Efficient Java Matrix Library

Peter Abeles

April 5, 2016

Contents

Abstract

Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating dense matrices. Its design goals are; 1) to be as computationally efficient as possible for both small and large matrices, and 2) to be accessible to both novices and experts. These goals are accomplished by dynamically selecting the best algorithms to use at runtime and by designing a clean API. EJML is free, written in 100% Java and has been released under an LGPL license.

http://code.google.com/p/efficient-java-matrix-library/

1 IMPORANT

This document is horribly out of date but for some reason its a top result on google. To get the latest documentation go to:

http://code.google.com/p/efficient-java-matrix-library/

2 Introduction

Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating dense matrices. Its design goals are; 1) to be as computationally efficient as possible for both small and large matrices, and 2) to be accessible to both novices and experts. These goals are accomplished by dynamically selecting the best algorithms to use at runtime and by designing a clean API. EJML is free, written in 100% Java and has been released under an LGPL license.

EJML has three distinct ways to interact with it. This allows a programmer to choose between simplicity and efficiency. 1) A simplified interface that allows a more object oriented way of programming. 2) Letting EJML select the best algorithm to use. 3) Directly calling specialized algorithms. In general EJML is one the fastest single threaded pure Java library. See Java Matrix Benchmark for a detailed comparison of different libraries.

The following is provided:

- Basic operators (addition, multiplication, ...).
- Linear Solvers (batch and incremental).
- Decompositions (LU, QR, Cholesky, SVD, Eigenvalue).
- Matrix Features (rank, symmetric, definitiveness, ...).
- Random Matrices (covariance, orthogonal, symmetric, ..

In addition there are many specialized algorithms for specific matrix sizes and types.

3 Data Structures and Algorithms

DenseMatrix64F is the most basic data structure in EJML. It is a dense matrix composed of doubles. Internally the matrix is stored as a single array using a row-major format, see Figure 1. SimpleMatrix is a wrapper on top of DenseMatrix64F that provides a simplified interface.

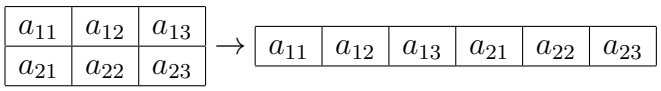

Figure 1: A matrix encoded in row-major format.

An operation, in this context, is just a mathematical function that takes at least one matrix in as an input. For example addition and subtraction are both operators. CommonOps and SpecializedOps are two classes which have several static functions that are operators for DenseMatrix64F. SimpleMatrix has all of its operators as part of its class for convenience. For a comparision of using operators with DenseMatrix64F and SimpleMatrix see Figure 2.

There are many operators in EJML. Figure 3 contains a list of some of the operators in this libraries. Figure 4 contains a list of operators that can be directly performed by SimpleMatrix. To make SimpleMatrix easier to program with, only a subset of the most essential operators are provided. However, by directly accessing the internal DenseMatrix64 in a SimpleMatrix, all the operators can be called on it.

Some of the operators do not specify which specific algorithm is used and some do. For example, many of the operators in CommonOps can call different algorithms depending on the structure of the matrix passed in. This allows the most efficient algorithm to be called. However, many the functions in MatrixMatrixMult are designed to be optimal for matrices of different sizes. Unless there is a very good reason not to, all calls should be made to the more generic operators.

One good reason not to use the generic operators is if you, the programmer, knows something they do not. For example, if a matrix is positive definite, then the generic inverse function is suboptimal. In that situation you should use CholeskyDecomposition directly because it is much faster.

Matrix multiplcation using operations.

```
DenseMatrix64F c =
  new DenseMatrix64F(a.numRows,b.numCols);
CommonOps.mult(a,b,c);
```

Matrix multiplcation using SimpleMatrix

```
SimpleMatrix c = a.times(b);
```

Figure 2: Comparision operations and SimpleMatrix.

4 Examples

In the examples directory three different implementations of a Kalman filter are provided along with a Levenberg-Marquardt optimization algorithm. These two algorithms are popular in various engineering disciplines.

The three different implementations of the Kalman filter are provided. Each one demonstrates different ways that EJML can be used. to show off the interfaces that one can use in EJML. In the following subsections the update function is shown and discussed.

4.1 KalmanFilterSimple.java

Figure 5 shows several concepts related to the simple interface. The wrap() function on the top demonstrates how a regular DenseMatrix64F can be changed into a SimpleMatrix with ease. Memory management is at a minimal since new matrices of the correct size are automatically created by each operation. The code is also noticeably easier to read.

4.2 KalmanFilterOps.java

The operation interface, shown in Figure 6, is noticeably more complex. It requires matrices to be predeclared. For this to work the programmer needs to figure out the exact shape of these intermediate matrices. What was on

one line now takes up multiple lines.

Memory management is more complicated in this example. In Figure 7 matrices that store the intermediate results are declared in the constructor. This requires more though from the programmer, but greatly reduces the amount of memory created and destroyed each processing cycle.

4.3 KalmanFilterAlg.java

Finally there is the algorithm interface, Figure 8. Here specific algorithms are called. The most important is the use of the Cholesky decomposition, which speeds up the matrix inverse. In other cases it takes advantage of it knowing before hand the size and shape of the matrix, which results in a slight performance gain.

4.4 Notes

What is the benefit of the added complexity of using the operator and algorithm interfaces over the simplistic interface? The operator interface runs about 23% faster than the simplistic interface and the algorithm interface runs about 28% faster than the simplistic interface. The exact performance differences will vary greatly depending on the application.

A Design Philosophy

This library was written in response to perceived weaknesses in other Java matrix libraries. The three biggest are, 1) unnecessarily slow performance in small matrices, 2) lack of flexibility in memory management, and 3) needing to choose between ease of use and performance.

One of the areas EJML performs very well is when dealing with small matrices. This is accomplished by having very low overhead and by using specialized algorithms for small matrices. The overhead is reduce by not having abstraction that allow native code or java code to be run, or generic algorithms that can work on a wide variety of matrix types. Specialized algorithms vary from switching the order the matrix is traversed depending on its size to having hand coded algorithms for specific matrix sizes. EJML also performs well with large matrices, where it still most often performs significantly better than other related libraries.

One way to write efficient Java algorithms is to minimize the amount of memory that is created and destroyed. This smooths out the processing time by not having the garbage collector needing to run as often and reduces the number of cycles spent reinitializing the data. All of the algorithms in EJML have been coded such that they predeclared all the data they use. The reshape function is also provided. While extremely easy to implement, it is often neglected in other libraries forcing you to declare a new matrix when the data changes.

Jama[1] is still a popular library despite no longer being actively developed. The primary reason for this is that it is easy to use. Matrix Toolkit Java (MTJ)[2] gives the developer more control, but is harder to use. EJML uses ideas from both libraries to create a simple yet robust interface. A user can program in EJML using the following interfaces; simple, operator, and algorithm.

Simple is a wrapper on top of the operator interface. It hides much of the memory management from the user and allows more readable code. The operator interface simplifies the selection of which algorithm to use from the user by automatically selecting what it thinks is the best one. The algorithm interface gives complete control to the programmer, but requires more skill and knowledge to user effectively.

A.1 Why multiple implementations?

Optimizing an algorithm for processing small and large matrices requires different strategies. An optimal small matrix algorithm typically minimizes the number of operations. Large matrix algorithms need to minimize the number of operations and cache misses.

On a modern desktop computer missing a CPU cache entails a large performance hit. A cache miss is when a computer needs to access memory that is not available on one of its high speed memory caches. This forces it to access the much slower main memory.

Small matrices can often be stored entirely in the CPU's cache, making it much less likely to have a cache miss. Large matrix algorithms are designed to avoid cache misses, often by processing the data in a less straight forward way that takes more operations. Typically running a small matrix algorithm

[1] http://math.nist.gov/javanumerics/jama/

[2] http://code.google.com/p/matrix-toolkits-java/

on a large matrix will result in a very large peformance hit, but the inverse only results in a relatively small performance hit.

Class	function	description
CommonOps	mult	$C = AB$ and $C = \alpha AB$ and $C = A^T B$
CommonOps	multTranA	$C = A^T B$ and $C = \alpha A^T B$
CommonOps	multTranAB	$C = A^T B^T$ and $C = \alpha A^T B^T$
CommonOps	multTranB	$C = AB^T$ and $C = \alpha AB^T$
CommonOps	multAdd	$C = C + AB$ and $C = C + \alpha AB$
CommonOps	multAddTranA	$C = C + A^T B$ and $C = C + \alpha A^T B$
CommonOps	multAddTranAB	$C = C + A^T B^T$ and $C = C + \alpha A^T B^T$
CommonOps	multAddTranB	$C = C + AB^T$ and $C = C + \alpha AB^T$
CommonOps	multElement	$c_{ij} = a_{ij} b_{ij}$
CommonOps	add	$C = A + B$, $C = \alpha A + \beta B$
CommonOps	addEquals	$A = A + B$, $A = A + \beta B$
CommonOps	sub	$C = A - B$
CommonOps	subEquals	$A = A - B$
CommonOps	invert	Matrix inverse A^{-1}
CommonOps	solve	Solves for $X = A^{-1} B$
CommonOps	normF	Matrix norm
CommonOps	scale	$a_{ij} = \alpha a_{ij}$
CommonOps	transpose	$a_{ij} = a_{ji}$
CommonOps	trace	$\sum_i a_{ii}$
SpecializedOps	identity	Creates an identity matrix
SpecializedOps	diag	Creates a diagonal matrix
SpecializedOps	submatrix	Creates a submatrix of the original
SpecializedOps	diffNormF	$\sqrt{\sum_{ij}(a_{ij} - b_{ij})^2}$
SpecializedOps	diffNormP1	$\sum_{ij} \lvert a_{ij} - b_{ij} \rvert$
SpecializedOps	copyChangeRow	Swaps the rows of the original matrix
MatrixFeatures	hasNaN	Does the matrix contain a NaN
MatrixFeatures	hasUncountable	Does the matrix contain a NaN or Infinity
MatrixFeatures	isVector	Is the matrix a vector?
MatrixFeatures	isPositiveDefinite	Is the matrix positive definite?
MatrixFeatures	isPositiveSemidefinite	Is the matrix positive semidefinite?
MatrixFeatures	isSquare	Is the matrix square?
MatrixFeatures	isSymmetric	Is the matrix symmetric?
MatrixFeatures	isSimilar	$\lvert a_{ij} - b_{ij} \rvert \leq \sigma \; \forall \; i, j$
CovarianceOps	isValidFast	Quick covariance validity check.
CovarianceOps	isValid	Rigerous covariance validity check.
CovarianceOps	invert	Faster covariance matrix inversion.
CovarianceOps	randomVector	Draws a random vector from the covariance.

Figure 3: Some of the matrix operators included in EJML

function	description
wrap	Allows a DenseMatrix64F to be manipulated as a SimpleMatrix.
identity	Creates an identity matrix.
random	Creates a matrix with random elements.
diag	Creates a diagonal matrix.
getMatrix	Returns the internal DenseMatrix64F.
transpose	Returns the transpose of this matrix.
mult(B)	Returns AB
plus(B)	Returns $A + B$
minus(B)	Returns $A - B$
plus(β,B)	Returns $A + \beta B$
scale(α)	Returns αA
invert()	Returns A^{-1}
solve(B)	Returns $X = A^{-1}B$
set(val)	$a_{ij} = val$
zero	$a_{ij} = 0$
norm	Returns the matrix norm.
determinant	Returns the determinant.
trace	Returns the matrix's trace.
reshape	Changes the matrix's shape with out declaring new data.
computeSVD	Singular Value Decomposition
set(i,j,val)	$a_{ij} = val$
get(i,j)	Returns a_{ij}
numRows()	Returns number of rows.
numCols()	Returns number of columns.

Figure 4: Functions provided by SimpleMatrix.

```
public void update(DenseMatrix64F _z, DenseMatrix64F _R) {
   // a fast way to make the matrices usable by SimpleMatrix
   SimpleMatrix z = SimpleMatrix.wrap(_z);
   SimpleMatrix R = SimpleMatrix.wrap(_R);

   // y = z - H x
   SimpleMatrix y = z.minus(H.mult(x));

   // S = H P H' + R
   SimpleMatrix S = H.mult(P).mult(H.transpose()).plus(R);

   // K = PH'S^(-1)
   SimpleMatrix K = P.mult(H.transpose().mult(S.invert()));

   // x = x + Ky
   x = x.plus(K.mult(y));

   // P = (I-kH)P = P - KHP
   P = P.minus(K.mult(H).mult(P));
}
```

Figure 5: Code from KalmanFilterSimple.java

```
public void update(DenseMatrix64F z, DenseMatrix64F R) {
   // y = z - H x
   mult(H,x,y);
   sub(z,y,y);

   // S = H P H' + R
   mult(H,P,c);
   multTransB(c,H,S);
   addEquals(S,R);

   // K = PH'S^(-1)
   if( !invert(S,S_inv) ) throw new RuntimeException("Invert failed");
   multTransA(H,S_inv,d);
   mult(P,d,K);

   // x = x + Ky
   mult(K,y,a);
   addEquals(x,a);

   // P = (I-kH)P = P - (KH)P = P-K(HP)
   mult(H,P,c);
   mult(K,c,b);
   subEquals(P,b);
}
```

Figure 6: Code from KalmanFilterOps.java

```
a = new DenseMatrix64F(dimenX,1);
b = new DenseMatrix64F(dimenX,dimenX);
y = new DenseMatrix64F(dimenZ,1);
S = new DenseMatrix64F(dimenZ,dimenZ);
S_inv = new DenseMatrix64F(dimenZ,dimenZ);
c = new DenseMatrix64F(dimenZ,dimenX);
d = new DenseMatrix64F(dimenX,dimenZ);
K = new DenseMatrix64F(dimenX,dimenZ);
```

Figure 7: When directly working with the operation functions, matrices that contain the intermediate results need to be declared.

```
public void update(DenseMatrix64F z, DenseMatrix64F R) {
   // y = z - H x
   MatrixVectorMult.mult(H,x,y);
   sub(z,y,y);

   // S = H P H' + R
   MatrixMatrixMult.mult_small(H,P,c);
   MatrixMatrixMult.multTransB(c,H,S);
   addEquals(S,R);

   // K = PH'S^(-1)
   if( !chol.decompose(S) ) throw new RuntimeException("Invert failed");
   chol.setToInverse(S_inv);
   MatrixMatrixMult.multTransA_small(H,S_inv,d);
   MatrixMatrixMult.mult_small(P,d,K);

   // x = x + Ky
   MatrixVectorMult.mult(K,y,a);
   addEquals(x,a);

   // P = (I-kH)P = P - (KH)P = P-K(HP)
   MatrixMatrixMult.mult_small(H,P,c);
   MatrixMatrixMult.mult_small(K,c,b);
   subEquals(P,b);
}
```

Figure 8: Code from KalmanFilterAlg.java