Integrate library UNU.RAN into scipy.stats

Self-Introduction

I am Tirth (@tirthasheshpatel on GitHub), a third-year computer science undergrad student at Nirma University. I am quite familiar with Cython (e.g. I have used Cython to create N-Body simulations and completed OReilly's "*Cython: A Guide for Python Programmers*" book) and a lot of my college courses make use of C (e.g. '*Data Structures and Algorithms*' and '*Graph Theory*'). I have completed courses on statistics such as '*Probability and Statistics*' and '*Bayesian Statistics*'.

Open Source work: I have participated in GSoC with the PyMC team last year. Please find my work here: https://summerofcode.withgoogle.com/projects/6135416450711552
I have been a contributor to SciPy since May 2020 and a maintainer since March 2021. My PRs: https://github.com/scipy/scipy/pulls/tirthasheshpatel

Contents

1 Project Abstract

UNU.RAN is a performant C library with methods to sample from continuous, discrete, multivariate, and empirical distributions. It has been used by CERN in its ROOT project and an R interface called Runuran also exists. It will be a very nice addition to scipy.stats. The default method used by SciPy to

sample from any distribution requires integrating the PDF and then numerically inverting the CDF. The implementation in SciPy is too slow to be relied on for practical purposes and custom methods for sampling random variates need to be implemented for the distributions in SciPy. In contrast UNU.RAN offers several methods (e.g. TDR, PINV) that offer very good performance to sample from wide classes of distributions.

The goal of this project is to provide an object-oriented interface for methods present in UNU.RAN to sample from univariate continuous and discrete distributions using NumPy's BitGenerator as the Uniform Random Number Sampler (URNG). My proposal also includes writing a comprehensive test and documentation suite with tutorials. If time permits, I also propose to write benchmarks for the added methods.

2 Coding Proposal: Design

2.1 UNU.RAN Library

UNU.RAN is a C library for generating non-uniform pseudorandom variates from continuous, discrete, multivariate, empirical and matrix distributions. There are several methods for generating random numbers. The choice depends upon the

information one has about the distribution. For example, if we desire to generate samples from a continuous distribution and we have its PDF and its derivative w.r.t the random variate, we can use the Transformed Density Rejection method provided by UNU.RAN.

SciPy currently only has a pure python implementation of the Ratio-of-Uniforms (RoU) method to sample from a continuous distribution.

2.2 UNU.RAN Source Tree

UNU.RAN is modularized into 9 sub-directories:

- **`distr`** : Contains structures for representing different types of distributions (e.g. continuous, discrete, matrix) in C.
- **`distributions`** : Implements several probability distributions like the normal and beta distributions.
- **`methods`** : Contains methods (like TDR, DAU, PINV) for sampling from different types of distributions.
- **`parser`** : UNU.RAN provides a "String API" which is a higher level API for building distributions and sampling using string expressions. This directory contains a parser to parse such strings and create a distribution/sampler.
- **`specfunct`** : Contains some special mathematical functions like area under a gaussian curve.

- **`tests`** : Contains functions for convergence checks, error checks and other statistical tests.
- **`uniform`** : Contains wrappers for third party C libraries like GSL, RngStreams, ect that provide uniform random number generators.
- **`urng`** : Contains structures to represent a uniform random number generator used by sampling methods.
- **`utils`** : Utilities used throughout the library.

As discussed in the mailing lists, we don't have the permission to use default URNGs provided in the `uniform` sub-directory. This sub-directory can be removed before building. The `parser` sub-directory can also be removed as the String API doesn't offer all the functionality present in UNU.RAN and probably won't be used to wrap any functionality. Other sub-directories contain core structures and algorithms that need to be included in SciPy. Appendix B lists a pre-processing step that is required to build UNU.RAN with SciPy's build system.

2.3 Interfacing UNU.RAN

Once unwanted directories are removed and remaining files are preprocessed, a Cython wrapper can be written to provide an interface to the library. The following steps are involved to sample from any distribution using a method in UNU.RAN:

1. Create a new distribution.

2. Set the required attributes (e.g. PDF, CDF, etc) of the distribution needed by the method.
3. Create a "parameter" object using the distribution. (the parameter object is a representation of a method in UNU.RAN. So, changing the attributes of the parameter means changing the attributes of the method itself)
4. Change the default attributes of the parameter object if provided by the user.
5. Initialize a generator using the parameter object.
6. Start sampling from the generator. If the user hasn't provided a URNG (an instance of NumPy's BitGenerator), use the default URNG of the parameter object.

My proposal is to use an object-oriented interface for such a workflow. The first five steps are only setup and don't need to run every time one needs to sample from the distribution. Hence, they can be performed in the `__cinit__` method itself. A method can be provided to sample from the distribution. An example of such a API is shown below:

```
cdef class TDR:
    def __cinit__(self, pdf, dpdf, params/args, ..., c=-0.5, variant='ps',
...):
        cdef unur_distr *distr = unur_cont_new()
        # set the attributes of the distribution.
        ...
        # initialize a parameter object
        cdef unur_distr *par = unur_tdr_new(distr)
        # change the default attributes of the parameter
        # object `par` if provided by the user.
```

```
        ...
        # create a generator object.
        cdef unur_gen *rng = unur_init(par)

    def sample(self, size=1, random_state=None):
        # set the random state if given.
        cdef unur_urng *urng = self._get_urng(random_state)
        unur_chg_urng(rng, urng)
        # sample from the distribution.
        ...
```

Initially, parameters to set the most important attributes of the parameter object can be added. Another option is to add a parameter for each option that can be set. The latter might be a little superfluous and would need a lot of documentation and tests. Hence, the former is better initially and once it is usable new parameters, documentation, and tests can be added on top of it easily. The API might also need changes based on user feedback and consensus of the developers. We can iterate on top of this design to incorporate new ideas.

Moreover, we can add getter (property) methods to get the value of the attributes of the parameter object. Also, as SciPy provides `rv_discrete` and `rv_continuous` classes to create custom distributions, a class method that accepts instances of such distributions to initialize a generator can be created:

```
cdef class TDR:
    def __cinit__(self, ...): ...

    def sample(self, ...): ...
```

```
@classmethod
def from_scipy(cls, dist, dpdf, ...):
    # extract the pdf of the distribution
    pdf = dist.pdf
    # extract the domain
    domain = dist.a, dist.b
    # now use the cls.__cinit__ method to initialize
    # the generator.
    cls.__cinit__(pdf, dpdf, params, domain, ...)
```

Another thing that the API needs to handle is error codes thrown by UNU.RAN in case of failure. UNU.RAN throws certain codes to indicate a failure. For example, in case a value out of domain is passed to one of the methods of the distribution, UNU.RAN throws a UNUR_ERR_DISTR_DOMAIN error (which is an error code that expands to 20). Corresponding to each error code is a small description that can be fetched using `unur_get_stderror` call.

This is very helpful for debugging and reporting informative errors to the user directly from Python and can be added easily.

2.4 Building SciPy with UNU.RAN

The library along with its Cython wrapper can be built using NumPy distutils. A rudimentary setup file can be found under the Prototype Section of this proposal. Macros that UNU.RAN expects at compile time can be set using the `config.h.in` file. Some of these macros check for the existence of standard

headers like `limits.h` and `stdlib.h`. This can be achieved via Python using `scipy._build_utils.compiler_helper.try_compile` in the pre build hook:

```
def pre_build_hook(build_ext, ext):
    from scipy._build_utils.compiler_helper import try_compile
    has_stdlib = try_compile(cc, code='#include <stdlib.h>\n'
                                      'int main(int argc, char **argv) {}')
    if has_stdlib:
        ext.define_macros.append(('HAVE_STDLIB_H', '1'))
```

All the sources and headers can be fetched from the source tree of UNU.RAN and used to compile the Cython wrapper.

3 Coding Proposal: Prototype

A working prototype of the proposed API can be found here: https://github.com/tirthasheshpatel/scipy/tree/unuran/scipy/stats/_unuran

It contains a very basic documentation and test suite. Please see Appendix A for all the limitations of the prototype that are yet to be addressed.

It builds on Ubuntu 20.04 and provides both functional and class-based API. Both the APIs have been tested and documented. Please find the build logs here:

https://github.com/tirthasheshpatel/scipy/pull/5/checks?check_run_id=2308954489

3.1 Functional API

A functional API for TDR and DAU methods for sampling from continuous and discrete distributions respectively has been added. Usage:

```
>>> from scipy.stats._unuran import tdr
>>> rvs = tdr(lambda x: 1 - x*x, lambda x: -2*x, params=(),
...       domain=(-1, 1), size=10000)
>>> import matploblib.pyplot as plt
>>> pdf = lambda x: 1 - x*x
>>> x = np.linspace(-1, 1, 1000)
>>> px = pdf(x)
>>> plt.plot(x, px)
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.show()
```

The user is able to sample from the distribution but not change the attributes of the parameter object. This is one of the big limitations of a functional API that a class-based API addresses.

3.2 Class Based API

A class based API is more flexible with an additional benefit of one time setup cost only. Hardly losing any performance, it allows the user to set the attributes of a parameter object and

sample from the distribution multiple times without running the setup every time. Example:

```
>>> from scipy.stats._unuran import TDR
>>> rng = TDR(lambda x: 1 - x*x, lambda x: -2*x, params=(),
...       domain=(-1, 1), c=0., cpoints=10, variant='ia')
>>> rvs = rng.sample(size=100_000)
>>> import matploblib.pyplot as plt
>>> pdf = lambda x: 1 - x*x
>>> x = np.linspace(-1, 1, 1000)
>>> px = pdf(x)
>>> plt.plot(x, px)
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.show()
```

Variant and the number of construction points to use can be passed as parameters followed by a call to the `sample` method to sample from the distribution. This is a simple use case of such an API. Including more methods to get and set more parameters and even change them between calls to the sample method will be useful.

4 Coding Proposal: Timeline

4.1 Milestones

Milestone 1: UNU.RAN has been added as a submodule to SciPy. Deprecated and unnecessary files (e.g src/uniform

directory) have been removed. A pull request has been filed. SciPy builds on all the platforms.

Milestone 2: An object-oriented interface for methods to sample from univariate continuous and discrete distributions has been written. Basic functionality has been provided (e.g. methods to let the user change the most important parameters of the parameter object). A basic documentation and a test suite has been added.

Milestone 3: A more comprehensive documentation and test suite has been added. Tutorials have been written. Docs and tutorials build. Tests pass. An early merge to test the new features out on master. Reserve some time for bug reports and follow up PRs. If time permits, add a benchmark suite. At the end, the API should be usable and we should be able to test the performance of the interface.

4.2 Timeline

Community Bonding Period (May 17, 2021 - June 7, 2021): Get to know the mentors. Polish the prototype. Start to work on a pull request. Try to reach Milestone 1.

Phase 1 (June 7, 2021 - July 16, 2021): Start to wrap the proposed methods from UNU.RAN. Reach Milestone 2 or complete most of the tasks listed in Milestone 2.

Phase 2 (July 16, 2021 - August 16, 2021): Start writing tutorials. Address documentation failures, if any. Write more tests. Reach Milestone 3. If the CI is happy and there is a consensus to merge, merge the PR early. If time permits, write a benchmark suite for the new API. Reserve the last couple of weeks and post GSoC period for iterating on the proposed design.

4.3 Reserved time

Starting from May 17, 2021, I have my semester end examinations which would span a week or two (end date hasn't been provided). My contribution during that time will be fairly low. The timeline has been created accordingly.

Conclusion

Addition of UNU.RAN's methods to SciPy will be a valuable enhancement. I propose to add a class-based API to SciPy that enables users to access powerful universal sampling methods from UNU.RAN. A working prototype is also available that will

serve as the starting point for my work. Thanks to Christoph for helping me figure out the object-oriented design of the API.

Appendix A Limitations of the Prototype

Here are some of the limitations of the prototype that I aim to address during the coding period:

- Only builds on Ubuntu 20.04
- Unstructured code: The code is not structured properly. For example, the declarations which should ideally be present in a separate `.pxd` file hasn't been considered. There is also a lot of repeating code which could be converted into functions.
- Doesn't remove all the unwanted files/directories: Most of the files and directories under the UNU.RAN source tree have been kept intact.
- Doesn't check for UNU.RAN error codes: A minimal amount of error handling has been done.
- Doesn't check for existence of standard libraries: UNU.RAN checks for certain standard libraries before building. This behaviour needs to be replicated.

Appendix B The `.ch` Convention and Deprecated files

UNU.RAN uses a `.ch` file convention. These files contain implementations of routines whose declarations are present in a different `.c` file where the needed `.ch` file is `#include`d at the end of the declarations. This is an abuse of the `#include ` directive and needs to be addressed.

One of the possible solutions is to rename the `.ch` files to `.h` files and change the `#include ` statements to include the `.h` file instead. This simple approach should work as these `.ch` files are not included anywhere else nor are they included multiple times in the same file.

We also don’t want to build the deprecated files. Fortunately, all the deprecated files are named `deprecated_<file_name>.c` and can be removed easily before building.