Julia for high-performance simulation of optical pulses

Corentin Simon

Univesité Libre de Bruxelles

2026-02-10

Simulations of optical pulse

In general, we want to integrate an equation of the form

$$\partial_z u(z,t) = \sum_n a_n \partial_t^n u + \widehat{\boldsymbol{N}}(u)$$

$\Rightarrow$ Semilinear parabolic PDE

$$\partial_z u(z,t) = \hat{\boldsymbol{D}} u + \widehat{\boldsymbol{N}}(u)$$

- $\hat{\boldsymbol{D}}$ is a (stiff-linear) operator.
- $\widehat{\boldsymbol{N}}$ is some nonlinear function on u.

Stiffness problem

$\hat{\boldsymbol{D}}$ is stiff : **large eigenvalues**.

If transformed in the spectral domain $\partial_t \to -i\omega$

$$\hat{\boldsymbol{D}}(\omega) = \sum_n a_n(-i\omega)^n$$

- Standard explicit ODE Method (RK45, …) poorly suited due to stiffness.
- Implicit methods are better but required a nonlinear solver
  - Large system ($10^2 - 10^3$ or more points) $\Rightarrow$ **poor performance** !

Split-step method

For the NLSE equation:

$$\partial_z u(z,t) = i\frac{\beta_2}{2}\partial_t^2 u + i\gamma\ |u|^2\ u$$

Exact solution

- $u(L,t) = \exp\left(L\left(\hat{\boldsymbol{D}} + \widehat{\boldsymbol{N}}\right)\right)u(0,t)$
- Cannot expands the exponential as $\hat{\boldsymbol{D}}$ and $\widehat{\boldsymbol{N}}$ do not commute.
- Uses the BCH approximation

$$\exp\left(h\left(\hat{\boldsymbol{D}} + \widehat{\boldsymbol{N}}\right)\right)u \approx \exp\left(h\hat{\boldsymbol{D}}\right)\exp\left(h\widehat{\boldsymbol{N}}\right) + \mathcal{O}(\hbar^2)$$

for small step h

- Split the step between linear and nonlinear one for a small
- Repeats L/h times.

Split-step method

Split-step implementation

For the NLSE,

- $\exp\left(h\widehat{\boldsymbol{N}}\right)u_0 = \exp(i\gamma h\ |u_0|^2)u_0$

which can be directly computed

- For $\exp\left(h\hat{\boldsymbol{D}}\right)u_0$, $\hat{\boldsymbol{D}}$ is diagonal in the spectral representation

$\Rightarrow$ exponentiation in the spectral domain

$\Rightarrow \exp\left(h\hat{\boldsymbol{D}}\right)u_0 = \mathcal{F}^{-1}(\exp\left(h\hat{\boldsymbol{D}}(\omega)\right)\mathcal{F}(u_0))$

Fourier transform for performing the DFT $\mathcal{F}$ on u.

- Very performant $\mathcal{O}(n \log n)$
- Very fast/optimized implentation exists (MKL, FFTW, …)

Split-step method

Split-step: strength and weakness

Strength

- Easy to implement

Weaknesses

- Lack accuracy $\mathcal{O}(h^2)$
- No adaptative control: fixed time step.
  - adaptative variations exist but are harder to implement.
- Required n exponential each step for the nonlinear part
  - Exponential is expensive
- For more complex nonlinear part $\widehat{\boldsymbol{N}}$, no explicit formula for $\exp\left(\widehat{\boldsymbol{N}}\right)$
  - Numerical integration is required (RK45, …)

RKIP

RKIP - Stands for **Runge-Kutta in the Interaction Picture**

- Alternative method to split-step.
- Transform the stiff-problem into a non stiff one.
- Allow to use direct Runge-Kutta methods.

If we defined u_I as

$$u_I(z,t) = \exp\left(-t\hat{\boldsymbol{D}}\right)u$$

$$\begin{aligned}
\partial_t u_I &= -\hat{\boldsymbol{D}}u_I + \exp\left(-t\hat{\boldsymbol{D}}\right)\partial_t u \\
&= -\hat{\boldsymbol{D}}u_I + \exp\left(-t\hat{\boldsymbol{D}}\right)\left(\hat{\boldsymbol{D}}u + \widehat{\boldsymbol{N}}(u)\right) \\
&= -\hat{\boldsymbol{D}}u_I + \hat{\boldsymbol{D}}\exp\left(-t\hat{\boldsymbol{D}}\right)u + \exp\left(-t\hat{\boldsymbol{D}}\right)\widehat{\boldsymbol{N}}(u) \\
&= \exp\left(-t\hat{\boldsymbol{D}}\right)\widehat{\boldsymbol{N}}(u) = \exp\left(-t\hat{\boldsymbol{D}}\right)\widehat{\boldsymbol{N}}\left(\exp\left(t\hat{\boldsymbol{D}}\right)u_I\right)
\end{aligned}$$

RKIP

For a Runge-Kutta tableau $c_i, a_{ij}, \alpha_i, \tilde{\alpha}_i$, this gives

$$k_i = \exp\left(-c_i h \hat{\boldsymbol{D}}\right) \widehat{\boldsymbol{N}} \left(\exp\left(c_i h \hat{\boldsymbol{D}}\right) \left(u_n + \sum_{j<i} h a_{ij} k_j \right) \right)$$

$$u_{n+1} = \exp\left(h \hat{\boldsymbol{D}}\right) \left(u_n + \sum_i h \alpha_i k_i \right) \text{ (Update)}$$

$$\boldsymbol{e} = \exp\left(h \hat{\boldsymbol{D}}\right) \left(\sum_i h(\alpha_i - \tilde{\alpha}_i) k_i \right) \begin{array}{c}\text{(Error estimate} \\ \text{if adaptative)}\end{array}$$

The best tableau for this method seems to be the Verner 5(6) tableau.

RKIP: strength and weakness

Strength

- As high order as the RK tableau
  - Large repertoire or higher-order efficient tables
- Adaptative with error control
- Agnostic about $\widehat{\boldsymbol{N}}$

Weaknesses

- Required computing $\exp\left(h\hat{\boldsymbol{D}}\right)$ each steps
  - If not adaptative, can be precomputed
  - If adaptative, can be cached for a fixed number of time steps $h_1, h_2, …, h_S$
    $\rightarrow$ restrict h to theses
- **Hard to implement**
  - Good new : **you don't have to !**

RKIP

RKIP implentation

RKIP is now available as part of the Julia `OrdinaryDiffEq.jl` package.

Why Julia ?

- Julia is compiled Just-in-Time (JIT)
- Syntax close to Python and Matlab
- Faster interpreted language (even with tools/precompiled library)
- `OrdinaryDiffEq.jl` is the most performant ODE solver library available

How to use this code repo

How to use this code repo

1. Install Julia https://julialang.org/downloads/ and git https://git-scm.com/install/windows

For Windows:

```shell
winget install --name Julia --id 9NJNWW8PVKMN -e -s msstore
winget install --id Git.Git -e --source winget
```

2. Ensure you have VSCode installed (https://code.visualstudio.com/download), with its Julia extension.

3. Clone the folder, and open it in VSCode.

```shell
git clone https://github.com/Azercoco/NLSE-Julia-Example.git
cd NLSE-Julia-Example
code .
```

How to use this code repo

Once VSCode is open in the folder, open the command palette (Default : Ctrl+Shift+P)

Search for `> Julia : Restart REPL`, then execute with Enter.

This will open the **Julia REPL** for interacting with Julia.

In the Julia REPL, type

```julia
] # Open the Package manager
activate .
instantiate
```

This will download and install all the required packages.

You can now play or use one of the examples.

RKIP Examples

Example 1 : LLE

See : `../examples/example_1_lle.jl`

This example show how to solve the Lugiato-Levefer equation

$$\partial_T u(T, t) = (-1 + i\partial_t^2)u + i(\Delta - |u|^2)u + S$$

which has two parameters S and Δ.

The operators are:
- $\hat{\boldsymbol{D}} = -1 + i\partial_t^2$
- $\widehat{\boldsymbol{N}}(u) = i(\Delta - |u|^2)u + S$

Δ could have also been moved into $\hat{\boldsymbol{D}}$ but it is easier to include in $\widehat{\boldsymbol{N}}$ as we want to be able to vary it with time $\Delta(t)$.

RKIP Examples

Dependency loading.

```julia
import ProgressLogging
using Plots
include("../src/mod.jl")
```

- `ProgressLogging` is used to display a progress bar in VSCode and `Plots` for plotting.
- `src/mod.jl` contains all the routine used for the examples.

In Julia, a differential equation is defined as $f(u, p, t)$ where t is the integration variable and p a `NamedTuple` containing the equation parameters (here S and Δ).

```julia
p = (; S=1.1, Δ=1.1);
```

This code create a `NamedTuple` with field `S` and `Δ`.

RKIP Examples

Crating the initial conditions.

```julia
u0 = lle_hss(p.S, p.Δ, 2^12; noise=1e-2)
```

This creates a vector of size `2^12`= 2048 points, corresponding to the continuous stationary solution of the LLE equation with a random noise level of `noise`.

```julia
dτ = 0.05
LLE = SemilinearPDE(
        StandartLLE(),
        u0,
        300.0,
        dτ,
        p
);
```

This creates a semilinear PDE with initial solution `u0`, an integration bounds of `(0, 300.0)`, a time step `dτ` and parameters `p`.

RKIP Examples

We solve the problem

```julia
u = solve_rkip(LLE; abstol=1e-4, reltol=1e-8)
```

with relative tolerance `reltol` and absolute tolerance `abstol`. Changing these value will increase/ decrease the precision with a cost/gain in performance.

We can plot the solution:

```julia
plot(
    LLE.τ,
    abs2.(u),
)
```

`LLE` has two fields `τ` and `freq` storing the value of fast time and normalized frequencies.

RKIP Examples

We can also sample the solution at different time using the `saveat` keyword arguments.

```julia
t = 0.0:5:300 # sample every 5 unit of time between 0 and 300
u2 = solve_rkip(LLE; saveat=t, abstol=1e-4, reltol=1e-8);
```

`u2` is now matrix with a number of columns corresponding the size of `t`.

Similarly the solution is plotted.

```julia
heatmap(abs2.(u2))
```

`LLE` has two fields `τ` and `freq` storing the value of fast time and normalized frequencies.

RKIP Examples

Exemple 2: LLE-Scan with time varying Δ

See `:../examples/example_2_lle_scan.jl`

This example is similar to the first. The only difference is that Δ is now a function of `p` and `t`.

```julia
Δ_scan(p, t) = p.Δ₀ + p.Δ_rate * t
p = (;
        S=3.5,
        Δ₀=-2.0,
        Δ_rate=0.2,
        Δ = Δ_scan,
)
u0 = lle_hss(p.S, p.Δ₀, 1024; noise=1e-2)
```

We have introduced two new parameters, the initial detuning `Δ₀` and detuning scan rate `Δ_rate`. Otherwise, we solve it exactly the same manner as the first example.

RKIP Examples

Example 3: third-order dispersion

See : `../examples/example_3_lle_d3.jl`

In this example, we want to modify the linear operator $\hat{\boldsymbol{D}}$ to add a third-order dispersion to the LLE:

$$\partial_T u(T,t) = (-1 + i\partial_t^2 + \boldsymbol{i d_3 \partial_t^3})u + \dots.$$

We add this new parameters `d₃` to the set of parameters.

```julia
p = (;..., d₃=0.08)
```

```julia
LLE_d3 = SemilinearPDE(
    StandartLLE(;
        D̂=(ω, p) -> -1 - 1im * ω^2 + 1im * ω^3 * p.d₃
    ),...
```

We can overide the the definition of $\hat{\boldsymbol{D}}$ by passing a function of (ω, p) computing $\tilde{\hat{\boldsymbol{D}}}(\omega)$.

RKIP Examples

Example 4: Raman Nonlinearity

See :`../examples/example_4_lle_rarman.jl`

The Raman nonlineariyt adds an additional term to the LLE:

$$\partial_T u(T,t) = (-1 + i\partial_t^2)u + i(\Delta - |u|^2)u + S + \boldsymbol{i\tau_R(\partial_t|u|^2)u}$$

Similarly, we add a new parameter `tau_R`.

```julia
p = (;..., τ_R=2e-4)
```

And we can modify the nonlinearity used by doing

```julia
LLE_raman = SemilinearPDE(
    StandartLLE(;
        N̂=LLERamanNonlinearPart()
    ),...
```

The rest is similar to the previous examples.

Custom Nonlinearity

Adding a custom nonlinearity.

To add a new nonlinearity, declares a new empty struct corresponding this nonlinearity, which inherits `AbstractNonlinearPart` and implements `(nl::CustomNonlinearity)(du, u, p, t)`.

```julia
struct CustomNonlinearity <: AbstractNonlinearPart end

function (nl::CustomNonlinearity)(du, u, p, t)
  # Put your nonlinear function here and store the result in du
end
```

and then use:

```julia
custom_lle = SemilinearPDE(
    StandartLLE(;
         N̂=CustomNonlinearity()
    ),...
```

Custom Nonlinearity

Custom Nonlinearity: PDLNSE

Example nonlinearity for a parametric driven NLSE:

$$\partial_T u(T,t) = (-1 + i\partial_t^2)u + i(\Delta - |u|^2)u + \kappa u^*$$

```julia
struct PDLNSE <: AbstractNonlinearPart end

function (nl::PDLNSE)(du, u, p, t)
  Δ = get_var(nl, p.Δ, p, t)
  κ  = get_var(nl, p.κ, p, t)

  @. du = 1im * (abs2(u) - Δ) * u + κ * conj(u)
end
```

The helper `get_var(nl, var, p, t)` allows `var` to be either a scalar or a function of `(p, t)`. We used the *broadcast* macro `@.` which executes the operation on the array without allocations.

Also remember to add `κ=...` to `p`.

Custom Nonlinearity

Custom Nonlinearity: Cache

Sometimes we want to cache some array for the computation. In that case, we can implement `function get_cache(nl, u0, p)` which returns a `NamedTuple`.

```julia
struct CustomNonlinearityWithCache <: AbstractNonlinearPart end

function get_cache(::CustomNonlinearityWithCache, u0, p)
    return (; cached_array=similar(u0))
end
```

The cache content can be accessed in the main evaluation with `p.cache`:

```julia
function (nl::PDLNSE)(du, u, p, t)
  cached_array = p.cache.cached_array
  ....
```

Custom Nonlinearity

Custom Nonlinearity: Raman example

Complete implementation in `../src/raman.jl`.

```julia
struct LLERamanNonlinearPart <: AbstractNonlinearPart end

# For Raman, we need to implement a cache to store intermediate computation
function get_cache(::LLERamanNonlinearPart, u0, p)
    abs2_tmp = similar(u0) # pre-allocate an array for intermediate storage
    raman_fourier_resp = convert(
        typeof(u0),
        @. -1im * 2π * p.freq * p.τ_R
    ) # pre-computed the Rama kernel in spectral space
    return (; abs2_tmp, raman_fourier_resp) # cache are supplied as a NamedTuple
end
```

Custom Nonlinearity

```julia
@fastmath function (lle_raman::LLERamanNonlinearPart)(du, u, p, t)
    S = get_var(lle_raman, p.S, p, t) # fetch eventually time varying function
    Δ = get_var(lle_raman, p.Δ, p, t) # fetch eventually time varying function
    @unpack abs2_tmp, raman_fourier_resp = p.cache # We can acces the cache using p.cache
    abs2_tmp .= abs2.(u) # used form temporary storage
    @. du = 1im * (abs2_tmp - Δ) * u + S # LLE
    fft!(p.bifft_plan, abs2_tmp) # in-place preallocated FFT
    @. abs2_tmp = raman_fourier_resp * abs2_tmp # convolution = mutiplication in spectral domain
    ifft!(p.bifft_plan, abs2_tmp) # in-place preallocated IFFT
    @. du += 1im * abs2_tmp * u
    return du
end
```

To use `fft!` and `ifft` with pre-allocated storage and temporary array, the wrappers `ifft!/fft!(p.bifft_plan, target)` are available.

Performance tips:

- Pre-allocate every array you may need in the cache (allocations are the main bottleneck of code)
  - Use Julia `similar(u0)` to create new array to keep to code generic.
  - Always use in-place mutating operation (in-place function in Julia are indicated with a `!`) which does not create temporary array
- Use the broadcast macro `@.` when to apply an element-wise operation on one or several arrays.
- Use the `@fastmath` macro before your function.
- Make sure all of your array have a well-defined type.
- Avoid using `abs(u)^2` to compute $|u|^2$, as it compute a squared-root internally. Use Julia `abs2()` instead
- Similarly, avoid `exp(1im*x)` to compute $\exp(ix)$ if x is real, use `cis(x)` instead.
- Be sure to computed only once and cached expensive computation that do not need to be updated.
- On IntelCPU, uses MKL FFT with `FFTW.set_provider!("mkl")`.
- Launch Julia with multiple threads wiht `julia --threads ...`
- For very large problem ($> 10^4$ points), consider using a GPU.
  - All the code in this repo should be compatible with a GPU by converting `u0` to `CuArray` from `CUDA.jl` package.
  - On the GPU, only use `Float32` (32-bit float) array instead of `Float64`.

Conclusion

- Except for simplicity, there is no reason to use Split-Step instead of RKIP
- Give Julia a try ! (You can interface it with Python)
- All examples code are on my GitHub https://github.com/Azercoco/NLSE-Julia-Example
  - Contributions and suggestions are welcome !