NAME

Math::Trig - trigonometric functions

SYNOPSIS

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$halfpi = pi/2;

$rad = deg2rad(120);
```

DESCRIPTION

`Math::Trig` defines many trigonometric functions not defined by the core Perl which defines only the `sin()` and `cos()`. The constant **pi** is also defined as are a few convenience functions for angle conversions.

TRIGONOMETRIC FUNCTIONS

The tangent

tan

The cofunctions of the sine, cosine, and tangent (cosec/csc and cotan/cot are aliases)

csc, **cosec**, **sec**, **sec**, **cot**, **cotan**

The arcus (also known as the inverse) functions of the sine, cosine, and tangent

asin, **acos**, **atan**

The principal value of the arc tangent of y/x

atan2(y, x)

The arcus cofunctions of the sine, cosine, and tangent (acosec/acsc and acotan/acot are aliases)

acsc, **acosec**, **asec**, **acot**, **acotan**

The hyperbolic sine, cosine, and tangent

sinh, **cosh**, **tanh**

The cofunctions of the hyperbolic sine, cosine, and tangent (cosech/csch and cotanh/coth are aliases)

csch, **cosech**, **sech**, **coth**, **cotanh**

The arcus (also known as the inverse) functions of the hyperbolic sine, cosine, and tangent

asinh, **acosh**, **atanh**

The arcus cofunctions of the hyperbolic sine, cosine, and tangent (acsch/acosech and acoth/acotanh are aliases)

acsch, **acosech**, **asech**, **acoth**, **acotanh**

The trigonometric constant **pi** is also defined.

$pi2 = 2 * **pi**;

ERRORS DUE TO DIVISION BY ZERO

The following functions

```
acoth
acsc
acsch
asec
asech
atanh
cot
coth
csc
csch
sec
sech
tan
tanh
```

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 `atanh`, `acoth`, the argument cannot be `1` (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `tan`, `sec`, `tanh`, `sech`, the argument cannot be *pi/2 + k * pi*, where *k* is any integer.

SIMPLE (REAL) ARGUMENTS, COMPLEX RESULTS

Please note that some of the trigonometric functions can break out from the **real axis** into the **complex plane**. For example `asin(2)` has no definition for plain real numbers but it has definition for complex numbers.

In Perl terms this means that supplying the usual Perl numbers (also known as scalars, please see *perldata*) as input for the trigonometric functions might produce as output results that no more are simple real numbers: instead they are complex numbers.

The `Math::Trig` handles this by using the `Math::Complex` package which knows how to handle complex numbers, please see *Math::Complex* for more information. In practice you need not to worry about getting complex numbers as results because the `Math::Complex` takes care of details like for example how to display complex numbers. For example:

```
print asin(2), "\n";
```

should produce something like this (take or leave few last decimals):

```
1.5707963267949-1.31695789692482i
```

That is, a complex number with the real part of approximately `1.571` and the imaginary part of

approximately `-1.317`.

PLANE ANGLE CONVERSIONS

(Plane, 2-dimensional) angles may be converted with the following functions.

```
$radians  = deg2rad($degrees);
$radians  = grad2rad($gradians);

$degrees  = rad2deg($radians);
$degrees  = grad2deg($gradians);

$gradians = deg2grad($degrees);
$gradians = rad2grad($radians);
```

The full circle is 2 *pi* radians or *360* degrees or *400* gradians. The result is by default wrapped to be inside the [0, {2pi,360,400}[circle. If you don't want this, supply a true second argument:

```
$zillions_of_radians  = deg2rad($zillions_of_degrees, 1);
$negative_degrees     = rad2deg($negative_radians, 1);
```

You can also do the wrapping explicitly by rad2rad(), deg2deg(), and grad2grad().

RADIAL COORDINATE CONVERSIONS

Radial coordinate systems are the **spherical** and the **cylindrical** systems, explained shortly in more detail.

You can import radial coordinate conversion functions by using the `:radial` tag:

```
    use Math::Trig ':radial';

    ($rho, $theta, $z)     = cartesian_to_cylindrical($x, $y, $z);
    ($rho, $theta, $phi)   = cartesian_to_spherical($x, $y, $z);
    ($x, $y, $z)           = cylindrical_to_cartesian($rho, $theta, $z);
    ($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta, $z);
    ($x, $y, $z)           = spherical_to_cartesian($rho, $theta, $phi);
    ($rho_c, $theta, $z)   = spherical_to_cylindrical($rho_s, $theta,
$phi);
```

All angles are in radians.

COORDINATE SYSTEMS

Cartesian coordinates are the usual rectangular *(x, y, z)*-coordinates.

Spherical coordinates, *(rho, theta, pi)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a sphere surface. The radius of the sphere is **rho**, also known as the *radial* coordinate. The angle in the *xy*-plane (around the *z*-axis) is **theta**, also known as the *azimuthal* coordinate. The angle from the *z*-axis is **phi**, also known as the *polar* coordinate. The `North Pole' is therefore *0, 0, rho*, and the `Bay of Guinea' (think of the missing big chunk of Africa) *0, pi/2, rho*. In geographical terms *phi* is latitude (northward positive, southward negative) and *theta* is longitude (eastward positive, westward negative).

BEWARE: some texts define *theta* and *phi* the other way round, some texts define the *phi* to start from the horizontal plane, some texts use *r* in place of *rho*.

Cylindrical coordinates, *(rho, theta, z)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a cylinder surface. The radius of the cylinder is **rho**, also

known as the *radial* coordinate. The angle in the *xy*-plane (around the *z*-axis) is **theta**, also known as the *azimuthal* coordinate. The third coordinate is the *z*, pointing up from the **theta**-plane.

3-D ANGLE CONVERSIONS

Conversions to and from spherical and cylindrical coordinates are available. Please notice that the conversions are not necessarily reversible because of the equalities like *pi* angles being equal to *-pi* angles.

cartesian_to_cylindrical

```
            ($rho, $theta, $z) = cartesian_to_cylindrical($x, $y, $z);
```

cartesian_to_spherical

```
            ($rho, $theta, $phi) = cartesian_to_spherical($x, $y, $z);
```

cylindrical_to_cartesian

```
            ($x, $y, $z) = cylindrical_to_cartesian($rho, $theta, $z);
```

cylindrical_to_spherical

```
            ($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c,
    $theta, $z);
```

Notice that when `$z` is not 0 `$rho_s` is not equal to `$rho_c`.

spherical_to_cartesian

```
            ($x, $y, $z) = spherical_to_cartesian($rho, $theta, $phi);
```

spherical_to_cylindrical

```
            ($rho_c, $theta, $z) = spherical_to_cylindrical($rho_s,
    $theta, $phi);
```

Notice that when `$z` is not 0 `$rho_c` is not equal to `$rho_s`.

GREAT CIRCLE DISTANCES AND DIRECTIONS

You can compute spherical distances, called **great circle distances**, by importing the great_circle_distance() function:

```
  use Math::Trig 'great_circle_distance';

  $distance = great_circle_distance($theta0, $phi0, $theta1, $phi1, [,
$rho]);
```

The *great circle distance* is the shortest distance between two points on a sphere. The distance is in `$rho` units. The `$rho` is optional, it defaults to 1 (the unit sphere), therefore the distance defaults to radians.

If you think geographically the *theta* are longitudes: zero at the Greenwhich meridian, eastward positive, westward negative--and the *phi* are latitudes: zero at the North Pole, northward positive, southward negative. **NOTE**: this formula thinks in mathematics, not geographically: the *phi* zero is at the North Pole, not at the Equator on the west coast of Africa (Bay of Guinea). You need to subtract your geographical coordinates from *pi/2* (also known as 90 degrees).

```
  $distance = great_circle_distance($lon0, pi/2 - $lat0,
                                    $lon1, pi/2 - $lat1, $rho);
```

The direction you must follow the great circle can be computed by the great_circle_direction() function:

```
  use Math::Trig 'great_circle_direction';

  $direction = great_circle_direction($theta0, $phi0, $theta1, $phi1);
```

The result is in radians, zero indicating straight north, pi or -pi straight south, pi/2 straight west, and -pi/2 straight east.

Notice that the resulting directions might be somewhat surprising if you are looking at a flat worldmap: in such map projections the great circles quite often do not look like the shortest routes-- but for example the shortest possible routes from Europe or North America to Asia do often cross the polar regions.

EXAMPLES

To calculate the distance between London (51.3N 0.5W) and Tokyo (35.7N 139.8E) in kilometers:

```
        use Math::Trig qw(great_circle_distance deg2rad);

        # Notice the 90 - latitude: phi zero is at the North Pole.
@L = (deg2rad(-0.5), deg2rad(90 - 51.3));
        @T = (deg2rad(139.8),deg2rad(90 - 35.7));

        $km = great_circle_distance(@L, @T, 6378);
```

The direction you would have to go from London to Tokyo

```
        use Math::Trig qw(great_circle_direction);

        $rad = great_circle_direction(@L, @T);
```

CAVEAT FOR GREAT CIRCLE FORMULAS

The answers may be off by few percentages because of the irregular (slightly aspherical) form of the Earth. The formula used for grear circle distances

```
lat0 = 90 degrees - phi0
lat1 = 90 degrees - phi1
d = R * arccos(cos(lat0) * cos(lat1) * cos(lon1 - lon01) +
                     sin(lat0) * sin(lat1))
```

is also somewhat unreliable for small distances (for locations separated less than about five degrees) because it uses arc cosine which is rather ill-conditioned for values close to zero.

BUGS

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

The code is not optimized for speed, especially because we use `Math::Complex` and thus go quite near complex numbers while doing the computations even when the arguments are not. This, however, cannot be completely avoided if we want things like `asin(2)` to give an answer instead of giving a fatal runtime error.

AUTHORS

Jarkko Hietaniemi <*jhi@iki.fi*> and Raphael Manfredi <*Raphael_Manfredi@pobox.com*>.