Spice-Weasel photron image enhancement version 1.0

B.Dudson, UKAEA Fusion

July 12, 2006

Contents

1 MIT License

2 Introduction

In order to enhance interesting features present in photron fast-camera images, and to prepare stills for analysis or presentation, a code called "Spice-Weasel" (or just "The Weasel") has been developed. Anyone who watches far too many cartoons will get the name. For everyone else, it's not important, and you'll just have to live with it.

The code can read in either IPX videos as written by sergei shibaev's software or a set of still images (in bitmap or PNG format). How the images are processed is controlled by a script whose syntax has been designed to be (relatively) easy to write, to run efficiently, and to force the result to be sensible. A feature of the code is that it maintains a sliding window centred about a given frame-number, allowing subtraction of several different "background" images. The output is a set of images (again bitmap or PNG format) or an IPX video file. Separate images can be used in presentations or converted to a video using software such as VirtualDub. If both the input and output are IPX files then frame times and header information are preserved, although frame numbers will be different.

The weasel is designed to be as fast as i could make it: The amount of file reading and writing is minimized and the code is multi-threaded (input, processing and output threads) so that it can process images whilst waiting for file I/O to finish. Typically it will process around 10 images per second on the fusion network machines, though this varies depending on the processing steps and the size of the images.

This manual is divided into two initial sections describing how to use the code and the syntax of the processing scripts, followed by a technical section full of details most users won't need or want to know. If you feel like modifying the code, this is the section for you.

To compile the code you will need libpthread, libpng (both standard on the fusion machines), and the openjpeg10 library.

I hope people find this code useful, and if anyone finds a bug or writes improvements, please let me know at ben.dudson@ukaea.org.uk.

3 Using Spice-Weasel

To just run the weasel and see what happens, type

```
> spiceweasel 1000 1050 21
```

This will read in frames 1000 to 1050 inclusive from the default input file (currently set to "`$MAST_IMAGES/rbb/rbb015368.ipx`"). Using a window 21 frames wide, i.e. 10 frames either side of a middle frame, the images will be processed using the default script (see next section). This results in processed frames 1010 to 1040 which are output into the current directory as a set of PNG files named with the frame-number as "`processed_1010.png`" to "`processed_1040.png`".

To set the input there is the "`-i filename`" option. For example, to read in another nice photron file "`$MAST_IMAGES/rbb/rbb015232.ipx`" you could run

```
> spiceweasel 800 899 21 -i $MAST_IMAGES/rbb/rbb015232.ipx
```

To save unnecessary typing and stave off RSI for a little while longer, if you want to read data for a given shot number from the archive, there is the "`-s shot`" option. For example, to process the photron data for shot 15236 from frame 200 to 400 with a window width of 21, run:

```
> spiceweasel 200 400 21 -s 15236
```

In addition to reading IPX files, you may want to process a set of images from a different source. To do this, the files should be in either bitmap (24-bit color) or compressed PNG format. As long as the name of the file includes the frame number it doesn't matter. Instead of giving the name of an IPX file with "`-i`", you specify a C printf format string for the filename. Another example is called for:

If you have a set of images in PNG format named "`15232_frame_0500.png`" to "`15232_frame_1200.png`" in the directory "`../data/`" then to process these you would run:

```
> spiceweasel 500 1200 21 -i ../data/15232_frame_%04d.png
```

The term "`%04d`" will be replaced by the frame number. Just "`%d`" will result in frame 500 becoming input file "`15232_frame_500.png`", The number 4 specifies 4 digits, so "`%4d`" would become "`15232_frame_ 500.png`" (note space before 500). The zero specifies zero padding of the number to produce "`15232_frame_0500.png`". For more info on C printf formatting, see the manual page (`> man -S3 printf`).

The output settings work in the same way as the input except the option is "`-o filename`". To process shot 15100 and output to files of the form "`15100_frame_00200.png`" you would run something like:

```
> spiceweasel 100 300 21 -s 15100 -o 15100_frame_%05d.png
```

(Note 5 digits in frame number, hence `%05d`).

4 Processing scripts

The examples in the previous section used the default script to process the input images. Called "default.sps", this script is in the same directory as the spiceweasel executable.

Processing scripts currently have the extension `SPS` (for Spice-weasel Processing Script) but this is optional. The scripts are not case-sensitive (`INPUT` is the same frame as `input`), and a hash symbol comments out the rest of the line. The syntax will be familiar to anyone who has written a Makefile or coded in Haskell, consisting of blocks of code with the format:

```
<result image>: <input image(s)>
   <processing commands>
              .
              .
              .
```

Each of these blocks takes a set of input images, processes them and puts the final result into another image. One of the blocks has to result in a final output image (called `output`). The simplest possible script which would run is:

```
output: input
```

which just copies the input frame (centre of the sliding window) to the output. In addition to `input`, there are currently two other pre-defined frames: `minimum` and `average`. These are the pixel-wise minimum and average over the sliding window respectively, and can be used as background images because they smooth over transient events like filaments. To subtract the minimum background from the input frame, the processing command `SUBTRACT` can be used:

```
output: input
  SUBTRACT minimum
```

Because the difference between the input and the minimum images is probably quite small (and hence produces a dim image), the image could be amplified (in this case by a factor of 4):

```
output: input
  SUBTRACT minimum
  AMPLIFY 4.0
```

Any number of extra processing steps could be applied to the output, but what if we want to process the image in several different ways then combine them? Instead of writing the processed image straight into the output, we can define a separate image called `difference`:

```
difference: input
  SUBTRACT minimum
  AMPLIFY 4.0
```

This can then be combined with the original input image to produce the output:

```
output: input, difference
```

If more than one image is put after the colon (separated by commas) then these are put side-by-side (from left to right) to produce the image to be processed by the following commands. The above example produces an image with the original on the left and the background subtracted image `difference` on the right. Of course the combined image can be processed further so to gamma correct the output, the script would be:

```
output: input, difference
  GAMMA 1.5

difference: input
  SUBTRACT minimum
  AMPLIFY 4.0
```

The name of defined images can be any string except `input`, `minimum` and `average` since these are already defined. It can contain any characters except space, tab, '#', ':' and ','. Defined frames can be used as inputs for other images, or subtracted from each other. Obviously `output` cannot be used as an input to anything. The order in which the frames are defined doesn't matter. When the weasel first runs it will process the input script, producing errors if anything is wrong. If the script has "compiled" correctly then the syntax is designed so that it will always produce a sensible output without crashing (it may not look very good though!), although i'm sure there are still many bugs to work out. If you define frames but then don't use them then this will not slow down the processing; only frames which are needed to produce `output` are calculated. The following example produces the original frame, the minimum background and the difference (gamma corrected). A frame called `gamma_frame` is defined, but is not needed for the output and so isn't calculated.

```
output: input, minimum, difference

gamma_frame: input
  GAMMA 1.5

difference: input
  SUBTRACT minimum
  AMPLIFY 4.0
  GAMMA 2.0
```

To specify a processing script, use the "-p filename" option. This will search:

1. The local directory
2. The directory pointed to by the environment variable `SPS_PATH`
3. The directory of the executable

For example, say the spiceweasel executable is `/home/me/image_proc/spiceweasel` and you've set `SPS_PATH = '/home/me/sps_scripts/'`. Running spiceweasel from the home directory `/home/me/` with:

```
> image_proc/spiceweasel 1000 1300 21 -s 15368 -p standard.sps
```

will first look for `/home/me/standard.sps`, then `/home/me/sps_scripts/standard.sps` and finally `/home/me/image_proc/standard.sps` and use the first it finds. If none are found then the weasel will give up.

4.1 Processing commands

This section lists the processing commands which can be used in scripts and a brief description of what they do. Some examples of the results are shown using frame 1020 from shot 15368. When processing, pixels are converted into floating point values between 0.0 and 1.0, allowing processing operations and the output precision to be set independently of input precision.

4.1.1 SUBTRACT [subframe]

This subtracts the specified frame `subframe` from the result frame.

$$\texttt{frame} \rightarrow \texttt{frame} - \texttt{subframe}$$

4.1.2 NORMALIZE

Amplifies and offsets an image so that the entire range is used.

$$\texttt{frame} \rightarrow (\texttt{frame} - MIN(\texttt{frame})) \,/\, (MAX(\texttt{frame}) - MIN(\texttt{frame}))$$

This will amplify each frame by a different amount but is good if you don't plan on comparing magnitudes of different output frames.

4.1.3 AMPLIFY [factor]

Multiplies all frames by a constant value.

$$\texttt{frame} \rightarrow \texttt{frame} * factor$$

4.1.4 GAMMA [factor]

Applies gamma correction a frame. If factor is greater than 1.0 then this will enhance low-intensity pixels. Amplification is monotonic, so if one pixel is brighter than another beforehand then it will be after gamma correction too.

$$pixel \rightarrow pixel^{(1/\texttt{factor})}$$

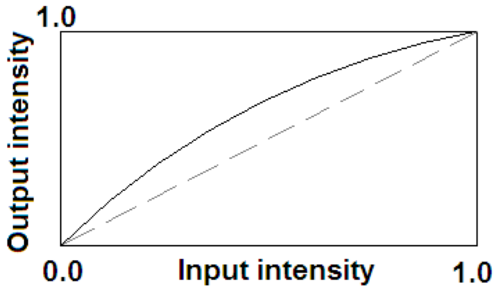

Figure 1: Gamma enhancement, `factor` > 1.0

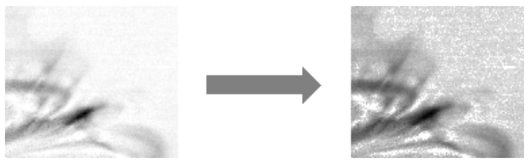

Figure 2: Gamma correction (2.0) of image 1020, shot 15368

4.1.5 OFFSET [value]

Adds a constant value to every pixel

$$\texttt{frame} \rightarrow \texttt{frame} + value$$

One use for this would be if you wanted to reverse the colors. The commands to do this are:

```
AMPLIFY -1.0
OFFSET 1.0
```

Another use is when subtracting frames except the minimum. This may result in negatives which would normally be set to zero at the end. To display both positives and negatives you could offset the frame to 0.5 (say).

4.1.6 GAUSS_BLUR [sigma]

This blurs an image by averaging over a gaussian filter. `sigma` is the standard deviation of the gaussian. The averaging is done over 3 sigma, so this can result in a lot of processing.

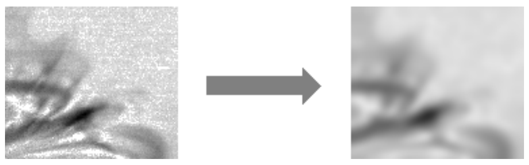

Figure 3: Gaussian blur (`sigma` = 3 pixels)

4.1.7 DESPECKLE_MEDIAN [radius]

For evey pixel in the image this calculates the median value of the pixel and radius pixels around it. For example, `DESPECKLE_MEDIAN 1` calculates the median value of the pixels in a 3x3 grid and puts the result in the central pixel. This is very good at removing "grainy" noise where some pixels are much brighter or dimmer than their neighbours.

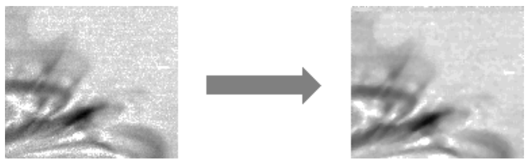

Figure 4: Despeckling (radius 1)

4.1.8 KUWAHARA [size]

This is a non-linear smoothing filter also calculating median pixels, designed to preserve edges. Slower than median despeckling with (i think) no real improvement. Try `KUWAHARA 1`.

4.1.9 SHARPEN [k]

This performs a simple sharpening of an image

4.1.10 UNSHARP_MASK [sigma] [amount]

Performs unsharp masking on the image. A type of sharpening, this can work quite well. I've found that `UNSHARP_MASK 4.0 1.0` are good parameters. Like all sharpening, this will tend to increase noise and so can be combined with a smoothing filter like despeckle.

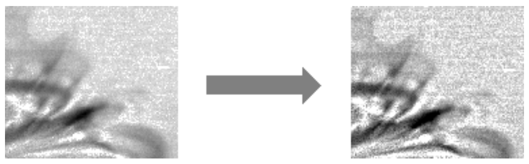

Figure 5: Unsharp masking (4.0, 1.0)

Unsharp masking involves a lot of processing so can take a while (due to gaussian blur step). This command is actually a little superfluous because unsharp masking can be done using the other commands:

```
usharp: input
  SUBTRACT blurmask

blurmask: input
  GAUSS_BLUR 4.0  # this is sigma
  SUBTRACT input
  AMPLIFY 0.5     # this is amount
```

which takes `input` and puts an unsharp masked image in `usharp`.

4.2 Useful script components

In addition to performing basic unsharp masking, scripts can be written to perform more complicated sharpening methods. A modification to the unsharp algorithm which i've found gives quite good results when applied to either the original frame or a differenced frame is:

```
usharp: input
  SUBTRACT blurmask

mask_input: input
  despeckle_median 1

blurmask: mask_input
  GAUSS_BLUR 4.0  # this is sigma
```

```
SUBTRACT mask_input
AMPLIFY 3.0     # this is amount
```

The mask to apply (`blurmask`) is calculated from an image with the noise removed (`mask_input`), and then applied to the original image. This does not then amplify the noise but tends to sharpen features of the image quite well.

5 Technical details

The spice-weasel code is written entirely in C. Parts of it are dependent on POSIX features like pthreads, so it will only compile on UNIX-like systems. It consists of four fairly separate parts: Input, processing, output and syncronisation/book-keeping code which keeps the first three parts working. Parsing and running processing scripts is part of processing, but since it's the largest single bit of source-code, it deserves a sub-section of its own.

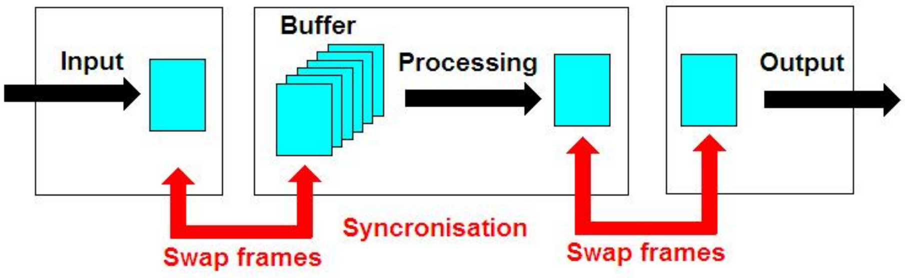

Figure 6: Spice-Weasel block diagram

Figure 6 shows how the parts of the code interact. The input thread reads a single frame into memory (a TFrame structure), the processing thread maintains a circular buffer of frames which are processed to produce an output frame and the output thread writes a frame to disk. These tasks are done simultaneously to speed things up. When a thread is finished it stops and waits for the others to finish too. Once all the threads have finished, the frame read by input is swapped with the oldest frame in the circular buffer, the result from the last processing is swapped with the frame in the output thread then the cycle begins again. When input reads the last frame, it sets a flag in the TFrame structure which tells first the processing thread then the output thread to stop.

Each image is stored in a structure called `TFrame`, defined in `spiceweasel.h`:

```
typedef struct {
  int number;          /* Frame number */

  int width, height;  /* Width and height of frame (image) */
  int allocated;      /* Indicates whether data has been allocated */
  float **data;       /* A 2D array of pixel values */
  int last;           /* Flag indicating last frame */
}TFrame;
```

Pixels are stored in columns as `data[column][row]`. This was to make processing quicker, but at the expense of making reading and writing slightly slower (since images are stored by rows). The allocated flag is there because initially no pixel data is allocated. When a frame is first used, memory is allocated and the flag is set. After this point the size of a frame does not change. This is to minimize the amount of memory allocation and freeing needed (again, speed was the deciding factor here).

5.1 Input

The input has to define three routines:

- `void read_init()` Called once at the start of the program, this should initialize any variables needed for reading. If reading an IPX file, this code reads the IPX file header so that frames can be read quickly later.
- `int read_frame(int number, TFrame *frame)` This is called every cycle and specifies which frame number to read and the location to put the data. Returns 0 if successful.
- `void read_finish()` is called once at the end of the code and should generally clean up. Currently just closes the input IPX file if used.

5.2 Processing

The processing code only has to supply two routines:

- `void process_init()` sets up any temporary variables needed during processing. For all but the simplest processing, intermediate frames are needed and this routine marks them as unallocated so that the memory is allocated when needed.
- `int process_frames(TFrame **framebuffer, int nframes, int centreframe, TFrame *output)` Takes an array of frames (the frame buffer), the

number of frames in the buffer, the index of the frame in the centre of the window and the location of the output frame.

A source file which supplies these is `process_main.c` and is not currently used, having been replaced by processing scripts (next sub-section). Nevertheless it's still there and useful for testing etc. To compile a version which uses this old code, just edit the `Makefile`, replacing `run_script.o` with `process_main.o`.

5.3 Processing scripts

This code takes a makefile-like processing script and produces a set of commands in a form which can be quickly executed for each frame. This is done in three stages:

- **Reading**: `parse_newline()` reads the script, removing white-space at the start of a line and comments and reducing all white-space to a single space. Letters are capitalised. The first line containing useful characters is returned along with the line-number.
- **Processing**: `parse_script()` first looks for target definitions which are the only lines containing colons (after target name). If the line is a target then a new target definition is begun (a `TTarget` structure), provided the name has not already been defined. If the line is not a target then it is checked against a list of commands and, if valid, added to the current target. All function names are converted into ID numbers which are defined in `script.h`, for example "AMPLIFY" is changed to the constant `PROC_AMPLIFY` (currently defined as 2). The end result is an array of targets with a list of dependencies and processing steps.
- **Resolving**: `resolve_script()` is a recursive algorithm which starts by finding the `output` target and its dependencies then the dependencies' dependencies and so on. When a target is found which depends only on defined targets (such as `input`) it is assigned a temporary storage and the list of commands needed to create it are copied to the final list. The result is a list of commands on temporary storage IDs instead of names which results in the required `output`.

As an example, here's a script for unsharp masking the input:

```
# Modified unsharp masking with some despeckling

mask_input: input
```

```
    despeckle_median 1

blurmask: mask_input
    GAUSS_blur 4.0 # this is sigma
    subtract mask_input
    # at this point this is the change
    # produced by bluring the image
    amplify 3.0  # this controls how much the image
                 # will be adjusted.

output: input
    # reverse change produced by bluring
    SUBTRACT blurmask
```

The first (reading) stage will reduce this to:

```
MASK_INPUT:INPUT
DESPECKLE_MEDIAN 1
BLURMASK:MASK_INPUT
GAUSS_BLUR 4.0
SUBTRACT MASK_INPUT
AMPLIFY 3.0
OUTPUT:INPUT
SUBTRACT BLURMASK
```

The second (parsing) stage will find the three lines containing colons and take these as definitions of `MASK_INPUT`, `BLURMASK` and `OUTPUT` targets. When an input is declared after a colon or used as input for a command (such as `SUBTRACT MASK_INPUT`), it is added to the dependency list for the target. This results in a tree-like structure shown in figure 7 where the dependencies are shown in red.

To resolve these dependencies and produce a list of commands to execute, the function `resolve_script()` calls `resolve_script_rec(char *name)`, passing `output` as the target name to resolve. This determines what targets `output` depends on and calls `resolve_script_rec(char *name)` with the names of those targets. The code works by travelling down the tree from `output`, following dependencies and then retracing its steps whilst copying the processing commands needed and allocating temporary storage spaces. For the current script, this produces the following steps:

1. First dependency of output is input. This is already defined so is set to a special identifier `INPUT_FRAME`.

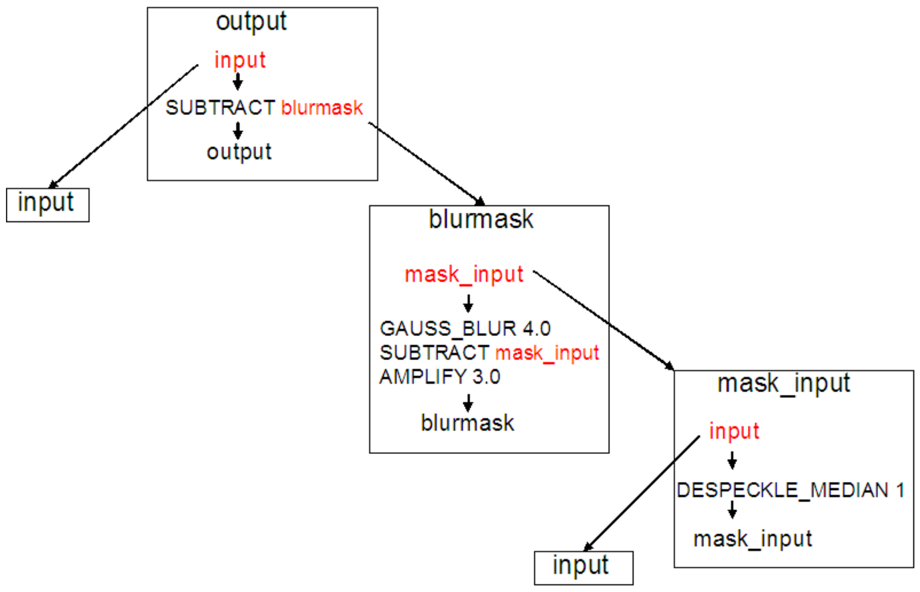

Figure 7: Target dependencies for unsharp masking script

2. Call `resolve_script_rec()` to resolve dependency `blurmask`
3. `blurmask` depends on `mask_input` so call `resolve_script_rec()`.
4. `mask_input` only depends on `input`, again set to `INPUT_FRAME`.

Now that `mask_input` has all its dependencies, the processing steps are copied into the final command list. Every function has an input and output frame and a number of arguments (which may be frames too). In this case the function is `DESPECKLE_MEDIAN`, and the input is set to `INPUT_FRAME`. Every target must have its own storage space which are given numbers, so `mask_input` is set to storage space ⟨0⟩ which is the output of `DESPECKLE_MEDIAN`. The command list is now:

```
Despeckle(INPUT, 1) => <0>
```

At this point `mask_input` has been assigned a storage space and so all dependencies of `blurmask` have been resolved. The set of commands for `blurmask` are are put at the end of the list. The first command, `GAUSS_BLUR` cannot have the same output as input since it operates on several points, and so the next available storage space ⟨1⟩ is used as output. `SUBTRACT` and `AMPLIFY` can have the same input as output, and so both operate only on ⟨1⟩. Finally, the end result is in ⟨1⟩ and so `blurmask` is given ID 1. Now output has all its dependencies and so the same thing happens, with the final result being given the special identifier `OUTPUT_FRAME`.

```
Despeckle(INPUT, 1) => <0>
Gaussian blur(<0>, 4.000000) => <1>
<1> - <0> => <1>
<1> * 3.000000 => <1>
INPUT - <1> => OUTPUT
```

In order to prevent circular dependencies, when a target is being resolved a flag (`TTarget.resolving`) is set so that if the code tries to resolve the same target twice without succeeding there must be a circular dependency and so an error is reported.

The end result of processing a script is a `TCommands` structure:

```
typedef struct { /* Set of sequential commands for processing frames */
  int ntemp;         /* Number of intermediate frames needed */
  int minimum_frame; /* The ID of the minimum frame */
  int average_frame; /* ID of average frame */
  int nsteps;        /* Number of processing steps */
  TProcess *step;    /* List of processing steps */
}TCommands;
```

If `minimum` or `average` are needed then their ID is set to be a storage location i.e. ≥ 0, otherwise they are not calculated. When `process_init()` is run, an array of `ntemp` frames is created (called `tmp_frame[]`) and frame IDs become indices into this array. Each `TProcess` structure consists of

```
typedef struct {  /* Define a processing step */
  int method;       /* Which method to use */
  int nargs;        /* Number of arguments */
  int input;        /* Input frame */
  TProcArg *args;   /* Array of arguments */
  int result;       /* Result frame */
}TProcess;
```

Because everything is specified as integers, a quick `switch` statement can be used to select processing method. The function `get_frame()` checks if a frame ID is is one of `INPUT_FRAME` or `OUTPUT_FRAME` or an index into `tmp_frame[]` and returns a pointer to the frame. The arguments are stored as:

```
typedef struct { /* An argument to a processing step */
  char *name;   /* Name of frame (not used when processing) */
  float fval;
  int ival;
  int frame;    /* ID of frame */
}TProcArg;
```

and so can be an integer, a float or a frame ID. The code which runs the processing steps assumes that the script parsing code has worked correctly and that the correct number and type of arguments are present without performing any checks.

5.3.1 Adding a processing command

If you have a favourite image processing algorithm which i haven't included and which can't be scripted then adding a new command may be worth it.

1. The first step is (obviously) to write the function to process the frames - see the file `process_frames.c` for the current set of functions. I strongly recommend using the following code:

   ```
   if(allocate_output(width, height, output)) {
     return(1);
   }
   ```

   where `output` is the output `TFrame*` pointer. This function checks if the output frame has been allocated and if not then allocates memory. If it has, the code checks that the sizes match and if not then returns nonzero. This ensures that frame sizes are consistent and that memory is allocated once only. Other useful functions are `shell_sort()` and a set of routines for defining and applying spatial filters to images which are used by `gauss_blur()`. Once you have written the function, put a prototype in `spiceweasel.h` with the others.

2. Around line 25 in `script.h` there is a list of `#define` statements like `#define PROC_KUWAHARA 6`. Add a new identifier for your function in the same way. If your function cannot have the same output as input then you need to add the identifier to the array `int proc_noio[6]` before the last `PROC_NULL`.

3. In function `parse_script` in file `process_script.c` starting around line 387 are a set of string comparisons like:

   ```
   }else if(strcmp(buffer, "AMPLIFY") == 0) {
     curproc->method = PROC_AMPLIFY;
     /* Should have one floating-point argument */
     if(nprocargs != 1) {
       printf("Error line %d: Amplify has one argument\n", linenr);
       return(1);
   ```

```
    }
    if(add_floatarg(curproc, procarg[0])) {
      printf("Error line %d: Argument to amplify is a floating point number
      return(1);
    }
  }
```

Add a similar section (remember the string must be in capitals). The number of arguments passed is in `nprocargs` and the arguments are stored as an array of strings `procarg[]`. Set `curproc->method` to your function identifier defined in script.h. For each argument you can use the functions `add_framearg`, `add_floatarg` and `add_intarg` to add a frame, float or integer argument respectively. These functions return non-zero in the event of an error and should be added in the order in which they appear in your function call.

4. Add a `case` statement for your function to `process_frames()` in run_script.c around line 137. This is a switch on function identifier (as defined in `script.h`). Already defined are `TFrame` pointers `in` and `out` for the input and output frames. `proc->args[]` is an array of `TProcArg` structures for the arguments with `ival` and `fval` elements for integer and floating point values respectively. To pass a pointer to a frame, use the macro `GETFRAME(proc->args[i].frame)`.

5. (optional) Add your function to `dummy_script()` in `process_script.c`. This prints out the processing steps to be performed at the start of the program but does no actual processing.

5.4 Output

Like the input, the output code defines three routines:

- `void write_init()`
- `int write_frame(TFrame *frame)`
- `void write_finish()`

5.5 Syncronisation code