Software Engineering Project “Bugfind”
Design Document

Project group
Perttu Haimi, Petteri Hintsanen, Philipp Masche, Toni Ruokolainen and Osku Salerma

Instructor
Peter Lamberg

Helsinki 9th May 2001
Design Document
UNIVERSITY OF HELSINKI
Department of Computer Science

Version history

- v0.1, 21.2.2001 (first draft)
- v0.2, 19.3.2001 (almost finished)
- v0.3, 21.3.2001 (given to the customer and Juha Taina, Gaussian main and configuration module description still finished)
- v0.9, 26.3.2001 (fixed flaws found in the FTR)
- v1.0, 2.4.2001 (frozen)

Contents

List of Figures

1 Preface

This paper contains the design description of the *Bugfind* program. It follows the requirements specification. Since some data structures like the different image file formats are specified in the requirements specification paper, they are not described here again to avoid redundancy.

All the primitive data types used in the following text are more or less the ones used in the C programming language. That is, `unsigned char` means an unsigned byte value, `double` means a double-precision floating point value, `int` means an integer value, and so on. This convention is used only for the sake of simplicity and consistency; it shouldn't produce any significant overhead for the design utilization with some other programming language.

The function prototypes are described using C-like formatting as well. Again, it should be pretty straightforward to think in terms of some other programming language than C, if necessary.

2 Architecture design

2.1 Architectural overview

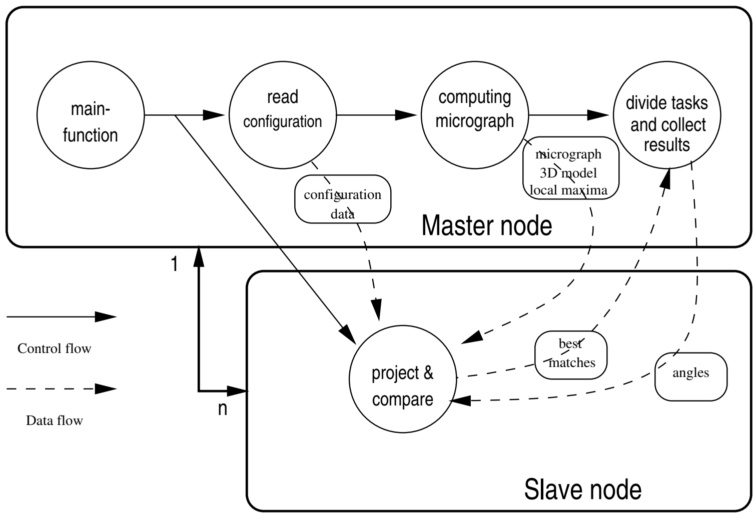

Figure 1: Abstract architecture diagram

The program runs distributed on different nodes. The machine where the user starts the program will be the master node and all other involved machines will be slave nodes (see big rectangles in Figure 1). The Figure 1 shows a very abstract control flow of the program. Each node has its own control flow. The dotted arrows and the small rectangles show the data flow between the master node and the slave nodes. The master node reads the configuration data and sends it to the slave nodes, does some image processing with the micrograph, transfers the local maxima, the micrograph and the 3D model to the other nodes and then divides the distributed tasks and collects the results calculated by the slave nodes. The slave nodes request angles to be processed from the main node.

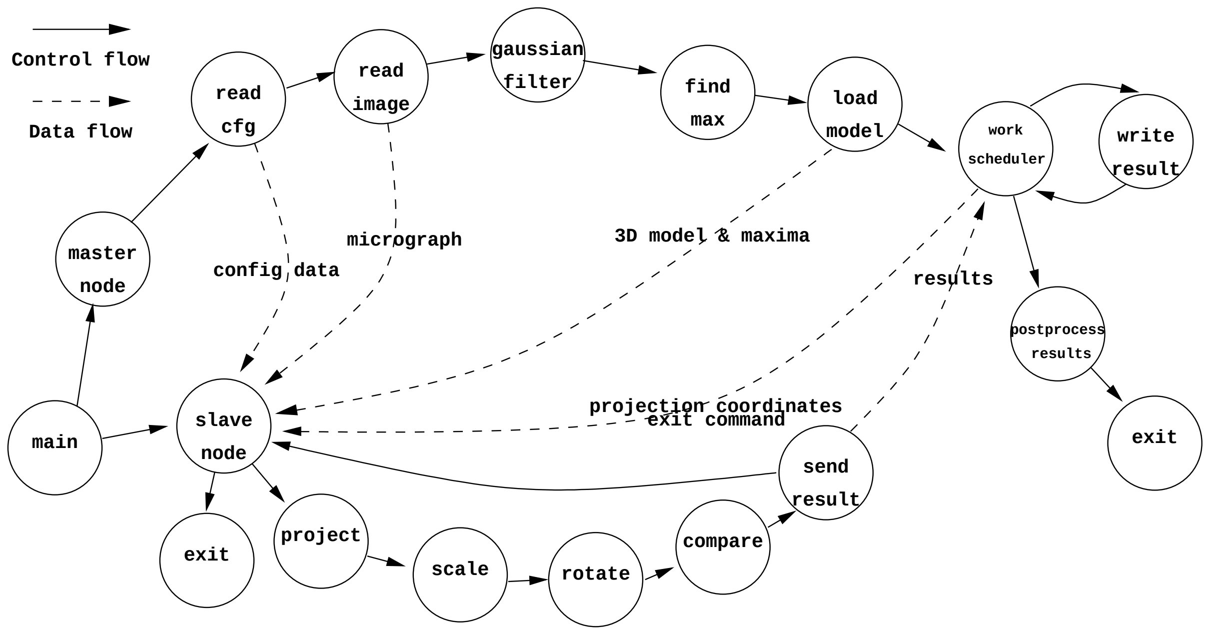

Figure 2: Data and control flow diagram

2.2 Data and control flow description

The data and control flow is quite simple, because the program does mainly a linear sequence of image processing tasks. But since there is some data flow between the distributed processes running on different nodes, the data flow diagram (see Figure 2) helps to understand the problem.

The master node reads and parses the configuration file and sends the needed information to the slave nodes. It then loads the micrograph, does Gaussian filtering and finds the local maxima (the particle diameter is used in the filtering phase). It sends those to slave nodes. Then it discards the filtered micrograph, loads the model and sends it to slaves, discards it, loads the micrograph, sends it to slaves, and discards it. We don't keep them around since the main node has no further use for them and they can take huge amounts of memory.

After sending all the needed initial data to slave nodes, the main node starts dividing up the processing between the slave nodes. Note that up to this point, each slave node has been sent the exact same bytes. Only now do they start getting individualized data from the main node. The distribution model is straightforward, there are a certain number of angles to search through, typically somewhere in the low thousands. The main node just sends angles to slave nodes to work through, and when a node is finished with an angle, it gets a new one to work on. This way slow/loaded machines don't hinder the progress of the whole program.

The main node loops, sending out new angles and waiting for the results from the slave nodes, until all the results are in. The results are written into a temporary file. Then the results are postprocessed: the main node sorts them, picks out the best by eliminating the ones with a low correlation value and/or being too close to another particle, and writes those out to the result file.

The slave node functionality is simple to describe. First it receives all the preliminary information from the master node (configuration data, micrograph, model, etc), then it just loops waiting for angles from the main node and searches through those angles.

To search through one angle, first the slave node does a projection of the model from 3D to 2D (this is where the angle is used). Then it scales the projection to a right size for the searched particle. Then it rotates the images and stores all the rotated versions in a table, since we need them repeatedly. Then it goes through each of the local maxima that the main node found from the micrograph, and tries to match all the rotated projections at each point. Match quality is measured by a correlation value between the projection and the micrograph. It returns the best match from the rotations found for each point to the main node. The slave nodes exit when they get a special message from the main node, telling them that there is no more work to be done.

3 Data design

3.1 Description of data structures

Since the design style of the software is a procedural one, the data structures do not encapsulate data in any way.

3.1.1 Point structure

The point structure `point` wraps together three `double` values representing a point in 3D space. These values are named `x`, `y` and `z`. The point structure is also used to represent 2D points.

3.1.2 Image and model structure

The same data structure, called `image`, is used for both 2D and 3D image data. This structure contains only the absolutely essential information about the image:

- The size of the image is given by three `int` fields: `x`, `y` and `z`. These values also define the dimensions of the image: for 2D images the `z`-value is 1, otherwise the image is considered to be a 3D model.
- Image data itself is just a linear buffer of bytes, pointed by a pointer `data`. The image data type is fixed to `unsigned char`.

3.1.3 Comparison match structure

The comparison module uses a specific data structure for storing the quality of a match. This structure, called `image_match`, has two fields:

- `point pt` stores the coordinates of the match.
- The quality of the match is stored by a `double` value `corr`. The quality is measured by a correlation value between the compared micrograph and the projection used in comparison. The quality is represented by a value between 0 and 1.

3.1.4 Orientation structure

The rotation module utilizes a data structure called `orientation`. It's just like the `point` structure, containing three floating point values, but instead of storing coordinate values it stores angles specifying a new orientation for different objects. The angles are named `alpha`, `beta` and `gamma`.

Figure 3 illustrates the axis orientation in the Bugfind system. All rotations are performed clockwise around the corresponding axises.

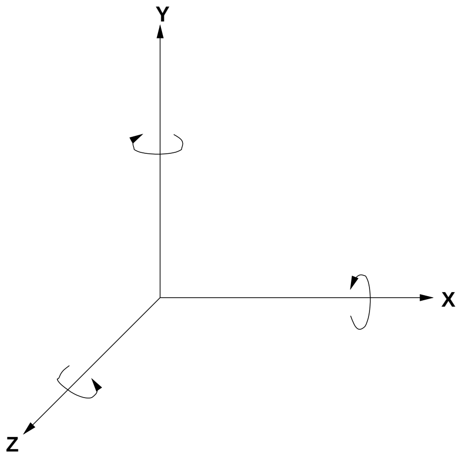

Figure 3: Orientation of the axises

3.1.5 List structure

The list structure is a standard double linked list. It consists of two substructures:

- `node` is a structure representing a node in the list. The node contains pointers to the next and previous node in the list. These pointers are labeled `next` and `prev`. The node's data is pointed by a `void` field `data`.
- `list` is the actual structure representing the double linked list. It stores the pointers to the list's head and tail nodes (`node`-pointers `first` and `last`, respectively), as well as the total number of nodes in the list (`int` field `nodes`).

3.2 Interfaces to structures

3.2.1 Point structure

There are no interface functions; all data access will be done directly.

3.2.2 Image and model structure

Three interface functions are provided:

- `image* create_image(int x, int y, int z)` creates a new image structure by allocating sufficient memory. The function returns a pointer to the image structure.
- `image* copy_image(image* img)` copies an image and returns a pointer to the copy.

- `void delete_image(image* img)` deletes the given image and frees the allocated memory.

3.2.3 Comparison match structure

There are no interface functions; all data access will be done directly.

3.2.4 Orientation structure

There are no interface functions; all data access will be done directly.

3.2.5 List structure

Following interface functions are provided for easy data access:

- `list* list_new()` creates a new, empty list, and returns a pointer to it.
- `node* list_add_last(list* ll, void* data)` creates a new node containing the data given and adds the new node to the tail of the given list.
- `node* list_add_first(list* ll, void* data)` behaves just like `list_add_last`, except that it adds the new node to the head of the list.
- `node* list_add(list* ll, void* data)` is just an alias for `list_add_last`.
- `node* list_add_before(list* ll, node* ptr, void* data)` creates a new list node with the data and adds the new node just before the node `ptr` in the given list.
- `node* list_add_after(list* ll, node* ptr, void* data)` behaves just like `list_add_before`, except that it adds the new node after the given node in the given list.
- `void list_remove(list* ll, node* ptr)` removes the given node from the given list.
- `void list_remove_wc(list* ll, node* ptr)` removes the given node from the given list and deallocates the node's data.
- `void list_delete(list* ll)` is like `list_remove`, except that it deletes the entire list.
- `void list_delete_wc(list* ll)` is like `list_remove_wc`, except that it deletes the entire list.
- `int list_get_size(list* ll)` returns the total number of nodes in the given list.
- `node* list_get_first(list* ll)` returns a pointer to the node at the head of the list.
- `node* list_get_last(list* ll)` returns a pointer to the node at the tail of the list.

3.3 Global data

Although global variables are deprecated in general guidelines of software design, the following global data is used for the sake of simplicity:

- `gv` is a composite variable of type `global_variables`. It holds general global variables used by the software. `global_variables` is a structure consisting of the following `int` fields:
  - `mpi_inited` stores the state of the MPI (initialized or not).
  - `nodes` stores the total number of nodes available.
  - `node` stores the index number of this particular node.
- `cfg` is a composite variable of type `config_data`. It represents the software's configuration, which is read from the configuration file. `config_data` is a structure consisting of the following fields:
  - `unsigned char *image_name` is a pointer to the string containing the name of the micrograph being processed.
  - `unsigned char *model_name` is a pointer to the string containing the name of the particle model.
  - `unsigned char *output_name` is a pointer to the string containing the name of the output file.
  - `diameter` is an `int` field storing the diameter of the particles in the micrograph.
  - `density` is an `int` field storing the projections-raster-density.
  - `searchradius` is an `int` field storing the diameter of the circular area sought around the candidate points.
  - `threshold` is a `double` field storing the threshold value in the postprocessing while eliminating the matches with too low correlation value.

3.4 References to data structures from the modules

The columns of the table represent the modules of the program, the rows represent the referenced data structures.

	Main	**Configuration**	**File**	**Gaussian**	**Findmax**	**Projection**	**Interpolation**	**Scaling**	**Rotation**	**Comparison**
Point	×				×	×	×		×	×
Image	×		×	×	×	×	×	×	×	×
Comparison match	×									×
Orientation						×			×	
List	×				×					

4 Interface design

The user can only run the program from the command line, setting parameters in the configuration file. There is no high level user interface. The configuration file, model file and micrograph file are described in detail in the specification requirement paper.

The interface description of the modules is placed by the module description in the following section.

5 Rationale behind the use of existing code

At a glance it might seem to the reader that there's a lot of re-used code (that is, code that has been taken from some other program instead of writing it by ourselves) in the modules. For instance, many of the subsections describing module internals have a sentence in the preface section saying "this module is based on the source code of ..." or words to that effect. This may be misleading, however. The reader might get a false image of this software being just a quick hack consisting of pieces of code coming from here and there; often referred as a "Saturday-night special" in hacker terms. Fortunately this is not the case here.

Most of the re-used code is taken from the "Beklu" project [K^{+}00]. There are two strong arguments for this: The Beklu project dealt with a problem closely related to our project, thus it has many ready solutions for us. The other point is that we have access to the full project documentation of the Beklu project, so we got an in-depth understanding of the inner workings of the code. In addition, the Beklu code is in production use, so it can be considered to be stable enough. However, most of the code taken from the Beklu project is more or less modified to make it fit into our software architecture, but the ideas behind are pretty much the same.

Another large module taken from "outside" is the Gaussian filtering. The code for the filter is taken from the *GNU Image Manipulation Program* [GIM], which is a free software distributed under the GNU General Public Licence [GPL]. There are several reasons for this:

- First attempt to create a custom gaussian filter algorithm didn't succeed. The filtering results were not good enough, and the algorithm was rather slow. The most irritating thing was that nobody really knew how it worked, so the chances to fix it were quite slim.

- The proposed GIMP filter produced very good results. It was about five times faster, and the code didn't look so cryptic as we feared.

- While testing the experimental comparison algorithm with the images filtered with the GIMP filter, it produced 100% accurate matches. This was *really* promising.

- The GIMP code is certainly much more tested than any algorithm we could produce by ourselves.

- The GIMP is free software.

These reasons convinced us to use the filtering algorithm from the GIMP. Frankly, it would have been insane to try to implement our own filtering, which almost surely would have been much slower, inefficient and unstable. Again, in the software-engineering sense, that would have been pointless.

6 Procedural design

The table shows, which module is presented in which source file and header file.

Module name	Source file name	Header file name
Main	`main.c`	`main.h`
Configuration	`config.c`	`config.h`
File	`file.c`	`file.h`
Gaussian	`gaussian.c`	`gaussian.h`
Findmax	`findmax.c`	`findmax.h`
Projection	`projection.c`	`projection.h`
Interpolation	`interpolation.c`	`interpolation.h`
Scaling	`scaling.c`	`scaling.h`
Rotation	`rotation.c`	`rotation.h`
Comparison	`comparison.c`	`comparison.h`
MPI	`bf_mpi.c`	`bf_mpi.h`
Miscellaneous	`misc.c`	`misc.h`
Double linked list	`list.c`	`list.h`

In addition to the modules described below, there are few general-purpose functions available:

- `void panic(const char* fmt, ...)` is the function to be called in the case of emergency stop. The `panic` function takes care of cleanup and graceful shutdown of the whole system. Essentially, it takes care of the nodes and ensures that no lockups occur.
- `void remove_newline(char* s)` removes a newline from the given string, if present.
- `char* bf_strdup(char* s)` returns a copy of the given string.
- `void* bf_malloc(size_t size)` is a safe `malloc` function, gracefully terminating the program with `panic` should the memory allocation fail.
- `void bf_free(void* ptr)` is a wrapper function for the standard `free`.

These functions are widely used by the modules; therefore their use is not explicitly indicated in the following subsections.

Few typedefs and constants are available as well:

- `NELEMS(x)` is a macro returning the number of elements in the static array `x`.
- `BF_PI` is a `double` value representing a mathematical constant π
- `MIN(x, y)` and `MAX(x, y)` return a smaller and larger value from the given values `x` and `y`.
- `uchar` is a `typedef` for `unsigned char`.

6.1 Module dependencies

The columns of the table represent the modules that use other modules, the rows represent the modules being used.

	Main	Configuration	File	Gaussian	Findmax	Projection	Interpolation	Scaling	Rotation	Comparison	MPI
Main											
Configuration	×										
File	×										
Gaussian	×										
Findmax	×										
Projection	×										
Interpolation						×		×			
Scaling	×										
Rotation	×					×					
Comparison	×										
MPI	×										

6.2 Main loop

6.2.1 Task description

The main module provides the general control flow of the program for both the main node and the slave nodes.

6.2.2 Interface description

The entry point of the main module is the `main` function, unsurprisingly. There it does some general initialization and then divides the control flow depending on whether the machine is the main node or a slave node.

6.2.3 References to other modules

Also unsurprisingly, the main module references pretty much every other module.

6.2.4 Internal data structures

The main module doesn't have any special internal data structures, just normal variables.

6.2.5 Pseudocode

```
main
        init global variables
```

```
        init mpi
        if master_node
                read config file
                main_node_loop
        else
                slave_node_loop
        close mpi

main_node_loop
        send configuration data to slaves
        load micrograph
        do gaussian filtering on micrograph
        do find_max on filtered micrograph
        delete filtered image
        load micrograph
        send microgaph to slave nodes
        delete micrograph
        load model
        send model to slave nodes
        send list of points found by find_max to slave nodes
        send work on to each slave node
        while work left to be done
                wait for results from some node
                save results
                update statistics
                if that node has no more work left
                        send that node more work
        send slave nodes exit command
        postprocess results

slave_node_loop
        receive configuration data
        receive micrograph
        receive model
        receive find_max list
        while main node has not told us to quit
                wait for an angle to work on from main node
                do projection
                do scaling
                do rotation
                for each point in find_max list
                        find best match, send its information to main node
```

6.3 Configuration management

6.3.1 Task description

The configuration module reads the configuration data from the configuration file, tests if there are some missing or unknown options and puts all the option parameters in the global variable `cfg` of the data structure `config_data`.

6.3.2 Interface description

The module provides one interface function:

- `void read_config_file(char* filename)` reads the configuration file specified as a runtime program parameter, tests for errors and finally writes the configuration data into the global variable `cfg`.

6.3.3 References to other modules

The module uses no services from other modules.

6.3.4 Internal data structures

The config module uses a table-driven approach. There's a structure `config_opt` that contains all the recognized variable names allowed in the config file. The contents of this struct are:

- `char* name` is the name of the variable
- `enum config_opt_type type` is the type of the variable (discussed later)
- `void* data` is a pointer to the configuration-data (also discussed later)
- `int required` is a flag to distinguish between required and optional parameters
- `int set` is a flag to keep track of which parameters have been set

The `config_opt_type type` variable needs a bit of explaining. It tells us what type of variable that parameter is. Possible values are CFG_TEXT, CFG_INT and CFG_DOUBLE. Using this information and the `void* data` pointer, the configuration file handling is quite mechanical. It just reads in a line, searches if there's an entry in the struct for a variable named so, if there is, it uses the known type of that variable to parse the value, and stores it in the location pointed to by `data`.

Internally, the module sorts the struct before starting and then uses binary search to locate the entries, but this is just an optimization.

6.4 Reading the model and the micrograph

6.4.1 Task description

A single module called *file* takes care of loading and saving model and micrograph files. It supports two image formats: the software's native *mrc*-format (See the requirement specification for description of mrc.) and the more common *pgm*-format. The image loading functions open an image file for reading and populate a new `image` structure with the image data. This approach makes the `image` structure independent of any explicit file format.

This module is based on the source code from the "Beklu" project [K+00]. Code will be simplified and checked thoroughly:

- The image loading functions have a pointer to the resulting image given to them as parameter. To avoid possible image size mismatches and memory allocation errors, load routines will be changed to always create new image.
- pgm-loading routine has four small finite automata which read the image description (see pgm-format) from the pgm header, ignoring comments. These will be merged to one. Also, the routine does not fully comply to the standard and expects the header elements to be separated by newlines.
- Error situations will be handled by call to the panic-utility function for a clean exit with a descriptive error message.
- The mrc-loading funtion can also load mode 3 (floating point) mrc-files. However, it does not normalize the values to [0, 255] if the original values fit to this range. This may lead to problems with some models. (Lets assume that the model is a density map with values ranging from 1.0234 to 1.8324. With this kind of data we will end up with an `image` structure with all pixels set to 1.)

6.4.2 pgm-format

The pgm (portable greymap) format as defined before July 2000 consists of the following elements, separated by white space: pgm identifier *"P5"*, width, height, maximum grey value (all in ASCII characters) and a raster of binary grey values, where each pixel is represented by a byte. Characters from a '#' to the next end-of-line, before the maxval line, are comments and are ignored. The July 2000 definition of pgm-files has support for multiple images per file and for 16 bit images. We will stick to the pre-July 2000 subset of the format.

6.4.3 Interface description

The following functions are defined in the user interface:

- `image* mrc_load(char* filename)`
  `image* pgm_load(char* filename)`
  These functions load an image from the file (`filename`) and return a new `image` structure containing the image data. The function name prefix defines the file format used.
- `int mrc_save(image* in, char* filename)`
  `int pgm_save(image* in, char* filename)`
  These functions save the image data given in the `image` structure into the specified file (`filename`). Function name prefix defines the file format used.

6.4.4 References to other modules

Image structures are created with the `create_image` function described in the section 3.2.2.

6.4.5 Internal data structures

The file module uses a rather complex data structure `MRCHead` while manipulating a mrc image. This structure is described in the requirements specification. Pgm -routines do not need any data structures except the image.

6.4.6 Pseudocode

The pseudocode is not given since the routines we use are already implemented and only need some improvements.

6.5 Gaussian filtering

6.5.1 Task description

Because the electron microscope images are full of white noise, the program needs to enhance signal-to-noise–ratio of the micrographs to increase the odds of finding the particles. Basicly this can be achieved with some kind of low-pass filtering because low-pass filtering the image suppresses high-frequency noise to some extent. We decided to use Gaussian filtering. The Gaussian module does the Gaussian filtering, which smoothes an images using for each pixel the surrounding pixels and multiplies them with a standard Gaussian kernel (*see fig.4*).

The user defined parameter for this algorithm is the radius (or $FWHM$) of the Gaussian kernel. The standard deviation of the normal distribution used to modify pixel values is calculated based on the supplied radius. The implementation is taken from Gimp [GIM] Gaussian IIR-plugin. The Gimp implementation uses approximation to achieve better performance. The Gimp Gaussian IIR filter routine is slightly adapted to suit better our needs: the filter routine takes an `image *` as an argument, the support for multibyte-imageformat is deleted and also the support for different horizontal and vertical radii is not needed.

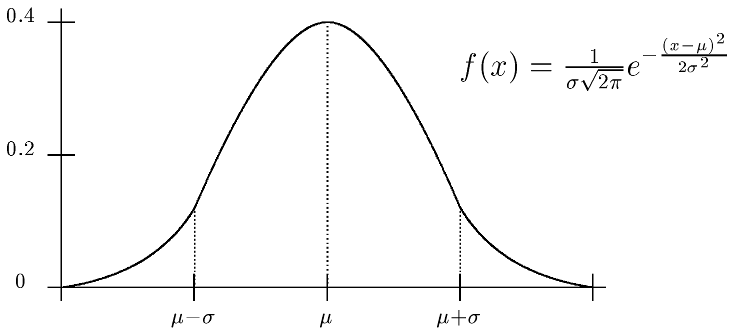

Figure 4: Standard Gaussian-Kernel

First the standard deviation σ for the given radius x is calculated from FWHM (*full width at half-maximum*):

$$\sigma = \frac{x}{8ln(2)} \tag{1}$$

The coefficients used in approximation of Gaussian operator are derived from the standard deviation σ and put into tables. This is done by the function `find_constants`.

6.5.2 Interface description

- `void do_gimp_gaussian(image* in, int radius)` does the 2-dimensional Gaussian filtering in the given image. This module does first a vertical pass through the image and then a horizontal pass with the resulting image from vertical pass. The function makes the changes into the given image.

6.5.3 References to other modules

This module doesn't use any external modules.

6.5.4 Internal functions

The Gaussian module has six internal functions:

- `void get_column(uchar* src, uchar* dest, int x, int width, int height)` Read column `x` from image `src` to buffer `dest`. `width` and `height` are the width and height of the source image.

- `void set_column(uchar *src, uchar* dest, int x, int width, int height)` Write a column from buffer `src` to column `x` in image `dest`. Destination image dimensions are defined in `width` and `height`.

- `void get_row(uchar* src, uchar* dest, int y, int width)` Read line `y` from image `src` to buffer `dest`. Source image width is given in `width`.

- `void set_row(uchar* src, uchar* dest, int y, int width)` Write buffer `src` to `image` line `y`. Destination image width is given in `width`.

- `find_constants(double n_p[], double n_m[], double d_p[], double d_m[], double bd_p[], double bd_m[], double std_dev)` Derive constants from the given standard deviation `std_dev` into buffers. The constants are used when the module is approximating the Gaussian distribution function in each point of the image. All the buffers (`n_p, n_m, d_p, d_m, bd_p, bd_m`) contain five values because the code is using a 4th degree approximation.

- `void transfer_pixels(double *src1, double *src2, uchar* dest, int width)` Transfers the temporary values from `src1` and `src2` to buffer `dest`. The temporary values are added together and then written to `dest`. Destination buffer width is given in `width`.

6.5.5 Internal data structures

There are no internal data structures in use.

6.6 Finding the maxima

6.6.1 Task description

The *find_max* module finds the maximum intensity value from each box in the prefiltered micrograph. The box dimension is a user defined parameter which is relative to the diameter of the particle being searched. The micrograph is first divided to boxes to decrease the amount of searching. There can't be more than one particle in each box. Overlapping particles are identified as garbage. The function first finds the maximum value in each box. After that all the instances of that value are searched in the box. Finally, the center of those points' mass is calculated and this value is returned as the point of maximum intensity value.

6.6.2 Interface description

The module interface is:

- `list* find_max(image* img, int diameter)` The module returns a pointer to a list of the maximum intensity value, for each box. The nodes in the list are of type `point`. The module has two input parameters:
  - `image* img` is a pointer to the prefiltered micrograph.
  - `int diameter` is the box diameter measured in pixels.

6.6.3 References to other modules

This module doesn't use any external modules.

6.6.4 Internal functions

The `find_max`-module has two internal functions:

- `point* new_point(int x, int y)` which returns a pointer to a new `point` structure which is initialized with the input parameters `int int x`, `int y`.
- `void handle_box(image* img, list* l, int x, int y, int diameter)` This function finds the maximum intensity value in each box. The input parameters are:
  - `image* img` is a pointer to the prefiltered micrograph.
  - `list* l` is a pointer to the `list`-structure which will hold the maximum intensity value coordinates for each box.
  - `int x` and `int y` The left and upper x– and y–coordinates for the box.
  - `int diameter` The box diameter.

6.6.5 Internal data structures

There are no internal data structures.

6.6.6 Pseudocode

```
function find_max(image, diameter) begin
        list := list_new()
        for each diameter sized box in image do begin
                handle_box(box, list)
        end for
        find_max := list
end function
```

```
function handle_box(box, list) begin
        maximum_value := -1
        for each line in box do begin
                for each column in line do begin
                        if value in (line, column) > maximum_value then begin
                                maximum_value := value(line, column)
                        end if
                end for
        end for

        maximum_x := 0
        maximum_y := 0
        maximum_values_in_box = 0

        for each line in box do begin
                for each column in line do begin
                if value in (line,column) = maximum value then begin
                        maximum_y := maximum_y + line
                        maximum_x := maximum_x + column
                        maximum_values_in_box := maximum_values_in_box + 1
                end if
                end for
        end for

        mass_center_x := maximum_x / maximum_values_in_box
        mass_center_y := maximum_y / maximum_values_in_box

        list_add(list, new_point(mass_center_x, mass_center_y)
end function
```

6.7 Projection

This module is based on the source code from the “Beklu” project [K$^+$00].

6.7.1 Task description

The projection module projects a 3D model from the desired angle and produces a 2D projection image from it. The projection is made by rotating the model around its center by two angles; first around the X-axis and then around the Y-axis. Rotation around the Z-axis is not performed; it is faster to rotate the produced 2D image later, thus achieving the same result. Then the rotated model is “shot through” with vectors parallel to the Z-axis. Each of the vectors contribute one pixel to the resulting projection. A pixel's value in the projection image is the average of the intensities of the pixels intersected by the corresponding vector.

In practice the actual model rotation (that is, creating a new model structure) is not performed, but simulated instead. Only the corner point $(0, 0, 0)$ is transferred to the desired location and used as an origin for the first projection vector. For the following vectors preprocessed (that is, correctly rotated) unit vectors are used to determine the correct origins and directions. Model rotation around its center point is performed by transferring the origo temporarily. The interpolation routine assumes this and returns corresponding points in the original coordinate system.

6.7.2 Interface description

The module provides one interface function:

- `image* project(image* model, double alpha, double beta)` does the projection from the given model. First the model is rotated around the X-axis clockwise by `alpha` degrees and then around the Y-axis by `beta` degrees (clockwise as well). The function returns a new image representing the produced projection.

6.7.3 References to other modules

The module uses services from two other modules:

- Rotating the model is done with `rotate_point` function, implemented in the rotation module (section 6.10).
- Interpolation (determining the intensity of some point in the model space) is done with `interpolate_nn` function, implemented in the interpolation module (section 6.8).
- Image structures are created with the `create_image` function described in section 3.2.2.

6.7.4 Internal data structures

Since the intensity sums calculated during the projection (that is, the sum of the intensities intersected by the vectors) are almost certainly beyond the range of single `byte`, a temporary buffer consisting of `long int` values is used (namely `ibuf`). This buffer is then scaled down to the single-byte value range.

To determine the origin points for projection vectors, a correctly oriented unit vector base is required to simulate the rotated model. These unit vectors are called `ux`, `uy` and `uz`.

Finally a new image structure `proj` is created and populated with the generated 2D projection data. This structure is then returned to the calling procedure.

The variables above are the most significant ones in the module. In addition there are some straightforward miscellaneous variables such as loop counters, etc.

6.7.5 Pseudocode

```
function project(model, alpha, beta) begin
        imin := ∞
        imax := -1

        orient := alpha, beta, 0
        ux := rotate_point(1, 0, 0, orient)
        uy := rotate_point(0, 1, 0, orient)
        uz := rotate_point(0, 0, 1, orient)

        p := -model.x / 2, -model.y / 2, -model.z / 2
```

```
        min := rotate_point(p, orient)

        for i := 0 to model.x - 1 do begin
                p.x := i * ux.x + min.x
                p.y := i * ux.y + min.y
                p.z := i * ux.z + min.z

                for j := 0 to model.y - 1 do begin
                        np.x := j * uy.x + p.x
                        np.y := j * uy.y + p.y
                        np.z := j * uy.z + p.z

                        for k := 0 to model.z - 1 do begin
                                ibuf[i + j * model.x] :=
                                        ibuf[i + j * model.x] +
                                        interpolate(model, np)
                                np.x := np.x + uz.x
                                np.y := np.y + uz.y
                                np.z := np.z + uz.z
                        end for
                end for

                if ibuf[i + j * model.x] > imax then
                        imax := ibuf[i + j * model.x]
                else if ibuf[i + j * model.x] < imin then
                        imin := ibuf[i + j * model.x]
                end if
        end for

        proj := create_image(model.x, model.y, 1)

        for i := 0 to model.x * model.y - 1 do begin
                proj.data[i] := 255 * (ibuf[i] - imin) / (imax - imin)
        end for

        project := proj
end function
```

6.8 Interpolation

This module is partially based (nearest-neighbour-method) on the source code from the "Beklu" project [K^{+}00].

6.8.1 Task description

As the point coordinates are integral values and the coordinates used in image manipulating procedures are often floating point values, it's not straightforward what actual (integral) point should represent the floating point coordinate. There exists several approaches in this

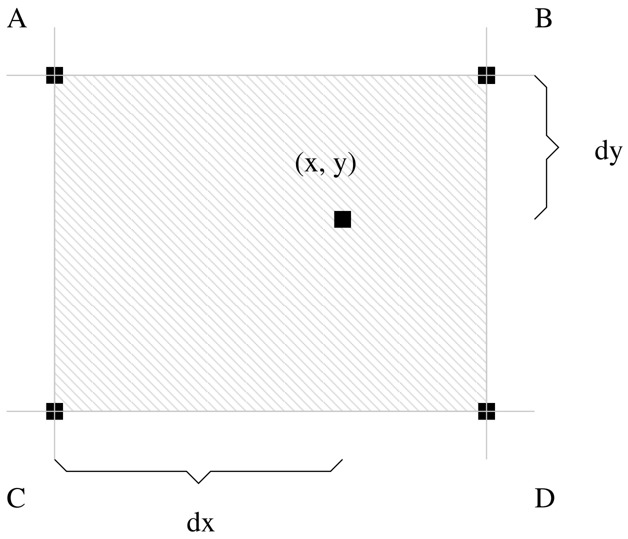

Figure 5: Linear interpolation.
The intensity for a floating point coordinate (x, y) is to be determined. A, B, C and D represent the neighbouring integral points. dx and dy are the corresponding distances between the floating point and integral points. These values are used to calculate the weighted average value.

situation: for example, one can round to the nearest (integral) neighbor and use its intensity or calculate an average of the intensities of the nearby (integral) neighbors.

The interpolation module implements two simple interpolation methods:

- Nearest-neighbor-rounding (discussed above), where the given (possibly floating point) coordinate point is just rounded to the nearest integral coordinate point and its intensity value is returned. This is probably the most intuitive approach.

- Linear interpolation method, where the intensity value at given floating-point coordinate is determined by calculating a weighted average from the intensities of the nearest four integral neighbours. Figure 5 illustrates this procedure. Every floating point coordinate has (excluding points near the image's bottom and right edges) four integral neighbours (labeled A, B, C and D in the figure). By calculating distances (dx and dy in the figure) from the floating point value to these integral points, it is possible to calculate a weighted average of the intensities contributed by each of the neighbouring points using the following formula:

  $$I_{x,y} = 1/2 + (dx * (I_B - I_A) + I_A) + ((dx * (I_D - I_C) + I_C) - (dx * (I_B - I_A) + I_A)) * dy$$

  where I_x represents an intensity at the point x, dx is the horizontal distance to A from the floating point coordinate (x, y) and dy is the corresponding vertical distance. $I_{x,y}$ is the result of the interpolation and is returned to the caller.

  At boundary areas, where the pixel to be interpolated has only one or two integral neighbouring pixels, the other (missing) pixels are substituted by the existing pixels (that is, the interpolation function doesn't try to predict any non-existing intensity values). This induces a small error which just has to be accepted. However, for our purposes the error is not significant.

  The idea for the interpolation was taken from the Lindley's textbook "Digital Image Processing in C" [Lin91]. More comprehensive discussion about the subject is presented there.

In addition, the nearest-neighbour-interpolation function assumes that the coordinate system is oriented so that the image center point is at the origo $(0, 0, 0)$. If the given coordinates are illegal (that is, they point "outside the image"), an intensity value of zero is returned.

6.8.2 Interface description

The module provides two interface functions:

- `unsigned char interpolate_nn(image* img, point p)` returns the intensity of point in the given image using the interpolation method described above (hence the name suffix "nn").
- `unsigned char interpolate_linear(image* img, point p)` returns the intensity of point in the given image using the linear interpolation method described above.

6.8.3 References to other modules

There are none.

6.8.4 Internal data structures

In addition to a few straightforward miscellaneous variables there are none.

6.8.5 Pseudocode

```
function interpolate_nn(image, point) begin
        i := integer(point.x + image.x / 2 + 0.5)
        j := integer(point.y + image.y / 2 + 0.5)
        k := integer(point.z + image.z / 2 + 0.5)

        if i < 0 or j < 0 or k < 0 or
                i >= image.x or j >= image.y or k >= image.z then begin
                interpolate_nn := image.data[0]
                return
        end if

        interpolate_nn := image.data[i + j * image.x +
                k * image.x * image.y]
end function

function interpolate_linear(image, point) begin
        ix := integer(point.x)
        iy := integer(point.y)
        dx := point.x - ix
        dy := point.y - iy

        if ix + 1 >= image.x then begin
                ul := image.data[ix + iy * image.x]
                ur := ul
                if iy + 1 >= image.y then begin
```

```
                ll := ul
                lr := ul
            else begin
                ll := image.data[ix + (iy + 1) * image.x]
                lr := ll
            end if
        else begin
            ul := image.data[ix + iy * image.x]
            ur := image.data[ix + 1 + iy * image.x]
            if iy + 1 >= image.y then begin
                ll := ul
                lr := ur
            else begin
                ll := image.data[ix + (iy + 1) * image.x]
                lr := image.data[ix + 1 + (iy + 1) * image.x]
            end if
        end if

        intupper := dx * (ur - ul) + ul
        intlower := dx * (lr - ll) + ll

        interpolate_linear := 0.5 + intupper + (intlower - intupper) * dy
end function
```

6.9 Scaling

6.9.1 Task description

The scaling module provides a function for image scaling. The image can be magnified or shrinked. Shrinking is accomplished by using the straightforward method: the target image is made row by row, and every i-th pixel is taken from every horizontal pixel row from the original image, where i is the shrinking factor. Magnifying is a bit more complicated task, since some kind of interpolation is obviously needed. Again the target image is manufactured row by row, and every pixel from the original image is used in the corresponding point in the destination image. The gaps between these "real" pixels from the original image are filled by using a linear interpolation, implemented in the interpolation module (section 6.8).

More detailed description of the scaling process can be found from the Lindley's textbook "Digital Image Processing in C" [Lin91].

6.9.2 Interface description

The module provides one interface function:

- `image* scale(image* img, int x, int y)` returns a new image of size (x, y). The image is magnified and/or shrinked to achieve the desired geometry values.

6.9.3 References to other modules

The scaling function uses a linear interpolation function implemented in the interpolation module (section 6.8).

6.9.4 Internal data structures

In addition to a few straightforward miscellaneous variables there are none.

6.9.5 Pseudocode

```
function scale(image, x, y) begin
        if x = image.x and y = image.y then return copy_image(img)

        fx := x / image.x
        fy := y / image.y

        new_image := create_image(x, y, 1)

        for j := 0 to y - 1 do begin
                point.y := j / fy
                for i := 0 to x - 1 do begin
                        point.x := i / fx
                        new_image.data[i + j * new_image.x] :=
                                interpolate_linear(image, point)
                end for
        end for

        scale := new_image
}
```

6.10 Rotation

This module is based on the source code from the "Beklu" project [K+00].

6.10.1 Task description

The rotation module performs coordinate rotations using linear transformations. Rotations are needed during projecting and searching. All rotations are performed clockwise around the axes. Angles are assumed to be radians.

The following rotation matrices are used, where subscripts indicate the corresponding axes and α is the amount of degrees to rotate:

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

6.10.2 Interface description

The module provides the following interface functions:

- `void rotate_2d_image(image* img, double alpha)` rotates the image `img` around its centerpoint by `alpha` degrees.
- `point rotate_point(point p, orientation o)` rotates the given point to the new orientation. The angles used in the rotation are given in the `orientation` structure (see section 3.1.4).
- `point rotate_point_around_x(point p, double alpha)`
  `point rotate_point_around_y(point p, double alpha)`
  `point rotate_point_around_z(point p, double alpha)`
  These functions rotate the given point around the corresponding axis by `alpha` degrees.

6.10.3 References to other modules

The module needs an interpolation function, since the rotated coordinates are non-integral in most cases. `interpolate_nn` function in the interpolation module (section 6.8) is adequate.

6.10.4 Internal data structures

All the variables used are pretty straightforward. The `rotate_2d_image`-function uses a temporary buffer `ibuf` during the construction of the rotated image.

6.10.5 Pseudocode

```
function rotate_point_around_x(point, alpha) begin
        cos_a := cos(alpha)
        sin_a := sin(alpha)

        np.x := point.x
        np.y := cos_a * point.y - sin_a * point.z
        np.z := sin_a * point.y + cos_a * point.z

        rotate_point_around_x := np
end function

function rotate_point_around_y(point, alpha) begin
        cos_a := cos(alpha)
```

```
        sin_a := sin(alpha)

        np.x := cos_a * point.x - sin_a * point.z
        np.y := point.y
        np.z := sin_a * point.x + cos_a * point.z

        rotate_point_around_y := np
end function

function rotate_point_around_z(point, alpha) begin
        cos_a := cos(alpha)
        sin_a := sin(alpha)

        np.x := cos_a * point.x - sin_a * point.y
        np.y := sin_a * point.x + cos_a * point.y
        np.z := point.z

        rotate_point_around_z := np
end function

function rotate_point(point, orientation) begin
        rotate_point := rotate_point_around_z(
                rotate_point_around_y(
                rotate_point_around_x(point, orientation.alpha),
                orientation.beta),
                orientation.gamma)
end function

function rotate_2d_image(image, alpha) begin
        cos_a := cos(alpha)
        sin_a := sin(alpha)

        for j := 0 to image.y - 1 do begin
                for i := 0 to image.x - 1 do begin
                        x := (i - image.x / 2) * cos_a +
                                (j - image.y / 2) * sin_a
                        y := -(i - image.x / 2) * sin_a +
                                (j - image.y / 2) * cos_a
                        for k := 0 to image.z - 1 do begin
                                p.x := x
                                p.y := y
                                p.z := k
                                buf[i + j * image.x +
                                        k * image.x * image.y] :=
                                        interpolate_nn(image, p)
                        end for
                end for
        end for
```

```
        image.data := buf
end function
```

6.11 Image comparison

6.11.1 Task description

The task for image comparison module is to recognize actual particles from potential candidate coordinate results from *find_max*–module (see section 6.6). To do this, the module will receive two different images: projection of the 3D model and the micrograph, search point and search radius. The module will then calculate Pearson's correlation coefficient between these pictures and find the best match in the specified search radius (a search radius of 0 means not to search around the point).

6.11.2 Interface description

The module's interface is:

- `image_match compare(image* img, image* proj, point p, int search_radius)`
  Input parameters:
  - `image* img` Pointer to the micrograph image
  - `image* proj` Pointer to the projection image of the 3D model
  - `point p` The center (x,y coordinates) of the search area
  - `int search_radius` The search radius

  Return value:
  - `image_match` A structure holding the (x,y) coordinates and the correlation coefficient value [0.0,1.0] for the best match in the area.

6.11.3 References to other modules

This module doesn't reference any other modules.

6.11.4 Internal functions

The modules uses an internal function to calculate the Pearson's correlation coefficient. The correlation coefficient is calculated in the area which has the same dimensions as the projection image.

- `double pearson(image* mgraph, image* proj, int sx, int sy)`
  Input parameters:
  - `image* mgraph` Pointer to the micrograph image
  - `image* proj` Pointer to the projection image
  - `int sx` and `int sy` The upper left coordinate of the square where the correlation coefficient will be calculated.

Return value:

– A `double`, which is in range [0.0,1.0] if everything went right, or -1.0 if there was an error (`int sx` or `int sy` were invalid).

6.11.5 Internal data structures

This module doesn't have any complex internal data structures.

6.11.6 Pseudocode

```
function compare(micrograph, projection, center, radius)
        best_correlation := -1.0
        best_match_x := 0
        best_match_y := 0

        for each (x, y) within radius from point center do begin
                corr := pearson(micrograph,projection,comparison_coords)
                if corr > best_correlation then begin
                        best_correlation := corr
                end if
        end for

        compare := { best_match_x, best_match_y, best_correlation }
end function

function pearson(micrograph,projection,comparison_coords)
```

$$\texttt{pearson} := \frac{\sum xy - \frac{\sum(x)\sum(y)}{n}}{\sqrt{(\sum(x^2) - \frac{(\sum(x))^2}{n})(\sum(y^2) - \frac{(\sum(y))^2}{n})}} \tag{2}$$

```
end function
```

7 Testing plan

The testing will, as mentioned in the requirement specification, take place as a bottom up modular testing.

From each module the following properties will be tested:

- Module interface
- Buffer over- and underruns
- Memory allocation and freeing.
- Handling of exceptional situations.

Integration testing will take place as well as testing with different distribution parameters.

For a final abstract level testing, a test image generator will be programmed. The performance of the program with pictures with various noise levels will be assessed. Test pictures with projections from slightly different models will be used to test the selectivity of the system.

Various real test images and models from the client will be obtained and the performance of the system will be evaluated. The client will have the possibility to test the system with his data.

A more detailed description of the testing will be done in the testing plan document.

References

[GIM] The GNU Image Manipulation Program. WWW hypertext, http://www.gimp.org.

[GPL] The GNU General Public Licence. WWW hypertext, http://www.gnu.org/copyleft/gpl.html.

[K+00] I. Koskenniemi et al. *Beklu: Makromolekyylikompleksien rekonstruktioprosessin rinnakkaistaminen Beowulf-ympäristössä.* University of Helsinki, Department of Computer Science, 2000.

[Lin91] Craig A. Lindley. *Practical Image Processing in C: Acquistion, Manipulation and Storage.* John Wiley & Sons, Inc., 1991.