pygtc Documentation

Release 0.3

Sebastian Bocquet and Faustin W. Carter

Nov 13, 2018

Contents

Make a publication-ready giant-triangle-confusogram (GTC) with just one line of code!

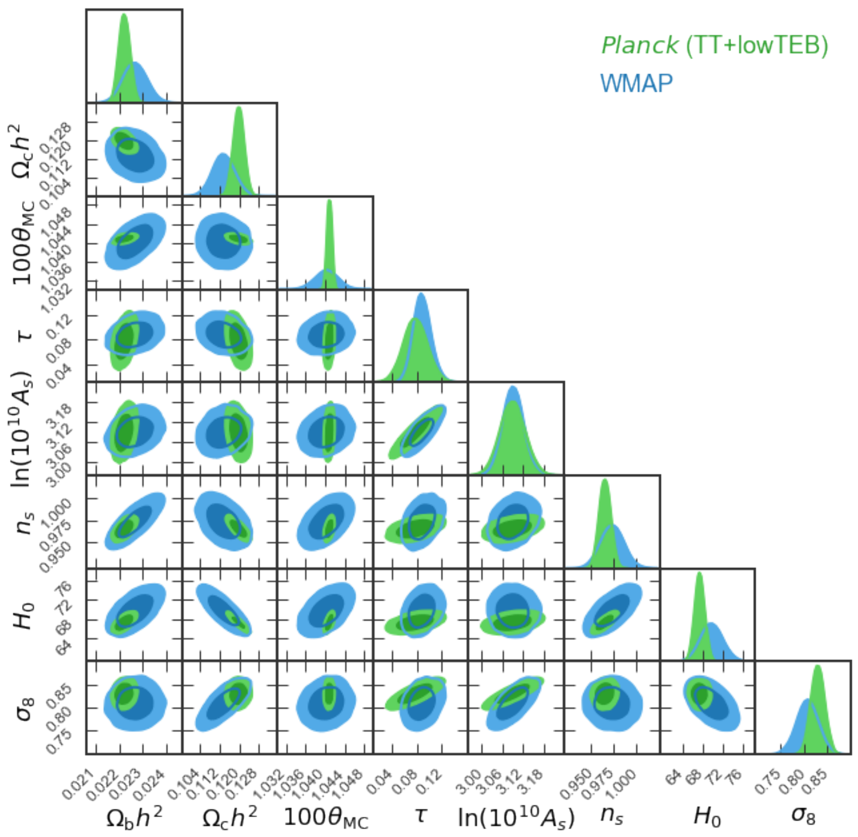
This documentation is for the `pygtc` package, which is hosted at GitHub.

What is a Giant Triangle Confusogram?

A Giant-Triangle-Confusogram (GTC, aka triangle plot) is a way of displaying the results of a Monte-Carlo Markov Chain (MCMC) sampling or similar analysis. The recovered parameter constraints are displayed on a grid in which the diagonal shows the one-dimensional posteriors (and, optionally, priors) and the lower-left triangle shows the pairwise projections. You might want to look at a plot like this if you are fitting model to data and want to see the parameter covariances along with the priors.

Although several other packages exists to make such a plot, we were unsatisfied with the amount of extra work required to massage the result into something we were happy to publish. With `pygtc`, we hope to take that extra legwork out of the equation by providing a package that gives a figure that is publication-ready on the first try!

Here's an example of a GTC (generated from Planck and WMAP data):



Note about contour levels In the above example, you see two sets of contour levels at 68% and 95% (in addition, pygtc could also show the 99% contour level if you want). Note that these are indeed the 68% and 95% contour levels, and not 1, 2, and 3 sigma levels (which, in two dimensions, correspond to the 39%, 86%, and 99% contour levels). Your posterior distributions will in general not be Gaussian, and so there is no right or wrong in choosing one or the other definition. We therefore recommend that you stick to the definition that is commonly used in your field, and clearly state what quantities you are showing in your figure caption. You can switch to displaying sigma levels by setting *sigmaContourLevels* to *True*.

CHAPTER 1

Contents:

1.1 Installation

1.1.1 Required packages

pygtc is compatible with Python 2.7 and 3.6 and requires the following packages:

- numpy >= 1.5
- matplotlib >= 1.5.3 (preferably >= 2.0)
- scipy (optional)
- nose (optional – only needed for running unit tests)
- nose-exclude (optional – only needed for running unit tests)

1.1.2 Downloading and installing

The latest stable version of pygtc is hosted at PyPi.

If you like pip, you can install with:

```
pip install pygtc
```

Or, alternatively, download and extract the tarball from the above link and then install from source with:

```
cd /path/to/pygtc
python setup.py install
```

1.1.3 Development

Development happens at github. You can clone the repo with:

```
git clone https://github.com/SebastianBocquet/pygtc
```

And you can install it in developer mode with pip:

```
pip install -e /path/to/pygtc
```

or from source:

```
cd /path/to/pygtc
python setup.py develop
```

1.1.4 Running tests

For tests to *for sure* run properly, you'll want to have matplotlib v2 or greater installed. If you are using an earlier version you'll need at least matplotlib v1.5.3, as they fixed a bug in their `image_comparison` decorator, but several tests will fail anyhow due to slightly different image sizes (v1.5 adds 5 pixels in the x-dimension for some reason). You'll need `nose` installed to run the tests, although pygtc functions fine without it. You should also install `nose-exclude` to make sure that the right matplotlib backend gets loaded with the tests. You also should have the Arial font installed, as that is pygtc's default font and tests will "fail" if matplotlib falls back on Bitstream Vera Sans (even though the images produced might look perfectly fine). Test base images were produced on Mac OSX using the `Agg` backend and if you are on another system there is no guarantee that you will get a pixel-perfect copy of what the Agg backend produces. However, the images produced by the tests should still look great! They are saved to a folder called `result_images` in whatever directory you ran the tests from.

There are two ways to run the test suite. You can use the nosetests utility from within the pygtc package directory:

```
cd /path/to/pygtc
nosetests
```

Or, you can run the tests as a script:

```
python /path/to/pygtc/tests/test_plotGTC.py
```

In either case, there are 25 tests to run, and it should take between 20-30 seconds to run them all. If the first test fails, there may be something wrong with your matplotlib install in general (or maybe something weird in your rcParams). If you are missing pandas or scipy, a few tests will be skipped. If you get a ton of errors make sure you read the first paragraph in this section and you have all the prerequisites installed. If matplotlib can't find Arial and you recently installed it, delete your matplotlib font cache and try again. If errors persist, let us know at GitHub.

1.1.5 Contribution and/or bug reports

If you like this project and want to contribute, send a message over at the gitHub repo and get involved. If you find a bug, please open an issue at gitHub. Better yet, you can try and fix the bug and submit a pull request!

1.2 Example 1: Making a GTC/triangle plot with pygtc

1.2.1 Import dependencies

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina' # For mac users with Retina display
from matplotlib import pyplot as plt
import numpy as np
import pygtc
```

1.2.2 Generate fake data

Let's create two sets of fake sample points with 8 dimensions each. Note that chains are allowed to have different lengths.

```
# Create Npoints samples from random multivariate, nDim-dimensional Gaussian
def create_random_samples(nDim, Npoints):
    means = np.random.rand(nDim)
    cov = .5 - np.random.rand(nDim**2).reshape((nDim,nDim))
    cov = np.triu(cov)
    cov += cov.T - np.diag(cov.diagonal())
    cov = np.dot(cov,cov)
    samples =  np.random.multivariate_normal(means, cov, Npoints)
    return samples

# Create two sets of fake data with 8 parameters
np.random.seed(0) # To be able to create the same fake data over and over again
samples1 = create_random_samples(8, 50000)
samples2 = 1+create_random_samples(8, 70000)
```

1.2.3 Omit one parameter for one chain

Let's assume the samples1 does not include the second to last parameter. In the figure, we only want to show this parameter for samples2. pygtc will omit parameters that only contain nan.

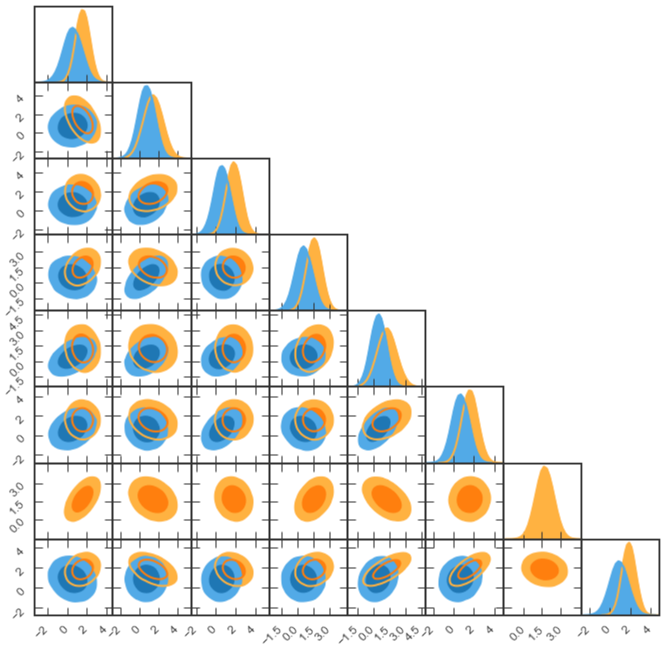
```
samples1[:,6] = None
```

1.2.4 Minimal example

Note that numpy throws a `RuntimeWarning` because we set one of the axes of `samples1` to `None` just above. As we understand the warning, let's move on!

```
GTC = pygtc.plotGTC(chains=[samples1,samples2])
```

```
pygtc/pygtc.py:558: RuntimeWarning: All-NaN axis encountered
  for k in range(nChains)]), axis=0)
pygtc/pygtc.py:560: RuntimeWarning: All-NaN slice encountered
  for k in range(nChains)]), axis=0)
```



1.2.5 Complete the figure

Now let's add: * axis and data labels * lines marking some important points in parameter space * Gaussian distributions on the 1d histograms that could indicate Gaussian priors we assumed

Note that all these must match number of parameters!

```
# List of parameter names, supports latex
# NOTE: For capital greek letters in latex mode, use \mathsf{}
names = ['param name',
         '$B_\mathrm{\lambda}$',
         '$E$', '$\\lambda$',
         'C',
         'D',
         '$\mathsf{\Omega}$',
         '$\\gamma$']

# Labels for the different chains
chainLabels = ["data1 $\lambda$",
               "data 2"]

# List of Gaussian curves to plot
#(to represent priors): mean, width
# Empty () or None if no prior to plot
priors = ((2, 1),
          (-1, 2),
          (),
          (0, .4),
          None,
          (1,1),
          None,
          None)
```

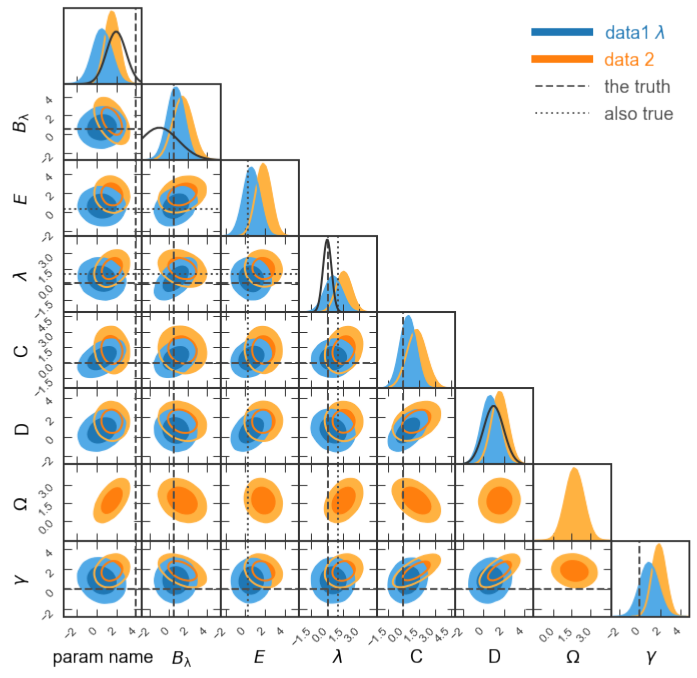
(continues on next page)

(continued from previous page)

```
# List of truth values, to mark best-fit or input values
# NOT a python array because of different lengths
# Here we choose two sets of truth values
truths = ((4, .5, None, .1, 0, None, None, 0),
          (None, None, .3, 1, None, None, None, None))

# Labels for the different truths
truthLabels = ( 'the truth',
               'also true')

# Do the magic
GTC = pygtc.plotGTC(chains=[samples1,samples2],
                    paramNames=names,
                    chainLabels=chainLabels,
                    truths=truths,
                    truthLabels=truthLabels,
                    priors=priors)
```

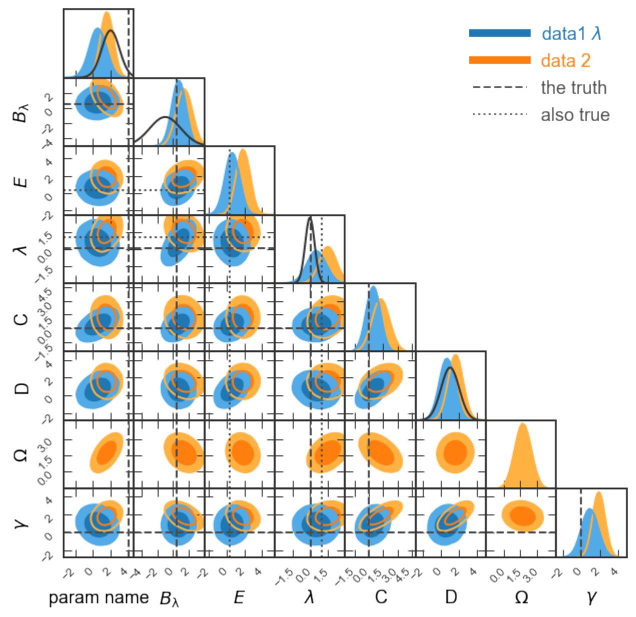


1.2.6 Make figure publication ready

- See how the prior for B_λ is cut off on the left? Let's display B_λ in the range (-5,4). Also, we could show a narrower range for λ like (-3,3).
- Given that we're showing two sets of truth lines, let's show the line styles in the legend (`legendMarker=True`).
- Finally, let's make the figure size publication ready for MNRAS. Given that we're showing eight parameters, we'll want to choose `figureSize='MNRAS_page'` and show a full page-width figure.
- Save the figure as `fullGTC.pdf` and paste it into your publication!

```
# List of parameter ranges to show,
# empty () or None to let pyGTC decide
paramRanges = (None,
               (-5,4),
               (),
               (-3,3),
               None,
               None,
               None,
               None)

# Do the magic
GTC = pygtc.plotGTC(chains=[samples1,samples2],
                    paramNames=names,
                    chainLabels=chainLabels,
                    truths=truths,
                    truthLabels=truthLabels,
                    priors=priors,
                    paramRanges=paramRanges,
                    figureSize='MNRAS_page',
                    plotName='fullGTC.pdf')
```



1.2.7 Single 2d panel

See how the covariance between C and D is a ground-breaking result? Let's look in more detail! Here, we'll want single-column figures.

```
# Redefine priors and truths
priors2d = (None,(1,1))
truths2d = (0,None)

# The 2d panel and the 1d histograms
GTC = pygtc.plotGTC(chains=[samples1[:,4:6], samples2[:,4:6]],
```

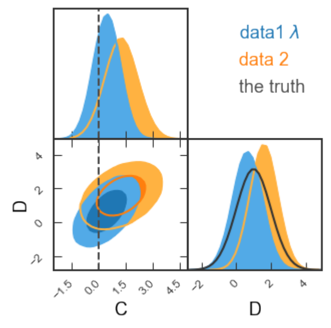
(continues on next page)

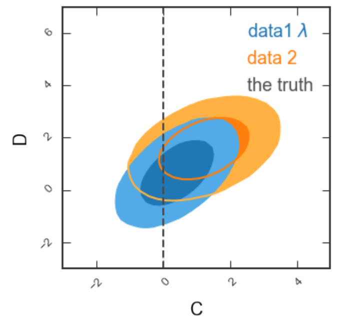
(continued from previous page)

```
                              paramNames=names[4:6],
                              chainLabels=chainLabels,
                              truths=truths2d,
                              truthLabels=truthLabels[0],
                              priors=priors2d,
                              figureSize='MNRAS_column')

# Only the 2d panel
Range2d = ((-3,5),(-3,7)) # To make sure there's enough space for the legend

GTC = pygtc.plotGTC(chains=[samples1[:,4:6],samples2[:,4:6]],
                              paramNames=names[4:6],
                              chainLabels=chainLabels,
                              truths=truths2d,
                              truthLabels=truthLabels[0],
                              priors=priors2d,
                              paramRanges=Range2d,
                              figureSize='MNRAS_column',
                              do1dPlots=False)
```





1.2.8 Single 1d panel

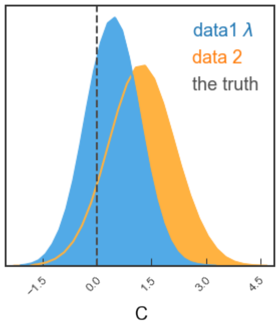
Finally, let's just plot the posterior on C

```
# Bit tricky, but remember each data set needs shape of (Npoints, nDim)
inputarr = [np.array([samples1[:,4]]).T,
            np.array([samples2[:,4]]).T]
truth1d = [0.]
GTC = pygtc.plotGTC(chains=inputarr,
                              paramNames=names[4],
```

(continues on next page)

(continued from previous page)

```
                        chainLabels=chainLabels,
                        truths=truth1d,
                        truthLabels=truthLabels[0],
                        figureSize='MNRAS_column',
                        doOnly1dPlot=True)
```



1.3 Example 2: Making a GTC/triangle plot with Planck and WMAP data!

1.3.1 Download the data

The full set of chains from the Planck 2015 release is available at http://pla.esac.esa.int/pla/#cosmology. You will want to download `COM_CosmoParams_fullGrid_R2.00.tar.gz`. Careful, that's a huge file to download (3.6 GB)!

Extract everything into a directory, cd into that directory, and run this notebook.

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina' # For mac users with Retina display

import numpy as np
from matplotlib import pyplot as plt
import pygtc
```

1.3.2 Read in and format the data

```
WMAP, Planck = [],[]
for i in range(1,5):
    WMAP.append(np.loadtxt('./base/WMAP/base_WMAP_'+str(i)+'.txt'))
    Planck.append(np.loadtxt('./base/plikHM_TT_lowTEB/base_plikHM_TT_lowTEB_'+str(i)+
↪'.txt'))
```

```
# Copy all four chains into a single array
WMAPall = np.concatenate((WMAP[0],WMAP[1],WMAP[2],WMAP[3]))
Planckall = np.concatenate((Planck[0],Planck[1],Planck[2],Planck[3]))
```

1.3.3 Select the parameters and make labels

In the chain directories, there are `.paramnames` files that allow you to find the parameters you are interested in.

```
WMAPplot = WMAPall[:,[2,3,4,5,6,7,9,15]]
Planckplot = Planckall[:,[2,3,4,5,6,7,23,29]]
```

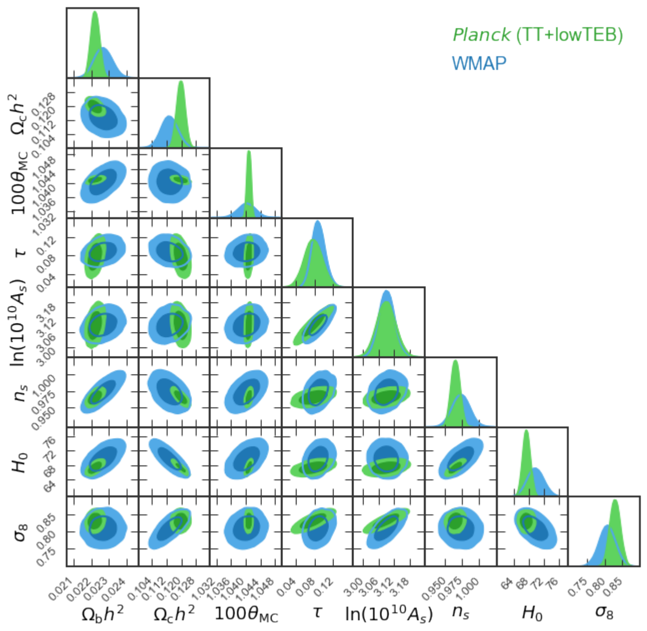
```
# Labels, pyGTC supports Tex enclosed in $..$
params = ('$\Omega_\mathrm{b}h^2$',
          '$\Omega_\mathrm{c}h^2$',
          '$100\\theta_\mathrm{MC}$',
          '$\\tau$',
          '$\ln(10^{10}A_s)$',
          '$n_s$','$H_0$',
          '$\\sigma_8$')

chainLabels = ('$Planck$ (TT+lowTEB)','WMAP')
```

1.3.4 Make the GTC!

Produce the plot and save it as `Planck-vs-WMAP.pdf`.

```
GTC = pygtc.plotGTC(chains=[Planckplot,WMAPplot],
                    weights=[Planckall[:,0],
                    WMAPall[:,0]],
                    paramNames=params,
                    chainLabels=chainLabels,
                    colorsOrder=('greens','blues'),
                    figureSize='APJ_page',
                    plotName='Planck-vs-WMAP.pdf')
```



1.4 API

`pygtc` only has one user-facing function `pygtc.plotGTC()`, which generates the GTC/triangle plot. There are some helper functions that generate the individual panels in the main GTC, but `plotGTC` has options to access these, so they are only minimally documented here.

1.4.1 plotGTC()

`pygtc.`**`plotGTC`**(*chains*, ***kwargs*)

Make a great looking Giant Triangle Confusogram (GTC) with one line of code! A GTC is a lot like a triangle (or corner) plot, but you get to put as many sets of data, and overlay as many truths as you like. That's what can make it so *confusing*!

Parameters **`chains`** (*`array-like[nSamples,nDims] or a`*) – list[[nSamples1,nDims], [nSamples2,nDims], ...] All chains (where a chain is [nSamples,nDims]) in the list must have the same number of dimensions. Note: If you are using `emcee` (http://dan.iel.fm/emcee/current/) - and you should! - each element of chains is an `EnsembleSampler.flatchain` object.

Keyword Arguments

- **`weights`** (*`array-like[nSamples] or a list[[nSamples1], ..]`*) – Weights for the sample points. The number of 1d arrays passed must correspond to the number of *chains*, and each *weights* array must have the same length nSamples as its corresponding chain.
- **`chainLabels`** (*`array-like[nChains]`*) – A list of text labels describing each chain passed to chains. len(chainLabels) must equal len(chains). chainLabels supports LaTex commands enclosed in $..$. Additionally, you can pass None as a label. Default is `None`.
- **`paramNames`** (*`list-like[nDims]`*) – A list of text labels describing each dimension of chains. len(paramNames) must equal nDims=chains[0].shape[1]. paramNames supports LaTex commands enclosed in $..$. Additionally, you can pass None as a label. Default is None, however if you pass a `pandas.DataFrame` object, *paramNames* defaults to the `DataFrame` column names.
- **`truths`** (*`list-like[nDims] or [[nDims], ..]`*) – A list of parameter values, one for each parameter in *chains* to highlight in the GTC parameter space, or a list of lists of values to highlight in the parameter space. For each set of truths passed to *truths*, there must be a value corresponding to every dimension in *chains*, although any value may be `None`. Default is `None`.
- **`truthLabels`** (*`list-like[nTruths]`*) – A list of labels, one for each list passed to truths. truthLabels supports LaTex commands enclosed in $..$. Additionally, you can pass `None` as a label. Default is `None`.
- **`truthColors`** (*`list-like[nTruths]`*) – User-defined colors for the truth lines, must be one per set of truths passed to *truths*. Default color is gray `#4d4d4d` for up to three lines.
- **`truthLineStyles`** (*`list-like[nTruths]`*) – User-defined line styles for the truth lines, must be one per set of truths passed to *truths*. Default line styles are `['--',':','dashdot']`.
- **`priors`** (*`list of tuples [(mu1, sigma1), ..]`*) – Each tuple describes a Gaussian to be plotted over that parameter's histogram. The number of priors must equal the number of dimensions in *chains*. Default is `None`.

- **plotName** (*string*) – A path to save the GTC to in pdf form. Default is `None`.
- **nContourLevels** (*int*) – The number of contour levels to plot in the 2d histograms. May be 1, 2, or 3. Default is 2.
- **sigmaContourLevels** (*bool*) – Whether you want 2d "sigma" contour levels (39%, 86%, 99%) instead of the standard contour levels (68%, 95%, 99%). Default is `False`.
- **nBins** (*int*) – An integer describing the number of bins used to compute the histograms. Default is 30.
- **smoothingKernel** (*float*) – Size of the Gaussian smoothing kernel in bins. Default is 1. Set to 0 for no smoothing.
- **filledPlots** (*bool*) – Whether you want the 2d contours and the 1d histograms to be filled. Default is `True`.
- **plotDensity** (*bool*) – Whether you want to see the 2d density of points. Default is `False`.
- **figureSize** (*float or string*) – A number in inches describing the length = width of the GTC, or a string indicating a predefined journal setting and whether the figure will span one column or the full page width. Default is 70/dpi where `dpi = plt.rcParams['figure.dpi']`. Options to choose from are `'APJ_column'`, `'APJ_page'`, `'MNRAS_column'`, `'MNRAS_page'`, `'AandA_column'`, `'AandA_page'`.
- **panelSpacing** (*string*) – Options are `'loose'` or `'tight'`. Determines whether there is some space between the subplots of the GTC or not. Default is `'tight'`.
- **legendMarker** (*string*) – Options are `'All'`, `'None'`, `'Auto'`. `'All'` and `'None'` force-show or force-hide all label markers. `'Auto'` shows label markers if two or more truths are plotted.
- **paramRanges** (*list of tuples [nDim]*) – Set the boundaries of each parameter range. Must provide a tuple for each dimension of *chains*. If `None` is provided for a parameter, the range defaults to the width of the histogram.
- **labelRotation** (*tuple [2]*) – Rotate the tick labels by 45 degrees for less overlap. Sets the x- and y-axis separately. Options are `(True,True)`, `(True,False)`, `(False,True)`, `(False,False)`, `None`. Using `None` sets to default `(True,True)`.
- **tickShifts** (*tuple [2]*) – Shift the x/y tick labels horizontally/vertically by a fraction of the tick spacing. Example tickShifts = (0.1, 0.05) shifts the x-tick labels right by ten percent of the tick spacing and shifts the y-tick labels up by five percent of the tick spacing. Default is (0.1, 0.1). If tick rotation is turned off for either axis, then the corresponding shift is set to zero.
- **colorsOrder** (*list-like[nDims]*) – The color order for chains passed to *chains*. Default is `['blues', 'oranges','greens', 'reds', 'purples', 'browns', 'pinks', 'grays', 'yellows', 'cyans']`. Currently, `pygtc` is limited to these color values, so you can reorder them, but can't yet define your own colors. If you really love the old colors, you can get at them by calling: `['blues_old', 'greens_old', ...]`.
- **do1dPlots** (*bool*) – Whether or not 1d histrograms are plotted on the diagonal. Default is `True`.
- **doOnly1dPlot** (*bool*) – Plot only ONE 1d histogram. If this is True, then chains must have shape `(samples,1)`. Default is `False`.

- **mathTextFontSet** (*string*) – Set font family for rendering LaTex. Default is `'stixsans'`. Set to `None` to use the default setting in your matplotlib rc. See Notes for known issues regarding this keyword.
- **customLabelFont** (`matplotlib.fontdict`) – Full customization of label fonts. See matplotlib for full documentation. Default is `{'family':'Arial', 'size':9}`.
- **customLegendFont** (`matplotlib.fontdict`) – Full customization of legend fonts. See matplotlib for full documentation. Default is `{'family':'Arial', 'size':9}`.
- **customTickFont** (`matplotlib.fontdict`) – Full customization of tick label fonts. See matplotlib for full documentation. Default is `{'family':'Arial', 'size':6}`. Attempting to set the color will result in an error.
- **holdRC** (*bool*) – Whether or not to reset rcParams back to default. You may wish to set this to `True` if you are working in interactive mode (ie with IPython or in a JuPyter notebook) and you want the plots that display to be identical to the plots that save in the pdf. See Notes below for more information. Default is `False`.

Returns fig – You can do all sorts of fun things with this in terms of customization after it gets returned. If you are using a `JuPyter` notebook with inline plotting enabled, you should assign a variable to catch the return or else the figure will plot twice.

Return type `matplotlib.figure` object

Note: If you are calling `plotGTC` from within an interactive python session (ie via IPython or in a JuPyter notebook), the label font in the saved pdf may differ from the plot that appears when calling `matplotlib.pyplot.show()`.

This will happen if the mathTextFontSet keyword sets a value that is different than the one stored in `rcParams['mathtext.fontset']` and you are using equations in your labels by enclosing them in $..$. The output pdf will display correctly, but the interactive plot will use whatever is stored in the rcParams default to render the text that is inside the $..$. Unfortunately, this is an oversight in matplotlib's design, which only allows one global location for specifying this setting. As a workaround, you can set `holdRC = True` when calling `plotGTC` and it will *not* reset your rcParams back to their default state. Thus, when the figure renders in interactive mode, it will match the saved pdf. If you wish to reset your rcParams back to default at any point, you can call `matplotlib.rcdefaults()`. However, if you are in a jupyter notebook and have set `%matplotlib inline`, then calling `matplotlib.rcdefaults()` may not set things back the way they were, but rerunning the line magic will.

This is all due to a bug in matplotlib that is slated to be fixed in the upcoming 2.0 release.

1.4.2 Helper functions

__plot2d()

`pygtc.pygtc.`**`__plot2d`**`()`

Plot a 2D histogram in a an axis object and return the axis with plot.

Parameters

- **ax** (*matplotlib.pyplot.axis*) – The axis on which to plot the 2D histogram
- **nChains** (*int*) – The number of chains to plot.
- **chains2d** (*list-like*) – A list of pairs of sample points in the form: [[chain1_x, chain1_y], [chain2_x, chain2_y], . . .].

- **weights** (*list-like*) – Weights for the chains2d.
- **nBins** (*int*) – Number of bins (per side) for the 2d histogram.
- **smoothingKernel** (*int*) – Size of the Gaussian smoothing kernel in bins. Set to 0 for no smoothing.
- **filledPlots** (*bool*) – Just contours, or filled contours?
- **colors** (*list-like*) – List of *nChains* tuples. Each tuple must have at least nContourLevels colors.
- **nContourLevels** (*int {1,2,3}*) – How many contour levels?
- **confLevels** (*list-like*) – List of at least *nContourLevels* values for contour levels.
- **truths2d** (*list-like*) – A list of nChains tuples of the form: [(truth1_x, truth1_y), etc…].
- **truthColors** (*list-like*) – A list of colors for the truths.
- **truthLineStyles** (*list-like*) – A list of matplotlib linestyle descriptors, one for each truth.
- **plotDensity** (*bool*) – Whether to show points density in addition to contours.
- **myColorMap** (*list-like*) – A list of *nChains* matplotlib colormap specifiers, or actual colormaps.

Note: You should really just call this from the plotGTC function unless you have a strong need to work only with an axis instead of a figure…

__plot1d()

`pygtc.pygtc.`**`__plot1d`**`()`

Plot the 1d histogram and optional prior.

Parameters

- **ax** (*matplotlib.pyplot.axis*) – Axis on which to plot the histogram(s)
- **nChains** (*int*) – How many chains are you passing?
- **chains1d** (*list-like*) – A list of *nChains* 1d chains: [chain1, chain2, etc…]
- **weights** (*list-like*) – A list of *nChains* weights.
- **nBins** (*int*) – How many histogram bins?
- **smoothingKernel** (*int*) – Number of bins to smooth over, 0 for no smoothing.
- **filledPlots** (*bool*) – Want the area under the curve filled in?
- **colors** (*list-like*) – List of *nChains* tuples. Each tuple must have at least two colors.
- **truths1d** (*list-like*) – List of truths to overplot on the histogram.
- **truthColors** (*list-like*) – One color for each truth.
- **truthLineStyles** (*list-like*) – One matplotlib linestyle specifier per truth.
- **prior1d** (*tuple*) – Normal distribution paramters (mu, sigma)
- **priorColor** (*color*) – The color to plot the prior.

Note: You should really just call this from the plotGTC function unless you have a strong need to work only with an axis instead of a figure…

CHAPTER 2

Citation

If you use pygtc to generate plots for a publication, please cite as:

```
@article{Bocquet2016,
  doi = {10.21105/joss.00046},
  url = {http://dx.doi.org/10.21105/joss.00046},
  year  = {2016},
  month = {oct},
  publisher = {The Open Journal},
  volume = {1},
  number = {6},
  author = {Sebastian Bocquet and Faustin W. Carter},
  title = {pygtc: beautiful parameter covariance plots (aka. Giant Triangle␣
↪Confusograms)},
  journal = {The Journal of Open Source Software}
}
```

CHAPTER 3

Contributions, problems, questions, recommendations

Please report any problems or pitch new ideas on GitHub where pygtc is being developed. Bugs will be squashed ASAP and feature requests will be seriously considered.

Index