fast linear algebra for Java
http://jblas.org

Mikio L. Braun
mikio@cs.tu-berlin.de
TU Berlin, Franklinstr. 28/29, 10587 Berlin

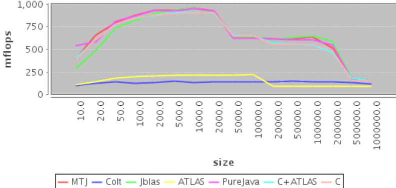
June 23, 2010

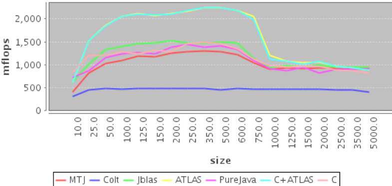
Overview

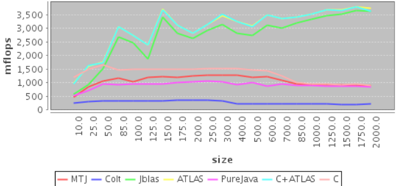
Main features:

- Matrix library for Java based on native BLAS and LAPACK.
- Support for single and double precision floats, real and complex matrices.
- Vectors and two-dimensional dense matrices only.
- For large matrix practically native performance (for example, 10 GFLOPS on a 2GHz Intel Core2Duo processor for matrix-matrix multiplication).
- Precompiled "fat jar" for Linux (Intel), Mac OS, and Windows.
- Parses FORTRAN code to automatically generate JNI stubs.

Benchmarks







Structure

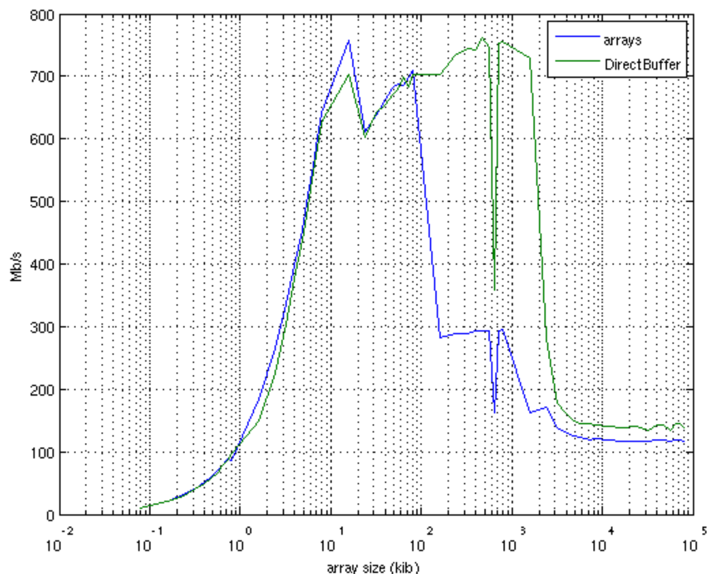
Main classes

- FloatMatrix, DoubleMatrix – real matrices.
- ComplexFloatMatrix, ComplexDoubleMatrix – complex matrices.

Computational routines as static methods

- Eigen – eigendecomposition
- Solve – solving linear equations
- Singular – singular value decomposition
- Decompose – decompositions like LU, Cholesky, ...
- Geometry – centering, normalizing, ...

Background on Native Calls in Java



Native calls are expensive:

- Arrays are always copied on native calls.
- For large arrays lots of cache misses.
- Doesn't make sense to call native code if operation is $O(1)$!

What about direct buffers?

- Direct buffers aren't garbage collected properly!
- Only makes sense to have a fixed number of direct buffers used for copying $\rightarrow$ directly use JNI with arrays.

Nevertheless, Java is as fast as C for simple things like vector addition!

Matrix creation

```
DoubleMatrix a = new DoubleMatrix(10, 5);
                        // 10 * 5 matrix
DoubleMatrix x = new DoubleMatrix(10);
                        // vector of length 10
DoubleMatrix y = DoubleMatrix.zeros(10, 5);
DoubleMatrix z = DoubleMatrix.ones(3, 4);
DoubleMatrix g = DoubleMatrix.randn(3, 4);
DoubleMatrix u = DoubleMatrix.rand(3);
```

- Dimensions: rows, columns, length
- Duplicating and copying: dup(), copy()
- Transposing: transpose()

Element access

```
DoubleMatrix a = new DoubleMatrix(10, 5);

// accessing elements by row and column
a.put(3, 2, 10.0);
a.get(2, 3);

// accessing element in linear order
a.get(15);
a.put(20, 1.0);

// for example, for implementing elementwise operations:
for (int i = 0; i < a.length; i++)
   a.put(i, a.get(i) * 3);
```

- Accessing rows and columns: putRow(), getRow(), ...

Pass a buffer object to prevent spurious object creation:

```
DoubleMatrix a = DoubleMatrix.randn(10, 100);
DoubleMatrix buf = new DoubleMatrix(10);

for (int i = 0; i < a.columns; i++) {
  a.getColumn(i, buf);
  // do something with buf
}
```

Simple Arithmetic

```
DoubleMatrix a = new DoubleMatrix(3, 3, 1, 2, 3,
                                        4, 5, 6,
                                        7, 8, 9);
DoubleMatrix x = new DoubleMatrix(3, 1, 10, 11, 12);

DoubleMatrix y = a.mmul(x);
DoubleMatrix z = x.add(y);
```

Supported Operations:

- Basic arithmetics: add, sub, mul, mmul, div, dot
- Element-wise logical operations: and, or, xor, not, lt, le, gt, ge, eq, ne
- Rows and vectors to a matrix: addRowVector, addColumnVector, ...

Machine Learning: Kernel Ridge Regression with Gaussian Kernel

Let's implement the following: Noisy sinc data set learned with Kernel Ridge Regression with a Gaussian Kernel.

Sinc Function

$$\mathrm{sinc}(x) = \begin{cases} \sin(x)/x & x \neq 0 \\ 1 & x = 0. \end{cases}$$

```
DoubleMatrix sinc(DoubleMatrix x) {
  return sin(x).div(x);
}
```

Not safe, what about $x = 0$?

```
DoubleMatrix safeSinc(DoubleMatrix x) {
  return sin(x).div(x.add(x.eq(0)));
}
```

Computing a sinc data set

$$X \sim \text{uniformly from -4, 4}$$
$$Y = \text{sinc}(x) + \sigma_\varepsilon^2 \varepsilon$$
$$\varepsilon \sim \mathcal{N}(0, 1).$$

```
DoubleMatrix[] sincDataset(int n, double noise) {
  DoubleMatrix X = rand(n).mul(8).sub(4);
  DoubleMatrix Y = sinc(X) .add( randn(n).mul(noise) );
  return new DoubleMatrix[] {X, Y};
}
```

Gaussian kernel

$$k(x, z) = \exp\left(-\frac{\|x - z\|^2}{w}\right)$$

```
DoubleMatrix gaussianKernel(double w,
                            DoubleMatrix X,
                            DoubleMatrix Z) {
  DoubleMatrix d =
    Geometry.pairwiseSquaredDistances(X.transpose(),
                                      Z.transpose());
  return exp(d.div(w).neg());
}
```

Kernel Ridge Regression

KRR learns a "normal" kernel model

$$f(x) = \sum_{i=1}^{n} k(x, X_i)\alpha_i$$

with

$$\alpha = (K + \lambda I)^{-1} Y,$$

where K is the kernel matrix

$$K_{ij} = k(X_i, X_j).$$

```
DoubleMatrix learnKRR(DoubleMatrix X, DoubleMatrix Y,
                    double w, double lambda) {
  int n = X.rows;
  DoubleMatrix K = gaussianKernel(w, X, X);
  K.addi(eye(n).muli(lambda));
  DoubleMatrix alpha = Solve.solveSymmetric(K, Y);
  return alpha;
}

DoubleMatrix predictKRR(DoubleMatrix XE, DoubleMatrix X,
                        double w, DoubleMatrix alpha) {
  DoubleMatrix K = gaussianKernel(w, XE, X);
  return K.mmul(alpha);
}
```

Computing the mean-squared error

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - Y_i')^2$$

```
double mse(DoubleMatrix Y1, DoubleMatrix Y2) {
  DoubleMatrix diff = Y1.sub(Y2);
  return pow(diff, 2).mean();
}
```

Conjugate Gradients

1. $r \leftarrow b - Ax$
2. $p \leftarrow r$
3. repeat
4. $\qquad \alpha \leftarrow \frac{r^T r}{p^T A p}$
5. $\qquad x \leftarrow x + \alpha p$
6. $\qquad r' \leftarrow r - \alpha A p$
7. $\qquad$ if r' is sufficiently small, exit loop
8. $\qquad \beta \leftarrow \frac{r'^T r'}{r^T r}$
9. $\qquad p \leftarrow r + \beta p$
10. $\qquad r \leftarrow r'$.
11. end repeat

```
DoubleMatrix cg(DoubleMatrix A, DoubleMatrix b,
                DoubleMatrix x, double thresh) {
  int n = x.length;
  DoubleMatrix r = b.sub(A.mmul(x)); // 1
  DoubleMatrix p = r.dup();          // 2
  double alpha = 0, beta = 0;
  DoubleMatrix r2 = zeros(n), Ap = zeros(n);
  while (true) {                     // 3
    A.mmuli(p, Ap);
    alpha = r.dot(r) / p.dot(Ap);  // 4
    x.addi(p.mul(alpha));          // 5
    r.subi(Ap.mul(alpha), r2);     // 6
    double error = r2.norm2();     // 7
    System.out.printf("Residual error = %f\n", error);
    if (error < thresh)
      break;
    beta = r2.dot(r2) / r.dot(r);  // 8
    r2.addi(p.mul(beta), p);       // 9
    DoubleMatrix temp = r;         // 10
    r = r2;
    r2 = temp;
  }
```

Outlook

- Better integration of different matrix classes.
- Better performance through caching.
- Support for sparse matrices and more compact storage schemes.