Homework 2

Peter McGill | github: petermcgill94 |UW : peterm28

May 2016

Question 1

The time complexity of solve_upper_triangular and solver_lower_triangular is defined as the the number of machine operations needed to be done as a function of N. Both of the these methods have a double nested for loop structure. The inner for loop contains a content time operation that is performed N times. The outer for loop contains the inner for loop and two other constant in time operations The outer loop is iterated N times, therefore we have:

$$T(N) = N[N \cdot 1 + 2 \cdot 1] \tag{1}$$

$$T(N) = N^2 + 2N \tag{2}$$

Because we care about complexity as N tends to a larger number then N^2 dominates:

$$T(N) \sim O(N^2) \tag{3}$$

Question 2

The code for my solve upper triangular method is shown below.

```
void solve_upper_triangular(double* out, double* U, double* b, int N) {

   //cycle overs rows
   for (int i = N - 1; i >= 0; i--) {
      out[i] = b[i];

      // cycle over columns
      for (int j = i + 1; j < N; j++) {
         out[i] -= U[i*N + j] * out[j];
      }
      out[i] /= U[i*N + i];
   }
}
```

This code solves an upper triangular system $Ux = b$ for x. (It returns x by reference in the variable out.) Where x and b are vectors of size N and U is a (N×N) matrix, with the only non zero elements being on the diagonal and above. Specifically it performs the following algorithm to find the elements of x:

$$x_m = \frac{b_m - \sum_{i=1}^{m-1} U_{mi} x_i}{U_{mm}} \tag{4}$$

In my method memory is accessed somewhat contiguously. Consider the place where U is accessed with U[i*N + j]. If we assume N to be 3 (could assume it to be any integer) we can see cycling through the for loops we access U in the following way: U[8] U[4],U[5], U[0], U[1] , U[2]. We can see it is accessing the rows contiguously. The vectors out and b are being accessed contiguously as i and j are decremented / incremented by one each time.

Assuming that the cache creates a copy of memory at and after a requested / particular location the method would be less efficient. This is because the first element of U that is requested and copied is the last element, and therefore future access would required getting memory before the requested location. Since we only have memory from after the first request stored in the cache we would have to got to ram for subsequent calls, Overall decreasing efficiency.

Question 3

The main method for my Gauss Seidel solver is shown below.

```
int gauss_seidel(double* out, double* A, double* b, int N, double epsilon)
{

  double * L = malloc(N*N*sizeof(double*));
  double * U = malloc(N*N*sizeof(double*));
  double * D = malloc(N*N*sizeof(double*));
  double * diff = malloc(N*sizeof(double*));
  // create L U D
  decompose(L, U, D, A, N);
  int count = 1;


  //initalize current
  double current[N];
  for(int i = 0; i < N;++i) {
     current[i] = 0.0;
     out[i] = 0.0;
  }

  //compute D + U
  double DU[N*N];
  for(int i = 0; i < N*N;++i) {
     DU[i] = 0.0;
  }
  mat_add(DU, D, U, N, N);

  //first iteration.
  gauss_step(out, DU, L, b, current, N);
  vec_sub(diff, out, current, N);
  double norm = vec_norm(diff, N);

  while (norm > epsilon) {
     count++;
     for (int i = 0; i < N; i++) {
        current[i] = out[i];
     }

     gauss_step(out, DU, L, b, current, N);
     vec_sub(diff, out, current, N);
     norm = vec_norm(diff, N);
  }

  free(D);
  free(U);
```

```
  free(L);
  free(diff);
  return count;
}
```

This method computes the Solution to the linear $Ax = b$ problem using the Gauss Seidal algorithm. It iteratively finds better and better solutions until they converge within radius epsilon. It returns 'x' by reference in out variable. It uses two helper methods, the first of which is decompose:

```
void decompose(double* L, double* U, double* D, double* A, int N)
{
  for (int i = 0; i < N; i++) {
     for(int j = 0; j < N; j++) {
        int index = i*N +j;
        //Lower matrix
        if (i > j) {
           L[index] = A[index];
           D[index] = 0.0;
           U[index] = 0.0;
        //upper matrix
        } else if (i < j) {
           U[index] = A[index];
           L[index] = 0.0;
           D[index] = 0.0;
        //diagonal
        } else {
           D[index] = A[index];
           L[index] = 0.0;
           U[index] = 0.0;
        }
     }
  }
}
```

This method simply decomposes the $(N \times N)$ matrix A into its diagonal component D, lower component L, and upper component U. It returns them by reference.

The other helper method is the gauss_step method:

```
void gauss_step(double* out, double* DU, double* L, double* b, double* xk, int N)
{
  //initialize tempory variable.
  double jvec[N];
  for( int i =0; i < N; i++) {
      jvec[i] = 0.0;
  }

  //computer iteration
  mat_vec(jvec, L, xk, N, N);
  vec_sub(jvec, b, jvec, N);
  solve_upper_triangular(out, DU, jvec, N);
}
```

This method performs one iteration of the Gauss Seidel algorithm. Namely is solves the upper triangular system $(D + U)x_{k+1} = b - L(x_k)$. It returns x_{k+1} by reference in the variable out.

Addressing the memory requirements. The method overall creates 4 ($N \times N$) matrices (D,U,L,DU) and 3 vectors of size N. (out, current, jvec).

$$Mem(N) \sim 4N^2 + 3N \tag{5}$$

So if we were to double N our memory requirement would increase by a factor of $\simeq 3.5$.

Contiguously memory considerations. The decompose method accesses contiguous memory as D, L, U are build from A row by row. Which means we are accessing and writing to elements adjacent in to each other in memory. In the gauss_step method memory is accessed contiguously in mat_vec vec_sub, but not in solve_upper_triangular. (See question 2). In the main method gauss_seidel matt_add and all initializations of variable are happening row by row and therefore contiguously. Also in the while loop in the main method vec_norm and vec_sub accesses memory contiguously.

Overall, they way to improve the gauss_seidel method is to have complete contiguous memory access in the solve_upper_triangular method. This would involve a re-implementation of the algorithm - possibly a re-arrangement of the for loops, or a recursive approach to this algorithm may be a better solution.