

The Filter Foundry

Revision: 11 November 2024 by Daniel Marschall

Table of Contents

Introduction.....	2
Downloading and installing Filter Foundry.....	3
Windows version 1.7	3
Mac OS version 1.6.....	3
Compatibility	3
About digital images.....	4
Non-RGB modes	4
16-bit color mode and 32-bit color modes.....	5
Creating custom filters	6
Creating a filter from scratch or loading an existing filter	6
Making and deploying standalone filters	7
Protecting filters.....	7
Supported file formats	8
Expressions for creating filters	8
Components of expressions	8
Number constants	8
Variables	8
Functions	10
Operators.....	14
Comments	18
Providing user-controlled sliders.....	18
Examples.....	19
Affecting a single channel (filter).....	19
Affecting channels using sliders (filter)	19
Adding noise to channels using sliders and random values (filter).....	19
Amplifying or toning down channels (filter).....	19
Averaging the channel values of neighboring pixels (filter)	20
Implementation detail differences.....	20
YUV color space variables.....	20
Polar coordinate system D, dmin, dmax	21
get(i) function	21
r, g, b of fully transparent pixels.....	21
rst(i) function	21
Evaluation of conditional branches	22
License	23

Introduction

Filter Foundry is a compatible replacement for **Filter Factory** (developed by the Adobe employee Joe Ternasky in 1994). Initially written by Toby Thain ([Telegraphics](#)) in 2003 - 2009, the development has been continued by [Daniel Marschall](#) ([ViaThinkSoft](#)) since 2018. Several advancements and improvements have been made, such as support for 64-bit Windows, and additional color modes including 16-bit color and 32-bit color.

With Filter Foundry, you can create your own plugins by determining how you want the filter to affect the channels (red, green, blue, and alpha) of each pixel in the image by specifying arithmetic expressions.

The filters you create can also include Settings dialog boxes, which provide up to eight sliders for adjusting the filter's effect. When you design a filter, you include user-supplied slider information in the expression. You also determine the number of sliders and whether they appear in the Settings dialog box individually or in pairs.

When you create a filter, you can save its expressions in a text file ("AFS parameter file"). Doing so lets you use the Filter Foundry to edit the expressions later. Filter Foundry can also extract the expressions out of standalone plugins created by Filter Factory or Filter Foundry, as long as the plugin is not protected by the creator.

The plugin is free and can be downloaded at:

https://www.viathinksoft.com/projects/filter_foundry

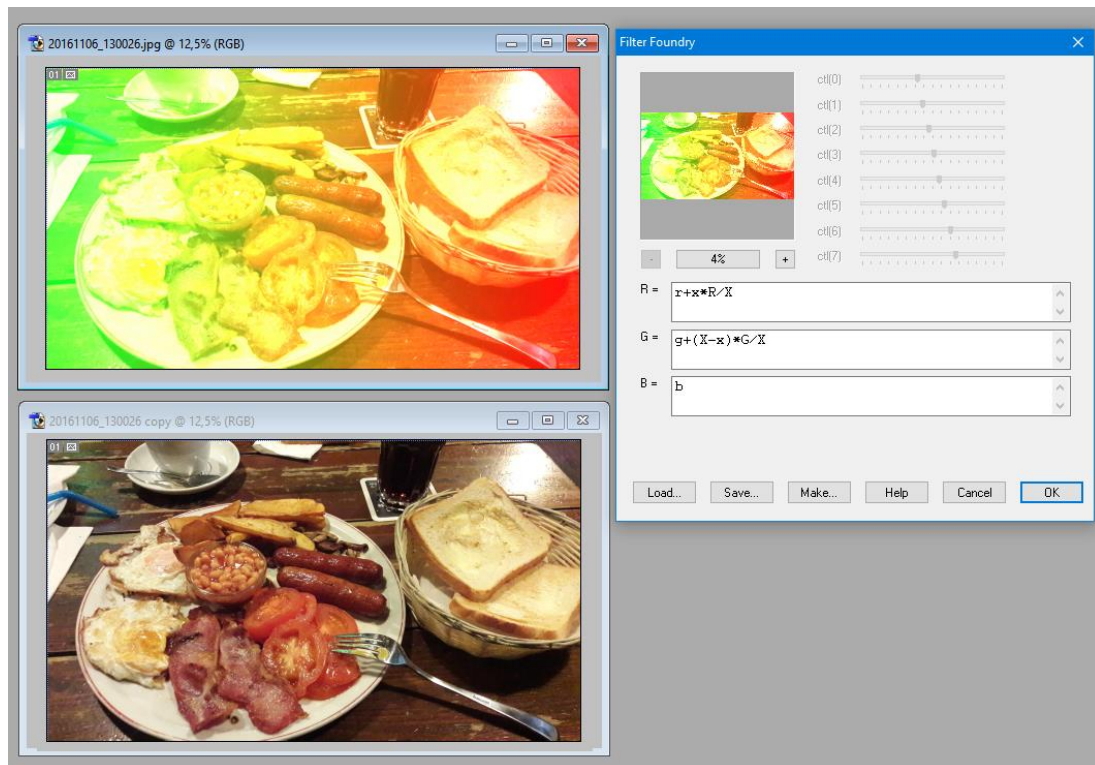
Since the plugin is licensed under the license terms of the GNU General Public License, Version 2, you can download the source code and modify it:

https://github.com/danielmarschall/filter_foundry/

More information about the Filter Factory can be obtained from The Filter Factory Programming Guide by Werner D. Streidt and Harald Heim. The guide is for Filter Factory, but most parts also apply to Filter Foundry.

<https://thepluginsite.com/knowhow/ffpg/ffpg.htm>

The following picture shows an example of a filter created using only a few keystrokes!



Downloading and installing Filter Foundry

Windows version 1.7

Filter Foundry 1.7 comes with a 32-Bit Windows plugin (FilterFoundry.8bf) and a 64-Bit Windows plugin (FilterFoundry64.8bf).



FilterFoundry.8bf



FilterFoundry64.8bf

To install the plugin to Photoshop, simply place the appropriate 8BF file into the **Plug-Ins\Filters** subdirectory of your Adobe Photoshop program files path and restart Photoshop. For other host applications like GIMP, look at the manual on how to install “.8bf” Photoshop filters. If you have information on whether or whether not Filter Foundry works on a non-Photoshop host application, please send us a message!

Mac OS version 1.6

The Mac OS version could not be taken over because Apple removed the “Carbon” API, and the new “Cocoa” API is not compatible with the current code-base. ***If you would like to help porting Filter Foundry to the latest OS X version, it would be highly appreciated!*** An old version of Filter Foundry for Mac 68k (requires 68020 or later CPU and FPU) and Mac Classic (PowerPC) can be obtained here: <https://www.telegraphics.net/sw/> or at [GitHub](#).

Compatibility

Filter Foundry works with nearly all programs that can handle “.8bf” Photoshop filters.

Tested with the following hosts:

- Photoshop 3.0.0 (32 bit) through 2025 (64 bit)
- IrfanView 4.53 (32/64 bit)
- JASC PSP 9, Corel's Paint Shop Pro XI
- The Gimp 2.2 with [PSPI.exe](#) extension to run Photoshop .8bf files
- Serif PhotoPlus 6
- PluginCommander 1.62 (Revision 2)
- Paint.Net using [PSFilterPdn](#) to run Photoshop .8bf files

Also recognized by the following code emulators:

- [pdn-filter-factory](#)
- I.C.NET Plugin-Manager 2.x
- Filters Unlimited
- Plugin Galaxy

Tested with the following Operating Systems:

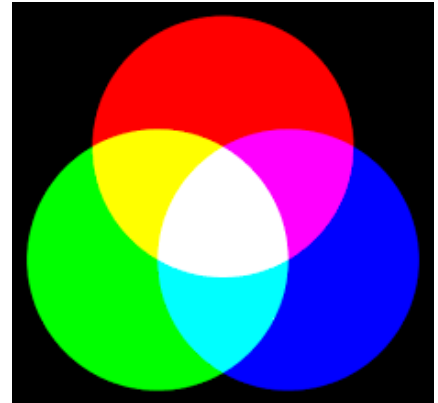
- Windows 95, 98, Me, NT4, 2000, XP, Vista, 7, 8, 8.1, 10, and 11 are fully compatible (32- and 64-bit).
- Windows 3.11 (using Win32s), Windows NT 3.1, and Windows NT 3.5x are compatible (without preview panning and help button does not work)
- Linux using the emulator “Wine” is supported
- Mac OS is currently not supported

About digital images

A digital image is a conglomeration of tiny picture elements, called *pixels*. Pixels project the color and brightness of the image. Each pixel in an image is uniquely identified by its coordinates. The first coordinate is the horizontal position of the pixel (“x”), and the second coordinate is the vertical position of the pixel (“y”). The horizontal coordinates start counting at the left edge of the image and increase as you move to the right. The vertical coordinates start counting at the top of the image and increase as you move down. Therefore, the top left corner of the image has the coordinates (0,0). The range of coordinates for an image depends on its resolution.

In **RGB color mode**, the color of a pixel is stored as three numbers: the amount of red, the amount of green, and the amount of blue. The three-color values are called *channels*. Channel values can range from 0 to 255. Pixels can have a fourth channel, an alpha channel, which controls the transparency of the image.

1. If a channel value is set to 0, none of its colors is present in the pixel.
2. If a channel value is set to 255, the maximum amount of that color is present in the pixel. For example, if a pixel has the channel values (255,0,0), the pixel is entirely red: 255 red, 0 green, 0 blue.
3. If all three channels have the same value, the pixel is a shade of gray. For example, (80,80,80) is a dark gray, (128,128,128) is a medium gray, and (200,200,200) is a light gray.
4. If all three channels are 0, the pixel is black. If all three channels are 255, the pixel is white.



The filters you create affect the channel values of the pixels in an image. You specify an expression for each channel, and each operation is performed on the appropriate channel for every pixel in the image. Expressions can include specific pixel coordinates whose channel values are evaluated and used in the calculation.

Note: If an expression evaluates to a number greater than 255, the channel is set to 255.

Likewise, if an expression evaluates to a number less than 0, the channel is set to 0.

Non-RGB modes

Unlike Filter Factory, Filter Foundry can also work with non-RGB mode images. However, there are some things to consider if you are using an image mode different than RGB:

1. The variables “r” (red), “g” (green), “b” (blue), and “a” (alpha) lose their original meaning and need to be interpreted as “channel 1”, “channel 2”, “channel 3”, and “channel 4” (see table below).
2. The variables “i”, “u”, and “v” of the YUV are meaningless since they are calculated based on “r”, “g”, “b”, assuming they are RGB. Hence, you must not use “i”, “u”, or “v” when working with non-RGB images.

For example, if you have a **CYMK image**, the channels in the Graphical User Interface will have the labels “C”, “Y”, “M”, and “K”. However, the formulas still must contain the variables “r” (for channel #1), “g” (for channel #2), “b” (for channel #3), and “a” (for channel #4). So, with CYMK, the expression “r” would refer to the Cyan channel, “b” refers to the “Yellow” channel, “g” refers to the “Magenta” channel, and “a” refers to the “Black/Key” channel. Unfortunately, the 5th channel (Alpha-channel) cannot be used in this color mode.

Other image modes are the CIELAB color space (channels “L”, “a”, and “b”), Duotone, Greyscale, and Multichannel (where only up to 4 channels can be used with Filter Foundry), which are listed in the following table:

Image Mode	“r” represents	“g” represents	“b” represents	“a” represents
RGB (8/16/32 bits)	R (red)	G (green)	B (blue)	A (alpha)
CMYK (8/16 bits)	C (cyan)	M (magenta)	Y (yellow)	K (black)
Gray (8/16/32 bits)	K (black)	(unused, zero)	(unused, zero)	A (alpha)
Multichannel (8/16 bits)	1 (channel 1)	2 (channel 2)	3 (channel 3)	4 (channel 4)
Duotone (8/16 bits)	D (tone)	(unused, zero)	(unused, zero)	A (alpha)
Lab (8/16 bits)	L* (lightness)	a* (green-red axis)	b* (blue-yellow axis)	A (alpha)

Not supported are the color modes Bitmap (1-bit image) and Indexed Colors.

16-bit color mode and 32-bit color modes

(Note: This chapter is intended for advanced users; please read the other chapters first to learn about the basics of formulas, constants, variables, etc.)

The following table shows how Filter Foundry handles different color bit depths:

Bit Depth	Internal Range (Integer)	Clamped Output Range*	Output Channel Range*
8-bit	-2,147,483,648 – 2,147,483,647 (32bit signed integer)	0 – 255	0 – 255
16-bit	-2,147,483,648 – 2,147,483,647 (32bit signed integer)	0 – 32,768 (15bit+1; has an integer midpoint)	0 – 32,768
32-bit	-2,147,483,648 – 2,147,483,647 (32bit signed integer)	0 – 8,388,608 (23bit+1; has an integer midpoint)	0.0 – 1.0 (float)

Internal Range: All calculations use signed integer values from -2,147,483,648 to 2,147,483,647, regardless of the color mode.

Clamped Output Range: In each color mode, values above the specified maximum become that maximum, and values below 0 become 0. For 32-bit, due to technical limitations described below, 8,388,608 has been chosen instead of a higher value.

Output Channel Range: For 8-bit and 16-bit, the output channel range is equal to the clamped output range. For 32-bit, the clamped output range (0 – 8,388,608) will be normalized to the Photoshop float representation 0.0 – 1.0.

* *Note:* Special case for **Lab color mode:** The channel L* as well as the alpha channel have an output range of 0 – 255 for 8-bit and 0 – 32,768 for 16-bit, while the a* and b* channels have an output range of -128 – 127 for 8-bit and -16,384 – 16,256 for 16-bit. In Filter Foundry you should use these values. A 32-bit mode for Lab does not exist.

⚠ Attention: Due to the higher output range (such as 32,768 for 16-bit filters instead of 255 for 8-bit filters), filters with the same formulas will most likely look different depending on the chosen color space, which may be an undesired result.

The plugin author should consider the following steps to make sure the filter looks equal with all bit depths:

1. Take care not to hardcode the value 255 to define a maximum channel value, and instead use the constants “R”, “G”, “B”, “A”, or “C”. Note that R, B, and C are not equal to G and B in Lab color mode.
2. Functions like “sin”, “cos”, and variables such as “x”, “y”, “d” (angle) and “m” (distance) do not change their output value, while functions like “i”, “u”, “v” do change their output value based on the bit depth (because the “i”, “u”, and “v” variables are calculated based on the channel values “r”, “g”, and “b”).
To make a variable such as “d” and “m” look equal in any bit depth, the plugin author should use the following technique to make an 8-bit expression compatible with 16-bit and 32-bit:

Search:	m
Replace with:	max(0,min(255,m)) * C / 255

Explanation:

- “max(0,min(255,m))” takes care that values smaller than 0 become 0 and values greater than 255 become 255. This is important because in 8-bit filters, variables like “m” get maxed out at 255, while for 16-bit they get maxed out at 32,768 and for 32-bit they get maxed out at 8,388,608. To get the same result as in 8-bit filters, the output will be clamped to the range 0 – 255.

- “C” is the channel’s maximum value such as 255 for 8-bit, 32,768 for 16-bit, and 8,388,608 for 32-bit. 255 is the 8-bit channel’s maximum. Multiplying a variable/constant/formulae with the desired color mode’s maximum range and the dividing by 255 (the 8-bit maximum range) should result in the filter looking similar in 8-bit, 16-bit, and 32-bit. Take care to first multiply and then divide, because the division is applied to integers, so there might be a rounding.
Note that “xxx * C / 255” is only safe if $xxx \leq 255$, because of the following edge case: Internally, Filter Foundry internally calculates using 32bit signed integers (max value 2,147,483,647). For 32-bit, the value C = 8,388,608. Multiplying 255 with 8,388,608 gives 2,139,095,040 which is inside the signed 32bit integer range. Multiplying 256 or greater with 8,388,608 will cause an integer overflow and therefore results in wrong values. This is the reason why the step “max(0,min(255,m))” must be combined with “xxx * C / 255”.

Alternatively, if you don’t want the expression get clamped to the range 0..255, calculate “C/255” first to avoid an integer overflow during multiplication. The 16/32-bit result should be similar but not equal to the 8-bit result:

Search:	m
Replace with:	m * (C / 255)

Creating custom filters

The procedures in this section explain how to use the Filter Foundry to load, apply, edit, and save filters.


Creating a filter from scratch or loading an existing filter

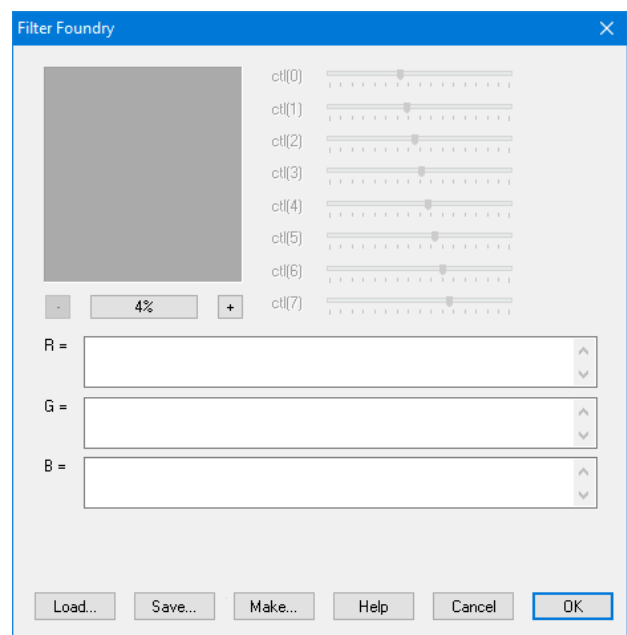
1. Choose **Telegraphics > Filter Foundry...** from the Filter menu. The Filter Foundry Settings dialog box appears.

2. If you want to load an existing filter to edit and/or apply it, click **Load**. Use the Open dialog box to load a file. You can open the following file types: AFS, 8BF, PFF, PRM, FFX, FFL, GUF, TXT (see section “Supported file formats” for descriptions). If you want to write a filter from scratch, continue with step 3.

3. Specify an expression for each channel in the channel fields. Even if you specify the same expression in all three channels (“R”, “G”, “B”), their evaluations will probably be different. The dialog box will include channel fields for any alpha channels in the image. If you are working in a layer, a channel field (“A”) will appear in the dialog box.

For information on how to use expressions to achieve a result, see “Expressions for Creating Filters”.

As you type an expression, a small yellow caution sign  appears. It will remain visible until you have typed a legal expression. If the caution sign does not disappear, it means that there is an error in the expression. To see which part of the expression is in error, click the caution sign to see the error message and select the incorrect portion.



4. If the expressions include user-supplied slider information, drag the appropriate Map sliders to preview the effects. The Map 0 sliders correspond to sliders 0 and 1; the Map 1 sliders correspond to sliders 2 and 3, and so on. For information on including user-supplied slider information in expressions, see “Providing User-Controlled Sliders”.

5. When you have correctly set up the filter, click **Save** to save the expressions in a text file (“AFS parameter file”). Saving the expression allows you to load and edit the filter in the future. (However, you can also load the standalone filter as long as it is not protected).

6. If you want to use this one instance of the filter only, click **OK** to apply the filter. If you want to use the filter more than once, see the next procedure, “Making and deploying standalone filters”.

Making and deploying standalone filters

1. Follow steps 1 through 4 of the previous procedure, “To create a custom filter.”

2. Click **Make**. The “Make Standalone Filter” dialog box appears.

3. Name the filter using the **Category** and **Title** fields. The name will appear in the Filter menu under the submenu specified in the Category field.

4. Use the **Copyright** and **Author** fields to include credits or copyright information in the filter’s **About...** dialog box.

5. If the filter’s expressions include user-supplied slider information, select the appropriate number of Control or Map options and specify labels for the sliders in the corresponding text boxes. The labels will appear with the sliders in the filter’s Settings dialog box.

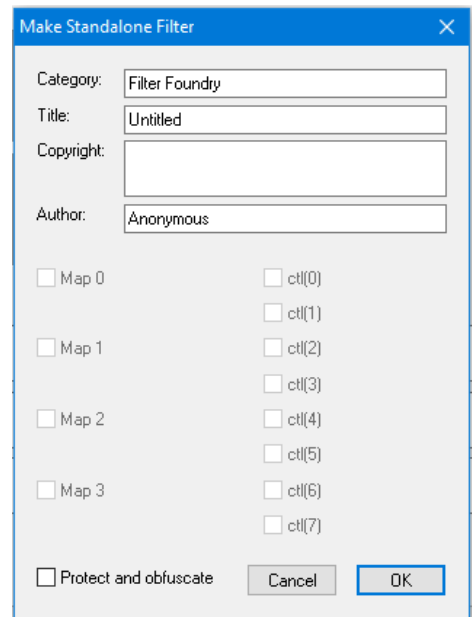
To display the sliders individually in the Settings dialog box, use the Control options. To display the sliders in pairs, use the Map options. Whether you should use individual or paired sliders depends on the type of filter you are creating and the expressions you have written.

6. Click **OK**. A standard Save dialog box appears. Save the filter somewhere on your disk and/or publish it accordingly.

Filter Foundry will automatically create a 32-bit and a 64-bit filter for you. The 64-bit filter will always have the suffix “64”. For example, if you name your filter “Example”, then two files will be created: “Example.8bf” (32 bit) and “Example64.8bf” (64-bit).

7. Copy the appropriate 8BF file into the Adobe Photoshop Plug-Ins folder.

8. To make the filter available, restart the Adobe Photoshop program.



Protecting filters

In the step of creating a standalone filter, you can select the option “**Protect and obfuscate**”. This will encrypt the filter formulas in the plugin (to avoid that the formulas are read by hex-editors or other tools) and protect it to avoid that it can be opened using Filter Foundry.

⚠ Attention: Please make sure that you save your filter in an TXT or AFS parameter file since you will NOT be able to recover the formulas from a protected 8BF plugin file! Note that it is *technically* possible to decrypt a protected filter; however, it is rather complex and needs software development skills. The plugin author did his best to make the protect feature as secure as possible and did not create an unprotect/decrypt tool.

Supported file formats

Filter Foundry support a variety of different file formats:

File extension	Description	"Make"	"Save"	"Load"
AFS or TXT	Parameter file created by Filter Foundry or Filter Factory for Adobe Photoshop .		Yes	Yes
8BF	Adobe Photoshop plugin created by Filter Foundry or Filter Factory for Adobe Photoshop .	Yes		Yes ¹
PFF	Parameter file created by Filter Factory for Adobe Premiere or Transaction Factory for Adobe Premiere .		Yes	Yes
PRM	Adobe Premiere plugin created by Filter Factory for Adobe Premiere or Transaction Factory for Adobe Premiere .			Yes
FFX	"Filters Unlimited" file.			Yes ²
TXT	A text file created by " Plugin Commander " or " FFDecomp ".		Yes ³	Yes
GUF	A filter file created by " GIMP UserFilter ".		Yes	Yes ²
FFL	"Filter Factory Library" by " Plugin Commander ".			Yes ⁴
BIN	Standalone filter created by Filter Factory/Foundry for Mac.			Yes

Currently not supported are FFP (FilterMeister) files.

¹ Loading is only possible if the 8BF file was created by Filter Factory, or by Filter Foundry without protection.

² Note that Filter Foundry only implements the basic Filter Factory commands. Therefore, most "Filters Unlimited" filters will not work with Filter Foundry.

³ Title, Category, Author, Copyright, and Slider/Map names are left empty and must be added using a text editor.

⁴ The FFL files are automatically extracted into TXT files, so that you can read the TXT files in the next step.

Expressions for creating filters

This section explains how to set up the arithmetic expressions that describe what a custom filter will do.

Components of expressions

A filter performs an operation on the channels of each pixel in an image. These channel operations are described by arithmetic expressions. Expressions are made up of combinations of four types of components: number constants, variables, functions, and operators. The following sections describe these four components.

The Filter Foundry allows only integer numbers in expressions – no fractions or decimal numbers are allowed. Variables and functions will always evaluate to integers. The operators operate on signed, 32-bit integers.

Number constants

A number constant is a number that is supplied directly in the expression. Constants can be used to construct simple expressions such as 10+5, and these expressions can always be replaced by another constant; in this case, 15.

Constants can also be written in hexadecimal form. To use hexadecimal values, prefix the number by 0x, as in 0xff which is the same as 255 in decimal notation.

Variables

A variable is a short name, such as *x*, that can be evaluated. The value of a variable depends on the current image, the current pixel, or the current channel for which an expression is being evaluated. For example, the variable *x* always evaluates to the horizontal coordinate of the current pixel, and the variable *y* always evaluates to the vertical coordinate. The variables for the current pixel's channel values are *r* (red), *g* (green), *b* (blue), and optionally *a* (alpha).

You can combine variables and constants to form expressions. For example, the expression *r+g* retrieves the red and green channel values for the current pixel and adds them together.

You can use the following variables in your expressions:

Variables	Definitions
r, g, b	For RGB color mode: Red, green, and blue channel values for the current pixel. For other color modes: Channel 1, 2, 3.
a	For RGB, Duotone, Greyscale, and Lab color mode: Alpha channel (transparency) value for the current pixel. 0 (<i>amin</i>) is fully transparent, max channel value (<i>A</i> , or <i>amax</i>) is fully opaque. Does not work on the background layer or layers where transparency is locked. For CMYK, “a” maps to the black channel. The 5 th channel (transparency) cannot be controlled with Filter Foundry. For Multichannel color mode: “a” maps to the 4 th color channel.
c	Value of the current channel, whichever channel the expression is defining.
R, G, B, A, C (rmax, gmax, ...)	Maximal channel values. For 8-bit images, it is 255, for 16-bit images it is 32,768, and for 32-bit images it is 8,388,608. Special case of Lab color mode: G/gmax and B/bmax are 127 for 8-bit Lab mode and 16,256 for 16-bit Lab mode.
rmin, gmin, bmin, amin, cmin	Minimum channel values. They are always 0, except for gmin and bmin at Lab color mode which are -128 for 8-bit Lab mode and -16,384 for 16-bit Lab mode.
i, u, v	The <i>i</i> , <i>u</i> , and <i>v</i> variables represent the channels in the YUV color mode. YUV channels do not exist in an RGB image, so they are calculated from the RGB channels. Because this calculation takes some time, using these variables is slower than using the <i>r</i> , <i>g</i> , and <i>b</i> variables. The following formulas are used to convert from RGB to YUV: $i = (299*r + 587*g + 114*b) / 1000$ $u = (-147407*r - 289391*g + 436798*b) / 2000000$ $v = (614777*r - 514799*g - 99978*b) / 2000000$ <p>According to these formulas, the output range of <i>i</i>, <i>u</i>, and <i>v</i> is (for 8-bit images):</p> $0 \leq i \leq 255$ $-55 \leq u \leq 55$ $-78 \leq v \leq 78$ <p>⚠ Attention: The values are only valid in the RGB image mode.</p>
imin	Smallest possible value of <i>i</i> . It is always 0.
umin	Smallest possible value of <i>u</i> . It is -55 (8-bit), -7,156 (16-bit), or -1,832,063 (32-bit). In Filter Factory, it was erroneously 0.
vmin	Smallest possible value of <i>v</i> . It is -78 (8-bit), -10,072 (16-bit), -2,578,561 (32-bit). In Filter Factory, it was erroneously 0.
imax	Largest possible value of <i>i</i> . It is 255 (8-bit), 32,768 (16-bit), or 8,388,608 (32-bit). In Filter Factory, it was 255 (Win) and 256 (Mac).
umax	Largest possible value of <i>u</i> . It is 55 (8-bit), 7,156 (16-bit), or 1,832,063 (32-bit). In Filter Factory, it was erroneously 255 (Win) and 256 (Mac).
vmax	Largest possible value of <i>v</i> . It is 78 (8-bit), 10,072 (16-bit), 2,578,561 (32-bit). In Filter Factory, it was erroneously 255 (Win) and 256 (Mac).
I	Defined as <i>imax-imin</i> , which is 255 (8-bit), 32,768 (16-bit), or 8,388,608 (32-bit). In Filter Factory it was equal to <i>imax</i> (255 for Win, and 256 for Mac).
U	Defined as <i>umax-umin</i> , which is 110 (8-bit), 14,312 (16-bit), or 3,664,126 (32-bit). In Filter Factory it was equal to <i>umax</i> (255 for Win, and 256 for Mac).
V	Defined as <i>vmax-vmin</i> , which 156 (8-bit), 20,144 (16-bit), or 5,157,122 (32-bit). In Filter Factory it was equal to <i>vmax</i> (255 for Win, and 256 for Mac).
d	Direction (angle) of the current pixel from the center of the image, where <i>d</i> is an integer between -512 and 512, inclusive. The Filter Factory manual erroneously states the range as -511 and 512.

dmax	Max value of d . It is always 512. In Filter Factory, it was erroneously 1024.
dmin	Min value of d . It is always -512. In Filter Factory it was erroneously 0.
D	Returns the total amount of angles within the image. Is always 1024 ($dmax-dmin$)
m	Distance (magnitude) from the center of the image to the current pixel
M (mmax)	Range of magnitudes with the image. It is always one-half the diagonal size of the image.
mmin	Is always 0
x	X-Coordinate (horizontal) of the current pixel
X (xmax)	Range of horizontal coordinates over the width of the image. The X variable represents the largest possible value for the x variable. This range is closed on the minimum and open on the maximum: $0 \leq x < X$.
xmin	Is always 0
y	Y-Coordinate (vertical) of the current pixel
Y (ymax)	Range of vertical coordinates over the height of the image. The Y variable represents the largest possible value for the y variable. This range is closed on the minimum and open on the maximum: $0 \leq y < Y$.
ymin	Is always 0
z	Channel index for the current expression
Z (zmax)	Range of channel indexes within one pixel The Z variable represents the largest possible value for the z variable. This range is closed on the minimum and open on the maximum: $0 \leq z < Z$.
zmin	Is always 0

Note: Some of these variables were present in Filter Factory, but they were undocumented!
The variables in brackets are synonyms.

Functions

Functions are short names that can be evaluated, and they require one or more arguments.

For example, the *rnd* function requires two arguments. It evaluates to a number that is greater than or equal to the first argument and less than or equal to the second argument. The expression *rnd(1,10)* evaluates to a number between 1 and 10, inclusive. (The *rnd* function is called a random number generator, and it is useful for adding noise or texture to an image.)

Arguments are written within parenthesis and are separated by commas. The arguments can be expressions. For example, the expression *rnd(r-10,r+10)* evaluates to the red channel of the current pixel, plus or minus 10.

Another function, *src* (source), retrieves channel values for a particular pixel. It requires three arguments: the horizontal coordinate of the pixel, the vertical coordinate, and the channel index. (The index for the red channel is 0; the green channel is 1; the blue channel is 2; the alpha channel is 3.) For example, the expression *src(10,20,0)* retrieves the red channel value for the pixel at coordinates (10,20). The expression *src(x,y,0)* retrieves the red channel value for the current pixel. The expression *src(x+1,y,0)* retrieves the red channel for the pixel to the right of the current pixel.

You can use the following functions in your expressions. Many functions place restrictions on the possible values of their arguments. If an argument is out of range, the expression will return a 0. For example, the expression *ctl(8)* evaluates to 0 because the *ctl* (control) function requires an argument between 0 and 7.

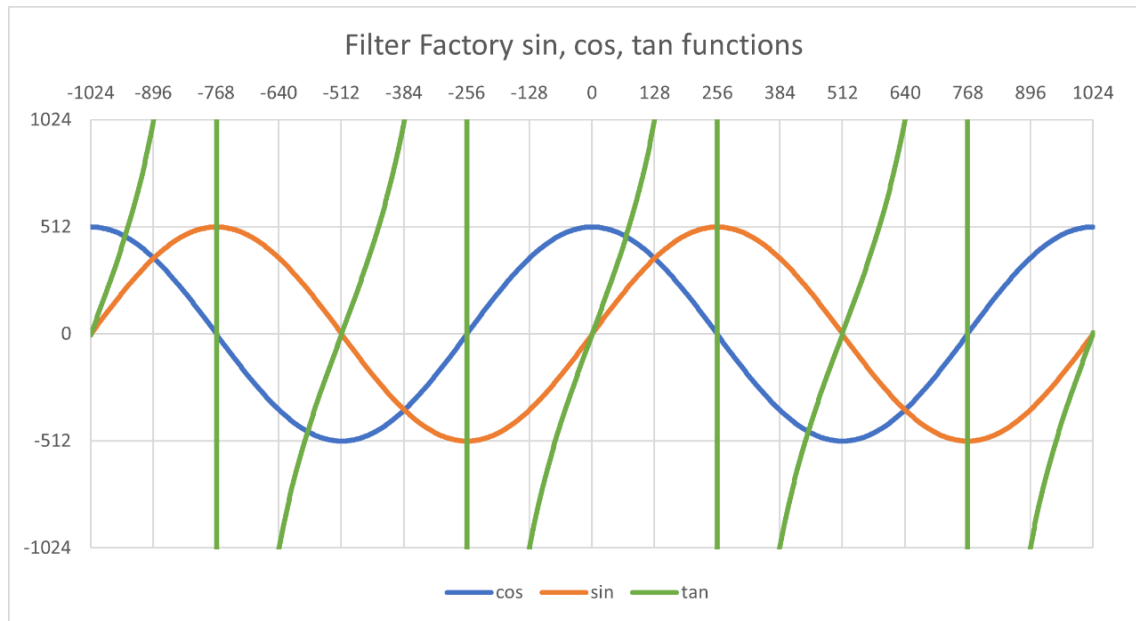
Functions	Definitions
<code>src(x,y,z)</code>	Pixel value of channel <i>z</i> (0 red, 1 green, 2 blue, 3 alpha) for the pixel at coordinates <i>x, y</i> The <i>src</i> (source) function is slow compared to the other operators and functions. Several evaluations of the <i>src</i> function in one expression can noticeably slow down the processing of an image. The coordinates passed to the <i>src</i> function should be within the ranges specified by the <i>X, Y</i> , and <i>Z</i> variables; otherwise, the coordinates will be pinned.
<code>rad(d,m,z)</code>	Pixel value of channel <i>z</i> (0 red, 1 green, 2 blue, 3 alpha) in the source image, which is <i>m</i> units away, at an angle of <i>d</i> , from the center of the image
<code>ctl(i)</code>	Value of slider <i>i</i> , where <i>i</i> is an integer between 0 and 7, inclusive.
<code>val(i,a,b)</code>	Value of slider <i>i</i> , mapped onto the range <i>a</i> to <i>b</i> . $\text{val}(i, a, b) = \text{ctl}(i) \cdot \frac{b - a}{255} + a$ The <i>val</i> (value) function converts the requested slider setting into a value in the requested range. The slider setting is multiplied by the size of the range (<i>b-a</i>) and offset by the start of the range (<i>a</i>). This function is useful when the range returned by the sliders (always 0 to 255, inclusive) does not match the range of values you want to use. For example, if the requested range is 1 to 10, a slider setting of 0 returns a value of 1, a setting of 255 returns a value of 10, and a setting of 127 returns a value of 5. The start of the requested range does not have to be less than the end of the range. For example, the expression <i>val(0,10,-10)</i> returns values between 10 and -10.
<code>map(i,n)</code>	Item <i>n</i> from the virtual mapping table <i>i</i> , where <i>i</i> is an integer between 0 and 3, inclusive, and <i>n</i> is an integer between 0 and 255, inclusive. The <i>map</i> (mapping) function uses virtual mapping tables which are constructed according to the slider settings. Each mapping table uses a pair of sliders: Table <i>i</i> uses sliders <i>2i</i> and <i>2i+1</i> for the high and low values, respectively. The tables are calculated as follows, where <i>L</i> is the value of the low slider, <i>H</i> is the value of the high slider, and <i>n</i> is an entry: $\text{map}(i, n) = \begin{cases} 0, & n \leq \text{ctl}(2i + 1) \\ 255, & n \geq \text{ctl}(2i) \\ \frac{n - \text{ctl}(2i+1)}{\text{ctl}(2i) - \text{ctl}(2i+1)} \cdot 255, & \text{ctl}(2i + 1) < n < \text{ctl}(2i) \end{cases} \quad \quad 0 \leq i \leq 3$
<code>min(a,b)</code>	Lesser of <i>a</i> and <i>b</i>
<code>max(a,b)</code>	Greater of <i>a</i> and <i>b</i>
<code>abs(a)</code>	Absolute value of <i>a</i>
<code>add(a,b,c)</code>	Sum of <i>a</i> and <i>b</i> , or <i>c</i> , whichever is lesser
<code>sub(a,b,c)</code>	Difference of <i>a</i> and <i>b</i> , or <i>c</i> , whichever is greater
<code>dif(a,b)</code>	Absolute value of the difference of <i>a</i> and <i>b</i>
<code>rnd(a,b)</code>	Random number between <i>a</i> and <i>b</i> , inclusive. The <i>rnd</i> (random) function returns a different random number each time it is called, but the entire function resets each time an image is processed. As a result, a filter that uses the <i>rnd</i> function will have the same effect each time it is used on the same image.

rst(i)	<p>Resets the random-number-generator (RNG) using seed i (0..32767). The RNG is also reset every time the filter is invoked. The initial seed is 0.</p> <p>The <i>rst</i> (random state) function can be used to change the initial seed of the random-number-generator (<i>rnd</i> function). More information on how <i>rst</i> works in Filter Factory and Filter Foundry can be found in the chapter “Implementation detail differences”.</p>
mix(a,b,n,d)	<p>Mixture of a and b by fraction n/d.</p> <p>The definition is:</p> $\text{mix}(a, b, n, d) = \frac{a \cdot n}{d} + \frac{b \cdot (d - n)}{d}$ <p>The <i>mix</i> (mixture) function combines the two input values using the specified fraction. A fraction of 1/2 returns the average of the two input values. A fraction close to 1 returns the first input value, and a fraction close to 0 returns the second input value.</p>
scl(a,il,ih,ol,oh)	<p>Scale a from input range (il to ih) to output range (ol to oh).</p> $\text{scl}(a, il, ih, ol, oh) = \begin{cases} 0, & ih = il \\ ol + \frac{(oh - ol)(a - il)}{(ih - il)}, & ih \neq il \end{cases}$ <p>The <i>scl</i> (scale) function maps a value from an input range onto an output range. For example, an input range of 0 to 255 could be mapped onto an output range of -100 to 100 by the expression <i>scl</i>(c, 0, 255, -100, 100). In this example, channel values close to 0 are mapped starting at -100, and channel values close to 255 are mapped up to 100. Note that $\text{val}(0, a, b) = \text{scl}(\text{ctl}(0), 0, 255, a, b)$.</p>
sqr(x) or sqrt(x)	<p>Square root of x</p> <p>Note: Filter Factory did calculate <i>sqr</i>(x)=x for the illegal input value $x < 0$. Filter Foundry took over this behavior to keep compatibility.</p>
sin(x)	<p>Sine function of x, where x is an integer between 0 and 1024, inclusive, and the value returned is an integer between -512 and 512, inclusive (Windows) or -1024 and 1024, inclusive (Mac OS)</p>
cos(x)	<p>Cosine function of x, where x is an integer between 0 and 1024, inclusive, and the value returned is an integer between -512 and 512, inclusive (Windows) or -1024 and 1024, inclusive (Mac OS)</p>
tan(x)	<p>Tangent function of x, where x is an integer between -256 and 256. The value returned can go down to $-\infty$ or up to ∞ (actually, the range is -167772 to 167772).</p> <p>The Filter Factory documentation described this function to be bounded, but this is not true.</p>
r2x(d,m)	<p>x displacement of the pixel m units away, at an angle of d, from an arbitrary center.</p> <p>The <i>r2x</i> and <i>r2y</i> functions convert radial coordinates to cartesian coordinates.</p>
r2y(d,m)	<p>y displacement of the pixel m units away, at an angle of d, from an arbitrary center.</p> <p>The <i>r2x</i> and <i>r2y</i> functions convert radial coordinates to cartesian coordinates.</p>
c2d(x,y)	<p>Angle displacement of the pixel at coordinates x,y.</p> <p>The <i>c2d</i> and <i>c2m</i> functions convert Cartesian coordinates to radial coordinates.</p>
c2m(x,y)	<p>Magnitude displacement of the pixel at coordinates x,y.</p> <p>The <i>c2d</i> and <i>c2m</i> functions convert Cartesian coordinates to radial coordinates.</p>
get(i)	Returns the current cell value at i
put(v,i)	Puts the new value v into cell i

<code>cnv(m11,m12,m13, m21,m22,m23, m31,m32,m33,d)</code>	Convolution kernel where $m11$ – $m33$ define the weights of pixels surrounding the current pixel ($m22$) and d scales the resulting value.
<code>pow(b,e)</code>	Calculates the base to the exponent power, that is, b^e . (This function was added in the inofficial Filter Factory patch 3.1.1 by Daniel Marschall.)

Trigonometric functions

The following graph shows the relationship between *sin*, *cos*, and *tan* functions.



Storage functions

The *get* and *put* functions can be used to store intermediate values into one of the cells (indexed from 0 to 255). Because the channel expressions are evaluated in order (R,G,B,A), the cells can be filled with values in an expression, and the values are available for the following functions. For example, the following expressions:

```
R: put(i,0),get(0)
G: get(0)
B: get(0)
A: a
```

are an efficient way to convert a color image into a monochrome image. Using *get* and *put* in this case speeds things up because the expression i is only evaluated once per pixel, rather than for every channel of every pixel.

Convolution kernel

The *cnv* (convolution) function is like Photoshop’s “Custom” filter and modifies the value of a pixel by assigning weights to the pixel and its eight surrounding pixels. The value of each pixel is multiplied by a weight ($m11$ through $m33$). The resulting values are then added and divided by d (typically the sum of the weights) to produce the new pixel value.

The pixels are arranged as follows, where $m22$ is the weight assigned to the target pixel:

$m11$	$m12$	$m13$
$m21$	$m22$	$m23$
$m31$	$m32$	$m33$

For example, the following expressions blur and sharpen a pixel, respectively:

Blur: $cnv(0, 1, 0, 4, 1, 0, 1, 0, 8)$

0	1	0
4	1	0
1	0	8

$d=8$

Sharpen: $cnv(0, -1, 0, -1, 5, -1, 0, -1, 0, 1)$

0	-1	0
-1	5	-1
0	-1	0

$d=1$

Polar coordinate functions

Functions in Filter Foundry may use one of two distinct coordinate models to interpret image data.

One of these is the **Cartesian coordinate model** described in “About Digital Images”.

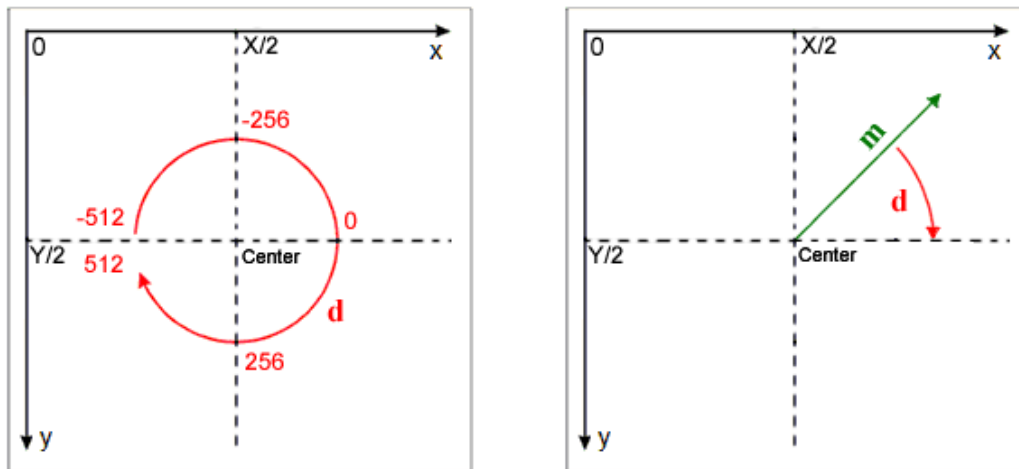
The other is the **polar coordinate model**, in which the origin is the center of the image, and coordinates are defined by an angle d from the horizontal axis and a distance, or magnitude, m from the origin.

In this model, -512 degrees corresponds to the 9 o'clock position, -256 to the 12 o'clock position, 0 to the 3 o'clock position, 256 to the 6 o'clock position, and 512 to the full rotation back to the 9 o'clock position. Negative values correspond to the appropriate counter-clockwise position.

The $r2x$ and $r2y$ functions convert radial coordinates to cartesian coordinates.

The $c2d$ and $c2m$ functions convert Cartesian coordinates to radial coordinates.

The functions rad , sin , cos , tan , $r2x$, $r2y$, $c2d$, and $c2m$ use polar coordinates.



Note: Filter Factory defines $c2d(x,y) := \arctan2(y,x)$, but $d := \arctan2(-y,-x)$

Operators

The operators include all of the arithmetic that can be used in an expression. There are five types of operators:

1. Arithmetic
2. Relational
3. Logical
4. Conditional
5. Bitwise

Precedence

You can use the following operators in your expressions. The operators are presented in their order of precedence. Precedence determines which operators are evaluated first within an expression when the order of evaluation is ambiguous. For example, in the expression $2+3*4$, the $*$ operator is evaluated first because it has higher precedence than the $+$ operator.

You can use brackets to change the precedence of the operators.

Operators	Definitions
,	Sequence
?:	Conditional
&&,	Logical “and”, logical “or”
&, ^,	Bitwise “and”, bitwise “exclusive or”, bitwise “or”
==, !=	Equal to, not equal to
<, <=, >, >=	Less than, less than or equal to, greater than, greater than or equal to
<<, >>	Shift left, shift right
*, /, %	Multiply, divide, modulo
+, -	Add, subtract
!, ~	Logical “not”, bitwise “not”

Arithmetic operators

Operator	Definitions
a+b	Add
a-b	Subtract
a*b	Multiply
a/b	Divide (divide on integer; will round down!)
a%b	Modulo (Remainder of a division; e.g. the expression $11\%3$ evaluates to 2)
-a	Negate

Relational operators

Operator	Definitions
a==b	Equal to
a!=b	not equal to
a<b	Less than
a<=b	less than or equal to
a>b	greater than
a>=b	greater than or equal to

Relational operators compare two expressions and evaluate to 0 (“false”) or 1 (“true”). For example, the $<$ operator evaluates to 1 if the expression on the left is less than the expression on the right. The expression $r < g$ evaluates to 1 when the red channel of the current pixel has a lower value than the green channel. Otherwise, it evaluates to 0. The set of relational operators includes $<$, $<=$, $>$, $>=$, $==$, and $!=$.

The $==$ (“equal-to”) operator evaluates to 1 when the two expressions surrounding it evaluate to the same thing. The $!=$ (“not-equal-to”) operator evaluates to 1 when the two expressions surrounding it evaluate to something different. For example, the expression $1 == 1$ evaluates to 1, and the expression $2 == 1$ evaluates to 0. The expression $1 != 2$ evaluates to 1, and the expression $1 != 1$ evaluates to 0.

Logical operators

Operators	Definitions															
a&&b	Logical “and” operation which has the following truth table: <table><tr><th>a</th><th>b</th><th>a&&b</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	a&&b	0	0	0	0	1	0	1	0	0	1	1	1
a	b	a&&b														
0	0	0														
0	1	0														
1	0	0														
1	1	1														
a b	Logical “or” operation which has the following truth table: <table><tr><th>a</th><th>b</th><th>a b</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	a	b	a b	0	0	0	0	1	1	1	0	1	1	1	1
a	b	a b														
0	0	0														
0	1	1														
1	0	1														
1	1	1														
!a	Logical “not” operation which has the following truth table: <table><tr><th>a</th><th>!a</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	a	!a	0	1	1	0									
a	!a															
0	1															
1	0															

Logical operators let you combine several relational expressions. For example, you could evaluate whether the horizontal coordinate of a pixel is between 10 and 30, inclusive. The appropriate relational expressions are $x \geq 10$ and $x \leq 30$. You can use the logical && operator to combine them into the single expression $(x \geq 10) \&\& (x \leq 30)$.

The && operator evaluates the expressions on both sides. If neither expression evaluates to 0, the && operator evaluates to 1. If either expression evaluates to 0, the && operator evaluates to 0.

The logical operators &&, ||, and ! treat all expressions as either true or false. Any value other than 0 is considered true, and only a 0 value is considered false.

The logical operator || is like the operator &&, but it performs a slightly different logical operation. The || operator is also placed between two relational expressions. If either of the expressions evaluates to anything but 0, the || operator evaluates to 1. If both expressions evaluate to 0, the || operator evaluates to 0. For example, the expression $(x > 10) || (y > 10)$ evaluates to 1 when the horizontal coordinate is greater than 10. The only time this expression evaluates to 0 is when the horizontal coordinate is ≤ 10 and the vertical coordinate is ≤ 10 .

The following truth table shows the difference between the && and || operators:

Left expression	Right expression	Left && Right	Left Right
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Finally, you place the **!** operator before an expression to invert the expression's evaluation. If the expression evaluates to 0, the **!** operator evaluates to 1. If the expression evaluates to anything but 0, the **!** operator evaluates to 0.

Conditional operators

Operators	Definitions
$a ? b : c$	Conditional. Evaluate "b" if "a" is non-zero, otherwise, evaluate "c".
a,b a,b,c ...	Sequence The comma operator evaluates both the left and right expressions and returns the value of the right expression.

The single **conditional operator** **?** lets you make a choice between two alternatives. A conditional expression includes a selection expression and two alternative expressions. The conditional operator evaluates the selection expression and uses the result to decide which of the two alternatives it should evaluate. If the selection expression evaluates to anything but 0, the first alternative is evaluated. If the selection expression evaluates to 0, the second alternative is evaluated.

For example, in the expression $(x \% 2) ? r : g$, the **?** conditional operator separates the selection expression $(x \% 2)$ from the two alternative expressions *r* and *g*. The alternative expressions are separated by a colon (:). The selection expression divides the horizontal coordinate of the current pixel by 2 and returns the remainder of the division. If the horizontal coordinate is an odd number, the result is something other than 0, and if the horizontal coordinate is an even number, the result is 0. Therefore, if the pixel has an odd horizontal coordinate, the conditional operator returns the value of the red channel. If the current pixel has an even horizontal coordinate, the conditional operator returns the value of the green channel.

Bitwise operators

Operators	Definitions
$a \& b$	Bitwise "and"
$a \wedge b$	Bitwise "exclusive or"
$a b$	bitwise "or"
$a \ll b$	Shift left
$a \gg b$	Shift right
$\sim a$	Bitwise "not"

Bitwise operators directly manipulate the bits in a value. The bitwise operators include **&**, **|**, **^**, **~**, **<<**, and **>>**. You place the **&**, **|**, and **^** operators between two expressions. The **&** operator performs a logical-and operation on the corresponding bits of the evaluated expressions; the **|** operator performs a logical-or, and the **^** operator performs a logical-exclusive-or. The **~** operator takes only one expression, and it performs a logical-not on each bit of the evaluated expression.

The **<<** and **>>** expressions are placed between two expressions. Both operators shift the bits in the left expression's evaluation by some number, which is specified by the right expression's evaluation. The **<<** operator shifts bits to the left. The **>>** operator shifts bits to the right.

The shifting operators (**<<** and **>>**) perform logical, not arithmetic, shifts so the sign of the shifted operand is not preserved.

Comments

Comments are human-readable text placed inside the formula which is not evaluated by Filter Foundry. A comment will start with a double slash (//) and will end with the end of the line. The next line will be part of the formula again.

Example (comments are marked in blue):

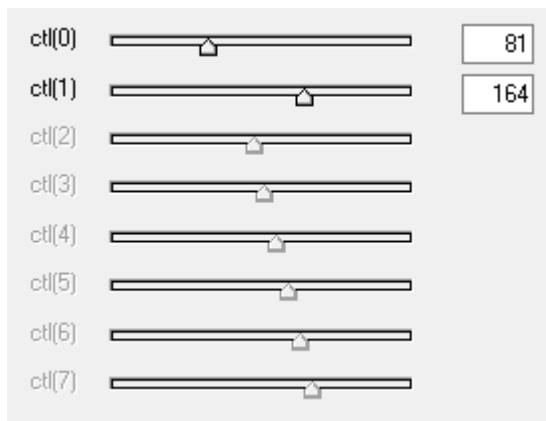
```
R: r*ctl(0) // Slider for multiplication
    + ctl(1) // Slider for addition
G: g * 22/7 // 22/7 is approximately Pi
B: b
```

A few examples where comments might be useful:

- To add a “TODO” entry to the formula
- To place a copyright notice into an AFS filter file
- To “comment out” a part of the formula that you want to remove temporarily
- To make an explanation for a part of your formula which might be confusing to other programmers
- To write down what a slider is intended to

Providing user-controlled sliders

When you create a filter, you can provide up to eight slider controls for the user to adjust when applying the effect. Slider values can range from 0 to 255.



You set up the effect’s sliders by using the *ctl* (control), *val* (value), and *map* (mapping) functions in your expressions.

If you use these functions to retrieve slider information, you should set up the Control or Map options in the Filter Foundry’s Build Custom dialog box. For information on using the Build Custom Filter dialog box, see “Creating Custom Filters”.

1. The *ctl* function retrieves the specified slider’s current value. This function requires one argument: the slider index, which is a number between 0 and 7. For example, the expression *ctl(0)* evaluates the current value of the first slider.
2. The *val* function converts the range of possible slider values (always 0 to 255) into a range that you specify. For example, to get a value between 1 and 100 from a slider, you would use the expression *val(0,1,100)*. If slider 0 is set to 0, the *val* function evaluates to 0. If slider 0 is set to 255, the *val* function evaluates to 100. Slider values between 0 and 255 are converted into values between 1 and 100.
3. The *map* function groups the sliders into pairs. Each even/odd slider pair sets the values in a virtual mapping table, which is accessed by the *map* function. There are four mapping tables – one for each slider pair. Sliders 0 and 1 set the values for mapping table 0; sliders 2 and 3 set the values for mapping table 1, and so on. Each table contains 256 entries.

The *map* function takes two arguments: the table index and the item index. For example, the expression *map(1,20)* returns item 20 from table 1. The table index must be between 0 and 3. The item index must be between 0 and 255, inclusive.

Examples

This section provides several examples of using expressions to achieve a result. The examples are presented in the order of their complexity. The Adobe Photoshop program also provides some sample filter expressions. You can use the Filter Foundry to load a sample file and observe its effect.

Affecting a single channel (filter)

To make an image redder, you could use the following expressions:

```
R: r+100
G: g
B: b
A: 0
```

The first expression evaluates the red channel of each pixel and adds 100 to each one. The next two expressions evaluate the other two channels and leave them unchanged.

Affecting channels using sliders (filter)

To add a user-controlled slider value to the current channel values, you could use the following expressions:

```
R: r+ctl(0)
G: g+ctl(1)
B: b+ctl(2)
A: 0
```

The first expression evaluates the red channel of each pixel and adds the value of slider 0 to each one. The next two expressions do the same thing to the green and blue channels, using the values of slider 1 and slider 2, respectively.

Adding noise to channels using sliders and random values (filter)

To use slider values to determine the range of random numbers, you could use the following expressions:

```
R: r+rnd(-ctl(0),ctl(0))
G: g+rnd(-ctl(1),ctl(1))
B: b+rnd(-ctl(2),ctl(2))
A: 0
```

The filter defined by these expressions adds noise to all three channels. The amount of noise in each channel is determined by the slider controls. The first expression evaluates the slider setting from slider 0. This value is used as the argument for the *rnd* function. If the slider setting is 0, the *rnd* function evaluates to 0. If the slider setting is 100, the *rnd* function can return any number between -100 and 100, inclusive. The result of the *rnd* function is then added to the current value of the red channel. As the slider setting is raised from 0 to 255, the random numbers are selected from a wider and wider range, resulting in more and more noise being added to the red channel.

The next two expressions perform the same operation on the green and blue channels, using sliders 1 and 2 for input, respectively.

Amplifying or toning down channels (filter)

To amplify or tone down a channel based on the values of a different channel, you could use the following expressions:

```
R: (b>100) ? r+50 : r-50
G: g
B: b
A: 0
```

The first expression evaluates the blue channel to determine if it is greater than 100. If it is greater than 100, the entire expression evaluates to the red channel value plus 50. If it is not greater than 100, the expression evaluates to the red channel value minus 50. The other two expressions do nothing. Therefore, this filter amplifies the red channel if there is bluer than you want, or it tones down the red channel if there is less blue than you want.

You could also use slider values in a similar type of filter, as follows:

```
R: (b>ctl(0)) ? r+ctl(1) : r-ctl(1)
G: g
B: (b>ctl(0)) ? b-ctl(1) : b+ctl(1)
A: 0
```

The first expression uses the setting from slider 0 as a “cutoff” value. If the blue channel is greater than the cutoff value, the red channel is amplified by the value of slider 1. If the blue channel is less than the cutoff value, the red channel is toned down by the value of slider 1. The third expression is the opposite of the first one, but it works on the blue channel instead of the red channel. The effect is that anything that is added to the red channel is subtracted from the blue channel and vice versa.

Averaging the channel values of neighboring pixels (filter)

You can use the *src* (source) function to retrieve the channel values from neighboring pixels and average them together, as follows:

```
R: (src(x-1,y,0)+src(x,y,0)+src(x+1,y,0))/3
G: (src(x-1,y,1)+src(x,y,1)+src(x+1,y,1))/3
B: (src(x-1,y,2)+src(x,y,2)+src(x+1,y,2))/3
A: 0
```

The first expression uses the *src* function to retrieve the red channel value for three different pixels: the pixel to the left of the current pixel, the current pixel, and the pixel to the right of the current pixel. These three values are added together and then divided by 3. The next two expressions do the same thing using the green and blue channels, respectively.

Implementation detail differences

Filter Foundry tries to be as compatible with Filter Factory as possible.

However, there are some differences that are explained in this documentation.

YUV color space variables

Filter Factory 3.0.x:

$i = ((76*r) + (150*g) + (29*b)) / 256$	Output range is $0 \leq i \leq 254$
$u = ((-19*r) + (-37*g) + (56*b)) / 256$	Output range is $-55 \leq u \leq 55$
$v = ((78*r) + (-65*g) + (-13*b)) / 256$	Output range is $-77 \leq v \leq 77$
$imin = 0$	
$imax = 255$ (Win), 256 (Mac)	
$umin = 0$	
$umax = 255$ (Win), 256 (Mac)	
$vmin = 0$	
$vmax = 255$ (Win), 256 (Mac)	
$I = 255$ (Win), 256 (Mac)	
$U = 255$ (Win), 256 (Mac)	
$V = 255$ (Win), 256 (Mac)	

Filter Factory 3.1.x and *Filter Foundry 1.7* use more accurate formulas which have higher accuracy, as well as correct min/max values which represent the actual range of the *i*, *u*, *v* variables:

$i = (299*r + 587*g + 114*b) / 1000$	Output range is $0 \leq i \leq 255$
$u = (-147407*r - 289391*g + 436798*b) / 2000000$	Output range is $-55 \leq u \leq 55$
$v = 614777*r - 514799*g - 99978*b / 2000000$	Output range is $-78 \leq v \leq 78$
$imin = 0$	(for 8-bit-images)
$imax = 255$	
$umin = -55$	
$umax = 55$	
$vmin = -78$	
$vmax = 78$	
$I := imax - imin = 255$	
$U := umax - umin = 110*$	
$V := vmax - vmin = 156*$	

*It is questionable if *I* was meant to be a synonym of *imax*, or if *I* was meant to be $I := imax - imin$. We have chosen the latter in Filter Foundry 1.7.0.9. Same thing with *U* and *V*.

Additional notes regarding the YUV color model (in 8-bit images):

- For U, the definition is $U := (B - Y) * 0.493$; the range would be $[-111..111]$. Filter Factory divided it by 2, resulting in a range of $[-55..55]$. Due to compatibility reasons, we adopt that behavior.
- For V, the definition is $V := (R - Y) * 0.877$; the range would be $[-156..156]$. Filter Factory divided it by 2, resulting in a range of $[-77..77]$. Due to compatibility reasons, we adopt that behavior (just change to $-78..78$).

Polar coordinate system D, dmin, dmax

The Filter Factory manual writes:

d=0 corresponds to the 3 o'clock position
d=256 to the 6 o'clock position
d=512 to the 9 o'clock position
d=768 to the 12 o'clock position
d=1024 to the full rotation back to the 3 o'clock position

But this does neither match the Windows implementation of Filter Factory nor its Mac OS implementation! In the original Windows & Mac OS Filter Factory implementations we can observe:

d=-512 is at 9 o'clock position
d=-256 is at 12 o'clock position
d=0 is at 3 o'clock position
d=256 is at 6 o'clock position
d=512 is the full rotation back to the 9 o'clock position

Therefore, $dmin$ has been changed from 0 to -512, and $dmax$ has been changed from 1024 to 512.

It is questionable if D was meant to be a synonym of $dmax$, or if D was meant to be $D := dmax - dmin$.

We have chosen the latter, so it stayed 1024.

get(i) function

Filter Factory on Windows: `get(i)=i if i>255 or i<0`
(The result “i” was most likely not intended but a result of undefined behavior)

Filter Factory on Mac OS: `get(i)=0 if i>255 or i<0`

Filter Foundry: get(i)=0 if i>255 or i<0

r, g, b of fully transparent pixels

In Filter Factory (Windows and Mac implementation), a fully transparent pixel is initialized (r, g, b variables) with the current background color.

Filter Foundry initializes it as $r,g,b=255$

rst(i) function

Filter Factory contains an undocumented function that sets the seed for the random number generator. Filter Factory and Filter Foundry beginning with 1.7.0.8 accept a seed between 0 and 32767, inclusively. If the argument is not within this range, the operation lowest 15 bits are taken.

There are many differences in the implementation between Filter Factory and Filter Foundry in regards $rst(i)$, which is documented below.

Filter Factory:

If $rst(i)$ is called in Filter Factory, an internal Seed-Variable is set. It does NOT influence any calls of $rnd(a,b)$, because a lookup-table needs to be built first. The building of the lookup-table is probably done before the processing of the first pixel $(x,y,z=0)$. It is suspected that the call of $rst(i)$ will take effect on the next calculation. Due to a bug (or feature?), the random state is not reset to its initial state (0) before the filter is applied. The preview image processing will modify the random state, and once the filter is actually applied (pressing "OK"), the random state that was set in the preview picture, will be used. This could be considered as a bug, but it is probably required, otherwise, the call of $rst(i)$ (inside the preview calculation) won't affect the $rnd(a,b)$ in the real run. However, in a standalone filter without dialog/preview, there is no preview that could set the internal seed, so the $rnd(a,b)$ functions will always work using the default seed 0, and only the subsequent calls will use the $rst(i)$ of the previous call.

Filter Foundry:

In Filter Foundry, the function `rnd(a,b)` retrieves a random number in “realtime”; therefore, if the seed is changed via `rst(i)`, there is an immediate effect on the next call of the `rnd(a,b)` function.

For example, the following filter would generate a one-colored picture without any randomness:

```
R: rst(123), rnd(0,255)
G: rnd(0,255)
B: rnd(0,255)
```

If you want to generate a random pixel image with a non-default seed, you need to make sure that `rst(i)` is called only once at the beginning (channel 0, coordinate 0,0):

```
R: (x== 0 && y ==0) ? rst(123) : 0, rnd(0,255)
G: rnd(0,255)
B: rnd(0,255)
```

In Filter Foundry, `rst(i)` can be called by branches, and variables/sliders can be used as arguments of `rst(i)`.

Evaluation of conditional branches

Filter Foundry:

Only the branches which will be chosen due to the conditional expression will be evaluated.

This means that following filter would generate a black canvas:

```
R: 1==0 ? put(255,0) : 0
G: get(0)
B: 0
```

In Boolean expressions, the evaluation will be aborted if the result is already determined.

So, this will also generate a black canvas:

```
R: 1==0 && put(255,0) ? 0 : 0
G: get(0)
B: 0
```

This will also generate a black canvas:

```
R: 1==1 || put(255,0) ? 0 : 0
G: get(0)
B: 0
```

Filter Factory:

Each branch inside an if-then-else expression will be evaluated.

This means that following filter would generate a green canvas:

```
R: 1==0 ? put(255,0) : 0
G: get(0)
B: 0
```

Also, all arguments of a Boolean expression will be fully evaluated.

So, this will also generate a green canvas:

```
R: 1==0 && put(255,0) ? 0 : 0
G: get(0)
B: 0
```

This will also generate a green canvas:

```
R: 1==1 || put(255,0) ? 0 : 0
G: get(0)
B: 0
```

License

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version:

<https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>

<https://www.gnu.org/licenses/gpl-3.0>

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.