Class to do math on complex numbers.

/* ——————————————————————————
*
* License
*
* The contents of this file are subject to the Jabber Open Source License
* Version 1.0 (the “License”). You may not copy or use this file, in either
* source code or executable form, except in compliance with the License. You
* may obtain a copy of the License at http://www.jabber.com/license/ or at
* http://www.opensource.org/.
*
* Software distributed under the License is distributed on an “AS IS” basis,
* WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License
* for the specific language governing rights and limitations under the
* License.
*
* Copyrights
*

```
*
* Acknowledgements
*
* Special thanks to the Jabber Open Source Contributors for their
* suggestions and support of Jabber.
*
* ————————————————————————*/
namespace bedrock.util
{
using System;
///

/// Class to do math on complex numbers. Lots of optimizations, many from
/// the numerical methods literature. Sorry, but I've lost the citations by now.
///
public class Complex : IFormattable
{
private double m_real;
private double m_imag;

// Double.Epsilon is too small
private static double s_tolerance = 1E-15;

///

/// Create a complex number from a real part and an imaginary part.
/// Both parts use double-precision.
///
/// Real part /// Imaginary part. Multiplied by “i” and added to real. public Complex(double real, double imag)
{
m_real = real;
m_imag = imag;
}
///
```

```
/// Complex number with imaginary part of 0.
///
/// Real part public Complex(double real) : this(real, 0.0)
{
}
///

/// Create a complex number from a polar representation.
///
/// The magnitude of the polar representation /// The angle,
in radians, of the polar representation public static Complex
Polar(double magnitude, double radianAngle)
{
return new Complex(magnitude * Math.Cos(radianAngle),
magnitude * Math.Sin(radianAngle));
}

///

/// The real part of the complex number
///
public double Real
{
get { return m_real; }
set { m_real = value; }
}

///

/// The imaginary part of the complex number
///
public double Imaginary
{
get { return m_imag; }
set { m_imag = value; }
}
///

/// Get a new complex number that is the conjugate (Imaginary
*= -1) of the current.
```

```
///
public Complex Conjugate()
{
return new Complex(m_real, -m_imag);
}
///

/// Return the absoulte value of the complex number.
///
public double Abs()
{
return Abs(m_real, m_imag);
}
///

/// sqrt(first^2 + second^2), with optimizations
///
/// first number /// second number private static double
Abs(double first, double second)
{
// avoid double math wherever possible…
//return Math.Sqrt((first * first) + (second * second));
first = Math.Abs(first);
second = Math.Abs(second);

if (first == 0d)
{
return second;
}
if (second == 0d)
{
return first;
}
if (first > second)
{
double temp = second / first;
return first * Math.Sqrt(1d + (temp * temp));
}
```

```
else
{
double temp = first / second;
return second * Math.Sqrt(1d + (temp * temp));
}
}

///

/// Angle, in radians, of the current value.
///
public double Arg()
{
return Math.Atan2(m_imag, m_real);
}

///

/// The square root of the current value.
///
public Complex Sqrt()
{
//return Math.Sqrt(this.Abs()) *
// new Complex( Math.Cos(this.Arg()/2),
// Math.Sin(this.Arg()/2));
if ((m_real == 0d) && (m_imag == 0d))
{
return new Complex(0d, 0d);
}
else
{
double ar = Math.Abs(m_real);
double ai = Math.Abs(m_imag);
double temp;
double w;

if (ar >= ai)
{
temp = ai / ar;
```

```
w = Math.Sqrt(ar) *
Math.Sqrt(0.5d * (1d + Math.Sqrt(1d + (temp * temp))));
}
else
{
temp = ar / ai;
w = Math.Sqrt(ai) *
Math.Sqrt(0.5d * (temp + Math.Sqrt(1d + (temp * temp))));
}
if (m_real > 0d)
{
return new Complex(w, m_imag / (2d * w));
}
else
{
double r = (m_imag >= 0d) ? w : -w;
return new Complex(r, m_imag / (2d * r));
}
}
}
///

/// Raise the current value to a power.
///
/// The power to raise to. public Complex Pow(double exponent)
{
double real = exponent * Math.Log(this.Abs());
double imag = exponent * this.Arg();
double scalar = Math.Exp(real);
return new Complex(scalar * Math.Cos(imag), scalar *
Math.Sin(imag));
}
///

/// Raise the current value to a power.
///
/// The power to raise to. public Complex Pow(Complex
exponent)
```

```
{
double real = Math.Log(this.Abs());
double imag = this.Arg();
double r2 = (real * exponent.m_real) – (imag * exponent.m_imag);
double i2 = (real * exponent.m_imag) + (imag * exponent.m_real);
double scalar = Math.Exp(r2);
return new Complex(scalar * Math.Cos(i2), scalar * Math.Sin(i2));
}

///

/// Returns e raised to the specified power.
///
public Complex Exp()
{
return Math.Exp(m_real) *
new Complex( Math.Cos(m_imag), Math.Sin(m_imag));
}
///

/// 1 / (the current value)
///
public Complex Inverse()
{
double scalar;
double ratio;

if (Math.Abs(m_real) >= Math.Abs(m_imag))
{
ratio = m_imag / m_real;
scalar = 1d / (m_real + m_imag * ratio);
return new Complex(scalar, -scalar * ratio);
}
else
{
```

```
ratio = m_real / m_imag;
scalar = 1d / (m_real * ratio + m_imag);
return new Complex(scalar * ratio, -scalar);
}
}

///

/// Returns the natural (base e) logarithm of the current
value.
///
public Complex Log()
{
return new Complex(Math.Log(this.Abs()), this.Arg());
}
///

/// Returns the sine of the current value.
///
public Complex Sin()
{
Complex iz = this * Complex.i;
Complex izn = -iz;
return (iz.Exp() – izn.Exp()) / new Complex(0,2);
}
///

/// Returns the cosine of the current value.
///
public Complex Cos()
{
Complex iz = this * Complex.i;
Complex izn = -iz;
return (iz.Exp() + izn.Exp()) / 2.0;
}
///

/// Returns the tangent of the current value.
///
```

```
public Complex Tan()
{
return this.Sin() / this.Cos();
}
///

/// Returns the hyperbolic sin of the current value.
///
public Complex Sinh()
{
return (this.Exp() – (-this).Exp()) / 2d;
}
///

/// Returns the hyperbolic cosine of the current value.
///
public Complex Cosh()
{
return (this.Exp() + (-this).Exp()) / 2d;
}
///

/// Returns the hyperbolic tangent of the current value.
///
public Complex Tanh()
{
return this.Sinh() / this.Cosh();
}
///

/// Returns the arc sine of the current value.
///
public Complex Asin()
{
// TODO: if anyone cares about this function, some of it
// should probably be inlined and streamlined.
Complex I = i;
return -I * ((this*I) + (1 – (this * this)).Sqrt()).Log();
}
```

```
///

/// Returns the arc cosine of the current value.
///
public Complex Acos()
{
// TODO: if anyone cares about this function, some of it
// should probably be inlined and streamlined.
Complex I = i;
return -I * (this + I * (1 – (this*this)).Sqrt()).Log();
}
///

/// Returns the arc tangent of the current value.
///
public Complex Atan()
{
// TODO: if anyone cares about this function, some of it
// should probably be inlined and streamlined.
Complex I = i;
return -I/2 * ((I – this)/(I + this)).Log();
}
///

/// Returns the arc hyperbolic sine of the current value.
///
public Complex Asinh()
{
return (this + ((this*this) + 1).Sqrt()).Log();
}

///

/// Returns the arc hyperbolic cosine of the current value.
///
public Complex Acosh()
{
return 2d * (((this+1d) / 2d).Sqrt() +
((this-1) / 2d).Sqrt()).Log();
```

```
// Gar. This one didn't work. Perhaps it isn't returning the
// “pricipal” value.
//return (this + ((this*this) – 1).Sqrt()).Log();
}
///

/// Returns the arc hyperbolic tangent of the current value.
///
public Complex Atanh()
{
return ((1+this) / (1-this)).Log() / 2d;
}

///

/// Is the current value Not a Number?
///
public bool IsNaN()
{
return Double.IsNaN(m_real) || Double.IsNaN(m_imag);
}
///

/// Is the current value infinite?
///
public bool IsInfinity()
{
return Double.IsInfinity(m_real) || Double.IsInfinity(m_imag);
}

///

///
///
///
public override int GetHashCode()
{
return ((int)m_imag < /// Format as string like “x + yi”.
///
```

```
public override string ToString()
{
return this.ToString(null, null);
}

///

///
///
///  ///  ///
public string ToString(string format, IFormatProvider sop)
{
if (this.IsNaN())
return “NaN”;
if (this.IsInfinity())
return “Infinity”;

if (m_imag == 0d)
return m_real.ToString(format, sop);
if (m_real == 0d)
return m_imag.ToString(format, sop) + “i”;
if (m_imag < 0.0) { return m_real.ToString(format, sop) + " -
" + (-m_imag).ToString(format, sop) + "i"; } return
m_real.ToString(format, sop) + " + " + m_imag.ToString(format,
sop) + "i"; } ///
/// Do a half-assed job of assessing equality, using the
current Tolerance value.
/// Will work with other Complex numbers or doubles.
///

/// The other object to compare against. Must be double or
Complex. public override bool Equals(object other)
{
if (other is Complex)
{
Complex o = (Complex) other;
// performance optimization for “identical” numbers”
if ((o.m_real == m_real) && (o.m_imag == m_imag))
```

```
return true;
return Equals(o, s_tolerance);
}
double d = (double) other; // can fire exception
if (m_imag != 0.0)
return false;
return Math.Abs(m_real – d) < s_tolerance; } ///
/// Is this number within a tolerance of being equal to
another Complex number?
///

/// The other Complex to comapare against. /// The tolerance
to be within. public bool Equals(Complex other, double
tolerance)
{
return (this – other).Abs() < tolerance; } ///
/// Calls Equals().
///

/// Complex /// Complex public static bool operator==(Complex
first, Complex second)
{
return first.Equals(second);
}
///

/// Calls !Equals().
///
/// Complex /// Complex public static bool operator!=(Complex
first, Complex second)
{
return !first.Equals(second);
}

///

/// Adds two complex numbers.
///
/// Complex /// Complex public static Complex
```

```
operator+(Complex first, Complex second)
{
return new Complex(first.m_real + second.m_real,
first.m_imag + second.m_imag);
}
///

/// Subtracts two complex numbers.
///
/// Complex /// Complex public static Complex operator-
(Complex first, Complex second)
{
return new Complex(first.m_real – second.m_real,
first.m_imag – second.m_imag);
}
///

/// Negates a complex number.
///
/// Complex public static Complex operator-(Complex first)
{
return new Complex(-first.m_real, -first.m_imag);
}
///

/// Multiplies two complex numbers.
///
/// Complex /// Complex public static Complex
operator*(Complex first, Complex second)
{
return new Complex((first.m_real * second.m_real) –
(first.m_imag * second.m_imag),
(first.m_real * second.m_imag) +
(first.m_imag * second.m_real));
}
///

/// Multiplies a complex number and a Complex.
///
```

```
/// Complex /// double public static Complex operator*(Complex first, double second)
{
return new Complex(first.m_real * second, first.m_imag * second);
}
///

/// Divides two Complex numbers.
///
/// Complex /// Complex public static Complex operator/(Complex first, Complex second)
{
//return (first * second.Conjugate()) /
// ((second.m_real * second.m_real) +
// (second.m_imag * second.m_imag));
double scalar;
double ratio;

if (Math.Abs(second.m_real) >= Math.Abs(second.m_imag))
{
ratio = second.m_imag / second.m_real;
scalar = 1d / (second.m_real + (second.m_imag * ratio));
return new Complex(scalar * (first.m_real + (first.m_imag*ratio)),
scalar * (first.m_imag – (first.m_real*ratio)));
}
else
{
ratio = second.m_real / second.m_imag;
scalar = 1d / ((second.m_real * ratio) + second.m_imag);
return new Complex(scalar * (first.m_real*ratio + first.m_imag),
scalar * (first.m_imag*ratio – first.m_real));
}
}
///
```

```
/// Divides a Complex number by a double.
///
/// Complex /// double public static Complex operator/(Complex
first, double second)
{
return new Complex(first.m_real / second, first.m_imag /
second);
}
///

/// Converts a double to a real Complex number.
///
/// Real part public static implicit operator Complex(double
real)
{
return new Complex(real);
}
///

/// Constant for sqrt(-1).
///
public static Complex i
{
get { return new Complex(0, 1); }
}
///

/// Tolerance value for Equals().
///
public static double Tolerance
{
get { return s_tolerance; }
set
{
if (value <= 0) throw new ArgumentOutOfRangeException
("Tolerance must be greater than 0"); s_tolerance = value; } }
} } [/csharp]
```