gmic-py

Release 2.9.4-alpha1

Jonathan-David Schröder, David Tschumperlé

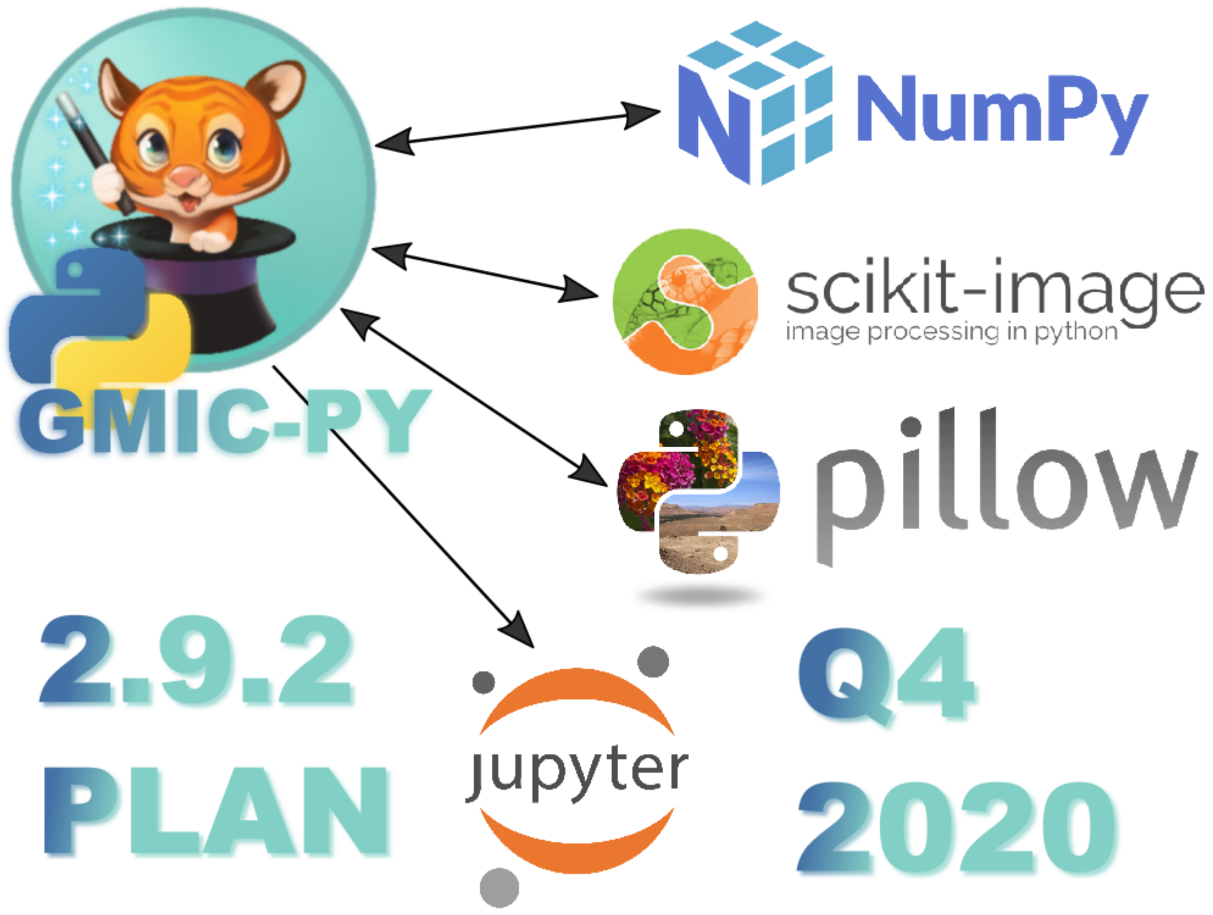
Apr 09, 2021

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:

```python
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:

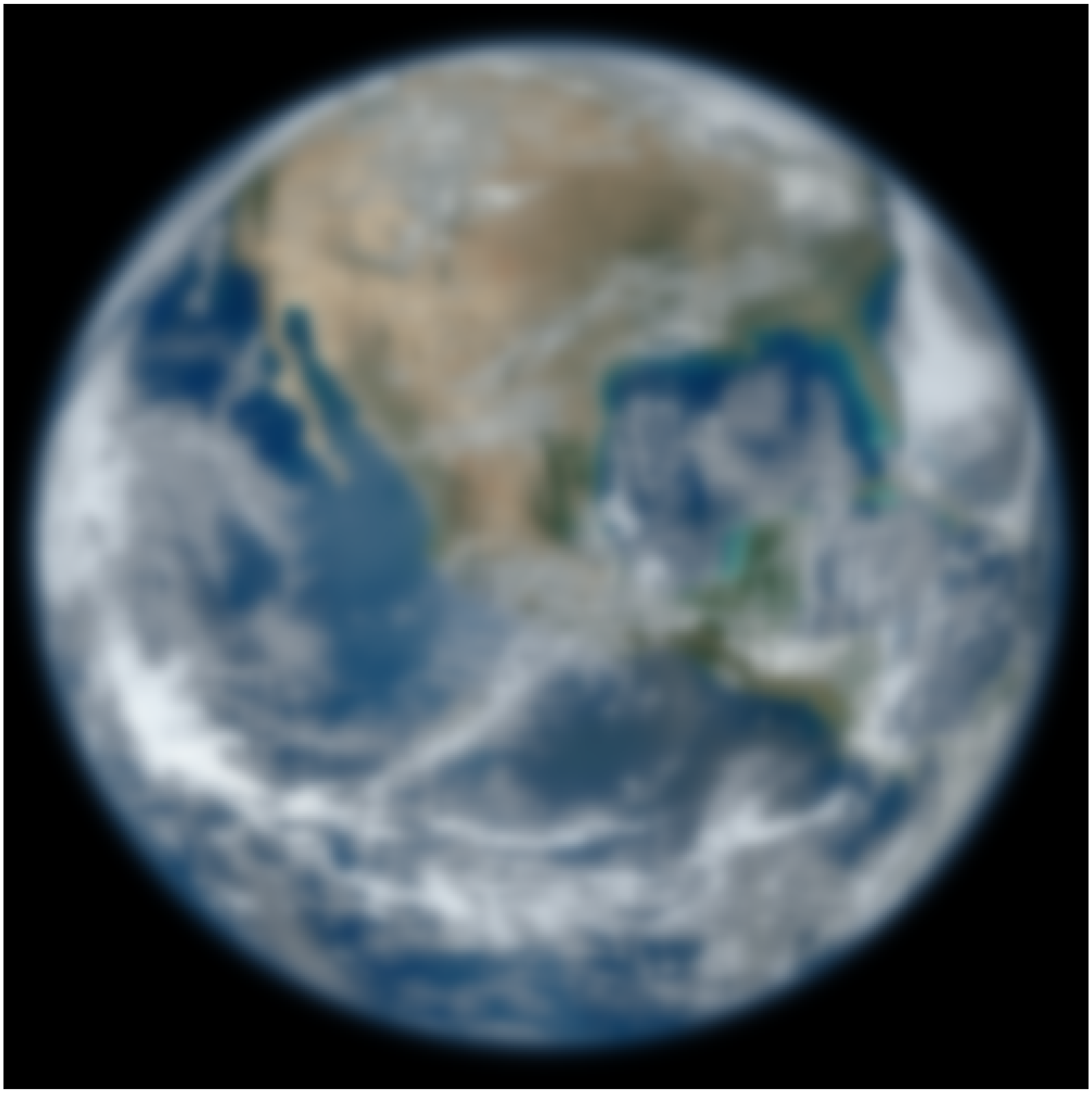


Fig. 1: sp earth blur 4

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).
```

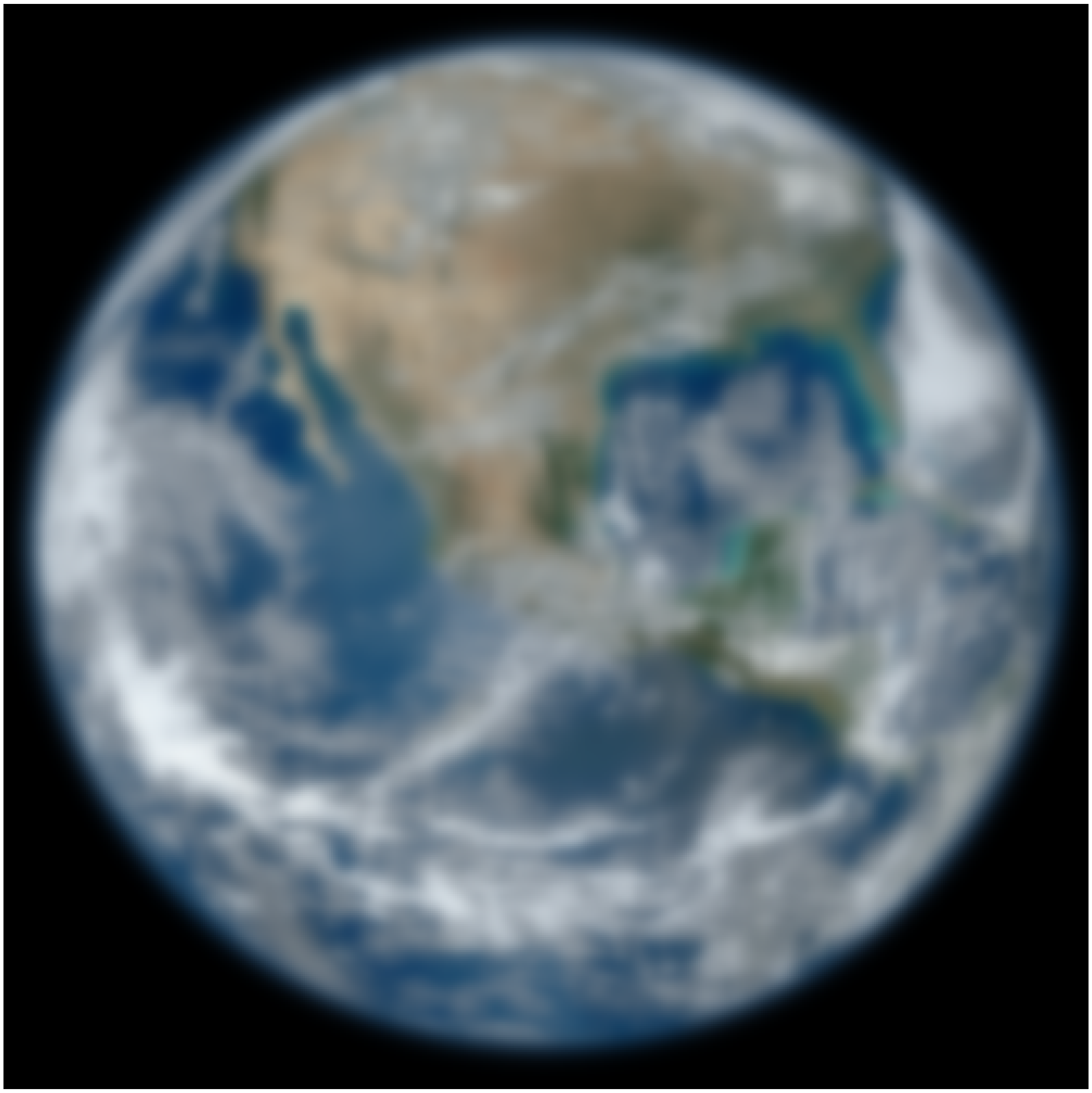


Fig. 2: sp earth blur 4

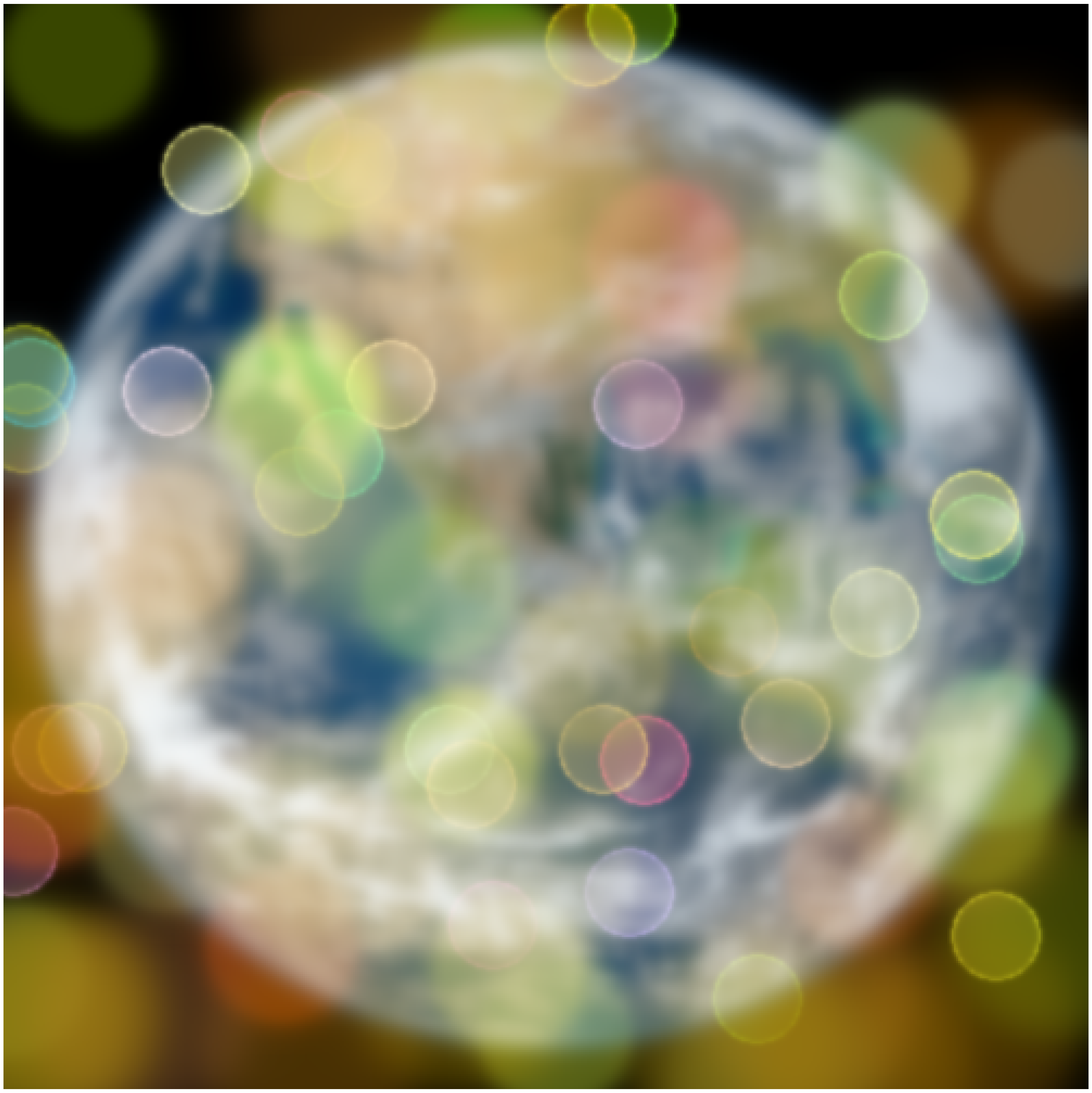


Fig. 3: sp earth blur 4 fx_bokeh 3,8,0,30,8,4,0.3,0.2,210,210,80,160,0.7,30,20,20,1,2,170,130,20,110,0.15,0

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).

```python
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
```

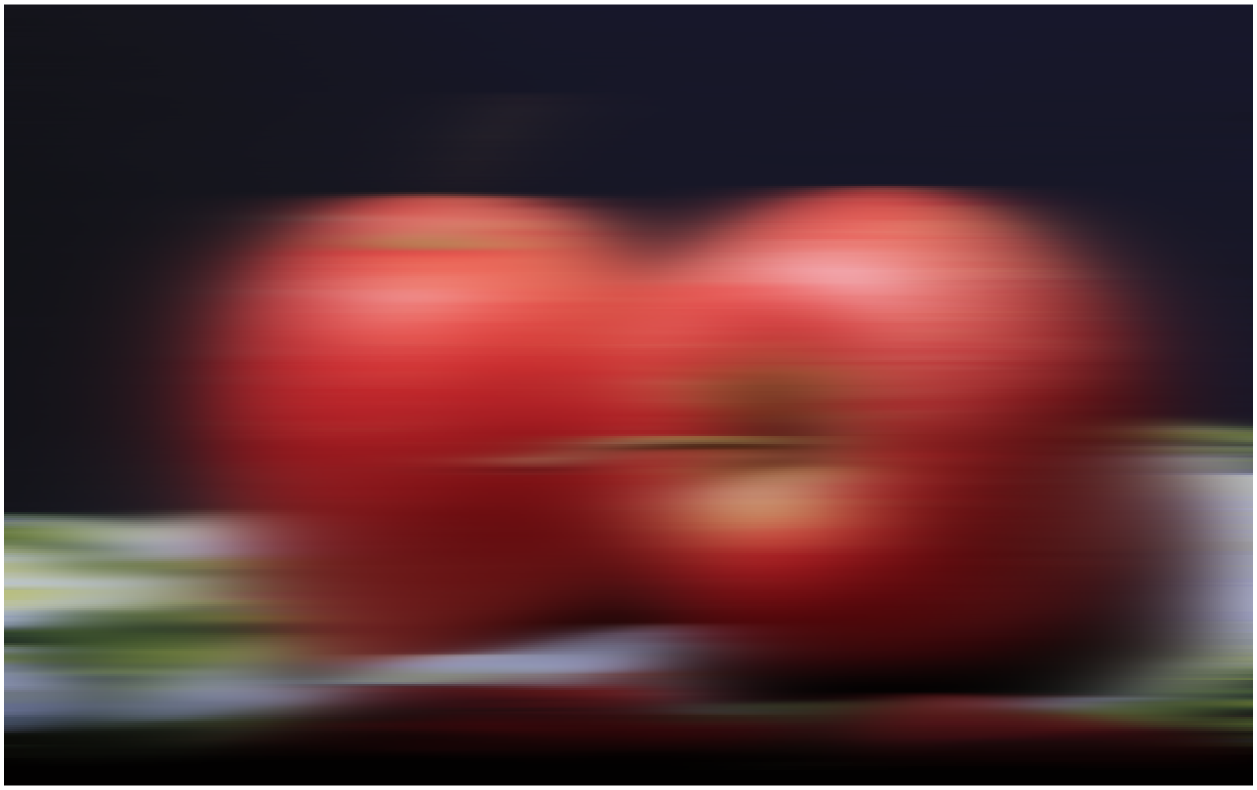


Fig. 4: sp apples blur_x 30

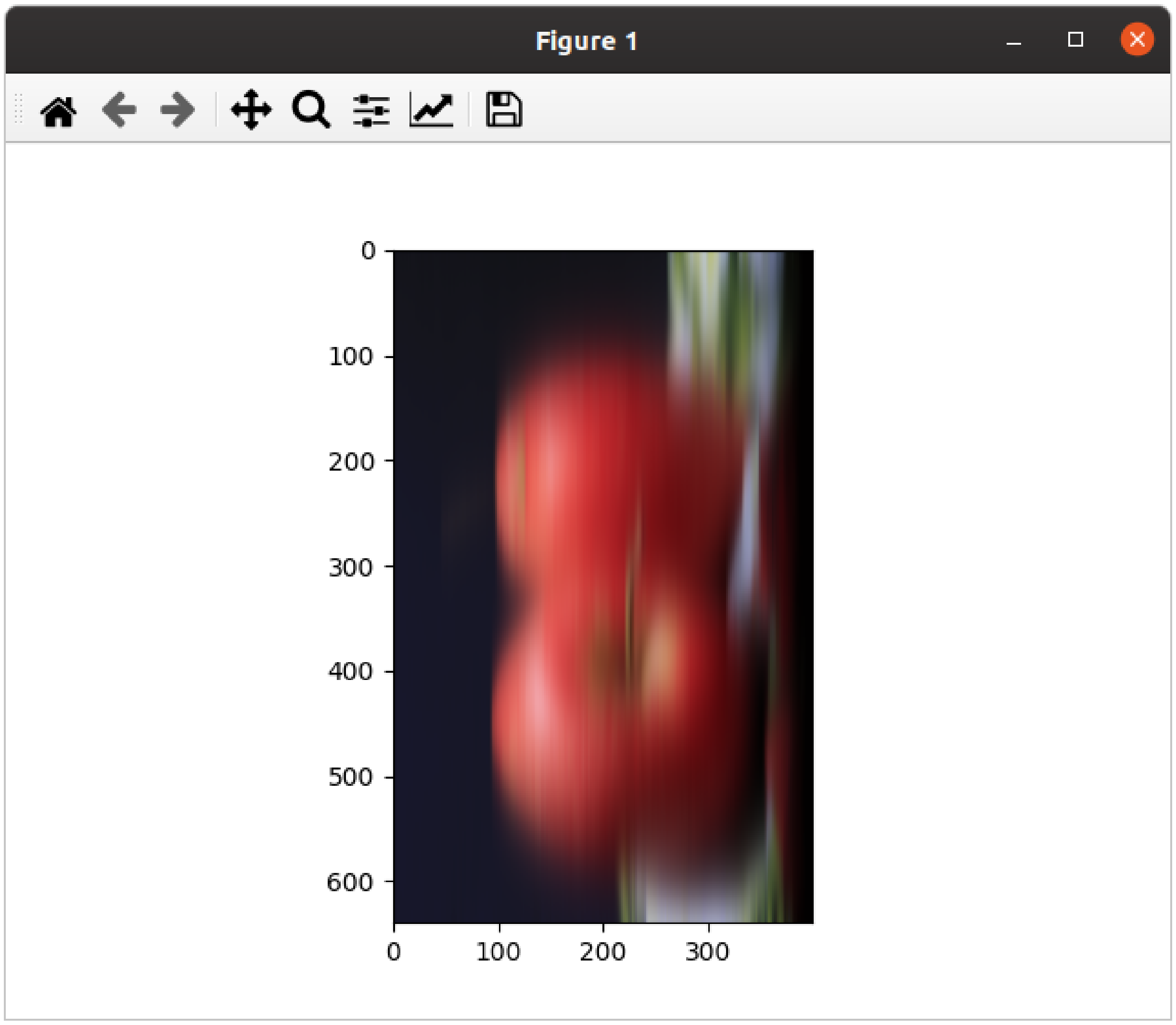
Now let us try a display from Numpy using Matplotlib

```python
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)

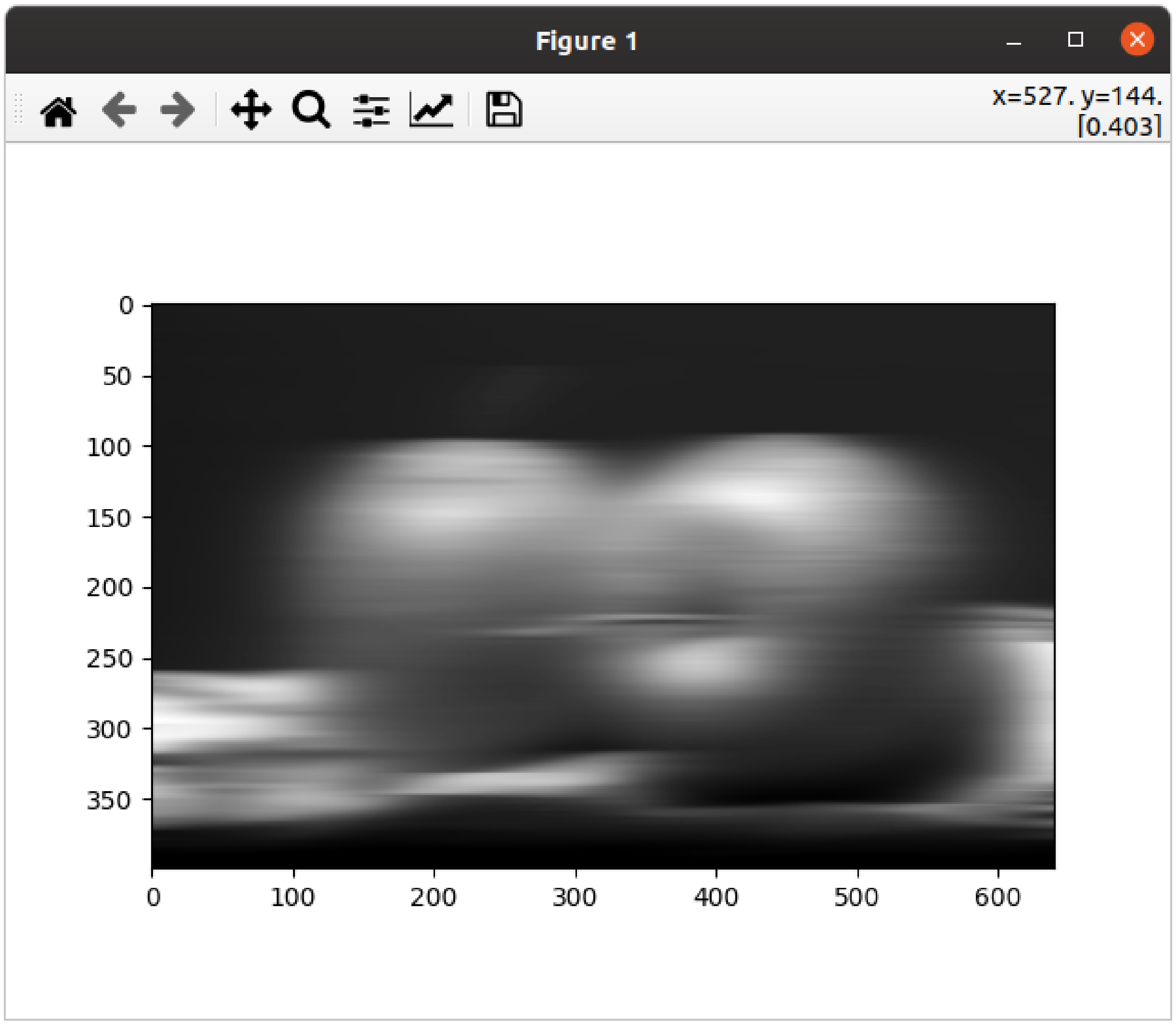
(continued from previous page)

```
# 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()
```

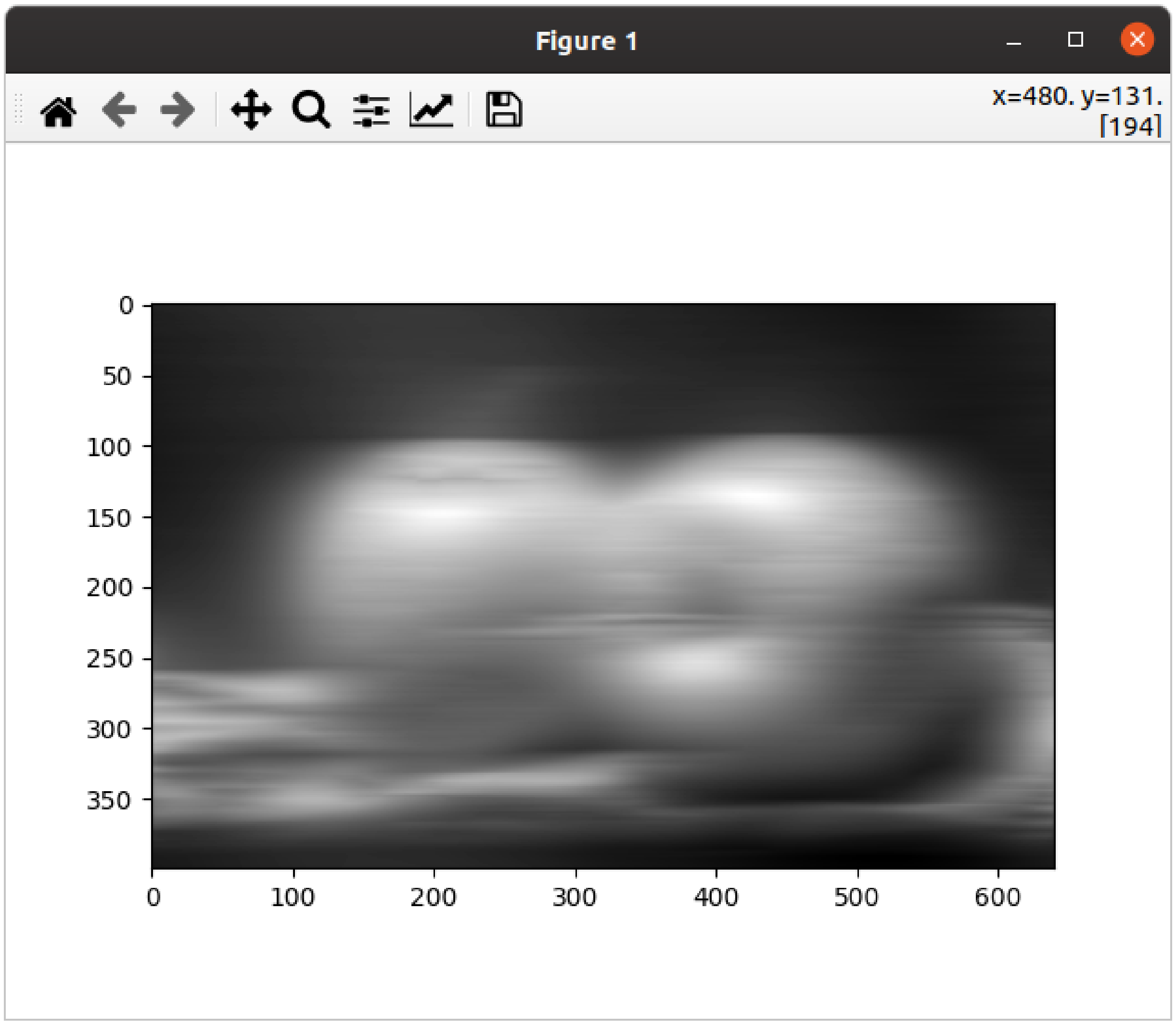


Figure 1
x=480. y=131.
[194]
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600

CHAPTER

THREE

COOKBOOK FOR G'MIC BEGINNERS

This cookbook is intended for newcomers to the G'MIC language. It takes you by the hand step by step, starting from your Python (optionnally Numpy, Scipy..) language skills and making you learn the G'MIC image processing language the easy way a smooth methodology. `gmic-py` allows to you evaluate G'MIC expressions and scripts easily, but breaking the walls of the G'MIC syntax is key for newcomers to leverage the power of the G'MIC engine.

The progression of this cookbook follows more or less closely five gmic.eu website's articles (Basics, Images, Selections and Command Decorations, Images have edges - Now What?, Conjuring Images out of the Aether and Other Generators). It was thought of originally in this pixls.us community discussion.

3.1 Basics

TODO

3.2 Images

TODO

3.3 Selections and Command Decorations

TODO

3.4 Images have Edges - Now What?

TODO

3.5 Conjuring Images out of the Aether and Other Generators

TODO

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
```

```
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

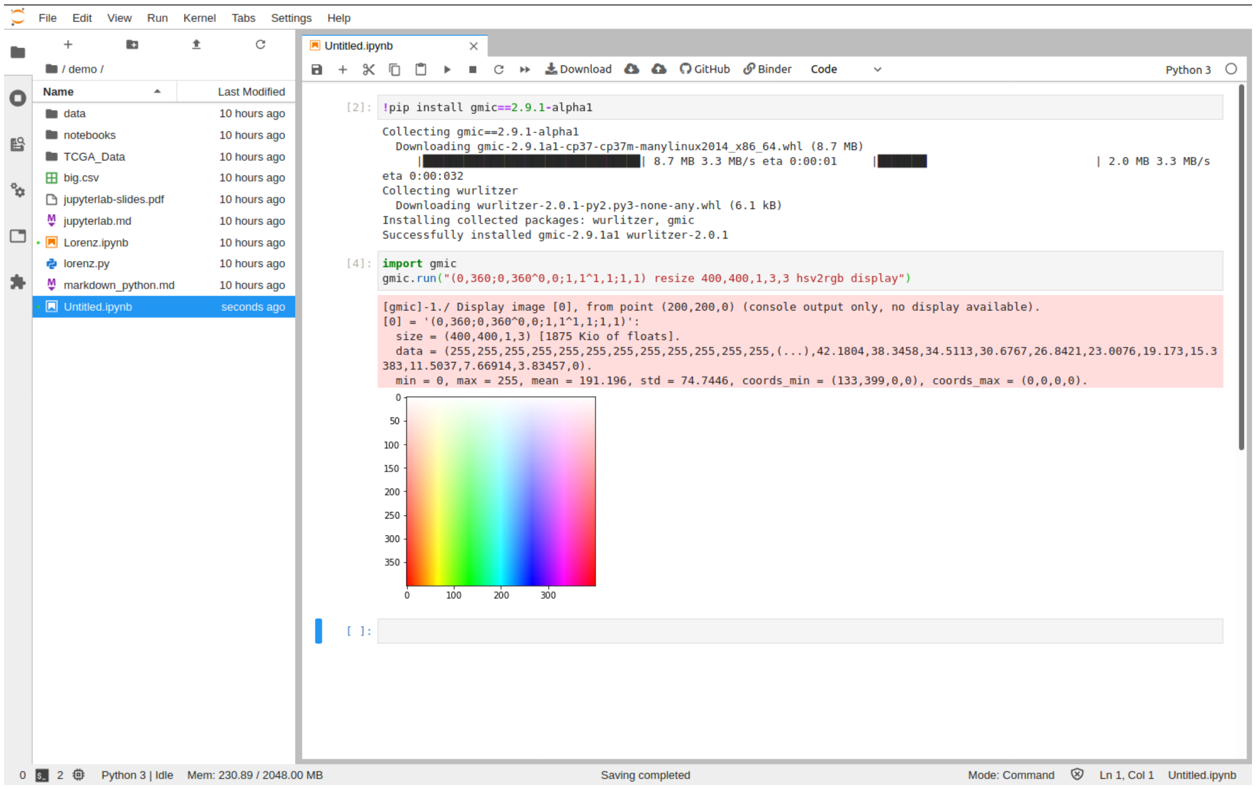
SEVEN

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:


File Edit View Run Kernel Tabs Settings Help
/ demo /
Untitled.ipynb
Download GitHub Binder Code
Python 3
0 2 Python 3 | Idle Mem: 230.89 / 2048.00 MB Saving completed Mode: Command Ln 1, Col 1 Untitled.ipynb

Name	Last Modified
data	10 hours ago
notebooks	10 hours ago
TCGA_Data	10 hours ago
big.csv	10 hours ago
jupyterlab-slides.pdf	10 hours ago
jupyterlab.md	10 hours ago
Lorenz.ipynb	10 hours ago
lorenz.py	10 hours ago
markdown_python.md	10 hours ago
Untitled.ipynb	seconds ago

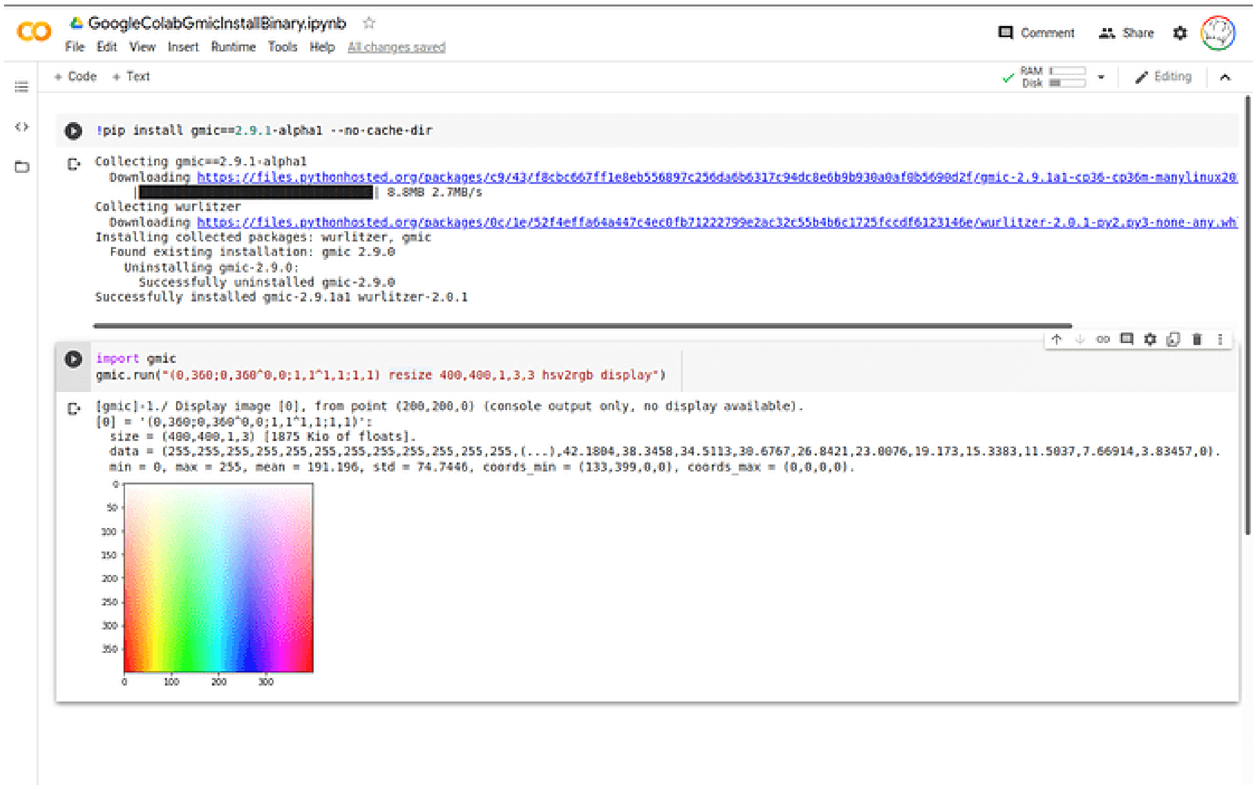
```
[2]: !pip install gmic==2.9.1-alpha1
```

```
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).
```

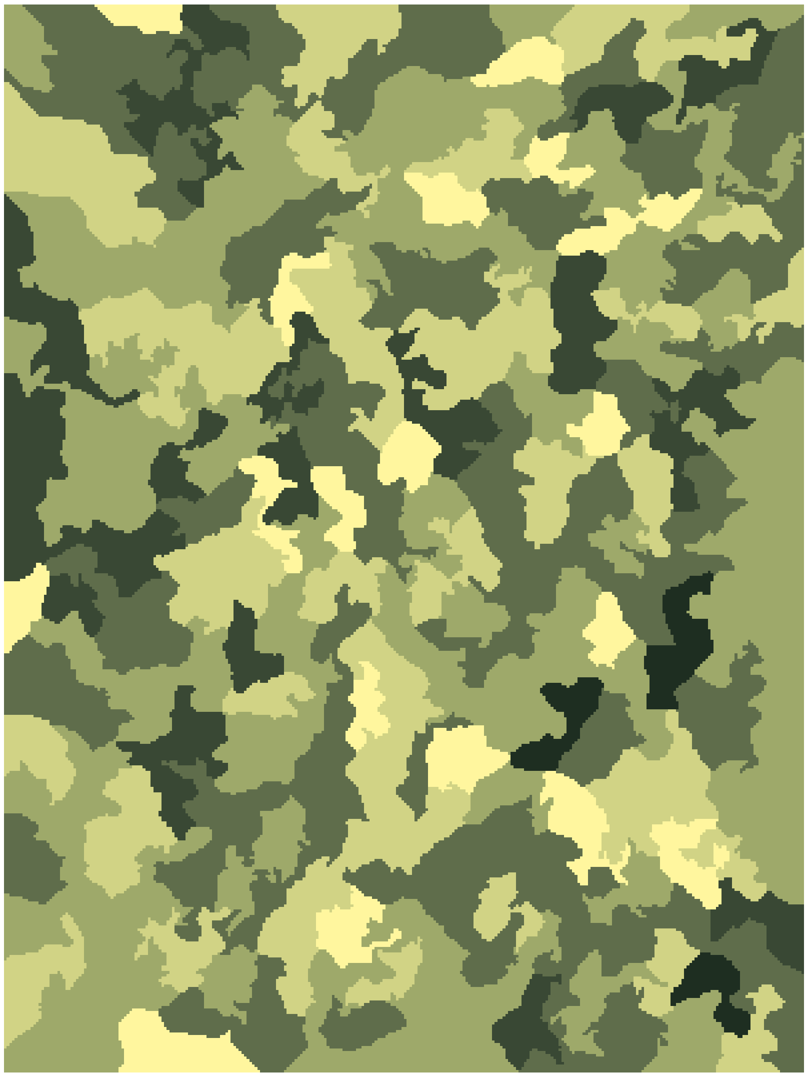


Fig. 3: 300,400,1,3 fx_camouflage 9,12,100,30,46,33,75,90,65,179,189,117,255,246,158

```
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>]
```

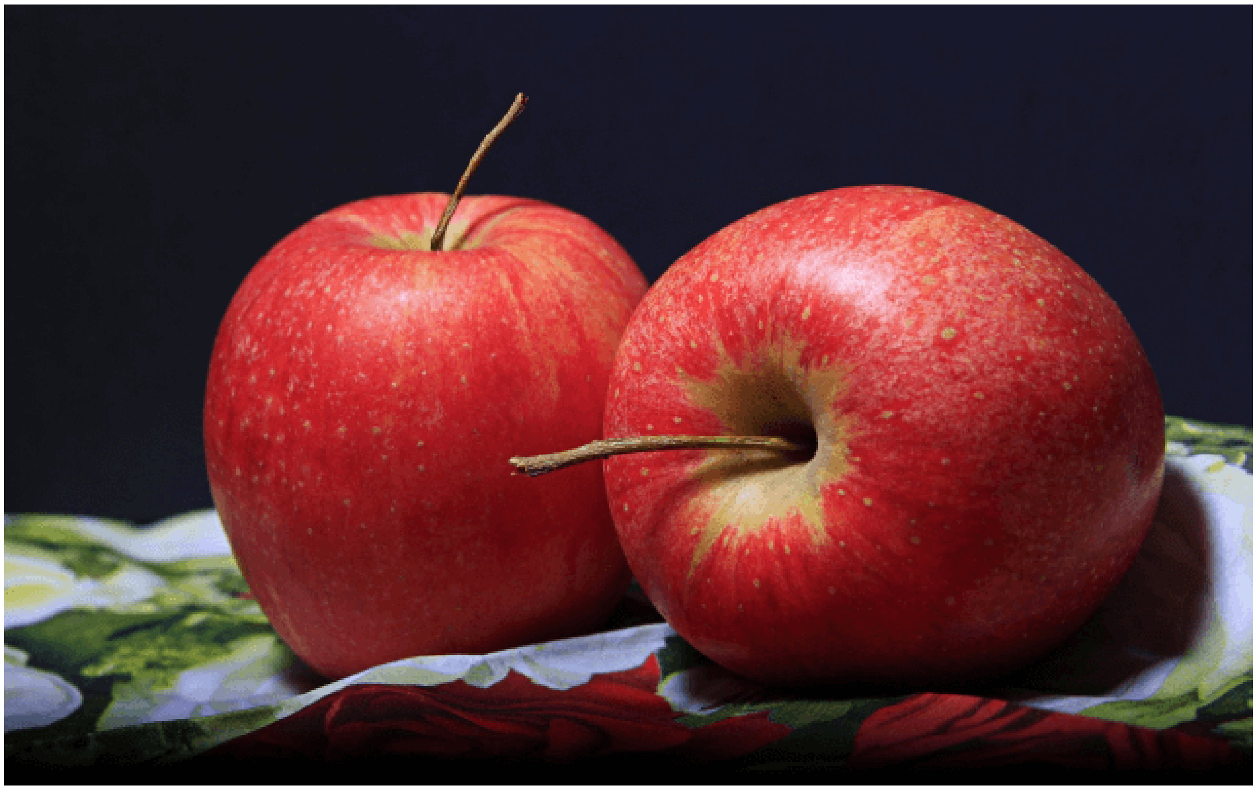


Fig. 4: sp apples

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:

```
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