C++ Matrix Class

E. Robert Tisdale

Abstract

This paper presents a simple, fast, efficient C++ Matrix class designed for scientists and engineers. The basic operations were implemented in C++ using the Gnu `g++` compiler (version 2.3.3) on Sun 4 computers running UNIX (SunOS Release 4.1.1). More sophisticated algorithms were implemented by interfacing with existing C routines. In order to demonstrate an application of the `Matrix` class, it was used to implement the backward error propagation algorithm for multi-layer, feed-forward artificial neural networks.

1 Introduction

The Array and Matrix classes permit two dimensional arrays to be treated as objects that can be included in arithmetic expressions with infix operators in much the same way as are scalar values. The array elements are stored in memory in row major order. Row (column) vectors are two dimensional arrays with just one row (column). A matrix with exactly one row and one column is a scalar.

1.1 class Array

An Array object contains four data members:

1. `int L` is the length of rows,
2. `int M` is the number of rows,
3. `int N` is the number of columns and
4. `double* X` is a pointer to the array.

An array object points to a contiguous block of memory organized into M rows of L numbers[1] which represents an $M \times N$ submatrix with $N \leq L$. See figure 1.

Figure 1: An array object points to a contiguous block of memory organized into M rows of L numbers which represents an $M \times N$ submatrix with $N \leq L$.

1 The Array class may actually be derived from single or double precision real or complex numbers.

Whenever an array object is declared explicitly, the user is responsible for allocating and deallocating memory for it. Almost all matrix operations are defined on class Array. But many operations return an object of class Matrix.

1.2 class Matrix

The derived class Matrix inherits all of the data members and operations of the base class Array. But it automatically allocates and deallocates memory for the array. Storage allocated for local variables (or intermediate results during expression evaluation) is deallocated as soon as they exit scope.

1.3 Indexing

Array and Matrix class objects are indexed in exactly the same way as two dimensional C arrays. If `A` is an $M \times N$ array, `A[i]` returns a pointer of type `double*` to the first element of row i and `A[i][j]` returns a reference of type `double&` to the element in column j of row i where $0 \le i < M$ and $0 \le j < N$.

1.4 Construction

The declaration `Matrix A(m, n)` creates an uninitialized $m \times n$ matrix. An optional third argument, scalar s, in the declaration `Matrix A(m, n, s)` initializes all the elements to s. The declaration `Matrix x(n)` is equivalent to the declaration `Matrix x(1, n)` and creates a uninitialized $1 \times n$ row vector. The declaration `Matrix B(A)` creates a new matrix B which is the same size as matrix A but allocates storage at a different location and copies the array. An optional third argument, pointer p, in the declaration `Matrix A(m, n, p)` creates an object which points to an $m \times n$ array at location p. This declaration should be used with care as the compiler will automatically generate code to delete the array at location p as soon as matrix A exits scope. The declaration `Array A(m, n, p)` should be used instead if the array at location p must survive matrix A.

1.5 Arithmetic Operations

Unary plus, `+A` $= A$, unary minus, `-A` $= -A$, scalar multiplication, `s*A` $= sA = As =$ `A*s`, scalar division, `A/s` $= A/s$, and element by element addition, `A + B` $= A+B$, subtraction, `A - B` $= A-B$, multiplication, `A*B` $= A \cdot B$, and division, `A/B` $= A/B$, all have the expected meanings. But other useful operations have been defined as well.

1.5.1 Matrix Multiplication

Matrix multiplication is effected by the infix operator, `%`, which computes an inner product, `A%B` , on the rows of the operands. This operation has indexing and memory referencing advantages[2] that tend to outweigh the cost of transposing either or both of the operands.

2 Striding across rows when multiplying large matrices results in frequent cache misses and page faults.

1.5.2 Scalar Operations

The same scalar operation is applied to every element of the array.

Scalar by Matrix

Addition

`s + A.`

Subtraction

`s - A.`

Division

`s/A.`

Matrix by Scalar

Addition

`A + s.`

Subtraction

`A - s.`

1.5.3 Vector Operations

Element by element operations between operands of different sizes are fatal errors unless one of the operands is a row (column) vector with the same number of columns (rows) as the other operand[3]. The row (column) vector is simply combined with each of the rows (columns) of the matrix.

Row Vectors

Addition

`x + A A + x.`

Subtraction

`x - A` and `A - x.`

Multiplication

`x*A` = $A\text{diag}(\vec{x})$ = `A*x.`

3 An operand with exactly one row and one column is treated as a scalar

Division

`x/A` and `A/x`.

Column Vectors

Addition

`x + A` $\rightrightarrows \vec{x}1 + A = A \rightrightarrows \vec{x}1$ `= A + x`.

Subtraction

`x - A` $\rightrightarrows \vec{x}1 - A$ and `A - x` $= A \rightrightarrows \vec{x}1$.

Multiplication

`x*A` $= \text{diag}(\vec{x})A =$ `A*x`.

Division

`x/A` $\rightrightarrows \vec{x}1/A$ and `A/x` $= A/(\vec{x}1)$.

1.6 Shift Operators

If the left operand is an array, *A*, and the right operand is an integer, *n*, the shift operator << (>>) returns a copy of *A* shifted *n* columns left (right). The operator << fills the empty columns with zeros. The operator >> copies the leftmost column into the empty columns. If the left operand is an `istream` and the right operand is an array, the right shift operator, >>, inputs numbers separated by white spaces in row major order until the entire array is filled. If the left operand is an `ostream` and the right operand is an array, the left shift operator, <<, outputs the entire array in row major order using the default output format and terminating each row with a new line. The `format` function specifies the default output format and returns an empty character string. It has three arguments. The first argument specifies the output format (initially `"%g"`) for each element of the array. The second argument (optional) specifies the number (initially `4`) of elements to display on each line. The third argument (optional) specifies the character string (initially `" "`) which separate each element displayed on a line.

1.7 Other Operators

Two arrays with the same number of columns can be stacked, `(A^B)`. Two arrays with the same number of rows can be adjoined, `(A|B)`. The outer product, `(A&B)`, on the rows of the operands should be avoided unless both operands are row vectors.

1.8 Assignment Operators

The assignment operator, =, copies each element of its right hand side into the corresponding element on its left hand side if both matrices are the same size. If the right hand side is a row (column) vector with the same number of rows (columns) of

the matrix on the left hand side, then it is copied into each of the rows (columns) of the matrix. If the right hand side is a scalar, then it is copied into every element of the matrix on the left hand side. Operators `+=`, `-=`, `*=`, `/=`, `<<=` and `>>=` were implemented with the expected meaning. But operator `%=` was not implemented.

1.9 Member Functions

1.9.1 Data Members

Although access to the data members, `L`, `M`, `N` and `X`, is not restricted in the implementation, the user must admit the possibility of an alternate implementation and should reference them only through the respective member functions, `l()`, `m()`, `n()` and `x()`.

1.9.2 Sub Array

`A.s(i, m, j, n)` is an $m \times n$ sub array beginning with row i and column j of array (matrix) A. `A.s(i, m, j)` is an $m \times (N-j)$ sub array beginning with row i and column j. `A.s(i, m)` is m rows beginning with row i. `A.s(i)` is row i. And `A.s()` is just row 0.

1.9.3 Sub Matrix

Since the sub array member function is an array object, it should only be applied to Array or Matrix objects declared explicitly by the user. It should never be applied to intermediate results in expressions because the storage allocated for the intermediate result may be reallocated upon return from the sub array function. The sub matrix member function is similar to the sub array function except that it actually allocates storage and copies the sub array into it. The intermediate result can safely be destroyed since it is no longer needed.

`A._(i, m, j, n)` returns an $m \times n$ sub matrix beginning with row i and column j of array (matrix) A. `A._(i, m, j)` returns an $m \times (N-j)$ sub matrix beginning with row i and column j. `A._(i, m)` returns m rows beginning with row i. `A._(i)` is row i. And `A._()` returns just row 0.

1.9.4 Transpose

`A.t()`.

1.9.5 Inverse

`A.i()`.

1.9.6 Row Sum

`A.sum()`.

1.9.7 Row Sum of Squares

`A.sumsq()`.

1.9.8 Mapping

`A.map(f)` .

1.9.9 Row Minimums

`A.min()` .

1.9.10 Row Maximums

`A.max()` .

1.9.11 Minimum Index

`A.min_index()` $=iL+j$ where .

1.9.12 Maximum Index

`A.max_index()` $=iL+j$ where .

1.9.13 Index

`A.index(s)` $=iL+j$ where .

1.9.14 Resize

`A.resize(m, n)` converts A to an $m \times n$ matrix. But `resize` can be called with any of the arguments passed to Matrix constructors.

1.10 Comparison Operators

A comparison is always valid when one of the operands in a comparison is a scalar. For example, the expression `s == A` is true if and only if for all i and j. A comparison of the form `A @ B` where `@` $\in$ {<, <=, ==, >=, >} is valid if and only if `B - A` is valid and true if and only if `0 @ B - A` is true.

2 Basetypes

The Matrix class was implemented for `double` and `complex` basetypes. The fast fourier cosine transform, `ffct`, and the signum function, `sgn`, are included in the `doubleArray` class. The `complexArray` class has no <, <=, > or >= operators and no `min`, `max`, `min_index` or `max_index` member functions but includes `fft`, `polar`, `conj`, `real`, `imag`, `norm` and `arg` functions. Both classes include `exp`, `log`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `tanh`, `sqrt` and `abs` functions.

3 Genclass

Instead of maintaining separate sources for both basetypes, the Gnu container class prototype mechanism[Error: Reference source not found] was used. The `Matrix.hP` header prototype file contains the class definitions and short inline functions. The `Matrix.ccP` source prototype file contains code for the longer

library routines. The Gnu `genclass` utility replaces every occurrence of `<T>` in these files with the desired basetype. But since the basetype is not the only difference between the container classes, the `genclass` shell script was modified to use the `m4` macro processor which selectively includes code depending upon the basetype. The `genclass double val Matrix` or `genclass complex val Matrix` command generates the respective header and source files, `double.Matrix.h` and `double.Matrix.C` or `complex.Matrix.h` and `complex.Matrix.C`, for the `double` or `complex` basetypes respectively.

4 Matrix Library

The Matrix class is intended to serve as a convenient interface to existing software. Matrix inversion and fast fourier transforms have been implemented with routines adapted from "Numerical Recipes in C"[Error: Reference source not found]. The singular value decomposition algorithm, `svdcmp`, was used to implement matrix inversion. The `four1`, `realft` and `cosft` routines were used to implement fourier transforms. The `doubleMatrix.C`, `complexMatrix.C`, `four1.C`, `realft.C` and `cosft.C` source files are compiled to object files using the `g++` compiler. Then the `ar` program is used to combine the object files into a single library archive, `libMatrix.a`. And the `ranlib` program is used to add a table of contents so that the routines can be linked more rapidly.

5 Backprop

The backward error propagation algorithm is illustrated in figure 2. An input/output pair is selected at random from the training sample. The input, $\vec{x}$, is fed into the bottom layer, 0, of the network. The signals, , feed forward, up through the network. The output of each layer is the input for the next layer. The difference between the desired output, $\vec{y}$, and the output, , of the top layer, $K-1$, is used to compute an equivalent error, , which is fed back into the top layer. The equivalent errors, , propagate backward, down through the network. The equivalent errors are used to update the biases, , and connection weights, , in each layer.

Figure 2: The signals, , feed forward, up through the layers. The equivalent errors, , feed backward, down through the layers. The equivalent errors are used to update the biases, , and the connection weights, .

The backward error propagation (backprop) algorithm can be summarized in three steps:

1. Feed Forward

(1)

(2)

(3)

2. Back Propagate

(4)

(5)

3. Update

(6)

(7)

where the learning rate, η, is a small number.

A multi-layer, feed-forward, artificial neural network can approximate any continuous function arbitrarily closely provided that there are enough (non-linear) neurons in the hidden layers. The output neurons are linear and the hyperbolic tangent, tanh(·), is used for the non-linear hidden neurons so that the derivatives, , are easily calculated.

The backprop algorithm itself is easy to implement with the C++ Matrix class.

```
x[0] = x;
for (k = 0; k < K-1; k++)     // Feed Forward
   x[k+1] = tanh(b[k] + x[k]%W[k]);
x[K] = b[K-1] + x[K-1]%W[K-1];

d[K-1] = eta*(y - x[K]);     // Back Propagate
for (k = K-1; k > 0; k--)
   d[k-1] = d[k]%W[k].t()*(1.0 - x[k]*x[k]);

for (k = 0; k < K; k++)      // Update
   {
      b[k] += d[k];
      W[k] += d[k]&x[k];
   };
```

The rest of the `backprop` program processes command line options, constructs arrays of matrices for the network and the training sample and reads data into them then writes the network matrices out again before it terminates. Network and training sample data are kept in separate files with different formats detailed in table 1 and table 2 respectively.

K	# of layers
	# of inputs to layer 0
	# of inputs to layer 1
	layer 0 biases & weights
	# of inputs to layer 2
	layer 1 biases & weights

⋮	
	number of outputs
	layer $K-1$ biases & weights

Table 1: Network file format.

m	# of examples
	example 0
	example 1
⋮	
	example $m-1$

Table 2: Sample file format.

5.1 Spiral Problem

A three layer network, $N(2,5,10,2)$, with two inputs, five hidden neurons in the first layer, ten hidden neurons in the second layer and two output neurons was used trained to compute the Cartesian coordinates,

$$(x,y)=\vec{f}(\varrho,\varphi)=(\varrho\cos(\varrho-\varphi),\varrho\sin(\varrho-\varphi)), \tag{8}$$

of a point on a spiral where $0\leq\varrho<\pi$ and $0\leq\varphi<\pi$ are the radius and angle respectively.

The network biases and connection weights were initialized to small random numbers and stored in file `ffnet.new`. The training sample is stored in file `ffnet.S`. Training is accomplished in three steps:

1. `mv ffnet.new ffnet.old`

   Move the file `ffnet.new` to file `ffnet.old`.

2. `cat ffnet.old ffnet.S | (backprop -v > ffnet.new) >>& ffnet.err &`

   The program reads from standard input and writes to standard output. The `ffnet.old` and `ffnet.S` files are concatenate together and piped to the `backprop` program. The `backprop` output is redirected to file `ffnet.new`. And any error messages are appended to file `ffnet.err`. It runs in the background so that the user can continue working until the program terminates.

3. `cat ffnet.new ffnet.S | evaluate | graph -m 0 | plot`

   In order to evaluate the network performance, the user plots the network output versus the desired output for every example in the training sample. The

`ffnet.new` and `ffnet.S` files are concatenated together, piped through `evaluate` and `graph` to `plot` which displays the plot on the user's terminal.

The previous steps can be repeated until the user concludes that the performance is acceptable or that the network will not learn the function.

A Notation

An N element row or column vector is usually denoted by a lower case symbol (i.e. $\vec{a}$). And an $M \times N$ matrix is usually denoted by an upper case symbol (i.e. A). Row $0 \leq i < M$ of matrix A is denoted . Column $0 \leq j < N$ of matrix A is denoted . And denotes the element in column j of row i. The inverse and transpose of matrix A are denoted and respectively. The vector $\vec{1}$ is a row vector of 1s. The matrix $\mathrm{diag}(\vec{a})$ is an $N \times N$ diagonal matrix where the diagonal elements are the N elements of vector $\vec{a}$. The expression is a row vector and is a matrix composed of scalar elements, and respectively. But is a matrix composed of column vectors, . Matrix multiplication, AB, and scalar multiplication, $sA = As$, are denoted by juxtaposition. But scalar division is denoted, A/s. Element by element addition, subtraction, multiplication, division and function evaluation are denoted $A+B$, $A-B$, $A \cdot B$, A/B and $f(A)$ respectively.