James D. McCaffrey
Software Research, Development, Testing, and Education

Inverting a Matrix using C#

Posted on March 6, 2015

Inverting a matrix is a surprisingly difficult challenge. I have my own library of C# matrix routines. I like to control my own code rather than relying on magic black box implementations, and I generally prefer to implement matrices using a plain array-of-arrays style rather than using an OOP approach.

I tested the code below by generating one million random matrices, with a dimension between 10 and 100, where each cell is a random value between -100.0 and +100.0. For each random matrix, I computed its inverse, then multiplied the inverse by the original matrix, and checked if the result was the identity matrix.

```
int n = rnd.Next(10, 100);
double[][] m = MatrixRandom(n, n, seed);
double[][] i = MatrixInverse(m);
double[][] I = MatrixIdentity(n);
double[][] p = MatrixProduct(m, i);
if (MatrixAreEqual(p, I, 1.0E-8))
  // pass
else
  // fail
```

The code passed 999,9999 out of 1,000,000 test cases. It failed once because of a round-off error. One of the problems with matrix inversion is that sometimes it just doesn't work.

The code is listed below. I always have trouble with the less-than and greater-than symbols so I did a text replacement for those symbols.

```
static double[][] MatrixCreate(int rows, int cols)
{
  double[][] result = new double[rows][];
  for (int i = 0; i less-than rows; ++i)
    result[i] = new double[cols];
  return result;
}

// --------------------------------------------------

static double[][] MatrixRandom(int rows, int cols,
  double minVal, double maxVal, int seed)
{
  // return a matrix with random values
  Random ran = new Random(seed);
  double[][] result = MatrixCreate(rows, cols);
  for (int i = 0; i less-than rows; ++i)
    for (int j = 0; j less-than cols; ++j)
      result[i][j] = (maxVal - minVal) *
        ran.NextDouble() + minVal;
  return result;
}

// --------------------------------------------------
```

```
static double[][] MatrixIdentity(int n)
{
  // return an n x n Identity matrix
  double[][] result = MatrixCreate(n, n);
  for (int i = 0; i less-than n; ++i)
    result[i][i] = 1.0;

  return result;
}

// --------------------------------------------------

static string MatrixAsString(double[][] matrix, int dec)
{
  string s = "";
  for (int i = 0; i less-than matrix.Length; ++i)
  {
    for (int j = 0; j less-than matrix[i].Length; ++j)
      s += matrix[i][j].ToString("F" + dec).PadLeft(8) + " ";
    s += Environment.NewLine;
  }
  return s;
}

// --------------------------------------------------

static bool MatrixAreEqual(double[][] matrixA,
  double[][] matrixB, double epsilon)
{
  // true if all values in matrixA == values in matrixB
  int aRows = matrixA.Length; int aCols = matrixA[0].Length;
  int bRows = matrixB.Length; int bCols = matrixB[0].Length;
  if (aRows != bRows || aCols != bCols)
    throw new Exception("Non-conformable matrices");

  for (int i = 0; i less-than aRows; ++i) // each row of A and B
    for (int j = 0; j less-than aCols; ++j) // each col of A and B
      //if (matrixA[i][j] != matrixB[i][j])
      if (Math.Abs(matrixA[i][j] - matrixB[i][j]) greater-than epsilon)
        return false;
  return true;
}

// --------------------------------------------------

static double[][] MatrixProduct(double[][] matrixA, double[][] matrixB)
{
  int aRows = matrixA.Length; int aCols = matrixA[0].Length;
  int bRows = matrixB.Length; int bCols = matrixB[0].Length;
  if (aCols != bRows)
    throw new Exception("Non-conformable matrices in MatrixProduct");

  double[][] result = MatrixCreate(aRows, bCols);

  for (int i = 0; i less-than aRows; ++i) // each row of A
    for (int j = 0; j less-than bCols; ++j) // each col of B
      for (int k = 0; k less-than aCols; ++k) // could use k less-than bRows
        result[i][j] += matrixA[i][k] * matrixB[k][j];

  //Parallel.For(0, aRows, i =greater-than
  //  {
  //    for (int j = 0; j less-than bCols; ++j) // each col of B
  //      for (int k = 0; k less-than aCols; ++k) // could use k less-than bRows
  //        result[i][j] += matrixA[i][k] * matrixB[k][j];
  //  }
  //);

  return result;
```

```
}

// --------------------------------------------------

static double[] MatrixVectorProduct(double[][] matrix,
  double[] vector)
{
  // result of multiplying an n x m matrix by a m x 1
  // column vector (yielding an n x 1 column vector)
  int mRows = matrix.Length; int mCols = matrix[0].Length;
  int vRows = vector.Length;
  if (mCols != vRows)
    throw new Exception("Non-conformable matrix and vector");
  double[] result = new double[mRows];
  for (int i = 0; i less-than mRows; ++i)
    for (int j = 0; j less-than mCols; ++j)
      result[i] += matrix[i][j] * vector[j];
  return result;
}

// --------------------------------------------------

static double[][] MatrixDecompose(double[][] matrix, out int[] perm,
  out int toggle)
{
  // Doolittle LUP decomposition with partial pivoting.
  // rerturns: result is L (with 1s on diagonal) and U;
  // perm holds row permutations; toggle is +1 or -1 (even or odd)
  int rows = matrix.Length;
  int cols = matrix[0].Length; // assume square
  if (rows != cols)
    throw new Exception("Attempt to decompose a non-square m");

  int n = rows; // convenience

  double[][] result = MatrixDuplicate(matrix);

  perm = new int[n]; // set up row permutation result
  for (int i = 0; i less-than n; ++i) { perm[i] = i; }

  toggle = 1; // toggle tracks row swaps.
  // +1 -greater-than even, -1 -greater-than odd. used by MatrixDeterminant

  for (int j = 0; j less-than n - 1; ++j) // each column
  {
    double colMax = Math.Abs(result[j][j]); // find largest val in col
    int pRow = j;
    //for (int i = j + 1; i less-than n; ++i)
    //{
    //  if (result[i][j] greater-than colMax)
    //  {
    //    colMax = result[i][j];
    //    pRow = i;
    //  }
    //}

    // reader Matt V needed this:
    for (int i = j + 1; i less-than n; ++i)
    {
      if (Math.Abs(result[i][j]) greater-than colMax)
      {
        colMax = Math.Abs(result[i][j]);
        pRow = i;
      }
    }
    // Not sure if this approach is needed always, or not.

    if (pRow != j) // if largest value not on pivot, swap rows
```

```
      {
        double[] rowPtr = result[pRow];
        result[pRow] = result[j];
        result[j] = rowPtr;

        int tmp = perm[pRow]; // and swap perm info
        perm[pRow] = perm[j];
        perm[j] = tmp;

        toggle = -toggle; // adjust the row-swap toggle
      }

      // --------------------------------------------------
      // This part added later (not in original)
      // and replaces the 'return null' below.
      // if there is a 0 on the diagonal, find a good row
      // from i = j+1 down that doesn't have
      // a 0 in column j, and swap that good row with row j
      // --------------------------------------------------

      if (result[j][j] == 0.0)
      {
        // find a good row to swap
        int goodRow = -1;
        for (int row = j + 1; row less-than n; ++row)
        {
          if (result[row][j] != 0.0)
            goodRow = row;
        }

        if (goodRow == -1)
          throw new Exception("Cannot use Doolittle's method");

        // swap rows so 0.0 no longer on diagonal
        double[] rowPtr = result[goodRow];
        result[goodRow] = result[j];
        result[j] = rowPtr;

        int tmp = perm[goodRow]; // and swap perm info
        perm[goodRow] = perm[j];
        perm[j] = tmp;

        toggle = -toggle; // adjust the row-swap toggle
      }
      // --------------------------------------------------
      // if diagonal after swap is zero . .
      //if (Math.Abs(result[j][j]) less-than 1.0E-20)
      //  return null; // consider a throw

      for (int i = j + 1; i less-than n; ++i)
      {
        result[i][j] /= result[j][j];
        for (int k = j + 1; k less-than n; ++k)
        {
          result[i][k] -= result[i][j] * result[j][k];
        }
      }


    } // main j column loop

    return result;
  } // MatrixDecompose

  // --------------------------------------------------

  static double[][] MatrixInverse(double[][] matrix)
  {
```

```
  int n = matrix.Length;
  double[][] result = MatrixDuplicate(matrix);

  int[] perm;
  int toggle;
  double[][] lum = MatrixDecompose(matrix, out perm,
    out toggle);
  if (lum == null)
    throw new Exception("Unable to compute inverse");

  double[] b = new double[n];
  for (int i = 0; i less-than n; ++i)
  {
    for (int j = 0; j less-than n; ++j)
    {
      if (i == perm[j])
        b[j] = 1.0;
      else
        b[j] = 0.0;
    }

    double[] x = HelperSolve(lum, b); //

    for (int j = 0; j less-than n; ++j)
      result[j][i] = x[j];
  }
  return result;
}

// --------------------------------------------------

static double MatrixDeterminant(double[][] matrix)
{
  int[] perm;
  int toggle;
  double[][] lum = MatrixDecompose(matrix, out perm, out toggle);
  if (lum == null)
    throw new Exception("Unable to compute MatrixDeterminant");
  double result = toggle;
  for (int i = 0; i less-than lum.Length; ++i)
    result *= lum[i][i];
  return result;
}

// --------------------------------------------------

static double[] HelperSolve(double[][] luMatrix, double[] b)
{
  // before calling this helper, permute b using the perm array
  // from MatrixDecompose that generated luMatrix
  int n = luMatrix.Length;
  double[] x = new double[n];
  b.CopyTo(x, 0);

  for (int i = 1; i less-than n; ++i)
  {
    double sum = x[i];
    for (int j = 0; j less-than i; ++j)
      sum -= luMatrix[i][j] * x[j];
    x[i] = sum;
  }

  x[n - 1] /= luMatrix[n - 1][n - 1];
  for (int i = n - 2; i greater-than-equal 0; --i)
  {
    double sum = x[i];
    for (int j = i + 1; j less-than n; ++j)
      sum -= luMatrix[i][j] * x[j];
```

```
    x[i] = sum / luMatrix[i][i];
  }

  return x;
}

// --------------------------------------------------

static double[] SystemSolve(double[][] A, double[] b)
{
  // Solve Ax = b
  int n = A.Length;

  // 1. decompose A
  int[] perm;
  int toggle;
  double[][] luMatrix = MatrixDecompose(A, out perm,
    out toggle);
  if (luMatrix == null)
    return null;

  // 2. permute b according to perm[] into bp
  double[] bp = new double[b.Length];
  for (int i = 0; i less-than n; ++i)
    bp[i] = b[perm[i]];

  // 3. call helper
  double[] x = HelperSolve(luMatrix, bp);
  return x;
} // SystemSolve

// --------------------------------------------------

static double[][] MatrixDuplicate(double[][] matrix)
{
  // allocates/creates a duplicate of a matrix.
  double[][] result = MatrixCreate(matrix.Length, matrix[0].Length);
  for (int i = 0; i less-than matrix.Length; ++i) // copy the values
    for (int j = 0; j less-than matrix[i].Length; ++j)
      result[i][j] = matrix[i][j];
  return result;
}

// --------------------------------------------------
```


Related

[Lower-Upper Matrix Decomposition Implementation in C#](#)
In "Machine Learning"

[Matrix Multiplication in Parallel with C# and the TPL](#)
In "Machine Learning"

[Finding the Inverse of a Matrix using Swarm Optimization](#)
In "Machine Learning"

This entry was posted in Machine Learning. Bookmark the permalink.

2 Responses to *Inverting a Matrix using C#*

Rafael Setragni *says:*
March 12, 2015 at 6:15 am

Hi James,

I completely understand your concerns about use a magic black box to do common operations on your code, and i think you're right about it. But, i believe there is another way to do this on C.

Instead of use the annotation "YourMatrix[line][column]", put inside your Matrix class a one dimensional array like this "YourArrayMatrix[line*column]". The access is more faster and you avoid to do a for inside another for. That is more slower than do a single for passing by all elements, even when the steps number are the same in both cases.

And believe, its really more faster find the algorithms, like "dotProduct" and a "inverse", respecting the one array logic than use a faster to understand for inside a for.

jamesdmccaffrey *says:*

March 14, 2015 at 7:52 am

Good points Rafael. The code I presented in this blog post is not very fast and I use it only when performance isn't a big concern. In the rare situations where I need my code to run as quickly as possible, I'll use C instead of C# (usually makes a big improvement). I don't often implement matrices as single arrays, but yes, I agree with you, in many situations that approach is very performant.

James D. McCaffrey

Blog at WordPress.com.