

```
1: /*
2:  * gauss.cpp / Gaussian Elimination and Jacobi Method Implementation
3:  * Gaussian Elimination vs Jacobi Method Project, MCS 471 F2008
4:  * by William R. Fraser <wfraser@cs.uic.edu> 11/21/2008
5:  */
6:
7: #include <stdio.h>
8: #include <math.h>
9: #include <string.h>
10: #include <limits>
11: #include <boost/multi_array.hpp>
12:
13: #define DOUBLE 1
14: #define FLOAT 2
15:
16: #ifndef TYPE
17: #define TYPE DOUBLE
18: #endif
19:
20: #if TYPE == FLOAT
21: #   define ELEM_TYPE float
22: #   define ELEM_PRINT "%.6f"
23: #   define ELEM_SCIENT "%.10e"
24: #   define ELEM_DUMP "%a"
25: #elif TYPE == DOUBLE
26: #   define ELEM_TYPE double
27: #   define ELEM_PRINT "%.6lf"
28: #   define ELEM_SCIENT "%.10le"
29: #   define ELEM_DUMP "%la"
30: #endif
31:
32: // how many times do we let the Jacobi Iteration's residual increase (or remain
33: // constant) before we consider it to be diverging (or stagnating) and terminate
34: // it?
35: #define DIVERGE_DETECT 15
36:
37: using std::numeric_limits;
38:
39: typedef boost::multi_array<ELEM_TYPE, 2> Matrix;
40: typedef Matrix::array_view<2>::type MatrixView;
41: typedef Matrix::extent_gen Dimension;
42: typedef Matrix::index_gen Indices;
43:
44: enum Mode {GAUSS, JACOBI};
45: enum TestMode {TEST, NOTEST};
46:
47: void usage(char *appname)
48: {
49:     printf("usage: %s <file> [<file2> <gauss | gauss_test | "
50:           "jacobi <tolerance> | jacobi_test <tolerance>]\n"
51:           "1 file:\n"
52:           " Inverts the matrix in the given file.\n"
53:           " Raw dump goes to stdout, human-readable info goes to stderr.\n"
54:           "2 files:\n"
55:           " Solves the system given by <file1>x = <file2>\n"
56:           " The matrices in file1 and file2 must have the same number of rows.\n"
57:           " Raw dump goes to stdout, human-readable info goes to stderr.\n"
58:           "\n"
59:           "The format of matrix files is as follows:\n"
60:           " It must start with two integers, which give the row and column dimensions\n"
61:           " of the matrix.\n"
62:           " Then it must contain the entries of the matrix in row-major order, in any\n"
63:           " format recognizable by scanf(%%a). This is just about any float format,\n"
```

```
64: " with or without decimal points, with or without scientific notation, but\n"
65: " most interestingly, it allows raw float representation, where the number is\n"
66: " given in hexadecimal and is the exact machine representation of the float.\n"
67: " Read the man page for scanf(3) for more info.\n"
68: " The entries may be separated by any amount of whitespace/newlines.\n",
69:     appname);
70:     exit(1);
71: }
72:
73: /*
74:  * Prints a matrix in human-readable form.
75:  * Optionally give it a name to make the output even prettier.
76:  * Prints to stderr.
77:  */
78: void print(Matrix & m, const char *name = "")
79: {
80:     size_t rows = m.shape()[0];
81:     size_t cols = m.shape()[1];
82:
83:     for (size_t i = 0; i < rows; i++) {
84:         for (size_t j = 0; j < cols; j++) {
85:             fprintf(stderr, "%s(%zu,%zu):"ELEM_PRINT"\n",
86:                     name, i, j, m[i][j]);
87:         }
88:         fprintf(stderr, "\n");
89:     }
90: }
91:
92: /*
93:  * Print a column vector in human-readable form.
94:  * Optionally give it a name to make the output even prettier.
95:  * Prints to stderr.
96:  */
97: void print_vector(Matrix & v, const char *name = "")
98: {
99:     size_t rows = v.shape()[0];
100:
101:     for (size_t i = 0; i < rows; i++) {
102:         fprintf(stderr, "%s_%zu = "ELEM_PRINT"\n", name, i, v[i][0]);
103:     }
104: }
105:
106: /*
107:  * Dump a matrix in machine-readable form.
108:  * Prints to stdout.
109:  */
110: void dump(Matrix & m)
111: {
112:     size_t rows = m.shape()[0];
113:     size_t cols = m.shape()[1];
114:
115:     printf("%zu %zu\n", rows, cols);
116:     for (size_t i = 0; i < rows; i++) {
117:         for (size_t j = 0; j < cols; j++) {
118:             printf(ELEM_DUMP "\n", m[i][j]);
119:         }
120:     }
121: }
122: /*
123:  * Matrix subtraction.
124:  * Takes n^2 time.
125:  */
126: Matrix & subtract(Matrix &a, Matrix &b, unsigned long long *nops = NULL)
```

```
127: {
128:     size_t rows = a.shape()[0];
129:     size_t cols = a.shape()[1];
130:     unsigned long long n = 0;
131:
132:     if (rows != b.shape()[0] || cols != b.shape()[1]) {
133:         printf("cannot subtract matrices of different dimensions!\n");
134:         exit(4);
135:     }
136:
137:     Dimension d;
138:     Matrix *r = new Matrix(d[rows][cols]);
139:
140:     for (size_t i = 0; i < rows; i++) {
141:         for (size_t j = 0; j < cols; j++) {
142:             n++;
143:             (*r)[i][j] = a[i][j] - b[i][j];
144:         }
145:     }
146:
147:     if (nops != NULL)
148:         *nops = n;
149:
150:     return *r;
151: }
152:
153: /*
154:  * General-case matrix multiply.
155:  * Takes n^3 time.
156:  */
157: Matrix & multiply(Matrix &a, Matrix &b, unsigned long long *nops = NULL)
158: {
159:     size_t a_rows = a.shape()[0];
160:     size_t a_cols = a.shape()[1];
161:     size_t b_rows = b.shape()[0];
162:     size_t b_cols = b.shape()[1];
163:     unsigned long long n = 0;
164:
165:     if (a_cols != b_rows) {
166:         printf("error: attempt to multiply %zux%zu and %zux%zu matrices. :(\n",
167:             a_rows, a_cols, b_rows, b_cols);
168:         exit(4);
169:     }
170:
171:     size_t inner_dim = a_cols;
172:
173:     Dimension d;
174:     Matrix *r = new Matrix(d[a_rows][b_cols]);
175:
176:     for (size_t i = 0; i < a_rows; i++) {
177:         for (size_t j = 0; j < b_cols; j++) {
178:             (*r)[i][j] = 0;
179:             for (size_t k = 0; k < inner_dim; k++) {
180:                 n++;
181:                 (*r)[i][j] += a[i][k] * b[k][j];
182:             }
183:         }
184:     }
185:
186:     if (nops != NULL)
187:         *nops = n;
188:
189:     return *r;
```

```
190: }
191:
192: /*
193:  * This is a special case of matrix multiply where the first is a strictly
194:  * diagonal matrix and the second is a matrix where the diagonal is all zeroes.
195:  * In this case, the operation takes not the usual  $n^3$  time, but only  $n^2$  time.
196:  */
197: Matrix & special_multiply(Matrix &d, Matrix &lu,
198:     unsigned long long *nops = NULL)
199: {
200:     size_t d_rows = d.shape()[0];
201:     size_t d_cols = d.shape()[1];
202:     size_t lu_rows = lu.shape()[0];
203:     size_t lu_cols = lu.shape()[1];
204:     unsigned long long n = 0;
205:
206:     if (d_cols != lu_rows) {
207:         printf("error: attempt to multiply %zux%zu and %zux%zu matrices. :(\n",
208:             d_rows, d_cols, lu_rows, lu_cols);
209:         exit(4);
210:     }
211:
212:     Dimension dim;
213:     Matrix *r = new Matrix(dim[d_rows][lu_cols]);
214:
215:     for (size_t i = 0; i < d_rows; i++) {
216:         for (size_t j = 0; j < lu_cols; j++) {
217:             n++;
218:             if (i == j)
219:                 (*r)[i][j] = 0;
220:             else
221:                 (*r)[i][j] = lu[i][j] * d[i][i];
222:         }
223:     }
224:
225:     if (nops != NULL)
226:         *nops = n;
227:
228:     return *r;
229: }
230:
231: /*
232:  * This is a special case of matrix multiply where the first is a strictly
233:  * diagonal matrix and the second is a column vector.
234:  * In this case, the operation takes not the usual  $n^3$  time, but only  $n$  time.
235:  */
236: Matrix & vector_diag_multiply(Matrix &d, Matrix &b,
237:     unsigned long long *nops = NULL)
238: {
239:     size_t d_rows = d.shape()[0];
240:     size_t d_cols = d.shape()[1];
241:     size_t b_rows = b.shape()[0];
242:     size_t b_cols = b.shape()[1];
243:     unsigned long long n = 0;
244:
245:     if (d_cols != b_rows) {
246:         printf("error: attempt to multiply %zux%zu and %zux%zu matrices. :(\n",
247:             d_rows, d_cols, b_rows, b_cols);
248:         exit(4);
249:     }
250:
251:     if (b_cols != 1) {
252:         printf("error: vector_diag_multiply() only works if the 2nd arg is a "
```

```
253:         "column vector. :(\n");
254:     exit(4);
255: }
256:
257: Dimension dim;
258: Matrix *r = new Matrix(dim[d_rows][1]);
259:
260: for (size_t i = 0; i < d_rows; i++) {
261:     n++;
262:     (*r)[i][0] = d[i][i] * b[i][0];
263: }
264:
265: if (nops != NULL)
266:     *nops = n;
267:
268: return *r;
269: }
270:
271: /*
272:  * Computes the infinity norm of the given matrix.
273:  * I.e., the sum of the largest row.
274:  * Takse n^2 time.
275:  */
276: ELEM_TYPE infinity_norm(Matrix &m)
277: {
278:     size_t rows = m.shape()[0];
279:     size_t cols = m.shape()[1];
280:
281:     ELEM_TYPE max = 0;
282:
283:     for (size_t i = 0; i < rows; i++) {
284:         ELEM_TYPE row = 0;
285:         for (size_t j = 0; j < cols; j++) {
286:             row += m[i][j];
287:         }
288:         if (fabs(row) > fabs(max))
289:             max = row;
290:     }
291:
292:     return max;
293: }
294:
295: /*
296:  * Perform a Jacobi iteration to solve the system Ax = b.
297:  * tolerance: stops computing when the residual is below this value
298:  * iters: stops computing when this number of iterations has been done
299:  * nops: optional pointer to store the number of operations done
300:  * niters: optional pointer to store the number of iterations done
301:  */
302: Matrix & jacobi_solve(Matrix & A, Matrix & b,
303:     ELEM_TYPE tolerance = 0.0,
304:     unsigned long long iters = 0,
305:     unsigned long long *nops = NULL,
306:     unsigned long long *niters = NULL,
307:     bool verbose = true)
308: {
309:     size_t a_rows, a_cols, b_rows, b_cols;
310:     unsigned long long n = 0;
311:
312:     if (iters == 0)
313:         iters = numeric_limits<unsigned long long>::max();
314:
315:     a_rows = A.shape()[0];
```

```
316:     a_cols = A.shape()[1];
317:     b_rows = b.shape()[0];
318:     b_cols = b.shape()[1];
319:
320:     if (a_rows != b_rows) {
321:         printf("Can't do the Jacobi iteration on %zux%zu and %zux%zu matrices:"
322:             " the row counts must match. :(\n",
323:             a_rows, a_cols, b_rows, b_cols);
324:         exit(3);
325:     }
326:
327:     Dimension d;
328:     // note that "D" will really mean  $-D^{-1}$ , and L and U will be combined
329:     // into matrix "LU".
330:     Matrix LU(d[a_rows][a_cols]), D(d[a_rows][a_cols]), nD(d[a_rows][a_cols]);
331:
332:     // split A into upper, diagonal, and lower matrices
333:     for (size_t i = 0; i < a_rows; i++) {
334:         for (size_t j = 0; j < a_cols; j++) {
335:             n++;
336:             if (i == j) {
337:                 LU[i][j] = 0.0;
338:                 D[i][j] = -1.0 / A[i][j]; // negate and invert
339:             } else {
340:                 LU[i][j] = A[i][j];
341:                 D[i][j] = 0.0;
342:             }
343:         }
344:     }
345:
346:     unsigned long long n2 = 0;
347:     Matrix M = special_multiply(D, LU, &n2);
348:     n += n2;
349:
350:     Matrix Db = vector_diag_multiply(D, b, &n2);
351:     n += n2;
352:
353:     Matrix *x = new Matrix(d[b_rows][b_cols]);
354:
355:     ELEM_TYPE old_residual = numeric_limits<ELEM_TYPE>::max();
356:     ELEM_TYPE new_residual;
357:     unsigned long long i;
358:     unsigned long long ndiverge = 0;
359:
360:     if (verbose)
361:         fprintf(stderr, "%llu ops before iteration begins.\n", n);
362:
363:     for (i = 0; i < iters; i++) {
364:         n2 = 0;
365:
366:         Matrix Mx = multiply(M, *x, &n2);
367:         n += n2;
368:         *x = subtract(Mx, Db, &n2);
369:         n += n2;
370:
371:         new_residual = infinity_norm(subtract(multiply(A, *x), b));
372:
373:         if (fabs(new_residual) <= fabs(tolerance)) {
374:             break;
375:         }
376:
377:         if (fabs(new_residual) >= fabs(old_residual)) {
378:             ndiverge++;
```

```
379:         if (verbose) {
380:             fprintf(stderr, "Jacobi Iteration residual increased. "
381:                 "(iteration %llu) (repeat %llu)"
382:                 " (by \"ELEM_SCIENT\")\n", i, ndiverge,
383:                 fabs(new_residual) - fabs(old_residual));
384:             fflush(stderr);
385:         }
386:         if (ndiverge >= DIVERGE_DETECT) {
387:             if (verbose)
388:                 fprintf(stderr, "Jacobi Iteration is diverging -- residual"
389:                     " is increasing. Aborting iteration.\n");
390:             break;
391:         }
392:     } else {
393:         if (verbose) {
394:             fprintf(stderr, "%llu:\"ELEM_SCIENT\"\n", i, fabs(new_residual));
395:             fflush(stderr);
396:         }
397:         old_residual = new_residual;
398:         ndiverge = 0;
399:     }
400: }
401:
402: if (niters != NULL)
403:     *niters = i;
404:
405: if (nops != NULL)
406:     *nops = n;
407:
408: return *x;
409: }
410:
411: /*
412:  * Finds the solution x in the equation Ax = b, via Gaussian Elimination (using
413:  * only row operations).
414:  *
415:  * Note that while b is traditionally a column vector, this implementation does
416:  * not care what it is, so long as the number of rows match up.
417:  *
418:  * So to compute the inverse of A, run it with b as the identity matrix with the
419:  * same number of rows as M.
420:  *
421:  * Also, if the ULL pointer *nops is not null, it will be set to the number of
422:  * operations the process took.
423:  */
424: Matrix & gaussian_solve(Matrix & A, Matrix & b, unsigned long long *nops = NULL)
425: {
426:     if (A.shape()[0] != b.shape()[0]) {
427:         printf("Error: attempted to gaussian eliminate on matrices of differing"
428:             " row counts: (%zux%zu and %zux%zu).\n",
429:             A.shape()[0], A.shape()[1], b.shape()[0], b.shape()[1]);
430:         exit(4);
431:     }
432:
433:     size_t dim = A.shape()[0];
434:     size_t b_cols = b.shape()[1];
435:     size_t total_cols = dim + b_cols;
436:
437:     unsigned long long n = 0;
438:
439:     // adjoin the M with b
440:     Matrix adj = A;
441:     Dimension d;
```

```
442:     adj.resize(d[dim][dim + b_cols]);
443:
444:     for (size_t i = 0; i < dim; i++) {
445:         for (size_t j = 0; j < b_cols; j++) {
446:             n++;
447:             adj[i][dim + j] = b[i][j];
448:         }
449:     }
450:
451:     // make upper triangular.
452:     for (size_t i = 0; i < dim - 1; i++) {
453:         ELEM_TYPE pivot = adj[i][i];
454:         for (size_t j = i + 1; j < dim; j++) {
455:             ELEM_TYPE target = adj[j][i];
456:             ELEM_TYPE scale = target / pivot;
457:             if (target == 0.0)
458:                 continue;
459:             // subtract (row i times scale) from row j
460:             for (size_t k = 0; k < total_cols; k++) {
461:                 n++;
462:                 adj[j][k] -= adj[i][k] * scale;
463:             }
464:         }
465:     }
466:
467:     // make diagonal.
468:     for (size_t i = dim - 1; i >= 1; i--) {
469:         ELEM_TYPE pivot = adj[i][i];
470:         if (pivot == 0.0)
471:             continue;
472:         for (size_t j = i - 1; j + 1 >= 1; j--) {
473:             ELEM_TYPE target = adj[j][i];
474:             ELEM_TYPE scale = target / pivot;
475:             if (target == 0.0)
476:                 continue;
477:             // subtract (row i times scale) from row j
478:             for (size_t k = 0; k < total_cols; k++) {
479:                 n++;
480:                 adj[j][k] -= adj[i][k] * scale;
481:             }
482:         }
483:
484:         // divide the rows by the leading variable
485:         if (pivot != 1.0) {
486:             for (size_t k = 0; k < total_cols; k++) {
487:                 n++;
488:                 adj[i][k] /= pivot;
489:             }
490:         }
491:     }
492:
493:     // divide the first row by the leading variable
494:     if (adj[0][0] != 1.0 && adj[0][0] != 0.0) {
495:         ELEM_TYPE scale = adj[0][0];
496:         for (size_t k = 0; k < total_cols; k++) {
497:             n++;
498:             adj[0][k] /= scale;
499:         }
500:     }
501:
502:     // copy the right half of the adjoined matrix (the solution) into a new
503:     // matrix
504:     Matrix *r = new Matrix(d[dim][b_cols]);
```

```
505:     for (size_t i = 0; i < dim; i++) {
506:         for (size_t j = 0; j < b_cols; j++) {
507:             n++;
508:             (*r)[i][j] = adj[i][dim + j];
509:         }
510:     }
511:
512:     if (nops != NULL) {
513:         *nops = n;
514:     }
515:
516:     return *r;
517: }
518:
519: /*
520:  * Reads a matrix from a file.
521:  *
522:  * The format of the file is as follows:
523:  * it must start with two integers, which give the row and column dimensions of
524:  * the matrix.
525:  * Then it must contain the entries of the matrix in row-major order, in any
526:  * format recognizable by scanf(%a). This is just about any float format, with
527:  * or without decimal points, with or without scientific notation, but most
528:  * interestingly, it allows raw float representation, where the number is
529:  * given in hexadecimal and is the exact machine representation of the float.
530:  * Read the man page for scanf(3) for more info.
531:  * The entries may be separated by any amount of whitespace/newlines.
532:  */
533: Matrix & read_matrix(char *filename)
534: {
535:     FILE *input = fopen(filename, "r");
536:     if (input == NULL) {
537:         char buf[512];
538:         snprintf(buf, 512, "fopen(%s)", filename);
539:         perror(buf);
540:         exit(2);
541:     }
542:
543:     size_t rows, cols;
544:     fscanf(input, "%zu %zu", &rows, &cols);
545:
546:     Dimension d;
547:     Matrix *m = new Matrix(d[rows][cols]);
548:
549:     for (size_t i = 0; i < rows; i++) {
550:         for (size_t j = 0; j < cols; j++) {
551:             ELEM_TYPE x;
552:             fscanf(input, ELEM_DUMP, &x);
553:             (*m)[i][j] = x;
554:         }
555:     }
556:
557:     return *m;
558: }
559:
560: int main(int argc, char **argv)
561: {
562:     if (argc < 2) {
563:         usage(argv[0]);
564:     } else if (argc == 2) {
565:         /*
566:          * 1 argument: invert matrix
567:          */
```

```
568:
569: Matrix A = read_matrix(argv[1]);
570:
571: // make the Identity Matrix of appropriate size
572: Dimension d;
573: Matrix I(d[A.shape()[0]][A.shape()[0]]);
574: for (size_t i = 0; i < A.shape()[0]; i++) {
575:     for (size_t j = 0; j < A.shape()[0]; j++) {
576:         if (i == j)
577:             I[i][j] = 1;
578:         else
579:             I[i][j] = 0;
580:     }
581: }
582:
583: Matrix inv = gaussian_solve(A, I, NULL);
584:
585: dump(inv);
586: print(inv);
587:
588: } else {
589:     /*
590:     * 2 arguments: solve system
591:     */
592:
593:     Mode mode;
594:     TestMode testmode;
595:     if (argc < 4) {
596:         mode = GAUSS;
597:         testmode = NOTEST;
598:     } else if (strcmp(argv[3], "gauss") == 0) {
599:         mode = GAUSS;
600:         testmode = NOTEST;
601:     } else if (strcmp(argv[3], "jacobi") == 0) {
602:         mode = JACOBI;
603:         testmode = NOTEST;
604:     } else if (strcmp(argv[3], "gauss_test") == 0) {
605:         mode = GAUSS;
606:         testmode = TEST;
607:     } else if (strcmp(argv[3], "jacobi_test") == 0) {
608:         mode = JACOBI;
609:         testmode = TEST;
610:     } else {
611:         fprintf(stderr, "Unknown mode. Must be either"
612:             " \"gauss\" or \"jacobi\".\n");
613:         exit(1);
614:     }
615:
616:     Matrix A = read_matrix(argv[1]);
617:     Matrix b = read_matrix(argv[2]);
618:
619:     if (A.shape()[0] != A.shape()[1]) {
620:         printf("Sorry, but this program will only work on square matrices."
621:             " :(\n");
622:         exit(3);
623:     }
624:
625:     if (b.shape()[1] != 1) {
626:         printf("Sorry, but this program needs the second matrix to be a"
627:             " column vector. :(\n");
628:         exit(3);
629:     }
630:
```

```
631:     if (A.shape()[0] != b.shape()[0]) {
632:         printf("Sorry, but the row count of the matrix and vector don't"
633:             " match. :(\n");
634:         exit(3);
635:     }
636:
637:     unsigned long long nops;
638:     unsigned long long niters;
639:
640:     Dimension d;
641:     Matrix x(d[A.shape()[0]][1]);
642:
643:     if (mode == GAUSS) {
644:         x = gaussian_solve(A, b, &nops);
645:         niters = 1;
646:     } else {
647:         ELEM_TYPE tolerance;
648:         if (argc < 5)
649:             tolerance = 1.0e-10;
650:         else
651:             sscanf(argv[4], ELEM_DUMP, &tolerance);
652:
653:         bool verb = true;
654:         if (testmode == TEST)
655:             verb = false;
656:         x = jacobi_solve(A, b, tolerance, 0, &nops, &niters, verb);
657:     }
658:
659:     //dump(x);
660:     //print_vector(x, "x");
661:
662:     Matrix Ax = multiply(A, x);
663:     Matrix sub = subtract(Ax, b);
664:     ELEM_TYPE residual = infinity_norm(sub);
665:
666:     double C = nops / pow((double) A.shape()[0], 3.0);
667:
668:     if (testmode == NOTEST) {
669:         dump(x);
670:         print_vector(x, "x");
671:         fprintf(stderr, "Residual = " ELEM_SCIENT "\nTook %llu (%lf * n^3)"
672:             " operations in %llu iterations.\n", residual, nops, C, nite
rs);
673:     } else {
674:         printf("%llu\n", nops);
675:     }
676:
677:
678:     }
679:
680:     return 0;
681: }
682:
683: // vim: sts=4 expandtab
684:
```

```
1: /*
2:  * genmatrix.cpp / Matrix Generator
3:  * Gaussian Elimination vs Jacobi Method Project, MCS 471 F2008
4:  * by William R. Fraser <wfraser@cs.uic.edu> 11/21/2008
5:  */
6:
7: #include <stdio.h>
8: #include <stdlib.h>
9: #include <string.h>
10: #include <math.h>
11: #include <sys/time.h>
12: #include <limits>
13:
14: using std::numeric_limits;
15:
16: #define DOUBLE 1
17: #define FLOAT 2
18:
19: #ifndef TYPE
20: #define TYPE DOUBLE
21: #endif
22:
23: #if TYPE == DOUBLE
24: #   define ELEM_TYPE double
25: #   define ELEM_DUMP "%la"
26: #elif TYPE == FLOAT
27: #   define ELEM_TYPE float
28: #   define ELEM_DUMP "%la"
29: #endif
30:
31: enum Mode {RANDOM, DIAGONALLY_DOMINANT, STRICTLY_DIAGONAL, MANY_ZEROES};
32:
33: int main(int argc, char **argv)
34: {
35:     Mode mode = RANDOM;
36:     ELEM_TYPE zero_point;
37:
38:     if (argc < 3) {
39:         printf("usage: %s <n> <m> [random | diagonally_dominant | strictly_diagonal | many_zeroes <zero_density>]\n"
40: "Generates a square nxn matrix of random numbers for use by gauss.\n"
41: "On x86, the numbers are floats. On AMD64, the numbers are doubles.\n"
42: "The numbers are output in raw float format (see %%a format in printf(3)).\n"
43: "Additionally, the matrix generated is guaranteed to be diagonally dominant.\n"
44: "The diagonal entries are the sum of the row plus 1.\n",
45:             argv[0]);
46:         exit(1);
47:     } else if (argc >= 4) {
48:         if (strcmp(argv[3], "diagonally_dominant") == 0) {
49:             mode = DIAGONALLY_DOMINANT;
50:         } else if (strcmp(argv[3], "strictly_diagonal") == 0) {
51:             mode = STRICTLY_DIAGONAL;
52:         } else if (strcmp(argv[3], "many_zeroes") == 0) {
53:             mode = MANY_ZEROES;
54:             if (argc >= 5)
55:                 sscanf(argv[4], ELEM_DUMP, &zero_point);
56:             else
57:                 zero_point = 0.10;
58:         }
59:     }
60:
61:     timeval tv;
62:     gettimeofday(&tv, NULL);
```

```
63:     srand48(tv.tv_sec);
64:
65:     size_t n = (size_t) atoi(argv[1]);
66:     size_t m = (size_t) atoi(argv[2]);
67:     printf("%zu %zu\n", n, m);
68:
69:     fprintf(stderr, "Making a %zux%zu ", n, m);
70:     switch (mode) {
71: case RANDOM:
72:     fprintf(stderr, "totally random matrix.\n");
73:     break;
74: case DIAGONALLY_DOMINANT:
75:     fprintf(stderr, "strictly diagonally dominant matrix.\n");
76:     break;
77: case STRICTLY_DIAGONAL:
78:     fprintf(stderr, "strictly diagonal matrix.\n");
79:     break;
80: case MANY_ZEROES:
81:     fprintf(stderr, "strictly diagonally dominant matrix "
82:             "with %f%% zeroes.\n", 100.0 * zero_point);
83:     break;
84: }
85:
86: for (size_t i = 0; i < n; i++) {
87:     ELEM_TYPE x;
88:     ELEM_TYPE row_total = 0;
89:     ELEM_TYPE row[m];
90:
91:     for (size_t j = 0; j < m; j++) {
92:         x = (ELEM_TYPE) drand48();
93:
94:         if (mode == STRICTLY_DIAGONAL && j != i) {
95:             row[j] = 0.0;
96:             continue;
97:         }
98:
99:         if (mode == MANY_ZEROES && i != j) {
100:             if (x > (1.0 - zero_point))
101:                 x = 0.0;
102:         }
103:
104:         row[j] = x;
105:         row_total += x;
106:     }
107:
108:     // diagonal is the sum of the row plus a random amount
109:     if ((mode == DIAGONALLY_DOMINANT || mode == MANY_ZEROES) && i < m)
110:         row[i] = row_total;
111:
112:     // print the row
113:     for (size_t j = 0; j < m; j++) {
114:         printf(ELEM_DUMP"\n", row[j]);
115:     }
116: }
117:
118: return 0;
119: }
120:
121: // vim: sts=4 expandtab
122:
```