

PyRigi—A General-Purpose Python Package for the Rigidity and Flexibility of Bar-and-Joint Frameworks

MATTHIAS ADRIAN-HIMMELMANN, Institute for Computational and Experimental Research in Mathematics, Brown University, Providence, Rhode Island, USA and Institut für Analysis und Algebra, Technische Universität Braunschweig, Braunschweig, Germany

MATTEO GALLET, Department of Mathematics, Informatics and Geosciences, University of Trieste, Trieste, Italy

GEORG GASEGGER, Johann Radon Institute for Computational and Applied Mathematics (RICAM), Austrian Academy of Sciences, Linz, Austria

JAN LEGERSKÝ, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic

We present PyRigi, a novel PYTHON package designed to study the rigidity properties of graphs and frameworks. Among many other capabilities, PyRigi can determine whether a graph admits only finitely many ways, up to isometries, of placing the vertices in the plane so that the corresponding edge lengths are not altered, it can check whether the graph satisfies such a property even after the removal of any of its edges, and whether it has a unique vertex placement. By implementing algorithms from the scientific literature, PyRigi enables the exploration of rigidity properties of structures that would be out of reach for computations by hand. With reliable and robust algorithms, as well as clear, well-documented methods that are closely connected to the underlying mathematical definitions and results, PyRigi aims to be a practical and powerful general-purpose tool for the working mathematician interested in rigidity theory. PyRigi is open source and easy to use, and awaits researchers to benefit from its computational potential.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Theory of computation** → **Computational geometry**; • **Mathematics of computing** → **Graph theory**;



This research was funded in part by the Austrian Science Fund (FWF) 10.55776/I6233. For open access purposes, the authors have applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. M. Adrian-Himmelman acknowledges the support by the National Science Foundation under Grant No. DMS-1929284 while the author was in residence at the Institute for Computational and Experimental Research in Mathematics in Providence, RI, during the Geometry of Materials, Packings and Rigid Frameworks semester program. J. Legerský was supported by the Czech Science Foundation

(Grantová agentura České republiky), project No. 22-04381L. We would like to thank Zulip (<https://zulip.com>) for the Zulip Cloud Standard plan provided free of charge.

Authors' Contact Information: Matthias Adrian-Himmelman, Institute for Computational and Experimental Research in Mathematics, Brown University, Providence, Rhode Island, USA and Institut für Analysis und Algebra, Technische Universität Braunschweig, Braunschweig, Germany; e-mail: matthias.himmelman@tu-braunschweig.de; Matteo Gallet, Department of Mathematics, Informatics and Geosciences, University of Trieste, Trieste, Italy; e-mail: matteo.gallet@units.it; Georg Grasegger, Johann Radon Institute for Computational and Applied Mathematics (RICAM), Austrian Academy of Sciences, Linz, Austria; e-mail: georg.grasegger@ricam.oeaw.ac.at; Jan Legerský (corresponding author), Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic; e-mail: jan.legersky@fit.cvut.cz.



This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).

ACM 1557-7295/2026/9-ART14

<https://doi.org/10.1145/3815171>

Additional Key Words and Phrases: rigidity theory, frameworks, software package, symbolic and numerical computations, graph methods, continuous motions

ACM Reference format:

Matthias Adrian-Himmelfmann, Matteo Gallet, Georg Grasegger, and Jan Legerský. 2026. PyRigi—A General-Purpose Python Package for the Rigidity and Flexibility of Bar-and-Joint Frameworks. *ACM Trans. Math. Softw.* 52, 3, Article 14 (September 2026), 22 pages.
<https://doi.org/10.1145/3815171>

1 Introduction

This article serves as an introduction to PyRIGI, a general-purpose PYTHON package. It aims to help researchers in rigidity and flexibility theory in their day-to-day work by implementing various algorithmic and combinatorial aspects of the mathematical theory, allowing users to perform computations that would be infeasible to carry out by hand. This document presents an overview of PyRIGI’s functionality, design philosophy, and implementation details.

Rigidity theory is the study of graph-theoretic, matroid-theoretic, geometric, and metric properties related to the various forms of structural rigidity. Concretely, one starts from a simple, loopless graph and assigns to each of its vertices a point in $\mathbb{R}^d$, namely a *realization*; in this way, a *framework* is obtained. Then, one can ask whether it is possible to assign other realizations to the same graph so that the Euclidean distance between points corresponding to an edge of the graph is the same as in the original realization. Clearly, this is always possible by applying an isometry, i.e., a rototranslation, to the original realization. If this is the only possible way to obtain such “compatible” realizations in a sufficiently small neighborhood, then the framework is called *rigid*; it is called *flexible* otherwise. In addition to this first notion of rigidity, many other can be introduced: for example, if a framework is rigid, does there exist a unique realization (up to isometries) with given distances between points connected by an edge, or are there finitely many of them? In the first case, the framework is called *globally rigid*. If the framework is rigid and removing any edge preserves this property, then it is called *redundantly rigid*. All these notions can be transferred from frameworks to abstract graphs by considering what the behavior of a “generic” framework is.

Rigidity theory sits at the interface between graph theory, matroid theory, and metric geometry, and benefits from contributions from algebraic and tropical geometry, functional analysis, semi-definite programming, and symbolic computation. In addition to its theoretical interest, rigidity theory plays a role in applications, such as protein and molecule analysis [Hermans et al., 2017; Thorpe and Duxbury, 2002], formation control [Zelazo and Zhao, 2019], matrix completion [Singer and Cucuringu, 2010], foldable structures [Tachi, 2016], chemistry, and modern art [Graver, 2001].

Rigidity theory is currently undergoing an exciting period of growth. For example, the average number of yearly entries tagged with MSC code 52C25 “Rigidity and flexibility of structures (aspects of discrete geometry)” on the *zbMATH Open* abstracting and reviewing service¹ increased from 28.4 in the 5-year period 2015–2019 to 50.6 in 2019–2024. Moreover, several major international scientific gatherings focusing on rigidity theory and related topics have been organized in recent years. These include the “Thematic Program on Geometric Constraint Systems, Framework Rigidity, and Distance Geometry” (2021) and the subsequent “Focus Program on Geometric Constraint Systems” (2023), both hosted by the Fields Institute for Research in Mathematical Sciences (Toronto, Canada), the “Special Semester on Rigidity and Flexibility” (2024) at RICAM (Linz, Austria) and the program “Geometry of Materials, Packings, and Rigid Frameworks” (2025) at ICERM (Providence, USA).

¹<https://zbmath.org>.

Parallel to the scientific activity, the last 15 years have seen the emergence of a wide range of software tools devoted to rigidity theory. Their primary scope can be broadly categorized into:

Applications: *FIRST/MSU ProFlex* [Jacobs et al., 2001] and *KINARI* [Fox and Streinu, 2012; Fox et al., 2011] are tools for the rigidity analysis of proteins.

Special-Purpose Software Packages: An overview of the functionality of the following packages is described in greater detail in Table 1. Some of these packages were developed primarily for a rigidity analysis. Among them, the graph-rigidity-checker [Dewar, 2021] and RIGIDPY [Hagh and Sadjadi, 2022] are tools built in PYTHON for checking the rigidity of graphs and frameworks, respectively. RIGICOMP [Grasegger, 2022] is a MATHEMATICA package for testing rigidity and for computing the number of realizations for minimally rigid graphs and GPART [Zhang et al., 2023] is a MATLAB package that computes partitions into globally/redundantly rigid components. Finally, Larsson [2024] provides a NAUTY plugin for enumerating (k, ℓ) -sparse and -tight graphs. Other packages focus on flexibility and motions. FLEXRiLoG [Grasegger and Legerský, 2020] is a SAGEMATH package for producing NAC-colorings and continuous motions, while DEFORMATIONPATHS [Himmelmann, 2026] is a JULIA package for approximating continuous motions of geometric constraint systems.

Supplementary Material for Research Articles: Bartzos et al. [2020], Capco et al. [2018a], Clarke et al. [2026], Gallet et al. [2022], and Grasegger et al. [2022] provide algorithms for bounding or counting the realization number of minimally rigid graphs in Euclidean space and on the sphere, Mc Glue and Stokes [2022] enumerate computationally rigid circuits and Assur graphs up to a fixed size, CAYMos [Wang and Sitharam, 2015] is a JAVA package for the parametrization of continuous motions for certain frameworks, and EASAL [Ozkan et al., 2018] is a software for describing the structure of the configuration space of frameworks.

Smaller Software Projects: Anand [2024] investigates the existence of circle packings whose contact graph is a given minimally rigid graph, Dance [2024] tests the rigidity properties of graphs, Islam [2024] generates continuous motions for graphs with a single nontrivial degree of freedom, McKenzie [2020] provides miscellaneous code snippets for the rigidity and flexibility of frameworks, and Poetzsch [2024] gives a method for finding the maximum rigid components of a graph.

Despite the growing interest in developing software tools for rigidity theory, it seemed to us that there is a lack of general-purpose software to support researchers in their day-to-day work. This lack of basic functionality in previously existing software can, for instance, be seen in Table 1. A general-purpose software package should allow testing for various rigidity properties, be able to manipulate graphs with rigidity-oriented operations such as k -extensions or coning, provide functionality for constructing continuous motions, and include databases of graphs with interesting rigidity properties.

This motivation led to the workshop “Code of Rigidity” during the 2024 Special Semester on Rigidity and Flexibility in Linz, which was dedicated to presenting and spreading knowledge about software in rigidity theory. A survey was prepared and circulated before the workshop among possibly interested researchers about their needs and wishes regarding software for rigidity theory, and the answers were the starting point for the workshop discussions. The (approximately 25) workshop participants agreed on the importance and usefulness of starting to develop a general-purpose software platform for rigidity theory and set associated goals. Via rounds of consultations, it was decided that the software should be standalone (and not embedded as a package of a wider already existing project) and developed in PYTHON (which had the broadest base of potential users and contributors). The name PYRIGI was selected from a shortlist of proposals through a vote by

Table 1. Comparison of Rigidity Software Packages and Their Functionality

	Infinitesimal Rigidity	2nd-order Rigidity	Motions	Generic Rigidity	Redundant Rigidity	Global Rigidity	Graph Properties	Visualization
DEFORMATIONPATHS	✓ ⁿ	✓	✓	✗	✗	✗	✗	✓
FLEXRiLoG	✗	✗	✓	(✓)	✗	✗	*	✓
GPART	✗	✗	✗	✗	✓	✓	✗	✗
graph-rigidity-checker	✗	✗	✗	✓	✗	✓	✗	✓ ⁺
RIGICOMP	✓ ^s	✗	✗	✓	✓	✓	*	✓
RIGIDPY	✓ ⁿ	(✓)	✗	✗	✗	✗	✗	✓
PyRIGI	✓ ^{s+n}	✓	✓	✓	✓	✓	*	✓ ⁺

Major functionality is indicated by ✓, while ✗ means no or just very little. Some packages check the infinitesimal rigidity of frameworks by using exact coordinates ✓^s, whereas others compute the rank of the rigidity matrix using numerical means ✓ⁿ. While the package RIGIDPY does not technically allow for checking second-order rigidity, it provides functionality that could be extended to such certificates; similarly, FLEXRiLoG provides functionality for checking minimal 2-rigidity (✓). The symbol * denotes that RIGICOMP relies on MATHEMATICA, FLEXRiLoG on SAGEMATH and PyRIGI uses NETWORKX for general graph properties. Many packages let the user visualize frameworks, but only some allow a graphical input option where the user can draw a graph, as indicated by ✓⁺.

the workshop participants as a portmanteau combining “Python” and “Rigidity.” Finally, the four authors of this article volunteered to act as maintainers of the project.

The main goal of PyRIGI is to provide a versatile software package built in PYTHON that integrates mathematical theory with algorithmic implementations. In doing so, it uses the backends NETWORKX, NUMPY, and SYMPY, but also provides native algorithms for checking properties of graphs and for linear-algebraic computations. In contrast, FLEXRiLoG and RIGICOMP are based on the computer algebra systems SAGEMATH and MATHEMATICA, respectively, which lets them benefit from sophisticated algebraic and symbolic computations. Meanwhile, DEFORMATIONPATHS takes advantage of the high-performance numerical linear algebra capabilities of LAPACK within JULIA to approximate continuous motions. However, these dependencies often come at the expense of broader accessibility and user-friendliness.

After a year, PyRIGI has reached its first stable release, and this article is meant to introduce the readers to its features and (current) limitations, and to hopefully convince researchers in rigidity theory or nearby fields to consider adopting it in their work. The main strengths of PyRIGI can be summarized as follows:

- it is *general-purpose*: a user finds a wealth of commands, all in the same software;
- it is *easy to use*: as a PYTHON package, it is easy to install on many platforms (simply by running `pip install pyrigi`), and it provides an intuitive interface, with method names derived from the mathematical concepts they implement;
- it is the result of *community involvement*: as a *GitHub* repository² anyone can contribute, and we provide a public Zulip channel³ to discuss issues, feature requests, and ongoing development; the package originated from a workshop, and several events have already taken place to enhance it;
- it is *well-documented*, including a *mathematical background*: we emphasize detailed interface instructions and link them to the related mathematical concepts;
- it *implements scientific results* from the recent literature.

²<https://github.com/PyRigi/PyRigi>.

³<https://pyrigi.zulipchat.com>.

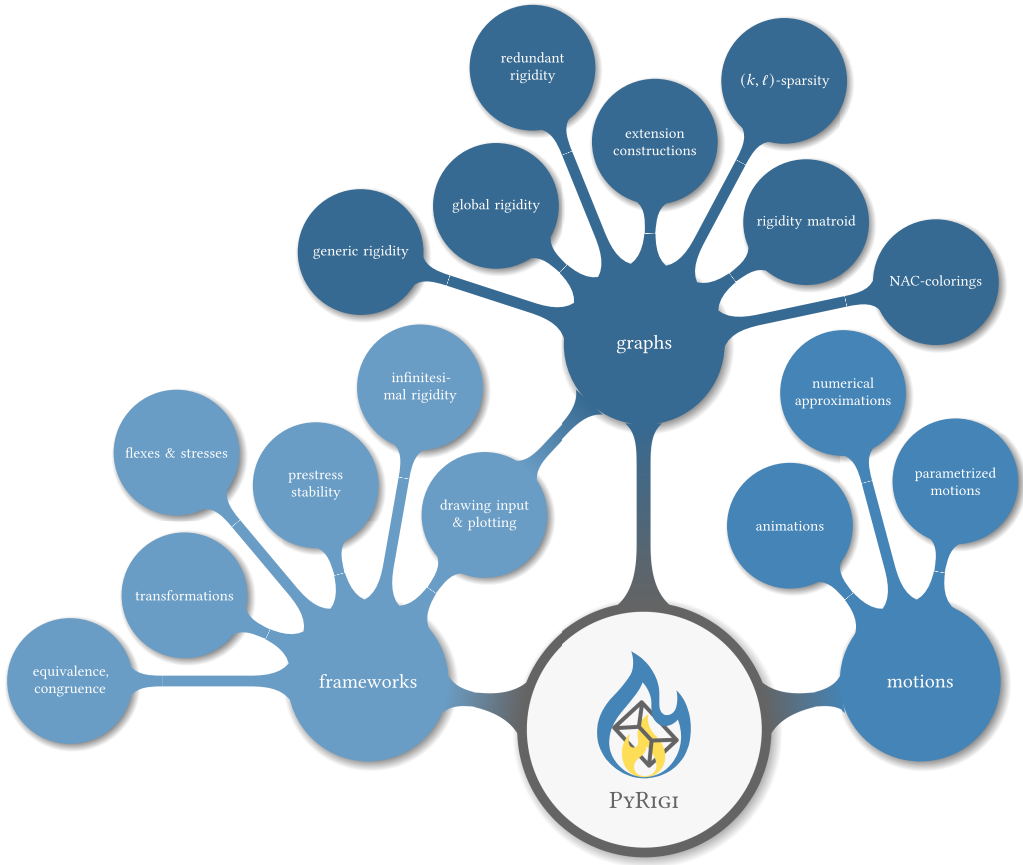


Fig. 1. Overview of PyRigi's functionality.

Figure 1 provides a glimpse of the current capabilities of PyRigi. As one may notice, PyRigi rests on the three pillars of graphs, frameworks, and motions. Each of these pillars corresponds to a PYTHON class that exposes methods which can be used to create and modify graphs and frameworks, both by explicitly entering their vertices, edges, and realizations, and by fetching them from databases of known examples; users are able to call methods to check for infinitesimal and generic rigidity, in addition to their global and redundant counterparts. A non-exhaustive list of possible constructions on graphs includes k -extensions, intersections, sums, and coning. Extensive plotting features are provided, and one can numerically generate motions of flexible frameworks.

PyRigi is designed to be user-friendly, especially for people with little to no experience with symbolic computations and programming: our chosen programming language, PYTHON, is known to be easy to learn also for beginners, and we strive to provide clear and detailed documentation for the introduced objects. In addition, the package documentation is enhanced by the integration of mathematical definitions, which provide precise mathematical meaning to the inputs and outputs of the PYTHON computations.

PyRigi is an open source software package. We believe that the outcomes of research should be publicly and freely accessible, in accordance with the FAIR Principles for Research Software [Barker et al., 2022], that is why all PyRigi's code is accessible via its GitHub page⁴, and the version 1.2

⁴<https://github.com/PyRigi/PyRigi>.

described in this article is stored in Zenodo [The PyRigi Developers, 2026]. Everyone is welcome to join this project. PyRigi is released under an MIT license, which grants several rights to use, modify, and copy the software. We kindly ask users and developers to carefully read the license and be aware of its consequences.

This article serves as an introduction to PyRigi by describing its core functionalities, guiding philosophy, and internal architecture. Section 2 highlights PyRigi’s unique feature of integrating formal mathematical statements with its interface documentation to give a precise meaning to the quantities used in its methods. Section 3 provides a detailed overview of the functionalities implemented in PyRigi. Section 4 sheds light on the implementation details of the main classes and methods in PyRigi and their internal structure. Finally, Section 5 outlines potential directions for future development and extensions of PyRigi.

2 Code Meets Math

A basic part of software documentation is the description of the code and the available functions and methods. PyRigi goes further, adding a second part that integrates the mathematical concepts and theorems of rigidity theory with computational tools. This means that each method and function in PyRigi—unless it is already well known—is linked to its respective mathematical definition. For algorithmic concepts and implementation decisions, there is usually a mathematical theorem and an algorithm description in the documentation. The mathematical documentation collects all necessary information in one place in a handbook-like style organized in topical pages. Definitions and theorems can therefore be referenced both from the code and the documentation. This gives the user a precise understanding of what is computed and how. The interactions between the theoretical and the computational side are visualized in Figures 2 and 3 for a graph and a framework example, respectively. In the following paragraphs, we describe these interactions in more detail. Besides the in-depth documentation, which is intended for more advanced users, there are, of course, also tutorials for getting started.

Let us start by discussing the interplay of mathematical concepts, documentation, and code on an example graph depicted in Figure 2. A crucial definition in rigidity theory is that of a minimally rigid graph. Out of the many possible equivalent definitions, we chose to adopt one related to the rigidity of frameworks (top left of the figure). However, the genericity that is mentioned in this definition is, from a coding point of view, difficult to use directly. The theorem of Pollaczek-Geiringer and Laman (top right of the figure) yields a better strategy for computation. Bibliographic references for the theorem guarantee scientific standards of the mathematical documentation. The theorem links to another concept of graph theory, namely tightness, for which there is (for some important cases) a polynomial-time algorithm. This algorithm is one option to check rigidity in the method `is_min_rigid` as can be seen in the docstring of the method (bottom left of the figure). The docstring also refers to the respective definitions and theorems. Additionally, the docstring shows an explanation of the method and the parameters, and it gives examples for their usage, fulfilling common coding standards, such as PEP 8. All of this is also presented in the online documentation in a well-formatted and accessible manner. Note that in the figure, some parts are left out (indicated by “...”) for space and illustration purposes. The example in the figure takes a predefined graph from the database of graphs `graphDB` that accompanies PyRigi.

The example structure for frameworks (Figure 3) shows two particularly relevant definitions, namely infinitesimal flexes and stresses (top of the figure). Frameworks are implemented in PyRigi as their own class. Two of its methods are shown: `inf_flexes` computes all infinitesimal flexes of a given framework, and `stresses` does so with the equilibrium stresses (center of the figure). References to definitions are considered an important aspect of the documentation in PyRigi, and hence, there is a standard section for them in the docstrings, as can be seen in the figure. The

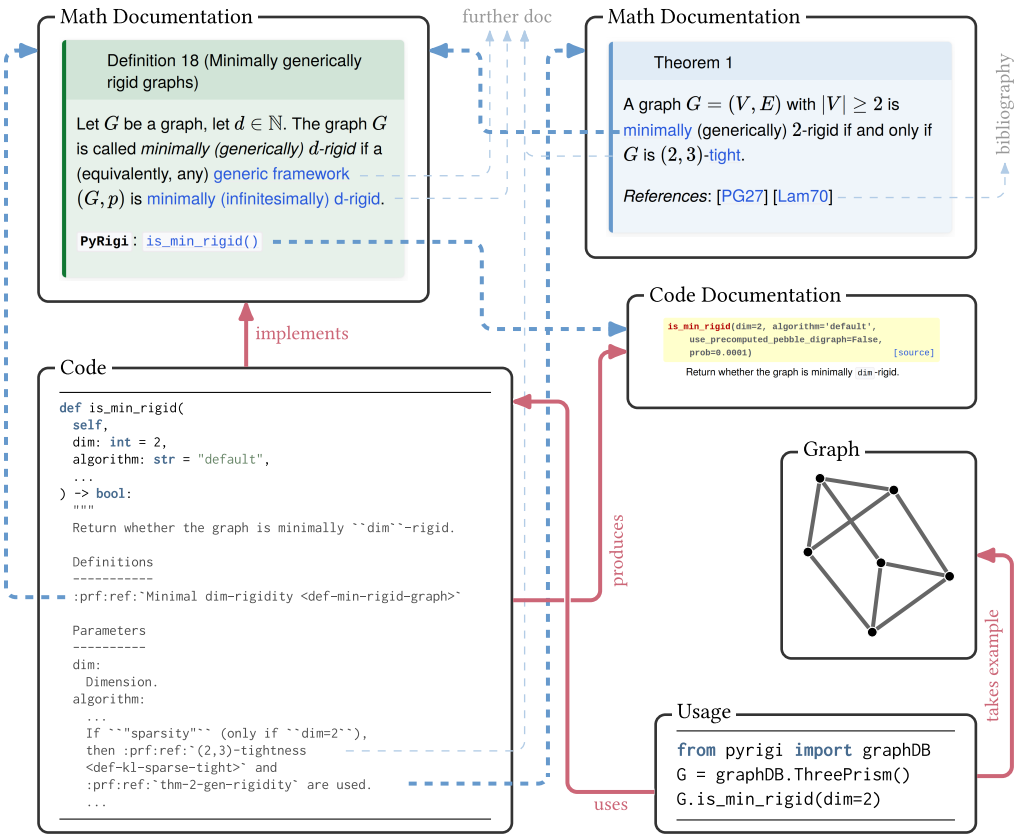


Fig. 2. Example structure of graph rigidity in PyRIGI. The rounded rectangles describe different parts of PyRIGI. Blue dashed arrows $\dashrightarrow$ symbolize references between elements, while red arrows $\rightarrow$ show structural relations.

example is chosen to show that computational results of frameworks can easily be visualized in `PyRIGI`. The input framework is obtained from the framework database `frameworkDB` that is part of `PyRIGI`. The output figure shows both the infinitesimal flexes (as green arrows) as well as the stresses (as labels on the edges).

Besides the documentation on the code and the mathematical background, PyRIGI’s documentation also provides a tutorial on the rigidity of graphs and frameworks.⁵ This tutorial serves as an introduction to the functionality of PyRIGI. It contains a description and the usage of some basic commands, including the elements described in the figures. Again, references to the mathematical definitions and the method descriptions help the user to easily navigate through the documentation. In this article, we offer a brief example in Section 3.4, which outlines the package’s usage.

3 Functionality

In this section, we illustrate the current functionality of `PyRIGI`. We emphasize what `PyRIGI` can do and refer the reader to the online documentation of `PyRIGI` to see how this can be used in practice. We start by describing how frameworks are handled by `PyRIGI`, and then move to graphs, concluding with motions. We chose this order, since many notions related to graph rigidity are

⁵<https://pyrigi.github.io/PyRigi/userguide/tutorials/rigidity.html>.

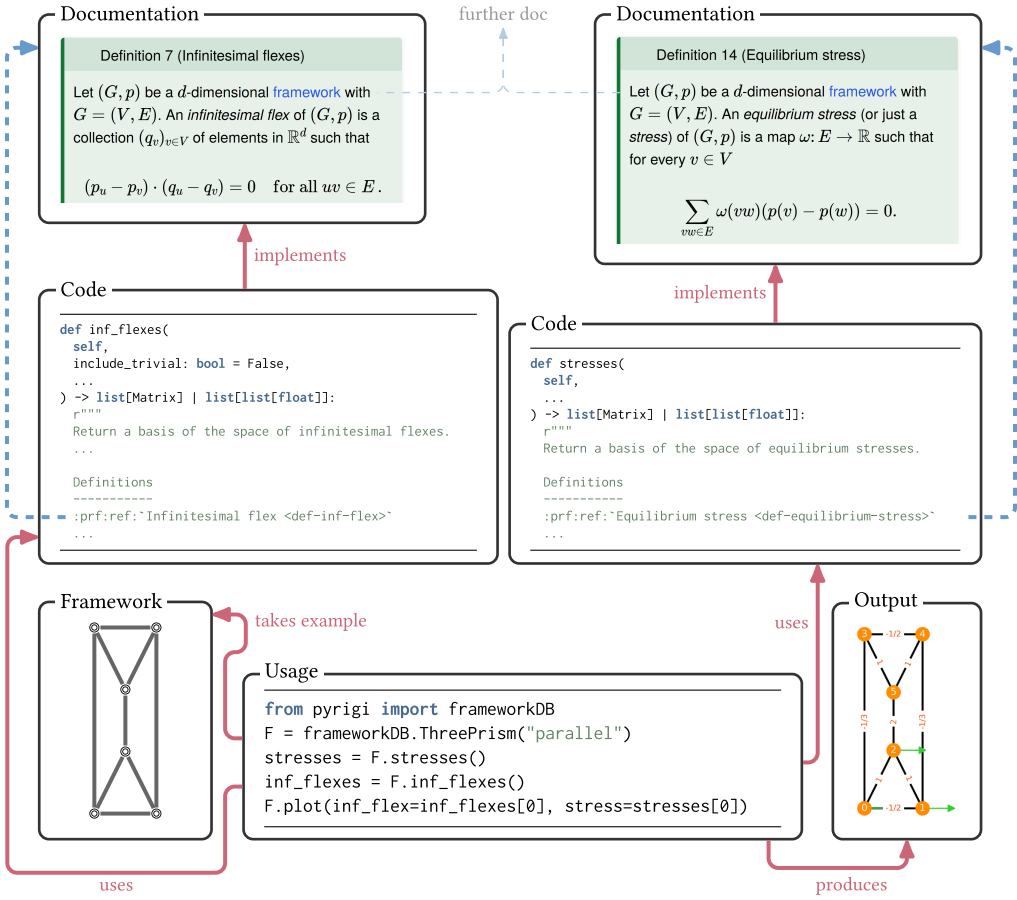


Fig. 3. Framework rigidity in PyRIGI. The rounded rectangles describe different parts of PyRIGI. Arrow styles are as in Figure 2.

first introduced for frameworks and later defined for graphs by considering so-called “generic” realizations.

3.1 Frameworks

Let $G = (V, E)$ be a simple graph, $d \in \mathbb{N}$ and let $p: V \rightarrow \mathbb{R}^d$ be a d -dimensional realization of G . A d -dimensional framework is then defined as the pair (G, p) . In PyRIGI, a framework object can be instantiated using the class `Framework`. It expects a `Graph` and a realization given by a dictionary as input. For instance, a framework on a 4-cycle can be constructed by calling

```
from pyrigi import Framework, Graph
G = Graph([(0,1), (1,2), (2,3), (3,0)])
realization = {0: [0,0], 1: [sqrt(2),0], 2: [1,1], 3: [0,3/4]}
F = Framework(G, realization)
```

The coordinates in the realization can be provided as `SYMPY` expressions using their string representation, so that the computations on the framework can be performed symbolically. Specific

relevant frameworks are stored in the internal frameworkDB. For instance, we can construct a parallel configuration of the 3-prism using the following commands:

```
from pyrigi import frameworkDB
TP = frameworkDB.ThreePrism(realization="parallel")
```

This framework consists of two flipped and translated triangles, with the property that corresponding vertices are connected by edges that lie on three parallel lines. It is depicted in Figure 3. We use the frameworks F and TP as examples throughout this subsection.

3.1.1 Equivalence and Congruence. To distinguish whether a given framework is flexible or rigid and to say that one framework is a deformed version of another framework, we employ a theoretical notion to compare realizations of the same graph up to rigid motions. Two d -dimensional frameworks (G, p) and (G, p') on a graph $G = (V, E)$ are called *equivalent* if

$$\|p(u) - p(v)\| = \|p'(u) - p'(v)\| \quad \text{for all } \{u, v\} \in E.$$

They are called *congruent* if

$$\|p(u) - p(v)\| = \|p'(u) - p'(v)\| \quad \text{for all } u, v \in V.$$

In PyRIGI, these concepts can be checked by calling the methods `is_equivalent` and `is_congruent`. By default, these checks are performed symbolically. When faster computation is required, users may switch to numerical methods by setting the appropriate keyword. This converts the internal coordinates to floating-point numbers and runs numerical checks instead of symbolic ones. The computation uses a predefined numeric tolerance, which also can be specified.

We then say that a d -dimensional framework (G, p) is (*continuously*) *rigid* if every framework (G, q) equivalent to (G, p) , and sufficiently close to it in the Euclidean distance, is actually congruent to it; otherwise, it is called (*continuously*) *flexible*.

3.1.2 Transformations. As we are often interested in comparing properties of different frameworks, PyRIGI comes with various tools for transforming frameworks. First and foremost, we can modify the underlying graph by adding and deleting vertices or edges. We can project a realization to a lower dimension by calling `projected_realization`. In addition, it is possible to rescale all coordinates by a factor, to translate a framework, and to rotate it in $\mathbb{R}^2$ and $\mathbb{R}^3$.

3.1.3 Infinitesimal Rigidity. Determining whether a given framework with arbitrary coordinates is rigid in $\mathbb{R}^d$ is coNP-hard for $d \geq 2$ [Abbott, 2008], meaning that checking whether a framework is flexible is NP-hard. Therefore, stronger conditions are typically employed to determine whether a framework is rigid.

The concept of infinitesimal rigidity provides an important sufficient condition that arises from showing that a framework's realization is a non-singular real isolated solution of the underlying Euclidean distance constraint system. In other words, a d -dimensional framework (G, p) on a graph $G = (V, E)$ is *infinitesimally rigid* if the rank of the Jacobian corresponding to the polynomial system of the associated edge-length equations—also referred to as the *rigidity matrix*—is equal to $d|V| - \binom{d+1}{2}$. This matrix is the $|E| \times d|V|$ matrix whose row labelled by $e = \{u, v\}$ is

$$\begin{pmatrix} 0 & \cdots & 0 & \overset{u}{p(u) - p(v)} & 0 & \cdots & 0 & \overset{v}{p(v) - p(u)} & 0 & \cdots & 0 \end{pmatrix}.$$

It turns out that the previously constructed framework F on the 4-cycle graph is not infinitesimally rigid. When adding the edge $\{1, 3\}$ by calling `F.add_edge([1, 3])`, it becomes a Diamond framework which is *minimally infinitesimally rigid*, meaning that it is infinitesimally rigid and the removal of any edge from the framework's underlying graph causes it to become infinitesimally

flexible. This fact can be verified using the command `F.is_min_inf_rigid()`. If we further add the edge $\{0, 2\}$, the framework F becomes a realization of the complete graph on four vertices K_4 . It is *redundantly infinitesimally rigid*, meaning that the removal of any edge still produces an infinitesimally rigid framework, as can be verified by calling `F.is_redundantly_inf_rigid()`. The keyword `numerical` can be used in all of these cases to speed up the computations by converting the symbolic coordinates to floating-point numbers and computing the rigidity matrix's kernel using NumPy [Harris et al., 2020].

3.1.4 Stresses and Flexes. Conversely, the framework TP on the 3-prism graph is not infinitesimally rigid, as can be checked via `TP.is_inf_rigid()`, which returns `False`.

We can verify this result by computing the framework's *infinitesimal flexes*, which are given by the kernel of the rigidity matrix. In other words, the infinitesimal flexes $(q(v))_{v \in V}$ of framework (G, p) are vectors at each vertex satisfying $(p(u) - p(v)) \cdot (q(u) - q(v)) = 0$ for each edge $\{u, v\} \in E$. They are called *trivial*, if they extend to an ambient isometry, *nontrivial* otherwise. In particular, an infinitesimally rigid framework possesses no nontrivial infinitesimal flexes. For instance, all infinitesimal flexes of the 3-prism framework TP can be computed by calling `TP.inf_flexes(include_trivial=True)`.

An associated concept is *equilibrium stresses*, which form the cokernel of the rigidity matrix. Computing

```
inf_flexes = TP.inf_flexes()
stresses = TP.stresses()
```

demonstrates that the framework has both a 1D space of nontrivial infinitesimal flexes and equilibrium stresses. We can then verify that these flexes and stresses are indeed nontrivial infinitesimal flexes and equilibrium stresses by calling `TP.is_nontrivial_flex(inf_flexes[0])` and `TP.is_stress(stresses[0])`, respectively. These computations can be performed numerically as well.

3.1.5 Prestress Stability and Second-Order Rigidity. Even though the parallel realization of the 3-prism graph is not infinitesimally rigid, the framework is prestress stable. This concept constitutes a sufficient criterion, weaker than infinitesimal rigidity, for the continuous rigidity of a framework. It depends on the idea that stresses may *block* certain infinitesimal flexes from extending to continuous motions. More precisely, if there exists an equilibrium stress ω such that for every nontrivial infinitesimal motion q it holds that

$$\sum_{\{u,v\} \in E} \omega_{\{u,v\}} \cdot \|q(u) - q(v)\|^2 > 0,$$

the framework is called *prestress stable* [Connelly and Whiteley, 1996]. For the 3-prism framework TP , this can be checked by calling `TP.is_prestress_stable()`, which returns `True`, so it is indeed rigid. If instead, for every nontrivial infinitesimal motion q there exists an equilibrium stress ω such that the inequality above holds, we obtain the weaker concept of *second-order rigidity* [Connelly and Whiteley, 1996]. This criterion can be verified via the method `is_second_order_rigid` (the `numerical` keyword can be used here as well). Since prestress stability and second-order rigidity both imply (continuous) rigidity [Connelly and Gortler, 2017], the framework cannot be continuously deformed. This check is based on the sums of nonnegative circuits decomposition [Ilmanen and de Wolff, 2016], which is a general sufficient condition for the nonnegativity (or positivity) of polynomials and is reliant on the corresponding Newton polytope. In our case, it produces an effective criterion for the second-order rigidity of bar-and-joint frameworks, which exploits the particular structure of the problem. In PyRIGI, it is currently required that either the space of

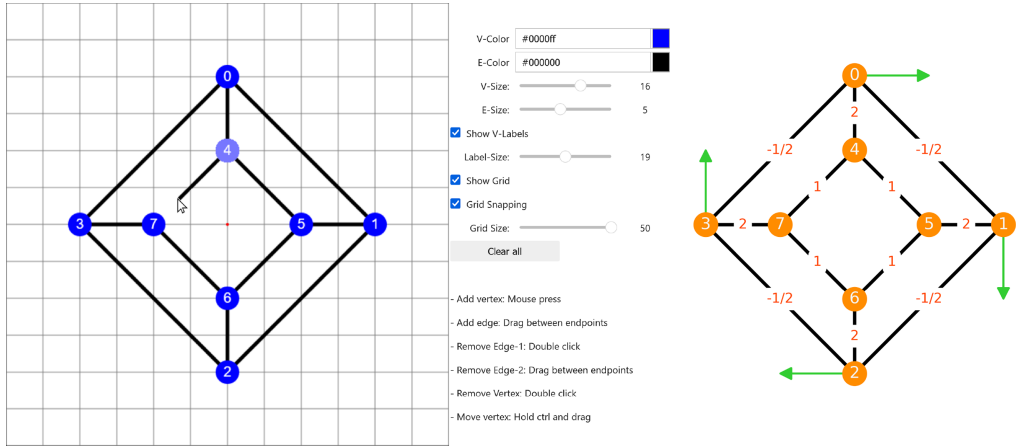


Fig. 4. The GraphDrawer GUI makes it possible to input a graph with a realization in PyRIGI by drawing it (left). In this screenshot from a Jupyter Notebook, the edge $\{4, 7\}$ is currently drawn. The resulting framework is then visualized using the internal plotting routines, while showing an infinitesimal flex and an equilibrium stress (right).

infinitesimal flexes or the space of equilibrium stresses is 1D. However, we intend to extend this approach to more general systems in the future, which amounts to showing that the stress matrix (see Section 3.1.6) is positive definite in the case of prestress stability [Connolly and Whiteley, 1996], and in the case of second-order rigidity requires showing that a certain polynomial system only has the origin as a real solution.

3.1.6 Stress Matrix. Given an equilibrium stress ω associated with a d -dimensional framework (G, p) on a graph $G = (V, E)$, we can construct an object called the *stress matrix* Ω . This is an $n \times n$ symmetric matrix, where $n = |V|$, with off-diagonal elements $-\omega_{u,v}$ and diagonal elements $\sum_{v \in V} \omega_{u,v}$ for all vertices $u, v \in V$. This construction ensures that the matrix's row- and column-sums are equal to 0. For the framework TP, the stress matrix of the stress stresses[0] can be constructed by calling `TP.stress_matrix(stresses[0])`. This matrix has ramifications for the rigidity properties of the corresponding framework. If the realization p of an infinitesimally rigid framework (G, p) is generic and the rank of Ω is $n - d - 1$, then (G, p) is globally rigid [Gortler et al., 2010]. In PyRIGI, the rank of Ω can be computed via `TP.stress_matrix_rank()`. For this function, the optional keyword `numerical` can be specified.

3.1.7 Drawing and Plotting. In PyRIGI, it is further possible to input a graph via the provided GraphDrawer class and simultaneously provide it with a realization. The GUI can be opened, for instance, in a Jupyter Notebook with the command

```
from pyrigi import GraphDrawer
GD = GraphDrawer()
```

and an example canvas are depicted in Figure 4 (left).

Interaction with the GUI is driven by left-clicking the canvas to create vertices that are numbered in ascending order. Furthermore, we can create edges by clicking and holding an existing vertex and dragging the cursor to another vertex. Conversely, vertices and edges can be removed by double-clicking. Additionally, it is possible to add a grid to the canvas of a chosen fineness and to snap the vertices to the grid. This allows for better control of the vertex positions. The created

graph and framework can be cast to PyRIGI objects by calling the methods `graph` and `framework`, respectively. For example, the output from Figure 4 (left) can be obtained with the command `F4=GD.framework()`.

Moreover, PyRIGI's `Framework` class comes with a variety of visualization options. By calling `plot2D` or `plot3D`, a given framework can be depicted in $\mathbb{R}^2$ or $\mathbb{R}^3$, respectively. Frameworks in d -dimensional space with large $d \in \mathbb{N}$ can be projected to 2D or 3D by providing a projection matrix with appropriate dimensions. Additionally, it is possible to depict infinitesimal flexes and equilibrium stresses, see Figure 4 (right). Regarding style adjustments, for instance, edges can be depicted as curved arcs or can be assigned custom edge colors. The edge colors can also be provided as a partition of the underlying graph's edges, which automatically creates distinct colors for each partition suitable for red-green color blind people. These parameters can be saved in an object of the `PlotStyle` class so that the particular style becomes reproducible across multiple instances of frameworks. Many additional keywords from the `MATPLOTLIB` library can be used here, as well.

3.1.8 Exports. PyRIGI makes it possible to export frameworks in two separate formats. First, the method `to_tikz` creates the TikZ⁶ code to embed the framework in PyRIGI's style in a scientific article that is written in L^AT_EX. Second, it is possible to create an STL file that can be used to 3D print the framework via the method `generate_stlBars`, which renders simple bars with holes that can be connected by screws and stop-nuts.

3.2 Graphs

Graphs are one of the main objects in PyRIGI. A graph object can be constructed using the class `Graph`. The most common way of constructing a graph is by providing a list of edges of the graph as we have seen at the beginning of Section 3.1. There are also other ways of constructing a `Graph`, as for example via the method `from_vertices_and_edges` that accepts a pair of lists indicating the vertices and the edges. This allows one to construct graphs with isolated vertices.

3.2.1 Generic Rigidity. As one may expect, PyRIGI can handle generic d -rigidity of graphs, where d denotes the dimension, with algorithms that are tailored to the considered dimension. Here, by *generic d -rigidity* we mean the following. A graph G is called *d -rigid* if any d -dimensional framework (G, p) such that all entries of the image of p are algebraically independent (i.e., the realization p is *generic*) is infinitesimally rigid. Otherwise, the graph is *d -flexible*.

Here, the most important method is `is_rigid`. In dimension 2, PyRIGI adopts by default an algorithm that checks whether there exists a $(2, 3)$ -tight subgraph of the given graph with the same vertex set; see Laman [1970], Lee and Streinu [2008], and Pollaczek-Geiringer [1927]. A graph $G = (V, E)$ is *$(2, 3)$ -tight* if $|E| = 2|V| - 3$ and it is *$(2, 3)$ -sparse*, meaning that every subset of n' vertices of G that determines at least one edge spans at most $2n' - 3$ edges in G .

In dimension 1, instead, PyRIGI uses by default the well-known characterization stating that a graph is rigid if and only if it is connected. In addition to these two algorithms, for every dimension PyRIGI may adopt a randomized method (using parameter `algorithm="randomized"`), based on creating a framework with the given underlying graph whose realization has randomly chosen integer coordinates, and checking whether the rigidity matrix kernel has the expected dimension; see Gortler et al. [2010, Algorithm 5.2, Proposition 5.7]. This randomized method never gives false positives. The theoretical foundations of this algorithm give a precise estimate of the probability of obtaining a false negative in terms of the range in which the random coordinates are chosen. By this, it is possible to specify in PyRIGI an upper bound for this probability when calling the algorithm.

⁶<https://github.com/pgf-tikz/pgf>.

Similar to `is_rigid`, one may use the method `is_min_rigid` to check whether a graph is minimally d -rigid. As a step further, PyRIGI is able to determine the rigid components of a graph, which are the maximal d -rigid subgraphs: the method `rigid_components` returns a list of their vertices. Finally, the number of complex realizations in the plane of a minimally 2-rigid graph up to the action of the complexification of the group of isometries of $\mathbb{R}^2$ can be computed via the method `number_of_realizations`—the latter is a wrapper for the analogous function from the package `lnumber` [Capco, 2024]; see Capco et al. [2018b] for the theoretical background. If the parameter `spherical` is set to `True`, the algorithm computes the number of complex realizations on the sphere; see Gallet et al. [2020] for the theoretical background.

3.2.2 (k, ℓ) -Sparsity and Tightness. As already mentioned while discussing generic rigidity, the notions of (k, ℓ) -sparsity and -tightness play a prominent role in rigidity theory, and PyRIGI ships efficient methods for checking both of them, based on the so-called *pebble game* algorithm [Jacobs and Hendrickson, 1997; Lee and Streinu, 2008]. Here, (k, ℓ) -sparsity and -tightness are generalizations of the corresponding $(2, 3)$ -concepts introduced before and are defined analogously.

3.2.3 Redundant Rigidity. We say that a graph G is *redundantly d -rigid* if removing any edge from G yields a d -rigid graph. In a similar manner, one can define *vertex redundant d -rigidity* and the counterparts of both these concepts involving removing k edges or vertices. PyRIGI has implemented methods for checking all these properties, based on the works of Jordán [2016, 2021], Jordán et al. [2022], Kaszanitzky and Király [2016], Motevallian et al. [2015], Servatius [1989], Summers et al. [2009], and Yu and Anderson [2009].

3.2.4 Global Rigidity. A framework (G, p) is called *globally d -rigid* if every d -dimensional realization q that is equivalent to p is also congruent to p .

A celebrated theorem by Gortler et al. [2010] shows that either all frameworks (G, p) , when p is generic, are globally d -rigid, or none is. Therefore, one can speak of global rigidity as a property of a graph. Detecting whether a graph is globally rigid or not is implemented in PyRIGI via the method `is_globally_rigid`. In dimension 2, PyRIGI relies on the result by Jackson and Jordán [2005] stating that a graph is globally 2-rigid if and only if it is 3-connected and redundantly rigid; a user may access this functionality by calling the method `is_globally_rigid` with the parameter `algorithm="redundancy"`. For higher dimensions, PyRIGI relies on a probabilistic algorithm developed in the paper by Gortler et al. As is the case for `is_rigid`, the user may also enter the optional parameter `prob`, which serves as an upper bound for the probability of false negatives.

A concept related to global rigidity is the one of *global linkedness*. If (G, p) is a framework and u, v are vertices of G , we say that u and v are globally linked in (G, p) if the distance between $p(u)$ and $p(v)$ equals the distance between $q(u)$ and $q(v)$ for any other realization q of G equivalent to p . Then, two vertices u and v of a graph G are said to be *weakly globally linked* if they are globally linked for at least one generic realization of G . This notion turned out to be useful in proving the generalization of a conjecture of Lovász and Yemini stating that every $d(d + 1)$ -connected graph is globally rigid in $\mathbb{R}^d$ [Villányi, 2025]. Jordán and Villányi [2024] provide an algorithm to check whether a pair of vertices is weakly globally linked. This algorithm is implemented in PyRIGI in the method `is_weakly_globally_rigid`.

As it is mentioned in the paper by Gortler et al. [2010], the randomized linear algebra computations, that at the moment are implemented using integer numbers, would benefit from a modular approach: one first solves the equations modulo a prime p , then the obtained solution is lifted to one modulo p^2 , and so on. This approach may be considered in some future versions of PyRIGI.

3.2.5 Extension Constructions. Common means of constructing rigid graphs are given by the coning operation and k -extensions. *Coning* a graph $G = (V, E)$ adds a new vertex $v^* \notin V$ and edges from v^* to each original vertex $v \in V$. Coning carries over the generic and global d -rigidity of a graph to $\mathbb{R}^{d+1}$ [Connelly and Whiteley, 2010]. The cone graph can be constructed via the method `cone`.

Given a number $k \in \mathbb{N}$ and a vertex $v \notin V$, a d -dimensional k -extension of $G = (V, E)$ is obtained by removing from E a subset $F \subset E$ of edges with $|F| = k$, adding the vertex v to V and, subsequently, adding edges from v to any subset of V with $d + k$ elements that contains all vertices incident to the edges in F . These extensions are important because, in many cases, they preserve some rigidity properties. For instance, 0- and 1-extensions preserve d -rigidity [Tay and Whiteley, 1985].

3.2.6 Rigidity Matroid. Rigidity theory has always been intimately connected with matroid theory: many of the concepts we have been discussing can be defined in terms of matroids, which are combinatorial objects that generalize both linear independence and cycles in a graph. The (generic) rigidity matroid $\mathcal{R}_d(G)$ of a graph G is defined as the linear matroid induced by the rows of the rigidity matrix of (G, p) where p is a generic d -dimensional realization of G ; the definition does not depend on the choice of a generic p . The bases of the generic rigidity matroid $\mathcal{R}_d(K_n)$ of the complete graph K_n are precisely the edge sets of the minimally d -rigid graphs on n vertices.

PyRIGI offers some basic functionality regarding computations in the generic rigidity matroid $\mathcal{R}_d(K_n)$. The Graph methods `is_Rd_dependent`, `is_Rd_independent` and `is_Rd_circuit` allow the user to determine whether the set of edges E of an n -vertex graph is dependent, independent or a circuit in $\mathcal{R}_d(K_n)$. Similarly, `is_Rd_closed` determines whether E is closed, and `Rd_closure` computes the closure of E in $\mathcal{R}_d(K_n)$.

3.2.7 NAC-Colorings. A generically rigid graph might still have non-generic realizations that are flexible. For any connected graph, such a flexible realization in $\mathbb{R}^2$ exists if and only if there exists an NAC-coloring [Grasegger et al., 2019]. This is a coloring of the edges in two colors, say red and blue, such that both colors occur and every cycle is either monochromatic or it contains both colors at least twice.

In general, the existence problem for NAC-colorings is NP-complete [Garamvölgyi, 2022], even for graphs with degree at most five [Laštovička and Legerský, 2024]. Nevertheless, the method `NAC_colorings` is able to compute all NAC-colorings for graphs that are not too large. This is done by determining classes of edges that must be monochromatic in every NAC-coloring. The naive approach is then to color all these classes in all possible ways and to check which ones yield an NAC coloring. Laštovička and Legerský improve this by an incremental approach, considering coloring of subgraphs, supported by new heuristics. The code supporting their paper, which is much faster than the one in FLEXRiLoG [Grasegger and Legerský, 2020], has been included in PyRIGI and extended by checks whether an NAC-coloring exists (method `has_NAC_coloring`) using some known sufficient or necessary conditions.

The existence of an NAC-coloring is, for instance, guaranteed if a graph has a stable separating set, i.e., a set of vertices that have no edges in between and whose removal disconnects the graph. For 2-flexible graphs, such a set can be found by a polynomial-time algorithm [Clinch et al., 2024], which is implemented in PyRIGI within the method `stable_separating_set`.

3.3 Motions

Beyond visualizing static frameworks, it is possible to visualize deformation paths of frameworks in PyRIGI. Given a framework $F = (G, p)$, with $G = (V, E)$ and $p: V \rightarrow \mathbb{R}^d$, a *motion* is a continuous map $\alpha: I \rightarrow (\mathbb{R}^d)^V$ for an interval $I \subset \mathbb{R}$ with $0 \in I$ such that $\alpha(0) = p$ and such that (G, p)

and $(G, \alpha(t))$ are equivalent for every $t \in \mathcal{I}$. A motion is called *trivial*, if (G, p) and $(G, \alpha(t))$ are congruent for all $t \in \mathcal{I}$.

3.3.1 Parametrized Motions. The most straightforward way to define a continuous motion is to provide a parametrization for it. When constructing a `ParametricMotion` object, we need to specify a parametrization in a single parameter and a (potentially unbounded) interval. For example, almost all frameworks F on the 4-cycle have a 1D deformation space that contains equivalent realizations of F . In this case, the corresponding system of polynomials is sufficiently simple so that one can parametrize a curve in this space by symbolically solving the bar-length equations. At the moment, the specific parametrization of the curve has to be provided by the user.

3.3.2 Numerical Approximations. However, the system of polynomial equations describing a framework's configuration space quickly becomes too complicated to be solved symbolically. If we still want to approximate continuous motion, we need to rely on numerical methods. Using homotopy continuation enables us to numerically track solution paths [Breiding and Timme, 2018]. Intuitively, paths are approximated by iteratively applying a *predictor-step* that anticipates the next value on the curve e.g., Euler's method) followed by a sequence of *corrector steps* to refine the previous prediction, usually by applying Newton's method.

Since the framework's deformation space naturally lives in $(\mathbb{R}^d)^V$, to approximate a curve in it, we can apply the Euclidean distance retraction [Heaton and Himmelmann, 2025]. This retraction is given by the metric projection to the closest point on the configuration space. It works as follows: Given a flexible realization p and a nontrivial infinitesimal flex q , we apply the path-tracking method to the closest point using the Lagrange multiplier system for the parameter $p + \varepsilon q$ and the step size $\varepsilon > 0$. This approach comes with theoretical convergence guarantees, as long as we stay sufficiently close to the constraint set. There are further theoretical considerations about singularities, vector transport, and stressed frameworks that are incorporated in this method, though their specifics are beyond the scope of this article. In the following example, we approximate a continuous motion of the complete bipartite graph $K_{2,4}$ with 348 steps, a step size of 0.1 and starting with the first infinitesimal flex (chosen_flex=0) from a basis that is internally computed so that the pair of vertices $\{0, 1\}$ is constrained to the direction $(1, 0)$:

```
from pyrigi.motion import ApproximateMotion
F = frameworkDB.CompleteBipartite(2,4)
motion = ApproximateMotion(F, 348, chosen_flex=0, step_size=0.1, fixed_pair=[0,1])
```

The number and size of the steps are chosen so that the approximated continuous motion becomes periodic.

3.3.3 Animations. The computed motions can be animated using the internal `animate` procedure in `PyRigi`. By default, this method produces an SVG animation (restricted to frameworks in $\mathbb{R}$ and $\mathbb{R}^2$), but it can also produce a `matplotlib.FuncAnimation` by setting the parameter `animation_format` to "matplotlib". This method works regardless of whether it is called on an instance of the `ParametricMotion` or `ApproximateMotion` class. While for objects of the former class, the `animate` method needs to first sample realizations from the parametric curve, objects from the latter class already come with that information. Most `PlotStyle` parameters that are admissible for framework plotting can be used here to change the style.

3.4 Using PyRigi for Rigidity Theory

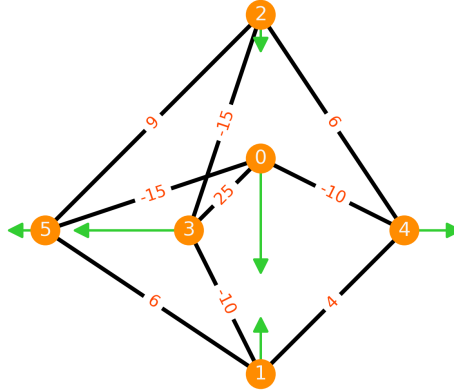
Let us consider an example to illustrate how to interact with `PyRigi`. For that, let us construct a framework F on the complete bipartite graph $K_{3,3}$. We equip it with a special realization called

‘dixonI’ so that the resulting framework contains nontrivial flexes. We propose one such infinitesimal flex and check that it is indeed nontrivial and an infinitesimal flex.

```
from pyrigi import frameworkDB
F = frameworkDB.CompleteBipartite(3, 3, realization='dixonI')
inf_flex = {0: (0, -6), 1: (0, 3), 2: (0, -2), 3: (-6, 0), 4: (3, 0), 5: (-2, 0)}
F.is_nontrivial_flex(inf_flex) # Output: True
```

Therefore, the framework F is infinitesimally flexible. We can plot the framework F alongside an equilibrium stress and the previously constructed infinitesimal flex inf_flex . We adjust the length of the vectors in inf_flex for visualization purposes.

```
F.plot(inf_flex=inf_flex, stress=9*F.stresses()[0], flex_length=0.19) # Output: see Figure
```



Subsequently, we extract the underlying graph G from the framework. The realization associated with the framework F is non-generic, which explains why it can be flexible. However, generic realizations of the graph G are infinitesimally rigid, which is why the graph itself is considered to be rigid.

```
G = F.graph
G.is_rigid() # Output: True
```

Still, the graph G is not globally rigid. To make G globally rigid, we can simply add an edge to it.

```
G.is_globally_rigid() # Output: False
G.add_edge(0, 1)
G.is_globally_rigid() # Output: True
```

Further tutorials on how to use PyRIGI can be found in the corresponding part of the online documentation.⁷

4 Implementation Details

In this section, we briefly discuss how PyRIGI is structured and what external tools it uses. PyRIGI uses several other PYTHON packages: in particular, we take advantage of the graph theory functionality implemented in NETWORKX [Hagberg et al., 2008]. For exact and numerical computations with frameworks we use SYMPY [Meurer et al., 2017] and NUMPY [Harris et al., 2020] respectively, while MATPLOTLIB [Hunter, 2007] is used for visualizations.

⁷<https://pyrigi.github.io/PyRigi/userguide/tutorial.html>.

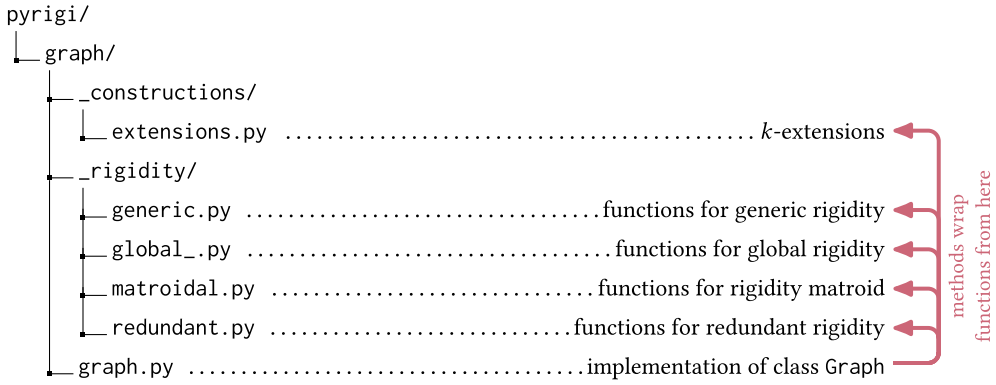


Fig. 5. An example of how the package is structured into modules according to topics.

From a user’s perspective, PyRIGI follows an object-oriented design. The functionality of the three areas depicted in Figure 1 can be accessed via the methods of the following classes:

- Graph—graph-related functionality, inherited from `networkx.Graph`.
- Framework—framework-related functionality, inherited from `FrameworkBase` (which is composed of `Graph` and a dictionary representing a realization).
- `ParametricMotion` and `ApproximateMotion`—functionality related to continuous motions, both inherited from `Motion`.

Moreover, the class `GraphDrawer` provides functionality for drawing graphs and 2D frameworks, and `PlotStyle2D`, `PlotStyle3D`, and their parent `PlotStyle` can be used for specifying the style of plots.

However, in order to have extensible and maintainable code, most of the graph algorithms are implemented as functions accepting a `networkx.Graph` instance as the first parameter. These functions are then wrapped as `pyrigi.Graph` methods. Hence, they are suggested by autocompletion tools once a `pyrigi.Graph` instance is available and, therefore, easy to search for and use. On the other hand, this approach allows the implementation of functionality in separate non-public modules according to their topics, which roughly correspond to the structure of Figure 1 and Section 3, see Figure 5 for an illustration. Similarly, the functions operating with frameworks accept `FrameworkBase` (which implements only the basic methods like adding vertices or edges) and are then wrapped as methods of `Framework`.

Since PyRIGI aims to be a larger, longer-term project that allows incorporating new areas of rigidity theory, several development tools and processes have been set up. In this document, we just summarize them. More details can be found in the development guide in the documentation.

Following the current standard in software development, the source code control is done using GRT, with the remote repository hosted on GitHub.⁸ Since libraries are explicitly versioned rather than continuously developed, the GRTFLOW⁹ approach is employed, namely, the branch `main` contains stable code, while the development is done on the branch `dev` via feature branches. Whenever appropriate, a new version is released by tagging that follows a *major.minor.patch* numbering scheme. Tag creation automatically triggers a GitHub action that deploys the release to

⁸<https://github.com/PyRigi/PyRigi>.

⁹Vincent Driessen, <https://nvie.com/posts/a-successful-git-branching-model>.

the Python Package Index.¹⁰ Hence, users can easily install the latest released version of PyRIGI via a `pip` command.

Consistent code formatting is achieved by the code formatter BLACK,¹¹ and compliance with the PEP 8 standard¹² is checked by FLAKE8.¹³ In order to maintain dependencies, we use POETRY,¹⁴ which also allows easy creation of virtual environments with clean installation of the required dependencies.

We aim to have high test coverage of the code. For this purpose, we distinguish three levels of tests: the standard ones are checked on every pull request to the development branch (automatically via a GitHub action), more tests are launched when merging to the branch `main` in order to create a new stable version (also automatic), and the last group of tests are long ones that are launched manually to test extensively a specific functionality during the development. The testing is done using PYTEST [Krekel et al., 2004].

We want to create both the math and code documentation as illustrated in Section 2 simultaneously with a single compilation. The standard way of documenting PYTHON modules is to use docstrings directly in the source code. These are compiled by SPHINX¹⁵ into an HTML-based documentation. It includes compilation of the mathematical and non-interface parts, which are written in MYST format,¹⁶ with tutorials being generated from Jupyter Notebooks via MYST-NB.¹⁷ The online documentation is updated automatically on every merge to the branch `main`. Notice that function docstrings are placed in corresponding modules, while the compiled documentation presents the methods wrapping them, as this is the interface that we provide to users. To avoid duplication, the docstring of a method that wraps a function is copied automatically using a decorator from the wrapped function. Namely, the actual source code is not as in Figures 2 and 3, where the docstrings are for illustration purposes hard copied from the wrapped functions.

5 Concluding Remarks

While PyRIGI has already proven to be useful, many important features are not yet present and await implementation. Among them, we mention:

- gain graphs and support for rigidity respecting symmetry and periodicity;
- implementations of different rigidity notions, other than bar-and-joint, such as linearly constrained or body-hinge frameworks;
- improvements in the efficiency of the currently implemented methods;
- implementations of more general criteria for prestress and second-order rigidity.

Both the generalization and specialization of admissible structures beyond bar-joint frameworks have the potential to significantly expand our user base. On the one hand, many real-world applications of rigidity theory require more general structures than mere bar-and-joint frameworks, such as body-bar and body-hinge frameworks. These geometric objects are composed of polyhedral rigid bodies that are connected by *hinges*, which allow their rotation around a mutual edge, or rigid bars. Examples of their applications include computer-aided design, robotics, and the rigidity of proteins and biomolecules, where groups of atoms may behave like rigid bodies. On the other

¹⁰<https://pypi.org/project/pyrigi>.

¹¹<https://black.readthedocs.io>.

¹²<https://peps.python.org/pep-0008>.

¹³<https://flake8.pycqa.org>.

¹⁴<https://python-poetry.org>.

¹⁵<https://sphinx-doc.org>.

¹⁶<https://myst-parser.readthedocs.io/>.

¹⁷<https://myst-nb.readthedocs.io>.

hand, special types of bar-and-joint frameworks, such as molecular frameworks and contact graphs coming from sphere packings, are highly relevant for applications in biochemistry and soft matter physics. Therefore, producing effective algorithms that exploit these objects' special structure may significantly increase the possible applications of PyRigi and make the package more relevant to researchers who are applying concepts from rigidity theory. Both of these observations provide meaningful directions for extending the package beyond bar-and-joint rigidity. We therefore plan to incorporate the corresponding methods and classes in future releases of PyRigi.

The best way to show interest in the development of PyRigi is to join its Zulip channel¹⁸ (or contact the maintainers) and propose new features or bug issues there. Our long-term goal is for PyRigi to become an environment in which researchers create modules devoted to their specific research needs. Ideally, these modules would then become part of PyRigi to be easily available to the community.

Acknowledgments

We would like to thank Mafalda Dal Cin for her code contribution related to global rigidity, Hakan Güler for the graph drawing functionality, Petr Laštovička for NAC-colorings and help with the package structure, and András Mihálykó for implementing pebble game algorithms. We would like to acknowledge the contributions to the PyRigi code by Jose Capco, Angelo El Saliby, Markéta Hošmáňková, Daniel Huczala, Lenka Kušnířová, Jan Rubeš, and Johannes Siegle.

We appreciate the comments, tests, and contributions to the math documentation by Sean Dewar, Alison La Porta, Rebecca Monks, and Anthony Nixon.

We acknowledge everybody who has contributed to PyRigi in any sense, including the many discussions we have had, especially the participants of the workshop Code of Rigidity (March 11–15, 2024), which was part of the Special Semester on Rigidity and Flexibility at RICAM, Linz, Austria. We thank RICAM for funding this event.

References

- Timothy G. Abbott. 2008. *Generalizations of Kempe's Universality Theorem*. Master's thesis. Massachusetts Institute of Technology. Retrieved from <http://hdl.handle.net/1721.1/44375>
- Ritwik Anand. 2024. *Minimally Rigid Graphs and Their Circle Packings*. GitHub. Retrieved from <https://github.com/Titwik/Dissertation>
- Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, et al. 2022. Introducing the FAIR principles for research software. *Scientific Data* 9, 1 (2022), 622. DOI: <https://doi.org/10.1038/s41597-022-01710-x>
- Evangelos Bartzos, Ioannis Z. Emiris, and Josef Schicho. 2020. On the multihomogeneous Bézout bound on the number of embeddings of minimally rigid graphs. *Applicable Algebra in Engineering, Communication and Computing* 31, 5–6 (2020), 325–357. DOI: <https://doi.org/10.1007/s00200-020-00447-7>
- Paul Breiding and Sascha Timme. 2018. HomotopyContinuation.jl: A package for homotopy continuation in Julia. In *Proceedings of the 6th International Congress on Mathematical Software (ICMS '18)*. James H. Davenport, Manuel Kauers, George Labahn, and Josef Urban (Eds.), Springer, 458–465. DOI: https://doi.org/10.1007/978-3-319-96418-8_54
- Jose Capco. 2024. *Lnumber: Toolkit for Computing the Laman Number*. Zenodo. DOI: <https://doi.org/10.5281/zenodo.8301012>
- Jose Capco, Matteo Gallet, Georg Grasegger, Christoph Koutschan, Niels Lubbes, and Josef Schicho. 2018a. *An Algorithm for Computing the Number of Realizations of a Laman Graph*. Zenodo. DOI: <https://doi.org/10.5281/zenodo.1245506>
- Jose Capco, Matteo Gallet, Georg Grasegger, Christoph Koutschan, Niels Lubbes, and Josef Schicho. 2018b. The number of realizations of a Laman graph. *SIAM Journal on Applied Algebra and Geometry* 2, 1 (2018), 94–125. DOI: <https://doi.org/10.1137/17M1118312>
- Oliver Clarke, Sean Dewar, Daniel Green Tripp, James Maxwell, Anthony Nixon, Yue Ren, and Ben Smith. 2026. A tropical approach to rigidity: Counting realisations of frameworks. *Journal of the London Mathematical Society* 113, 2 (2026), e70438. DOI: <https://doi.org/10.1112/jlms.70438>

¹⁸<https://pyrigi.zulipchat.com>.

- Katie Clinch, Dániel Garamvölgyi, John Haslegrave, Tony Huynh, Jan Legerský, and Anthony Nixon. 2026. Stable cuts, NAC-colourings and flexible realisations of graphs. To appear in *Journal of Graph Theory* (2026). DOI : <https://doi.org/10.1002/jgt.70064>
- Robert Connelly and Steven J. Gortler. 2017. Prestress stability of triangulated convex polytopes and universal Second-Order rigidity. *SIAM Journal on Discrete Mathematics* 31, 4 (2017), 2735–2753. DOI : <https://doi.org/10.1137/15M1054833>
- Robert Connelly and Walter Whiteley. 2010. Global rigidity: The effect of coning. *Discrete & Computational Geometry* 43, 4 (2010), 717–735. DOI : <https://doi.org/10.1007/s00454-009-9220-0>
- Robert Connelly and Walter Whiteley. 1996. Second-Order rigidity and prestress stability for tensegrity frameworks. *SIAM Journal on Discrete Mathematics*, 9, 3 (1996), 453–491. DOI : <https://doi.org/10.1137/S0895480192229236>
- Ronan Dance. 2024. *PhD-Research*. GitHub. Retrieved from <https://github.com/RonanD10/PhD-Research>
- Sean Dewar. 2021. *Graph-Rigidity-Checker*. GitHub. Retrieved from <https://github.com/dewar28/graph-rigidity-checker>
- Naomi Fox, Filip Jagodzinski, Yang Li, and Ileana Streinu. 2011. KINARI-Web: A server for protein rigidity analysis. *Nucleic Acids Research* 39 (2011), W177–W183. DOI : <https://doi.org/10.1093/nar/gkr482>
- Naomi Fox and Ileana Streinu. 2012. Towards accurate modeling for protein rigidity analysis. In: *2013 IEEE 3rd International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*. IEEE Computer Society, Los Alamitos, CA, 1–6. DOI : <https://doi.org/10.1109/ICCABS.2012.6182632>
- Matteo Gallet, Georg Grasegger, and Josef Schicho. 2020. Counting realizations of Laman graphs on the sphere. *The Electronic Journal of Combinatorics* 27, 2 (2020), 18. DOI : <https://doi.org/10.37236/8548>
- Matteo Gallet, Georg Grasegger, and Josef Schicho. 2022. *Software for Counting Realizations of Minimally Rigid Graphs on the Sphere*. Zenodo. DOI : <https://doi.org/10.5281/zenodo.6810642>
- Dániel Garamvölgyi. 2022. Global rigidity of (quasi-)injective frameworks on the line. *Discrete Mathematics* 345, 2 (2022), 112687. DOI : <https://doi.org/10.1016/j.disc.2021.112687>
- Steven J. Gortler, Alexander D. Healy, and Dylan P. Thurston. 2010. Characterizing generic global rigidity. *American Journal of Mathematics* 132, 4 (2010), 897–939. DOI : <https://doi.org/10.1353/ajm.0.0132>
- Georg Grasegger. 2022. *RigiComp—A Mathematica Package for Computational Rigidity of Graphs*. Zenodo. DOI : <https://doi.org/10.5281/zenodo.7457820>
- Georg Grasegger, Boulos El Hilany, and Niels Lubbes. 2022. *Calligraphs and Counting Realizations of Minimally Rigid Graphs*. Zenodo. DOI : <https://doi.org/10.5281/zenodo.6421148>
- Georg Grasegger and Jan Legerský. 2020. FlexRiLoG—A SageMath package for motions of graphs. In *Proceedings of the 7th International Conference on Mathematical Software (ICMS '20)*. *Lecture Notes in Computer Science*, Vol. 12097. Springer International Publishing, 442–450. DOI : https://doi.org/10.1007/978-3-030-52200-1_44
- Georg Grasegger, Jan Legerský, and Josef Schicho. 2019. Graphs with flexible labelings. *Discrete & Computational Geometry* 62, 2 (2019), 461–480. DOI : <https://doi.org/10.1007/s00454-018-0026-9>
- Jack E. Graver. 2001. *Counting on Frameworks. Mathematics to Aid the Design of Rigid Structures*. The Dolciani Mathematical Expositions. Vol. 25. Mathematical Association of America, Washington, DC. DOI : <https://doi.org/10.1090/dol/025>
- Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*. Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.), SciPy, 11–15. DOI : <https://doi.org/10.25080/TCWV9851>
- Varda Faghir Haghighi and Mahdi Sadjadi. 2022. RigidPy: Rigidity analysis in python. *Computer Physics Communications* 275 (2022), 108306. DOI : <https://doi.org/10.1016/j.cpc.2022.108306>
- Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362. DOI : <https://doi.org/10.1038/s41586-020-2649-2>
- Alexander Heaton and Matthias Himmelfmann. 2025. Computing Euclidean distance and maximum likelihood retraction maps for constrained optimization. *Computational Geometry* 126 (2025), 102147. DOI : <https://doi.org/10.1016/j.comgeo.2024.102147>
- Susanne M. A. Hermans, Christopher Pflieger, Christina Nutschel, Christian A. Hanke, and Holger Gohlke. 2017. Rigidity theory for biomolecules: Concepts, software, and applications. *WIREs Computational Molecular Science* 7, 4 (2017), e1311. DOI : <https://doi.org/10.1002/wcms.1311>
- Matthias Himmelfmann. 2026. Approximating continuous motions of geometric constraint systems. arXiv:2602.08016. Retrieved from <https://arxiv.org/abs/2602.08016>
- John D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. DOI : <https://doi.org/10.1109/MCSE.2007.55>
- Sadik Ilman and Timo de Wolff. 2016. Amoebas, nonnegative polynomials and sums of squares supported on circuits. *Research in the Mathematical Sciences* 3, 1 (2016), 9. DOI : <https://doi.org/10.1186/s40687-016-0052-2>
- Sartaj Islam. 2024. *Motion Generation for Flexible and Rigid Graphs*. GitHub. Retrieved from <https://github.com/Sartazz7/Graph-Rigidity>

- Bill Jackson and Tibor Jordán. 2005. Connected rigidity matroids and unique realizations of graphs. *Journal of Combinatorial Theory. Series B* 94, 1 (2005), 1–29. DOI: <https://doi.org/10.1016/j.jctb.2004.11.002>
- Donald J. Jacobs and Bruce Hendrickson. 1997. An algorithm for two-dimensional rigidity percolation: The pebble game. *Journal of Computational Physics* 137, 2 (1997), 346–365. DOI: <https://doi.org/10.1006/jcph.1997.5809>
- Donald J. Jacobs, Andrew J. Rader, Leslie A. Kuhn, and Michael F. Thorpe. 2001. Protein flexibility predictions using graph theory. *Proteins* 44, 2 (2001), 150–165. DOI: <https://doi.org/10.1002/prot.1081>
- Tibor Jordán. 2016. Combinatorial rigidity: Graphs and matroids in the theory of rigid frameworks. In *Discrete Geometric Analysis*. Martin T. Barlow, Tibor Jordán, and Andrzej Zuk (Eds.), *MSJ Memoirs*, Vol. 34, The Mathematical Society of Japan, 33–112. DOI: <https://doi.org/10.2969/msjmemoirs/03401c020>
- Tibor Jordán. 2021. Minimum size highly redundantly rigid graphs in the plane. *Graphs and Combinatorics* 37, 4 (2021), 1415–1431. DOI: <https://doi.org/10.1007/s00373-021-02327-4>
- Tibor Jordán, Christopher Poston, and Ryan Roach. 2022. Extremal families of redundantly rigid graphs in three dimensions. *Discrete Applied Mathematics* 322 (2022), 448–464. DOI: <https://doi.org/10.1016/j.dam.2022.03.006>
- Tibor Jordán and Soma Villányi. 2024. Globally linked pairs of vertices in generic frameworks. *Combinatorica* 44, 4 (2024), 817–838. DOI: <https://doi.org/10.1007/s00493-024-00094-3>
- Viktória E. Kaszanitzky and Csaba Király. 2016. On minimally highly vertex-redundantly rigid graphs. *Graphs and Combinatorics* 32, 1 (2016), 225–240. DOI: <https://doi.org/10.1007/s00373-015-1560-3>
- Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laughner, and Florian Bruhin. 2004. *Pytest* 8.3.5. Retrieved from <https://github.com/pytest-dev/pytest>
- Gerard Laman. 1970. On graphs and rigidity of plane skeletal structures. *Journal of Engineering Mathematics* 4 (1970), 331–340. DOI: <https://doi.org/10.1007/BF01534980>
- Martin Larsson. 2024. *Nauty Laman Plugin*. GitHub. Retrieved from <https://github.com/martinkjarlsson/nauty-laman-plugin>
- Petr Laštovička and Jan Legerský. 2024. Flexible realizations existence: NP-completeness on sparse graphs and algorithms. arXiv:2412.13721. Retrieved from <https://arxiv.org/abs/2412.13721>
- Audrey Lee and Ileana Streinu. 2008. Pebble game algorithms and sparse graphs. *Discrete Mathematics* 308, 8 (2008), 1425–1437. DOI: <https://doi.org/10.1016/j.disc.2007.07.104>
- Ciaran Mc Glue and Klara Stokes. 2022. Generating all rigidity circuits on at most 10 vertices and all assur graphs on at most 11 vertices. *Journal of Integer Sequences* 25, Article 221.3 (2022), 1–22.
- Ian McKenzie. 2020. *Nearly Isostatic Networks and Allosteric Effects*. GitHub. Retrieved from https://github.com/naimenz/rigidity_masters
- Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, et al. 2017. SymPy: Symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103. DOI: <https://doi.org/10.7717/peerj-cs.103>
- S. Alireza Motevallian, Changbin Yu, and Brian D. O. Anderson. 2015. On the robustness to multiple agent losses in 2D and 3D formations. *International Journal of Robust and Nonlinear Control* 25, 11 (2015), 1654–1687. DOI: <https://doi.org/10.1002/rnc.3167>
- Aysegül Ozkan, Rahul Prabhu, Troy Baker, James Pence, Jorg Peters, and Meera Sitharam. 2018. Algorithm 990: Efficient atlasing and search of configuration spaces of point-sets constrained by distance intervals. *ACM Transactions on Mathematical Software* 44, 4 (2018), 1–30. DOI: <https://doi.org/10.1145/3204472>
- Johannes Poetzsch. 2024. *Rigid Subgraphs*. GitHub. Retrieved from <https://github.com/johannes-poetzsch/Rigid-Subgraphs>
- Hilda Pollaczek-Geiringer. 1927. Über die Gliederung ebener Fachwerke. *Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik Und Mechanik* 7 (1927), 58–72. DOI: <https://doi.org/10.1002/zamm.19270070107>
- Brigitte Servatius. 1989. Birigidity in the plane. *SIAM Journal on Discrete Mathematics* 2, 4 (1989), 582–589. DOI: <https://doi.org/10.1137/0402050>
- Amit Singer and Mihai Cucuringu. 2010. Uniqueness of low-rank matrix completion by rigidity theory. *SIAM Journal on Matrix Analysis and Applications* 31, 4 (2010), 1621–1641. DOI: <https://doi.org/10.1137/090750688>
- Tyler H. Summers, Changbin Yu, and Brian D. O. Anderson. 2009. Addressing agent loss in vehicle formations and sensor networks. *International Journal of Robust and Nonlinear Control* 19, 15 (2009), 1673–1696. DOI: <https://doi.org/10.1002/rnc.1400>
- Tomohiro Tachi. 2016. Designing rigidly foldable horns using Bricard’s octahedron. *Journal of Mechanisms and Robotics* 8, 3 (2016), 031008. DOI: <https://doi.org/10.1115/1.4031717>
- Tiong-Seng Tay and Walter Whiteley. 1985. Generating isostatic frameworks. *Structural Topology* 11 (1985), 21–69. Retrieved from <https://hdl.handle.net/2099/1047>
- The PyRigi Developers. *PyRigi 1.2.0*. Zenodo. 2026. DOI: <https://doi.org/10.5281/zenodo.19613778>
- Michael F. Thorpe and Phillip M. Duxbury. 2002. *Rigidity Theory and Applications*. Springer US, Boston, MA. DOI: <https://doi.org/10.1007/b115749>

- Soma Villányi. 2025. Every $d + 1$ -connected graph is globally rigid in $\mathbb{R}^d$. *Journal of Combinatorial Theory: Series B* 173 (2025), 1–13. DOI: <https://doi.org/10.1016/j.jctb.2025.01.005>
- Menghan Wang and Meera Sitharam. 2015. Algorithm 951: Cayley analysis of mechanism configuration spaces using CayMos: Software functionalities and architecture. *ACM Transactions on Mathematical Software* 41, 4 (2015), 1–8. DOI: <https://doi.org/10.1145/2699462>
- Changbin Yu and Brian D. O. Anderson. 2009. Development of redundant rigidity theory for formation control. *International Journal of Robust and Nonlinear Control* 19, 13 (2009), 1427–1446. DOI: <https://doi.org/10.1002/rnc.1386>
- Daniel Zelazo and Shiyu Zhao. 2019. Formation control and rigidity theory. *Snapshots of Modern Mathematics from Oberwolfach* 17 (2019), 1–16. DOI: <https://doi.org/10.14760/SNAP-2019-017-EN>
- Yu Zhang, Qinhan Wei, Yongcai Wang, Haodi Ping, and Deying Li. 2023. GPART: Partitioning maximal redundant rigid and maximal global rigid components in generic distance graphs. *ACM Transactions on Sensor Networks* 19, 4 (2023), 1–26. DOI: <https://doi.org/10.1145/3594668>

Received 5 June 2025; revised 23 April 2026; accepted 26 April 2026