User Manual for the FdeSolver Package in Julia

Moein Khalighi, Giulio Benedetti, Leo Lahti
Department of Computing, Faculty of Technology, University of Turku, Finland

Abstract

This guide explains the usage and functionality of the FdeSolver Julia package for solving fractional differential equations accompanying the article

> Moein Khalighi, Giulio Benedetti, Leo Lahti. Algorithm xxx: FdeSolver, a Julia Package for Solving Fractional Differential Equations. ACM Trans. Math. Softw. xx, x, Article xx (202x), xx pages.

1 Software Implementation with Julia

We have implemented the numerical methods described in Section 2 in the main manuscript in the FdeSolver Julia package, available at `https://github.com/JuliaTurkuDataScience/FdeSolver.jl`. It is released with the permissive MIT open-source license, which has been recommended for research software in [3]. The package is also listed on Julia's General Registry at `https://github.com/JuliaRegistries/General/tree/master/F/FdeSolver`. Our current representation is based on the v1.0.7 release of the FdeSolver package.

The FdeSolver package takes advantage of several features of Julia. This is a compiled language, which means it is interactive and can be used with a read-evaluate-process-loop (REPL) interface. Anyone can contribute to improving the open-source FdeSolver package and use it in their applications. Multiple dispatch is one of the useful features that we use in our package. In multiple dispatch, a function can have multiple implementations allocated for different parameters dispatched at runtime and determined based on the precise parameter types [1]. Multiple dispatch relies on two other performance-enhancing features, composite types and dynamic types: 1) There is a unique feature of Julia's composite types (like objects or structs in other languages) in that functions do not get bound to objects, nor are they bundled with them. This is essential for multiple dispatches and leads to more flexibility. 2) Julia can assign a type to a variable, similar to static programming, or support dynamic types determined during execution, contrary to other high-level languages.

1.1 Installing FdeSolver

FdeSolver v1.0.7 is available for Julia version 1 and higher. Julia has a built-in package manager named `Pkg` that can handle operations. By typing a `]` in Julia REPL, you can enter the Pkg REPL and type `add FdeSolver@v1.0.7` to install the package for this specific version, and use `up FdeSolver` or `rm FdeSolver` to update the version or remove the package, respectively.

1.2 Third Party Supporting Packages

FdeSolver uses `FFTW` package version 1.2 for fast Fourier transforms, `LinearAlgebra` for constructing diagonal and identity matrices and applying norm functions, and `SpecialFunctions` version 1 or 2 for using Γ function. All dependencies of FdeSolver are listed in `Project.toml`:

```
[deps]
FFTW = "7a1cc6ca-52ef-59f5-83cd-3a7055c09341"
LinearAlgebra = "37e2e46d-f89d-539d-b4ee-838fcccc9c8e"
SpecialFunctions = "276daf66-3868-5448-9aa4-cd146d93841b"
```

and the compatibility constraints for the mentioned dependencies are listed as follows:

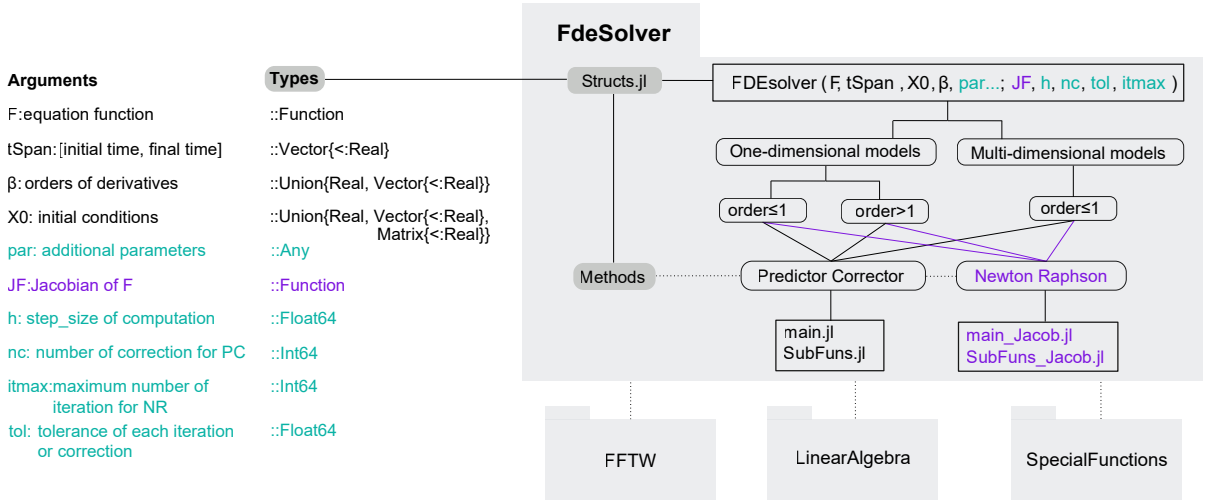

Figure 1: A schematic of FdeSolver package v1.0.7. This package can solve one-dimensional fractional models for the order derivative $\beta > 0$ and multi-dimensional ones for the order derivatives $0 < \beta \leq 1$ of a class of problem (3), detailed in the main manuscript. The name of the solver function is `FDEsolver`. There are ten arguments where the four positional arguments are `F`: the equation function, `tSpan`: the domain of the problem, `X0`: the initial conditions, and β: the orders of derivatives and the four optional arguments are `h`: step-size of computation, `nc`: the number of correction for the predictor-corrector method, `itmax`: the maximum number of iterations for the Newton-Raphson method, and `tol`: the tolerance of each iteration or correction. The input `par` is any additional parameters related to the problem, and `JF` is a Jacobian function needed for the Netwon-Raphson method. The predictor-corrector method is encoded in `main.jl` and `SubFuns.jl`, and the Newton-Raphson method in `main_Jacob.jl` and `SubFuns_Jacob.jl`. `Structs.jl` leverages multiple dispatch to choose a proper numerical algorithm based on the presence or the lack of a Jacobian function and tune its functionality based on arguments' types. The `FFTW`, `LinearAlgebra`, and `SpecialFunctions` Julia packages support the FdeSolver package.

```
[compat]
FFTW = "1.2"
SpecialFunctions = "1, 2"
julia = "^1"
```

1.3 Practices for Reproducible Software Development

FdeSolver follows state-of-the-art methods for building and sharing scientific computing software, as encouraged by the Software Carpentry and Data Carpentry communities [4]. The main repository of the package is created on GitHub (with Zenodo DOI: `https://doi.org/10.5281/zenodo.7462094`). Unit testing is automatically performed on Ubuntu and MacOS x64 machines via GitHub Actions when the content of one or multiple files is modified. At the same time, Codecov, an open-source third-party service, is used to keep track of the coverage, that is, the percentage of code actually run by the unit tests. Our continuous integration approach simplifies and speeds up the collaborative work around FdeSolver, to which any member of the Julia community can readily contribute. The functionality of our package is thoroughly described in `README.md` and documented more comprehensively in its official manual, where users can rapidly get started with the four provided usage examples of the solver.

1.4 FdeSolver Basics

In Julia, a *struct* defines a type composed of other types, e.g., `Float64`, `Int64`, or even other structs. Parametric structs are used throughout FdeSolver to adjust the algorithms for different model classes. The types implemented by the package are shown in Fig. 1. The inputs are divided into four sets of arguments:

- Positional arguments

  - `F`: derivative function, or the right side of the system of differential equations. Depending on the problem, it could be expressed as a function and return a vector function with the same number of entries of order derivatives. This function can also include a vector of additional parameters.
  - `tSpan`: the time along which computation is performed. It must be a vector containing the two values for the initial and final times.
  - `X0`: the initial conditions. The values in the type of a row vector for $\beta \leq 1$ and a matrix for $\beta > 1$, where each column corresponds to the initial values of one differential equation and each row to its order of derivation.
  - β: the orders derivatives in a type of scalar or vector, where each element corresponds to the order of one differential equation. It could be an integer value.

```
struct PositionalArguments
   F::Function
   tSpan::Vector{<:Real}
   X0::Union{Real, Vector{<:Real}, Matrix{<:Real}}
   β::Union{Real, Vector{<:Real}}
end
```

- Optional arguments
  - `h`: the step size of the computation. The default is 2^{-6}.
  - `nc`: the desired number of corrections for the PC method when there is no Jacobian. The default is 2.
  - `tol`: the tolerance of errors taken from the infinity norm of each iteration for the NR method or correction when nc¿10 for the PC method. The default is 10^{-6}.
  - `itmax`: the maximal number of iterations for the NR method when the user defines a Jacobian. The default is 100.

```
struct OptionalArguments
    h::Float64
    nc::Int64
    tol::Float64
    itmax::Int64
end
```

- `JF`: the Jacobian of F for switching to the NR method. The solver will evaluate the solution using the PC method if not provided.
- `par`: additional parameters for the functions F and JF.

2 Benchmarking Repository Orientation

The repository has been structured to facilitate easy navigation and comprehension of the benchmarking and simulation processes. Below is an overview of the key directories and functionalities.

2.1 Main Directories

The repository is primarily organized into three main directories:

- `data`: Contains raw and preprocessed data files.
- `results`: Stores the plots and outputs generated from the scripts.
- `src`: Houses the source code and scripts for benchmarking and simulations.

2.1.1 data Directory

The `data` directory holds preprocessed data for generating figures and the raw output from benchmarks in the `data/raw` subdirectory.

2.1.2 results Directory

The `results` directory encompasses three subdirectories corresponding to the various benchmarks and simulations:

- `benchmarking`
- `community_simulation`
- `covid_fitting`

Within these subdirectories, users can find the related plots for each case.

2.1.3 src Directory

The `src` directory consists of three core subdirectories:

1. **benchmarking**:
    - `R`: Incorporates two R scripts utilised within Julia for data rearrangement and plotting.
    - `julia`: Contains six Julia scripts for solutions of the six benchmarking examples presented in the main manuscript.
    - `matlab`: This is where all MATLAB codes used for benchmarking are stored.
    - `plotters`: Consists of three Julia scripts dedicated to plotting the three benchmarking classes: One-dimension (`Plt1D.jl`), Multi-dimensions (`PltMD.jl`), and Random parameters (`PltRnd.jl`).
2. **community_simulation**: Dedicated to the microbial community simulation as detailed in the main manuscript.
3. **covid_fitting**: Specifically for performing COVID-19 transmission fitting, further explained in the main manuscript.

2.2 Julia Scripts Overview

Here is a brief rundown of the core Julia scripts found in the repository:

- `plot_benchmarking_results.jl`: Generates the benchmarking figures presented in the main manuscript.
- `run_benchmark.jl`: Executes the benchmarks again for validation or further analysis.
- `run_community_simulation.jl`: Produces the related figure for microbial community simulation.
- `run_covid_fitting.jl`: Renders the associated figure for Covid-19 transmission fitting.

3 Usage and benchmarks

All the model classes shown in Figure 1 are solved using FdeSolver. The availability of source code for replicating the results in the manuscript is presented in GitHub with Zenodo DOI: `https://doi.org/10.5281/zenodo.10037076`. Here, we provide implementation details and relevant codes for user guidance. The user needs to type

```
using FdeSolver, Plots
```

at the top of the code for solving the examples and plotting the results.

We measure the performance of Matlab's codes using function `timeit()` and separately run a group of benchmarks for Julia's codes using `BenchmarkTools.jl` package. Benchmark results in Julia are measured in nanoseconds. We multiply the mean by 10^{-9} to compare them with Matlab. A sample of benchmarking methods M1 and J1 is presented as follows:

```julia
## Julia
    # Computing the run time
    t= @benchmark FDEsolver(F, $(tSpan), $(X0), $(β), $(par), h=$(h)) seconds=1
    T= mean(t).time / 10^9 # convert from nanoseconds to seconds
    # Computing the error
    tt, X = FDEsolver(F, tSpan, X0, β, par, h=h)
    ery=norm(X - map(Exact, tt), 2) # Error: 2-norm

```

```matlab
%% Matlab
    % Computing the run time
    Bench(1,1) = timeit(@() FDE_PI12_PC(beta,F,t0,T,X0,h,param));
    % Computing the error
    [t,X]=FDE_PI12_PC(beta,F,t0,T,X0,h,param);
    Bench(1,2)=norm((X-Ex));
```

Similarly, we can add other methods and include the settings explained in the examples. Finding the exact solution of the multi-dimensional models is difficult or not possible, so we measure the accuracy of the methods by comparing the obtained results with the results secured by fine step size in Matlab:

```matlab
    [t,Yex]=FDE_PI2_Im(beta,F,JF,t0,T,x0,2^(-10),par, 1e-12);%Solution with a fine step size in
    ↪  Matlab
```

and Julia:

```julia
    t, Yex=FDEsolver(F, tSpan, X0, β, par, JF = JF, h=2^-10, tol=1e-12)#Solution with a fine
    ↪  step size in Julia
```

The benchmarking examples utilize the following packages in conjunction with our own:

- `BenchmarkTools`: Streamlines Julia performance tracking and benchmark comparisons.
- `FractionalDiffEq`: Benchmarks an alternative Julia package and employs the Mittag-Leffler function.
- `Plots`: For result visualisations.
- `LinearAlgebra`, `SpecialFunctions`: Provides special operators.
- `CSV`, `DataFrames`, `Tables`: Modifies and compares results.

3.1 One-Dimensional Models

The codes corresponding to the functions and conditions of benchmarking of one-dimensional examples are detailed in this section.

3.1.1 Non-Stiff Example

For this example, we use the SpecialFunctions package in Julia to leverage the gamma function in our calculations. The time should be encoded using the format [initial time, final time].

```julia
    using SpecialFunctions
    tSpan = [0, 1]      # [intial time, final time]
    X0 = 0              # intial value
    β = 0.5             # order of the derivative
    # Equation
    par = β
    F(t, X, par) = (40320 / gamma(9 - par) * t ^ (8 - par) - 3 * gamma(5 + par / 2)
           / gamma(5 - par / 2) * t ^ (4 - par / 2) + 9/4 * gamma(par + 1) +
           (3 / 2 * t ^ (par / 2) - t ^ 4) ^ 3 - X ^ (3 / 2))
    # Jacobian function
    JF(t, X, par) = -(3 / 2) * X ^ (1 / 2)
    # Solution
    t1, X1 = FDEsolver(F, tSpan, X0, β, par)
    _, X2= FDEsolver(F, tSpan, X0, β, par, JF = JF)
    [...]
```

The results include the time `t1`, the solutions derived from the PC method (`X1`), and the NR method (`X2`). Both solutions share the same time span because the computational step size `h` is unspecified for both and defaults to 2^{-6}. The notation `[...]` indicates that some portions of the codes have been omitted for brevity.

3.1.2 Stiff Example

For the Mittag-Leffler function, we recommend utilizing the `mittleff` function from the `FractionalDiffEq (v0.3.1)` package over the `MittagLeffler.jl` package. The latter faces compatibility constraints with most versions of the `SpecialFunctions` package.

We include `nc=4` in the optional arguments for 4 corrections using the PC method and set `tol=1e-8` to establish an iteration tolerance of 10^{-8} for the NR method.

```
using FractionalDiffEq # to get MittagLeffler function
    [...]
    λ = -10
    par = λ
    F(t, X, par)= par * X # equation F
    JF(t, X, par) = par # Jacobian F
    t1, X1 = FDEsolver(F, tSpan, y0, β, par, nc=4)
    _, X2 = FDEsolver(F, tSpan, y0, β, par, JF = JF, tol=1e-8)
    # exact solution: mittag-leffler
    Exact(t) = mittleff(β, λ * t .^ β)
    X_exact=map(Exact, t1) # the exact solution corresponding to the time steps t1
    [...]
```

3.1.3 High-Order Example

The initial conditions must adhere to the format $[X_0 \quad X_0']$ for second-order cases. Here, X_0 represents the initial value of the variable X, while X_0' corresponds to the initial value of its first derivative.

```
  [...]
  ## inputs
  tSpan = [0, 10]      # [intial time, final time]
  X0 = [1 1]              # intial value ([of order 0      of order 1])
  β = 1.90             # order of the derivative
  par = [16.0, 4.0] # [spring constant for a mass on a spring, inertial mass]
  ## Equations
  function F(t, x, par)
    K, m = par
    - K / m * x
  end
  function JF(t, x, par)
    K, m = par
    - K / m
  end
  [...]
  Exact(t) = X0[1] .* mittleff(β,1, -par[1]./par[2] .* t .^ β) .+ X0[2] .* t .* mittleff(β,2,
  ↪  -par[1]./par[2] .* t .^ β) # the exact solution
  [...]
```

3.2 Multi-Dimensional Systems

The section details the codes for benchmarking multidimensional examples.

3.2.1 Non-Oscillation Example

For a system of dimension n with variables $X_1, \ldots, X_n$, the initial values should be arranged as: $[X_1(0), X_2(0), \ldots, X_n(0)]$ with commas between the values.

```
[...]
## inputs
    I0 = 0.1              # initial value of infected
    tSpan = [0, 100]        # [intial time, final time]
    y0 = [1 - I0, I0, 0]    # initial values [S0,I0,R0]
    α = [.9, .6, .7]           # order derivatives
## ODE model
par = [0.4, 0.04] # parameters [infectious rate, recovery rate]
function F(t, y, par)
    # parameters
    β, γ = par   # infection rate,  recovery rate
    S, I, R = y   # Susceptible, Infectious, Recovered
    # System equation
    dSdt = - β * S * I
    dIdt = β * S * I - γ * I
    dRdt = γ * I
    return [dSdt, dIdt, dRdt]
end
function JF(t, y, par) # Jacobian of ODE system
    # parameters
    β, γ = par   # infection rate,  recovery rate
    S, I, R = y   # Susceptible, Infectious, Recovered
    # System equation
    J11 = - β * I;    J12 = - β * S;    J13 =  0
    J21 =  β * I;    J22 =  β * S - γ;    J23 =  0
    J31 =  0;    J32 =  γ;    J33 =  0
    J = [J11 J12 J13
         J21 J22 J23
         J31 J32 J33]
    return J
end
[...]
```

3.2.2 Sharp Oscillation Example

The computational methods used are consistent with the previous examples, so further elaboration is unnecessary.

3.2.3 Random Values

We have considered four examples with random values for the parameters. Employing "`i`" as a counter for each configuration of a random variable, the parameters and conditions for Example 3.1.1 are presented in the following code:

```
    [...]
    tSpan = [0, 1]
    h =M_Ex1.HR[i] # step size
    nc=M_Ex1.NcR[i]
    tol=M_Ex1.TolR[i]
    β=M_Ex1.AlphaR[i] # order of the derivative
    par = β #parameter
    [...]
```

In which, `M_Ex1.HR`, `M_Ex1.NcR`, `M_Ex1.TolR`, and `M_Ex1.AlphaR` represent the sets of random vectors extracted from benchmarking Matlab routines, regulated by the random number generator command `rng(1,'twister')`. Moving forward, it should be noted that all instances of `M_Ex*` correspond to the sets of random vectors that are similarly extracted from the benchmarking Matlab routines.

The codes are configured for Example 3.1.2 as follows:

```
    [...]
    h =M_Ex2.HR[i] # step size
    nc=M_Ex2.NcR[i]
```

```
    tol=M_Ex2.TolR[i]
    β=M_Ex2.AlphaR[i] # order of the derivative
    par=M_Ex2.LambdaR[i]
    T=M_Ex2.TR[i]; tSpan=[0,T]
    y0 =1
    [...]
```

The codes are set up for Example 3.1.3 in the following way:

```
    [...]
    h =M_Ex3.HR[i] # step size
    nc=M_Ex3.NcR[i]
    tol=M_Ex3.TolR[i]
    β=2 # order of the derivative
    par=[16.0, 4.0]
    T=M_Ex3.TR[i]; tSpan=[0,T]
    y0 = [1 1]
    [...]
```

The codes for Example 3.2.1 are arranged in the following manner:

```
    [...]
    h=2.0^(- M_Ex4.HR[i])# step size
    nc=M_Ex4.NcR[i]
    tol=M_Ex4.TolR[i]
    β=[M_Ex4.alp1R[i],M_Ex4.alp2R[i],M_Ex4.alp3R[i]]
    par=[M_Ex4.BetaR[i], M_Ex4.GamaR[i]]
    T=M_Ex4.TR[i]; tSpan=[0,T]
    y0 = [1-M_Ex4.II0[i], M_Ex4.II0[i], 0]
    Tspan = (0, T)
    [...]
```

4 Applications

The following section contains the codes relating to the functions and conditions of the application examples.

4.1 Simulation of Microbial Community Dynamics

In our simulation, we utilize the following packages along with FdeSolver:

- `SpecialFunctions`, `Statistics`: For specific functions.
- `Plots`, `ColorTypes`, `LaTeXStrings`: For plotting purposes.

The code below replicates Figure 2 from Ref. [2], also depicted in our main manuscript.

```
## inputs
tSpan = [0, 300]   # time span
h = 0.05           # time step
N = 3         # number of species
# order of derivatives
μ1=[1;1;1] # No memory
μ2=.9*[1;1;1] # With memory

X0 = [.99;.01;.01]   # initial abundances

## System definition

# parameterization
par1 = [2,
       [1;.95;1.05],
       ones(N),
```

```
      .1*ones(N,N),
       N,
       1]
par2 = copy(par1) #deepcopy is another option
par2[6]= 2

function F(t, x, par)
    l = par[1] # Hill coefficient
    b = par[2] # growth rates
    k = par[3] # death rates
    K = par[4] # inhibition matrix
    N = par[5] # number of species
    pulse = par[6] # pulse type

    # pulse perturbation
    B=copy(b)
    if pulse==1
        bG=2
    elseif pulse==2
        bG=2.2
    end
    bB=0.5

    if t>20.0 && t<60.0
        B[1]=bB
        B[3]=bG
    end

# ODE
    Fun = zeros(N)
    for i in 1:N
    # inhibition functions
    f = prod(K[i,1:end .!= i] .^ l ./
             (K[i,1:end .!= i] .^ l .+ x[ 1:end .!= i] .^l))
    # System of equations
    Fun[i] = x[ i] .* (B[i] .* f .- k[i] .* x[ i])
    end

    return Fun

end

t, x1 = FDEsolver(F, tSpan, X0, μ1, par1, h = h) # Solution without memory under the weaker
  ↪  perturbation
_, x2 = FDEsolver(F, tSpan, X0, μ2, par1, h = h) # Solution with memory under the weaker
  ↪  perturbation
_, x12 = FDEsolver(F, tSpan, X0, μ1, par2, h = h) # Solution without memory under the stronger
  ↪  perturbation
_, x22 = FDEsolver(F, tSpan, X0, μ2, par2, h = h) # Solution with memory under the weaker
  ↪  perturbation


# Perturbations
function Fun_pert1(t)
    bb,br,bg=  [1,.95, 1.05]
    if t > 20 && t <60
            bg=2
            bb=0.5
    end
    return [bb,br,bg]
end
#functions for plotting the patches
```

```
fbb(x)=20<x<60 ? 0.5 : 1
fbr(x)=1.05
fbg1(x)=20<x<60 ? 2 : 0.95
fbg2(x)=20<x<60 ? 2.2 : 0.95
[...]
```

To delve deeper, the model can handle higher-dimensional communities by setting `N` to a value exceeding 3. Additionally, for exploring other perturbation types, we provide code exemplifying the integration of a periodic perturbation.

```
[...]
# periodic perturbation
    B = copy(b)
    mm = 20
    m = ceil(mod(t / (mm * 4), mm))
    bB1 = 0.2
    bB2 = 4.5

    if pulse == 1

        if t > 60.0 && t < 100.0
            B[1] = bB1
        elseif t > 200 && t < 330.0
            B[1] = bB2
        end

    elseif pulse == 2

        if t >= mm * (4 * m - 3) && t < mm * (4 * m - 2)
            B[1] = bB1
        elseif t >= mm * (4 * m - 1) && t < mm * (4 * m)
            B[1] = bB2
        end

    end
[...]
```

4.2 Epidemiological Analysis of Covid-19 Transmission Dynamics

For disease transmission modeling and data fitting, we combine the following packages with our own:

- `CSV`, `HTTP`, `DataFrames`, `Dates`: For data importation and managements.
- `Optim`, `StatsBase`: For optimisation and value fitting.
- `SpecialFunctions`: For specialised functions.
- `Plots`, `StatsPlots`, `StatsPlots.PlotMeasures`: For plotting tasks.

To retrieve data directly from the online repository, the following code accesses the relevant URL and then parses the content into a CSV format for use in Julia:

```
# dataset of Covid from CSSE
url = "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/" *
      "csse_covid_19_data/csse_covid_19_time_series/time_series_covid19_confirmed_global.csv"
repo = HTTP.get(url)
dataset_CC = CSV.read(repo.body, DataFrame) # all data of confirmed
```

To preprocess the data specifically for Portugal, the following code filters out the relevant subset of data and then performs necessary adjustments, such as handling negative values and outliers:

```
    # Dataset subset
    Confirmed=dataset_CC[dataset_CC[!,2].=="Portugal",45:121] #cumulative confirmed data of
    ↪ Portugal from 3/2/20 to 5/17/20
```

```
    C=diff(Float64.(Vector(Confirmed[1,:])))# Daily new confirmed cases

#prepossessing (map negative values to zero and remove outliers)
    _Ind=findall(C.<0)
    C[_Ind].=0.0
    outlier=findall(C.>1500)
    C[outlier]=(C[outlier.-1]+C[outlier.-1])/2
```

The following code preprocesses the dataset specifically for Spain. It extracts relevant data, computes daily new confirmed cases, and corrects any negative values:

```
Confirmed=dataset_CC[dataset_CC[!,2].=="Spain",38:142] #confirmed of spain from 2/24/20 to
↪  6/7/20
C=diff(Float64.(Vector(Confirmed[1,:])))# Daily new confirmed cases

#preporcessing (map negative values to zero)
_Ind=findall(C.<0)
C[_Ind].=0.0
```

The code provided is used to model COVID-19 transmission in Portugal. The function `SIR()` simulates how the compartments evolve.

```
## System definition

# parameters
    β=2.55 # Transmission coefficient from infected individuals
    l=1.56 # Relative transmissibility of hospitalized patients
    β′=7.65 # Transmission coefficient due to super-spreaders
    κ=0.25 # Rate at which exposed become infectious
    ρ1=0.58 # Rate at which exposed people become infected I
    ρ2=0.001 # Rate at which exposed people become super-spreaders
    γa=0.94 # Rate of being hospitalized
    γi=0.27 # Recovery rate without being hospitalized
    γr=0.5 # Recovery rate of hospitalized patients
    δi=1/23 # Disease-induced death rate due to infected class
    δp=1/23 # Disease-induced death rate due to super-spreaders
    δh=1/23 # Disease-induced death rate due to hospitalized class
# Define SIR model
function SIR(t, u, par)
    # Model parameters.
        N,β,l,β′,κ,ρ1,ρ2,γa,γi,γr,δi,δp,δh=par

    # Current state.
    S, E, I, P, A, H, R, F = u

# ODE
    dS = - β * I * S/N - l * β * H * S/N - β′* P * S/N # susceptible individuals
    dE = β * I * S/N + l * β * H * S/N + β′ *P* S/N - κ * E # exposed individuals
    dI = κ * ρ1 * E - (γa + γi )*I - δi * I #symptomatic and infectious individuals
    dP = κ* ρ2 * E - (γa + γi)*P - δp * P # super-spreaders individuals
    dA = κ *(1 - ρ1 - ρ2 )* E # infectious but asymptomatic individuals
    dH = γa *(I + P ) - γr *H - δh *H # hospitalised individuals
    dR = γi * (I + P ) + γr* H # recovery individuals
    dF = δi * I + δp* P + δh *H # dead individuals
    return [dS, dE, dI, dP, dA, dH, dR, dF]
end

#initial conditions
    N=10280000/875 # Population Size
    S0=N-5; E0=0; I0=4; P0=1; A0=0; H0=0; R0=0; F0=0
    X0=[S0, E0, I0, P0, A0, H0, R0, F0] # initial values
    tspan=[1,length(C)] # time span [initial time, final time]

    par=[N,β,l,β′,κ,ρ1,ρ2,γa,γi,γr,δi,δp,δh] # parameters
```

Three optimization approaches are provided for fitting the observed data. 1) Integer order model, 2) commensurate fractional order model, and 3) incommensurate fractional order model

```
## Optimisation of β for integer order model
function loss_1(b)# loss function
        par[2]=b[1]
        _, x = FDEsolver(SIR, tspan, X0, ones(8), par, h = .1)
    appX=vec(sum(x[1:10:end,[3,4,6]], dims=2))
    rmsd(C, appX; normalize=:true) # Normalized root-mean-square error
end

    p_lo_1=[1.4] #lower bound for β
    p_up_1=[4.0] # upper bound for β
    p_vec_1=[2.5] #  initial guess for β
    Res1=optimize(loss_1,p_lo_1,p_up_1,p_vec_1,Fminbox(BFGS()),# Broyden{Fletcher{Goldfarb{Shanno algorithm
# Result=optimize(loss_1,p_lo_1,p_up_1,p_vec_1,SAMIN(rt=.99), # Simulated Annealing algorithm (sometimes it has better performance than (L-)BFGS)
                        Optim.Options(outer_iterations = 10,
                                                iterations=10000,
                                                show_trace=true,
                                                show_every=1))
p1=vcat(Optim.minimizer(Res1))
par1=copy(par); par1[2]=p1[1]

## Optimisation of β and order of commensurate fractional order model
function loss_F_1(pμ)
        par[2] = pμ[1] # infectious rate
        μ = pμ[2] # order of derivatives

        _, x = FDEsolver(SIR, tspan, X0, μ*ones(8), par, h = .1)
    appX=vec(sum(x[1:10:end,[3,4,6]], dims=2))
    rmsd(C, appX; normalize=:true)
end

p_lo_f_1=vcat(1.4,.5)
p_up_f_1=vcat(4,1)
p_vec_f_1=vcat(2.5,.9)
ResF1=optimize(loss_F_1,p_lo_f_1,p_up_f_1,p_vec_f_1,Fminbox(LBFGS()), # LBFGS is suitable for large scale problems
# Result=optimize(loss,p_lo,p_up,pvec,SAMIN(rt=.99),
                        Optim.Options(outer_iterations = 10,
                                                iterations=10000,
                                                show_trace=true,
                                                show_every=1))
pμ=vcat(Optim.minimizer(ResF1))
parf1=copy(par); parf1[2]=pμ[1]; μ1=pμ[2]

## Optimisation of β and order of incommensurate fractional order model
function loss_F_8(pμ)
        par[2] = pμ[1] # infectious rate
        μ = pμ[2:9] # order of derivatives
        if size(X0,2) != Int64(ceil(maximum(μ))) # to prevent any errors regarding orders higher than 1
                indx=findall(x-> x>1, μ)
                μ[indx]=ones(length(indx))
        end
        _, x = FDEsolver(SIR, tspan, X0, μ, par, h = .1)
    appX=vec(sum(x[1:10:end,[3,4,6]], dims=2))
    rmsd(C, appX; normalize=:true)
end

```

```
        p_lo=vcat(1.4,.5*ones(8))
        p_up=vcat(4,ones(8))
        pvec=vcat(2.5,.9*ones(8))
        ResF8=optimize(loss_F_8,p_lo,p_up,pvec,Fminbox(LBFGS()), # LBFGS is suitable for large
        ↪  scale problems
    # Result=optimize(loss,p_lo,p_up,pvec,SAMIN(rt=.99),
                            Optim.Options(outer_iterations = 10,
                                                    iterations=10000,
                                                    show_trace=true,
                                                    show_every=1))
    pp=vcat(Optim.minimizer(ResF8))
    parf8=copy(par); parf8[2]=pp[1]; μ8=pp[2:9]
```

After optimization, we visualize the results. We employ the `FDEsolver()` to generate model solutions for each approach, and the Root Mean Square Deviation (RMSD) evaluates the quality of fits.

```
    ## plotting
        DateTick=Date(2020,3,3):Day(1):Date(2020,5,17)
        DateTick2= Dates.format.(DateTick, "d u")

    t1, x1 = FDEsolver(SIR, tspan, X0, ones(8), par1, h = .1) # solve ode model
    _, xf1 = FDEsolver(SIR, tspan, X0, μ1*ones(8), parf1, h = .1) # solve commensurate fode
    ↪  model
    _, xf8 = FDEsolver(SIR, tspan, X0, μ8, parf8, h = .1) # solve incommensurate fode model

        X1=sum(x1[1:10:end,[3,4,6]], dims=2)
        Xf1=sum(xf1[1:10:end,[3,4,6]], dims=2)
        Xf8=sum(xf8[1:10:end,[3,4,6]], dims=2)

        Err1=rmsd(C, vec(X1)) # RMSD for ode model
        Errf1=rmsd(C, vec(Xf1)) # RMSD for commensurate fode model
        Errf8=rmsd(C, vec(Xf8)) # RMSD for incommensurate fode model
        [...]
```

References

[1] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM Review 59*, 1 (2017), 65–98.

[2] Khalighi, M., Sommeria-Klein, G., Gonze, D., Faust, K., and Lahti, L. Quantifying the impact of ecological memory on the dynamics of interacting communities. *PLOS Computational Biology 18*, 6 (06 2022), 1–21.

[3] Morin, A., Urban, J., and P, S. A quick guide to software licensing for the scientist-programmer. *PLoS Computational Biology 8*, 7 (2012), e1002598.

[4] Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., and Teal, T. K. Good enough practices in scientific computing. *PLoS computational biology 13*, 6 (2017), e1005510.