Java's new math, Part 1: Real numbers

Elliotte Rusty Harold 28 October 2008

Join Elliotte Rusty Harold for a look into "new" features in the classic `java.lang.Math` class in this two-part article. Part 1 focuses on more purely mathematical functions. Part 2 will explore the functions designed for operating on floating-point numbers.

View more content in this series

Sometimes you're so familiar with a class you stop paying attention to it. If you could write the documentation for `java.lang.Foo`, and Eclipse will helpfully autocomplete the functions for you, why would you ever need to read its Javadoc? Such was my experience with `java.lang.Math`, a class I thought I knew really, really well. Imagine my surprise, then, when I recently happened to be reading its Javadoc for possibly the first time in half a decade and realized that the class had almost doubled in size with 20 new methods I'd never heard of. Obviously it was time to take another look.

Version 5 of the Java™ Language Specification added 10 new methods to `java.lang.Math` (and its evil twin `java.lang.StrictMath`), and Java 6 added another 10. In this article, I focus on the more purely mathematical functions provided, such as `log10` and `cosh`. In Part 2, I'll explore the functions more designed for operating on floating point numbers as opposed to abstract real numbers.

The distinction between an abstract real number such as π or 0.2 and a Java `double` is an important one. First of all, the Platonic ideal of the number is infinitely precise, while the Java representation is limited to a fixed number of bits. This is important when you deal with very large and very small numbers. For example, the number 2,000,000,001 (two billion and one) can be represented exactly as an `int`, but not as a `float`. The closest you can get in a float is 2.0E9 — that is, two billion. `double`s do better because they have more bits (which is one reason you should almost always use `double`s instead of `float`s); but there are still practical limits to how accurate they can be.

The second limitation of computer arithmetic (the Java language's and others') is that it's based on binary rather than decimal. Fractions such as 1/5 and 7/50 that can be represented exactly in decimal (0.2 and 0.14, respectively) become repeating fractions when expressed in binary notation. This is exactly like the way 1/3 becomes 0.3333333... when expressed in decimal. In base 10, any fraction whose denominator has the prime factors 5 and 2 (and no others) is exactly

expressible. In base 2, only fractions whose denominators are powers of 2 are exactly expressible: 1/2, 1/4, 1/8, 1/16, and so on.

These imprecisions are one of the big reasons a math class is needed in the first place. Certainly you could define the trigonometric and other functions with Taylor series expansions using nothing more than the standard + and * operators and a simple loop, as shown in Listing 1:

Listing 1. Calculating sines with a Taylor series

```
public class SineTaylor {

    public static void main(String[] args) {
        for (double angle = 0; angle <= 4*Math.PI; angle += Math.PI/8) {
            System.out.println(degrees(angle) + "\t" + taylorSeriesSine(angle)
                    + "\t" + Math.sin(angle));
        }
    }

    public static double degrees(double radians) {
        return 180 * radians/ Math.PI;
    }

    public static double taylorSeriesSine(double radians) {
        double sine = 0;
        int sign = 1;
        for (int i = 1; i < 40; i+=2) {
            sine += Math.pow(radians, i) * sign / factorial(i);
            sign *= -1;
        }
        return sine;
    }

    private static double factorial(int i) {
        double result = 1;
        for (int j = 2; j <= i; j++)  {
            result *= j;
        }
        return result;
    }
}
```

This starts off well enough with only a small difference, if that, in the last decimal place:

```
0.0       0.0       0.0
22.5       0.3826834323650897       0.3826834323650898
45.0       0.7071067811865475       0.7071067811865475
67.5       0.923879532511287        0.9238795325112867
90.0       1.0000000000000002       1.0
```

However, as the angles increase, the errors begin to accumulate, and the naive approach no longer works so well:

```
630.0000000000003       -1.0000001371557132       -1.0
652.5000000000005       -0.9238801080153761       -0.9238795325112841
675.0000000000005       -0.7071090807463408       -0.7071067811865422
697.5000000000006       -0.3826922100671368       -0.3826834323650824
```

The Taylor series here actually proved more accurate than I expected. However as the angle increases to 360 degrees, 720 degrees (4 pi radians), and more, the Taylor series requires

progressively more terms for accurate computation. The more sophisticated algorithms used by `java.lang.Math` avoid this.

The Taylor series is also inefficient compared to the built-in sine function of a modern desktop chip. Proper calculations of sine and other functions that are both accurate and fast require very careful algorithms designed to avoid accidentally turning small errors into large ones. Often these algorithms are embedded in hardware for even faster performance. For example, almost every X86 chip shipped in the last 10 years has hardware implementations of sine and cosine that the X86 VM can just call, rather than calculating them far more slowly based on more primitive operations. HotSpot takes advantage of these instructions to speed up trigonometry operations dramatically.

Right triangles and Euclidean norms

Every high school geometry student learns the Pythagorean theorem: the square of the length of hypotenuse of a right triangle is equal to the sum of the squares of the lengths of the legs. That is, $c^2 = a^2 + b^2$

Those of us who stuck it out into college physics and higher math learned that this equation shows up a lot more than in just right triangles. For instance, it's also the square of the Euclidean norm on $\mathbf{R}^2$, the length of a two-dimensional vector, a part of the triangle inequality, and quite a bit more. (In fact, these are all really just different ways of looking at the same thing. The point is that Euclid's theorem is a lot more important than it initially looks.)

Java 5 added a `Math.hypot` function to perform exactly this calculation, and it's a good example of why a library is helpful. The naive approach would look something like this:

```
public static double hypot(double x, double y){
  return Math.sqrt (x*x + y*y);
}
```

The actual code is somewhat more complex, as shown in Listing 2. The first thing you'll note is that this is written in native C code for maximum performance. The second thing you should note is that it is going to great lengths to try to minimize any possible errors in this calculation. In fact, different algorithms are being chosen depending on the relative sizes of `x` and `y`.

Listing 2. The real code that implements `Math.hypot`

```
/*
 * ====================================================
 * Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
 *
 * Developed at SunSoft, a Sun Microsystems, Inc. business.
 * Permission to use, copy, modify, and distribute this
 * software is freely granted, provided that this notice
 * is preserved.
 * ====================================================
 */

#include "fdlibm.h"

#ifdef __STDC__
```

```
	double __ieee754_hypot(double x, double y)
#else
	double __ieee754_hypot(x,y)
	double x, y;
#endif
{
	double a=x,b=y,t1,t2,y1,y2,w;
	int j,k,ha,hb;

	ha = __HI(x)&0x7fffffff;	/* high word of  x */
	hb = __HI(y)&0x7fffffff;	/* high word of  y */
	if(hb > ha) {a=y;b=x;j=ha; ha=hb;hb=j;} else {a=x;b=y;}
	__HI(a) = ha;	/* a <- |a| */
	__HI(b) = hb;	/* b <- |b| */
	if((ha-hb)>0x3c00000) {return a+b;} /* x/y > 2**60 */
	k=0;
	if(ha > 0x5f300000) {	/* a>2**500 */
	   if(ha >= 0x7ff00000) {	/* Inf or NaN */
	       w = a+b;			/* for sNaN */
	       if(((ha&0xfffff)|__LO(a))==0) w = a;
	       if(((hb^0x7ff00000)|__LO(b))==0) w = b;
	       return w;
	   }
	   /* scale a and b by 2**-600 */
	   ha -= 0x25800000; hb -= 0x25800000;	k += 600;
	   __HI(a) = ha;
	   __HI(b) = hb;
	}
	if(hb < 0x20b00000) {	/* b < 2**-500 */
	    if(hb <= 0x000fffff) {	/* subnormal b or 0 */
	       if((hb|(__LO(b)))==0) return a;
	       t1=0;
	       __HI(t1) = 0x7fd00000;	/* t1=2^1022 */
	       b *= t1;
	       a *= t1;
	       k -= 1022;
	    } else {		/* scale a and b by 2^600 */
	        ha += 0x25800000;	/* a *= 2^600 */
	       hb += 0x25800000;	/* b *= 2^600 */
	       k -= 600;
	          __HI(a) = ha;
	          __HI(b) = hb;
	       }
	}
    /* medium size a and b */
	w = a-b;
	if (w>b) {
	    t1 = 0;
	    __HI(t1) = ha;
	    t2 = a-t1;
	    w  = sqrt(t1*t1-(b*(-b)-t2*(a+t1)));
	} else {
	    a  = a+a;
	    y1 = 0;
	    __HI(y1) = hb;
	    y2 = b - y1;
	    t1 = 0;
	    __HI(t1) = ha+0x00100000;
	    t2 = a - t1;
	    w  = sqrt(t1*y1-(w*(-w)-(t1*y2+t2*b)));
	}
	if(k!=0) {
	    t1 = 1.0;
	    __HI(t1) += (k<<20);
	    return t1*w;
	} else return w;
}
```

Actually, whether you end up in this particular function or one of a few other similar ones depends on details of the JVM on your platform. However, more likely than not this is the code that's invoked in Sun's standard JDK. (Other implementations of the JDK are free to improve on this if they can.)

This code (and most of the other native math code in Sun's Java Development Library) comes from the open source `fdlibm` library written at Sun about 15 or so years ago. This library is designed to implement the IEE754 floating point precisely and to have very accurate calculations, even at the cost of some performance.

Logarithms in base 10

A logarithm tells you what power a base number must be raised to in order to produce a given value. That is, it is the inverse of the `Math.pow()` function. Logs base 10 tend to appear in engineering applications. Logs base *e* (natural logarithms) appear in the calculation of compound interest, and numerous scientific and mathematical applications. Logs base 2 tend to show up in algorithm analysis.

The `Math` class has had a natural logarithm function since Java 1.0. That is, given an argument x, the natural logarithm returns the power to which *e* must be raised to give the value x. Sadly, the Java language's (and C's and Fortran's and Basic's) natural logarithm function is misnamed as `log()`. In every math textbook I've ever read, log is a base-10 logarithm, while ln is a base *e* logarithm and lg is a base-2 logarithm. It's too late to fix this now, but Java 5 did add a `log10()` function that takes the logarithm base 10 instead of base *e*.

Listing 3 is a simple program to print the log-base 2, 10, and *e* of the integers from 1 to 100:

Listing 3. Logarithms in various bases from 1 to 100

```
public class Logarithms {
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++) {
            System.out.println(i + "\t" +
                               Math.log10(i) + "\t" +
                               Math.log(i) + "\t"  +
                               lg(i));
        }
    }

    public static double lg(double x) {
        return Math.log(x)/Math.log(2.0);
    }
}
```

Here are the first 10 rows of the output:

```
1    0.0                                0.0                                0.0
2    0.3010299956639812  0.6931471805599453   1.0
3    0.47712125471966244 1.0986122886681096  1.584962500721156
4    0.6020599913279624  1.3862943611198906   2.0
5    0.6989700043360189  1.6094379124341003   2.321928094887362
6    0.7781512503836436  1.791759469228055     2.584962500721156
7    0.8450980400142568  1.9459101490553132   2.807354922057604
8    0.9030899869919435  2.0794415416798357   3.0
9    0.9542425094393249  2.1972245773362196   3.1699250014423126
10  1.0                              2.302585092994046     3.3219280948873626
```

`Math.log10()` has the usual caveats of logarithm functions: taking the log of 0 or any negative number returns NaN.

Cube roots

I can't say that I've ever needed to take a cube root in my life, and I'm one of those rare people who does use algebra and geometry on a daily basis, not to mention the occasional foray into calculus, differential equations, and even abstract algebra. Consequently, the usefulness of this next function escapes me. Nonetheless, should you find an unexpected need to take a cube root somewhere, you now can — as of Java 5 — with the `Math.cbrt()` method. Listing 4 demonstrates by taking the cube roots of the integers from -5 to 5:

Listing 4. Cube roots from -5 to 5

```
public class CubeRoots {
    public static void main(String[] args) {
        for (int i = -5; i <= 5; i++) {
            System.out.println(Math.cbrt(i));
        }
    }
}
```

Here's the output:

```
-1.709975946676697
-1.5874010519681996
-1.4422495703074083
-1.2599210498948732
-1.0
0.0
1.0
1.2599210498948732
1.4422495703074083
1.5874010519681996
1.709975946676697
```

As this output demonstrates, one nice feature of cube roots compared to square roots: Every real number has exactly one real cube root. This function only returns NaN when its argument is NaN.

The hyperbolic trigonometric functions

The hyperbolic trigonometric functions are to hyperbolae as the trigonometric functions are to circles. That is, imagine you plot these points on a Cartesian plane for all possible values of t:

```
x = r cos(t)
y = r sin(t)
```

You will have drawn a circle of radius *r*. By contrast, suppose you instead use sinh and cosh, like so:

```
x = r cosh(t)
y = r sinh(t)
```

You will have drawn a rectangular hyberbola whose point of closest approach to the origin is r.

Another way of thinking of it: Where sin(x) can be written as $(e^{ix} - e^{-ix})/2i$ and cos(x) can be written as $(e^{ix} + e^{-ix})/2$, sinh and cosh are what you get when you remove the imaginary unit from those formulas. That is, $\sinh(x) = (e^{x} - e^{-x})/2$ and $\cosh(x) = (e^{x} + e^{-x})/2$.

Java 5 adds all three: `Math.cosh()`, `Math.sinh()`, and `Math.tanh()`. The inverse hyperbolic trigonometric functions — acosh, asinh, and atanh — are not yet included.

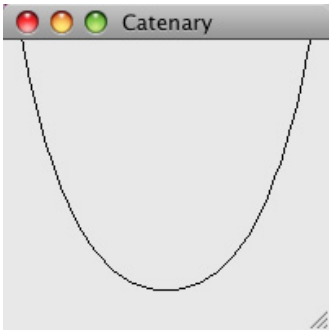
In nature, cosh(z) is the equation for the shape of a hanging rope connected at two ends, known as a *catenary*. Listing 5 is a simple program that draws a catenary using the `Math.cosh` function:

Listing 5. Drawing a catenary with `Math.cosh()`

```
import java.awt.*;

public class Catenary extends Frame {

    private static final int WIDTH = 200;
    private static final int HEIGHT = 200;
    private static final double MIN_X = -3.0;
    private static final double MAX_X = 3.0;
    private static final double MAX_Y = 8.0;

    private Polygon catenary = new Polygon();

    public Catenary(String title) {
        super(title);
        setSize(WIDTH, HEIGHT);
        for (double x = MIN_X; x <= MAX_X; x += 0.1) {
            double y = Math.cosh(x);
            int scaledX = (int) (x * WIDTH/(MAX_X - MIN_X) + WIDTH/2.0);
            int scaledY = (int) (y * HEIGHT/MAX_Y);
            // in computer graphics, y extends down rather than up as in
            // Caretesian coordinates' so we have to flip
            scaledY = HEIGHT - scaledY;
            catenary.addPoint(scaledX, scaledY);
        }
    }

    public static void main(String[] args) {
        Frame f = new Catenary("Catenary");
        f.setVisible(true);
    }

    public void paint(Graphics g) {
        g.drawPolygon(catenary);
    }

}
```

Figure 1 shows the drawn curve:

Figure 1. A catenary curve in the Cartesian plane



The sinh, cosh, and tanh functions also all appear in various calculations in special and general relativity.

Signedness

The `Math.signum` function converts positive numbers into 1.0, negative numbers into -1.0, and zeroes into zeroes. In essence, it extracts just the sign from a number. This can be useful when you're implementing the `Comparable` interface.

There's a `float` and a `double` version to maintain the type. The reason for this rather obvious function is to handle special cases of floating-point math, NaN, and positive and negative zero. NaN is also treated like zero, and positive and negative zero should return positive and negative zero. For example, suppose you were to implement this function naively as in Listing 6:

Listing 6. Buggy implementation of `Math.signum`

```
public static double signum(double x) {
  if (x == 0.0) return 0;
  else if (x < 0.0) return -1.0;
  else return 1.0;
}
```

First, this method would turn all negative zeroes into positive zeroes. (Yes, negative zeroes are a little weird, but they are a necessary part of the IEEE 754 specification.) Second, it would claim that NaN is positive. The actual implementation shown in Listing 7 is more sophisticated and careful for handling these weird corner cases:

Listing 7. The real, correct implementation of `Math.signum`

```
public static double signum(double d) {
    return (d == 0.0 || isNaN(d))?d:copySign(1.0, d);
}

public static double copySign(double magnitude, double sign) {
    return rawCopySign(magnitude, (isNaN(sign)?1.0d:sign));
}

public static double rawCopySign(double magnitude, double sign) {
    return Double.longBitsToDouble((Double.doubleToRawLongBits(sign) &
                                    (DoubleConsts.SIGN_BIT_MASK)) |
                                    (Double.doubleToRawLongBits(magnitude) &
                                    (DoubleConsts.EXP_BIT_MASK |
                                     DoubleConsts.SIGNIF_BIT_MASK)));
}
```

Do less, get more

The most efficient code is the code you never write. Don't do for yourself what experts have already done. Code that uses the `java.lang.Math` functions, new and old, will be faster, more efficient, and more accurate than anything you write yourself. Use it.