Numerical Linear Algebra : Design and Implementation

An Object-oriented Approach

Rick Cui

Wonderful World

CONTENTS

CHAPTER 1

MATRIX DESIGN

Contents

TO DO LIST

amsfont are better.

java code needs a better format: font and layout position.

There are already various packages available in Java for linear algebra, but most of them are dedicated to algorithms, such as solving linear equation systems. The main purpose of this package is to design matrices and their related behaviors at the proper abstract level from a software engineering point of view. Mathematical programming is harder than general software programming because of the heavy optimization embedded in the code. Designing matrix related classes are among the worst because matrix computing involves both performance and memory management. However, in the same spirit in mathematics where we always divide bigger problems into smaller problems, we hope we achieve a better design by utilizing some software design principles without sacrificing performance. Our goal is to design a reusable, flexible and easy to extend and test package.

1.1 Scope

This package has the following scope:

- All matrices are two dimensional.

- All matrix cells are of type double.
- Banded/Block matrices are not implemented.
- Any parallel or divide-and-conquer based algorithms are not in the scope.

There are several reasons we limit ourselves to this scope:

- Though Java has a root class, Number, for all numeric classes, it's not operable arithmetically, such as $a + b$ where a and b are of type Number. Therefore, even we can extend Number class to Rational, Complex, or Quaternion, we really can't operate on them. The consequence is that we can't define a generic matrix interface which has getCell() and setCell() methods depending on Number class. If we have to extend Number, we would have to extend arithmetic operations through interfaces, such as addition and subtraction. This would make the code hard to read and maintain.
- The Java array is continuous in the memory. But arrays of array is not. This has a big performance hit, even in multiplications of matrices of type double. If we have array of arrays of Objects, the performance is getting worse.
- Java doesn't support operator overloading and so even if we extend from the Number class to a Complex, we can't utilize the same code for doubles on Complex, such as arithmetic operations. Furthermore, we can't overload matrix operations either, otherwise we could easily extend the cell-based matrices to block-based matrices.
- One way to expand to higher dimensional matrices, such as double[][][], one way is to define a new class to encapsulate the index various, e.g., a MatrixIndex class, then map the objects of this class to the matrix data storage. But we need to create a lot of objects of this class, one for each matrix cell. If the matrix is fairly large, the performance is hardly satisfied.
- Because Java can't be extended to create new types of number natively, like in C++, and operators can be overloaded like in C++, creating new types of numbers, like Rational, Complex, or Quaternion will have performance hit. If we create MatrixValue class to handle different types of values(with subclasses like DoubleMatrixValue, ComplexMatrixValue), then the math operations(plus, minus, multiply, and division) will create a lot of small objects. To make things worse, MatrixValue[] array copy will be slow too. So another unknown is how to map MatrixValue[] to double[] arrays.

Another problem is when a real matrix times a complex number, then the storage has to be expanded. The general interface is matrix<N extends Number>. The design of the implementation should satisfy the following:

- easy to add new types of matrices with minimal effort. For example,
- easy to add new matrix operations.

- easy to change implementations of matrix operations.

In addition, it should be easy to test each class and every line of code. We should avoid duplicate code, reuse whenever possible. For example, a permutation matrix cares only multiplication, so its addition should be the default. However, it should be easy to overwrite this method if needed.

A look back on design after the implementation is always a good health check. Sometimes we need a refactoring at the design level - design refactoring, beside the code refactoring.

1.2 Fundamental Classes

The fundamental classes/interfaces are:

```
public interface NumericMatrix implements Serializable;
public class NumericVector implements Serializable;
public class NumericMatrixException extends RuntimeException;
```

The interface NumericMatrix is an interface and thus it gives us the freedom to implement various types of matrices with different memory storage. Vector is just a class since there is no need for different types of storage. The last class is a subclass of RuntimeException. This class is the root class of all exceptions in this package, it provides a convenience for users to catch all runtime exceptions if needed, rather than catching each particular exception in a series of tedious try-catch blocks, unless users need to do so. The runtime exception is chosen because normally there is no way to recover from these value related errors but keep throwing them up in the chain.

1.3 Matrix Interface

There are so many matrix operations, and probably more will be added in the future. A proper abstraction and design of the matrix behaviors are well deserved. We will present the interface first and then explain the rationales behind it. Here is the list of behaviors:

```
public interface NumericMatrix extends Serializable
{
    int getNumberOfRows();
    int getNumberOfColumns();

    NumericMatrix transpose();

    double getCell(int row, int column);
    void setCell(int row, int column, double value);

    // return the sum of all diagonal cells.
    double trace();

    // return the cell-wise sum.
    NumericMatrix add(NumericMatrix numericMatrix);

    // return the cell-wise difference, this - matrix.
    NumericMatrix sub(NumericMatrix numericMatrix);

    // return the cell-wise difference, matrix - this.
    NumericMatrix leftSub(NumericMatrix numericMatrix);

    // return the product, this * matrix.
    NumericMatrix multi(NumericMatrix numericMatrix);

    // return the product, matrix * this.
    NumericMatrix leftMulti(NumericMatrix numericMatrix);

    // return the product, this * vector.
    NumericVector multi(NumericVector numericVector);

    // return the product, vector^T * this.
    NumericVector leftMulti(NumericVector numericVector);

    // return the product, this * scaler.
    NumericMatrix multi(double scaler);

    NumericMatrix newCopy();
    void accept(NumericMatrixVisitor numericMatrixVisitor);
}
```

These behaviors are divided into two major groups, data related and arithmetic. Note that these methods have no dependency on how the data(the cells of matrices) is stored.

The first five methods are data related. Note that the setter and getter for cells depend on the type double. If we want to extend this interface to a more generic operable numeric class, then we have to change these two methods. The setter method also indicates that matrix implementations are mutable. Making it immutable like the *String* class would have big impact in memory on practical usage. However, we will try our best to preserve the immunity because it does provide benefits on making the code healthy and solid. Part of this effort is to eliminate the methods to get a row, or a column, or a portion of them, or a sub-matrix, because either we have to expose the internal memory reference or duplicate the data. Exposing the internal memory reference would make tracing value changes very hard. Duplicating data would restrict the applicable usage of this package.

Though the method conjugate() is close to transpose() we don't include it because it is a special method for complex matrices, not apply to other types of matrices.

Other properties of matrices are omitted too, such as whether a matrix is a square matrix, an orthogonal matrix, or a symmetric matrix. These properties are left out to separate classes.

The next nine methods are arithmetic operations. Besides the normal operations, there are a few left- operations, leftSub and leftMulti. At first thought, if matrix1 and matrix2 are two matrices, then

```
matrix1.multi(matrix2)
```

has the effect to multiply matrix2 by matrix1 from the left and thus there is no need to have these extra left- operations. However, if we think about matrices as both transformations and objects being transformed, it would be beneficial to the users to have these methods. Let's say we have a huge matrix matrix1 as the object being transformed, we want to apply a series of transforms T_i onto it. We could do it like this:

```
matrix1.multi(T_1).leftMulti(T_2) ...
```

This means matrix1 is changed. The rule we enforce here is the input parameters T_i are not changed by these methods, only the invoking objects could be changed.

Note that the methods

```
multi(double scaler)
double trace()
```

depends on the type of double, combining with the get() and set() methods, totally we have 4 methods depending on the type double. If in the future we want to expand this package to to other types of matrices, we need to change these these method signatures.

The BLAS related methods are not here either, such as $\alpha AB + C$.

There are a lot of behaviors we intentionally leave out of the above interfaces. There are different reasons to leave them out and we propose a different way to handle them.

- matrix norms: Though each of the norms, 1-norm, 2-norm, infinity-norm, etc, is trivial, but these norm variances should be left out of the interface and dealt with separately.
- matrix inverse: not all matrices have inverse(e.g., a rectangle matrix). Even if there is an inverse, the implementation is not as trivial as the above methods. Plus there is a pseudo inverse consideration as well.
- determinant: is only for square matrices and its computation is not as trivial as the above methods.
- solving linear equation system, again this is more complex and there are several concerns, such as pivoting(no/partial/full), under/over constraint. Besides, this is more like a way to use matrix, not an internal behavior of a matrix.
- matrix decomposition: There are many different types of decompositions, the most commonly used are LU, QR, SVD, and eigen decompositions. Others are polar, etc. We may have more in the future. Squeezing all of them in the above interface is not a good practice to follow OOP.

We should have a better way to deal with the above features of matrices in a uniform manner. Furthermore, the design should make it easy to add more features and compose different features together. For example, when we estimate the condition number of a matrix using the formula

$$cond(A) = \frac{1}{|A|_p |A^{-1}|_p}$$

we should be able to specify the number p = 1, 2, or infinity.

The last two methods have special purposes. The newCopy() method is to create a new copy of the invoking object without knowing its concrete type. In contrast, if we know the type of a matrix, we could directly invoke its constructors. The accept() method is for the visitor design pattern. This method is the center of the purpose of this package. Using a visitor pattern, also called double dispatch, we could delegate each different case to a different class, rather than having a deeply nested if-else. Knowing that there are some side effects stated in GoF, the benefit is far more outweighed.

CHAPTER 2

MATRIX IMPLEMENTATION

Contents

Once we have the matrix interface ready, we could think about the implementation. Keeping in mind, matrices could be very large and thus we have to make all effort to preserve the memory. Another twist is that there are so many types of matrices, such as diagonal matrix, tridiagonal matrix, symmetric matrix, to name a few.

2.1 Matrix Storage

General matrices could take a lot of memory, e.g., a single 7000 ×7000 matrix could take ~390MB. In order to save memory, the following special matrices are implemented, in addition to the general purpose matrix. Here, we follow the java array index convention, i.e., index runs from 0 to length - 1.

- **DiagonalMatrix**: all off diagonal cells are zeros, i.e.,

  $$A(i,j) = 0, \ \ if \ \ i \neq j$$

  So the storage is just a double[].

- **AntiDiagonalMatrix**: all off secondary diagonal cells are zeros, i.e.

  $$A(i,j) = 0, \ \ if \ \ i \neq n-1-j$$

  So the storage is a double[].

- **ToeplitzMatrix**: each descend diagonal has the same value, i.e.,

  $$A(i,j) = A(i-1, j-1)$$

  So we only need to store the secondary diagonal cells and thus it requires the same storage as AntiDiagonalMatrix, i.e., a double[].

- **HankelMatrix**: each skew diagonal has the same value, i.e.,

  $$A(i,j) = A(i-1, j+1)$$

  So we need a double[] to storage the different values.

- **BidiagonalMatrix**: there are two version, UpperBidiagonalMatrix and LowerBidiagonalMatrix. For the UpperBidiagonalMatrix, all off diagonal and off super-diagonal cells are zeros, i.e.,

  $$A(i,j) = 0, \ \ if i \neq j \ \ and i \neq j+1$$

  For the LowerBidiagonalMatrix, all off diagonal and off sub-diagonal cells are zeros, i.e.,

  $$A(i,j) = 0, \ \ if i \neq j \ \ and \ \ i \neq j-1$$

  So the storage is two $double[$.

- **TridiagonalMatrix**: all off diagonal, off super-diagonal, and off sub-diagonal cells are zeros, i.e.,

  $$A(i,j) = 0 \ \ if i \neq j \ \ and \ \ i \neq j+1 \ \ and \ \ i \neq j-1$$

  So the storage of this class is 3 double[].

- **TriangularMatrix**: there are two versions, UpperTriangularMatrix and LowerTriangularMatrix. For the UpperTriangularMatrix, all cells below diagonal are zeros, i.e.,

  $$A(i,j) = 0 \ \ if \ \ i > j$$

  For the LowerTriangularMatrix, all cells above diagonal are zeros, i.e.,

  $$A(i,j) = 0 \ \ if \ \ i < j$$

  So the storage for these classes are $double[][]$ with each $double[i]$ having length i, so the total size of the storage is about the half of the matrix size (the size is half of the matrix size minus the diagonal size).

- **SymmetricMatrix**: symmetric about the diagonal,

$$A(i,j) = A(j,i)$$

  So we need the same storage as triangular matrices.

- **AntiSymmetricMatrix**: since it satisfies

$$A(i,j) = -A(j,i)$$

  the diagonal cells are all zero. So we need to store only the strictly upper half of the matrix.

- **GeneralNumericMatrix**: we have to store the entire matrix in this case.

- **SparseMatrix**: nonzero cells are sparse.

The guideline is to save at least a half of the storage(minus one diagonal).

In java, there are two storage types to store a matrix, double[][] and double[]. There are also two ways to store a matrix using these types, row majored or column majored. There is a performance impact on accessing the cells in a row or a column.

There are several ways to store a matrix, using a double[][] with row or column major, or using a double[] with row or column major. Choosing a row or column major also depends on the dimensions of the given matrix. For example, if row numbers > column numbers, then we could choose column majored.

Sparse matrices have a different way to store data.

Other issues are views, slice and dice matrices.

Matrix readers and writers

RealArray and ComplexArray

RealMatrix and ComplexMatrix.

ComplexMatrix { double[][] real; double[][] imag; } Sometimes, we have conjuates(in the real field). a pair of conjugates uses only 2 matrices like a +- bi, but two complex general matrix use 4 matrices.

nice printing(fixed number of digits)

quadratic form

2.2 Matrix Constructors

There are several constructors in each of the matrix classes. The first one is taking the size parameters (row size and column size) so that users can use getters and setters to retrieve and set matrix values. The second one is taking arrays, which will be deeply cloned. The third one is a copy constructor. The forth one is the general form: NumericMatrix newCopy(). The purpose of this is that we could create a new copy of the invoking matrix without knowing its concrete class. So this method is added to the NumericMatrix interface. Given each implementation, the data related methods can be implemented straight forward.

The key consideration here is the setCell(), because it makes matrices mutable. The conscious and balanced decision is to keep this so that we could reuse some of the matrix objects without creating a lot of large matrices. Another reason is that without it, we have to rely on the constructors to pass in matrix data and thus we have to clone a lot of data if we want to enforce data encapsulation. Though most of the time there is a constructor that takes data structure, such as double[] or double[][], users in general should use the constructor that takes size(s) because of the deep cloning. The main purpose of the constructors taking arrays is for unit testing. Keep in mind, matrices are mutable.

How to create a matrix from vectors.

2.3 Arithmetic Operations

Since there are different types of matrices, we try to preserve the types under these operations whenever possible. The visitor pattern naturally breaks down the different cases. Since all operations have no dependency on other components, these visitors can be stateful. In constrast, in the next chapter we have to utilize a stateless visitor because it could have dependencies which could have changed during runtime. Using a stateful visitor is not thread safe, but the visitor interface stays the same, i.e., visit(Visitable v). Using a stateless visitor has to change the interface whenever the returning result is different, e.g., matrix additions return a matrix, but determinant calculation return a real number(assuming all cells are real).

In the above session, in order to save memory space, we choose to implement several matrix classes for different data storage. These different implementations introduce several complexities when we operate them:

- When we add a diagonal matrix to a tridiagonal matrix, the result is still a tridiagonal matrix. But when we add a upper triangular matrix to a lower triangular matrix, it's likely we get a general, full size matrix. This means the add() method has to return a new object of a different class.

- If we enforce that all math operations to return a new object, then we run out of memory pretty quickly when matrix has large sizes. So the compromised way is to return new objects sometimes and return the old object other times. For all special matrices, since we already save the storage by implementing them, we could return a new object. But for the general matrix, since the storage could be huge, we choose to return itself. When using the interface NumericMatrix, sometimes we could get lost to track different matrix classes(and we shouldn't do this at all since it defeats the purpose of the interface), so the rule of thumb is that the invoking object of the math operations could be changed after the operations. If users have to keep the invoking object, call newCopy() first to get a fresh object. The passing parameter of math operations are never modified, because we don't want to implicitly modify the parameter, so the so-called "side effect".

- Because of returning itself in the case of a general matrix for math operations, it introduce another complexity when we multiply a general matrix by a special matrix from left, such as d.multiply(g), where d is a diagonal matrix and g is a general matrix. By the rule, this will return a new object, though g.multiply(d) will return g itself after modification. So we have to introduce two more methods, leftSubstract() and leftMultiplication(). Basically, we treat general matrix as an object that can be transformed by other matrices, which are treated as transformers.

- Sometimes, we want to precise results, such as a diagonal matrix multiply a diagonal matrix should be also a diagonal matrix, rather than a general matrix. With the combinations of different types of invoking objects and parameters, a naive implementation will end up a lot of embedded if-else blocks. A better approach is to use visitor design patterns. So we add the visitor pattern to the matrix interface. All complex math operations are using this pattern.

return new result object

general matrix return same object and export

Of all the combinations, we need to implement only half of the operations.

We want to abstract the math operations from the data storage, such as C = A + B, we want to construct the result object first (maybe in different ways, a new object, or the invoking object), and then fill the result. Adding two different types/classes of matrices should be in one place for each combination.

multiplication taks $O(n^3)$ operations, very expensive.

2.4 Class consideration

The matrix classes are designed to be value objects, i.e., as long as all cell values are equal, they are considered to be equal. Our classes of matrices are value objects. There are several considerations due to java language: equals() method: Because of the choice of implementations, when check whether two matrices are equal, we should check the cell values, not the classes. For example, the identity matrix can be created with many classes. In the same class, we need to maintain the consistency of the equals() method and hashCode() method, but for different classes, we can safely ignore this.

toString()

visitor

export general matrix internal double[][] to hook into 3rd party libs.

Do we need valueEquals() so that we separate equals() for the systemwide.

Root exceptions.

2.5 Transformation Matrices

We also implement several special matrices, transformation matrices.

2.5.1 Identity Matrix

pass in a dimension.

2.5.2 Permutation Matrix

There are two special cases, exchange matrix and

2.5.3 Elementary Modifier Matrix

There are actually two different operations, scale a row/column, or modify a row/-column with another. There are 3 types: type 1, 2, 3. Type 3 modify another row/-column by multipling a row and then adding to that row. the determinant is 1 but the norm is not 1 in general.

2.5.4 Projection Matrix

Projection

2.5.5 Rotation Matrix

Givens Rotation and Jacobi Rotation.

We want to distinguish the rotation behavior and the creation of rotations.

General rotation is $RR^T = 1$ and $det(R) = 1$.

2.5.6 Householder Reflection Matrix

Reflection

CHAPTER 3

MORE MATRIX OPERATIONS

Contents

Now we are ready to consider these operations. We provide our own implementation and Jlapack hookup. There is no point to reinvent the wheel if there is a high quality package available. Better algorithms are added over time, such as divide-and-conquer, parallel.

Template methods for iterative methods.

Estimate run time $O(n^3)$

3.1 Matrix Norms

There are many norms

3.2 Matrix Decompositions

We consider only the commonly used decompositions.

3.2.1 LU Decomposition

Pivoting: no partial or full.

Full LU decomposition is numerical stable for well conditioned matrices.

Lapack SGETRF

3.2.2 Cholesky Decomposition

If a matrix is symmetric, then the LU decomposition can be simplified

$$A = L * D * L^T$$

where L is an unit lower triangular matrix and D is diagonal. If A is positive definite, then we can absorb D into L by taking the square root of the cells of D,

$$A = L * L^T$$

Numerically, Cholesky decomposition takes a shortcut using the above expression and thus save time. If A is not positive definite, in theory we still have the first expression, but due to pivoting we can't use it numerically. The best shot is to make D is block diagonal with 2 X 2 blocks.

3.2.3 QR Decomposition

Using householder transformation.

3.2.4 Singular Value Decomposition(SVD)

Do we care symmetric?

Polar decomposition of a matrix A is A = PQ, where P is symmetric positive semidefinite and Q is orthogonal. We can derive this from the SVD.

3.2.5 Eigen Decomposition

This part has several applications:

- find eigenvalues only
- find eigenvalues and eigenvectors
- find eigenvalues, eigenvectors, and the transformation matrix.

If the matrix is symmetric, we could use householder matrix to transform it to a tridiagonal matrix, then use QL + implicit shift.

If the matrix is nonsymmetric, use householder to transform it to a Hessenberg form, then transform it to a Schur form.

3.3 Determinant

We could use LU decomposition to compute Determinant for general matrices.

3.4 Linear Equation Systems

Use LU decomposition to solve. pivoting.

general solver()

–> LU

–> cholesky

–> Iterative

Another method is iterative method. Here are the two packages:

SPARSKIT: http://www-users.cs.umn.edu/ saad/software/

Templates: http://www.netlib.org/templates/

3.5 Matrix Inverse and Condition Number

The inverse of a matrix can be deducted from the linear solver. The condition number is defined as

$$\kappa = \frac{1}{|A||A^{-1}}|$$

condition number estimate.

3.6 Testing

test different compositions.

3.6.1 Correctness and Error Bound

3.6.2 Algorithm Stability

3.6.3 Performance

In general, there are two places where we can optimize:

* 1. time spent to access elements

* 2. time spent to calculate.

Java array bound checking.

Test the difference between:

Matrix and double[][]

and

Vector and double[]
for getter/setter.

3.7 Reference

Java implementations:

- JLapack: http://www.netlib.org/lapack/
- Colt implementation:
- Jsci implementation:
- ojAlgo
- Jama
- Jampack
- UJMP: http://www.ujmp.org/

Other implementations:

- Aglib: for C++, .net
- newmat: a good C++ implementation
- BLAS: http://www.netlib.org/blas/
- BOOST: http://www.boost.org/
- PLapack C implementation: http://www.cs.utexas.edu/ plapack/

BLAS

3.8 Bibliography

[1] Carpenter, R.H.S., *M*ovements of the Eyes, 2nd Edition, Pion Publishing, 1988.

[2] Franklin, G.F., Powel, J.D., Workman, M.L., *D*igital Control of Dynamic Systems, Second Edition, Addison-Wesley, 1990.

[3] Oh, P.Y., Allen, P.K., “Design a Partitioned Visual Feedback Controller,” *I*EEE Int Conf Robotics & Automation, Leuven, Belgium, pp. 1360-1365 5/98

[4] Oh, P.Y., Allen, P.K., “Visual Servoing by Partitioning Degrees of Freedom,” *I*EEE Trans on Robotics & Automation, V17, N1, pp. 1-17, 2/01

CHAPTER 4

MATRIX ALGORITHMS

Contents

Here are some of the explanation of the algorithms used in the package, at the minimal level. For details, check the references.

4.1 LU Decomposition

LU decomposition

4.2 Cholesky Decomposition

Cholesky Decomposition

4.3 Geometric Transformations

Here are the geometric transformations rotation has determinant 1 and norm 1.
reflection has determinant -1 and norm 1.
projection has determinant != 1 or -1, the norm is not 1.

4.3.1 Projection

We are going to derive the matrix representation of the projection of a vector on another vector. Given two vectors u and v, the projection of u on v is

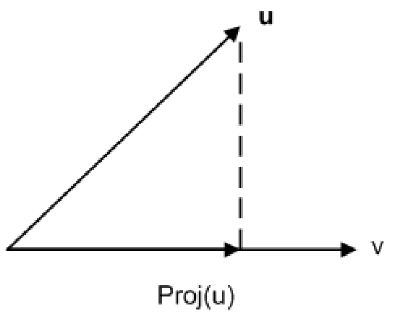

Figure 4.1: Projections of u on v

$$proj(u) = |u|\cos\theta \cdot \frac{v}{|v|}$$

Recall that the inner product of u and v is

$$vu = v^T u = |v||u|\cos\theta$$

by eliminating $\cos\theta$, the projection can be expressed as

$$proj(u) = |u|\frac{v^T u}{|v||u|}\frac{v}{|v|} = \frac{v^T u}{|v|}^2 v = \frac{v^T u}{v^T v} v$$

The first factor is really a scaler, so we could rewrite it

$$proj(u) = v\frac{v^T u}{v^T v} = \frac{vv^T}{v^T v} u$$

So the project matrix on v is

$$P_v = \frac{vv^T}{v^T v} \tag{4.1}$$

where the denominator is just a scaler. Several properties of P

- P is symmetric, $P^T = P$.

- P is idempotent, $P^2 = P$.

In numerical practice, when we deal with multiplications we don't need to store the entire P, instead we can store only the vector v. For an arbitrary vector x,

$$Px = \frac{vv^T}{v^T v} x = v \frac{v^T x}{v^T v} = \frac{v^T x}{v^T v} v$$

The left factor of the above is purely a number. Similarly

$$x^T P = x^T \frac{vv^T}{v^T v} = \frac{x^T v}{v^T v} v^T$$

Again, the left factor is a scaler. If A is a matrix, then

$$PA = \frac{vv^T}{v^T v} A = \frac{v(v^T A)}{v^T v}$$

and

$$AP = A\frac{vv^T}{v^T v} = \frac{(Av)v^T}{v^T v}$$

These formulae simplify the calculations when transforming other matrices.

Interestingly, we can extend the above result to higher dimensions. For a vector u, we want to project it to a subspace.

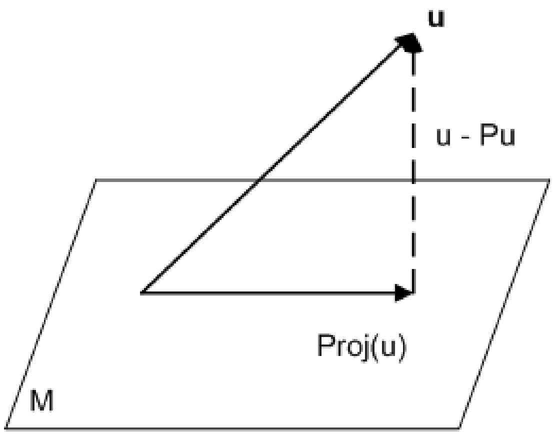

Figure 4.2: Projections of u on M

Lemma 4.3.1. *Suppose*

$$M = span\{v_1,\ v_2,\ \cdots,\ v_m\} \subset R^n$$

where $m \leq n$ be a subspace spanned by linear independent $\{v_j\}$. The the projection matrix associated with M is

$$P = A\left(A^T A\right)^{-1} A^T$$

where $A = \left(v_1,\ v_2,\ \cdots,\ v_m\right)$, a $n \times m$ matrix, P is thus a $n \times n$ matrix. The matrix $\left(A^T A\right)^{-1} A^T$ is the pseudo inverse of A, denoted by A^+.

Proof. : For any vector u,

$$Pu = A\left(A^T A\right)^{-1} A^T u = A\left[\left(A^T A\right)^{-1} A^T u\right] \triangleq Aw$$

So

$$Pu = (v_1,\ v_2,\ \cdots,\ v_m)\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = \sum_{i=1}^{m} w_i v_i \in M$$

Next we need to prove that $u - Pu$ is perpendicular to M. In fact

$$A^T(u - Pu) = A^T u - A^T Pu = A^T u - A^T A\left(A^T A\right)^{-1} A^T u = A^T u - A^T u = 0$$

So $u - Pu$ is perpendicular to all v_j and thus to M too. It is easy to verify that Pu and $u - Pu$ are perpendicular too

$$(Pu)^T(u - Pu) = u^T P^T (u - Pu) = u^T P^T u - u^T P^T Pu = 0$$

because $P^T = P$ and $P^2 = P$. $\square$

Another interesting fact is the following.

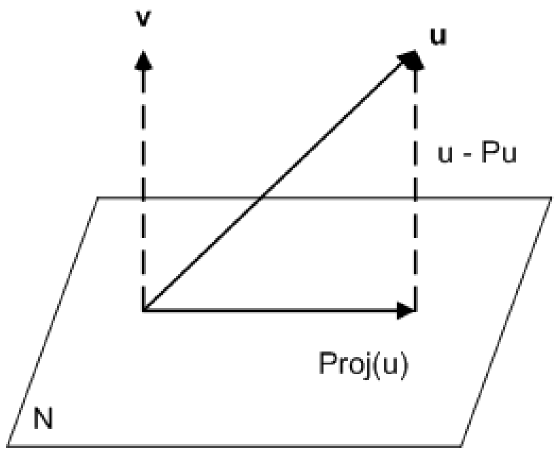

Figure 4.3: Projections of u on N

Lemma 4.3.2. *Suppose $v_1,\ v_2,\ \cdots,\ v_m \in R^n$ be linear independent and $m \le n$, let*

$$A = \left(v_1^T,\ v_2^T,\ \cdots,\ v_m^T\right)$$

be a $m \times n$ matrix. Now consider the following subspace:

$$N = \left\{\ x \mid Ax = 0\ \right\}$$

the perpendicular subspace to all $\{v_j\}$, i.e., $N = span\{\ v_j \mid j = 1,\ 2,\ \cdots\ m\}^{\perp}$. The projection matrix on N is given by

$$P = I - A^T\left(AA^T\right)^{-1}A$$

where I is the $m \times m$ identity matrix.

Proof. : For any vector u,

$$A(Pu) = A\left(I - A^T\left(AA^T\right)^{-1}A\right)u = Au - AA^T\left(AA^T\right)^{-1}Au = Au - Au = 0$$

It implies that Pu lies in N. Furthermore,

$$u - Pu = A^T\left(AA^T\right)^{-1}Au \triangleq A^T w = \left(v_1,\ v_2,\ \cdots\ v_m\right)\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = \sum_{i=1}^{m} w_i v_i$$

It implies that u-Pu lies in $span\{v_i\}$ and perpendicular to N. □

4.3.2 Reflection

Given a vector v, let $M = span\{v\}^{\perp}$, the perpendicular space to v. For any vector u, the reflection of u through M is given by

$$refl(u) = u - 2\ proj(u) = u - 2\frac{vv^T}{v^Tv}u = (I - 2\frac{vv^T}{v^Tv})u$$

Here we use the result from last section. So the reflection matrix for M is

$$Q_v = I - 2\frac{vv^T}{v^Tv} \tag{4.2}$$

It is easy to verify these properties:

- Q is symmetric, i.e., $Q^T = Q$.
- Q is orthogonal, i.e., $Q^TQ = I$.
- Q is involuntary, i.e., $Q^2 = I$, reflecting a vector twice should get back to the original vector.

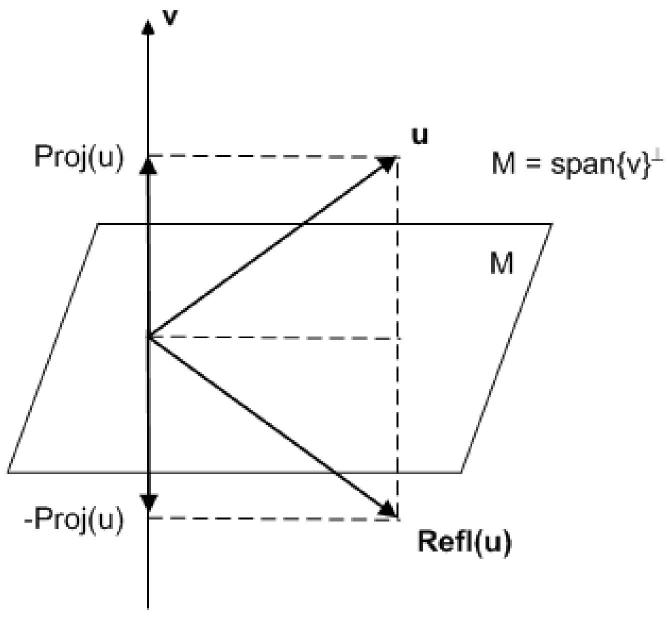

Figure 4.4: Reflection of u through M

Similar to the projection case, in numerical practice, we need to store only the vector v when applying the reflection to a vector or a matrix.

The following are several useful facts, the first one is used in the QR decomposition, the second one is used in the SVD, and the third one is used when we accumulate Householder matrices.

Lemma 4.3.3. *Given two vectors x and y with same norm, the reflection matrix that maps x to y is* Q_{x-y}.

Proof. : From the above picture, if we treat u as x and refl(u) as y, it's easy to see x-y = 2proj(u) is parallel to v. □

If x and y don't have same norm, we could normalize both vectors first.

A typical usage of this lemma is to map a given vector to a vector of our choice, such as e_1, the basis in R^n, so that we can eliminate all components along other basis. Repeating this step we can transform a matrix to a triangular matrix.

Lemma 4.3.4. *Let x be a vector in* R^n. *For any k with* $1 \le k \le n-2$, *we can find a* *vector u_k such that*

$$Q_{u_k}x = Q_{u_k}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ -S \\ 0 \\ \vdots \\ 0 \end{pmatrix} \triangleq = y$$

This means we can eliminate all components but the first two in x.

Proof. If we pick

$$S^2 = x_{k+1}^2 + x_{k+2}^2 + \cdots + x_n^2$$

Then x and y have the same norm. By the first lemma,

$$u_k = x - y = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ x_{k+1} \\ x_{k+2} \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ -S \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ x_{k+1} + S \\ x_{k+2} \\ \vdots \\ x_n \end{pmatrix}$$

To avoid rounding error, the sign of S is chosen to be the same as x_{k+1}. Furthermore, it is easy to verify

$$\left|u_k\right|_2^2 = (x_{k+1} + S)^2 + x_{k+2}^2 + \cdots + x_n^2 = 2x_{k+1}S + 2S^2$$

□

Lemma 3.

4.3.3 Rotation

Givens rotation matrix is defined as

$$G(i,j,c,s) = \begin{array}{cc} & \begin{array}{ccccccccccc} & & & i & & & & j & & & \end{array} \\ \begin{array}{c} \\ \\ \\ i \\ \\ \\ \\ j \\ \\ \\ \\ \end{array} & \left(\begin{array}{ccccccccccc} 1 & & & & & & & & & & \\ & . & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & c & . & . & . & s & & & \\ & & & . & 1 & & & & & & \\ & & & . & & . & & & & & \\ & & & . & & & 1 & & & & \\ & & & -s & . & . & . & c & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & . & \\ & & & & & & & & & & 1 \end{array}\right) \end{array}$$

where $c^2 + s^2 = 1$. Every rotation can be decomposed into a product of a series of the above rotations.

The Householder matrix is a reflection because $det(Q_v) = -1$. Then $-Q_v$ is a rotation(with the angle π).

The Jacobi rotation is a similar transformation using G, i.e., $G^T AG$. See the link.

4.4 QR decomposition

Let A be a n ×m matrix with n > m and $rank(A) = m$, the QR decomposition of A is

$$A = QR$$

where Q is a m ×n orthogonal matrix, i.e., $Q^T Q = I_n$, the n ×n identity matrix and R is an n ×n upper triangular matrix. Denote A as

$$A = \left(\vec{a}_1,\ \vec{a}_2,\ \cdots,\ \vec{a}_m\right)$$

where all $\vec{a}_j$ are linear independent in R^n with basis $\{\vec{e}_1,\ \vec{e}_2,\ \cdots,\ \vec{e}_n\}$ and

$$\vec{a}_j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{nj} \end{pmatrix}$$

The QR algorithm, using Householder transformations, is based on the Lemma 1. After the execution, R is stored in the strictly upper triangular portion of A and R's diagonal cells are stored in a separate array. Q is stored in the lower triangular portion indirectly, actually, the vectors generating the Householder transformations are stored. In order to get Q we need to multiply these Householder transformations together. For all j, we work on the sub-matrix from j to m:

1. compute $|\vec{a}_j|$, the 2-norm, square root of the sum of all components,

$$\sqrt{\sum_{i=1}^{n} a_{ij}}$$

   We further select the norm to have the same sign as a_{jj} to avoid the rounding error with subtractions in the step 3.

2. overwrite $|\vec{a}_j|$ with $\frac{\vec{a}_j}{|\vec{a}_j|}$ such that now $\vec{a}_j$ is a unit vector. Note that we are working on the sub vector below the diagonal.

3. construct the Householder transformation for the jth sub-column so that it is mapped to $\vec{e}_j$, the jth axis in R_n. By Lemma 1, the transformation is

$$Q_v = I - 2\frac{\vec{v}\vec{v}^T}{\vec{v}^T\vec{v}}$$

   where $\vec{v} = \vec{a}_j + \vec{e}_j$. Note that $vecv$ is the same as $veca_j$ except the first component is $a_{jj}+1$. So we just overwrite the first component of $\vec{a}_j$ with $a_{jj}+1$ and thus now the jth column is $\vec{v}$. This value will be used in the next step. As we progress j from 1 to m, the vector v has the form $\left(0,\ 0,\ \cdots,\ v_j,\ v_{j+1},\ \cdots,\ v_n\right)$, i.e., anything above the diagonal is zero.

4. Note that $\vec{a}_j$ now is a unit vector, in the Q_v expression above,

$$\vec{v}^T\vec{v} = \left(\vec{a}_j + \vec{e}_j\right)^T\left(\vec{a}_j + \vec{e}_j\right) = 1 + a_{jj} + a_{jj} + 1 = 2\left(a_{jj}+1\right) = 2v_j$$

   So now Q_v can be expressed

$$Q_v = I - 2\frac{\vec{v}\vec{v}^T}{\vec{v}^T\vec{v}} = I - 2\frac{\vec{v}\vec{v}^T}{2v_j} = I - \frac{\vec{v}\vec{v}^T}{v_j}$$

   where v_j is the overwritten value stored from the step 3.

5. We leave the jth column with the vector $\vec{v}$, and apply the Householder transformation Q_v to the rest of the columns using the formula:

$$Qx = \left(I - \frac{\vec{v}\vec{v}^T}{v_j}\right)x = x - \frac{\vec{v}^T x}{v_j}\vec{v}$$

   which is just a difference of two vectors x and v with a scaler.

6. store the norm from step 1 in a separate vector, this is part of the upper triangular matrix R because $Q\vec{a}_j = |\vec{a}_j| e_j$

Since we store all the vectors v's in the lower portion of A, we need to multiply them to get Q.

4.5 Singular Value Decomposition

The common practice of SVD is to decompose the given matrix to a bidiagonal matrix, then compute the singular values of the reduced bidiagonal matrix.

4.6 Eigen Decomposition

scaling

4.7 Reference

For mathematical treatment,

- Matrix Analysis, Horn and Johnson
- Topics in Matrix Analysis, Horn and Johnson
-

For numerical algorithms

- Lapack documents: http://www.netlib.org/lapack/lawns/downloads/
- Matrix Perturbation Theory, Stewart and Sun
- Matrix Algorithms, G.W. Stewart
- Fundamentals of Matrix Computations, D. Watkins

4.8 Bibliography

[1] Demmel, J.W., *Applied Numerical Linear Algebra*, SIAM, 1997.

[2] Golub, G.H., Loan, C.F., *Matrix Computations*, 3rd Edition, The Johns Hopkins University Press, 1996.

[3] Teukolsky, A., Vetterling, W.T., Flannery, B.P., *Numerical Recipes*, 3rd Edition, Cambridge University Press, 1997

[4] Higham, N.J., *Accuracy and Stability of Numerical Algorithms*, SIAM, 1996

[5] Sparse matrix

[6] Lawn 95.