map (\x -> (exactp x and compvalp x)) l; // "has exact complex value"
[1,1,1,0,0,1,1,0,0,0]
> map ratcompvalp l;
```

```
[1,1,1,1,1,1,1,1,0,0]
> map (\x -> (exactp x and ratcompvalp x)) l;
[1,1,1,0,0,1,1,0,0,0]
> map intcompvalp l;
[1,1,0,0,0,1,0,0,0,0]
> map (\x -> (exactp x and intcompvalp x)) l;
[1,1,0,0,0,1,0,0,0,0]
> map ratvalp l;
[1,1,1,1,1,0,0,0,0,0]
> map (\x -> (exactp x and ratvalp x)) l;
[1,1,1,0,0,0,0,0,0,0]
> map intvalp l; // for comparison
[1,1,0,0,0,0,0,0,0,0]
> map (\x -> (exactp x and intvalp x)) l;
[1,1,0,0,0,0,0,0,0,0]
>

See ‘Type and Value Tests‘_ for some details of rational type and value tests.
```

14.3 `rational complex` Arithmetic Operators and Relations

The standard arithmetic operators (+), (−), (*), (/), (%), (), (==) and (˜=) are overloaded to have at least one complex and/or rational operand, but (<), (<=), (>), (>=) are not, as complex numbers are unordered.

Example 63 `rational` complex arithmetic:

```
> let w = 1%2 +: 3%4;
> let z = 5%6 +: 7%8
> w + z;
4%3+:13%8
> w % z;
618%841+:108%841
> w / z
0.734839476813318+:0.128418549346017
> w ^ 2;
-0.3125+:0.75
> w == z
0
> w == w
1
>
```

14.4 `rational complex` Maths

The standard functions `re` and `im` work with rational complex numbers (see Rational Complex Constructors and 'Deconstructors'). The standard functions `polar`, `abs` and `arg` work with rational complex numbers, but the results are inexact.

Example 64 `rational` complex maths:

```
> polar (1%2) (1%2);
0.438791280945186+:0.239712769302102
> abs (4%2 +: 3%2);
2.5
```

```
> arg (-1%1);
3.14159265358979
>
```

There are some additional useful functions for calculating with rational complex numbers and more general mathematical values.

norm_gauss c: returns the Gaussian norm $||c||$ of any `complex` (or `real`) number c; this is the square of the absolute value, and is returned as an (exact) `integer`.

div_mod_gauss n d: performs Gaussian integer division, returning (q, r) where q is a (not always unique) quotient, and r is a (not always unique) remainder. q and r are such that $n = q * d + r$ and $||r|| < ||d||$ (equivalently, $|r| < |d|$).

n_div_gauss d: returns just a quotient from Gaussian integer division as produced by div_mod_gauss n d.

n_mod_gauss d: returns just a remainder from Gaussian integer division as produced by div_mod_gauss n d.

gcd_gauss c1 c2: returns a GCD G of the Gaussian integers c1,c2. This is chosen so that s.t. $\Re(G) > 0$, and either $|\Im(G)| < \Re(G)$ or $\Im(G) = \Re(G) > 0$;

euclid_gcd zerofun modfun x y: returns a (non-unique) GCD calculated by performing the Euclidean algorithm on the values x and y (of any type) where zerofun is a predicate for equality to 0, and modfun is a binary modulus (remainder) function.

euclid_alg zerofun divfun x y: returns (g, a, b) where the g is a (non-unique) GCD and a, b are (arbitrary, non-unique) values such that $a * x + b * y = g$ calculated by performing the generalised Euclidean algorithm on the values x and y (of any type) where zerofun is a predicate for equality to 0, and div is a binary quotient function.

Example 65 More rational complex and other maths:

```
> norm_gauss (1 +: 3);
10
> abs (1 +: 3);
3.16227766016838
> norm_gauss (-5);
25
> let (q, r) = div_mod_gauss 100 (12 +: 5);
> (q, r);
(7+:(-3),1+:1)
> q * (12 +: 5) + r;
100+:0
> 100 div_gauss (12 +: 5);
7+:(-3)
> 100 mod_gauss (12 +: 5);
1+:1
> div_mod_gauss 23 5;
(5+:0,-2+:0)
> gcd_gauss (1 +: 2) (3 +: 4);
1+:0
> gcd_gauss 25 15;
5+:0
```

```
> euclid_gcd (==0) (mod_gauss) (1+: 2) (3 +: 4);
1+:0
> euclid_gcd (==0) (mod) 25 15;
5
> let (g, a, c) = euclid_alg (==0) (div_gauss) (1 +: 2) (3 +: 4);
g;
1+:0
>  (a, b);
(-2+:0,1+:0)
> a * (1 +: 2) + B * (3 +: 4);
1+:0
> let (g, a, b) = euclid_alg (==0) (div) 25 15;
g;
5
>  (a, b);
(-1,2)
> a * 25 + b * 15;
5
>
```

14.5 `rational complex` Type Simplification

comp_simplify c: returns q with `complex` numbers simplified to `reals`, if possible.

ratcomp_simplify c: returns q with `rationals` simplified to `integers`, and `complex` numbers simplified to `reals`, if possible.

Example 66 `rational` complex number type simplification:

```
> let l = [9+:1, 9%1+:1, 9%2+:1, 4.5+:1, 9%1+:0, 9%2+:0, 4.5+:0.0];
> l;
[9+:1,9%1+:1,9%2+:1,4.5+:1,9%1+:0,9%2+:0,4.5+:0.0]
> map_comp_simplify l;
[9+:1,9%1+:1,9%2+:1,4.5+:1,9%1,9%2,4.5+:0.0]
> map ratcomp_simplify l;
[9+:1,9+:1,9%2+:1,4.5+:1,9,9%2,4.5+:0.0]

See ‘Rational Type Simplification‘_ for some details of rational type
simplification.
```

15 String Formatting and Evaluation

15.1 The Naming of the String Conversion Functions

There are several families of functions for converting between `strings` and `rationals`.

The functions that convert from `rationals` to strings have names based on that of the standard function `str`. The `str_*` functions convert to a formatted string, and depend on a 'format structure' parameter (see Internationalisation and Format Structures). The `strs_*` functions convert to a tuple of string fragments.

The functions that convert from strings to `rationals` have names based on that of the standard function `val`. The `val_*` functions convert from a formatted string, and depend on a format structure parameter. The `sval_*` functions convert from a tuple of string fragments.

There are also `join_*` and `split_*` functions to join string fragments into formatted strings, and to split formatted strings into string fragments, respectively; these depend on a format structure parameter. These functions are not always invertible, because some of the functions reduce an error term to just a sign, e.g. str_real_approx_dp may round a value. Thus sometimes the `join_*` and `split_*` pairs, and the `str_*` and `val_*` pairs are not quite mutual inverses.

15.2 Internationalisation and Format Structures

Many of the string formatting functions in the following sections are parameterised by a 'format structure'. Throughout this document, the formal parameter for the format structure will be fmt. This is simply a map from some string 'codes' to functions as follows. The functions are mostly from strings to a string, or from a string to a tuple of strings.

"sm": a function mapping a sign and an unsigned mantissa (or integer) strings to a signed mantissa (or integer) string.

"se": a function mapping a sign and an unsigned exponent string to a signed exponent string.

"-s": a function mapping a signed number string to a pair containing a sign and the unsigned number string.

"gi": a function mapping an integer representing the group size and an integer string to a grouped integer string.

"gf": a function mapping an integer representing the group size and a fraction-part string to a grouped fraction-part string.

"-g": a function mapping a grouped number string to an ungrouped number string.

"zi": a function mapping an integer number string to a number string. The input string representing zero integer part is "", which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

"zf": a function mapping a fraction-part number string to a number string. The input string representing zero fraction part is "", which should be mapped to the desired representation of zero. All other number strings should be returned unaltered.

"ir": a function mapping initial and recurring parts of a fraction part to the desired format.

"-ir": a function mapping a formatted fraction part to the component initial and recurring parts.

"if": a function mapping an integer string and fraction part string to the radix-point formatted string.

"-if": a function mapping a radix-point formatted string to the component integer fraction part strings

"me": a function mapping a mantissa string and exponent string to the formatted exponential string.

"-me": a function mapping a formatted exponential string to the component mantissa and exponent strings.

"e": a function mapping an 'error' number (not string) and a number string to a formatted number string indicating the sign of the error.

"-e": a function mapping a formatted number string indicating the sign of the error to the component 'error' string (not number) and number strings.

Depending upon the format structure, some parameters of some of the functions taking a format structure may have no effect. For example, an `intGroup` parameter specifying the size of the integer digit groups will have no effect if the integer group separator is the empty string.

create_format options: is a function that provides an easy way to prepare a 'format structure' from the simpler 'options structure'. The options structure is another dictionary, but from more descriptive strings to a string or tuple of strings.

For example, `format_uk` is generated from `options_uk` as follows:

```
public options_uk;
const options_uk =
  dict [
    ("sign", ("-","","")),              //alternative: ("-"," ","+")
    ("exponent sign", ("-","","")),     //alternative: ("-","","+")
    ("group separator", ","),           //might be " " or "." or "'" else-
where
    ("zero", "0"),
    ("radix point", "."),               //might be "," elsewhere
    ("fraction group separator", ","),
    ("fraction zero", "0"),             //alternative: ""
    ("recur brackets", ("[","...]")),
    ("exponent", "*10^"),               //(poor) alternative: "e"
    ("error sign", ("-","","+")),
    ("error brackets", ("(",")")),
  ];

public format_uk;
const format_uk = create_format options_uk;
```

The exponent string need not depend on the radix, as the numerals for the number radix in that radix are always "10".

Beware of using "e" or "E" as an exponent string as these have the potential of being treated as digits in, e.g., hexadecimal.

Format structures do not have to be generated via create format; they may also be constructed directly.

15.3 Digit Grouping

Some functions take `group` parameters. A value of 0 means "don't group".

15.4 Radices

The functions that produce a decimal expansion take a Radix argument. The fraction parts are expanded in that radix (or 'base'), in addition to the integer parts. The parameter Radix is not restricted to the usual {2, 8, 10, 16}, but may be any `integer` from 2 to 36; the numerals ('digits') are chosen from ["0", . . . , "9", "A", . . . , "Z"]. The letter-digits are always upper case.

The functions do not attach a prefix (such as "0x" for hexadecimal) to the resulting string.

15.5 Error Terms

Some functions return a value including an 'error' term (in a tuple) or sign (at the end of a string). Such an error is represents what the next digit would be as a fraction of the radix.

Example 67 Error term in the tuple of string 'fragments':

```
> strs_real_approx_sf 10 floor 3 (234567%100000);
("+","2","34",567%1000)
> strs_real_approx_sf 10 ceil 3 (234567%100000);
("+","2","35",-433%1000)
```

```
>
```

(See the function strs_real_approx_sf.)

In strings, only the sign of the error term is given. A "+" should be read as "and a bit more"; "-" as "but a bit less".

Example 68 Error sign in the string:

```
> str_real_approx_sf format_uk 10 0 0 floor 3 (234567%100000);
"2.34(+)"
> str_real_approx_sf format_uk 10 0 0 ceil 3 (234567%100000);
"2.35(-)"
>
```

(See the function str_real_approx_sf.)

16 Q ↔ Fraction String ("i + n/d")

16.1 Formatting to Fraction Strings

str_vulgar q: returns a String representing the `rational` (or `integer`) q in the form

- "[−]n/d"

str_vulgar_or_int q: returns a String representing the `rational` (or `integer`) q in one of the forms

- "[−]n/d"
- "[−]i"

str_mixed q: returns a String representing the `rational` (or `integer`) q in one of the forms

- "i + n/d"
- "−(i + n/d)"
- "[−]n/d"
- "[−]i"

Example 69 The fraction string representations:

```
> let l = while (<= 3%2) (+(1%2)) (-3%2);
> l;
[-3%2,-1%1,-1%2,0%1,1%2,1%1,3%2]
> map str_vulgar l;
["-3/2","-1/1","-1/2","0/1","1/2","1/1","3/2"]
> map str_vulgar_or_int l;
["-3/2","-1","-1/2","0","1/2","1","3/2"]
> map str_mixed l;
["-(1+1/2)","-1","-1/2","0","1/2","1","1+1/2"]
>
```

These might be compared to the behaviour of the standard function `str`.

str x: returns a `string` representing the value x.

Example 70 The standard function str:

```
> map str l;
["-3%2","-1%1","-1%2","0%1","1%2","1%1","3%2"]
>
```

16.2 Evaluation of Fraction Strings

val_vulgar strg: returns a `rational` q represented by the string strg in the form

- "[−]n/d"

Such strings can also be evaluated by the `val_mixed` function.

val_mixed strg: returns a `rational` q represented by the `string` strg

- "i + n/d"
- "−(i + n/d)"
- "[−]n/d" — thus val_mixed strictly extends val_vulgar
- "[−]i"

Example 71 Evaluating fraction strings:

```
> val_vulgar "-22/7";
-22%7
> val_mixed "1+5/6";
11%6
>
```

These might be compared to the behaviour of the standard function val.

val s: evaluates the `string` s.

Example 72 The standard function val:

```
> val "1+5%6";
11%6
> val "1+5/6";
1.83333333333333
>
```

17 Q ↔ Recurring Numeral Expansion String ("I.FR")

See Internationalisation and Format Structures for information about the formatting structure to be supplied in the fmt parameter.

17.1 Formatting to Recurring Expansion Strings

str_real_recur fmt radix intGroup q: returns a `string` (exactly) representing the `rational` (or `integer`) q as base-Radix expansion of one the forms

- "[−]int.frac"
- "[−]int.init frac part[smallest recurring frac part. . .]"

Note that there is no fracGroup parameter.

Beware that the string returned by this function can be very long. The length of the recurring part of such a decimal expansion may be up to one less than the simplest denominator of q.

Example 73 The recurring radix expansion-type string representations:

```
> str_real_recur format_uk 10 3 (4000001%4); // grouped with commas
"1,000,000.25"
> str_real_recur format_uk 10 0 (4000001%4); // no grouping
"1000000.25"
```

```
> str_real_recur format_uk 10 3 (1000000%3);
"333,333.[3...]"
> str_real_recur format_uk 10 3 (1000000%7);
"142,857.[142857...]"
> str_real_recur format_uk 10 3 (-1%700);
"-0.00[142857...]"
> str_real_recur format_uk 10 3 (127%128);
"0.9921875"
> str_real_recur format_uk 2 4 (-127%128);
"-0.1111111"
> str_real_recur format_uk 16 4 (127%128);
"0.FE"
> str_real_recur format_uk 10 0 (70057%350) // 1%7 + 10001%50;
"200.16[285714...]"
>
```

The function allows expansion to different radices (bases).
Example 74 The recurring radix expansion in decimal and hexadecimal:

```
> str_real_recur format_uk 10 0 (1%100);
"0.01"
> str_real_recur format_uk 16 0 (1%100);
"0.0[28F5C...]"
>
```

Example 75 The recurring radix expansion in duodecimal:

```
> str_real_recur format_uk 12 0 (1%100);
"0.0[15343A0B62A68781B059...]"
>
```

Note that this bracket notation is not standard in the literature. Usually the recurring numerals are indicated by a single dot over the initial and final numerals of the recurring part, or an overline over the recurring part. For example $1/70 = 0.0\dot{}14285\dot{}7 = 0.0142857$ and $1/3 = 0.\dot{}3 = 0.3$.

strs_real_recur radix q: returns a quadruple of the four strings:

- the sign,
- integer part (which is empty for 0),
- initial fraction part
- and recurring fraction part (either and both of which may be empty).

Example 76 The recurring radix expansion in decimal — the fragments:

```
> strs_real_recur 10 (100%7);
("+","14","","285714")
> strs_real_recur 10 (-1%700);
("-","","00","142857")
> strs_real_recur 10 (70057%350);
("+","200","16","285714")
>
```

This function may be used to also, e.g. format the integer part with comma-separated groupings.

:join_str_real_recur fmt intGroup sign i fracInit fracRecur“ formats the parts in the quadruple returned by strs_real_recur to the sort of string as returned by str_real_recur.

17.2 Evaluation of Recurring Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between `rationals` and differently formatted strings if a suitable alternative format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

val_real_recur fmt radix strg: returns the `rational` q represented by the base-radix expansion `string` strg of one the forms

- "[−]int.frac"
- "[−]int.init frac part[recurring frac part. . .]"

Example 77 Conversion from the recurring radix expansion-type string representations:

```
> val_real_recur format_uk 10 "-12.345";
-2469%200
> val_real_recur format_uk 10 "0.3";
3%10
> val_real_recur format_uk 10 "0.[3...]";
1%3
> val_real_recur format_uk 10 ".333[33...]";
1%3
> val_real_recur format_uk 10 ".[9...]";
1%1
```

sval_real_recur radix sign iStr fracStr recurPartStr: returns the `rational` q represented by the parts

- sign
- integer part
- initial fraction part
- recurring fraction part

split_str_real_recur Fmt strg: returns a tuple containing the parts

- sign
- integer part
- initial fraction part
- recurring fraction part of one the forms - "[−]int.frac" - "[−]int.init frac part[recurring frac part. . .]"

18 Q ↔ Numeral Expansion String ("I.F × 10E")

See Internationalisation and Format Structures for information about the formatting structure to be supplied in the fmt parameter.

The exponent string "*10ˆ" need not depend on the radix, as the numerals for the number radix in that radix are always "10".

18.1 Formatting to Expansion Strings

18.1.1 Functions for Fixed Decimal Places

str_real_approx_dp fmt radix intGroup fracGroup roundfun dp q: returns a string representing a numeral expansion approximation of q to dp decimal places, using rounding mode roundfun (see Rounding to Integer) roundfun is usually round or round_unbiased.

(dp may be positive, zero or negative; non-positive dps may look misleading — use e.g. scientific notation instead.)

Example 78 Decimal places:

```
> str_real_approx_dp format_uk 10 3 3 round 2 (22%7);
"3.14(+)"
> str_real_approx_dp format_uk 10 3 3 ceil 2 (22%7);
"3.15(-)"
>
```

strs_real_approx_dp radix roundfun do q: returns a tuple of strings

- sign
- integer part
- fraction part

representing an expansion to a number of decimal places, together with

- the rounding "error": a fraction representing the next numerals.

Example 79 Decimal places — the fragments:

```
> strs_real_approx_dp 10 round 2 (22%7);
("+","3","14",2%7)
> strs_real_approx_dp 10 ceil 2 (22%7);
("+","3","15",-5%7)
>
```

join_str_real_approx fmt intGroup fracGroup sign i frac err: formats the parts in the quadruple returned by strs_real_approx_dp or strs_real_approx_sf to the sort of string as returned by str_real_approx_dp or str_real_approx_sf.

18.1.2 Functions for Significant Figures

str_real_approx_sf fmt radix intGroup fracGroup roundfun sf q: returns a string representing a numeral expansion approximation of q to sf significant figures, using rounding mode roundfun (see Rounding to Integer).

Round is usually round or round_unbiased. (sf must be positive.)
Example 80 Significant figures:

```
> str_real_approx_sf format_uk 10 3 3 floor 2 (22%7);
"3.1(+)"
> str_real_approx_sf format_uk 10 3 3 floor 2 ((-22)%7);
"-3.2(+)"
>
```

strs_real_approx_sf radix roundfun sf q: returns a tuple of strings

- sign,
- integer part,
- fraction part, representing an expansion to a number of significant figures, together with
- the rounding "error": a fraction representing the next numerals

join_str_real_approx: see join_str_real_approx.

18.1.3 Functions for Scientific Notation and Engineering Notation

str_real_approx_sci fmt radix intGroup fracGroup roundfun sf q: returns a string expansion with a number of significant figures in scientific notation, using rounding mode roundfun (see Rounding to Integer).

(sf must be positive; expStep is usually 3, radix is usually 10, roundfun is usually round or round_unbiased; str_real_approx_sci is equivalent to str_real_approx_eng (below) with expStep = 1.)

strs_real_approx_sci radix roundfun sf q: returns a tuple of strings:

- sign of mantissa,
- integer part of mantissa,
- fraction part of mantissa,
- sign of exponent,
- exponent magnitude

representing an expansion to a number of significant figures in scientific notation together with

- the rounding "error": a fraction representing the next numerals.

str_real_approx_eng fmt expStep radix intGroup fracGroup round sf q: returns a string expansion with a number of significant figures in engineering notation, using rounding mode roundfun.

The ExpStep parameter specifies the granularity of the exponent; specifically, the exponent will always be divisible by expStep.

(sf must be positive; expStep is usually 3 and must be positive, radix is usually 10, roundfun is usually round or round_unbiased.)

Example 81 Engineering notation:

```
> str_real_approx_eng format_uk 3 10 3 3 round 7 (rational 999950);
"999.950,0*10^3"
> str_real_approx_eng format_uk 3 10 3 3 round 4 999950;
"1.000*10^6(-)"
>
```

strs_real_approx_eng expStep radix roundfun sf q: returns a tuple of strings:

- sign of mantissa,
- integer part of mantissa,
- fraction part of mantissa,
- sign of exponent,
- exponent magnitude

representing an expansion to a number of significant figures in engineering notation together with

- the rounding "error": a fraction representing the next numerals.

Example 82 Engineering notation — the fragments:

```
> strs_real_approx_eng 3 10 round 7 (rational 999950);
("+","999","9500","+","3",0%1)
> strs_real_approx_eng 3 10 round 4 999950;
("+","1","000","+","6",-1%20)
>
```

join_str_real_eng fmt intGroup fracGroup mantSign mantI mantF rac expSign expI err: formats the parts in the quadruple returned by `strs_real_approx_eng` or strs_real_approx_sci to the sort of string as returned by str_real_approx_eng or *str_real_approx_sci*.

18.2 Evaluation of Expansion Strings

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between `rationals` and differently formatted strings if a suitable alternative format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

val_real_eng fmt radix strg: returns the `rational` q represented by the base-radix expansion `string` strg of one the forms

- "[−]int.frac"
- "[−]int.frace[−]exponent"

Example 83 Conversion from the recurring radix expansion-type string representations:

```
> val_real_eng format_uk 10 "-12.345";
-2469%200
> val_real_eng format_uk 10 "-12.345*10^2";
-2469%2
>
```

sval_real_eng radix signStr mantIStr mantF racStr expSignStr expStr: returns the `rational` q represented by the parts

- sign
- integer part of mantissa
- fraction part of mantissa
- sign of exponent
- exponent

:split_str_real_eng fmt strg — returns a tuple containing the string parts

- sign
- integer part of mantissa
- fraction part of mantissa
- sign of exponent
- exponent
- the "error" sign

of one the forms

- "[−]int.frac"
- "[−]int.frac ×10ˆ[−]exponent"

These functions can deal with the fixed decimal places, the significant figures and the scientific notation in addition to the engineering notation.

19 Numeral String → Q — Approximation

This section describes functions to approximate by a `rational` a real number represented by a string. See R → Q — Approximation for approximation by a `rational` of a `double`.

The `str_*` and `val_*` functions depend on a 'format structure' parameter (fmt) such as format uk. Conversions may be performed between `rationals` and differently formatted strings if a format structure is supplied. See Internationalisation and Format Structures for information about formatting structures.

:val_eng_approx_epsilon fmt radix epsilon strg Find the least complex `rational` approximation q to the number represented by the base-radix expansion `string` str in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10ˆ[−]exponent"

that is "-close. That is find a q such that |q − val str| ≤ .

Example 84 `rational` from a long string:

```
> let strg = "123.456,789,876,543,212,345,678,987,654,321*10^27";
> let x = val_real_eng format_uk 10 str;
> x;
123456789876543212345678987654321%1000
> let q = val_eng_approx_epsilon format_uk 10 (1%100) strg;
> q;
1975308638024691397530863802469%16
> double (x - q);
0.0085
> str_real_approx_eng format_uk 3 10 3 3 round 30 q;
"123.456,789,876,543,212,345,678,987,654*10^27(+)"
> str_real_approx_eng format_uk 3 10 3 3 round 42 q;
"123.456,789,876,543,212,345,678,987,654,312,500,000,000*10^27"
> double q;
1.23456789876543e+029
>
```

val_eng_interval_epsilon fmt radix epsilon strg: Find the least complex `rational` interval containing the number represented by the base-radix expansion `string` strg in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10ˆ[−]exponent"

that is "-small.

val_eng_approx_max_den fmt radix maxDen strg: Find the closest `rational` approximation to the number represented by the base-rRadix expansion `string` strg in one of the forms

- "[−]int.frac"
- "[−]int.frac ×10ˆ[−]exponent"

that has a denominator no greater than maxDen. (maxDen > 0)

val_eng_interval_max_den fmt radix maxDen strg: Find the smallest `rational` interval containing the number represented by the base-radix expansion `string` strg in one of the forms

- "[−]int.frac"
- "[−]int.frac $\times 10$ˆ[−]exponent"

that has endpoints with denominators no greater than maxDen. (maxDen > 0)

Example 85 Other `rationals` from a long string:

```
> val_eng_approx_epsilon format_uk 10 (1%100) strg;
19753086380246913975308638024 69%16
> val_eng_interval_epsilon format_uk 10 (1%100) strg;
interval::Ivl 3086419746913580308641974691358%25 3456790116543209945679011654321%28
> val_eng_approx_max_den format_uk 10 100 strg;
999999998000000001999999998000000%81
> val_eng_interval_max_den format_uk 10 100 strg;
interval::Ivl 999999998000000001999999998000000%81 3456790116543209945679011654321%28
>
```

20 Index