Homework 8

Elizabeth Hunt

December 9, 2023

1 Question One

See `UTEST(jacobi, solve_jacobi)` in `test/jacobi.t.c` and the entry `Jacobi / Gauss-Siedel -> solve_jacobi` in the LIZFCM API documentation.

2 Question Two

We cannot just perform the Jacobi algorithm on a Leslie matrix since it is obviously not diagonally dominant - which is a requirement. It is certainly not always the case, but, if a Leslie matrix L is invertible, we can first perform gaussian elimination on L augmented with n_{k+1} to obtain n_k with the Jacobi method. See `UTEST(jacobi, leslie_solve)` in `test/jacobi.t.c` for an example wherein this method is tested on a Leslie matrix to recompute a given initial population distribution.
In terms of accuracy, an LU factorization and back substitution approach will always be as correct as possible within the limits of computation; it's a direct solution method. It's simply the nature of the Jacobi algorithm being a convergent solution that determines its accuracy.
LU factorization also performs in order $O(n^3)$ runtime for an $n \times n$ matrix, whereas the Jacobi algorithm runs in order $O(kn^2) = O(n^2)$ on average but with the con that k is given by some function on both the convergence criteria and the number of nonzero entries in the matrix - which might end up worse in some cases than the LU decomp approach.

3 Question Three

See `UTEST(jacobi, gauss_siedel_solve)` in `test/jacobi.t.c` which runs the same unit test as `UTEST(jacobi, solve_jacobi)` but using the `Jacobi / Gauss-Siedel -> gauss_siedel_solve` method as documented in the LIZFCM API reference.

4 Question Four, Five

We produce the following operation counts (by hackily adding the operation count as the last element to the solution vector) and errors - the sum of each vector elements' absolute value away from 1.0 using the proceeding patch and unit test.

N	JAC opr	JAC err	GS opr	GS err	LU opr	LU err
5	1622	0.001244	577	0.000098	430	0.000000
6	2812	0.001205	775	0.000080	681	0.000000
7	5396	0.001187	860	0.000178	1015	0.000000
8	5618	0.001468	1255	0.000121	1444	0.000000
9	7534	0.001638	1754	0.000091	1980	0.000000
10	10342	0.001425	1847	0.000435	2635	0.000000
11	12870	0.001595	2185	0.000368	3421	0.000000
12	17511	0.001860	2912	0.000322	4350	0.000000
13	16226	0.001631	3362	0.000270	5434	0.000000
14	34333	0.001976	3844	0.000121	6685	0.000000
15	38474	0.001922	4358	0.000311	8115	0.000000
16	40405	0.002061	4904	0.000204	9736	0.000000
17	58518	0.002125	5482	0.000311	11560	0.000000
18	68079	0.002114	6092	0.000279	13599	0.000000
19	95802	0.002159	6734	0.000335	15865	0.000000
20	85696	0.002141	7408	0.000289	18370	0.000000
21	89026	0.002316	8114	0.000393	21126	0.000000
22	101537	0.002344	8852	0.000414	24145	0.000000
23	148040	0.002323	9622	0.000230	27439	0.000000
24	137605	0.002348	10424	0.000213	31020	0.000000
25	169374	0.002409	11258	0.000894	34900	0.000000
26	215166	0.002502	12124	0.000564	39091	0.000000
27	175476	0.002616	13022	0.000535	43605	0.000000
28	268454	0.002651	13952	0.000690	48454	0.000000
29	267034	0.002697	14914	0.000675	53650	0.000000
30	277193	0.002686	15908	0.000542	59205	0.000000
31	336792	0.002736	16934	0.000390	65131	0.000000
32	293958	0.002741	17992	0.000660	71440	0.000000
33	323638	0.002893	19082	0.001072	78144	0.000000
34	375104	0.003001	20204	0.001018	85255	0.000000
35	436092	0.003004	21358	0.000912	92785	0.000000
36	538143	0.003005	22544	0.000954	100746	0.000000
37	511886	0.003029	23762	0.000462	109150	0.000000
38	551332	0.003070	25012	0.000996	118009	0.000000
39	592750	0.003110	26294	0.000989	127335	0.000000
40	704208	0.003165	27608	0.000583	137140	0.000000

```
diff --git a/src/matrix.c b/src/matrix.c
index 901a426..af5529f 100644
--- a/src/matrix.c
+++ b/src/matrix.c
@@ -144,20 +144,54 @@ Array_double *solve_matrix_lu_bsubst(Matrix_double *m, Array_double *b)
    assert(b->size == m->rows);
    assert(m->rows == m->cols);

+  double opr = 0;
+
+  opr += b->size;
    Array_double *x = copy_vector(b);
+
+  size_t n = m->rows;
```

```
+  opr += n * n;     // (u copy)
+  opr += n * n;     // l_empty
+  opr += n * n + n; // copy + put_identity_diagonal
+  opr += n;         // pivot check
+  opr += m->cols;
+  for (size_t x = 0; x < m->cols; x++) {
+    opr += (m->rows - (x + 1));
+    for (size_t y = x + 1; y < m->rows; y++) {
+      opr += 1;
+      opr += 2;     // -factor
+      opr += 4 * n; // scale, add_v, free_vector
+      opr += 1;     // -factor
+    }
+  }
+  opr += n;
   Matrix_double **u_l = lu_decomp(m);
+
   Matrix_double *u = u_l[0];
   Matrix_double *l = u_l[1];

+  opr += n;
+  for (int64_t row = n - 1; row >= 0; row--) {
+    opr += 2 * (n - row);
+    opr += 1;
+  }
   Array_double *b_fsub = fsubst(l, b);
+
+  opr += n;
+  for (size_t x = 0; x < n; x++) {
+    opr += 2 * (x + 1);
+    opr += 1; // /= l->data[row]->data[row]
+  }
   x = bsubst(u, b_fsub);
-  free_vector(b_fsub);

+  free_vector(b_fsub);
   free_matrix(u);
   free_matrix(l);
   free(u_l);

-  return x;
+  Array_double *copy = add_element(x, opr);
+  free_vector(x);
+  return copy;
 }

 Matrix_double *gaussian_elimination(Matrix_double *m) {
@@ -231,18 +265,36 @@ Array_double *jacobi_solve(Matrix_double *m, Array_double *b,
   assert(b->size == m->cols);
   size_t iter = max_iterations;
```

```
+  double opr = 0;
+
+  opr += 2 * b->size; // to initialize two vectors with the same dim of b twice
   Array_double *x_k = InitArrayWithSize(double, b->size, 0.0);
   Array_double *x_k_1 =
       InitArrayWithSize(double, b->size, rand_from(0.1, 10.0));

+  // add since these wouldn't be accounter for after the loop
+  opr += 1; // iter decrement
+  opr +=
+      3 * x_k_1->size; // 1 to perform x_k_1, x_k and 2 to perform ||x_k_1||_2
   while ((--iter) > 0 && l2_distance(x_k_1, x_k) > l2_convergence_tolerance) {
+    opr += 1; // iter decrement
+    opr +=
+        3 * x_k_1->size; // 1 to perform x_k_1, x_k and 2 to perform ||x_k_1||_2
+
+    opr += m->rows; // row for add oprs
     for (size_t i = 0; i < m->rows; i++) {
       double delta = 0.0;
+
+      opr += m->cols;
       for (size_t j = 0; j < m->cols; j++) {
         if (i == j)
           continue;
+
+        opr += 1;
         delta += m->data[i]->data[j] * x_k->data[j];
       }
+
+      opr += 2;
       x_k_1->data[i] = (b->data[i] - delta) / m->data[i]->data[i];
     }

@@ -251,8 +303,9 @@ Array_double *jacobi_solve(Matrix_double *m, Array_double *b,
     x_k_1 = tmp;
   }

-  free_vector(x_k);
-  return x_k_1;
+  Array_double *copy = add_element(x_k_1, opr);
+  free_vector(x_k_1);
+  return copy;
 }

 Array_double *gauss_siedel_solve(Matrix_double *m, Array_double *b,
@@ -262,30 +315,48 @@ Array_double *gauss_siedel_solve(Matrix_double *m, Array_double *b,
   assert(b->size == m->cols);
   size_t iter = max_iterations;

+  double opr = 0;
+
```

```
+  opr += 2 * b->size; // to initialize two vectors with the same dim of b twice
   Array_double *x_k = InitArrayWithSize(double, b->size, 0.0);
   Array_double *x_k_1 =
       InitArrayWithSize(double, b->size, rand_from(0.1, 10.0));

   while ((--iter) > 0) {
+    opr += 1; // iter decrement
+
+    opr += x_k->size; // copy oprs
     for (size_t i = 0; i < x_k->size; i++)
       x_k->data[i] = x_k_1->data[i];

+    opr += m->rows; // row for add oprs
     for (size_t i = 0; i < m->rows; i++) {
       double delta = 0.0;
+
+      opr += m->cols;
       for (size_t j = 0; j < m->cols; j++) {
         if (i == j)
           continue;
+
+        opr += 1;
         delta += m->data[i]->data[j] * x_k_1->data[j];
       }
+
+      opr += 2;
       x_k_1->data[i] = (b->data[i] - delta) / m->data[i]->data[i];
     }

+    opr +=
+        3 * x_k_1->size; // 1 to perform x_k_1, x_k and 2 to perform ||x_k_1||_2
     if (l2_distance(x_k_1, x_k) <= l2_convergence_tolerance)
       break;
   }

   free_vector(x_k);
-  return x_k_1;
+
+  Array_double *copy = add_element(x_k_1, opr);
+  free_vector(x_k_1);
+  return copy;
 }
```

And this unit test:

```
UTEST(hw_8, p4_5) {
  printf("| N | JAC opr | JAC err | GS opr | GS err | LU opr | LU err | \n");

  for (size_t i = 5; i < 100; i++) {
    Matrix_double *m = generate_ddm(i);
    double oprs[3] = {0.0, 0.0, 0.0};
    double errs[3] = {0.0, 0.0, 0.0};
```

```
Array_double *b_1 = InitArrayWithSize(double, m->rows, 1.0);
Array_double *b = m_dot_v(m, b_1);
double tolerance = 0.001;
size_t max_iter = 400;

// JACOBI
{
  Array_double *solution_with_opr_count =
      jacobi_solve(m, b, tolerance, max_iter);
  Array_double *solution = slice_element(solution_with_opr_count,
                                         solution_with_opr_count->size - 1);

  for (size_t i = 0; i < solution->size; i++)
    errs[0] += fabs(solution->data[i] - 1.0);

  oprs[0] =
      solution_with_opr_count->data[solution_with_opr_count->size - 1];

  free_vector(solution);
  free_vector(solution_with_opr_count);
}

// GAUSS-SIEDEL
{
  Array_double *solution_with_opr_count =
      gauss_siedel_solve(m, b, tolerance, max_iter);
  Array_double *solution = slice_element(solution_with_opr_count,
                                         solution_with_opr_count->size - 1);

  for (size_t i = 0; i < solution->size; i++)
    errs[1] += fabs(solution->data[i] - 1.0);

  oprs[1] =
      solution_with_opr_count->data[solution_with_opr_count->size - 1];

  free_vector(solution);
  free_vector(solution_with_opr_count);
}

// LU-BSUBST
{
  Array_double *solution_with_opr_count = solve_matrix_lu_bsubst(m, b);
  Array_double *solution = slice_element(solution_with_opr_count,
                                         solution_with_opr_count->size - 1);

  for (size_t i = 0; i < solution->size; i++)
    errs[2] += fabs(solution->data[i] - 1.0);

  oprs[2] =
      solution_with_opr_count->data[solution_with_opr_count->size - 1];
```

```
      free_vector(solution);
      free_vector(solution_with_opr_count);
    }
    free_matrix(m);
    free_vector(b_1);
    free_vector(b);

    printf("| %zu | %f | %f | %f | %f | %f | %f | \n", i, oprs[0], errs[0],
           oprs[1], errs[1], oprs[2], errs[2]);
  }
}
```