Jedy: Julia Evolutionary Dynamics

Lucien Rae Gentil
lgen0003@student.monash.edu

Julián García
julian.garcia@monash.edu

October 10, 2022

Abstract

Jedy is a package developed as a lightweight framework for building reliable and efficient agent-based simulations in Julia. The primary focus is to provide abstractions that allow for quick, intuitive, and reliable development of agent-based evolutionary models in a way that leverages the distinct design and strengths of the Julia programming language. This report, adapted from the package's online documentation, overviews the key background information, design philosophy, and a guided example for the Jedy package.

Source code: https://github.com/lucrae/Jedy.jl
Online documentation: https://lucrae.github.io/Jedy.jl

1 Background

1.1 Game theory

The field of game theory studies the strategic interactions between agents, most commonly analysing behaviour in models that result from certain actions and payoffs. The analysis of game theory scenarios lead into reasoning real-world complex systems in economics, political science, biology, computing, and social structures, primarily dissecting processes of decision making and behavioural relations.

1.2 Agent-based simulation

Agent-based simulations are a highly important scientific computing task in game theory. While mathematical analysis can provide closed-form or sound theoretical results to problems, developing and running simulations to play out the actions and interactions of autonomous agents in the theoretical of game theory are extremely useful in proving or disproving hypotheses, or revealing new insights into the model and the greater problems of social conflict, dilemmas, and cooperation that the model represents. Agent-based simulations combined with evolutionary processes are highly useful to analyse the outcomes and *dynamics* of repeated actions and payoffs between agents.

1.3 Scientific computing and the Julia language

The compromises between low-level and high-level programming languages presents a challenge for scientific computing.

Relatively lower-level languages (by modern standards) such as C/C++ provide highly efficient performance, but because their design is oriented towards systems programming they have a strong specificity for low-level or "machine-oriented" operations such as memory allocation that bloats development time and makes them very unsuitable for fast or intuitive prototyping—an attribute very important for scientific computing.

Higher-level languages such as Python have consequently become popular in many areas of modern scientific computing due to a "friendlier" approach to readability and automation of low-level operations, but the implementation of being easy and dynamic, mainly through being interpreted—rather than compiled—also produces far less optimal performance. Especially in computing tasks that rely on optimised, repeated operations, such as simulations, the slowness of Python and other popular high-level languages produces magnitudes of inefficiency.

First appearing in 2012–relatively recent for a programming language–the Julia programming language has emerged as a solution to this rift between ease and performance, providing the high-level dynamic readability of Python, but through sophisticated design decisions such as a just-in-time (JIT) compiler and eager evaluation, it delivers on the optimised performance seen by the likes of C/C++, achieving petaFLOP computations. Julia, as a result, presents new and exciting potential for new works in scientific computing.

2 Design of the Jedy package

Jedy (name originating from from **J**ulia **E**volutionary **DY**namics) is a package developed to provide a **lightweight framework for building reliable and efficient agent-based simulations in Julia**. The primary focus is to provide abstractions that allow for quick, intuitive, and reliable development of agent-based ideas, as well as already implemented and tested functions to run commonly-used evolutionary algorithms such as imitation processes and Wright-Fisher evolution.

A very important aspect to the development of this package is an understanding of Julia's design as a language, aiming for Jedy to be *idiomatic* to the language to fully leverage its strengths.

2.1 Typing

The type system of a programming language regulates the usage of different data types/structures in a program. Formally, Julia is *strongly typed*, that is each variable/object must remain the same type in its lifetime (as opposed to weak, like Python, where terms can change type), and *dynamically type-checked*, that is the validity of this typing is checked at run-time (as opposed to static type-checking, like C++, done at compile time).

Dynamic type checking allows the types of variables to be "inferred", such that Julia variables can be defined without any explicit type specified:

```
a = 4
b = "hello"
```

While convenient, a notable concern is the ambiguity of variable types, especially in the context of parameters of functions where using the correct type may be useful, or different types may require different functions (overloading). To solve this, Julia allows for rich, in-built type hinting. For example:

```
function divide(a::Int, b::Int) :: Float64
    return a/b
end
```

Here it is clear that the divide function will take in two integers, and return a floating point number.

For the goal of "reliability" for Jedy, this element of Julia's typing is highly important. Consider:

```
function imitation_process!(agents::Array{Jedy.Agent},
                            fitnesses::Array{Float64},
                            w::Float64)
    # ...
end
```

In this function all the types are clearly set out, including the core custom type of the "Agent." Also note there is no returning type, thus this function does not return anything, and the Julia convention of this function's naming using a "!" denotes that it uses in-place operations on the parameter types (i.e. the mutable array of agents can be manipulated).

The philosophy of clear typing, even when the implicit typing is possible, is the clear *Julianic* way of a package such as Jedy to promote reliability.

2.2 Composition over inheritance

While object-oriented languages such as Java will greatly favour inheritance, using children of parents to represent sophisticated objects, Julia does not support inheritance. Instead, there aren't sophisticated objects but rather sophisticated combinations of objects, most ideally implemented functionally (that is, using functions).

For Jedy, this means not creating seperate special different types of ‘Agent‘s or encapsulating simulations within an object, but rather focusing on various functions and specifying what types can be given to them. For scientific computing, this approach has a huge advantage on reliability, and at times efficiency when evading overly-clunky object operations, by minimising the complexity of relationships.

2.3 Thorough documentation

Documentation is extremely important to a package’s utility. For Jedy, a modern use of Julia’s `Documenter.jl` was used to generate a sophisticated static website, hosted at `https://lucrae.github.io/Jedy.jl/stable` in a conventional documentation structure with an introduction, usage, examples, as well as API documentation generated from Jedy’s in-script docstrings with the `@autodocs` feature. Linked with the use of git tagging to clearly version documentation, this makes for a thorough resource to learn and understand Jedy with easy scalibility to future developments.

2.4 Open-sourcing

Julia makes a very interesting choice for packages (importable libraries/tools developed for Julia) to naturally be git repositories (contribution/version control directories). As this makes open-sourcing extremely easy, almost every Julia repository is open-source. Jedy is no different, and the source-code is accessible and able to be contributed to at: `https://github.com/lucrae/Jedy.jl`.

2.5 Logging customisation

Being able to record intermediate and resulting data is key to scientific computing, so Julia implements as a first-class feature the use of a custom log, with the flexible solution of providing a function that takes in the agents and returns some array that will constitute each line in the outputted CSV.

3 Example

The Prisoner’s Dilemma is a staple problem in game theory. Two agents must choose to either defect or cooperate with the other, without knowing the others’ choice. Defecting when the other cooperates provides the highest reward T and lowest S to the other, but both defecting provides a low reward P and both cooperating provides a decent reward R such that $T > R > P > S$. The key insight from analysing behaviour in the Prisoner’s Dilemma is that agents will tend to defect instead of cooperate as the Nash Equilibrium, even though this will result in an ultimately overall worse payoff than cooperation. This is of fundamental interest to the study of how unoptimal consequences of cooperation/defection in the real world can occur.

We can use Jedy to simulate agents playing the Repeated Prisoner's Dilemma with an evolutionary process to see the success of defecting vs. cooperating.

3.1 Define population

First, let's make a population of 100 agents with a randomly-assigned Boolean attribute `behaviour` that defines if they defect (true) or cooperate (false):

```
import Jedy
import Random

N = 100
agents = Array{Jedy.Agent}(undef, N)
for i in 1:N
    agents[i] = Jedy.Agent(Dict("behaviour" => rand(Bool)))
end
```

3.2 Define way of determining payoffs/fitness

We use a straight-forward function for two agents to play the Prisoner's Dilemma:

```
function play_prisoners_dilemma(agent_a::Jedy.Agent, agent_b::Jedy.Agent) :: Tuple{Int, Int}
    T, R, P, S = 3, 2, 1, 0
    action_a = agent_a.body["behaviour"]
    action_b = agent_b.body["behaviour"]
    if action_a && action_b
        return (P, P)
    elseif action_a && !action_b
        return (T, S)
    elseif !action_a && action_b
        return (S, T)
    else
        return (R, R)
    end
end
```

We can then use this for our `compute_fitness` function. Here we compute fitness by the total payoff each agent accumulates against 1000 random opponents.

```
function compute_fitnesses(agents::Array{Jedy.Agent}) :: Array{Float64}
    n = length(agents)
    fitnesses = zeros(n)

    for f in 1:n
        for _ in 1:100 # Agent[f] plays 100 prisoner's dilemmas
            r = f
            while r == f # Pick non-self opponent
                r = rand(1:n)
            end
            payoff_f, payoff_r = play_prisoners_dilemma(agents[f], agents[r])
            fitnesses[f] += payoff_f
            fitnesses[r] += payoff_r
        end
    end
    return fitnesses
end
```

3.3 Define evolution process

Let's use an imitation process:

```
# Imitation process with w=0.2 (weak selection)
imitation_process! = Jedy.EvolutionProcesses.create_imitation_process(0.2)
```

3.4 Run simulation

Our final simulation puts all these parts together and runs for 200 epochs:

```
Jedy.run_simulation!(agents,
                     compute_fitnesses,
                     imitation_process!,
                     200)
```

Printing out the agents array will show that now all agents have behaviour `1` (defecting), evidencing (with repetitions) the convergence to defection.

(Optional) Logging to CSV

It's very useful to be able to log values from each epoch's agents. We can add a highly-customisable logger function easily:

```
# Logs the behaviour of each agent
function logger(agents::Array{Jedy.Agent})
    vals = map(a -> a.body["behaviour"], agents)
    return "temp.log", vals
end

# Run simulation with logger
Jedy.run_simulation!(agents,
                     compute_fitnesses,
                     perform_imitation_process!,
                     200,
                     logger)
```

Full code is available at: https://github.com/lucrae/Jedy.jl/tree/master/examples/pd_imitation.jl.

4 Future work

The project so far has been on the groundwork and approach of Jedy, but there are many more features and developments that could contribute well to the package in future work. Most prominently:

- Implementation of more evolutionary processes.
- Direct visualisation tools (i.e. shorthands for common plots).
- Advanced debugging (i.e. customisable verification tools).

As Jedy is open-source (https://github.com/lucrae/Jedy.jl), the progression of new features are clear, and of course open to contributions as a living project.