Structure and Interpretation of Classical Mechanics with Python and Sagemath

Nicky van Foreest

May 23, 2025

CONTENTS

0 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 all worked, I copied the code to an Org file and make code blocks. Then I tangled, for instance, generally useful code of `secton1.4.org` to `utils1.4.sage` and to `section1.4.sage` for code specific for Section 1.4 of the book. This way I can load the utils files at later stages.

I found it convenient to test things in a `tests.sage` file. Then, I could edit 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 todo 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.

../sage/show_expression.sage

```
import re

latex.matrix_delimiters(left='[', right=']')
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.

../sage/show_expression.sage

```
def simplify_latex(s):
    s = re.sub(r"\\frac{\\partial}{\\partial t}", r"\\dot ", s)
    s = re.sub(r"\\left\(t\\right\)", r"", s)
    s = re.sub(
        r"\\frac\{\\partial\^\{2\}\}\{\(\\partial t\)\^\{2\}\}",
        r"\\ddot ",
        s,
    )
    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 LaTeX 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.

../sage/show_expression.sage
```
def show_expression(s, simplify=True):
    s = latex(s)
    if simplify:
        s = simplify_latex(s)
    res = r"\begin{dmath*}"
    res += "\n" + s + "\n"
    res += r"\end{dmath*}"
    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 LaTeX output. However, when executing a code block in org mode, I *do* want to get LaTeX 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.

../sage/show_expression.sage
```
def show(s, simplify=True):
    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.

../sage/tuples.sage
```
"""
This is a copy of tuples.py from
https://github.com/jtauber/functional-differential-geometry.
"""

from sage.structure.element import Matrix, Vector

class Tuple:
    def __init__(self, *components):
        self._components = components

    def __getitem__(self, index):
        return self._components[index]

    def __len__(self):
        return len(self._components)

    def __eq__(self, other):
        if (
                isinstance(other, self.__class)
                and self._components == other._components
        ):
            return True
        else:
            return False

    def __ne__(self, other):
        return not (self.__eq__(other))

    def __add__(self, other):
        if isinstance(self, Tuple):
            if not isinstance(other, self.__class__) or len(self) != len(
                    other
            ):
                raise TypeError("can't add incompatible Tuples")
            else:
                return self.__class__(
                    *(
                        s + o
                        for (s, o) in zip(self._components, other._components)
                    )
                )
        else:
            return self + other

    def __iadd__(self, other):
        return self + other

    def __neg__(self):
        return self.__class__(*(-s for s in self._components))

    def __sub__(self, other):
        return self + (-other)

    def __isub__(self, other):
        return self - other

    def __call__(self, **kwargs):
        return self.__class__(
            *(
                (c(**kwargs) if isinstance(c, Expr) else c)
                for c in self._components
            )
        )
```

```python

    def subs(self, args):
        # substitute variables with args
        return self.__class__(*(c.subs(args) for c in self._components))

    def list(self):
        "convert tuple and its components to one list."
        result = []
        for comp in self._components:
            if isinstance(comp, (Tuple, Matrix, Vector)):
                result.extend(comp.list())
            else:
                result.append(comp)
        return result

    def derivative(self, var):
        "Compute the derivative of all components and put the result in a tuple."
        return self.__class__(
            *[derivative(comp, var) for comp in self._components]
        )
```

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

../sage/tuples.sage

```python
class UpTuple(Tuple):
    def __repr__(self):
        return "up({})".format(", ".join(str(c) for c in self._components))

    def _latex_(self):
        "Print up tuples vertically."
        res = r"\begin{array}{c}"
        for comp in self._components:
            res += r"\begin{array}{c}"
            res += latex(comp)
            res += r"\end{array}"
            res += r" \\"
        res += r"\end{array}"
        return res

class DownTuple(Tuple):
    def __repr__(self):
        return "down({})".format(", ".join(str(c) for c in self._components))

    def _latex_(self):
        "Print down tuples horizontally."
        res = r"\begin{array}{c}"
        for comp in self._components:
            res += r"\begin{array}{c}"
            res += latex(comp)
            res += r"\end{array}"
            res += r" & "
        res += r"\end{array}"
        return res

up = UpTuple
down = DownTuple

up._dual = down
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 donwloaded from the said github repo.

../sage/tuples.sage
```
def ref(tup, *indices):
    if indices:
        return ref(tup[indices[0]], *indices[1:])
    else:
        return tup


def component(*indices):
    def _(tup):
        return ref(tup, *indices)

    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*

../sage/functions.sage
```
load("tuples.sage")
```

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

../sage/functions_tests.sage
```
load("functions.sage")
```

We load `show_expression` to control the LaTeX output in this org file.

don't tangle

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

0.4.2 *The* `Function` *class*

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

../sage/functions.sage

```
class Function:
    def __init__(self, func):
        self._func = func

    def __call__(self, *args):
        return self._func(*args)

    def __add__(self, other):
        return Function(lambda *args: self(*args) + other(*args))

    def __neg__(self):
        return Function(lambda *args: -self(*args))

    def __sub__(self, other):
        return self + (-other)

    def __mul__(self, other):
        if isinstance(other, Function):
            return Function(lambda *args: self(*args) * other(*args))
        return Function(lambda *args: other * self(*args))

    def __rmul__(self, other):
        return self * other

    def __pow__(self, exponent):
        if exponent == 0:
            return Function(lambda x: 1)
        else:
            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`.

../sage/functions.sage

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

    return wrapper
```

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

../sage/functions.sage
```
@Func
def compose(*funcs):
    if len(funcs) == 1:
        return lambda x: funcs[0](x)
    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.

../sage/functions_tests.sage
```
def f(x):
    return 5 * x


F = Function(lambda x: f(x))
```

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

../sage/functions.sage
```
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.

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

We will use quadratic functions often.

../sage/functions.sage
```
from functools import singledispatch


@singledispatch
def _square(x):
    raise TypeError(f"Unsupported type: {type(x)}")


@_square.register(int)
@_square.register(float)
@_square.register(Expression)
@_square.register(Integer)
def _(x):
    return x ^ 2


```

```
@_square.register(Vector)
@_square.register(list)
@_square.register(tuple)
def _(x):
    v = vector(x)
    return v.dot_product(v)


@_square.register(Matrix)
def _(x):
    if x.ncols() == 1:
        return (x.T * x)[0, 0]
    elif x.nrows() == 1:
        return (x * x.T)[0, 0]
    else:
        raise TypeError(
            f"Matrix must be a row or column vector, got shape {x.nrows()}×{x.ncols()}"
        )


square = Function(lambda x: _square(x))
```

To use Sagemath functions we make an abbreviation.

../sage/functions.sage

```
function = sage.symbolic.function_factory.function
```

Now we can make symbolic functions like so.

../sage/functions_tests.sage

```
V = Function(lambda x: function("V")(x))
```

0.4.4 *Examples*

../sage/functions_tests.sage

```
x, y = var("x y", domain = RR)

show((square)(x + y).expand())
```

$$x^2 + 2\,xy + y^2$$

../sage/functions_tests.sage

```
show((square + square)(x + y))
```

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

../sage/functions_tests.sage

```
show((square * square)(x))
```

$$x^4$$

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

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

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

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

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

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

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

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

$$0$$

Next, we test differentiation and integration.

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

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

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

Arithmetic with symbolic functions works too.

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

../sage/functions_tests.sage
```
show((U + V)(x))
show((V + V)(x))
show((V(U(x))))
show((compose(V, U)(x)))
```

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

$$2\,V(x)$$

$$V(U(x))$$

$$V(U(x))$$

../sage/functions_tests.sage

```
def f(x):
    def g(y):
        return x * y ^ 2

    return g
```

../sage/functions_tests.sage

```
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'.

don't tangle

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

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

../sage/functions_tests.sage

```
@Func
def f(x):
    def g(y):
        return x * y ^ 2

    return g
```

../sage/functions_tests.sage

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

$$80$$

Indeed: $(3+2) * 4^2 = 80$.
Decorating with `@Func` is the same as this.

../sage/functions_tests.sage

```
def f(x):
    def g(y):
        return x * y ^ 2

    return Function(lambda y: g(y))
```

../sage/functions_tests.sage

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

$$80$$

0.5 DIFFERENTIATION

0.5.1 *Standard imports*

../sage/differentiation.sage

```
load(
    "functions.sage",
    "tuples.sage",
)
```

../sage/differentiation_tests.sage

```
load("differentiation.sage")

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

don't tangle

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

0.5.2 *Examples with matrices, functions and tuples*

../sage/differentiation_tests.sage

```
_ = var("a b c x y", domain=RR)
M = matrix([[a, b], [b, c]])
b = vector([a, b])
v = vector([x, y])
F = 1 / 2 * v * M * v + b * v + c
```

../sage/differentiation_tests.sage

```
show(F)
```

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

../sage/differentiation_tests.sage

```
show(F.expand())
```

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

../sage/differentiation_tests.sage

```
show(diff(F, x))
```

$$ax + by + a$$

Repeated differentiation works nicely.

../sage/differentiation_tests.sage

```
show(diff(F, [x, y]))
```

$$b$$

This is the Jacobian.

../sage/differentiation_tests.sage

```
show(jacobian(F, [x, y]))
```

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

../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.

don't tangle

```
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.

../sage/differentiation_tests.sage

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

../sage/differentiation_tests.sage

```
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.

don't tangle

```
jacobian(F, v) # F has no arguments
jacobian(F(v), v) # v is not a list
```

The `Tuple` class supports differentiation.

../sage/differentiation_tests.sage

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

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

0.5.3 *Differentation with respect to time*

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

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

../sage/differentiation.sage
```
@Func
def D(f):
    return lambda t: diff(f(t), t)
    #return derivative(expr, t)
```

Here is an example.

don't tangle
```
q = Function(lambda t: function("q")(t))

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 by 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:

../sage/differentiation_tests.sage
```
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.

don't tangle
```
F = 5 * q + 3 * t

show(diff(F, r)) # does not work
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 protection. 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. Now there is one further problem: `subs` does not work on lists or tuples. However, `subs` *does work* on vectors and matrices. Therefore, we cast all relevant lists to vectors, which suffices for our goal.

3. 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.

4. 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

```
def Jacobian(F):
    def f(args, vrs):
        if isinstance(args, (list, tuple)):
            args = vector(args)
        if isinstance(vrs, (list, tuple)):
            vrs = vector(vrs)
        subs = {
            v: var(f"v{id(v)}", domain=RR)
            for v in args.list()
            if not v.is_symbol()
        }
        result = jacobian(F(args.subs(subs)), vrs.subs(subs).list())
        inverse_subs = {v: k for k, v in subs.items()}
        return result.subs(inverse_subs)

    return f
```

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.

../sage/differentiation_tests.sage
```
v = var("v", domain=RR)


def F(v):
    r, t = v.list()
    return 5 * r ^ 3 + 3 * t ^ 2 * r


show(Jacobian(F)([v, t], [t]))
show(Jacobian(F)([v, t], [v, t]))
```

$$\begin{bmatrix} 6tv \end{bmatrix}$$

$$\begin{bmatrix} 3t^2 + 15v^2 & 6tv \end{bmatrix}$$

This works. Now we try the same with a function like argument. Recall, `v` must a be list for `partial` on which `gradient` depends.

../sage/differentiation_tests.sage
```
q = Function(lambda t: function("q")(t))
v = [q(t), t]
show(Jacobian(F)(v, v))
```

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

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.

don't tangle
```
def gradient(F, v):
    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.

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

../sage/differentiation_tests.sage
```
show(gradient(F)(v))
```

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

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

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

$$\begin{bmatrix} 2q\mathrm{D}_0(U)\left(t^2 + q^2\right) \\ 2t\mathrm{D}_0(U)\left(t^2 + q^2\right) \end{bmatrix}$$

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

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

../sage/differentiation_tests.sage
```
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.

../sage/differentiation.sage
```
@Func
def partial(f, slot):
    def wrapper(local):
        if slot == 0:
            selection = [time(local)]
        elif slot == 1:
            selection = coordinate(local)
        elif slot == 2:
            selection = velocity(local)
        return Jacobian(f)(local, selection)

    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.

../sage/utils1.4.sage

```
import numpy as np

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.

../sage/section1.4.sage

```
load("utils1.4.sage")

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.

don't tangle

```
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.

../sage/utils1.4.sage

```
def L_free_particle(mass):
    def Lagrangian(local):
        v = velocity(local)
        return 1 / 2 * mass * square(v)

    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, $t \to x(t)$. Since we need to perform arithmetic with literal functions, see below for some examples, we encapsulate it in a `Function`.

../sage/utils1.4.sage

```
@Func
def literal_function(name):
    return lambda t: function(name)(t)
```

It's a function.

don't tangle

```
x = literal_function("x")
print(x)
```

```
<__main__.Function object at 0x71122066e470>
```

Here are some operations on `x`.

don't tangle

```
show(x(t))
show((x+x)(t))
show(square(x)(t))
```

Note that, to keep the notation brief, the t is suppressed in the LaTeX 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`.

../sage/utils1.4.sage

```
@Func
def column_path(lst):
    #return lambda t: vector([l(t) for l in lst])
    return lambda t: column_matrix([l(t) for l in lst])
```

don't tangle
```
q = column_path(
    [
        literal_function("x"),
        literal_function("y"),
    ]
)
```

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

don't tangle
```
show(q(t))
```

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

don't tangle
```
show((q + q)(t))
```

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

don't tangle
```
show((2 * q)(t))
```

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

don't tangle
```
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,

$$\Gamma[q](\cdot) = (\cdot, q(\cdot), v(\cdot), \dots),$$
$$\Gamma[q](t) = (t, q(t), v(t), \dots).$$

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 $a = Dv, \dots$. 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
```
def Gamma(q, n=3):
    # if isinstance(q, np.ndarray):
    #     q = vector(q.tolist()) # todo, is this still needed?

    if n < 2:
        raise ValueError("n must be > 1")
    Dq = [q]
    for k in range(2, n):
        Dq.append(D(Dq[-1]))
    return lambda t: up(t, *[v(t) for v in Dq])
```

When applying Gamma to a path, we get this.

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

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

We can include the acceleration too.

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

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

todo: revise the definitions of time, coordiante, velocity, below.
Finally, here are some projections operators from the local tuple to supspaces.

../sage/utils1.4.sage
```
time = Function(lambda local: local[0])
coordinate = Function(lambda local: local[1])
velocity = Function(lambda local: local[2])
```

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

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

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.

```
../sage/section1.4.sage
q = column_path(
    [
        literal_function("x"),
        literal_function("y"),
        literal_function("z"),
    ]
)
```

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

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

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

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

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

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

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.

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

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

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)`.

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

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

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.

../sage/utils1.4.sage
```
from sage.symbolic.integration.integral 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 $\mathrm{d}t$; for this reason I overwrite the LaTeX formatting of `definite_integral`.

../sage/utils1.4.sage
```
def integral_latex_format(*args):
    expr, var, a, b = args
    return (
        fr"\int_{{{a}}}^{{{b}}} "
        + latex(expr)
        + r"\, \textrm{d}\,"
        + latex(var)
    )


definite_integral._print_latex_ = integral_latex_format
```

Here is the action along a generic path `q`.

../sage/section1.4.sage
```
T = var("T", domain="positive")

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

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

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

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`.

../sage/section1.4.sage
```
test_path = Function(lambda t: vector([4 * t + 7, 3 * t + 5, 2 * t + 1]))
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.

../sage/section1.4.sage
```
hard_path = lambda t: vector([4 * t + 7, 3 * t + 5, 2 * exp(-t) + 1])

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

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

$$377.9999999938165$$

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.

../sage/section1.4.sage
```
@Func
def make_eta(nu, t1, t2):
    return lambda t: (t - t1) * (t - t2) * nu(t)


nu = Function(lambda t: vector([sin(t), cos(t), t ^ 2]))

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.)

../sage/section1.4.sage
```
def varied_free_particle_action(mass, q, nu, t1, t2):
    eta = make_eta(nu, t1, t2)

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

    return f

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

$$436.291214285714$$

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

../sage/section1.4.sage
```
res = find_local_minimum(
    varied_free_particle_action(3.0, test_path, nu, 0.0, 10.0), -2.0, 1.0
)
show(res)
```

$$(435.000000000000, 0.0)$$

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.

../sage/utils1.4.sage
```
def Lagrangian_polynomial(ts, qs):
    return RR['x'].lagrange_polynomial(list(zip(ts, qs)))
```

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

../sage/section1.4.sage
```
ts = np.linspace(0, pi / 2, 5)
qs = [cos(t).n() for t in ts]
lp = Lagrangian_polynomial(ts, qs)
ts = np.linspace(0, pi / 2, 20)
Cos = [lp(x=t).n() for t in ts]
Sin = [lp.derivative(x)(x=t).n() for t in ts]
Zero = [abs(Cos[i] ^ 2 + Sin[i] ^ 2 - 1) for i in range(len(ts))]
show(max(Zero))
```

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.

../sage/section1.4.sage
```
def make_path(t0, q0, t1, q1, qs):
    ts = np.linspace(t0, t1, len(qs) + 2)
    qs = np.r_[q0, qs, q1]
    return lambda t: vector([Lagrangian_polynomial(ts, qs)(t)])
```

Here is the harmonic Lagrangian.

../sage/utils1.4.sage
```
def L_harmonic(m, k):
    def Lagrangian(local):
        q = coordinate(local)
        v = velocity(local)
        return (1 / 2) * m * square(v) - (1 / 2) * k * square(q)

    return Lagrangian
```

../sage/section1.4.sage
```
def parametric_path_action(Lagrangian, t0, q0, t1, q1):
    def f(qs):
        path = make_path(t0, q0, t1, q1, qs=qs)
        return Lagrangian_action(Lagrangian, path, t0, t1)

    return f
```

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

../sage/section1.4.sage
```
t0, t1 = 0, pi / 2
q0, q1 = cos(t0), cos(t1)
T = np.linspace(0, pi / 2, 5)
initial_qs = [cos(t).n() for t in T][1:-1]
parametric_path_action(L_harmonic(m=1, k=1), t0, q0, t1, q1)(initial_qs)
```

What is the quality of the path obtained by the Lagrangian interpolation? (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]`.)

../sage/section1.4.sage
```
def find_path(Lagrangian, t0, q0, t1, q1, n):
    ts = np.linspace(t0, t1, n)
    initial_qs = np.linspace(q0, q1, n)[1:-1]
    minimizing_qs = minimize(
        parametric_path_action(Lagrangian, t0, q0, t1, q1),
        initial_qs,
    )
    return make_path(t0, q0, t1, q1, minimizing_qs)

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

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

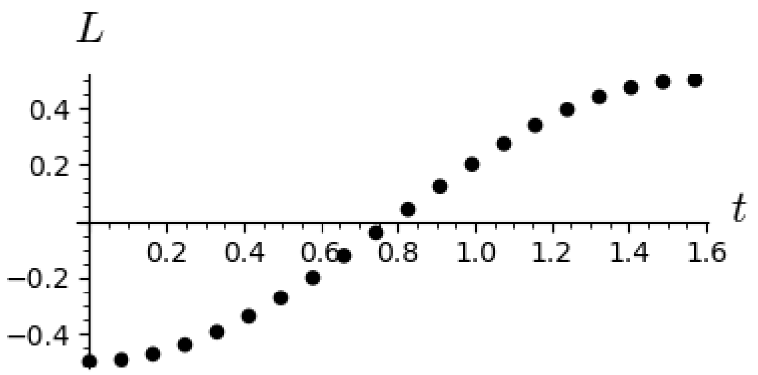

Figure 1.1: The harmonic Lagrangian as a function of the optimal path $q(t) = \cos t$, $t \in [0, \pi/2]$.

1.5 THE EULER-LAGRANGE EQUATIONS

1.5.1 *Standard imports*

../sage/utils1.5.sage

```
load("utils1.4.sage")
```

../sage/section1.5.sage
```
load("utils1.5.sage")

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

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

1.5.2 *Derivation of the Lagrange equations*

Harmonic oscillator

Here is a test on the harmonic oscillator.

../sage/section1.5.sage
```
load("utils1.4.sage")
k, m = var('k m', domain="positive")
q = column_path([literal_function("x")])
```

../sage/section1.5.sage
```
L = L_harmonic(m, k)
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 `partial(L_harmonic(m, k), 1)` maps a local tuple to a real number, and `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.)

../sage/section1.5.sage
```
show(partial(L, 1)(Gamma(q)(t)))
```

$$\left[\begin{array}{c} -kx \end{array}\right]$$

../sage/section1.5.sage
```
show(partial(L, 2)(Gamma(q)(t)))
```

$$\left[\begin{array}{c} m\dot{x} \end{array}\right]$$

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

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

../sage/section1.5.sage
```
show(compose(partial(L, 1), Gamma(q))(t))
show(compose(partial(L, 2), Gamma(q))(t))
```

$$\left[\ -kx\ \right]$$

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

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:

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

In more classical notation, we compute this:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial}{\partial_{\dot{q}}} L\left(\Gamma(q)\right)\right)(t)$$

../sage/section1.5.sage
```
show(D(compose(partial(L, 2), Gamma(q)))(t))
```

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

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

Orbital motion

../sage/section1.5.sage
```
q = column_path([literal_function("xi"), literal_function("eta")])
```

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

def L_orbital(m, mu):
    def Lagrangian(local):
        q = coordinate(local)
        v = velocity(local)
        return (1 / 2) * m * square(v) + mu / sqrt(square(q))

    return Lagrangian
```

../sage/section1.5.sage
```
L = L_orbital(m, mu)
show(L(Gamma(q)(t)))
```

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

../sage/section1.5.sage
```
show(partial(L, 1)(Gamma(q)(t)))
```

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

../sage/section1.5.sage
```
show(partial(L, 2)(Gamma(q)(t)))
```

$$\left[\begin{array}{cc} m\dot{\xi} & m\dot{\eta} \end{array}\right]$$

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

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

../sage/section1.5.sage
```
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.

../sage/utils1.5.sage
```
var("m g l", domain="positive")


def L_planar_pendulum(m, g, l):
    def Lagrangian(local):
        theta = coordinate(local).list()[0]
        theta_dot = velocity(local).list()[0]
        T = (1 / 2) * m * l ^ 2 * square(theta_dot)
        V = m * g * l * (1 - cos(theta))
        return T - V

    return Lagrangian
```

../sage/section1.5.sage
```
L = L_planar_pendulum(m, g, l)
show(L(Gamma(q)(t)))
```

../sage/section1.5.sage
```
show(partial(L, 1)(Gamma(q)(t)))
```

../sage/section1.5.sage
```
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.

../sage/utils1.5.sage
```
def L_Henon_Heiles(m):
    def Lagrangian(local):
        x, y = coordinate(local).list()
        v = velocity(local)
        T = (1 / 2) * square(v)
        V = 1 / 2 * (square(x) + square(y)) + square(x) * y - y**3 / 3
        return T - V

    return Lagrangian
```

../sage/section1.5.sage
```
L = L_Henon_Heiles(m)
q = column_path([literal_function("x"), literal_function("y")])
show(L(Gamma(q)(t)))
```

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

../sage/section1.5.sage
```
show(partial(L, 1)(Gamma(q)(t)))
```

$$\left[\begin{array}{cc} -2\,xy - x & -x^2 + y^2 - y \end{array}\right]$$

../sage/section1.5.sage
```
show(partial(L, 2)(Gamma(q)(t)))
```

$$\left[\begin{array}{cc} \dot{x} & \dot{y} \end{array}\right]$$

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

../sage/section1.5.sage
```
var('R', domain="positive")


def L_sphere(m, R):
    def Lagrangian(local):
        theta, phi = coordinate(local).list()
        alpha, beta = velocity(local).list()
        L = m * R * (square(alpha) + square(beta * sin(theta))) / 2
        return L

    return Lagrangian
```

../sage/section1.5.sage
```
q = column_path([literal_function("phi"), literal_function("theta")])
L = L_sphere(m, R)

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

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

../sage/section1.5.sage
```
show(partial(L, 1)(Gamma(q)(t)))
```

$$\left[\begin{array}{cc} Rm\cos\left(\phi\right)\sin\left(\phi\right)\dot{\theta}^{2} & 0 \end{array}\right]$$

../sage/section1.5.sage
```
show(partial(L, 2)(Gamma(q)(t)))
```

$$\left[\begin{array}{cc} Rm\dot{\phi} & Rm\sin\left(\phi\right)^{2}\dot{\theta} \end{array}\right]$$

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.

../sage/section1.5.sage
```
q = column_path(
    [
        literal_function("x"),
        literal_function("y"),
    ]
)

L = L_free_particle(m)
show(compose(partial(L, 1), Gamma(q))(t))
show(compose(partial(L, 2), Gamma(q))(t))
show(D(compose(partial(L, 2), Gamma(q)))(t))
show(
    (D(compose(partial(L, 2), Gamma(q))) - compose(partial(L, 1), Gamma(q)))(t)
)
```

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

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

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

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

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

../sage/utils1.5.sage
```
def Lagrange_equations(L):
    def f(q):
        return D(compose(partial(L, 2), Gamma(q))) - compose(
            partial(L, 1), Gamma(q)
        )

    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
```
var("a b c a0 b0 c0", domain="real")
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
l_eq = Lagrange_equations(L_free_particle(m))(test_path)
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
q = column_path([literal_function("x")])
l_eq = Lagrange_equations(L_free_particle(m))(q)
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

```
../sage/section1.5.sage
var("A phi omega", domain="real")
assume(A > 0)

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.

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

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

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

```
../sage/section1.5.sage
show(l_eq[0][0])
```

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

Let's factor out the cosine.

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

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

Kepler's third law

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

../sage/section1.5.sage
```
var("G m m1 m2", domain="positive")


def L_central_polar(m, V):
    def Lagrangian(local):
        r, phi = coordinate(local).list()
        rdot, phidot = velocity(local).list()
        T = 1 / 2 * m * (square(rdot) + square(r * phidot))
        return T - V(r)

    return Lagrangian


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

    return f
```

../sage/section1.5.sage
```
q = column_path([literal_function("r"), literal_function("phi")])
V = gravitational_energy(G, m1, m2)
L = L_central_polar(m, V)
show(L(Gamma(q)(t)))
```

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

../sage/section1.5.sage
```
l_eq = Lagrange_equations(L)(q)(t)
```

../sage/section1.5.sage
```
show(l_eq[0, 1] == 0)
```

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

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

What can we say about the other equation?

../sage/section1.5.sage
```
show(l_eq[0, 0] == 0)
```

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

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

1.6 HOW TO FIND LAGRANGIANS

1.6.1 *Standard imports*

../sage/utils1.6.sage

```
load("utils1.5.sage")
```

../sage/section1.6.sage

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

don't tangle

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

1.6.2 *Constant acceleration*

We start with a point in a uniform gravitational field.

../sage/utils1.6.sage

```
var("t", domain="real")
var("g m", domain="positive")


def L_uniform_acceleration(m, g):
    def Lagrangian(local):
        x, y  = coordinate(local).list()
        v = velocity(local)
        T = 1 / 2 * m * square(v)
        V = m * g * y
        return T - V

    return Lagrangian
```

../sage/section1.6.sage

```
q = column_path([literal_function("x"), literal_function("y")])
l_eq = Lagrange_equations(L_uniform_acceleration(m, g))(q)
show(l_eq(t))
```

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

1.6.3 *Central force field*

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

    return Lagrangian
```

Let us first try this on a concrete potential function.

../sage/section1.6.sage
```
def U(r):
    return 1 / r
```

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

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

$$\begin{bmatrix} m\ddot{x} + \frac{x\mathrm{D}_0(U)\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}} & m\ddot{y} + \frac{y\mathrm{D}_0(U)\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}} \end{bmatrix}$$

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.

../sage/utils1.6.sage
```
def F_to_C(F):
    def f(local):
        return up(
            time(local),
            F(local),
            partial(F, 0)(local) + partial(F, 1)(local) * velocity(local),
        )

    return f
```

1.6.5 *polar coordinates*

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

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

../sage/section1.6.sage
```
r = literal_function("r")
phi = literal_function("phi")
q = column_path([r, phi])
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_to_r` does not depend explicitly on t, the result should be a column matrix of zeros.

../sage/section1.6.sage
```
show((partial(p_to_r, 0)(Gamma(q)(t))))
```

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

Next is the derivative wrt r and ϕ.

../sage/section1.6.sage
```
show((partial(p_to_r, 1)(Gamma(q)(t))))
```

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

../sage/section1.6.sage
```
show(F_to_C(p_to_r)(Gamma(q)(t)))
```

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

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

../sage/utils1.6.sage
```
def L_central_polar(m, U):
    def Lagrangian(local):
        return compose(L_central_rectangular(m, U), F_to_C(p_to_r))(local)

    return Lagrangian
```

../sage/section1.6.sage
```
# show(L_central_polar(m, U)(Gamma(q)(t)))
show(L_central_polar(m, U)(Gamma(q)(t)).simplify_full())
```

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

../sage/section1.6.sage
```
expr = Lagrange_equations(L_central_polar(m, U))(q)(t)
show(expr.simplify_full().expand())
```

$$\left[\begin{array}{cc} -mr\dot\phi^2 + m\ddot r + \frac{r\mathrm{D}_0(U)\left(\sqrt{r^2}\right)}{\sqrt{r^2}} & mr^2\ddot\phi + 2mr\dot\phi\dot r \end{array}\right]$$

1.6.6 *Coriolis and centrifugal forces*

../sage/utils1.6.sage
```
def L_free_rectangular(m):
    def Lagrangian(local):
        v = velocity(local)
        return 1 / 2 * m * square(v)

    return Lagrangian


def L_free_polar(m):
    def Lagrangian(local):
        return L_free_rectangular(m)(F_to_C(p_to_r)(local))

    return Lagrangian


def F(Omega):
    def f(local):
        t = time(local)
        r, theta = coordinate(local).list()
        return vector([r, theta + Omega * t])

    return f


def L_rotating_polar(m, Omega):
    def Lagrangian(local):
        return L_free_polar(m)(F_to_C(F(Omega))(local))

    return Lagrangian



def r_to_p(local):
    x, y = coordinate(local).list()
    return column_matrix([sqrt(x * x + y * y), atan(y / x)])


def L_rotating_rectangular(m, Omega):
    def Lagrangian(local):
        return L_rotating_polar(m, Omega)(F_to_C(r_to_p)(local))

    return Lagrangian
```

../sage/section1.6.sage

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

../sage/section1.6.sage

```
show(expr)
```

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

The simplification of the Lagrange equations takes some time.

don't tangle

```
expr = Lagrange_equations(L_rotating_rectangular(m, Omega))(q)(t)
show(expr.simplify_full())
```

I edited the result a bit by hand.

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

1.6.7 *Constraints, a driven pendulum*

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

../sage/utils1.6.sage
```
def dp_coordinates(l, ys):
    "From theta to x, y coordinates."
    def f(local):
        t = time(local)
        theta = coordinate(local)[0, 0]
        return column_matrix([l * sin(theta), ys(t) - l * cos(theta)])

    return f
```

../sage/utils1.6.sage
```
def L_pend(m, l, g, ys):
    def Lagrangian(local):
        return L_uniform_acceleration(m, g)(
            F_to_C(dp_coordinates(l, ys))(local)
        )

    return Lagrangian
```

../sage/section1.6.sage
```
_ = var("l", domain="positive")

theta = column_path([literal_function("theta")])
ys = literal_function("y")

expr = L_pend(m, l, g, ys)(Gamma(theta)(t)).simplify_full()
show(expr)
```

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

1.7 EVOLUTION OF DYNAMICAL STATE

1.7.1 *Standard imports*

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

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

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

don't tangle
```
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.

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

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

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

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

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

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

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

Convert to vector.

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

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

../sage/utils1.7.sage
```
def Lagrangian_to_acceleration(L):
    def f(local):
        P = partial(L, 2)
        F = compose(transpose, partial(L, 1))
        M = (F - partial(P, 0)) - partial(P, 1) * velocity
        return partial(P, 2)(local).solve_right(M(local))

    return f
```

We apply this to the harmonic oscillator.

../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
```
def de_rhs(x, y):
    return [y, -x]


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

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

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

The solution is to replace the numbers by expressions.

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

And then define the function of differentials like this.

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

Now things work as they should.

1.7.4 *Continuing with the oscillator*

The next function computes the state derivative of the Lagrangian. For the purpose of numerical integration, we cast the result of the derivative of $\mathrm{d}t/\mathrm{d}t = 1$ to an expression, more specifically, by the above intermezzo we should set the derivative of t to `convert_to_expr(1)`.

../sage/utils1.7.sage

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

../sage/section1.7.sage

```
show(Lagrangian_to_state_derivative(L)(local))
```

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

../sage/section1.7.sage

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

../sage/section1.7.sage

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

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

../sage/utils1.7.sage

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

../sage/utils1.7.sage

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

    return f
```

../sage/section1.7.sage
```
res = Lagrange_equations_first_order(L_harmonic(m, k))(
    column_path([literal_function("x"), literal_function("y")]),
    column_path([literal_function("v_x"), literal_function("v_y")]),
)
show(res(t))
```

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

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
```
def make_dummy_vector(name, dim):
    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
```
def evolve(state_derivative, ics, times):
    dim = coordinate(ics).nrows()
    coordinates = make_dummy_vector("q", dim)
    velocities = make_dummy_vector("v", dim)
    space = up(t, coordinates, velocities)
    soln = desolve_odeint(
        des=state_derivative(space).list(),
        ics=ics.list(),
        times=times,
        dvars=space.list(),
        atol=1e-13,
    )
    return soln
```

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

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

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

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

```
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.

../sage/utils1.7.sage
```
def periodic_drive(amplitude, frequency, phase):
    def f(t):
        return amplitude * cos(frequency * t + phase)

    return f
```

With this we make the Lagrangian.

../sage/utils1.7.sage
```
_ = var("m l g A omega")


def L_periodically_driven_pendulum(m, l, g, A, omega):
    ys = periodic_drive(A, omega, 0)

    def Lagrangian(local):
        return L_pend(m, l, g, ys)(local)

    return Lagrangian
```

../sage/section1.7.sage
```
q = column_path([literal_function("theta")])
show(
    L_periodically_driven_pendulum(m, l, g, A, omega)(
        Gamma(q)(t)
    ).simplify_full()
)
```

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

../sage/section1.7.sage
```
expr = Lagrange_equations(L_periodically_driven_pendulum(m, l, g, A, omega))(
    q
)(t).simplify_full()
show(expr)
```

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

../sage/section1.7.sage
```
show(
    Lagrangian_to_acceleration(
        L_periodically_driven_pendulum(m, l, g, A, omega)
    )(Gamma(q)(t)).simplify_full()
)
```

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

../sage/section1.7.sage
```
def pend_state_derivative(m, l, g, A, omega):
    return Lagrangian_to_state_derivative(
        L_periodically_driven_pendulum(m, l, g, A, omega)
    )
```

../sage/section1.7.sage
```
expr = pend_state_derivative(m, l, g, A, omega)(Gamma(q)(t))
show(time(expr))
show(coordinate(expr).simplify_full())
show(velocity(expr).simplify_full())
```

$$1$$

$$\left(\; \dot{\theta} \;\right)$$

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

../sage/utils1.7.sage
```
def principal_value(cut_point):
    def f(x):
        return (x + cut_point) % (2 * np.pi) - cut_point

    return f
```

../sage/section1.7.sage

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

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

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

```

So now we make the plot.

../sage/section1.7.sage

```
plot_driven_pendulum(A=0.1, T=100, step_size=0.005)
```

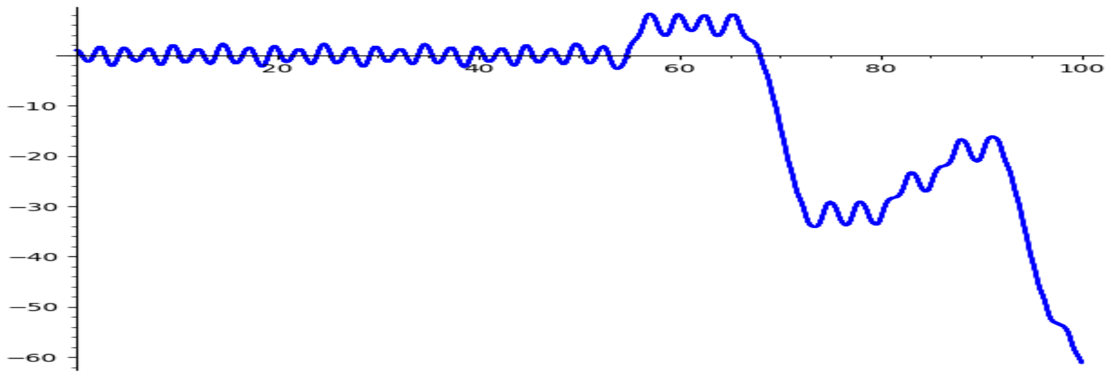

Figure 1.2: The angle of the vertically driven pendulum as a function of time. Obviously, around $t = 80$, the pendulum makes a few revolutions, and then starts to wobble again.

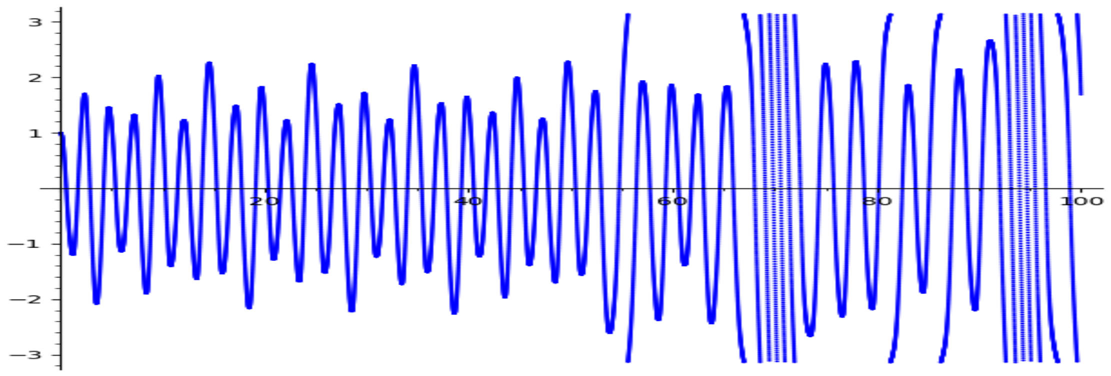

Figure 1.3: The angle on $(-\pi, \pi]$.

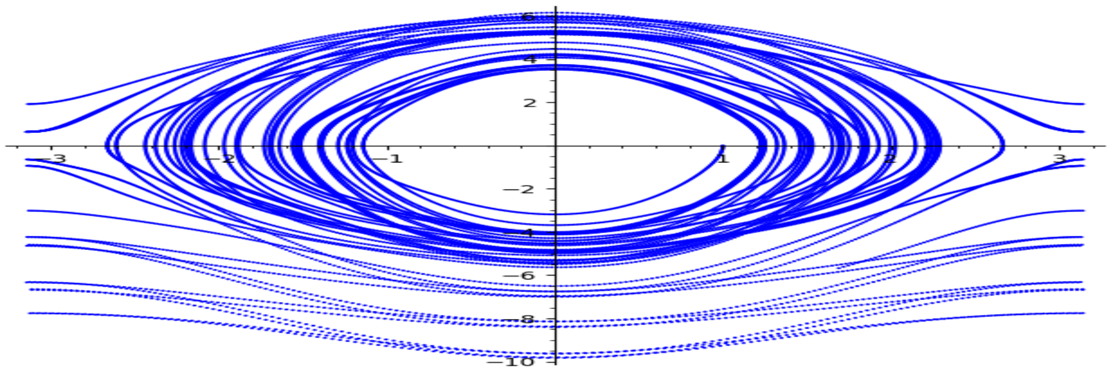

Figure 1.4: The trajectory of θ and $\dot{\theta}$.

1.8 conserved quantities

1.8.1 *Standard imports*

../sage/utils1.8.sage

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

../sage/section1.8.sage

```
load("utils1.8.sage")

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

don't tangle

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

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`.

../sage/section1.8.sage
```
def Lagrangian_to_energy(L):
    P = partial(L, 2)
    LL = Function(lambda local: L(local))
    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.

../sage/section1.8.sage
```
q = column_path(
    [
        literal_function("r"),
        literal_function("theta"),
        literal_function("phi"),
    ]
)
```

Next the transformation from spherical to 3D rectangular coordinates.

../sage/section1.8.sage
```
def s_to_r(sperical_state):
    r, theta, phi = coordinate(spherical_state).list()
    return vector(
        [r * sin(theta) * cos(phi), r * sin(theta) * sin(phi), r * cos(theta)]
    )
```

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

../sage/section1.8.sage
```
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.

../sage/section1.8.sage
```
V = Function(lambda r: function("V")(r))

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

    return Lagrangian
```

../sage/section1.8.sage
```
show(partial(L_3D_central(m, V), 1)(Gamma(q)(t)).simplify_full())
```

$$\left[\begin{array}{ccc} -\frac{r\mathrm{D}_0(V)\left(\sqrt{r^2}\right)-\left(mr\sin(\theta)^2\dot{\phi}^2+mr\dot{\theta}^2\right)\sqrt{r^2}}{\sqrt{r^2}} & m\cos(\theta)\, r^2\sin(\theta)\,\dot{\phi}^2 & 0 \end{array}\right]$$

../sage/section1.8.sage
```
show(partial(L_3D_central(m, V), 2)(Gamma(q)(t)).simplify_full())
```

$$\left[\begin{array}{ccc} m\dot{r} & mr^2\dot{\theta} & mr^2\sin(\theta)^2\dot{\phi} \end{array}\right]$$

../sage/section1.8.sage
```
def ang_mom_z(m):
    def f(rectangular_state):
        xyx = vector(coordinate(rectangular_state))
        v = vector(velocity(rectangular_state))
        return xyx.cross_product(m * v)[2]

    return f


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 $E = T + V$.

../sage/section1.8.sage
```
show(Lagrangian_to_energy(L_3D_central(m, V))(Gamma(q)(t)).simplify_full())
```

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

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.

../sage/section1.8.sage
```
var("G M0 M1 a", domain="positive")


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


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


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

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

    return f

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

    return f
```

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

../sage/section1.8.sage

```
q = column_path([literal_function("x"), literal_function("y")])
expr = (sqrt(G*M0 + G*M1)*t) / a^(3/2)
A = var('A')

show(
    Lagrangian_to_energy(L0(m, V(a, M0, M1, m)))(Gamma(q)(t))
    .simplify_full()
    .expand()
    .subs({expr: A})
)
```

$$\left[-\frac{\sqrt{M_0^2+2\,M_0M_1+M_1^2}GM_0m}{\sqrt{2\,M_0M_1a\cos(A)x+2\,M_1^2a\cos(A)x+2\,M_0M_1a\sin(A)y+2\,M_1^2a\sin(A)y+M_1^2a^2+M_0^2x^2+2\,M_0M_1x^2+M_1^2x^2+M_0^2y^2+2\,M_0M_1y^2+M_1^2y^2}}\right.$$

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.

../sage/utils1.8.sage

```
def rotation_matrix(axis, theta):
    """
    Return the 3x3 rotation matrix for a rotation of angle theta (in radians)
    about the given axis. The axis is specified as an iterable of 3 numbers.
    """
    # Convert the axis to a normalized vector
    axis = vector(axis).normalized()
    x, y, z = axis
    c = cos(theta)
    s = sin(theta)
    t = 1 - c  # common factor

    # Construct the rotation matrix using Rodrigues' formula
    R = matrix(
        [
            [c + x**2 * t, x * y * t - z * s, x * z * t + y * s],
            [y * x * t + z * s, c + y**2 * t, y * z * t - x * s],
            [z * x * t - y * s, z * y * t + x * s, c + z**2 * t],
        ]
    )
    return R
```

../sage/section1.8.sage

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

    return f
```

../sage/section1.8.sage

```
q = column_path(
    [literal_function("x"), literal_function("y"), literal_function("z")]
)
```

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

../sage/section1.8.sage

```
def Rx(s):
    return lambda local: F_tilde(s, 0, 0)(local)


s, u, v = var("s u v")
latex.matrix_delimiters(left='[', right=']')
latex.matrix_column_alignment("c")
show(Rx(s)(Gamma(q)(t)))
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` is somewhat complicated. Observe that `F_tilde` is a function of the rotation angles, and returns a function that takes `local` as argument. Now we want to differentiate `F_tilde` 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` to `local`, and use a lambda function to provide the angles as the variables. This gives us `Ftilde` (note that I drop the underscore in this name). There is one further point: `F_tilde` expects three angles, while the Jacobian provides the list `[s, u, v]` as the argument to `Ftilde`. 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 `Gamma(q)(t)` to `F_tilde`, the function depends also on t via the path $t \to \Gamma(q, t)$ 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.

../sage/section1.8.sage

```
U = Function(lambda r: function("U")(r))


def the_Noether_integral(local):
    L = L_central_rectangular(m, U)
    Ftilde = lambda x: F_tilde(*x)(local)
    DF0 = Jacobian(Ftilde)([s, u, v], [s, u, v])(s=0, u=0, v=0)
    return partial(L, 2)(local) * DF0
```

../sage/section1.8.sage

```
show(the_Noether_integral(Gamma(q)(t)).simplify_full())
```

$$\begin{bmatrix} -mz\dot{y} + my\dot{z} & mz\dot{x} - mx\dot{z} & -my\dot{x} + mx\dot{y} \end{bmatrix}$$

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*

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

../sage/section1.9.sage
```
load("utils1.9.sage")

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

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

1.9.2 *Understanding `F_to_C`*

The Scheme code starts with defining `Gamma_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.

../sage/section1.9.sage
```
r, theta = literal_function("r"), literal_function("theta")
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.

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

$$(r(t), \theta(t))$$

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.)

../sage/section1.9.sage
```
q_prime = column_path([r, theta])
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:

$$\Gamma[q] : t \to l = (t, q(t), v(t), \ldots).$$

../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.

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

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

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.

```
../sage/section1.9.sage
F = p_to_r
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.

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

```
../sage/section1.9.sage
Q = lambda t: compose(p_to_r, Gamma(q_prime))(t)
show(Gamma(Q)(t), simplify=False)
```

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

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

```
../sage/utils1.9.sage
def f_bar(q_prime):
    q = lambda t: compose(F, Gamma(q_prime))(t)
    return lambda t: Gamma(q)(t)
```

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

```
../sage/section1.9.sage
show(f_bar(q_prime)(t))
```

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

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:

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

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 $\mathrm{d}t^n$ 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 $\Gamma[q](t) = (t, q, \ldots)$. This local tuple has length 2; the local tuple $l = (t, q, v)$ has length 3.

../sage/utils1.9.sage

```
def osculating_path(local):
    t = time(local)
    q = coordinate(local)

    def wrapper(t_prime):
        res = q
        dt = 1
        factorial = 1
        for k in range(2, len(local)):
            factorial *= k
            dt *= t_prime - t
            res += local[k] * dt / factorial
        return res

    return wrapper
```

Here is an example.

../sage/section1.9.sage

```
t_prime = var("tt", domain="positive", latex_name="t'")
q = column_path([literal_function("r"), literal_function("theta")])
local = Gamma(q)(t)
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`.

../sage/utils1.9.sage

```
def Gamma_bar(f_bar):
    def wrapped(local):
        t = time(local)
        q_prime = osculating_path(local)
        return f_bar(q_prime)(t)

    return wrapped
```

../sage/section1.9.sage

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

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

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

../sage/utils1.9.sage
```
def F_to_C(F):
    def C(local):
        n = len(local)

        def f_bar(q_prime):
            q = lambda t: compose(F, Gamma(q_prime))(t)
            return lambda t: Gamma(q, n)(t)

        return Gamma_bar(f_bar)(local)

    return C
```

../sage/section1.9.sage
```
show(F_to_C(p_to_r)(local))
```

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

Here is the total time derivative.

../sage/utils1.9.sage
```
@Func
def Dt(F):
    def DtF(local):
        n = len(local)

        def DF_on_path(q):
            return D(lambda t: F(Gamma(q, n - 1)(t)))

        return Gamma_bar(DF_on_path)(local)

    return lambda state: DtF(local)
```

1.9.3 *Lagrange equations at a moment*

../sage/utils1.9.sage
```
def Euler_Lagrange_operator(L):
    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.

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

$$\begin{array}{c} t \\ \left[\, x \,\right] \\ \left[\, \dot{x} \,\right] \\ \left[\, \ddot{x} \,\right] \end{array}$$

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

$$\left[\, kx + m\ddot{x} \,\right]$$

2

CHAPTER 2

I skipped this one.

CHAPTER 3

3

3.1 HAMILTON'S EQUATIONS

3.1.1 *Standard imports*

../sage/utils3.1.sage

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

../sage/section3.1.sage

```
load("utils3.1.sage")

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

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.

../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.

../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.

../sage/utils3.1.sage

```
def transpose(M):
    return M.T


def row_path(lst):
    return lambda t: transpose(column_path(lst)(t))


def row_matrix(lst):
    return transpose(column_matrix(lst))
```

Let's try what we built.

../sage/section3.1.sage

```
q = column_path([literal_function("q_x"), literal_function("q_y")])
p = row_path([literal_function("p_x"), literal_function("p_y")])
```

../sage/section3.1.sage

```
show(p(t))
```

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

../sage/section3.1.sage

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

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

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

../sage/utils3.1.sage

```
def Hamiltonian_to_state_derivative(Hamiltonian):
    def f(H_state):
        return up(
            SR(1),
            partial(Hamiltonian, 2)(H_state).T,
            -partial(Hamiltonian, 1)(H_state),
        )

    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.

../sage/utils3.1.sage
```
var("m")

def H_rectangular(m, V):
    def f(state):
        q, p = coordinate(state), momentum(state)
        return square(p) / 2 / m + V(q)

    return f
```

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

../sage/utils3.1.sage
```
momentum = Function(lambda H_state: H_state[2])
```

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

../sage/section3.1.sage
```
V = Function(lambda x: function("V")(*x.list()))
```

This is the Hamiltonian.

../sage/section3.1.sage
```
H = H_rectangular
show(H(m, V)(H_state))
```

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

Partial derivatives work.

../sage/section3.1.sage
```
show(partial(H(m, V), 1)(H_state))
```

$$\left[\begin{array}{cc} \mathrm{D}_0\left(V\right)\left(q_x, q_y\right) & \mathrm{D}_1\left(V\right)\left(q_x, q_y\right) \end{array}\right]$$

../sage/section3.1.sage
```
show(Hamiltonian_to_state_derivative(H(m, V))(H_state))
```

$$\begin{array}{c} 1 \\ \left[\begin{array}{c} \frac{p_x}{m} \\ \frac{p_y}{m} \end{array}\right] \\ \left[\begin{array}{cc} -\mathrm{D}_0\left(V\right)\left(q_x, q_y\right) & -\mathrm{D}_1\left(V\right)\left(q_x, q_y\right) \end{array}\right] \end{array}$$

../sage/section3.1.sage

```
show(Hamilton_equations(H(m, V))(q, p)(t))
```

$$\begin{array}{c} 0 \\ \left[\begin{array}{c} -\frac{p_x}{m} + \dot{q}_x \\ -\frac{p_y}{m} + \dot{q}_y \end{array}\right] \\ \left[\begin{array}{cc} \mathrm{D}_0\left(V\right)\left(q_x, q_y\right) + \dot{p}_x & \mathrm{D}_1\left(V\right)\left(q_x, q_y\right) + \dot{p}_y \end{array}\right] \end{array}$$

3.1.3 *The Legendre Transformation*

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

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

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 0 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 0, 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.

../sage/utils3.1.sage

```
def Legendre_transform(F):
    def G(w):
        zeros = {var(f"v_{i}"): 0 for i in range(w.ncols())}
        b = gradient(F)(list(zeros.keys())).subs(zeros)
        M = Hessian(F)(list(zeros.keys())).subs(zeros)
        v = M.solve_right(w.T - b)
        return w * v - F(v)

    return G
```

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

../sage/utils3.1.sage

```
def Lagrangian_to_Hamiltonian(Lagrangian):
    def f(H_state):
        t = time(H_state)
        q = coordinate(H_state)
        p = momentum(H_state)

        def L(qdot):
            return Lagrangian(up(t, q, qdot))

        return Legendre_transform(L)(p)

    return f
```

../sage/section3.1.sage

```
res = Lagrangian_to_Hamiltonian(L_central_rectangular(m, V))(H_state)
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

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

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

../sage/section3.1.sage

```
var("m g l")
q = column_path([literal_function("theta")])
p = row_path([literal_function("p")])
```

Here is exercise 3.1.

../sage/section3.1.sage

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

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

../sage/section3.1.sage

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

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

../sage/section3.1.sage
```python
def L_sphere(m, R):
    def Lagrangian(local):
        theta, phi = coordinate(local).list()
        thetadot, phidot = velocity(local).list()
        return 1 / 2 * m * R ^ 2 * (
            square(thetadot) + square(phidot * sin(theta))
        )

    return Lagrangian


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

../sage/section3.1.sage
```python
q = column_path([literal_function("theta"), literal_function("phi")])
p = row_path([literal_function("p_x"), literal_function("p_y")])
H_state = qp_to_H_state_path(q, p)(t)
show(Lagrangian_to_Hamiltonian(L_sphere(m, R))(H_state).simplify_full())
```

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

3.2 POISSON BRACKETS

3.2.1 *The standard imports.*

3.2.2 *Standard imports*

../sage/utils3.2.sage
```python
load("utils3.1.sage")
```

../sage/section3.2.sage
```python
load("utils3.2.sage")

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

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

3.2.3 *The Poisson Bracket*

This is the Poisson bracket.

../sage/utils3.2.sage
```python
@Func
```

```
def Poisson_bracket(F, G):
    def f(state):
        left = (partial(F, 1) * compose(transpose, partial(G, 2)))(state)
        right = (partial(F, 2) * compose(transpose, partial(G, 1)))(state)
        return (left - right).simplify_full()

    return f
```

We can make general state functions like so.

../sage/utils3.2.sage

```
@Func
def state_function(name):
    return lambda H_state: function(name)(
        time(H_state), *coordinate(H_state).list(), *momentum(H_state).list()
    )
```

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

```
q = column_matrix([var("q_x"), var("q_y")])
p = row_matrix([var("p_x"), var("p_y")])
sigma = up(t, q, p)
H = state_function("H")

show(Poisson_bracket(coordinate, H)(sigma))
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.

3.2.4 *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

```
show(Poisson_bracket(H, H)(sigma))
```

$$\left[\begin{array}{c}0\end{array}\right]$$

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

../sage/section3.2.sage
```
F = state_function("F")
G = state_function("G")

show((Poisson_bracket(F, G) + Poisson_bracket(G, F))(sigma))
```

$$\left[\begin{array}{c}0\end{array}\right]$$

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

../sage/section3.2.sage
```
show(
    (
        Poisson_bracket(F, G + H)
        - Poisson_bracket(F, G)
        - Poisson_bracket(F, H)
    )(sigma)
)
```

$$\left[\begin{array}{c}0\end{array}\right]$$

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.

../sage/section3.2.sage
```
constant = Function(lambda H_state: function("c")())
```

Is it indeed constant?

../sage/section3.2.sage
```
show(Jacobian(constant)(sigma, sigma))
```

$$\left[\begin{array}{ccccc}0 & 0 & 0 & 0 & 0\end{array}\right]$$

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

../sage/section3.2.sage
```
show(
    (Poisson_bracket(F, constant * G) - constant * Poisson_bracket(F, G))(
        sigma
    ).simplify_full()
)
```

$$\left[\,0\,\right]$$

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

../sage/section3.2.sage
```
jacobi = (
    Poisson_bracket(F, Poisson_bracket(G, H))
    + Poisson_bracket(G, Poisson_bracket(H, F))
    + Poisson_bracket(H, Poisson_bracket(F, G))
)

show(jacobi(sigma).simplify_full())
```

$$\left[\,0\,\right]$$

3.2.5 *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.

../sage/section3.2.sage
```
def f(H_state):
    return function("f")(
        *coordinate(H_state).list(), *momentum(H_state).list()
    )
```

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

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

$$0$$

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

../sage/section3.2.sage
```
V = Function(lambda q: function("V")(*q.list()))

var(m, domain="positive")

H = H_rectangular(m, V)
```

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

../sage/section3.2.sage
```
q = column_matrix([var("q")])
p = row_matrix([var("p")])
sigma = up(t, q, p)

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

$$\dot{p}\frac{\partial f}{\partial p} + \dot{q}\frac{\partial f}{\partial q} = \frac{\mathrm{d}f}{\mathrm{d}t}.$$

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

CHAPTER 4

I skipped this one.