gmic-py

Release 2.9.4-alpha1

Jonathan-David Schröder, David Tschumperlé

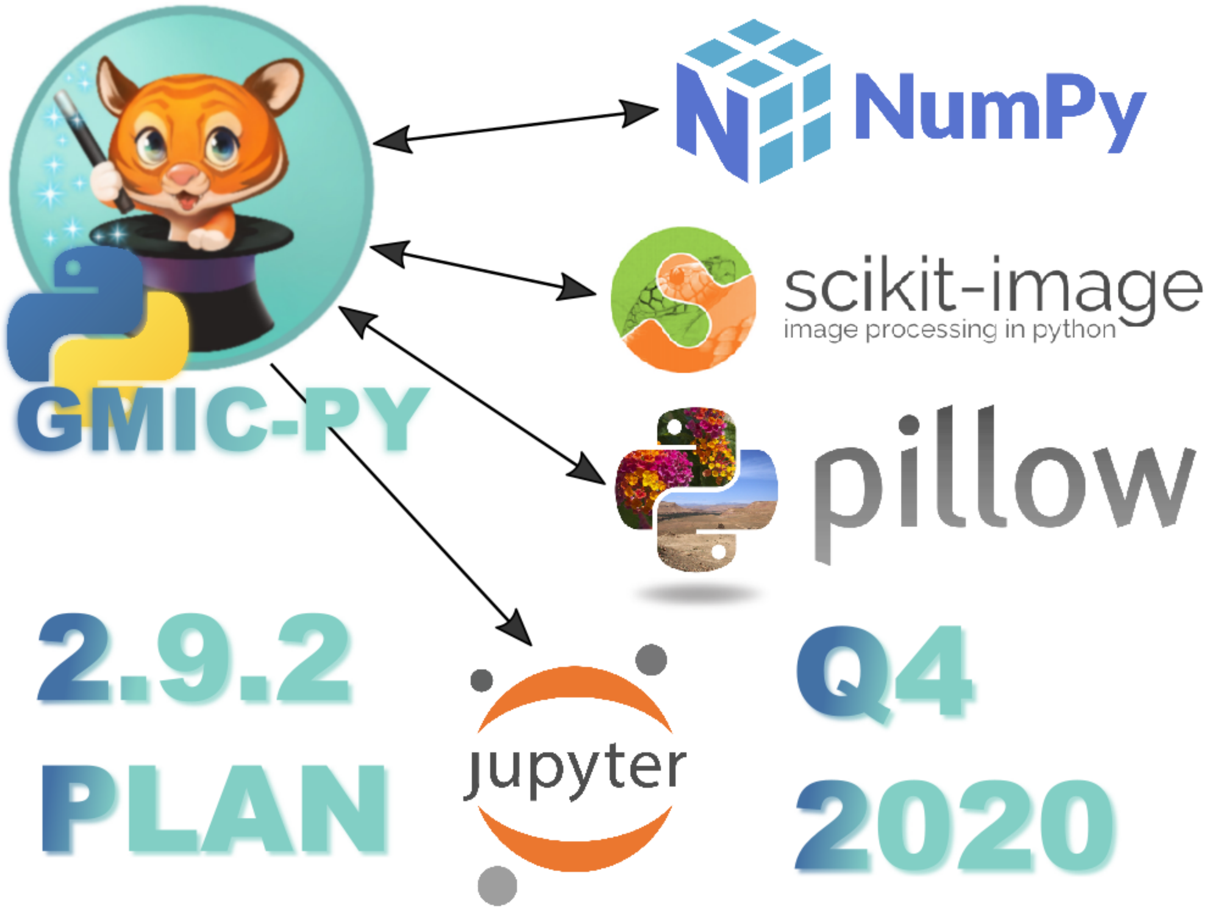
Jan 14, 2021

CONTENTS

`gmic-py` is a Python 3 binding for the G'MIC Image Processing Framework written in C++.

`gmic` is the package name of gmic-py on pypi.org, so you should `pip install gmic` (see *Installing*).

G'MIC provides image-processing commands for 1D to 4D images, as well as many graphical filters. It is thus targetted at both artists and data-scientists.



This documentation showcases various uses of `gmic-py`:

- gmic-py alone in pure Python,
- with Numpy,
- with the Python Imaging Library (PIL),
- with scikit-image,
- (soon) with pygame and Tkinter,
- (soon) with p5 processing for Python.

Head over to the *Quickstart* section for examples.

CHAPTER

ONE

INSTALLING

Here is how to install the `gmic-py` binary Python module on your operating system.

For now only Mac OS (till 2.8.3) and Linux / Unix (2.9.x) are supported.

For `sp` / `sample` and `update` commands to work, the curl or wget executable must be installed.

If you cannot install anything on your machine, you may also install `gmic-py` from a Jupyter Notepad or Google Colab, see *Jupyter/IPython/GColab support*

1.1 For Linux / Unix

If you have pip:

```
pip install gmic
```

Or conda:

```
conda install gmic
```

1.2 For Mac OS

If you have pip:

```
pip install gmic
```

If the OpenMP library is not installed yet, you may want to install it first (for parallelization speedup):

```
brew install libomp # Or possibly clang-omp if failing
```

1.3 For Windows (future)

gmic-py's support for Windows is planned but not ready yet.

CHAPTER TWO

QUICKSTART

Here is how to fiddle with gmic-py in five minutes.

`gmic-py` is a cross-platform Python binding for the G'MIC C++ library. G'MIC is nowadays mostly used for its image processing domain-specific language (DSL), or through its plug-ins for graphical software. In our case, only the language is available, but it is very powerful though!

Using `gmic-py` always boils down to five steps:

1. install the module
2. import it
3. (instantiate a G'MIC language interpreter)
4. evaluate a G'MIC expression against G'MIC samples or your own images
5. retrieve your output images

2.1 1. Install `gmic-py`

This works on Linux or Mac OS for now. You need no compiler, just Python >= 3.6.

In your favorite shell, run:

```
pip install gmic
```

This will install the G'MIC pre-compiled module for Python:

```
Collecting gmic
  Downloading gmic-2.9.0-cp36-cp36m-manylinux2014_x86_64.whl (8.8 MB)
     || 8.8 MB 6.8 MB/s
Installing collected packages: gmic
Successfully installed gmic-2.9.0
```

2.2 2. Run a simple G'MIC effect and view it

You are now ready to work, open a Python 3 terminal or edit your own `.py` file and type in the following:

```
import gmic
gmic.run("sp earth blur 4")
```

What that does is:

1. import the `gmic` Python module
2. create a default G'MIC language interpreter and give it an expression to evaluate:
   - `sp earth` will load a sample image from G'MIC library, named `earth`
   - `blur 4` will apply a `blur` effect to the image(s) before, here with a force of 4.

The G'MIC language's commands are all listed in its reference documentation on gmic.eu. *Here is for the ''blur'* command itself. <https://gmic.eu/reference.shtml#blur>'_ If you are on Linux (sorry, not Mac OS), you will see a window popping up with the following image:

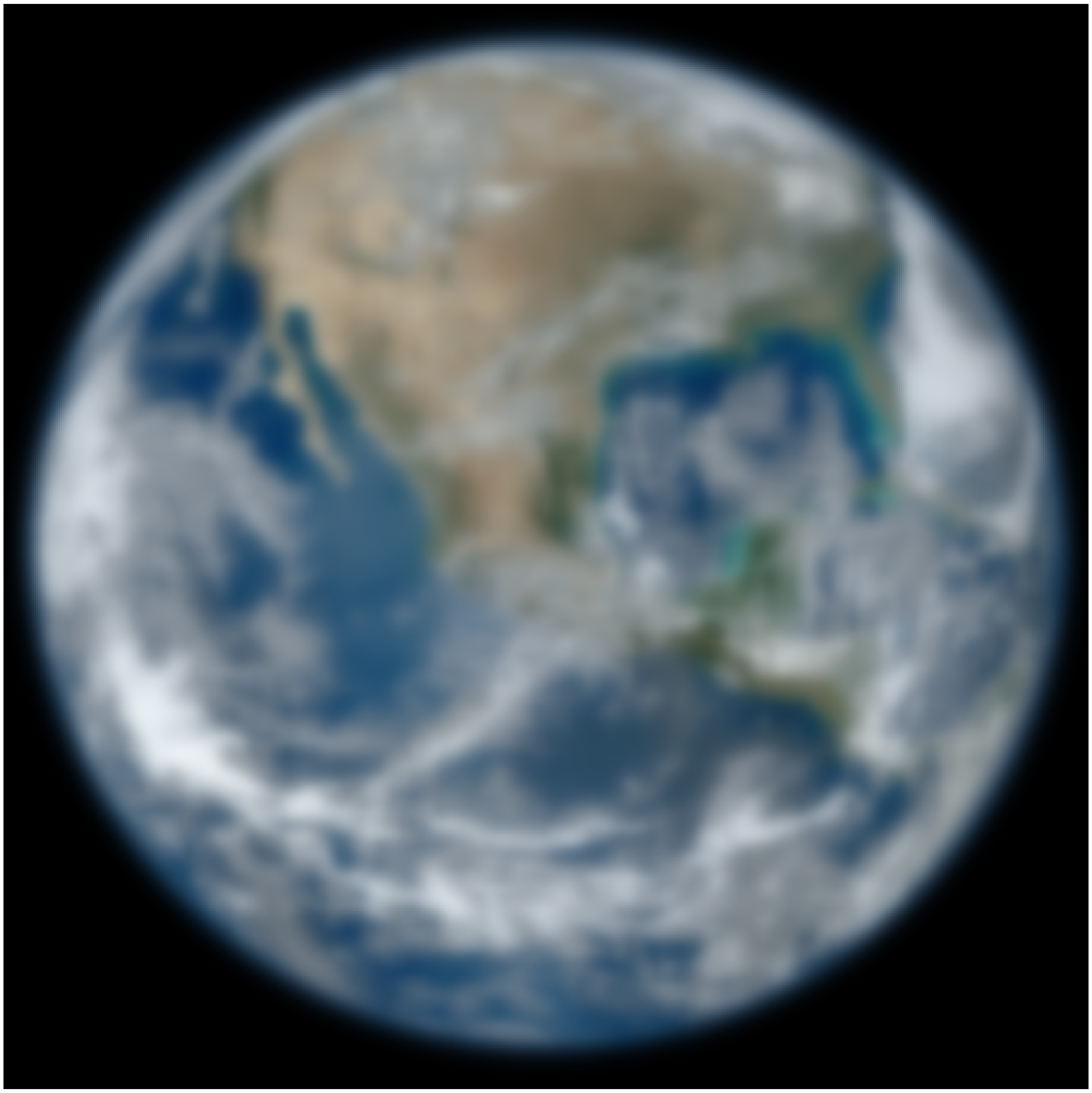


Fig. 1: sp earth blur 4

If you would like to use your own file instead, just write its path first instead of `sp earth`:

```
gmic.run("/home/me/myimage.jpg blur 4")
```

2.3 3. Save your result to a file

Whether you are on Linux or MacOS you can also save your image with the `output` G'MIC command.

```
import gmic
gmic.run("sp earth blur 4 output myblurredearth.png") # will save in the current␣
↪working directory
```

2.4 4. Dealing with `GmicImage` and `GmicImage` lists

Now you may want to use and analyse your result images without having to save them first!

For this, pass in as a second parameter a standard Python list, which will be filled with the result. You can then use this image list as input or output for other commands. The G'MIC Python binding works on image lists *in place* similarly to the C++ version. Those lists of images, named `gmic_list` in C++, are plain-oldlists the Python world. G'MIC will not rewrite GmicImage data in place, instead it replaces GmicImage by new ones inside image lists, or appends new images.

```
import gmic
image_results = [] # This is my image set that G'MIC can read and write in place. Its␣
↪contents can grow and reduce.
gmic.run("sp earth blur 4", image_results) # will save in the current working␣
↪directory
print(image_results)
# [<gmic.GmicImage object at 0x7f6b8f27e3f0 with _data address at 0x1fdfe00, w=500␣
↪h=500 d=1 s=3 shared=0>]
gmic.run("display", image_results) # will show the image a window (for Linux only)
gmic.run("print", image_results) # will print image statistics
# [gmic]-1./ Print image [0] = '[unnamed]'.
# [0] = '[unnamed]':
#   size = (500,500,1,3) [2929 Kio of floats].
#   data = (2.59494e-26,3.91551e-26,5.90707e-26,8.91042e-26,1.34385e-25,2.02649e-25,3.
↪05511e-25,4.6054e-25,6.94078e-25,1.04589e-24,1.57568e-24,2.3734e-24,(...),5.74726e-
↪24,3.81723e-24,2.53479e-24,1.68282e-24,1.117e-24,7.41318e-25,4.91866e-25,3.26308e-
↪25,2.16433e-25,1.43531e-25,9.51711e-26,6.30933e-26).
#   min = 6.66209e-27, max = 243.369, mean = 88.0979, std = 63.7667, coords_min = (0,
↪499,0,0), coords_max = (84,345,0,2).
```

```
gmic.run("fx_bokeh 3,8,0,30,8,4,0.3,0.2,210,210,80,160,0.7,30,20,20,1,2,170,130,20,
↪110,0.15,0", image_results)
gmic.run("display", image_results) # will show the image with bokeh effect applied a␣
↪window
gmic.run("print", image_results) # will print image statistics
# [gmic]-1./ Print image [0] = '[unnamed]'.
# [0] = '[unnamed]':
#   size = (500,500,1,4) [3906 Kio of floats].
#   data = (32.1635,38.7346,46.0117,53.8523,62.053,70.3621,78.5205,86.3153,93.5843,
↪100.211,106.123,111.297,(...),255,255,255,255,255,255,255,255,255,255,255,255).
#   min = 0, max = 255, mean = 150.29, std = 95.1853, coords_min = (84,0,0,0), coords_
↪max = (0,0,0,3).
```

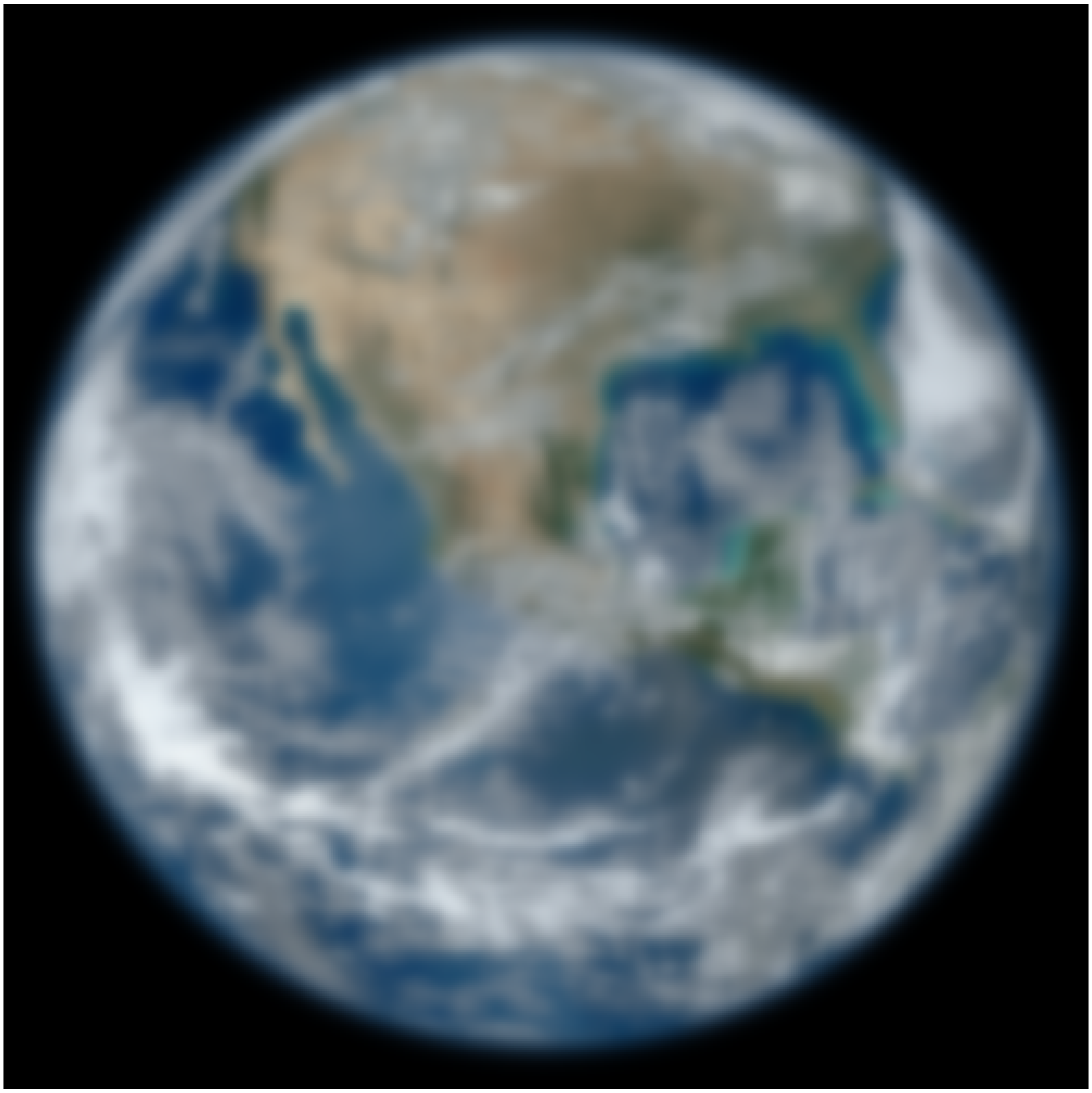


Fig. 2: sp earth blur 4

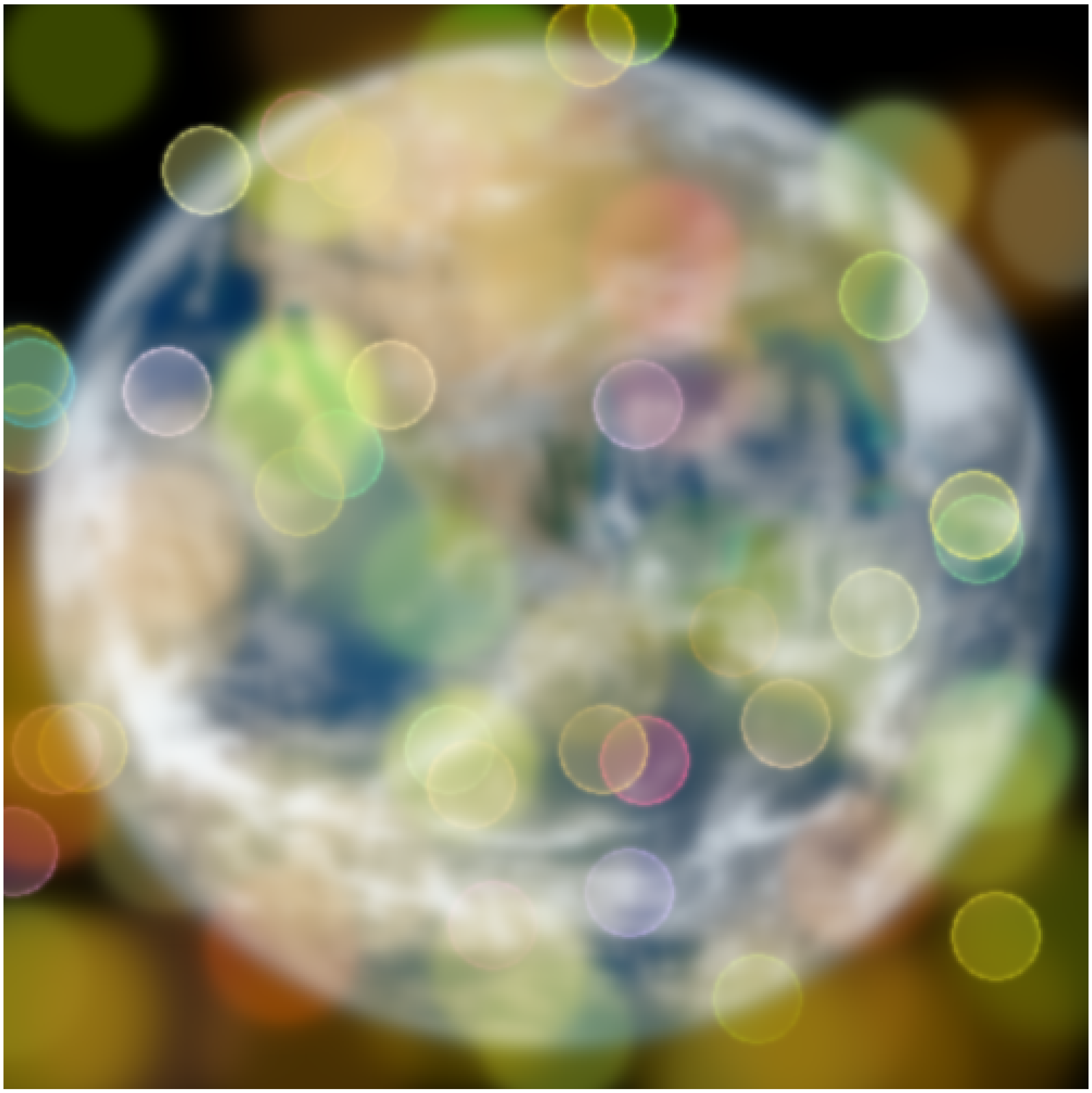


Fig. 3: sp earth blur 4 fx_bokeh 3,8,0,30,8,4,0.3,0.2,210,210,80,160,0.7,30,20,20,1,2,170,130,20,110,0.15,0

2.5 5. Working with the Numpy support

A chapter of this documentation is dedicated to the Numpy support of the G'MIC Python version. Head over to the *Numpy support* page for details. For now here is a foretaste of it showcasing one-direction conversion from G'MIC to Numpy.

You can input and output 1D to 3D matrices between G'MIC and Numpy. Note that *gmic-py* supports data input/output with not only *Numpy support*, but also *PIL support* and *Scikit-Image support*.

You must install the `numpy` module yourself, G'MIC will detect and import it from there, leveraging some of its functions. Data-scientists should keep in mind that G'MIC works internally with `float32` pixel values, arranged in a non-interleaved way (eg. RRR,GGG,BBB), with matrices whose shape is always 4D and prints as `(width, height, depth, spectrum)` where `spectrum` is the number of channels (ie. per-pixel values).

```
import numpy
from matplotlib import pyplot as plt
import gmic

images = []
gmic.run("sp apples blur_x 30", images) # Handy way to generate an image into file
```

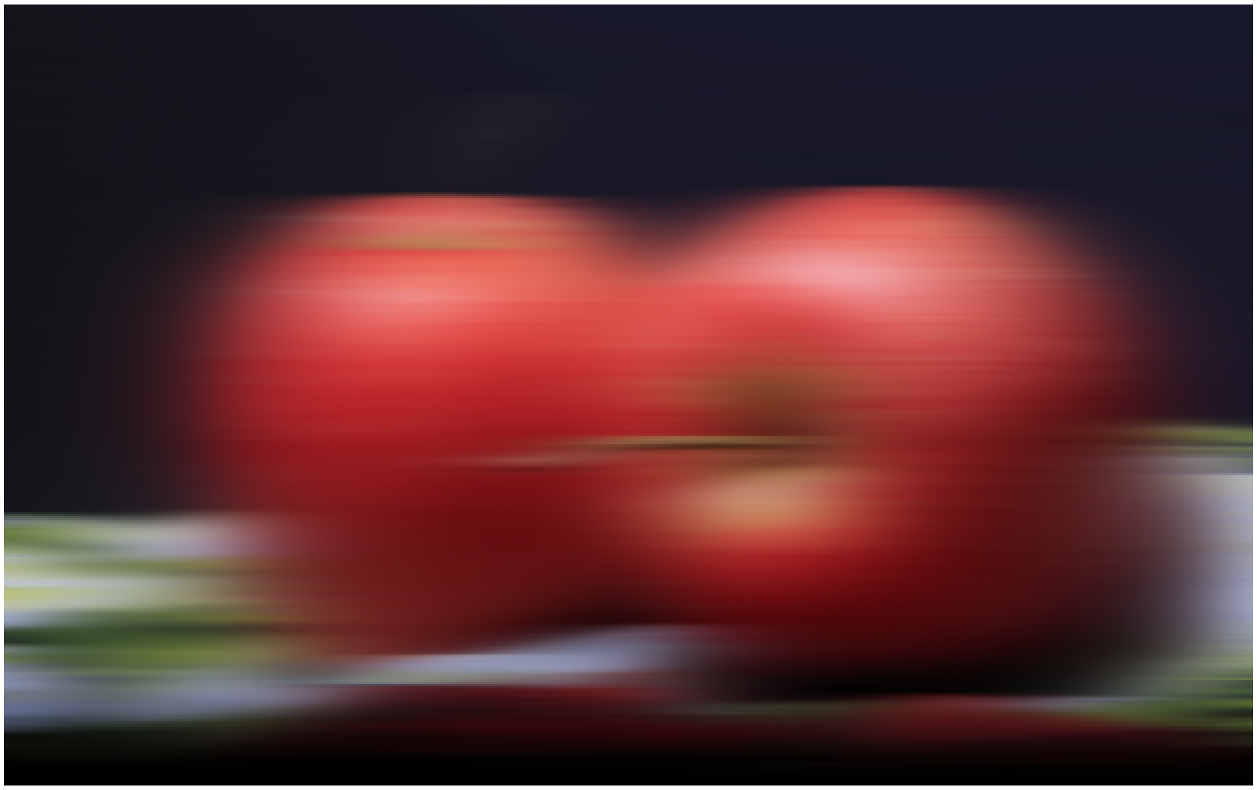


Fig. 4: sp apples blur_x 30

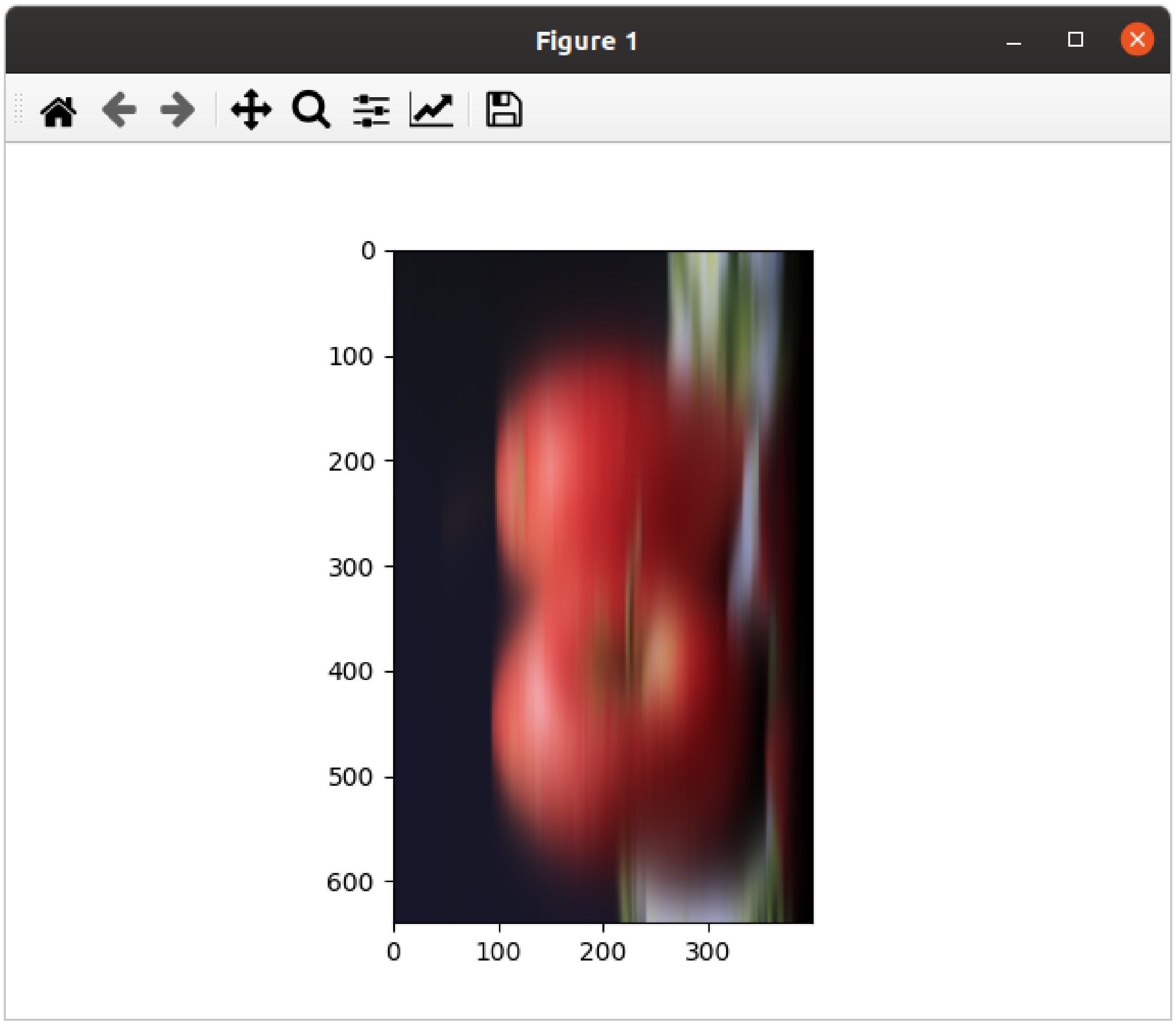
Now let us try a display from Numpy using Matplotlib

```
numpy_im = images[0].to_numpy() # deinterleaves, keeps G'MIC w,h,d,s shape
print(numpy_im).shape # (400, 640, 1, 3)

# prepare a Matplotlib display with the dimensions=1 hidden to have a proper 2D image
```

(continues on next page)

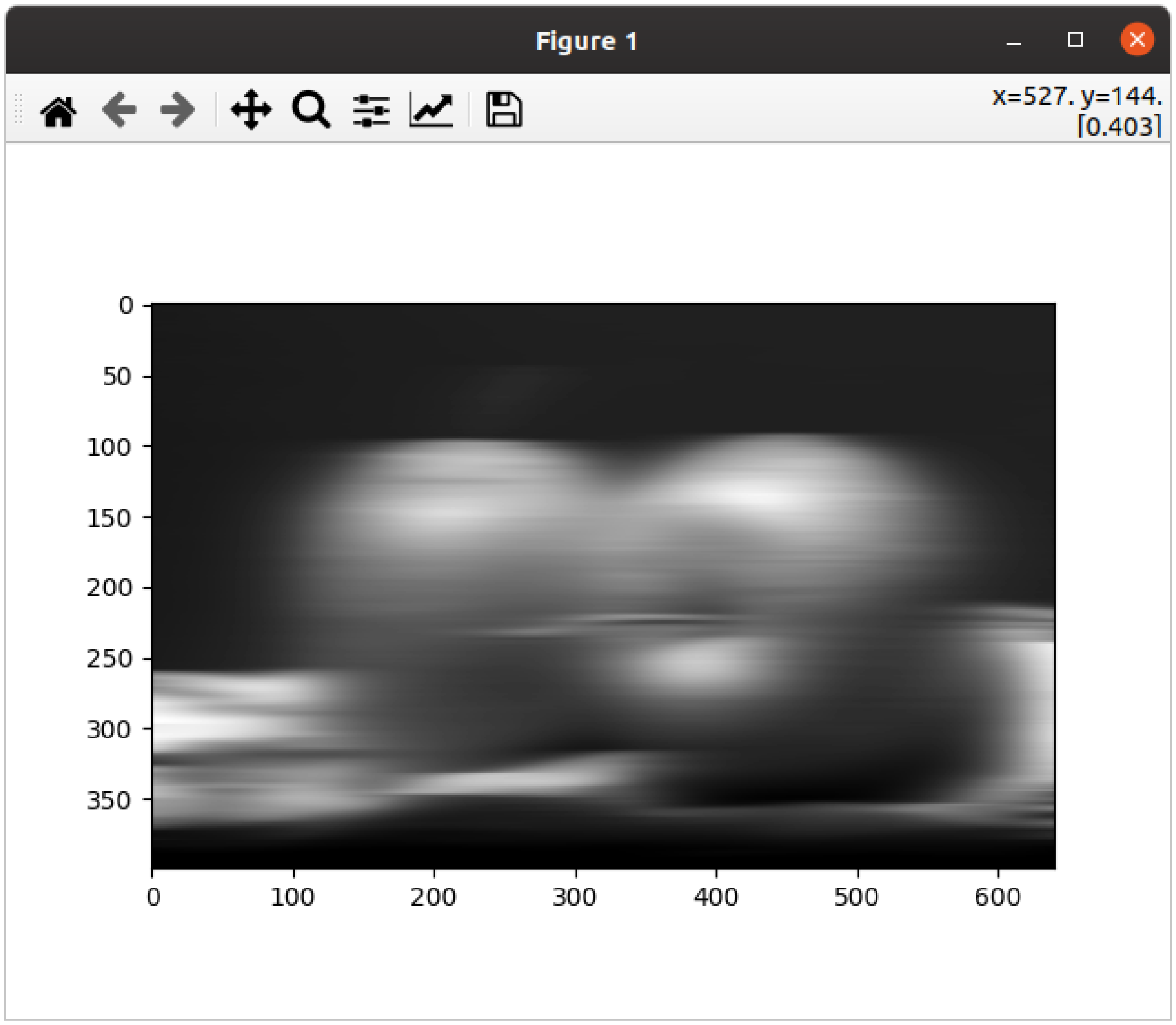
(continued from previous page)

```
# by default, Matplotlib does not like float with a range in 0-1, so we normalize␣
↪values by dividing by 255
numpy_for_plt = numpy_im.squeeze()/255

# time for displaying
plt.imshow(numpy_for_plt)
plt.show() # See rotated apples
```



Now let us fix those apples orientation and use numpy for greyscale filtering

```
# Pure-numpy greyscaling per https://stackoverflow.com/a/51571053/420684
numpy_for_plt = numpy.dot(numpy_for_plt[... , :3] , [0.299 , 0.587, 0.114])
numpy_for_plt.transpose((1,0,2)) # flipping x and y axes

# Time for redisplaying
plt.imshow(numpy_for_plt, plt.get_cmap('gray'))
plt.show()
```



To sum up, here was different fast way to work from G'MIC, with more work on the G'MIC side

```
import numpy
from matplotlib import pyplot as plt
import gmic

images = []
# The gmic.eu website shows various types of B&W: https://gmic.eu/gallery/
↪blackandwhite.shtml
gmic.run("sp apples blur_x 30 fx_freaky_bw 90,20,0,0,0,0", images) # horizontal␣
↪blur+special black&white
numpy_im = images[0].to_numpy_helper(interleave=True,permute="yxzc", squeeze_
↪shape=True, astype=numpy.uint8)
plt.imshow(numpy_im, plt.get_cmap('gray'))
plt.show()
```

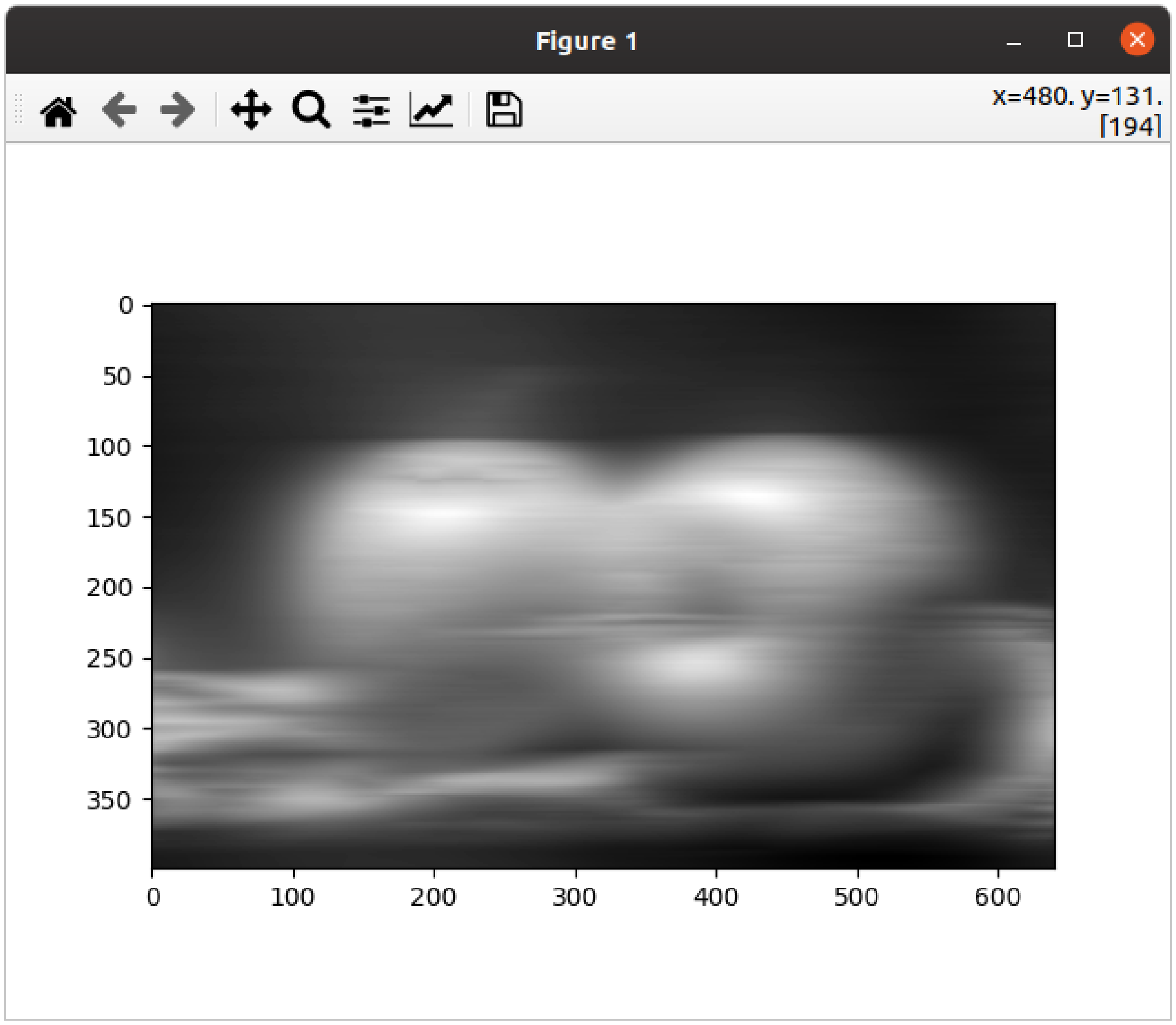


Figure 1
x=480. y=131.
[194]
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600

CHAPTER

THREE

TUTORIALS

This is the good place to start if you know a tiny bit of Python but very little about G'MIC commands.

If you are in a hurry, you may head to the related G'MIC Python script and run it right away after a `pip install gmic`.

3.1 Tutorial 1 - G'MIC commands syntax

Here is a little theory about G'MIC commands signature, commands piping and buffer indexes. This sounds complicated but is not at all. This tutorial is for very beginner and easy to follow.

3.1.1 Little theory on G'MIC commands

For this section, you do not need to code or run anything at this point. Just meditate.

G'MIC provides more that 500 commands or filters. Which can get updated from the internet by running the *update* or *up* command. This spares you from updating your gmic binary runtime frequently. Instead, a definitions file of many G'MIC commands gets refreshed on your machine, downloaded from gmic.eu.

G'MIC stores images internally in the x,y,z,c space (or width, height, depth, channels), allowing you to work with 2D greyscale images, 2D color images, or voxels (3D points) for many applications.

For executing expressions, the G'MIC language parser allows you to pipe commands from left to right:

```
<command 1> <command 2> <command 3>
```

where the result(s) of command 1 is piped as the input of command 2 which outputs in turn to command 3.

In G'MIC each command outputs most of the time a G'MIC Image (or G'MIC Image list if several images are there).

Commands outputting corresponds most of the time to **in-place** transformation within an input images list. Some times (or always when prefixed with a + sign, eg. *+blur*), commands can append a new result image to an input images list.

G'MIC commands thus take as input: G'MIC image(s) (or "buffers"), coming from their left side, and also G'MIC image(s) and/or scalar parameters (integers, floats, strings..) on the right side:

```
<former command's output images> <command 1> <command 1 image and/or scalar␣
↪parameters>
```

Example:

```
input myimage.png blur 3,1
```

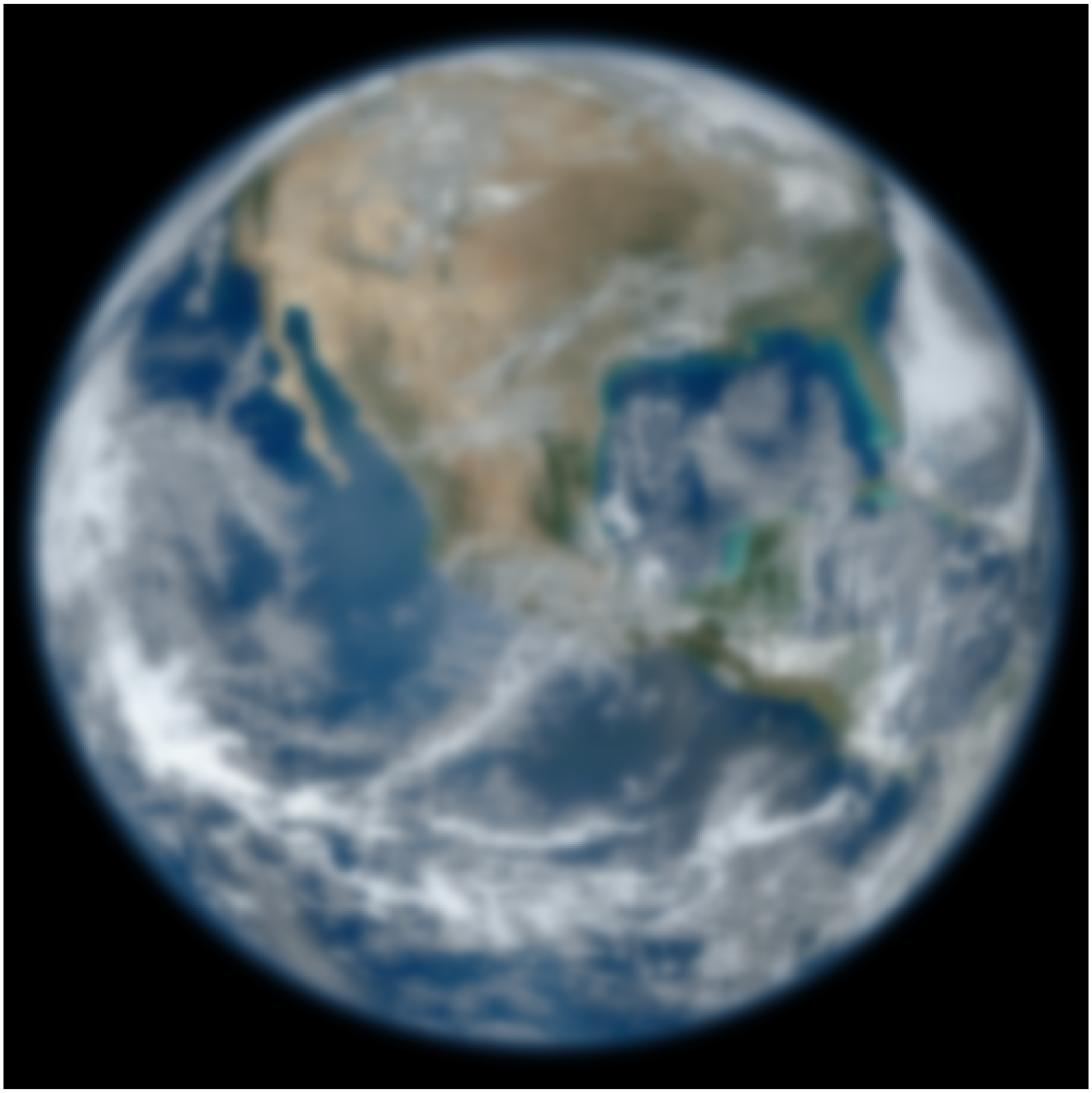


Fig. 1: sample earth blur 3,1

The above command will:

1. open *myimage.png* from the G'MIC executable current directory,
2. load it into an G'MIC Images list, which will have 1 item only,
3. then pipe the list as input to the blur command with parameters *3* (standard deviation) and *1* (Neumann boundary conditions).

From there you will see nothing unless you use the G'MIC command line executable (it pops a display window automatically), so *output <filename>* or *display* are your friends for writing or showing a result window.

Here is a more complex example with more commands piping, where the G'MIC List of 2 G'MIC Images is passed modified in place from left to right:

```
input image1.png image2.png blur 4 sharpen 30 smooth 200 display
```

You can add a '-' prefix before your G'MIC commands to make them stand out a little more. It changes absolutely nothing for the result.

```
-input image1.png image2.png -blur 4 -sharpen 30 -smooth 200 -display
```

If you want any command in your chain to use only a subset of the leftside results, use the *[]* index suffix, it will keep the images in your list in place though

```
-input image1.png image2.png blur[1] 3 display[0]
```

The above command will actually blur *image2.png* but display *image1.png* only, which is not blurred.

Note that G'MIC is a full fledged scripting language, with variables and control flow: repeat, if condition etc... You may also create custom commands or filters.

The gmic.eu website technical's reference and the G'MIC community Github repository are good friends to learn and ask about G'MIC.

Now is the time to learn some very basic foundational commands for learning and working with G'MIC.

3.1.2 The `help <some command>` command

```
import gmic
gmic.run("help blur")

# Outputs
"""


  gmic: GREYC's Magic for Image Computing: command-line interface
        (https://gmic.eu)
        Version 2.9.1

        Copyright (c) 2008-2020, David Tschumperlé / GREYC / CNRS.
        (https://www.greyc.fr)

    blur (+):
        std_deviation>=0[%],_boundary_conditions,_kernel |
        axes,std_deviation>=0[%],_boundary_conditions,_kernel

      Blur selected images by a quasi-gaussian or gaussian filter (recursive␣
↪implementation).
```

(continues on next page)

(continued from previous page)

```
        (eq. to 'b').

        'boundary_conditions' can be { 0=dirichlet | 1=neumann }.
        'kernel' can be { 0=quasi-gaussian (faster) | 1=gaussian }.
        When specified, argument 'axes' is a sequence of { x | y | z | c }.
        Specifying one axis multiple times apply also the blur multiple times.
        Default values: 'boundary_conditions=1' and 'kernel=0'.

        Example: [#1]  image.jpg +blur 5,0 +blur[0] 5,1

                 [#2]  image.jpg +blur y,10%

        Tutorial: https://gmic.eu/tutorial/_blur.shtml

"""
```

Just in case you are new to Python, you can use the help() command on any G'MIC object.

```
import gmic
help(gmic)
# Outputs:
"""
Help on module gmic:

NAME
    gmic - G'MIC Image Processing library Python binding

DESCRIPTION
    Use gmic.run(...), gmic.GmicImage(...), gmic.Gmic(...).
    Make sure to visit https://github.com/myselfhimself/gmic-py for examples and␣
↪documentation.
"""

help(gmic.run)
# Outputs:
"""
Help on built-in function run in module gmic:

run(...)
    run(command: str[, images: GmicImage|List[GmicImage], image_names:␣
↪str|List[str]]) -> None
"""
```

3.1.3 The sample or sp command

The sample or sp command allows you to load sample images provided by G'MIC. None are pre-downloaded by default.

If you see network errors when running examples for this section, you should install the curl or wget executable, change your firewall settings or just skip using the sample command for now. You may also directly read the section about input instead below, which allows you to load your own local images.

First pop some help about the command, and notice how many sample image names you can use as a first parameter.

```
import gmic
gmic.run("help sample") # or "help sp"

# OUTPUTS
"""
  gmic: GREYC's Magic for Image Computing: command-line interface
        (https://gmic.eu)
        Version 2.9.1

        Copyright (c) 2008-2020, David Tschumperlé / GREYC / CNRS.
        (https://www.greyc.fr)

    sample:
        _name1={ ? | apples | balloons | barbara | boats | bottles | butterfly | \
         cameraman | car | cat | cliff | chick | colorful | david | dog | duck |␣
↪eagle | \
         elephant | earth | flower | fruits | gmicky | gmicky_mahvin | gmicky_wilber␣
↪| \
         greece | gummy | house | inside | landscape | leaf | lena | leno | lion | \
         mandrill | monalisa | monkey | parrots | pencils | peppers | portrait0 | \
         portrait1 | portrait2 | portrait3 | portrait4 | portrait5 | portrait6 | \
         portrait7 | portrait8 | portrait9 | roddy | rooster | rose | square | swan |␣
↪\
         teddy | tiger | tulips | wall | waterfall | zelda },_name2,...,_nameN,_width=
↪{ \
         >=0 | 0 (auto) },_height = { >=0 | 0 (auto) } |
        (no arg)

      Input a new sample RGB image (opt. with specified size).
      (eq. to 'sp').

      Argument 'name' can be replaced by an integer which serves as a sample index.

      Example: [#1]  repeat 6 sample done
"""
```

Let us run the command once, to download an `apples` file if not on your computer yet, and loading it into our now empty images buffer:

```
import gmic
gmic.run("sample apples") # A display window would pop up in gmic's command line␣
↪executable, but not in Python that is intended!
```

3.1.4 The `display` command

The display command is twofolds: - it displays textual information about an image, - if the environment (operating system, terminal, IPython-like shell…) allows it, tries to show the image in some G'MIC image window, matplotlib view etc..

```
import gmic
gmic.run("sample apples display") # This will pop up a display window showing your␣
↪image, without it needing to be saved anyway on your drive
gmic.run("sample duck sample apples display[0]") # Same but will show only index 0␣
↪image, ie. the duck
```

(continues on next page)

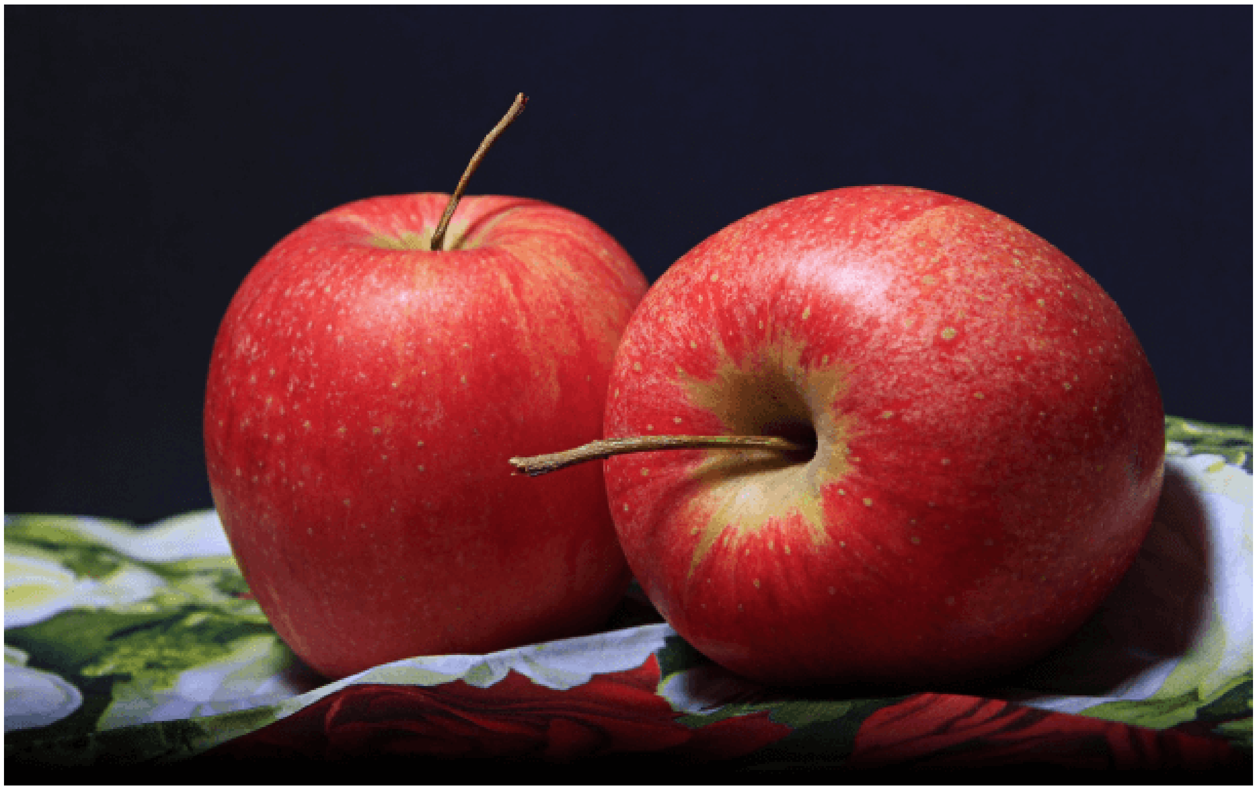


Fig. 2: sample apples

(continued from previous page)

```
# OUTPUTS
"""
[gmic]-1./ Display image [0] = 'apples', from point (320,200,0).
[0] = 'apples':
  size = (640,400,1,3) [3000 Kio of floats].
  data = (20,22,20,20,20,22,22,22,22,22,22,20,(...),1,1,1,1,1,1,1,1,1,1,1,1).
  min = 1, max = 250, mean = 58.5602, std = 59.8916, coords_min = (317,306,0,1),␣
↪coords_max = (430,135,0,0).
[gmic]-2./ Display image [0] = 'duck', from point (320,240,0).
[0] = 'duck':
  size = (640,480,1,3) [3600 Kio of floats].
  data = (89,89,74,89,89,89,74,89,89,89,89,89,(...),177,190,177,215,181,194,206,201,
↪153,201,161,209).
  min = 1, max = 253, mean = 125.444, std = 57.4846, coords_min = (364,72,0,2),␣
↪coords_max = (413,123,0,0).
"""
```

3.1.5 The `print` command

This command is similar to the `display` command except that it shows no picture, it just outputs text.

```
import gmic
gmic.run("sp leno print")
```

(continues on next page)

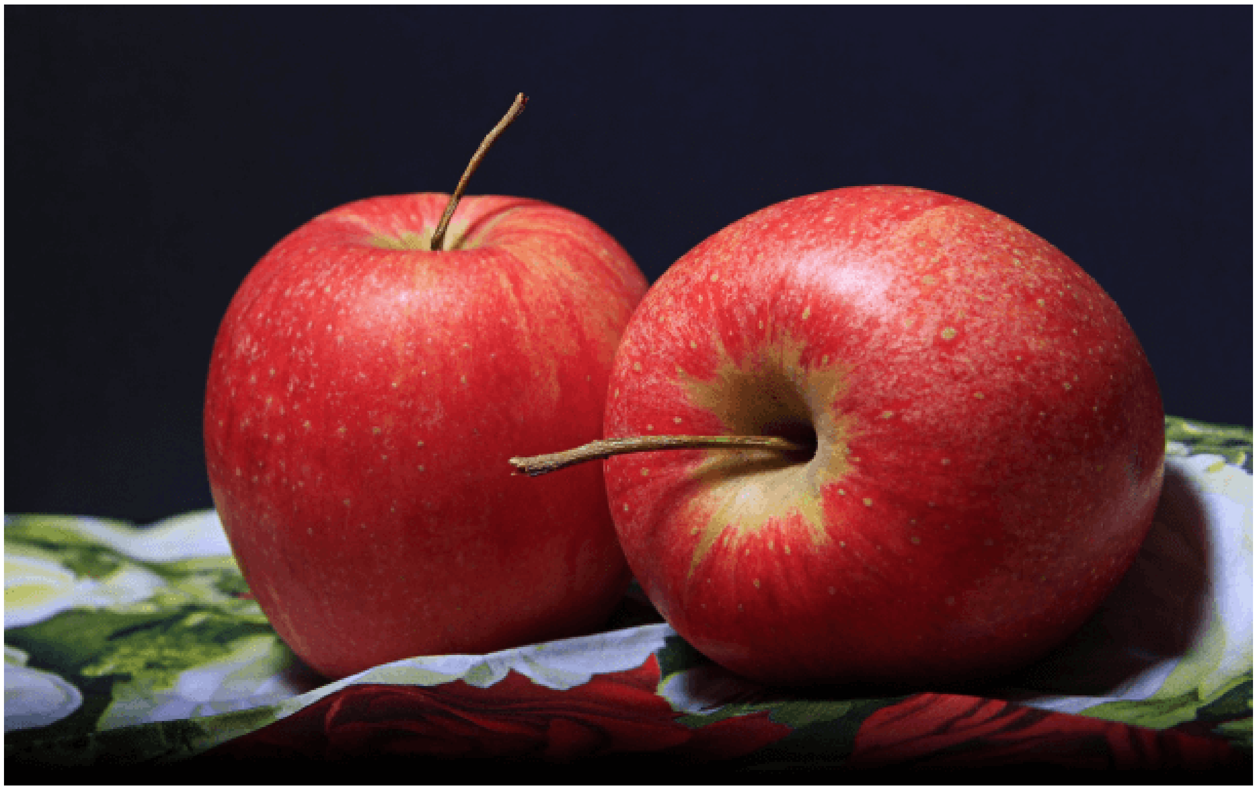


Fig. 3: sample apples

(continued from previous page)

```
# OUTPUTS
"""
[gmic]-1./ Print image [0] = 'leno'.
[0] = 'leno':
  size = (512,512,1,3) [3072 Kio of floats].
  data = (224,224,223,224,224,225,224,224,224,224,224,224,(...),69,85,85,79,87,79,85,
↪90,77,77,79,84).
  min = 1, max = 255, mean = 128.318, std = 58.3599, coords_min = (508,71,0,1),␣
↪coords_max = (124,189,0,0).
"""
```

3.1.6 The *output <file>* command

This command writes your images list's contents to files, using file extension detection.

```
import gmic
gmic.run("sample earth output myearth.png") # outputs the result of the earth sample␣
↪to a path you want (.png, .jpeg, .tiff, .bmp, .pbm and more are supported)
gmic.run("sample earth elephant output mysamples.jpeg") # outputs the result to␣
↪mysamples_NNNN.jpeg
gmic.run("sample earth elephant output[1] myelephant.jpeg") # outputs the second␣
↪image (index 1, starting at 0) to a single JPEG file
```

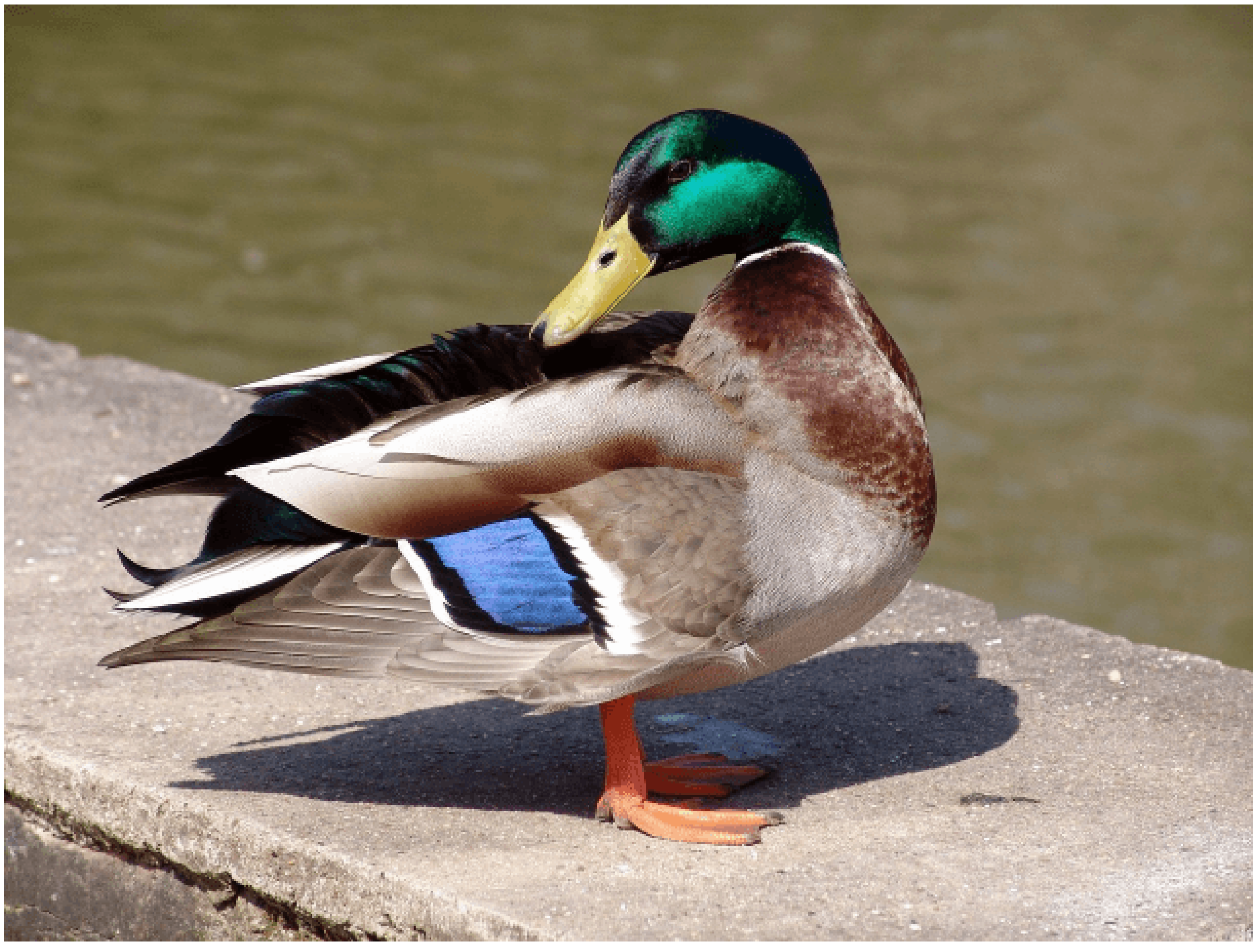


Fig. 4: sample duck

3.1.7 The `input <somefile>` command (simple and short form)

This command fills your image(s) list with the contents of files. Note that G'MIC may also allows to open video files directly, especially if OpenCV is linked, although the official gmic-py release does not link to OpenCV.

```
import gmic

# LOADING AND SHOWING A SINGLE IMAGE
gmic.run("input myearth.png display") # opens myearth.png and then trying a display
gmic.run("myearth.png display") # here is the short form, where 'input' can be␣
↪omitted. Note that the 'display' command is not obligatory, it is left for you as a␣
↪proof that it works.

# LOADING AND SAVING MULTIPLE IMAGES
gmic.run("sample earth sample apples output myimages.png display") # saves to␣
↪myimages_000000.png  myimages_000001.png. The display command is optional.
gmic.run("myimages_000000.png myimages_000001.png display") # loads myimages_000000.
↪png  myimages_000001.png and displays them. Note the 'input' command name was␣
↪omitted.
```

3.1.8 Applying a one or more filter(s)

Filtering images is what G'MIC is good at, and especially what most users do with G'MIC.

Official filters and commands are listed at: https://gmic.eu/reference/, especially in the Filtering section.

The G'MIC QT plug-in for GIMP and other graphic software provide more filters, which usually wrap those official filters and have internal layer management specificities. If you use the latter (they are usually prefixed in `fx_` or `gimp_` or `gui_`, beware their community authors do not always care about stability or allowing the same parameters' order or meaning! A Gist file explains this in more technical detail if you are very curious.

To get inspiration for commands to run, you may also head to the G'MIC gallery and click the images to see corresponding commands.

Here are some example commands and filters:

```
import gmic
gmic.run("sample apples blur 4 display") # blur's documentation with a nice preview␣
↪is also at https://gmic.eu/reference/blur.html not just through the "help blur"␣
↪command
```

```
import gmic
gmic.run("sample apples rodilius 10 display") # more at https://gmic.eu/reference/
↪rodilius.html
```

```
# SELECTING IMAGES BY INDEX
import gmic
gmic.run("sample apples sample earth blur[1] 4 display") # Here the blur of strength␣
↪4 was applied only to image with second position
```

```
import gmic
# APPLYING THE FAMOUS 'STYLIZE' STYLE TRANSFER FILTER
gmic.run("sp leno display") # this is the portrait we will want to be stylized
gmic.run("_fx_stylize landscapenearantwerp display") # let us be hackish and use the␣
↪internal _fx_stylize function to preview one of Antwerp's painting as a future style
```

(continues on next page)

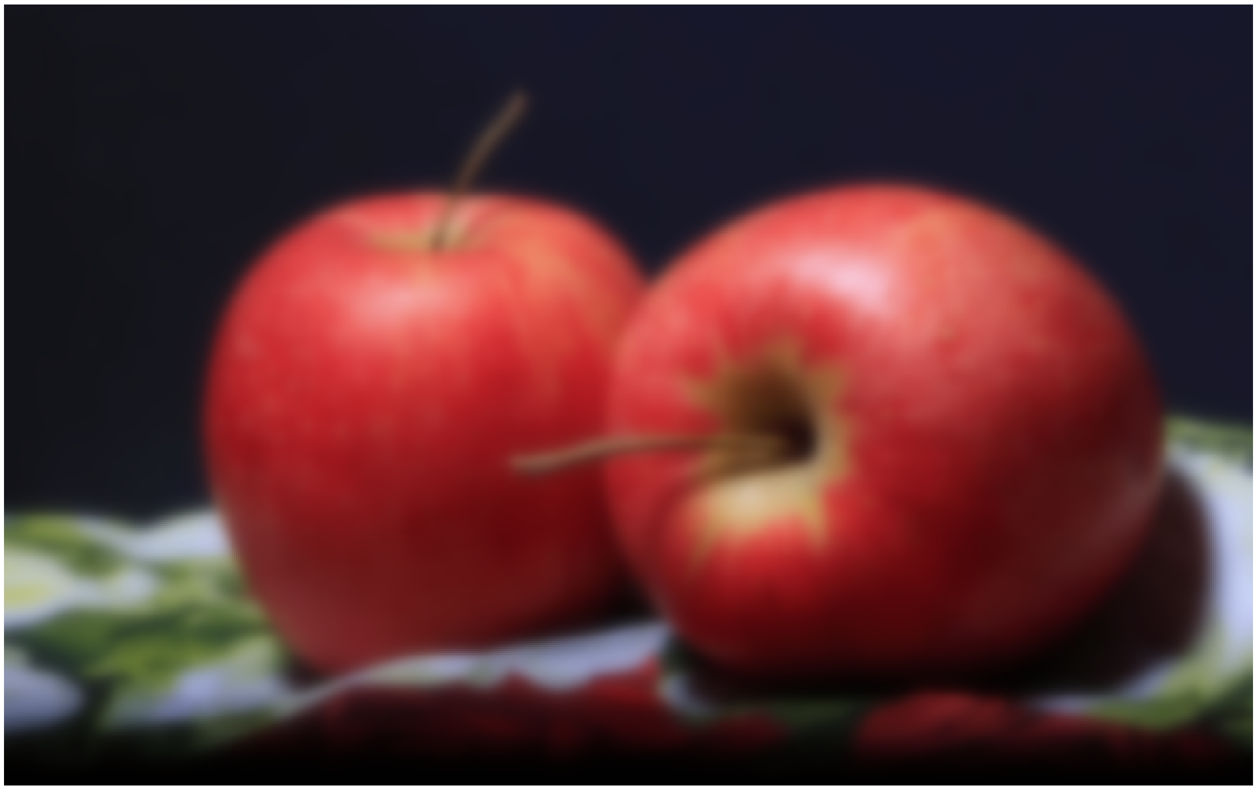


Fig. 5: sample apples blur 4

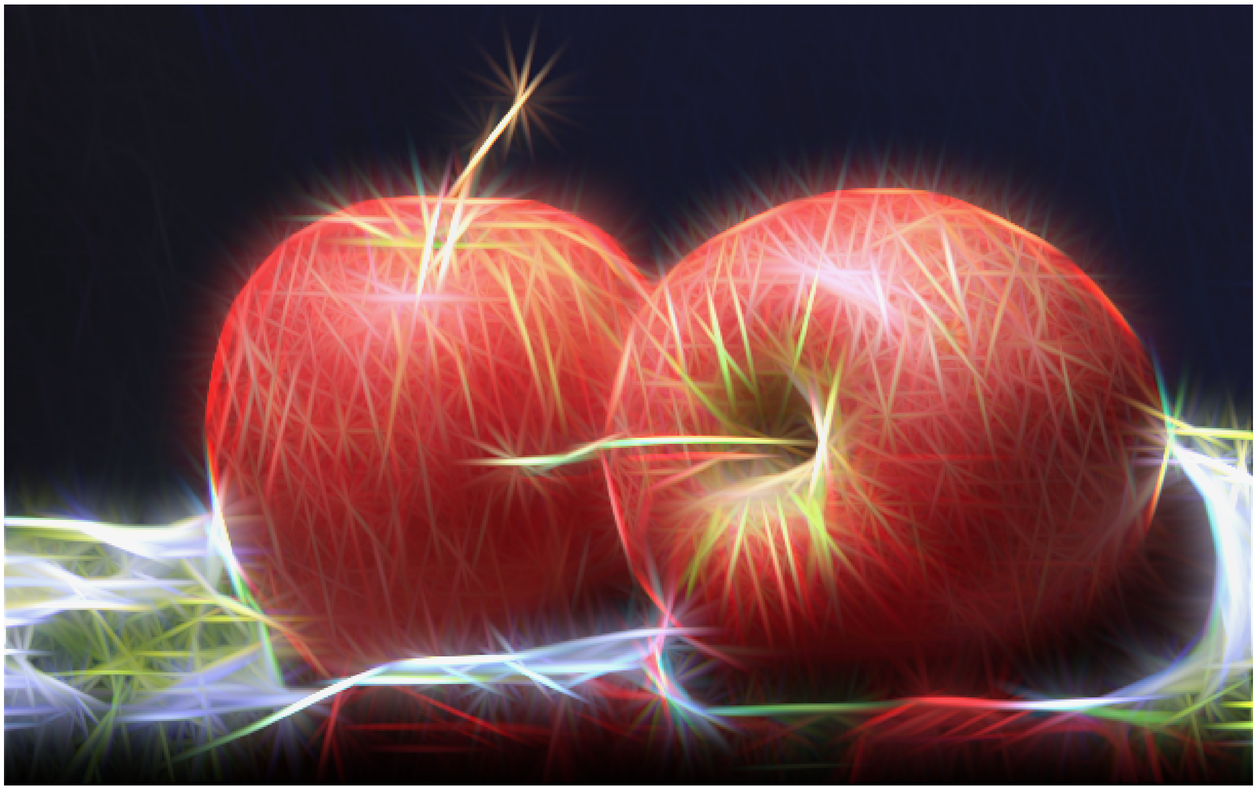


Fig. 6: sample apples rodilius 10

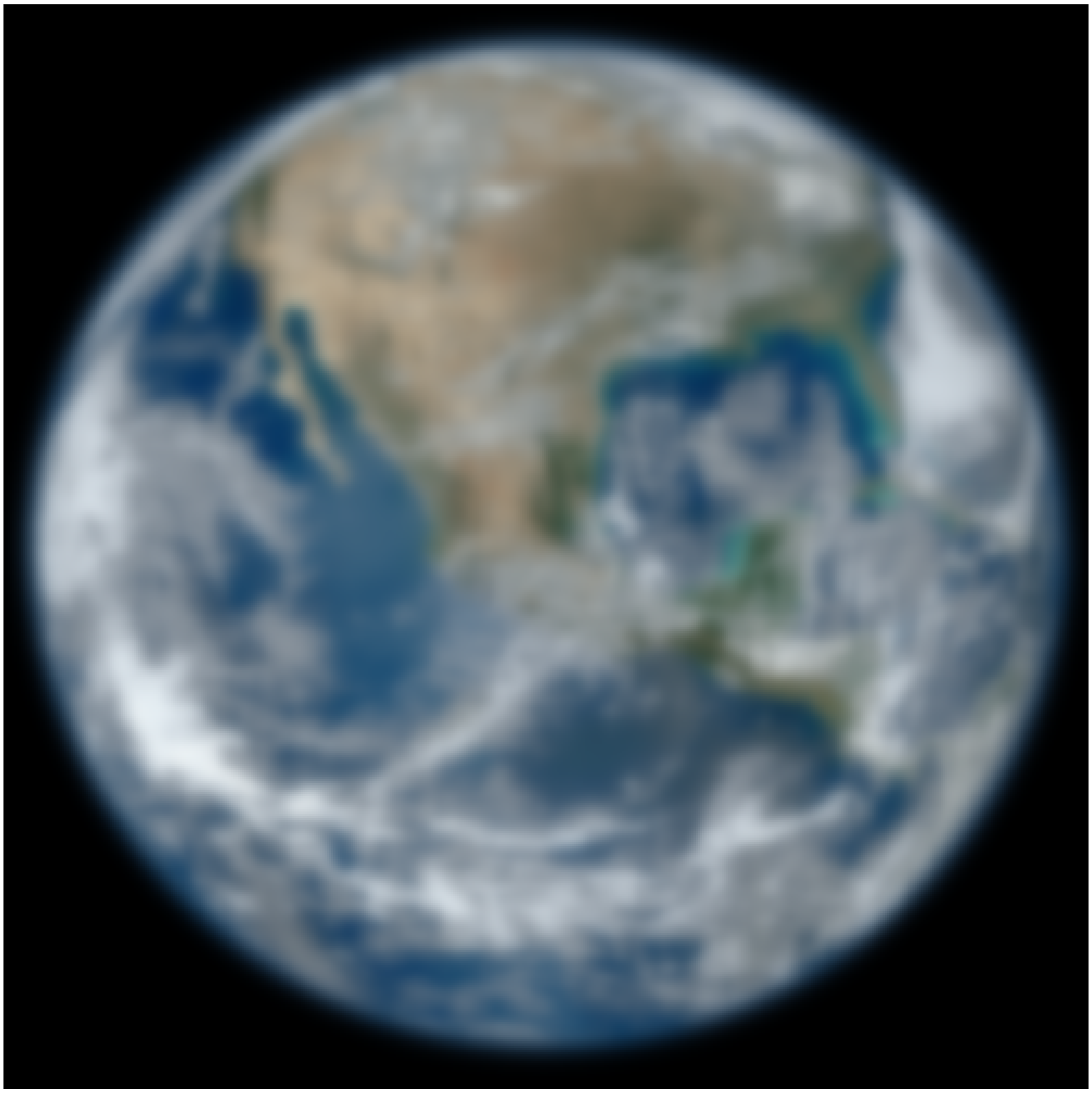


Fig. 7: sample apples sample earth blur[1] 4 keep[1]

(continued from previous page)

```
gmic.run("sample leno _fx_stylize landscapenearantwerp stylize[0] [1] display")
```

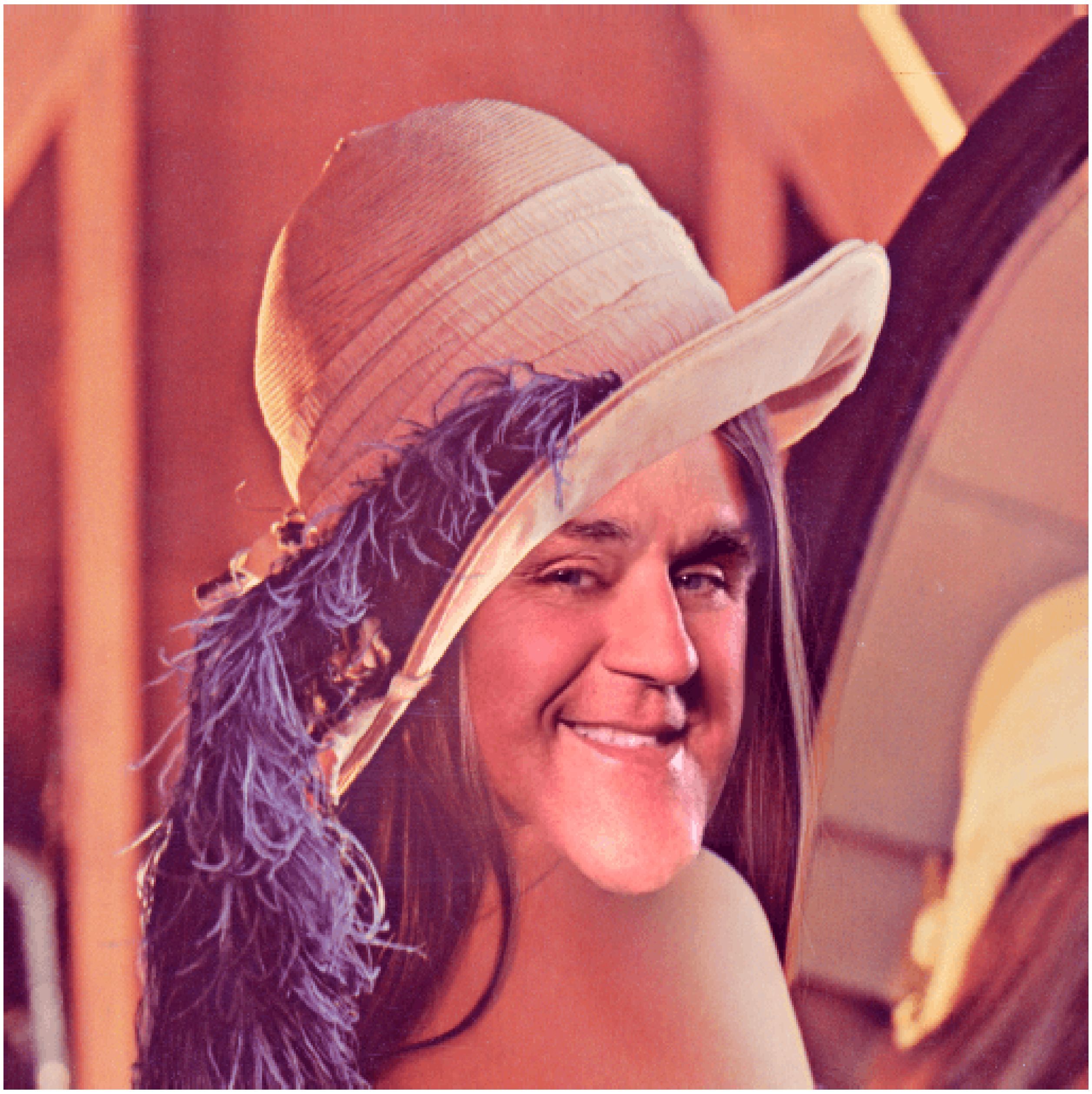


Fig. 8: sp leno

```
# APPLYING MULTIPLE FILTERS
# ONCE
gmic.run("sample duck smooth 40,0,1,1,2 display")
# 3 TIMES
gmic.run("sample duck repeat 3 smooth 40,0,1,1,2 done display")
# SEVERAL FILTERS IN A ROW
gmic.run("sample duck repeat 3 smooth 40,0,1,1,2 done blur xy,5 rodilius , display")
```

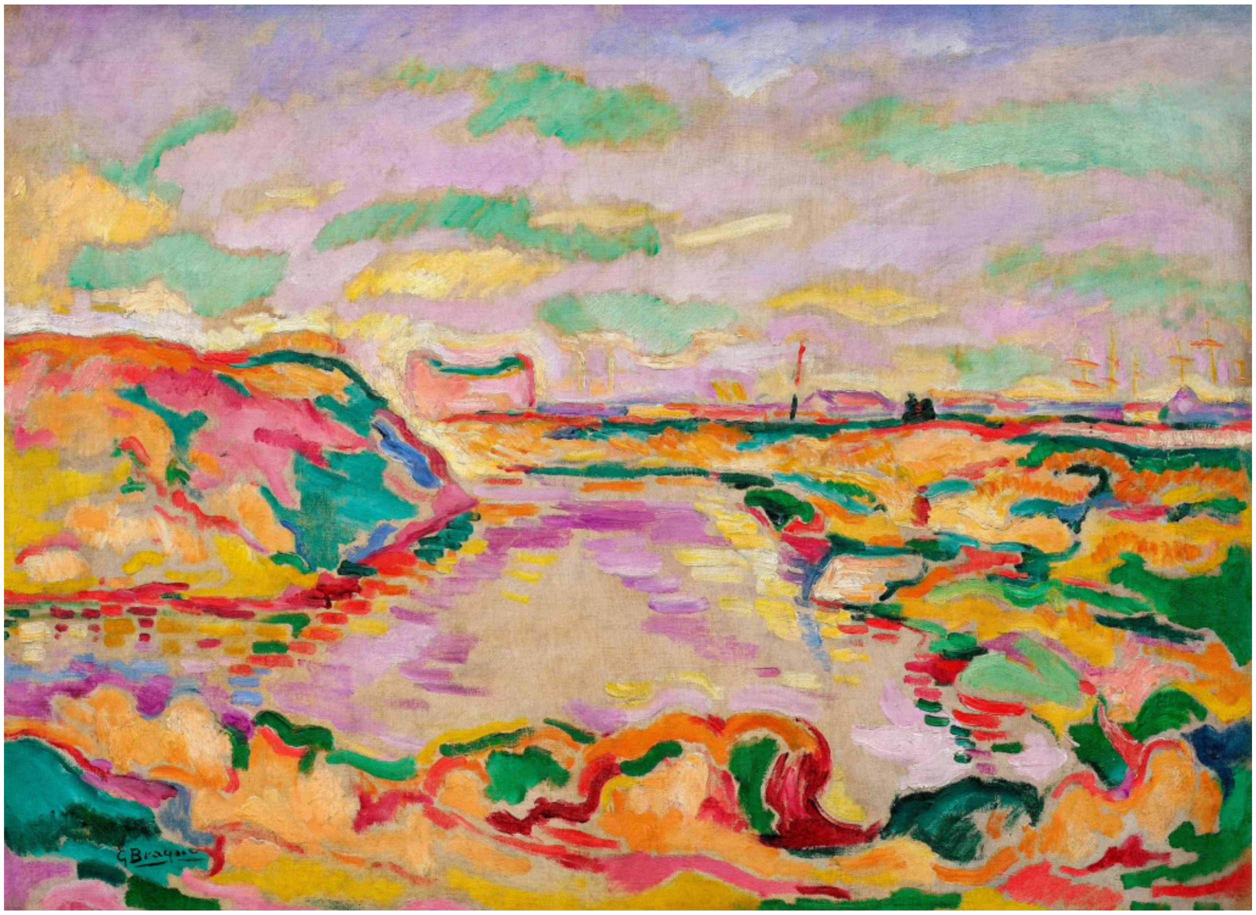


Fig. 9: _fx_stylize landscapenearantwerp

../../../../../../../../../tmp/gmic_sphinxe40pa8mv/99096a

Fig. 10: sample leno _fx_stylize landscapenearantwerp stylize[0] [1]

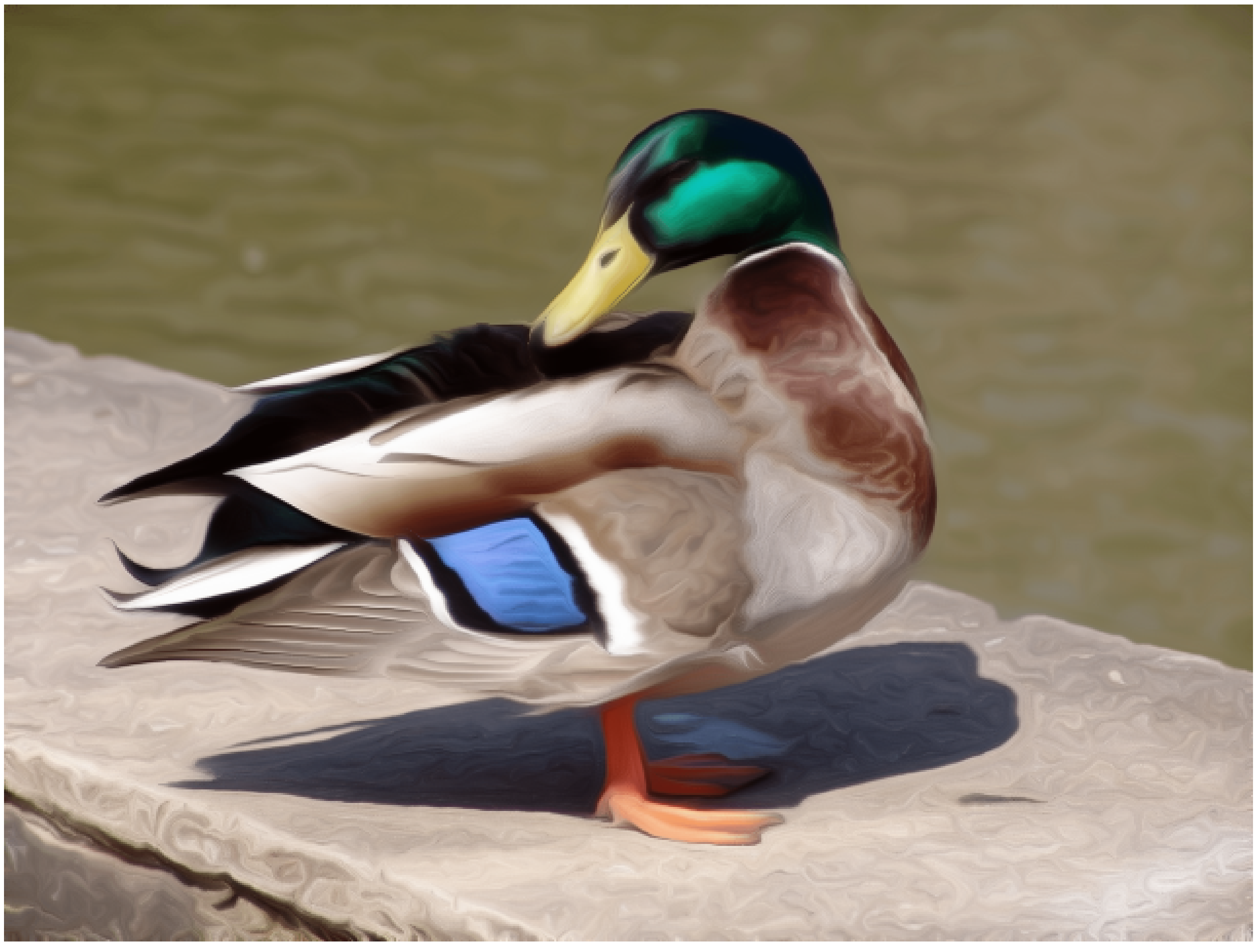


Fig. 11: sample duck smooth 40,0,1,1,2

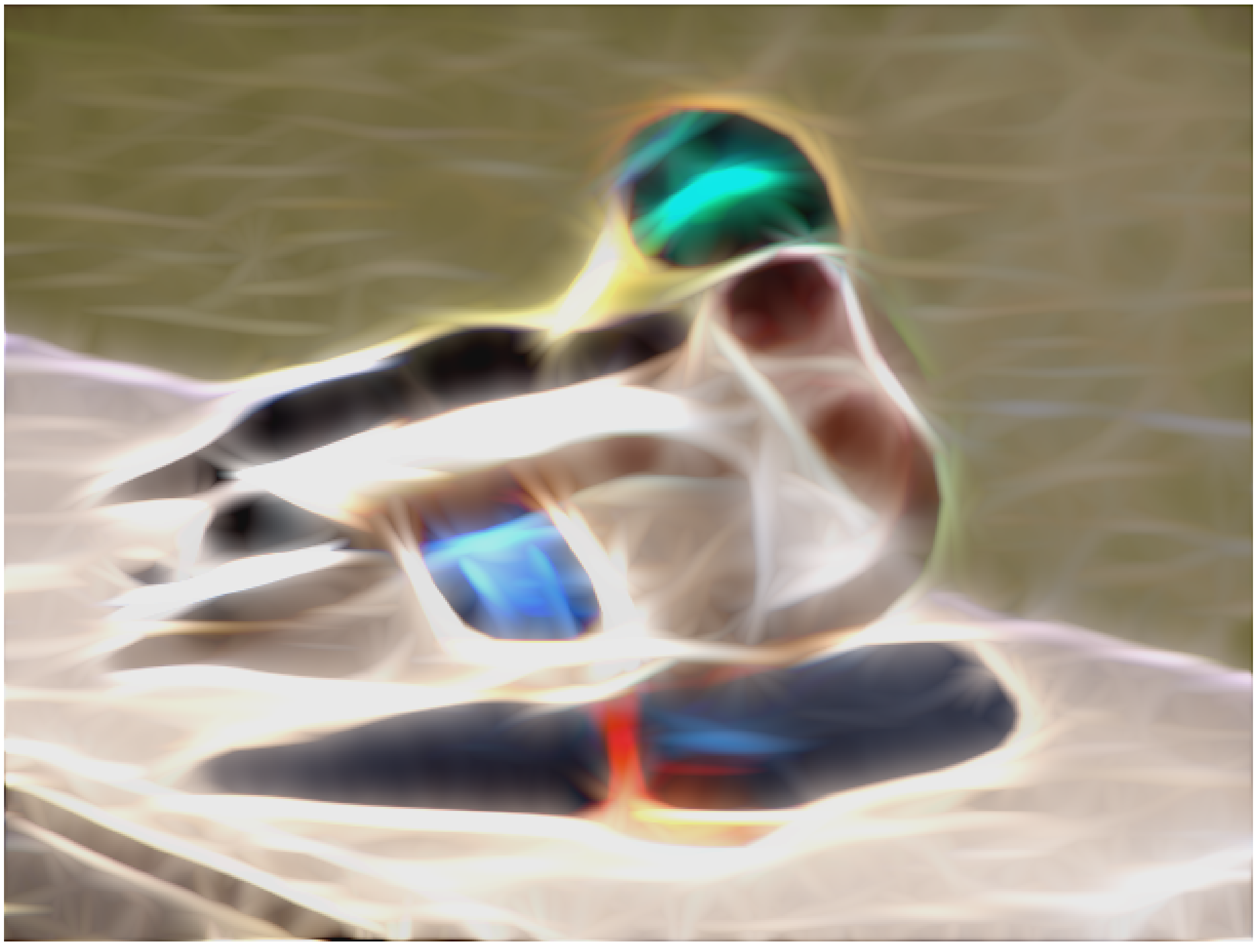


Fig. 12: sample duck repeat 3 smooth 40,0,1,1,2 done blur xy,5 rodilius ,

3.2 Tutorial 2 - Optimization, GmicImage, lists, stylize

The Python binding for G'MIC or `gmic-py` (although you "pip install gmic" and "import gmic") is quite rudimentary. `gmic-py` tries to bring together the advantages of the `gmic` command line tool (a sort of G'MIC language evaluator) with the speed and API-similarity of G'MIC's C++ library.

Below you will discover core knowledge of `gmic-py` to optimize your scripts' processing speed a bit. This will maybe be boring, but investing time there will allow you to spare CPU time and avoid superfluous file reads-writes, especially if you use `gmic-py` in some bigger back-end or front-end applications.

Note though, that some of knowledge of how the `GmicImage` class works is needed, so you might want to read the related section below beforehand.

In this tutorial, let us see how the 3 building blocks of gmic-py can be used together: the interpreter, single images, and images lists. In *Tutorial 1 - G'MIC commands syntax*, you have used the G'MIC interpreter mostly, without noticing how it was instantiated, and used file input and output to avoid Python-level images management.

3.2.1 The G'MIC module - for debugging's sake mostly

Let us dive into the Python `gmic` module elements.

```
import gmic

print(dir(gmic))

# Outputs:
"""
['Gmic', 'GmicException', 'GmicImage', '__build__', '__doc__', '__file__', '__loader__
↪', '__name__', '__package__', '__spec__', '__version__', 'run']
"""
```

Most important objects you see in this module-level list are:

- `Gmic` - the G'MIC language intepreter class
- `run` - is the `gmic.run()` call you see in most `gmic-py`'s examples out there. It is a shortcut to `gmic.Gmic().run` or `gmic.Gmic(commands)` for beginners to kick-off running G'MIC expressions right away
- `GmicException` - an generic exception thrown by most G'MIC classes (along with standard Python exceptions, such as ValueError etc)
- `GmicImage` - a wrapper around C++'s gmic_image class (a CImg alias)

Now some quick and dirty notes on module variables:

```
import gmic

print(gmic.__spec__)  # path were your compiled G'MIC Python loaded shared library␣
↪lives. Mostly useful to people installing gmic-py several times.
# Outputs:
# ModuleSpec(name='gmic', loader=<_frozen_importlib_external.ExtensionFileLoader␣
↪object at 0x7fd7f45ab0b8>, origin='/export/home/AAA/.virtualenvs/gmic-sphinx/lib/
↪python3.6/site-packages/gmic.cpython-36m-x86_64-linux-gnu.so')

print(gmic.__version__)  # version of the embedded the libgmic C++ interpreter
# Outputs:
# 2.9.0
```

(continues on next page)

(continued from previous page)

```
print(gmic.__build__)  # flags that were used for compilation.
"""This allows to understand fast if your gmic-py provides jpeg, png, tiff support
interesting flags are: openMP is for parallel computing
fftw3 is needed for spectrum-based computing and managing images with dimensions not␣
↪in power of 2
OpenCV is by default not linked to gmic-py, although you could rebuild gmic-py easily␣
↪and use it
"""
# Outputs:
# zlib_enabled:1 libpng_enabled:1 display_enabled:1 fftw3_enabled:1 libcurl_enabled:1␣
↪openmp_enabled:1 cimg_OS:1 numpy_enabled:1 OS_type:unix

help(gmic) # shows an introduction about gmic-py
# G'MIC language help can be read using gmic.run("help <somecommand>")
```

3.2.2 Gmic - The G'MIC interpreter class

`gmic.run` is a function which spawns a G'MIC interpreter object for you, evaluates your command, then deletes the interpreter object. For those literate in computer science, there is no singleton design pattern in use here and no interpreter gets cached.

Let us see working but unoptimized example of evaluating several commands:

```
import gmic
gmic.run("sp apples rodilius 3 display")
gmic.run("sp earth blur 5 display")
```

In pure Python, the above two lines would be the same as doing (being unsure of when garbage collection for memory-living G'MIC interpreters would happen):

```
import gmic

g1 = gmic.Gmic()
g1.run("sp apples rodilius 3 display")
del g1
g2 = gmic.Gmic()
g2.run("sp earth blur 5 display")
del g2
```

`gmic.Gmic()` instantiates a G'MIC intepreter class. Under the hood, the G'MIC c++ library is made to read its configuration, set up internal variables, detect operating system capabilities etc..

This is a bit heavy and you may not want to repeat that! For simplicity though, most `gmic-py` beginner tutorials just write gmic.run().

Here is the better way to evaluate several commands in a row using a single G'MIC interpreter instance:

```
import gmic

g = gmic.Gmic()  # First create a G'MIC interpreter instance using the Gmic class,␣
↪and attach to a variable by a simple assignment
g.run("sp apples rodilius 3")  # Reuse your variable as many times as you want, and␣
↪call its run() method.
g.run("sp apples blur 5")  # Here you are, a 2nd call, where the G'MIC interpreter␣
↪was not recreated for nothing!
```

YAY!!! Optimization!!

The G'MIC interpreter does not store images between each evaluation, they are destroyed unless you keep them attached to a Python variable.

Passing in a pure-Python list of G'MIC images is the way to keep track of your images in memory. This will be shown a bit further in the next two sections.

Especially, as the run() method actually takes 3 parameters: - a command(s) string, - an optional list of G'MIC images, - an optional list of G'MIC image names. You can read more about this by running `help(gmic.Gmic)` or visiting the API reference.

3.2.3 GmicImage - The G'MIC Image class

After discovering the `gmic.Gmic` interpreter class, the G'MIC Image is the other building block of `gmic-py` (and G'MIC C++).

Here is how to create one from scratch with no data:

```
import gmic

im = gmic.GmicImage() # without specifying any dimensions
im_ = gmic.GmicImage(width=640, height=480, spectrum=3) # with some dimensions␣
↪provided

# By the way:
help(gmic.GmicImage)  # Some mini-doc on how to call the GmicImage class
# Outputs:
"""
Help on class GmicImage in module gmic:

class GmicImage(builtins.object)
 |  GmicImage([data: bytes = None, width: int = 1, height: int = 1, depth: int = 1,␣
↪spectrum: int = 1, shared: bool = False]) -> bool
"""
```

Now let us take a look at the first image's properties (attributes):

```
print(dir(im))
# Outputs:
# ['__call__', '__class__', '__copy__', '__delattr__', '__dir__', '__doc__', '__eq__',
↪ '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__
↪init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_
↪ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
↪data', '_data_str', '_depth', '_height', '_is_shared', '_spectrum', '_width', 'from_
↪PIL', 'from_numpy', 'from_numpy_helper', 'from_skimage', 'to_PIL', 'to_numpy', 'to_
↪numpy_helper', 'to_skimage']
```

Here the most important attributes you see are:

- `_data`: a read-only 'bytes' buffer, which G'MIC reads and writes as a list of interleaved 32bit float. Interleaving and non-interleaving is a big topic, and G'MIC seems to an exception compared to many other graphics processing libraries: it stores pixel channel values, channel after channel. For example, for a 3x1 pixels RGB image, the _data would look like: `R1R2R3G1G2G3B1B2B3` instead of `R1G1B1R2G2B2R3G3B3`.
- `_width`, `_height`, `_depth`, `_spectrum`: read-only integers. G'MIC works in 3D if needed and stores its channels (eg. RGB, HSV, or a few million other channels) in the _spectrum dimensions. So an RGB 1024x768 screenshot would have those attributes as: 1024, 768, 1, 3. Any dimension must be >=1.

- `from_*` and `to_*`: conversion methods to and from other graphics libraries!! (as long as you install them first in your virtual environment or machine). Indeed, gmic-py was designed so that you spend more time using other famous tools you already love (numpy and PIL namely..) than working with the less famous `gmic-py`! Interoperability FTW!

Here are less important methods:

- `_data_str`: is not so important, but for your curiosity, it helps to decode the _data attribute as a unicode string!!! (in same some people wanted to store text in a G'MIC Image… the parse_gui command does this actually)
- `_is_shared` is never used in Python, is relevant when two interpreters or threads work on the `GmicImage`.

Let us see how a GmicImage is represented as a string:

```
print(im)
# Outputs:
# <gmic.GmicImage object at 0x7f5d6a028a70 with _data address at 0x26d0180, w=1 h=1␣
↪d=1 s=1 shared=0>
```

So we have just created a `GmicImage` with default parameters, that is an empty `_data` buffer, and dimensions `width=height=depth=spectrum=1` and no sharing.

If you know `numpy`, `GmicImage` object look like `numpy`'s ndarrays, though the former are much less practical to manipulate!!! They are actually a very superficial binding of G'MIC's C++ `gmic_image` / `cimg` image class.

To instantiate a `GmicImage`, you can pass in a `bytes` buffer, as well as the optional dimensions seen before: width, height, depth. Numpy does that as well. Here is a complex way to create a GmicImage from fixed bytes-level data and introspect it:

```
import gmic
import struct  # a handy native Python module to parse and build buffers

# Here we set up a GmicImage with 6 floats and dimensions 3x2x1
im2 = gmic.GmicImage(struct.pack("6f", 1, 2, 3, 4, 5, 6), 3, 2, 1)
# Let us print that image
print(im2)
# Outputs:
# <gmic.GmicImage object at 0x7f5d6a028b10 with _data address at 0x26d0f20, w=3 h=2␣
↪d=1 s=1 shared=0>

# You may print the _data buffer and its length, if you are very curious:
print(im2._data)
# Outputs:
# b'\x00\x00\x80?\x00\x00\x00@\x00\x00@@\x00\x00\x80@\x00\x00\xa0@\x00\x00\xc0@'

print(len(im2._data))
# Outputs:
# 24 # Remember a 3x2x1 G'MIC Image makes up 6 floats (always 32 bits or 4-bytes␣
↪long), so 6x4=24 bytes

# Just in case you wanted to read your GmicImage's data as floats at once, here is a␣
↪pythonic way
# Reminder: G'MIC stores pixel values in float32 which is the default float type's␣
↪length on most OSes
floats = struct.unpack(
    "6f", im2._data
)  # "6f" for six floats or str(len(im2._data)/4)+"f"
```

(continues on next page)

(continued from previous page)

```
print(floats)
# Outputs:
# (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

The `GmicImage` class has no method to print its pixels into console nicely as you would in `numpy` with `print(mynparray)`.

For accessing pixels, `numpy` provides a `[]` coordinates accessor `numpy.ndarray[x,y,z,....]` to read matrix cell values.

The `GmicImage` class pixel accessor is just `\(\)` parentheses call on a `GmicImage` instance. That is to say, each `GmicImage` object is callable. The signature for this accessor is `mygmicimage(x=0,y=0,z=0,s=0)`, each parameter is optional and defaults to 0. **Important:** for now, the `GmicImage` pixel accessor is read-only. You are encouraged to use I/O functions described in *Numpy support* or *PIL support* or *Scikit-Image support*.

Side note: **s** stands for spectrum, it is interchangeable with c for channel in most G'MIC literature.

Here are two examples of accessing one or all pixel values:

```
print(im2(y=1))  # reads at x=0,y=1,z=0,s=0
# Outputs:
# 4.0

for z in range(im2._depth):
    for y in range(im2._height):
        for x in range(im2._width):
            for c in range(im2._spectrum):
                print((x, y, z, c), "=", im2(x, y, z, c))

"""
Outputs:
(0, 0, 0, 0) = 1.0
(1, 0, 0, 0) = 2.0
(2, 0, 0, 0) = 3.0
(0, 1, 0, 0) = 4.0
(1, 1, 0, 0) = 5.0
(2, 1, 0, 0) = 6.0
"""
```

You may also want to view your image with your own eyes:

```
gmic.run(
    "display", images=im2
)  # Or try gmic.run("print", im2) or gmic.run("output myimage.png", im2) if your␣
↪environment has no display
"""
[gmic]-1./ Display image [0] = '[unnamed]', from point (1,1,0).
[0] = '[unnamed]':
  size = (3,2,1,1) [24 b of floats].
  data = (1,2,3;4,5,6).
  min = 1, max = 6, mean = 3.5, std = 1.87083, coords_min = (0,0,0,0), coords_max =␣
↪(2,1,0,0).
"""
```

3.2.4 The G'MIC images list (and image names)

Ooops!!! In the former section, we forgot to talk about G'MIC's intepreter images list parameter and started using it!!

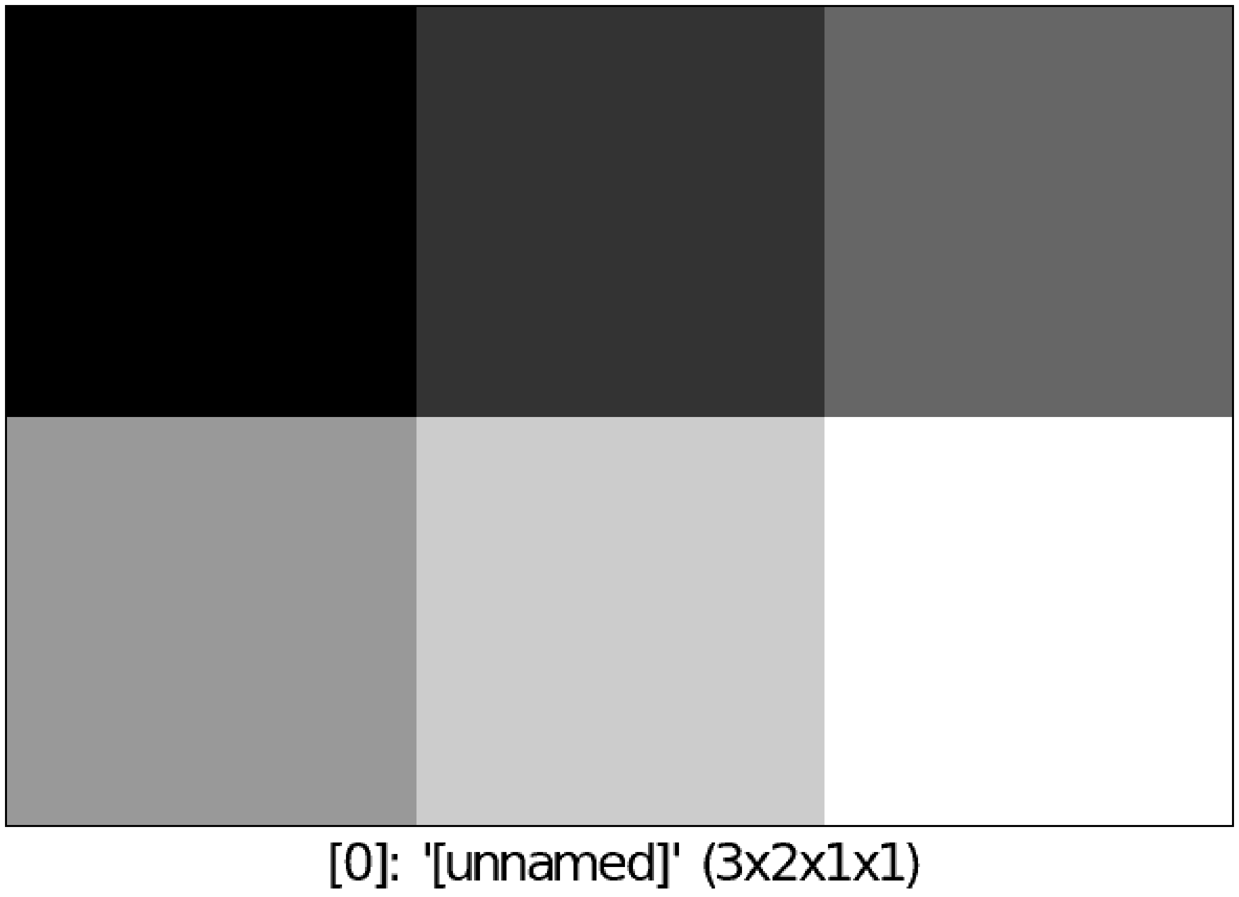


Fig. 13: tuto2_im2.png

Just for now, here is a little trick which we have done.

`gmic.run()`, `gmic.Gmic().run()` or `gmic.Gmic()` all have the same signature `(commands, images, image_names)`. Their second parameter, the `images` parameter accepts a rewritable list of `GmicImage` objects or a single read-only `GmicImage`:

- If a list is passed, it will be emptied and refilled in place…
- If a single `GmicImage`, which will be read only (no in-place modification)

Let us call the G'MIC interpreter with both single or lists or images:

```
import gmic
import struct

im2 = gmic.GmicImage(struct.pack("6f", 1, 2, 3, 4, 5, 6), 3, 2, 1)

gmic.run("display", im2)  # is a read-only operation, we can pass a single GmicImage

# But the proper way to see a change is to put your single image in a list
images_list = [im2]

gmic.run("add 1 display", images_list)  # add value 1 to each pixel then display
# Note above that the min and max value are properly shifted by 1, compared to our␣
↪first 'display' of im2, before in that same tutorial:
"""
Outputs:
gmic.run("add 1 display", images_list) # add value 1 to each pixel then display
[gmic]-1./ Display image [0] = '[unnamed]', from point (1,1,0).
[0] = '[unnamed]':
  size = (3,2,1,1) [24 b of floats].
  data = (2,3,4;5,6,7).
  min = 2, max = 7, mean = 4.5, std = 1.87083, coords_min = (0,0,0,0), coords_max =␣
↪(2,1,0,0).
"""
```

Let us continue our in-place changed image list study:

```
print(images_list)
# Not all commands have the same behaviour in terms or adding, replacing or removing␣
↪images in the input images list
# Here the 'add' command changes input images in place by default, so our images_list␣
↪is still 1-item long
# Outputs:
# [<gmic.GmicImage object at 0x7fbbe9fd3d30 with _data address at 0x1c16550, w=3 h=2␣
↪d=1 s=1 shared=0>]"""

# Let us check if our images list's single item has the same content or address as␣
↪the original im2 GmicImage... NO! And this is EXPECTED!
print(im2 == images_list[0])
# gmic-py does not change your images in place at all!
# Just empty and refill your list of GmicImages, so keep references around if you␣
↪want to keep them!
```

3.2.5 Image names

When we run the G'MIC `display` or `print` commands, you may notice in your console or with your mouse in the image display window, that our images so far are all `unnamed`.

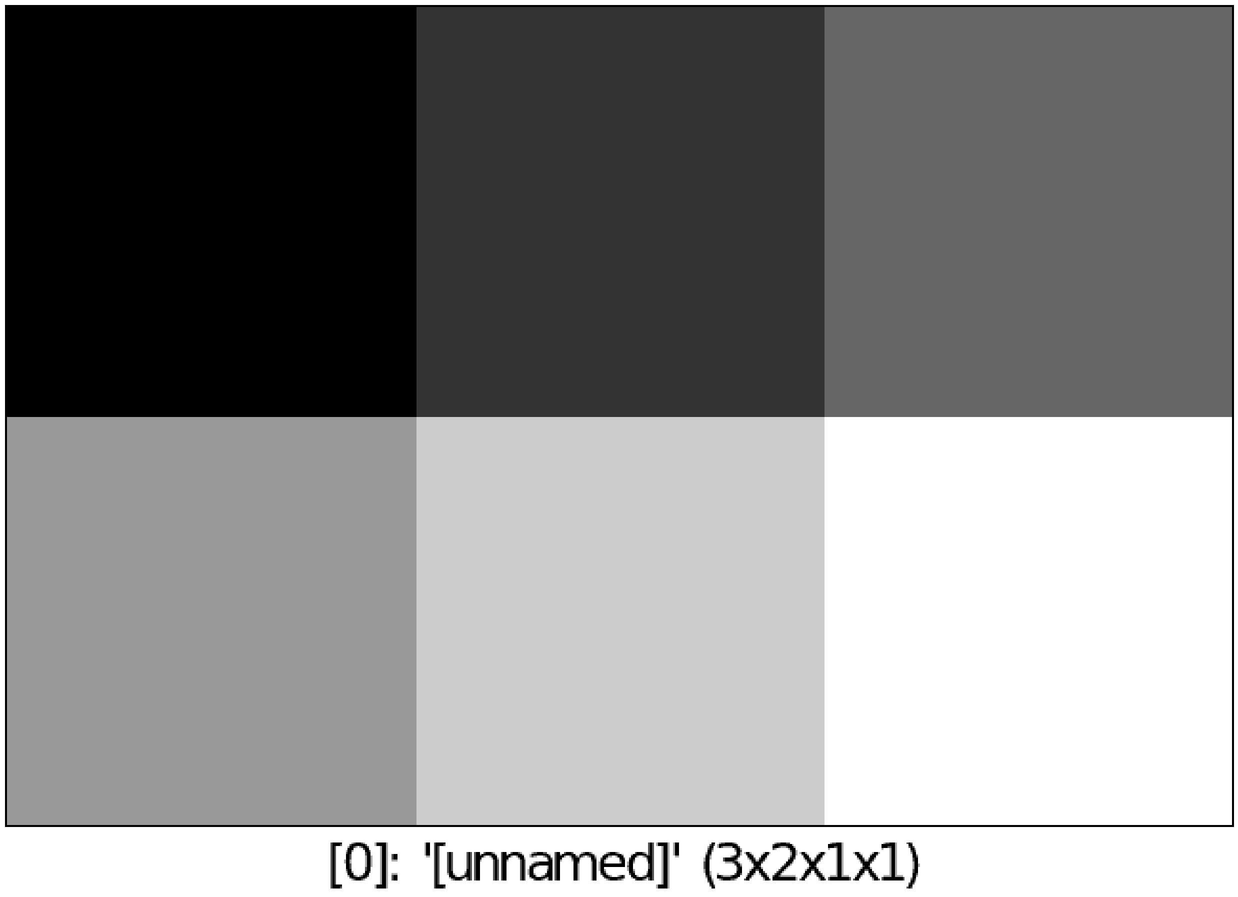


Fig. 14: tuto2_im2_add1.png

```
[gmic]-1./ Display image [0] = '[unnamed]', from point (1,1,0).
```

This is not an issue, though you can give names if you prefer, and refer to those names for indexing:

```
images_list = []
images_names = ["myapples", "myearth"]
gmic.run("sp apples sp earth print", images_list, images_names)  # No names given
```

# TODO continue + fix: https://github.com/myselfhimself/gmic-py/issues/81

3.2.6 Wrapping up - stylized fruits example

Here is an example of a stylized nature montage with some parameters injection.

To prepare this example, the following tricks have been used:

- the command line command `gmic help sp` (or `gmic.run("help sp")`), to help decide which samples would nice to pick.
- The gmic.eu Gallery page for Stylization shows sample names parameter supported by the _fx_stylize (which is a non-documented G'MIC internal command providing famous painting samples..)
  - Actually, since this is a G'MIC internal command, its code can be found here (look for _fx_stylize)
- The List of Commands page from the G'MIC online reference, especially the stylize command page

```
import gmic

g = gmic.Gmic()

# I- Stylization pass
nature_config = [
    {"sample": "apples", "style": "convergence"},
    {"sample": "fruits", "style": "redwaistcoat"},
    {"sample": "flower", "style": "lesdemoisellesdavignon"},
]

result_images = []
for conf in nature_config:
    images_list = []
    g.run(
        "sp {} _fx_stylize {} stylize[0] [1]".format(conf["sample"], conf["style"]),
        images_list,
    )
    print(images_list)
    result_images.append(images_list[0])


g.run("display", result_images)

# II- Montage pass
# Build a 3x3-bordered pixels frame around images, white, and make an automatic␣
↪montage, display it and save to file
g.run("frame 3,3,255 montage X display output mymontage.png", result_images)
```

That was it for tutorial number 2!

The tutorial's source code files can be found in gmic-py's Github's example directory.

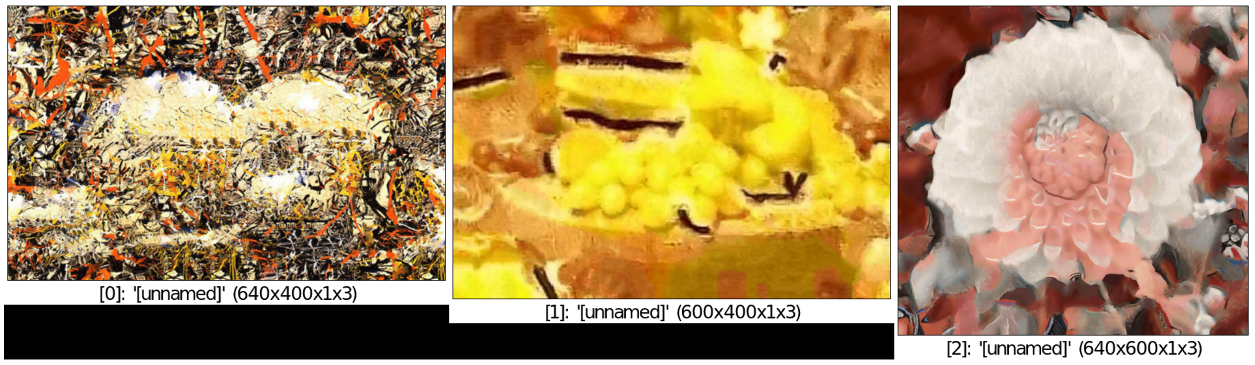

[0]: '[unnamed]' (640x400x1x3)
[1]: '[unnamed]' (600x400x1x3)
[2]: '[unnamed]' (640x600x1x3)

Fig. 15: tuto2_stylization.png

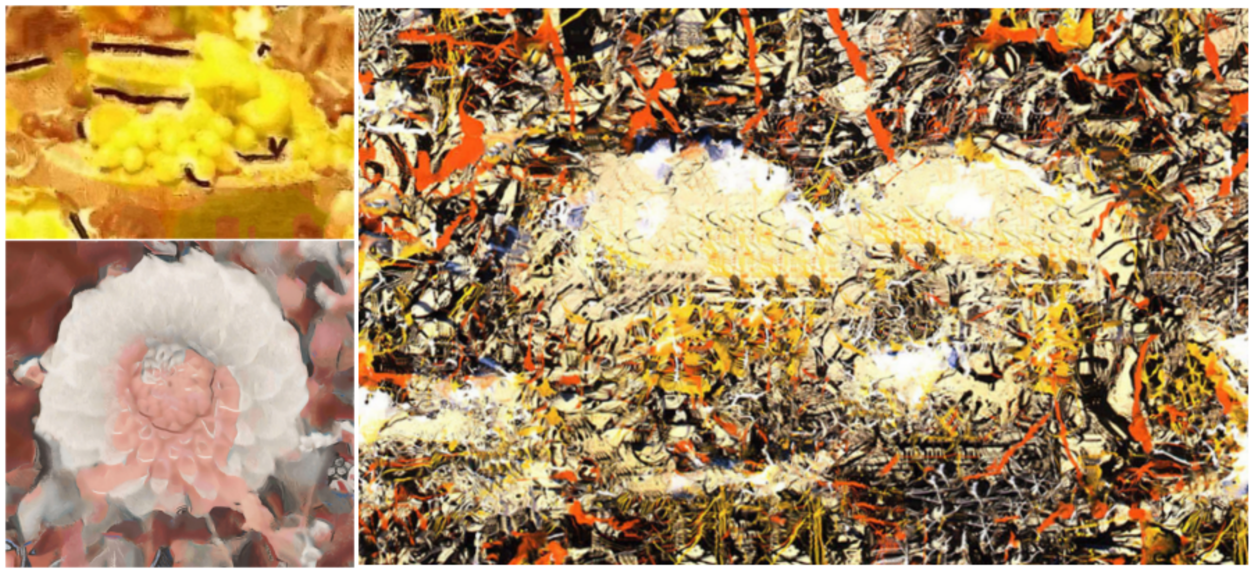


Fig. 16: tuto2_montage.png

Now you know more about reusing a G'MIC interpreter handle and calling it several times on a GmicImage list.

Congratulations!

3.3 Tutorial 3 - filtering a GIF into a flipbook

This tutorial will lead you into filtering frames of animated content using `gmic-py`:

- a moon phases GIF file to make a flip book
- a video file to apply time-controlled filters

If you are in a hurry, this tutorial's source code is on Github in the examples folder.

3.3.1 Goal: Making a moon phases flip book

The strategy behind transforming a GIF into a flip book is to:

1. split the GIF file into frames
2. modify each frame to our taste to make them fancier (stars, increasing blur)
3. make a big montage grid of the frames
4. save it to PNG and print it on quite thick a A4 paper sheet
5. cut, assemble and snap them with a binder
6. time to flip!

If you lack a printer that accepts thick pages, you may as well laser cut your PNG file.

3.3.2 Loading a GIF into a list of GmicImages

Here is a GIF of moon phases by Thomas Bresson under Creative Commons License 4.0 BY, obtained from Wikimedia:

G'MIC does not have internal support for GIF, because the GIF file format has many variants. Instead it relies on ImageMagick's `convert` executable if installed.

Let us try to open and display that GIF renamed `moonphases.gif` (download it from this link directly) into separate frames.

```
import gmic
gmic.run("moonphases.gif display")
```

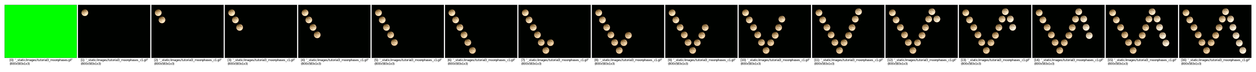


Fig. 17: _static/images/tutorial3_moonphases.gif _parse_cli_images append x

Note that we have a green frame which we do not want. Let us remove that first frame systematically.

```
import gmic
gmic.run("moonphases.gif remove[0] display")
```

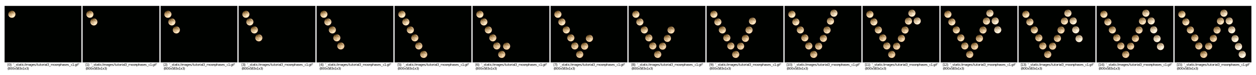


Fig. 18: _static/images/tutorial3_moonphases.gif remove[0] _parse_cli_images append x

If this GIF import leveraging `convert` does not work on your machine, let us try another way using PIL (or Pillow). (You might otherwise install `convert`).

Here we take benefit from `gmic-py`'s PIL input/output converter which uses numpy under the hood. The magic piece of code involved is: `gmic.GmicImage.from_PIL()` - a static method of the `GmicImage` class, which returns a `GmicImage` object from a `PIL.Image.Image` object .

If the `convert`-executable technique does work, but you see a green frame as the first frame, skip the PIL step and continue.

```
pip install Pillow
pip install numpy # compulsory for gmic-py PIL I/O converter to work
```

```python
import gmic
import numpy
from PIL import Image, ImageSequence

im = Image.open("moonphases.gif")

images_list = []

for frame in ImageSequence.Iterator(im):
    images_list.append(gmic.GmicImage.from_PIL(frame))

gmic.run("display", images_list)
```

Here is a synthetic adaptive version of both ways:

```python
import gmic
import shutil

g = gmic.Gmic()
images_list = []
GIF_FILENAME = 'moonphases.gif'

# If 'convert' is installed
if shutil.which('convert'):
    g.run(GIF_FILENAME, images_list)
else:
    # If convert is absent
    # PIL and numpy must be installed for this to work
    import numpy
    from PIL import Image, ImageSequence

    im = Image.open(GIF_FILENAME)

    for frame in ImageSequence.Iterator(im):
        images_list.append(gmic.GmicImage.from_PIL(frame))

g.run("remove[0]", images_list) # discard the first black or green frame
# equivalent to this Python expression which works too: images_list = images_list[1:]
g.run("display", images_list)
```

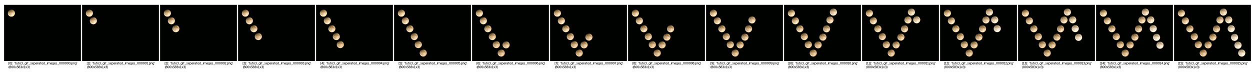


Fig. 19: input_glob tuto3_gif_separated_images_0*.png _parse_cli_images append x

Now we have the `images_list` variable filled with a GIF's frames except for the first void frame.

If you are curious to playback your `GmicImage` list, you may use the animate command with `g.run("animate", images_list)` (works on a Linux OS with a GUI).

3.3.3 Filtering individual animation frames

TIME FOR SPECIAL EFFECTS!!! (simple)

We will be adding two special effects:

1. blinking stars
2. a growing blur

Adding blinking stars

First let us add random stars on each frame, to make things blink as the Eiffel tower every nighttime hour. *G'MIC's ''stars'* command <https://gmic.eu/reference/stars.html>'_ does just that, and its default input arguments (simplified by a comma `,`) are sufficient.

Remember that G'MIC pipes commands from left to right on all the images in the current list.

Since our images list is stored in Python list, we can reuse it in a new command, without having to build a very long G'MIC expression though.

Append the following line to your script, so stars get generated differently for each frame with default settings:

```
g.run("stars ,", images_list)
```

Here are three variants for displaying the `stars`-filtered `images_list`. We have been using them since tutorial 1 more or less explicitly:

```
# basic G'MIC 'display' command
g.run("display", images_list) # The basic one.
# On Jupyter/Google Colab use 'output somefilename' instead, see known issue: https://
↪github.com/myselfhimself/gmic-py/issues/63#issuecomment-704396555

# '_document_gmic' (or '_parse_cli_images append x' from gmic>=2.9.4), for debugging
# beware this replaces your list's contents with single image!
# hence the optional backup with copy.copy() (GmicImage objects are copy()-compatible)
import copy
images_list_2 = [copy.copy(im) for im in images_list]
g.run("_parse_cli_images append x display", images_list_2)
print(len(images_list_2)) # prints '1'

# 'animate' for playing back images (will not work on Jupyter/Google Colab)
g.run("animate", images_list)
```

Now let us make things unclear with blurring!!

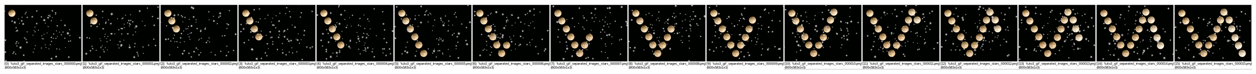


Fig. 20: input_glob tuto3_gif_separated_images_stars_0*.png _parse_cli_images append x display

Adding a blur effect

We will use the G'MIC blur command with a strength = frame index * 2.

For looping over the images and applying a different blur strength for each, we would do a Python for loop. Unfortunately `GmicImage` objects are immutable in G'MIC and `GmicImage` lists are emptied and refilled in place (but its items are not edited in place).

A - Pythonic way

Here is a Pythonic not so short way of applying a growing blur on a list of images in place.

```python
for pos, image in enumerate(images_list):
    single_image_list = [image]
    g.run("blur {strength}".format(strength=(pos*2)), single_image_list)
    images_list[pos] = single_image_list[0]
```

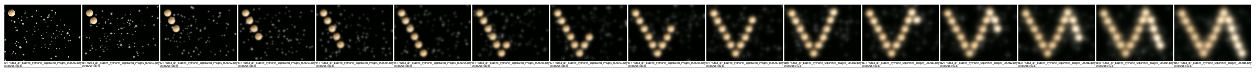


Fig. 21: input_glob tuto3_gif_blurred_pythonic_separated_images_0*.png _parse_cli_images append x display

Now, do not believe that G'MIC is slow as hell. It uses OpenMP for parallelization for more and more commands in each new release.

B - G'MIC one-liner way

In case you are satisfied with the former result, you may skip to the next section.

Out of curiosity, here is just a pure-G'MIC syntax alternative for the growing blur, which stands as a one-liner.

```python
g.run("repeat $! blur[$>] {$>*2} done animate", images_list)
```

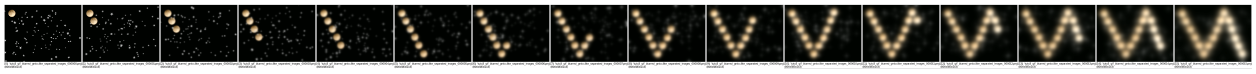


Fig. 22: input_glob tuto3_gif_blurred_gmicclike_separated_images_0*.png _parse_cli_images append x display

You have added two special effects to our animation!!! Congratulations!!!

Now is the time to flatten all our frames onto a sheet of paper :)

3.3.4 Making a montage of the frames

An A4 sheet of paper is what we typically print on in any office or house in Europe. Its dimensions are: 21 x 29.7 millimeters.

Here is a related montage-making specialists conversation on pixls.us, a community forum.

Let us skip dots-per-inch printing resolution calculation and assume that our final images will be 100 times this ratio in pixels: 2100 x 2970 px.

For our work, the following commands will be used:

- *frame (aka frame_xy) <https://gmic.eu/reference/frame_xy.html>* for adding vertical and horizontal margins to each image,
- append_tiles (instead of the more famous montage command) for combining images into a 2D grid,
- resize_ratio2d to make the result image fit into our pixel dimensions, without stretching.

Here is what each step with the aforementioned G'MIC commands looks like.

Build frames (padding or margins) :

```
# Adding a 40px horizontal margin to each frame, and 3px high
g.run("frame_xy 40,3", images_list) # frame and frame_xy commands are the same
```

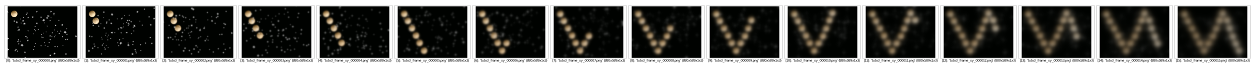


Fig. 23: input_glob tuto3_frame_xy_0*.png _parse_cli_images append x display

Build a grid (or sort of montage) :

```
# Make 4-items wide grid of all images,
# 4 is the x-count, the second y-count parameter is omitted (hence ',' comma followed␣
↪by nothing)
g.run("append_tiles 4,", images_list)
```

Resize it to fit nicely on an A4 page :

```
g.run("resize_ratio2d 2100,2970", images_list)
```

As you notice, paper space is not fully used and our image is 2100x1406 pixels big.

You may skip the next section if you are please with this result.

Actually arranging our images in the vertical direction yields use bigger images and less paper space wasting.

For this we use the G'MIC rotate command and swap the `append_tiles` columns (y) and rows (x) parameters.

Here is a one-liner that spares a bit more space.

```
g.run("frame_xy 40,3 append_tiles ,4 rotate 90 resize_ratio2d 2100,2970", images_list)
```

Congratulations !! Your montage is now ready !!!

3.3.5 Saving to PNG and printing

Using Imagemagick's `convert` and G'MIC's `output someFile.pdf` command, you may output a PDF file for our image. However, there seems to be canvas or view size output misconfigurations.

Instead, G'MIC's native PNG output will print well using any regular desktop's printing tools.

The output command is just `output yourFileName.someFileExtension`.

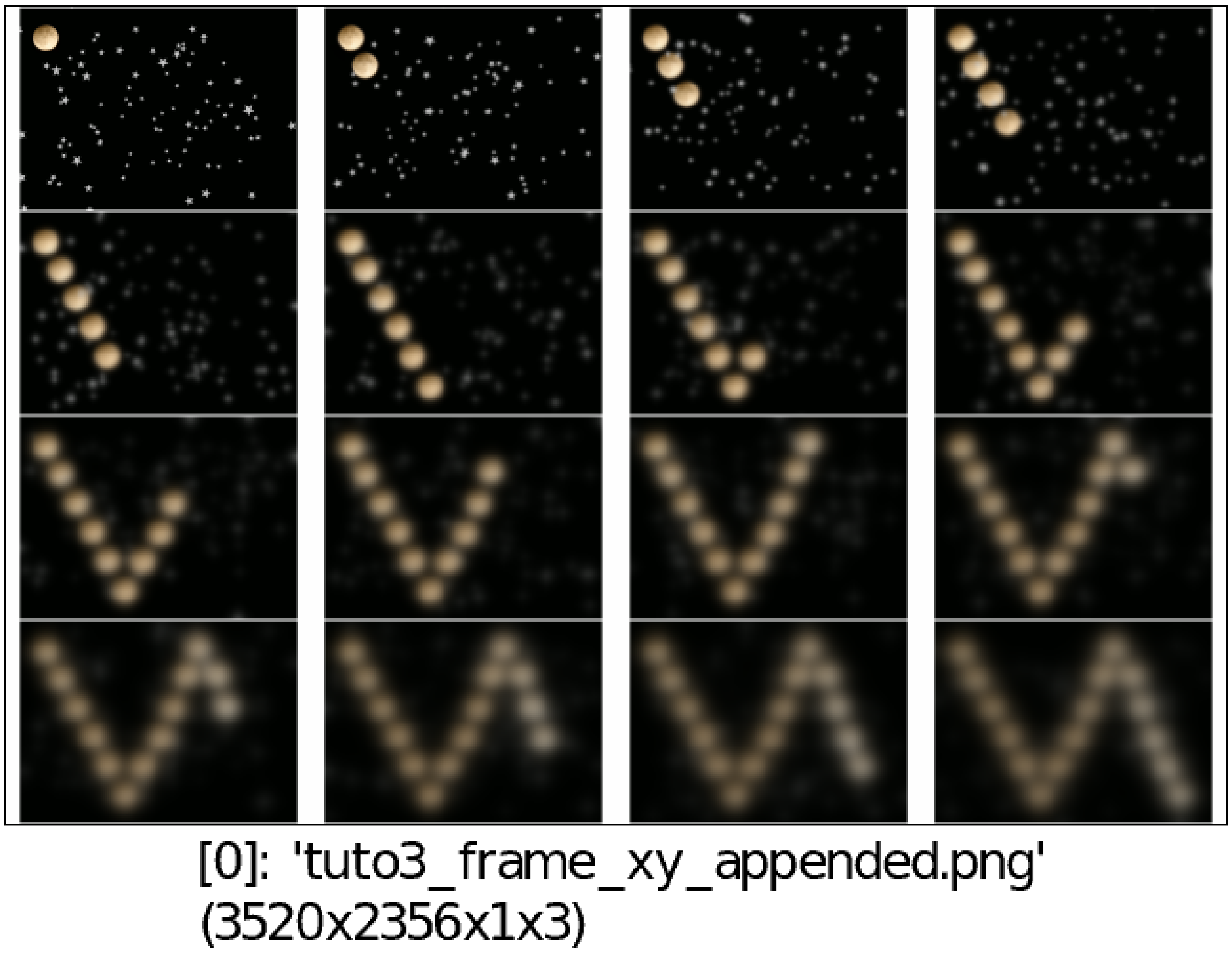
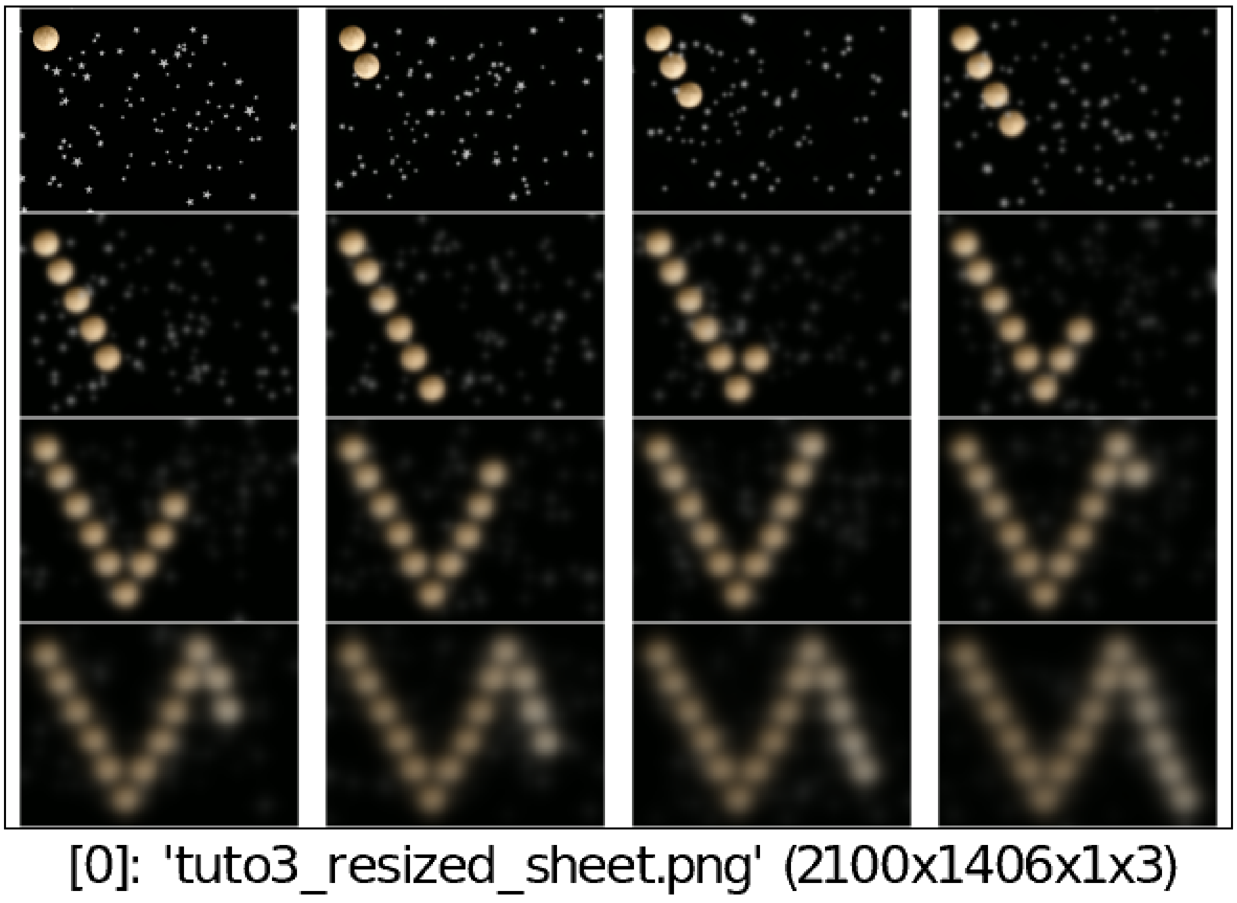
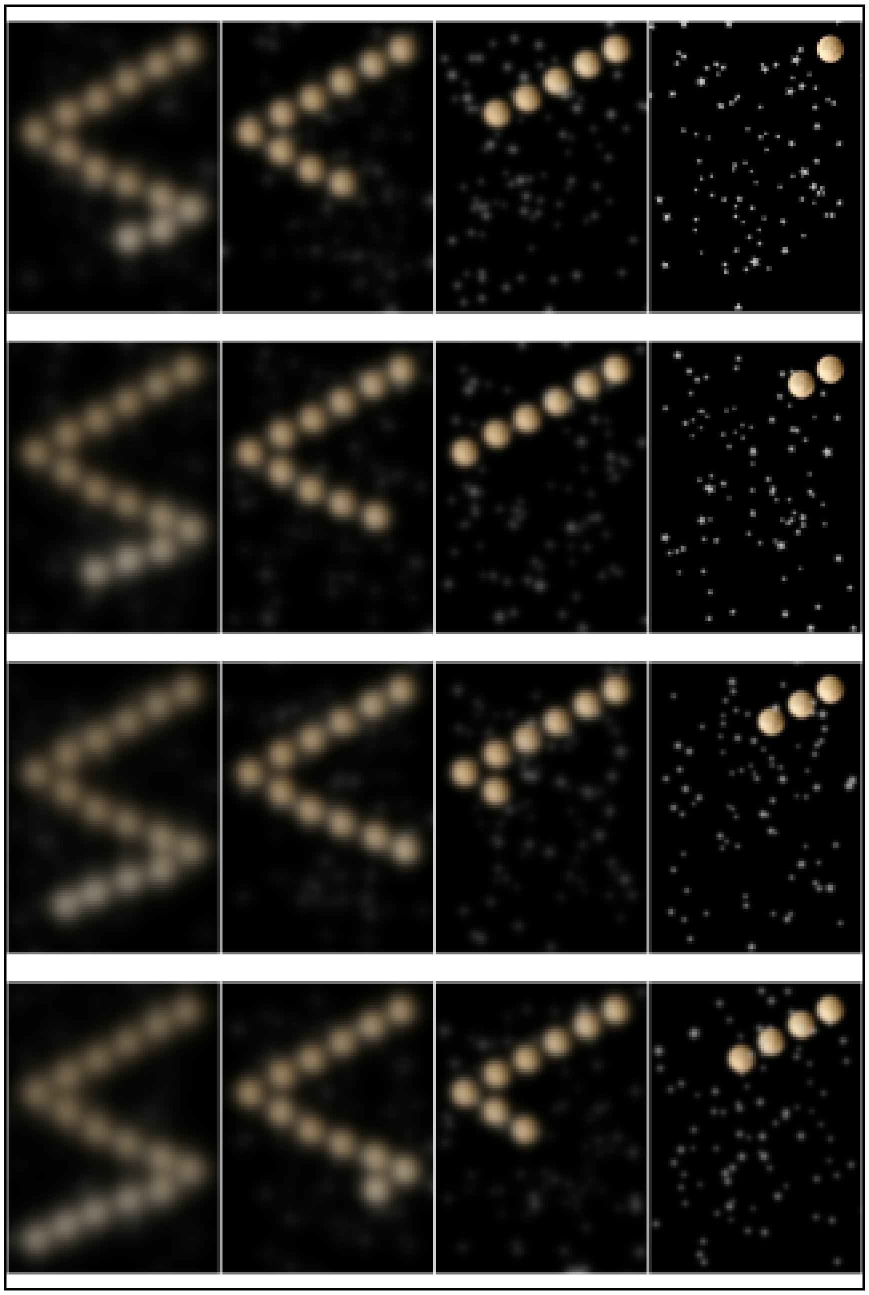

[0]: 'tuto3_frame_xy_appended.png'
(3520x2356x1x3)

Fig. 24: input_glob tuto3_frame_xy_appended.png _parse_cli_images append x display



[0]: 'tuto3_resized_sheet.png' (2100x1406x1x3)

Fig. 25: input_glob tuto3_resized_sheet.png _parse_cli_images append x display



```
[0]: 'tuto3_resized_sheet2.png'
(1988x2970x1x3)
```

Fig. 26: input_glob tuto3_resized_sheet2.png _parse_cli_images append x display

```
print(len(images_list)) # images_list has just 1 item: the montage'd GIF frames
g.run("output myGifAsA4Paper.png", images_list) # saves into the current directoy
```

Note that output works also if you have multiple images in your *images_list* (or left-side resulting images in the G'MIC expression). In that case, the filename you provide as *output* parameter will be used as a pattern (eg. *myfilename.png* -> *myfilename_0000.png*).

Three printing tips: - print on the thickest paper as you can, as long as your printer can stand it without being stuffed; - try to disable any kind of page margin in your printing setup, we do not want to squeeze down our frames; - page resizing is OK though as long as your image ratio is respected.

_static/images/tutorial3_print_screenshot.png

3.3.6 Cutting and assembling paper frames

Take your best cutter or scissors and cut each frames without mixing up their order :)

We want no white margin to appear.

Stack your paper frames and bind them on the left or right side with a binder (or your fingers hey!!).

3.3.7 Time to flip

This is how my early flip book ended up!! Not that bad!!

To improve it, I could have more frames (spanning on several printed sheets), colors (or laser engraving), thicker paper (>200g/m2) and a proper binder

In case you missed it, this tutorial's source code is on Github in the examples folder.

CHAPTER

FOUR

NUMPY SUPPORT

Numpy stands for "numeric Python" and is a very famous data processing library for scientists. Since *gmic-py* 2.9.1, you can convert a `GmicImage` from and to a `numpy.ndarray` for simpler manipulation. The *numpy.ndarray* type is used in turn by many graphics processing toolkits.

Numpy input/output support for *gmic-py* is broken down into 4 methods:

- simplified input/output:
  - *`gmic.GmicImage.from_numpy`*
  - *`gmic.GmicImage.to_numpy`*
- full-control, also used for PIL and Scikit-image support:
  - *`gmic.GmicImage.from_numpy_helper`*
  - *`gmic.GmicImage.to_numpy_helper`*

All those methods are fully documented in the *API Reference*. If you want to implement new *gmic-py* conversion methods for some library that uses *numpy.ndarray'*‘*s a lot, you might just want to import and call the* ‘**_numpy_helper* methods with your own very parameters.

4.1 Must-know

- G'MIC works in 1D, 2D, 3D, or 4D. Numpy can work from 0D (scalar) to N dimensions (>4D).
- G'MIC has the following array shapes' dimension order: `(width, height, depth, spectrum)`. The `spectrum` (or channels) dimension represents the number of values per pixel (eg. for RGB images, `spectrum=3`). Numpy is shape-agnostic.
- G'MIC works in float32 (ie. 4-bytes float pixel values). Casts from and to *numpy.ndarray* will be done for you using numpy.ndarray.astype(). This can be tuned by parameter `GmicImage().to_numpy_helper(astype=...)`.
- G'MIC can store a few billions of values per pixel (eg. not just R,G,B,A).
- G'MIC is not made for real-time image processing but is quite fast though :).
- G'MIC stores pixel values internally in a non-interleaved format, eg. `R,R,R,G,G,G,B,B,B` for `(3,1,3)` image shape.
- For now the shape is not altered between within input/output methods provided by G'MIC. To alter those, you can use either the *permute*= parameter or pre- or post-process you numpy array with a numpy transpose() call containing several axes.
- `numpy` is not a requirement for the G'MIC's Python binding to install, start and work. But is must be installed if you happen to call the *to_/from_/numpy_** methods.

```
pip install numpy
```

- Use numpy.expand_dims and numpy.atleast_2d, numpy.atleast_3d, numpy.squeeze to fix your numpy arrays's dimensions.
- Advanced note: the compile flag for enabling or disabling numpy support in gmic-py is *gmic_py_numpy* (see setup.py).

4.2 How-to

- The usual way to convert a Numpy array to G'MIC is as follows:

```
pip install numpy
pip install gmic
```

```
import gmic
import numpy.random
arr = numpy.random.rand(512,256,3)
gmic_image_from_numpy = gmic.GmicImage.from_numpy(arr)
# You might have identically called gmic.GmicImage.from_numpy_helper(arr,␣
↪deinterleave=True)
print(gmic_image_from_numpy)
gmic.run("display", gmic_image_from_numpy)
```

- The usual way to convert a G'MIC Image to Numpy is as follows:

```
pip install numpy
pip install gmic
pip install matplotlib
```

```
import gmic
import numpy
from matplotlib import pyplot as plt
gmic_images = []
gmic.run("sp apples", gmic_images) # store apples image into our list
numpy_image_from_gmic = gmic_images[0].to_numpy()
# You might have identically called gmic.GmicImage.to_numpy_helper(arr,␣
↪interleave=True)
print(numpy_image_from_gmic)
plt.imshow(numpy_image_from_gmic)
plt.show()
```

CHAPTER

FIVE

PIL SUPPORT

PIL is the Python Imaging Library and allows to load, save and modify files in many formats. Nowadays Python programmers install its Pillow fork mostly. Since gmic-py 2.9.1 you can convert a *GmicImage* from and to a *PIL.Image.Image*.

This support is limited and does not intend to cover all types of image buffer formats accepted by PIL. Because of PIL's buffer codec limitations highlighted here, an intermediate pass which will be invisible to you will leverage methods *GmicImage.to_numpy_helper* and *GmicImage.from_numpy_helper*.

We have tested only 8-bit RGB 2D files in PIL, more or fewer channels and smaller or bigger pixel values should work. Feel free to add an issue on our tracker for things you really need related to PIL or gmic-py!

G'MIC Python's PIL input/output conversion methods are simply:

- `gmic.GmicImage.from_PIL`
- `gmic.GmicImage.to_PIL`

Those are fully documented in the *API Reference*.

You are encouraged to write your own version of *to_PIL()* and *from_PIL()* in pure Python by copy-pasting the expressions listed in those API definitions documentation, and tinkering with them.

5.1 Must-know

- G'MIC's images are 3D (volumetric) non-interleaved with an almost unlimited number of 32-bit float pixel values. Their shape axes order is x,y,z,c (or width, height, depth, channels).
- PIL works mostly in 2D interleaved, assuming here only 8-bit float pixel values (because of our limited support). PIL Images shape are assumed to be y,x,c (or height, width, channels), which may equate to a squeezed 3D array down to 2D.. Shape squeezing and axes flipping is what the *to_PIL()* will do for you.
- G'MIC PIL Image input and output methods will import *numpy* on the fly for you, so that module must be installed too in addition to eg. *Pillow*.

5.2 How-to

- The usual way to convert a PIL image to G'MIC is as follows:

```
pip install Pillow
pip install gmic
```

```
import gmic
import PIL.Image
#have some myfile.png somewhere or anything that PIL can open
gmic_image_from_PIL = gmic.GmicImage.from_PIL(PIL.Image.open("myfile.png"))
print(gmic_image_from_PIL)
gmic.run("display", gmic_image_from_PIL)
```

- The usual way to convert a G'MIC Image to PIL is as follows:

```
pip install Pillow
```

```
import gmic
import PIL.Image
gmic_images = []
gmic.run("sp apples", gmic_images) # store apples image into our list
PIL_image_from_gmic = gmic_images[0].to_PIL() # to_PIL can take 3 config parameters,␣
↪see its documentation or run help(gmic.GmicImage.to_PIL)
print(PIL_image_from_gmic)
PIL_image_from_gmic.show()
```

CHAPTER SIX

SCIKIT-IMAGE SUPPORT

Scikit-image (or *skimage*) is an image processing framework tied to Scikit. Luckily Its images are of type numpy.ndarray. Since gmic-py 2.9.1 you can convert a *GmicImage* from and to a *PIL.Image.Image*.

The *skimage* support is limited for now. It relies on fine-tuned calls to the *GmicImage.from_numpy_helper* and *GmicImage.to_numpy_helper* generic methods.

G'MIC Python's Scikit-image input/output conversion methods are simply:

- `gmic.GmicImage.from_skimage`
- `gmic.GmicImage.to_skimage`

Those are fully documented in the *API Reference*.

You are encouraged to write your own version of *to_skimage()* and *from_skimage()* in pure Python by copy-pasting the expressions listed in those API definitions documentation, and tinkering with them. You can also help improve the converters upstream with suggestions or patches on the project repository.

6.1 Must-know

- G'MIC's images are 3D (volumetric) non-interleaved with an almost unlimited number of 32-bit float pixel values. Their shape axes order is x,y,z,c (or width, height, depth, channels).
- Scikit images are the same, with pixel-type agnosticity and different shape: z,y,x,c (depth or layers, height, width, channels (or spectrum)).

6.2 How-to

- The usual way to convert a Scikit image to G'MIC is as follows:

```
pip install scikit-image
pip install gmic
```

```python
import gmic
import skimage
astronaut = skimage.data.astronaut
gmic_image_from_skimage = gmic.GmicImage.from_skimage(astronaut)
print(gmic_image_from_skimage)
gmic.run("display", gmic_image_from_skimage)
```

- The usual way to convert a G'MIC Image to PIL is as follows:

```
pip install scikit-image
pip install gmic
```

```
import gmic
import skimage
from skimage.viewer import ImageViewer
gmic_images = []
gmic.run("sp apples", gmic_images) # store apples image into our list
skimage_from_gmic = gmic_images[0].to_skimage() # to_PIL can take 3 config parameters,
↪ see its documentation or run help(gmic.GmicImage.to_PIL)
print(skimage_from_gmic)
viewer = ImageViewer(skimage_from_gmic) # you might want to call the image's .
↪squeeze() method first to have it 2D
viewer.show()
```

CHAPTER

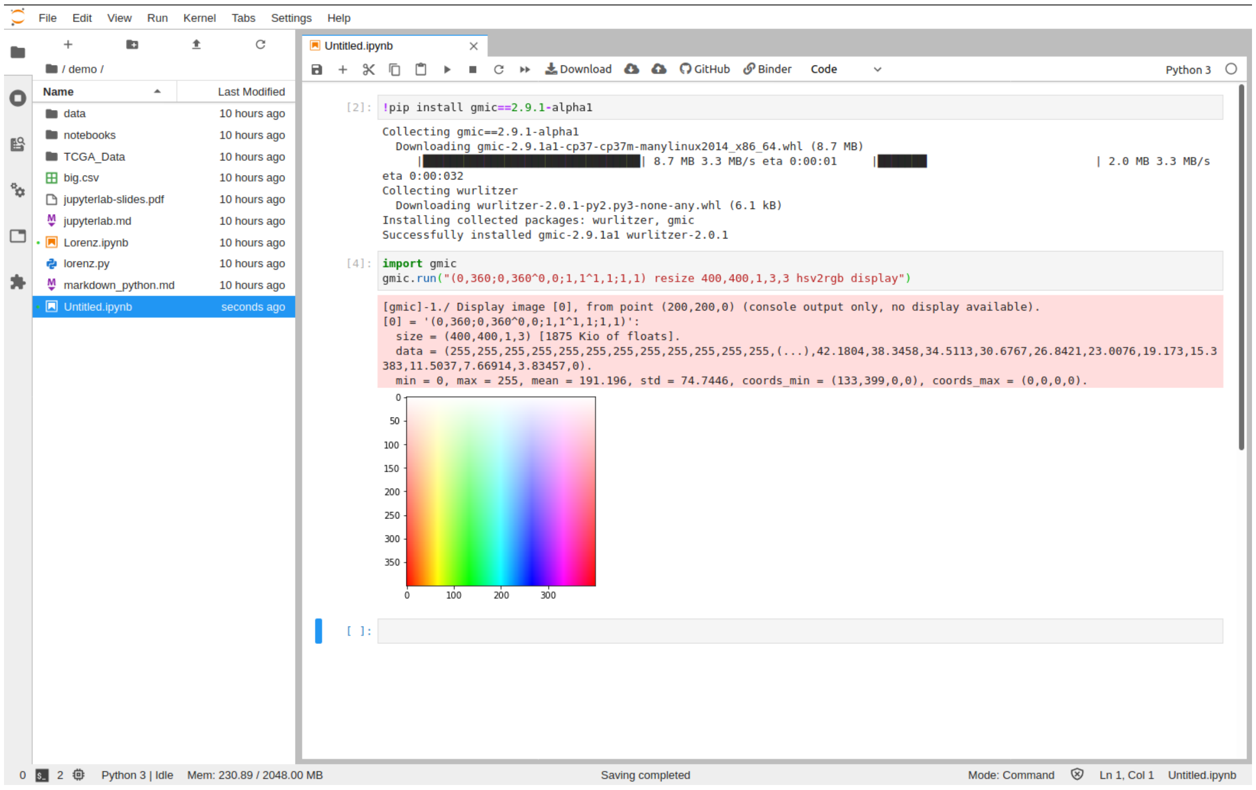
SEVEN

JUPYTER/IPYTHON/GCOLAB SUPPORT

Since version 2.9.1, *gmic-py* has been improved so that you can use *gmic* as a daily tool from any the following IPython-based shells and possibly more:

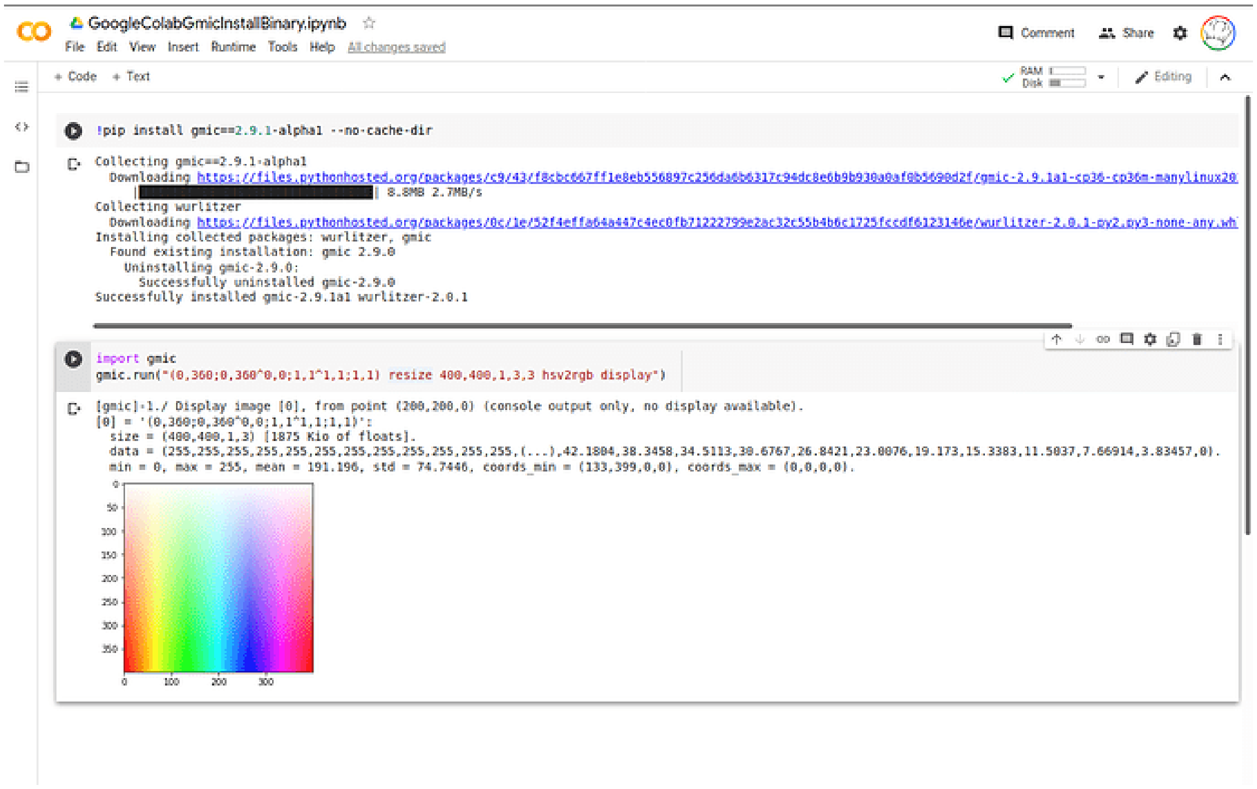
- IPython is an open-source framework improving Python shell experience in consoles, web browsers and desktop user interfaces.
- Jupyter is a sub-project of the IPython-based initiative providing amongst others *JupyterLab* (aka *Jupyter Notebooks*), a web-browser application for editing scientifical Python scripts in runnable sub-steps.
- Jupyter QtConsole is an implementation of the JupyterLab as as desktop application using the Qt GUI framework.
- Google Colab is a Google alternative to *JupyterLab*, also leveraging the IPython technology.

Example screenshots:



```
[2]: !pip install gmic==2.9.1-alpha1

     Collecting gmic==2.9.1-alpha1
       Downloading gmic-2.9.1a1-cp37-cp37m-manylinux2014_x86_64.whl (8.7 MB)
          |████████████████████████████████| 8.7 MB 3.3 MB/s eta 0:00:01     |███████                         | 2.0 MB 3.3 MB/s
     eta 0:00:032
     Collecting wurlitzer
       Downloading wurlitzer-2.0.1-py2.py3-none-any.whl (6.1 kB)
     Installing collected packages: wurlitzer, gmic
     Successfully installed gmic-2.9.1a1 wurlitzer-2.0.1

[4]: import gmic
     gmic.run("(0,360;0,360^0,0;1,1^1,1;1,1) resize 400,400,1,3,3 hsv2rgb display")

     [gmic]-1./ Display image [0], from point (200,200,0) (console output only, no display available).
     [0] = '(0,360;0,360^0,0;1,1^1,1;1,1)':
       size = (400,400,1,3) [1875 Kio of floats].
       data = (255,255,255,255,255,255,255,255,255,255,255,255,(...),42.1804,38.3458,34.5113,30.6767,26.8421,23.0076,19.173,15.3
     383,11.5037,7.66914,3.83457,0).
       min = 0, max = 255, mean = 191.196, std = 74.7446, coords_min = (133,399,0,0), coords_max = (0,0,0,0).

[ ]:
```

Fig. 1: _static/images/jupyter_example.png display


GoogleColabGmicInstallBinary.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Editing

```
!pip install gmic==2.9.1-alpha1 --no-cache-dir
```

```
Collecting gmic==2.9.1-alpha1
  Downloading https://files.pythonhosted.org/packages/c9/43/f8cbc667ff1e8eb556897c256da6b6317c94dc8e6b9b930a0af0b5690d2f/gmic-2.9.1a1-cp36-cp36m-manylinux20
     |████████████████████████████████| 8.8MB 2.7MB/s
Collecting wurlitzer
  Downloading https://files.pythonhosted.org/packages/0c/1e/52f4effa64a447c4ec0fb71222799e2ac32c55b4b6c1725fccdf6123146e/wurlitzer-2.0.1-py2.py3-none-any.wh
Installing collected packages: wurlitzer, gmic
  Found existing installation: gmic 2.9.0
    Uninstalling gmic-2.9.0:
      Successfully uninstalled gmic-2.9.0
Successfully installed gmic-2.9.1a1 wurlitzer-2.0.1
```

```
import gmic
gmic.run("(0,360;0,360^0,0;1,1^1,1;1,1) resize 400,400,1,3,3 hsv2rgb display")
```

```
[gmic]-1./ Display image [0], from point (200,200,0) (console output only, no display available).
[0] = '(0,360;0,360^0,0;1,1^1,1;1,1)':
  size = (400,400,1,3) [1875 Kio of floats].
  data = (255,255,255,255,255,255,255,255,255,255,255,255,(...),42.1804,38.3458,34.5113,30.6767,26.8421,23.0076,19.173,15.3383,11.5037,7.66914,3.83457,0).
  min = 0, max = 255, mean = 191.196, std = 74.7446, coords_min = (133,399,0,0), coords_max = (0,0,0,0).
```

Fig. 2: _static/images/gcolab_example.png display

7.1 How-to

7.1.1 Installing

Unless you are using a local *JupyterLab* (ie. web interface) or *Jupyter QtConsole* from your own virtual-environment where *gmic-py* can be pre-installed using *pip install gmic*, here is an example command to install *gmic-py*:

```
!pip install gmic # or gmic==2.9.1-alpha6
```

```
Collecting gmic==2.9.1-alpha1
  Downloading https://files.pythonhosted.org/packages/c9/43/
↪f8cbc667ff1e8eb556897c256da6b6317c94dc8e6b9b930a0af0b5690d2f/gmic-2.9.1a1-cp36-
↪cp36m-manylinux2014_x86_64.whl (8.7MB)
     || 8.8MB 2.8MB/s
Collecting wurlitzer
  Downloading https://files.pythonhosted.org/packages/0c/1e/
↪52f4effa64a447c4ec0fb71222799e2ac32c55b4b6c1725fccdf6123146e/wurlitzer-2.0.1-py2.
↪py3-none-any.whl
Installing collected packages: wurlitzer, gmic
Successfully installed gmic-2.9.1a1 wurlitzer-2.0.1
```

7.1.2 Using

The following examples have nothing special compared to a regular *gmic-py* usage and were tested on Google Colab, a web type of Jupyter/IPython notepad.

```
import gmic
# You might optionnally see a message as follows:
# gmic-py: wurlitzer found (for G'MIC stdout/stderr redirection) and enabled␣
↪automatically through IPython '%load_ext wurlitzer'.
images = []
gmic.run("300,400,1,3 fx_camouflage 9,12,100,30,46,33,75,90,65,179,189,117,255,246,
↪158 display", images)
# A matplotlib or other type of image view should pop inline
# Expected text output:
# [gmic]-1./ Display image [0], from point (150,200,0) (console output only, no␣
↪display available).
# [0] = '[unnamed]':
#   size = (300,400,1,3) [1406 Kio of floats].
#   data = (95.8,95.8,95.8,95.8,95.8,95.8,95.8,95.8,95.8,95.8,95.8,95.8,(...),75.4,75.
↪4,75.4,75.4,75.4,75.4,75.4,75.4,75.4,75.4,75.4,75.4).
#   min = 30, max = 255, mean = 111.497, std = 51.1507, coords_min = (125,0,0,0),␣
↪coords_max = (167,18,0,0).
```

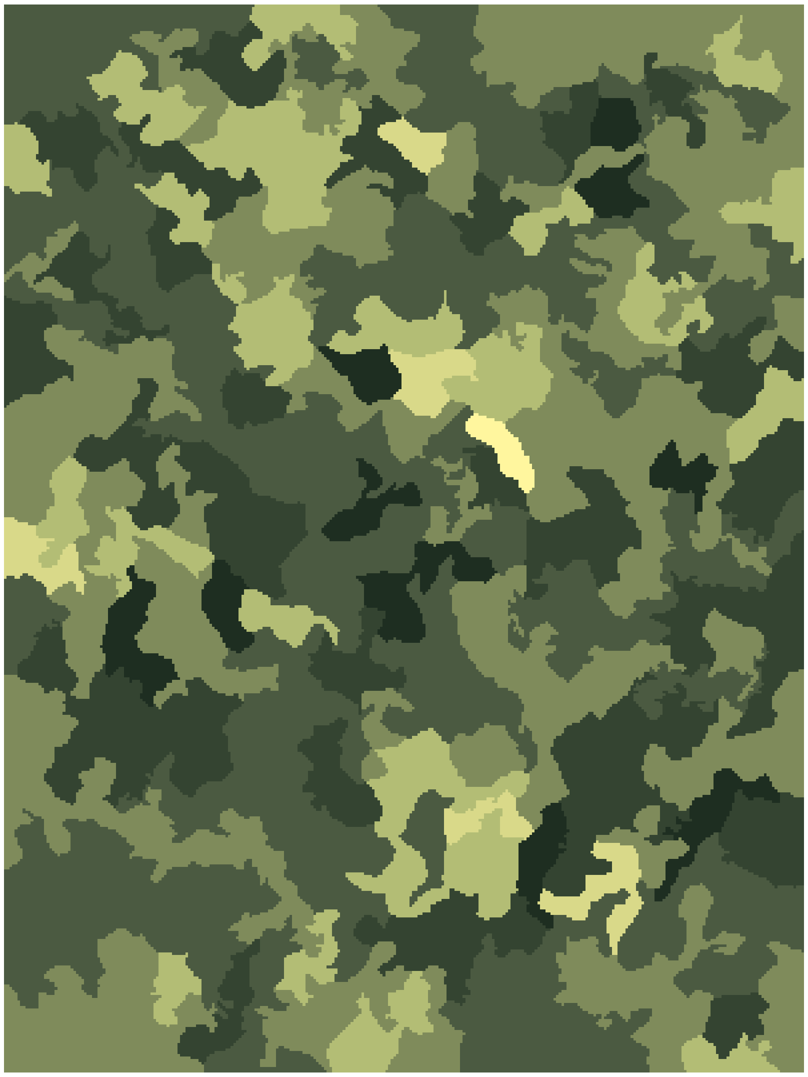


Fig. 3: 300,400,1,3 fx_camouflage 9,12,100,30,46,33,75,90,65,179,189,117,255,246,158

```
import gmic
images = []
# Note that the "sample" command relies exclusively on gmic.eu online images␣
↪reachability
# So the following might not work if you are using a remote JupyterLab or Google␣
↪Colab shell
# depending on your platforms' proxy parameters
gmic.run("sp apples print", images)
# No image view should pop (the print command only prints textual information?
# Outputs:
# [gmic]-1./ Print image [0] = 'apples'.
# [0] = 'apples':
#   size = (640,400,1,3) [3000 Kio of floats].
#   data = (20,22,20,20,20,22,22,22,22,22,22,20,(...),1,1,1,1,1,1,1,1,1,1,1,1).
#   min = 1, max = 250, mean = 58.5602, std = 59.8916, coords_min = (317,306,0,1),␣
↪coords_max = (430,135,0,0).
print(images)
# Outputs:
# [<gmic.GmicImage object at 0x7f23fc2f6d30 with _data address at 0x7f23fae17010,␣
↪w=640 h=400 d=1 s=3 shared=0>]
```

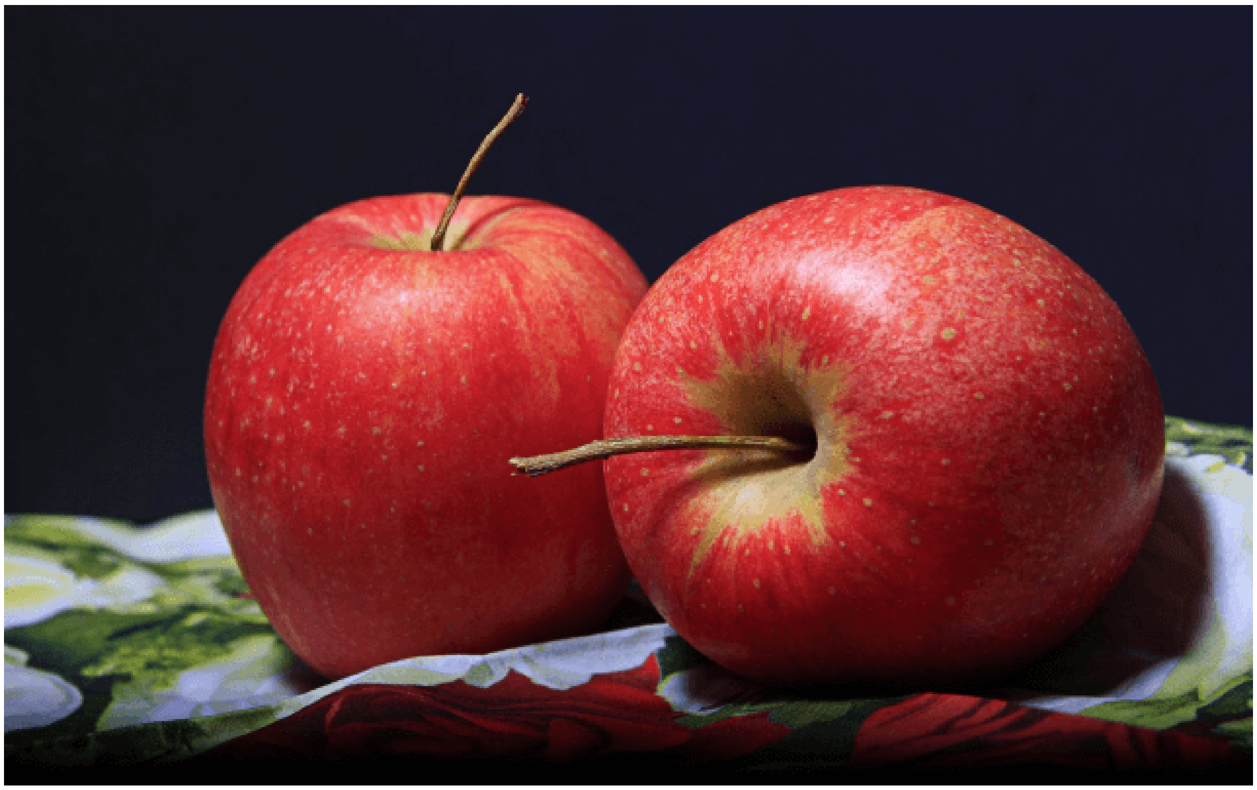


Fig. 4: sp apples

7.2 Implementation details

The core tricks of *gmic-py*'s support for IPython-based web-based graphical shells are:

1. for text display (eg. for the display and print commands: G'MIC standard output redirection towards the IPython

user output. For this the Python wurlitzer cross-platform module has been used and added as a *gmic-py* permanent dependency, leveraging its IPython enabling macro, if an IPython shell is detected.

2. for non-popping G'MIC image display window: transparent replacement of G'MIC display command calls into output calls as *PNG* format into your (or the host) computer's temporary directory, followed by IPython or Matplotlib display calls. For this, a pure C/Python simple adaptor code has been added.

For desktop UI implementations such as Jupyter QtConsole, since your operating systems' *DISPLAY* environment variable is set, above point 1. is still relevant, but the G'MIC native display will probably pop up instead of the *PNG* trick.

The Jupyter support in *gmic-py* can be disabled before module compilation by unsetting the *gmic_py_jupyter_ipython_display* compiler variable. See setup.py.

CHAPTER

EIGHT

DEVELOPING, COMPILING, TESTING, RELEASING

gmic-py is a C/Python binding of C++ which must be compiled or pre-compiled for any target machines in order to work. Third-party frameworks for writing bindings such as Pybind11 or Pyrex have not been used because of the binding's simplicity.

Any Linux / Mac OS / Unix operating system with Python >= 3.6 (possibly less) should be able to compile the binding without any non-standard Python tooling. 32bit architectures are not supported since Q3 2020, in favor of pure x86_64, but the project used to compile well against the former.

8.1 TL;DR building *gmic-py* on Linux

You can build G'MIC by forcing pip install to build it from a source tarball:

```
pip install gmic --compile
```

You can build G'MIC from a Git repository clone. For this run the following lines one by one, deciding on your options:

```
git clone https://github.com/myselfhimself/gmic-py --depth=1

# For Centos / Redhat / Fedora..:
yum install fftw-devel libpng-devel zlib-devel libgomp libtiff-devel libjpeg-devel␣
↪wget

# For Ubuntu
sudo apt-get install libfftw3-dev libcurl4-openssl-dev libpng-dev liblz-dev libgomp1␣
↪libtiff-dev libjpeg-dev wget

# Download libgmic's preassembled source archive (equates to 2 git clone commands + 2-
↪3 make commands..)
bash build_tools.bash 1_clean_and_regrab_gmic_src

# For building linux-repaired wheels, using the manylinux docker images, run:
bash build_tools.bash 33_build_manylinux # to focus on one version, add eg. 'cp36' as␣
↪a final parameter
ls wheelhouse/ # here you have .whl files

# For building just a non-repaired debug .so file the current machine:
bash build_tools.bash 2b_compile_debug
ls build/lib* # cd into any directory with a .so file and run python3 in it, to be␣
↪able to 'import gmic'

# Same but optmimized non-repaired .so file
```

(continues on next page)

(continued from previous page)

```
bash build_tools.bash 2_compile
ls build/lib*
```

8.2 `gmic-py` development to release lifecycle (overview)

In very short, the G'MIC Python binding per-version lifecycle is as follows: #. grab libgmic's C++ targetted version #. tune binding code and pytest cases #. compile and test locally #. git push with a tag to trigger optimized releases building and sending to G'MIC's pypi.org project

Go to `` ` ``gmic-py development to release lifecycle (detailed)'_ for more details on the right tooling to use for each step.

Note: Steps 1-3 correspond to the `bash build_tools.bash 00_all_steps` command.

8.3 Github Actions Continuous integration workflows

Looking that the Github project's Action's tab or the .github/workflows files, you will notice the following discting workflows:

- Linux debug (the fastest to yield a result)
- MacOS optimized
- Manylinux optimized
- Manylinux optimized, on Git tag push optimized with release (to pypi.org)
- MacOS optimized on Git tag push with release (to pypi.org)

All of them leverage `build_tools.bash` and show the needed package for each OS.

8.4 `build_tools.bash` - a developer's Swiss army knife

Located in the Git repository's root, build_tools.bash is used for developing, building and releasing `gmic-py`.

Before running *build_tools.bash*, you should install the developer requirements first:

```
pip install -r dev-requirements.txt
```

Then, a running the former script without parameters or with `--help` shows the targeted G'MIC version and the available commands.

8.4.1 Centralized version for development and continuous-integration-based releasing

The targeted G'MIC version is the available version of G'MIC (according to its source archives and pre-releases) for which we are developing a binding and preparing a release. It is stored in the `VERSION` file (add no trailing character after the version number there!) for use by build_tools.bash, setup.py the continuous integration scripts.

8.4.2 Calling build_tools.bash

To call any command just append its name as a first parameter:

```
$ bash build_tools.bash <the command name>
$ # For example:
$ bash build_tools.bash1_clean_and_regrab_gmic_src # Will grab the libgmic C++ code
```

8.4.3 Rapid sub-commands overview and explanations

Exhaustive commands documentation will not be covered hereafter. In order to understand them, you should look at their implementations within the bash script and their use within the .github/worfklows/ Github Action continuous integration recipes. In it, one function equates to one command.

- `00_all_steps`: Use this if you are a beginner with `build_tools.bash` and have time on a Linux machine with a Python virtual environment, it will grab G'MIC's C++ source, compile, test and bundle it without making any release. More experienced developer in the project will likely run single steps only. This can also be run from a Docker image, *although the related Dockerfile now only survives in Git history <https://github.com/myselfhimself/gmic-py/blob/fc12cb74f4b02fbfd83e9e9fba44ba7a4cee0d93/Dockerfile>*_ because it is used very rarely.
- `1_clean_and_regrab_gmic_src`: download libgmic's C++ code into the src/ directory (which is emptied beforehand)
- `11_send_to_pypi`: send built wheels (`.whl`) to pypi.org using twine
- `2_compile`: compile with optimization (long). On Linux a `.so` file is generated in the build/ directory.
- `2b_compile_debug`: compile without optimization (fast) and with debug symbols.
- `20_reformat_all`: reformat both Python and C code (note this is not done after compile time in `manylinux` to avoid crashes). You usually run this by hand before doing a Git commit.
- `21_check_c_style`: using clang-format.
- `22_reformat_c_style`: using `clang-format`.
- `23i_install_black_python_formatter`: installed a locked version of the black Python formatter and checker.
- `23_check_python_style`: using `black`.
- `24_reformat_python_style`: using `black`.
- `33_build_manylinux`: build `gmic-py` with optimized compiling using the PEP 571 standard for old Linux distributions. This technique nicknamed manylinux ships with a Docker image we use on Github Actions. Rarely run locally because it is super long, but this is safe as it is dockerized. Check for your built wheels in the *wheels/* directory.
- `3_test_compiled_so`: runs pytest cases from `tests/` onto your `build/lib*` shared `gmic-py` library.
- `3b_test_compiled_so_no_numpy`: similar by omitting the Numpy-support test suite.
- `31_test_compiled_so_filters_io`: very long experimental test suite with G'MIC `gmic` cli command vs `gmic-py` module output images result comparison.
- `4_build_wheel`: build a .whl wheel without embedding external shared libraries (ie. doing a "repair" step as needed on Linuxes, but not on MacOS or Windows). When run, head over to the *build/dist** directory.
- `5_test_wheel`: runs pytest cases over the last built wheel.

8.5 Recommended compilers

For proper OpenMP support - which is highly recommended, our build bots use GCC for Linux (CLang should work) and CLang version 6 (not newer) on MacOS.

For the upcoming Windows support, MSYS2 - mimicking the UNIX standards - will be the envisioned environment, instead of MSVC. The former compiler works already best with G'MIC (C++).

8.6 Library requirements

`gmic-py` embeds libgmic C++ library and has the same library needs as the latter. Namely zlib and libpng, optionally libfftw3, libjpeg, libtiff, OpenMP. `gmic-py`'s setup.py file shows the use of the Unix-compatible pkgconfig module, for available libraries detection and toggling in order to run a smooth compilation with you having to tune compile flags at all.

Note that our releases are all built against: zlib, libpng, libopenmp, libtiff, libjpeg, similarly to libgmic releases. Libgmic IS embedded inside the `gmic-py` binding.

8.7 Optimized vs. debugging

For testing and daily development, `gmic-py` can be compiled faster with no optimization and with debug symbols attached. This is down through a hackish `--debug` flag. This is what is run through

From `setup.py`:

```
debug_enabled = "--debug" in sys.argv
```

For releases, an optimized build is generated, just by omitting the `--debug` flag.

For debugging segfaults or other situations, you can run *gdb python* and explore with the gdb command line. You can also use CLion (or any C++ editor), load the C source and Python script of your own using the *gmic* module, run your Python script in Debug mode or with some blocking *input()* or other pure-python breakpoing for example, and attach with your C++ IDE to the latest Python process run. Here is a similar very barebone way of debugging with IPython and lldb (or gdb).

8.8 On the fly compiling with pip

You can compile automatically through a `pip` which will run the `setup.py` compiling steps for you, it will download `gmic-py`'s source from its most stable origin: pypi.org.

```
pip install --no-binary gmic
```

8.9 Compiling from a git clone

Compiling locally from a Git clone is usually done with GCC/CLang and gets inspiration from libgmic's own Makefile. There are no special tricks, but Python tools are used best instead of direct compiler calling.

```
setup.py build # will need a pip install pkgconfig first
```

Which is done by `build_tools.bash 2_compile` or `2b_compile_debug` variant as well.

Though you will libgmic's source first. See the next section instead for doing first things first.

8.10 `gmic-py` development to release lifecycle (detailed)

1. once for all, install developer's requirements in a project own virtual environment:

```
pip install -r dev-requirements.txt
```

2. change the targetted G'MIC version number (we follow libgmic's versioning) in VERSION. `build_tools.bash`, `setup.py` and the Github Actions workflow files will all rely on this central piece of data!

```
echo "2.9.1" > VERSION
```

Note: this version can be overriden on a per-command basis for `build_tools.bash` by setting the `GMIC_VERSION` environment variable. Read `build_tools.bash` code for details.

3. grab the related libgmic C++ source

```
bash build_tools.bash 1_clean_and_regrab_gmic_src
```

4. edit `gmicpy.cpp` `gmicpy.h` `setup.py` the pytest `tests/`
5. edit the documentation in `docs/` (it gets deployed to readthedocs.io on each Git push)
6. rebuild documentation for previewing:

```
pip install sphinx # one time only
cd docs/; make html
```

7. compile in debug mode

```
bash build_tools.bash 2b_compile_debug
```

8. run few or all unit tests locally

```
bash build_tools.bash 3_test_compiled_so # for all tests
bash build_tools.bash 3b_test_compiled_so_no_numpy # for all tests, except numpy ones
bash build_tools.bash 3b_test_compiled_so_no_numpy openmp # all tests the name of␣
↪which matches the *openmp* wildcard
```

9. hand test interactively (outside any Python virtual environment, or using an environment with *gmic* uninstalled)

```
cd build/lib.linux-x86_64-3.6/
ls # shows gmic.cpython-36m-x86_64-linux-gnu.so
python3
# import gmic
# gmic.run("sp earth") # etc
```

10. check linked shared libraries

```
cd build/lib.linux-x86_64-3.6/
ldd gmic.cpython-36m-x86_64-linux-gnu.so
```

11. Git push without any tag to trigger Github Actions for Mac OS and Linux debug and optimized builds, as well as readthedocs.io documentation building

```
git push # (origin master) or any other Github branch
```

12. set a Git tag and Git push to trigger the former Github Actions + identical ones optimized with pypi.org release wheels upload

```
git tag -a v2.9.1 # In this project, the tag must start with v for releasing
git push # origin master or any other Github branch
```

13. look at pypi.org's gmic module released contents
14. test online releases by hand (in a Python environment without gmic installed)

```
pip install gmic # or gmic==2.9.1 in our case
python3
# import gmic
# gmic.run("sp earth") # etc
py.test tests/
```

CHAPTER

NINE

API REFERENCE

G'MIC image processing library Python binary module.

Use `gmic.run` or `gmic.Gmic` to run G'MIC commands inside the G'MIC C++ interpreter, manipulate `gmic.GmicImage` which has `numpy`/`PIL` input/output support, assemble lists of `gmic.GmicImage` items inside read-writeable pure-Python *list* objects.

class gmic.**Gmic**

Bases: `object`

run (*command*, *images=None*, *image_names=None*)

Run G'MIC interpreter following a G'MIC language command(s) string, on 0 or more namable `GmicImage` items.

Note (single-image short-hand calling): if `images` is a `GmicImage`, then `image_names` must be either a `str` or be omitted.

Example

Here is a long example describing several use cases:

```
import gmic
import struct
import random
instance1 = gmic.Gmic('echo_stdout \'instantiation and run all in one\')
instance2 = gmic.Gmic()
instance2.run('echo_stdout \'hello world\'') # G'MIC command without images␣
↪parameter
a = gmic.GmicImage(struct.pack(*('256f',) + tuple([random.random() for a in␣
↪range(256)])), 16, 16) # Build 16x16 greyscale image
instance2.run('blur 12,0,1 resize 50%,50%', a) # Blur then resize the image
a._width == a._height == 8 # The image is half smaller
instance2.run('display', a) # If you have X11 enabled (linux only), show the␣
↪image in a window
image_names = ['img_' + str(i) for i in range(10)] # You can also name your␣
↪images if you have several (optional)
images = [gmic.GmicImage(struct.pack(*((str(w*h)+'f',) + (i*2.0,)*w*h)), w,␣
↪h) for i in range(10)] # Prepare a list of image
instance1.run('add 1 print', images, image_names) # And pipe those into the␣
↪interpreter
instance1.run('blur 10,0,1 print', images[0], 'my_pic_name') # Short-hand 1-
↪image calling style
```

Parameters

- **command** (`str`) – An image-processing command in the G'MIC language
- **images** (`Optional[Union[List[gmic.GmicImage], gmic.GmicImage]]`) – A list of `GmicImage` items that G'MIC will edit in place, or a single `gmic.GmicImage` which will used for input only. Defaults to None. Put a list variable here, not a plain `[]`. If you pass a list, it can be empty if you intend to fill or complement it using your G'MIC command.
- **image_names** (`Optional[List<str>]`) – A list of names for the images, defaults to None. In-place editing by G'MIC can happen, you might want to pass your list as a variable instead.

Returns Returns `None` or raises a `GmicException`.

Return type None

Raises ***`GmicException`*** – This translates' G'MIC C++ same-named exception. Look at the exception message for details.

exception gmic.**GmicException**

Bases: `Exception`

Only exception class of the Gmic module.

This wraps G'MIC's C++ gmic_exception. Refer to the exception message itself.

class gmic.**GmicImage**(*data=None*, *width=1*, *height=1*, *depth=1*, *spectrum=1*, *shared=False*)

Bases: `object`

Simplified mapping of the C++ `gmic_image` type. Stores a binary buffer of data, a height, width, depth, spectrum.

Example

Several ways to use a GmicImage simply:

```
import gmic
empty_1x1x1_black_image = gmic.GmicImage() # or gmic.GmicImage(None,1,1,1,1) for␣
↪example
import struct
i = gmic.GmicImage(struct.pack('2f', 0.0, 1.5), 1, 1) # 2D 1x1 image
gmic.run('add 1', i) # GmicImage injection into G'MIC's interpreter
i # Using GmicImage's repr() string representation
# Output: <gmic.GmicImage object at 0x7f09bfb504f8 with _data address at␣
↪0x22dd5b0, w=1 h=1 d=1 s=1 shared=0>
i(0,0) == 1.0 # Using GmicImage(x,y,z) pixel reading operator after initialization
gmic.run('resize 200%,200%', i) # Some G'MIC operations may reallocate the image␣
↪buffer in place without risk
i._width == i._height == 2 # Use the _width, _height, _depth, _spectrum, _data, _
↪data_str, _is_shared read-only attributes
```

Parameters

- **data** (`Optional[bytes]`) – Raw data for the image (must be a sequence of 4-bytes floats blocks, with as many blocks as all the dimensions multiplied together).
- **width** (`Optional[int]`) – Image width in pixels. Defaults to 1.
- **height** (`Optional[int]`) – Image height in pixels. Defaults to 1.
- **depth** (`Optional[int]`) – Image height in pixels. Defaults to 1.

- **spectrum** (*Optional[int]*) – Number of channels per pixel. Defaults to 1.
- **shared** (*Optional[bool]*) – C++ option: whether the buffer should be shareable between several GmicImages and operations. Defaults to False.

Note: GmicImage(x=0, y=0, z=0, s=0)

This instance method allows you to read pixels in a `GmicImage` for given coordinates.

You can read, but cannot write pixel values by passing some or all coordinates the following way:

```python
import gmic
images = []
gmic.run("sp apples", images)
image = images[0]
print(image(0,2,0,2)) # or image(y=2,z=2)
print(image(0,0,0,0)) # or image()
for x in range(image._width):
    for y in range(image._height):
        for z in range(image._depth):
            for c in range(image._spectrum):
                print(image(x,y,z,c))
```

`__copy__`()
Copy method for copy.copy() support. Deepcopying and pickle-ing are not supported.

`from_PIL`(*pil_image*)
Make a `GmicImage` from a 2D `PIL.Image.Image` object. Equates to `gmic.GmicImage.from_numpy_helper(numpy.array(pil_image), deinterleave=True)`. Will import `PIL.Image` and `numpy` for conversion.

Parameters **pil_image** (*PIL.Image.Image*) – An image to convert into `GmicImage`.

Returns A new `gmic.GmicImage` based on the input `PIL.Image.Image` data.

Return type *gmic.GmicImage*

`from_numpy`(*numpy_array*)
Make a GmicImage from a 1-4 dimensions numpy.ndarray. Simplified version of `GmicImage.from_numpy_helper` with `deinterleave=True`.

Parameters **numpy_array** (*numpy.ndarray*) – A non-empty 1D-4D Numpy array.

Returns A new `GmicImage` based the input `numpy.ndarray` data.

Return type *GmicImage*

Raises ***GmicException***, **TypeError** – Look at the exception message for details. Matrices with dimensions <1D or >4D will be rejected.

`from_numpy_helper`(*numpy_array*, *deinterleave=False*, *permute=''*)
Make a GmicImage from a 1-4 dimensions numpy.ndarray.

G'MIC works with (width, height, depth, spectrum/channels) matrix layout, with 32bit-float pixel values deinterleaved (ie. RRR,GGG,BBB). If your matrix is less than 4D, G'MIC will tentatively add append void dimensions to it (eg. for a shape of (3,1) -> (3,1,1,1)). You can avoid this by using `numpy.expand_dims` or `numpy.atleast_*d` functions yourself first. If your pixel values (ie. `numpy.ndarray.dtype`) are not in a `float32` format, G'MIC will tentatively call `numpy.astype(numpy_array, numpy.float32)` to cast its contents first.

Example

Several ways to use a GmicImage simply:

```
import gmic
empty_1x1x1_black_image = gmic.GmicImage() # or gmic.GmicImage(None,1,1,1,1)␣
↪for example
import struct
i = gmic.GmicImage(struct.pack('2f', 0.0, 1.5), 1, 1) # 2D 1x1 image
gmic.run('add 1', i) # GmicImage injection into G'MIC's interpreter
i # Using GmicImage's repr() string representation
# Output: <gmic.GmicImage object at 0x7f09bfb504f8 with _data address at␣
↪0x22dd5b0, w=1 h=1 d=1 s=1 shared=0>
i(0,0) == 1.0 # Using GmicImage(x,y,z) pixel reading operator after␣
↪initialization
gmic.run('resize 200%,200%', i) # Some G'MIC operations may reallocate the␣
↪image buffer in place without risk
i._width == i._height == 2 # Use the _width, _height, _depth, _spectrum, _
↪data, _data_str, _is_shared read-only attributes
```

Parameters

- **numpy_array** (*numpy.ndarray*) – A non-empty 1D-4D Numpy array.
- **deinterleave** (*Optional[bool]*) – If `True`, pixel channel values will be deinterleaved inside the GmicImage data. If `False`, pixel channels vector values will be untouched. Defaults to `False`.
- **permute** (*Optional[str]*) – If non-empty, a G'MIC `permute` operation will be run with this parameter (eg. yxzc) on the input matrix before saving into the GmicImage. See https://gmic.eu/reference.shtml#permute Defaults to "" (no permutation).

Returns A new `GmicImage` based the input `numpy.ndarray` data.

Return type *GmicImage*

Raises ***GmicException***, **TypeError** – Look at the exception message for details. Matrices with dimensions <1D or >4D will be rejected.

from_skimage()

GmicImage.from_numpy_helper(numpy_array, deinterleave=False, permute='')

Make a GmicImage from a 1-4 dimensions numpy.ndarray.

G'MIC works with (width, height, depth, spectrum/channels) matrix layout, with 32bit-float pixel values deinterleaved (ie. RRR,GGG,BBB). If your matrix is less than 4D, G'MIC will tentatively add append void dimensions to it (eg. for a shape of (3,1) -> (3,1,1,1)). You can avoid this by using `numpy.expand_dims` or `numpy.atleast_*d` functions yourself first. If your pixel values (ie. `numpy.ndarray.dtype`) are not in a `float32` format, G'MIC will tentatively call `numpy.astype(numpy_array, numpy.float32)` to cast its contents first.

Example

Several ways to use a GmicImage simply:

```
import gmic
empty_1x1x1_black_image = gmic.GmicImage() # or gmic.GmicImage(None,1,1,1,1)␣
↪for example
```

(continues on next page)

(continued from previous page)

```
import struct
i = gmic.GmicImage(struct.pack('2f', 0.0, 1.5), 1, 1) # 2D 1x1 image
gmic.run('add 1', i) # GmicImage injection into G'MIC's interpreter
i # Using GmicImage's repr() string representation
# Output: <gmic.GmicImage object at 0x7f09bfb504f8 with _data address at␣
↪0x22dd5b0, w=1 h=1 d=1 s=1 shared=0>
i(0,0) == 1.0 # Using GmicImage(x,y,z) pixel reading operator after␣
↪initialization
gmic.run('resize 200%,200%', i) # Some G'MIC operations may reallocate the␣
↪image buffer in place without risk
i._width == i._height == 2 # Use the _width, _height, _depth, _spectrum, _
↪data, _data_str, _is_shared read-only attributes
```

Parameters

- **numpy_array** (*numpy.ndarray*) – A non-empty 1D-4D Numpy array.
- **deinterleave** (*Optional[bool]*) – If `True`, pixel channel values will be deinterleaved inside the GmicImage data. If `False`, pixel channels vector values will be untouched. Defaults to `False`.
- **permute** (*Optional[str]*) – If non-empty, a G'MIC `permute` operation will be run with this parameter (eg. yxzc) on the input matrix before saving into the GmicImage. See https://gmic.eu/reference.shtml#permute Defaults to "" (no permutation).

Returns A new `GmicImage` based the input `numpy.ndarray` data.

Return type *GmicImage*

Raises ***GmicException***, **TypeError** – Look at the exception message for details. Matrices with dimensions <1D or >4D will be rejected.

to_PIL(*astype=numpy.uint8*, *squeeze_shape=True*, *mode='RGB'*)

Make a 2D 8-bit per pixel RGB PIL.Image from any GmicImage. Equates to `PIL.Image.fromarray(self.to_numpy_helper(astype=astype, squeeze_shape=squeeze_shape, interleave=True, permute='zyxc'), mode)`. Will import `PIL.Image` and `numpy`.

This method uses `numpy` for conversion. Thus `astype` is used in a `numpy.ndarray.astype()` `conversion pass and samewise for ``squeeze_shape`. :param astype: Will be used for casting your image's pixel. :type astype: type :param squeeze_shape: if True, your image shape has '1' components removed, is usually necessary to convert from G'MIC 3D to PIL.Image 2D only. :type squeeze_shape: bool :param mode: the PIL Image mode to use. see https://pillow.readthedocs.io/en/stable/handbook/concepts.html#concept-modes :type mode: str

Returns A new `PIL.Image` based on the instance `GmicImage` data from which you call this method.

Return type PIL.Image

to_numpy()

Make a numpy.ndarray from a GmicImage. Simplified version of `GmicImage.to_numpy_helper` with `interleave=True`.

Returns A new `numpy.ndarray` based the input `GmicImage` data.

Return type numpy.ndarray

`to_numpy_helper`(*astype=numpy.float32*, *interleave=False*, *permute=”*, *squeeze_shape=False*)

Make a numpy.ndarray from a GmicImage. G’MIC does not squeeze dimensions internally, so unless you use the `squeeze_shape` flag calling `numpy.squeeze` for you, the output matrix will be 4D.

Parameters

- **astype** (*numpy.dtype*) – The type to which G’MIC’s float32 pixel values will cast to for the output matrix.
- **interleave** (*Optional[bool]*) – If `True`, pixel channel values will be interleaved (ie. RGB, RGB, RGB) within the numpy array. If `False`, pixel channels vector values will be untouched/deinterleaved (ie. RRR,GGG,BBB). Defaults to `False`.
- **permute** (*Optional[str]*) – If non-empty, a G’MIC `permute` operation will be run with this parameter (eg. yxzc) on the output matrix before saving into the GmicImage. See https://gmic.eu/reference.shtml#permute Defaults to “” (ie. no permutation).

Returns A new `numpy.ndarray` based the input `GmicImage` data.

Return type numpy.ndarray

`to_skimage`()

GmicImage.to_numpy_helper(astype=numpy.float32, interleave=False, permute=”, squeeze_shape=False)

Make a numpy.ndarray from a GmicImage. G’MIC does not squeeze dimensions internally, so unless you use the `squeeze_shape` flag calling `numpy.squeeze` for you, the output matrix will be 4D.

Parameters

- **astype** (*numpy.dtype*) – The type to which G’MIC’s float32 pixel values will cast to for the output matrix.
- **interleave** (*Optional[bool]*) – If `True`, pixel channel values will be interleaved (ie. RGB, RGB, RGB) within the numpy array. If `False`, pixel channels vector values will be untouched/deinterleaved (ie. RRR,GGG,BBB). Defaults to `False`.
- **permute** (*Optional[str]*) – If non-empty, a G’MIC `permute` operation will be run with this parameter (eg. yxzc) on the output matrix before saving into the GmicImage. See https://gmic.eu/reference.shtml#permute Defaults to “” (ie. no permutation).

Returns A new `numpy.ndarray` based the input `GmicImage` data.

Return type numpy.ndarray

`gmic.`**`run`**(*command*, *images=None*, *image_names=None*)

Run the G’MIC interpreter with a G’MIC language command(s) string, on 0 or more nameable GmicImage(s). This is a short-hand for calling `gmic.Gmic().run` with the exact same parameters signature.

Note (single-image short-hand calling): if `images` is a `GmicImage`, then `image_names` must be either a `str` or be omitted.

Note (interpreter warm-up): calling `gmic.run` multiple times is inefficient as it spawns then drops a new G’MIC interpreter instance for every call. For better performance, you can tie a `gmic.Gmic` G’MIC interpreter instance to a variable instead and call its `run` method multiple times. Look at `gmic.Gmic.run` for more information.

Example

Several ways to use the module-level `gmic.run()` function:

```
import gmic
import struct
import random
gmic.run('echo_stdout \'hello world\'') # G'MIC command without images parameter
a = gmic.GmicImage(struct.pack(*('256f',) + tuple([random.random() for a in␣
↪range(256)])), 16, 16) # Build 16x16 greyscale image
gmic.run('blur 12,0,1 resize 50%,50%', a) # Blur then resize the image
a._width == a._height == 8 # The image is half smaller
gmic.run('display', a) # If you have X11 enabled (linux only), show the image in␣
↪a window
image_names = ['img_' + str(i) for i in range(10)] # You can also name your␣
↪images if you have several (optional)
images = [gmic.GmicImage(struct.pack(*((str(w*h)+'f',) + (i*2.0,)*w*h)), w, h)␣
↪for i in range(10)] # Prepare a list of image
gmic.run('add 1 print', images, image_names) # And pipe those into the interpreter
gmic.run('blur 10,0,1 print', images[0], 'my_pic_name') # Short-hand 1-image␣
↪calling style
```

Parameters

- **command** (*str*) – An image-processing command in the G'MIC language
- **images** (*Optional[Union[List[gmic.GmicImage], gmic.GmicImage]]*) – A list of `GmicImage` items that G'MIC will edit in place, or a single `gmic.GmicImage` which will used for input only. Defaults to None. Put a list variable here, not a plain `[]`. If you pass a list, it can be empty if you intend to fill or complement it using your G'MIC command.
- **image_names** (*Optional[List<str>]*) – A list of names for the images, defaults to None. In-place editing by G'MIC can happen, you might want to pass your list as a variable instead.

Returns Returns `None` or raises a `GmicException`.

Return type None

Raises ***GmicException*** – This translates' G'MIC C++ same-named exception. Look at the exception message for details.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX