Paganini

Release 1.1.1

Dec 05, 2019

Contents:

Paganini is a lightweight python library for tuning multiparametric combinatorial specifications.

Given a combinatorial specification, expressed using a domain-specific language closely resembling Flajolet and Sedgewick's *symbolic method*, Paganini gives its users some additional control over the distribution of structures constructed using the designed samplers.

CHAPTER 1

Installation

Paganini is available as a *Python* package for both *Python2* and *Python3*.

Warning: We strongly recommend using *Python3*, and we do not guarantee the required decimal precision for *Python2*. Moreover, the support for Python2 drops in 2020, and as a consequence, some of the packages that we use are not anymore maintained, and the usage is at your own risk. In particular, two of our tests fail on Python2 for the reasons of numerical precision.

Note: We assume that the user is familiar with *Python* and its package manager *pip*. If you are new to *Python*, please visit the official installation webpage https://www.python.org/downloads/. For new versions of *Python*, *pip* is already pre-installed. If you don't have it, check https://pip.pypa.io/en/stable/installing/

The latest release of *Paganini* can be installed using *pip*:

```
>>> pip install paganini
```

Tip: If you want to update to the recent version of paganini, use

pip3 install –upgrade paganini

You may also want to upgrade *pip* so that the installation works properly

pip3 install –upgrade pip

1.1 Installation from sources

In order to install from sources, you need *git*, or you can download and install the code manually from *github*.

```
git clone git://github.com/maciej-bendkowski/paganini.git
cd paganini
python3 setup.py install
```

1.2 Testing

In order to verify that paganini works, run the following in the command line

```
python3 -m paganini.tests
```

You can get more examles in the tests folder

```
>>> import paganini
>>> help(paganini.tests)
```

CHAPTER 2

Tutorial

Tip: Interactive environments like `jupyter notebook` are extremely helpful in code testing and experimenting. Check them out!

Note: Throughout the tutorial, it is assumed that at the beginning of the session, all the contents of the package *Paganini* have been imported:

```
>>> from paganini import *
```

Alternatively, in order to avoid polluting the global namespace, a synonym import can be used. In this case, all the functions should be referenced as sub-items of this namespace

```
>>> import paganini as pg
>>> spec = pg.Specification()
```

2.1 Introduction

Consider the following example. Suppose that we are interested in designing an sampler for plane trees of unbounded degree (i.e. with an arbitrary number of children), specified as

```
T = Z SEQ(T)
```

where `SEQ(T)` stands for a (possibly empty) sequence of trees and `Z` marks the size of a node. In Paganini, we write the following snippet defining the same combinatorial class:

```
>>> spec = Specification()
>>> z, T = Variable(), Variable()
>>> spec.add(T, z * Seq(T))
```

Now, if we we want to construct a corresponding sampler, say *analytic (or Boltzmann) sampler*, we have to find a specific value of `Z` and use it to compute branching probabilities governing the random choices of our sampler (essentially the number of children for each of the constructed nodes). What value of `Z` should be choose if we are interested in large, uniform, and unbounded in size trees? With Paganini, this task amounts to invoking

```
>>> spec.run_singular_tuner(z)
```

... and that's it! Paganini determines the corresponding value of z for us. Once *tuned*, variables are decorated with appropriate numerical values:

```
>>> z.value
0.25
>>> T.value
0.5
```

Paganini allows its users to focus on the design of specifications, taking care of the rest.

2.2 Target expectation tuning

With the help of Paganini, users can demand the computation of tuning variables with specific, finite target expectations. Suppose that we are interested in designing an analytic sampler for Motzkin trees (i.e. plane unary-binary trees) however we would also like to demand that the outcome trees consists of around 1000 nodes, among which around 200 are unary. To achieve this goal, we construct the following specification:

```
>>> from paganini import *
>>> spec = Specification()
>>> z, u, M = Variable(1000), Variable(200), Variable()
>>> spec.add(M, z + u * z * M + z * M ** 2)
>>> spec.run_tuner(M)
```

Here `z` and `u` are two *marking variables* standing for the tree size and the number of unary nodes, respectively. Once we run the tuner, all three variables are decorated with respective numerical values, which the user can then use to compute respective branching probabilities. A sampler designed with such values is *guaranteed* to output Motzkin trees for which the expected size and mean number of unary nodes obey the design specification.

2.3 Examples

Paganini is under constant development, supporting a growing class of so-called admissible constructors. Below you can find a handful of examples supported by Paganini. For more specifications, please visit our *tests* folder.

Polya trees

The specification is `T = Z * MSET(T)`.

```
>>> spec = Specification()
>>> z, T = Variable(), Variable()
>>> spec.add(T, z * MSet(T))
>>> spec.run_singular_tuner(z)
>>> z.value
```

(continues on next page)

(continued from previous page)

```
0.338322112871298
>>> T.value
1.0
```

Bounded (unlabelled) cyclic compositions

A non-recursive specification `C = CYC_{= 12}(Z * SEQ(Z))` with mean size around 20.

```
>>> spec = Specification()
>>> z, C = Variable(20), Variable()
>>> spec.add(C, UCyc(z * Seq(z), eq(12)))
>>> spec.run_tuner(z)
>>> z.value
0.405765659263783
```

Cayley trees with finite expected size.

The specification is `T = Z * SET(T)`.

```
>>> spec = Specification()
>>> z, T = Variable(1024), Variable()
>>> spec.add(T, z * Set(K))
>>> spec.run_tuner(T)
>>> z.value
0.367879265638609
```

CHAPTER 3

Paganini's API

3.1 Module contents

3.1.1 Paganini

Paganini is a lightweight python library for tuning of multiparametric combinatorial systems.

All the necessary documentation can be found on-line on https://paganini.readthedocs.io/

Use

```
>>> help(paganini.tutorial)
```

to see some examples of code usage.

class paganini.expressions.**Expr**(*coeff=1*, *variables={}*)

Bases: `object`

Algebraic expressions (multivariate monomials) in form of $cx_1^{k_1}x_2^{k_2}\cdots x_m^{k_m}$.

static cast(*other*)

Casts its input to an expression.

is_constant

True iff the expression represents a constant.

related(*other*)

Checks if the two expressions are related, i.e. have the same exponents and perpahs a different coefficient.

class paganini.expressions.**VariableType**

Bases: `enum.Enum`

An enumeration.

PLAIN = 1

TYPE = 2

class paganini.expressions.**Variable**(*tuning_param=None*)
Bases: *paganini.expressions.Expr*

Symbolic variables.

is_type_variable
True iff the variable represents a type variable. In other words, if it admits a defining equation.

set_expectation(*tuning_param*)

class paganini.expressions.**Polynomial**(*expressions*)
Bases: object

Polynomials of multivariate algebraic expressions.

static cast(*other*)
Casts its input to a polynomial.

is_one()
Checks if the polynomial represents a constant one.

static simplify(*polynomial*)
Simplifies the given polynomial.

specification(*no_variables*)
Composes a sparse matrix specification of the polynomial. Requires as input a number dictating the number of columns of the constructed matrix (usually the number of variables in the corresponding optimisation problem).

Its output is a tuple consisting of:

(1) a sparse matrix representing the polynomial,

(2) a vector of logarithms of monomial coefficients,

(3) a (collective) constant term representing constant monomials.

The matrix represents expoenents of respective variables.

static sum(*series*)
Evaluates the sum of the given series.

class paganini.specification.**Seq**(*expression*, *constraint=None*)
Bases: *paganini.expressions.Variable*

Sequence variables.

register(*spec*)
Unfolds the Seq definition and registers it in the given system.

class paganini.specification.**UCyc**(*expression*, *constraint=None*)
Bases: *paganini.expressions.Variable*

Unlabelled Cyc variables.

register(*spec*)
Unfolds the UCyc definition and registers it in the given system.

class paganini.specification.**MSet**(*expression*, *constraint=None*)
Bases: *paganini.expressions.Variable*

MSet variables.

register(*spec*)
Unfolds the MSet definition and registers it in the given system.

class paganini.specification.**Set**(*expression*, *constraint=None*)
Bases: *paganini.expressions.Variable*

Labelled Set variables.

register(*spec*)
Unfolds the Set definition and registers it in the given system.

class paganini.specification.**Cyc**(*expression*, *constraint=None*)
Bases: *paganini.expressions.Variable*

Labelled Cyc variables.

register(*spec*)
Unfolds the Cyc definition and registers it in the given system.

class paganini.specification.**Operator**
Bases: enum.Enum

Enumeration of supported constraint signs.

EQ = 2

GEQ = 3

LEQ = 1

UNBOUNDED = 4

class paganini.specification.**Constraint**(*operator*, *value*)
Bases: object

Supported constraints for classes such as SEQ.

static normalise(*constraint=None*)

class paganini.specification.**Type**
Bases: enum.Enum

Enumeration of supported system types.

ALGEBRAIC = 1

RATIONAL = 2

class paganini.specification.**Params**(*sys_type*)
Bases: object

CVXPY solver parameters initalised with some defaults.

class paganini.specification.**Specification**(*series_truncate=20*)
Bases: object

Symbolic system specifications.

add(*var*, *expression*)
Includes the given definition in the specification.

check_type()
Checks if the system is algebraic or rational. Note: the current method is heuristic.

discharged_variables
Number of variables discharged in the system.

run_singular_tuner(*z*, *params=None*)

Given a (size) variable and a set of tuning parameters, composes an optimisation problem corresponding to an approximate sampler meant for structures of the given type. Variables are tuned so to achieve (in expectation) the marked variable frequencies.

Consider the following example:

```
>>> sp = Specification()
>>> z, u, M = Variable(), Variable(0.4), Variable()
>>> sp.add(M, z + u * z * M + z * M **2)
>>>
>>> params = Params(Type.ALGEBRAIC)
>>> sp.run_singular_tuner(z, params)
```

Here, the variable u is marked with a *frequency* 0.4. The type M represents the type of Motzkin trees, i.e. unary-binary plane trees. Variable z marks their size, whereas u marks the occurrences of unary nodes. The tuning goal is to obtain specific values of z, u, and M, such that the induced branching probabilities lead to a sampler which generates, in expectation, Motzkin trees of infinite (i.e. unbounded) size and around 40% of unary nodes.

Respective variables (including type variables) are decorated with a proper 'value'. The method returns the CVXPY solution (i.e. the optimal value for the problem, or a string indicating why the problem could not be solved).

run_tuner(*t*, *params=None*)

Given the type variable and a set of tuning parameters, composes a (tuning) optimisation problem corresponding to an approximate sampler meant for structures of the given type. Variables are tuned so to achieve (in expectation) the marked variable values.

Consider the following example:

```
>>> sp = Specification()
>>> z, u, M = Variable(1000), Variable(200), Variable()
>>> sp.add(M, z + u * z * M + z * M **2)
>>> params = Params(Type.ALGEBRAIC)
>>> sp.run_tuner(M, params)
```

Here, the variables z and u are marked with *absolute* values 1000 and 200, respectively. The input type represents the type of Motzkin trees, i.e. plane unary-binary trees. Variable z marks their size, whereas u marks the occurrences of unary nodes. The tuning goal is to obtain specific values of z, u, and M, such that the induced branching probabilities lead to a sampler which generates Motzkin trees of size 1000 with around 200 unary nodes (both in expectation).

Respective variables (including type variables) are decorated with a proper 'value'. The method returns the CVXPY solution (i.e. the optimal value for the problem, or a string indicating why the problem could not be solved).

paganini.specification.**leq**(*n*)

Creates a less or equal constraint for the given input.

paganini.specification.**geq**(*n*)

Creates a greater or equal constraint for the given input.

paganini.specification.**eq**(*n*)

Creates an equal constraint for the given input.

CHAPTER 4

Citing Paganini

If you use *Paganini* or its components for published work, we encourage you to cite the accompanying paper:

Maciej Bendkowski, Olivier Bodini, Sergey Dovgal Polynomial tuning of multiparametric combinatorial samplers

You can import the following BibTeX citation:

```
@inproceedings{paganini,
  title        = {Polynomial tuning of multiparametric combinatorial samplers},
  author       = {Bendkowski, Maciej and Bodini, Olivier and Dovgal, Sergey},
  booktitle    = {2018 Proceedings of the Fifteenth Workshop on Analytic Algorithmics␣
↪and Combinatorics (ANALCO)},
  pages        = {92--106},
  year         = {2018},
  organization = {SIAM}
}
```

CHAPTER 5

References

Paganini relies on published work of numerous excellent authors. Below, you can find a short (and definitely inexhaustive) list of papers on the subject:

- P. Flajolet, R. Sedgewick: Analytic Combinatorics
- P. Duchon, P. Flajolet, G. Louchard. G. Schaeffer: Boltzmann Samplers for the random generation of combinatorial structures
- C. Pivoteau, B. Salvy, M. Soria: Algorithms for Combinatorial Systems: Well-Founded Systems and Newton Iterations
- O.Bodini, J. Lumbroso, N. Rolin: Analytic samplers and the combinatorial rejection method
- S. Diamond and S. Boyd: CVXPY: A Python-Embedded Modeling Language for Convex Optimization

If you are interested in the practical design of analytic samplers, we encourage you to check out the related Boltzmann Brain software.

CHAPTER 6

About

6.1 Maciej Bendkowski

Theoretical Computer Science Department,
Jagiellonian University in Kraków, Poland.

6.2 Sergey Dovgal

Université Paris 13,
Laboratoire d'Informatique de Paris Nord.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

T

U

V