More Power (and fun!) in Java Numbers

Frédéric Bapst and Lucy Linder

Haute école d'ingénierie et d'architecture de Fribourg
Pérolles 80, CP 32, CH-1705 Fribourg (Switzerland)
`frederic.bapst@hefr.ch`

Abstract

*We present **COJAC** (Climbing Over Java Arithmetic Capabilities, `https://github.com/Cojac/Cojac`), a free tool that lets you boost at runtime the power of Java numbers (`float/double`)*

No source code modification, no recompilation needed

*At runtime, **COJAC** makes floating point numbers able to*

- *signal anomalies (cancellation, smearing...)*
- *apply interval computation or discrete stochastic arithmetic*
- *extend the precision to hundreds of significant digits*
- *convey automatic differentiation*
- *perform symbolic computation (ongoing work)*
- *imitate Chebfun (ongoing work)*

1. Standard `float` and `double` boosted at runtime

COJAC can reconcile two points of view

- programmers a priori use native number types (`float/double`)
- scientific computing has its specific needs and toolboxes

Simple to use — *Just tune a JVM option at runtime* to make standard numbers powerful (via *on-the-fly* instrumentation)

Example — Rump polynomial at a "bad" point, naive version

$$f(x,y) = \frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

```
public class Hello {
 static double pow(double base, int exp) {
    double r=1.0;
    while(exp-- > 0) r*=base;
    return r;
  }
  static double myPolyn(double x, double y) {
    return 1335.0*(pow(y, 6))/4.0
     + x*x*(11*x*x*y*y -pow(y, 6) -121*...
  }
  public static void main(String[] args) {
    double x=77617.0, y=33096.0;
    double r=myPolyn(x, y);
    System.out.println(r); // Gives a silly result:
  }                        -1.18E+21 instead of
}                          -0.827 !!!
```

2. Detection of floating point anomalies

To help debugging, **COJAC** emits **warnings** when

- a type casting results in precision or information loss
- an operation gives ∞ or NaN
- an addition causes *smearing*
- there is a strong *cancellation* effect
- an operation *underflows*
- two very close numbers are compared
- an *integer overflow* occurs

```
$ java -javaagent:cojac.jar Hello
COJAC: Smearing: DSUB      Hello.myPolyn(Hello.java:13)
COJAC: Cancellation: DADD Hello.myPolyn(Hello.java:12)
COJAC: Smearing: DADD      Hello.myPolyn(Hello.java:12)
-1.1805916207174113E21
```

3. Interval computation

With another option, you get *intervals* or *discrete stochastic arithmetic* (less pessimistic). **Now the uncertainty is quantified**

```
$ java -javaagent:cojac.jar="-Ri" Hello
COJAC: Interval detects instability [-3.5E+22;3.4E+22]
                Hello.myPolyn(Hello.java:12)
-1.1805916207174113E21
```

4. Higher precision

Maybe you want `BigDecimal` instead of `double`, to have **full control** over the underlying precision (here 40 significant digits)

```
$ java -javaagent:cojac.jar="-Rb 40" Hello
-0.8273960599468214                     Correct result!
```

5. Automatic differentiation

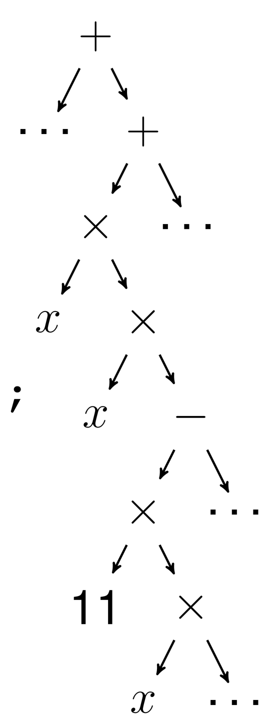
One additional line automatically brings the derivative in x. This is *automatic differentiation*, a technique completely different from both numerical and symbolic differentiation

```
  public static void main(String[] args) {
    double r, x=2.0, y=3.0;
    x=COJAC_MAGIC_asDerivationTarget(x);
    r=myPolyn(x, y);
    System.out.println("f (x,3): "+r);
    System.out.println("f' (x,3): "+COJAC_derivative(r));
  }

$ java -javaagent:cojac.jar="-Ra" Hello
f (x,3): 238845.58333333334
f' (x,3): -38954.0
```

6. Symbolic processing and Chebfun (ongoing work)

We are able to store the whole **symbolic expression tree** of a result, a first step towards symbolic processing (finding roots, computing integrals...). **Numbers become functions**

```
  public static void main(String[] args) {
    double x=COJAC_MAGIC_identity();
    double r=myPolyn(x, 3.0);
    System.out.println(COJAC_toString(r));
    System.out.println("r(2):"
                  +COJAC_evaluateAt(r, 2.0));
  }

$ java -javaagent:cojac.jar="-Ry" Hello
r(x)=ADD(...,ADD(MUL(x,...(SUB(MUL(11.0,MUL(x...
r(2): 238845.58333333334
```



We'll try to port Matlab's *Chebfun* and use Chebyshev interpolation to find roots efficiently *and* accurately

Conclusion — we need you!

COJAC gives super powers to `float/double`... Is it useful? Dear colloquium participants, **please give us feedback** ☺

`https://github.com/Cojac/Cojac`
`frederic.bapst@hefr.ch`

Colloque Numérique Suisse / Schweizer Numerik Kolloquium, Université de Fribourg, April 22, 2016