Structure and Interpretation of Classical Mechanics with Python and Sagemath

Nicky van Foreest

February 28, 2025

CONTENTS

0 PRELIMINARIES

0.1 INTRODUCTION

This is a translation to Python and Sagemath of (most of) the Scheme code of the book 'Structure and interpretation of classical mechanics' by Sussman and Wisdom. When referring to *the book*, I mean their book. I expect the reader to read the related parts of the book, and use the Scheme code of the book to understand the Python code below. In other words, I don't explain much of the logic of the code in this document. I'll try to stick to the naming of functions and variables as used in the book. I also try to keep the functional programming approach of the book; consequently, I don't strive to the most pythonic code possible. To keep the code clean, I never protect functions against stupid input; realize that this is research project, the aim is not to produce a fool-proof software product.

I tend to place explanations, comments, and observations about the code and the results *above* the code blocks.

I wrote this document in emacs and org mode. When developing, I first made a sage file with all code for a specific section of the book. Once all worked, I copied the code to an org file and make code blocks. Once done, I tangled to code of the org mode to a utils.sage file containing general functions and code that is useful for other sections, and section specific code to a section.sage file.

I also found it convenient to test things in a `tests.sage` file. Then, by opening a sage session in the command prompt and attaching the session to the file (`attach("tests.sage")`), I could edit within emacs and see the consequences directly in the sage session.

- An online version of the book: https://tgvaughan.github.io/sicm/
- An org file of the book with Scheme: https://github.com/mentat-collective/sicm-book/blob/main/org/chapter001.org
- https://doc.sagemath.org/html/en/reference/
- Handy tuples: https://github.com/jtauber/functional-differential-geometry
- A port to Clojure: https://github.com/sicmutils/sicmutils
- ChatGPT proved to be a great help in the process of becoming familiar with Scheme and Sagemath.

0.2 UTILITIES

Here we provide the Python and Sagemath code for the functions that the book does not explain.

0.2.1 *Standard import*

We need some standard imports to run the code in the code blocks below. Since I run the sage scripts on the command line like `$> sage section1.4.sage` it appears not to be necessary to import `sage.all`.

BTW, don't load `from sage.all import *` after loading `utils.sage`, because that will lead to name space conflict with the `Gamma` function.

Below I explain why the next files are loaded in this way. Note that the titles of the code blocks correspond to the file to which the code is written when tangled.

utils.sage

```
import numpy as np
from tuples import up, Tuple # See below why this import.
```

utils_tests.sage

```
load("utils.sage")
```

don't tangle

```
load("show_expression.sage")
```

0.2.2 *Output to LaTeX*

To keep the formulas short in LaTeX, I remove all strings like (t), and replace $\partial x/\partial t$ by $\dot{x}$. There is a caveat, though. When a string is returned, rather than printed, org mode, or Python, adds many escape symbols, thereby ruining the LaTeX output. For this reason, I call `print`, which for my purposes (with org mode) works.

show_expression.sage

```
import re
import tuples # see below

def show_expression(s):
    s = latex(s)
    s = re.sub(r"\\frac{\\partial}{\\partial t}", r"\\dot ", s)
    s = re.sub(r"\\left\(t\\right\)", r"", s)
    s = re.sub(
        r"\\frac\{\\partial\^\{2\}\}\{\(\\partial t\)\^\{2\}\}", r"\\ddot ", s
    )
    print(r"\[" + s + r"\]")

```

```
17
18 # def show_expression(s):
19 #     return r"\[" + latex(s) + r"\]"
```

The book uses `up` tuples, which I want to be printed vertically. I guess I should implement the formatting of `Tuple` on the `Tuple` class, but here I take the lazy route.

```
20
21 def show_tuple(tup):
22     res = r"\begin{align*}"
23     for component in tup:
24         res += "& " + latex(component) + r"\\"
25     res += r"\end{align*}"
26     return res
```

There is another subtlety. When working in sage files, I call `show(expr)` to have some expression printed to screen. In this case, I do *not* want to see LaTeX output. However, when executing a code block in org mode, I *do* want to get LaTeX output, and for this, I could use `show_expression` (just like in the book) in the code blocks in the org file. So far so good, but now comes the subtlety. When I *tangle* the code from the org file to a sage file, and I don't want to see `show_expression`, but just `show`. Thus, I should use `show` throughout, but in the org mode file, `show` should call `show_expression`. To achieve this, I load the file `show_expression.sage` *only in the org mode file* so that `show` gets routed to `show_expression` in the org file.

```
27 def show(s):
28     if isinstance(s, tuples.Tuple):
29         return show_tuple(s)
30     return show_expression(s)
```

0.2.3 *Literate functions*

The function `literal_function` will depend *always* on t. We therefore provide a `(t)` after the `function`. To print a literal function with LaTeX, I suppress the dependence on (t); the function `print_lit_to_latex` achieves this.

utils.sage

```
31 var('t', domain="real")
32
33
34 def literal_function(name):
35     return function(name, nargs=1, print_latex_func=print_lit_f_to_latex)(t)
36
37
38 def print_lit_f_to_latex(name, *args):
39     return name
```

0.2.4 *Coordinate paths*

We implement a generic coordinate path `q` as a `vector` whose elements are `literal_function`s. Note that a `vector` expects expressions as elements; this is another reason to attach `(t)` as a argument in definition of `literal_function` (without the `(t)`, literal functions cannot be stored in a vector.)

utils_tests.sage

```
q = vector(
    [
        literal_function("q_x"),
        literal_function("q_y"),
        literal_function("q_z"),
    ]
)
```

Here is an example to see how to evaluate `q`.

utils_tests.sage

```
show(q(t=t))
```

$$(q_x, q_y, q_z)$$

I discovered it is best not to use `x` and `y` in the definition of literal functions because the variables `x` and `y` seem to get overwritten. Hence, don't do this.

don't tangle

```
# Avoid this use of x and y.
q = vector([literal_function("x"), literal_function("y")])
```

Sometimes we need to lift a coordinate `q` and a velocity vector `v` to a *local tuple*. For this, we use the `up` tuple; this makes it easy to stick to the book. However, we don't build the coordinate path nor the velocity as tuples because I find Sagemath vectors more convenient.

utils.sage

```
def qv_to_state(q, v):
    return up(t, q, v)
```

The next function allows us to lift coordinate paths and velocity paths to local tuples paths. (The idea of returning a function within a function is known as *currying*.)

utils.sage

```
def qv_to_state_path(q, v):
    def f(t):
        return up(t, q(t=t), v(t=t))

    return f
```

Using a `vector` proves handy because Sagemath offers element-wise differentiation of vectors.

utils_tests.sage

```
57 show(q.diff(t))
```

$$(\dot{q}_x, \dot{q}_y, \dot{q}_z)$$

Before we can continue with constructing a coordinate path $q(t)$ and its velocity path $\dot{q}(t)$, we need to figure out how to set up differentiation with Sagemath so that we can follow the notation of the book. This in turn requires to construct (coordinate) spaces, so this we will do that first.

0.2.5 *Spaces*

To use the differentiation functionality of Sagemath, we need variable names, for instance, in the expressions `f(x).diff(x)` or `diff(f(x), x)` we need to provide an argument such as `x`. Now, without a space I don't see a simple way to support in Sagemath the notation of the book like $\partial_1 L(t,q,p)$, where the 1 refers to the *slot* of the arguments of L. So I decided to build a `space` that specifies variable names to make up a space. This brings us to the problem of defining a space.

The next function makes coordinates with a list of given names, for instance `["\\phi", "\\theta"]`. Note that these names are used as arguments in `latex_name`, so we need to include the full LaTeX name, and escape the backslash. In the name of the Sagemath variable, we strip the backslash. The velocities follow the same pattern. As I prefer to read $\dot{\phi}$ instead of $\partial\phi/\partial t$, I use the dot in the `latex_name`. Once all coordinates and velocities are made for the variables in the named list, the next function casts them to vectors and turns them into a local tuple.

utils.sage

```
58 def make_named_coordinates(coordinate_names, latex_names=None):
59     vars = []
60     if latex_names == None:
61         latex_names = [name for name in coordinate_names]
62 
63     stripped = [f'{name.lstrip(r"\\")}' for name in coordinate_names]
64     for name, latex in zip(stripped, latex_names):
65         q = var(name, latex_name=f"{latex}", domain='real')
66         vars.append(q)
67     return vector(vars)
68 
69 
70 def make_named_velocities(coordinate_names, latex_names=None):
71     names = [f"{name}dot" for name in coordinate_names]
72     if latex_names == None:
73         latex_names = [fr"\dot {name}" for name in coordinate_names]
```

```
74      return make_named_coordinates(names, latex_names)
75
76
```

utils_tests.sage

```
77  show(make_named_coordinates(["q", "r"]))
78  show(make_named_velocities(["q", "r"]))
79  show(make_named_coordinates(["\\phi", r"\theta"]))
80  show(make_named_velocities(["\\phi", r"\theta"]))
```

$$(q, r)$$

$$(\dot{q}, \dot{r})$$

$$(\phi, \theta)$$

$$(\dot{\phi}, \dot{\theta})$$

Sometimes its easier to use variable names with an index, like $x_1, x_2, \ldots$. With these function we can just provide the symbol and the dimension of the coordinate space.

utils.sage

```
81  def make_coordinates(coordinate_name, dim):
82      names = [f"{coordinate_name}_{i}" for i in range(1, dim + 1)]
83      return make_named_coordinates(names)
84
85
86  def make_velocities(coordinate_name, dim):
87      names = [f"{coordinate_name}_{i}" for i in range(1, dim + 1)]
88      return make_named_velocities(names)
```

utils_tests.sage

```
89  show(make_coordinates("q", dim=3))
90  show(make_velocities("q", dim=3))
```

$$(q_1, q_2, q_3)$$

$$(\dot{q}_1, \dot{q}_2, \dot{q}_3)$$

The book of V.I. Arnold on classical mechanics provides two ways to define the Lagrangian as a map on a space. The first is this: $L : \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}, (t, q, p) \to L(t, q, p)$. The other is to define it as a map from the tangent bundle TM of the manifold M on which a system moves. The potential is then a function from M to $\mathbb{R}$, and the kinetic energy a quadratic form on each tangent space $T_x M$. As the first approach seems the simplest, at least at the moment, we follow this definition.

utils.sage

```
91
92
93  def make_named_space(coordinate_names):
94      coordinates = make_named_coordinates(coordinate_names)
95      velocities = make_named_velocities(coordinate_names)
96      return qv_to_state(coordinates, velocities)
97
98
99  def make_space(coordinate_name, dim):
100     coordinates = make_coordinates(coordinate_name, dim)
101     velocities = make_velocities(coordinate_name, dim)
102     return qv_to_state(coordinates, velocities)
```

The results are printed vertically because it's an up tuple.

utils_tests.sage

```
103 show(make_space("q", dim=2))
```

$$
\begin{pmatrix} t \\ (q_1, q_2) \\ (\dot{q}_1, \dot{q}_2) \end{pmatrix}
$$

Here is another example.

utils_tests.sage

```
104 space = make_named_space(["\\phi", "\\theta"])
105 show(space)
```

$$
\begin{pmatrix} t \\ (\phi, \theta) \\ (\dot{\phi}, \dot{\theta}) \end{pmatrix}
$$

0.2.6 *Differentiation*

Let's start with some basic examples of differentiation to see how things work in Sagemath. We need some variables to define a function `F`.

utils_tests.sage

```
106 var("a b c x y", domain="real")
```

We will use quadratic functions often, so let's make a function for this.

utils.sage

```
def square(x):
    return x * x
```

utils_tests.sage

```
F = a * square(x) + b * x + c
show(diff(F, x))
show(diff(F, x)(x=0))
show(diff(F, x, 2))
show(diff(F, x, x))
```

$$2ax + b$$

$$b$$

$$2a$$

$$2a$$

We can ask the argument names of a function, but this is not always helpful. The intention is to treat `c` as a constant, not as an argument of `F`. So, we need to be careful when using `args()`.

utils_tests.sage

```
M = matrix([[3, 4], [4, 5]])
b = vector([8, 9])
v = vector([x, y])
F = 1 / 2 * v * M * v + b * v + c
show(F.args())
```

$$(c, x, y)$$

Here are some ways to compute the gradient of `F`.

utils_tests.sage

```
show(jacobian(F, (x, y)))
show(jacobian(F, v))
show(jacobian(F, (x, y))(x=0, y=0))
show(jacobian(F(x=x, y=y), (x, y)))
show(jacobian(F(x=x, y=y), (x, y))(x=0, y=0))
```

$$\begin{pmatrix} 3x + 4y + 8 & 4x + 5y + 9 \end{pmatrix}$$

$$\begin{pmatrix} 3x + 4y + 8 & 4x + 5y + 9 \end{pmatrix}$$

$$\begin{pmatrix} 8 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 3x+4y+8 & 4x+5y+9 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 9 \end{pmatrix}$$

We can find the Hessian by using the `jacobian` function twice.

utils_tests.sage
```
124 show(jacobian(jacobian(F, (x, y)), (x, y)))
```

$$\begin{pmatrix} 3 & 4 \\ 4 & 5 \end{pmatrix}$$

We can also define a function in the normal Pythonic way, and take derivatives. The quoted example does not work because `F` does not receive an explicit variable name wrt which to take the derivative.

utils_tests.sage
```
125 def F(v):
126     return 1 / 2 * v * M * v + b * v + c
127
128 show(jacobian(F(v), (x, y)))
129 show(jacobian(F(v), v))
130 # show(jacobian(F, v))  # This does not work
```

$$\begin{pmatrix} 3x+4y+8 & 4x+5y+9 \end{pmatrix}$$
$$\begin{pmatrix} 3x+4y+8 & 4x+5y+9 \end{pmatrix}$$

Differentiation of symbolic functions works different from what I expected. For instance, if `U = function("U")`, then `jacobian(U(v), (x, y))` gives a coercion error. However, this does work:

utils_tests.sage
```
131 U = function("U")
132 show(jacobian(U(*v), (x, y)))
```

$$\begin{pmatrix} \frac{\partial}{\partial x}U(x,y) & \frac{\partial}{\partial y}U(x,y) \end{pmatrix}$$

So when differentiating a symbolic function, the arguments need to be unpacked with the $*$ operator. We can now do two things to get around this problem. We can protect each function in which we take a derivative by testing whether the given function is a symbolic function or not, and then act accordingly. Another strategy is to wrap a symbolic function in a python function and then pass an unpacked argument to the symbolic function. We follow this approach, one reason being that we will not use symbolic functions all that often. Here is an example to see how this approach works.

utils_tests.sage

```
def U(q):
    return function("U")(*q)

show(jacobian(U(v), (x, y)))
```

$$\left(\frac{\partial}{\partial x} U(x,y) \quad \frac{\partial}{\partial y} U(x,y) \right)$$

So now the interface to the `jacobian` stays the same, but we have to be careful on how to use the function of which we take the derivative.

0.2.7 *Gradient and Hessian*

Next we build the gradient and the Hessian. We can use Sagemath's `jacobian`, but as is clear from above, we need to indicate explicitly the variable names with respect to which to differentiate. An easy solution is to make a space with dummy variable names, and then select the set of variables that correspond to the slot. We use the length of the function argument to find out the dimension of the coordinate space. Once we have computed the Jacobian, we replace (by substitution) the dummy variables by their proper names. Finally, as the gradient is a (co-)vector, we cast it to a vector.

To ensure that (numerical) values for the variables are passed on properly, we need unique names for the variables that make up the space. Therefore we use `id(F)` in the variable names. As these are internal names, the actual variable names are irrelevant; as long as they are unique, it's OK.

utils.sage

```
def gradient(F, v):
    cds = make_coordinates(f"q_{id(F)}", dim=len(v))
    deriv = jacobian(F(cds), cds)
    return vector(deriv.subs(dict(zip(cds, v))))
```

Here is the version that handles differentiation of symbolic functions explicitly, but we will not use it.

utils_tests.sage

```
def gradient(F, v):
    cds = make_coordinates(f"qq_{id(F)}", dim=len(v))
    if isinstance(F, sage.symbolic.function_factory.SymbolicFunction):
        deriv = jacobian(F(*cds), cds)  # Unpack coordinates if F is symbolic
    else:
        deriv = jacobian(F(cds), cds)  # Otherwise, call F with a vector
    return deriv.subs(dict(zip(cds, v)))
```

utils_tests.sage

```
show(gradient(F, v))
```

$$\begin{pmatrix} 3x+4y+8 & 4x+5y+9 \end{pmatrix}$$

Let's substitute some values.

utils_tests.sage

```
show(gradient(F, v)(x=0, y=0))
show(gradient(F, v).subs({v[0]: 0, v[1]: 0}))
show(gradient(F, [0, 0]))
```

$$\begin{pmatrix} 8 & 9 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 9 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 9 \end{pmatrix}$$

Sometimes we want to take the gradient of `F` and use a path as argument.

utils_tests.sage

```
q = vector([literal_function("q_1"), literal_function("q_2")])
show(gradient(F, q))
```

$$\begin{pmatrix} 3q_1+4q_2+8 & 4q_1+5q_2+9 \end{pmatrix}$$

Note that the dependence on `t` is suppressed in the LaTeX output.

To apply the gradient to symbolic functions, we wrap it in a Python function and do the unpacking of the arguments in the function body.

utils_tests.sage

```
def U(q):
    return function("U")(*q)

show(gradient(U, q))
```

$$\begin{pmatrix} \mathrm{D}_0(U)(q_1,q_2) & \mathrm{D}_1(U)(q_1,q_2) \end{pmatrix}$$

This is the Hessian.

utils.sage

```
def hessian(F, v):
    cds = make_coordinates(f"q_{id(F)}", dim=len(v))
    hes = jacobian(jacobian(F(cds), cds), cds)
    return matrix(hes.subs(dict(zip(cds, v))))
```

utils_tests.sage

```
show(hessian(F, q))
```

$$\begin{pmatrix} 3 & 4 \\ 4 & 5 \end{pmatrix}$$

utils_tests.sage

```
def U(q):
    return function("U")(*q)

show(hessian(U, q))
```

$$\begin{pmatrix} \mathrm{D}_{0,0}(U)(q_1,q_2) & \mathrm{D}_{0,1}(U)(q_1,q_2) \\ \mathrm{D}_{0,1}(U)(q_1,q_2) & \mathrm{D}_{1,1}(U)(q_1,q_2) \end{pmatrix}$$

0.2.8 *Differentiation with respect to slots*

To follow the notation of the book, we need to define a python function that computes partial derivatives with respect to the slot of a function; for example, in $\partial_1 L$ the `1` indicates that the partial derivatives are supposed to be taken wrt the coordinate variables. The examples above show us how to approach this problem.

The function `partial` can be called recursively.

utils.sage

```
def partial(f, slot):
    def wrapper(local):
        space = make_space(f"q_{id(f)}_{slot}", dim=len(coordinate(local)))
        if slot == 0:
            selection = [time(space)]
        elif slot == 1:
            selection = coordinate(space)
        elif slot == 2:
            selection = velocity(space)
        deriv = jacobian(f(space), selection)
        return deriv.subs(
            {
                t: time(local),
                **dict(zip(coordinate(space), coordinate(local))),
                **dict(zip(velocity(space), velocity(local))),
            }
        )

    return wrapper
```

Here are some applications.

utils_tests.sage

```
def L_harmonic(m, k):
    def Lagrangian(local):
```

```
188         q = coordinate(local)
189         v = velocity(local)
190         return (1 / 2) * m * square(v) - (1 / 2) * k * square(q)
191 
192     return Lagrangian
193 
194 
195 var('k m', domain="positive")
196 L = L_harmonic(m, k)
```

utils_tests.sage

```
197 space = make_space("x", dim=2)
198 show(space)
199 show(partial(L, 1)(space))
200 show(partial(L, 2)(space))
```

$$t$$

$$(x_1, x_2)$$

$$(\dot{x}_1, \dot{x}_2)$$

$$\begin{pmatrix} -kx_1 & -kx_2 \end{pmatrix}$$

$$\begin{pmatrix} m\dot{x}_1 & m\dot{x}_2 \end{pmatrix}$$

What happens if we use a symbolic potential function `U`?

utils_tests.sage

```
201 def L_generic(m, U):
202     def Lagrangian(local):
203         q = coordinate(local)
204         v = velocity(local)
205         return (1 / 2) * m * square(v) - U(q)
206 
207     return Lagrangian
```

utils_tests.sage

```
208 def U(q):
209     return function("U")(*q)
210 
211 
212 L_gen = L_generic(m, U)
213 show(partial(L_gen, 1)(space))
```

$$\begin{pmatrix} -\frac{\partial}{\partial x_1} U(x_1, x_2) & -\frac{\partial}{\partial x_2} U(x_1, x_2) \end{pmatrix}$$

If we want to substitute a path, we use the function `Gamma` that we build below.

utils_tests.sage

```
214 q = vector([literal_function("q_1"), literal_function("q_2")])
215 show(partial(L, 1)(Gamma(q)(t)))
216 show(partial(L_gen, 1)(Gamma(q)(t)))
```

$$\begin{pmatrix} -kq_1 & -kq_2 \end{pmatrix}$$
$$\begin{pmatrix} -\mathrm{D}_0(U)(q_1,q_2) & -\mathrm{D}_1(U)(q_1,q_2) \end{pmatrix}$$

Later we want to apply differentiation with respect to t to more general objects. As `diff` does not always seem work (to matrices for instance), we use `derivative` for the definition of the operator `D`. We need to implement differentiation of local tuples. Here I take a lazy route, by not deferring this to the `Tuple` class. This works as long as our up tuples will only consist of a time, a coordinate vector, and velocity vector.

utils.sage

```
217 def D(expr):
218     "Derivative wrt time t."
219     if isinstance(expr, Tuple):
220         return up(
221             D(time(expr)),
222             D(coordinate(expr)),
223             D(velocity(expr)),
224         )
225     return derivative(expr, t)
```

Here is an example.

utils_tests.sage

```
226 show(D(q))
```

$$(\dot{q}_1, \dot{q}_2)$$

0.2.9 *Sums, products and composition of functions*

We want to support the summation, products and composition of functions:

$$\begin{aligned}(f+g)(x) &= f(x)+g(x),\\ (fg)(x) &= f(x)g(x),\\ (f\circ g)(x) &= f(g(x).\end{aligned}$$

With recursion, this is easy to code. I use `Sum` rather than `sum` because the latter is a built-in function of Python. For consistency, I defined `Product`, `Compose`, and `Min` also with a capital.

utils.sage

```
227 def Sum(*funcs):
228     if len(funcs) == 1:
229         return lambda x: funcs[0](x)
230     return lambda x: funcs[0](x) + Sum(*funcs[1:])(x)
231
232
233 def Product(*funcs):
234     if len(funcs) == 1:
235         return lambda x: funcs[0](x)
236     return lambda x: funcs[0](x) * Product(*funcs[1:])(x)
237
238
239 def Compose(*funcs):
240     if len(funcs) == 1:
241         return lambda x: funcs[0](x)
242     return lambda x: funcs[0](Compose(*funcs[1:])(x))
243
244
245 def Min(func):
246     return lambda x: -func(x)
```

Here is how this works on general functions.

utils_tests.sage

```
247 f = function('f')
248 g = function('g')
249 h = function('h')
250
251 show(Sum(f, g, h)(x))
252 show(Product(f, g, h)(x))
253 show(Compose(f, g, h)(x))
254 show(diff(Sum(f, g, h)(x), x))
255 show(diff(Product(f, g, h)(x), x))
256 show(diff(Compose(f, g, h)(x), x))
```

$$f(x)+g(x)+h(x)$$

$$f(x)\,g(x)\,h(x)$$

$$f(g(h(x)))$$

$$\frac{\partial}{\partial x}f(x)+\frac{\partial}{\partial x}g(x)+\frac{\partial}{\partial x}h(x)$$

$$g(x)\,h(x)\,\frac{\partial}{\partial x}f(x)+f(x)\,h(x)\,\frac{\partial}{\partial x}g(x)+f(x)\,g(x)\,\frac{\partial}{\partial x}h(x)$$

$$\mathrm{D}_0(f)(g(h(x)))\,\mathrm{D}_0(g)(h(x))\,\frac{\partial}{\partial x}h(x)$$

And some concrete examples.

utils_tests.sage

```
f = lambda x: sin(x)
g = lambda x: x ^ 2 + 1
h = lambda x: 3 * x


show(Sum(f, g, h)(x))
show(Product(f, g, h)(x))
show(Compose(f, g, h)(x))
show(diff(Sum(f, g, h)(x), x))
show(diff(Product(f, g, h)(x), x))
show(diff(Compose(f, g, h)(x), x))
```

$$x^2 + 3x + \sin(x) + 1$$
$$3\left(x^2 + 1\right)x\sin(x)$$
$$\sin\left(9x^2 + 1\right)$$
$$2x + \cos(x) + 3$$
$$3\left(x^2 + 1\right)x\cos(x) + 6x^2\sin(x) + 3\left(x^2 + 1\right)\sin(x)$$
$$18x\cos\left(9x^2 + 1\right)$$

0.2.10 *Local tuples and paths*

The function `Gamma` lifts the coordinate path `q` to a *local tuple*.

There is a subtlety here.

In numerical work, the `vector q` can be implicitly converted to a numpy array whose elements still may depend on the variable t.

As a numpy array cannot be differentiated, we cast `q` to a vector, so that we can apply `D` again.

utils.sage

```
def Gamma(q):
    q = vector(q)
    v = D(q)
    return qv_to_state_path(q, v)
```

When applying to a path, we get this.

utils_tests.sage

```
show(Gamma(q)(t))
```

$$t$$

$$(q_1, q_2)$$

$$\left(\frac{\partial}{\partial t}q_1, \frac{\partial}{\partial t}q_2\right)$$

Finally, here are the projections from Γ to its components.

utils.sage

```
273 def time(local):
274     return local[0]
275
276 def coordinate(local):
277     return local[1]
278
279 def velocity(local):
280     return local[2]
```

utils_tests.sage

```
281 show(velocity(Gamma(q)(t)))
```

$$(\dot{q}_1, \dot{q}_2)$$

0.2.11 *Rotations*

At some point in the book we need to rotate around a given axis in 3D space. ChatGPT gave me the code right away.

utils.sage

```
282 def rotation_matrix(axis, theta):
283     """
284     Return the 3x3 rotation matrix for a rotation of angle theta (in radians)
285     about the given axis. The axis is specified as an iterable of 3 numbers.
286     """
287     # Convert the axis to a normalized vector
288     axis = vector(axis).normalized()
289     x, y, z = axis
290     c = cos(theta)
291     s = sin(theta)
292     t = 1 - c  # common factor
293
294     # Construct the rotation matrix using Rodrigues' formula
295     R = matrix(
296         [
```

```
297             [c + x**2 * t, x * y * t - z * s, x * z * t + y * s],
298             [y * x * t + z * s, c + y**2 * t, y * z * t - x * s],
299             [z * x * t - y * s, z * y * t + x * s, c + z**2 * t],
300         ]
301     )
302     return R
```

CHAPTER 1

1.4 COMPUTING ACTIONS

1.4.1 *Standard import*

I make an org file for each separate section of the book. The code gets tangled to `utils1.4.sage` when the code serves a goal for later sections, and otherwise to `section1.4.sage`. This allows me to run the sage scripts on the prompt, however this implies that I have to load the relevant utility files.

section1.4.sage

```
303 import numpy as np
304
305 load(
306     "utils.sage",
307     "utils1.4.sage",
308 )
```

don't tangle

```
309 load("show_expression.sage")
```

1.4.2 *The Lagrangian for a free particle.*

The function `L_free_particle` takes `mass` as an argument and returns the function `Lagrangian` that takes a `local` tuple as an argument and projects it to a velocity.

utils1.4.sage

```
310 def L_free_particle(mass):
311     def Lagrangian(local):
312         v = velocity(local)
313         return 1 / 2 * mass * square(v)
314
315     return Lagrangian
```

For the next step, we need the variable m for the mass of the particle and a coordinate path.

section1.4.sage

```
316 var('m', domain='positive')
317
318 q = vector(
```

```
    [
        literal_function("q_x"),
        literal_function("q_y"),
        literal_function("q_z"),
    ]
)
```

The Lagrangian of a free particle with mass m applied to the path `Gamma` gives

section1.4.sage

```
show(L_free_particle(m)(Gamma(q)(t)))
```

$$\frac{1}{2} m \sin^2$$

We now compute the integral of Lagrangian `L` along the path `q`, but for this we need a function to carry out 1D integration (along time in our case).

don't tangle

```
def definite_integral(func, start, end, symbolic=False):
    if symbolic:
        return integrate(func(t), t, start, end)
    res, error = numerical_integral(func(t), start, end)
    return res


def Lagrangian_action(L, q, t1, t2, symbolic=False):
    return definite_integral(Compose(L, Gamma(q)), t1, t2, symbolic)
```

However, it turns out that Sagemath already support the definite integral in a library. I don't like to read dt at the end of the integral because it reads like the product of the variables d and t. Instead, I prefer to read $\mathrm{d}t$; for this reason I overwrite the LaTeX formatting of `definite_integral`.

utils1.4.sage

```
from sage.symbolic.integration.integral import definite_integral

def integral_latex_format(*args):
    expr, var, a, b = args
    return (
        fr"\int_{{{a}}}^{{{b}}} "
        + latex(expr)
        + r"\, \textrm{d}\,"
        + latex(var)
    )


definite_integral._print_latex_ = integral_latex_format
```

utils1.4.sage

```
def Lagrangian_action(L, q, t1, t2):
    return definite_integral(Compose(L, Gamma(q))(t), t, t1, t2)
```

Here is the action along a generic path q.

section1.4.sage

```

var("T", domain="positive")
```

section1.4.sage

```
show(latex(Lagrangian_action(L_free_particle(m), q, 0, T)))
```

$$\frac{1}{2}m\left(\int_0^T \dot{q}_x^2\,\mathrm{d}t + \int_0^T \dot{q}_y^2\,\mathrm{d}t + \int_0^T \dot{q}_z^2\,\mathrm{d}t\right)$$

To get a numerical answer, we take the test path of the book.

section1.4.sage

```
test_path = vector([4 * t + 7, 3 * t + 5, 2 * t + 1])
Lagrangian_action(L_free_particle(mass=3), test_path, 0, 10)
```

435

Let's try a harder path.

section1.4.sage

```
hard_path = vector([4 * t + 7, 3 * t + 5, 2 * exp(-t) + 1])

result = Lagrangian_action(L_free_particle(mass=3), hard_path, 0, 10)
show(result)
show(float(result))
```

$$3\left(125e^{20}-1\right)e^{(-20)}+3$$

$$377.9999999938165$$

The value of the integral is different from 435 because the end points on this harder path are not the same as the end points of the test path.

1.4.3 *Path of minimum action*

First some experiments to see whether the code works as intended.

section1.4.sage

```
def make_eta(nu, t1, t2):
    return (t - t1) * (t - t2) * nu(t=t)

nu = vector([sin(t), cos(t), t ^ 2])
show(1 / 3 * make_eta(nu, 3, 4)  + test_path)
```

$$\left(\frac{1}{3}(t-3)(t-4)\sin+4t+7, \frac{1}{3}(t-3)(t-4)\cos+3t+5, \frac{1}{3}(t-3)(t-4)t^2+2t+1\right)$$

By adding the `n()` we force the result into one floating point number. (If we don't, the result is long expression with lots of cosines and sines.)

section1.4.sage

```
def varied_free_particle_action(mass, q, nu, t1, t2):
    eta = make_eta(nu, t1, t2)

    def f(eps):
        return Lagrangian_action(L_free_particle(mass), q + eps * eta, t1, t2).n()

    return f

show(varied_free_particle_action(3.0, test_path, nu, 0.0, 10.0)(0.001))
```

436.291214285714

By comparing our result with that of the book, we see we are still on track. Now use Sagemath's `find_local_minimum` to minimize over ϵ.

section1.4.sage

```
res = find_local_minimum(
    varied_free_particle_action(3.0, test_path, nu, 0, 10), -2, 1
)
show(res)
```

(435.000000000000, 0.0)

We see that the optimal $\epsilon = 0$, and we retrieve our earlier value of the Lagrangian action.

1.4.4 *Finding minimal trajectories*

The `make_path` function uses a Lagrangian polynomial to interpolate a given set of data.

utils1.4.sage

```
def Lagrangian_polynomial(ts, qs):
    return RR['x'].lagrange_polynomial(list(zip(ts, qs)))
```

While a Lagrangian polynomial gives an excellent fit on the fitted points, its behavior in between these points can be quite wild. Let us test the quality of the fit before using this interpolation method. From the book we know we need to fit $\cos(t)$ on $t \in [0, \pi/2]$, so let us try this first before trying to find the optimal path for the harmonic Lagrangian. Since $\cos^2 x + \sin^2 x = 1$, we can use this relation to check the quality of derivative of the fitted polynomial at the same time. The result is better than I expected.

section1.4.sage

```
ts = np.linspace(0, pi / 2, 5)
qs = [cos(t).n() for t in ts]
lp = Lagrangian_polynomial(ts, qs)
ts = np.linspace(0, pi / 2, 20)
Cos = [lp(x=t).n() for t in ts]
Sin = [-lp.derivative(x)(x=t).n() for t in ts]
Zero = [abs(Cos[i] ^ 2 + Sin[i] ^ 2 - 1) for i in range(len(ts))]
show(max(Zero))
```

$$0.00735247812614714$$

For `math_path` we use numpy's `linspace` instead of the linear interpolants of the book. Note that the coordinate paths above are `vector` functions, so `make_path` should return this also.

section1.4.sage

```
def make_path(t0, q0, t1, q1, qs):
    ts = np.linspace(t0, t1, len(qs) + 2)
    qs = np.r_[q0, qs, q1]
    return vector([Lagrangian_polynomial(ts, qs)(t)])
```

Here is the harmonic Lagrangian.

utils1.4.sage

```
def L_harmonic(m, k):
    def Lagrangian(local):
        q = coordinate(local)
        v = velocity(local)
        return (1 / 2) * m * square(v) - (1 / 2) * k * square(q)

    return Lagrangian
```

section1.4.sage

```
def parametric_path_action(Lagrangian, t0, q0, t1, q1):
    def f(qs):
        path = make_path(t0, q0, t1, q1, qs=qs)
        return Lagrangian_action(Lagrangian, path, t0, t1)

    return f
```

Let's try this on the path $\cos(t)$. The intermediate values `qs` will be optimized below, whereas `q0` and `q1` remain fixed. Thus, we strip the first and last element of `linspace` to make `qs`. The result tells us what we can expect for the minimal value for the integral over the Lagrangian along the optimal path.

section1.4.sage

```
t0, t1 = 0, pi / 2
q0, q1 = cos(t0), cos(t1)
ts = np.linspace(0, pi / 2, 5)
initial_qs = [cos(t).n() for t in ts][1:-1]
parametric_path_action(L_harmonic(m=1, k=1), t0, q0, t1, q1)(initial_qs)
```

What is the quality of the path obtained by the Lagrangian interpolation?

section1.4.sage

```
def find_path(Lagrangian, t0, q0, t1, q1, n):
    ts = np.linspace(t0, t1, n)
    initial_qs = np.linspace(q0, q1, n)[1:-1]
    minimizing_qs = minimize(
        parametric_path_action(Lagrangian, t0, q0, t1, q1),
        initial_qs,
    )
    return make_path(t0, q0, t1, q1, minimizing_qs)

best_path = find_path(L_harmonic(m=1, k=1), t0=0, q0=1, t1=pi / 2, q1=0, n=5)
result = [
    abs(best_path(t=t).n()[0] - cos(t).n()) for t in np.linspace(0, pi / 2, 10)
]
show(max(result))
```

$$0.000172462354236957$$

Great. All works!
Instead of Exercise 1.5 I plot the Lagrangian as a function of $q(t)$.

section1.4.sage

```
ts = np.linspace(0, pi / 2, 20)
q = vector([cos(t)])
lvalues = [L_harmonic(m=1, k=1)(Gamma(q)(t)).n() for t in ts]
points = list(zip(ts, lvalues))
plot = list_plot(points, color="black", size=30)
plot.axes_labels(["$t$", "$L$"])
plot.save("../figures/Lagrangian.png", figsize=(4, 2))
```

1.5 THE EULER-LAGRANGE EQUATIONS

1.5.1 *The standard imports.*

section1.5.sage

```
import numpy as np

load(
```

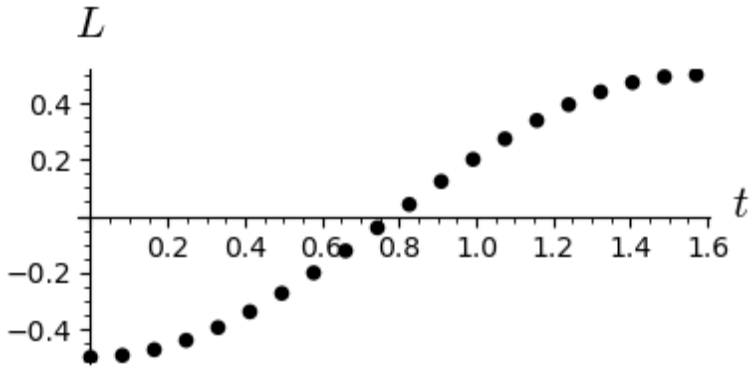

Figure 1: The harmonic Lagrangian as a function of the optimal path $q(t) = \cos t$, $t \in [0, \pi/2]$.

```
    "utils.sage",
    "utils1.4.sage",
    "utils1.5.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

1.5.2 *Derivation of the Lagrange equations*

Harmonic oscillator

Here is a test on the harmonic oscillator. First the path to which we apply `Gamma`.

section1.5.sage

```
var('k m')
assume(k > 0, m > 0)
space = make_named_space(["q_x"])
```

section1.5.sage

```
show(L_harmonic(m, k)(space))
```

$$-\frac{1}{2}kq_x{}^2 + \frac{1}{2}m\dot{q}_x^2$$

We can apply $\partial_1 L$ and $\partial_2 L$ to the formal parameters of the space.

section1.5.sage

```
show(partial(L_harmonic(m, k), 1)(space))
```

$$\left(-kq_x \right)$$

section1.5.sage
```
show(partial(L_harmonic(m, k), 2)(space))
```

$$\begin{pmatrix} m\dot{q}_x \end{pmatrix}$$

It also works on a configuration path $q = (x, y)$ that we lift to a local tuple.

section1.5.sage
```
q = vector([literal_function("q_x"), literal_function("q_y")])

show(partial(L_harmonic(m, k), 1)(Gamma(q)(t)))
```

$$\begin{pmatrix} -kq_x & -kq_y \end{pmatrix}$$

Observe that `partial(L_harmonic(m, k), 1)` maps a local tuple to real number, and `Gamma(q)` maps a time t to a local tuple. Thus, we can consider the compositions of these two functions.

section1.5.sage
```
show(Compose(partial(L_harmonic(m, k), 1), Gamma(q))(t))
```

$$\begin{pmatrix} -kq_x & -kq_y \end{pmatrix}$$

Observe that these results are functions of t, so we can take the derivative with respect to t, which forms the last step to check before building the Euler-Lagrange equations.

section1.5.sage
```
show(D(partial(L_harmonic(m, k), 2)(Gamma(q)(t))))
```

$$\begin{pmatrix} m\ddot{q}_x & m\ddot{q}_y \end{pmatrix}$$

There we are! We can now try the other examples of the book.

Orbital motion

section1.5.sage
```
space = make_named_space(["\\xi", "\\eta"])
q = vector([literal_function("\\xi"), literal_function("\\eta")])
```

section1.5.sage
```
var("mu", domain="positive")

def L_orbital(m, mu):
    def Lagrangian(local):
```

```
458         q = coordinate(local)
459         v = velocity(local)
460         return (1 / 2) * m * square(v) + mu / sqrt(square(q))
461
462     return Lagrangian
```

section1.5.sage

```
463 L = L_orbital(m, mu)
464 show(L(space))
```

$$\frac{1}{2}\left(\dot{\eta}^2+\dot{\xi}^2\right)m+\frac{\mu}{\sqrt{\eta^2+\xi^2}}$$

section1.5.sage

```
465 show(partial(L, 1)(Gamma(q)(t)))
```

$$\left(\begin{array}{cc} -\frac{\mu\xi}{(\eta^2+\xi^2)^{\frac{3}{2}}} & -\frac{\mu\eta}{(\eta^2+\xi^2)^{\frac{3}{2}}} \end{array}\right)$$

section1.5.sage

```
466 show(partial(L, 2)(Gamma(q)(t)))
```

$$\left(\begin{array}{cc} m\dot{\xi} & m\dot{\eta} \end{array}\right)$$

An ideal planar pendulum, Exercise 1.9.a of the book

We need a new space and path in terms of θ and $\dot{\theta}$.

section1.5.sage

```
467 space = make_named_space(["\\theta"])
468 q = vector([literal_function("\\theta")])
```

Here is the Lagrangian. Recall that the coordinates of the space are a vector. As the motion is in one dimension, we just need the first component of `theta`. For `thetadot` we don't have to this since we consider $\dot{\theta}^2$.

utils1.5.sage

```
469 var("m g l")
470 assume(m > 0, g > 0, l > 0)
471
472
473 def L_planar_pendulum(m, g, l):
474     def Lagrangian(local):
475         theta = coordinate(local)[0]
476         theta_dot = velocity(local)
477         T = (1 / 2) * m * l ^ 2 * square(theta_dot)
```

```
478         V = m * g * l * (1 - cos(theta))
479         return T - V
480
481     return Lagrangian
```

section1.5.sage

```
482 L = L_planar_pendulum(m, g, l)
483 show(L(space))
```

$$\frac{1}{2}\,l^2 m\dot{\theta}^2 + glm(\cos(\theta) - 1)$$

section1.5.sage

```
484 show(partial(L, 1)(Gamma(q)(t)))
```

$$\left(\; -glm\sin(\theta)\;\right)$$

section1.5.sage

```
485 show(partial(L, 2)(Gamma(q)(t)))
```

$$\left(\; l^2 m\dot{\theta}\;\right)$$

Henon Heiles potential, Exercise 1.9.b of the book

utils1.5.sage

```
486 def L_Henon_Heiles(m):
487     def Lagrangian(local):
488         q = coordinate(local)
489         x, y = q[:]
490         v = velocity(local)
491         T = (1 / 2) * square(v)
492         V = 1 / 2 * (square(x) + square(y)) + square(x) * y - y**3 / 3
493         return T - V
494
495     return Lagrangian
```

section1.5.sage

```
496 L = L_Henon_Heiles(m)
497 space = make_space("x", dim=2)
498 show(L(space))
```

$$-{x_1}^2 x_2 + \frac{1}{3}\,{x_2}^3 - \frac{1}{2}\,{x_1}^2 + \frac{1}{2}\,\dot{x}_1^2 - \frac{1}{2}\,{x_2}^2 + \frac{1}{2}\,\dot{x}_2^2$$

section1.5.sage

```
499 show(partial(L, 1)(space))
```

$$\begin{pmatrix} -2x_1x_2 - x_1 & -{x_1}^2 + {x_2}^2 - x_2 \end{pmatrix}$$

section1.5.sage

```
500 show(partial(L, 2)(space))
```

$$\begin{pmatrix} \dot{x}_1 & \dot{x}_2 \end{pmatrix}$$

Motion on the 2d sphere, Exercise 1.9.c of the book

section1.5.sage

```
501 var('R', domain="positive")
502
503
504 def L_sphere(m, R):
505     def Lagrangian(local):
506         q = coordinate(local)
507         theta, phi = q[:]
508         v = velocity(local)
509         alpha, beta = v[:]
510         L = m * R * (square(alpha) + square(beta * sin(theta))) / 2
511         return L
512
513     return Lagrangian
```

section1.5.sage

```
514 space = make_named_space(["\\phi", "\\theta"])
515 L = L_sphere(m, R)
516 show(L(space))
```

$$\frac{1}{2}\left(\dot{\theta}^2 \sin(\phi)^2 + \dot{\phi}^2\right) Rm$$

section1.5.sage

```
517 show(partial(L, 1)(space))
```

$$\begin{pmatrix} Rm\dot{\theta}^2 \cos(\phi)\sin(\phi) & 0 \end{pmatrix}$$

section1.5.sage

```
518 show(partial(L, 2)(space))
```

$$\begin{pmatrix} Rm\dot{\phi} & Rm\dot{\theta}\sin(\phi)^2 \end{pmatrix}$$

Higher order Lagrangians

I recently read the books of Larry Susskind on the theoretical minimum for physics. He claims that Lagrangians up to first order derivatives suffice to understand nature, so I skip this part.

1.5.3 *Computing Lagrange's equation*

The Euler-Lagrange equations are simple to implement now that we have a good function for computing partial derivatives. Besides this, for the moment, I prefer `diff(t)` over the `D` operator of the book. Hence, I did not (yet) implement `D` as a function.

EL equations

Here are two EL equations with a slight detail. The first returns a function that takes `q` as an argument and then returns a real number since the argument t is already provided (BTW, not a a number, but as a variable on which the returned function depends.)

don't tangle

```
519 def Lagrange_equations(Lagrangian):
520     def f(q):
521         res = D(partial(Lagrangian, 2)(Gamma(q)(t)))
522         res -= partial(Lagrangian, 1)(Gamma(q)(t))
523         return res
524 
525     return f
```

This second implementation follows the book. It returns a function that takes `q` as an argument, and then returns a function that still depends on t.

utils1.5.sage

```
526 def Lagrange_equations(Lagrangian):
527     def f(q):
528         return Sum(
529             Compose(D, partial(Lagrangian, 2), Gamma(q)),
530             Min(Compose(partial(Lagrangian, 1), Gamma(q))),
531         )
532 
533     return f
```

The free particle

We compute the Lagrange equation for a path linear in t for the Lagrangian of a free particle..

section1.5.sage

```
space = make_space("x", dim=3)
var("a b c a0 b0 c0", domain="real")
test_path = vector([a * t + a0, b * t + b0, c * t + c0])
```

Note that if we do not provide the argument `t` to `l_eq` we receive a function instead of vector.

section1.5.sage

```
l_eq = Lagrange_equations(L_free_particle(m))(test_path)
show(l_eq)
show(l_eq(t))
```

```
<function Sum.<locals>.<lambda> at 0x788badebc5e0>
```

$$\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

section1.5.sage

```
l_eq = Lagrange_equations(L_free_particle(m))(
    [literal_function("x"), literal_function("y"), literal_function('z')]
)
show(l_eq(t))
```

$$\begin{pmatrix} m\ddot{x} & m\ddot{y} & m\ddot{z} \end{pmatrix}$$

Equating this to $(0,0,0)$ shows that the solution of these differential equation are linear in t.

The harmonic oscillator

section1.5.sage

```
var("A phi omega", domain="real")
assume(A > 0)

proposed_path = vector([A * cos(omega * t + phi)])
```

`Lagrange_equations` returns a matrix whose elements correspond to the components of the configuration path q.

section1.5.sage

```
l_eq = Lagrange_equations(L_harmonic(m, k))(proposed_path)(t)
show(l_eq)
```

$$\begin{pmatrix} -Am\omega^2\cos(\omega t+\phi)+Ak\cos(\omega t+\phi) \end{pmatrix}$$

To obtain the contents of this 1×1 matrix, we take the element `[0][0]`.

section1.5.sage

```
show(l_eq[0][0])
```

$$-Am\omega^2\cos(\omega t+\phi)+Ak\cos(\omega t+\phi)$$

Let's factor out the cosine.

section1.5.sage

```
show(l_eq[0][0].factor())
```

$$-\left(m\omega^2-k\right)A\cos(\omega t+\phi)$$

Kepler's third law

section1.5.sage

```
space = make_named_space(["r", "\\phi"])
var("G m m1 m2")
assume(G > 0, m > 0, m1 > 0, m2 > 0)


def L_central_polar(m, V):
    def Lagrangian(local):
        r, phi = coordinate(local)
        # r, phi = q[:]
        qdot = velocity(local)
        rdot, phidot = velocity(local)  # qdot[:]
        T = 1 / 2 * m * (square(rdot) + square(r * phidot))
        return T - V(r)

    return Lagrangian


def gravitational_energy(G, m1, m2):
    def f(r):
        return -G * m1 * m2 / r

    return f
```

section1.5.sage

```
V = gravitational_energy(G, m1, m2)
L = L_central_polar(m, gravitational_energy(G, m1, m2))
show(L(space))
```

$$\frac{1}{2}\left(\dot{\phi}^2r^2+\dot{r}^2\right)m+\frac{Gm_1m_2}{r}$$

section1.5.sage

```
l_eq = Lagrange_equations(L_central_polar(m, gravitational_energy(G, m1, m2)))(
    [literal_function("r"), literal_function("\\phi")]
)(t)
```

section1.5.sage

```
show(l_eq[0][1].factor() == 0)
```

$$(r\ddot{\phi} + 2\dot{\phi}\dot{r})mr = 0$$

In this equation, let's divide by mr to get $r\ddot{\phi} + 2\dot{\phi}\dot{r} = 0$, which is equal to $\partial_t(\dot{\phi}r^2) = 0$. This implies that $\dot{\phi}r^2 = C$, i.e., a constant. If $r \neq 0$ and constant, which we should assume according to the book, then we see that $\dot{\phi}$ is constant, so the two bodies rotate with constant angular speed around each other.

What can we say about the other equation?

section1.5.sage

```
show(l_eq[0][0] == 0)
```

$$-mr\dot{\phi}^2 + m\ddot{r} + \frac{Gm_1m_2}{r^2} = 0$$

As r is constant according to the book, $\ddot{r} = 0$. By dividing by $m := m_1m_2/(m_1 + m_2)$, this equation reduces to $r^3\dot{\phi}^2 = G(m_1 + m_2)$, which is the form we were to find according to the exercise.

Exercise 1.12 and 1.13

Exercise 1.12 is just copy paste work; I skip this. Exercise 1.13 seems not relevant for physics, so I skip this one too.

1.6 HOW TO FIND LAGRANGIANS

1.6.1 *Standard imports*

section1.6.sage

```
import numpy as np

load(
    "utils.sage",
    "utils1.4.sage",
    "utils1.5.sage",
    "utils1.6.sage",
)

var("t x y", domain="real")
```

don't tangle
```
load("show_expression.sage")
```

1.6.2 *Constant acceleration*

We start with making the coordinates and velocities.

section1.6.sage
```
space = make_named_space(["x", "y"])
```

utils1.6.sage
```
var("g m")

def L_uniform_acceleration(m, g):
    def Lagrangian(local):
        q = coordinate(local)
        v = velocity(local)
        y = q[1]
        T = 1 / 2 * m * v * v
        V = m * g * y
        return T - V

    return Lagrangian
```

section1.6.sage
```
l_eq = Lagrange_equations(L_uniform_acceleration(m, g))(
    [literal_function("x"), literal_function("y")]
)
show(l_eq)
```

$$\begin{pmatrix} m\ddot{x} & gm + m\ddot{y} \end{pmatrix}$$

1.6.3 *Central force field*

utils1.6.sage
```
def L_central_rectangular(m, U):
    def Lagrangian(local):
        q = coordinate(local)
        v = velocity(local)
        T = 1 / 2 * m * square(v)
        return T - U(sqrt(square(q)))

    return Lagrangian
```

Let us first try this on a concrete potential function.

section1.6.sage

```
def U(r):
    return 1 / r
```

section1.6.sage

```
show(
    Lagrange_equations(L_central_rectangular(m, U))(
        [literal_function("x"), literal_function("y")]
    )
)
```

$$\left(m\ddot{x} - \frac{x}{(x^2+y^2)^{\frac{3}{2}}} \quad m\ddot{y} - \frac{y}{(x^2+y^2)^{\frac{3}{2}}} \right)$$

Now we try it on a general central potential. We should a different name for the Python function than the one we use in the `function`. Thus, we take `V`, not `U`. Recall that to differentiate symbolic function that receives multiple arguments, the list of arguments needs to be unpacked.

section1.6.sage

```
def U(r):
    return function("V")(r)

show(
    Lagrange_equations(L_central_rectangular(m, U))(
        [literal_function("x"), literal_function("y")]
    )
)
```

$$\left(m\ddot{x} + \frac{x\mathrm{D}_0(V)\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}} \quad m\ddot{y} + \frac{y\mathrm{D}_0(V)\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}} \right)$$

1.6.4 *Coordinate transformations*

To get things straight, the function F is the transformation of the coordinates x' to x, i.e., $x = F(t, x')$. The function C lifts the transformation F to the phase space, so it transforms $\Gamma(q')$ to $\Gamma(q)$. Note that `partial(F, 0)` is a matrix while while `partial(F, 1)(local) * velocity(local)` is a vector. To add these two, we need to cast `partial(F, 0)` to a vector.

utils1.6.sage

```
def F_to_C(F):
    def f(local):
        return up(
            time(local),
            F(local),
```

```
638             vector(partial(F, 0)(local))
639             + partial(F, 1)(local) * velocity(local),
640         )
641
642     return f
```

1.6.5 *polar coordinates*

utils1.6.sage

```
643 def p_to_r(local):
644     r, phi = coordinate(local)
645     return vector([r * cos(phi), r * sin(phi)])
```

We apply `F_to_C` and `p_to_r` to several examples, to test and to understand how they collaborate. We need to make the appropriate variables for the space in terms of r and ϕ.

section1.6.sage

```
646 space = make_named_space(["r", "\\phi"])
647 show(p_to_r(space))
```

$$(r\cos(\phi), r\sin(\phi))$$

section1.6.sage

```
648 show((partial(p_to_r, 0)(space)))
```

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

section1.6.sage

```
649 show((partial(p_to_r, 1)(space)))
```

$$\begin{pmatrix} \cos(\phi) & -r\sin(\phi) \\ \sin(\phi) & r\cos(\phi) \end{pmatrix}$$

section1.6.sage

```
650 show(F_to_C(p_to_r)(space))
```

$$\begin{aligned} &t \\ &(r\cos(\phi), r\sin(\phi)) \\ &(-\dot{\phi}r\sin(\phi) + \dot{r}\cos(\phi), \dot{\phi}r\cos(\phi) + \dot{r}\sin(\phi)) \end{aligned}$$

We can see what happens for the Lagrangian for the central force in polar coordinates.

utils1.6.sage

```
651 def L_central_polar(m, U):
652     def Lagrangian(local):
653         return Compose(L_central_rectangular(m, U), F_to_C(p_to_r))(local)
654         # return L_central_rectangular(m, U)(F_to_C(p_to_r)(local))
655 
656     return Lagrangian
```

section1.6.sage

```
657 show(L_central_polar(m, U)(space).simplify_full())
```

$$\frac{1}{2}m\dot{\phi}^2r^2+\frac{1}{2}m\dot{r}^2-V\left(r\right)$$

section1.6.sage

```
658 expr = Lagrange_equations(L_central_polar(m, U))(
659     [literal_function("r"), literal_function("\\phi")]
660 ).simplify_full().expand()
661 
662 show(expr[0][0])
663 show(expr[0][1])
```

$$-mr\dot{\phi}^2+m\ddot{r}+\frac{r\mathrm{D}_0\left(V\right)\left(\sqrt{r^2}\right)}{\sqrt{r^2}}$$

$$mr^2\ddot{\phi}+2mr\dot{\phi}\dot{r}$$

1.6.6 *Coriolis and centrifugal forces*

utils1.6.sage

```
664 def L_free_rectangular(m):
665     def Lagrangian(local):
666         v = velocity(local)
667         return 1 / 2 * m * v * v
668 
669     return Lagrangian
670 
671 
672 def L_free_polar(m):
673     def Lagrangian(local):
674         return L_free_rectangular(m)(F_to_C(p_to_r)(local))
675 
676     return Lagrangian
677 
678 
679 def F(Omega):
680     def f(local):
```

```
681             t = time(local)
682             r, theta = coordinate(local)
683             return vector([r, theta + Omega * t])
684
685         return f
686
687
688     def L_rotating_polar(m, Omega):
689         def Lagrangian(local):
690             return L_free_polar(m)(F_to_C(F(Omega))(local))
691
692         return Lagrangian
693
694
695
696     # atan2(y/x) is not accepted when computing the L-ea
697     def r_to_p(local):
698         x, y = coordinate(local)
699         return vector([sqrt(x * x + y * y), atan(y / x)])
700
701
702     def L_rotating_rectangular(m, Omega):
703         def Lagrangian(local):
704             return L_rotating_polar(m, Omega)(F_to_C(r_to_p)(local))
705
706         return Lagrangian
```

section1.6.sage

```
707 space = make_named_space(["x", "y"])
708 var("m Omega r")
709 expr = L_rotating_rectangular(m, Omega)(space).simplify_full()
```

section1.6.sage

```
710 show(expr)
```

$$\frac{1}{2}\Omega^2 m x^2 + \frac{1}{2}\Omega^2 m y^2 - \Omega m \dot{x} y + \Omega m x \dot{y} + \frac{1}{2} m \dot{x}^2 + \frac{1}{2} m \dot{y}^2$$

The simplification of the Lagrange equations takes some time.

don't tangle

```
711 expr = Lagrange_equations(L_rotating_rectangular(m, Omega))(
712     [literal_function("x"), literal_function("y")]
713 ).simplify_full()
714 show(expr)
```

I edited the result a bit by hand.

$$0 = -m\Omega^2 x - 2m\Omega\dot{y} + m\ddot{x},$$
$$0 = -m\Omega^2 y + 2m\Omega\dot{x} + m\ddot{y}.$$

1.6.7 *Constraints, a driven pendulum*

Rather than implementation the formulas of the book at this place, we follow the idea they explain: formulate a Lagrangian in practical coordinates, then formulate the problem in practical coordinates *for that problem*, and then use a coordinate transformation from the problem's coordinates to the Lagrangian coordinates.

Here, the Lagrangian is easiest to express in terms of x and y coordinates, while the pendulum needs an angle θ. So, we need a transformation from θ to x and y. Note that the function `coordinate` returns a vector, and the vector here just contains θ. So, we have to pick the 0 element. Another point is that here `ys` needs to be evaluated at `t`; in the other functions `ys` is just passed on as a function.

utils1.6.sage

```
def dp_coordinates(l, ys):
    "From theta to x, y coordinates."
    def f(local):
        t = time(local)
        theta = coordinate(local)[0]
        return vector([l * sin(theta), ys(t=t) - l * cos(theta)])

    return f
```

utils1.6.sage

```
var('g l m')
def L_pend(m, l, g, ys):
    def Lagrangian(local):
        return L_uniform_acceleration(m, g)(
            F_to_C(dp_coordinates(l, ys))(local)
        )

    return Lagrangian
```

section1.6.sage

```
space = make_named_space(["\\theta"])
ys = literal_function("y")

expr = L_pend(m, l, g, ys)(space).simplify_full()
show(expr)
```

$$\frac{1}{2}\,l^2 m\dot{\theta}^2 + lm\dot{\theta}\sin\left(\theta\right)\dot{y} + glm\cos\left(\theta\right) - gmy + \frac{1}{2}\,m\dot{y}^2$$

1.7 EVOLUTION OF DYNAMICAL STATE

1.7.1 *Standard imports*

utils1.7.sage

```
load(
    "utils.sage",
    "utils1.6.sage",
)
```

section1.7.sage

```
import numpy as np

load(
    "utils.sage",
    "utils1.4.sage",
    "utils1.5.sage",
    "utils1.7.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

1.7.2 *The harmonic oscillator*

utils1.7.sage

```
def Lagrangian_to_accelaration(L):
    def f(local):
        P = partial(L, 2)  # (local)
        F = partial(L, 1)  # (local)
        res = partial(P, 2)(local).solve_left(
            vector(F(local))
            - vector(partial(P, 0)(local))
            - partial(P, 1)(local) * velocity(local)
        )
        return res

    return f
```

We apply this to the harmonic oscillator.

section1.7.sage

```
var("m k")
space = make_space("x", 2)

L = L_harmonic(m, k)
```

section1.7.sage
```
show(Lagrangian_to_accelaration(L)(space))
```

$$\left(-\frac{kx_1}{m}, -\frac{kx_2}{m}\right)$$

1.7.3 *Intermezzo, numerically integrating ODEs with Sagemath*

At a later stage, we want to numerically integrate the system of ODEs that result from the Lagrangian. This works a bit different from what I expected; here are two examples to see the problem.

This code plots a circle.

don't tangle
```
def de(x, y):
    return [y, -x]


sol = desolve_odeint(de(x, y), [1, 0], srange(0, 100, 0.05), [x, y])
pp = list(zip(sol[:, 0], sol[:, 1]))
p = points(pp, color='blue', size=3)
p.save(f'circle.png')
```

However, if replace the differentials by `x` and `y` by constants, I get an error that the integration variables are unknown.

don't tangle
```
def de(x, y):
    return [1, -1]
```

The solution is to replace the numbers by expressions.

utils1.7.sage
```
def convert_to_expr(n):
    return SR(n)
```

And then define the function of differentials like this.

don't tangle
```
def de(x, y):
    return [convert_to_expr(1), convert_to_expr(-1)]
```

Now things work as they should.

1.7.4 *Continuing with the oscillator*

The next function computes the state derivative of the Lagrangian. For the purpose of numerical integration, we cast the result of the derivative of $\mathrm{d}t/\mathrm{d}t = 1$ to an expression.

utils1.7.sage

```
def Lagrangian_to_state_derivative(L):
    acceleration = Lagrangian_to_acceleration(L)
    return lambda state: up(
        convert_to_expr(1), velocity(state), acceleration(state)
    )
```

section1.7.sage

```
show_tuple(Lagrangian_to_state_derivative(L)(space))
```

$$\left(1,\ (\dot{x}_1, \dot{x}_2),\ \left(-\frac{kx_1}{m}, -\frac{kx_2}{m}\right)\right)$$

section1.7.sage

```
def harmonic_state_derivative(m, k):
    return Lagrangian_to_state_derivative(L_harmonic(m, k))
```

section1.7.sage

```
show(harmonic_state_derivative(m, k)(space))
```

```
up(1, (x_1dot, x_2dot), (-k*x_1/m, -k*x_2/m))
```

utils1.7.sage

```
def Lagrange_equations_first_order(L):
    def f(q, v):
        state_path = qv_to_state_path(q, v)(t)
        res = D(state_path)
        res -= Lagrangian_to_state_derivative(L)(state_path)
        return res

    return f
```

section1.7.sage

```
res = Lagrange_equations_first_order(L)(
    vector([literal_function("x"), literal_function("y")]),
    vector([literal_function("v_x"), literal_function("v_y")]),
)
show_tuple(res)
```

$$
\begin{array}{c}
0 \\
\left(-v_x + \frac{\partial}{\partial t}x, -v_y + \frac{\partial}{\partial t}y\right) \\
\left(\frac{kx}{m} + \frac{\partial}{\partial t}v_x, \frac{ky}{m} + \frac{\partial}{\partial t}v_y\right)
\end{array}
$$

1.7.5 *Numerical integration*

For the ode solver, we need to map the state to a plain list.

utils1.7.sage

```
804 def state_to_list(state):
805     return [time(state), *coordinate(state), *velocity(state)]
```

For the `state_advacer` we can use `evolve`, which we will use later too.

utils1.7.sage

```
806 def evolve(state_derivative, ics, times):
807     space = make_space("qq", dim=len(coordinate(ics)))
808     soln = desolve_odeint(
809         des=state_to_list(state_derivative(space)),
810         ics=state_to_list(ics),
811         times=times,
812         dvars=state_to_list(space),
813         rtol=1e-13,
814     )
815     return soln
```

utils1.7.sage

```
816 def state_advancer(state_derivative, ics, T):
817     init_time = time(ics)
818     times = [init_time, init_time + T]
819     soln = evolve(state_derivative, ics, times)
820     return soln[-1]
```

section1.7.sage

```
821 state_advancer(
822     harmonic_state_derivative(m=2, k=1),
823     ics=up(1, vector([1, 2]), vector([3, 4])),
824     T=10,
825 )
```

array([11. , 3.71279173, 5.42062092, 1.61480313, 1.81891042])

1.7.6 *The driven pendulum*

Here is the driver for the pendulum.

utils1.7.sage

```
826 def periodic_drive(amplitude, frequency, phase):
827     def f(t):
828         return amplitude * cos(frequency * t + phase)
829 
830     return f
```

utils1.7.sage

```
831 var("m l g A omega")
832 
833 
834 def L_periodically_driven_pendulum(m, l, g, A, omega):
835     ys = periodic_drive(A, omega, 0)
836 
837     def Lagrangian(local):
838         return L_pend(m, l, g, ys)(local)
839 
840     return Lagrangian
```

section1.7.sage

```
841 space = make_named_space(["\\theta"])
842 show(L_periodically_driven_pendulum(m, l, g, A, omega)(space).simplify_full())
```

$$\frac{1}{2}A^2m\omega^2\sin\left(\omega t\right)^2 - Alm\omega\dot{\theta}\sin\left(\omega t\right)\sin\left(\theta\right) + \frac{1}{2}l^2m\dot{\theta}^2 - Agm\cos\left(\omega t\right) + glm\cos\left(\theta\right)$$

section1.7.sage

```
843 expr = Lagrange_equations(L_periodically_driven_pendulum(m, l, g, A, omega))(
844     [literal_function("\\theta")]
845 ).simplify_full()
846 show(expr)
```

$$\left(\; l^2m\frac{\partial^2}{(\partial t)^2}\theta\left(t\right) - \left(Alm\omega^2\cos\left(\omega t\right) - glm\right)\sin\left(\theta\left(t\right)\right) \;\right)$$

section1.7.sage

```
847 show(
848     Lagrangian_to_accelaration(
849         L_periodically_driven_pendulum(m, l, g, A, omega)
850     )(space).simplify_full()
851 )
```

$$\left(\frac{\left(A\omega^2\cos\left(\omega t\right)-g\right)\sin\left(\theta\right)}{l}\right)$$

section1.7.sage

```
def pend_state_derivative(m, l, g, A, omega):
    return Lagrangian_to_state_derivative(
        L_periodically_driven_pendulum(m, l, g, A, omega)
    )
```

section1.7.sage

```
expr = pend_state_derivative(m, l, g, A, omega)(space)
show(velocity(expr).simplify_full())
```

$$\left(\frac{\left(A\omega^2\cos\left(\omega t\right)-g\right)\sin\left(\theta\right)}{l}\right)$$

utils1.7.sage

```
def principal_value(cut_point):
    def f(x):
        return (x + cut_point) % (2 * np.pi) - cut_point

    return f
```

section1.7.sage

```
def plot_driven_pendulum(A, T, step_size=0.01):
    times = srange(0, T, step_size, include_endpoint=True)
    soln = evolve(
        pend_state_derivative(m=1, l=1, g=9.8, A=A, omega=2 * sqrt(9.8)),
        ics=up(0, vector([1]), vector([0])),
        times=times,
    )
    thetas = soln[:, 1]
    pp = list(zip(times, thetas))
    p = points(pp, color='blue', size=3)
    p.save(f'../figures/driven_pendulum_{A:.2f}.png')

    thetas = principal_value(np.pi)(thetas)
    pp = list(zip(times, thetas))
    p = points(pp, color='blue', size=3)
    p.save(f'../figures/driven_pendulum_{A:.2f}_principal_value.png')

    thetadots = soln[:, 2]
    pp = list(zip(thetas, thetadots))
    p = points(pp, color='blue', size=3)
    p.save(f'../figures/driven_pendulum_{A:.2f}_trajectory.png')

```

So now we make the plot.

section1.7.sage

```
plot_driven_pendulum(A=0.1, T=100, step_size=0.01)
```

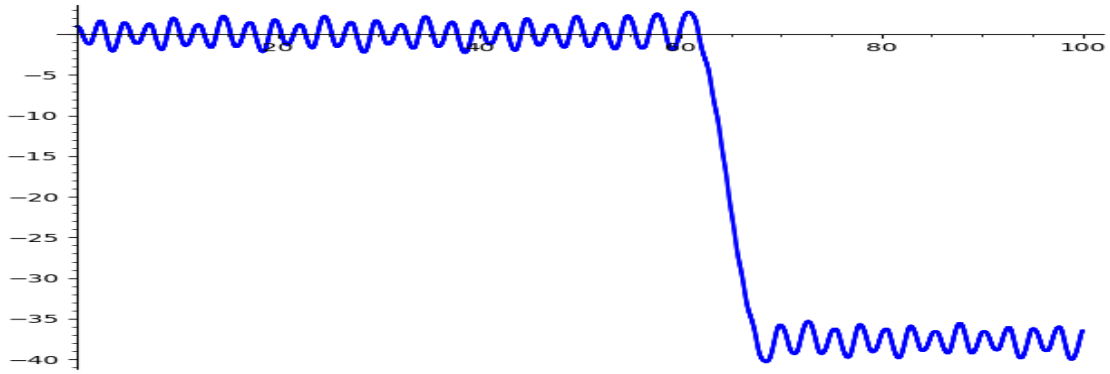

Figure 2: The angle of the vertically driven pendulum as a function of time. Obviously, around $t = 80$, the pendulum makes a few revolutions, and then starts to wobble again.

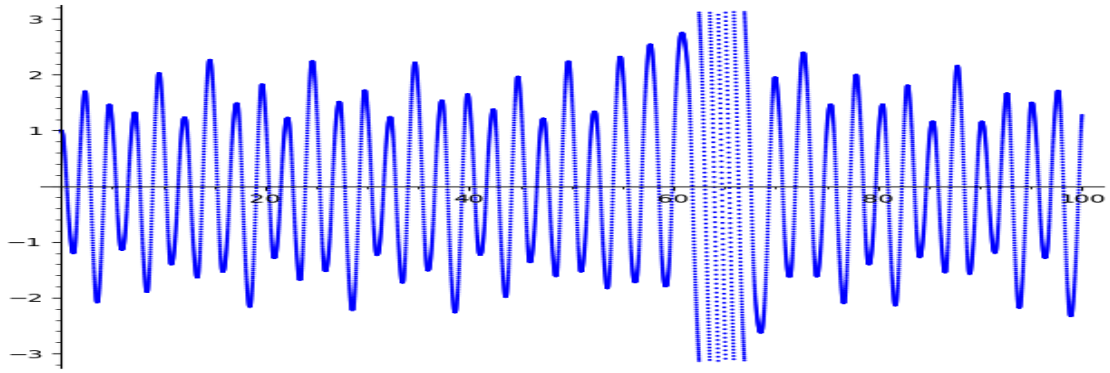

Figure 3: The angle on $(-\pi, \pi]$.

1.8 CONSERVED QUANTITIES

1.8.1 *Standard imports*

section1.8.sage

```
886 import numpy as np
887
888 load(
889     "utils.sage",
890     "utils1.4.sage",
891     "utils1.5.sage",
892     "utils1.6.sage",
893 )
```

don't tangle

```
894 load("show_expression.sage")
```

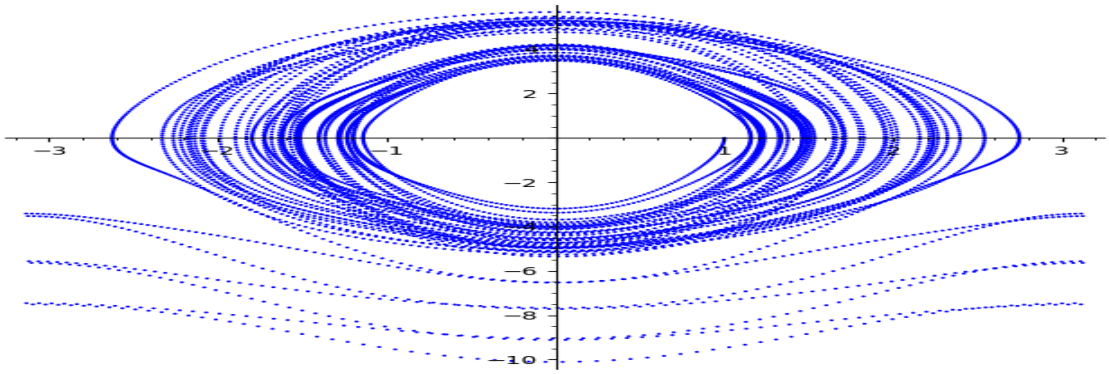

Figure 4: The trajectory of θ and $\dot{\theta}$.

1.8.2 *1.8.2 Energy Conservation*

From the Lagrangian we can construct the energy function. Here are two implementations, where the second is closed to the one in the book. I have my doubts whether this is carrying functional programming a bit too far; I prefer the first implementation. Note that we should cast $P = \partial_2 L$ to a `vector` so that `P * v` becomes a number instead of a 1×1 matrix.

section1.8.sage

```
895 def Lagrangian_to_energy(L):
896     P = partial(L, 2)
897
898     def f(local):
899         v = velocity(local)
900         return vector(P(local)) * v - L(local)
901
902     return f
903
904
905 def Lagrangian_to_energy(L):
906     P = partial(L, 2)
907     return lambda local: Sum(Product(Compose(vector, P), velocity), Min(L))(
908         local
909     )
```

1.8.3 *Central Forces in Three Dimensions*

Instead of building the kinetic energy in spherical coordinates, I am going to use the ideas that have been expounded in the book in earlier sections: define the Lagrangian in convenient coordinates, and then use a coordinate transform to obtain it in coordinates that show the symmetries of the system.

section1.8.sage

```
var("t x y", domain="real")

space = make_named_space(["r", "\\theta", "\\phi"])
```

This the transformation from spherical to 3D rectangular coordinates.

section1.8.sage

```
def s_to_r(sperical_state):
    r, theta, phi = coordinate(sperical_state)[:]
    return vector(
        [r * sin(theta) * cos(phi), r * sin(theta) * sin(phi), r * cos(theta)]
    )
```

section1.8.sage

```
def U(r):
    return function("V")(r)

def L_3D_central(m, V):
    def Lagrangian(local):
        return L_central_rectangular(m, V)(F_to_C(s_to_r)(local))

    return Lagrangian
```

section1.8.sage

```
show(partial(L_3D_central(m, U), 1)(space).simplify_full())
```

$$\left(\frac{m\dot{\phi}^2 r|r|\sin(\theta)^2 + mr\dot{\theta}^2|r| - r\mathrm{D}_0(V)(|r|)}{|r|} \quad m\dot{\phi}^2 r^2 \cos(\theta)\sin(\theta) \quad 0 \right)$$

section1.8.sage

```
show(partial(L_3D_central(m, U), 2)(space).simplify_full())
```

$$\left(m\dot{r} \quad mr^2\dot{\theta} \quad m\dot{\phi}r^2\sin(\theta)^2 \right)$$

section1.8.sage

```
def ang_mom_z(m):
    def f(rectangular_state):
        xyx = coordinate(rectangular_state)
        v = velocity(rectangular_state)
        return xyx.cross_product(m * v)[2]

    return f


show(Compose(ang_mom_z(m), F_to_C(s_to_r))(space).simplify_full())
```

$$m\dot{\phi}r^2 \sin(\theta)^2$$

This is check that $E = T + V$.

section1.8.sage

```
show(Lagrangian_to_energy(L_central_spherical(m, U))(space).simplify_full())
```

1.8.4 *The Restricted Three-Body Problem*

I decompose the potential energy function into smaller functions; I find the implementation in the book somewhat heavy.

section1.8.sage

```
var("G M0 M1 a")
assume(G > 0, M0 > 0, M1 > 0, a > 0)


def distance(x, y):
    return sqrt(square(x - y))


def angular_freq(M0, M1, a):
    return sqrt(G * (M0 + M1) / a ^ 3)


def V(a, M0, M1, m):
    Omega = angular_freq(M0, M1, a)
    a0, a1 = M1 / (M0 + M1) * a, M0 / (M0 + M1) * a

    def f(t, origin):
        pos0 = -a0 * vector([cos(Omega * t), sin(Omega * t)])
        pos1 = a1 * vector([cos(Omega * t), sin(Omega * t)])
        r0 = distance(origin, pos0)
        r1 = distance(origin, pos1)
        return -G * m * (M0 / r0 + M1 / r1)

    return f


def L0(m, V):
    def f(local):
        t, q, v = time(local), coordinate(local), velocity(local)
        return 1 / 2 * m * square(v) - V(t, q)

    return f
```

For the computer it's easy to compute the energy, but the formula is pretty long.

section1.8.sage

```
space = make_named_space(["x", "y"])
show(Lagrangian_to_energy(L0(m, V(a, M0, M1, m)))(space).simplify_full().expand())
```

I skip the rest of the code of this part as it is just copy work from the mathematical formulas.

1.8.5 *Noether's theorem*

section1.8.sage

```
def F_tilde(angle_x, angle_y, angle_z):
    def f(local):
        return (
            rotation_matrix([1, 0, 0], angle_x)
            * rotation_matrix([0, 1, 0], angle_y)
            * rotation_matrix([0, 0, 1], angle_z)
            * coordinate(local)
        )

    return f
```

section1.8.sage

```
space = make_named_space(["x", "y", "z"])

var("s t u")
```

Let's see what we get when we exercise a rotation of s radians round the x axis.

section1.8.sage

```
def Rx(s):
    return F_tilde(s, 0, 0)(space)


show(Rx(s))
show(diff(Rx(s), s)(s=0))
```

$$(x, y\cos(s) - z\sin(s), z\cos(s) + y\sin(s))$$

$$(0, -z, y)$$

And now we check the result of the book. The computation of `D F_tilde` is somewhat complicated. Observe that `F_tilde` is a function of the angles, but returns a function that takes `local` as argument. We want to differentiate the function $(s,t,u) \to \tilde{F}(s,t,u)(l)$ where l is the fixed local point. This gives the first part: `jacobian(F_tilde(s, t, u)(space), (s, t, u))`. Finally, we can fill in $s = t = u = 0$.

As for the result, I don't see why my result differs by a minus sign from the result in the book.

section1.8.sage

```
992 def the_Noether_integral(local):
993     L = L_central_rectangular(m, U)
994     DF0 = jacobian(F_tilde(s, t, u)(local), (s, t, u))(s=0, t=0, u=0)
995     return partial(L, 2)(local) * DF0
996
997
998 show(the_Noether_integral(space))
```

$$\begin{pmatrix} -m\dot{y}z + my\dot{z} & m\dot{x}z - mx\dot{z} & -m\dot{x}y + mx\dot{y} \end{pmatrix}$$

CHAPTER 3

3.1 HAMILTON'S EQUATIONS

3.1.1 *Standard imports*

utils3.1.sage

```
load(
    "utils.sage",
    "utils1.6.sage",
)
```

section3.1.sage

```
import numpy as np


load(
    "utils.sage",
    "utils1.5.sage",
    "utils1.6.sage",
    "utils3.1.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

3.1.2 *Exercise 3.1*

We will use the code below to solve Exercise 3.1.

3.1.3 *Computing Hamilton's equations*

utils3.1.sage

```
def Hamilton_equations(Hamiltonian):
    def f(q, p):
        state_path = qp_to_H_state_path(q, p)(t)
        return D(state_path) - Hamiltonian_to_state_derivative(Hamiltonian)(
            state_path
        )
```

```
1021
1022        return f
```

utils3.1.sage

```
1023    def Hamiltonian_to_state_derivative(Hamiltonian):
1024        def f(H_state):
1025            return up(
1026                SR(1),
1027                vector(partial(Hamiltonian, 2)(H_state)),
1028                -vector(partial(Hamiltonian, 1)(H_state)),
1029            )
1030
1031        return f
```

utils3.1.sage

```
1032    def qp_to_H_state_path(q, p):
1033        def f(t):
1034            return up(t, q(t=t), p(t=t))
1035
1036        return f
```

utils3.1.sage

```
1037    def momentum(H_state):
1038        return H_state[2]
```

In the book, they use just the first and second component of `q` to pass on to the potential. I don't see why, so I keep it general.

section3.1.sage

```
1039    def H_rectangular(m, V):
1040        def f(state):
1041            q, p = coordinate(state), velocity(state)
1042            return square(p) / 2 / m + V(q)
1043
1044        return f
```

I discovered that if I use the same name for a python function (here `V`) as the symbolic function (here `U`), I get errors. Therefore I use `U` instead of `V` in `function("U")`. Recall, to use symbolic functions in differentiation, the symbolic function want the argument list unpacked.

section3.1.sage

```
1045    def V(q):
1046        return function("U")(*q)
```

Let's try what we built.

section3.1.sage

```
var("m")
q = vector([literal_function("q_x"), literal_function("q_y")])
p = vector([literal_function("p_x"), literal_function("p_y")])
```

section3.1.sage

```
H_state = qp_to_H_state_path(q, p)(t)
show(H_state)
```

```
up(t, (q_x(t), q_y(t)), (p_x(t), p_y(t)))
```

section3.1.sage

```
H = H_rectangular
show(H(m, V)(H_state))
```

$$\frac{p_x^2 + p_y^2}{2\,m} + U\left(q_x, q_y\right)$$

section3.1.sage

```
show(partial(H(m, V), 1)(H_state))
```

$$\begin{pmatrix} \mathrm{D}_0\left(U\right)\left(q_x, q_y\right) & \mathrm{D}_1\left(U\right)\left(q_x, q_y\right) \end{pmatrix}$$

section3.1.sage

```
show_tuple(Hamiltonian_to_state_derivative(H(m, V))(H_state))
```

$$\begin{aligned} &1 \\ &\left(\frac{p_x}{m}, \frac{p_y}{m}\right) \\ &\left(-\mathrm{D}_0\left(U\right)\left(q_x, q_y\right), -\mathrm{D}_1\left(U\right)\left(q_x, q_y\right)\right) \end{aligned}$$

section3.1.sage

```
show_tuple(Hamilton_equations(H(m, V))(q, p))
```

$$\begin{aligned} &0 \\ &\left(-\frac{p_x}{m} + \frac{\partial}{\partial t} q_x, -\frac{p_y}{m} + \frac{\partial}{\partial t} q_y\right) \\ &\left(\mathrm{D}_0\left(U\right)\left(q_x, q_y\right) + \frac{\partial}{\partial t} p_x, \mathrm{D}_1\left(U\right)\left(q_x, q_y\right) + \frac{\partial}{\partial t} p_y\right) \end{aligned}$$

3.2 THE LEGENDRE TRANSFORMATION

To understand the code of the book, observe the following. When $F(v) = 1/2 v^T M v + b^T v + c$, then $\partial_v F = Mv + b$, and so $\partial_v F(v = 0) = b$, because $M0 = 0$. Likewise, if F has orders of v higher than 2, then $\partial_v^2 F(v = 0) = M$. In code, $\partial_v F(v = 0)$ is `gradient(F, zero)`. Note further that—at least for the examples we consider—the argument `w` corresponds to a moment, and `zero` is a velocity vector with the same dimension of `w`.

utils3.1.sage

```
1057 def Legendre_transform(F):
1058     def G(w):
1059         zero = [0] * len(w)
1060         b = gradient(F, zero)
1061         M = hessian(F, zero)
1062         v = M.solve_left(w - b)
1063         return w * v - F(v)
1064 
1065     return G
```

utils3.1.sage

```
1066 def Lagrangian_to_Hamiltonian(Lagrangian):
1067     def f(H_state):
1068         t = time(H_state)
1069         q = coordinate(H_state)
1070         p = momentum(H_state)
1071 
1072         def L(qdot):
1073             return Lagrangian(up(t, q, qdot))
1074 
1075         return Legendre_transform(L)(p)
1076 
1077     return f
```

section3.1.sage

```
1078 res = Lagrangian_to_Hamiltonian(L_central_rectangular(m, V))(H_state)
1079 show(res)
```

$$-\frac{1}{2} m \left(\frac{p_x^2}{m^2} + \frac{p_y^2}{m^2} \right) + \frac{p_x^2}{m} + \frac{p_y^2}{m} + U(q_x, q_y)$$

section3.1.sage

```
1080 show(res.simplify_full())
```

$$\frac{2mU(q_x, q_y) + p_x^2 + p_y^2}{2m}$$

section3.1.sage

```
var("m g l")
q = vector([literal_function("q_x")])
p = vector([literal_function("p_x")])
```

Here is exercise 3.1.

section3.1.sage

```
# space = make_named_space(["\\theta"])
H_state = qp_to_H_state_path(q, p)(t)
show(Lagrangian_to_Hamiltonian(L_planar_pendulum(m, g, l))(H_state))
```

$$-glm(\cos(q_x) - 1) + \frac{p_x^2}{2l^2m}$$

section3.1.sage

```
q = vector([literal_function("q_x"), literal_function("q_y")])
p = vector([literal_function("p_x"), literal_function("p_y")])
H_state = qp_to_H_state_path(q, p)(t)
show(Lagrangian_to_Hamiltonian(L_Henon_Heiles(m))(H_state))
```

$$q_x^2 q_y - \frac{1}{3}q_y^3 + \frac{1}{2}p_x^2 + \frac{1}{2}p_y^2 + \frac{1}{2}q_x^2 + \frac{1}{2}q_y^2$$

section3.1.sage

```
def L_sphere(m, R):
    def Lagrangian(local):
        theta, phi = coordinate(local)
        thetadot, phidot = velocity(local)
        return 1 / 2 * m * R ^ 2 * (
            square(thetadot) + square(phidot * sin(theta))
        )

    return Lagrangian


var("R")
```

section3.1.sage

```
space = make_named_space(["\\theta", "\\phi"])
show(Lagrangian_to_Hamiltonian(L_sphere(m, R))(H_state).simplify_full())
```

$$\frac{p_x^2 \sin(q_x)^2 + p_y^2}{2R^2 m \sin(q_x)^2}$$

3.4 PHASE SPACE REDUCTION

3.4.1 *The standard imports.*

section3.4.sage

```
1105 import numpy as np
1106
1107 load(
1108     "utils.sage",
1109     "utils3.1.sage",
1110 )
1111
1112 var("t x y", domain="real")
```

don't tangle

```
1113 load("show_expression.sage")
```

3.4.2 *Motion in a central potental*

section3.4.sage

```
1114 var("m")
1115
1116 V = function("V")
1117
1118 def L_polar(m, V):
1119     def Lagrangian(local):
1120         r, phi = coordinate(local)
1121         rdot, phidot = velocity(local)
1122         T = 1 / 2 * m * (square(rdot) + square(r * phidot))
1123         return T - V(r)
1124
1125     return Lagrangian
```

section3.4.sage

```
1126 space = make_named_space(["r", "\\phi"])
1127 show(space)
```

$$t$$
$$(r, \phi)$$
$$(\dot{r}, \dot{\phi})$$

section3.4.sage

```
1128 q = vector([literal_function("r"), literal_function("\\phi")])
1129 p = vector([literal_function(r"p_r"), literal_function(r"p_{\phi}")])
```

```
H_state = qp_to_H_state_path(q, p)(t)
show(H_state)
```

$$\begin{aligned} &t \\ &(r, \phi) \\ &(p_r, p_\phi) \end{aligned}$$

section3.4.sage

```
H = Lagrangian_to_Hamiltonian(L_polar(m, V))  # (H_state)
show(H(H_state))
```

$$-\frac{1}{2}m\left(\frac{p_r^2}{m^2}+\frac{p_\phi^2}{m^2r^2}\right)+\frac{p_r^2}{m}+\frac{p_\phi^2}{mr^2}+V(r)$$

For some reason I run into a bug of Sage when calling `simplify_full()` on the above expression. However, by using `expand()` the expression is simplified.

section3.4.sage

```
show(H(H_state).expand())
```

$$\frac{p_r^2}{2m}+\frac{p_\phi^2}{2mr^2}+V(r)$$

Here are the Hamilton equations.

section3.4.sage

```
HE = Hamilton_equations(Lagrangian_to_Hamiltonian(L_polar(m, V)))(q, p)
show(HE)
```

$$\begin{aligned} &0 \\ &\left(-\frac{p_r}{m}+\frac{\partial}{\partial t}r, -\frac{p_\phi}{mr^2}+\frac{\partial}{\partial t}\phi\right) \\ &\left(-\frac{p_\phi^2}{mr^3}+\mathrm{D}_0(V)(r)+\frac{\partial}{\partial t}p_r, \frac{\partial}{\partial t}p_\phi\right) \end{aligned}$$

Realize, we have obtained the LHS of the system of differential equations $Dz(t) - F(t, z(t)) = 0$.

3.5 PHASE SPACE EVOLUTION

3.5.1 *The standard imports.*

section3.5.sage

```
import numpy as np

load(
    "utils.sage",
    "utils1.7.sage",
    "utils3.1.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

3.5.2 *The Hamiltonian for the driven pendulum*

section3.5.sage

```
q = vector([literal_function("\\theta")])
p = vector([literal_function(r"p_\theta")])
H_state = qp_to_H_state_path(q, p)(t)
show(H_state)
```

$$t \mapsto \begin{pmatrix} \theta(t) \\ p_\theta(t) \end{pmatrix}$$

section3.5.sage

```
H = Lagrangian_to_Hamiltonian(
    L_periodically_driven_pendulum(m, l, g, A, omega)
)
expr = H(H_state).simplify_full()
show(expr.expand())
```

$$-\frac{1}{2}A^2 m\omega^2 \cos(\theta(t))^2 \sin(\omega t)^2 + Agm\cos(\omega t) - glm\cos(\theta(t)) + \frac{A\omega p_\theta(t)\sin(\omega t)\sin(\theta(t))}{l} + \frac{p_\theta(t)^2}{2l^2 m}$$

section3.5.sage

```
DH = Hamiltonian_to_state_derivative(H)(H_state)
show(DH[1].simplify_full())
show(DH[2].simplify_full())
```

$$\left(\frac{Alm\omega\sin\left(\omega t\right)\sin\left(\theta\left(t\right)\right)+p_{\theta}\left(t\right)}{l^{2}m}\right)$$

$$\left(-\frac{A\omega\cos\left(\theta\left(t\right)\right)p_{\theta}\left(t\right)\sin\left(\omega t\right)+\left(A^{2}lm\omega^{2}\cos\left(\theta\left(t\right)\right)\sin\left(\omega t\right)^{2}+gl^{2}m\right)\sin\left(\theta\left(t\right)\right)}{l}\right)$$

This is the system derivative, i.e., the LHS of the Hamilton equations.

section3.5.sage

```
def H_pend_sysder(m, l, g, A, omega):
    Hamiltonian = Lagrangian_to_Hamiltonian(
        L_periodically_driven_pendulum(m, l, g, A, omega)
    )

    def f(state):
        return Hamiltonian_to_state_derivative(Hamiltonian)(state)

    return f
```

Then last step is to numerically integrate the HE and make a graph of θ and p_θ.

section3.5.sage

```
times = srange(0, 100, 0.01, include_endpoint=True)
soln = evolve(
    H_pend_sysder(m=1, l=1, g=9.8, A=0.1, omega=2 * sqrt(9.8)),
    ics=up(0, vector([1]), vector([0])),
    times=times,
)
thetas = principal_value(np.pi)(soln[:, 1])
thetadots = soln[:, 2]
pp = list(zip(thetas, thetadots))
p = points(pp, color='blue', size=3)
p.save(f'./../figures/hamiltonian_driven_pendulum_0.01.png')
```

section3.5.sage

```
times = srange(0, 100, 0.005, include_endpoint=True)
soln = evolve(
    H_pend_sysder(m=1, l=1, g=9.8, A=0.1, omega=2 * sqrt(9.8)),
    ics=up(0, vector([1]), vector([0])),
    times=times,
)
thetas = principal_value(np.pi)(soln[:, 1])
```

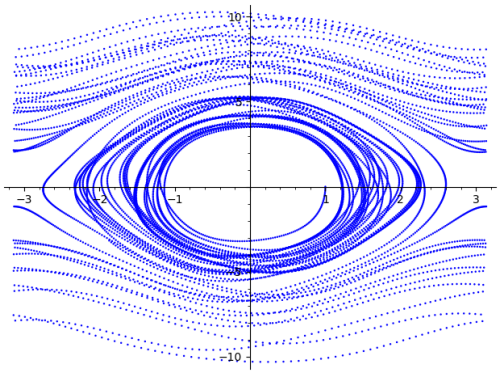

Figure 5: The driven pendulum obtained from numerically integrating the Hamilton equations. The graph is not identital to the one in the book, because of the inherent chaotic behavior.

```
thetadots = soln[:, 2]
pp = list(zip(thetas, thetadots))
p = points(pp, color='blue', size=3)
p.save(f'./../figures/hamiltonian_driven_pendulum_0.005.png')
```

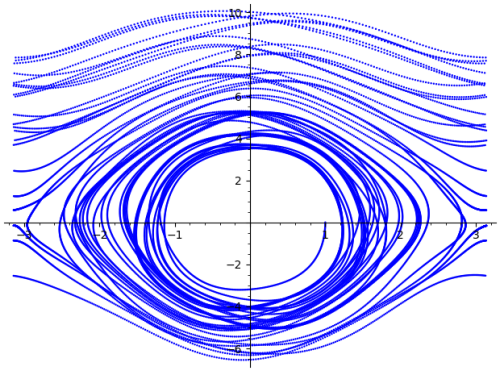

Figure 6: The driven pendulum obtained from numerically integrating the Hamilton equations, but now with time steps 0.005 instead of 0.01. The graphs for both step-sizes seem to be qualitatively the same, but the details are different.

3.9 THE STANDARD MAP

We take a number of uniformly distributed starting points for the paths. The result is very nice.

```
import numpy as np
import matplotlib.pyplot as plt


```

```
n_points = 10000
I = np.zeros(n_points)
theta = np.zeros(n_points)

K = 0.9
two_pi = 2 * np.pi

plt.figure(figsize=(10, 10), dpi=300)

for _ in range(500):
    theta[0] = np.random.uniform(0, two_pi)
    I[0] = np.random.uniform(0, two_pi)
    for i in range(1, n_points):
        I[i] = (I[i - 1] + K * np.sin(theta[i - 1])) % two_pi
        theta[i] = (theta[i - 1] + I[i]) % two_pi
    plt.scatter(theta, I, s=0.01, color='black', alpha=0.1, marker='.')


plt.axis('off')
plt.savefig('../figures/standard_map.png', bbox_inches='tight')
```

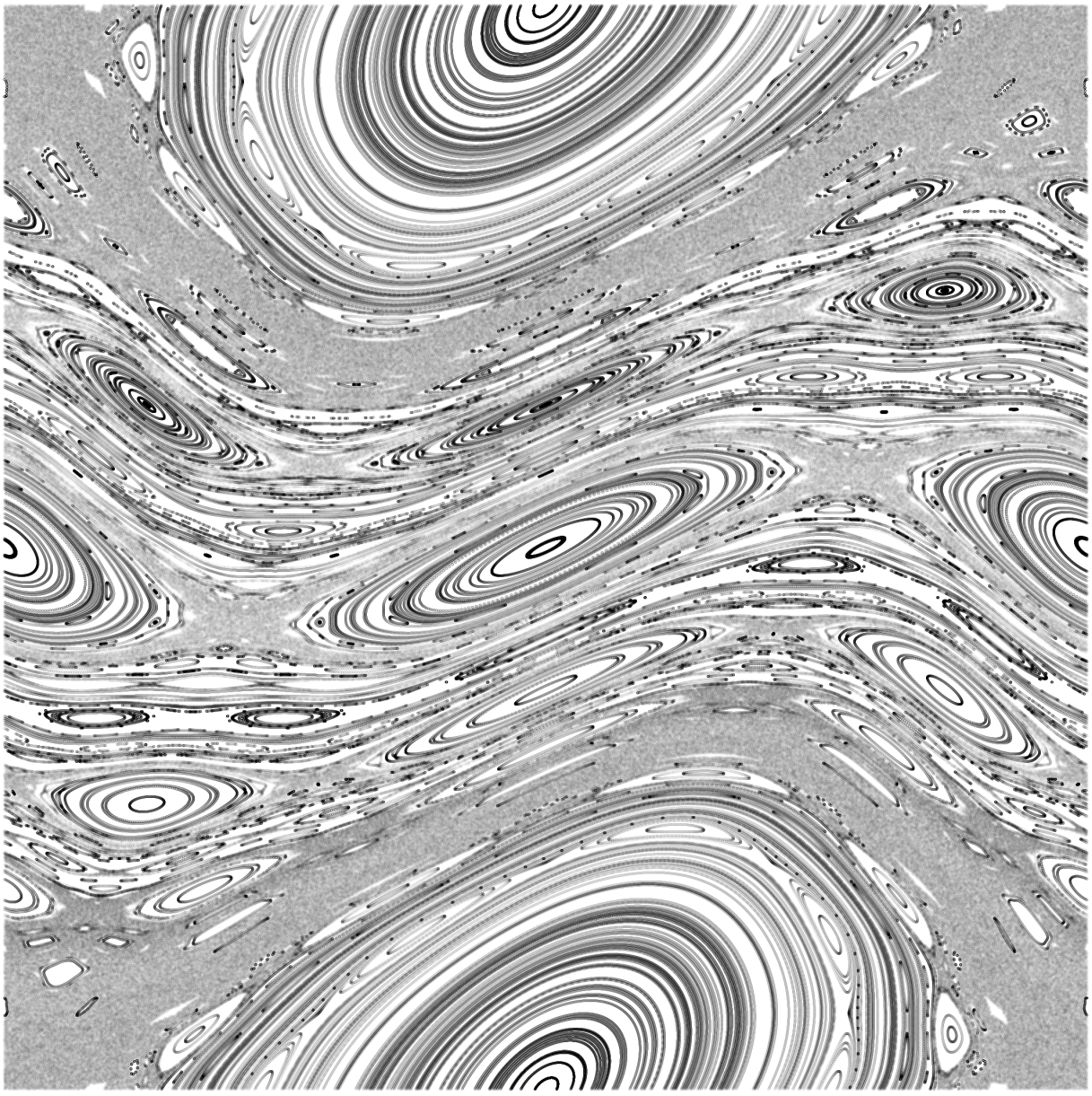

Figure 7: The standard map with $K = 0.6$

CHAPTER 5

5.1 POINT TRANSFORMATIONS

5.1.1 *The standard imports.*

section5.1.sage

```
load(
    "utils.sage",
    "utils1.6.sage",
    "utils3.1.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

5.1.2 *Implementing point transformations*

utils5.1.sage

```
def F_to_CH(F):
    M = partial(F, 1)

    def f(state):
        return up(time(state), F(state), M(state).solve_left(momentum(state)))

    return f
```

Let's test this function.

section5.1.sage

```
var("r, phi, p_r, p_phi", domain="real")
assume(r > 0)

q = vector([r, phi])
p = vector([p_r, p_phi])
local = up(t, q, p)
```

section5.1.sage

```
show((F_to_CH(p_to_r))(local)[0].simplify_full())
show((F_to_CH(p_to_r))(local)[1].simplify_full())
show((F_to_CH(p_to_r))(local)[2].simplify_full())
```

$$t$$

$$(r\cos(\phi), r\sin(\phi))$$

$$\left(\frac{p_r r\cos(\phi) - p_\phi \sin(\phi)}{r}, \frac{p_r r\sin(\phi) + p_\phi \cos(\phi)}{r}\right)$$

The central Hamiltonian in rectangular coordinates.

section5.1.sage

```
def H_central(m, V):
    def f(state):
        x, p = coordinate(state), momentum(state)
        return square(p) / (2 * m) + V(sqrt(square(x)))

    return f
```

Now we convert it to polar coordinates.

section5.1.sage

```
var("r, phi, p_r, p_phi", domain="real")
assume(r > 0)
var("m", domain="positive")
```

section5.1.sage

```
show(
    Compose(H_central(m, function("V")), (F_to_CH(p_to_r)))(local)
    .simplify_full()
    .expand()
)
```

$$\frac{p_r^2}{2m} + \frac{p_\phi^2}{2mr^2} + V(r)$$

The correction term for time dependent Hamiltonians.

utils5.1.sage

```
def F_to_K(F):
    M = partial(F, 1)

    def f(state):
        p = M(state).solve_left(momentum(state))
        return -vector(p) * vector(partial(F, 0)(state))

    return f
```

We apply this to a 2D translation.

utils5.1.sage

```
1260 def translating(v):
1261     def f(state):
1262         return coordinate(state) + v * time(state)
1263 
1264     return f
```

section5.1.sage

```
1265 var("q_x q_y v_x v_y p_x p_y", domain="real")
1266 q = vector([q_x, q_y])
1267 v = vector([v_x, v_y])
1268 p = vector([p_x, p_y])
1269 local = up(t, q, p)
```

section5.1.sage

```
1270 show(F_to_K(translating(v))(local))
```

$$-p_x v_x - p_y v_y$$

Finally, we transform the Hamiltonian of a particle not subject to forces due to a potential field.

section5.1.sage

```
1271 def H_free(m):
1272     def f(state):
1273         return square(momentum(state)) / (2 * m)
1274 
1275     return f
1276 
1277 
1278 def H_prime():
1279     return Sum(
1280         Compose(H_free(m), F_to_CH(translating(v))), F_to_K(translating(v))
1281     )
1282 
1283 
1284 show(H_prime()(local))
```

$$-p_x v_x - p_y v_y + \frac{p_x^2 + p_y^2}{2m}$$

5.2 GENERAL CANONICAL TRANSFORMATIONS

In Sagemath we work with matrices and vectors instead of up and down tuples. Thus, if we want to convert the code examples of the earlier parts of Section 5.2 of the book, we have to build functions to convert tuples to vectors and then convert back. But the later parts of Section 5.2 do the same work, but directly in terms of symplectic matrices. I therefore move on to the symplectic matrix version directly.

5.2.1 *The standard imports.*

section5.2.sage

```
load(
    "utils.sage",
    "utils1.6.sage",
    "utils3.1.sage",
    "utils5.1.sage",
    "utils5.2.sage",
)

var("t x y", domain="real")
```

don't tangle

```
load("show_expression.sage")
```

5.2.2 *Symplectic matrices*

The main task is how to support the computation of DC_H where C_H is a phase-space transformation, that is, a map from $up(t,q',p')$ to $up(t,q,p)$. For this, we can use the jacobian of Sagemath, but this requires to map tuples like (t,q',p') to a vector, then apply the jacobian, and then convert back to a tuple. So, we start with making the functions that carry out these conversions. Then we concentrate on jacobian. When that is done, we can consider symplectic matrices.

Here are the mappings between vectors and up tuples.

utils5.2.sage

```
def vector_to_up(vec):
    n = len(vec)
    dim = n // 2
    return up(vec[0], vec[1 : dim + 1], vec[dim + 1 : n + 1])


def up_to_vector(tup):
    return vector([time(tup), *coordinate(tup), *velocity(tup)])
```

Here is a test.

section5.2.sage

```
var("r, phi, p_r, p_phi", domain="real")
assume(r > 0)

r_phi = up(t, vector([r, phi]), vector([p_r, p_phi]))
```

The functions should be each others inverse.

section5.2.sage

```
vec = up_to_vector(r_phi)
show(vec)
```

```
show(up_to_vector(vector_to_up(vec)))
show(vector_to_up(up_to_vector(r_phi)))
```

$$(t, r, \phi, p_r, p_\phi)$$
$$(t, r, \phi, p_r, p_\phi)$$

$$t$$
$$(r, \phi)$$
$$(p_r, p_\phi)$$

Point transformations can be used as canonical transformations. So this case we should get working anyway.

section5.2.sage
```
show(up_to_vector(F_to_CH(p_to_r)(r_phi)).simplify_full())
```

$$\left(t, r\cos(\phi), r\sin(\phi), \frac{p_r r\cos(\phi) - p_\phi \sin(\phi)}{r}, \frac{p_r r\sin(\phi) + p_\phi \cos(\phi)}{r}\right)$$

First I try to differentiate wrt t (to build up some confidence).

section5.2.sage
```
CH = lambda t: up_to_vector(F_to_CH(p_to_r)(r_phi))

res = diff(CH(t), t)
show(res)
```

$$(1, 0, 0, 0, 0)$$

This works.

Now, recall that the jacobian needs a vector as input, but our canonical transformations expect up-tuples. The next function realizes the necessary lifting.

utils5.2.sage
```
def to_vector_map(C):
    "Lift the structure map C such that it maps a vector to a vector."
    return Compose(up_to_vector, C, vector_to_up)
```

It's time to test all this. The result looks great.

section5.2.sage
```
DC_H = jacobian(to_vector_map(F_to_CH(p_to_r))(vec), vec).simplify_full()
show(DC_H)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -r\sin(\phi) & 0 & 0 \\ 0 & \sin(\phi) & r\cos(\phi) & 0 & 0 \\ 0 & \frac{p_\phi \sin(\phi)}{r^2} & -\frac{p_r r \sin(\phi)+p_\phi \cos(\phi)}{r} & \cos(\phi) & -\frac{\sin(\phi)}{r} \\ 0 & -\frac{p_\phi \cos(\phi)}{r^2} & \frac{p_r r \cos(\phi)-p_\phi \sin(\phi)}{r} & \sin(\phi) & \frac{\cos(\phi)}{r} \end{pmatrix}$$

And now we can build the function `D_as_matrix` from the book.

utils5.2.sage

```
def D_as_matrix(C):
    def f(state):
        vec = up_to_vector(state)
        vmap = to_vector_map(C)
        return jacobian(vmap(vec), vec).simplify_full()

    return f
```

We continue with the tests on the symplectic matrices. As our examples are low dimensional, we don't need sparse matrices.

utils5.2.sage

```
def symplectic_unit(n):
    I = identity_matrix(n)
    return block_matrix([[zero_matrix(n), I], [-I, zero_matrix(n)]])
```

section5.2.sage

```
show(symplectic_unit(2))
```

$$\left(\begin{array}{cc|cc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{array}\right)$$

The test on whether a matrix is symplectic or not. Sometimes Sagemath seems to miss that $\sqrt{x}/\sqrt{x} = 1$, even after adding the assumption that x is positive. It turned out that expanding the matrix M resolved this type of problem. I like to see the output in case the transformation is not symplectic.

utils5.2.sage

```
def is_symplectic_matrix(M):
    n = M.nrows()
    J = symplectic_unit(n // 2)
    M = M.expand()
    res = (M * J * M.transpose()).simplify_full()
    if res == J:
        return True
    print(res - J)
    return False
```

Since $J^{-1} = J^T$, we can fill in $J = A$ in the relation $J = AJA^T = JJJ^T = JJJ^{-1} = J$, to conclude that J is a symplectic matrix itself. This is our first test of `is_symplectic`.

section5.2.sage

```
J = symplectic_unit(2)
show(is_symplectic_matrix(J))
```

Here are the rest of the functions of the section.

utils5.2.sage

```
def qp_submatrix(M):
    return M[1:, 1:]

def is_symplectic_transform(C):
    return lambda state: Compose(
        is_symplectic_matrix, qp_submatrix, D_as_matrix(C)
    )(state)
```

We can test whether the transformation to polar coordinates is canonical.

section5.2.sage

```
show(is_symplectic_transform(F_to_CH(p_to_r))(r_phi))
```

True

This is a test for a general 2D point transformation. It took me a bit of time to see how to translate the next Scheme code. (BTW, I suspect that the third closing bracket at the end of line 3 is incorrect.)

```
(define (F s)
  ((literal-function 'F
    (-> (X Real (UP Real Real)) (UP Real Real))
    (time s)
    (coordinates s)))
```

In Sagemath this becomes

```
def F(local):
    t, q = time(local), coordinate(local)
    return vector([function("f")(t, *q), function("g")(t, *q)])
```

The next check takes some time to complete.

section5.2.sage

```
_ = var("q_x q_y v_x v_y p_x p_y", domain="real")
xy = up(t, vector([x, y]), vector([p_x, p_y]))

```

```
1362
1363 def F(local):
1364     t = time(local)
1365     q = coordinate(local)
1366     return vector([function("f")(t, *q), function("g")(t, *q)])
1367
1368
1369 show(is_symplectic_transform(F_to_CH(F))(xy))
```

True

We can do some tests on earlier examples. Formally we know already the answer, but why use them to test our code (and our understanding)?

This one tests the polar-canonical transformation. BTW, this example reported false when the matrix `M` in `is_symplectic_matrix` was not expanded.

```
1370 def polar_canonical(alpha):
1371     def f(state):
1372         t = time(state)
1373         theta = coordinate(state)[0]
1374         I = momentum(state)[0]
1375         x = sqrt(2 * I / alpha) * sin(theta)
1376         p = sqrt(2 * I * alpha) * cos(theta)
1377         return up(t, vector([x]), vector([p]))
1378
1379     return f
1380
1381
1382 _ = var("theta", domain="real")
1383 _ = var("I alpha", domain="positive")
1384 theta_I = up(t, vector([theta]), vector([I]))
1385 show(is_symplectic_transform(polar_canonical(alpha))(theta_I))
```

True

This is a non canonical transformation.

```
1386 def a_non_canonical_transform(state):
1387     t = time(state)
1388     theta = coordinate(state)[0]
1389     I = momentum(state)[0]
1390     x = I * sin(theta)
1391     p = I * cos(theta)
1392     return up(t, vector([x]), vector([p]))
1393
1394
1395 show(is_symplectic_transform(a_non_canonical_transform)(theta_I))
```

```
[ 0 I - 1] [-I + 1 0]
```

False