# pyScannerBit

***Release 0.0.26***

**Ben Farmer, Greg Martinez**

**Mar 19, 2020**

# CONTENTS


| | | |
|---|---|---|
| 1 | Installation | 3 |
| 2 | Quick Start | 5 |
| 3 | Low-level API | 9 |
| 4 | Test run of all scanners | 11 |
| 5 | Indices and tables | 15 |

pyScannerBit is a Python interface to the GAMBIT library of scanning algorithms, called ScannerBit

It is still in an "alpha" stage of development, but if you are brave then you are encouraged to give it a try! Please report any bugs/problems you encounter on github.

For installation instructions see *Installation*, and see the *Quick Start* guide to get your first scan up and running!

# CHAPTER ONE

# INSTALLATION

PyPI installation:

```
pip install pyscannerbit
```

From github:

```
git clone https://github.com/bjfar/pyscannerbit.git
pip install ./pyscannerbit
```

Either way it is recommended to use pip to install the package since this should generally be compatible with anaconda environments. ScannerBit will built from source, so don't worry if it takes a couple of minutes. If there is an error during the build then there is a good chance that some non-python dependency was missing, so check out the CMake output and see if you can figure out what that might have been. Feel free to ask for help on our github page!

# CHAPTER TWO

# QUICK START

Import the module:

```
import pyscannerbit.scan as sb

# And some other stuff for use in this example
import numpy as np
import math
import matplotlib.pyplot as plt
```

Define the function to be scanned. Make sure the first argument is called `scan`, as a reference to a special interface object will be passed into this argument. The rest of the arguments should be the parameters you wish to scan:

```
# Test function
def rastrigin(scan,x,y,z):
    X = [x,y,z]
    A = 10
    scan.print("x+y+z",x+y+x) # Send extra results to output file.
    return - (A + sum([(x**2 - A * np.cos(2 * math.pi * x)) for x in X]))
```

Internally, scans are conducted inside a unit hypercube with a uniform prior weighting. This needs to be "stretched" into the parameter space you want to scan via a "prior transformation" function, in a fashion analogous to inverse transform sampling. Below a simple uniform prior over the range [-4,4] is defined for each parameter:

```
# Prior transformation from unit hypercube
def prior(vec, map):
    map["x"] = -4 + 8*vec[0] # flat prior over [-4,4]
    map["y"] = -4 + 8*vec[1]
    map["z"] = -4 + 8*vec[2]
```

Here `vec` will contain a sample from the unit hypercube (the input of the transform) at each iteration of the scan, and `map` should be filled with the transformed parameter values, which will in turn be passed on to the matching arguments of the target function.

Next, set some options for your preferred scanning algorithm, to be passed as a dictionary:

```
twalk_options = {"sqrtR": 1.05}
```

Now create and run your scan!:

```
myscan = sb.Scan(rastrigin, prior_func=prior, scanner="twalk", scanner_options=twalk_
↪options)
myscan.scan()
```

By default results will be output to a HDF5 file called `results.hdf5`, located in the directory `pyscannerbit_run_data/samples` relative to where you launched your driver script. Analyse these with whatever tools you like! You will probably need at least the `h5py` package to read the data back into numpy arrays for analysis.

If you just want to plot something quickly, we provide a few small helper tools to retrieve results and make simple plots. For example for a simple scatter plot:

```
results = myscan.get_hdf5()
results.make_plot("x", "y") # Simple scatter plot of samples
```

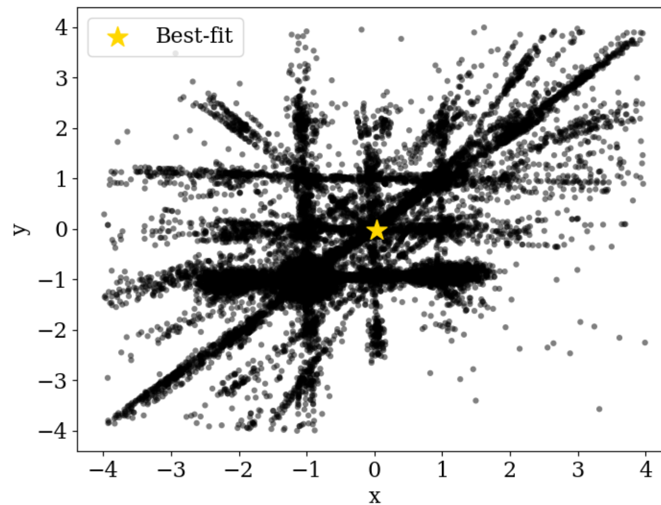


Fig. 1: Simple 2D scatter plot of 3D T-Walk scan results.

Or for a binned profile likelihood plot with 68%/95% asympotic confidence regions:

```
fig = plt.figure()
ax = fig.add_subplot(111)
results.plot_profile_likelihood(ax,"x","y") # Profile likelihood
fig.savefig("x_y_prof_like.png")
```

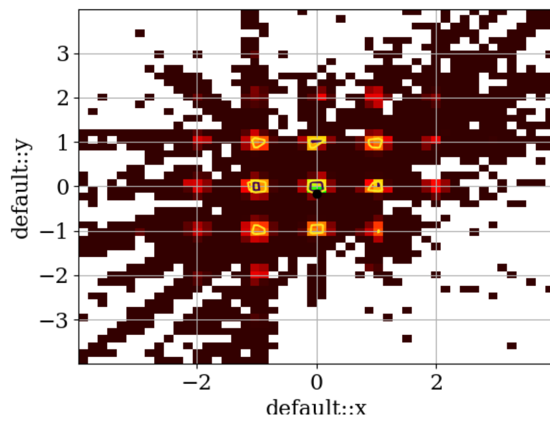


Fig. 2: Profile likelihood plot of rastrigin test function, scanned quickly by Twalk

# CHAPTER THREE

# LOW-LEVEL API

For user-friendliness we recommend using the high-level API described in *Quick Start* and most of the rest of these docs. However, if for some reason you don't want to use this API, then you have the option of digging deeper, into a lower level API that is exposed by ScannerBit. Below is a short description of how this API can be used.

First, you need to set some flags for `dlopen` to help than scanner plugin libraries be dynamically loaded correctly (this is done automatically in the high-level API):

```
import sys
import ctypes
flags = sys.getdlopenflags()
sys.setdlopenflags(flags | ctypes.RTLD_GLOBAL)
```

Next import the 'bare-bones' interface from a few levels down in the package:

```
# Dig past the extra python wrapping to the direct ScannerBit.so shared library␣
↪interface level
from pyscannerbit.ScannerBit.python import ScannerBit
```

Define the log-likelihood function you wish to scan, and (if desired) a prior transformation function:

```
def loglike(m):
    a = m["model1::x"]
    ScannerBit.print("my_param", 0.5) # Send extra data to the output file at each␣
↪point
    return -a*a/2.0

def prior(vec, map):
    # tell ScannerBit that the hypergrid dimension is 1
    ScannerBit.ensure_size(vec, 1) # this needs to be the first line!
    map["model1::x"] = 5.0 - 10.0*vec[0]
```

These look superficially similar to the functions that should be supplied to the high-level API, however please note that there is no sanity/error checking in this low-level API, so mistakes will result in cryptic errors from inside ScannerBit. The model name in e.g. `model1::x` is also non-optional in this interface, and similarly you must call the `ensure_size` function in the prior function to tell ScannerBit the dimension of your parameter space.

Next generate a settings dictionary. The structure of this should match the YAML format required by ScannerBit when run via GAMBIT (this is one thing that we will simplify in the nice wrapper. . . .):

```
settings = {
"Parameters": {
  "model1": {
    "x": None,
    }
```

(continues on next page)

(continued from previous page)

```
  },
"Priors": {
  "x_prior": {
    "prior_type": 'flat',
    "parameters": ['model1::x'],
    "range": [1.0, 40.0],
    }
  },
"Printer": {
  "printer": "hdf5",
  "options": {
    "output_file": "results.hdf5",
    "group": "/",
    "delete_file_on_restart": "true",
    }
  },
"Scanner": {
  "scanners": {
    "twalk": {
      "plugin": "twalk",
      "like": "LogLike",
      "tolerance": 1.003,
      "kwalk_ratio": 0.9,
      "projection_dimension": 4
      }
    },
  },
"KeyValues": {
  "default_output_path": "pyscannerbit_run_data/",
  "likelihood": {
    "model_invalid_for_lnlike_below": -1e6
    }
  }
}
```

Run your scan!:

```
myscan = ScannerBit.scan(True)
myscan.run(inifile=settings, lnlike={"LogLike": like}, prior=prior, restart=True)
```

If all went well, your scan should begin, and generate HDF5 format output in `pyscannerbit_run_data/samples/results.hdf5`.

# CHAPTER FOUR

# TEST RUN OF ALL SCANNERS

The following is a complete script to test-run all the scanning algorithms wrapped by pyScannerBit. It can be found in the git repo at `tests/test_all_scanners.py`. NOTE: external scanners are current non-functional so they are left out of this test:

```
"""Demo script which runs all the (serious) scanners to which pyScannerBit has access"
↪""

import numpy as np
import math
import pyscannerbit.scan as sb
import matplotlib.pyplot as plt

from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()

# Regenerate scan data?
new_scans = True

# Test function
def rastrigin(scan,x,y,z):
    X = [x,y,z]
    A = 10
    return - (A + sum([(x**2 - A * np.cos(2 * math.pi * x)) for x in X]))

# Prior transformation from unit hypercube
def prior(vec, map):
    map["x"] = -4 + 8*vec[0] # flat prior over [-4,4]
    map["y"] = -4 + 8*vec[1]
    map["z"] = -4 + 8*vec[2]

# Settings for quick and dirty scans. Won't do very well, because the test function is
# actually rather tough!
# Don't have to specify all scanner options; anything missing will revert to defaults␣
↪(see defaults.py)
scanner_options = {}
scanner_options["multinest"] = {"tol": 0.5, "nlive": 100}
scanner_options["polychord"] = {"tol": 1.0, "nlive": 20}
scanner_options["diver"]     = {"convthresh": 1e-2, "NP": 300}
scanner_options["twalk"]     = {"sqrtR": 1.05}
scanner_options["random"]    = {"point_number": 10000}
scanner_options["toy_mcmc"]  = {"point_number": 10} # Acceptance ratio is really bad␣
↪with this scanner, so don't ask for much
scanner_options["badass"  ]  = {"points": 1000, "jumps": 10}
```

(continues on next page)

(continued from previous page)

```
scanner_options["pso"]       = {"NP": 400}

scanners = ["twalk","badass","pso"]
colors = ["r","b","g"]
if size is 1:
    scanners += ["random","toy_mcmc"] # "random" and "toy_mcmc" are not MPI␣
↪compatible.
    colors += ["c","y"]

Nscans = len(scanners)
results = {}

# Do all scans
for s in scanners:
    # Create scan manager object
    myscan = sb.Scan(rastrigin, prior_func=prior, scanner=s, scanner_options=scanner_
↪options[s])
    if new_scans:
        print("Running scan with {}".format(s))
        myscan.scan()
    else:
        print("Retrieving results from previous {} scan".format(s))
    results[s] = myscan.get_hdf5()

# Plot results
# Only want to do this on one process
if rank is 0:
    fig = plt.figure(figsize=(4*Nscans,8))
    for i,(s,c) in enumerate(zip(scanners,colors)):
        x,y = results[s].get_params(["x","y"])
        ax = fig.add_subplot(2,Nscans,i+1)
        ax.set_title("{0} (N={1})".format(s,len(x)))
        ax.scatter(x,y,c=c,label=s,s=0.5)
        ax = fig.add_subplot(2,Nscans,i+1+Nscans)
        results[s].plot_profile_likelihood(ax,"x","y")

    ax.legend(loc=1, frameon=True, framealpha=1, prop={'size':10})
    plt.tight_layout()
    fig.savefig("test_all_scanners.png")
```

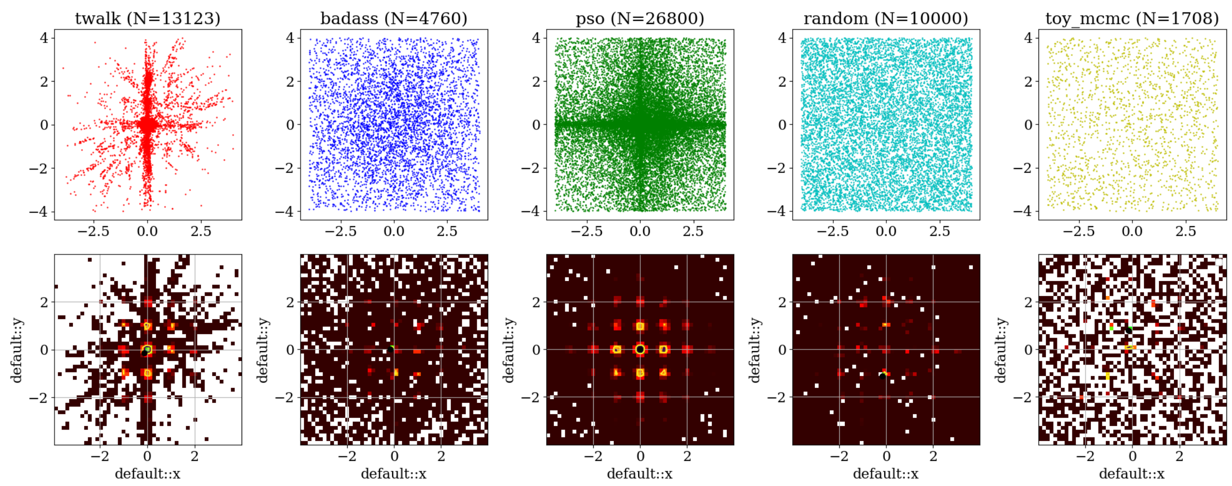


Fig. 1: Profile likelihood and scatter plots of rastrigin test function, scanned quickly by all available algorithms

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search