Activity 3 - Free Induction Decay (3 hours)

✏ **Task 0:** Create a new folder `BlochFromScratch` and activate its environment. Create a file `scripts.jl` inside it to start coding.

Magnetic Resonance Imaging (MRI)

For this activity, we will simulate the Bloch equations, which represent the physics of the magnetic resonance phenomena:

$$\frac{\mathrm{d}M}{\mathrm{d}t} = \gamma M \times B - \frac{M_x \hat{x} + M_y \hat{y}}{T_2} - \frac{M_z - M_0}{T_1} \hat{z}.$$

We will consider a simple MRI experiment, where $B = (0, 0, B_z)$ and $M(0) = (M_0, 0, 0)$.

This experiment is also called **Free Induction Decay** (FID), and we will expect to see $M(t)$ precessing and decaying around the z-axis.

All the simulations will cover the time interval $t \in [0, 3]\,\mathrm{s}$. Other parameters include:

- $\gamma = 2\pi \cdot 42.58 \cdot 10^6\,\mathrm{rad/(s \cdot T)}$ (use the Unicode `π` 😉)
- $M_0 = 1$
- $T_1 = 1\,\mathrm{s}$
- $T_2 = 0.5\,\mathrm{s}$
- $B_z = 10^{-7}\,\mathrm{T}$

Numerical solutions to differential equations

To solve the Bloch equations, we will write a function `solve`, which will progress the state of the magnetization `m` from its initial state `m0`:

```julia
function solve(m0, dt, tmax, method)
    Nsteps = ...
    m = ...
    mt = ...
    for i in 1:Nsteps
        m = step(dt, m, method)
        mt[:, i] = m
    end
    return mt
end
```

This function will behave differently depending on the *type* of the input argument `method`. Inside `solve`, the `step` function will be in charge of updating our magnetization

$$M_{i+1} = M_i + \int_{t_i}^{t_{i+1}} \mathrm{bloch}(M(t))\,\mathrm{d}t,$$

where `bloch` is a function that calculates the right-hand side of the Bloch equations. To generate numerical methods, we will change the way `step` is integrating in the previous expression (Figure 1).

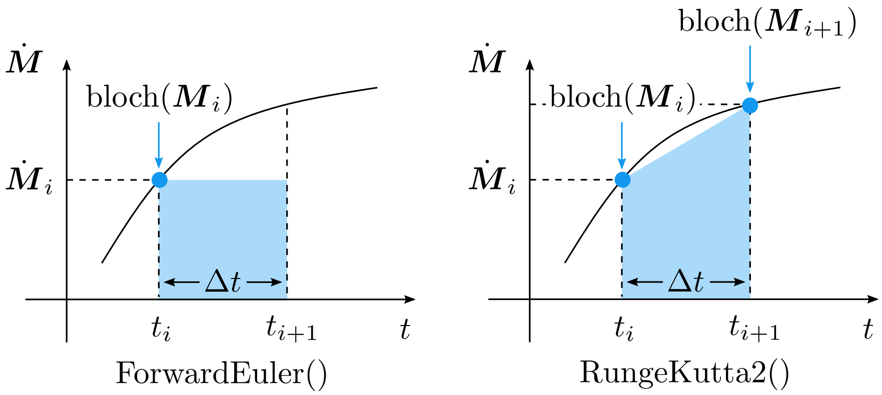

Figure 1: Integration strategies for the different numerical methods.

> ✎ **Task 1:** Complete the code for `solve` and write a function `bloch` that gives the right-hand side of the Bloch equations (explained in the "Magnetic Resonance Imaging (MRI)" section).

Theoretical solution

The Bloch equations does not have a closed solution for every set of conditions, but it has one for our problem:

$$\begin{aligned} M_x(t) &= M_0 \cdot \cos(\gamma B_z t) \cdot \exp(-t/T_2) \\ M_y(t) &= -M_0 \cdot \sin(\gamma B_z t) \cdot \exp(-t/T_2) \\ M_z(t) &= M_0 \cdot (1 - \exp(-t/T_1)) \end{aligned}$$

Note that this solution is only valid for a constant $B = (0, 0, B_z)$ and $M(0) = (M_0, 0, 0)$.

> ✎ **Task 2:** Define a method for `solve` that computes the theoretical solution using array **broadcasting**:

```
solve(m0, dt, tmax, method::Theoretical)
```

Numerical method 1: Forward Euler

The Forward Euler method is described by

$$M_{i+1} = M_i + \Delta t \cdot \mathrm{bloch}(M_i).$$

> ✎ **Task 3:** Define a method for `step` that computes a Forward Euler update:

```
step(dt, m, method::ForwardEuler)
```

Numerical method 2: Runge-Kutta 2nd-order

The Runge-Kutta order 2 method is described by

$$M_{i+1} = M_i + \Delta t \cdot \left(\frac{\mathrm{bloch}(M_i) + \mathrm{bloch}(M_{i+1})}{2} \right).$$

Written like this, the method is implicit (M_{i+1} depends on itself), to make it explicit (to only depend on M_i) we can approximate M_{i+1} by the Forward Euler scheme $M_{i+1} = M_i + \Delta t \cdot \mathrm{bloch}(M_i)$. This is also called Heun's method.

✎ **Task 4:** Define a method for `step` that computes a Runge-Kutta 2nd-order update:

```
step(dt, m, method::RungeKutta2)
```

✎ **Task 5:** How many **methods** does the **functions** `solve` and `step` have? Use `methods(f)`.

Comparison between methods

To compare results, we will use the `Plots` package.

✎ **Task 6:** Install the package `Plots` in your global environment `@v1.10`. Be careful, do not add it to the `BlochFromScratch` environment.

✎ **Task 7:** Choose a `dt` and compare the numerical methods:

```
sol1 = solve(m0, dt, tmax, ForwardEuler())
sol2 = solve(m0, dt, tmax, RungeKutta2())
```

Plot both results in the same figure using `plot` and `plot!`, with the x-axis being time and y-axis the magnetization, include M_x, M_y, and M_z.

✎ **Task 8:** Do the same using $T_2 = 100\,\mathrm{s}$. Do you see any changes to the stability of the solutions?

Creating a Julia package

Now we will create our own Julia package. This package will contain some minor **documentation** and **tests**.

Currently, you should have a file `scripts.jl`. We will move some of its function definitions inside a package `MyPkg`.

✎ **Task 9:** Generate a package inside the `BlochFromScratch` folder by using `] generate MyPkg`. Open the generated folder `MyPkg` in VSCode, which environment is activated by default?

✎ **Task 10:** Copy the definition of `solve` (general and `Theoretical`), `step` (only for `ForwardEuler`), and `bloch` to `MyPkg/src/MyPkg.jl`. Export the functions `solve`, `step`, and the types `Theoretical` and `ForwardEuler`. Modify the `script.jl` file to use this local package instead.

Extending a package

One of the cool things about Julia is that we can extend a package without needing to touch the package internals. As you may have noticed, we haven't included the definition of `RungeKutta2` in `MyPkg`. Let's imagine you are a user of `MyPkg` and want to include this new `RungeKutta2` method.

> **Task 11:** In `script.jl`, extend `MyPkg.step` to be able to perform a 2nd-order Runge-Kutta method. Do **not** touch **any** file inside `MyPkg/`. Replicate the plots from Task 7.

Documenting my package

The simplest way of adding documentation to a package is to add a **docstring**. So, for example,

```julia
"Writes a friendly message."
greet() = print("Hello World!")
```

> **Task 12:** Add a docstring to the function `solve`. Check if `? solve` in the Julia REPL gives you the documentation of the function.

The recommended way of creating documentation in Julia is by using the package Documenter.jl.

Create a folder `MyPkg/docs/`, and inside create a `make.jl` file:

```julia
using Documenter, MyPkg
makedocs(
    sitename = "MyPkg.jl",
    remotes = nothing # For local packagees
)
deploydocs(
    repo = nothing # For local packagees
)
```

Also, in `MyPkg/docs/src/index.md`:

````markdown
# MyPkg

```@autodocs
Modules = [MyPkg]
```
````

> **Task 13:** Add `Documenter` to the `MyPkg/docs/` environment. Add `MyPkg` to the same environment with `dev ..`. Then run `include("docs/make.jl")` to generate the HTML documentation.
> *Hint*: If you need help, take a look at https://github.com/JuliaLang/Example.jl.

Testing my package

To set up tests in Julia, a file `MyPkg/test/runtests.jl` is needed. A minimal test setup can be seen below:

```julia
using Test
@test [1, 1, 1] ≈ ones(3)
```

> ✏ **Task 14:** As a test, check that the `solve` method for `ForwardEuler` gives the same results as `Theoretical`, if not, adjust the tolerance. You will need to add `Test` as a dependency in the `MyPkg/test/` environment. Run the tests with `] test`. If the test fails, adjust `Δt`.

Saving your results 🏁

Congratulations! If you are reading this, you managed to create your first basic Julia package. The idea of this part is to save your performance results to participate for a prize 🏆.

For this, give your GitHub email to the instructor so you can create a pull request (PR) to:

https://github.com/Stockless/pr-execution-leaderboard

This repository will run your code and save the benchmark results. You can add new commits to your PR multiple times, and only the last results will be considered for the leaderboard.

Clone the repository and modify the following files:

- `pr-execution-leaderboard/src/MyPkg.jl`: Include the code of your package.
- `pr-execution-leaderboard/test/runtests.jl`: Add your test code that was passing locally.

If you have any problems, ask the instructor for help.

> ✏ **Task 15:** Include your results in the leaderboard!