On Computational Poisson Geometry II: Numerical Methods

M. EVANGELISTA-ALVARADO, J. C. RUÍZ-PANTALEÓN AND P. SUÁREZ-SERRATO

We present twelve numerical methods for evaluation of objects and concepts from Poisson geometry. We describe how each method works with examples, and explain how it is executed in code. These include methods that evaluate Hamiltonian and modular vector fields, compute the image under the coboundary and trace operators, the Lie bracket of differential 1-forms, gauge transformations, and normal forms of Lie–Poisson structures on $\mathbf{R}^3$. The complexity of each of our methods is calculated, and we include experimental verification on examples in dimensions two and three.

Contents

1   Introduction

This is our second paper in a series on the computational aspects of Poisson geometry. In the first paper we showed how fundamental concepts from Poisson geometry could be operationalized into symbolic code [18]. We also provided an associated Python module with our implementations, for ease of execution[1].

A *Poisson manifold* [49, 60, 16, 34, 39] is a smooth manifold M equipped with a contravariant skew–symmetric 2–tensor field Π, *called Poisson bivector field*, satisfying the equation

$$(1\text{–}1) \qquad\qquad [\![\Pi, \Pi]\!] = 0,$$

with respect to the Schouten–Nijenhuis bracket $[\![\,,\,]\!]$ for multivector fields [46, 16]. Let $m = \dim M$, and $x = (U; x^1, \ldots, x^m)$ be local coordinates on M, then Π has the following representation [42, 58]:

$$(1\text{–}2) \qquad\qquad \Pi = \tfrac{1}{2}\Pi^{ij}\frac{\partial}{\partial x^i} \wedge \frac{\partial}{\partial x^j} \; = \sum_{1 \leq i < j \leq m} \Pi^{ij}\frac{\partial}{\partial x^i} \wedge \frac{\partial}{\partial x^j}$$

The functions $\Pi^{ij} = \Pi^{ij}(x) \in C_U^\infty$ are called the coefficients of Π, and $\{\partial/\partial x^i\}$ is the canonical basis for vector fields on $U \subseteq M$.

Poisson manifolds are generalizations of symplectic manifolds. A Poisson manifold can be though of informally as a space that is foliated by symplectic leaves. It is then possible to define Hamiltonian dynamics relative to the symplectic forms defined on each leaf, via the Poisson bracket. Comprehensive treatments are available for interested readers [16, 39].

---

[1]Available via https://github.com/appliedgeometry/poissongeometry

This geometric context provides a route that describes Hamiltonian dynamics rigorously. Applications of Hamiltonian dynamics are almost ubiquitous in every scientific domain. More specifically, uses of symplectic forms provide a formalism for diverse phenomena. This theory was introduced by Poisson himself to describe celestial mechanics [49]. It was then used by Dirac to investigate quantum mechanics [15], and notably Kontsevich showed that Poisson manifolds admit deformation quantizations [33].
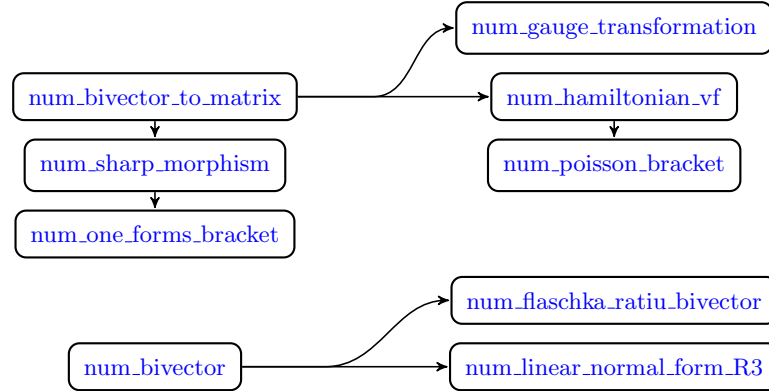
In this paper, we present twelve methods that allow for numerical computations of concepts from Poisson geometry. These are listed in the following Table 1.1, along with their respective algorithms, and a list of (not exhaustive) relevant references for each one. We have indicated with an asterisk (*) the seven methods that depend explicitly on our module for symbolic computation PoissonGeometry [18].

Table 1.1: Our numerical methods, with their corresponding algorithms, and examples where they are used. The right column is an informal summary of the algorithmic complexities, computed and presented in detail in Section 3.

Method	**Algorithm**	**Examples**	**Complexity**
num_bivector_field	2.1	[16, 8, 34, 39]	$O(2)$
num_bivector_to_matrix *	2.2	[16, 8, 34, 39]	$O(2)$
num_hamiltonian_vf	2.3	[36, 8, 56, 7, 26]	$O(2)$
num_poisson_bracket	2.4	[37, 16, 34, 39]	$O(2)$
num_sharp_morphism	2.5	[16, 34, 39]	$O(2)$
num_coboundary_operator *	2.6	[47, 16, 34, 2, 45]	$O(2^m)$
num_modular_vf *	2.7	[1, 16, 34, 27, 39, 3, 48]	$O(2^m)$
num_curl_operator *	2.8	[24, 13, 16, 39]	$O(2^m)$
num_one_forms_bracket *	2.9	[16, 34, 39, 25]	$O(2)$
num_gauge_transformation	2.10	[9, 4, 14]	$O(7)$
num_linear_normal_form_R3 *	2.11	[43, 47, 23, 16, 54, 8, 39, 14, 45, 20]	$O(1)$
num_flaschka_ratiu_bivector *	2.12	[24, 13, 22, 55, 17]	$O(6)$

The following diagram illustrates the internal functional dependencies of the methods available in NumPoissonGeometry [2].

---

[2]Avalaible via https://github.com/appliedgeometry/NumericalPoissonGeometry

```
num_bivector_to_matrix ──────┬──────→ num_gauge_transformation
         │                   └──────→ num_hamiltonian_vf
         ↓                                      │
num_sharp_morphism                              ↓
         │                           num_poisson_bracket
         ↓
num_one_forms_bracket


                              ┌──────→ num_flaschka_ratiu_bivector
num_bivector ─────────────────┤
                              └──────→ num_linear_normal_form_R3
```

The methods presented here have classical applications to Mechanics—see for example [8]—and also to recent advances in computer-aided techniques for determining normal forms for Hamiltonian systems [11]. Furthermore, there has been a recent surge of interest in understanding Hamiltonian dynamics as uses of this theory start to appear in the data analysis and machine learning communities [50]. Without attempting to be exhaustive, recent domains of application include; the development of Hamiltonian Monte Carlo techniques [28], applications of symplectic integration to optimization [32], inference of symbolic models from data [29], and the development of Hamiltonian Neural Networks [26, 57].

Our work has been specifically designed to be compatible with popular machine learning frameworks, as our code can be integrated into NumPy, Pytorch, or TensorFlow environments. Moreover, as we rely on lattice meshes for our evaluations, our results will also be of interest to researchers that use finite element methods.

We hope to contribute an additional dimension to the understanding of Poisson geometry, enabling everyone to carry out numerical experiments with our freely available open-source code, motivating the expansion of techniques that have so far been incorporated. To the best of our knowledge this is the first comprehensive implementation of these methods.

Our numerical techniques can inform and complement researchers' intuition and provide further insights. For example, if a certain vector field is not trivial it can imply that a given Poisson structure is not unimodular, and this can be verified numerically. Moreover, our module only needs the algebraic expression for the Poisson bivector to carry out this verification (and not the complete, explicit vector field).

We have included specific examples for each of the methods above where numerical computations would be desirable, or are relevant in published work. We also strongly believe that our algorithms in this paper can be useful in the following related fields.

Our methods could aid in the development of deep learning systems for flows on tori and spheres [30], in particular for the groups $U(1) \cong \mathbb{S}^1$ and $SU(2) \cong \mathbb{S}^3$. In a related direction, normalized flows on Lie groups have been investigated with respect to distributions of data points [19].

We expect our methods to be useful for evaluating inference problems related to the n-body problem, which has been approached recently [6].

Hamiltonian dynamics have been used to learn systems of simulations [51]. This innovative work pioneers deep learning techniques that combine Hamiltonian dynamics and integrators of Ordinary Differential Equations. With our methods, such an overarching program could be extended to include more diverse equations, such as the ones that define a Poisson structure, or that verify conservation of specific quantities, for instance, a unimodular flow.

Normalized flows and techniques from symplectic geometry have been used to find canonical Darboux coordinates [41]. This highlights that the use of symplectic, and more generally, Poisson techniques is computationally efficient because these structures preserve volume.

Models that learn to respect conservative laws, through the use of Hamiltonian dynamics have been recently developed [26]. These could also be extended to more ample classes using our methods.

Hamiltonian functions themselves, relative to a symplectic form, have been modelled using recurrent neural networks [10]. We expect extensions of these works to be possible using our methods as well, where now the functions may be Hamiltonian relative to a Poisson structure. Moreover, systems that learn the dynamics from observed orbit trajectory data could be potentially designed.

There are two more applications that are relevant to the computation of statistical quantities on Lie groups. First, Hamiltonian flows that are equivariant with respect to an action can be used to learn densities that are invariant under the action of a Lie algebra. This strategy has been realized already for very specific algebras [31], and could be extended to more with our methods. Second, these ideas could also lead to a general method for finding reparametrizable densities on arbitrary Lie groups, using their associated Lie-Poisson structure [12].

This paper is organized in the following way. In section 2, we describe our algorithms and point to domains of application for each one. In section 3, detailed analyses of the complexities of our methods are presented, as well as experimental run-times. A summary with explicit upper bounds is found in Table 3.1.

2 Numerical Methods

|In this section we describe the implementation of all functions in NumPoissonGeometry and present numerical examples with classical/relevant Poisson bivector fields.

2.1 NumPoissonGeometry: Evaluation

Our methods work with regular and *irregular* meshes that must be entered as NumPy arrays: a (k, m) NumPy array for a mesh with k points in $\mathbf{R}^m$, for $k \in \mathbf{N}$.

Irregular meshes can be used to implement probabilistic/statistical methods in Poisson geometry. In several of the subsections below we generate irregular meshes by means of random samples drawn from a uniform distribution over the interval $[0, 1)$. Sometimes for simplicity we use the 'corners' of the unit cube on $\mathbf{R}^m$,

(2–1)
$$Q^m := \{0, 1\} \underbrace{\times \cdots \times}_{m \text{ times}} \{0, 1\},$$

preloaded in Python into a NumPy array called Qmesh.

2.2 NumPoissonGeometry: Syntax

The instance and syntax of NumPoissonGeometry are the same as that of PoissonGeometry. We recall these here for the reader's convenience.

Coordinates. By default, to emulate canonical coordinates on $\mathbf{R}^m$, we use symbolic variables that are just the juxtaposition of the symbol x and an index of the set $\{1, \ldots, m\}$: (x1, ..., xm).

Scalar Functions. A scalar function is written using *string literal expressions*. For example, the function $f = a(x^1)^2 + b(x^2)^2 + c(x^3)^2$ should be written exactly as follows: 'a*x1**2 + b*x2**2 + c*x3**2'. All characters that are not coordinates are treated as (symbolic) parameters: a, b and c for the previous example.

Multivector Fields and Differential forms. Both multivector fields and differential forms are written using *dictionaries* with *tuples of integers* as *keys* and *string* type *values*. If the coordinate expression of an a–multivector field A, with $a \in \mathbb{N}$ is given by

$$A = \sum_{1 \leq i_1 < i_2 < \cdots < i_a \leq m} A^{i_1 i_2 \cdots i_a} \frac{\partial}{\partial x^{i_1}} \wedge \frac{\partial}{\partial x^{i_2}} \wedge \cdots \wedge \frac{\partial}{\partial x^{i_a}}, \quad A^{i_1 \cdots i_a} = A^{i_1 \cdots i_a}(x),$$

then A should be written using a dictionary, as follows:

$$(2\text{–}2) \qquad \left\{ (1, ..., a) : \mathcal{A}^{1 \cdots a}, \; ..., \; (i_1, ..., i_a) : \mathcal{A}^{i_1 \cdots i_a}, \; ..., \; (m - a + 1, ..., m) : \mathcal{A}^{m-a+1 \cdots m} \right\}$$

Here, each key $(i_1, \ldots, i_a)$ is a tuple containing ordered indices $1 \leq i_1 < \cdots < i_a \leq m$ and the corresponding value $\mathcal{A}^{i_1 \cdots i_a}$ is the string expression of the scalar function (coefficient) $A^{i_1 \cdots i_a}$ of A.

The syntax for differential forms is the same. It is important to remark that we can only write the keys and values of *non–zero coefficients*. See the documentation for more details.

2.3 NumPoissonGeometry: Python Implementation

First let us briefly describe the syntax of the module NumPoissonGeometry. The inputs of the twelve methods in Table 1.1 have to be string literal expressions. The sympify method converts such string expressions into symbolic variables, and the lambdify method transforms the symbolic expressions into functions that allow a (fast) numerical evaluation.

The output of our methods can be chosen to be a numpy array (by default) or a pytorch/tensorflow tensor containing the evaluation of the input data at each point in a mesh on $\mathbf{R}^m$.

Next we will explain each or our numerical methods, how they relate to the theoretical objects being implemented, present the corresponding algorithm, and show how they may be used with helpful examples.

2.4   Bivector Fields

The evaluation of a (Poisson) bivector field Π at a point in M is defined by

$$\Pi_p := \tfrac{1}{2}\Pi^{ij}(p)\left.\frac{\partial}{\partial x^i}\right|_p \wedge \left.\frac{\partial}{\partial x^j}\right|_p, \quad p \in M.$$

Observe that the coefficients of Π (1–2) at p determines the evaluation above.

The function $\mathrm{num\_bivector}$ evaluates a (Poisson) bivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.1 $\mathrm{num\_bivector}$(*bivector, mesh*)

Input: a (Poisson) bivector field and a mesh
Output: evaluation of the bivector field at each point of the mesh

---

1: **procedure**
2:      $m \leftarrow$ dimension of the manifold
3:      bivector $\leftarrow$ a variable encoding the (Poisson) bivector field
4:      mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
5:      TRANSFORM each bivector item to a function that allows a numerical evaluation
6:      **return** an array containing the evaluation of bivector at each m–array of mesh
7: **end procedure**

Example 2.1 Consider the Lie–Poisson bivector field on $\mathbf{R}^3_x$

$$(2\text{–}3) \qquad \Pi_{\mathfrak{so}(3)} = x_3\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} - x_2\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} + x_1\frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3},$$

associated to the 3–dimensional Lie algebra $\mathfrak{so}(3)$, which is used in studies of the 3–body problem [7], and more generally in geometric mechanics [8]. To evaluate $\Pi_{\mathfrak{so}(3)}$ at points of Q^3 (2–1) we compute:

```
>>> npg3 = NumPoissonGeometry(3)                    # NumPoissonGeometry instance
>>> P_so3 = {(1, 2): 'x3', (1, 3): '-x2', (2, 3): 'x1'}
                                    # dictionary for Π_so(3) in (2–3) according to (2–2)
>>> npg3.num_bivector(P_so3, Qmesh)                 # run num_bivector function

[{(1, 2): 0.0, (1, 3):  0.0, (2, 3): 0.0},  {(1, 2): 1.0, (1, 3):  0.0, (2, 3): 0.0},
 {(1, 2): 0.0, (1, 3): -1.0, (2, 3): 0.0},  {(1, 2): 1.0, (1, 3): -1.0, (2, 3): 0.0},
 {(1, 2): 0.0, (1, 3):  0.0, (2, 3): 1.0},  {(1, 2): 1.0, (1, 3):  0.0, (2, 3): 1.0},
 {(1, 2): 0.0, (1, 3): -1.0, (2, 3): 1.0},  {(1, 2): 1.0, (1, 3): -1.0, (2, 3): 1.0}]
```

Note that the output preserves the PoissonGeometry syntax (2–2). For example, to produce a PyTorch tensor encoding this information we use the $\mathrm{pt\_output}$ flag:

>>> npg3.num_bivector(P_so3, Qmesh, pt_output=True)

run num_bivector function with pt_output flag

tensor([[[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]], [[0., 1., 0.], [-1., 0., 0.], [0., 0., 0.]],
         [[0., 0., -1.], [0., 0., 0.], [1., 0., 0.]], [[0., 1., -1.], [-1., 0., 0.], [1., 0., 0.]],
         [[0., 0., 0.], [0., 0., 1.], [0., -1., 0.]], [[0., 1., 0.], [-1., 0., 1.], [0., -1., 0.]],
         [[0., 0., -1.], [0., 0., 1.], [1., -1., 0.]], [[0., 1., -1.], [-1., 0., 1.], [1., -1., 0.]]],
         dtype=torch.float64)
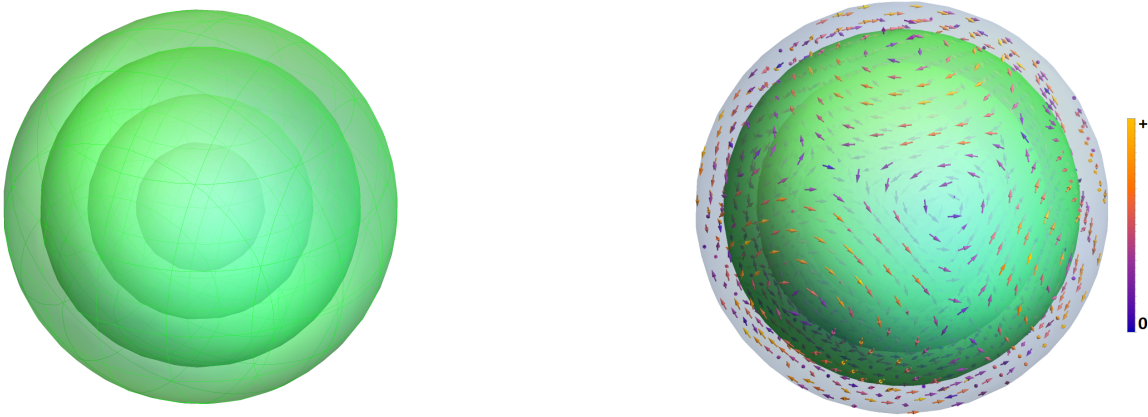

Figure 2.1: *Left*: Symplectic foliation of $\Pi_{\mathfrak{so}(3)}$ in (2–3). *Right*: Modular vector field of Π in (2–13) relative to the Euclidean volume form on $\mathbf{R}_x^3$. The color scale indicates the magnitude of the vectors.

2.5   Matrix of a Bivector Field

The value of the matrix $[\Pi^{ij}]$ of a (Poisson) bivector field Π at a point in M is given by

(2–4) $$\left[\Pi^{ij}\right]_p = \left[\Pi^{ij}(p)\right], \quad p \in M.$$

Hence, we just need to know the value of the coefficients of Π (1–2) at p.

The function num_bivector_to_matrix evaluates the matrix of a (Poisson) bivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.2 num_bivector_to_matrix(*bivector, mesh*)

Input: a (Poisson) bivector field and a mesh
Output: evaluation of the matrix of the (Poisson) bivector field at each point of the mesh

---

1: **procedure**
2:      $m \leftarrow$ dimension of the manifold
3:      bivector $\leftarrow$ a variable encoding the (Poisson) bivector field
4:      mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
5:      variable_1 $\leftarrow$ bivector_to_matrix(bivector)   ▷
     bivector_to_matrix: function of PoissonGeometry

6:      TRANSFORM each variable_1 item to a function that allows a numerical evaluation
7:      **return** an array containing the evaluation of variable_1 at each m–array of mesh
8: **end procedure**

Example 2.2 Consider the Lie–Poisson bivector field on $\mathbf{R}^3_x$

$$(2\text{–}5) \qquad \Pi_{\mathfrak{sl}(2)} = -x_3 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} - x_2 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} + x_1 \frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3},$$

associated to the 3–dimensional Lie algebra $\mathfrak{sl}(2)$, used in the classification of rigid motions [14], and in other mechanical systems [47, 8]. To evaluate the matrix of $\Pi_{\mathfrak{sl}(2)}$ at points of Q^3 (2–1) we compute:

```
>>> npg3 = NumPoissonGeometry(3)              # NumPoissonGeometry instance
>>> P_sl2 = {(1, 2): '-x3', (1, 3): '-x2', (2, 3): 'x1'}
                                   # dictionary for Π_sl(2) in (2–5) according to (2–2)
>>> npg3.num_bivector_to_matrix(P_sl2, Qmesh, pt_output=True)
                          # run num_bivector_to_matrix function with pt_output flag

tensor([[[0., 0., 0.], [0., 0., 0.], [ 0.,  0., 0.]], [[0., 1., 0.], [-1., 0., 0.], [ 0.,  0., 0.]],
        [[0., 0., 1.], [0., 0., 0.], [-1.,  0., 0.]], [[0., 1., 1.], [-1., 0., 0.], [-1.,  0., 0.]],
        [[0., 0., 0.], [0., 0., 1.], [ 0., -1., 0.]], [[0., 1., 0.], [-1., 0., 1.], [ 0., -1., 0.]],
        [[0., 0., 1.], [0., 0., 1.], [-1., -1., 0.]], [[0., 1., 1.], [-1., 0., 1.], [-1., -1., 0.]]],
        dtype=torch.float64)
```

2.6   Hamiltonian Vector Fields

The Hamiltonian vector field $X_h := \mathbf{i}_{\mathrm{d}h}\Pi$ of a scalar function h, and relative to a Poisson bivector field Π on M, at a point in M can be determined by the following (coordinate) formula:

$$(2\text{–}6) \qquad X_h|_p \simeq - \left[\Pi^{ij}(p)\right] \left[\frac{\partial h}{\partial x^k}(p)\right], \quad p \in M; \quad k = 1, \ldots, m$$

Here, $[\Pi^{ij}]$ is the matrix of Π (2–4), and $[\partial h/\partial x^k]$ is the gradient vector (field) of h.

The function num_hamiltonian_vf evaluates a Hamiltonian vector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.3 num_hamiltonian_vf(*bivector, ham_function, mesh*)

Input: a Poisson bivector field Π, a scalar function h and a mesh
Output: evaluation of the Hamiltonian vector field of h respect to Π at each point of the mesh

1: **procedure**
2:      $m \leftarrow$ dimension of the manifold
3:      bivector $\leftarrow$ a variable encoding the Poisson bivector field Π
4:      ham_function $\leftarrow$ a (string) expression representing the scalar function h
5:      mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
6:      CONVERT ham_function to a symbolic variable
7:      variable_1 $\leftarrow$ the (symbolic) gradient vector field of ham_function

8:        TRANSFORM each variable_1 item to a function that allows a numerical evaluation
9:        variable_2 ← an array containing the evaluation of variable_1 at each m−array of mesh
10:       variable_3 ← num_bivector_to_matrix(bivector, mesh)   ▷ see Algorithm 2.2
11:       variable_4 ← an empty container
12:       **for** $0 \leq i < k$ **do**
13:           variable_4[i] ← $(-1) *$ variable_3[i] $*$ variable_2[i]   ▷ matrix–vector product
14:       **end for**
15:       **return** variable_4
16: **end procedure**

Example 2.3 Consider the Hamiltonian vector field on $\mathbf{R}_x^6$, that arises in a particular case of the three body problem [7],

$$X_h = - x_4 \frac{\partial}{\partial x_1} - x_5 \frac{\partial}{\partial x_2} - x_6 \frac{\partial}{\partial x_3} + \left[\frac{1}{(x_1 - x_2)|x_1 - x_2|} + \frac{1}{(x_1 - x_3)|x_1 - x_3|} \right] \frac{\partial}{\partial x_4}$$

$$+ \left[\frac{1}{(x_1 - x_2)|x_1 - x_2|} + \frac{1}{(x_2 - x_3)|x_2 - x_3|} \right] \frac{\partial}{\partial x_5}$$

$$- \left[\frac{1}{(x_1 - x_3)|x_1 - x_3|} + \frac{1}{(x_2 - x_3)|x_2 - x_3|} \right] \frac{\partial}{\partial x_6},$$

with Hamiltonian function

$$(2-7) \qquad h = \frac{1}{x_1 - x_2} + \frac{1}{x_1 - x_3} + \frac{1}{x_2 - x_3} + \frac{x_4^2 + x_5^2 + x_6^2}{2},$$

and relative to the canonical Poisson bivector field on $\mathbf{R}_x^6$

$$(2-8) \qquad \Pi = \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_4} + \frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_5} + \frac{\partial}{\partial x_3} \wedge \frac{\partial}{\partial x_6}.$$

To evaluate X_h avoiding singularities we use the mesh, for Q^3 being as in (2–1),

$$\{-2, -1\} \times \{0, 1\} \times \{2, 3\} \times Q^3, \qquad \mathbf{R}^6 \simeq \mathbf{R}^3_{(x_1, x_2, x_3)} \times \mathbf{R}^3_{(x_4, x_5, x_6)} :$$

```
>>> npg6 = NumPoissonGeometry(6)                    # NumPoissonGeometry instance
>>> P = {(1, 4): 1, (2, 5): 1, (3, 6): 1}
                                    # dictionary for Π in (2–8) according to (2–2)
>>> h = '1/(x1 - x2) + 1/(x1 - x3) + 1/(x2 - x3) + (x4**2 + x5**2 + x6**2)/2'
                                               # string expression for h in (2–7)
>>> npg6.num_hamiltonian_vf(P, h, mesh, pt_output=True)
                              # run num_hamiltonian_vf function with pt_output flag

tensor([[[ 0.0000], [ 0.0000], [ 0.0000], [-0.3125], [0.0000], [0.3125]],

        ...,
        [[-1.0000], [-1.0000], [ 0.0000], [-0.3125], [0.0000], [0.3125]],
        [[-1.0000], [-1.0000], [-1.0000], [-0.3125], [0.0000], [0.3125]]],
        dtype=torch.float64)
```

2.7 Poisson Brackets

By definition and (2–6), the Poisson bracket $\{f, g\}_\Pi$ of two scalar functions f and g, induced by a Poisson bivector field Π on M, at a point in M can be calculated in coordinates as follows:

$$\{f, g\}_\Pi(p) = -\left[\tfrac{\partial g}{\partial x^k}(p)\right]^\top \left[\Pi^{ij}(p)\right]\left[\tfrac{\partial f}{\partial x^l}(p)\right], \quad p \in M; \quad k, l = 1, \ldots, m$$

Here, $[\Pi^{ij}]$ is the matrix of Π (2–4), $[\partial f / \partial x^l]$ and $[\partial g / \partial x^k]$ are the gradient vector (fields) of f and g, in that order.

The function $\mathrm{num\_poisson\_bracket}$ evaluates the Poisson bracket of two scalar functions on a mesh in $\mathbf{R}^m$.

Algorithm 2.4 num_poisson_bracket(*bivector, function_1, function_2, mesh*)

Input: a Poisson bivector field Π, two scalar functions f, g and a mesh
Output: evaluation of the Poisson bracket of f and g induced by Π at each point of the mesh

---

1: **procedure**
2:     $m \leftarrow$ dimension of the manifold
3:     bivector $\leftarrow$ a variable encoding the Poisson bivector field Π
4:     function_1, function_2 $\leftarrow$ string expressions representing the scalar functions f and g
5:     mesh $\leftarrow$ a (k, m) array encoding the mesh $\triangleright$ k: number of points in the mesh
6:     **if** function_1 $==$ function_2 **then**
7:         **return** 0 $\triangleright$ if $f = g$, then its Poisson bracket is zero
8:     **else**
9:         CONVERT function_2 to a symbolic variable
10:        variable_1 $\leftarrow$ the (symbolic) gradient vector field of function_2
11:        TRANSFORM each variable_1 item to a function that allows a numerical evaluation
12:        variable_2 $\leftarrow$ an array containing the evaluation of variable_1 at each m–array of mesh
13:        variable_3 $\leftarrow$ num_hamiltonian_vf(bivector, function_1, mesh)
                                                       $\triangleright$ see Algorithm 2.3
14:        variable_4 $\leftarrow$ an empty container
15:        **for** $0 \leq i < k$ **do**
16:            variable_4[i] $\leftarrow$ variable_2[i] $*$ variable_3[i] $\triangleright$ scalar vector product
17:        **end for**
18:        **return** variable_4
19:    **end if**
20: **end procedure**

Example 2.4 Consider the Poisson bivector field on $\mathbf{R}_x^6$, obtained as a deformation of an almost Poisson structure analyzed in relation to plasma [38] (see, also [53]),

$$(2\text{–}9) \qquad \Pi = \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_4} + \frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_5} + \frac{\partial}{\partial x_3} \wedge \frac{\partial}{\partial x_6} + x_2^2 \frac{\partial}{\partial x_5} \wedge \frac{\partial}{\partial x_6}.$$

Observe that the Poisson bracket $\{x_6, x_5\}_\Pi = -1$, at points $x \in \mathbf{R}^6$ such that $|x_2| = 1$. We can check this fact using random meshes of the form

$$\{a_1, b_1\} \times \{1\} \times \{a_2, b_2\} \times \cdots \times \{a_5, b_5\}, \qquad a_i, b_i \in \text{random } [0, 1).$$

Here, random samples are taken from a uniform distribution over the interval $[0, 1)$:

```
>>> npg6 = NumPoissonGeometry(6)                    # NumPoissonGeometry instance
>>> P = {(1, 4): 1, (2, 5): 1, (3, 6): 1, (5, 6): 'x2**2'}
                                        # dictionary for Π in (2–9) according to (2–2)
>>> f, g = 'x6', 'x5'      # string expressions for canonical coordinates x6 and x5, in that order
>>> npg6.num_poisson_bracket(P, f, g, meshes, pt_output=True)
                                    # run num_poisson_bracket function with pt_output flag
```

```
tensor(-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
       -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
       -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
       -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, dtype=torch.float64)
```

Observe that we use a probabilistic method to verify a particular property of a Poisson bracket. These methods can be used to determine other characteristics, for example, singular points.

2.8   Sharp Morphism

In the context of Lie algebroids, the vector bundle morphism $\Pi^\sharp : \alpha \mapsto \mathbf{i}_\alpha \Pi$, induced by a Poisson bivector field Π on M, is the anchor map of the Poisson Lie algebroid corresponding to Π [21].

Similarly to (2–6), we can evaluate the image $\Pi^\sharp(\alpha)$ of a differential 1–form α at a point in M as follows:

(2–10) $$\Pi^\sharp(\alpha)\big|_p \simeq - \left[\Pi^{ij}(p)\right]\left[\alpha_k(p)\right], \quad p \in M; \quad k = 1, \ldots, m$$

Here, $[\Pi^{ij}]$ is the matrix of Π (2–4), and $[\alpha_k]$ is the coefficient vector of $\alpha = \alpha_k \mathrm{d}x^k$.

The function num_sharp_morphism evaluates a vector field $\Pi^\sharp\alpha$ on a mesh in $\mathbf{R}^m$.

Algorithm 2.5 num_sharp_morphism(*bivector, one_form, mesh*)

Input: a Poisson bivector field Π, a differential 1–form α and a mesh
Output: evaluation of the vector field $\Pi^\sharp(\alpha)$ at each point of the mesh

---

1: **procedure**
2:     $m \leftarrow$ dimension of the manifold
3:     bivector $\leftarrow$ a variable encoding the Poisson bivector field Π
4:     one_form $\leftarrow$ a variable encoding the differential 1–form $\alpha = \alpha_1 \mathrm{d}x^1 + \cdots + \alpha_m \mathrm{d}x^m$
5:     mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
6:     variable_1 $\leftarrow$ a container with items $(\alpha_1, \ldots, \alpha_m)$
7:     TRANSFORM each variable_1 item to a function that allows a numerical evaluation
8:     variable_2 $\leftarrow$ an array containing the evaluation of variable_1 at each m–array of mesh

---

9:      variable_3 ← num_bivector_to_matrix(bivector, mesh)   ▷ see Algorithm 2.2
10:     variable_4 ← an empty container
11:     **for** $0 \leq i < k$ **do**
12:         variable_4[i] ← (−1) ∗ variable_3[i] ∗ variable_2[i]   ▷ matrix–vector product
13:     **end for**
14:     **return** variable_4
15: **end procedure**

Example 2.5 Consider the Casimir function $K = 1/2(x_1^2 + x_2^2 + x_3^2)$ of the Poisson bivector field $\Pi_{\mathfrak{so}(3)}$ in (2–3) [13, 7]. By definition, the exterior derivative of K,

(2–11) $$dK = x_1 dx_1 + x_2 dx_2 + x_3 dx_3,$$

belongs to the kernel of $\Pi_{\mathfrak{so}(3)}$, that is, $\Pi_{\mathfrak{so}(3)}^{\sharp} dK = 0$. We can check this fact by using random meshes:

```
>>> npg3 = NumPoissonGeometry(3)                        # NumPoissonGeometry instance
>>> P_so3 = {(1, 2): 'x3', (1, 3): '-x2', (2, 3): 'x1'}
                                    # dictionary for Π_so(3) in (2–3) according to (2–2)
>>> dK = {(1,): 'x1', (2,): 'x2', (3,): 'x3'}
                                    # dictionary for dK in (2–11) according to (2–2)
>>> mesh = numpy.random.rand(10**6, 3)
          # # (10^6, 3) numpy array with random samples from a uniform distribution over [0,1)
>>> npg6.num_sharp_morphism(P, dK, mesh, pt_output=True)
                                    # run num_sharp_morphism function with pt_output flag

tensor([ [[0.], [0.], [0.]],  [[0.], [0.], [0.]],  [[0.], [0.], [0.]],  …,
         [[0.], [0.], [0.]],  [[0.], [0.], [0.]],  [[0.], [0.], [0.]] ], dtype=torch.float64)
```

2.9   Coboundary Operator

The coboundary operator $\delta_{\Pi} : \Gamma \wedge^{\bullet} TM \to \Gamma \wedge^{\bullet+1} TM$ induced by a Poisson bivector field Π on M [42] is defined by

$$\delta_{\Pi}(A) := [\![\Pi, A]\!], \quad A \in \Gamma \wedge TM.$$

Here, $\Gamma \wedge TM$ denotes the module of multivector fields on M.

The function num_coboundary_operator evaluates the image under δ_{Π} of an arbitrary multivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.6 num_coboundary_operator(*bivector, multivector, mesh*)

Input: a Poisson bivector field Π, a multivector field A and a mesh
Output: evaluation of the multivector field $[\![\Pi, A]\!]$ at each point of the mesh

---

1: **procedure**

2:    $m \leftarrow$ dimension of the manifold
3:    bivector $\leftarrow$ a variable encoding the Poisson bivector field Π
4:    multivector $\leftarrow$ a variable encoding the multivector field A, or a (string) expression if degree $A = 0$
5:    mesh $\leftarrow$ a (k, m) array encoding the mesh $\triangleright$ k: number of points in the mesh
6:    variable_1 $\leftarrow$ lichnerowicz_poisson_operator(bivector, multivector)
                               $\triangleright$ lichnerowicz_poisson_operator: function of PoissonGeometry
7:    TRANSFORM each variable_1 item to a function that allows a numerical evaluation
8:    **return** an array containing the evaluation of variable_1 at each m–array of mesh
9: **end procedure**

Example 2.6 The vector field on $\mathbf{R}_x^3 \setminus \{x_3\text{–axis}\}$

$$(2\text{–}12) \qquad W = \frac{x_1 x_3 e^{-1/(x_1^2+x_2^2-x_3^2)^2}}{x_1^2 + x_2^2} \frac{\partial}{\partial x_1} + \frac{x_2 x_3 e^{-1/(x_1^2+x_2^2-x_3^2)^2}}{x_1^2 + x_2^2} \frac{\partial}{\partial x_2} + e^{-1/(x_1^2+x_2^2-x_3^2)^2} \frac{\partial}{\partial x_3}$$

arises in the study of the first cohomology group of the Poisson bivector field $\Pi_{\mathfrak{sl}(2)}$ in (2–5) to construct a 1–cocycle that is not Hamiltonian [47, 45]. To check the cocycle property of W under the coboundary operator induced by $\Pi_{\mathfrak{sl}(2)}$, we evaluate the (image) bivector field $[\![\Pi_{\mathfrak{sl}(2)}, W]\!]$ on random meshes:

```
>>> npg3 = NumPoissonGeometry(3)                    # NumPoissonGeometry instance
>>> P_sl2 = {(1, 2): '-x3', (1, 3): '-x2', (2, 3): 'x1'}
                               # dictionary for Π_sl(2) in (2–5) according to (2–2)
>>> W = {(1,): 'x1 * x3 * exp(-1/(x1**2 + x2**2 - x3**2)**2) / (x1**2 + x2**2)',
         (2,): 'x2 * x3 * exp(-1/(x1**2 + x2**2 - x3**2)**2) / (x1**2 + x2**2)',
         (3,): 'exp(-1/(x1**2 + x2**2 - x3**2)**2)'}
                               # dictionary for W in (2–12) according to (2–2)
>>> mesh = numpy.random.rand(10**6, 3)
        # # (10^6, 3) numpy array with random samples from a uniform distribution over [0,1)
>>> npg3.num_coboundary_operator(P_sl2, W, mesh, pt_output=True)
                       # run num_coboundary_operator function with pt_output flag

tensor([ [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],  [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],
        . . .,
        [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],  [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],
        [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]],  [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]] ],
        dtype=torch.float64)
```

The characteristic foliation of $\Pi_{\mathfrak{sl}(2)}$ can be described using the Casimir function $2K = x_1^2 + x_2^2 - x_3^2$ (see, Figure 2.2). Observe that W is orthogonal to the gradient vector field of K, $x_1 \partial/\partial x_1 + x_2 \partial/\partial x_2 - x_3 \partial/\partial x_3$. Hence, W is tangent to the symplectic foliation of $\Pi_{\mathfrak{sl}(2)}$ (see, Figure 2.2). However, it known that W is not a Hamiltonian vector field [47]. Therefore the first Poisson cohomology group of $\Pi_{\mathfrak{sl}(2)}$ is non–trivial.
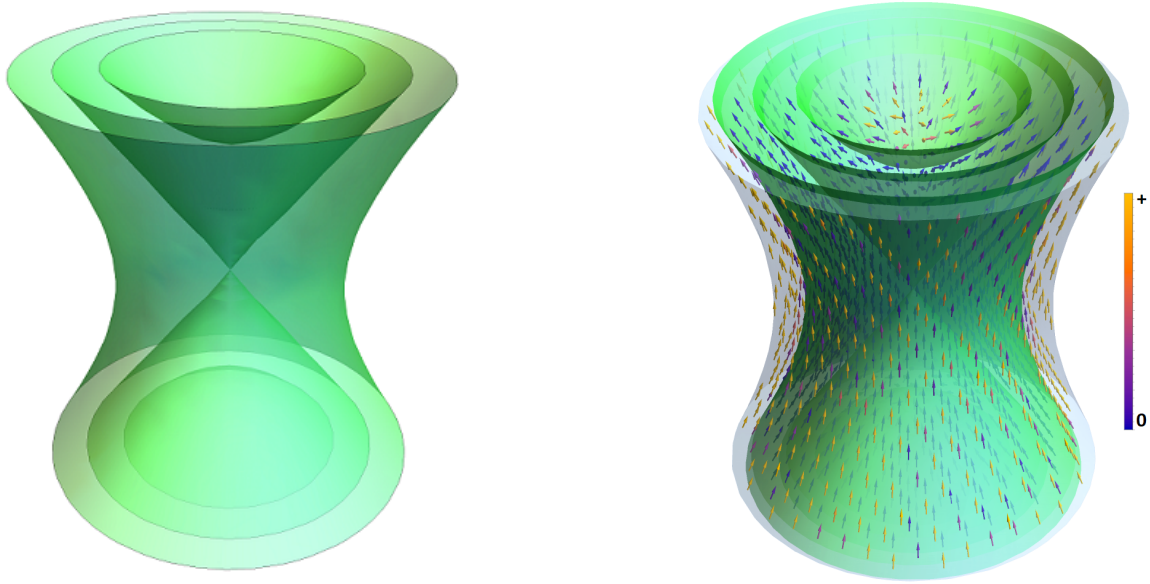
Figure 2.2: *Left*: Symplectic foliation of $\Pi_{\mathfrak{sl}(2)}$ in (2–5). *Right*: Vector field W in (2–12), tangent to the symplectic foliation of $\Pi_{\mathfrak{sl}(2)}$ in (2–5). The color scale indicates the magnitude of the vectors.

2.10 Modular Vector Field

The modular vector field of an orientable Poisson manifold (M, Π) is an infinitesimal automorphism of Π determined by the choice of a volume form Ω [59], it is defined by the linear map

$$Z_{\Pi,\Omega} : h \longmapsto \mathrm{div}_\Omega X_h, \quad h \in C_M^\infty.$$

Here, $\mathrm{div}_\Omega X_h \in C_M^\infty$ denotes the divergence of the Hamiltonian vector field X_h (2–6) with respect to Ω.

The modular vector field measures how far Hamiltonian flows are from preserving a given volume form [59]. In the regular case, the existence of a volume form that remains invariant under every Hamiltonian vector field only depends on the characteristic (symplectic) foliation of the Poisson manifold, rather than the leaf–wise symplectic form [1]. In particular, for regular codimension–one symplectic foliations, a characteristic class controls the existence of such a volume form [27].

The function num_modular_vf evaluates the modular vector field of a Poisson bivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.7 num_modular_vf(*bivector, function, mesh*)

Input: a Poisson bivector field Π, a non–zero scalar function f_0 and a mesh
Output: evaluation of the modular vector field of Π relative to the volume form $f_0\Omega_0$ at each
        point of the mesh. Where Ω_0 is the Euclidean volume form on $\mathbf{R}^m$

1: **procedure**
2:    $m \leftarrow$ dimension of the manifold

3: bivector ← a variable encoding the Poisson bivector field Π
4: function ← a (string) expression representing the scalar function f_0
5: mesh ← a (k,m) array encoding the mesh   ▷ k: number of points in the mesh
6: variable_1 ← modular_vf(bivector, function) ▷ modular_vf: function of PoissonGeometry
7: TRANSFORM each variable_1 item to a function that allows a numerical evaluation
8: **return** an array containing the evaluation of variable_1 at each m−array of mesh
9: **end procedure**

Example 2.7 The characteristic (symplectic) foliation of the following homogeneous Poisson bivector field on $\mathbf{R}_x^3$ coincides with that of $\Pi_{\mathfrak{so}(3)}$ in (2–3):

(2–13)
$$\Pi = \tfrac{1}{4}x_3\left(x_1^4 + x_2^4 + x_3^4\right)\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} - \tfrac{1}{4}x_2\left(x_1^4 + x_2^4 + x_3^4\right)\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3}$$
$$+ \tfrac{1}{4}x_1\left(x_1^4 + x_2^4 + x_3^4\right)\frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3}$$

However, $(\mathbf{R}^3, \Pi_{\mathfrak{so}(3)})$ admits a Hamiltonian–invariant volume form, while Π does not [48]. Consequently, the following vector field,

$$Z_\Pi = \left(x_2x_3^3 - x_2^3x_3\right)\frac{\partial}{\partial x_1} + \left(x_3x_1^3 - x_3^3x_1\right)\frac{\partial}{\partial x_2} + \left(x_1x_2^3 - x_1^3x_2\right)\frac{\partial}{\partial x_3},$$

which is the modular vector field of Π with respect to the Euclidean volume form on $\mathbf{R}_x^3$, cannot be a Hamiltonian vector field.

We can now use random meshes to check numerically that there are points for which the modular vector field of Π with respect to the Euclidean volume form is not zero, which implies that Π is not unimodular, as follows:

```
>>> npg3 = NumPoissonGeometry(3)                    # NumPoissonGeometry instance
>>> P = {(1, 2): '1/4*x3*(x1**4 + x2**4 + x3**4)', (1, 3): '- 1/4*x2* (x1**4 +
        x2**4 + x3**4)', (2, 3): '1/4*x1*(x1**4 + x2**4 + x3**4)'}
                                       # dictionary for Π in (2–13) according to (2–2)
>>> mesh = numpy.random.rand(10**6, 3)
       # # (10^6, 3) numpy array with random samples from a uniform distribution over [0,1)
>>> npg3.num_modular_vf(P, 1, mesh, pt_output=True)
                              # run num_modular_vf function with pt_output flag

tensor([ [[ 0.0538], [ 0.0545], [-0.1005]],  [[ 0.0031], [ 0.3838], [-0.0149]],
        [[-0.2559], [-0.0204], [ 0.0575]],  [[-0.0910], [ 0.0093], [ 0.1988]],
        ...,
        [[-0.0013], [-0.0095], [ 0.0107]],  [[-0.0467], [-0.0012], [0.0296]] ],
        dtype=torch.float64)
```

Observe that Z_Π is orthogonal to the radial vector field $x_i\partial/\partial x_i$ on $\mathbf{R}_x^3$, for $i = 1,2,3$. As the characteristic foliation of Π consists of concentric spheres (see, Figure 2.1), Z_Π is tangent to the symplectic leafs of Π.

2.11 Curl Operator

On an oriented manifold M with volume form Ω, the *divergence* of an a–multivector field A on M [35], and relative to Ω, is the unique $(a-1)$–multivector field $\mathcal{D}_\Omega(A)$ on M such that

$$(2\text{--}14) \qquad\qquad\qquad \mathbf{i}_{\mathcal{D}_\Omega(A)}\Omega = \mathrm{d}\mathbf{i}_A\Omega.$$

The function $\mathrm{num\_curl\_operator}$ evaluates the divergence of a multivector field on a mesh in $\mathbf{R}^m$. Let Ω_0 denote the standard volume form on $\mathbf{R}^m$.

Algorithm 2.8 $\mathrm{num\_curl\_operator}$(*multivector, function, mesh*)

Input: a multivector field A, a non–zero scalar function f_0 and a mesh
Output: evaluation of the the divergence of A with respect to the volume form $f_0\Omega_0$ at each point
   of the mesh.

1: **procedure**
2:    $m \leftarrow$ dimension of the manifold
3:    multivector $\leftarrow$ a variable encoding the multivector field A, or a (string) expression if
   degree $A = 0$
4:    function $\leftarrow$ a (string) expression representing the scalar function f_0
5:    mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
6:    variable_1 $\leftarrow$ curl_operator(multivector, function)
                                        ▷ curl_operator: function of PoissonGeometry
7:    TRANSFORM each variable_1 item to a function that allows a numerical evaluation
8:    **return** an array containing the evaluation of variable_1 at each m–array of mesh
9: **end procedure**

Example 2.8 The next Poisson bivector field on $\mathbf{R}^4_x$, has been applied to the analyze the orbital stability of the Pais–Uhlenbeck oscillator [3],

$$
\begin{aligned}
(2\text{--}15) \qquad \Pi =\;& 2x_4\frac{\partial}{\partial x_1}\wedge\frac{\partial}{\partial x_3} + 2x_3\frac{\partial}{\partial x_1}\wedge\frac{\partial}{\partial x_4} - 2x_4\frac{\partial}{\partial x_2}\wedge\frac{\partial}{\partial x_3} + 2x_3\frac{\partial}{\partial x_2}\wedge\frac{\partial}{\partial x_4} \\
& + (x_1 - x_2)\frac{\partial}{\partial x_3}\wedge\frac{\partial}{\partial x_4}.
\end{aligned}
$$

It is unimodular on (the whole of) $\mathbf{R}^4_x$, and has trivial modular vector field with respect to the Euclidean volume form. For a fixed volume form, the divergence of a Poisson bivector field coincides with its modular vector field. Hence, the unimodularity of Π (2–15) may be numerically verified, at least in a sample, using random meshes. In this example the output is a Pytorch tensor:

```
>>> npg4 = NumPoissonGeometry(4)              # NumPoissonGeometry instance
>>> P = {(1, 3): '2*x4', (1, 4): '2*x3', (2, 3): '-2*x4', (2, 4): '2*x3',
         (3, 4): 'x1 - x2'}
                                   # dictionary for Π in (2–15) according to (2–2)
>>> mesh = numpy.random.rand(10**6, 4)
         # # (10^6, 4) numpy array with random samples from a uniform distribution over [0, 1)
```

```
>>> npg4.num_curl_operator(P, 1, mesh, pt_output=True)
                                    # run num_curl_operator function with pt_output flag

tensor([ [[0.], [0.], [0.], [0.]],  [[0.], [0.], [0.], [0.]],  [[0.], [0.], [0.], [0.]],  ...,
        [[0.], [0.], [0.], [0.]],  [[0.], [0.], [0.], [0.]],  [[0.], [0.], [0.], [0.]] ],
        dtype=torch.float64)
```

2.12 Differential 1–Forms Bracket

In the context of Lie algebroids, the Koszul bracket of 1–forms is the Lie bracket on the space of sections of a Poisson Lie algebroid [35, 25].

By definition and (2–10), the Lie bracket of two differential 1–forms α and β, induced by a Poisson bivector field Π on M, at a point in M can be determined by the following (coordinate) formula:

$$\{\alpha, \beta\}_\Pi(p) \simeq \left[(\mathrm{d}\beta)_{kl}(p)\right]\left[\Pi^{ij}(p)\right]\left[\alpha_r(p)\right] - \left[(\mathrm{d}\alpha)_{kl}(p)\right]\left[\Pi^{ij}(p)\right]\left[\beta_r(p)\right]$$
$$- \left[\tfrac{\partial}{\partial x^r}\left([\beta_k]^\top\left[\Pi^{ij}\right][\alpha_l]\right)(p)\right], \qquad p \in M; \quad k, l, r = 1, \ldots, m$$

Here, $[\Pi^{ij}]$ is the matrix of Π (2–4), $[(\mathrm{d}\alpha)_{kl}]$ and $[(\mathrm{d}\beta)_{kl}]$ are the matrix of the differential 2–forms $\mathrm{d}\alpha = 1/2(\mathrm{d}\alpha)_{kl}\mathrm{d}x^k \wedge \mathrm{d}x^l$ and $\mathrm{d}\beta = 1/2(\mathrm{d}\beta)_{kl}\mathrm{d}x^k \wedge \mathrm{d}x^l$, $[\alpha_r]$ and $[\beta_r]$ are the coefficient vectors of $\alpha = \alpha_k\mathrm{d}x^k$ and $\beta = \beta_k\mathrm{d}x^k$, and $[\partial/\partial x^r(\cdot)]$ is the gradient vector (field) operator.

The function $\mathrm{num\_one\_forms\_bracket}$ evaluates the differential 1–form $\{\alpha, \beta\}_\Pi$ on a mesh in $\mathbf{R}^m$.

Algorithm 2.9 num_one_forms_bracket(*bivector, one_form_1, one_form_2, mesh*)

Input: a Poisson bivector field Π, two differential 1–forms α, β and a mesh
Output: evaluation of the Lie bracket of α and β induced by Π at each point of the mesh

---

1: **procedure**
2:     $m \leftarrow$ dimension of the manifold
3:     bivector $\leftarrow$ a variable encoding the Poisson bivector field Π
4:     one_form_1 $\leftarrow$ a variable encoding the differential 1–form $\alpha = \alpha_1\mathrm{d}x^1 + \cdots + \alpha_m\mathrm{d}x^m$
5:     one_form_2 $\leftarrow$ a variable encoding the differential 1–form $\beta = \beta_1\mathrm{d}x^1 + \cdots + \beta_m\mathrm{d}x^m$
6:     mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ *k*: number of points in the mesh
7:     variable_1 $\leftarrow$ a container with items $(\alpha_1, \ldots, \alpha_m)$
8:     variable_2 $\leftarrow$ a container with items $(\beta_1, \ldots, \beta_m)$
9:     **for** $i \in \{1, 2\}$ **do**
10:         CONVERT each item in variable_i to a symbolic variable
11:         variable_3_i $\leftarrow$ the (symbolic) Jacobian matrix of variable_i
                    ▷ variable_i thought as a vector field
12:         TRANSFORM each variable_3_i item to a function that allows a numerical evaluation
13:         variable_4_i $\leftarrow$ an array containing the evaluation of variable_3_i at each *m*–array of mesh

---

14:    variable_5_j ← num_sharp_morphism(bivector, one_form_j, mesh)
                                         ▷ $i \neq j = 1, 2$. See Algorithm 2.2

15: **end for**

16: variable_6, variable_7 ← empty containers

17: **for** $0 \leq i < k$ **do**

18:    variable_6[i] ← $\big($variable_4_1 − transpose(variable_4_1)$\big)$[i] ∗ variable_5_2[i]

19:    variable_7[i] ← $\big($variable_4_2 − transpose(variable_4_2)$\big)$[i] ∗ variable_5_1[i]
                                          ▷ matrix–vector products

20: **end for**

21: variable_8 ← sharp_morphism(bivector, one_form_1)
                             ▷ sharp_morphism: function of PoissonGeometry

22: variable_9 ← a container with all the *values* available in the dictionary variable_8

23: variable_10 ← variable_9 ∗ variable_2
                                         ▷ scalar vector product

24: variable_11 ← the (symbolic) gradient vector field of variable_10

25: TRANSFORM each variable_11 item to a function that allows a numerical evaluation

26: variable_12 ← an array containing the evaluation of variable_11 at each m−array of mesh

27: variable_13 ← an empty container

28: **for** $0 \leq i < k$ **do**

29:    variable_13[i] ← variable_7[i] − variable_6[i] + variable_12[i]
                                 ▷ vector sum

30: **end for**

31: **return** variable_13

32: **end procedure**

Example 2.9 By definition, the Lie bracket induced by the Poisson bivector field Π in (2–9) [38], see also [53], of the (basic) differential 1–forms dx_5 and dx_6 is given by

$$\{dx_5, dx_6\}_\Pi = 2x_2 dx_2.$$

Hence, $\{dx_5, dx_6\}_\Pi = 2dx_2$ at points $x \in \mathbf{R}^6$ such that $x_2 = 1$. This coincides with the following computation involving random meshes,

$$\{a_1, b_1\} \times \{1\} \times \{a_2, b_2\} \times \cdots \times \{a_5, b_5\}, \qquad a_i, b_i \in \text{random}\,[0, 1]:$$

```
>>> npg6 = NumPoissonGeometry(6)                    # NumPoissonGeometry instance
>>> P = {(1, 4): 1, (2, 5): 1, (3, 6): 1, (5, 6): 'x2**2'}
                                       # dictionary for Π in (2–9) according to (2–2)
>>> alpha = {(5,): 1}                           # dictionary for dx5 according to (2–2)
>>> beta = {(6,): 1}                            # dictionary for dx6 according to (2–2)
>>> npg6.num_one_forms_bracket(P, alpha, beta, mesh, pt_output=True)
                                # run num_one_forms_bracket function with pt_output flag
```

tensor([[[0.], [2.], [0.], [0.], [0.], [0.]], [[0.], [2.], [0.], [0.], [0.], [0.]],

       ...,

       [[0.], [2.], [0.], [0.], [0.], [0.]], [[0.], [2.], [0.], [0.], [0.], [0.]],

       [[0.], [2.], [0.], [0.], [0.], [0.]], [[0.], [2.], [0.], [0.], [0.], [0.]]],

       dtype=torch.float64)

2.13 Gauge Transformations

Gauge transformations are used to simplify dynamical equations, they aid in reduction methods for dynamical systems. They are also used for the Hamiltonization of nonholonomic systems [4]. Recall that the *Hamiltonization problem* consists in determining conditions that represent a dynamical system in Hamiltonian form.

Given a differential 2–form λ on M, if the morphism $\mathrm{id}_{T^*M} - \lambda^\flat \circ \Pi^\sharp$ is invertible, the λ–gauge transformation of a bivector field Π on M is the bivector field $\overline{\Pi}$ determined by the vector bundle morphism $\overline{\Pi}^\sharp = \Pi^\sharp \circ (\mathrm{id}_{T^*M} - \lambda^\flat \circ \Pi^\sharp)^{-1}$ [53]. This morphism can be evaluated at a point in M as follows:

$$\overline{\Pi}_p^\sharp \simeq -\left[\Pi^{ij}(p)\right] \left(\mathrm{Id} - \left[\lambda_{kl}(p)\right]\left[\Pi^{ij}(p)\right]\right)^{-1}, \qquad p \in M; \quad k, l = 1, \ldots, m$$

Here, $[\Pi^{ij}]$ is the matrix of Π (2–4), $[\lambda_{kl}]$ is the matrix of $\lambda = 1/2\lambda_{kl}\mathrm{d}x^k \wedge \mathrm{d}x^l$ and Id denotes the $m \times m$ identity matrix. The morphism $\lambda^\flat : TM \to T^*M$ above is given by $X \mapsto \mathbf{i}_X\lambda$.

The function num_gauge_transformation evaluates the gauge transformation of a bivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.10 num_gauge_transformation(*bivector, two_form, mesh*)

Input: a (Poisson) bivector field Π, a differential 2–form λ and a mesh
Output: evaluation of the gauge transformation of Π induced by λ at each point of the mesh

---

1: **procedure**
2:     $m \leftarrow$ dimension of the manifold
3:     bivector $\leftarrow$ a variable encoding the (Poisson) bivector field Π
4:     two_form $\leftarrow$ a variable encoding the (Poisson) the differential 2–form λ
5:     mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
6:     variable_1 $\leftarrow$ num_bivector_to_matrix(bivector, mesh)   ▷ see Algorithm 2.2
7:     variable_2 $\leftarrow$ num_bivector_to_matrix(two_form, mesh)   ▷ see Algorithm 2.2
8:     variable_3 $\leftarrow$ the identity $m \times m$ matrix
9:     variable_4 $\leftarrow$ an empty container
10:     **for** $0 \leq i < k$ **do**
11:         variable_4[i] $\leftarrow$ variable_3 $-$ variable_2[i] $*$ variable_1[i]

                                        ▷ matrix sum/multiplication
12:     **end for**
13:     variable_5 $\leftarrow$ an empty container

```
14:      for 0 ≤ i < k do
15:          if determinant(variable_4[i]) ≠ 0 then
16:              variable_5[i] ← variable_1[i] * inverse(variable_4[i])          ▷ matrix multiplication
17:          else
18:              variable_5[i] ← False
19:          end if
20:      end for
21:      return variable_5
22: end procedure
```

Example 2.10   For this example we will use the following result that we proved in [18]:

Proposition 1 *Let Π be a bivector field on a 3–dimensional smooth manifold M. Then, given a differential 2–form λ on M, the λ–gauge transformation $\overline{\Pi}$ of Π is well defined on the open subset,*

(2–16) $$\{F := 1 + \langle \lambda, \Pi \rangle \neq 0\} \subseteq M.$$

Moreover, $\overline{\Pi}$ is given by

$$\overline{\Pi} = \tfrac{1}{F}\Pi.$$

In consequence, if Π is Poisson, then $\overline{\Pi}$ is also Poisson.

Observe that if $\langle \lambda, \Pi \rangle = 0$ in (2–16), then Π remains unchanged under the gauge transformation induced by λ. This holds for the Poisson bivector field $\Pi_{\mathfrak{so}(3)}$ in (2–3) [7] and the differential 2–form on $\mathbf{R}^3$ given by

(2–17) $$\lambda = (x_1 - x_2)\mathrm{d}x_1 \wedge \mathrm{d}x_2 + (x_1 - x_3)\mathrm{d}x_1 \wedge \mathrm{d}x_3 + (x_2 - x_3)\mathrm{d}x_2 \wedge \mathrm{d}x_3.$$

Then, for Q^3 being as in (2–1), we can check the invariance of $\Pi_{\mathfrak{so}(3)}$ under λ as follows:

```
>>> npg3 = NumPoissonGeometry(3)                          # NumPoissonGeometry instance
>>> P_so3 = {(1, 2): 'x3', (1, 3): '-x2', (2, 3): 'x1'}
                                             # dictionary for Π_so(3) in (2–3) according to (2–2)
>>> lambda = {(1, 2): 'x1 - x2', (1, 3): 'x1 - x3', (2, 3): 'x2 - x3'}
                                             # dictionary for λ in (2–17) according to (2–2)
>>> npg3.num_gauge_transformation(P_so3, lambda, Qmesh, pt_output=True)
                            # run num_gague_transformation function with pt_output flag

tensor([ [[0., 0.,  0.], [0., 0., 0.], [0.,  0., 0.]],  [[0., 1.,  0.], [-1., 0., 0.], [0.,  0., 0.]],
         [[0., 0., -1.], [0., 0., 0.], [1.,  0., 0.]],  [[0., 1., -1.], [-1., 0., 0.], [1.,  0., 0.]],
         [[0., 0.,  0.], [0., 0., 1.], [0., -1., 0.]],  [[0., 1.,  0.], [-1., 0., 1.], [0., -1., 0.]],
         [[0., 0., -1.], [0., 0., 1.], [1., -1., 0.]],  [[0., 1., -1.], [-1., 0., 1.], [1., -1., 0.]] ],
       dtype=torch.float64)
```

Notice that the output is the same as the second one in Example 2.1, which encodes the evaluation of the bivector field $\Pi_{\mathfrak{so}(3)}$ at points of Q^3.

2.14   Lie–Poisson Normal Forms on $\mathbf{R}^3$

Two Poisson bivector fields Π and $\widetilde{\Pi}$ on M are said to be equivalent (or isomorphic) if there exists a diffeomorphism $F : M \to M$ such that $\widetilde{\Pi} = F^*\Pi$. Under this equivalence relation there exist 9 non–trivial normal forms of Lie–Poisson bivector fields on $\mathbf{R}^3$ [43, 54].

The function $\mathrm{num\_linear\_normal\_form\_R3}$ evaluates a normal form of a given Lie–Poisson bivector field on a mesh in $\mathbf{R}^3$.

Algorithm 2.11   num_linear_normal_form_R3(*linear_bivector, mesh*)

Input: a Lie–Poisson bivector field Π on $\mathbf{R}^3$ and a mesh
Output: evaluation of a normal form of Π at each point of the mesh

---

1: **procedure**
2:      $m \leftarrow$ dimension of the manifold
3:      linear_bivector $\leftarrow$ a variable that encodes the Lie–Poisson bivector field Π
4:      mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
5:      variable_1 $\leftarrow$ linear_normal_form_R3(linear_bivector)
                                                                    ▷ linear_normal_form_R3: function of PoissonGeometry
6:      **return** num_bivector(variable_1, mesh)   ▷ see Algorithm 2.1
7: **end procedure**

Example 2.11   Using the function $\mathrm{linear\_normal\_form\_R3}$ of PoissonGeometry we can verify that the Lie–Poisson bivector field on $\mathbf{R}^3_x$

$$(2\text{–}18) \qquad \Pi = 2(x_2 + x_3)\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} + (x_1 - x_2)\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} + (x_1 + x_2 + 2x_3)\frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3},$$

admits the following Poisson bivector field as a normal form:

$$(2\text{–}19) \qquad \Pi_N = (x_1 - 4ax_2)\frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} + (4ax_1 + x_2)\frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3}, \qquad a > 0.$$

To evaluate this normal form of Π at points of Q^3 (2–1) we compute:

```
>>> npg3 = NumPoissonGeometry(3)                    # NumPoissonGeometry instance
>>> P = {(1, 2): '2*(x2 + x3)', (1, 3): 'x1 - x2', (2, 3): 'x1 + x2 +2*x3'}
                                        # dictionary for Π in (2–18) according to (2–2)
>>> npg3.num_linear_normal_form_R3(P, Qmesh)
                                        # run num_linear_normal_form function

[{(1,3):  0.0, (2,3): 0.0}, {(1,3):  0.0, (2,3): 0.0}, {(1,3): -4.0*a, (2,3): 1.0},
 {(1,3): -4.0*a, (2,3): 1.0}, {(1,3): 1.0, (2,3): 4.0*a}, {(1,3): 1.0, (2,3): 4.0*a},
 {(1,3): 1.0-4.0*a, (2,3): 4.0*a+1.0},  {(1,3): 1.0-4.0*a, (2,3): 4.0*a+1.0}]
```

The equivalence between Π (2–18) and Π_N (2–19) implies that the characteristic foliation of Π is a open book foliation [20], as shown in Figure 2.3 below. In particular, Π does not admit global, non–constant, Casimir functions.
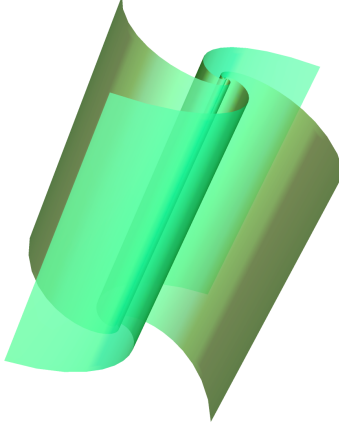
Figure 2.3: Symplectic (open book) foliation of Π (2–18).

2.15 Flaschka–Ratiu Poisson Bivector Field

If M is a oriented manifold with volume form Ω, the Poisson bivector field Π determined by $m - 2$ prescribed Casimir functions $K_1, ..., K_{m-2}$ on M, and defined by the formula

$$\mathbf{i}_\Pi \Omega = \mathrm{d}K_1 \wedge \cdots \wedge \mathrm{d}K_{m-2},$$

is called Flaschka–Ratiu bivector field [24, 13]. Observe that Π is non–trivial on the open subset of M where $K_1, \ldots, K_{m-2}$ are (functionally) independent.

The function $\mathrm{num\_flaschka\_ratiu\_bivector}$ evaluates a Flaschka–Ratiu bivector field on a mesh in $\mathbf{R}^m$.

Algorithm 2.12 num_flaschka_ratiu_bivector(*casimirs_list, mesh*)

Input: a set of scalar functions and a mesh
Output: evaluation of the Flaschka–Ratiu bivector field induced by these functions at each point
of the mesh

1: **procedure**
2:     $m \leftarrow$ dimension of the manifold
3:     casimirs_list $\leftarrow$ a container with $m - 2$ (string) expressions representing the set of scalar functions   ▷ each string expression represents a scalar function
4:     mesh $\leftarrow$ a (k, m) array encoding the mesh   ▷ k: number of points in the mesh
5:     variable_1 $\leftarrow$ flaschka_ratiu_bivector(functions_set)
                                                    ▷ flaschka_ratiu_bivector: function of PoissonGeometry
6:     **return** num_bivector(variable_1, mesh)   ▷ see Algorithm 2.1
7: **end procedure**

Example 2.12 Consider the following (Poisson) Flaschka–Ratiu bivector field on $\mathbf{R}^4_x$

$$\Pi = x_3 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} - x_2 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} - x_1 \frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3},$$

that appears as a local model around singularities of a broken Lefschetz fibration on smooth 4–manifolds [22] (see, also [55, 17]). It is induced by the functions

(2–20) $$K_1 = \tfrac{1}{2}x_4, \quad K_2 = -x_1^2 + x_2^2 + x_3^2.$$

To evaluate Π at points of Q^4 (2–1) we compute:

```
>>> npg4 = NumPoissonGeometry(4)                    # NumPoissonGeometry instance
>>> functions = ['1/2*x4', '-x1**2 + x2**2 + x3**2']
                  # list containing string expressions for K1 and K2 in (2–20), in that order
>>> npg4.num_flaschka_ratiu_bivector(functions, Qmesh)
                                      # run num_flaschka_ratiu_bivector function
```

[{(1, 2): 0.0, (1, 3): 0.0, (2, 3): 0.0}, {(1, 2): 0.0, (1, 3): 0.0, (2, 3): 0.0},
 {(1, 2): 1.0, (1, 3): 0.0, (2, 3): 0.0}, {(1, 2): 1.0, (1, 3): 0.0, (2, 3): 0.0},
 {(1, 2): 0.0, (1, 3): -1.0, (2, 3): 0.0}, {(1, 2): 0.0, (1, 3): -1.0, (2, 3): 0.0},
 {(1, 2): 1.0, (1, 3): -1.0, (2, 3): 0.0}, {(1, 2): 1.0, (1, 3): -1.0, (2, 3): 0.0},
 {(1, 2): 0.0, (1, 3): 0.0, (2, 3): -1.0}, {(1, 2): 0.0, (1, 3): 0.0, (2, 3): -1.0},
 {(1, 2): 1.0, (1, 3): 0.0, (2, 3): -1.0}, {(1, 2): 1.0, (1, 3): 0.0, (2, 3): -1.0},
 {(1, 2): 0.0, (1, 3): -1.0, (2, 3): -1.0}, {(1, 2): 0.0, (1, 3): -1.0, (2, 3): -1.0},
 {(1, 2): 1.0, (1, 3): -1.0, (2, 3): -1.0}, {(1, 2): 1.0, (1, 3): -1.0, (2, 3): -1.0}]

3   Algorithmic Complexity and Performance

In this section we will present an *approximation* to the (worst–case) time complexity of the twelve algorithms in Table 1.1, as well as a time performance analysis of our Python implementation of the functions in the module NumPoissonGeometry.

3.1   Complexity

The time complexity of certain algorithms depends on the nature and structure of the input data. In our methods two important processes in the implementation depend on the *length* of the input data items: converting string expressions to symbolic variables, and transforming these symbolic expressions into functions that allow a numerical evaluation. Therefore, for the analysis of our algorithms we define,

$$|C| := \max_{\text{for } x \text{ in } C} \{\text{len}(x)\},$$

for a container C with (string expression) items that encode the coefficients (scalar functions) of the coordinate expression of a multivector field or a differential form. For example, as illustrated

in (2–2), we use dictionaries for our Python implementation. In this case, C is a tuple (or list) that contains all the values of a such dictionary.

In the following Table 3.1 we record the approximate worst–case time complexities for the twelve methods of our NumPoissonGeometry module:

Table 3.1: Worst–case time complexity of NumPoissonGeometry methods. In the second column: m denotes the dimension of $\mathbf{R}^m$, k is the number of points in a mesh on $\mathbf{R}^m$, and $[\cdot]$ is the integer part function.

Method	Time Complexity
1. num_bivector_field	$\mathcal{O}(m^2 k \lvert bivector \rvert)$
2. num_bivector_to_matrix	$\mathcal{O}(m^2 k \lvert bivector \rvert)$
3. num_hamiltonian_vf	$\mathcal{O}(mk(m \lvert bivector \rvert + \mathrm{len}(ham\_function)))$
4. num_poisson_bracket	$\mathcal{O}(mk(m \lvert bivector \rvert + \mathrm{len}(function\_1) + \mathrm{len}(function\_2)))$
5. num_sharp_morphism	$\mathcal{O}(m(mk \lvert bivector \rvert + \lvert one\_form \rvert))$
6. num_coboundary_operator	$\mathcal{O}(\mathrm{comb}(m, [m/2]) \lvert bivector \rvert \mathrm{len}(function)(m^5 + k))$
7. num_modular_vf	$\mathcal{O}(\mathrm{comb}(m, [m/2]) \lvert bivector \rvert \mathrm{len}(function)(m + k))$
8. num_curl_operator	$\mathcal{O}(\mathrm{comb}(m, [m/2]) \lvert multivector \rvert \mathrm{len}(function)(m + k))$
9. num_one_forms_bracket	$\mathcal{O}(m^2 k \lvert bivector \rvert \lvert one\_form\_1 \rvert \lvert one\_form\_2 \rvert)$
10. num_gauge_transformation	$\mathcal{O}\big(m^2 k(m^5 + \lvert bivector \rvert + \lvert two\_form \rvert)\big)$
11. num_linear_normal_form_R3	$\mathcal{O}(k \lvert bivector \rvert)$
12. num_flaschka_ratiu_bivector	$\mathcal{O}(m^6 k \lvert bivector \rvert)$

Remark 3.1 Observe that the time complexities in Table 3.1 depend linearly on the number of points in the mesh (k).

Recall that the time complexity of independent processes equals the sum of the respective time complexities of each process. With this in mind, we present the deduction of the time complexities of our numerical methods presented in Table 3.1.

3.1.1 Polynomial Complexity

The following methods have polynomial time complexities.

Lemma 3.2 *The time complexity of the* num_bivector_field *method is approximately* $\mathcal{O}(m^2 k \lvert bivector \rvert)$.

Proof Consider the Algorithm 2.1 with *bivector* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 5 → $\mathcal{O}(m^2|bivector|)$: we iterate over *bivector*, and the transformation of each *bivector* item depends on its length.

ii. Line 6 → $\mathcal{O}(km^2|bivector|)$: we iterate over *mesh*, and the evaluation of *bivector* depends on its length and on the length of its items.

Hence, the time complexity of num_bivector_field is $\mathcal{O}(m^2k|bivector|)$. □

Lemma 3.3 *The time complexity of the* num_bivector_to_matrix *method is approximately* $\mathcal{O}(m^2k|bivector|)$.

Proof Consider the Algorithm 2.2 with *bivector* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 5 → $\mathcal{O}(m^2|bivector|)$: the bivector_to_matrix method has time complexity $\mathcal{O}(m^2|bivector|)$.

ii. Line 6 → $\mathcal{O}(m^2|bivector|)$: we iterate over the $m \times m$ matrix *variable_1*, and the transformation of each *bivector* item depends on its length.

iii. Line 7 → $\mathcal{O}(km^2|bivector|)$: we iterate over *mesh*, and the evaluation of *variable_1* depends on its dimension and on the length of the *bivector* items.

Hence, the time complexity of num_bivector_to_matrix is $\mathcal{O}(m^2k|bivector|)$. □

Lemma 3.4 *The time complexity of the* num_hamiltonian_vf *method is approximately* $\mathcal{O}(mk(m|bivector|+$ len(*ham_function*)))*.*

Proof Consider the Algorithm 2.3 with *bivector*, *ham_function* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 6 → $\mathcal{O}(\text{len}(ham\_function))$: converting *ham_function* to a symbolic expression depends on its length.

ii. Line 7 → $\mathcal{O}(m\text{len}(ham\_function))$: calculate the gradient of *ham_function* depends on the length of this symbolic expression and on the dimension of $\mathbf{R}^m$.

iii. Line 8 → $\mathcal{O}(m\text{len}(ham\_function))$: we iterate over the $m \times 1$ matrix *variable_1*, and the transformation of each of its items depends on the length of *ham_function*.

iv. Line 9 → $\mathcal{O}(km\text{len}(ham\_function))$: we iterate over *mesh*, and the evaluation of *variable_1* depends on its dimension and on the length of *ham_function*.

v. Line 10 → $\mathcal{O}(m^2k|bivector|)$: the num_bivector_to_matrix method has time complexity $\mathcal{O}(m^2k|bivector|)$.

vi. Lines 12-14 → $\mathcal{O}(km^2)$: we iterate over the set of indices $\{0,\ldots,k-1\}$, and the product of the *variable_2* and *variable_3* items has time complexity $\mathcal{O}(m^2)$.

Hence, the time complexity of num_hamiltonian_vf is

$$\mathcal{O}(mk(m|bivector| + \text{len}(ham\_function))).$$

□

Lemma 3.5 *The time complexity of the* num_poisson_bracket *method is approximately* $\mathcal{O}(mk(m|bivector|+ \text{len}(function\_1) + \text{len}(function\_2)))$.

Proof Consider the Algorithm 2.4 with *bivector*, *function_1*, *function_2* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 9 → $\mathcal{O}(\text{len}(function\_2))$: converting *function_2* to a symbolic expression depends on its length.

ii. Line 10 → $\mathcal{O}(m\text{len}(function\_2))$: calculate the gradient of *function_2* depends on the length of this symbolic expression and on the dimension of $\mathbf{R}^m$.

iii. Line 11 → $\mathcal{O}(m\text{len}(function\_2))$: we iterate over the $m \times 1$ matrix *variable_1*, and the transformation of each of its items depends on the length of *function_2*.

iv. Line 12 → $\mathcal{O}(km\text{len}(function\_2))$: we iterate over *mesh*, and the evaluation of *variable_1* depends on its dimension and on the length of *function_2*.

v. Line 13 → $\mathcal{O}(mk(m|bivector| + \text{len}(function\_1)))$: as num_hamiltonian_vf has time complexity $\mathcal{O}(mk(m|bivector| + \text{len}(ham\_function)))$.

vi. Lines 15-17 → $\mathcal{O}(km^2)$: we iterate over the set of indices $\{0, \ldots, k-1\}$, and the product of the *variable_2* and *variable_3* items has time complexity $\mathcal{O}(m^2)$.

Hence, the time complexity of num_poisson_bracket is

$$\mathcal{O}(mk(m|bivector| + \text{len}(function\_1) + \text{len}(function\_2))).$$

$\square$

Lemma 3.6 *The time complexity of the* num_sharp_morphism *method is approximately*

$$\mathcal{O}(m(mk|bivector| + |one\_form|)).$$

Proof Consider the Algorithm 2.5 with *bivector*, *one_form* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 7 → $\mathcal{O}(m|one\_form|)$: we iterate over the $(m, 1)$ container *variable_1*, and the transformation of each of its items depends on the length of *one_form* items.

ii. Line 9 → $\mathcal{O}(m^2k|bivector|)$: the num_bivector_to_matrix method has time complexity $\mathcal{O}(m^2k|bivector|)$.

iii. Lines 11-13 → $\mathcal{O}(km^2)$: we iterate over the set of indices $\{0, \ldots, k-1\}$, and the product of the *variable_2* and *variable_3* items has time complexity $\mathcal{O}(m^2)$.

Hence, the time complexity of num_sharp_morphism is

$$\mathcal{O}(m(mk|bivector| + |one\_form|)).$$

$\square$

Lemma 3.7 *The time complexity of the* num_one_forms_bracket *method is approximately*

$$\mathcal{O}(m^2k|bivector||one\_form\_1||one\_form\_2|).$$

Proof   Consider the Algorithm 2.9 with *bivector*, *one_form_1*, *one_form_2* and *mesh* inputs. The time complexity of our implementation depends on:

   i.   Line 10 $\rightarrow$ $\mathcal{O}(\text{len}(one\_form\_1)) + \text{len}(one\_form\_2))$: converting *variable_i* to a symbolic expression depends on the length of *one_form_1* and *one_form_2* items.

  ii.   Line 11 $\rightarrow$ $\mathcal{O}(m^2(|one\_form\_1| + |one\_form\_2|))$: calculate the Jacobian matrix of *variable_i* depends on the length of *one_form_1* and *one_form_2* items and on the dimension of $\mathbf{R}^m$.

 iii.   Line 12 $\rightarrow$ $\mathcal{O}(m^2(|one\_form\_1| + |one\_form\_2|))$: we iterate over the $m \times m$ matrix *variable_3_i*, and the transformation of each of its items depends on the length of *one_form_1* and *one_form_2* items.

  iv.   Line 14 $\rightarrow$ $\mathcal{O}(m(mk|bivector| + |one\_form\_1| + |one\_form\_2|))$: the num_sharp_ morphism method has time complexity $\mathcal{O}(m(mk|bivector| + |one\_form|))$.

   v.   Lines 17-19 $\rightarrow$ $\mathcal{O}(km^2)$: we iterate over the set of indices $\{0, \dots, k-1\}$, and the product of the *variable_4_i* and *variable_5_j* items has time complexity $\mathcal{O}(m^2)$, for $i, j = 1, 2$.

  vi.   Line 21 $\rightarrow$ $\mathcal{O}(m^2|bivector||one\_form\_1|)$: the sharp_morphism method has time complexity $\mathcal{O}(m^2$
$|bivector||one\_form|)$.

 vii.   Line 24 $\rightarrow$ $\mathcal{O}(m|one\_form\_1||one\_form\_2|)$: calculate the gradient of *variable_10* depends on the length of *one_form_1* and *one_form_2* items, and on the dimension of $\mathbf{R}^m$.

viii.   Line 25 $\rightarrow$ $\mathcal{O}(m(|one\_form\_1||one\_form\_2|))$: we iterate over the $m \times 1$ matrix *variable_11*, and the transformation of each of its items depends on the length of *one_form_1* and *one_form_2* items.

  ix.   Lines 28-30 $\rightarrow$ $\mathcal{O}(km)$: we iterate over the set of indices $\{0, \dots, k-1\}$, and the sum of variables *variable_6*, *variable_7* and *variable_12* items has time complexity $\mathcal{O}(m)$.

Hence, the time complexity of num_one_forms_bracket is

$$\mathcal{O}(m^2 k |bivector||one\_form\_1||one\_form\_2|).$$

<div align="right">□</div>

Lemma 3.8   *The time complexity of the* num_gauge_transformation *method is approximately*

$$\mathcal{O}(m^2 k (m^5 + |bivector| + |two\_form|)).$$

Proof   Consider the Algorithm 2.10 with *bivector*, *two_form* and *mesh* inputs. The time complexity of our implementation depends on:

   i.   Line 6 $\rightarrow$ $\mathcal{O}(m^2 k |bivector|)$: the num_bivector_to_matrix method has time complexity $\mathcal{O}(m^2 k |bivector|)$.

  ii.   Line 7 $\rightarrow$ $\mathcal{O}(m^2 k |two\_form|)$: the num_bivector_to_matrix method has complexity $\mathcal{O}(m^2 k |bivector|)$.

 iii.   Lines 10-12 $\rightarrow$ $\mathcal{O}(km^2)$: we iterate over the set of indices $\{0, \dots, k-1\}$, and the matrix operations between the *variable_1*, *variable_2* and *variable_3* items has time complexity $\mathcal{O}(m^2)$.

iv. Lines 14-20 $\to \mathcal{O}(km^{10})$: we iterate over the set of indices $\{0, \ldots, k-1\}$, calculate the determinant of each item of *variable_4* has time complexity $\mathcal{O}(m^4)$, the inverse has time complexity $\mathcal{O}(m^3)$, and the matrix product in line 16 has time complexity $\mathcal{O}(m^3)$.

Hence, the time complexity of num_gauge_transformation is $\mathcal{O}(m^2 k(m^5 + |bivector| + |two\_form|))$.

$\square$

Lemma 3.9 *The time complexity of the num_linear_normal_form_R3 method is approximately* $\mathcal{O}(k|bivector|)$.

Proof Consider the Algorithm 2.11 with *linear_bivector* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line $5 \to \mathcal{O}(|bivector|)$: the linear_normal_form_R3 method has time complexity $\mathcal{O}(|bivector|)$.

ii. Line $6 \to \mathcal{O}(k|bivector|)$: the num_bivector_field method has time complexity $\mathcal{O}(m^2 k|bivector|)$.

Hence, the time complexity of num_linear_normal_form_R3 is $\mathcal{O}(k|bivector|)$. $\square$

Lemma 3.10 *The time complexity of the num_flaschka_ratiu_bivector method is approximately*

$$\mathcal{O}(m^6 k|bivector|).$$

Proof Consider the Algorithm 2.12 with *casimir_list* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line $5 \to \mathcal{O}(m^6|casimir\_list|)$: the flaschka_ratiu_bivector method has complexity $\mathcal{O}(m^6|casimir\_list|)$.

ii. Line $6 \to \mathcal{O}(m^2 k|casimir\_list|)$: the num_bivector_field method has time complexity $\mathcal{O}(m^2 k|bivector|)$.

Hence, the time complexity is $\mathcal{O}(m^6 k|bivector|)$. $\square$

3.1.2 Exponential Complexity

Due to the nature of the concatenated loops in our methods, and because we need to calculate ordered index permutations, the following methods have exponential time complexities.

Lemma 3.11 *The time complexity of the num_coboundary_operator method is approximately*

$$\mathcal{O}(\text{comb}(m, [m/2])|bivector|\text{len}(function)(m^5 + k)).$$

Proof Consider the Algorithm 2.6 with *bivector*, *multivector* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line $6 \to \mathcal{O}(m^5 \text{comb}(m, [m/2])|bivector||multivector|)$: the lichnerowicz_poisson_ operator method has time complexity

$$\mathcal{O}(m^5 \text{comb}(m, [m/2])|bivector||multivector|).$$

ii. Line 7 → $\mathcal{O}(\text{comb}(m, [m/2])|bivector||multivector|)$: we iterate over *variable_1*, and the transformation of each of its items depends on the length of the *bivector* and *multivector* items.

iii. Line 8 → $\mathcal{O}(k\text{comb}(m, [m/2])|bivector||multivector|)$: we iterate over *mesh*, and the evaluation of *variable_1* depends on its length and on the length of the *bivector* and *multivector* items.

Hence, the time complexity of num_coboundary_operator is

$$\mathcal{O}(\text{comb}(m, [m/2])|bivector|\text{len}(function)(m^5 + k)).$$

□

Lemma 3.12 *The time complexity of the* num_modular_vf *method is approximately*

$$\mathcal{O}(\text{comb}(m, [m/2])|bivector|\text{len}(function)(m + k)).$$

Proof Consider the Algorithm 2.7 with *bivector*, *function* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 6 → $\mathcal{O}(m\,\text{comb}(m, [m/2])|bivector|\text{len}(function))$: the modular_vf method has time complexity

$$\mathcal{O}(m\,\text{comb}(m, [m/2])|bivector|\text{len}(function)).$$

ii. Line 7 → $\mathcal{O}(\text{comb}(m, [m/2])|bivector|\text{len}(function))$: we iterate over *variable_1*, and the transformation of each of its items depends on the length of the *bivector* items and the *function* string expression.

iii. Line 8 → $\mathcal{O}(k\text{comb}(m, [m/2]|bivector|\text{len}(function)))$: we iterate over *mesh*. The evaluation of *variable_1* depends on its length, and on the length of the *bivector* items and the *function* string expression.

Hence, the time complexity of num_modular_vf is

$$\mathcal{O}(\text{comb}(m, [m/2])|bivector|\text{len}(function)(m + k)).$$

□

Lemma 3.13 *The time complexity of the* num_curl_operator *method is approximately*

$$\mathcal{O}(\text{comb}(m, [m/2])|multivector|\text{len}(function)(m + k)).$$

Proof Consider the Algorithm 2.8 with *bivector*, *function* and *mesh* inputs. The time complexity of our implementation depends on:

i. Line 6 → $\mathcal{O}(m\,\text{comb}(m, [m/2])|multivector|\text{len}(function))$: the curl_operator method has time complexity $\mathcal{O}(m\,\text{comb}(m, [m/2])|multivector|\text{len}(function))$.

ii. Line 7 → $\mathcal{O}(\text{comb}(m, [m/2])|multivector|\text{len}(function))$: we iterate over *variable_1*, and the transformation of each of its items depends on the length of the *multivector* items and the *function* string expression.

iii. Line 8 $\rightarrow \mathcal{O}(k\text{comb}(m, [m/2]|multivector|\text{len}(function)))$: we iterate over *mesh*. The evaluation of *variable_1* depends on its length, and on the length of the *multivector* items and the *function* string expression.

Hence, the time complexity of num_curl_operator is

$$\mathcal{O}(\text{comb}(m, [m/2])|multivector|\text{len}(function)(m+k)).$$

$\square$

3.2 Performance

The time performance of each function in NumPoissonGeometry was experimentally measured by evaluating concrete examples on $\mathbf{R}^2$, and $\mathbf{R}^3$, on precalculated (irregular) meshes with 10^κ points, for $\kappa = 3, \ldots, 7$. These meshes were generated by means of random samples extracted from a uniform distribution in the interval $[0, 1)$.

All the numerical experiments were performed on a workstation equipped with 48 GB of main memory in a 3×16 GB ddr4 module configuration and an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz CPU, running at 3.4 GHz for a peak theoretical performance of four cores.

3.2.1 Two Dimensional Case

For the performance tests of our methods in dimension two, we have set the non–degenerate Poisson bivector field

$$\Pi_0 = \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2},$$

induced by the standard symplectic structure $dx_1 \wedge dx_2$ on $\mathbf{R}_x^2$.

Table 3.2: Input data used for the time performance tests of functions 1-10 in Table 3.3.

Function	Input	Function	Input
1	Π_0	6	$\Pi_0, W = x_2\frac{\partial}{\partial x_1} - x_1\frac{\partial}{\partial x_2}$
2	Π_0	7	$\Pi_0, f = 1$
3	$\Pi_0, h = x_1^2 + x_2^2$	8	$\Pi_0, f = 1$
4	$\Pi_0, f = x_1^2 + x_2^2, g = x_1 + x_2$	9	$\Pi_0, \alpha = x_1 dx_1 + x_2 dx_2, \beta = dx_1 + dx_2$
5	$\Pi_0, \alpha = x_1 dx_1 + x_2 dx_2$	10	$\Pi_0, \lambda = dx_1 \wedge dx_2$

Table 3.3 lists the mean time in seconds (with standard deviation) it takes to evaluate the first ten functions in NumPoissonGeometry (see, Table 1.1) on a irregular mesh on $\mathbf{R}_x^3$ with 10^κ points, computed by taking twenty-five samples, for $\kappa = 3, \ldots, 7$.

Table 3.3: Summary of the time performance of NumPoissonGeometry functions, in dimension two.

Function	Points in mesh/Processing time (in seconds)				
	10^3	10^4	10^5	10^6	10^7
1. num_bivector_field	0.004 ± 0.689	0.038 ± 0.009	0.356 ± 0.002	3.545 ± 0.026	35.711 ± 0.164
2. num_bivector_to_matrix	0.006 ± 3.633	0.046 ± 0.001	0.438 ± 0.004	4.442 ± 0.037	45.155 ± 1.466
3. num_hamiltonian_vf	0.014 ± 0.001	0.112 ± 0.006	1.096 ± 0.021	10.867 ± 0.044	108.460 ± 0.726
4. num_poisson_bracket	0.021 ± 0.006	0.169 ± 0.001	1.652 ± 0.008	16.721 ± 0.049	168.110 ± 1.637
5. num_sharp_morphism	0.014 ± 0.658	0.111 ± 0.001	1.068 ± 0.007	10.725 ± 0.142	107.275 ± 0.667
6. num_coboundary_operator	0.001 ± 0.087	0.008 ± 0.001	0.084 ± 0.006	0.848 ± 0.011	8.638 ± 0.045
7. num_modular_vf	0.004 ± 0.754	0.030 ± 0.009	0.280 ± 0.001	2.805 ± 0.016	28.057 ± 0.107
8. num_curl_operator	0.022 ± 0.009	0.196 ± 0.024	1.923 ± 0.004	18.487 ± 0.136	182.774 ± 1.260
9. num_one_forms_bracket	0.058 ± 0.006	0.420 ± 0.007	4.278 ± 0.027	43.257 ± 0.071	434.450 ± 0.589
10. num_gauge_transformation	0.051 ± 0.001	0.446 ± 0.010	4.380 ± 0.016	43.606 ± 0.212	434.704 ± 1.234

To illustrate how fast the NumPoissonGeometry functions can be performed, we use the data in Table 3.5 to plot the time versus the number of points in each 10^κ-point (irregular) mesh on base 10 *log-log graphs*:

Figure 3.1: Log-log graphs of the execution time in seconds versus the number of points in 10^κ-point (irregular) meshes of the NumPoissonGeometry functions 1–10 in Table 3.3, for $\kappa = 3, \ldots, 7$. In red, the fitted linear model used to predict the asymptotic behavior of the runtime for each function, with the corresponding determination coefficient (R-squared) indicated in each legend. We include a zoom-graph in each plot due to the accumulation of runtime values.
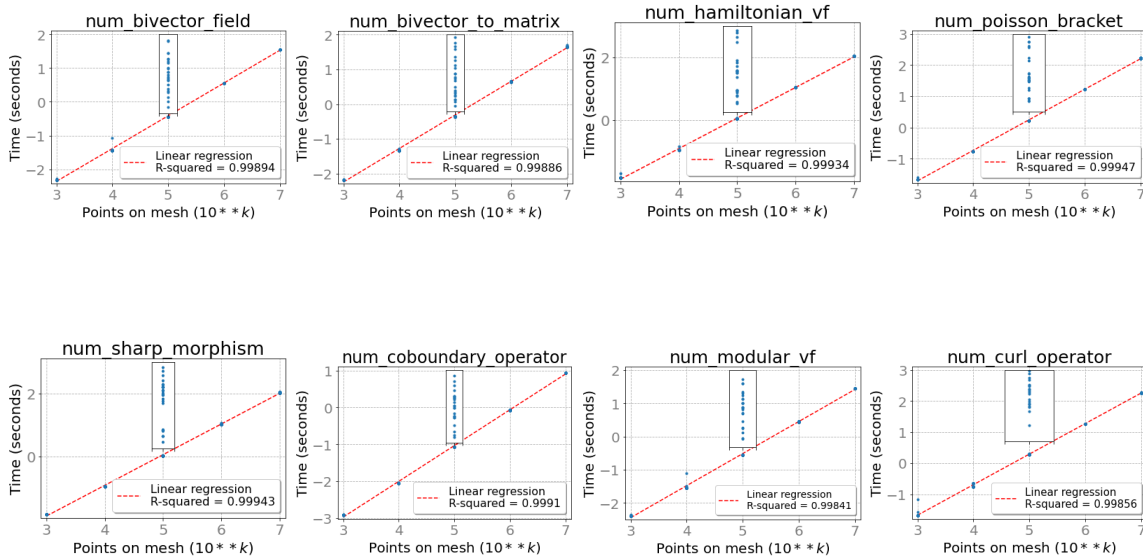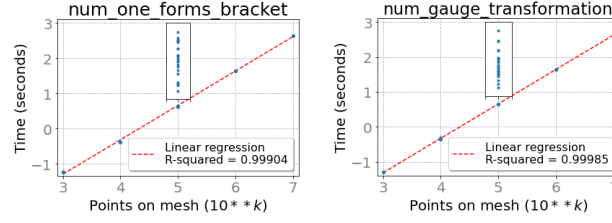
We deduce from the graphs in Figure 3.1 that, for the input data in Table 3.4, all of our methods in Table 3.3 were executed experimentally in polynomial time. Power-law relationships appear as straight lines in a log-log graph. Therefore the degree of the polynomial complexities are deduced by fitting a linear model and estimating its coefficient, which we carry out to 0.99 accuracy.

3.2.2 Three Dimensional Case

For the performance tests of our methods in dimension three, we have set the Lie–Poisson bivector field $\Pi_{\mathfrak{sl}(2)}$ in (2–5),

$$\Pi_{\mathfrak{sl}(2)} = -x_3 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_2} - x_2 \frac{\partial}{\partial x_1} \wedge \frac{\partial}{\partial x_3} + x_1 \frac{\partial}{\partial x_2} \wedge \frac{\partial}{\partial x_3},$$

associated to the 3–dimensional Lie algebra $\mathfrak{sl}(2)$.

Table 3.4: Input data used for the time performance tests of functions 1-11 in Table 3.5.

Function	Input
1	$\Pi_{\mathfrak{sl}(2)}$
2	$\Pi_{\mathfrak{sl}(2)}$
3	$\Pi_{\mathfrak{sl}(2)}, h = x_1^2 + x_2^2 - x_3^2$
4	$\Pi_{\mathfrak{sl}(2)}, f = x_1^2 + x_2^2 - x_3^2, g = x_1 + x_2 + x_3$
5	$\Pi_{\mathfrak{sl}(2)}, \alpha = x_1 dx_1 + x_2 dx_2 - x_3 dx_3$
6	$\Pi_{\mathfrak{sl}(2)}, W = e^{-1/(x_1^2+x_2^2-x_3^2)^2}\left[x_1 x_3/(x_1^2 + x_2^2)\frac{\partial}{\partial x_1} + x_2 x_3/(x_1^2 + x_2^2)\frac{\partial}{\partial x_2} + \frac{\partial}{\partial x_3}\right]$
7	$\Pi_{\mathfrak{sl}(2)}, f = 1$
8	$\Pi_{\mathfrak{sl}(2)}, f = 1$
9	$\Pi_{\mathfrak{sl}(2)}, \alpha = x_1 dx_1 + x_2 dx_2 - x_3 dx_3, \beta = dx_1 + dx_2 + dx_3$
10	$\Pi_{\mathfrak{sl}(2)}, \lambda = (x_2 - x_1)dx_1 \wedge dx_2 + (x_3 - x_1)dx_1 \wedge dx_3 + (x_2 - x_3)dx_2 \wedge dx_3$
11	$-\Pi_{\mathfrak{sl}(2)}$

Table 3.5 lists the mean time in seconds (with standard deviation) it takes to evaluate the first eleven functions in NumPoissonGeometry (see, Table 1.1) on a irregular mesh on $\mathbf{R}_x^3$ with 10^κ points, computed by taking twenty-five samples, for $\kappa = 3, \dots, 7$.

Table 3.5: Summary of the time performance of NumPoissonGeometry functions in dimension 3.

| Function	Points in mesh/Processing time (in seconds)				
	10^3	10^4	10^5	10^6	10^7
1. num_bivector_field	0.009 ± 0.009	0.051 ± 0.004	0.496 ± 0.002	4.984 ± 0.023	49.565 ± 0.222
2. num_bivector_to_matrix	0.008 ± 3.164	0.057 ± 0.002	0.553 ± 0.019	5.442 ± 0.023	55.249 ± 1.690
3. num_hamiltonian_vf	0.017 ± 0.002	0.129 ± 0.001	1.263 ± 0.022	12.518 ± 0.064	126.091 ± 0.583
4. num_poisson_bracket	0.036 ± 0.001	0.299 ± 0.010	2.936 ± 0.067	29.600 ± 0.933	292.625 ± 6.094
5. num_sharp_morphism	0.017 ± 0.006	0.128 ± 0.005	1.252 ± 0.005	12.384 ± 0.038	124.851 ± 1.809
6. num_coboundary_operator	1.589 ± 0.016	1.705 ± 0.029	2.815 ± 0.032	12.972 ± 0.166	111.034 ± 1.365
7. num_modular_vf	0.050 ± 0.001	0.103 ± 0.004	0.645 ± 0.006	6.025 ± 0.013	59.652 ± 0.146
8. num_curl_operator	0.019 ± 0.010	0.129 ± 0.027	1.199 ± 0.032	10.911 ± 0.181	105.841 ± 1.230
9. num_one_forms_bracket	0.093 ± 0.001	0.738 ± 0.007	7.285 ± 0.159	72.802 ± 1.474	724.514 ± 13.594
10. num_gauge_transformation	0.051 ± 0.001	0.445 ± 0.010	4.395 ± 0.013	43.794 ± 0.173	437.326 ± 0.824
11. num_linear_normal_form_R3	0.016 ± 0.438	0.061 ± 0.002	0.504 ± 0.012	4.903 ± 0.017	48.786 ± 0.219

Now, to illustrate how fast the NumPoissonGeometry functions can be performed, we use the data in Table 3.5 to plot the time versus the number of points in each 10^κ-point (irregular) mesh on base 10 *log-log graphs*. These (execution time) plots are presented in Figures 3.2 and 3.3.

3.2.3   Polynomial Time

The log-log graphs presented in Figure 3.2 correspond to NumPoissonGeometry methods that are executed in polynomial time of some degree. As power-law relationships appear as straight lines in a log-log graph, the complexities are deduced by fitting a linear model and estimating its coefficient to 0.99 accuracy.

Figure 3.2: Log-log graphs of the execution time in seconds versus the number of points in 10^κ-point (irregular) meshes of the NumPoissonGeometry functions 1–5 and 8–11 in Table 3.5, for $\kappa = 3, \ldots, 7$. In red, the fitted linear model used to predict the asymptotic behavior of the runtime for each function, with the corresponding determination coefficient (R-squared) indicated in each legend. We include a zoom-graph in each plot due to the accumulation of runtime values.
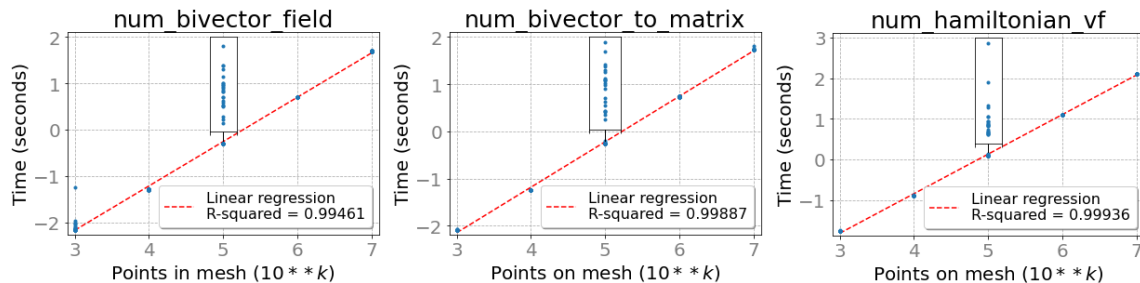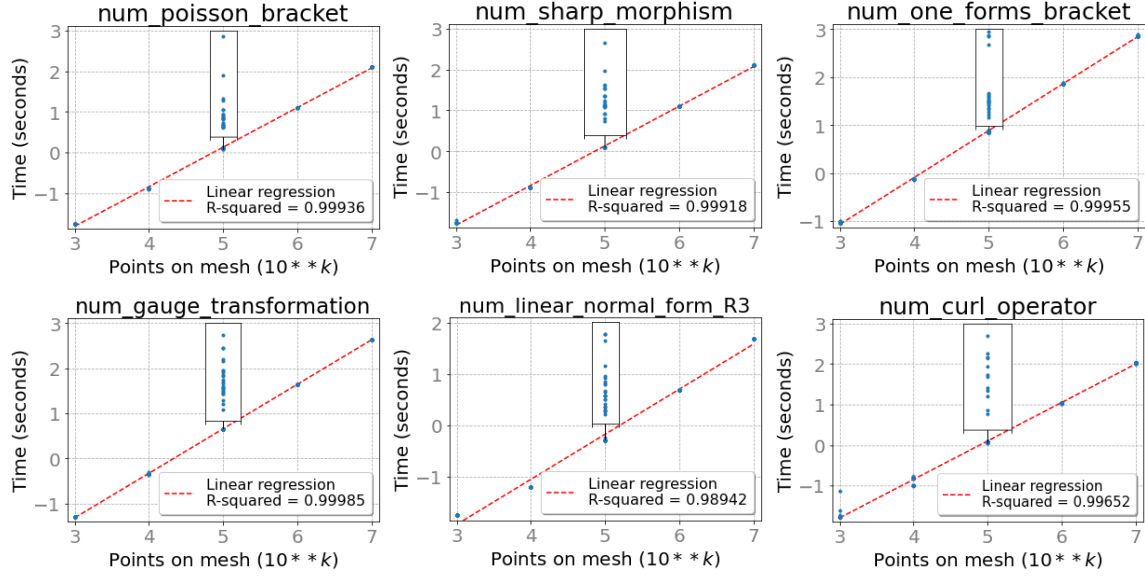
Observe that the experimentally deduced time complexities of the $\mathrm{NumPoissonGeometry}$ functions in Figure 3.2 coincide with their theoretical time complexities in the following sense: the time complexities of methods 1–5 and 9–11 in Table 3.1 are polynomially dependent on variables m and k.

Remark 3.14 In our experiments, $\mathrm{num\_curl\_operator}$ approached a polynomial time complexity, although its theoretical time complexity is exponential (see, Table 3.1). This is not *a fortiori* a contradiction, because in Table 3.1 we present an approximation of the *worst–case* time complexity of the $\mathrm{NumPoissonGeometry}$ methods. In fact, it is an example that the execution time of our algorithms depends (naturally) on their inputs and that in some cases they run faster than expected.

3.2.4 Exponential Time

The log-log graphs presented in Figure 3.2 correspond to $\mathrm{NumPoissonGeometry}$ methods that are executed in exponential time. As exponential relationships trace polynomial curves in a log-log graph, the complexities are deduced by fitting a (non-linear) polynomial regression and estimating its coefficients to 0.99 accuracy.

Figure 3.3: Log-log graphs of the execution time in seconds versus the number of points in 10^κ-point (irregular) meshes of the $\mathrm{NumPoissonGeometry}$ functions 6 and 7 in Table 3.5, for $\kappa = 3, \ldots, 7$. In red, the fitted model used to predict the asymptotic behavior of the runtime for each function, with the corresponding determination coefficient (R-squared) indicated in each legend. We include a zoom-graph in each plot due to the accumulation of runtime values.
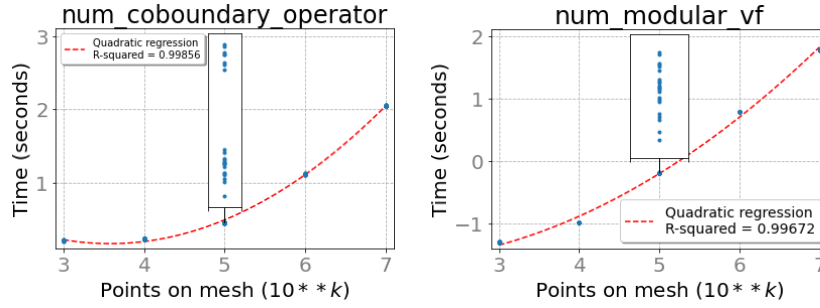
Observe that although the $\mathrm{NumPoissonGeometry}$ functions presented in Figure 3.3 are executed in exponential time, they are relatively fast since their evaluation on a 10 million point (irregular) mesh takes at most 2 minutes in the experiments on our desktop workstation. Furthermore, the execution times of the $\mathrm{num\_modular\_vf}$ function are fitted to a linear model in the interval [4,7], as we illustrate in the following figure:

Figure 3.4: Log-log graph of the execution time in seconds versus the number of points in a 10^κ-point (irregular) mesh of the $\mathrm{NumPoissonGeometry}$ function $\mathrm{num\_modular\_vf}$, for $\kappa = 4, \ldots, 7$. In red, the fitted linear model used to predict the asymptotic behavior of the runtime, with the corresponding determination coefficient (R-squared) indicated in the legend.
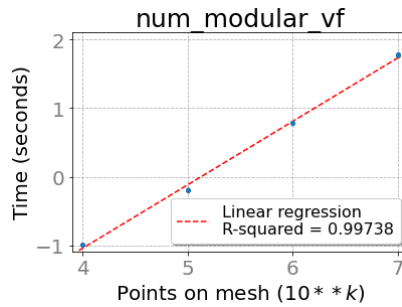
3.2.5 Flaschka–Ratiu Bivector Fields

For the performance tests of the method $\mathrm{num\_flaschka\_ratiu\_bivector}$, we have used as inputs the following scalar functions on $\mathbf{R}_x^4$ (see Example 2.12):

$$K_1 = \tfrac{1}{2}x_4, \qquad K_2 = -x_1^2 + x_2^2 + x_3^2.$$

Table 3.6: Mean time in seconds (with standard deviation) it takes to evaluate the num_flaschka_ratiu_bivector method on a irregular mesh on $\mathbf{R}^4$ with 10^κ points, computed by taking twenty-five samples, for $\kappa = 3, \ldots, 7$.

Function	Points in mesh/Processing time (in seconds)				
	10^3	10^4	10^5	10^6	10^7
num_flaschka_ratiu_bivector	0.0158 ± 0.105	0.057 ± 0.003	0.505 ± 0.003	4.993 ± 0.029	49.563 ± 0.207

Figure 3.5 below illustrates that the num_flaschka_ratiu_bivector method is executed in polynomial time of some degree, which coincides with its theoretical complexity presented in Table 3.1 in the sense that it is polynomially dependent on variables m and k. As power-law relationships appear as straight lines in a log-log graph, the complexity is deduced by fitting a linear model and estimating its coefficient to 0.99 accuracy.

Figure 3.5: Log-log graph of the execution time in seconds versus the number of points in a 10^κ-point (irregular) mesh of the num_flaschka_ratiu_bivector method, for $\kappa = 3, \ldots, 7$. In red, the fitted linear model used to predict the asymptotic behavior of the runtime, with the corresponding determination coefficient (R-squared) indicated in the legend. We include a zoom-graph due to the accumulation of runtime values.
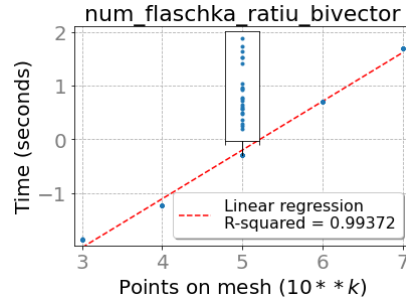
Acknowledgements

References

[1] A. Abouqateb, M. Boucetta, *The Modular Class of a Regular Poisson Manifold and the Reeb Class of its Symplectic Foliation*, C. R. Math. Acad. Sci. Paris **337** (1), 61-66 (2003).

[2] M. Ammar, G. Kass, M. Masmoudi, N. Poncin, *Strongly r-Matrix Induced Tensors, Koszul Cohomology, and Arbitrary-Dimensional Quadratic Poisson Cohomology*, Pac. J. Math. **245** (1), 1-23 (2010).

[3]   M. Avendaño-Camacho, J. A. Vallejo, Yu. Vorobiev, *A Perturbation Theory Approach to the Stability of the Pais-Uhlenbeck Oscillator*, J. Math. Phys. **58** (9), (2017).

[4]   P. Balseiro, L. C. García-Naranjo, *Gauge Transformations, Twisted Poisson Brackets and Hamiltonization of Nonholonomic Systems*, Arch. Ration Mech. An. **205**, 267-310 (2012).

[5]   A. Barp, F. X. Briol, A. D. Kennedy, M. Girolami, *Geometry and Dynamics for Markov Chain Monte Carlo*, Annual Review of Statistics and Its Application **5** (1), 451-471 (2018).

[6]   P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, K. Kavukcuoglu, *Interaction Networks for Learning about Objects, Relations and Physics*, Proc. of the 30th International Conference on Neural Information Proc. Systems **NIPS 16**, 4509–4517 (2016).

[7]   P. G. Breen, C. N. Foley, T. Boekholt, S. P. Zwart, *Newton versus the Machine: Solving the Chaotic Three-Body Problem using Deep Neural Networks*, Mon. Not. R. Astron. Soc. **494** (2), 2465–2470 (2020).

[8]   F. Bullo, A. D. Lewis, *Geometric Control of Mechanical Systems: Modeling, Analysis, and Design for Simple Mechanical Control Systems*, Springer, New York, NY (2005).

[9]   H. Bursztyn, *On Gauge Transformations of Poisson Structures*, In Quantum Field Theory and Noncommutative Geometry, Lecture Notes in Phys. U. Carow-Watamura, Y. Maeda, and S. Watamura, Eds. Springer, Berlin, Heidelberg, 89-112 (2005).

[10]   Z. Chen, J. Zhang, M. Arjovsky, L. Bottou, *Symplectic Recurrent Neural Networks*, arXiv:1909.13334 [stat.ML] (2019).

[11]   C. Caracciolo, U. Locatelli, *Computer-assisted estimates for Birkhoff normal forms*, Journal of Computational Dynamics,No. 7, Vol.2,425–460 (2020).

[12]   M. Cranmer, D. Tamayo, S. Hadden, H. Rein, P. Battaglia, A. Obertas, P. Armitage, S Ho, D. N. Spergel, C. Gilbertson, N. Hussain, A. Silburt, D. Jontof-Hutter, K. Menou, *Predicting the long-term stability of compact multiplanet systems*, Proceedings of the National Academy of Sciences **117**, (31) 18194-18205 (2020).

[13]   P. A. Damianou, F. Petalidou, *Poisson Brackets with Prescribed Casimirs*, Canad. J. Math. **64** (5), 991-1018 (2012).

[14]   M. de la Cruz, N. Gaspar, L. Jiménez-Lara, R. Linares, *Classification of the Classical SL*(2, **R**) *Gauge Transformations in the Rigid Body*, Ann. Phys. **379**, 112{130 (2017).

[15]   P. A. M. Dirac, *Quelques problèmes de mécanique quantique*, Annales de l'I. H. P., tome 1, no 4 (1930), p. 357–400.

[16]   J. P. Dufour, N. T. Zung, *Poisson Structures and their Normal Forms*, Birkhäuser Basel (2005).

[17]   M. Evangelista-Alvarado M, P. Suárez-Serrato, J. Torres-Orozco, R. Vera, *On Bott-Morse Foliations and their Poisson Structures in Dimension 3*, Jour. Singular. **19**, 19-33 (2019).

[18]   M. Evangelista-Alvarado, J. C. Ruíz-Pantaleón, P. Suárez-Serrato, *On Computational Poisson Geometry I: Symbolic Foundations*, arXiv:1912.01746 [math.DG] (2019).

[19]   L. Falorsi, P. De Haan, T. R. Davidson, P. Forré, *Reparameterizing Distributions on Lie Groups*, Proceedings of Machine Learning Research. **89**, 3244-3253 (2019).

[20]   P. Frejlich, I. Marcut, *The Homology Class of a Poisson Transversal*, Int. Math. Res. Notices **2020** (10), 2952-2976 (2020).

[21]   B. Fuchssteiner, *The Lie Algebra Structure of Degenerate Hamiltonian and Bi–Hamiltonian Systems*, Progr. Theoret. Phys. **68**, 1082–1104 (1982).

[22] L. C. García-Naranjo, P. Suárez-Serrato, R. Vera, *Poisson Structures on Smooth 4-manifolds*, Lett. Math. Phys. **105** (105), 1533-1550 (2015).

[23] V. L. Ginzburg, A. Weinstein, *Lie-Poisson Structure on Some Poisson Lie Groups*, J. Amer. Math. Soc. **5** (2), 445-453 (1992).

[24] J. Grabowski, G. Marmo, A. M. Perelomov, *Poisson Structures: Towards a Classification*, Mod. Phys. Lett. A **08** (18), 1719-1733 (1993).

[25] J. Grabowski, *Brackets*, Int. J. Geom. Methods Mod. Phys. **10** (08), (2013).

[26] S. Greydanus, M. Dzamba, J. Yosinski, *Hamiltonian neural networks*, Advances in Neural Information Processing Systems (2019), 15379–15389.

[27] V. Guillemin, E. Miranda, A. R. Pires, *Codimension One Symplectic Foliations and Regular Poisson Stuctures*, Bull. Braz. Math. Soc. **42** (4), 607-623 (2011).

[28] M.D. Hoffman, A. Gelman, *The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo* Journal of Machine Learning Research. 15 (1), (2014) 1593–1623.

[29] M. Cranmer, A. Sanchez-Gonzalez, P. Battaglia, R. Xu, K. Cranmer, *Discovering symbolic models from deep learning with inductive biases*, arXiv preprint arXiv:2006.11287.

[30] D. Jimenez, G. Papamakarios, S. Racaniere, M. S. Albergo, G. Kanwar, P. E. Shanahan, K. Cranmer, *Normalizing Flows on Tori and Spheres*, arXiv:2002.02428 [stat.ML] (2020).

[31] D. Jimenez, S. Racaniere, I. Higgins, P. Toth, *Equivariant Hamiltonian Flows*, arXiv:1909.13739 [stat.ML] (2019).

[32] Y.A. Ma, Y. Chen, C. Jin, N. Flammarion, M.I. Jordan, *Sampling can be faster than optimization*, Proceedings of the National Academy of Sciences 116 (42), (2019) 20881–20885.

[33] M. Kontsevich, *Deformation quantization of Poisson manifolds,* Lett. Math. Phys. 66 (2003), no. 3, 157–216.

[34] Y. Kosmann-Schwarzbach, *Poisson Manifolds, Lie Algebroids, Modular Classes: a Survey*, SIGMA **4**, 005-034 (2008).

[35] J. L. Koszul, *Crochet de Schouten-Nijenhuis et Cohomologie*, Astérisque **S131**, 257-271 (1985).

[36] V. V. Kozlov, *Symmetries, Topology and Resonances in Hamiltonian Dynamics*, Springer-Verlag Berlin Heidelberg, Berlin (1995).

[37] M. Kröger, M. Hütter, H. C. Öttinger, *Symbolic Test of the Jacobi Identity for Given Generalized 'Poisson' Bracket*, Comput. Phys. Commun. **137** (2), 325-340 (2001).

[38] M. Lainz, C. Sardón, A. Weinstein, *Plasma in a Monopole Background does not have a Twisted Poisson Structure*, Phys. Rev. D **100** (105016), 1-5 (2019).

[39] C. Laurent-Gengoux, A. Pichereau, P. Vanhaecke, *Poisson Structures*, Springer-Verlag Berlin Heidelberg, Berlin (2013).

[40] J. P. Lessard, J. Berg, *Rigorous Numerics in Dynamics*, Notices of the American Mathematical Society **69**, 1057-1061 (2015).

[41] S. Li, C. Dong, L. Zhang, L. Wang, *Neural Canonical Transformation with Symplectic Flows*, Phys. Rev. X, **10** (021020), 1-13 (2020).

[42] A. Lichnerowicz, *Les Variétés de Poisson et Leurs Algèbres de Lie Associetes*, J. Differential Geom. **12** (2), 253-300 (1977).

[43] Z. J. Liu, P. Xu, *On Quadratic Poisson Structures*, Lett. Math. Phys. **26** (1), 33-42 (1992).

[44] Yi-An Ma, Y. Chen, C. Jin, N. Flammarion, M. I. Jordan, *Sampling can be faster than optimization*, Proc. of the National Academy of Sciences **116**, (42) 20881-20885 (2019).

[45] I. Marcut, F. Zeiser, *The Poisson Cohomology of $\mathfrak{sl}^*(2, \mathbf{R})$*, arXiv:1911.11732 [math.SG] (2020).

[46] P. Michor. *Topics in Differential Geometry*, Am. Math. Soc. (2008).

[47] N. Nakanishi, *On the Structure of Infinitesimal Automorphisms of Linear Poisson Manifolds I*, J. Math. Kyoto Univ. **31** (1), 281-287 (1991).

[48] A. Pedroza, E. Velasco-Barreras, Yu. Vorobiev, *Unimodularity Criteria for Poisson Structures on Foliated Manifolds*, Lett. Math. Phys. **108**, 861–882 (2018).

[49] S.-D. Poisson, *Sur la Variation des Constantes Arbitraires dans les Questions de Mécanique*, J. Ecole Polytechnique **8** (15), 266-344 (1809).

[50] D. Pozharskiy, N.J. Wichrowski, A.B. Duncan, G.A. Pavliotis, I.G. Kevrekidis, *Manifold learning for accelerating coarse-grained optimization*, Journal of Computational Dynamics,Vol. 7, No. 2, 511–536 (2020).

[51] A. Sanchez-Gonzalez, V. Bapst, K. Cranmer, P. Battaglia, *Hamiltonian Graph Networks with ODE Integrators*, arXiv:1909.12790 [cs.LG] (2019).

[52] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, P. Battaglia, *Learning to Simulate Complex Physics with Graph Networks*, arXiv:2002.09405 [cs.LG] (2020).

[53] P. Severa, A. Weinstein, *Poisson Geometry with a 3-Form Background*, Progr. Theoret. Phys. Suppl. **144**, 145-154 (2001).

[54] Y. Sheng, *Linear Poisson Structures on $\mathbf{R}^4$*, J. Geom. Phys. **57**, 2398–2410 (2007).

[55] P. Suárez-Serrato, J. Torres-Orozco, *Poisson Structures on Wrinkled Fibrations*. Bol. Soc. Mat. Mex. **22** (1), 263-280 (2016).

[56] S. Takato and J.A. Vallejo, *Hamiltonian Dynamical Systems: Symbolical, Numerical and Graphical Study*, Math. in Comp. Sci. **13** (1-2), 281-295 (2019).

[57] P.Toth, D.J. Rezende, A. Jaegle, S. Racanière, A. Botev, I. Higgins, *Hamiltonian Generative Networks*, International Conference on Learning Representations (2020).

[58] A. Weinstein, *The Local Structure of Poisson Manifolds*, J. Diff. Geom. **18** (3), 523-557 (1983).

[59] _____ , *The Modular Automorphism Group of a Poisson Manifold*, J. Geom. Phys. **23** (3-4), 379-394 (1997).

[60] _____ , *Poisson Geometry*, Diff. Geom. Appl. **9** (1-2), 213-238 (1998).

[61] A. Wibisono, A. C. Wilson, M. I. Jordan, *A variational perspective on accelerated methods in optimization*, Proc. of the National Academy of Sciences **113**, (47) 7351-7358 (2016).

[62] J. H. Wilkinson, C. Reinsch, *Handbook for Automatic Computation: Volume II: Linear Algebra*, **186** Springer Science & Business Media (2012).

Instituto de Matemáticas, Universidad Nacional Autónoma de México (UNAM), Mexico City, Mexico

miguel.eva.alv @matem.unam.mx, jcpanta @im.unam.mx, pablo @im.unam.mx