LIZFCM Software Manual (v0.6)

Elizabeth Hunt

December 11, 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:

- Implementations 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 "modules" that follow a form of separation of concerns in files, and not individual files per function.

2 Compilation

A provided `Makefile` is added for convencience. It has been tested on an `arm`-based M1 machine running MacOS as well as `x86` Arch Linux.

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

Then, as of homework 5, the testing routines are provided in `test` and utilize the `utest` "micro"library. They compile to a binary in `./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` and 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

```c
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`

```c
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

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

3.3.8 add_element

- Author: Elizabeth Hunt
- Name: `add_element`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double`
- Output: a new `Array_double` with element `x` appended.

```c
Array_double *add_element(Array_double *v, double x) {
  Array_double *pushed = InitArrayWithSize(double, v->size + 1, 0.0);
  for (size_t i = 0; i < v->size; ++i)
    pushed->data[i] = v->data[i];
  pushed->data[v->size] = x;
  return pushed;
}
```

3.3.9 slice_element

- Author: Elizabeth Hunt
- Name: `slice_element`
- Location: `src/vector.c`
- Input: a pointer to an `Array_double`
- Output: a new `Array_double` with element `x` sliced.

```c
Array_double *slice_element(Array_double *v, size_t x) {
  Array_double *sliced = InitArrayWithSize(double, v->size - 1, 0.0);
  for (size_t i = 0; i < v->size - 1; ++i)
    sliced->data[i] = i >= x ? v->data[i + 1] : v->data[i];
  return sliced;
}
```

3.3.10 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`

```c
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;
}
```

3.3.11 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: Fails assertions 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_matrix(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");
      assert(false);
    }
  }
```

```c
  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");
          assert(false);
        }

        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

```c
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_lu_bsubst`

- 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_lu_bsubst(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);
  free(u_l);

  return x;
}
```

3.4.5 gaussian_elimination

- Author: Elizabeth Hunt
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` m
- Output: a pointer to a copy of m in reduced echelon form

This works by finding the row with a maximum value in the column k. Then, it uses that as a pivot, and applying reduction to all other rows. The general idea is available at `https://en.wikipedia.org/wiki/Gaussian_elimination`.

```
Matrix_double *gaussian_elimination(Matrix_double *m) {
  uint64_t h = 0, k = 0;

  Matrix_double *m_cp = copy_matrix(m);

  while (h < m_cp->rows && k < m_cp->cols) {
    uint64_t max_row = h;
    double max_val = 0.0;

    for (uint64_t row = h; row < m_cp->rows; row++) {
      double val = fabs(m_cp->data[row]->data[k]);
      if (val > max_val) {
        max_val = val;
        max_row = row;
      }
    }

    if (max_val == 0.0) {
      k++;
      continue;
    }

    if (max_row != h) {
      Array_double *swp = m_cp->data[max_row];
      m_cp->data[max_row] = m_cp->data[h];
      m_cp->data[h] = swp;
    }

    for (uint64_t row = h + 1; row < m_cp->rows; row++) {
      double factor = m_cp->data[row]->data[k] / m_cp->data[h]->data[k];
      m_cp->data[row]->data[k] = 0.0;
```

```
      for (uint64_t col = k + 1; col < m_cp->cols; col++) {
        m_cp->data[row]->data[col] -= m_cp->data[h]->data[col] * factor;
      }
    }

    h++;
    k++;
  }

  return m_cp;
}
```

3.4.6 `solve_matrix_gaussian`

- Author: Elizabeth Hunt
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` m and a target `Array_double` b
- Output: a pointer to a vector x being the solution to the equation $mx = b$

We first perform `gaussian_elimination` after augmenting m and b. Then, as m is in reduced echelon form, it's an upper triangular matrix, so we can perform back substitution to compute x.

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

  Matrix_double *m_augment_b = add_column(m, b);
  Matrix_double *eliminated = gaussian_elimination(m_augment_b);

  Array_double *b_gauss = col_v(eliminated, m->cols);
  Matrix_double *u = slice_column(eliminated, m->rows);

  Array_double *solution = bsubst(u, b_gauss);

  free_matrix(m_augment_b);
  free_matrix(eliminated);
  free_matrix(u);
  free_vector(b_gauss);

  return solution;
}
```

3.4.7 `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;
}
```

3.4.8 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;
}
```

3.4.9 slice_column

- 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 with column at x sliced

```
Matrix_double *slice_column(Matrix_double *m, size_t x) {
  Matrix_double *sliced = copy_matrix(m);

  for (size_t row = 0; row < m->rows; row++) {
    Array_double *old_row = sliced->data[row];
    sliced->data[row] = slice_element(old_row, x);
    free_vector(old_row);
  }
  sliced->cols--;

  return sliced;
}
```

3.4.10 `add_column`

- Author: Elizabet Hunt
- Location: `src/matrix.c`
- Input: a pointer to a `Matrix_double` and a new vector representing the appended column `x`
- Output: a pointer to a copy of the given `Matrix_double` with a new column `x`

```
Matrix_double *add_column(Matrix_double *m, Array_double *v) {
  Matrix_double *pushed = copy_matrix(m);

  for (size_t row = 0; row < m->rows; row++) {
    Array_double *old_row = pushed->data[row];
    pushed->data[row] = add_element(old_row, v->data[row]);
    free_vector(old_row);
  }

  pushed->cols++;
  return pushed;
}
```

3.4.11 `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`

```
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;
}
```

3.4.12 `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.

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

3.4.13 `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`

```
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[5192];
    strcpy(row_s, "");

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

3.5 Root Finding Methods

3.5.1 `find_ivt_range`

- Author: Elizabeth Hunt
- Name: `find_ivt_range`
- Location: `src/roots.c`
- Input: a pointer to a oneary function taking a double and producing a double, the beginning point in R to search for a range, a `delta` step that is taken, and a `max_steps` number of maximum iterations to perform.
- Output: a pair of `double`'s in an `Array_double` representing a closed closed interval `[beginning, end]`

```
// f is well defined at start_x + delta*n for all n on the integer range [0,
// max_iterations]
Array_double *find_ivt_range(double (*f)(double), double start_x, double delta,
                             size_t max_iterations) {
```

```
  double a = start_x;

  while (f(a) * f(a + delta) >= 0 && max_iterations > 0) {
    max_iterations--;
    a += delta;
  }

  double end = a + delta;
  double begin = a - delta;

  if (max_iterations == 0 && f(begin) * f(end) >= 0)
    return NULL;
  return InitArray(double, {begin, end});
}
```

3.5.2 `bisect_find_root`

- Author: Elizabeth Hunt
- Name(s): `bisect_find_root`
- Input: a one-ary function taking a double and producing a double, a closed interval represented by `a` and `b`: `[a, b]`, a `tolerance` at which we return the estimated root once $b - a <$ tolerance, and a `max_iterations` to break us out of a loop if we can never reach the `tolerance`.
- Output: a vector of size of 3, `double`'s representing first the range `[a,b]` and then the midpoint, `c` of the range.
- Description: recursively uses binary search to split the interval until we reach `tolerance`. We also assume the function `f` is continuous on `[a, b]`.

```
// f is continuous on [a, b]
Array_double *bisect_find_root(double (*f)(double), double a, double b,
                               double tolerance, size_t max_iterations) {
  assert(a <= b);
  // guarantee there's a root somewhere between a and b by IVT
  assert(f(a) * f(b) < 0);

  double c = (1.0 / 2) * (a + b);
  if (b - a < tolerance || max_iterations == 0)
    return InitArray(double, {a, b, c});

  if (f(a) * f(c) < 0)
    return bisect_find_root(f, a, c, tolerance, max_iterations - 1);
  return bisect_find_root(f, c, b, tolerance, max_iterations - 1);
}
```

3.5.3 `bisect_find_root_with_error_assumption`

- Author: Elizabeth Hunt
- Name: `bisect_find_root_with_error_assumption`

- Input: a one-ary function taking a double and producing a double, a closed interval represented by a and b: [a, b], and a tolerance equivalent to the above definition in bisect_find_root
- Output: a double representing the estimated root
- Description: using the bisection method we know that $e_k \leq (\frac{1}{2})^k(b_0 - a_0)$. So we can calculate k at the worst possible case (that the error is exactly the tolerance) to be $\frac{log(tolerance) - log(b_0 - a_0)}{log(\frac{1}{2})}$. We pass this value into the max_iterations of bisect_find_root as above.

```
double bisect_find_root_with_error_assumption(double (*f)(double), double a,
                                              double b, double tolerance) {
  assert(a <= b);

  uint64_t max_iterations =
      (uint64_t)ceil((log(tolerance) - log(b - a)) / log(1 / 2.0));

  Array_double *a_b_root = bisect_find_root(f, a, b, tolerance, max_iterations);
  double root = a_b_root->data[2];
  free_vector(a_b_root);

  return root;
}
```

3.5.4 fixed_point_iteration_method

- Author: Elizabeth Hunt
- Name: fixed_point_iteration_method
- Location: src/roots.c
- Input: a pointer to a oneary function f taking a double and producing a double of which we are trying to find a root, a guess x_0, and a function g of the same signature of f at which we "step" our guesses according to the fixed point iteration method: $x_k = g(x_{k-1})$. Additionally, a max_iterations representing the maximum number of "steps" to take before arriving at our approximation and a tolerance to return our root if it becomes within [0 - tolerance, 0 + tolerance].
- Assumptions: $g(x)$ must be a function such that at the point x^* (the found root) the derivative $|g'(x^*)|1$
- Output: a double representing the found approximate root $\approx x^*$.

```
double fixed_point_iteration_method(double (*f)(double), double (*g)(double),
                                    double x_0, double tolerance,
                                    size_t max_iterations) {
  if (max_iterations <= 0)
    return x_0;

  double root = g(x_0);
  if (tolerance >= fabs(f(root)))
    return root;
```

```
  return fixed_point_iteration_method(f, g, root, tolerance,
                                      max_iterations - 1);
}
```

3.5.5 `fixed_point_newton_method`

- Author: Elizabeth Hunt
- Name: `fixed_point_newton_method`
- Location: `src/roots.c`
- Input: a pointer to a oneary function f taking a double and producing a double of which we are trying to find a root and another pointer to a function fprime of the same signature, a guess x_0, and a `max_iterations` and `tolerance` as defined in the above method are required inputs.
- Description: continually computes elements in the sequence $x_n = x_{n-1} - \frac{f(x_{n-1})}{f'p(x_{n-1})}$
- Output: a double representing the found approximate root $\approx x^*$ recursively applied to the sequence given

```
double fixed_point_newton_method(double (*f)(double), double (*fprime)(double),
                                 double x_0, double tolerance,
                                 size_t max_iterations) {
  if (max_iterations <= 0)
    return x_0;

  double root = x_0 - f(x_0) / fprime(x_0);
  if (tolerance >= fabs(f(root)))
    return root;

  return fixed_point_newton_method(f, fprime, root, tolerance,
                                   max_iterations - 1);
}
```

3.5.6 `fixed_point_secant_method`

- Author: Elizabeth Hunt
- Name: `fixed_point_secant_method`
- Location: `src/roots.c`
- Input: a pointer to a oneary function f taking a double and producing a double of which we are trying to find a root, a guess x_0 and x_1 in which a root lies between $[x_0, x_1]$; applying the sequence $x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$. Additionally, a `max_iterations` and `tolerance` as defined in the above method are required inputs.
- Output: a double representing the found approximate root $\approx x^*$ recursively applied to the sequence.

```
double fixed_point_secant_method(double (*f)(double), double x_0, double x_1,
                                 double tolerance, size_t max_iterations) {
  if (max_iterations == 0)
```

```
    return x_1;

  double root = x_1 - f(x_1) * ((x_1 - x_0) / (f(x_1) - f(x_0)));

  if (tolerance >= fabs(f(root)))
    return root;

  return fixed_point_secant_method(f, x_1, root, tolerance, max_iterations - 1);
}
```

3.5.7 fixed_point_secant_bisection_method

- Author: Elizabeth Hunt
- Name: fixed_point_secant_method
- Location: src/roots.c
- Input: a pointer to a oneary function f taking a double and producing a double of which we are trying to find a root, a guess x_0, and a x_1 of which we define our first interval $[x_0, x_1]$. Then, we perform a single iteration of the fixed_point_secant_method on this interval; if it produces a root outside, we refresh the interval and root respectively with the given bisect_find_root method. Additionally, a max_iterations and tolerance as defined in the above method are required inputs.
- Output: a double representing the found approximate root $\approx x^*$ continually applied with the constraints defined.

```
double fixed_point_secant_bisection_method(double (*f)(double), double x_0,
                                           double x_1, double tolerance,
                                           size_t max_iterations) {
  double begin = x_0;
  double end = x_1;
  double root = x_0;

  while (tolerance < fabs(f(root)) && max_iterations > 0) {
    max_iterations--;

    double secant_root = fixed_point_secant_method(f, begin, end, tolerance, 1);

    if (secant_root < begin || secant_root > end) {
      Array_double *range_root = bisect_find_root(f, begin, end, tolerance, 1);

      begin = range_root->data[0];
      end = range_root->data[1];
      root = range_root->data[2];

      free_vector(range_root);
      continue;
    }

    root = secant_root;
```

```
    if (f(root) * f(begin) < 0)
      end = secant_root; // the root exists in [begin, secant_root]
    else
      begin = secant_root;
  }

  return root;
}
```

3.6 Linear Routines

3.6.1 `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 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;
}
```

3.7 Eigen-Adjacent

3.7.1 `dominant_eigenvalue`

- Author: Elizabeth Hunt
- Name: `dominant_eigenvalue`
- Location: `src/eigen.c`
- Input: a pointer to an invertible matrix `m`, an initial eigenvector guess `v` (that is non zero or orthogonal to an eigenvector with the dominant eigenvalue), a `tolerance` and `max_iterations` that act as stop conditions
- Output: the dominant eigenvalue with the highest magnitude, approximated with the Power Iteration Method

```
double dominant_eigenvalue(Matrix_double *m, Array_double *v, double tolerance,
                           size_t max_iterations) {
  assert(m->rows == m->cols);
  assert(m->rows == v->size);

  double error = tolerance;
  size_t iter = max_iterations;
  double lambda = 0.0;
  Array_double *eigenvector_1 = copy_vector(v);

  while (error >= tolerance && (--iter) > 0) {
    Array_double *eigenvector_2 = m_dot_v(m, eigenvector_1);
    Array_double *normalized_eigenvector_2 =
        scale_v(eigenvector_2, 1.0 / linf_norm(eigenvector_2));
    free_vector(eigenvector_2);
    eigenvector_2 = normalized_eigenvector_2;

    Array_double *mx = m_dot_v(m, eigenvector_2);
    double new_lambda =
        v_dot_v(mx, eigenvector_2) / v_dot_v(eigenvector_2, eigenvector_2);

    error = fabs(new_lambda - lambda);
    lambda = new_lambda;
    free_vector(eigenvector_1);
    eigenvector_1 = eigenvector_2;
  }

  return lambda;
}
```

3.7.2 `shift_inverse_power_eigenvalue`

- Author: Elizabeth Hunt
- Name: `least_dominant_eigenvalue`
- Location: `src/eigen.c`
- Input: a pointer to an invertible matrix `m`, an initial eigenvector guess `v` (that is non zero or orthogonal to an eigenvector with the dominant eigenvalue), a `shift` to act as the shifted δ, and `tolerance` and `max_iterations` that act as stop conditions.
- Output: the eigenvalue closest to `shift` with the lowest magnitude closest to 0, approximated with the Inverse Power Iteration Method

```
double shift_inverse_power_eigenvalue(Matrix_double *m, Array_double *v,
                                      double shift, double tolerance,
                                      size_t max_iterations) {
  assert(m->rows == m->cols);
  assert(m->rows == v->size);

  Matrix_double *m_c = copy_matrix(m);
  for (size_t y = 0; y < m_c->rows; ++y)
```

```
    m_c->data[y]->data[y] = m_c->data[y]->data[y] - shift;

  double error = tolerance;
  size_t iter = max_iterations;
  double lambda = shift;
  Array_double *eigenvector_1 = copy_vector(v);

  while (error >= tolerance && (--iter) > 0) {
    Array_double *eigenvector_2 = solve_matrix_lu_bsubst(m_c, eigenvector_1);
    Array_double *normalized_eigenvector_2 =
        scale_v(eigenvector_2, 1.0 / linf_norm(eigenvector_2));
    free_vector(eigenvector_2);

    Array_double *mx = m_dot_v(m, normalized_eigenvector_2);
    double new_lambda =
        v_dot_v(mx, normalized_eigenvector_2) /
        v_dot_v(normalized_eigenvector_2, normalized_eigenvector_2);

    error = fabs(new_lambda - lambda);
    lambda = new_lambda;
    free_vector(eigenvector_1);
    eigenvector_1 = normalized_eigenvector_2;
  }

  return lambda;
}
```

3.7.3 `least_dominant_eigenvalue`

- Author: Elizabeth Hunt
- Name: `least_dominant_eigenvalue`
- Location: `src/eigen.c`
- Input: a pointer to an invertible matrix `m`, an initial eigenvector guess `v` (that is non zero or orthogonal to an eigenvector with the dominant eigenvalue), a `tolerance` and `max_iterations` that act as stop conditions.
- Output: the least dominant eigenvalue with the lowest magnitude closest to 0, approximated with the Inverse Power Iteration Method.

```
double least_dominant_eigenvalue(Matrix_double *m, Array_double *v,
                                 double tolerance, size_t max_iterations) {
  return shift_inverse_power_eigenvalue(m, v, 0.0, tolerance, max_iterations);
}
```

3.7.4 `partition_find_eigenvalues`

- Author: Elizabeth Hunt
- Name: `partition_find_eigenvalues`
- Location: `src/eigen.c`

- Input: a pointer to an invertible matrix `m`, a matrix whose rows correspond to initial eigenvector guesses at each "partition" which is computed from a uniform distribution between the number of rows this "guess matrix" has and the distance between the least dominant eigenvalue and the most dominant. Additionally, a `max_iterations` and a `tolerance` that act as stop conditions.
- Output: a vector of `doubles` corresponding to the "nearest" eigenvalue at the midpoint of each partition, via the given guess of that partition.

```
Array_double *partition_find_eigenvalues(Matrix_double *m,
                                         Matrix_double *guesses,
                                         double tolerance,
                                         size_t max_iterations) {
  assert(guesses->rows >=
         2); // we need at least, the most and least dominant eigenvalues

  double end = dominant_eigenvalue(m, guesses->data[guesses->rows - 1],
                                   tolerance, max_iterations);
  double begin =
      least_dominant_eigenvalue(m, guesses->data[0], tolerance, max_iterations);

  double delta = (end - begin) / guesses->rows;
  Array_double *eigenvalues = InitArrayWithSize(double, guesses->rows, 0.0);
  for (size_t i = 0; i < guesses->rows; i++) {
    double box_midpoint = ((delta * i) + (delta * (i + 1))) / 2;

    double nearest_eigenvalue = shift_inverse_power_eigenvalue(
        m, guesses->data[i], box_midpoint, tolerance, max_iterations);

    eigenvalues->data[i] = nearest_eigenvalue;
  }

  return eigenvalues;
}
```

3.7.5 `leslie_matrix`

- Author: Elizabeth Hunt
- Name: `leslie_matrix`
- Location: `src/eigen.c`
- Input: two pointers to `Array_double`'s representing the ratio of individuals in an age class x getting to the next age class $x + 1$ and the number of offspring that individuals in an age class create in age class 0.
- Output: the leslie matrix generated from the input vectors.

```
Matrix_double *leslie_matrix(Array_double *age_class_surivor_ratio,
                             Array_double *age_class_offspring) {
  assert(age_class_surivor_ratio->size + 1 == age_class_offspring->size);

  Matrix_double *leslie = InitMatrixWithSize(double, age_class_offspring->size,
```

```
                                                     age_class_offspring->size, 0.0);

  free_vector(leslie->data[0]);
  leslie->data[0] = age_class_offspring;

  for (size_t i = 0; i < age_class_surivor_ratio->size; i++)
    leslie->data[i + 1]->data[i] = age_class_surivor_ratio->data[i];
  return leslie;
}
```

3.8 Jacobi / Gauss-Siedel

3.8.1 jacobi_solve

- Author: Elizabeth Hunt
- Name: jacobi_solve
- Location: src/matrix.c
- Input: a pointer to a diagonally dominant square matrix m, a vector representing the value b in $mx = b$, a double representing the maximum distance between the solutions produced by iteration i and $i+1$ (by L2 norm a.k.a cartesian distance), and a max_iterations which we force stop.
- Output: the converged-upon solution x to $mx = b$

```
Array_double *jacobi_solve(Matrix_double *m, Array_double *b,
                           double l2_convergence_tolerance,
                           size_t max_iterations) {
  assert(m->rows == m->cols);
  assert(b->size == m->cols);
  size_t iter = max_iterations;

  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 && l2_distance(x_k_1, x_k) > l2_convergence_tolerance) {
    for (size_t i = 0; i < m->rows; i++) {
      double delta = 0.0;
      for (size_t j = 0; j < m->cols; j++) {
        if (i == j)
          continue;
        delta += m->data[i]->data[j] * x_k->data[j];
      }
      x_k_1->data[i] = (b->data[i] - delta) / m->data[i]->data[i];
    }

    Array_double *tmp = x_k;
    x_k = x_k_1;
    x_k_1 = tmp;
  }
```

```
  free_vector(x_k);
  return x_k_1;
}
```

3.8.2 `gauss_siedel_solve`

- Author: Elizabeth Hunt
- Name: `gauss_siedel_solve`
- Location: `src/matrix.c`
- Input: a pointer to a diagonally dominant or symmetric and positive definite square matrix m, a vector representing the value b in $mx = b$, a double representing the maximum distance between the solutions produced by iteration i and $i+1$ (by L2 norm a.k.a cartesian distance), and a `max_iterations` which we force stop.
- Output: the converged-upon solution x to $mx = b$
- Description: we use almost the exact same method as `jacobi_solve` but modify only one array in accordance to the Gauss-Siedel method, but which is necessarily copied before due to the convergence check.

```
Array_double *gauss_siedel_solve(Matrix_double *m, Array_double *b,
                                 double l2_convergence_tolerance,
                                 size_t max_iterations) {
  assert(m->rows == m->cols);
  assert(b->size == m->cols);
  size_t iter = max_iterations;

  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) {
    for (size_t i = 0; i < x_k->size; i++)
      x_k->data[i] = x_k_1->data[i];

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

    if (l2_distance(x_k_1, x_k) <= l2_convergence_tolerance)
      break;
  }

  free_vector(x_k);
```

```
  return x_k_1;
}
```

3.9 Appendix / Miscellaneous

3.9.1 Random

- Author: Elizabeth Hunt
- Name: `rand_from`
- Location: `src/rand.c`
- Input: a pair of doubles, min and max to generate a random number min $\leq$ x $\leq$ max
- Output: a random double in the constraints shown

```
double rand_from(double min, double max) {
  return min + (rand() / (RAND_MAX / (max - min)));
}
```

3.9.2 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
   ```

3.9.3 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

```
#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

```
#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;                                                                 \
  })
```