Systems of Linear Equations

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: March 2, 1999

I needed some standard matrix inversion routines for linear system solving, so I added the Numerical Recipes in C code. Modifications were made so that array indexing is zero–based. The class header is in `solve.h`:

```
class mgcLinearSystem
{
public:
    mgcLinearSystem() {;}

    float** NewMatrix (int N);
    void DeleteMatrix (int N, float** A);
    float* NewVector (int N);
    void DeleteVector (int N, float* B);

    int Inverse (int N, float** A);
    // Input:
    //     A[N][N], entries are A[row][col]
    // Output:
    //     return value is TRUE if successful, FALSE if pivoting failed
    //     A[N][N], inverse matrix

    int Solve (int N, float** A, float* b);
    // Input:
    //     A[N][N] coefficient matrix, entries are A[row][col]
    //     b[N] vector, entries are b[row]
    // Output:
    //     return value is TRUE if successful, FALSE if pivoting failed
    //     A[N][N] is inverse matrix
    //     b[N] is solution

    int SolveTri (int N, float* a, float* b, float* c, float* r, float* u);
    // Input:
    //     Matrix is tridiagonal.
    //     Lower diagonal a[N-1]
    //     Main  diagonal b[N]
    //     Upper diagonal c[N-1]
    //     Right-hand side r[N]
    // Output:
```

```
    //     return value is TRUE if successful, FALSE if pivoting failed
    //     u[N] is solution

    int SolveConstTri (int N, float a, float b, float c, float* r, float* u);
    // Input:
    //     Matrix is tridiagonal.
    //     Lower diagonal is constant, a
    //     Main  diagonal is constant, b
    //     Upper diagonal is constant, c
    //     Right-hand side r[N]
    // Output:
    //     return value is TRUE if successful, FALSE if pivoting failed
    //     u[N] is solution
};
```

Routine `Inverse` uses Gaussian elimination to invert a matrix. The computation is done in–place, so if you do not want the original matrix lost, save a copy. Routine `Solve` is the general solver which uses Gaussian elimination with pivoting. Routine `SolveTri` is a fast solver for tridiagonal matrices. Routine `SolveConstTri` is the same fast solver, but for tridiagonal matrices whose diagonals are constant values.

Here is an example for solving a general system. The abstract problem is

$$A\vec{x} = \begin{bmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \vec{R}$$

and has solution $\vec{x} = (1/4, 1/2, -1/4)$. The code is

```
const int N = 3;
int row;
mgcLinearSystem sys;

float** A = sys.NewMatrix(N);
float*  R = sys.NewVector(N);

// initialize A
A[0][0] =  1;  A[0][1] = 1;  A[0][2] = -1;
A[1][0] =  1;  A[1][1] = 0;  A[1][2] =  1;
A[2][0] = -1;  A[2][1] = 1;  A[2][2] =  1;

// initialize R
R[0] = 1;  R[1] = 0;  R[2] = 0;

// solve the system
if ( sys.Solve(N,A,R) ) {
    cout << "inverse is" << endl;
    for (row = 0; row < N; row++) {
```

```
        for (int col = 0; col < N; col++)
            cout << A[row][col] << ' ';
        cout << endl;
    }
    cout << "solution is:  ";
    for (row = 0; row < N; row++)
        cout << R[row] << ' ';
    cout << endl;
    }
else
    cout << "system is singular, cannot solve it" << endl;

sys.DeleteVector(N,R);
sys.DeleteMatrix(N,A);
```

Here is an example for solving a tridiagonal system. The abstract problem is

$$A\vec{x} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -1 & 2 \\ 0 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \vec{R}$$

and has solution $\vec{x} = (7/8, 1/8, -3/8)$. The code is

```
const int N = 3;
int row;
mgcLinearSystem sys;

// initialize the diagonals
float A[N-1] = { 1, 3 };
float B[N]   = { 1, -1, 1 };
float C[N-1] = { 1, 2 };
float R[N]   = { 1, 0, 0 };
float U[N];

// solve the system
if ( sys.SolveTri(N,A,B,C,R,U) ) {
    cout << "solution is:  ";
    for (row = 0; row < N; row++)
        cout << U[row] << ' ';
    cout << endl;
}
else
    cout << "pivoting failed, cannot solve it" << endl;
```

Here is an example for solving a constant tridiagonal system. The abstract problem is

$$A\vec{x} = \begin{bmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \vec{R}$$

and has solution $\vec{x} = (-3/4, -1/2, -1/4)$. The code is

```
const int N = 3;
int row;
mgcLinearSystem sys;

// initialize the diagonals
float A =  1;
float B = -2;
float C =  1;
float R[N] = { 1, 0, 0 };
float U[N];

// solve the system
if ( sys.SolveTri(N,A,B,C,R,U) ) {
    cout << "solution is:  ";
    for (row = 0; row < N; row++)
        cout << U[row] << ' ';
    cout << endl;
}
else
    cout << "pivoting failed, cannot solve it" << endl;
```

In the special case that A is a nonsingular symmetric matrix, it is possible to reduce the calculations (in Gaussian elimination) approximately in half. Use the $A = LDL^t$ decomposition where L is a lower triangular matrix whose diagonal entries are all 1 and where D is a diagonal matrix whose diagonal entries are positive. The basic algorithm can be found in Matrix Computations by G. Golub and C. Van Loan.

An abstract problem is

$$A\vec{x} = \begin{bmatrix} 10 & 20 & 30 \\ 20 & 45 & 80 \\ 30 & 80 & 171 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \vec{R}$$

and has solution $\vec{x} = (25.9, -20.4, 5)$. The code is

```
const int N = 3;
int row;
mgcLinearSystem sys;
```

```
// allocate matrix A
float** A = sys.NewMatrix(N);
float*  B = sys.NewVector(N);

// initialize A
A[0][0] = 10;  A[0][1] = 20;  A[0][2] =  30;
A[1][0] = 20;  A[1][1] = 45;  A[1][2] =  80;
A[2][0] = 30;  A[2][1] = 80;  A[2][2] = 171;

// initialize B
B[0] = 1;  B[1] = 0;  B[2] = 0;

// solve the system
if ( sys.SolveSymmetric(N,A,B) ) {
    cout << "output is" << endl;
    for (row = 0; row < N; row++) {
        for (int col = 0; col < N; col++)
            cout << A[row][col] << ' ';
        cout << endl;
    }
    cout << "solution is:  ";
    for (row = 0; row < N; row++)
        cout << B[row] << ' ';
    cout << endl;
}
else
    cout << "system is singular, cannot solve it" << endl;

// deallocate B
sys.DeleteVector(N,B);
sys.DeleteMatrix(N,A);
```

The matrix $A = LDL^t$ on output has $A_{ii} = D_{ii}$, $A_{ij} = L_{ij}$ for $i > j$, and its original entries for $i < j$.

To invert the same matrix A, the code is

```
// allocate matrices
float** A = sys.NewMatrix(N);
float** Ainv = sys.NewMatrix(N);

// initialize A
A[0][0] = 10;  A[0][1] = 20;  A[0][2] =  30;
A[1][0] = 20;  A[1][1] = 45;  A[1][2] =  80;
A[2][0] = 30;  A[2][1] = 80;  A[2][2] = 171;

// invert the matrix
if ( sys.SymmetricInverse(N,A,Ainv) ) {
    cout << "inverse is:  ";
```

```
    for (row = 0; row < N; row++) {
        for (int col = 0; col < N; col++)
            cout << Ainv[row][col] << ' ';
        cout << endl;
    }
    cout << endl;
}
else
    cout << "matrix is singular, cannot solve it" << endl;

sys.DeleteMatrix(N,A);
```

Unlike the Gaussian elimination for computing inverses, this routine does not compute the inverse in–place and requires an additional matrix to be allocated beforehand.