LIZFCM Software Manual (v0.1)

Elizabeth Hunt

October 13, 2023

Contents

1 Design

The LIZFCM static library (at [https://github.com/Simponic/math-4610 is a successor to my attempt at writing codes for the Fundamentals of Computational Mathematics course in Common Lisp, but the effort required to meet the requirement of creating a static library became too difficult to integrate outside of the `ASDF` solution that Common Lisp already brings to the table.
All of the work established in `deprecated-cl` has been painstakingly translated into the C programming language. I have a couple tenets for its design:

- Implemntations of routines should all be done immutably in respect to arguments.
- Functional programming is good (it's... rough in C though).
- Routines are separated into "module" c files, and not individual files per function.

2 Compilation

A provided `Makefile` is added for convencience. It has been tested on an M1 machine running MacOS as well as Arch Linux.

1. `cd` into the root of the repo
2. `make`

Then, as of homework 4, the testing routine in `test/main.c` can be run via `./dist/lizfcm.test`.
Execution of the Makefile will perform compilation of individual routines.
But, in the requirement of manual intervention (should the little alien workers inside the computer fail to do their job), one can use the following command to produce an object file:

```
gcc -Iinc/ -lm -Wall -c src/<the_routine>.c -o build/<the_routine>.o
```

Which is then bundled into a static library in `lib/lizfcm.a` which can be linked in the standard method.

3 The LIZFCM API

3.1 Simple Routines

3.1.1 `smaceps`

- Author: Elizabeth Hunt
- Name: `smaceps`
- Location: `src/maceps.c`
- Input: none
- Output: a `float` returning the specific "Machine Epsilon" of a machine on a single precision floating point number at which it becomes "indistinguishable".

```
float smaceps() {
  float one = 1.0;
  float machine_epsilon = 1.0;
  float one_approx = one + machine_epsilon;

  while (fabsf(one_approx - one) > 0) {
    machine_epsilon /= 2;
    one_approx = one + machine_epsilon;
  }

  return machine_epsilon;
}
```

3.1.2 `dmaceps`

- Author: Elizabeth Hunt
- Name: `dmaceps`
- Location: `src/maceps.c`
- Input: none
- Output: a `double` returning the specific "Machine Epsilon" of a machine on a double precision floating point number at which it becomes "indistinguishable".

```
double dmaceps() {
  double one = 1.0;
  double machine_epsilon = 1.0;
  double one_approx = one + machine_epsilon;

  while (fabs(one_approx - one) > 0) {
    machine_epsilon /= 2;
    one_approx = one + machine_epsilon;
  }

  return machine_epsilon;
}
```

3.2 Derivative Routines

3.2.1 `central_derivative_at`

- Author: Elizabeth Hunt
- Name: `central_derivative_at`
- Location: `src/approx_derivative.c`
- Input:
  - `f` is a pointer to a one-ary function that takes a double as input and produces a double as output
  - `a` is the domain value at which we approximate `f'`
  - `h` is the step size

- Output: a `double` of the approximate value of `f'(a)` via the central difference method.

```
double central_derivative_at(double (*f)(double), double a, double h) {
  assert(h > 0);

  double x2 = a + h;
  double x1 = a - h;

  double y2 = (*f)(x2);
  double y1 = (*f)(x1);

  return (y2 - y1) / (x2 - x1);
}
```

3.2.2 `forward_derivative_at`

- Author: Elizabeth Hunt
- Name: `forward_derivative_at`
- Location: `src/approx_derivative.c`
- Input:
  - `f` is a pointer to a one-ary function that takes a double as input and produces a double as output
  - `a` is the domain value at which we approximate `f'`
  - `h` is the step size
- Output: a `double` of the approximate value of `f'(a)` via the forward difference method.

```
double forward_derivative_at(double (*f)(double), double a, double h) {
  assert(h > 0);

  double x2 = a + h;
  double x1 = a;

  double y2 = (*f)(x2);
  double y1 = (*f)(x1);

  return (y2 - y1) / (x2 - x1);
}
```

3.2.3 `backward_derivative_at`

- Author: Elizabeth Hunt
- Name: `backward_derivative_at`
- Location: `src/approx_derivative.c`
- Input:
  - `f` is a pointer to a one-ary function that takes a double as input and produces a double as output

  - a is the domain value at which we approximate f'
  - h is the step size

- Output: a double of the approximate value of f'(a) via the backward difference method.

```
double backward_derivative_at(double (*f)(double), double a, double h) {
  assert(h > 0);

  double x2 = a;
  double x1 = a - h;

  double y2 = (*f)(x2);
  double y1 = (*f)(x1);

  return (y2 - y1) / (x2 - x1);
}
```

3.3 Vector Routines

3.3.1 Vector Arithmetic: add_v, minus_v

- Author: Elizabeth Hunt
- Name(s): add_v, minus_v
- Location: src/vector.c
- Input: two pointers to locations in memory wherein Array_double's lie
- Output: a pointer to a new Array_double as the result of addition or subtraction of the two input Array_double's

```
Array_double *add_v(Array_double *v1, Array_double *v2) {
  assert(v1->size == v2->size);

  Array_double *sum = copy_vector(v1);
  for (size_t i = 0; i < v1->size; i++)
    sum->data[i] += v2->data[i];
  return sum;
}

Array_double *minus_v(Array_double *v1, Array_double *v2) {
  assert(v1->size == v2->size);

  Array_double *sub = InitArrayWithSize(double, v1->size, 0);
  for (size_t i = 0; i < v1->size; i++)
    sub->data[i] = v1->data[i] - v2->data[i];
  return sub;
}
```

3.3.2 Norms: l1_norm, l2_norm, linf_norm

- Author: Elizabeth Hunt
- Name(s): l1_norm, l2_norm, linf_norm

- Location: `src/vector.c`
- Input: a pointer to a location in memory wherein an `Array_double` lies
- Output: a `double` representing the value of the norm the function applies

```
double l1_norm(Array_double *v) {
  double sum = 0;
  for (size_t i = 0; i < v->size; ++i)
    sum += fabs(v->data[i]);
  return sum;
}

double l2_norm(Array_double *v) {
  double norm = 0;
  for (size_t i = 0; i < v->size; ++i)
    norm += v->data[i] * v->data[i];
  return sqrt(norm);
}

double linf_norm(Array_double *v) {
  assert(v->size > 0);
  double max = v->data[0];
  for (size_t i = 0; i < v->size; ++i)
    max = c_max(v->data[i], max);
  return max;
}
```

3.3.3 `vector_distance`

- Author: Elizabeth Hunt
- Name: `vector_distance`
- Location: `src/vector.c`
- Input: two pointers to locations in memory wherein `Array_double`'s lie, and a pointer to a one-ary function `norm` taking as input a pointer to an `Array_double` and returning a double representing the norm of that `Array_double`

```
double vector_distance(Array_double *v1, Array_double *v2,
                       double (*norm)(Array_double *)) {
  Array_double *minus = minus_v(v1, v2);
  double dist = (*norm)(minus);
  free(minus);
  return dist;
}
```

3.3.4 Distances: `l1_distance`, `l2_distance`, `linf_distance`

- Author: Elizabeth Hunt
- Name(s): `l1_distance`, `l2_distance`, `linf_distance`
- Location: `src/vector.c`

- Input: two pointers to locations in memory wherein `Array_double`'s lie, and the distance via the corresponding `l1`, `l2`, or `linf` norms
- Output: A `double` representing the distance between the two `Array_doubles`'s by the given norm.

```
double l1_distance(Array_double *v1, Array_double *v2) {
  return vector_distance(v1, v2, &l1_norm);
}

double l2_distance(Array_double *v1, Array_double *v2) {
  return vector_distance(v1, v2, &l2_norm);
}

double linf_distance(Array_double *v1, Array_double *v2) {
  return vector_distance(v1, v2, &linf_norm);
}
```

3.3.5 `sum_v`

- Author: Elizabeth Hunt
- Name: `sum_v`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double`
- Output: a `double` representing the sum of all the elements of an `Array_double`

```
double sum_v(Array_double *v) {
  double sum = 0;
  for (size_t i = 0; i < v->size; i++)
    sum += v->data[i];
  return sum;
}
```

3.3.6 `scale_v`

- Author: Elizabeth Hunt
- Name: `scale_v`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double` and a scalar `double` to scale the vector
- Output: a pointer to a new `Array_double` of the scaled input `Array_double`

```
Array_double *scale_v(Array_double *v, double m) {
  Array_double *copy = copy_vector(v);
  for (size_t i = 0; i < v->size; i++)
    copy->data[i] *= m;
  return copy;
}
```

3.3.7 `free_vector`

- Author: Elizabeth Hunt
- Name: `free_vector`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double`
- Output: nothing.
- Side effect: free the memory of the reserved `Array_double` on the heap

```
void free_vector(Array_double *v) {
  free(v->data);
  free(v);
}
```

3.3.8 `copy_vector`

- Author: Elizabeth Hunt
- Name: `copy_vector`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double`
- Output: a pointer to a new `Array_double` whose `data` and `size` are copied from the input `Array_double`

```
Array_double *copy_vector(Array_double *v) {
  Array_double *copy = InitArrayWithSize(double, v->size, 0.0);
  for (size_t i = 0; i < copy->size; ++i)
    copy->data[i] = v->data[i];
  return copy;
}
```

3.3.9 `format_vector_into`

- Author: Elizabeth Hunt
- Name: `format_vector_into`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double` and a pointer to a c-string `s` to "print" the vector out into
- Output: nothing.
- Side effect: overwritten memory into `s`

```
void format_vector_into(Array_double *v, char *s) {
  if (v->size == 0) {
    strcat(s, "empty");
    return;
  }

  for (size_t i = 0; i < v->size; ++i) {
    char num[64];
    strcpy(num, "");

    sprintf(num, "%f,", v->data[i]);
    strcat(s, num);
  }
  strcat(s, "\n");
}
```

3.4 Matrix Routines

3.4.1 `lu_decomp`

- Author: Elizabeth Hunt
- Name: `lu_decomp`
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` m to decompose into a lower triangular and upper triangular matrix L, U, respectively such that $LU = m$.
- Output: a pointer to the location in memory in which two `Matrix_double`'s reside: the first representing L, the second, U.
- Errors: Exits and throws a status code of `-1` when encountering a matrix that cannot be decomposed

```
Matrix_double **lu_decomp(Matrix_double *m) {
  assert(m->cols == m->rows);

  Matrix_double *u = copy_matrix(m);
  Matrix_double *l_empt = InitMatrixWithSize(double, m->rows, m->cols, 0.0);
  Matrix_double *l = put_identity_diagonal(l_empt);
  free(l_empt);


  Matrix_double **u_l = malloc(sizeof(Matrix_double *) * 2);

  for (size_t y = 0; y < m->rows; y++) {
    if (u->data[y]->data[y] == 0) {
      printf("ERROR: a pivot is zero in given matrix\n");
      exit(-1);
    }
  }

  if (u && l) {
```

```
    for (size_t x = 0; x < m->cols; x++) {
      for (size_t y = x + 1; y < m->rows; y++) {
        double denom = u->data[x]->data[x];

        if (denom == 0) {
          printf("ERROR: non-factorable matrix\n");
          exit(-1);
        }

        double factor = -(u->data[y]->data[x] / denom);

        Array_double *scaled = scale_v(u->data[x], factor);
        Array_double *added = add_v(scaled, u->data[y]);
        free_vector(scaled);
        free_vector(u->data[y]);

        u->data[y] = added;
        l->data[y]->data[x] = -factor;
      }
    }
  }

  u_l[0] = u;
  u_l[1] = l;
  return u_l;
}
```

3.4.2 bsubst

- Author: Elizabeth Hunt
- Name: `bsubst`
- Location: `src/matrix.c`
- Input: a pointer to an upper-triangular `Matrix_double` u and a `Array_double` b
- Output: a pointer to a new `Array_double` whose entries are given by performing back substitution

```
Array_double *bsubst(Matrix_double *u, Array_double *b) {
  assert(u->rows == b->size && u->cols == u->rows);

  Array_double *x = copy_vector(b);
  for (int64_t row = b->size - 1; row >= 0; row--) {
    for (size_t col = b->size - 1; col > row; col--)
      x->data[row] -= x->data[col] * u->data[row]->data[col];
    x->data[row] /= u->data[row]->data[row];
  }
  return x;
}
```

3.4.3 `fsubst`

- Author: Elizabeth Hunt
- Name: `fsubst`
- Location: `src/matrix.c`
- Input: a pointer to a lower-triangular `Matrix_double` l and a `Array_double` b
- Output: a pointer to a new `Array_double` whose entries are given by performing forward substitution

```
Array_double *fsubst(Matrix_double *l, Array_double *b) {
  assert(l->rows == b->size && l->cols == l->rows);

  Array_double *x = copy_vector(b);

  for (size_t row = 0; row < b->size; row++) {
    for (size_t col = 0; col < row; col++)
      x->data[row] -= x->data[col] * l->data[row]->data[col];
    x->data[row] /= l->data[row]->data[row];
  }

  return x;
}
```

3.4.4 `solve_matrix`

- Author: Elizabeth Hunt
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` m and a pointer to an `Array_double` b
- Output: x such that $mx = b$ if such a solution exists (else it's non LU-factorable as discussed above)

Here we make use of forward substitution to first solve $Ly = b$ given L as the L factor in `lu_decomp`. Then we use back substitution to solve $Ux = y$ for x similarly given U.
Then, $LUx = b$, thus x is a solution.

```
Array_double *solve_matrix(Matrix_double *m, Array_double *b) {
  assert(b->size == m->rows);
  assert(m->rows == m->cols);

  Array_double *x = copy_vector(b);
  Matrix_double **u_l = lu_decomp(m);
  Matrix_double *u = u_l[0];
  Matrix_double *l = u_l[1];

  Array_double *b_fsub = fsubst(l, b);
  x = bsubst(u, b_fsub);
  free_vector(b_fsub);
```

```
  free_matrix(u);
  free_matrix(l);

  return x;
}
```

3.4.5 m_dot_v

- Author: Elizabeth Hunt
- Location: src/matrix.c
- Input: a pointer to a Matrix_double m and Array_double v
- Output: the dot product mv as an Array_double

```
Array_double *m_dot_v(Matrix_double *m, Array_double *v) {
  assert(v->size == m->cols);

  Array_double *product = copy_vector(v);

  for (size_t row = 0; row < v->size; ++row)
    product->data[row] = v_dot_v(m->data[row], v);

  return product;
}
```

3.4.6 put_identity_diagonal

- Author: Elizabeth Hunt
- Location: src/matrix.c
- Input: a pointer to a Matrix_double
- Output: a pointer to a copy to Matrix_double whose diagonal is full of 1's

```
Matrix_double *put_identity_diagonal(Matrix_double *m) {
  assert(m->rows == m->cols);
  Matrix_double *copy = copy_matrix(m);
  for (size_t y = 0; y < m->rows; ++y)
    copy->data[y]->data[y] = 1.0;
  return copy;
}
```

3.4.7 copy_matrix

- Author: Elizabeth Hunt
- Location: src/matrix.c
- Input: a pointer to a Matrix_double
- Output: a pointer to a copy of the given Matrix_double

```c
Matrix_double *copy_matrix(Matrix_double *m) {
  Matrix_double *copy = InitMatrixWithSize(double, m->rows, m->cols, 0.0);
  for (size_t y = 0; y < copy->rows; y++) {
    free_vector(copy->data[y]);
    copy->data[y] = copy_vector(m->data[y]);
  }
  return copy;
}
```

3.4.8 `free_matrix`

- Author: Elizabeth Hunt
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double`
- Output: none.
- Side Effects: frees memory reserved by a given `Matrix_double` and its member `Array_double` vectors describing its rows.

```c
void free_matrix(Matrix_double *m) {
  for (size_t y = 0; y < m->rows; ++y)
    free_vector(m->data[y]);
  free(m);
}
```

3.4.9 `format_matrix_into`

- Author: Elizabeth Hunt
- Name: `format_matrix_into`
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` and a pointer to a c-string `s` to "print" the vector out into
- Output: nothing.
- Side effect: overwritten memory into `s`

```c
void format_matrix_into(Matrix_double *m, char *s) {
  if (m->rows == 0)
    strcpy(s, "empty");

  for (size_t y = 0; y < m->rows; ++y) {
    char row_s[256];
    strcpy(row_s, "");

    format_vector_into(m->data[y], row_s);
    strcat(s, row_s);
  }
  strcat(s, "\n");
}
```

3.5 Linear Routines

3.5.1 least_squares_lin_reg

- Author: Elizabeth Hunt
- Name: least_squares_lin_reg
- Location: src/lin.c
- Input: two pointers to Array_double's whose entries correspond two ordered pairs in $\mathbb{R}^2$
- Output: a linear model best representing the ordered pairs via least squares regression

```
Line *least_squares_lin_reg(Array_double *x, Array_double *y) {
  assert(x->size == y->size);

  uint64_t n = x->size;
  double sum_x = sum_v(x);
  double sum_y = sum_v(y);
  double sum_xy = v_dot_v(x, y);
  double sum_xx = v_dot_v(x, x);
  double denom = ((n * sum_xx) - (sum_x * sum_x));

  Line *line = malloc(sizeof(Line));
  line->m = ((sum_xy * n) - (sum_x * sum_y)) / denom;
  line->a = ((sum_y * sum_xx) - (sum_x * sum_xy)) / denom;

  return line;
}
```

3.6 Appendix / Miscellaneous

3.6.1 Data Types

1. Line

   - Author: Elizabeth Hunt
   - Location: inc/types.h

   ```
   typedef struct Line {
     double m;
     double a;
   } Line;
   ```

2. The Array_<type> and Matrix_<type>

   - Author: Elizabeth Hunt
   - Location: inc/types.h

   We define two Pre processor Macros DEFINE_ARRAY and DEFINE_MATRIX that take as input a type, and construct a struct definition for the given type for convenient access to the vector or matrices dimensions.

   Such that DEFINE_ARRAY(int) would expand to:

```
typedef struct {
  int* data;
  size_t size;
} Array_int
```

And `DEFINE_MATRIX(int)` would expand a to `Matrix_int`; containing a pointer to a collection of pointers of `Array_int`'s and its dimensions.

```
typedef struct {
  Array_int **data;
  size_t cols;
  size_t rows;
} Matrix_int
```

3.6.2 Macros

1. `c_max` and `c_min`

   - Author: Elizabeth Hunt
   - Location: `inc/macros.h`
   - Input: two structures that define an order measure
   - Output: either the larger or smaller of the two depending on the measure

```
#define c_max(x, y) (((x) >= (y)) ? (x) : (y))
#define c_min(x, y) (((x) <= (y)) ? (x) : (y))
```

2. `InitArray`

   - Author: Elizabeth Hunt
   - Location: `inc/macros.h`
   - Input: a type and array of values to initialze an array with such type
   - Output: a new `Array_type` with the size of the given array and its data

```
#define InitArray(TYPE, ...)                                              \
  ({                                                                      \
    TYPE temp[] = __VA_ARGS__;                                            \
    Array_##TYPE *arr = malloc(sizeof(Array_##TYPE));                     \
    arr->size = sizeof(temp) / sizeof(temp[0]);                           \
    arr->data = malloc(arr->size * sizeof(TYPE));                         \
    memcpy(arr->data, temp, arr->size * sizeof(TYPE));                    \
    arr;                                                                  \
  })
```

3. `InitArrayWithSize`

   - Author: Elizabeth Hunt
   - Location: `inc/macros.h`
   - Input: a type, a size, and initial value
   - Output: a new `Array_type` with the given size filled with the initial value

```c
#define InitArrayWithSize(TYPE, SIZE, INIT_VALUE)                            \
  ({                                                                         \
    Array_##TYPE *arr = malloc(sizeof(Array_##TYPE));                        \
    arr->size = SIZE;                                                        \
    arr->data = malloc(arr->size * sizeof(TYPE));                            \
    for (size_t i = 0; i < arr->size; i++)                                   \
      arr->data[i] = INIT_VALUE;                                             \
    arr;                                                                     \
  })
```

4. `InitMatrixWithSize`

   - Author: Elizabeth Hunt
   - Location: `inc/macros.h`
   - Input: a type, number of rows, columns, and initial value
   - Output: a new `Matrix_type` of size `rows x columns` filled with the initial value

```c
#define InitMatrixWithSize(TYPE, ROWS, COLS, INIT_VALUE)                     \
  ({                                                                         \
    Matrix_##TYPE *matrix = malloc(sizeof(Matrix_##TYPE));                   \
    matrix->rows = ROWS;                                                     \
    matrix->cols = COLS;                                                     \
    matrix->data = malloc(matrix->rows * sizeof(Array_##TYPE *));            \
    for (size_t y = 0; y < matrix->rows; y++)                                \
      matrix->data[y] = InitArrayWithSize(TYPE, COLS, INIT_VALUE);           \
    matrix;                                                                  \
  })
```