MIDvec Doc, v1.0.3

Introduction:

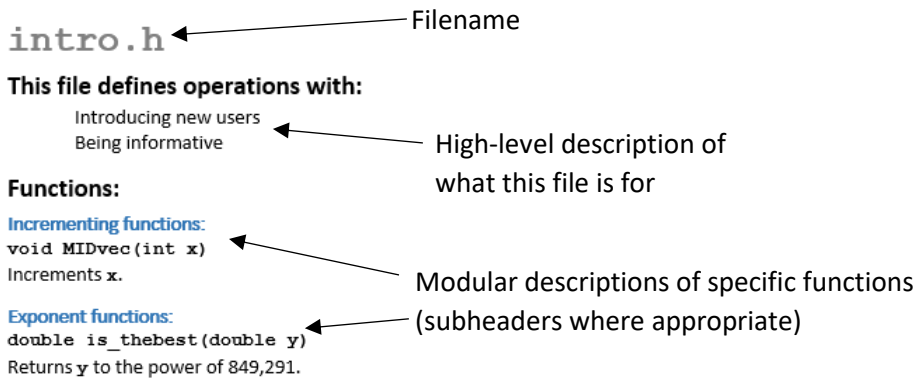
In general, how MIDvec works:

Run MIDvec by writing code in the `main()` function of `midvec.c`, (or writing your own .c file which includes the library's .h files) then compiling and executing `midvec.c`. Use any output from MIDvec as you will.

How to approach this guide:

This guide is organized primarily by the names of the files in the MIDvec package, going roughly in order of most fundamental to least. Under each filename heading is a listing on what the file defines, followed by modular descriptions of the functions within said file.

A typical structure for a file description in this guide generally looks like the following:



With that said, thank you for considering or purchasing MIDvec.

midvec.c

This is the file one compiles to run MIDvec.
In the `main()` function for this file, write the code to be executed.

cpx_vec.h

This file defines operations with:

Complex numbers: Construction and arithmetic
Vectors: Storing primitives and complex numbers, menial arithmetic and operations

Objects:

MIDvec abstracts array creation with structs named **xvec**.
Each **vec** struct has a pointer and a length, eg:

```
typedef struct bvec {
  unsigned char* arr;
  int len;
} bvec_o;
```

> Tip: The nomenclature here goes backwards. eg, `bvec` is a *vector* of *bytes*. `cpx_vec` is a *vector* of *complex numbers*.

`xvec` structs are named as follows:

bvec: Vector of unsigned bytes
ivec: Vector of signed `int`
dvec: Vector of `double`
cpx_vec: Vector of complex numbers

MIDvec also abstracts holding *lists of* vectors of primitives.
Each **xvec_list** has a pointer to a vec and a length, eg:

```
typedef struct cpx_vec_list {
  cpx_vec_o* arr;
  int len;
} cpx_vec_list_o;
```

> Tip: Again, backwards naming, eg. `bvec_list` is a *list* of *vector*s of *bytes*.

bvec_list: List of unsigned `char` vectors
ivec_list: List of `int` vectors
dvec_list: List of `double` vectors
cpx_vec_list: List of complex number vectors

Constructors:

`cpx_o new_cpx_polar(double mag, double phase)`
Returns complex number with magnitude **mag** and phase **phase**.

`cpx_o new_cpx_rect(double r, double i)`
Returns complex number with real component **r** and imaginary component **i**.

`xvec_o new_xvec(int length)`
Returns new vector of type **x** ∈ [**b**, **i**, **v**, **cpx_**].

`bvec_list_o new_bvec_list(int num_vecs, int vec_length)`
Returns new list of vectors of type **x** ∈ [**b**, **i**, **v**, **cpx_**].

`dvec_o cpx_to_vec(cpx_o z)`
Takes complex number and outputs vector **`v`** where **`v[0]`** = **`z.real`** and **`v[1]`** = **`z.imag`**. **`v`** will be two elements long exactly.

`cpx_o vec_to_cpx(dvec_o v)`
Takes vector **`v`** and outputs complex number such that **`z.real`** = **`v[0]`** and **`z.imag`** = **`v[1]`**. **`v`** need not be 2 elements long.

Functions - Complex number operations:

Gathering information on a complex number:

`double mag_cpx(cpx_o a)`
Find the magnitude of a complex number **`a`**.

`double mag2_cpx(cpx_o a)`
Returns magnitude squared of complex number **`a`**. Saves step of square rooting.

`double phase_cpx(cpx_o a)`
Find the phase of a complex number **`a`**.

Complex arithmetic involving one complex number:

`cpx_o cpx_recip(cpx_o a)`
Returns **`1 / a.`**

`cpx_o cpx_conj(cpx_o a)`
Returns **`a*.`**

`cpx_o cpx_scale(cpx_o a, double c)`
Multiply a complex number by a scalor **`c`**.

`cpx_vec_o cpx_vec_scale(cpx_vec_o xn, double c)`
Multiply a vector of complex numbers by a scalor.

Operations involving two complex numbers:

`cpx_o cpx_add(cpx_o a, cpx_o b)`
Returns **`a + b.`**

`cpx_o cpx_sub(cpx_o a, cpx_o b)`
Returns **`a - b.`**

`cpx_o cpx_mul(cpx_o a, cpx_o b)`
Returns **`a * b.`**
`cpx_o cpx_div(cpx_o a, cpx_o b)`
Returns **`a / b.`**

Typecasting involving complex numbers:

`cpx_vec_o upcast_dvec(dvec_o xn)`
Store a vector of double values as the real components to a vector of complex numbers.

dvec_o downcast_cpx(cpx_vec_o zn, char* select)
Store either the real, imaginary, norm, norm-squared, or phase of a vector of complex numbers to a **double** vector.

Workable string values for **select** are:
"**real**": Store real components.
"**imag**": Store imaginary components.
"**mag**": Store norms.
"**mag2**": Store norm-squareds.
"**phase**": Store phases.

Functions - Vector operations:

Printing:

void print_xvec(xvec_o v)
Prints vector of type **x** ∈ [**b**, **i**, **v**, **cpx_**].

void fprint_xvec(xvec_o v, FILE* fout)
Prints vector of type **x** ∈ [**b**, **i**, **v**, **cpx_**] to a file.

Trivial memory management:

void copy_xvec(xvec_o src, xvec_o dst)
Overwrites vector **src** (of type **x** ∈ [**b**, **i**, **v**, **cpx_**]) into vector **dst**. Segmentation fault will occur if **src** is longer than **dst**.

void clear_xvec(xvec_o vec)
Sets all elements of vector **vec** (of type **x** ∈ [**b**, **i**, **v**, **cpx_**]) to zero.

xvec_o zeropad_xvec(xvec_o vec, int new_length)
Copies vector vec vector **vec** (of type **x** ∈ [**b**, **i**, **v**, **cpx_**]) into a vector of length **new_length**, filling empty space with zeros.

Intra-vector comparison:

int min(dvec_o xn, int start_ind, int end_ind)
Finds the index within double vector **xn** which holds the minimum value on the interval [**start_ind**, **end_ind**], endpoint-inclusive.

int max(dvec_o xn, int start_ind, int end_ind)
Finds the index within double vector **xn** which holds the maximum value on the interval [**start_ind**, **end_ind**], endpoint-inclusive.

int min_abs(dvec_o xn, int start_ind, int end_ind)
Finds the index within double vector **xn** which holds the minimum absolute value (closest to zero) on the interval [**start_ind**, **end_ind**], endpoint-inclusive.

`int max_abs(dvec_o xn, int start_ind, int end_ind)`
Finds the index within double vector **`xn`** which holds the maximum absolute value (farthest from zero) on the interval [**`start_ind`**, **`end_ind`**], endpoint-inclusive.

`int numequiv_x(xvec_o x1, xvec_o x2)`
Returns the amount of indices where **`x1[n]`** equals **`x2[n]`**.

Indexing.

`void insert_intra_vec(dvec_o xn, int src_ind, int dst_ind)`
Pull out **`xn[src_ind]`** and shift terms into its empty spot until it can be put in the place of **`xn[dst_ind]`**. Useful for insertion sort in **`sort.h`**.

`void swap_intra_vec(dvec_o xn, int n1, int n2)`
`xn[n1]` gets the value at **`xn[n2]`** and **`xn[n2]`** gets the value at **`xn[n1]`**. Useful for selection sort in sort.h.

`xvec_o vecinsert_x(xvec_o x_dst, xvec_o x_src, int offset)`
Insert vector **`x_src`** (where **`x`** ∈ [**`b`**, **`i`**, **`v`**, **`cpx_`**]) into vector **`x_dst`** at index **`offset`**.

`void vecoverwrite_x(xvec_o x_dst, xvec_o x_src, int offset)`
Overwrite vector **`x_src`** (where **`x`** ∈ [**`b`**, **`i`**, **`v`**, **`cpx_`**]) into **`x_dst`** at index **`offset`**.

`xvec_o subvecfromvec_x(xvec_o in, int start, int end)`
Return a subvector of vector **`in`** of type **`x`** from indices [**`start`**, **`end`**], endpoint-inclusive.

Menial math operations.

`void flipsign_x(xvec_o vec)`
Multiplies all elements of vector **`vec`** of type **`x`** by -1.

`double norm_vec(xvec_o vec)`
Returns Euclidean norm of vector vec of type **`x`**, that is:
norm = sqrt(**`v`**[0]^2 + **`v`**[1]^2 + **`v`**[2]^3 + ... **`v`**[len - 1]^2)

`dvec_o generate_linear_vec(int num_samples, double low, double high)`
Returns vector of linearly spaced values given number of samples **`num_samples`**, low value **`low`**, and high value **`high`**.

`dvec_o generate_linear_vec2(double stepsize, double low, double high, int include_end)`
Returns vector of linearly spaced values given low value **`low`**, high value **`high`**, desired step size **`stepsize`**, and integer **`include_end`**.

Input values for **`include_end`**:
0: **Do not include** final point in linearly spaced vector - eg: [-1, -0.5 , 0, 0.5]
1: **Include** final point in linearly spaced vector - eg: [-1, -0.5, 0, 0.5, 1]

`double sum_x(xvec_o v)`
Return sum of all terms in vector v of type x.

`double avg_x(xvec_o v)`
Return average of all terms in vector v of type x.

Term by term vector operations.

`xvec_o termbyterm_yyy(xvec_o x1n, xvec_o x2n)`
Performs term by term addition, subtraction, multiplication, or division between vectors **`x1n`** and **`x2n`**, with yyy ∈ [**`add`**, **`sub`**, **`mul`**, **`div`**], respectively.

File IO.

`xvec_o datatovec_x(char* filename, int length)`
Returns vector of length **`length`** and type **`x`** whose data is read from file **`filename`**.

dsp.h

This file defines operations with:

Convolution / correlation
FFT / IFFT
Filter generation functions

Functions:

`dvec_o conv(dvec_o xn, dvec_o hn)`
Convolve input signal **`xn`** with impulse response **`hn`**. puts of equal length to input signal.

Note: This particular function is *not commutative* because the vector it outputs is forced to be the same length as the input signal **`xn`**. The tapering *after* the input signal stops existing is ignored.

`dvec_o corr(dvec_o xn, dvec_o hn)`
Correlate input signal **`xn`** with impulse response **`hn`**. Returns correlation of equal length to input signal.

Note: This function is *not commutative* because the vector it outputs is forced to be the same length as the input signal **`xn`**. The tapering *before* the input signal comes *into* existence is ignored.

`dvec_o conv_full_length(dvec_o xn, dvec_o hn)`
Convolve input signal **`xn`** with impulse response **`hn`**. Returns full convolution.

This function is commutative.

`cpx_vec_o fft(cpx_vec_o xn)`
Return Fast Fourier Transform of complex signal **`xn`**.
Note: This function will not work if the length of **`xn`** is not a power of two.

`cpx_vec_o ifft(cpx_vec_o Xk)`
Return Inverse Fast Fourier Transform of complex signal **`Xk`**.
Note: This function will not work if the length of **`Xk`** is not a power of two.

`int pick_window(double stopband_gain)`
Returns integer denoting optimal window function given desired stopband gain. Automatically determines window function which attains necessary sidelobe roll-off while minimizing main lobe width.

Values for output are:
Rectangular: 0
Hanning: 1
Hamming: 2
Blackman: 3
Error: -1

`dvec_o generate_window(int which_window, int window_length)`
Returns particular window function (as selected by **`which_window`**) with **`window_length`** samples.

`int get_filter_length(double w_1, double w_2, double stopband_gain)`
Determines necessary filter length for given pass and stop frequencies **`w_1`** and **`w_2`**, denoted in radians, and desired stopband gain **`stopband_gain`**, denoted as voltage gain factor.
Note: Order of placing arguments **`w_1`** and **`w_2`** does *not* matter as they are only used to compute wc (which is their mean) and **`dw`** (which is the absolute value of their difference).

`dvec_o generate_lowpass_impulse_response(int filter_length, double wc)`
Generates lowpass impulse response given filter length **`filter_length`** and cutoff frequency **`wc`** in radians.

`dvec_o generate_highpass_impulse_response(int filter_length, double wc)`
Generates highpass impulse response given filter length **`filter_length`** and cutoff frequency **`wc`** in radians.

`dvec_o generate_lowpass_filter(double wL_1, double wL_2, double stopband_gain)`
Generates lowpass filter impulse response given boundary frequency **`wL_1`**, boundary frequency **`wL_2`** (both in radians), and desired stopband gain **`stopband_gain`** (designated as voltage gain factor).

Note: Order of placing arguments **`wL_1`** and **`wL_2`** does *not* matter as they are only used to compute wc (which is their mean) and **`dw`** (which is the absolute value of their difference).

`dvec_o generate_highpass_filter(double wH_1, double wH_2, double stopband_gain)`
Generates highpass filter impulse response given boundary frequency **`wH_1`**, boundary frequency **`wH_2`** (both in radians), and desired stopband gain **`stopband_gain`** (designated as voltage gain factor).

Note: Order of placing arguments **`wH_1`** and **`wH_2`** does *not* matter as they are only used to compute wc (which is their mean) and **`dw`** (which is the absolute value of their difference).

`dvec_o generate_bandstop_filter(double wL_1, double wL_2, double wH_1, double wH_2, double stopband_gain)`
Generates bandstop filter impulse response given boundary frequencies **`wL_1`**, **`wL_2`**, **`wH_1`**, and **`wH_2`** (all in radians), and desired stopband gain **`stopband_gain`** (designated as voltage gain factor).

Note: **`wL_1`** and **`wL_2`** are interchangeable, and **`wH_1`** and **`wH_2`** are interchangeable, as each pair of frequencies corresponds to their own filter which creates the composite impulse response.

`dvec_o generate_bandpass_filter(double wL_1, double wL_2, double wH_1, double wH_2, double stopband_gain)`
Generates bandpass filter impulse response given boundary frequencies **`wL_1`**, **`wL_2`**, **`wH_1`**, and **`wH_2`** (all in radians), and desired stopband gain **`stopband_gain`** (designated as voltage gain factor).

Note: **`wL_1`** and **`wL_2`** are interchangeable, and **`wH_1`** and **`wH_2`** are interchangeable, as each pair of frequencies corresponds to their own filter which creates the composite impulse response.

ann.h

This file defines operations with:

Artificial neural net (ANN) initialization, training, testing, evaluation, saving, and loading.

Objects:

The node:

```
typedef struct node {
	double output;

	dvec_o weight;
	dvec_o prev_dw;
	double bias;
	double prev_db;

	double error;
} node_o;
```

Every node has an output value. Active nodes hold a weight vector and bias, a vector of previous weight and bias changes (for momentum factor calculations), and an error measurement.

The layer:

```
typedef struct layer {
	node_o* arr_nodes;
	int num_nodes;
} layer_o;
```

A layer is a group of nodes, active or inactive.

The ANN:

```
typedef struct ann {
	layer_o* arr_layers;
	int num_layers;

	double learning_rate;
	double momentum_factor;
} ann_o;
```

An ANN is a group of layers. The hyperparameters learning rate and momentum factor are also placed here.

Constructors / Initialization:

`void init_weights_and_biases_ann(ann_o ann)`

Randomly select weights and biases for all nodes in the ANN **`ann`** on the range (-0.5, 0.5). Various heuristics exist for weight and bias initialization. This function is easy to modify accordingly.

`ann_o new_ann_o(ivec_o nodes_per_layer)`

Allocate the memory structure for an ANN and randomly initialize its weights and biases.
The length of **`nodes_per_layer`** represents the amount of layers the ANN will have.

`nodes_per_layer[i]` represents the amount of nodes in layer **`i`** of the ANN.

If the length of **nodes_per_layer** is **L**, then **nodes_per_layer[L - 1]** represents the amount of nodes in the output layer of the ANN.

This function also initializes the learning rate and momentum factor, however these are trivial to set at will.

Functions:

Primary ANN operation functions:

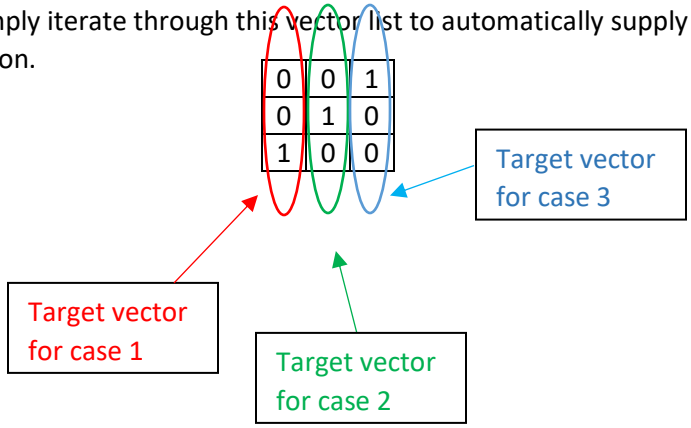
void feed_forward_ann(ann_o ann)
Given a loaded input layer, perform the full feed-forward computation through to the output nodes of the ANN ann. This function uses sigmoid activation.

int backpropagate_adjust(ann_o ann, dvec_o targets)
Given a particular target vector, perform backpropagation to adjust the weights and biases of the ANN. This function uses cross-entropy cost.

Aux. operation functions: Vector generation

dvec_list_o create_targets(int num_cases)
Return a list of **num_cases** vectors. This list of vectors holds the target values for the ANN, assuming that the problem the ANN is solving will only ever call for one output node to be ON at a time. During training, the user can simply iterate through this vector list to automatically supply the correct target vector for backpropagation.

For example:



dvec_list_o create_training_examples()
dvec_list_o create_test_items()
These are functions you must customize for your own specific purpose. Implementing these functions for your specific problem will expedite code implementation at the higher level.

Aux. operation functions: Memory

void preload_input_cpx(ann_o ann, cpx_o src)
Only useful if the ANN **ann** has two input nodes. This function loads **src.real** to the output of the first input node and **src.image** to the output of the second.

void take_output_cpx(ann_o ann, cpx_o* dst)
Only useful if the ANN **ann** has two output nodes. This function loads the output of the first output node to **dst.real** and the output of the second to **dst.imag**.

`int preload_layer_outputs(ann_o ann, dvec_o src, int l_dst)`
Loads **`src`** into the outputs of the nodes of layer **`l_dst`** in ANN **`ann`**. This is most often usable for loading values into the input layer of the ANN.
Returns 1 upon successful load.

`int load _weight_to_node(ann_o ann, int l_dst, int n_dst, dvec_o weight_src, double bias_src)`
Loads weight vector **`weight_src`** and bias **`bias_src`** to node **`n_dst`** in layer **`l_dst`** of ANN **`ann`**.
Retuns 1 upon successful load.

`void save_ann(ann_o ann, char* filename)`
Saves information of ANN **`ann`** to file **`filename`**. **`filename`** can simply be a .txt.

`ann_o load_ann(char* filename)`
Loads ANN information from file **`filename`**. Constructs and returns ANN with said information.

Evaluate performance:

`int max_output_node_ind(ann_o ann)`
Returns the index in the output layer of the output node of ANN **`ann`** which outputs the greatest value.

`int test_and_score(ann_o ann, dvec_list_o test_list, ivec_o answer_key, ivec_o* answer_ann)`
Test ANN ann using testing items in test_list to fill out answer_ann. answer_ann is then compared to answer_key to determine the number of correct guesses made, which is returned.

Misc:

`void print_ann(ann_o ann)`
Print every weight and bias of every node of every layer of the ANN ann to the console.

bitmap.h

This file defines operations with:

Bitmap generation, editing, saving and loading
Drawing, graphing

Objects:

```
typedef struct bmp {
	bvec_o header;
	bvec_o img;
	bvec_o full;

	int x_pix;
	int y_pix;
	int bytespp;

	int total_num_bytes;
	int num_padded_bytes_per_row;
} bmp_t;
```

Every bitmap has a header, an image, and a composite vector combining the two. Hyperparameters include horizontal and vertical pixels and bytes per pixel (currently, always three). At a higher level MIDvec tracks the total number of bytes in the bitmap as well as the number of padded bytes per row (an infrastructure detail).

```
typedef struct graph {
	double x_min;
	double x_max;
	double y_min;
	double y_max;
} graph_o;
```

A graph is characterized by its horizontal and vertical boundaries. These boundaries are how we normalize from the domain of pixels to the domain of any particular graph.

```
typedef struct color {
	unsigned char r;
	unsigned char g;
	unsigned char b;
} color_o;
```

A color is composed of red, green, and blue.

```
typedef struct line_segment {
	double x1;
	double y1;
	double x2;
	double y2;
} line_segment_o;
```

A line segment is an initial point and a final point. Line segments were designed to exist in the graph domain, hence the **`double`** values.

```
typedef struct circle {
      double xc;
      double yc;
      double r;
} circle_o;
```
A circle is its center and radius.

Constructors:

`void randomize_color(color_o* cc)`
Passing this function a color **`cc`** will cause the selection of new R, G, and B values for that color object.

`color_o new_color_string(char* s, double scalor)`
`s` defines the color of the output color object in absolute while **`scalor`** defines the tone. Inside the function, all colors are given at either zero or 255 * **`scalor`**.
Valid values for **`s`**: "**`red`**", "**`green`**", "**`blue`**", "**`magenta`**", "**`yellow`**", "**`cyan`**", "**`gray`**"

`color_o new_color_rgb(unsigned char r_d, unsigned char g_d, unsigned char b_d)`
Returns a color object defined by desired red, green, and blue components (**`r_d`**, **`g_d`**, and **`b_d`**, respectively).

`line_segment_o new_line_segment(double x1d, double y1d, double x2d, double y2d)`
Returns a new line segment object connecting endpoints **`(x1d, y1d)`** and **`(x2d, y2d)`**.

Functions:

Byte parsing:

`void parse_int(bvec_o bb, int x, int start_ind, int num_bytes)`

Parse an integer **`x`** of size **`num_bytes`** into the byte vector **`bb`**, starting at index **`start_ind`** in vector **`bb`**.

Key .bmp generation functions:

`bmp_o new_bmp(int num_cols, int num_rows)`
Initializes and returns bitmap object with **`num_cols`** columns and **`num_rows`** rows.

`void generate_header(bmp_t* bmp)`
Generate bitmap file header given bitmap object.

`void generate_full(bmp_t* bmp)`
Combine header with bitmap object and zeropad as necessary according to convention.

`void package_bitmap(bmp_t* bmp)`
Combines above two functions.

`void save_bmp(bmp_t bmp, char* name)`
Highest level of all above functions - saves bitmap object as filename **`name`**. Saving a bitmap requires only this function.

Simple image functions:

`void set_white(bmp_o bb)`
Set entire bitmap **`bb`** white.

`void set_black(bmp_o bb)`
Set entire bitmap **`bb`** black.

Graphing.

Index Access.

`int get_bmp_img_ind(bmp_o bmp, int x, int y, int which_color)`
Find correct byte in image array for **`bmp`** given **`(x, y)`** plus color byte offset **`which_color`**.

Points.

`int draw_point_onecolor(bmp_o bmp, int x, int y, int which_color, unsigned char val)`
Find point in bmp at **`(x, y)`**, color byte offset **`which_color`**. Change that color filter to **`val`**.

`void draw_point_rgb(bmp_o bmp, int x, int y, color_o cc)`
Find point in bmp at **`(x, y)`**. Change the color of that pixel to **`cc`**.

`void draw_thickpoint_rgb(bmp_o bmp, int x, int y, int halfnumpix, color_o cc)`
Find point in **`bmp`** at **`(x, y)`**. Make a square of color **`cc`** and side length 2*`halfnumpix` centered there.

Lines.

`void draw_line_seg_vert(bmp_o bmp, int x, int yL, int yH, color_o cc)`
Draw a vertical line segment at location **`x`** from **`yL`** to **`yH`** of color **`cc`** in **`bmp`**.

`void draw_line_seg_horiz(bmp_o bmp, int y, int xL, int xH, color_o cc)`
Draw a horizontal line segment at location **`y`** from **`xL`** to **`xH`** of color **`cc`** in **`bmp`**.

`void draw_ruleV(bmp_o bmp, int x, color_o cc)`
Draw a vertical line spanning the entirety of **`bmp`** at horizontal location **`x`** with color **`cc`**.

`void draw_ruleH(bmp_o bmp, int y, color_o cc)`
Draw a horizontal line spanning the entire of **`bmp`** at a vertical location **`y`** with color **`cc`**.

`int draw_line(bmp_o bmp, int x1, int y1, int x2, int y2, color_o cc)`
Draw a line segment from **`(x1, y1)`** to **`(x2, y2)`** of color **`cc`** in **`bmp`**.

More complex shapes.

`draw_circle(bmp_o bmp, int xc, int yc, int radius, color_o cc)`
Draws circle in **`bmp`** centered at **`(xc, yc)`** with radius **`radius`** of color **`cc`**.

`void draw_cross(bmp_o bmp, int xc, int yc, int radius, color_o cc)`
Draws cross in **`bmp`** centered at **`(xc, yc)`** with radius **`radius`** of color **`cc`**.

Coordinate graphing functions:

Mapping graph coordinates to pixel coordinates:

`int map_graph2pix_X(graph_o g, bmp_o bmp, double xd)`
Map **`xd`** from graph domain **`g`** to pixel domain of **`bmp`**; return result.

`int map_graph2pix_Y(graph_o g, bmp_o bmp, double yd)`
Map **`yd`** from graph domain **`g`** to pixel domain of **`bmp`**; return result.

`void map_graph2pix_2D(graph_o g, bmp_o bmp, double xd, double yd, int* xp, int* yp)`
Map **`xd`** and **`yd`** both from grpah domain **`g`** to pixel domain of **`bmp`**, and store the results in pointers **`xp`** and **`yp`**.

`double map_segment (graph_o g, bmp_o bmp, double len, double theta)`
Map line segment with given radius and angle from graph domain **`g`** to pixel domain of **`bmp`**.

Drawing in graph domain:

```
void draw_point_rgb_g(graph_o g, bmp_o bmp, double xd, double yd, color_o cc)
void draw_thickpoint_rgb_g(graph_o g, bmp_o bmp, double xd, double yd, int
halfnumpix, color_o cc)

void draw_ruleV_g(graph_o g, bmp_o bmp, double x, color_o cc)
void draw_ruleH_g(graph_o g, bmp_o bmp, double y, color_o cc)
int draw_line_g(graph_o g, bmp_o bmp, line_segment_o lin, color_o cc)

void draw_circle_g(graph_o g, bmp_o bmp, circle_o cir, color_o cc)
void draw_cross_g(graph_o g, bmp_o bmp, circle_o cir, color_o cc)
```

The above seven functions are all entirely analogous to their counterparts above, only with the addition of the graph argument to define the range over which the graph extends.

Marking axes in graph domain:

`draw_axes(graph_o g, bmp_o bmp, color_o cc)`
Marks on the bitmap the lines $y = 0$ and $x = 0$.

```
void markaxis_X(graph_o g, bmp_o bmp, double grid_interval)
void markaxis_Y(graph_o g, bmp_o bmp, double grid_interval)
```

The above two functions mark grid intervals of width **`grid_interval`** on the **`x`** and **`y`** axes respectively on **`bmp`**.

Low-level plotting:

`draw_xyplot_g(graph_o g, bmp_o bmp, dvec_o x_vals, dvec_o y_vals, color_o cc)`
Uses graph **`g`** to plot a **`cc`**-colored trace of **`y_vals`** as a function of **`x_vals`** on **`bmp`**.

`draw_rthetaplot_g(graph_o g, bmp_o bmp, dvec_o theta_vals, dvec_o r_vals, color_o cc)`

Uses graph **g** to plot a **cc**-colored trace of **r_vals** as a function of **theta_vals** on **bmp**.

int draw_stemplot_g(graph_o g, bmp_o bmp, int offset, dvec_o y_vals, color_o cc)
Uses graph **g** to plot a **cc**-colored stemplot of **y_vals**. Use integer values for **offset** to shift the stemplot right or left in the bitmap.

High-level plotting:

void signal_to_stemplot_bmp(dvec_o xn, char* name)
This function attempts to autoscale a view of signal **xn** before saving it as a stemplot named **name.bmp**.

void fft_mags_to_bmp(cpx_vec_o Xk, char* name, double x_grid_interval)
This function saves the magnitudes of frequency domain signal **Xk** to a bitmap **name.bmp** with horizontal grid interval **x_grid_interval**.

misc_math.h

This file defines operations with:

Miscellaneous math functions: Random numbers, computation, linear mapping, indexing.

Functions:

`double rand_given_bound(double lo, double hi)`
Outputs a random value on bounds (lo, hi); endpoint-exclusive due to continuity of double values.

`int int_rand_given_bound(int lo, int hi)`
Outputs a random integer on bounds [**`lo`**, **`hi`**], endpoint-inclusive.

`double sinc(double x)`
Returns $\frac{\sin(x)}{x}$. Returns 1 if **`x`** equals 0. Zeros occur at $n\pi$, n integer.

`double sincpi(double x)`
Returns $\frac{\sin(\pi x)}{x}$. Returns 1 if **`x`** equals 0. Zeros occur at n, n integer.

`double sigmoid(double x)`
Returns $\sigma(x) = \frac{1}{(1 + e^{-x})}$.
`double sigmoid_prime(double x)`
Returns derivative of sigmoid, $\sigma(x)[1 - \sigma(x)]$.

`int revolve_mod(int cur, int lim)`
Either returns **`cur + 1`** or 0, depending on whether **`cur + 1`** remains less than **`lim`**.

For example, the code:
```
for (;;) { x = revolve_mod(x, 3); printf("%d, " x); }
```
would show in console:
0, 1, 2, 0, 1, 2, 0, 1, 2, 0 ...

`int round_up_int(int x, int base)`
Round **`x`** up to the nearest multiple of base. If **`x`** is a multiple of base, do nothing.

`int less(int a, int b)`
Return the lesser of the two values **`a`** and **`b`**.

`int greater(int a, int b)`
Return the greater of the two values **`a`** and **`b`**.

`double map_val(double lo_1, double hi_1, double lo_2, double hi_2, double val)`
Return the value to which **`val`** maps linearly from the range [**`lo_1`**, **`hi_1`**] to [**`lo_2`**, **`hi_2`**].

sort.h

This file defines operations with:

Insertion sort
Selection sort

Functions:

`void insertion_sort(dvec_o xn)`
Performs insertion sort on **`xn`**.

`void selection_sort(dvec_o xn)`
Performs selection sort on **`xn`**.