Programming with GLAF:
A Step-by-Step Example

This guide provides a step-by-step example for developing a program using the GLAF programming framework. It covers many aspects of GLAF programming and attempts to highlight the mindset with which one should approach GLAF programming to take advantage of the framework's features.

Prepared by Konstantinos Krommydas (kokrommy@vt.edu)

July, 2016
Blacksburg, Virginia

Copyright 2016

In this tutorial, we present how to develop a simple program in order to showcase some of the main functionalities of the GLAF programming framework, as well as the GLAF programming paradigm.

Specifically, our program is related to image processing. Given an input image in the RGB format (i.e., an image with three components; Red, Green, Blue), we compute an output image that is scaled according to certain computations (as function of the pixel position). The details of the computation itself are not important. This is an intentionally dummy example that functions as an introduction to GLAF programming.

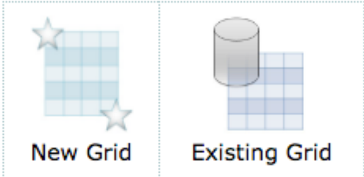
The pseudo-code of our program is given below:

```
//Declare RGB images (input and output)
rgb_image input_img[7][7]
rgb_image output_img[7][7]

//Initialize input_img
for each row and each column (7x7 pixels) of each component{
        if (row>3) {
                input_img[R component][row,col] = row*2
                input_img[G component][row,col] = row*3
                input_img[B component][row,col] = row*4
        } else {
                input_img[R component][row,col] = 0
                input_img[G component][row,col] = row^2
                input_img[B component][row,col] = row^2
        }
}

for each row and each column (7x7 pixels) of each component{
        output_img[R component][row,col] = input_img[R component][row,col]*2
        output_img[G component][row,col] = input_img[G component][row,col]*2
        output_img[B component][row,col] = input_img[R component][row,col]*2
}
```

You can follow the example step-by-step using the online version of GLAF available at http://glaf.cs.vt.edu/glaf_online/grid_main.html

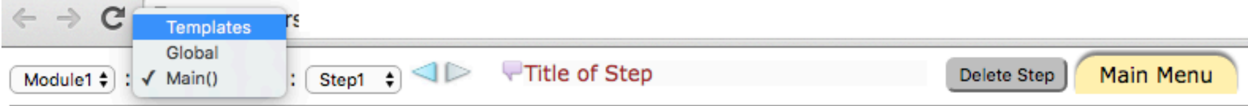
For any questions related to GLAF, please, contact Konstantinos Krommydas (kokrommy@vt.edu) or Ruchira Sasanka (ruchira.sasanka@intel.com).

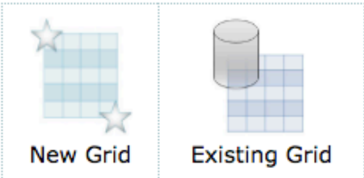
Every GLAF program starts with the main GLAF screen that defaults at the first step of the main function (Main()).



Select Output Grid for Step

Before starting a program it is wise to identify any type of grids (i.e., custom data-types) that is going to be re-used throughout the program, so that we avoid having to design it for every new declaration. In our case, we will use a grid that corresponds to an RGB image. This corresponds to the functionality of *templates*. To define a template, we click on the second drop-down menu that contains *Templates*, *Global Grids*, and *Functions* that belong to the current module of our program.

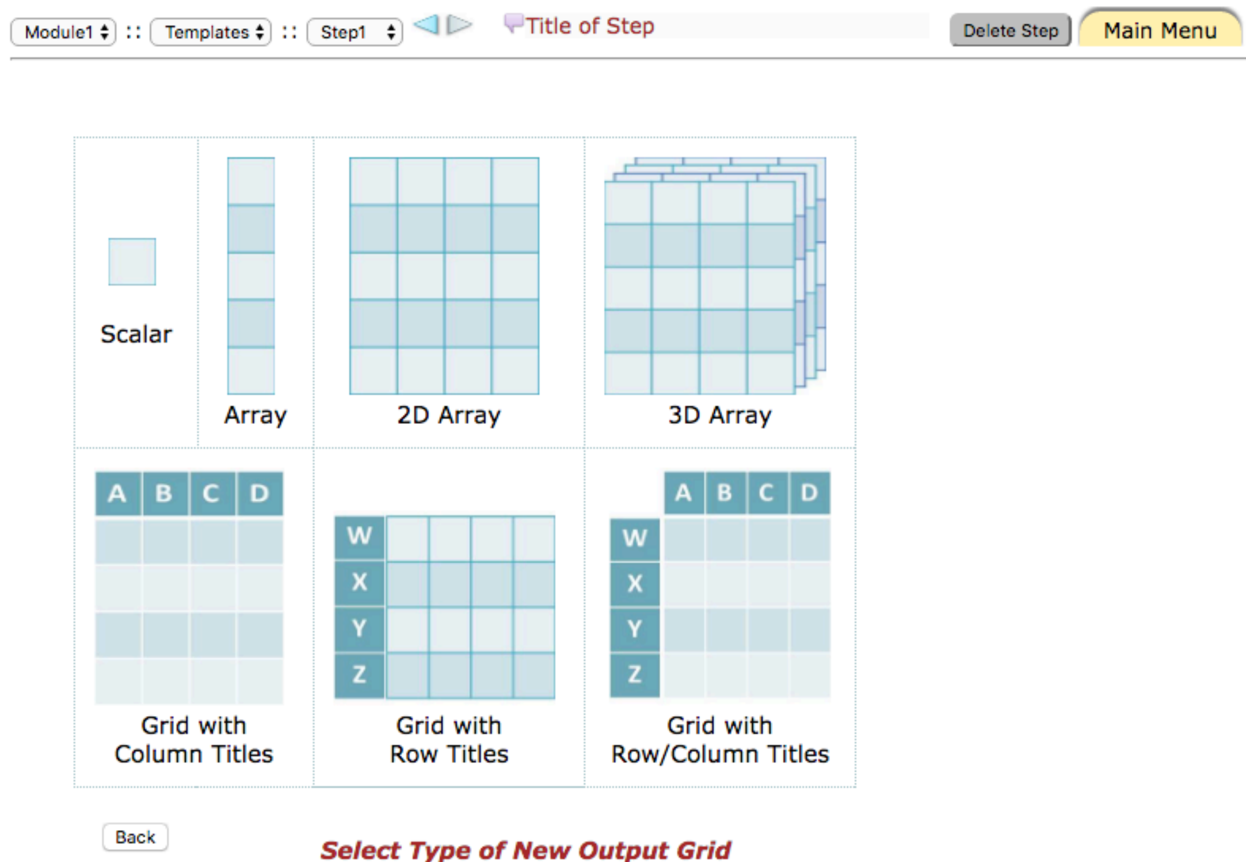




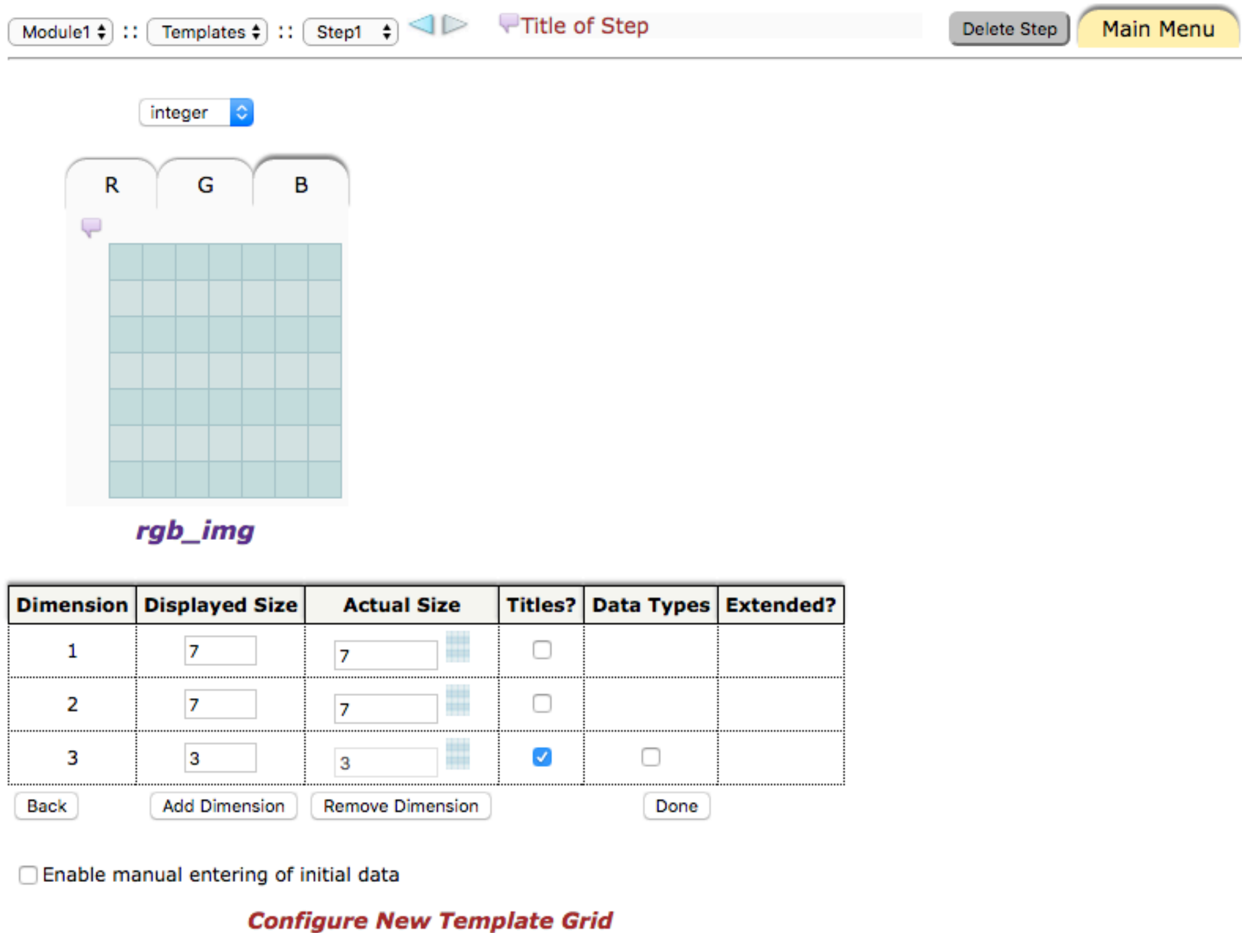
Select Output Grid for Step

Now, we are in the Templates section of our program. We click on *New Grid* to define a new template grid. We want to define a three-dimensional grid; two dimensions will correspond to the two dimensions of the image and the third dimension will correspond to the *Red (R)*, *Green (G)*, and *Blue (B)* component of the image.

Once, we click on *New Grid* we are presented with commonly used predefined grids (e.g., scalar, array, 2D array). Since none of the predefined grids apply in our case, we select the 2D Array (for every selection we can subsequently add/remove dimensions or change the attributes of the grid).



From the grid configuration screen, we click the *Add Dimension* button to add the third dimension. We specify the *Displayed Size* and *Actual Size* of each dimension. We leave dimensions 1 and 2 as they are and update the sizes for dimension 3 with the value *3* (for red, green, blue components of an image). The *Titles?* check-box is by default checked and predefined titles are added for the third dimension (*Tab00*, *Tab01*, ...). We change the default dimension titles to *R*, *G*, *B* by double-clicking on the pre-existing default names (*Tab00*, etc.) and writing the new titles. Also, we set a name for our template grid by clicking on the pre-existing default grid name (*Out*) and changing it to the desired new name (*rgb_img*). After finishing the grid configuration step, we click the *Done* button.

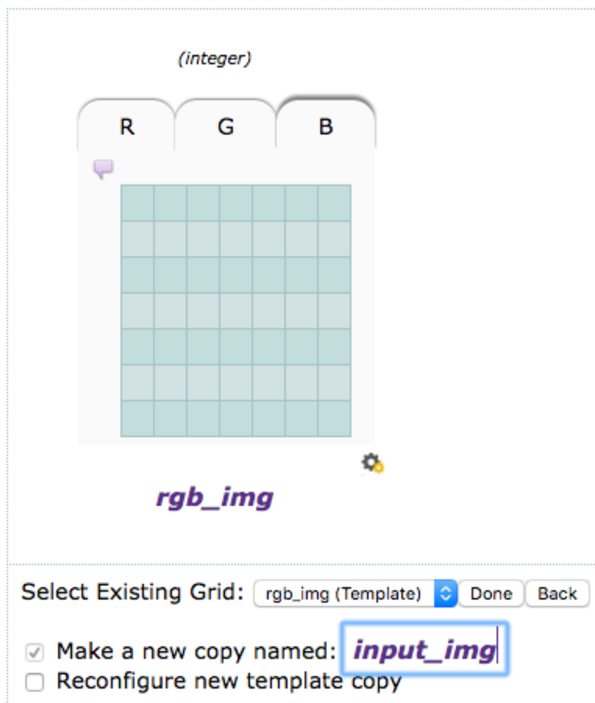


From the drop-down menu on the top of the GUI (where we selected *Templates* earlier) we now select the *Main()* function and click the *Insert New Step* button at the right side of the top bar. This will take us to the first step of our algorithm. Recall that a GLAF program contains one or more functions (starting with *Main()*) and each function contains one or more steps.

In the first step we will initialize the input image of our algorithm. The current screen asks us to select an *Output Grid*. Recall that each step contains one *Output Grid* and zero or more *Input Grids*. Computation within a step *flows* from the input grids to an output grid. Hence, it is suggested that users develop their programs keeping this in mind for every step. However, technically GLAF does not prohibit writing information to a grid selected as input grid, nor leave an output grid of a step unwritten.

Since we are going to use a template to define this step's output grid, we need to select *Existing Grid*. From the drop-down menu of existing grids, we select our desired template grid (*rgb_img (Template)*).

For templates, we use the template to create a *copy* of the template, so we need to give a name to this new grid by clicking on the pre-defined copy name (*Copy*) next to *Make a new copy named:* (make sure the check-box is checked). In our case, we name it *input_img*. Once we name the template copy grid, we click *Done*.



Now, this grid (*input_img*) has been selected as our output grid for the step. This denotes that this grid will get written in this step by the computation specified in said step. Since we are not going to use any source grid (or *input grid*), we click *Cancel* to finish specifying grids for the step. Subsequently, we are confronted with the computation part of the step.

Notice how the grids of a step are collocated with the computation pseudo-code in the same screen. Using only clicks via the mouse and typing any numbers we will now define the desired computation for the step and inspect the effect of the step's code (i.e., results) on the grids at the end.

Before doing this, though, let's pay closer attention to what we see in the current screen starting with the grids:

Note the *input_img* grid; on the horizontal and vertical dimension you can see predefined labels: *0*, *row*, *end0*, and *0*, *col*, *end1*, respectively. These labels can be used to easily address (i.e., refer) to cells within a grid or to set *start*, *step*, *end* values in loops (discussed later). By clicking on any blank space next to a dimension, we can also enter any other number or expression (that may optionally include *end0*, *end1*, or another scalar grid used in the step). The name of the *index variables* for each available dimension (*row*, *col*, *ind2*) is conveniently displayed above the statement boxes (clickable for easy use).

Moving to the statement boxes section we note the following:

There are three types of statement boxes that a step can contain; *Index Range* (which corresponds to a loop), *Condition* (which corrresponds to a conditional statement), and *Formula* (which corresponds to a statement that specifies computation using input/output grids). Each step by default contains one of each of the above types of statement boxes. The user may not need to use one or more of these, or (in the practical case) will want to add more boxes of a type (e.g., more condition boxes for complex conditional statements, more formula boxes). However, each step can *only* contain one *Index Range*. This is a conscious design decision in GLAF: if users want a nested loop structure at some point in the current step, said loops need to be separate steps of a new function (via a user function call).

Now, let's move to the actual implementation of our step's computation. Specifically, we want to initialize all components (R, G, B) of the image along the two dimensions. So, we keep the default Index Range statement box as is (*foreach row col*). As is, the loop iterates over all values of *row* from *0* to *end0* and all values of *col* from *0* to *end1*. This corresponds to the following C code:

```
for (row=0; row<=end0; row++)
        for (col=0; col<=end1; col++)
```

Note that the *end0*, *end1* are inclusive. You may have noticed that the *start*, *step*, *end* values for each index variable are not explicitly seen. To see and/or change any of these values, you need to click on the index variable used in the index range statement box. For example, clicking on *row* expands it as seen below:

You can see that *row=input_img(0:end0:1)* specifies that *row* will obtain values starting from *0* to *end0* (where *end0* corresponds to the size for dimension 0 for the *input_img* grid – *end0* may be different for a different grid, depending on the grid's specification during its creation). The loop's step for *row* is *1*. If we were to change the grid to which *row* refers, we would need to click the name of the grid on the expression for *row* in the Index Range statement box and subsequently click at the label of the grid of interest. Similarly, clicking on *0*, *end0*, *1*, would allow us to type a numerical value explicitly or to click on a label of a grid (e.g., *0*, *row*, *end0*). To select a value that is an expression of *end0*, *row*, etc., we need to click and hold on the labels' part of the grid (e.g., next or above a dimension for horizontal and vertical dimensions, respectively) and then "build" our epxression. For example, see below, where we defined *end0-1*. We can use this "*end0-1*" label anywhere in a statement box simply by clicking on it.

We now click the conditional statement box. *If()* is automatically filled in the box. We click within the parentheses, then click on *row* (from the *Index Variables* list), then click on the ">" sign and type *3*. We just built a simple conditional statement, whereby what follows in the subsequent statement boxes will take place when *row>3*.

So far, we worked with the *Index Range* and *Condition* statement boxes (i.e., added a loop and conditional statement). Now we proceed to the *Formula* statement box by clicking in it. We are going to assign values to the *R* component of our *input_img* grid. To do this, we click on the *R* label of the *input_img* grid. Then, we click the cell at the intersection of *row* and *col* and notice that we have addressed *input_img(row,col,R)*. Now, we insert a

computation for this cell (as looped over *row* and *col* per the *Index Range* statement box. We fill *row*2* (no practical meaning, just for the sake of the example). We do the same thing for components *G* and *B*. First, we add two new formula statement boxes by clicking the *Add Formula* button. We click on the *G* tab and fill in the formula for this component (in our example we use *row*3*) and repeat similarly for the *B* component.

Computation in this step so far occurs for *row>3*. We will add an *else* condition to show how to construct more complex conditional statements. We click the *Add Condition* button. By default this creates an *If()* conditional at the same level as the preceding statement box. By clicking on the *If()* we can see more options at the bottom of the screen: *if*, *else*, *else if*. We click on the *else* button. GLAF recognizes that this *else* corresponds to the closest *if* and matches them accordingly. For more complex conditionals with multiple nesting, the user can use the <= and => buttons (on the column left of the statement boxes), as needed. This will take care of indenting the statement boxes more on the left or the right (as allowed).

For now, we will add three more Formula statement boxes, for the *else* condition, by clicking the *Add Formula* button. For the R component, we intialize to zero. For the G and B components we will use a *library function* and a *user-defined function* to showcase the *function* usage.

First, we will look into library functions. We click on the f_{lib} button and a menu to select a library and function from that library appears, as shown in the next figure.

We select the *Math* library and the *pow* function from the drop-down menu and click the *Insert* button. We see the *Math.pow(number,number)* inserted in the formula statement box.

Module1 :: Main() :: Step1 Title of Step Insert New Step Duplicate Step Delete Step Main Menu

(integer)

R G B

0 col end1

0

row

end0-1

end0

input_img

Index Variables : [row,col,ind2]

Add Source Grid
Add Formula
Add Condition
Delete Formula
<= =>
Options

```
Index Range: foreach row col
Condition:   if ( row > 3 )
Formula:     input_img[row,col,R] = row * 2
Formula:     input_img[row,col,G] = row * 3
Formula:     input_img[row,col,B] = row * 4
Condition:   else
Formula:     input_img[row,col,R] = 0
Formula:     input_img[row,col,G] = Math.pow( number, number )
Formula:     input_img[row,col,B] =
```

+ - * / < > ≤ = != <- OR AND NOT ( ) f_{new} f f_{lib} 123 abc end Delete

We click on the first *number* and then click on *row* from the *Index Variables* list to insert this as the first number. Then, we click on the second *number* and press *2*. This math library function (*pow*) will raise *row* to the power of *2*.

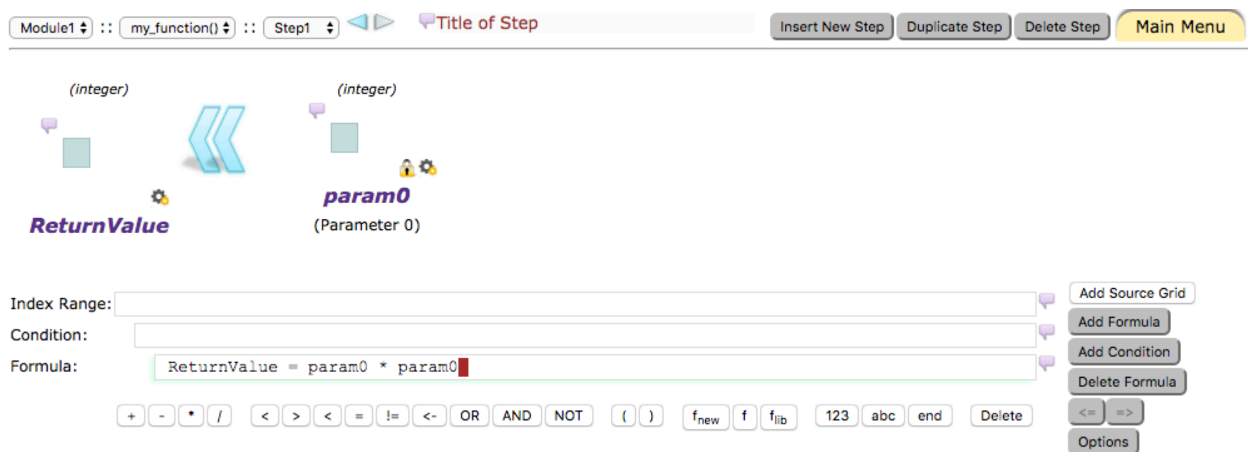
Last, let's see how user-defined functions work. Again, we will create a dummy function that takes an argument and raises it to the power of 2 (like the *pow* library function we used above). We click after the last formula statement box's "=" sign and then click the f_{new} button and give a name to our function. Here, we simply name it *my_function*. We see a function inserted in the statement box (*my_function()*) without any arguments. Now, we may add arguments. In our case, we will pass *row*, so we click within the parentheses and then on *row* from the *Index Variables* list. This will be the only parameter of the function.

Our current step up to this point is shown in the next figure.

To specify what the function we just created does, we will go to the function's drop-down menu at the top of the GLAF GUI and select the corrsponding function.

We are taken at the function header screen. By default, all functions have a return value (integer *ReturnValue*). If our function returns a value this would be the default value to be returned, unless a *return* clause is used with another grid cell value. Parameters are of the type passed during creation of the function and are automatically named *paramX* (where *X* a number). Of course, we can change the parameter name to something more intuitive by doube clicking on the name and typing our preference. Here, we leave as is.

We click *Insert New Step*. We select *Existing Grid* and then *ReturnValue*. As source grid, we select *param0*. Now, as before, we complete the step's computation (see image).



Optionally (since *ReturnValue* is returned by default), we can add another formula statement box to specify the return value. We click the *return* buton and then click *Return Value*. The above would be useful if we wanted to return different values/grids, based on conditional statements, an arithmetic function of a grid cell, etc.

Module1 :: my_function() :: Step1 Title of Step Insert New Step Duplicate Step Delete Step Main Menu

(integer) ReturnValue

(integer) param0 (Parameter 0)

Index Range:
Condition:

```
Formula: ReturnValue = param0 * param0
Formula: return ReturnValue
```

Add Source Grid | Add Formula | Add Condition | Delete Formula | <= | => | Options

+ - * / < > < = != <- OR AND NOT ( ) f_{new} f f_{lib} 123 abc end Delete

Now, we go back to the *Main()* function (by selecting it from the drop-down menu on the top of the GLAF GUI). Since we have concluded speficying computation for the step (including the called user-defined function), we can see the result of our step by clicking on the *Menu* button on the top right of the GLAF GUI, and subsequently selecting *Show Data*.

Module1 :: Main() :: Step1 Title of Step Parallelism: row:7 col:7 Insert New Step Duplicate Step Delete Step Main Menu

(integer)

R | G | B

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	4	4	4	4	4	4	4
3	9	9	9	9	9	9	9
4	16	16	16	16	16	16	16
5	20	20	20	20	20	20	20
6	24	24	24	24	24	24	24

input_img

Index Variables : [row,col,ind2]

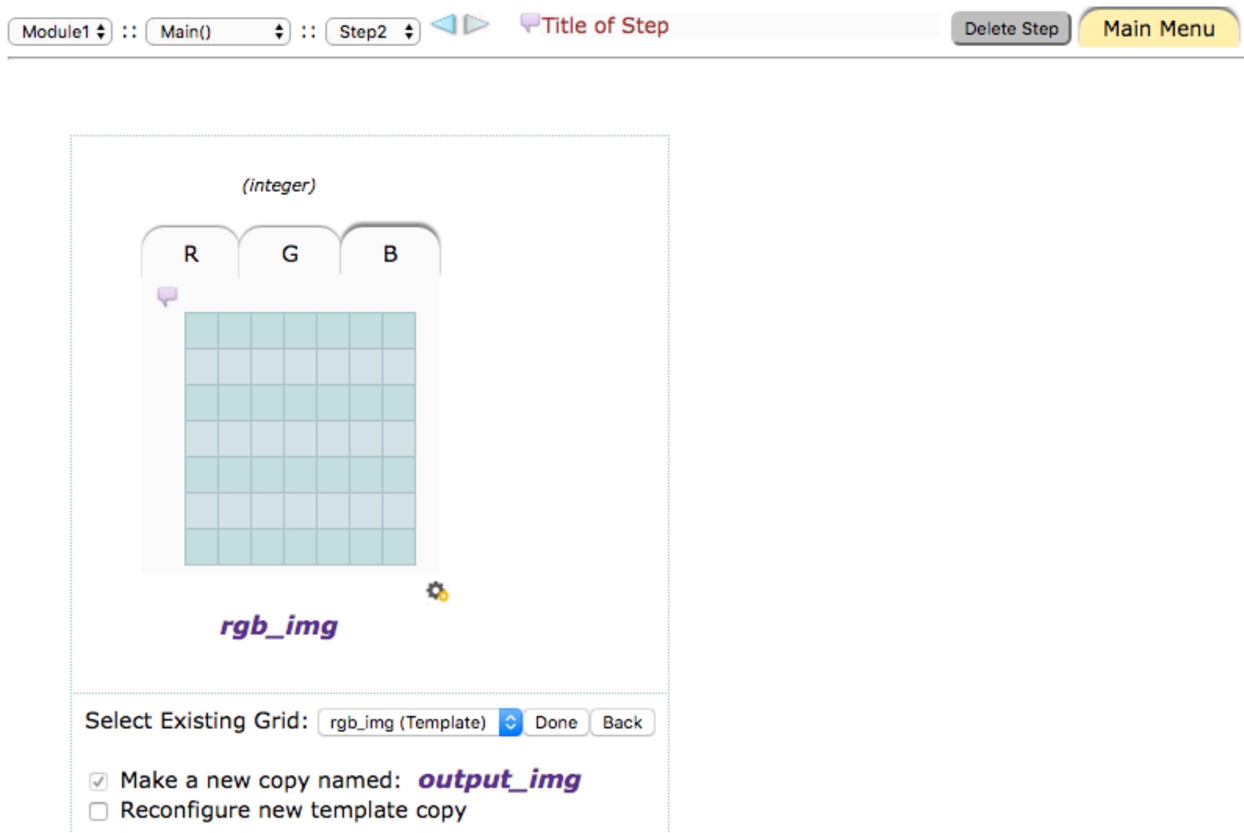
```
Index Range: foreach row col
Condition:   if ( row > 3 )
Formula:       input_img[row,col,R] = row * 2
Formula:       input_img[row,col,G] = row * 3
Formula:       input_img[row,col,B] = row * 4
Condition:   else
Formula:       input_img[row,col,R] = 0
Formula:       input_img[row,col,G] = Math.pow( row, 2 )
Formula:       input_img[row,col,B] = my_function( row )
```

Add Source Grid | Add Formula | Add Condition | Delete Formula | <= | => | Options

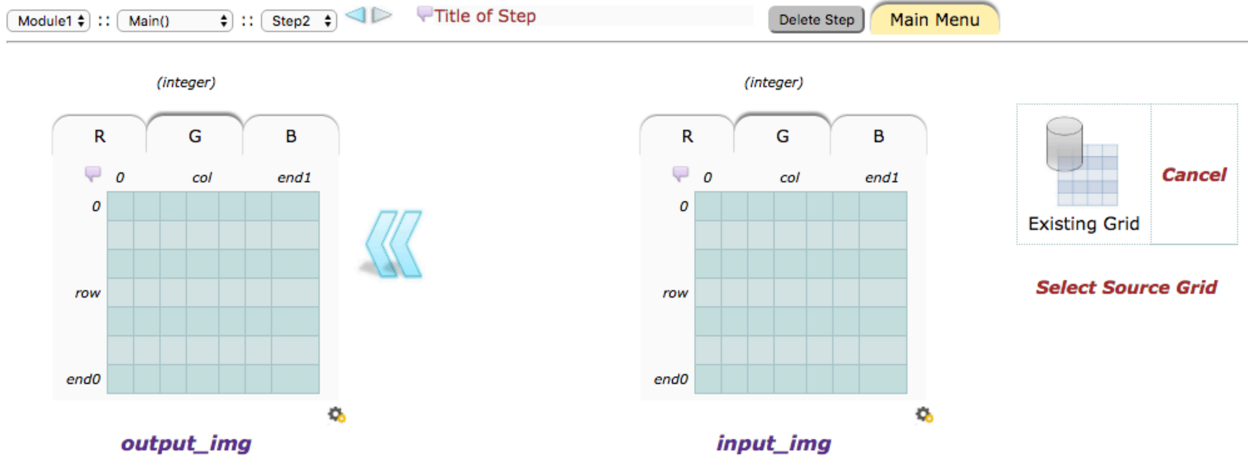
+ - * / < > < = != <- OR AND NOT ( ) f_{new} f f_{lib} 123 abc end Delete

As we can see, the results are evaluated correctly. At this point, if we didn't get the desired output, we should try to identify the problem in our program. Also, you can notice the *parallelism meter* on the top bar of the GLAF GUI that indicates available parallelism in the current step. Specifically, the index variables *row* and *col* are green, which means that both can be parallelized. The number *7* indicates that there are 7 parallel iterations for each of these two index variables. We click *Menu* again and select *Hide Data* to hide the results.

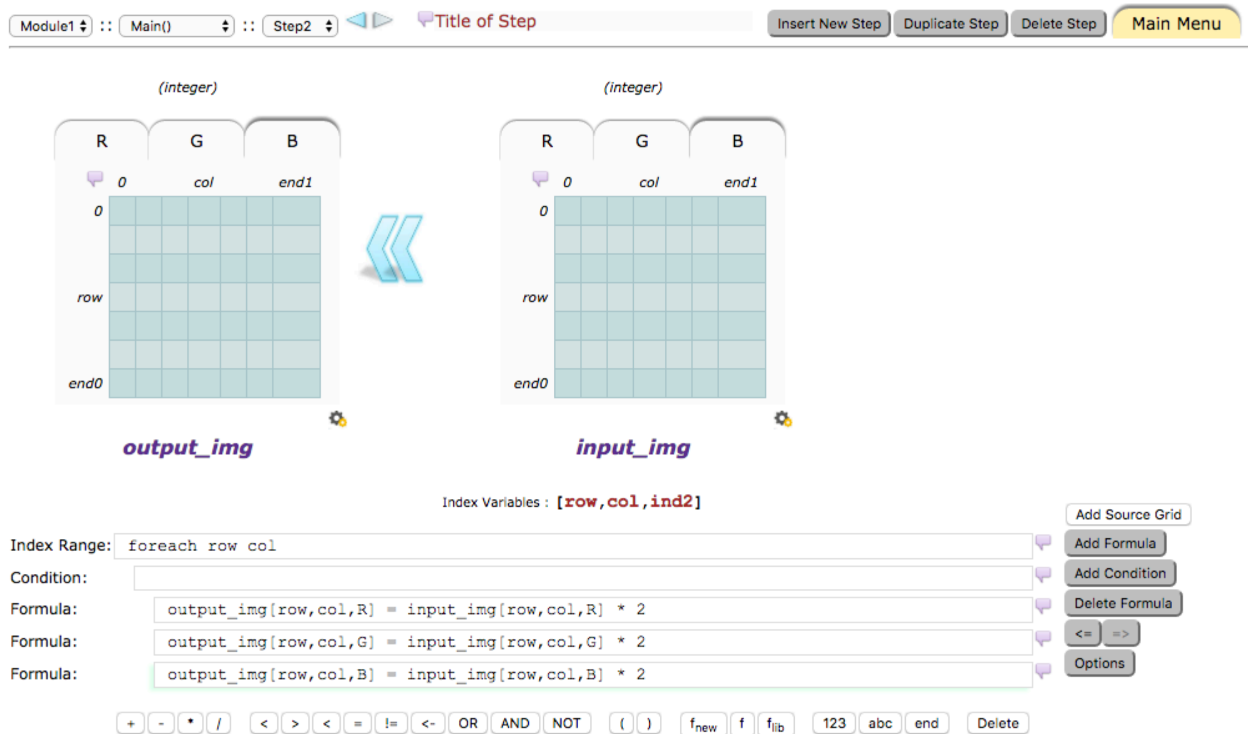
Up until this point, we have initialized our input image. We will insert a new step by clicking *Insert New Step*. Then, we will define a new grid to store the output image (i.e., the result of our computation). As we did for the input image, we will use our template, so as *Output Grid* for this step we select *Existing Grid* and do as before (giving a different name to the new grid: *output_img*). Remember that every grid (as do variables in traditional programming languages) is uninitialized. In our case, we will write on every cell of this newly defined grid, so intialiation is not necessary.



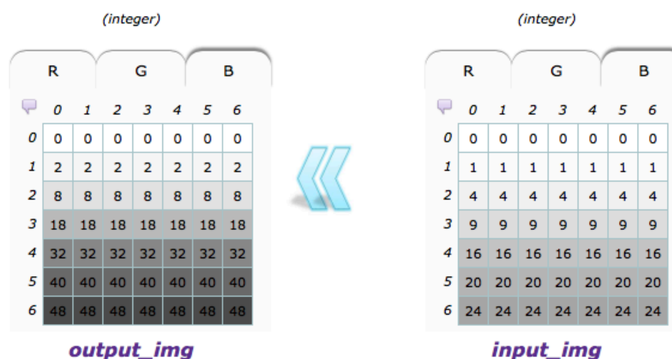
After selecting the *Output Grid* of the step, we select the *Source Grid*. We select the *input_img* grid we created (and initialized) in the previous step. Once we are done, we click *Cancel* to proceed with specifying the computation for the current step.

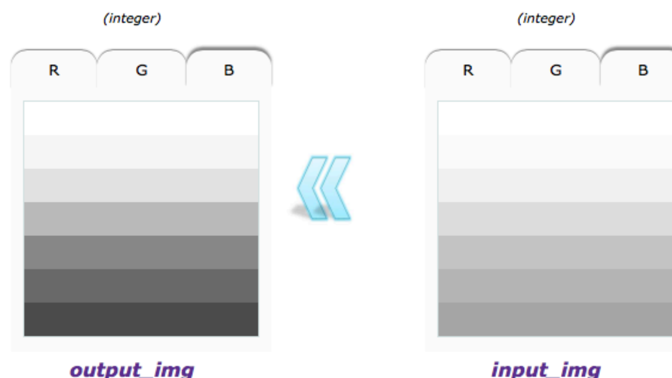


In this step, we will just scale each cell of the input image by multiplying with *2*. We define the desired computation by clicking appropriately on the grids (as we did in the previous step). We show what the step would look like in the figure below.



Clicking on *Menu* and then on *Show Data* from the menu, we can see the results and verify correctness. By clicking *Colorize* or *Data Image* in the menu, we can see the results using combinations of numerical values and/or colors (gray-scale). You can see that the output and input images are gradually darker and that the output image is overall darker than the input image (since we multiplied the input image's values by *2*).





Once we verify our program's correctness, we can generate code by selecting our desired target platform (e.g., CPU), language (e.g., Fortran, C), and any other options (e.g., parallel version of the code) from the *Menu* → *Generate code...* menu item.

Code generation options

Please, select your target platform, target language(s), and data layout optimization below.

Target Platform:
- CPU
- MIC

Target Languages:
- Fortran
- C

Basic Auto-Tuning Options:
- Serial version
- Parallel version (tool-generated)
- Parallel version (compiler-generated)

Extra Auto-Tuning Options:
- Data layout transformations (SoA/AoS)

Generate .glf file

To obtain the code we need to do the following:

1. Click the *Generate .glf file* button. This will perform the appropriate actions to generate the appropriate codes and pack them in a single file (*sourceCodes.glf*) that will be downloaded to your computer.

2. Download the *splitfiles* PERL script in the same directory where you downloaded the *sourceCodes.glf* file (you can find the link in the page from *Generate code...* menu item).

3. Run the PERL script with the command: *perl splitfiles*. The following are generated under the *prog* sub-directory:

    - All code implementations in an appropriate folder structure.
    - A Makefile that can be used to compile all code implementations.
    - A script (runScript.sh) to execute and measure execution time of all code implementations.

4. Run the make command from within the *prog* sub-directory: *make*

5. Run the execution and timing script from within the *prog* sub-directory: *sh runScript.sh*

6. View the execution time results by clicking the *results.html* file created in the *prog* sub-directory (it would open in your default web browser).

If you want to see the code that is generated you can navigate the *prog* sub-directory. For example, if we selected CPU and parallel implementation in C using OpenMP directives, we obtain the code shown below automatically generated:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

int ft_Main(char * *ft_StartUpArgs);
int ft_my_function(int ft_param0);


int ft_Main(char * *ft_StartUpArgs) {
	int ft_ReturnValue;
	int ft_row;
	int ft_col;
	int ft_end0;
	int ft_end1;
	int input_img_R=0, input_img_G=1, input_img_B=2;
	int output_img_R=0, output_img_G=1, output_img_B=2;
	int *ft_input_img;
	int *ft_output_img;
	ft_input_img = (int *)malloc(sizeof(int)*3*7*7);
	ft_end0 = 7-1;
	ft_end1 = 7-1;
	#pragma omp parallel for collapse(2)
	for (ft_row = 0; ft_row <= ft_end0; ft_row += 1) {
		for (ft_col = 0; ft_col <= ft_end1; ft_col += 1) {
			if(ft_row > 3) {
				ft_input_img[input_img_R*7*7 + ft_row*7 + ft_col] = ft_row * 2;
				ft_input_img[input_img_G*7*7 + ft_row*7 + ft_col] = ft_row * 3;
				ft_input_img[input_img_B*7*7 + ft_row*7 + ft_col] = ft_row * 4;
			} else {
				ft_input_img[input_img_R*7*7 + ft_row*7 + ft_col] = 0;
				ft_input_img[input_img_G*7*7 + ft_row*7 + ft_col] = pow(ft_row, 2);
				ft_input_img[input_img_B*7*7 + ft_row*7 + ft_col] = ft_my_function(ft_row);
			}
		}
	}

	ft_output_img = (int *)malloc(sizeof(int)*3*7*7);
	ft_end0 = 7-1;
	ft_end1 = 7-1;
	#pragma omp parallel for collapse(2)
	for (ft_row = 0; ft_row <= ft_end0; ft_row += 1) {
		for (ft_col = 0; ft_col <= ft_end1; ft_col += 1) {
			ft_output_img[output_img_R*7*7 + ft_row*7 + ft_col] = ft_input_img[input_img_R*7*7 + ft_row*7 + ft_col] * 2;
			ft_output_img[output_img_G*7*7 + ft_row*7 + ft_col] = ft_input_img[input_img_G*7*7 + ft_row*7 + ft_col] * 2;
			ft_output_img[output_img_B*7*7 + ft_row*7 + ft_col] = ft_input_img[input_img_B*7*7 + ft_row*7 + ft_col] * 2;
		}
	}

	free(ft_input_img);
	free(ft_output_img);
}

int ft_my_function(int ft_param0) {
	int fun_param0;
	int ft_ReturnValue;
	fun_param0 = ft_param0;
	ft_ReturnValue = fun_param0 * fun_param0;
	return  ft_ReturnValue;
}


int main(int argc, char *argv[]) {
	char *ft_StartUpArgs[4];
	int ft_ReturnValue;
	omp_set_nested(0);
	ft_ReturnValue = ft_Main(ft_StartUpArgs);
}
```

This concludes our GLAF tutorial! For more programs, please, look at the GLAF web-site at http://glaf.cs.vt.edu.