NAME

Math::Complex - complex numbers and associated mathematical functions

SYNOPSIS

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$t = 4 - 3*i + $z;
$j = cplxe(1, 2*pi/3);
```

DESCRIPTION

This package lets you create and manipulate complex numbers. By default, *Perl* limits itself to real numbers, but an extra `use` statement brings full complex support, along with a full set of mathematical functions typically associated with and/or extended to complex numbers.

If you wonder what complex numbers are, they were invented to be able to solve the following equation:

```
x*x = -1
```

and by definition, the solution is noted *i* (engineers use *j* instead since *i* usually denotes an intensity, but the name does not matter). The number *i* is a pure *imaginary* number.

The arithmetics with pure imaginary numbers works just like you would expect it with real numbers... you just have to remember that

```
i*i = -1
```

so you have:

```
5i + 7i = i * (5 + 7) = 12i
4i - 3i = i * (4 - 3) = i
4i * 2i = -8
6i / 2i = 3
1 / i = -i
```

Complex numbers are numbers that have both a real part and an imaginary part, and are usually noted:

```
a + bi
```

where a is the *real* part and b is the *imaginary* part. The arithmetic with complex numbers is straightforward. You have to keep track of the real and the imaginary parts, but otherwise the rules used for real numbers just apply:

```
(4 + 3i) + (5 - 2i) = (4 + 5) + i(3 - 2) = 9 + i
(2 + i) * (4 - i) = 2*4 + 4i -2i -i*i = 8 + 2i + 1 = 9 + 2i
```

A graphical representation of complex numbers is possible in a plane (also called the *complex plane*, but it's really a 2D plane). The number

```
z = a + bi
```

is the point whose coordinates are (a, b). Actually, it would be the vector originating from (0, 0) to (a, b). It follows that the addition of two complex numbers is a vectorial addition.

Since there is a bijection between a point in the 2D plane and a complex number (i.e. the mapping is unique and reciprocal), a complex number can also be uniquely identified with polar coordinates:

```
[rho, theta]
```

where `rho` is the distance to the origin, and `theta` the angle between the vector and the *x* axis. There is a notation for this using the exponential form, which is:

```
rho * exp(i * theta)
```

where *i* is the famous imaginary number introduced above. Conversion between this form and the cartesian form `a + bi` is immediate:

```
a = rho * cos(theta)
b = rho * sin(theta)
```

which is also expressed by this formula:

```
z = rho * exp(i * theta) = rho * (cos theta + i * sin theta)
```

In other words, it's the projection of the vector onto the *x* and *y* axes. Mathematicians call *rho* the *norm* or *modulus* and *theta* the *argument* of the complex number. The *norm* of `z` is marked here as `abs(z)`.

The polar notation (also known as the trigonometric representation) is much more handy for performing multiplications and divisions of complex numbers, whilst the cartesian notation is better suited for additions and subtractions. Real numbers are on the *x* axis, and therefore *y* or *theta* is zero or *pi*.

All the common operations that can be performed on a real number have been defined to work on complex numbers as well, and are merely *extensions* of the operations defined on real numbers. This means they keep their natural meaning when there is no imaginary part, provided the number is within their definition set.

For instance, the `sqrt` routine which computes the square root of its argument is only defined for non-negative real numbers and yields a non-negative real number (it is an application from **R+** to **R+**). If we allow it to return a complex number, then it can be extended to negative real numbers to become an application from **R** to **C** (the set of complex numbers):

```
sqrt(x) = x >= 0 ? sqrt(x) : sqrt(-x)*i
```

It can also be extended to be an application from **C** to **C**, whilst its restriction to **R** behaves as defined above by using the following definition:

```
sqrt(z = [r,t]) = sqrt(r) * exp(i * t/2)
```

Indeed, a negative real number can be noted `[x,pi]` (the modulus *x* is always non-negative, so `[x,pi]` is really `-x`, a negative number) and the above definition states that

```
sqrt([x,pi]) = sqrt(x) * exp(i*pi/2) = [sqrt(x),pi/2] = sqrt(x)*i
```

which is exactly what we had defined for negative real numbers above. The `sqrt` returns only one of the solutions: if you want the both, use the `root` function.

All the common mathematical functions defined on real numbers that are extended to complex numbers share that same property of working *as usual* when the imaginary part is zero (otherwise, it would not be called an extension, would it?).

A *new* operation possible on a complex number that is the identity for real numbers is called the *conjugate*, and is noted with a horizontal bar above the number, or ~z here.

```
 z = a + bi
~z = a - bi
```

Simple... Now look:

```
z * ~z = (a + bi) * (a - bi) = a*a + b*b
```

We saw that the norm of z was noted `abs(z)` and was defined as the distance to the origin, also known as:

```
rho = abs(z) = sqrt(a*a + b*b)
```

so

```
z * ~z = abs(z) ** 2
```

If z is a pure real number (i.e. `b == 0`), then the above yields:

```
a * a = abs(a) ** 2
```

which is true (`abs` has the regular meaning for real number, i.e. stands for the absolute value). This example explains why the norm of z is noted `abs(z)`: it extends the `abs` function to complex numbers, yet is the regular `abs` we know when the complex number actually has no imaginary part... This justifies *a posteriori* our use of the `abs` notation for the norm.

OPERATIONS

Given the following notations:

```
z1 = a + bi = r1 * exp(i * t1)
z2 = c + di = r2 * exp(i * t2)
z = <any complex or real number>
```

the following (overloaded) operations are supported on complex numbers:

```
z1 + z2 = (a + c) + i(b + d)
z1 - z2 = (a - c) + i(b - d)
z1 * z2 = (r1 * r2) * exp(i * (t1 + t2))
z1 / z2 = (r1 / r2) * exp(i * (t1 - t2))
z1 ** z2 = exp(z2 * log z1)
~z = a - bi
abs(z) = r1 = sqrt(a*a + b*b)
sqrt(z) = sqrt(r1) * exp(i * t/2)
exp(z) = exp(a) * exp(i * b)
log(z) = log(r1) + i*t
sin(z) = 1/2i (exp(i * z1) - exp(-i * z))
cos(z) = 1/2 (exp(i * z1) + exp(-i * z))
atan2(y, x) = atan(y / x) # Minding the right quadrant, note the order.
```

The definition used for complex arguments of atan2() is

```
       -i log((x + iy)/sqrt(x*x+y*y))
```

Note that atan2(0, 0) is not well-defined.

The following extra operations are supported on both real and complex numbers:

```
Re(z) = a
Im(z) = b
arg(z) = t
abs(z) = r

cbrt(z) = z ** (1/3)
log10(z) = log(z) / log(10)
logn(z, n) = log(z) / log(n)

tan(z) = sin(z) / cos(z)

csc(z) = 1 / sin(z)
sec(z) = 1 / cos(z)
cot(z) = 1 / tan(z)

asin(z) = -i * log(i*z + sqrt(1-z*z))
acos(z) = -i * log(z + i*sqrt(1-z*z))
atan(z) = i/2 * log((i+z) / (i-z))

acsc(z) = asin(1 / z)
asec(z) = acos(1 / z)
acot(z) = atan(1 / z) = -i/2 * log((i+z) / (z-i))

sinh(z) = 1/2 (exp(z) - exp(-z))
cosh(z) = 1/2 (exp(z) + exp(-z))
tanh(z) = sinh(z) / cosh(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-z))

csch(z) = 1 / sinh(z)
sech(z) = 1 / cosh(z)
coth(z) = 1 / tanh(z)

asinh(z) = log(z + sqrt(z*z+1))
acosh(z) = log(z + sqrt(z*z-1))
atanh(z) = 1/2 * log((1+z) / (1-z))

acsch(z) = asinh(1 / z)
asech(z) = acosh(1 / z)
acoth(z) = atanh(1 / z) = 1/2 * log((1+z) / (z-1))
```

arg, *abs*, *log*, *csc*, *cot*, *acsc*, *acot*, *csch*, *coth*, *acosech*, *acotanh*, have aliases *rho*, *theta*, *ln*, *cosec*, *cotan*, *acosec*, *acotan*, *cosech*, *cotanh*, *acosech*, *acotanh*, respectively. `Re`, `Im`, `arg`, `abs`, `rho`, and `theta` can be used also as mutators. The `cbrt` returns only one of the solutions: if you want all three, use the `root` function.

The *root* function is available to compute all the *n* roots of some complex, where *n* is a strictly positive integer. There are exactly *n* such roots, returned as a list. Getting the number mathematicians call `j` such that:

```
1 + j + j*j = 0;
```

is a simple matter of writing:

```
$j = ((root(1, 3))[1];
```

The *k*th root for `z = [r,t]` is given by:

```
(root(z, n))[k] = r**(1/n) * exp(i * (t + 2*k*pi)/n)
```

You can return the *k*th root directly by `root(z, n, k)`, indexing starting from *zero* and ending at *n - 1*.

The *spaceship* numeric comparison operator, <=>, is also defined. In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match.

CREATION

To create a complex number, use either:

```
$z = Math::Complex->make(3, 4);
$z = cplx(3, 4);
```

if you know the cartesian form of the number, or

```
$z = 3 + 4*i;
```

if you like. To create a number using the polar form, use either:

```
$z = Math::Complex->emake(5, pi/3);
$x = cplxe(5, pi/3);
```

instead. The first argument is the modulus, the second is the angle (in radians, the full circle is 2*pi). (Mnemonic: e is used as a notation for complex numbers in the polar form).

It is possible to write:

```
$x = cplxe(-3, pi/4);
```

but that will be silently converted into `[3,-3pi/4]`, since the modulus must be non-negative (it represents the distance to the origin in the complex plane).

It is also possible to have a complex number as either argument of the `make`, `emake`, `cplx`, and `cplxe`: the appropriate component of the argument will be used.

```
$z1 = cplx(-2,  1);
$z2 = cplx($z1, 4);
```

The `new`, `make`, `emake`, `cplx`, and `cplxe` will also understand a single (string) argument of the forms

```
    2-3i
    -3i
[2,3]
[2,-3pi/4]
[2]
```

in which case the appropriate cartesian and exponential components will be parsed from the string and used to create new complex numbers. The imaginary component and the theta, respectively, will default to zero.

The `new`, `make`, `emake`, `cplx`, and `cplxe` will also understand the case of no arguments: this means

plain zero or (0, 0).

DISPLAYING

When printed, a complex number is usually shown under its cartesian style *a+bi*, but there are legitimate cases where the polar style *[r,t]* is more appropriate. The process of converting the complex number into a string that can be displayed is known as *stringification*.

By calling the class method `Math::Complex::display_format` and supplying either `"polar"` or `"cartesian"` as an argument, you override the default display style, which is `"cartesian"`. Not supplying any argument returns the current settings.

This default can be overridden on a per-number basis by calling the `display_format` method instead. As before, not supplying any argument returns the current display style for this number. Otherwise whatever you specify will be the new display style for *this* particular number.

For instance:

```
use Math::Complex;

Math::Complex::display_format('polar');
$j = (root(1, 3))[1];
print "j = $j\n";  # Prints "j = [1,2pi/3]"
$j->display_format('cartesian');
print "j = $j\n";  # Prints "j = -0.5+0.866025403784439i"
```

The polar style attempts to emphasize arguments like *k*pi/n* (where *n* is a positive integer and *k* an integer within [-9, +9]), this is called *polar pretty-printing*.

For the reverse of stringifying, see the `make` and `emake`.

CHANGED IN PERL 5.6

The `display_format` class method and the corresponding `display_format` object method can now be called using a parameter hash instead of just a one parameter.

The old display format style, which can have values `"cartesian"` or `"polar"`, can be changed using the `"style"` parameter.

```
$j->display_format(style => "polar");
```

The one parameter calling convention also still works.

```
$j->display_format("polar");
```

There are two new display parameters.

The first one is `"format"`, which is a sprintf()-style format string to be used for both numeric parts of the complex number(s). The is somewhat system-dependent but most often it corresponds to `"%.15g"`. You can revert to the default by setting the `format` to `undef`.

```
# the $j from the above example

$j->display_format('format' => '%.5f');
print "j = $j\n";  # Prints "j = -0.50000+0.86603i"
$j->display_format('format' => undef);
print "j = $j\n";  # Prints "j = -0.5+0.86603i"
```

Notice that this affects also the return values of the `display_format` methods: in list context the

whole parameter hash will be returned, as opposed to only the style parameter value. This is a potential incompatibility with earlier versions if you have been calling the `display_format` method in list context.

The second new display parameter is `"polar_pretty_print"`, which can be set to true or false, the default being true. See the previous section for what this means.

USAGE

Thanks to overloading, the handling of arithmetics with complex numbers is simple and almost transparent.

Here are some examples:

```
use Math::Complex;

$j = cplxe(1, 2*pi/3); # $j ** 3 == 1
print "j = $j, j**3 = ", $j ** 3, "\n";
print "1 + j + j**2 = ", 1 + $j + $j**2, "\n";

$z = -16 + 0*i;   # Force it to be a complex
print "sqrt($z) = ", sqrt($z), "\n";

$k = exp(i * 2*pi/3);
print "$j - $k = ", $j - $k, "\n";

$z->Re(3);   # Re, Im, arg, abs,
$j->arg(2);   # (the last two aka rho, theta)
    # can be used also as mutators.
```

CONSTANTS

PI

The constant `pi` and some handy multiples of it (pi2, pi4, and pip2 (pi/2) and pip4 (pi/4)) are also available if separately exported:

```
use Math::Complex ':pi';
$third_of_circle = pi2 / 3;
```

Inf

The floating point infinity can be exported as a subroutine Inf():

```
use Math::Complex qw(Inf sinh);
my $AlsoInf = Inf() + 42;
my $AnotherInf = sinh(1e42);
print "$AlsoInf is $AnotherInf\n" if $AlsoInf == $AnotherInf;
```

Note that the stringified form of infinity varies between platforms: it can be for example any of

```
inf
infinity
INF
1.#INF
```

or it can be something else.

Also note that in some platforms trying to use the infinity in arithmetic operations may result in Perl

crashing because using an infinity causes SIGFPE or its moral equivalent to be sent. The way to ignore this is

```
local $SIG{FPE} = sub { };
```

ERRORS DUE TO DIVISION BY ZERO OR LOGARITHM OF ZERO

The division (/) and the following functions

```
log ln log10 logn
tan sec csc cot
atan asec acsc acot
tanh sech csch coth
atanh asech acsch acoth
```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be `0` (zero). For the logarithmic functions and the `atanh`, `acoth`, the argument cannot be `1` (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `atan`, `acot`, the argument cannot be `i` (the imaginary unit). For the `atan`, `acoth`, the argument cannot be `-i` (the negative imaginary unit). For the `tan`, `sec`, `tanh`, the argument cannot be *pi/2 + k * pi*, where *k* is any integer. atan2(0, 0) is undefined, and if the complex arguments are used for atan2(), a division by zero will happen if z1**2+z2**2 == 0.

Note that because we are operating on approximations of real numbers, these errors can happen when merely `too close' to the singularities listed above.

ERRORS DUE TO INDIGESTIBLE ARGUMENTS

The `make` and `emake` accept both real and complex arguments. When they cannot recognize the arguments they will die with error messages like the following

```
Math::Complex::make: Cannot take real part of ...
Math::Complex::make: Cannot take real part of ...
Math::Complex::emake: Cannot take rho of ...
Math::Complex::emake: Cannot take theta of ...
```

BUGS

Saying `use Math::Complex;` exports many mathematical routines in the caller environment and even overrides some (`sqrt`, `log`, `atan2`). This is construed as a feature by the Authors, actually... ;-)

All routines expect to be given real or complex numbers. Don't attempt to use BigFloat, since Perl has currently no rule to disambiguate a '+' operation (for instance) between two overloaded entities.

In Cray UNICOS there is some strange numerical instability that results in root(), cos(), sin(), cosh(), sinh(), losing accuracy fast. Beware. The bug may be in UNICOS math libs, in UNICOS C compiler, in Math::Complex. Whatever it is, it does not manifest itself anywhere else where Perl runs.

SEE ALSO

Math::Trig

AUTHORS

Daniel S. Lewart <*lewart!at!uiuc.edu*>, Jarkko Hietaniemi <*jhi!at!iki.fi*>, Raphael Manfredi <*Raphael_Manfredi!at!pobox.com*>, Zefram <zefram@fysh.org>

LICENSE