From paper to silicon: coding numerical simulations effectively

Sebastián J. Ferraro
Departamento de Matemática, Universidad Nacional del Sur, Bahía Blanca
CONICET, Argentina

XVII International ICMAT Summer School on Geometry, Dynamics and Field theory

Miraflores de la Sierra, June 2025

Some caveats for this pdf file

- This file was exported from an html file generated by Quarto.
- Some features do not work as in the “live” version:
  - There are no animations.
  - The interactive figures in the visualization section do not work.
  - For slides that change incrementally (such as those with portions of code that are consecutively highlighted), only the last state is shown, but this should not cause any confusion.

What is this course about?

Let’s say…

- You are reading some text on numerical methods, and would like to try one of them.
- You have an idea for a numerical method.
  - Want to test/show/plot/publish/…
- You have written simulations before, but want to make a better one next time.
- You don’t plan to write the code yourself, but you are intrigued about the possibilities.

What this course is not

- A Python course.
  - Python will be our language of choice, but most of the ideas could be implemented in any other language.
  - We will not focus too much on the syntax of the features. You can always look it up when you need to use it.
  - However, some Python-specific aspects are inevitable.
- A presentation of a new numerical algorithm.
  - We will limit ourselves to classic examples which you can adapt to your specific use case.

What can you expect from this course?

- I will present you with a set of tools and ideas for managing your thought process.
- You will gain a better understanding of what's behind a simulation, either as a "creator" or as a "user".
- As part of the course, we will work through an example, gradually improving it.
- You can download it to your computer to experiment or adapt to your needs.

Expected background?

- My hope is that it should be enough with your math background.

- Some basic notions regarding programs are nevertheless assumed:
  - Variables can hold data that can change
  - Iterations: for every number n for 1 to 10, print $2n$
  - There are some constructions called “functions”
  - …

Some things we'll cover

- Structuring your code with functions (at the end: classes)
- Discrete Lagrangian mechanics
- Preparing a simulation (to be improved as we advance)
- Floating point numbers and NumPy
- Automatic differentiation with JAX
- Version control with Git
- Testing with `pytest`
- Visualization
- Vectorization and Parallelization

The simplest “simulation”

- Take a number a
- Add some number b
- Repeat the last step N times
  - (addition step performed N times in total, each one over the previous result)

$$a, \quad a+b, \quad a+2b, \quad \ldots, \quad a+Nb$$

The simplest code

```
a = 5
b = 1
N = 20

for i in range(N):
    a = a + b
```

- What do we want the answer (output) to be?
- The numbers are hard-coded (sort of)
- *(And how are we supposed to run this code?)*

Print the final value

```
a = 5
b = 1
N = 20

for i in range(N):
    a = a + b

print(a)
```

Output:

```
25
```

Compute the full sequence

```
 1  a = 5
 2  b = 1
 3  N = 20
 4
 5  sequence = [a] + [0] * N  # Preallocate list: [a, 0, 0 ...]
 6
 7  for i in range(N):
 8      a = a + b
 9      sequence[i + 1] = a  # Store a (indices start at 0)
10
11  print(sequence)
```

Output:

```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25]
```

```python
def initialize_variables():
    a = 5
    b = 1
    N = 20
    return a, b, N

def next_a(a, b):
    return a + b

def calculate_sequence(a, b, N):
    sequence = [a] + [0] * N
    for i in range(N):
        a = next_a(a, b)
        sequence[i + 1] = a
    return sequence

def show_results(sequence):
    print(sequence)

# Initialization, computations and output
a, b, N = initialize_variables()
sequence = calculate_sequence(a, b, N)
show_results(sequence)
```

We could add the following:

```
def example_1():
    a = 5
    b = 1
    N = 20
    return a, b, N

def example_2():
    a = 7
    b = -2
    N = 1000
    return a, b, N

a, b, N = example_1()
c, d, M = example_2()
seq_1 = calculate_sequence(a, b, N)
seq_2 = calculate_sequence(c, d, M)
```

We could combine initialization and computation:

```
seq_1 = calculate_sequence(example_1())
```

Why structure the code?

- It is easier:
  - to read and understand
  - to maintain
  - to test
  - to reuse
- There’s a popular saying:

  “Code is read more often than it is written”

About functions

"function" in CS ≠ "function" in math

- In Computer Science, it is a "*callable unit of software logic that has a well-defined interface and behavior and can be invoked multiple times*" (Wikipedia)
- In other words, a bunch of lines of code which you can invoke/call/execute, possibly providing some input (arguments) on which it should work
- You can call a function twice with the same arguments and get different results.
  - We'll try to avoid this. Look up "functional programming".

```
def next_a(a, b):
    return a + b
```

This corresponds to a function

$$f : (a, b) \mapsto a + b$$

```
def calculate_sequence(a, b, N):
    sequence = [a] + [0] * N
    for i in range(N):
        a = next_a(a, b)
        sequence[i + 1] = a
    return sequence
```

This corresponds to, say,

$$g : \mathbb{R} \times \mathbb{R} \times \mathbb{N} \to \mathrm{FinSeq}(\mathbb{R})$$

Optional: type hints

There is a way of indicating domain / codomain:

```
1 def calculate_sequence(a: float, b: float, N: int) -> list[float]:
2     sequence = [a] + [0] * N
3     for i in range(N):
4         a = next_a(a, b)
5         sequence[i + 1] = a
6     return sequence
```

- For a `float` type hint, an argument of type `int` is acceptable
- This does not *enforce* the types.
  - Instructions for type-checker software / code editor.
- Considered good practice
- Python is a dynamically typed language (type checking happens at runtime)

```
def initialize_variables():
    a = 5
    b = 1
    N = 20
    return a, b, N
```

This is a 0-ary function (no arguments), that is, a constant, an element of $\mathbb{R} \times \mathbb{R} \times \mathbb{N}$.

```
def initialize_variables():
    a = 5
    b = 1
    N = 20
    return a, b, N
```

This is a 0-ary function (no arguments), that is, a constant, an element of $\mathbb{R} \times \mathbb{R} \times \mathbb{N}$.

Actually, this would more closely correspond to the case where `a` and `b` are `float`s:

```
def initialize_variables():
    a = 5.0
    b = 1.0
    N = 20
    return a, b, N
```

As a function,

$$h\colon \{\varnothing\} \to \mathbb{R} \times \mathbb{R} \times \mathbb{N}$$

Your turn!

We'll try to get some code running on your computers.

- Recommended
  - Install `uv` and `git` from your OS software manager, or
    - https://docs.astral.sh/uv/
    - https://git-scm.com/
- Also fine: try Python without installing anything
  - https://www.w3schools.com/python/ (scroll down to the green "Try it yourself" button)

If you go through the “install” route

- `git` is a version control system. For now, we’ll use it to get some examples in this presentation from GitHub.
- `uv` is a package and environment manager. It’s an alternative to `conda` (Anaconda, miniconda), `pip` and others.
- *Environments* are isolated installations of Python plus whatever packages you need to use.
  - Per project or per project type.
- Don’t mess with your “system” Python installation! Do not add and remove packages for your research or experiments.

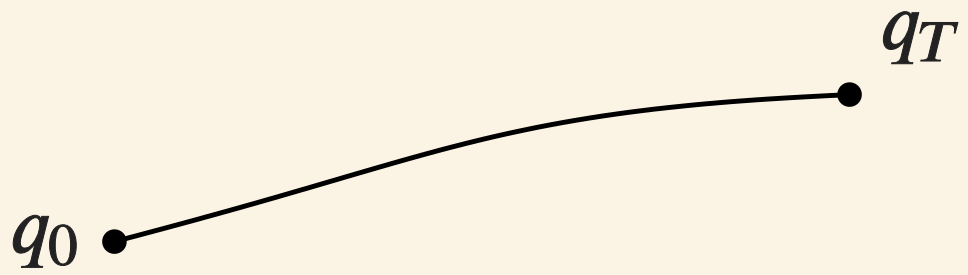
Editors

- You'll also need an editor. Some suggestions:
  - Any plain text editor (Text Editor, Notepad, …)
  - An IDE (PyCharm, Spyder, …). These provide better integration with your project.
  - VS Code, either installed or using the online version https://vscode.dev, which can open local files.

A bit of Lagrangian mechanics

- Q is the **configuration manifold** (for us, $\mathbb{R}^n$)
  - $q = (q^1, \ldots, q^n) \in Q$
- TQ is the tangent bundle of Q, that is, the space of configuration and velocities (in our case, $\mathbb{R}^n \times \mathbb{R}^n$)
  - $(q, v) = (q^1, \ldots, q^n; v^1, \ldots, v^n) \in TQ$
- $L: TQ \to \mathbb{R}$ is a function called the **Lagrangian function** (assumed smooth)
  - A typical case: kinetic minus potential energy

- Time interval $[0, T]$
- Two given points $q_0, q_T \in Q$
- $\mathcal{C}$ is the space of (smooth) curves $q\colon [0, T] \to Q$ such that $q(0) = q_0, q(T) = q_T$.



The **action functional** $\mathcal{S}\colon \mathcal{C} \to \mathbb{R}$ is

$$\mathcal{S}[q] = \int_0^T L(q(t), \dot{q}(t))\, dt.$$

Variational principle

Hamilton's principle: a **trajectory** of the system described by L is a **critical point of the action functional**

$$S[q] = \int_0^T L(q(t), \dot{q}(t))\, dt.$$

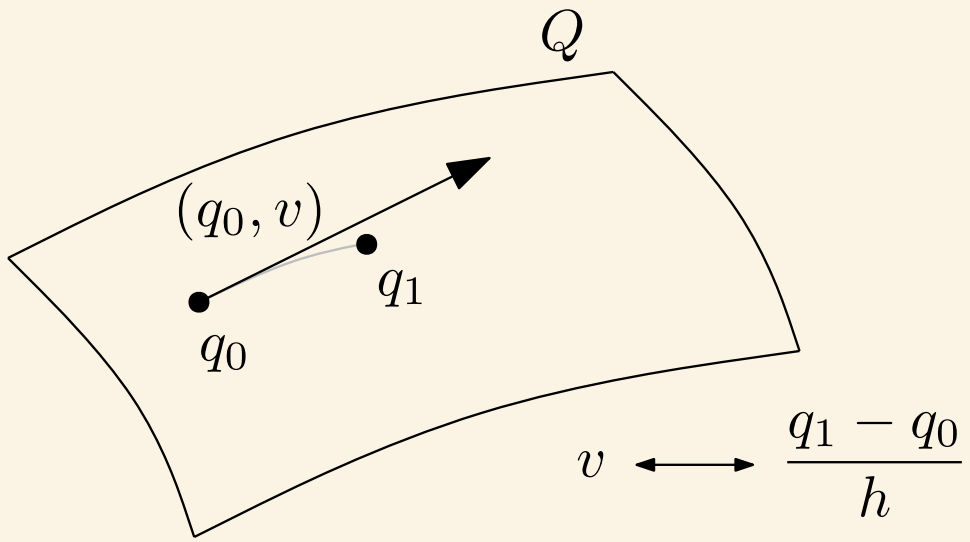
Trajectories $q(t)$ satisfy the **Euler–Lagrange equations**:

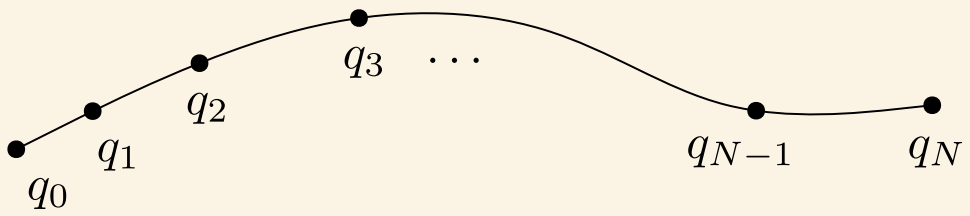
$$\frac{d}{dt}\frac{\partial L}{\partial v^i}(q(t), \dot{q}(t)) - \frac{\partial L}{\partial q^i}(q(t), \dot{q}(t)) = 0, \quad i = 1, \dots, n.$$

The fixed endpoints $q(0) = q_0, q(T) = q_T$, become boundary conditions for this ODE.

Discrete Lagrangian systems

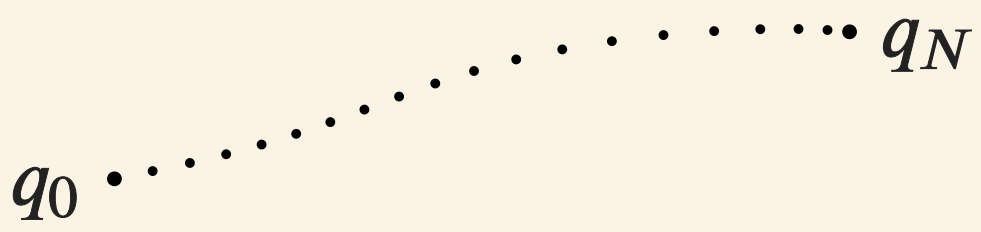
Replace TQ by $Q \times Q$ (velocities by nearby points) and curves by finite sequences of points.





- Given $L_d : Q \times Q \to \mathbb{R}$ (a **discrete Lagrangian**),
- a number of steps N, and
- two points $q_0, q_N \in Q$,

consider the space $\mathcal{C}_d$ of sequences $(q_0, q_1, \ldots, q_N)$.



Given a sequence, we can compute $L_d(q_k, q_{k+1}) \in \mathbb{R}$ for each pair of consecutive points.

Discrete Hamilton's principle

Define the **action sum** $S_d : C_d \to \mathbb{R}$

$$S_d(q_0, q_1, \dots, q_N) = \sum_{i=1}^{N} L_d(q_{k-1}, q_k)$$
$$= L_d(q_0, q_1) + L_d(q_1, q_2) + \cdots + L_d(q_{N-1}, q_N)$$

A **trajectory** of the system described by L_d is a sequence that is a critical point of S_d. This condition is expressed by the **discrete Euler–Lagrange equations**

$$D_1 L_d(q_k, q_{k+1}) + D_2 L_d(q_{k-1}, q_k) = 0, \quad k = 1, \dots, N-1.$$

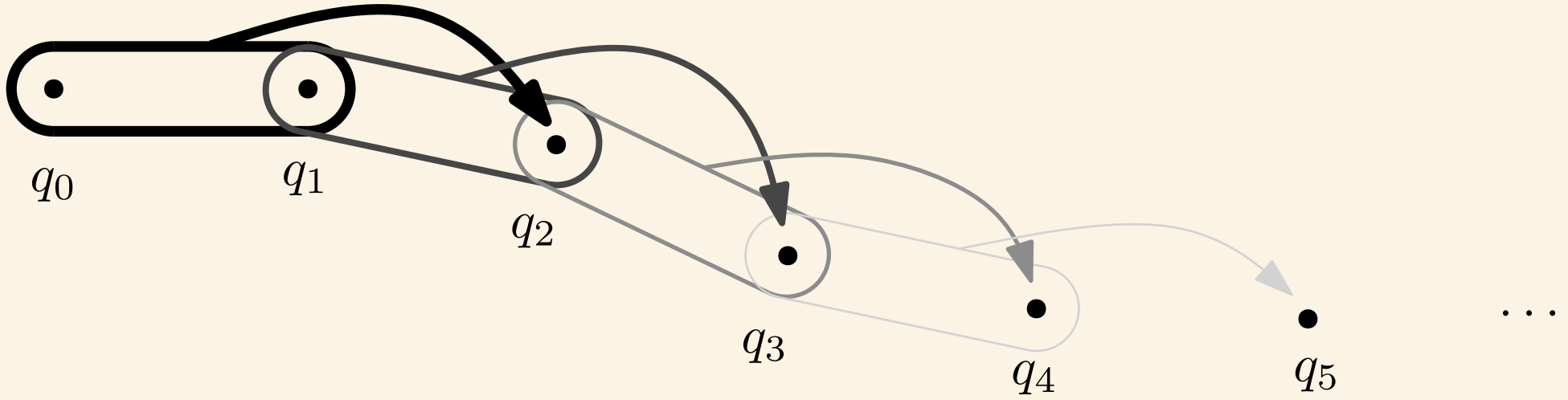
Solution strategy

$$D_1 L_d(q_k, q_{k+1}) + D_2 L_d(q_{k-1}, q_k) = 0, \quad k = 1, \dots N-1.$$

For each k, we try to solve for q_{k+1}. If it is possible and $D_{12} L_d(q_k, q_{k+1})$ is regular, which in coordinates reads

$$\det\left(\frac{\partial^2 L_d}{\partial q_k^i \partial q_{k+1}^j} \right) \neq 0,$$

then q_{k+1} is a smooth function of (q_{k-1}, q_k).


q_0
q_1
q_2
q_3
q_4
q_5
$\cdots$

From continuous to discrete

- Let $L: TQ \to \mathbb{R}$ and $[0, T]$ be given.
- Divide $[0, T]$ into N pieces of size $h = T/N$ (time step).
- For arbitrary (nearby) $q_0, q_1 \in Q$, define

$$L_d(q_0, q_1) \approx \int_0^h L(q(t), \dot{q}(t))\, dt,$$

where $q(t)$ is the trajectory of the continuous system joining q_0 to q_1 for time h. (Existence and uniqueness can be guaranteed under reasonable assumptions.)

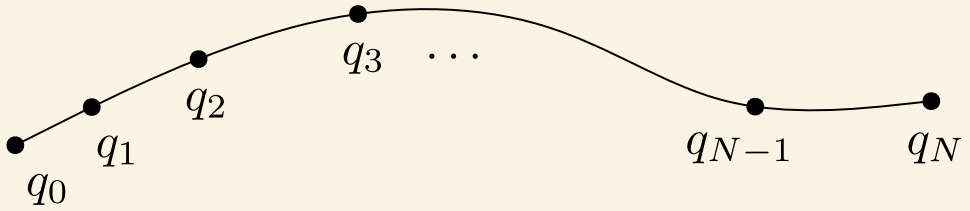
Since this connecting trajectory is not known in general, we must resort to an approximation such as the midpoint rule

$$q(t) \approx \frac{q_0 + q_1}{2}, \qquad \dot{q}(t) \approx \frac{q_1 - q_0}{h},$$

so we can define for instance

$$\begin{aligned} L_d(q_0, q_1) &= \int_0^h L\left(\frac{q_0 + q_1}{2}, \frac{q_1 - q_0}{h}\right) dt \\ &= hL\left(\frac{q_0 + q_1}{2}, \frac{q_1 - q_0}{h}\right). \end{aligned}$$

Note that the action sum approximates the action integral:



$$L_d(q_0, q_1) + L_d(q_1, q_2) + \cdots + L_d(q_{N-1}, q_N) \approx$$

$$\int_0^h L(q, \dot{q})\, dt + \int_h^{2h} L(q, \dot{q})\, dt + \cdots + \int_{(N-1)h}^{Nh} L(q, \dot{q})\, dt$$

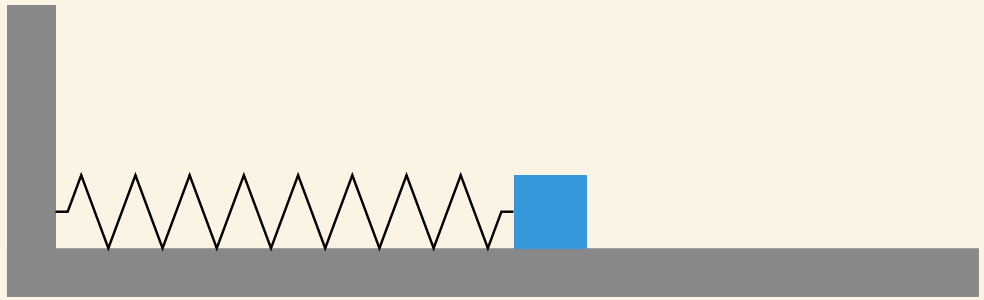
$$= \int_0^T L(q, \dot{q})\, dt$$

where each piece of $q(t)$ is the exact solution joining consecutive points.

An example: harmonic oscillator

- $Q = \mathbb{R}$
- $L(q, v) = \frac{1}{2}mv^2 - \frac{1}{2}kq^2$

- General solution: $q(t) = A \sin\left(\sqrt{\frac{k}{m}}t + \varphi\right)$



- Discretize nevertheless. Let us use the midpoint rule:

$$L(q, v) = \frac{1}{2} m v^2 - \frac{1}{2} k q^2$$

$$\begin{aligned} L_d(q_0, q_1) &= hL\left(\frac{q_0 + q_1}{2}, \frac{q_1 - q_0}{h}\right) \\ &= \frac{h}{2} m \left(\frac{q_1 - q_0}{h}\right)^2 - \frac{h}{2} k \left(\frac{q_0 + q_1}{2}\right)^2 \end{aligned}$$

Recall the DEL (Discrete Euler–Lagrange) equations. Here N is given, and so are $q_0, q_1 \in Q = \mathbb{R}$.

$$D_1 L_d(q_k, q_{k+1}) + D_2 L_d(q_{k-1}, q_k) = 0, \quad k = 1, \dots, N-1.$$

What should we do now?

1. Compute derivatives $D_i L_d$
   - By hand?
   - Using Maple / Mathematica / … (for more complex cases)
   - Using some Python package (`sympy`)
   - Using something else? (This one's for later: Automatic Differentiation)

2. Write the equations
    - Make a function that evaluates the left-hand side, with arguments (q_{k-1}, q_k, q_{k+1}) (which we'll rename to (q_0, q_1, q_2) for readability).
3. Solve the equations in a loop
    - If we can solve them explicitly (unlikely), do so.
    - In general, we can use some numerical solver within Python.

sympy approach to the derivatives

```python
from sympy import symbols, diff, simplify, latex

# Define symbols (internal name and display name)
q0, q1, q2, h, m, k = symbols('q0 q1 q2 h m k')

# Discrete Lagrangian
L_d = h * (m / 2 * ((q1 - q0) / h)**2 - k / 2 * ((q1 + q0) / 2)**2)

# Compute derivatives
D1_Ld = simplify(diff(L_d, q0))
D2_Ld = simplify(diff(L_d, q1))
D1_Ld = D1_Ld.subs({q0: q1, q1: q2}, simultaneous=True)
DEL_equations = simplify(D1_Ld + D2_Ld)

print(latex(D1_Ld), latex(D2_Ld))
```

$$\frac{-\frac{h^2 k(q_1+q_2)}{4} + m(q_1 - q_2)}{h}$$

$$\frac{-\frac{h^2 k(q_0+q_1)}{4} - m(q_0 - q_1)}{h}$$

Big downside of symbolic differentiation

(using `sympy` or any Computer Algebra System)

- **Expression swell**: for Lagrangians (and/or discretization procedures) that results in longer expressions, the derivatives are usually long and expensive to evaluate.

Advantages of symbolic differentiation

There are advantages too! It closely resembles the steps we would do manually:

- Compute the derivatives
- Change the points of evaluation
- Add the derivatives together to obtain the DEL equations
- Solve the DEL equations (we are going to do this next)

Solving (root finding)

- Numerically, not symbolically.
- We'll use two fundamental packages:
  - `numpy` enables working with multidimensional numerical arrays ("ndarrays") and performing computations on them efficiently.
  - `scipy` is built on top of it, and provides algorithms for optimization, root finding, ODE solving and much more.
- In order to use them, we add the following to our file:

```
import numpy as np
from scipy.optimize import root
```

Prepare the DEL equations for root finding

- The DEL equation is currently an *expression*, not a function
  - You can convert it into a function using `lambdify` (from `sympy`)
  - The name comes from lambda calculus, developed by Alonzo Church in the 1930s.

- At the end of the file, we add

```
DEL_equations = simplify(D1_Ld + D2_Ld)   # Repeated here for context

f_DEL = lambdify([q0, q1, q2, h, m, k], DEL_equations, "numpy")
```

“Prepare the numbers”

```
# This overwrites the old meaning of m, k, and h
# (but the expressions and functions do not change)
m = 1.0  # Mass
k = 1.0  # Spring constant
h = 0.1  # Time step

N = 100  # Number of steps
trajectory = np.zeros(N + 1)  # from 0 to N

# Set initial conditions
trajectory[0] = 1.0
trajectory[1] = 1.1
```

Root finding in a loop

```
for i in range(1, N):  # i goes from 1 to N-1
    q0 = trajectory[i - 1]
    q1 = trajectory[i]

    def solve_for_q2(q2):
        return f_DEL(q0, q1, q2, h, m, k)

    guess = 2 * q1 - q0  # guess for q2
    q2 = root(solve_for_q2, x0=guess)
    # (If you want, you can check if the root finding was successful)
    trajectory[i + 1] = q2.x[0]
```

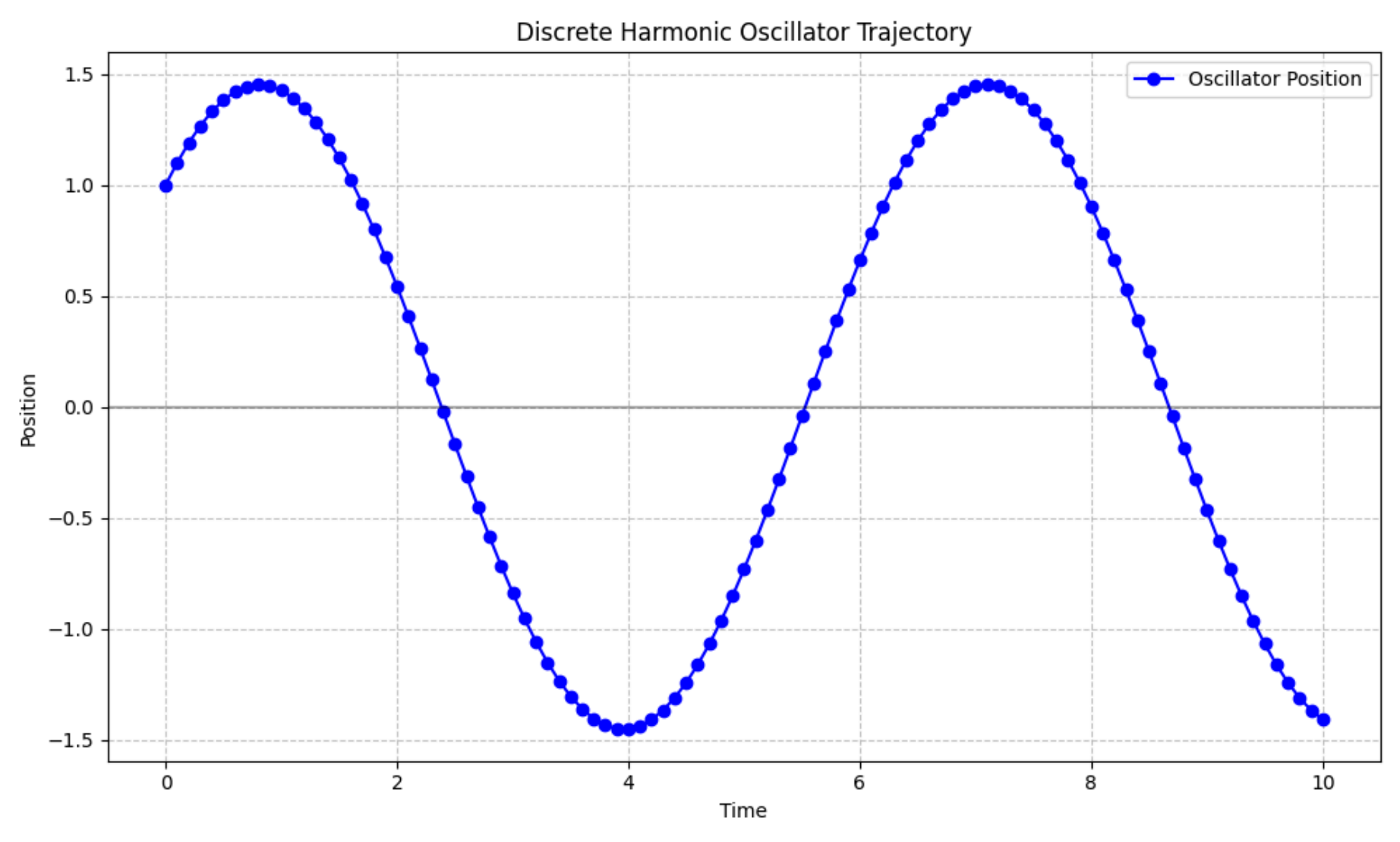
This is not too elegant, but it works for now.

Note: the guess for q_2 is wrong in the GitHub repo! It has an h that shouldn't be there. It will work anyway.

What did we miss?

- Structuring
- Visualization

Visualization



How I did this particular plot

- AI!
  - Very useful for this kind of mechanical tasks
    - Formerly: read the `matplotlib` documentation (plotting package) to find out how to produce the features in the plot (grid, colors, labels…)
    - The AI has read the docs for you, and knows how people use this package
  - Won’t write a new numerical method for you (yet?)
  - Use with caution

The AI plot code (almost verbatim)

At the beginning, add

```
import matplotlib.pyplot as plt
```

After the simulation:

```
plt.figure(figsize=(10, 6))
time_points = h * np.arange(N+1)
plt.plot(time_points, trajectory, 'bo-', label='Oscillator Position')
# Add a line at y=0 for reference
plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
plt.grid(True, linestyle='--', alpha=0.7)
plt.xlabel('Time')
plt.ylabel('Position')
plt.title('Discrete Harmonic Oscillator Trajectory')
plt.legend()
plt.tight_layout()
plt.savefig('harmonic_oscillator_trajectory.png')
```

Could you set up `git` and `uv`?

- The code for this example is available on GitHub:

  https://github.com/sebajf/ss2025_harm_osc

- In a terminal (on Windows use `cmd` or `PowerShell`):

```
# Get the code and set up the environment
mkdir summer_school_2025
cd summer_school_2025
git clone https://github.com/sebajf/ss2025_harm_osc
cd ss2025_harm_osc
uv sync
# Go to different versions of the code and run the examples
git checkout Unstructured
uv run harmonic_oscillator.py
git checkout Structured
uv run main.py
# Go to the latest version
git checkout main
```

Structure the code

- Define the discrete Lagrangian L_d, its derivatives and the DEL equations.
- Set up the simulation (N, h, initial conditions, …)
- Run the simulation loop, including the root finding procedure: $(q_0, q_1) \mapsto q_2$.
- Output results (e.g., as a plot)

One of the many ways we could structure our code. First, the (discrete) mechanical system itself:

```
def DEL_equations_harmonic_osc():
    # Define symbols (internal name and display name)
    q0, q1, q2, h, m, k = symbols("q0 q1 q2 h m k")

    # Discrete Lagrangian
    L_d = h * (m / 2 * ((q1 - q0) / h)**2 - k / 2 * ((q1 + q0) / 2)**2)

    # Compute derivatives and DEL equations
    D1_Ld = simplify(diff(L_d, q0))
    D2_Ld = simplify(diff(L_d, q1))
    D1_Ld = D1_Ld.subs({q0: q1, q1: q2}, simultaneous=True)
    DEL_equations = simplify(D1_Ld + D2_Ld)
    DEL_lambdified = lambdify([q0, q1, q2, h, m, k], DEL_equations, "numpy")

    def f_DEL_q2_first(q2, q0, q1, h, phys_params):
        # (Could check if phys_params contains the required keys)
        m, k = phys_params["m"], phys_params["k"]
        return DEL_lambdified(q0, q1, q2, h, m, k)

    return f_DEL_q2_first
```

Physical and simulation parameters (these store example-specific information)

```
def parameters_example_1():
    # Return physical parameters as a dictionary
    phys_params = {
        "m": 1.0,   # Mass
        "k": 1.0,   # Spring constant
    }
    return phys_params


def simulation_parameters_short_time():
    h = 0.1   # Time step
    N = 100   # Number of steps
    q0_q1 = np.array([1.0, 1.1])   # First two positions
    return h, N, q0_q1
```

The simulation loop

```
def perform_simulation(f_DEL_q2_first, phys_params, h, N, init_cond):
    trajectory = np.zeros(N + 1)  # from 0 to N
    # Set initial conditions
    trajectory[0:2] = init_cond  # Set first two values

    for i in range(1, N):  # i goes from 1 to N-1
        q0 = trajectory[i - 1]
        q1 = trajectory[i]

        guess = 2 * q1 - q0  # guess for q2
        q2 = root(f_DEL_q2_first, x0=guess, args=(q0, q1, h, phys_params))
        # (If you want, you can check if the root finding was successful)
        trajectory[i + 1] = q2.x[0]

    return trajectory
```

Plotting the results

```
def plot_results(trajectory, h, N, filename="harmonic_oscillator_trajectory.png
    plt.figure(figsize=(10, 6))
    time_points = h * np.arange(N + 1)
    plt.plot(time_points, trajectory, "bo-", label="Oscillator Position")
    plt.axhline(
        y=0, color="k", linestyle="-", alpha=0.3
    )
    plt.grid(True, linestyle="--", alpha=0.7)
    plt.xlabel("Time")
    plt.ylabel("Position")
    plt.title("Discrete Harmonic Oscillator Trajectory")
    plt.legend()
    plt.tight_layout()
    plt.savefig(filename)
```

Putting it all together

```
def main():
    f_DEL_q2_first = DEL_equations_harmonic_osc()
    phys_params = parameters_example_1()
    h, N, q0_q1 = simulation_parameters_short_time()
    trajectory = perform_simulation(f_DEL_q2_first, phys_params, h, N, q0_q1)
    plot_results(
        trajectory,
        h,
        N,
        filename="harm_osc_example1_short_time.png",
    )


if __name__ == "__main__":
    main()
```

Splitting into files

- The general integrator (assumes scalar q, but can be adapted later).

integrator.py

```
import numpy as np
from scipy.optimize import root
def perform_simulation(f_DEL_q2_first, phys_params, h, N, init_cond):
    trajectory = np.zeros(N + 1)  # from 0 to N
    # Set initial conditions
    trajectory[0:2] = init_cond  # Set first two values
    for i in range(1, N):  # i goes from 1 to N-1
        q0 = trajectory[i - 1]
        q1 = trajectory[i]
        guess = 2 * q1 - q0  # guess for q2
        q2 = root(f_DEL_q2_first, x0=guess, args=(q0, q1, h, phys_params))
        # (If you want, you can check if the root finding was successful)
        trajectory[i + 1] = q2.x[0]
    return trajectory
```

- The (discrete) physical system, which also “knows how to plot itself”.

harm_osc_system_sympy.py

```
from sympy import symbols, diff, simplify, lambdify
import matplotlib.pyplot as plt
import numpy as np


def DEL_equations_harmonic_osc():
    q0, q1, q2, h, m, k = symbols("q0 q1 q2 h m k")
    L_d = h * (m / 2 * ((q1 - q0) / h)**2 - k / 2 * ((q1 + q0) / 2)**2)
    # ... more code ...


def plot_results(trajectory, h, N, filename="harmonic_oscillator_trajectory.png
    # ... plotting code ...
```

- The main program, which is the “entry point”.

main.py

```
 1 import harm_osc_system_sympy
 2 import integrator
 3 import numpy as np
 4
 5 def parameters_example_1():
 6     # ... code for physical parameters ...
 7 def simulation_parameters_short_time():
 8     # ... code for h, N, q0_q1 ...
 9 def main():
10     f_DEL_q2_first = harm_osc_system_sympy.DEL_equations_harmonic_osc()
11     phys_params = parameters_example_1()
12     h, N, q0_q1 = simulation_parameters_short_time()
13     trajectory = integrator.perform_simulation(f_DEL_q2_first, phys_params,
14     harm_osc_system_sympy.plot_results(trajectory, h, N, filename="harm_osc
15 if __name__ == "__main__":
16     main()
```

Floating point numbers and where to store them

We'll discuss

- Floating point numbers
- `numpy` and its ndarrays

Fixed point?

Historically, several conventions have been used for representing (certain subsets of) real numbers in computer memory.

Fixed point refers to a representation that stores the integer part of a number and a fixed number of digits after the decimal point (*radix* point if not working in base 10).

$$\underbrace{00101101}_{8\text{ bits}}.\underbrace{10111000}_{8\text{ bits}}{}_2 = 45.71875_{10}$$

- The distance between representable numbers is always b^{-d} (base and number of “decimal” digits).

Floating point!

- It's similar to the familiar scientific notation:
  - the mass of the Earth is $5.9722 \cdot 10^{24}$ kg;
  - the diameter of a hydrogen atom is about $1.06 \cdot 10^{-10}$ m.
- Numbers in scientific notation have
  - a *sign*
  - a *significand* (here 5.9722 or 1.06)
  - an *exponent* (here 24 or -10)
- Floating point has restrictions on the number of digits in the significand and the range of the exponent.

A bit of history

Año LXII — Madrid 19 de Noviembre de 1914. — Núm. 2043

REVISTA DE OBRAS PÚBLICAS

PUBLICACION TECNICA DEL CUERPO DE INGENIEROS DE CAMINOS, CANALES Y PUERTOS

DIRECTOR

D. MANUEL MALUQUER Y SALVADOR

COLABORADORES

LOS INGENIEROS DE CAMINOS, CANALES Y PUERTOS

SE PUBLICA LOS JUEVES — Dirección y Administración: Plaza de Oriente, 6, primero derecha.

AUTOMÁTICA

COMPLEMENTO DE LA TEORIA DE LAS MÁQUINAS

POR

DON LEONARDO TORRES QUEVEDO

Ingeniero de caminos, canales y puertos.

I

La denominación de autómata se aplica á menudo á una má-

producen el desplazamiento de una pared que separa el depósito de aire del agua que le rodea; las variaciones de inclinación producen el movimiento, con relación al torpedo, del péndulo, que permanece vertical; el timón horizontal está unido al péndulo y á la pared del depósito por medio de mecanismos que le hacen tomar en cada momento la posición conveniente para que el torpedo vuelva á la profundidad que se desea.

Se trata, pues, de establecer entre tres móviles: el péndulo, la pared y el timón, enlaces mecánicos invariables. Este es un problema de la misma especie que todos los estudiados en la Cinemática aplicada á la construcción de máquinas. Su estudio no presenta aquí un interés especial.

En los autómatas del segundo grupo, el automatismo no se obtiene por medio de enlaces mecánicos invariables; se trata,

Leonardo Torres Quevedo

Germ of floating point representation

También á veces, para no escribir muchos ceros, se escriben las cantidades en esta forma: $n \times 10^m$.

Podríamos simplificar mucho esta notación estableciendo arbitrariamente tres reglas muy sencillas:

1.ª n tendrá siempre el mismo número de cifras (seis, por ejemplo).

2.ª La primera cifra de n será del orden de las décimas, la segunda del de las centésimas, etc.

3.ª Se escribirá cada cantidad en esta forma: n; m.

Así, en vez de 2.435,27 y de 0,00000341862 se escribirá, respectivamente, 243527; 4 y 341862; — 5.

No he señalado límite al valor del exponente; pero es evidente que en todos los cálculos usuales será menor de ciento; de modo que en este sistema se inscribirán todas las cantidades que intervienen en los cálculos sólo con ocho cifras.

Write numbers as $n \cdot 10^m$, with these rules:

1. n has a fixed number of digits (for example, six)
2. n should be interpreted as "zero point n"
3. Write as "$n; m$" (so $243527; 4$ means $0.243527 \cdot 10^4$)

More history

- 1938: Konrad Zuse completed the Z1, the first binary, programmable mechanical computer. He used floating point representation with 24 bits.
- 1946–1951: John von Neumann developed the Princeton IAS machine. He was against using floating point numbers, and the first version used fixed point. A later upgrade added floating point arithmetic.
- In the following decades, manufacturers were implementing their own variants of floating point arithmetic. Even different models had different implementations!

IEEE 754

- 1985: The IEEE 754 standard was established
  - IEEE: Institute of Electrical and Electronics Engineers
  - Updated and revised in 2008 and 2019.

William Kahan played the leading role in designing this standard. He has some very interesting writings on his homepage: people.eecs.berkeley.edu/~wkahan/index.htm

Single-precision

Other names: binary32, single, float32, …

Numbers are represented using 32 bits (binary digits).

- The first bit stores the sign: `0` is positive, `1` is negative.
- The next 8 bits store the exponent. The value encoded there (0 to 255) is shifted by -127 to obtain an exponent in the range -127 to 128.
- The remaining 23 bits represent the significand.

Remember: $\pm\,\text{significand} \cdot 2^{\text{exponent}}$

The significand is assumed to be “1 point something”

Double precision

Other names: binary64, double, float64, or even just float

Number of bits dedicated

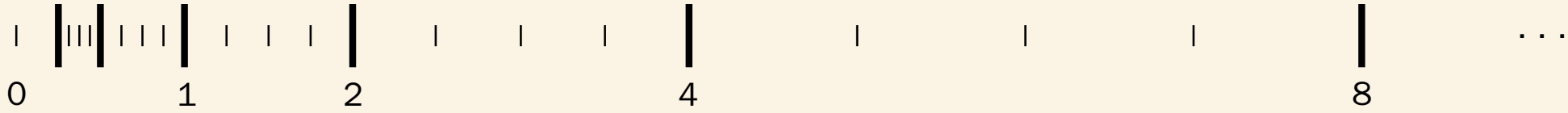
	sign	exponent	significand
`float32`	1	8	23
`float64`	1	11	52

Most programming languages use `float64` by default.

- There's quite a lot more to floating point (any precision). Special values represent `NaN` (Not a Number), $\pm\infty$, and even $+0$ and -0. There are *denormalized* or *subnormal* numbers (very close to zero), which slow down performance.

Distribution of numbers

- Representable numbers are not evenly spaced: the gaps get larger as we get far from 0.




```
from numpy import nextafter, inf

print(nextafter(.125, inf))
print(nextafter(1, inf))
print(nextafter(1000, inf))
print(nextafter(1000000, inf))
print(nextafter(10000000000, inf))
```

```
0.12500000000000003
1.0000000000000002
1000.0000000000001
1000000.0000000001
10000000000.000002
```

Distribution of numbers

- As the gaps get larger, eventually there appear integers that are not representable:

```
x = 2.0 ** 53       # "point zero" to force float
print(x == x + 1)   # Are these equal?
```

```
True
```

In the code above, $x + 1$ got truncated to x. In fact, $2^{53} + 1$ is the first (positive) nonrepresentable integer in `float64`.

- Compared to fixed point representation, you can deal with much larger numbers (about $1.7 \cdot 10^{308}$), and also much smaller (closer to 0) numbers.

Why should we care?

- Choice of `float32` vs `float64` in our simulations
  - `float64` (double precision) should be our first choice, and it is the default in most cases
  - You can even explore using quadruple or octuple precision (128 and 256 bits) if you need.
  - In some contexts, such as GPU computing, only `float32` is supported.
  - You may want to choose `float32` if performance is more important than accuracy, or if memory is limited.

Why should we care?

- Don’t compare floats using strict equality!
  - At least, floats that result from computations.

```
print(0.1 + 0.2 == 0.3)
```

```
False
```

However,

```
a = -3515.09834
b = -3515.09834
print(a == b)
```

```
True
```

So, how should we compare floats instead?

```
import numpy as np

print(np.isclose(0.1 + 0.2, 0.3))
```

```
True
```

You can also adjust the tolerance.

The periodic number we always use

- Everyone's favorite time-step, $h = 0.1$, is periodic in binary!

$$0.1_{10} = 0.0\overline{0011}_2$$

  It's not representable in binary floating point, and gets rounded. (The same goes for 0.2 and 0.3.)

- It does not mean that it's a bad choice for a time-step, but certain constructions should be avoided.

For example, `numpy.arange(start, stop, step)` generates an array with values within the half-open interval `[start, stop)`, with spacing between values given by `step`.

```
import numpy as np
x = np.arange(0, 3.14, 0.5)
```

produces an array containing $0, 0.5, 1, 1.5, 2, 2.5, 3$.

But see what happens here:

```
import numpy as np

h = 0.1
for N in range(1, 10):  # from 1 to 9
    x = np.arange(0, N * h, h)
    print(len(x), end=' ')
```

```
1 2 4 4 5 7 7 8 9
```

We were hoping for lengths 1 2 3 4 5 6 7 8 9.

However, feel free to use `numpy.arange` with integers, even if they may be interpreted as floats.

```
import numpy as np

h = 0.1
for N in range(1, 10):  # from 1 to 9
    # We'll force np.arange to produce floats to show it works
    x = h * np.arange(0.0, N)
    print(len(x), end=' ')

print(f"\nThe last x was {x}")
```

```
1 2 3 4 5 6 7 8 9
The last x was [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8]
```

Common misconceptions

- *A floating-point number represents a range of values around its “real value”.*

  False. A floating-point number represents a specific rational number exactly.

- *A small, unpredictable random number is added to every result.* $1 + 1$ *may not be exactly* 2*.*

  False. In particular, integer arithmetic is always exact (as long as you don't go beyond 2^{53} in absolute value). The case $0.1 + 0.2 \neq 0.3$ is surprising, but it's an artifact of the decimal to binary conversion.

- However, your interpreter might perform optimizations that change the order of operations, which might lead to different results due to rounding.

- *“Machine epsilon” means the smallest positive value the machine can handle.*

  False. It’s the distance between 1 and the next representable value. It serves as an upper bound to the relative approximation error due to rounding.

  For `float64`:

  - Machine epsilon: $2^{-52} \approx 2.22 \cdot 10^{-16}$
  - Smallest (“normal”) number: $2^{-1022} \approx 2.23 \cdot 10^{-308}$

    Moral: use machine epsilon for “relative” stuff (tolerances, roundoff estimates), not “absolute” comparisons.

Some recommended links

- William Kahan homepage: https://people.eecs.berkeley.edu/~wkahan/index.htm
  - How Java's Floating-Point Hurts Everyone Everywhere https://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf
  - An Interview with the Old Man of Floating-Point https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html
- What every Computer Scientist Should Know About Floating-Point Arithmetic: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- https://float.exposed/
- https://fabiensanglard.net/floating_point_visually_explained/

`numpy`'s n-dimensional arrays

- The fundamental data type in the `numpy` Python package is the ndarray:
  - A multidimensional container of data (usually numbers), designed to be efficient for numerical operations.
  - The natural data type for vectors, matrices, tensors… mostly anything indexed.

Axes and shape

```
import numpy as np

a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
print(a.ndim)
print(a.shape)
print(a[1, 2])
```

```
2
(3, 4)
7
```

- `a` has two “axes” (it’s 2D).
- Its `shape` is a tuple of length 2 because `ndim` is 2.
- Axes are numbered starting from 0:
  - axis 0, axis 1 (and so on)

As a function,

$$a\colon \{0, 1, 2\} \times \{0, 1, 2, 3\} \to \mathbb{R}$$

- Each factor in the cartesian product corresponds to an axis.
- Factors are of the form $\{0, 1, \ldots, n - 1\}$, where n is the size of the axis.

Are vectors row/column matrices?

Neither:

```
import numpy as np

b = np.array([1, 2, 3, 4])
print(b.ndim)
print(b.shape)
```

```
1
(4,)
```

$$b\colon \{0, 1, 2, 3\} \to \mathbb{R}$$

- When confused, don't think in terms of rows and columns, but in terms of axes!

More than two axes

- You can nest arrays (of consistent sizes).

```
import numpy as np

a = np.array([1, 2, 3])
b0 = np.stack((a, a, a, a, a))  # Along axis 0 by default
b1 = np.stack((a, a, a, a, a), axis=1)  # Along axis 1
c = np.stack((b1, b1))
print(a.shape, b0.shape, b1.shape, c.shape)
```

```
(3,) (5, 3) (3, 5) (2, 3, 5)
```

- Useful for block matrices (with blocks of the same order)
  - For example, an ndarray of `shape` (2, 3, 4, 5) can be seen as a 2×3 block matrix, each block of size 4×5.
- Useful for batches of data.

Operations

You can do operations without loops:

- Of course, sums, multiplications (`np.matmul` or `@` as a shorthand).
- Linear algebra operations (inverses, SVD, …)
- Einstein summation convention!
  - `np.einsum('ij,jk', a, b)` performs the matrix multiplication of `a` and `b`
  - Very flexible (see the documentation if you need to use it)
- Element-wise operations: `np.sin`, `np.exp`, `np.sqrt`, etc. are applied element-wise to ndarrays.

Automatic differentiation

- Automatic differentiation, also called autodiff or AD is **not**
  - symbolic (like `sympy`)
  - numeric (no difference quotients $\frac{\Delta f}{\Delta x}$)
- What is it, then?

Question:

If a program is something that takes numbers and outputs numbers… can you differentiate a program?

Idea behind autodiff

A program (in some subclass of “differentiable” programs)

```
def f(x):
    # ... lots of computations, including loops, branching (if/else) ...
    return final_value
```

is a function

$$f: \mathbb{R}^n \to \mathbb{R}^m.$$

(May be defined only in some subset, of course.) We want to build the derivative of `f` as a *program*, not as an expression. We can think of

$$Df: \mathbb{R}^n \to \mathbb{R}^{m \times n}$$ (matrices, or linear operators)

- If we know the rules for differentiation of “basic” functions:
  - the derivative of `np.sin` is `np.cos`
  - …
- and “structural” differentiation rules:
  - derivatives of sums, products…
  - chain rule!

Then we could transform the program `f` into the program `Df`.

`f` can even involve branching (`if`/`else`), loops, recursion, or calling other functions.

Software packages that do this for us

- Python packages: `jax`, `PyTorch`, `TensorFlow` (not Python-specific)
- We'll use `jax`
  - By Google + Nvidia + community contributors.
  - `numpy`-compatible syntax
  - GPU/TPU acceleration
  - Just-In-Time (JIT) compilation

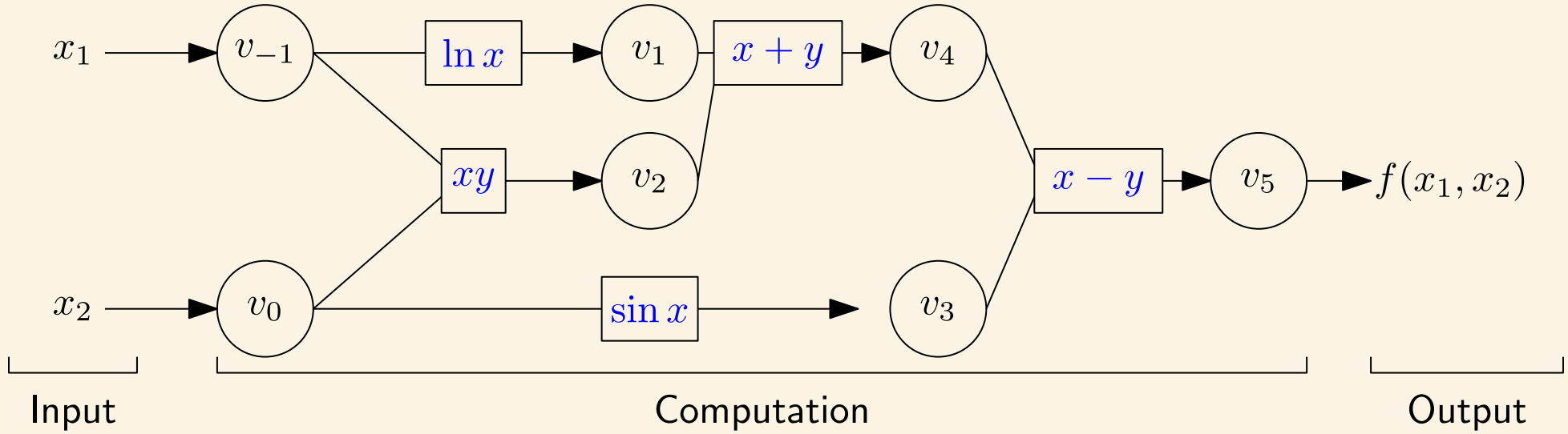
Autodiff theory

Recommended read: Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind. “Automatic differentiation in machine learning: a survey”. *The Journal of Machine Learning Research*, 18(153):1–43, 2018. https://arxiv.org/abs/1502.05767.

The following example is from that paper, adding some geometry.

Computational graph

$$f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2$$



In the graph, $\ln x$ denotes the function $x \mapsto \ln x$, etc.

Primals and tangents

Each $v_i \in \mathbb{R}$ is either a constant or a function of some of the previous ones. The variables v_i are called *primals*. To each v_i we can associate a value $\dot{v}_i$ called its *tangent*.

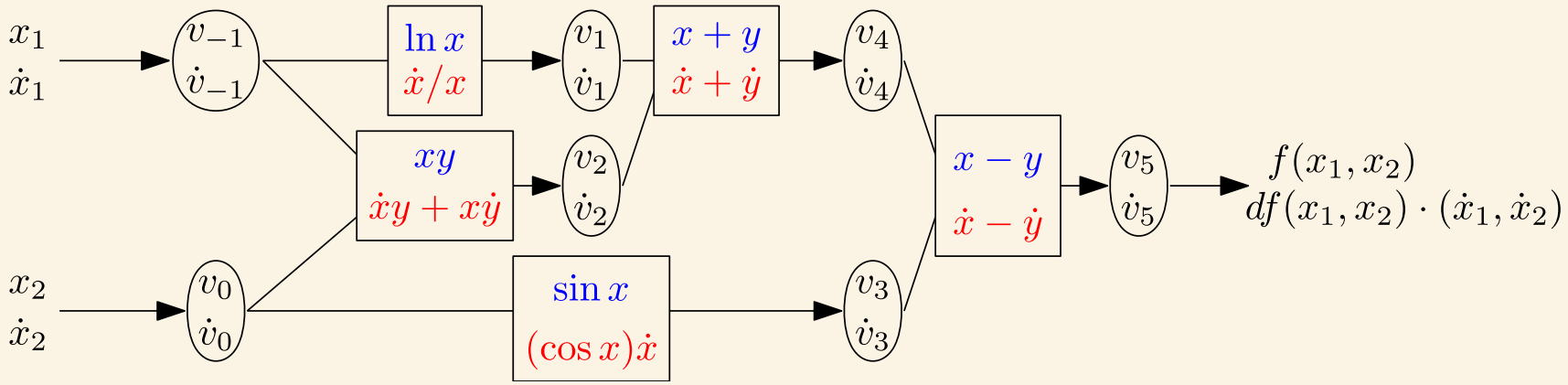
In other words, we have, for each i,

$$(v_i, \dot{v}_i) \in T\mathbb{R} \equiv \mathbb{R} \times \mathbb{R},$$

and also

$$(x_1, x_2, \dot{x}_1, \dot{x}_2), \text{ or equivalently, } (x_1, \dot{x}_1), (x_2, \dot{x}_2)$$

Let us enrich our computational graph by replacing variables by primal/tangent pairs, and operations by their tangent maps.



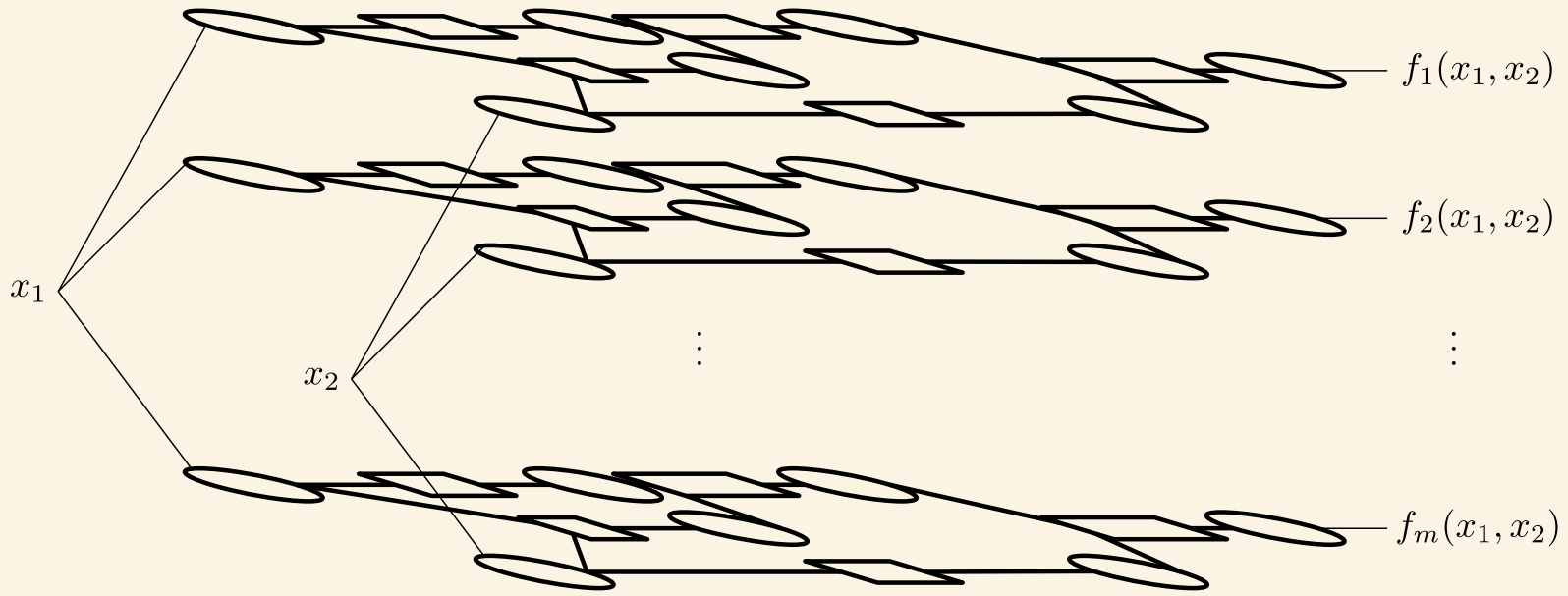
We pass from a diagram through which real numbers "flow" through functions ϕ, to one where elements of $T\mathbb{R}$ "flow" through tangent maps $T\phi$ (with the same topology!).

We claim that this computes $df(x_1, x_2) \cdot (\dot{x}_1, \dot{x}_2)$, and this is because of the chain rule: $T(f \circ g) = Tf \circ Tg$.

- This is called **forward mode autodiff**.
- It computes f at $x = (x_1, x_2)$ and its directional derivative w.r.t. $\dot{x} = (\dot{x}_1, \dot{x}_2)$ in one single pass.
- The results are numerical values, not symbolic expressions. They are the exact numerical value we would compute if we differentiated by hand and then evaluated the expressions.
- If $\dot{x}$ is $(1, 0)$ or $(0, 1)$ we get the partial derivatives at x.
- The value of $\dot{x}$ is sometimes called the *seed*.

Vector-valued functions

- For $f\colon \mathbb{R}^n \to \mathbb{R}^m$, $f = (f_1, \ldots, f_m)$, we have a computation graph for each component. They make up a big graph, to which we can apply our “tangent enrichment”.



(Just an illustration, the computations could be intertwined.)

- Given $x \in \mathbb{R}^n$, using $\mathbf{e}_1$ as the seed, we can compute

$$\frac{\partial f_1}{\partial x_1}, \cdots, \frac{\partial f_m}{\partial x_1}$$

in one pass, to compute column 1 of the Jacobian matrix

$$Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}(x)$$

Bonus!

- We can compute Jacobian-vector products (JVP) in one single pass, without ever computing the Jacobian matrix.
- To compute $Df(x) \cdot v$, just use v as the seed.
- (In comparison, the full Jacobian matrix requires n passes.)

JAX provides:

- `jax.grad`, `jax.value_and_grad`, for scalar f; and `jax.jacfwd`, the Jacobian evaluated column by column. These take a function and return a function.
- `jax.numpy`, a submodule that behaves like `numpy` but is adapted to work well with `jax`.
  - Change `import numpy as np` → `import jax.numpy as jnp`,
  - and change `np.whatever(...)` → `jnp.whatever(...)`.
- Much, much more.

AD for harmonic oscillator DEL eqns

The old `sympy` version had:

```
1 from sympy import symbols, diff, simplify, lambdify
2 import numpy as np
3 from scipy.optimize import root
4 import matplotlib.pyplot as plt
```

Change it to

```
1 import jax
2 import jax.numpy as jnp
3 from scipy.optimize import root
4 import matplotlib.pyplot as plt
```

With `sympy`, we had

```
def DEL_equations_harmonic_osc():
    # Define symbols (internal name and display name)
    q0, q1, q2, h, m, k = symbols("q0 q1 q2 h m k")

    # Discrete Lagrangian
    L_d = h * (m / 2 * ((q1 - q0) / h) ** 2 - k / 2 * ((q1 + q0) / 2) ** 2)

    # Compute derivatives
    D1_Ld = simplify(diff(L_d, q0))
    D2_Ld = simplify(diff(L_d, q1))
    D1_Ld = D1_Ld.subs({q0: q1, q1: q2}, simultaneous=True)
    DEL_equations = simplify(D1_Ld + D2_Ld)

    DEL_lambdified = lambdify([q0, q1, q2, h, m, k], DEL_equations, "numpy")

    def f_DEL(q0, q1, q2, h, phys_params):
        # (Could check if phys_params contains the required keys)
        m, k = phys_params["m"], phys_params["k"]
        return DEL_lambdified(q0, q1, q2, h, m, k)

    return f_DEL
```

In the JAX version, we do this:

```
def L_d(q0, q1, h, phys_params):
    m, k = phys_params["m"], phys_params["k"]
    return h * (m / 2 * ((q1 - q0) / h) ** 2 - k / 2 * ((q1 + q0) / 2) ** 2)

D1_Ld = jax.jacfwd(L_d, argnums=0)  # The default, actually
D2_Ld = jax.jacfwd(L_d, argnums=1)

def DEL_equations_harmonic_osc(q0, q1, q2, h, phys_params):
    return D1_Ld(q1, q2, h, phys_params) + D2_Ld(q0, q1, h, phys_params)
```

We could go even further, and change the definition of `L_d` to

```
def L(q, v, phys_params):
    m, k = phys_params["m"], phys_params["k"]
    return m / 2 * v ** 2 - k / 2 * q ** 2

# midpoint discretization for any L
def L_d(q0, q1, h, phys_params):
    return h * L((q1 + q0) / 2, (q1 - q0) / h, phys_params)
```

- Everything else (setting up the physical and simulation parameters, performing the simulation loop, plotting the results…) stays the same!

- Well, mostly the same. You need to change `np.zeros` to `jnp.zeros`, `np.array` to `jnp.array`, etc.

- The first time you call a transformed (differentiated) function such as `D1_Ld`, it will take a bit longer. The function is “traced” (JAX does the procedure explained above). Subsequent calls will be much faster.

- Speaking of speed, we can use `jax.jit` (Just-In-Time compilation) to speed up the execution.

```
@jax.jit
def DEL_equations_harmonic_osc(q0, q1, q2, h, phys_params):
    return D1_Ld(q1, q2, h, phys_params) + D2_Ld(q0, q1, h, phys_params)
```

We'll talk about JIT in more detail later.

Immutability

There is one big difference with `numpy`. JAX's arrays are *immutable* by design choice. To change values inside them, a special syntax is needed. Instead of

```
def perform_simulation(f_DEL, phys_params, h, N, init_cond):
    # (...)
    trajectory[0:2] = init_cond
    # (...)
        trajectory[i + 1] = q2.x[0]
```

we need to write

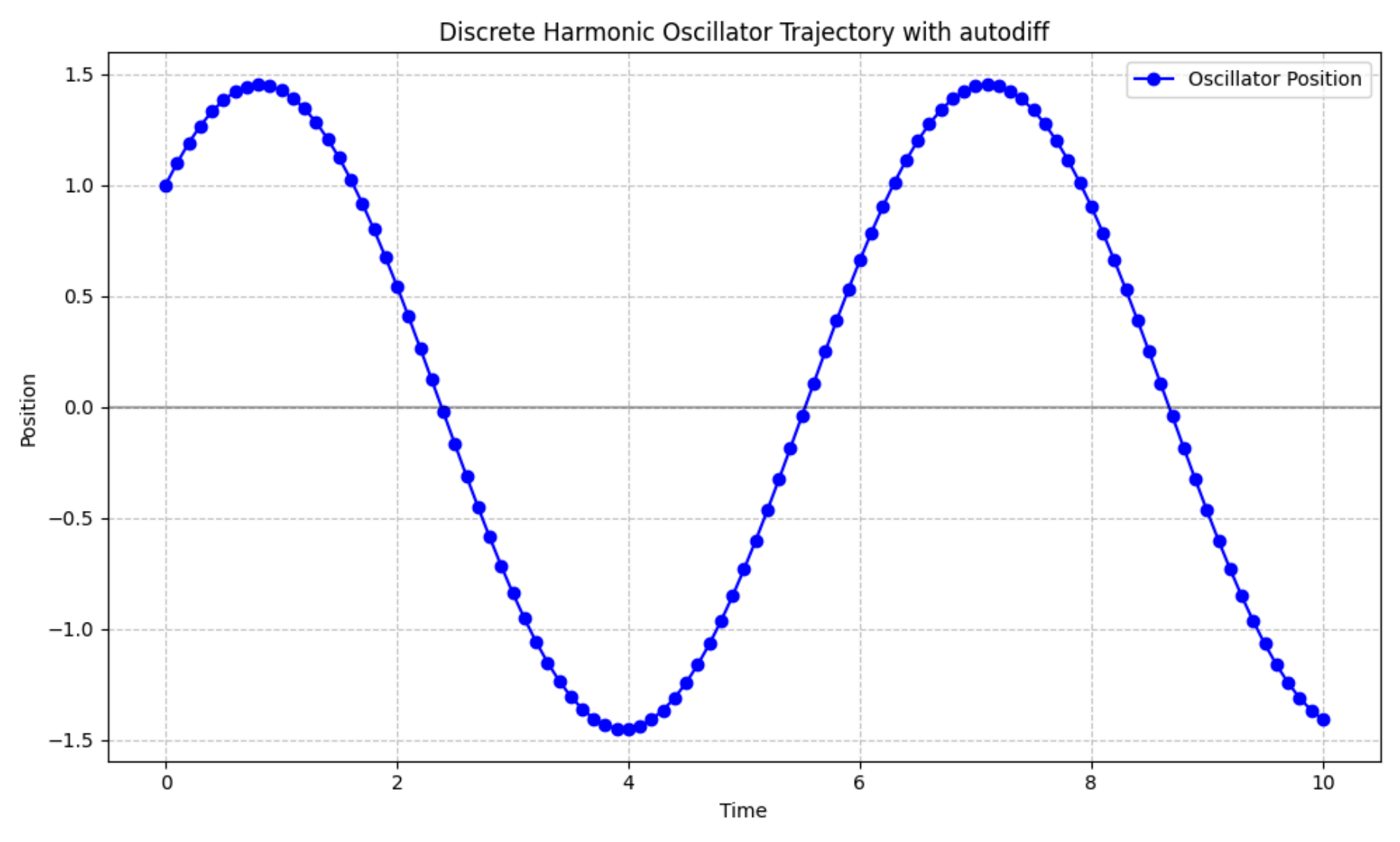
```
def perform_simulation(f_DEL, phys_params, h, N, init_cond):
    # (...)
    trajectory = trajectory.at[0:2].set(init_cond)
    # (...)
        trajectory = trajectory.at[i + 1].set(q2.x[0])
```

Another difference: default is `float32`

```
import jax
import jax.numpy as jnp

print(f"Default float type in jax: {jnp.array(1.0).dtype}")

# Enable or disable float64 precision
jax.config.update("jax_enable_x64", True)
print(f"Float type with jax_enable_x64=True: {jnp.array(1.0).dtype}")

jax.config.update("jax_enable_x64", False)
print(f"Float type with jax_enable_x64=False: {jnp.array(1.0).dtype}")
```

```
Default float type in jax: float32
Float type with jax_enable_x64=True: float64
Float type with jax_enable_x64=False: float32
```

The output looks the same as before!



Some thoughts

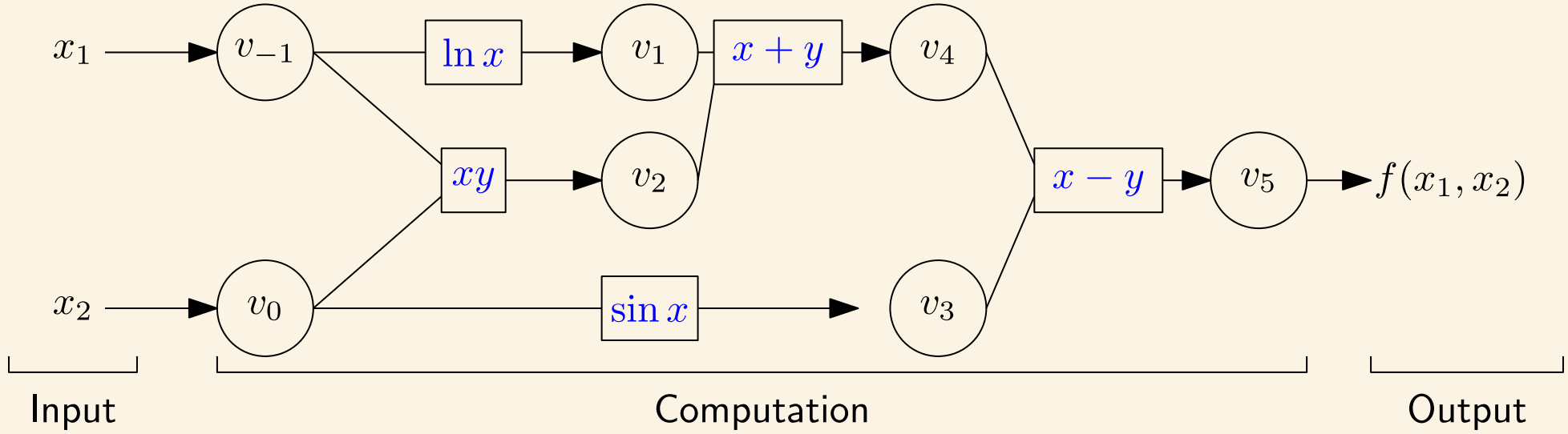
- This moves the line that divides what we as researchers must do manually (compute the derivatives, and even write the equations) and what the computer does for us.

- Once you write the equations for your numerical method in terms of derivatives, all you need to do is provide L_d.

- It feels a bit strange asking the computer to solve an equation we can't see.

Back to the tangent picture

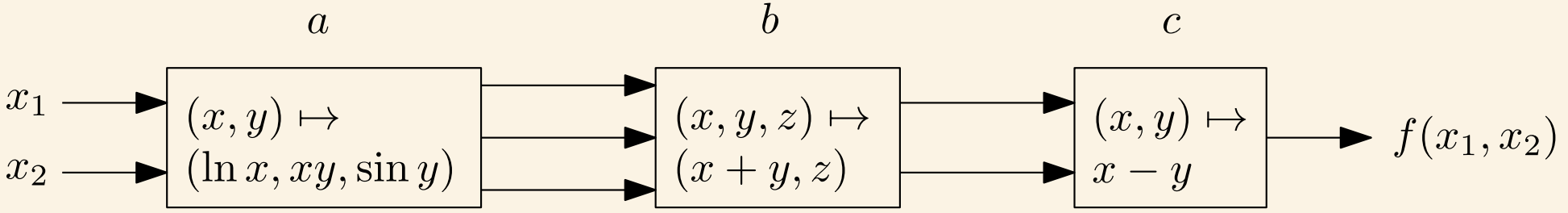
Recall our function $f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2$.



Back to the tangent picture

Recall our function $f(x_1, x_2) = \ln x_1 + x_1 x_2 - \sin x_2$.

We can replace the diagram by



- a could be further split into several consecutive blocks. The point is that f is a composition of simpler functions: $f = c \circ b \circ a$.
- Forward mode autodiff is just applying the tangent functor T to this diagram.

- The chain rule $Tf = Tc \circ Tb \circ Ta$ means that we can compute

$$Tf(x, v) = (f(x), Df(x) \cdot v)$$

  in steps:

  - take (x, v),
  - apply Ta,
  - then Tb,
  - then Tc.

- *Each step is simple enough for the autodiff software to produce automatically.*

With forward-mode AD, if $f = (f_1, \ldots, f_m)$, then we can compute column j in the Jacobian

$$Df(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}(x)$$

in one pass, using $\mathbf{e}_j$ as the seed.

- What if the Jacobian is very wide, that is, $n \gg m$?
- For example: compute the gradient of $f\colon \mathbb{R}^{1000} \to \mathbb{R}$.

Enter reverse mode autodiff

We are denoting the standard basis of $\mathbb{R}^n$ by $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$. Let $\{\mathbf{e}^1, \ldots, \mathbf{e}^m\}$ denote the dual of the standard basis of $\mathbb{R}^m$ (not $\mathbb{R}^n$!). Let us represent these by column and row matrices:

- $\mathbf{e}_j$: column matrix with a 1 at position j (the rest are 0s).
- $\mathbf{e}^i$: row matrix with a 1 at position i (the rest are 0s).

$$\frac{\partial f_i}{\partial x_j}(x) = \mathbf{e}^i \, Df(x) \mathbf{e}_j = \mathbf{e}^i \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{bmatrix} \mathbf{e}_j$$

Rows and columns

$Df(x)\mathbf{e}_j$ gives us a column of the Jacobian, but $\mathbf{e}^i Df(x)$ gives us a *row*.

Recall our function $f = c \circ b \circ a$. Denote the intermediate values by

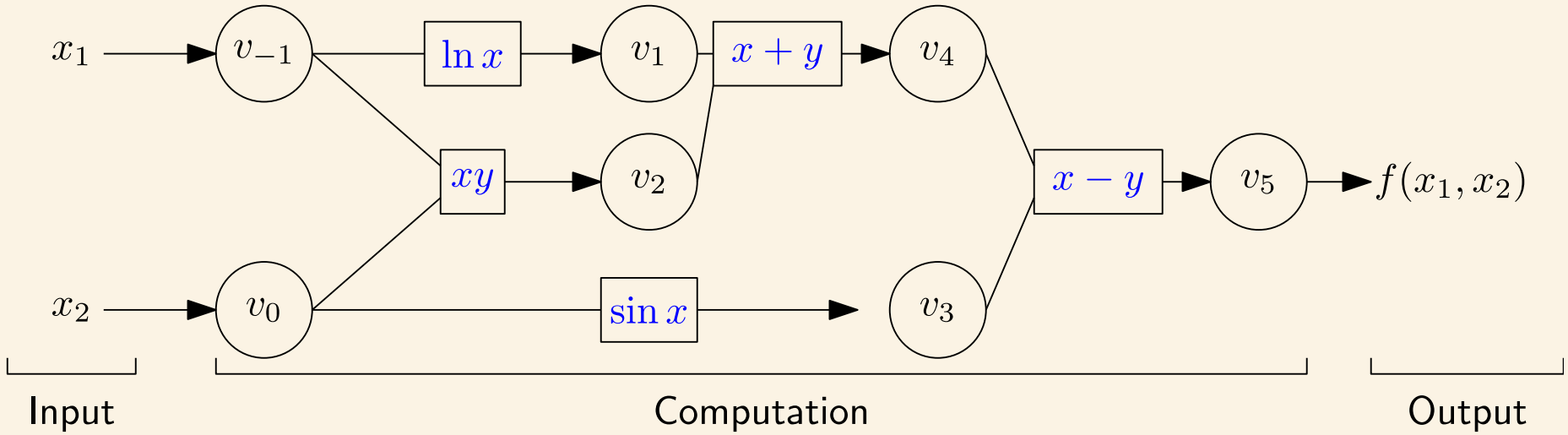
$$x \overset{a}{\longmapsto} w \overset{b}{\longmapsto} z \overset{c}{\longmapsto} y = f(x)$$

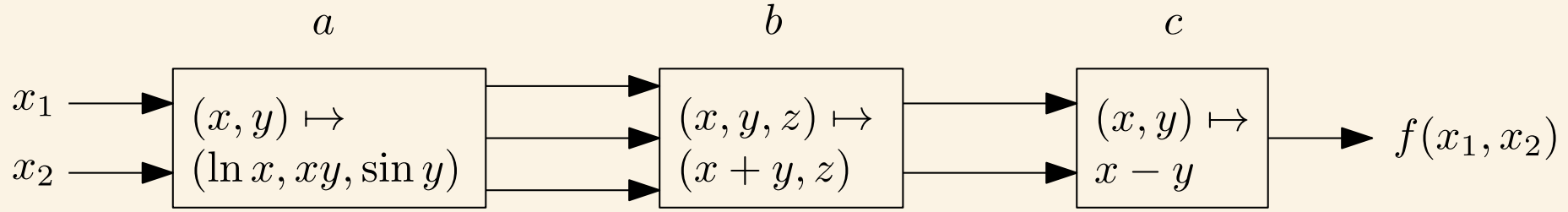
Then

$$\mathbf{e}^i Df(x) = \mathbf{e}^i Dc(z)\, Db(w)\, Da(x)$$

This is the core of reverse mode autodiff! Let's see what this means.

Back to the graphs





"Detailed" and "single stream" versions of the computational graph.

Step by step

1. Do one forward pass through the graph (just with the primals $x = (x_1, x_2)$, no tangents here), to compute the intermediate values w and z (and the final value y).
2. Choose the seed at the end of the graph, associated to y. Now it has to be a covector! For example, $\mathbf{e}^i$.
3. Apply $Dc(z)$ (as matrix multiplication on the right), then $Db(w)$, then $Da(x)$. This requires traversing the graph *backwards*. We obtain

$$\mathbf{e}^i \, Dc(z) \, Db(w) \, Da(x)$$

(this is the ith row of $Df(x)$.)

- This means that we can compute row i of the Jacobian in just 2 passes (forward, then backward), no matter how large n is.
- In practice, the “detailed” version of the computational graph is traversed backwards. The intermediate results that originate from the seed and accompany the values of the primals are called the *adjoints*.
- Reverse mode autodiff is also called adjoint or cotangent linear mode.
- In machine learning, for neural network training, it is called backpropagation.

In geometric terms…

we are taking the constant 1-form $\mathbf{e}^i$ and pulling it back by c, then b, then a. For the automatic differentiation of $f = c \circ b \circ a$,

- forward mode autodiff relies on $Tf = Tc \circ Tb \circ Ta$
- reverse mode autodiff relies on $f^* = a^* \circ b^* \circ c^*$

- In JAX, this is implemented by `jax.jacrev`.
- If instead of $\mathbf{e}^i$ we take any seed, we get what is called a "vector-Jacobian product" (VJP) or "transposed Jacobian–vector product" in the literature.
- For computing the Hessian of a scalar function `f`, the most efficient way is to do `jax.jacfwd(jax.jacrev(f))`
  - Or just `jax.hessian(f)` which does exactly that internally.

Some cool derivatives

- JAX can differentiate through the *whole simulation loop* to get $\partial q_N / \partial q_0$ and $\partial q_N / \partial q_1$.

```
  # ( ... a couple of changes later (see GitHub repo) ... )
def final_point(f_DEL_q2_first, phys_params, h, N, q0, q1):
    trajectory = integrator.perform_simulation(
        f_DEL_q2_first, phys_params, h, N, q0, q1
    )
    return trajectory[-1]  # The -1 "wraps around" to the last element
partial_qN_partial_q0 = jax.jacfwd(final_point, argnums=4)
partial_qN_partial_q1 = jax.jacfwd(final_point, argnums=5)
print(
    "Sensitivity of final point wrt q0:",
    partial_qN_partial_q0(f_DEL_q2_first, phys_params, h, N, q0, q1),
)
print(
    "Sensitivity of final point wrt q1:",
    partial_qN_partial_q1(f_DEL_q2_first, phys_params, h, N, q0, q1),
)
```

This produces:

```
Sensitivity of final point wrt q0: 4.513205
Sensitivity of final point wrt q1: -5.383626
```

- The most important change I made:
  - JAX can’t differentiate through the `scipy.optimize.root` “black-box” zero-finding procedure.
  - I added a Newton-Raphson method for solving the DEL equations (see GitHub repo for the course).

- Our `perform_simulation` function does

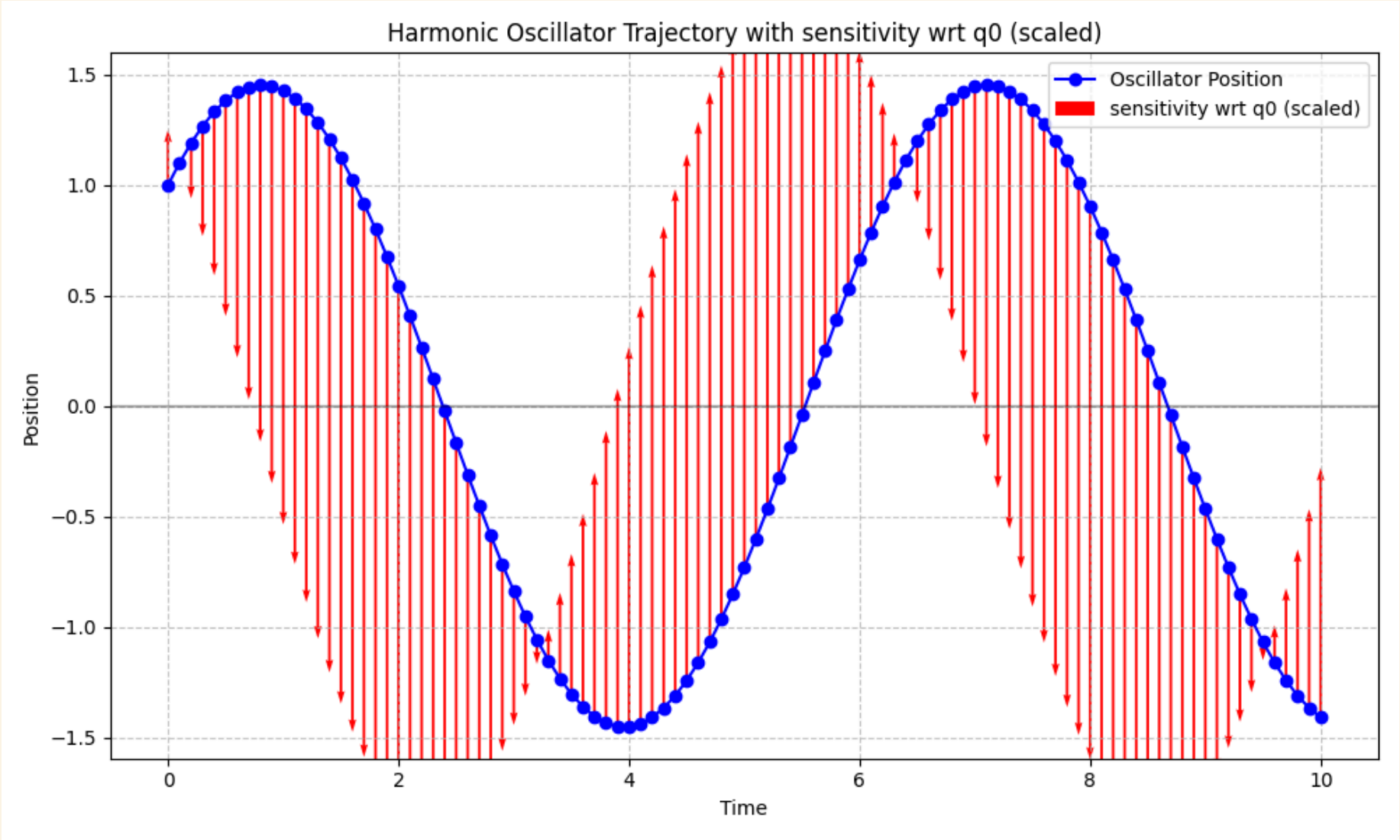
(DEL eqns, phys_params, h, N, q0, q1) ↦ trajectory

so let's compute

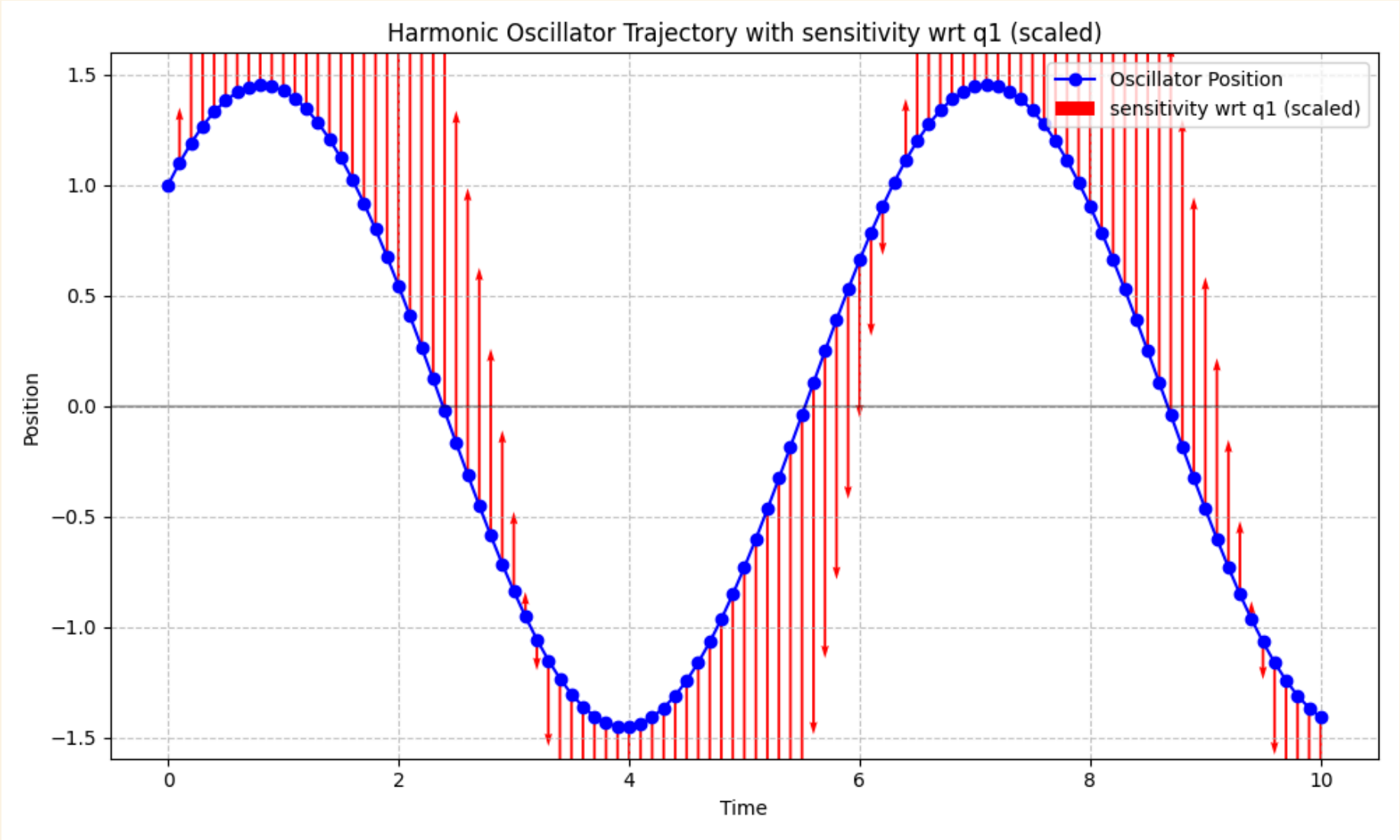
$$\frac{\partial \text{trajectory}}{\partial q_0}, \quad \frac{\partial \text{trajectory}}{\partial q_1}$$

```
partial_trajectory_partial_q0 = jax.jacfwd(
    integrator.perform_simulation, argnums=4
)
partial_trajectory_partial_q1 = jax.jacfwd(
    integrator.perform_simulation, argnums=5
)
```

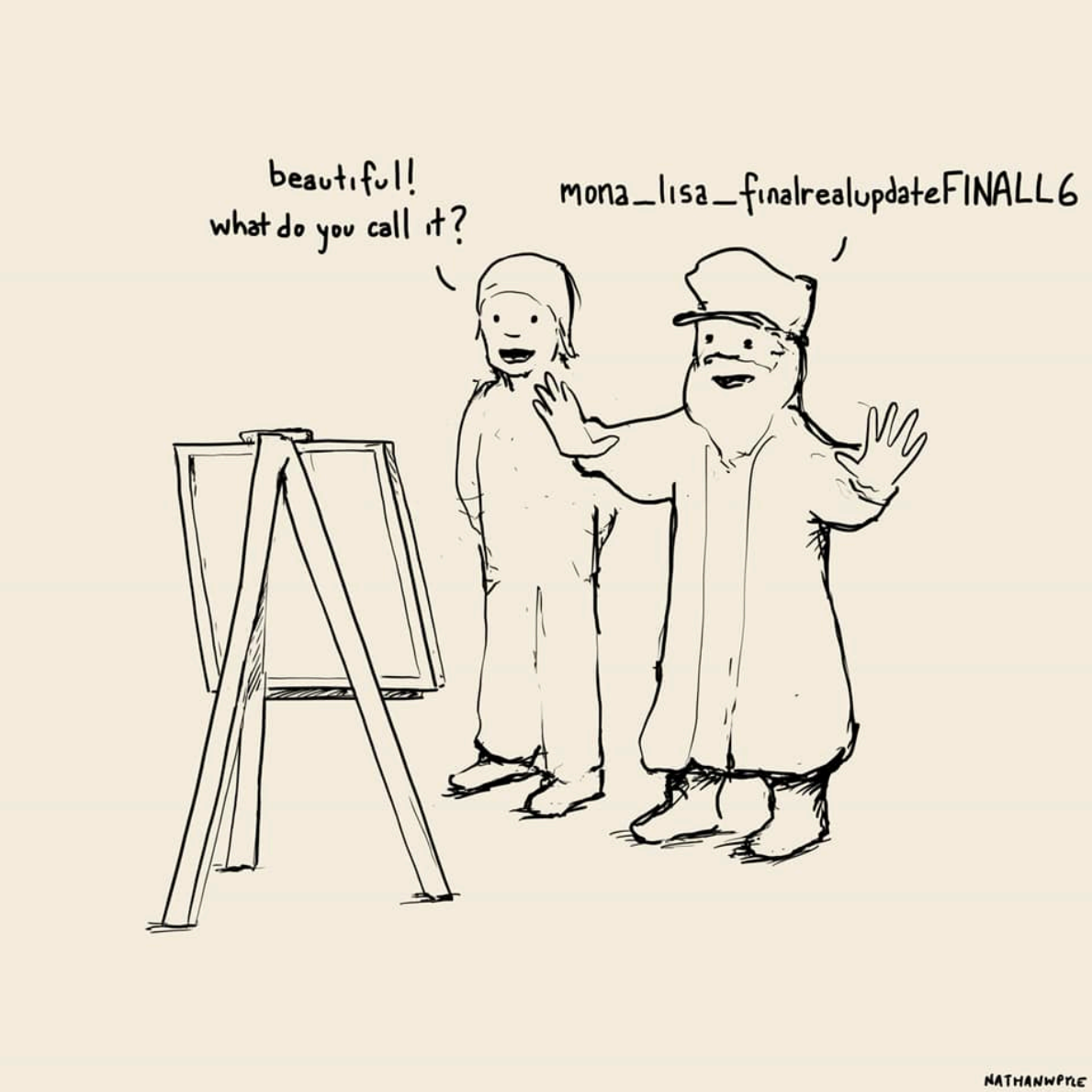
Trajectory sensitivity wrt q_0



Trajectory sensitivity wrt q_1



Version control



(Credit: Nathan W. Pyle)

How do we manage our code history?

- We had a first version using `sympy`
- We made a second one with `jax`
- We'll surely make some improvements in the future
- We want to save some intermediate versions
- How do we manage this?

Some things to take into account

- Code is more sensitive to modifications than, say, LaTeX files.
- Collaboration is an issue if everyone is editing the same files.
- We should feel free to experiment without worring about breaking the code.

Git

- Git is a distributed version control system.
- Created in 2005 by Linus Torvalds, of Linux fame.
- Mantained since 2005 by Junio Hamano.
- Free and open-source software.
- It solves many problems that existing software had.
- It’s the most widely used version control system,
- and the industry standard in software development.

Why you should use Git

For individual development

- Even for non-code, like LaTeX files
- Undo mistakes: reset to any previous state of your project
- Track changes: compare versions of files, not only the contents but also the behavior
  - See if performance has improved or worsened
  - Find which change made things stop working correctly
- Experiment safely: create branches to try new ideas without affecting your main work

More reasons

For team collaboration

- Each person can work on the same project simultaneously.
- The contributions are combined (“merged”) when you ask for that to happen, not automatically.
- Useful even for non-coding collaborators.
  - The current version, “the version we used for the paper”, etc, are always accessible

“Git is to writing software what LaTeX is to writing math.”

- There’s a learning curve, but everyone on the other side say it’s worth it.

Git and GitHub

- You probably have heard of GitHub.
- GitHub is a cloud service that *uses* Git, but it's not Git.
- You can use Git on your own computer (offline).
  - Git can "talk" to GitHub to upload and download data.
- You can use GitHub for personal backups and for collaboration.
  - Never upload passwords, etc! Even if you delete them, they are still accessible by people.
- There are other similar options (Gitea, GitLab, Bitbucket…)

What about Dropbox, GDrive… ?

- When you are working on code, the project often remains in a broken state until you finish rounding up your changes.
- You will probably duplicate the project into a new folder.
- You end up having several copies, and it’s very likely that you edit an old one by mistake.
- Dropbox lets you access earlier versions (just the last 30 days for free accounts), but every single save is stored.
- With Git, you can take snapshots when you want.
- Mixing Git and Dropbox is not recommended.

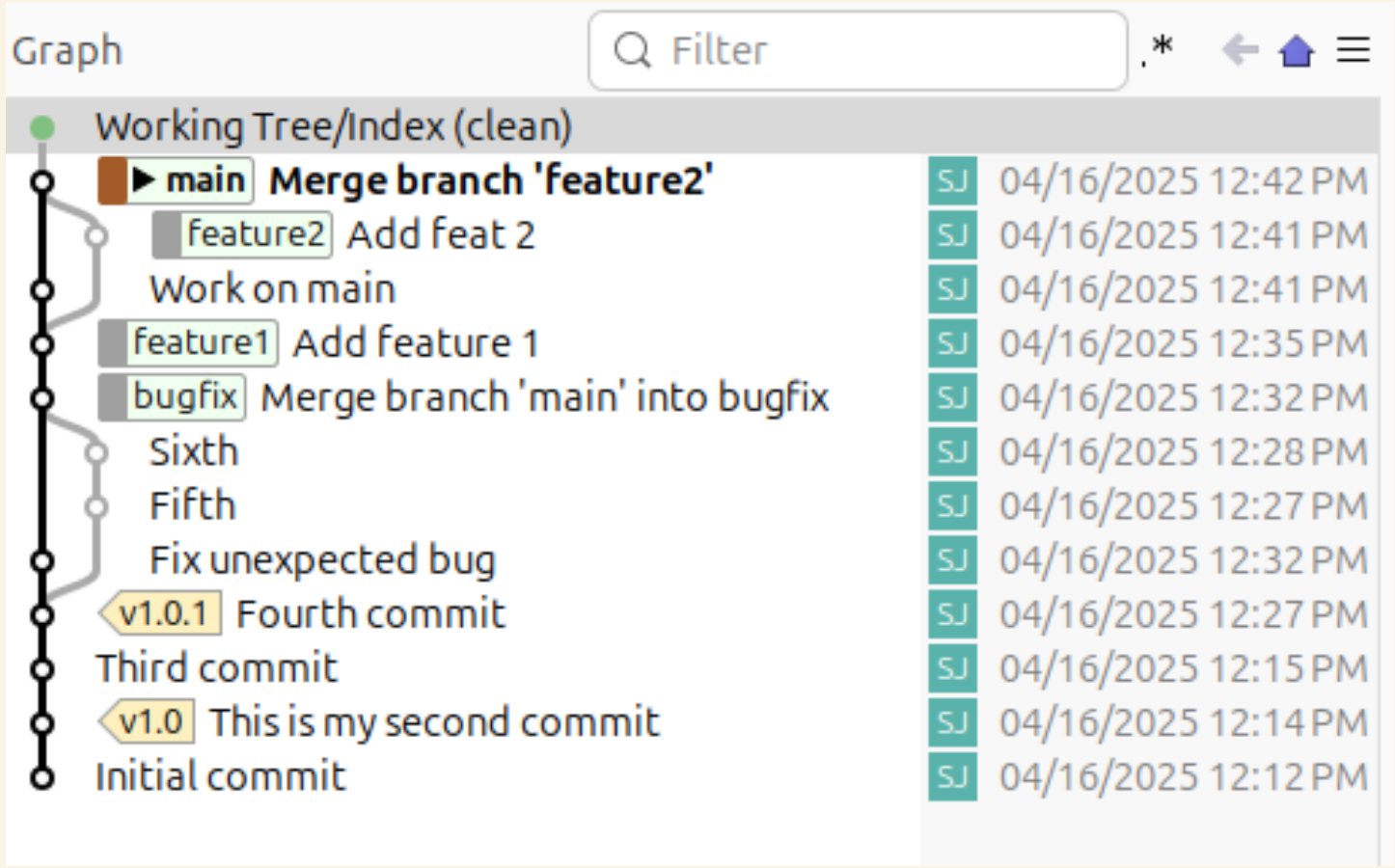
The Git model

- Git has a reputation for having a steep learning curve. However, the basic operation is quite simple.
- There are lots of pages and videos that teach you Git.
  - “If you want to do A, just run `git some_command`”
- Knowing how Git represents your files and history is crucial for understanding what you can do.

Your life is a directed acyclic graph

- Git’s version history is structured as a Directed Acyclic Graph (DAG), not a simple linear timeline.
- Each node represents a state of the project (a “commit”)
- Edges between nodes represent relationships between states, but the direction goes **back in history**!
  - Edges go from newer to older commits
  - If you take a specific state, you can access its whole previous history.
- No circular references in history.

Visualization of some Git graphs



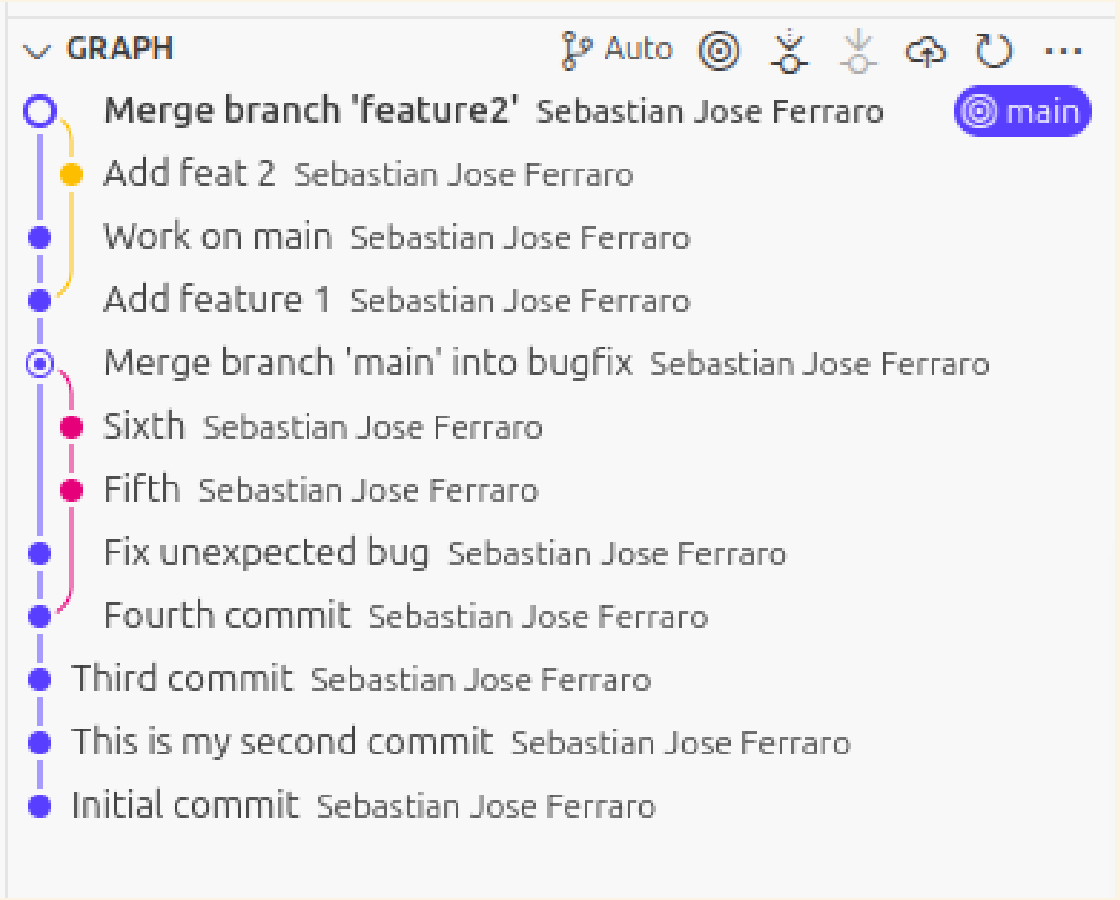
A simple repo opened in SmartGit. Time goes ↑, edges point ↓.

Visualization of some Git graphs

```
seba@seba-UX310UQ:~/Documents/github_repos_nobackup/trunk_based$ git log --all --graph --decorate --oneline
*   fe2e398 (HEAD -> main) Merge branch 'feature2'
|\
| * 95d3af1 (feature2) Add feat 2
* | a3dd707 Work on main
|/
* ce8224b (feature1) Add feature 1
*   9b4aaf1 (bugfix) Merge branch 'main' into bugfix
|\
| * 58fef85 Sixth
| * 9a11703 Fifth
* | ebbebd1 Fix unexpected bug
|/
* f2fb361 (tag: v1.0.1) Fourth commit
* 66e0b98 Third commit
* 33fedf7 (tag: v1.0) This is my second commit
* b8eb1e1 Initial commit
```

The same repo on the command line.

Visualization of some Git graphs

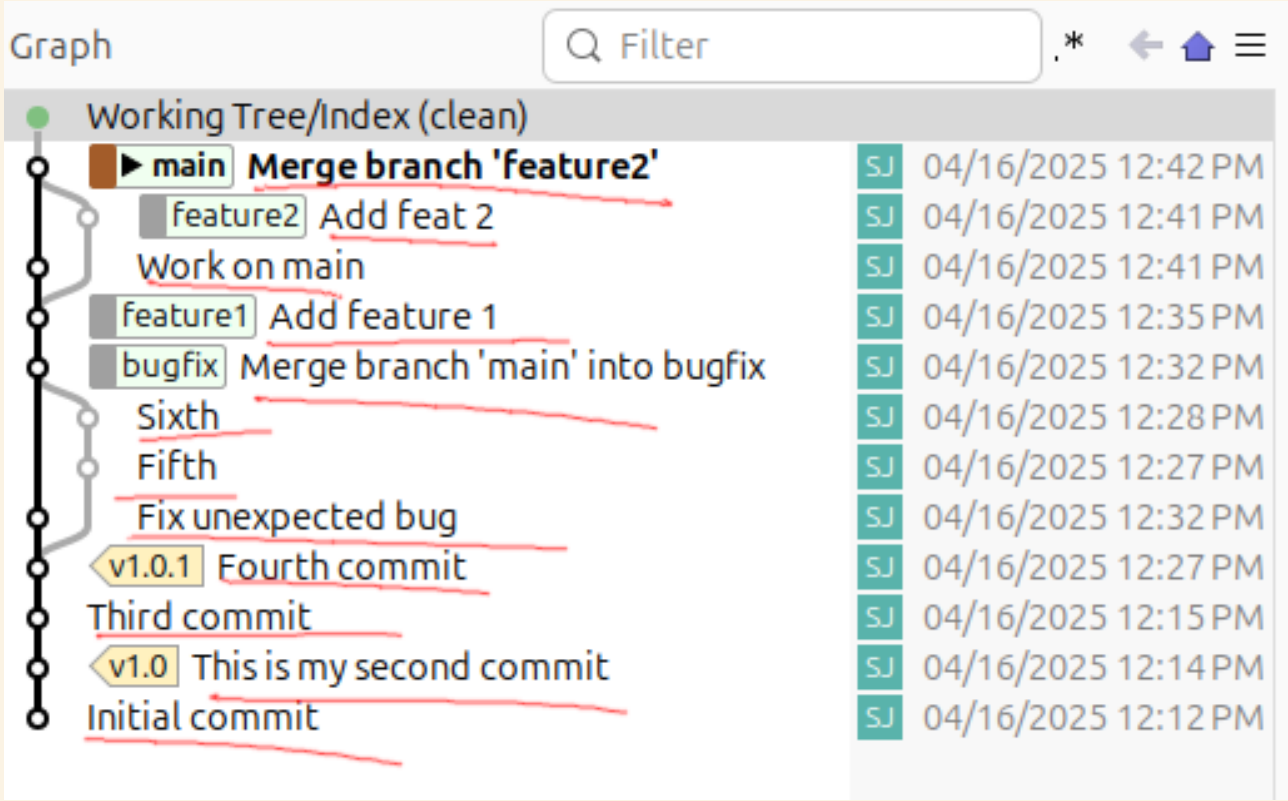


And in VS Code.

Commits (the nodes)

- A commit is a snapshot of your full project at the time of the commit. However, it is storage-efficient.
- It has a unique identifier: A 40-character SHA-1 hash (like 9b4aaf1a30e87967f366d3adc9f024cb3044c114) generated based on the content and metadata. Often abbreviated to the first few characters.
- It contains metadata, including author, date, and a message explaining the change.
- It has a pointer to the parent commit (or commits).
- It is immutable: cannot be changed without altering its ID.

Commit messages



Here you write something (hopefully) meaningful about the changes you made.

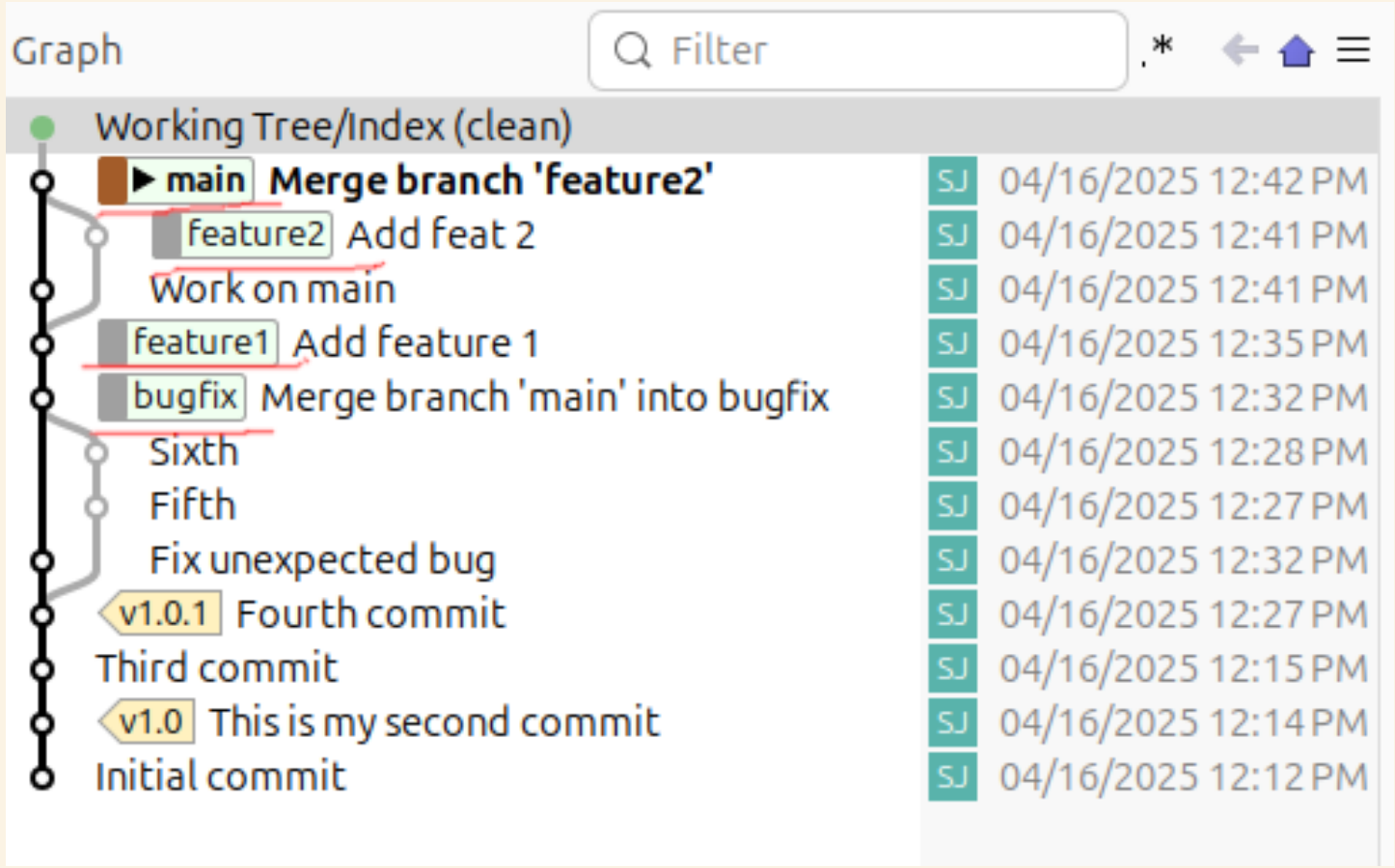
Branches

- Conceptually, branches represent different lines of development.
- They can diverge and merge.
- Branches are lightweight and can be created, deleted, and switched easily.
- For example, if you are working on transitioning from a `sympy` version to a `jax` version, you can create a branch called `sympy-to-jax`, work on that branch, and then merge it back into the main branch when you are done.

Branches

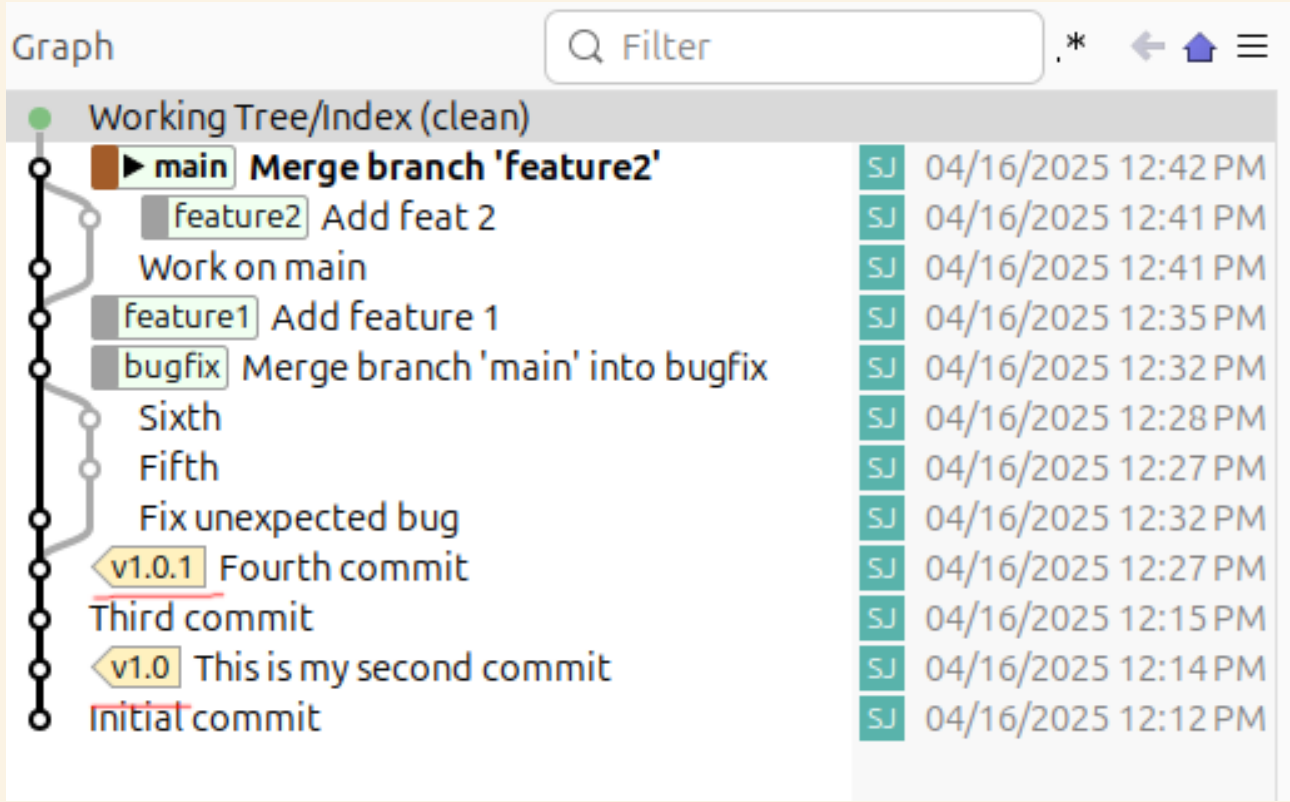
- When you create a new repository, it starts with a default branch, usually called `main`.
- A branch is a reference to a specific commit in the history.
- Merging branches creates a new commit that has two parent commits (the tips of the branches being merged).

Branch names



You can name branches anything you want.

Tags



Tags are like branches, but they are not meant to change. Often used to mark important points in history.

How to begin

- If this is a *new* repository (maybe with some files already in it), you can run `git init` to create a new Git repository.

```
seba@seba-UX310UQ:~/Documents$ mkdir summerschool_demo
seba@seba-UX310UQ:~/Documents$ cd summerschool_demo/
seba@seba-UX310UQ:~/Documents/summerschool_demo$ git init
Initialized empty Git repository in /home/seba/Documents/summerschool_demo/.git/
seba@seba-UX310UQ:~/Documents/summerschool_demo$
```

- If you want to *clone* (copy) an existing repository, you can run `git clone <url>` to create a local copy of the repository.
  - The URL can be a local path or a remote URL (like a GitHub repository).
  - We did that with the harmonic oscillator code.
- Now you have a `.git` folder in your project directory, which contains all the metadata and history of the repository.
  - Don’t modify this folder manually! (Here’s where Dropbox might interfere and cause problems.)

Contents of the `.git` folder

Among other things, it contains:

- The object database, which stores the contents of files and commits.
- The references (branches and tags).
  - These are very small files that contain the SHA-1 hash of the commit they point to.
  - In our example, we have for instance

```
.git/refs/heads/bugfix
```

```
9b4aaf1a30e87967f366d3adc9f024cb3044c114
```

- There is a special reference called `HEAD`, which points to the current branch or commit you are working on.

```
seba@seba-UX310UQ:~/Documents/github_repos_nobackup/trunk_based$ git log --all --graph --decorate --oneline
*   fe2e398 (HEAD -> main) Merge branch 'feature2'
|\
| * 95d3af1 (feature2) Add feat 2
* | a3dd707 Work on main
|/
* ce8224b (feature1) Add feature 1
*   9b4aaf1 (bugfix) Merge branch 'main' into bugfix
|\
| * 58fef85 Sixth
| * 9a11703 Fifth
* | ebbebd1 Fix unexpected bug
|/
* f2fb361 (tag: v1.0.1) Fourth commit
* 66e0b98 Third commit
* 33fedf7 (tag: v1.0) This is my second commit
* b8eb1e1 Initial commit
```

Working with git

I want to avoid giving you a list of commands. You can work with Git using the command line or a GUI tool. Some words you need to know:

- **Repository**: A Git repository (“repo”) is a directory that contains all the files and history of a project. It can be local (on your computer) or remote (on a server like GitHub).
- **Staging**: Before committing changes, you need to *stage* them. This means selecting which files you want to include in the next commit. Git calls this the “index” or “staging area”.

- **Committing**: After staging, you can commit the changes with a descriptive message.
- **Working tree**: The “usual” directory with your current working files, probably with uncommitted changes.
- **Pushing**: If you are working with a remote repository (like GitHub), you can send your commits to the remote repository.
- **Pulling**: Update your local repository with changes from the remote repository.

Experimenting with Git concepts

- A nice interactive visualization of commands:

  Explain Git with D3:

  https://onlywei.github.io/explain-git-with-d3/#commit

- A tutorial:

  Learn Git Branching:

  https://learngitbranching.js.org/

  (Here the graph is inverted, time goes downwards)

Guiding principles

- Commit early, commit often.
- Use meaningful commit messages.
  - For example, “Fix bug in harmonic oscillator simulation” is better than “Fix stuff”.
  - The recommended practice is to use the imperative mood, like “Fix bug” instead of “Fixed bug”.
- Use branches for new features, bug fixes, or experiments.
  - You can create a branch for each paper, presentation…

Assorted advice

- Non-canonical tips:
  - If Git is overwhelming, **the very least you should do** is to make a single-branch repo with your code. No branches, just a chain of commits.
  - If you worked for days without committing, forgive yourself and commit with a message like *“Add lots of stuff I can’t remember right now”*. It’s better than nothing. (Not recommended for collaboration.)
- Split your code into several files, the history will be easier to follow and changes will be confined to their own files.

Also…

- Don’t commit sensitive information (like passwords, API keys, etc) to GitHub. The whole commit history is accessible.
- Learn how to use `.gitignore` to exclude files and directories that you don’t want to commit.

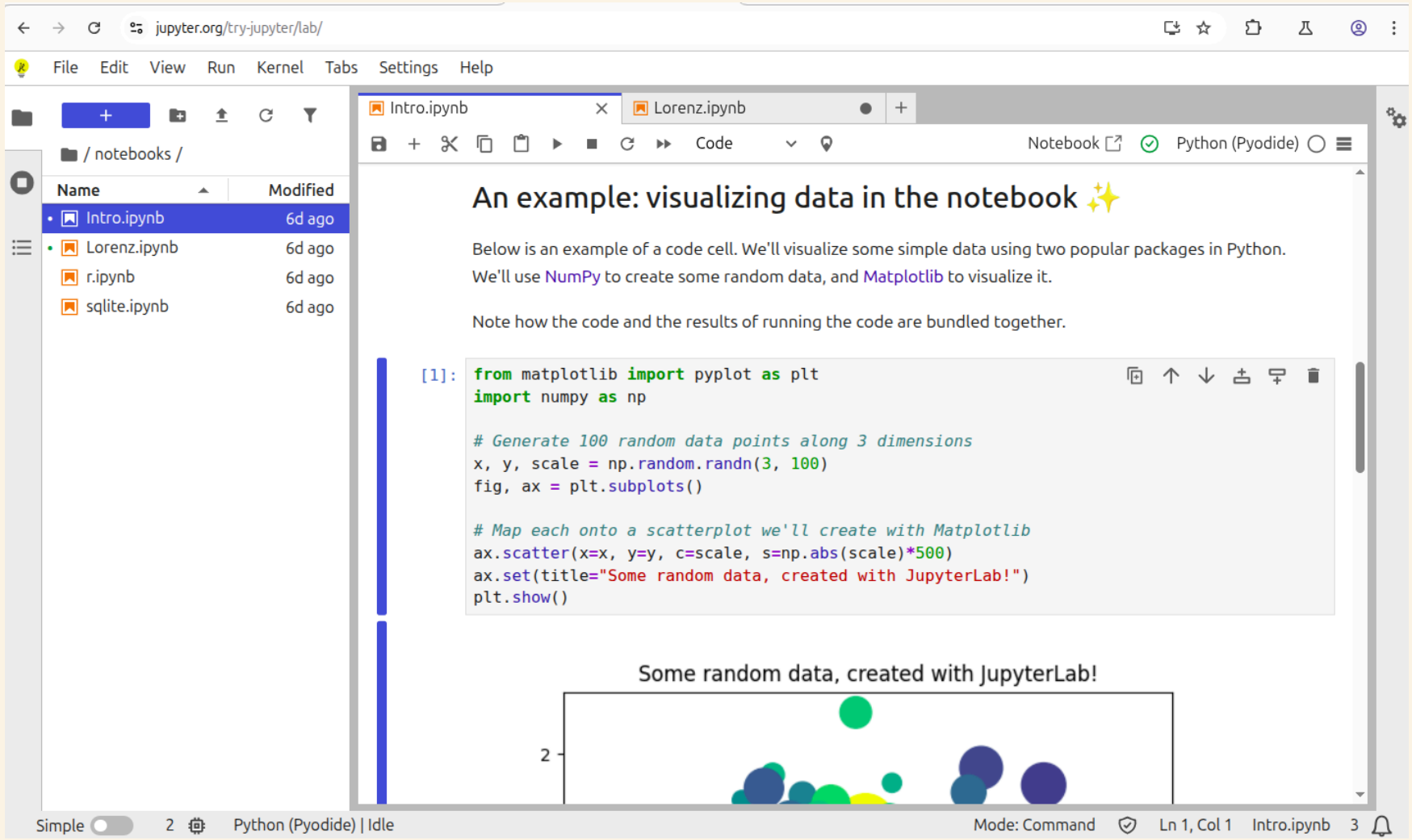
Git’s own git graph

Backups

But what about continuous backups (like Dropbox)?

- Git is not a backup system. It is a version control system.
- You should also have a backup strategy.
- For example, you can push your repository to a remote server (like GitHub) regularly.
- Something that worked for me is to make symlinks (symbolic links) of important files or folders into a Dropbox folder, as an emergency backup between commits.

Aside: Jupyter notebooks?



- So far, we have been using Python scripts (`.py` files, plain text).
- Jupyter notebooks (including Google Colab) are popular since you can mix code, text, and visualizations.
- They serve different purposes:
  - Scripts are better for writing reusable code (and for version control!)
  - Jupyter notebooks are better for interactive exploration and presentations

Further reading

- Pro Git book (official book, highly recommended)

  https://git-scm.com/book/en/v2
- Think like (a) Git (clarifies many concepts)

  https://think-like-a-git.net/
- The Git Parable (how Git's approach appears naturally)

  https://tom.preston-werner.com/2009/05/19/the-git-parable.html

Testing

Why testing?

- We want to be sure that our code works as expected.
  - No big deal, just be focused when writing it, right?
- When update/improve parts of the code, other parts may stop working correctly, and we need to detect this.
  - This is called *regression testing*.
- For example, we changed the harmonic oscillator simulation from `sympy` to `jax`. Did the results change?
- Testing is a way to automate this process, so we don't have to remember to do it manually every time we change something.

Testing in Python

We'll use the package `pytest`, a popular testing framework.

toy_example.py

```
def add(a, b):
    return a + b

def add_many_times(a, b, N):
    result = a
    for _ in range(N):
        result = add(result, b)
    return result

def main():
    a, b, N = 1, 2, 30
    result = add_many_times(a, b, N)
    print(f"{a} plus {N} times {b} is {result}")

if __name__ == "__main__":
    main()
```

```
1 plus 30 times 2 is 61
```

Preparing the tests

The file containing the tests should start with `test_` or end with `_test.py`, for automatic discovery by `pytest`. Also, the functions should start with `test_`.

test_toy_example.py

```
from toy_example import add, add_many_times

def test_add():
    assert add(1, 1) == 2
    assert add(-1.0, 1.0) == 0.0
    assert add(1, 2.5) == 3.5    # We know this will be exact!
    assert add([1, 2], [3, 4]) == [1, 2, 3, 4]

def test_add_many_times():
    assert add_many_times(1, 2, 0) == 1
    assert add_many_times(1, 2, 10) == 21
    assert add_many_times(1, 2.5, 10) == 26.0
```

Running the tests

- Just execute `pytest` in the terminal. It will automatically discover and run all the tests in files that match the naming conventions (this is called “test discovery”).

```
(toy-example) seba@seba-UX310UQ:~/Documents/github_repos/toy_example$ pytest
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-8.4.0, pluggy-1.6.0
rootdir: /home/seba/Documents/github_repos/toy_example
configfile: pyproject.toml
collected 2 items

test_toy_functions.py ..                                                 [100%]

============================== 2 passed in 0.01s ===============================
```

The two green dots indicate that both tests passed.

Keeping the code tested

- Write tests as you write the code, including “extreme” cases.
- Run tests frequently, especially after making changes.
- Ideally, you should test every single function, not just the “top-level” ones.

Repetitive tests?

If you have many similar tests, you can use `pytest.mark.parametrize` to run the same test with different inputs.

- You give a list of tuples with the inputs and expected outputs.
- `pytest` will run the test for each tuple, and report if any of them fail.
- Each tuple is a different test case.

Parametrization

test_toy_example.py

```
1  from toy_example import add, add_many_times
2  import pytest
3
4  @pytest.mark.parametrize("a, b, expected", [
5      (1, 1, 2),
6      (-1.0, 1.0, 0.0),
7      (1, 2.5, 3.5),
8      ([1, 2], [3, 4], [1, 2, 3, 4])
9  ])
10 def test_add(a, b, expected):
11     assert add(a, b) == expected
12
13 @pytest.mark.parametrize("a, b, N, expected", [
14     (1, 2, 0, 1),
15     (1, 2, 10, 21),
16     (1, 2.5, 10, 26.0)
17 ])
18 def test_add_many_times(a, b, N, expected):
19     assert add_many_times(a, b, N) == expected
```

Aside: decorators

- A decorator takes a function and returns another function

Instead of the generic syntax

```
def my_decorator(func):
    decorated_f = build_some_new_function_using(func)
    return decorated_f

def g(x):
    return x + 1

g = my_decorator(g)
```

there's the special syntax

```
@my_decorator
def g(x):
    return x + 1
```

Running the tests again

```
(toy-example) seba@seba-UX310UQ:~/Documents/github_repos/toy_example$ pytest
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-8.4.0, pluggy-1.6.0
rootdir: /home/seba/Documents/github_repos/toy_example
configfile: pyproject.toml
collected 7 items

test_toy_functions.py .......                                            [100%]

============================== 7 passed in 0.02s ===============================
```

Let’s make a change

Change the `add` function to this:

toy_example.py

```
def add(a, b):
    return a - (-b)
```

and run `pytest` again.

```
collected 7 items

test_toy_functions.py ...F...                                            [100%]

================================== FAILURES ===================================
_____________________ test_add[a3-b3-expected3] _____________________

a = [1, 2], b = [3, 4], expected = [1, 2, 3, 4]

    @pytest.mark.parametrize("a, b, expected", [
        (1, 1, 2),
        (-1.0, 1.0, 0.0),
        (1, 2.5, 3.5),
        ([1, 2], [3, 4], [1, 2, 3, 4])
    ])
    def test_add(a, b, expected):
>       assert add(a, b) == expected
               ^^^^^^^^^

test_toy_functions.py:11:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = [1, 2], b = [3, 4]

    def add(a, b):
>       return a - (-b)
                 ^^
E       TypeError: bad operand type for unary -: 'list'

toy_example.py:2: TypeError
=========================== short test summary info ===========================
FAILED test_toy_functions.py::test_add[a3-b3-expected3] - TypeError: bad operand
 type for unary -: 'list'
======================== 1 failed, 6 passed in 0.07s ==========================
```

What about simulation results?

- We cannot just write down what the results should be. What trajectory values should we expect from the harmonic oscillator with our initial conditions?

- Yet we want to be able to compare the results of the simulation before and after making changes.

- The approach is to save the results of the simulation in a file, and then compare the new results with the saved ones.

- Fortunately, we do not need to do this manually: we can use a `pytest` plugin called `pytest-regressions`.

`pytest-regressions`

https://pytest-regressions.readthedocs.io/en/latest/overview.html

- Helps you make tests that involve the generation of data files, ndarrays, images, etc.
- The first time you run the tests, it will create a “baseline” file with the results.
- The next time you run the tests, it will compare the new results against the baseline file.

Parametrization

- As before, you can run the same test with different inputs using `pytest.mark.parametrize`.
  - For example, you can test the harmonic oscillator with different initial conditions or physical parameters.
- You can nest parametrizations.

test_harm_osc.py

```
import harm_osc_system_sympy as the_system  # Switch sympy and jax versions her
import integrator
import pytest
@pytest.mark.parametrize(
    "phys_params",
    [
        {"m": 1.0, "k": 1.0},
        {"m": 2.0, "k": 1.0},
    ],
)
@pytest.mark.parametrize(
    "h, N, init_cond",
    [
        (0.1, 100, [0.0, 0.0]),
        (0.1, 20, [1.0, 1.1]),
        (0.01, 100, [1.0, 1.1]),
    ],
)
def test_harmonic_oscillator(phys_params, h, N, init_cond, ndarrays_regression)
    trajectory = integrator.perform_simulation(
        the_system.f_DEL_q2_first, phys_params, h, N, init_cond
    )
    ndarrays_regression.check({"values": trajectory})
```

Test the `sympy` version

```
(harm-osc-ss2025) seba@seba-UX310UQ:~/Documents/github_repos/harm_osc_ss2025$ pytest
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-8.4.1, pluggy-1.6.0
rootdir: /home/seba/Documents/github_repos/harm_osc_ss2025
configfile: pyproject.toml
plugins: regressions-2.8.0, datadir-1.7.2
collected 6 items

test_harm_osc.py ......                                                  [100%]

============================== 6 passed in 1.70s ===============================
```

This is actually the second run. The first run always *fails* because it creates the baseline file. (This is expected.)

If you want to try it yourself, run `source .venv/bin/activate` before.

Retest with the `jax` version

Change the import to `import harm_osc_system_jax as the_system` and run the tests again.

```
(harm-osc-ss2025) seba@seba-UX310UQ:~/Documents/github_repos/harm_osc_ss2025$ pytest
============================= test session starts ==============================
platform linux -- Python 3.13.3, pytest-8.4.1, pluggy-1.6.0
rootdir: /home/seba/Documents/github_repos/harm_osc_ss2025
configfile: pyproject.toml
plugins: regressions-2.8.0, datadir-1.7.2
collected 6 items

test_harm_osc.py ......                                                  [100%]

============================== 6 passed in 4.32s ===============================
```

All good!

Careful with the floats!

- The results are arrays of floating point numbers, so `pytest-regressions` does not check for exact equality.
- It uses `np.allclose` to compare the arrays (but you can change the tolerance).
- JAX uses `float32` by default, while `sympy`+`numpy` use `float64`, so you may get “false failures”!
  - `float32` numbers get upgraded to `float64` for the comparison.
  - The binary “decimals” are completed with zeros.

A false failure

With $h = 1, N = 100, m = k = 1$ and $q_0 = 1, q_1 = 1.1$:

```
E       AssertionError: Values are not sufficiently close.
E       To update values, use --force-regen option.
E
E       values:
E         Shape: (101,)
E         Number of differences: 2 / 101 (2.0%)
E         Statistics are computed for differing elements only.
E         Stats for abs(obtained - expected):
E           Max:     2.97866684406331e-07
E           Mean:    2.400311227652649e-07
E           Median:  2.400311227652649e-07
E         Stats for abs(obtained - expected) / abs(expected):
E           Max:     4.6160897377383616e-05
E           Mean:    3.018832322791304e-05
E           Median:  3.018832322791304e-05
E         Individual errors:
E                 Index              Obtained              Expected            Difference
E                    26  -0.012816640907574414  -0.01281645871201329  -1.8219556112419877e-07
E                    87   0.006452494460361425  0.006452792327045831  -2.97866684406331e-07

test_harm_osc.py:28: AssertionError
```

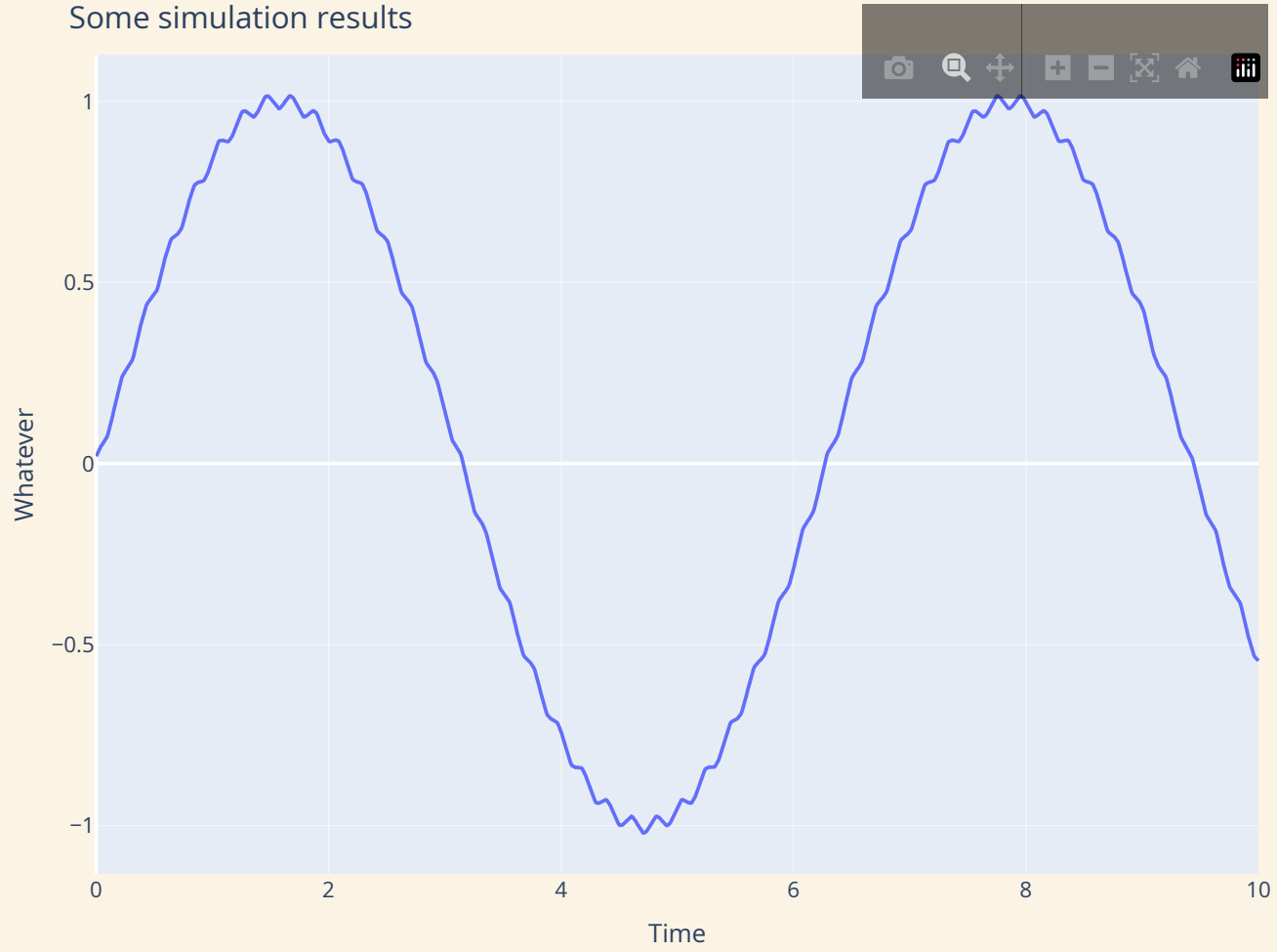
`pytest` fixtures

- For extra flexibility and modularity, you can use something called *fixtures*.
- We won't go into details here, but you can look it up in the `pytest` documentation.
- Fixtures allow you to define reusable pieces of code that can be used in multiple tests.
- In this way, you can arrange the context in which the tests run. For example, you can do a *long simulation once* and reuse the result in *multiple tests*.

Visualization and analysis

- Recommended packages for visualization:
  - `matplotlib`: mature, stable, widely used. Good for static plots. Very customizable.
  - `plotly`: interactive, web-based, good for presentations.

An interactive plot with `plotly`



A 3D plot with `plotly`

Another one


WebGL is not supported by your browser - visit https://get.webgl.org for more info

Tips

- Separate the actual simulation code from the visualization or analysis code.
- The simulation should save the results in a file:
  - Use the `np.save` and `np.load` functions to save and load arrays.
  - Also `np.savez` and `np.savez_compressed` to save multiple arrays in a single file.
  - See also the `pickle` module for saving and loading arbitrary Python objects.

More tips

- Very long simulations can benefit from saving intermediate results.
  - For example, make a button to save the current state of the simulation, along with whatever data you need to resume it later.
  - Or save the results every certain number of steps.
- When making a figure, don't include a bazillion points!
  - `matplotlib` and `plotly` will downsample large data automatically.
- Both packages can do animations as well.

General advice

- Document your code.
  - Use comments and docstrings (look it up).
- Make it clear to yourself and to others how to run the simulations.
  - For example, write a README file with instructions.
  - Do not just walk away once it works.
- Use breakpoints and learn how to debug (look into the `logging` module, for example).

Vectorization and parallelization

- These are two related but slightly different concepts.
- Vectorization: Instead of loops, use a single operation that applies to all elements of an array.
  - The underlying libraries (`numpy`, `jax`, etc) will use optimized code to perform the operation.
- Parallelization: Split the computation into smaller tasks that can be executed *simultaneously*.
  - This is done with multiple CPU cores, GPUs or TPUs.
- `jax` can do both automatically.

Basic idea of vectorization

- If $f\colon A \to B$, we can think of the family of functions

$$f^k = (f, \dots, f)\colon A^k \to B^k$$

  that is, $f^k(x_1, \dots, x_k) = (f(x_1), \dots, f(x_k))$.
- The vectorization of f is essentially this family of functions.
- It can accept any number of inputs, and returns as many outputs.
- (It chooses a function from the family depending on the number of inputs.)

- But if you have

$$f\colon A_0 \times \cdots \times A_n \to B$$

  you can choose to vectorize along one of the factors, that is, you vectorize $f(a_0, \ldots, \bullet, \ldots, a_n)\colon A_i \to B$.

- In other words,

$$\begin{aligned} f^k\colon A_0 \times \cdots \times (A_i)^k \times \cdots \times A_n &\to B^k, \\ (a_0, \ldots, (a_i^1, \ldots, a_i^k), \ldots, (a_n)) &\mapsto \\ (f(a_0, \ldots, a_i^1, \ldots, a_n), \ldots, f(a_0, \ldots, a_i^k, \ldots, a_n)). & \end{aligned}$$

- You can even do this along several factors at the same time.

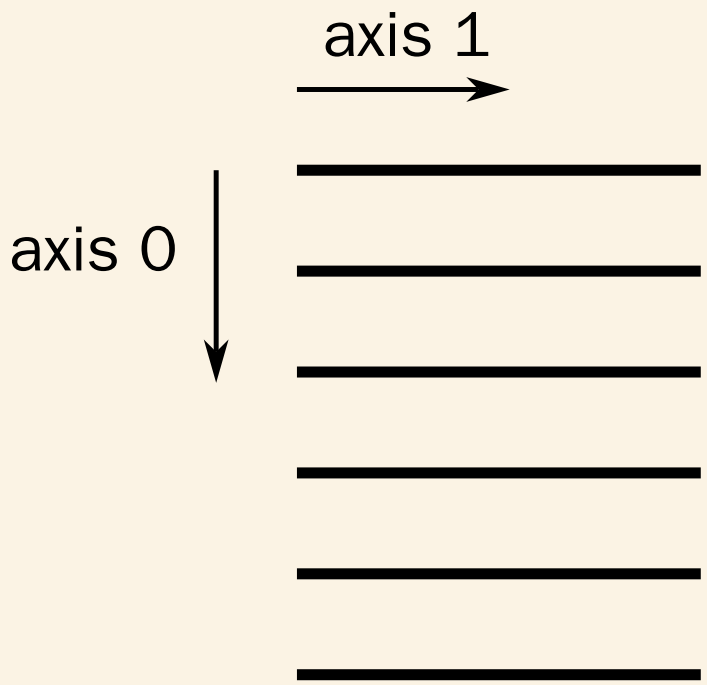
JAX's vectorizing map

- JAX has a function called `vmap` ("vectorizing map") for that.
- You have a function that takes several arguments, and you want to apply it to many different values of one of the arguments while keeping the others fixed.
- For example, you want to run a simulation with many different initial conditions, but the same physical parameters, time step, and total number of steps.

- Suppose you have `f(x0, ..., xn)`,
- where each argument is an ndarray.
- Choose one of the arguments, say `xn` for definiteness.
- And let's say, also for definiteness, that `xn` is 1-D (remember: not a column, not a row).
- So you have for example something that looks like `f(matrix, matrix, vector, vector)`, and returns a matrix.
- You want to apply `f` to many different values of `xn`, while keeping the other arguments fixed.

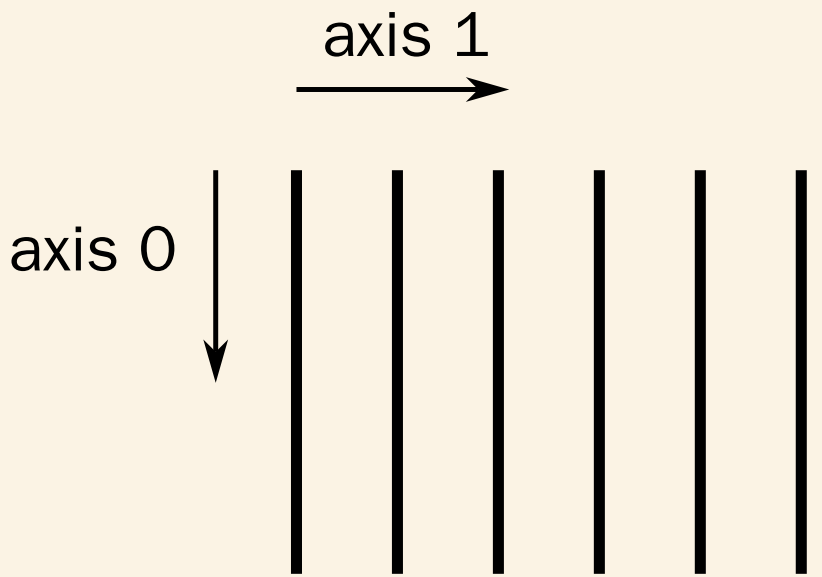
- You prepare a “batch” of values for `xn`. You stack all those k vectors along a new axis, which can be 0 (the default) or 1. Your choice.
  - `batch = jnp.stack([vector1, vector2, ..., vectork], axis=0)` or `axis=1`.
  - If you want, think of rows and columns, but thinking in terms of axes is less confusing.

The batch



stacked along axis 0

or

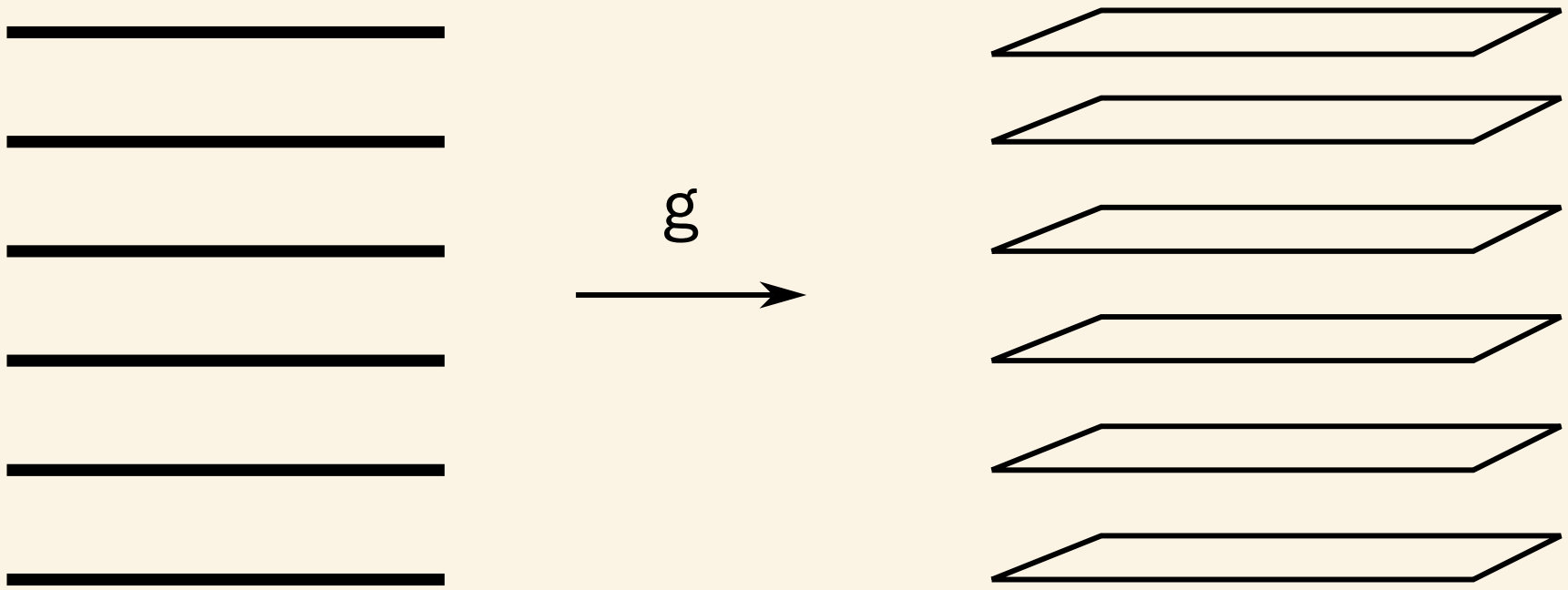


stacked along axis 1

The `vmap` function

- `g = jax.vmap(f, in_axes=(None, None, None, 0))` makes a new function `g` that does the following:
- Calling `g(matrix, matrix, vector, batch)` will accept your batch, and you have told it that the first three arguments are fixed (`None`), and the last one has your data stacked along axis 0 (or use `(None, None, None, 1)` if you chose axis 1).
- Since the output of our `f` is a 2D array, `g` will return the collection of all the outputs, in stacked form, as a 3D array.

Vectorized function output



(with the rest of the arguments fixed)

In a sense, `g` “unstacks” the batch and applies `f` individually.

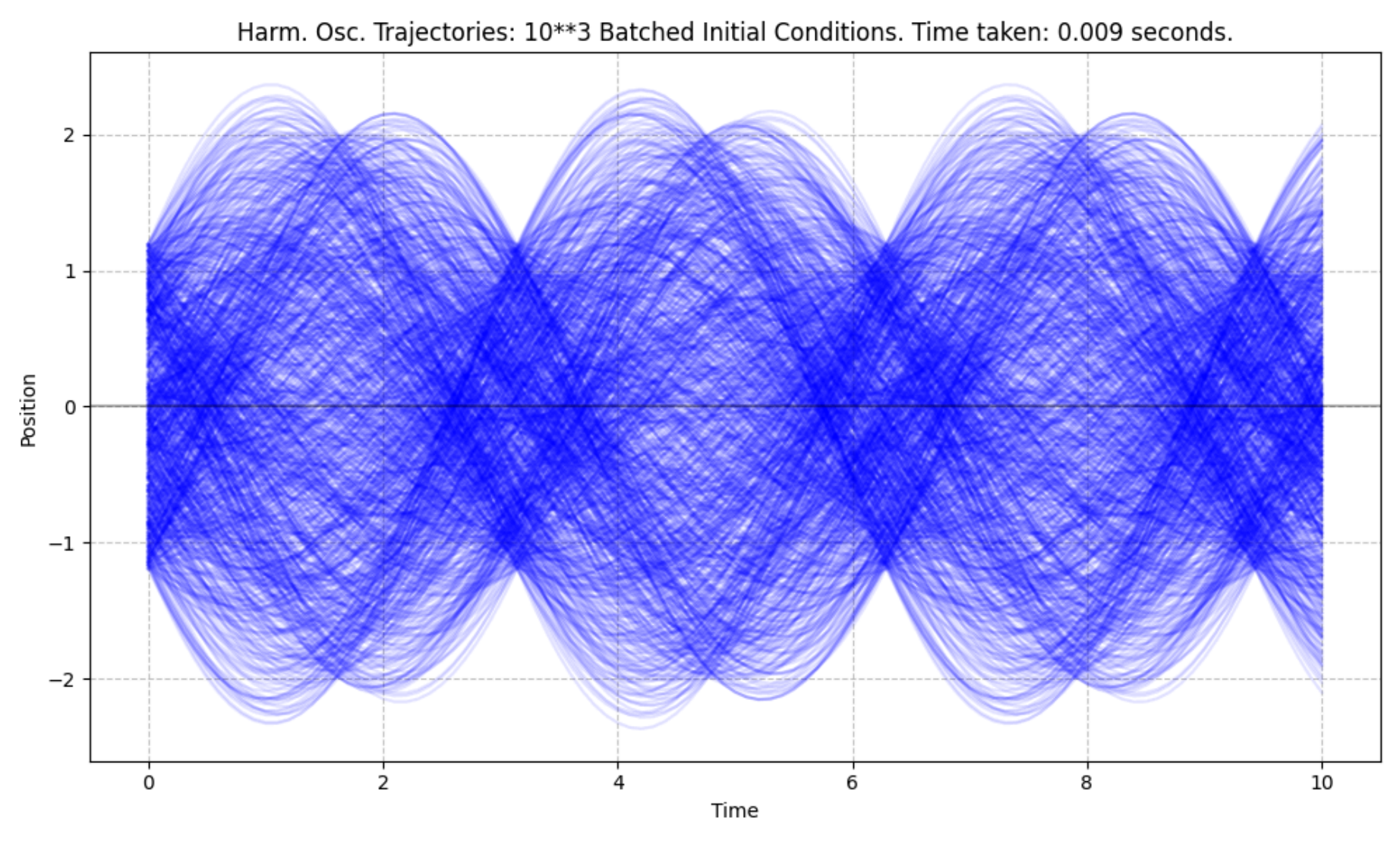
Vectorizing along multiple arguments

- Take our `f(matrix, matrix, vector, vector)`
- If you do `g = jax.vmap(f)`, it will vectorize `f` with respect to all the arguments, each one stacked along axis 0 by default.
- You'll get all the combinations of results, in a suitably-shaped ndarray.

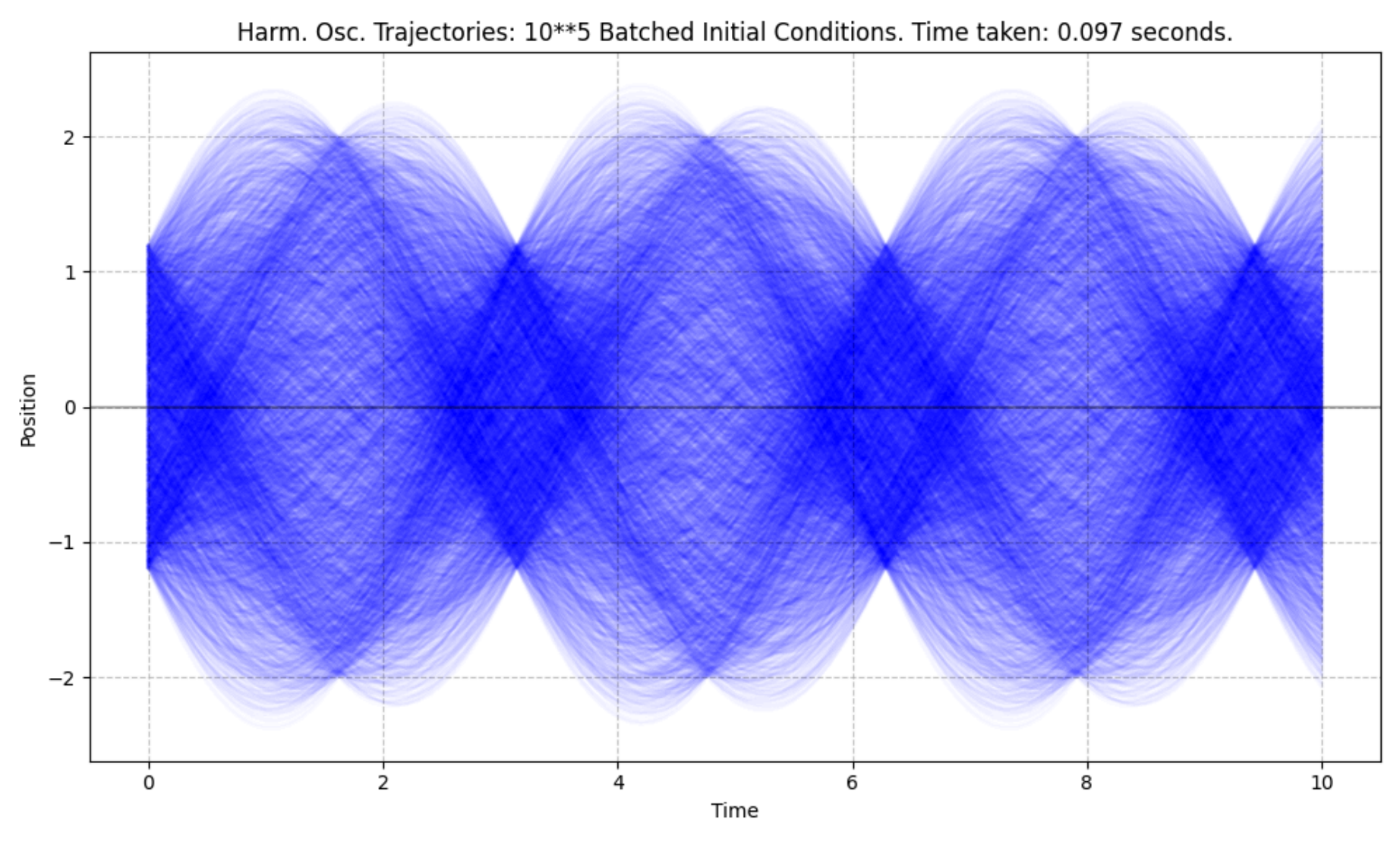
“2D” batching

- To work with a 2D batch of the same argument, you can apply `vmap` twice.

Batched random initial conditions



Batched random initial conditions



Time to talk about JIT

- (Just-In-Time compilation)
- In our code (`integrator.py`), we have the lines

```
from jax import jit

# (... define the perform_simulation function ...)
perform_simulation = jit(
    perform_simulation,
    static_argnames=["f_DEL_q2_first", "N"],
)
# (...)
```

What does `jit` do?

- It *prepares* the function for compilation into very efficient code (adapted to your GPU if you have one!).
- The actual compilation is triggered when the function is called.
- If you call the “jitted” function again but the inputs have a *different array shape* than before (e.g. a new batch with more or less data), it gets compiled again.
- Try to jit functions whenever you can…
- … but read the documentation (https://docs.jax.dev/) because there are several things to take into account.

Too slow?

Why is my “jitted” JAX code using all my CPU and memory, and why doesn’t it even start? Wasn’t it supposed to be fast?

- Chances are, you have a long `for` loop somewhere, and that’s not `jit`-friendly: the loop gets *unrolled*.

- For instance, if you try to `jit` this:

```
for i in range(20):
    for j in range(50):
        do_something(i, j)
```

it will be compiled into 1000 consecutive calls to `do_something`!

So what should we do instead?

- Use `jax.lax.fori_loop` or `jax.lax.scan` instead of `for`.
- These can handle the kind of loops that usually appear in simulations:
  - Initialize some variable,
  - update it in each iteration,
  - return the value at *all* the steps (or just keep the final one).

Any other similar substitutions?

Python construct	JAX equivalent
`for`	`jax.lax.fori_loop` or `jax.lax.scan`
`if`	`jax.lax.cond`
`while`	`jax.lax.while_loop`

For example: a conditional

```
import jax

def absolutevalue(x):
    return jax.lax.cond(x >= 0, lambda x: x, lambda x: -x, x)
```

(condition, function if true, function if false, argument)

Let’s take a derivative:

```
print(jax.grad(absolutevalue)(0.0))
```

And the answer is…

```
1.0
```

Quick ideas for structuring your code with classes

Recommended reading

- Object-oriented Programming in Python for Mathematicians (online book), by David A. Ham.

  https://object-oriented-python.github.io/index.html

  Some examples are from this book.

Classes

- Classes allow you to group together data (“attributes”) and functions (“methods”).
- Think of them as new custom data types.
- You can create objects (“instances”) of a class.

For example:

```
class Polynomial:
    def __init__(self, coeffs):
        self.coefficients = coeffs

    def degree(self):
        return len(self.coefficients) - 1


f = Polynomial((0, 1, 2))
print(f.degree())
```

```
2
```

We can add code to tell the class how to sum and multiply polynomials.

Inheritance

- You can create a hierarchy of classes.
- For example, you can have a `Manifold` class and a `Group` class, and then create a `LieGroup` class that inherits from both.

```
class Manifold:
    # (... define the manifold class ...)

class Group:
    # (... define the group class ...)

class LieGroup(Manifold, Group):
    # (... define the Lie group class ...)
```

- Or you can create a `SymplecticManifold` class that inherits from `Manifold` and adds the symplectic form and some related operations.

```
class SymplecticManifold(Manifold):
    # (...)
```

- Inheritance represents an “is-a” relationship (a `LieGroup` is a `Manifold` and a `Group`, etc.).

- You don’t have to use classes in this way for building your equations of motion, but it illustrates inheritance in a context that is familiar in mathematics.
- You can maybe use JAX to compute derivatives for your `Manifold` class.

Composition

It consists in building a complex object or operation out of several components.

- We know about composition for operations.
- We can compose objects by making one object an attribute of another one.
  - For example, the `Polynomial` class has the coefficients as an attribute.
  - Or you can create an `Point` class that has a `Manifold` attribute, meaning that it has a manifold associated with it.

```
class Point:
    def __init__(self, coordinates, manifold):
        self.coordinates = coordinates
        self.manifold = manifold
```

(maybe the coordinates consist of some numbers plus some `Chart` object!)

- Composition represents a “has-a” relationship (a `Point` has a `Manifold`, a `Polynomial` has coefficients…).

A different use for classes

- For example, you can make a `LagrangianSystem` class (for continuous systems!)
  - It has the Lagrangian as an attribute, and a method to compute the continuous Euler-Lagrange equations in some form (say as a second-order vector field).
  - Every instance can have its own Lagrangian.

```
class LagrangianSystem:
    def __init__(self, L):
        self.L = L
    def euler_lagrange(self, q, v):
        # Compute the left-hand side of the Euler-Lagrange equations
```

(This is just the skeleton of an example!)

- Maybe you can add a “plot” method that takes a configuration and returns a plottable object (these are called “artists” in `matplotlib`).
- So for example you can have

```
# suppose you have defined all the arguments below
pendulum = LagrangianSystem(L, my_pendulum_artist, physical_params)
eqn = pendulum.euler_lagrange(q, v)
artist = pendulum.plot_configuration(q)
```

(This doesn’t do much anyway)

A discrete version

- You can also have a `DiscreteLagrangianSystem` class, that has a:
  - Discrete Lagrangian
  - Method to compute the discrete Euler-Lagrange equations in some form
  - Method to plot the discrete configuration (maybe using the same artist as the continuous version, if it is related to one)

- You can add discretization methods to the `LagrangianSystem` class to produce a `DiscreteLagrangianSystem` object.

```
class LagrangianSystem:
    # (... existing code ...)

    def midpoint_discretization(self, h):
        def Ld(q0, q1):
            return h * self.L((q0 + q1) / 2, (q1 - q0) / h)

        return DiscreteLagrangianSystem(
            Ld, self.artist, self.physical_params
        )
```

Your main program

could look like this (at this point, this is already pseudocode!):

```
# (again, suppose that all the arguments below have already been defined)

discrete_pendulum = pendulum.midpoint_discretization(h)
f_DEL_q2_first = discrete_pendulum.DEL_equations_q2_first

trajectory = integrator.perform_simulation(
    f_DEL_q2_first, discrete_pendulum.physical_params, h, N, init_cond
)

# Make an animation
for i in range(N):
    make_an_animation_frame_using(
      discrete_pendulum.artist(trajectory[i])
    )
```

The end