

Structure and Interpretation of Classical Mechanics with Python and Sagemath

Nicky van Foreest

May 31, 2025

CONTENTS

0 PRELIMINARIES	2
0.1 Readme	2
0.2 Output to L ^A T _E X	3
0.3 The Tuple class	4
0.4 Functional programming with Python functions	7
0.5 Differentiation	14
1 CHAPTER 1	21
1.4 Computing Actions	21
1.5 The Euler-Lagrange Equations	30
1.6 How to find Lagrangians	38
1.7 Evolution of Dynamical State	44
1.8 Conserved Quantities	51
1.9 Abstraction of Path Functions	57
2 CHAPTER 2: SKIPPED	63
3 CHAPTER 3	64
3.1 Hamilton's Equations	64
3.2 Poisson Brackets	69
3.4 Phase Space Reduction	73
3.5 Phase Space Evolution	74
3.9 The standard map	76
4 CHAPTER 4: SKIPPED	79
5 CHAPTER 5	80
5.1 Point Transformations	80
5.2 General Canonical Transformations	82
6 CHAPTER 6	88
6.4 Lie Series	88
7 CHAPTER 7	93

PRELIMINARIES

0.1 README

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 Python code to understand the Scheme code of the book (and vice versa). I therefore 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.

- The file `sicm_sagemath.pdf` shows all code samples together with the output when running the code.
- The directory `org` contains the org files.
- The directory `sage` contains all sage files obtained from tangling the org files.

In the pdf file 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 everything worked, I copied the code to an Org file and make code blocks. Then I tangled, for instance, generally useful code of `section1.4.org` to `utils1.4.sage` and code specific for Section 1.4 to `section1.4.sage`. A util file does not contain code that will be executed when loading the util file. This way I can load the utils files at later stages, and by running, for instance `sage section1.4.sage` I can test all functionality offered by the ‘module’ `utils1.4.sage`.

I found it convenient to test things in a `tests.sage` file. I can edit `tests.sage` within Emacs and see the consequences directly in the sage session by opening a sage session on the command prompt and attaching the session to the file like so:

```
sage: attach("tests.sage")
```

Finally, here are some resources that were helpful to me:

- 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>
- A port to Clojure: <https://github.com/sicmutils/sicmutils>
- The Sagemath reference guide: <https://doc.sagemath.org/html/en/reference/>
- Handy tuples: <https://github.com/jtauber/functional-differential-geometry>
- ChatGPT proved to be a great help in the process of becoming familiar with Scheme and Sagemath.
- Some solutions to problems: <https://github.com/hnarayanan/sicm>

In the next sections we provide Python and Sagemath code for background functions that are used, but not defined, in the book.

0.2 OUTPUT TO LATEX

We need some tricks to adapt the LATEX output of Sagemath to our liking.

We use `re` to modify LateX strings. I discovered the two latex options from this site: [Sage, LateX and Friends](#).

```
1 import re
2
3 latex.matrix_delimiters(left='[', right=']')
4 latex.matrix_column_alignment("c")
```

Note in passing that the title of the code block shows the file to which the code is tangled, and if a code block is not tangled, the title says "don't tangle".

To keep the formulas short in LATEX, I remove all strings like (t) , and replace $\partial x/\partial t$ by \dot{x} . This is the job of the regular expressions below.

```
5 def simplify_latex(s):
6     s = re.sub(r"\frac{\partial}{\partial t}", r"\dot{", s)
7     s = re.sub(r"\left(t\right)", r"", s)
8     s = re.sub(
9         r"\frac{\partial^2}{\partial t^2}\{", r"\partial_t\}^2",
10        r"\ddot{",
11        s,
12    )
13    return s
```

The function `show_expression` prints expressions to LATEX. There is a caveat, though. When `show_expression` would return a string, org mode (or perhaps Python) adds

many escape symbols for the \ character, which turns out to ruin the L^AT_EX output in an org file. For this reason, I just call `print`; for my purposes (writing these files in emacs and org mode) it works the way I want.

```
14  def show_expression(s, simplify=True):
15      s = latex(s)
16      if simplify:
17          s = simplify_latex(s)
18      res = r"\begin{dmath*}"
19      res += "\n" + s + "\n"
20      res += r"\end{dmath*}"
21      print(res)
```

0.2.1 Printing with org mode

There is a subtlety with respect to printing in org mode and in tangled files. When working in sage files, and running them from the prompt, I call `show(expr)` to have some expression printed to the screen. So, when running Sage from the prompt, I do *not* want to see L^AT_EX output. However, when executing a code block in org mode, I *do* want to get L^AT_EX output. For this, I could use the book's `show_expression` 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, 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 include the following `show` function in org mode, but I don't *tangle* it to the related sage files.

```
22  def show(s, simplify=True):
23      return show_expression(s, simplify)
```

0.3 THE TUPLE CLASS

The book uses up tuples quite a bit. This code is a copy of `tuples.py` from <https://github.com/jtauber/functional-differential-geometry>. See `tuples.rst` in that repo for further explanations.

```
24  """
25  This is a copy of tuples.py from
26  https://github.com/jtauber/functional-differential-geometry.
27  """
28
29  from sage.structure.element import Matrix, Vector
30
31
```

```
32 class Tuple:
33     def __init__(self, *components):
34         self._components = components
35
36     def __getitem__(self, index):
37         return self._components[index]
38
39     def __len__(self):
40         return len(self._components)
41
42     def __eq__(self, other):
43         if (
44             isinstance(other, self.__class__)
45             and self._components == other._components
46         ):
47             return True
48         else:
49             return False
50
51     def __ne__(self, other):
52         return not (self.__eq__(other))
53
54     def __add__(self, other):
55         if isinstance(self, Tuple):
56             if not isinstance(other, self.__class__) or len(self) != len(
57                 other
58             ):
59                 raise TypeError("can't add incompatible Tuples")
60             else:
61                 return self.__class__(
62                     *(
63                         s + o
64                         for (s, o) in zip(self._components, other._components)
65                     )
66                 )
67             else:
68                 return self + other
69
70     def __iadd__(self, other):
71         return self + other
72
73     def __neg__(self):
74         return self.__class__(*(-s for s in self._components))
75
76     def __sub__(self, other):
77         return self + (-other)
78
79     def __isub__(self, other):
80         return self - other
81
82     def __call__(self, *args, **kwargs):
83         return self.__class__(
```

```

84         *(  
85             (c(*args, **kwargs) if isinstance(c, Expr) else c)  
86             for c in self._components  
87         )  
88     )  
89  
90     def subs(self, args):  
91         # substitute variables with args  
92         return self.__class__(*(c.subs(args) for c in self._components))  
93  
94     def list(self):  
95         "convert tuple and its components to one list."  
96         result = []  
97         for comp in self._components:  
98             if isinstance(comp, (Tuple, Matrix, Vector)):  
99                 result.extend(comp.list())  
100            else:  
101                result.append(comp)  
102        return result  
103  
104    def derivative(self, var):  
105        "Compute the derivative of all components and put the result in a tuple."  
106        return self.__class__()  
107        *[derivative(comp, var) for comp in self._components]  
108    )

```

We have up tuples and down tuples. They differ in the way they are printed.

```

----- ./sage/tuples.sage -----  
109 class UpTuple(Tuple):  
110     def __repr__(self):  
111         return "up({})".format(", ".join(str(c) for c in self._components))  
112  
113     def _latex_(self):  
114         "Print up tuples vertically."  
115         res = r"\begin{array}{c}"  
116         for comp in self._components:  
117             res += r"\begin{array}{c}"  
118             res += latex(comp)  
119             res += r"\end{array}"  
120             res += r" \\"  
121             res += r"\end{array}"  
122         return res  
123  
124 class DownTuple(Tuple):  
125     def __repr__(self):  
126         return "down({})".format(", ".join(str(c) for c in self._components))  
127  
128     def _latex_(self):  
129         "Print down tuples horizontally."  
130         res = r"\begin{array}{c}"  
131         for comp in self._components:

```

```

132         res += r"\begin{array}{c}"
133         res += latex(comp)
134         res += r"\end{array}"
135         res += r" & "
136         res += r"\end{array}"
137     return res
138
139 up = UpTuple
140 down = DownTuple
141
142 up._dual = down
143 down._dual = up

```

Here is some functionality to unpack tuples. I don't use it for the moment, but it is provided by the `tuples.py` package that I downloaded from the said github repo.

```

144 def ref(tup, *indices):
145     if indices:
146         return ref(tup[indices[0]], *indices[1:])
147     else:
148         return tup
149
150
151 def component(*indices):
152     def _(tup):
153         return ref(tup, *indices)
154
155     return _

```

0.4 FUNCTIONAL PROGRAMMING WITH PYTHON FUNCTIONS

In this section we set up some generic functionality to support the summation, product, and composition of functions:

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

This is easy to code with recursion.

0.4.1 Standard imports

```

156 load("tuples.sage")

```

We need to load `functions.sage` to run the examples in the test file.

```
157     load("functions.sage")
```

We load `show_expression` to control the L^AT_EX output in this org file.

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

0.4.2 *The Function class*

The `Function` class provides the functionality we need for functional programming.

```
159  class Function:
160      def __init__(self, func):
161          self._func = func
162
163      def __call__(self, *args):
164          return self._func(*args)
165
166      def __add__(self, other):
167          return Function(lambda *args: self(*args) + other(*args))
168
169      def __neg__(self):
170          return Function(lambda *args: -self(*args))
171
172      def __sub__(self, other):
173          return self + (-other)
174
175      def __mul__(self, other):
176          if isinstance(other, Function):
177              return Function(lambda *args: self(*args) * other(*args))
178          return Function(lambda *args: other * self(*args))
179
180      def __rmul__(self, other):
181          return self * other
182
183      def __pow__(self, exponent):
184          if exponent == 0:
185              return Function(lambda x: 1)
186          else:
187              return self * (self ** (exponent - 1))
```

The next function decorates a function `f` that returns another function `inner_f`, so that `inner_f` becomes a `Function`.

```
188  def Func(f):
189      def wrapper(*args, **kwargs):
190          return Function(f(*args, **kwargs))
```

```
191
192     return wrapper
```

Below I include an example to see how to use, and understand, this decorator. Composition is just a recursive call of functions.

```
193     @Func
194     def compose(*funcs):
195         if len(funcs) == 1:
196             return lambda x: funcs[0](x)
197         return lambda x: funcs[0](compose(*funcs[1:])(x))
```

0.4.3 Some standard functions

To use python functions as Functions, use `lambda` like this.

```
198     def f(x):
199         return 5 * x
200
201
202     F = Function(lambda x: f(x))
```

The identity is just interesting. Perhaps we'll use it later.

```
203     identity = Function(lambda x: x)
```

To be able to code things like $(\sin + \cos)(x)$ we need to postpone the application of `sin` and `cos` to their arguments. Therefore we override their definitions.

```
204     sin = Function(lambda x: sage.functions.trig.sin(x))
205     cos = Function(lambda x: sage.functions.trig.cos(x))
```

We will use quadratic functions often.

```
206     from functools import singledispatch
207
208
209     @singledispatch
210     def _square(x):
211         raise TypeError(f"Unsupported type: {type(x)}")
212
213     @_square.register(int)
```

```

215 @_square.register(float)
216 @_square.register(Expression)
217 @_square.register(Integer)
218 def _(x):
219     return x ^ 2
220
221
222 @_square.register(Vector)
223 @_square.register(list)
224 @_square.register(tuple)
225 def _(x):
226     v = vector(x)
227     return v.dot_product(v)
228
229
230 @_square.register(Matrix)
231 def _(x):
232     if x.ncols() == 1:
233         return (x.T * x)[0, 0]
234     elif x nrows() == 1:
235         return (x * x.T)[0, 0]
236     else:
237         raise TypeError(
238             f"Matrix must be a row or column vector, got shape {x nrows()}x{x ncols()}"
239         )
240
241
242 square = Function(lambda x: _square(x))

```

To use Sagemath functions we make an abbreviation.

```

..... ./sage/functions.sage
243 function = sage.symbolic.function_factory.function
.....
```

Now we can make symbolic functions like so.

```

..... ./sage/functions_tests.sage
244 V = Function(lambda x: function("V")(x))
.....
```

0.4.4 Examples

```

..... ./sage/functions_tests.sage
245 x, y = var("x y", domain = RR)
246 show((square)(x + y).expand())
.....
```

$$x^2 + 2xy + y^2$$

```

..... ./sage/functions_tests.sage
248 show((square + square)(x + y))
.....
```

$$2(x + y)^2$$

```
249      ..../sage/functions_tests.sage
show((square * square)(x))
```

$$x^4$$

```
250      ..../sage/functions_tests.sage
show((sin + cos)(x))
```

$$\cos(x) + \sin(x)$$

```
251      ..../sage/functions_tests.sage
show((square + V)(x))
```

$$x^2 + V(x)$$

```
252      ..../sage/functions_tests.sage
hh = compose(square, sin)
show((hh + hh)(x))
```

$$2 \sin(x)^2$$

We know that $2\sin x \cos x = \sin(2x)$.

```
254      ..../sage/functions_tests.sage
show((2 * (sin * cos)(x) - sin(2 * x)).simplify_full())
```

$$0$$

Next, we test differentiation and integration.

```
255      ..../sage/functions_tests.sage
show(diff(-compose(square, cos)(x), x))
256      show(integrate((2 * sin * cos)(x), x))
```

$$2 \cos(x) \sin(x)$$

$$-\cos(x)^2$$

Arithmetic with symbolic functions works too.

```
257      ..../sage/functions_tests.sage
U = Function(lambda x: function("U")(x))
258      V = Function(lambda x: function("V")(x))
```

```

259 show((U + V)(x))
260 show((V + V)(x))
261 show((V(U(x))))))
262 show((compose(V, U)(x)))

```

$$U(x) + V(x)$$

$$2V(x)$$

$$V(U(x))$$

$$V(U(x))$$

```

263 def f(x):
264     def g(y):
265         return x * y ^ 2
266
267     return g

```

```

268 show(f(3)(5))

```

75

However, we cannot apply algebraic operations on `f`. For instance, this does not work; it gives `TypeError: unsupported operand type(s) for +: 'function' and 'function'`.

```

269 show((f(3) + f(2))(4))

```

By decoration with `@Func` we get what we need.

```

270 @Func
271 def f(x):
272     def g(y):
273         return x * y ^ 2
274
275     return g

```

```

276 show((f(3) + f(2))(4))

```

80

Indeed: $(3 + 2) * 4^2 = 80$.

Decorating with `@Func` is the same as this.

```

277 def f(x):
278     def g(y):
279         return x * y ^ 2
280
281     return Function(lambda y: g(y))

```

```

282 show((f(3) + f(2))(4))

```

80

An example from the Appendix of the book.

```

283 cube = Function(lambda x: x * square(x))

```

```

284 h = compose(cube, sin)
285 a = var('a', domain=RR)
286 show(h(a))
287 show(h(float(2)))

```

0.7518269446689928

$$\sin(a)^3$$

```

288 g = cube * sin
289 show(g(2).n())

```

7.27437941460545

0.5 DIFFERENTIATION

0.5.1 Standard imports

```

291 _____ .. / sage / differentiation . sage
292 load(
293     "functions . sage",
294     "tuples . sage",
295 )
296 _____ .. / sage / differentiation _ tests . sage
297 load( "differentiation . sage" )
298 var( "t" , domain = "real" )
299 _____ don ' t tangle
300 load( "show _ expression . sage" )

```

0.5.2 Examples with matrices, functions and tuples

```

299 _____ .. / sage / differentiation _ tests . sage
300 _ = var( "a b c x y" , domain = RR )
301 M = matrix( [[a, b], [b, c]])
302 b = vector([a, b])
303 v = vector([x, y])
304 F = 1 / 2 * v * M * v + b * v + c
305 _____ .. / sage / differentiation _ tests . sage
306 show(F)

```

$$\frac{1}{2}(ax + by)x + ax + \frac{1}{2}(bx + cy)y + by + c$$

```

305 _____ .. / sage / differentiation _ tests . sage
306 show(F . expand())

```

$$\frac{1}{2}ax^2 + bxy + \frac{1}{2}cy^2 + ax + by + c$$

```

306 _____ .. / sage / differentiation _ tests . sage
307 show( diff(F , x) )

```

$$ax + by + a$$

Repeated differentiation works nicely.

```
307 ..../sage/differentiation_tests.sage
show(diff(F, [x, y]))
```

$$b$$

This is the Jacobian.

```
308 ..../sage/differentiation_tests.sage
show(jacobian(F, [x, y]))
```

$$\begin{bmatrix} ax + by + a & bx + cy + b \end{bmatrix}$$

```
309 ..../sage/differentiation_tests.sage
show(jacobian(F, v.list())) # convert the column matrix to a list
```

$$\begin{bmatrix} ax + by + a & bx + cy + b \end{bmatrix}$$

This expression gives an error.

```
310 ..../sage/differentiation_tests.sage
diff(F, v) # v is not a list, but a vector
```

To differentiate a Python function we need to provide the arguments to the function.

```
311 def F(v):
312     return 1 / 2 * v * M * v + b * v + c
```

```
313 ..../sage/differentiation_tests.sage
314 show(diff(F(v), x)) # add the arguments to F
show(jacobian(F(v), v.list()))
```

$$ax + by + a$$

$$\begin{bmatrix} ax + by + a & bx + cy + b \end{bmatrix}$$

The next two examples do not work.

```
315 ..../sage/differentiation_tests.sage
316 jacobian(F, v) # F has no arguments
jacobian(F(v), v) # v is not a list
```

The Tuple class supports differentiation.

```
317 T = up(t, t ^ 2, t ^ 3, sin(3 * t))
318 show(diff(T, t))
```

$$\begin{aligned} 1 \\ 2t \\ 3t^2 \\ 3 \cos(3t) \end{aligned}$$

0.5.3 Differentiation with respect to time

The function `D` takes a function (of time) as argument, and returns the derivative with respect to time:

$$D(f(\cdot)): t \rightarrow f'(t).$$

```
319 @Func
320 def D(f):
321     return lambda t: diff(f(t), t)
322     #return derivative(expr, t)
```

Here is an example.

```
323 q = Function(lambda t: function("q")(t))
324
325 show(D(q)(t))
```

$$\dot{q}$$

0.5.4 Differentiation with respect to function arguments

The Euler-Lagrange equations depend on the partial derivative of a Lagrangian L with respect to q and v , and a total derivative with respect to time. Now q and v will often be functions of time, so we need to find a way to differentiate with respect to *functions*, like $q(\cdot)$, rather than just symbols, like x . To implement this in Sagemath turned out to be far from easy, at least for me.

First, observe that the Jacobian in Sagemath takes as arguments a function and the variables with respect to which to take the derivatives. So, I tried this first:

```
326 q = Function(lambda t: function("q")(t))
```

But the next code gives errors saying that the argument q should be a symbolic function, which it is not.

```
327 F = 5 * q + 3 * t
328
329 show(diff(F, r)) # does not work
330 show(jacobian(F, [q, t])) # does not work
```

To get around this problem, I use the following strategy to differentiate a function F with respect to functions.

1. Make a list of dummy symbols, one for *each argument* of F that is *not a symbol*. To understand this in detail, observe that arguments like t or x are symbols, but such symbols need not be protected. In other words: we don't have to replace a symbol by another symbol, because Sagemath can already differentiate wrt symbols; it's the other 'things' are the things that have to be replaced by a variable. Thus, arguments like $q(t)$ that are *not* symbols have to be protected by replacing them with dummy symbols.
2. Replace in F the arguments by their dummy variables. We use the Sagemath `subs` functionality of Sagemath to substitute the dummy variables for the functions.
3. Now there is one further problem: `subs` does not work on lists or tuples. However, `subs` *does work* on matrices. Therefore, we cast all relevant lists to matrices. (We could have used vectors. However, when `args` is a vector, `results` will be a vector too, but we need a matrix to distinguish between standing and lying vectors.)
4. Take the Jacobian of F with respect to the dummy symbols. We achieve this by substituting the dummy symbols in the vector of arguments and the vector of variables.
5. Invert: Replace in the final result the dummy symbols by their related arguments.

We use `id(v)` to create a unique variable name for each dummy variable and store the mapping from the functions to the dummy variables in a dictionary `subs`. (As these are internal names, the actual variable names are irrelevant; as long as they are unique, it's OK.)

We know from the above that `jacobian` expects a *list* with the variables with respect to which to differentiate. Therefore, we turn the vector with substituted variables to a list.

.../sage/differentiation.sage

```

331 def Jacobian(F):
332     def wrap_Jacobian(args, vrs):
333         if isinstance(args, (list, tuple)):
334             args = matrix(args)
335         if isinstance(vrs, (list, tuple)):
336             vrs = matrix(vrs)
337         subs = {
338             v: var(f"v{id(v)}", domain=RR)
339             for v in args.list()
340             if not v.is_symbol()
341         }
342         result = jacobian(F(args.subs(subs)), vrs.subs(subs).list())
343         inverse_subs = {v: k for k, v in subs.items()}
344         return result.subs(inverse_subs)
345
346     return wrap_Jacobian

```

Here are some examples to see how to use this Jacobian. Note that Jacobian expects the arguments and variables to be *lists*, or list like. As a result, in the function F we have to unpack the list.

```
347 v = var("v", domain=RR)
348
349
350 def F(v):
351     r, t = v.list()
352     return 5 * r^3 + 3 * t^2 * r
353
354
355 show(Jacobian(F)([v, t], [t]))
356 show(Jacobian(F)([v, t], [v, t]))
```

$$[6tv]$$

$$[3t^2 + 15v^2 \quad 6tv]$$

This works. Now we try the same with a function-like argument. Recall, v must a be list for partial.

```
357 q = Function(lambda t: function("q")(t))
358 v = [q(t), t]
359 show(Jacobian(F)(v, v))
```

$$[3t^2 + 15q^2 \quad 6tq]$$

0.5.5 Gradient and Hessian

Next we build the gradient. 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. Moreover, we like to be able to take the gradient with respect to literal functions. Thus, we use the Jacobian defined above.

One idea for the gradient is like this. However, this does not allow to use gradient as a function in functional composition.

```
360 def gradient(F, v):
361     return Jacobian(F)(v, v).T
```

We therefore favor the next implementation. BTW, note that the gradient is a vector in a tangent space, hence it is column vector. For that reason we transpose the Jacobian.

```
362 _____ ./sage/differentiation.sage _____
363 def gradient(F):
364     return lambda v: Jacobian(F)(v, v).T
```

```
364 _____ ./sage/differentiation_tests.sage _____
365 show(gradient(F)(v))
```

$$\begin{bmatrix} 3t^2 + 15q^2 \\ 6tq \end{bmatrix}$$

The function may return a list too.

```
365 q = Function(lambda t: function("q")(t))
366 v = [q(t), t]
367
368 def G(v):
369     r, t = v.list()
370     return [5 * r ^ 3 + 3 * t ^ 2 * r, r^2 + t]
371
372
373 show(gradient(G)(v))
```

$$\begin{bmatrix} 3t^2 + 15q^2 & 2q \\ 6tq & 1 \end{bmatrix}$$

When differentiating a symbolic function, wrap such a function in a `Function`.

```
374 _____ ./sage/differentiation_tests.sage _____
375 U = Function(lambda x: function("U")(square(x)))
376 show(gradient(U)(v))
```

$$\begin{bmatrix} 2qD_0(U)(t^2 + q^2) \\ 2tD_0(U)(t^2 + q^2) \end{bmatrix}$$

The Hessian can now be defined as the composition of the gradient with itself.

```
376 _____ ./sage/differentiation.sage _____
377 def Hessian(F):
378     return lambda v: compose(gradient, gradient)(F)(v)
```

```
378 _____ ./sage/differentiation_tests.sage _____
379 show(Hessian(F)(v))
```

$$\begin{bmatrix} 30q & 6t \\ 6t & 6q \end{bmatrix}$$

0.5.6 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 Jacobian function built above allows us a very simple solution. Note that we return a Function so that we can use this operator in functional composition if we like.

```

379 _____ ./.sage/differentiation.sage _____
380 @Func
381 def partial(f, slot):
382     def wrapper(local):
383         if slot == 0:
384             selection = [time(local)]
385         elif slot == 1:
386             selection = coordinate(local)
387         elif slot == 2:
388             selection = velocity(local)
389         return Jacobian(f)(local, selection)
390
391     return wrapper

```

The main text contains many examples.

CHAPTER 1

1.4 COMPUTING ACTIONS

1.4.1 Standard setup

I create an Org file for each separate section of the book; for this section it's `section1.4.org`. Code that is useful for later sections is tangled to `utils1.4.sage` and otherwise to `section1.4.sage`. This allows me to run the sage scripts on the prompt. Note that the titles of the code blocks correspond to the file to which the code is written when tangled.

```
391 import numpy as np
392
393 load("functions.sage", "differentiation.sage", "tuples.sage")
```

BTW, don't do `from sage.all import *` because that will lead to name space conflicts, for instance with the Gamma function which we define below.

```
394 load("utils1.4.sage")
395
396 t = var("t", domain="real")
```

The next module is used for nice printing in org mode files; it should only be loaded in org mode files.

```
397 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 (curried) function `Lagrangian` that takes a `local` tuple as an argument.

```
398 def L_free_particle(mass):
399     def Lagrangian(local):
400         v = velocity(local)
401         return 1 / 2 * mass * square(v)
402
403     return Lagrangian
```

For the next step, we need a *literal functions* and *coordinate paths*.

1.4.3 Literal functions

A `literal_function` maps the time t to a coordinate or velocity component of the path, for instance, $t \rightarrow x(t)$. Since we need to perform arithmetic with literal functions, see below for some examples, we encapsulate it in a `Function`.

```
404     ..../sage/utils1.4.sage
405 @Func
406 def literal_function(name):
407     return lambda t: function(name)(t)


---


```

It's a function.

```
407 x = literal_function("x")          don't tangle
408 print(x)


---


```

`<__main__.Function object at 0x71122066e470>`

Here are some operations on x .

```
409 show(x(t))                      don't tangle
410 show((x+x)(t))
411 show(square(x)(t))


---


```

Note that, to keep the notation brief, the t is suppressed in the L^AT_EX output.

1.4.4 Paths

We will represent coordinate path functions q and velocity path functions v as functions that map time to vectors. Thus, `column_path` returns a function of time, not yet a path. We also need to perform arithmetic on paths, like $3q$, therefore we encapsulate the path in a `Function`.

```
412     ..../sage/utils1.4.sage
413 @Func
414 def column_path(lst):
415     return lambda t: column_matrix([l(t) for l in lst])


---


```

```
415 q = column_path()                don't tangle
416 [
417     literal_function("x"),
418     literal_function("y"),
419 ]
420 )


---


```

Here is an example to see how to use q .

don't tangle

`421 show(q(t))`

$\begin{bmatrix} x \\ y \end{bmatrix}$

don't tangle

`422 show((q + q)(t))`

$\begin{bmatrix} 2x \\ 2y \end{bmatrix}$

don't tangle

`423 show((2 * q)(t))`

$\begin{bmatrix} 2x \\ 2y \end{bmatrix}$

don't tangle

`424 show((q * q)(t))`

1.4.5 Gamma function

The Gamma function lifts a coordinate path to a function that maps time to a local tuple of the form $(t, q(t), v(t), \dots)$. That is,

$$\begin{aligned}\Gamma[q](\cdot) &= (\cdot, q(\cdot), v(\cdot), \dots), \\ \Gamma[q](t) &= (t, q(t), v(t), \dots).\end{aligned}$$

To follow the conventions of the book, we use an up tuple for Gamma. However, I don't build the coordinate path nor the velocity as up tuples because I find SageMath vectors more convenient.

Γ just receives q as an argument. Then it computes the velocity $v = Dq$, from which the acceleration follows recursively as $a = Dv$, Recall that D computes the derivative (wrt time) of a function that depends on time.

When $n = 3$, it returns a function of time that produces the first three elements of the local tuple $(t, q(t), v(t))$. This is the default. Once all derivatives are computed, we convert the result to a function that maps time to an up tuple.

.. /sage/utils1.4.sage

`425 def Gamma(q, n=3):`

`426 if n < 2:`

`427 raise ValueError("n must be > 1")`

`428 Dq = [q]`

`429 for k in range(2, n):`

`430 Dq.append(D(Dq[-1]))`

`431 return lambda t: up(t, *[v(t) for v in Dq])`

When applying `Gamma` to a path, we get this.

```
432   local = Gamma(q)(t)          don't tangle
433   show(local)
```

$$\begin{bmatrix} t \\ x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$$

We can include the acceleration too.

```
434   show(Gamma(q, 4)(t))        don't tangle
```

$$\begin{bmatrix} t \\ x \\ y \\ \dot{x} \\ \ddot{y} \\ \dot{\ddot{x}} \\ \ddot{\dot{y}} \end{bmatrix}$$

Finally, here are some projections operators from the local tuple to superspaces.

```
435   time = Function(lambda local: local[0])
436   coordinate = Function(lambda local: local[1])
437   velocity = Function(lambda local: local[2])
```

```
438   show(compose(velocity, Gamma(q))(t))      don't tangle
```

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}$$

1.4.6 Continuation with the free particle.

Now we know how to build literal functions and Γ , we can continue with the Lagrangian of the free particle.

```

439 q = column_path(           .../sage/section1.4.sage
440   [
441     literal_function("x"),
442     literal_function("y"),
443     literal_function("z"),
444   ]
445 )

```

```

446 show(q(t))           .../sage/section1.4.sage

```

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

```

447 show(D(q)(t))           .../sage/section1.4.sage

```

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}$$

```

448 show(Gamma(q)(t))           .../sage/section1.4.sage

```

$$\begin{bmatrix} t \\ x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}$$

The Lagrangian of a free particle with mass m applied to the path `Gamma` gives this. Our first implementation is like this: $L(\Gamma[q](t))$, that is, $\Gamma[q](t)$ makes a local tuple, and this is given as argument to L .

```

449 load("functions.sage")           .../sage/section1.4.sage
450 m = var('m', domain='positive')
451 show(L_free_particle(m)(Gamma(q)(t)))

```

$$\frac{1}{2} (\dot{x}^2 + \dot{y}^2 + \dot{z}^2) m$$

Here is the implementation of the book: $(L \circ \Gamma[q])(t)$, that is, $L \circ \Gamma[q]$ is a function that depends on t . Note how the brackets are placed after `Gamma(q)`.

```
452 _____ ./sage/section1.4.sage _____
show(compose(L_free_particle(m), Gamma(q))(t))
```

$$\frac{1}{2} (\dot{x}^2 + \dot{y}^2 + \dot{z}^2) m$$

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). Of course, Sagemath already supports a definite integral in a library.

```
453 _____ ./sage/utils1.4.sage _____
from sage.symbolic.integration import definite_integral
```

I don't like to read dt at the end of the integral because dt reads like the product of the variables d and t . Instead, I prefer to read dt ; for this reason I overwrite the L^AT_EX formatting of `definite_integral`.

```
454 _____ ./sage/utils1.4.sage _____
def integral_latex_format(*args):
    expr, var, a, b = args
    return (
        fr"\int_{{{{a}}}}^{{{{b}}}} "
        + latex(expr)
        + r"\, \text{d}\,"
        + latex(var))
461 )
462
463
464 definite_integral._print_latex_ = integral_latex_format
```

Here is the action along a generic path q .

```
465 _____ ./sage/section1.4.sage _____
T = var("T", domain="positive")
466
467 def Lagrangian_action(L, q, t1, t2):
468     return definite_integral(compose(L, Gamma(q))(t), t, t1, t2)
469
470 show(Lagrangian_action(L_free_particle(m), q, 0, T))
```

$$\frac{1}{2} m \left(\int_0^T \dot{x}^2 dt + \int_0^T \dot{y}^2 dt + \int_0^T \dot{z}^2 dt \right)$$

To get a numerical answer, we take the test path of the book. Below we'll do some arithmetic with `test_path`; therefore we encapsulate it in a Function.

```
471 _____ ./sage/section1.4.sage _____
test_path = Function(lambda t: vector([4 * t + 7, 3 * t + 5, 2 * t + 1]))
472 show(Lagrangian_action(L_free_particle(mass=3), test_path, 0, 10))
```

435

Let's try a harder path. We don't need this later, so the encapsulation in Function is not necessary.

```

473 hard_path = lambda t: vector([4 * t + 7, 3 * t + 5, 2 * exp(-t) + 1])
474
475 result = Lagrangian_action(L_free_particle(mass=3), hard_path, 0, 10)
476 show(result)
477 show(float(result))

```

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

377.9999999938165

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

1.4.7 Path of minimum action

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

```

478 @Func
479 def make_eta(nu, t1, t2):
480     return lambda t: (t - t1) * (t - t2) * nu(t)
481
482
483 nu = Function(lambda t: vector([sin(t), cos(t), t ^ 2]))
484
485 show((1 / 3 * make_eta(nu, 3, 4) + test_path)(t))

```

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

In the next code, I add the `n()` to force the result to a floating point number. (Without this, the result is a long expression with lots of cosines and sines.)

```

486 def varied_free_particle_action(mass, q, nu, t1, t2):
487     eta = make_eta(nu, t1, t2)
488
489     def f(eps):
490         return Lagrangian_action(L_free_particle(mass), q + eps * eta, t1, t2).n()
491
492     return f
493
494 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 .

```
495 res = find_local_minimum(
496     varied_free_particle_action(3.0, test_path, nu, 0.0, 10.0), -2.0, 1.0
497 )
498 show(res)
```

(435.0000000000000, 0.0)

We see that the optimal value for ϵ is 0, and we retrieve our earlier value of the Lagrangian action.

1.4.8 Finding minimal trajectories

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

```
499 def Lagrangian_polynomial(ts, qs):
500     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.

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

In the function `make_path` we use numpy's `linspace` instead of the linear interpolants of the book. Note that the coordinate paths above are column-vector functions, so `make_path` should return the same type.

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

Here is the harmonic Lagrangian.

```
513     def L_harmonic(m, k):
514         def Lagrangian(local):
515             q = coordinate(local)
516             v = velocity(local)
517             return (1 / 2) * m * square(v) - (1 / 2) * k * square(q)
518
519     return Lagrangian
```

```
520
521     def parametric_path_action(Lagrangian, t0, q0, t1, q1):
522         def f(qs):
523             path = make_path(t0, q0, t1, q1, qs=qs)
524             return Lagrangian_action(Lagrangian, path, t0, t1)
525
526     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.

```
526     t0, t1 = 0, pi / 2
527     q0, q1 = cos(t0), cos(t1)
528     T = np.linspace(0, pi / 2, 5)
529     initial_qs = [cos(t).n() for t in T][1:-1]
530     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? (Recall that a path is a vector; to extract the value of the element that corresponds to the path, we need to write `best_path(t=t)[0]`.)

```
531     def find_path(Lagrangian, t0, q0, t1, q1, n):
532         ts = np.linspace(t0, t1, n)
533         initial_qs = np.linspace(q0, q1, n)[1:-1]
534         minimizing_qs = minimize(
535             parametric_path_action(Lagrangian, t0, q0, t1, q1),
536             initial_qs,
537         )
538         return make_path(t0, q0, t1, q1, minimizing_qs)
539
540     best_path = find_path(L_harmonic(m=1, k=1), t0=0, q0=1, t1=pi / 2, q1=0, n=5)
541     result = [
542         abs(best_path(t)[0].n() - cos(t).n()) for t in np.linspace(0, pi / 2, 10)
543     ]
544     show(max(result))
```

```
0.000172462354236957
```

Great. All works!

Finally, here is a plot of the Lagrangian as a function of $q(t)$.

```
..... ./sage/section1.4.sage
545 T = np.linspace(0, pi / 2, 20)
546 q = lambda t: vector([cos(t)])
547 lvalues = [L_harmonic(m=1, k=1)(Gamma(q)(t))(t=ti).n() for ti in T]
548 points = list(zip(ts, lvalues))
549 plot = list_plot(points, color="black", size=30)
550 plot.axes_labels(["$t$", "$L$"])
551 plot.save("../figures/Lagrangian.png", figsize=(4, 2))
```

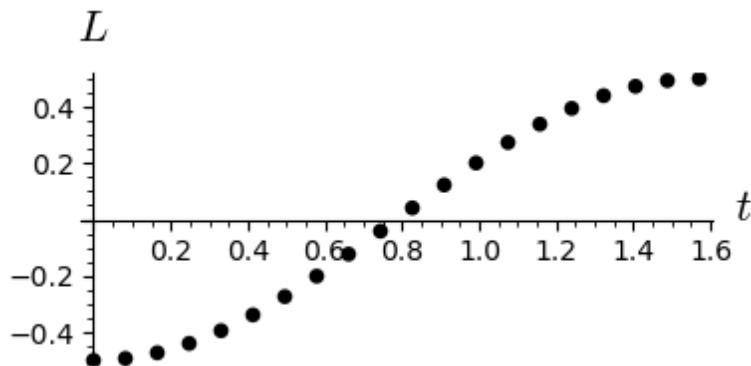


Figure 1.1: The harmonic Lagrangian as a function of the optimal path $q(t) = \cos t$, $t \in [0, \pi/2]$.

1.5 THE EULER-LAGRANGE EQUATIONS

1.5.1 Standard imports

```
..... ./sage/utils1.5.sage
552 load("utils1.4.sage")
..... ./sage/section1.5.sage
553 load("utils1.5.sage")
554 t = var("t", domain="real")
..... don't tangle
556 load("show_expression.sage")
```

1.5.2 Derivation of the Lagrange equations

Harmonic oscillator

Here is a test on the harmonic oscillator.

```
557 load("utils1.4.sage")
558 k, m = var('k m', domain="positive")
559 q = column_path([literal_function("x")])


---


560 L = L_harmonic(m, k)
561 show(L(Gamma(q)(t)))
```

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

We can apply $\partial_1 L$ and $\partial_2 L$ to a configuration path q that we lift to a local tuple by means of Γ . Realize therefore that $\text{partial}(L_{\text{harmonic}}(m, k), 1)$ maps a local tuple to a real number, and $\text{Gamma}(q)$ maps a time t to a local tuple. The next code implements $\partial_1 L(\Gamma(q)(t))$ and $\partial_2 L(\Gamma(q)(t))$. (Check how the brackets are organized.)

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


---


[ -kx ]
```

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


---


[ m\dot{x} ]
```

Here are the same results, but now with functional composition.

$$(\partial_1 L \circ \Gamma(q))(t), \quad (\partial_2 L \circ \Gamma(q))(t).$$

```
564 show(compose(partial(L, 1), Gamma(q))(t))
565 show(compose(partial(L, 2), Gamma(q))(t))


---



```

$$[-kx]$$

$$[m\dot{x}]$$

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. To understand this, note the following function mappings, where we write t for time, l for a local tuple, v a velocity-like vector, and a an acceleration-like vector:

$$\begin{aligned}\Gamma[q] &: t \rightarrow l, \\ \partial_2 L &: l \rightarrow v \\ \partial_2 L \circ \Gamma[q] &: t \rightarrow v \\ D(v) &: t \rightarrow a \\ D(\partial_2 L \circ \Gamma[q]) &: t \rightarrow a.\end{aligned}$$

In more classical notation, we compute this:

$$\frac{d}{dt} \left(\frac{\partial}{\partial q} L(\Gamma(q)) \right)(t)$$

566 `show(D(compose(partial(L, 2), Gamma(q)))(t))`

$$[m\ddot{x}]$$

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

Orbital motion

567 `q = column_path([literal_function("xi"), literal_function("eta")])`

568 `var("mu", domain="positive")`

569 `def L_orbital(m, mu):`

570 `def Lagrangian(local):`

571 `q = coordinate(local)`

572 `v = velocity(local)`

573 `return (1 / 2) * m * square(v) + mu / sqrt(square(q))`

574 `return Lagrangian`

575 `L = L_orbital(m, mu)`

576 `show(L(Gamma(q))(t))`

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

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

$$\left[-\frac{\mu \xi}{(\eta^2 + \xi^2)^{\frac{3}{2}}} \quad -\frac{\mu \eta}{(\eta^2 + \xi^2)^{\frac{3}{2}}} \right]$$

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

$$\begin{bmatrix} m\ddot{\xi} & m\dot{\eta} \end{bmatrix}$$

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

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

```
581 q = column_path([literal_function("theta")])
```

Here is the Lagrangian. Recall that the coordinates of the space form a vector. Here, `theta` is the only element of the vector, which we can extract by considering element 0. For `thetadot` we don't have to do this since we consider $\dot{\theta}^2$, and the `square` function accepts vectors as input and returns a real. However, for reasons of consistency, we choose to do this nonetheless.

```
582 var("m g l", domain="positive")
583
584
585 def L_planar_pendulum(m, g, l):
586     def Lagrangian(local):
587         theta = coordinate(local).list()[0]
588         theta_dot = velocity(local).list()[0]
589         T = (1 / 2) * m * l ^ 2 * square(theta_dot)
590         V = m * g * l * (1 - cos(theta))
591         return T - V
592
593     return Lagrangian
```

```
594 L = L_planar_pendulum(m, g, l)
595 show(L(Gamma(q)(t)))
```

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

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

Henon Heiles potential, Exercise 1.9.b of the book

As the potential depends on the x and y coordinate separately, we need to unpack the coordinate vector.

```
598     def L_Henon_Heiles(m):
599         def Lagrangian(local):
600             x, y = coordinate(local).list()
601             v = velocity(local)
602             T = (1 / 2) * square(v)
603             V = 1 / 2 * (square(x) + square(y)) + square(x) * y - y**3 / 3
604             return T - V
605
606     return Lagrangian
```

```
607 L = L_Henon_Heiles(m)
608 q = column_path([literal_function("x"), literal_function("y")])
609 show(L(Gamma(q)(t)))
```

$$-x^2y + \frac{1}{3}y^3 - \frac{1}{2}x^2 - \frac{1}{2}y^2 + \frac{1}{2}\dot{x}^2 + \frac{1}{2}\dot{y}^2$$

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

$$\begin{bmatrix} -2xy - x & -x^2 + y^2 - y \end{bmatrix}$$

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

$$\begin{bmatrix} \dot{x} & \dot{y} \end{bmatrix}$$

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

```
612 var('R', domain="positive")
613
614
615 def L_sphere(m, R):
616     def Lagrangian(local):
617         theta, phi = coordinate(local).list()
618         alpha, beta = velocity(local).list()
619         L = m * R * (square(alpha) + square(beta * sin(theta))) / 2
620         return L
621
622     return Lagrangian
```

```

623 q = column_path([literal_function("phi"), literal_function("theta")])
624 L = L_sphere(m, R)
625 show(L(Gamma(q)(t)))

```

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

```

626 show(partial(L, 1)(Gamma(q)(t)))

```

$$[Rm \cos(\phi) \sin(\phi) \dot{\theta}^2 \quad 0]$$

```

627 show(partial(L, 2)(Gamma(q)(t)))

```

$$[Rm\dot{\phi} \quad Rm \sin(\phi)^2 \dot{\theta}]$$

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.

The Euler Lagrange Equations

We work in steps to see how all components tie together.

```

629 q = column_path(
630     [
631         literal_function("x"),
632         literal_function("y"),
633     ]
634 )
635
636 L = L_free_particle(m)
637 show(compose(partial(L, 1), Gamma(q))(t))
638 show(compose(partial(L, 2), Gamma(q))(t))
639 show(D(compose(partial(L, 2), Gamma(q)))(t))
640 show(
641     (D(compose(partial(L, 2), Gamma(q))) - compose(partial(L, 1), Gamma(q)))(t)
642 )

```

$$\begin{bmatrix} 0 & 0 \\ m\ddot{x} & m\ddot{y} \\ m\ddot{x} & m\ddot{y} \\ m\ddot{x} & m\ddot{y} \end{bmatrix}$$

The last step forms the Euler-Lagrange equation, which we can now implement as a function.

```
..... ./sage/utils1.5.sage
643 def Lagrange_equations(L):
644     def f(q):
645         return D(compose(partial(L, 2), Gamma(q))) - compose(
646             partial(L, 1), Gamma(q)
647         )
648
649     return f
```

The free particle

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

```
..... ./sage/section1.5.sage
650 var("a b c a0 b0 c0", domain="real")
651 test_path = lambda t: column_matrix([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.

```
..... ./sage/section1.5.sage
652 l_eq = Lagrange_equations(L_free_particle(m))(test_path)
653 show(l_eq(t))
```

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

This is correct since a free particle is not moving in a potential field, hence only depends on the velocity but not the coordinates of the path. But since the velocity is linear in t , all components along the test path become zero.

Here are the EL equations for a generic 1D path.

```
..... ./sage/section1.5.sage
654 q = column_path([literal_function("x")])
655 l_eq = Lagrange_equations(L_free_particle(m))(q)
656 show(l_eq(t))
```

$$\begin{bmatrix} m\ddot{x} \end{bmatrix}$$

Equating this to (0) shows that the solution of these differential equations is linear in t .

The harmonic oscillator

```
657 var("A phi omega", domain="real")
658 assume(A > 0)
659
660 proposed_path = lambda t: vector([A * cos(omega * t + phi)])
```

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

```
661 _____ ..../sage/section1.5.sage
662 l_eq = Lagrange_equations(L_harmonic(m, k))(proposed_path)(t)
663 show(l_eq)
```

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

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

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

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

Let's factor out the cosine.

```
664 _____ ..../sage/section1.5.sage
665 show(l_eq[0, 0].factor())
```

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

Kepler's third law

Recall that to unpack the coordinates, we have to convert the vector to a Python list.

```
665 var("G m m1 m2", domain="positive")
666
667
668 def L_central_polar(m, V):
669     def Lagrangian(local):
670         r, phi = coordinate(local).list()
671         rdot, phidot = velocity(local).list()
672         T = 1 / 2 * m * (square(rdot) + square(r * phidot))
673         return T - V(r)
674
675     return Lagrangian
676
677
```

```

678 def gravitational_energy(G, m1, m2):
679     def f(r):
680         return -G * m1 * m2 / r
681
682     return f
683
684 ..../sage/section1.5.sage
685 q = column_path([literal_function("r"), literal_function("phi")])
686 V = gravitational_energy(G, m1, m2)
687 L = L_central_polar(m, V)
688 show(L(Gamma(q)(t)))
689
690 ..../sage/section1.5.sage
691 l_eq = Lagrange_equations(L)(q)(t)
692
693 ..../sage/section1.5.sage
694 show(l_eq[0, 1] == 0)

```

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

In this equation, let's divide by mr to get $r\ddot{\phi} + 2\dot{\phi}r = 0$, 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?

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

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

1.6 HOW TO FIND LAGRANGIANS

1.6.1 Standard imports

```
690 ..../sage/utils1.6.sage  
load("utils1.5.sage")  
  
691 ..../sage/section1.6.sage  
load("utils1.6.sage")  
  
692 don't tangle  
load("show_expression.sage")
```

1.6.2 Constant acceleration

We start with a point in a uniform gravitational field.

```

693 var("t", domain="real")           ../sage/utils1.6.sage
694 var("g m", domain="positive")
695
696
697 def L_uniform_acceleration(m, g):
698     def wrap_L_unif(local):
699         x, y = coordinate(local).list()
700         v = velocity(local)
701         T = 1 / 2 * m * square(v)
702         V = m * g * y
703         return T - V
704
705     return wrap_L_unif

```

```

706 q = column_path([literal_function("x"), literal_function("y")]) ..../sage/section1.6.sage
707 l_eq = Lagrange_equations(L_uniform_acceleration(m, g))(q)
708 show(l_eq(t))

```

$$[m\ddot{x} \quad gm + m\ddot{y}]$$

1.6.3 Central force field

```

709 def L_central_rectangular(m, U):
710     def Lagrangian(local):
711         q = coordinate(local)
712         v = velocity(local)
713         T = 1 / 2 * m * square(v)
714         return T - U(sqrt(square(q)))
715
716     return Lagrangian

```

Let us first try this on a concrete potential function.

```

717 def U(r):
718     return 1 / r

```

```

719 show(Lagrange_equations(L_central_rectangular(m, U))(q)(t))

```

$$\begin{bmatrix} m\ddot{x} - \frac{x}{(x^2+y^2)^{\frac{3}{2}}} & m\ddot{y} - \frac{y}{(x^2+y^2)^{\frac{3}{2}}} \end{bmatrix}$$

Now we try it on a general central potential.

```
720 _____ ..../sage/section1.6.sage
721 U = Function(lambda x: function("U")(x))
722 show(Lagrange_equations(L_central_rectangular(m, U))(q)(t))
```

$$\begin{bmatrix} m\ddot{x} + \frac{x D_0(U) (\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}} & m\ddot{y} + \frac{y D_0(U) (\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}} \end{bmatrix}$$

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 $\Gamma(q)$.

The result of $\partial_1 F v$ is a vector, because v is a vector. We have to cast $\partial_0 F$ into a vector to enable the summation of these two terms.

```
722 _____ ..../sage/utils1.6.sage
723 def F_to_C(F):
724     def wrap_F_to_C(local):
725         return up(
726             time(local),
727             F(local),
728             partial(F, 0)(local) + partial(F, 1)(local) * velocity(local),
729         )
730     return wrap_F_to_C
```

1.6.5 Polar coordinates

```
731 _____ ..../sage/utils1.6.sage
732 def p_to_r(local):
733     r, phi = coordinate(local).list()
734     return column_matrix([r * cos(phi), r * sin(phi)])
```

We apply $F_{\text{to_}C}$ and $p_{\text{to_}r}$ 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 .

```
734 _____ ..../sage/section1.6.sage
735 r = literal_function("r")
736 phi = literal_function("phi")
737 q = column_path([r, phi])
738 show(p_to_r(Gamma(q)(t)))
```

$$\begin{bmatrix} \cos(\phi)r \\ r\sin(\phi) \end{bmatrix}$$

This is the derivative wrt t . As the transformation $p_{\text{to_r}}$ does not depend explicitly on t , the result should be a column matrix of zeros.

```
738 show((partial(p_to_r, 0)(Gamma(q)(t))))
```

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Next is the derivative wrt r and ϕ .

```
739 show((partial(p_to_r, 1)(Gamma(q)(t))))
```

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

```
740 show(F_to_C(p_to_r)(Gamma(q)(t)))
```

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

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

```
741 def L_central_polar(m, U):
742     def Lagrangian(local):
743         return compose(L_central_rectangular(m, U), F_to_C(p_to_r))(local)
744
745     return Lagrangian
```

```
746 # show(L_central_polar(m, U)(Gamma(q)(t)))
747 show(L_central_polar(m, U)(Gamma(q)(t)).simplify_full())
```

$$\frac{1}{2}mr^2\dot{\phi}^2 + \frac{1}{2}m\dot{r}^2 - U(\sqrt{r^2})$$

```
748 expr = Lagrange_equations(L_central_polar(m, U))(q)(t)
749 show(expr.simplify_full().expand())
```

$$\begin{bmatrix} -mr\dot{\phi}^2 + m\ddot{r} + \frac{rD_0(U)(\sqrt{r^2})}{\sqrt{r^2}} & mr^2\ddot{\phi} + 2mr\dot{\phi}\dot{r} \end{bmatrix}$$

1.6.6 Coriolis and centrifugal forces

```
..... ./sage/utils1.6.sage .....
```

```

750 def L_free_rectangular(m):
751     def Lagrangian(local):
752         v = velocity(local)
753         return 1 / 2 * m * square(v)
754
755     return Lagrangian
756
757
758 def L_free_polar(m):
759     def Lagrangian(local):
760         return L_free_rectangular(m)(F_to_C(p_to_r)(local))
761
762     return Lagrangian
763
764
765 def F(Omega):
766     def f(local):
767         t = time(local)
768         r, theta = coordinate(local).list()
769         return vector([r, theta + Omega * t])
770
771     return f
772
773
774 def L_rotating_polar(m, Omega):
775     def Lagrangian(local):
776         return L_free_polar(m)(F_to_C(F(Omega))(local))
777
778     return Lagrangian
779
780
781
782 def r_to_p(local):
783     x, y = coordinate(local).list()
784     return column_matrix([sqrt(x * x + y * y), atan(y / x)])
785
786
787 def L_rotating_rectangular(m, Omega):
788     def Lagrangian(local):
789         return L_rotating_polar(m, Omega)(F_to_C(r_to_p)(local))
790
791     return Lagrangian

```

```
..... ./sage/section1.6.sage .....
```

```

792 _ = var("Omega", domain="positive")
793 q_xy = column_path([literal_function("x"), literal_function("y")])
794 expr = L_rotating_rectangular(m, Omega)(Gamma(q_xy)(t)).simplify_full()

```

```
795 show(expr) ..../sage/section1.6.sage
```

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

The simplification of the Lagrange equations takes some time.

```
796 expr = Lagrange_equations(L_rotating_rectangular(m, Omega))(q)(t) don't tangle
797 show(expr.simplify_full())
```

I edited the result a bit by hand.

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

1.6.7 Constraints, a driven pendulum

Rather than implementation the formulas of the book at this place, we follow the idea they explain at bit later in the book: 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.

For the driven pendulum, 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 (1×1) column matrix which just contains θ . So, we have to pick element $(0,0)$. Another point is that here `ys` needs to be evaluated at t ; in the other functions `ys` is just passed on as a function.

```
798 def dp_coordinates(l, ys):
799     "From theta to x, y coordinates."
800     def wrap_dp(local):
801         t = time(local)
802         theta = coordinate(local)[0, 0]
803         return column_matrix([l * sin(theta), ys(t) - l * cos(theta)])
804
805     return wrap_dp
```

```
806 def L_pend(m, l, g, ys):
807     def wrap_L_pend(local):
808         return L_uniform_acceleration(m, g)(
809             F_to_C(dp_coordinates(l, ys))(local)
810         )
811
812     return wrap_L_pend
```

```

813 ..../sage/section1.6.sage
814 _ = var("l", domain="positive")
815 theta = column_path([literal_function("theta")])
816 ys = literal_function("y")
817
818 expr = L_pend(m, l, g, ys)(Gamma(theta)(t)).simplify_full()
819 show(expr)

```

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

1.7 EVOLUTION OF DYNAMICAL STATE

1.7.1 Standard imports

```

820 ..../sage/utils1.7.sage
load("utils1.6.sage")

```

```

821 ..../sage/section1.7.sage
load("utils1.7.sage")

```

```

822 var("t", domain=RR)

```

```

823 don't tangle
824 load("show_expression.sage")

```

1.7.2 Acceleration and state derivative

We build the functions `Lagrangian_to_acceleration` and `Lagrangian_to_state_derivative` in steps.

```

825 ..../sage/section1.7.sage
q = column_path([literal_function("x"), literal_function("y")])
826 local = Gamma(q)(t)
827 m, k = var("m k", domain="positive")
828 L = L_harmonic(m, k)
829 show(L(local))

```

$$-\frac{1}{2} (x^2 + y^2)k + \frac{1}{2} (\dot{x}^2 + \dot{y}^2)m$$

```

830 ..../sage/section1.7.sage
F = compose(transpose, partial(L, 1))
831 show(F(local))
832 P = partial(L, 2)
833 show((F - partial(P, 0))(local))

```

$$\begin{pmatrix} -kx \\ -ky \end{pmatrix}$$

$$\begin{pmatrix} -kx \\ -ky \end{pmatrix}$$

```
834   _____ ..../sage/section1.7.sage _____
show((partial(P, 1) * velocity)(local))
```

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

Convert to vector.

```
835   _____ ..../sage/section1.7.sage _____
show((F - partial(P, 0) - partial(P, 1) * velocity)(local))
```

$$\begin{pmatrix} -kx \\ -ky \end{pmatrix}$$

```
836 def Lagrangian_to_acceleration(L):
837     def f(local):
838         P = partial(L, 2)
839         F = compose(transpose, partial(L, 1))
840         M = (F - partial(P, 0)) - partial(P, 1) * velocity
841         return partial(P, 2)(local).solve_right(M(local))
842
843     return f
```

We apply this to the harmonic oscillator.

```
844   _____ ..../sage/section1.7.sage _____
show(Lagrangian_to_acceleration(L)(local))
```

$$\begin{pmatrix} -\frac{kx}{m} \\ -\frac{ky}{m} \end{pmatrix}$$

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.

Consider the system of DEs for the circle: $\dot{x} = y$, $\dot{y} = -x$. This code implements the rhs:

```
don't tangle
845 def de_rhs(x, y):
846     return [y, -x]
847
848
849 sol = desolve_odeint(de_rhs(x, y), [1, 0], xrange(0, 100, 0.05), [x, y])
850 pp = list(zip(sol[:, 0], sol[:, 1]))
851 p = points(pp, color='blue', size=3)
852 p.save('circle.png')
```

However, if I replace the RHS of the DE by by constants,, I get an error that the integration variables are unknown.

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

The solution is to replace the numbers by expressions.

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

And then define the function of differentials like this.

```
don't tangle
857 def de_rhs(x, y):
858     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 $dt/dt = 1$ to an expression, more specifically, by the above intermezzo we should set the derivative of t to `convert_to_expr(1)`.

```
..../sage/utils1.7.sage
859 def Lagrangian_to_state_derivative(L):
860     acceleration = Lagrangian_to_acceleration(L)
861     return lambda state: up(
862         convert_to_expr(1), velocity(state), acceleration(state)
863     )
```

```
..../sage/section1.7.sage
864 show(Lagrangian_to_state_derivative(L)(local))
```

$$\begin{pmatrix} 1 \\ \dot{x} \\ \dot{y} \\ -\frac{kx}{m} \\ -\frac{ky}{m} \end{pmatrix}$$

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

```
867 show(harmonic_state_derivative(m, k)(local))
```

$$\begin{pmatrix} 1 \\ \dot{x} \\ \dot{y} \\ -\frac{kx}{m} \\ -\frac{ky}{m} \end{pmatrix}$$

```
868 def qv_to_state_path(q, v):
869     return lambda t: up(t, q(t), v(t))
```

```
870 def Lagrange_equations_first_order(L):
871     def f(q, v):
872         state_path = qv_to_state_path(q, v)
873         res = D(state_path)
874         res -= compose(Lagrangian_to_state_derivative(L), state_path)
875         return res
876
877     return f
```

```
878 res = Lagrange_equations_first_order(L_harmonic(m, k))(
879     column_path([literal_function("x"), literal_function("y")]),
880     column_path([literal_function("v_x"), literal_function("v_y")]),
881 )
882 show(res(t))
```

$$\begin{pmatrix} 0 \\ -v_x + \dot{x} \\ -v_y + \dot{y} \\ \frac{kx}{m} + \ddot{v}_x \\ \frac{ky}{m} + \ddot{v}_y \end{pmatrix}$$

1.7.5 Numerical integration

For the numerical integrator we have to specify the variables that appear in the differential equations. For this purpose we use dummy vectors.

```
.../sage/utils1.7.sage
883 def make_dummy_vector(name, dim):
884     return column_matrix([var(f"{name}{i}", domain=RR) for i in range(dim)])
```

The state_advancer needs an evolve function. We use the initial conditions ics to figure out the dimension of the coordinate space. Once we have the dimension, we construct a dummy up tuple with coordinate and velocity variables. The ode solver need plain lists; since space is an up tuple, the list method of Tuple can provide for this.

```
.../sage/utils1.7.sage
885 def evolve(state_derivative, ics, times):
886     dim = coordinate(ics).nrows()
887     coordinates = make_dummy_vector("q", dim)
888     velocities = make_dummy_vector("v", dim)
889     space = up(t, coordinates, velocities)
890     soln = desolve_odeint(
891         des=state_derivative(space).list(),
892         ics=ics.list(),
893         times=times,
894         dvars=space.list(),
895         atol=1e-13,
896     )
897     return soln
```

The state advancer integrates the orbit for a time T and starting at the initial conditions.

```
.../sage/utils1.7.sage
898 def state_advancer(state_derivative, ics, T):
899     init_time = time(ics)
900     times = [init_time, init_time + T]
901     soln = evolve(state_derivative, ics, times)
902     return soln[-1]
```

As a test, let's apply it to the one D harmonic oscillator.

```
.../sage/section1.7.sage
903 state_advancer(
904     harmonic_state_derivative(m=2, k=1),
905     ics=up(0, column_matrix([1, 2]), column_matrix([3, 4])),
906     T=10,
907 )
```

array([10. , 3.71279102, 5.42061989, 1.61480284, 1.8189101])

These are (nearly) the same results as in the book.

1.7.6 The driven pendulum

Here is the driver for the pendulum.

```
908     def periodic_drive(amplitude, frequency, phase):
909         def f(t):
910             return amplitude * cos(frequency * t + phase)
911
912     return f
```

With this we make the Lagrangian.

```
913 _ = var("m l g A omega")
914
915
916
917 def L_periodically_driven_pendulum(m, l, g, A, omega):
918     ys = periodic_drive(A, omega, 0)
919
920     def L_periodic(local):
921         return L_pend(m, l, g, ys)(local)
922
923     return L_periodic
```

```
924 q = column_path([literal_function("theta")])
925 show(
926     L_periodically_driven_pendulum(m, l, g, A, omega)(
927         Gamma(q)(t)
928     ).simplify_full()
929 )
```

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

```
930 expr = Lagrange_equations(L_periodically_driven_pendulum(m, l, g, A, omega))(
931     q
932 )(t).simplify_full()
933 show(expr)
```

$$(l^2 m \ddot{\theta} - (A l m \omega^2 \cos(\omega t) - g l m) \sin(\theta))$$

```
934 show(
935     Lagrangian_to_acceleration(
936         L_periodically_driven_pendulum(m, l, g, A, omega)
937     )(Gamma(q)(t)).simplify_full()
938 )
```

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

```

939 def pend_state_derivative(m, l, g, A, omega):
940     return Lagrangian_to_state_derivative(
941         L_periodically_driven_pendulum(m, l, g, A, omega)
942     )

```

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

```

943 expr = pend_state_derivative(m, l, g, A, omega)(Gamma(q)(t))
944 show(time(expr))
945 show(coordinate(expr).simplify_full())
946 show(velocity(expr).simplify_full())

```

1

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

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

```

947 def principal_value(cut_point):
948     def f(x):
949         return (x + cut_point) % (2 * np.pi) - cut_point
950
951     return f

```

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

```

952 def plot_driven_pendulum(A, T, step_size=0.01):
953     times = srange(0, T, step_size, include_endpoint=True)
954     soln = evolve(
955         pend_state_derivative(m=1, l=1, g=9.8, A=A, omega=2 * sqrt(9.8)),
956         ics=up(0, column_matrix([1]), column_matrix([0])),
957         times=times,
958     )
959     thetas = soln[:, 1]
960     pp = list(zip(times, thetas))
961     p = points(pp, color='blue', size=3)
962     p.save(f'../figures/driven_pendulum_{A:.2f}.png')
963
964     thetas = principal_value(np.pi)(thetas)
965     pp = list(zip(times, thetas))
966     p = points(pp, color='blue', size=3)
967     p.save(f'../figures/driven_pendulum_{A:.2f}_principal_value.png')
968
969     thetadots = soln[:, 2]
970     pp = list(zip(thetas, thetadots))
971     p = points(pp, color='blue', size=3)
972     p.save(f'../figures/driven_pendulum_{A:.2f}_trajectory.png')
973

```

So now we make the plot.

```
974          .../sage/section1.7.sage
plot_driven_pendulum(A=0.1, T=100, step_size=0.005)
```

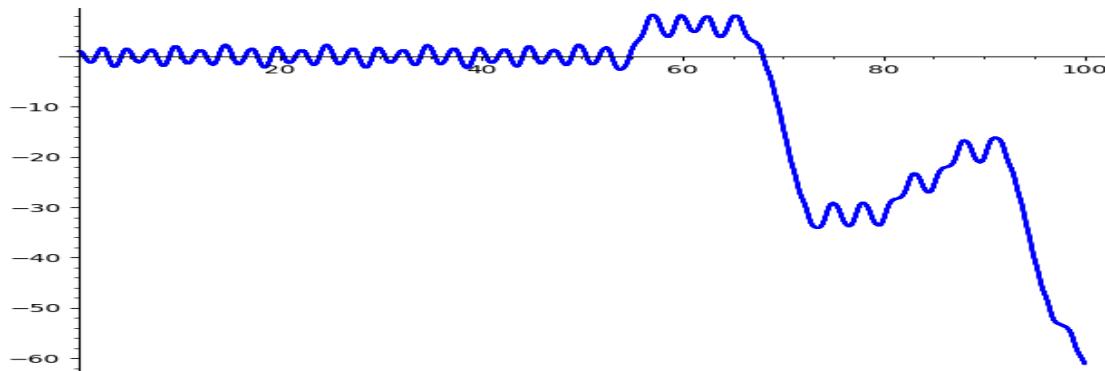


Figure 1.2: The angle of the vertically driven pendulum as a function of time. Obviously, around $t = 80$, the pendulum makes a few revolutions, and then starts to wobble again.

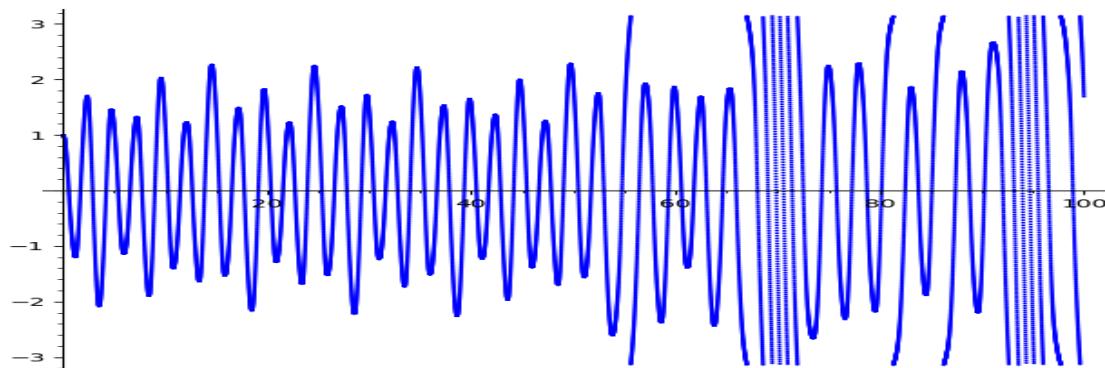


Figure 1.3: The angle on $(-\pi, \pi]$.

1.8 CONSERVED QUANTITIES

1.8.1 Standard imports

```
975          .../sage/utils1.8.sage
load("utils1.6.sage")
```

```
976          .../sage/section1.8.sage
load("utils1.8.sage")
```

```
977          var("t", domain=RR)
```

```
978          don't tangle
```

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

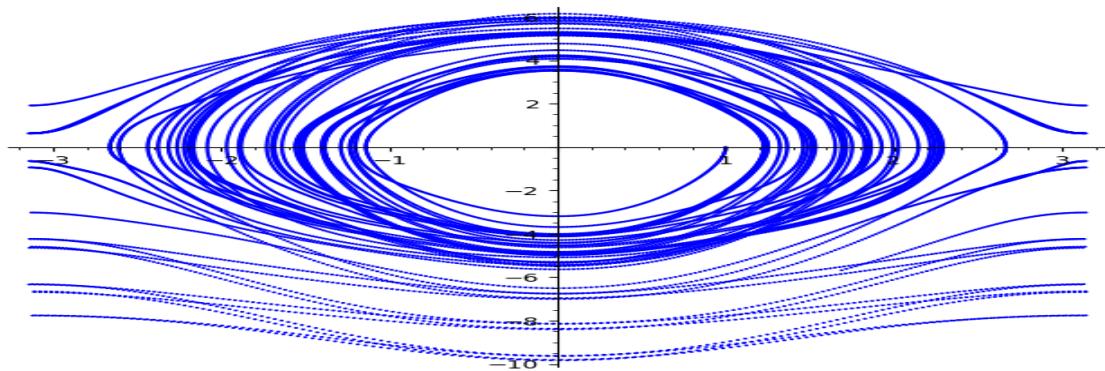


Figure 1.4: The trajectory of θ and $\dot{\theta}$.

1.8.2 1.8.2 Energy Conservation

From the Lagrangian we can construct the energy function. 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. As we use the Lagrangian in functional arithmetic, we convert L into a Function.

```
980 def Lagrangian_to_energy(L):
981     P = partial(L, 2)
982     LL = Function(lambda local: L(local))
983     return lambda local: (P * velocity - LL)(local)
```

1.8.3 Central Forces in Three Dimensions

Instead of building the kinetic energy in spherical coordinates, as in Section 1.8.3 of the book, I am going to use the ideas that have been expounded 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.

```
984 q = column_path(
985     [
986         literal_function("r"),
987         literal_function("theta"),
988         literal_function("phi"),
989     ]
990 )
```

Next the transformation from spherical to 3D rectangular coordinates.

```
991 def s_to_r(spherical_state):
992     r, theta, phi = coordinate(spherical_state).list()
```

```

993     return vector(
994         [r * sin(theta) * cos(phi), r * sin(theta) * sin(phi), r * cos(theta)])
995

```

For example, here are the velocities expressed in spherical coordinates.

```

996 show(velocity(F_to_C(s_to_r)(Gamma(q)(t))).simplify_full())

```

$$\begin{bmatrix} \cos(\phi)\cos(\theta)r\dot{\theta} - (r\sin(\phi)\dot{\phi} - \cos(\phi)\dot{r})\sin(\theta) \\ \cos(\theta)r\sin(\phi)\dot{\theta} + (\cos(\phi)r\dot{\phi} + \sin(\phi)\dot{r})\sin(\theta) \\ -r\sin(\theta)\dot{\theta} + \cos(\theta)\dot{r} \end{bmatrix}$$

Now we are ready to check the code examples of the book.

```

997 V = Function(lambda r: function("V")(r))
998
999 def L_3D_central(m, V):
1000     def Lagrangian(local):
1001         return L_center_rectangular(m, V)(F_to_C(s_to_r)(local))
1002
1003     return Lagrangian

```

```

1004 show(partial(L_3D_central(m, V), 1)(Gamma(q)(t)).simplify_full())

```

$$\begin{bmatrix} -\frac{rD_0(V)(\sqrt{r^2}) - (mr\sin(\theta)^2\dot{\phi}^2 + mr\dot{r}^2)\sqrt{r^2}}{\sqrt{r^2}} & m\cos(\theta)r^2\sin(\theta)\dot{\phi}^2 & 0 \end{bmatrix}$$

```

1005 show(partial(L_3D_central(m, V), 2)(Gamma(q)(t)).simplify_full())

```

$$\begin{bmatrix} m\dot{r} & mr^2\dot{\theta} & mr^2\sin(\theta)^2\dot{\phi} \end{bmatrix}$$

```

1006 def ang_mom_z(m):
1007     def f(rectangular_state):
1008         xyx = vector(coordinate(rectangular_state))
1009         v = vector(velocity(rectangular_state))
1010         return xyx.cross_product(m * v)[2]
1011
1012     return f
1013
1014
1015 show(compose(ang_mom_z(m), F_to_C(s_to_r))(Gamma(q)(t)).simplify_full())

```

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

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

```
1016 show(Lagrangian_to_energy(L_3D_central(m, V))(Gamma(q)(t)).simplify_full())
```

$$\left[\frac{1}{2} mr^2 \sin(\theta)^2 \dot{\phi}^2 + \frac{1}{2} mr^2 \dot{\theta}^2 + \frac{1}{2} m \dot{r}^2 + V(\sqrt{r^2}) \right]$$

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.

```
1017 var("G M0 M1 a", domain="positive")
1018
1019
1020 def distance(x, y):
1021     return sqrt(square(x - y))
1022
1023
1024 def angular_freq(M0, M1, a):
1025     return sqrt(G * (M0 + M1) / a ^ 3)
1026
1027
1028 def V(a, M0, M1, m):
1029     Omega = angular_freq(M0, M1, a)
1030     a0, a1 = M1 / (M0 + M1) * a, M0 / (M0 + M1) * a
1031
1032     def f(t, origin):
1033         pos0 = -a0 * column_matrix([cos(Omega * t), sin(Omega * t)])
1034         pos1 = a1 * column_matrix([cos(Omega * t), sin(Omega * t)])
1035         r0 = distance(origin, pos0)
1036         r1 = distance(origin, pos1)
1037         return -G * m * (M0 / r0 + M1 / r1)
1038
1039     return f
1040
1041 def L0(m, V):
1042     def f(local):
1043         t, q, v = time(local), coordinate(local), velocity(local)
1044         return 1 / 2 * m * square(v) - V(t, q)
1045
1046     return f
```

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

```

1047 q = column_path([literal_function("x"), literal_function("y")])
1048 expr = (sqrt(G*M0 + G*M1)*t) / a^(3/2)
1049 A = var('A')
1050
1051 show(
1052     Lagrangian_to_energy(L0(m, V(a, M0, M1, m)))(Gamma(q)(t))
1053     .simplify_full()
1054     .expand()
1055     .subs({expr: A})
1056 )

```

$$\left[-\frac{\sqrt{M_0^2+2 M_0 M_1+M_1^2} G M_0 m}{\sqrt{2 M_0 M_1 a \cos(A) x+2 M_1^2 a \cos(A) x+2 M_0 M_1 a \sin(A) y+2 M_1^2 a \sin(A) y+M_1^2 a^2+M_0^2 x^2+2 M_0 M_1 x^2+M_1^2 x^2+M_0^2 y^2+2 M_0 M_1 y^2+M_1^2 y^2}}$$

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

We need to rotate around a given axis in 3D space. ChatGPT gave me the code right away.

```

1057 def rotation_matrix(axis, theta):
1058     """
1059         Return the 3x3 rotation matrix for a rotation of angle theta (in radians)
1060         about the given axis. The axis is specified as an iterable of 3 numbers.
1061     """
1062     # Convert the axis to a normalized vector
1063     axis = vector(axis).normalized()
1064     x, y, z = axis
1065     c = cos(theta)
1066     s = sin(theta)
1067     t = 1 - c # common factor
1068
1069     # Construct the rotation matrix using Rodrigues' formula
1070     R = matrix(
1071         [
1072             [c + x**2 * t, x * y * t - z * s, x * z * t + y * s],
1073             [y * x * t + z * s, c + y**2 * t, y * z * t - x * s],
1074             [z * x * t - y * s, z * y * t + x * s, c + z**2 * t],
1075         ]
1076     )
1077     return R

```

```

1078 def F_tilde(angle_x, angle_y, angle_z):
1079     def f(local):

```

```

1080     return (
1081         rotation_matrix([1, 0, 0], angle_x)
1082         * rotation_matrix([0, 1, 0], angle_y)
1083         * rotation_matrix([0, 0, 1], angle_z)
1084         * coordinate(local)
1085     )
1086
1087     return f

```

```

1088     q = column_path(
1089         [literal_function("x"), literal_function("y"), literal_function("z")]
1090     )

```

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

```

1091 def Rx(s):
1092     return lambda local: F_tilde(s, 0, 0)(local)
1093
1094
1095 s, u, v = var("s u v")
1096 latex.matrix_delimiters(left='[', right=']')
1097 latex.matrix_column_alignment("c")
1098 show(Rx(s)(Gamma(q)(t)))
1099 show(diff(Rx(s)(Gamma(q)(t)), s)(s=0))

```

$$\begin{bmatrix} x \\ \cos(s)y - \sin(s)z \\ \sin(s)y + \cos(s)z \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ -z \\ y \end{bmatrix}$$

And now we check the result of the book. The computation of $D F_{\tilde{t}}$ is somewhat complicated. Observe that $F_{\tilde{t}}$ is a function of the rotation angles, and returns a function that takes $local$ as argument. Now we want to differentiate $F_{\tilde{t}}$ with respect to the angles, so these are the variables we need to provide to the Jacobian. For this reason, we bind the result of $F_{\tilde{t}}$ to $local$, and use a lambda function to provide the angles as the variables. This gives us $F_{\tilde{t}}$ (note that I drop the underscore in this name). There is one further point: $F_{\tilde{t}}$ expects three angles, while the Jacobian provides the list $[s, u, v]$ as the argument to $F_{\tilde{t}}$. Therefore we unpack the argument x of the lambda function to convert the list $[s, u, v]$ into three separate arguments. The last step is to fill in $s = u = v = 0$.

Note that we differentiate wrt s, u, v and not wrt t . In itself, using t would not be a problem, but since we pass $\text{Gamma}(q)(t)$ to F_{tilde} , the function depends also on t via the path $t \rightarrow \Gamma(q, t)$ which we should avoid.

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

```
1100 _____ ..../sage/section1.8.sage _____
1100 U = Function(lambda r: function("U")(r))
1101
1102
1103 def the_Noether_integral(local):
1104     L = L_central_rectangular(m, U)
1105     Ftilde = lambda x: F_tilde(*x)(local)
1106     DF0 = Jacobian(Ftilde)([s, u, v], [s, u, v])(s=0, u=0, v=0)
1107     return partial(L, 2)(local) * DF0
```

```
1108 _____ ..../sage/section1.8.sage _____
1108 show(the_Noether_integral(Gamma(q)(t)).simplify_full())
```

$$[-mz\dot{y} + my\dot{z} \quad mz\dot{x} - mx\dot{z} \quad -my\dot{x} + mx\dot{y}]$$

1.9 ABSTRACTION OF PATH FUNCTIONS

I found this section difficult to understand, so I work in small steps to the final result, and include checks to see what goes on.

1.9.1 Standard imports

```
1109 _____ ..../sage/utils1.9.sage _____
1109 load("utils1.6.sage")
```

```
1110 _____ ..../sage/section1.9.sage _____
1110 load("utils1.9.sage")
1111
1112 var("t", domain=RR)
```

```
1113 _____ don't tangle _____
1113 load("show_expression.sage")
```

1.9.2 Understanding $F_{\text{to}}C$

The Scheme code starts with defining $\text{Gamma}_{\text{bar}}$ in terms of f_{bar} and osculating_path . We build f_{bar} first and apply it to the example in which polar coordinates are converted to rectilinear coordinates.

Next, let's spell out the arguments of all functions to see how everything works together. A literal function maps time t to some part of the space, often to a coordinate, x say.

```
1114      _____ ..../sage/section1.9.sage
r, theta = literal_function("r"), literal_function("theta")
1115      show(r)
```

```
<__main__.Function object at 0x752ed4eb27a0>
```

So, r is a Function. We can evaluate r at t . I pass `simplify=False` to show to *not* suppress the dependence on t .

```
1116      _____ ..../sage/section1.9.sage
show((r(t), theta(t)), simplify=False)
```

```
(r(t),θ(t))
```

A `column_path` takes literal functions as arguments and returns a coordinate path. Hence, it is a function of t and returns $q(t)$. (I use the notation of the code examples of the book such as `q_prime` so that I can copy the examples into the functions I build later.)

```
1117      _____ ..../sage/section1.9.sage
q_prime = column_path([r, theta])
1118      show(q_prime(t), simplify=False)
```

$$\begin{bmatrix} r(t) \\ \theta(t) \end{bmatrix}$$

The function Γ takes a coordinate path q (which is a function of time) as input, and returns a function of t that maps to a local up tuple l :

$$\Gamma[q] : t \rightarrow l = (t, q(t), v(t), \dots).$$

```
1119      _____ ..../sage/section1.9.sage
show(Gamma(q_prime))
```

```
<function Gamma.<locals>.<lambda> at 0x752ed4ba0cc0>
```

Indeed, `Gamma` is a function, and has to be applied to some argument to result into a value. In fact, when $\Gamma(q)$ is applied to t , we get the local up tuple l . Observe, that a local tuple is *not* a functions of time, by that I mean, a local is not a Python function of time, and therefore does not take any further arguments.

```
1120      _____ ..../sage/section1.9.sage
show(Gamma(q_prime)(t), simplify=False)
```

$$\begin{bmatrix} t \\ r(t) \\ \theta(t) \\ \frac{\partial}{\partial t}r(t) \\ \frac{\partial}{\partial t}\theta(t) \end{bmatrix}$$

The coordinate transformation F in the example that transforms polar coordinates to rectilinear coordinates is `p_to_r`. This transform F maps a local tuple l to coordinates $q(t)$. Therefore, we can apply F to $\Gamma[q](t)$, and use composition like this:

$$F(\Gamma[q](t)) = (F \circ \Gamma[q])(t).$$

Observe that $F \circ \Gamma[q]$ is a function of t .

```
1121 F = p_to_r
1122 show(compose(F, Gamma(q_prime))(t), simplify=False)
```

$$\begin{bmatrix} \cos(\theta(t))r(t) \\ r(t)\sin(\theta(t)) \end{bmatrix}$$

Since $F \circ \Gamma[q]$ is a function of t to a coordinate path $q(t)$, this function has the same ‘protocol’ as a coordinate path function. We can therefore apply Γ to the composite function $F \circ \Gamma[q]$ to obtain a function that maps t to a local tuple in the transformed space.

$$Q : t \rightarrow \Gamma[F \circ \Gamma[q]](t).$$

```
1123 Q = lambda t: compose(p_to_r, Gamma(q_prime))(t)
1124 show(Gamma(Q)(t), simplify=False)
```

$$\begin{bmatrix} t \\ \cos(\theta(t))r(t) \\ r(t)\sin(\theta(t)) \\ -r(t)\sin(\theta(t))\frac{\partial}{\partial t}\theta(t) + \cos(\theta(t))\frac{\partial}{\partial t}r(t) \\ \cos(\theta(t))r(t)\frac{\partial}{\partial t}\theta(t) + \sin(\theta(t))\frac{\partial}{\partial t}r(t) \end{bmatrix}$$

Now that we have analyzed all steps, we can make `f_bar`.

```
1125 def f_bar(q_prime):
1126     q = lambda t: compose(F, Gamma(q_prime))(t)
1127     return lambda t: Gamma(q)(t)
```

Here is the check. I suppress the dependence on t again to keep the result easier to read.

```
1128 show(f_bar(q_prime)(t))
```

$$\begin{bmatrix} t \\ \cos(\theta)r \\ r\sin(\theta) \\ -r\sin(\theta)\dot{\theta} + \cos(\theta)\dot{r} \\ \cos(\theta)r\dot{\theta} + \sin(\theta)\dot{r} \end{bmatrix}$$

The second function to build is `osculating_path`. This is the Taylor series of the book in which a local tuple is mapped to coordinate space:

$$O(t, q, v, a, \dots)(\cdot) = q + v(\cdot - t) + a/2(\cdot - t)^2 + \dots$$

I write \cdot instead of t' to make explicit that $O(l)$ is still a function of t' in this case.

Clearly, the RHS is a sum of vectors all of which have the same dimension as the space of coordinates.

Rather than computing dt^n as $(t - t')^n$, and $n!$ for each n , I compute these values recursively. The implementation assumes that the local tuple $\Gamma[q](t)$ contains at least the elements t and q , that is $\Gamma[q](t) = (t, q, \dots)$. This local tuple has length 2; the local tuple $l = (t, q, v)$ has length 3.

```
1129 def osculating_path(local):
```

```
1130     t = time(local)
```

```
1131     q = coordinate(local)
```

```
1132     def wrapper(t_prime):
```

```
1133         res = q
```

```
1134         dt = 1
```

```
1135         factorial = 1
```

```
1136         for k in range(2, len(local)):
```

```
1137             factorial *= k
```

```
1138             dt *= t_prime - t
```

```
1139             res += local[k] * dt / factorial
```

```
1140         return res
```

```
1141     return wrapper
```

Here is an example.

```
1144 t_prime = var("tt", domain="positive", latex_name="t'")
```

```
1145 q = column_path([literal_function("r"), literal_function("theta")])
```

```
1146 local = Gamma(q)(t)
```

```
1147 show(osculating_path(local)(t_prime))
```

$$\begin{bmatrix} -\frac{1}{2}(t-t')\dot{r} + r \\ -\frac{1}{2}(t-t')\dot{\theta} + \theta \end{bmatrix}$$

With the above pieces we can finally build `Gamma_bar`.

```
1148 def Gamma_bar(f_bar):
1149     def wrapped(local):
1150         t = time(local)
1151         q_prime = osculating_path(local)
1152         return f_bar(q_prime)(t)
1153
1154     return wrapped
```

```
1155 show(Gamma_bar(f_bar)(local))
```

$$\begin{bmatrix} t \\ \cos(\theta)r \\ r\sin(\theta) \\ -r\sin(\theta)\dot{\theta} + \cos(\theta)\dot{r} \\ \cos(\theta)r\dot{\theta} + \sin(\theta)\dot{r} \end{bmatrix}$$

We can use `Gamma_bar` in to produce the transformation for polar to rectilinear coordinates.

```
1156 def F_to_C(F):
1157     def C(local):
1158         n = len(local)
1159
1160         def f_bar(q_prime):
1161             q = lambda t: compose(F, Gamma(q_prime))(t)
1162             return lambda t: Gamma(q, n)(t)
1163
1164         return Gamma_bar(f_bar)(local)
1165
1166     return C
```

```
1167 show(F_to_C(p_to_r)(local))
```

$$\begin{bmatrix} t \\ \cos(\theta)r \\ r\sin(\theta) \\ -r\sin(\theta)\dot{\theta} + \cos(\theta)\dot{r} \\ \cos(\theta)r\dot{\theta} + \sin(\theta)\dot{r} \end{bmatrix}$$

Here is the total time derivative.

```
1168 ..../sage/utils1.9.sage
1169 @Func
1170 def Dt(F):
1171     def DtF(local):
1172         n = len(local)
1173
1174         def DF_on_path(q):
1175             return D(lambda t: F(Gamma(q, n - 1))(t)))
1176
1177         return Gamma_bar(DF_on_path)(local)
1178
1179     return lambda state: DtF(local)
```

1.9.3 Lagrange equations at a moment

```
1179 ..../sage/utils1.9.sage
1180 def Euler_Lagrange_operator(L):
1181     return lambda local: (Dt(partial(L, 2)) - partial(L, 1))(local)
```

To apply this operator to a local tuple, we need to include the acceleration.

```
1181 ..../sage/section1.9.sage
1182 q = column_path([literal_function("x")])
1183 local = Gamma(q, 4)(t)
1184 show(local)
```

$$\begin{bmatrix} t \\ x \\ \dot{x} \\ \ddot{x} \end{bmatrix}$$

```
1184 ..../sage/section1.9.sage
1185 m, k = var("m k", domain="positive")
1186 L = L_harmonic(m, k)
1187 show(Euler_Lagrange_operator(L)(local))
```

$$[kx + m\ddot{x}]$$

2

CHAPTER 2: SKIPPED

CHAPTER 3

3.1 HAMILTON'S EQUATIONS

3.1.1 Standard imports

```
1187      ..... /sage/utils3.1.sage
load("utils1.6.sage")
```



```
1188      ..... /sage/section3.1.sage
load("utils3.1.sage")
```



```
1189      t = var("t", domain="real")
```



```
1191      ..... don't tangle
load("show_expression.sage")
```

3.1.2 Computing Hamilton's equations

The code in Section 3.1 of the book starts with the following function.

```
1192      ..... /sage/utils3.1.sage
def Hamilton_equations(Hamiltonian):
    def f(q, p):
        state_path = qp_to_H_state_path(q, p)
        return D(state_path) - compose(
            Hamiltonian_to_state_derivative(Hamiltonian), state_path
        )
    return f
```

This needs the next function.

```
1200      ..... /sage/utils3.1.sage
def qp_to_H_state_path(q, p):
    def f(t):
        return up(t, q(t), p(t))
    return f
```

Here p is a function that maps t to a momentum vector. Such vectors are represented as lying vectors (or down tuples in the book). To implement this, `row_path` takes a list

and returns a function that maps time to the transpose of a column path. In passing we define `row_matrix` as the transpose of `column_matrix`, which is the function provided by Sagemath, and a `transpose` function.

```
1205 def transpose(M):
1206     return M.T
1207
1208
1209 def row_path(lst):
1210     return lambda t: transpose(column_path(lst)(t))
1211
1212
1213 def row_matrix(lst):
1214     return transpose(column_matrix(lst))
```

Let's try what we built.

```
1215 q = column_path([literal_function("q_x"), literal_function("q_y")])
1216 p = row_path([literal_function("p_x"), literal_function("p_y")])
1217
1218 show(p(t))
```

$$\begin{bmatrix} p_x & p_y \end{bmatrix}$$

```
1218 H_state = qp_to_H_state_path(q, p)(t)
1219 show(H_state)
```

$$\begin{bmatrix} t \\ \begin{bmatrix} q_x \\ q_y \end{bmatrix} \\ \begin{bmatrix} p_x & p_y \end{bmatrix} \end{bmatrix}$$

The next function on which `Hamiltonian_equations` depends is `Hamiltonian_to_state_derivative`. The book prints the system of differential equations as a column vector. Therefore we transpose $\partial_2 H$.

```
1220 def Hamiltonian_to_state_derivative(Hamiltonian):
1221     def f(H_state):
1222         return up(
1223             SR(1),
1224             partial(Hamiltonian, 2)(H_state).T,
1225             -partial(Hamiltonian, 1)(H_state),
1226         )
1227
1228     return f
```

Here is an example with `H_rectangular`. For some reason, the book takes just the first and second component of `q`, i.e. (`req q 0`) and (`ref q 1`), to pass to the potential, but the general formula works just as well.

```
1229      var("m")
1230
1231  def H_rectangular(m, V):
1232      def f(state):
1233          q, p = coordinate(state), momentum(state)
1234          return square(p) / 2 / m + V(q)
1235
1236      return f
```

For this to work, we need a momentum projection operator. It's the same as the velocity projection.

```
1237  momentum = Function(lambda H_state: H_state[2])
```

Recall, to use symbolic functions in differentiation, the symbolic function requires an unpacked list of arguments.

```
1238  V = Function(lambda x: function("V")(*x.list()))
```

This is the Hamiltonian.

```
1239  H = H_rectangular
1240  show(H(m, V)(H_state))
```

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

Partial derivatives work.

```
1241  show(partial(H(m, V), 1)(H_state))
```

$$[D_0(V)(q_x, q_y) \quad D_1(V)(q_x, q_y)]$$

```
1242  show(Hamiltonian_to_state_derivative(H(m, V))(H_state))
```

$$\begin{bmatrix} 1 \\ \begin{bmatrix} \frac{p_x}{m} & \frac{p_y}{m} \end{bmatrix} \\ \begin{bmatrix} -D_0(V)(q_x, q_y) & -D_1(V)(q_x, q_y) \end{bmatrix} \end{bmatrix}$$

```
1243 ..../sage/section3.1.sage
1243 show(Hamilton_equations(H(m, V))(q, p)(t))
```

$$\begin{bmatrix} 0 \\ -\frac{p_x}{m} + \dot{q}_x \\ -\frac{p_y}{m} + \dot{q}_y \\ D_0(V)(q_x, q_y) + \dot{p}_x \quad D_1(V)(q_x, q_y) + \dot{p}_y \end{bmatrix}$$

3.1.3 The Legendre Transformation

To understand the code of the book, observe the following.

$$\begin{aligned} F(v) &= 1/2v^T M v + b^t v + c, \\ \partial_v F(v) &= M v + b, \\ \partial_v F(0) &= b, \\ \partial_v^2 F(v) &= M. \end{aligned}$$

Clearly, $\partial_v F$ is the gradient, and $\partial_v^2 F$ is the Hessian. Observe that under the operation of the gradient, the vector b changes shape: from b^t to b .

In the code, the argument w corresponds to a moment, hence is a lying vector. We need some dummy symbols with respect to which to differentiate, and then we set the dummy variables to o in the gradient and the Hessian. For this second step, Sagemath uses substitution with a dictionary when multiple arguments are involved, which is the case here because w is a vector. So, by making a `zeros` dictionary that maps symbols to o , we can use the keys of `zeros` as the dummy symbols, and then use `zeros` itself in the substitution. Then, to solve for v such that $Mv = w^t - b$, the lying vector w has to be transposed.

```
1244 ..../sage/utils3.1.sage
1244 def Legendre_transform(F):
1245     def G(w):
1246         zeros = {var(f"v_{i}"): 0 for i in range(w.ncols())}
1247         b = gradient(F)(list(zeros.keys())).subs(zeros)
1248         M = Hessian(F)(list(zeros.keys())).subs(zeros)
1249         v = M.solve_right(w.T - b)
1250         return w * v - F(v)
1251
1252     return G
```

Now we are equipped to convert a Lagrangian into a Hamiltonian.

```
1253 ..../sage/utils3.1.sage
1253 def Lagrangian_to_Hamiltonian(Lagrangian):
1254     def f(H_state):
1255         t = time(H_state)
```

```

1256     q = coordinate(H_state)
1257     p = momentum(H_state)
1258
1259     def L(qdot):
1260         return Lagrangian(up(t, q, qdot))
1261
1262     return Legendre_transform(L)(p)
1263
1264 return f

```

```

.../sage/section3.1.sage
1265 res = Lagrangian_to_Hamiltonian(L_central_rectangular(m, V))(H_state)
1266 show(res)

```

$$\left[-\frac{1}{2}m\left(\frac{p_x^2}{m^2} + \frac{p_y^2}{m^2}\right) + \frac{p_x^2}{m} + \frac{p_y^2}{m} + V\left(\sqrt{q_x^2 + q_y^2}\right) \right]$$

```

.../sage/section3.1.sage
1267 show(res.simplify_full())

```

$$\left[\frac{2mV(\sqrt{q_x^2 + q_y^2}) + p_x^2 + p_y^2}{2m} \right]$$

```

.../sage/section3.1.sage
1268 var("m g l")
1269 q = column_path([literal_function("theta")])
1270 p = row_path([literal_function("p")])

```

Here is exercise 3.1.

```

.../sage/section3.1.sage
1271 # space = make_named_space(["\\theta"])
1272 H_state = qp_to_H_state_path(q, p)(t)
1273 show(Lagrangian_to_Hamiltonian(L_planar_pendulum(m, g, l))(H_state))

```

$$\left[-g l m (\cos(\theta) - 1) + \frac{p^2}{2l^2 m} \right]$$

```

.../sage/section3.1.sage
1274 q = column_path([literal_function("q_x"), literal_function("q_y")])
1275 p = row_path([literal_function("p_x"), literal_function("p_y")])
1276 H_state = qp_to_H_state_path(q, p)(t)
1277 show(Lagrangian_to_Hamiltonian(L_Henon_Heiles(m))(H_state))

```

$$\left[q_x^2 q_y - \frac{1}{3} q_y^3 + \frac{1}{2} p_x^2 + \frac{1}{2} p_y^2 + \frac{1}{2} q_x^2 + \frac{1}{2} q_y^2 \right]$$

```

1278 _____ ./.sage/section3.1.sage _____
1279 def L_sphere(m, R):
1280     def Lagrangian(local):
1281         theta, phi = coordinate(local).list()
1282         thetadot, phidot = velocity(local).list()
1283         return 1 / 2 * m * R ^ 2 * (
1284             square(thetadot) + square(phidot * sin(theta)))
1285
1286     return Lagrangian
1287
1288
1289 var("R", domain="positive")

```

```

1290 _____ ./.sage/section3.1.sage _____
1291 q = column_path([literal_function("theta"), literal_function("phi")])
1292 p = row_path([literal_function("p_x"), literal_function("p_y")])
1293 H_state = qp_to_H_state_path(q, p)(t)
1294 show(Lagrangian_to_Hamiltonian(L_sphere(m, R))(H_state).simplify_full())

```

$$\left[\frac{p_x^2 \sin(\theta)^2 + p_y^2}{2 R^2 m \sin(\theta)^2} \right]$$

3.2 POISSON BRACKETS

3.2.1 Standard imports

```

1294 _____ ./.sage/utils3.2.sage _____
load("utils3.1.sage")

_____ ./.sage/section3.2.sage _____
1295 load("utils3.2.sage")
1296
1297 t = var("t", domain="real")

_____ don't tangle _____
1298 load("show_expression.sage")

```

3.2.2 The Poisson Bracket

This is the Poisson bracket.

```

1299 @Func
1300 def Poisson_bracket(F, G):
1301     def f(state):

```

```

1302     left = (partial(F, 1) * compose(transpose, partial(G, 2)))(state)
1303     right = (partial(F, 2) * compose(transpose, partial(G, 1)))(state)
1304     return (left - right).simplify_full()
1305
1306 return f

```

We can make general state functions like so.

```

.../sage/utils3.2.sage
1307 @Func
1308 def state_function(name):
1309     return lambda H_state: function(name)(
1310         time(H_state), *coordinate(H_state).list(), *momentum(H_state).list()
1311     )

```

The first test is to see whether $\{Q, H\} = \partial_2 H$ and $\{P, H\} = -\partial_1 H$, where Q and P are the coordinate and momentum selectors, and H is a general state function.

```

.../sage/section3.2.sage
1312 q = column_matrix([var("q_x"), var("q_y")])
1313 p = row_matrix([var("p_x"), var("p_y")])
1314 sigma = up(t, q, p)
1315 H = state_function("H")
1316
1317 show(Poisson_bracket(coordinate, H)(sigma))
1318 show(Poisson_bracket(momentum, H)(sigma))

```

$$\begin{bmatrix} \frac{\partial}{\partial p_x} H(t, q_x, q_y, p_x, p_y) \\ \frac{\partial}{\partial p_y} H(t, q_x, q_y, p_x, p_y) \end{bmatrix}$$

$$\begin{bmatrix} -\frac{\partial}{\partial q_x} H(t, q_x, q_y, p_x, p_y) \\ -\frac{\partial}{\partial q_y} H(t, q_x, q_y, p_x, p_y) \end{bmatrix}$$

All is correct. Note that both results are standing vectors.

3.2.3 Properties of the Poisson bracket

We know that $\{H, H\} = 0$ for any function. Let's test this for our implementation.

```

.../sage/section3.2.sage
1319 show(Poisson_bracket(H, H)(sigma))

```

$$[0]$$

The property $\{F, F\} = 0$ is actually implied when we can show that the Poisson bracket is anti-symmetric.

```
1320 F = state_function("F")
1321 G = state_function("G")
1322
1323 show((Poisson_bracket(F, G) + Poisson_bracket(G, F))(sigma))
```

$$[0]$$

How about $\{F, G + H\} = \{F, G\} + \{F, H\}$?

```
1324 show(
1325     (
1326         Poisson_bracket(F, G + H)
1327         - Poisson_bracket(F, G)
1328         - Poisson_bracket(F, H)
1329     )(sigma)
1330 )
```

$$[0]$$

To check the rule $\{F, cG\} = c\{F, G\}$ we need a constant function. By making the next function independent of any argument, it becomes constant.

```
1331 constant = Function(lambda H_state: function("c")())
```

Is it indeed constant?

```
1332 show(Jacobian(constant)(sigma, sigma))
```

$$[0 \ 0 \ 0 \ 0 \ 0]$$

So, next we can check $\{F, cG\} = c\{F, G\}$.

```
1333 show(
1334     (Poisson_bracket(F, constant * G) - constant * Poisson_bracket(F, G))(
1335         sigma
1336     ).simplify_full()
1337 )
```

$$[0]$$

Finally, here is the check on Jacobi's identity.

```

1338 _____ ./.sage/section3.2.sage _____
1339 jacobi = (
1340     Poisson_bracket(F, Poisson_bracket(G, H))
1341     + Poisson_bracket(G, Poisson_bracket(H, F))
1342     + Poisson_bracket(H, Poisson_bracket(F, G)))
1343 )
1344 show(jacobi(sigma).simplify_full())

```

[0]

3.2.4 Poisson bracket of a conserved quantity

To check that the Poisson bracket of a conserved quantity is conserved we need a function that does not depend on time.

```

1345 _____ ./.sage/section3.2.sage _____
1346 def f(H_state):
1347     return function("f")(
1348         *coordinate(H_state).list(), *momentum(H_state).list()
1349     )

```

Clearly, the derivative with respect to time of this function is zero, so it does what we need.

```

1349 _____ ./.sage/section3.2.sage _____
show(diff(f(sigma), time(sigma)))

```

0

Now consider $\{F, H\}$ where H is the rectangular Hamiltonian.

```

1350 _____ ./.sage/section3.2.sage _____
V = Function(lambda q: function("V")(*q.list()))
1351
1352 var(m, domain="positive")
1353
1354 H = H_rectangular(m, V)

```

I compute the Poisson bracket of F and H for one dimension so that the result remains small.

```

1355 _____ ./.sage/section3.2.sage _____
q = column_matrix([var("q")])
1356 p = row_matrix([var("p")])
1357 sigma = up(t, q, p)
1358 show(Poisson_bracket(f, H)(sigma).expand())

```

$$\left[-\frac{\partial}{\partial q} V(q) \frac{\partial}{\partial p} f(q, p) + \frac{p \frac{\partial}{\partial q} f(q, p)}{m} \right]$$

To complete the check, note that, by Hamilton's equation, $\dot{q} = \partial H / \partial p$, $\dot{p} = -\partial H / \partial q = -\partial V / \partial q$. If we replace that in the above equation we obtain

$$\dot{p} \frac{\partial f}{\partial p} + \dot{q} \frac{\partial f}{\partial q} = \frac{df}{dt}.$$

Since f is conserved, the total time derivative of F is zero, hence f and H commute.

3.4 PHASE SPACE REDUCTION

3.4.1 Standard imports

```
1360 load("utils3.1.sage") .../sage/section3.4.sage
1361 t = var("t", domain="real")
1362
1363 load("show_expression.sage")
```

3.4.2 Motion in a central potential

```
1364 var("m")
1365 V = function("V")
1366
1367
1368 def L_polar(m, V):
1369     def Lagrangian(local):
1370         r, phi = coordinate(local).list()
1371         rdot, phidot = velocity(local).list()
1372         T = 1 / 2 * m * (square(rdot) + square(r * phidot))
1373         return T - V(r)
1374
1375     return Lagrangian
```

```
1377 q = column_path([literal_function("r"), literal_function("phi")])
1378 p = row_path([literal_function("p_r"), literal_function("p_phi")])
1379 H_state = qp_to_H_state_path(q, p)(t)
1380 show(H_state)
```

$$\begin{bmatrix} t \\ r \\ \phi \\ [p_r \ p_\phi] \end{bmatrix}$$

```
1381 _____ ./.sage/section3.4.sage _____
1382 H = Lagrangian_to_Hamiltonian(L_polar(m, V))
1383 show(H(H_state).simplify_full())
```

$$\left[\frac{2mV(r)r^2 + p_r^2 r^2 + p_\phi^2}{2mr^2} \right]$$

Here are the Hamilton equations.

```
1383 _____ ./.sage/section3.4.sage _____
1384 HE = Hamilton_equations(Lagrangian_to_Hamiltonian(L_polar(m, V)))(q, p)(t)
1385 show(HE)
```

$$\begin{bmatrix} 0 \\ -\frac{p_r}{m} + \dot{r} \\ -\frac{p_\phi}{mr^2} + \dot{\phi} \\ -\frac{p_\phi^2}{mr^3} + D_0(V)(r) + \dot{p}_r \quad \dot{p}_\phi \end{bmatrix}$$

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.

```
1385 _____ ./.sage/section3.5.sage _____
1386 import numpy as np
1387
1388 load("utils1.7.sage", "utils3.1.sage")
1389 var("t", domain="real")
```

```
1390 _____ don't tangle _____
1391 load("show_expression.sage")
```

3.5.2 The Hamiltonian for the driven pendulum

```
1391 _____ ./.sage/section3.5.sage _____
1392 q = column_path([literal_function("theta")])
1393 p = row_path([literal_function(r"p_theta")])
1394 H_state = qp_to_H_state_path(q, p)(t)
1395 show(H_state)
```

$$\begin{bmatrix} t \\ \theta \\ p_\theta \end{bmatrix}$$

This is the Hamiltonian. The computations return a (1×1) matrix; we therefore unpack it.

```
_____ ..../sage/section3.5.sage _____
1395 _ = var("A g l m omega", domain="positive")
1396
1397 H = Lagrangian_to_Hamiltonian(
1398     L_periodically_driven_pendulum(m, l, g, A, omega)
1399 )
1400 show(H(H_state)[0,0].simplify_full().expand())
```

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

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

```
_____ ..../sage/section3.5.sage _____
1401 DH = Hamiltonian_to_state_derivative(H)(H_state)
1402 show(DH[1].simplify_full()[0,0])
1403 show(DH[2].simplify_full()[0,0])
```

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

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

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

```
_____ ..../sage/section3.5.sage _____
1404 def H_pend_sysder(m, l, g, A, omega):
1405     Hamiltonian = Lagrangian_to_Hamiltonian(
1406         L_periodically_driven_pendulum(m, l, g, A, omega)
1407     )
1408
1409     def f(state):
1410         return Hamiltonian_to_state_derivative(Hamiltonian)(state)
1411
1412     return f
```

```
_____ ..../sage/section3.5.sage _____
1413 times = strange(0, 100, 0.001, include_endpoint=True)
1414 soln = evolve(
1415     H_pend_sysder(m=1, l=1, g=9.8, A=0.1, omega=2 * sqrt(9.8)),
1416     ics=up(0, column_matrix([1]), row_matrix([0])),
```

```

1417     times=times,
1418 )
1419 thetas = principal_value(np.pi)(soln[:, 1])
1420 thetadots = soln[:, 2]
1421 pp = list(zip(thetas, thetadots))
1422 p = points(pp, color='blue', size=3)
1423 p.save(f'../../figures/hamiltonian_driven_pendulum_0.001.png')

```

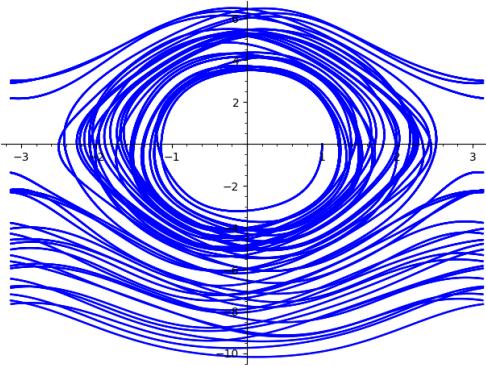


Figure 3.1: The driven pendulum obtained from numerically integrating the Hamilton equations. The graph is not identical to the one in the book, because of the inherent chaotic behavior.

```

.../sage/section3.5.sage
1424 times = strange(0, 100, 0.005, include_endpoint=True)
1425 soln = evolve(
1426     H_pend_sysder(m=1, l=1, g=9.8, A=0.1, omega=2 * sqrt(9.8)),
1427     ics=up(0, vector([1]), vector([0])),
1428     times=times,
1429 )
1430 thetas = principal_value(np.pi)(soln[:, 1])
1431 thetadots = soln[:, 2]
1432 pp = list(zip(thetas, thetadots))
1433 p = points(pp, color='blue', size=3)
1434 p.save(f'../../figures/hamiltonian_driven_pendulum_0.005.png')

```

3.9 THE STANDARD MAP

We take a number of uniformly distributed starting points for the paths. The result, shown in Fig. 3.3, is very nice.

```

1435 import numpy as np
1436 import matplotlib.pyplot as plt
1437

```

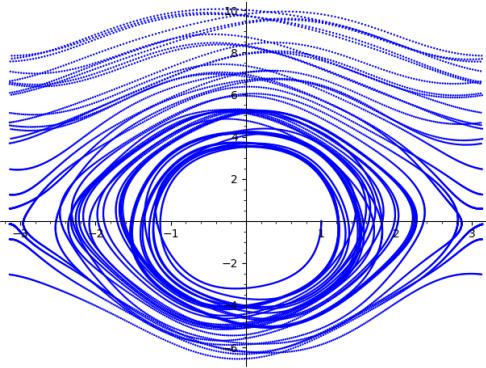


Figure 3.2: The driven pendulum obtained from numerically integrating the Hamilton equations, but now with time steps 0.005 instead of 0.001. The graphs for both step-sizes seem to be qualitatively the same, but the details are different.

```

1438
1439 n_points = 10000
1440 I = np.zeros(n_points)
1441 theta = np.zeros(n_points)
1442
1443 K = 0.9
1444 two_pi = 2 * np.pi
1445
1446 plt.figure(figsize=(10, 10), dpi=300)
1447
1448 for _ in range(500):
1449     theta[0] = np.random.uniform(0, two_pi)
1450     I[0] = np.random.uniform(0, two_pi)
1451     for i in range(1, n_points):
1452         I[i] = (I[i - 1] + K * np.sin(theta[i - 1])) % two_pi
1453         theta[i] = (theta[i - 1] + I[i]) % two_pi
1454         plt.scatter(theta, I, s=0.01, color='black', alpha=0.1, marker='.')
1455
1456
1457 plt.axis('off')
1458 plt.savefig('../figures/standard_map.png', bbox_inches='tight')
```

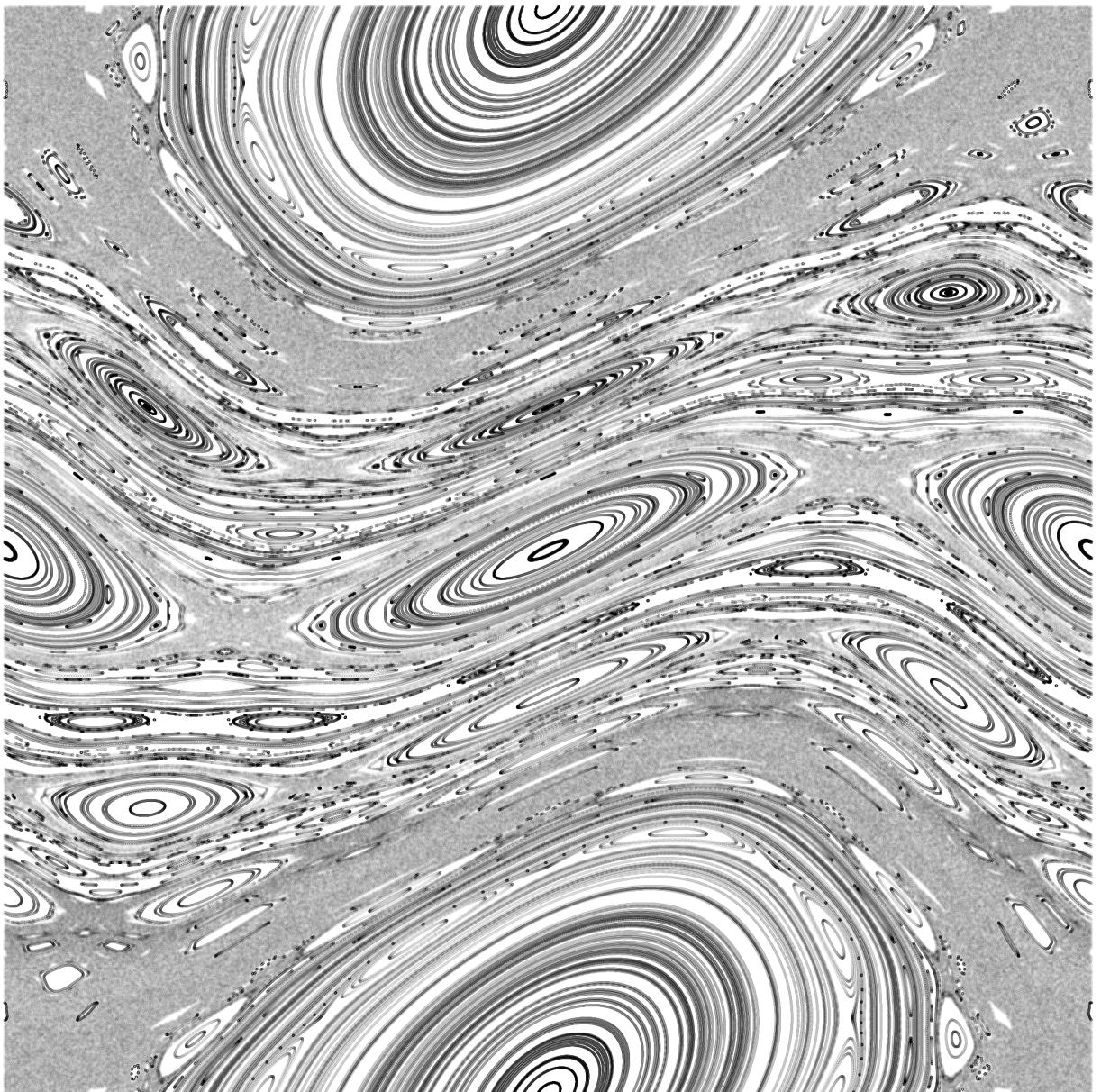


Figure 3.3: The standard map with $K = 0.6$.

4

CHAPTER 4: SKIPPED

CHAPTER 5

5.1 POINT TRANSFORMATIONS

5.1.1 *The standard imports.*

```
1459      ...../sage/utils5.1.sage
load("utils3.1.sage")
```



```
1460      ...../sage/section5.1.sage
load("utils5.1.sage")
```



```
1461      t = var("t", domain="real")
```



```
1463      .....don't tangle
load("show_expression.sage")
```

5.1.2 *Implementing point transformations*

```
...../sage/utils5.1.sage
1464 def F_to_CH(F):
1465     M = partial(F, 1)
1466
1467     def f(state):
1468         return up(time(state), F(state), M(state).solve_left(momentum(state)))
1469
1470     return f
```

Let's test this function.

```
...../sage/section5.1.sage
1471 var("r, phi, p_r, p_phi", domain="real")
1472 assume(r > 0)
1473
1474 q = column_matrix([r, phi])
1475 p = row_matrix([p_r, p_phi])
1476 state = up(t, q, p)
```



```
...../sage/section5.1.sage
1477 show((F_to_CH(p_to_r))(state)[0].simplify_full())
1478 show((F_to_CH(p_to_r))(state)[1].simplify_full())
1479 show((F_to_CH(p_to_r))(state)[2].simplify_full())
```

t

$$\begin{bmatrix} r \cos(\phi) \\ r \sin(\phi) \end{bmatrix}$$

$$\begin{bmatrix} p_r r \cos(\phi) - p_\phi \sin(\phi) \\ p_r r \sin(\phi) + p_\phi \cos(\phi) \end{bmatrix}$$

The central Hamiltonian in rectangular coordinates.

```
1480 def H_central(m, V):
1481     def f(state):
1482         x, p = coordinate(state), momentum(state)
1483         return square(p) / (2 * m) + V(sqrt(square(x)))
1484
1485     return f
1486
1487 var("m", domain="positive")
```

Now we convert it to polar coordinates.

```
1488 show(
1489     compose(H_central(m, function("V")), (F_to_CH(p_to_r)))(state)
1490     .simplify_full()
1491     .expand()
1492 )
```

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

The correction term for time dependent Hamiltonians.

```
1493 def F_to_K(F):
1494     M = partial(F, 1)
1495
1496     def f(state):
1497         p = M(state).solve_left(momentum(state))
1498         return -p * partial(F, 0)(state)
1499
1500     return f
```

We apply this to a 2D translation.

```
1501 def translating(v):
1502     def f(state):
1503         return coordinate(state) + v * time(state)
1504
1505     return f
```

```

1506 var("q_x q_y v_x v_y p_x p_y", domain="real")
1507 q = column_matrix([q_x, q_y])
1508 v = column_matrix([v_x, v_y])
1509 p = row_matrix([p_x, p_y])
1510 state = up(t, q, p)

```

```

1511 show(F_to_K(translating(v))(state)[0, 0])

```

$$-p_x v_x - p_y v_y$$

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

```

1512 def H_free(m):
1513     def f(state):
1514         return square(momentum(state)) / (2 * m)
1515
1516     return f
1517
1518
1519 def H_prime():
1520     return compose(H_free(m), F_to_CH(translating(v))) + F_to_K(translating(v))
1521
1522
1523 show(H_prime()(state)[0, 0])

```

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

5.2 GENERAL CANONICAL TRANSFORMATIONS

I found the analysis in the first part of Section 5.2 somewhat strange: why do all this coding when it turns out that it's easiest to express the canonical condition in terms of a matrix equation? I therefore skip this first part, and move to symplectic matrices straightforwardly.

5.2.1 *The standard imports.*

```

1525 load("utils5.1.sage")

```

```

1526   _____ ./.sage/section5.2.sage _____
1527   load("utils5.2.sage")
1528   t = var("t", domain="real")
1529   _____ don't tangle _____
load("show_expression.sage")

```

5.2.2 Symplectic matrices

We start with building the symplectic unit.

```

1530   _____ ./.sage/utils5.2.sage _____
def symplectic_unit(n):
    I = identity_matrix(n)
    return block_matrix([[zero_matrix(n), I], [-I, zero_matrix(n)]])
1533   _____ ./.sage/section5.2.sage _____
show(symplectic_unit(2))

```

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

Now we make the test on whether a matrix is symplectic or not. Two remarks:

1. Sometimes Sagemath seems to miss that $\sqrt{x}/\sqrt{x} = 1$, even after adding the assumption that x is positive. This turned out to be problematic when running this test on examples below. It turned out that expanding the matrix M resolved this type of problem.
2. I like to see the output in case the transformation is not symplectic, hence the print statement at the end.

```

1534   _____ ./.sage/utils5.2.sage _____
1535   def is_symplectic_matrix(M):
1536       n = M.nrows()
1537       J = symplectic_unit(n // 2)
1538       if isinstance(M, sage.matrix.matrix_symbolic_dense.Matrix_symbolic_dense):
1539           M = M.expand()
1540           res = (M * J * M.transpose()).simplify_full()
1541       else:
1542           res = M * J * M.transpose()
1543       if res == J:
1544           return True
1545           print(res - J)
1546       return False

```

We can test this code on a symplectic unit J . Let's fill in $J = A$ in the relation $J = AJA^T$. This then becomes $JJJ^T = JJJ^{-1} = J$, since $J^{-1} = J^T$. We conclude that J is a symplectic matrix itself.

```
1547 J = symplectic_unit(2)
1548 show(is_symplectic_matrix(J))
```

The next important function is the test on whether a transformation is symplectic, which depends on a few other functions that we will build below.

```
1549 def is_symplectic_transform(C):
1550     return lambda state: compose(
1551         is_symplectic_matrix, qp_submatrix, D_as_matrix(C)
1552     )(state)
```

The function `qp_submatrix` is easy.

```
1553 def qp_submatrix(M):
1554     return M[1:, 1:]
```

The function `D_as_matrix` requires some more work. It takes a phase-space transformation C as argument, that is, a map from $up(t, q', p')$ to $up(t, q, p)$, and returns a function that operates on the state $up(t, q', p')$ to produce the derivative DC in matrix form. Thus, we need a function to convert tuples to vectors, and another function that lets C operate on vectors instead of tuples. Once we have this, we can apply Sagemath's `jacobian` to compute the derivative of the functions `vmap(vec)` with respect to the vectorized `up`-tuple `state`.

```
1555 def D_as_matrix(C):
1556     def f(state):
1557         vec = up_to_vector(state)
1558         vmap = to_vector_map(C)
1559         #return jacobian(vmap(vec), vec).simplify_full()
1560         return jacobian(vmap(state), vec).simplify_full()
1561
1562     return f
```

In Sagemath we work with matrices and vectors instead of `up` and `down` tuples. Note that we give up on whether coordinates and momenta are standing or lying vectors. Everything becomes just a vector. This is of course a consequence of the use of symplectic matrices.

```
1563 def up_to_vector(state):
1564     return vector(
1565         [time(state), *coordinate(state).list(), *momentum(state).list()])
1566     )
```

Here is a test.

```
1567 var("r, phi, p_r, p_phi", domain="real")
1568 assume(r > 0)
1569 r_phi = up(t, column_matrix([r, phi]), row_matrix([p_r, p_phi]))
```

The functions should be each others inverse.

```
1571 vec = up_to_vector(r_phi)
1572 show(vec)
```

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

Point transformations can be used as canonical transformations. So this case forms a more serious test.

```
1573 show(up_to_vector(F_to_CH(p_to_r)(r_phi)).simplify_full())
```

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

The other function is `to_vector_map(C)`.

```
1574 def to_vector_map(C):
1575     return compose(up_to_vector, C)
```

```
1576 show(to_vector_map(F_to_CH(p_to_r))(r_phi).simplify_full())
```

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

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

```
1577 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.

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

In Sagemath this becomes:

```
1583 def F(local):
1584     t, q = time(local), coordinate(local)
1585     return vector([function("f")(t, *q), function("g")(t, *q)])
```

The next check takes some time to complete.

```
1586 _ = var("y p_x p_y", domain="real")
1587 xy = up(t, vector([x, y]), vector([p_x, p_y]))
1588 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.

```
1590 def polar_canonical(alpha):
1591     def f(state):
1592         t = time(state)
1593         theta = coordinate(state)[0]
1594         I = momentum(state)[0]
1595         x = sqrt(2 * I / alpha) * sin(theta)
1596         p = sqrt(2 * I * alpha) * cos(theta)
1597         return up(t, vector([x]), vector([p]))
1598
1599     return f
1600
1601
1602 _ = var("theta", domain="real")
1603 _ = var("I alpha", domain="positive")
1604 theta_I = up(t, vector([theta]), vector([I]))
1605 show(is_symplectic_transform(polar_canonical(alpha))(theta_I))
```

True

This is a *non-canonical* transformation.

```
1606 _____ . /sage/section5.2.sage _____
1607 def a_non_canonical_transform(state):
1608     t = time(state)
1609     theta = coordinate(state)[0]
1610     I = momentum(state)[0]
1611     x = I * sin(theta)
1612     p = I * cos(theta)
1613     return up(t, vector([x]), vector([p]))
1614
1615 show(is_symplectic_transform(a_non_canonical_transform)(theta_I))
```

[0 I - 1] [-I + 1 0]
False

Here I is a momentum, not the identity matrix.

6

CHAPTER 6

6.4 LIE SERIES

6.4.1 The standard imports.

```
1616      ..... /sage/utils6.4.sage
load("utils3.2.sage")  
  
1617      ..... /sage/section6.4.sage
load("utils6.4.sage")  
1618  
1619      var("t", domain="real")  
  
1620      ..... don't tangle
load("show_expression.sage")
```

6.4.2 Taylor expansions

We follow the examples of the book on the Taylor series.

```
1621      ..... /sage/section6.4.sage
f = function("f")
show(taylor(f(x), x, 0, 4))  
  
1623      ..... /sage/section6.4.sage
f = sin
show(taylor(f(x), x, 0, 6))
```

$$\frac{1}{24}x^4 D_{0,0,0,0}(f)(0) + \frac{1}{6}x^3 D_{0,0,0}(f)(0) + \frac{1}{2}x^2 D_{0,0}(f)(0) + x D_0(f)(0) + f(0)$$

$$\frac{1}{120}x^5 - \frac{1}{6}x^3 + x$$

Calling `taylor` on `f(x=x)` is wrong because `f(x=x)` tries to evaluate `f` at `x = x`. Instead, `f` should be treated as a symbolic function.

```
1625      ..... don't tangle
taylor(f(x=x), x, 0, 6) # does not work
```

When using `lambda`, we should put it in brackets and apply it to a variable like `x`. Therefore the first line does not work.

```
1626  _____ ..../sage/section6.4.sage _____
1627 # show(taylor(lambda x: sqrt(1 + x), x, 0, 6)) # this does not work
1627 show(taylor((lambda x: sqrt(1 + x))(x), x, 0, 6))
```

$$-\frac{21}{1024}x^6 + \frac{7}{256}x^5 - \frac{5}{128}x^4 + \frac{1}{16}x^3 - \frac{1}{8}x^2 + \frac{1}{2}x + 1$$

Here is Exercise 6.7

```
1628  _____ ..../sage/section6.4.sage _____
1629 n = var("n", domain="integer")
1629 f = lambda x: (1 + x)^n
1630 show(taylor(f(x), x, 0, 7))
```

$$\begin{aligned} & \frac{1}{5040} (n^7 - 21n^6 + 175n^5 - 735n^4 + 1624n^3 - 1764n^2 + 720n) x^7 \\ & + \frac{1}{720} (n^6 - 15n^5 + 85n^4 - 225n^3 + 274n^2 - 120n) x^6 \\ & + \frac{1}{120} (n^5 - 10n^4 + 35n^3 - 50n^2 + 24n) x^5 + \frac{1}{24} (n^4 - 6n^3 + 11n^2 - 6n) x^4 \\ & + \frac{1}{6} (n^3 - 3n^2 + 2n) x^3 + \frac{1}{2} (n^2 - n) x^2 + nx + 1 \end{aligned}$$

6.4.3 Computing Lie Series

The Lie derivative is straightforward to build. We decorate it with `@Func` so that we can use it in algebraic equations.

```
1631 @Func
1632 def Lie_derivative(H):
1633     def f(F):
1634         return Poisson_bracket(F, H)
1635
1636     return f
```

Before tring our hands on the Lie transform, we consider a few examples of the Lie derivative.

```
1637  _____ ..../sage/section6.4.sage _____
1638 _ = var("m k", domain="positive")
1639
1640 def H_harmonic(m, k):
1641     def f(state):
```

```

1642     return (
1643         square(momentum(state)) / (2 * m)
1644         + k * square(coordinate(state)) / 2
1645     )
1646
1647     return f
1648
1649 H = H_harmonic(m, k)

```

```

.../sage/section6.4.sage
1650 x0, p0 = var("x0 p0", domain="real")
1651 state = up(t, column_matrix([x0]), row_matrix([p0]))
1652 lie = Lie_derivative(H)
1653 show(lie(coordinate)(state))
1654 show(lie(lie(coordinate))(state))

```

$$\left[\frac{p_0}{m} \right]$$

$$\left[-\frac{kx_0}{m} \right]$$

We move on to the Lie transform. The transform function consumes as arguments a Hamiltonian (for the Lie derivative) and a time t . The `returnn` function, `outer`, asks for the function F to which to apply the Lie transform. The nested function `inner` requires a state and the order up to which to evaluate first the Lie derivative. (As the book only uses the series up to some order, there is no need to compute $e^{\epsilon L_H} F$ first and then take the series approximation for that exponential.)

```

.../sage/utils6.4.sage
1655 def Lie_transform(H, t):
1656     lie = Lie_derivative(H)
1657
1658     def outer(func):
1659         def inner(local, n):
1660             term = func
1661             factor = 1
1662             res = factor * term(local)
1663             for i in range(1, n + 1):
1664                 term = lie(term)
1665                 factor *= t / i
1666                 res += factor * term(local)
1667             return res
1668
1669         return inner
1670
1671     return outer

```

We can now move on to the examples of the book.

```

1672 _ = var('dt', domain="real", latex_name=r"\mathrm{d}\ t")
1673 show(Lie_transform(H, dt)(coordinate)(state, 4))
1674 show(Lie_transform(H, dt)(momentum)(state, 4))
1675 show(Lie_transform(H, dt)(H)(state, 4))

```

$$\left[\frac{\mathrm{d}t^4 k^2 x_0}{24m^2} - \frac{\mathrm{d}t^3 k p_0}{6m^2} - \frac{\mathrm{d}t^2 k x_0}{2m} + \frac{\mathrm{d}t p_0}{m} + x_0 \right]$$

$$\left[\frac{\mathrm{d}t^4 k^2 p_0}{24m^2} + \frac{\mathrm{d}t^3 k^2 x_0}{6m} - \frac{\mathrm{d}t^2 k p_0}{2m} - \mathrm{d}t k x_0 + p_0 \right]$$

$$\left[\frac{1}{2} k x_0^2 + \frac{p_0^2}{2m} \right]$$

The final example is the Hamiltonian for a central potential field, formulated in polar coordinates. We can build this Hamiltonian from our earlies work, like so.

```

1677 def V(q):
1678     return function("U")(q)
1679
1680 m = var("m", domain="positive")
1681
1682 def H_central_polar(m, V):
1683     def f(state):
1684         r, phi = coordinate(state).list()
1685         p_r, p_phi = momentum(state).list()
1686         T = 1 / 2 * square(p_r) / m + 1 / 2 * square(p_phi) / (m * square(r))
1687         return T + V(r)
1688
1689     return f
1690
1691
1692 H = H_central_polar(m, V)

```

First two elementary checks on our code.

```

1693 _ = var("r phi p_r p_phi", domain="real")
1694 assume(r > 0)
1695 q = column_matrix([r, phi])
1696 p = row_matrix([p_r, p_phi])
1697 H_state = up(t, q, p)
1698
1699 show(H(H_state).expand())
1700 show(partial(H, 1)(H_state))

```

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

$$\left[\begin{array}{cc} -\frac{p_\phi^2}{mr^3} + \frac{\partial}{\partial r}U(r) & 0 \end{array} \right]$$

Here is the result; it's the same as in the book. We unpack the matrix to remove the brackets in the printing.

```
1701 _____ ./.sage/section6.4.sage -
1702 res = Lie_transform(H, dt)(coordinate)(H_state, 3).expand()
1703 show(res[0][0])
show(res[1][0])
```

$$\begin{aligned} & -\frac{dt^3 p_r \frac{\partial^2}{(\partial r)^2} U(r)}{6m^2} - \frac{dt^2 \frac{\partial}{\partial r} U(r)}{2m} + \frac{dt p_r}{m} + r - \frac{dt^3 p_\phi^2 p_r}{2m^3 r^4} + \frac{dt^2 p_\phi^2}{2m^2 r^3} \\ & \phi + \frac{dt^3 p_\phi \frac{\partial}{\partial r} U(r)}{3m^2 r^3} + \frac{dt^3 p_\phi p_r^2}{m^3 r^4} - \frac{dt^2 p_\phi p_r}{m^2 r^3} + \frac{dt p_\phi}{mr^2} - \frac{dt^3 p_\phi^3}{3m^3 r^6} \end{aligned}$$

7

CHAPTER 7
