O código base para a simulação está num módulo que contém os tipos e procedimentos necessários.

```
[3]  module DyDeo2
     using LightGraphs, MetaGraphs, Distributions, DataFrames
     using Parameters, StatPlots, ProgressMeter, JLD2,  StatsBase

     # package code goes here
     include("basefns.jl")
     include("runfns.jl")

     end # module
```

O arquivo **basefns.jl** contêm o seguinte código:

```
[ ]  #Structs for Agents and Beliefs --------------------

     abstract type  AbstractAgent end
     abstract type AbstractBelief end


     "Concrete type for Agents' beliefs;
     comprised of opinion, uncertainty and an id (whichissue)"
     mutable struct Belief{T1 <: Real, T2 <: Integer}
         o::T1
         σ::T1
         whichissue::T2
     end


     """
         mutable struct Agent_o{T1 <: Integer, T2 <: Vector, T3 <: Real,
                                T4 <: Vector, T5 <: Tuple} <: AbstractAgent

     Concrete type for an Agent which only change its opinion.

     Fields:
      - id::Integer
      - ideo:: Vector
      - idealpoint::Real
      - neighbors::Vector
      - certainissues::Vector
      - certainparams::Tuple

     """
```

```julia
mutable struct Agent_o{T1 <: Integer, T2 <: Vector, T3 <: Real,
                       T4 <: Vector, T5 <: Tuple} <: AbstractAgent
    id::T1
    ideo::T2
    idealpoint::T3
    neighbors::T4
    certainissues::T4
    certainparams::T5
end


"Concrete type for an Agent which changes both opinion and uncertainty"
mutable struct Agent_oσ{T1 <: Integer,T2 <: Vector,T3 <: Real,
                        T4 <: Vector, T5 <: Tuple} <: AbstractAgent
    id::T1
    ideo::T2
    idealpoint::T3
    neighbors::T4
    certainissues::T4
    certainparams::T5
end


#= "Constructors" for Beliefs, Agents and Graphs
- All I need for the initial condition
=#
•

"""
     createbetaparams(popsize::Integer)

Creates a list of parameters for posterior instantiation of Belief
"""
function createbetaparams(popsize::Integer)
    αs = linspace(1.1, 100, popsize) |> shuffle
    βs = linspace(1.1, 100, popsize) |> shuffle
    betaparams = collect(zip(αs,βs))
    return(betaparams)
end

"""
    create_belief(σ::Real, issue::Integer, paramtuple::Tuple)

Instantiates beliefs;
note o is taken from a Beta while σ is global and an input
"""
function create_belief(σ::Real, issue::Integer, paramtuple::Tuple)
    o = rand(Beta(paramtuple[1],paramtuple[2]))
    belief = Belief(o, σ, issue)
end
```

```julia
function create_idealpoint(ideology)
    opinions = []
    for issue in ideology
        push!(opinions,issue.o)
    end
    ideal_point = mean(opinions)
end

"""
    create_agent(agent_type,n_issues::Integer,
id::Integer, σ::Real, paramtuple::Tuple)

Instantiates  agents; something missing in terms of design
"""
function create_agent(agent_type,n_issues::Integer,
        id::Integer, σ::Real, paramtuple::Tuple)

    ideology = [create_belief(σ, issue, paramtuple) for
                issue in 1:n_issues ]
    idealpoint = create_idealpoint(ideology)

    if agent_type == "mutating o"
        agent = Agent_o(id,ideology, idealpoint,[0], [0], paramtuple)
    elseif agent_type == "mutating o and sigma"
        agent = Agent_oσ(id,ideology, idealpoint,[0],[0], paramtuple)
    else
        println("specify agent type: mutating o or mutating o and sigma")
    end
    return(agent)
end

"""
    createpop(agent_type, σ::Real,  n_issues::Integer, size::Integer)

Creates an array of agents
"""
function createpop(agent_type, σ::Real,
        n_issues::Integer, size::Integer)
    betaparams = createbetaparams(size)
    population = [create_agent(agent_type, n_issues,
            i,σ, betaparams[i]) for i in 1:size]
end

"""
    createintransigents!(pop,propextremists::AbstractFloat)

turn some agents into extremists; that is,
given a number or proportion of extremists and
issues it makes the σ of some issues and some agents into ≈ 0 (1e-20)
"""
function createintransigents!(pop,propintransigents::AbstractFloat;
        position = "random")
```

```julia
    n_issues = length(pop[1].ideo)
    nintransigents = round(Int, length(pop) * propintransigents)

    if position == "random"
        whichintransigents = sample(1:length(pop),nintransigents,
                                    replace = false)
    elseif position == "extremes"
        # gives an error if nintransigents > len(extremistsid)
        extremistsid = map( x-> x.id,
                            filter(x-> ( x.idealpoint < 0.2) ||
                  (x.idealpoint > 0.8),
                                   pop))
        if nintransigents > length(extremistsid)
            error("there aren't enough agents on the extremes;
                  try a lower prop_intran")
            else
            whichintransigents  = sample(extremistsid,nintransigents,
                                         replace = false)
        end

    elseif position == "center"
        centristsid = map( x-> x.id,
                            filter(x-> ( x.idealpoint > 0.25) &&
                  (x.idealpoint < 0.75),
                                   pop))
        if nintransigents > length(centristsid)
            error("there aren't enough agents on the center;
                  try a lower prop_intran")
        else
            whichintransigents = sample(centristsid,nintransigents,
                                        replace = false)
        end
    else
        error("wrong position argument; correct: random,extremes,center")
    end

    for i in whichintransigents
        whichissues = sample(1:n_issues, 1,
                             replace = false)
        pop[i].certainissues = whichissues
        for issue in whichissues
            pop[i].ideo[issue].σ = 1e-20
        end
    end
end




"""
    creategraphfrompop(population, graphcreator)
```

```julia
Creates a graph; helper for add_neighbors!
"""
function creategraphfrompop(population, graphcreator)
    graphsize = length(population)
    nw = graphcreator(graphsize)
    return(nw)
end

"""
    add_neighbors!(population, nw)

adds the neighbors from nw to pop; the fn neighbors is from LightGraphs
"""
function add_neighbors!(population, nw)
    for i in population
        i.neighbors = neighbors(nw,i.id)
    end
end

#= Interaction functions
=#
•
"""
    getjtointeract(i::AbstractAgent,  population)
Chooses and returns a neighbor for i
"""
function getjtointeract(i::AbstractAgent,  population)
    whichj = rand(i.neighbors)
    j = population[whichj]
end



"""
    pick_issuebelief(i::AbstractAgent, j::AbstractAgent)

Takes two agents and returns a tuple with:
 * which issue they discuss
 * i and j beliefs
"""
function pick_issuebelief(i::AbstractAgent, j::AbstractAgent)
    whichissue= rand(1:length(i.ideo))
    i_belief = i.ideo[whichissue]
    j_belief = j.ideo[whichissue]
    return(whichissue, i_belief, j_belief)
end

"""
    calculate_pstar(i_belief::Belief, j_belief::Belief, p::AbstractFloat)

helper for posterior opinion and uncertainty
"""
function calculate_pstar(i_belief::Belief,
```

```
        j_belief::Belief, p::AbstractFloat)
    numerator = p * (1 / (sqrt(2 * π ) * i_belief.σ ) )*
    exp(-((i_belief.o - j_belief.o)^2 / (2*i_belief.σ^2)))
    denominator = numerator + (1 - p)
    pₚ  = numerator / denominator
    return(pₚ)
end

"""
    calc_posterior_o(i_belief::Belief, j_belief::Belief,
p::AbstractFloat)

Helper for update_step
Input = beliefs in an issue and confidence paramater;
Output = i new opinion
"""
function calc_posterior_o(i_belief::Belief, j_belief::Belief,
        p::AbstractFloat)
    pₚ = calculate_pstar(i_belief, j_belief, p)
    posterior_opinion = pₚ * ((i_belief.o + j_belief.o) / 2) +
        (1 - pₚ) * i_belief.o
end

"""
    calc_pos_uncertainty(i_belief::Belief,
j_belief::Belief, p::AbstractFloat)
helper for update_step
"""
function calc_pos_uncertainty(i_belief::Belief,
        j_belief::Belief, p::AbstractFloat)
    pₚ = calculate_pstar(i_belief, j_belief, p)
    posterior_uncertainty = sqrt(i_belief.σ^2 * ( 1 - pₚ/2) + pₚ *
                                 (1 - pₚ) *
                                 ((i_belief.o - j_belief.o)/2)^2)
end

"""
    update_o!(i::AbstractAgent, which_issue::Integer,
posterior_o::AbstractFloat)

 update_step for changing opinion but not belief

"""

function update_o!(i::AbstractAgent, which_issue::Integer,
        posterior_o::AbstractFloat)
    i.ideo[which_issue].o = posterior_o
    newidealpoint = create_idealpoint(i.ideo)
    i.idealpoint = newidealpoint
end

"""
```

```julia
    update_oσ!(i::AbstractAgent,issue_belief::Integer,
posterior_o::AbstractFloat, posterior_σ::AbstractFloat)

update_step for the version with
changing opinions and changing uncertainty
"""
function update_oσ!(i::AbstractAgent,issue_belief::Integer,
        posterior_o::AbstractFloat, posterior_σ::AbstractFloat)
    i.ideo[issue_belief].o = posterior_o
    i.ideo[issue_belief].σ = posterior_σ
    newidealpoint = create_idealpoint(i.ideo)
    i.idealpoint = newidealpoint
end



"""
    updateibelief!(i::Agent_o, population, p::AbstractFloat )

Main update fn; has two methods depending on the agent type

"""
function updateibelief!(i::Agent_o, population, p::AbstractFloat )

    j = getjtointeract(i,population)
    whichissue,ibelief,jbelief = pick_issuebelief(i,j)
    pos_o = calc_posterior_o(ibelief,jbelief, p)
    update_o!(i,whichissue,pos_o)
end



function updateibelief!(i::Agent_oσ, population,p::AbstractFloat )

    j = getjtointeract(i, population)
    whichissue,ibelief,jbelief = pick_issuebelief(i,j)
    pos_o = calc_posterior_o(ibelief,jbelief, p)
    pos_σ = calc_pos_uncertainty(ibelief, jbelief, p)
    update_oσ!(i,whichissue,pos_o, pos_σ)
end



"""
    ρ_update!(i::AbstractAgent,  σ::AbstractFloat, ρ::AbstractFloat)

fn for noise updating; note it returns
a randomly taken o(t+1) = o(t) + r,
but the new σ is the initial one
"""
function ρ_update!(i::AbstractAgent, ρ::AbstractFloat)
    whichissue = rand(1:length(i.ideo))
    r =  rand(Normal(0,ρ))
    if (i.ideo[whichissue].σ != 1e-20)
```

```
            if i.ideo[whichissue].o + r > 1.0
                i.ideo[whichissue].o = 1.0
            elseif i.ideo[whichissue].o + r < 0.0
                i.ideo[whichissue].o = 0.0
            else
                i.ideo[whichissue].o += r
            end
            newidealpoint = create_idealpoint(i.ideo)
            i.idealpoint = newidealpoint
        end
    end
```

Já o arquivo **runfns.jl** possui o seguinte código:

```
[ ]     # Types needed for the simulation

        "Parameters for the simulation; makes the code cleaner"
        @with_kw struct DyDeoParam{R<:Real}
            n_issues::Int = 1
            size_nw::Int = 2
            p::R = 0.9
            σ::R = 0.1
            time::Int = 2
            ρ::R = 0.01
            agent_type::String = "mutating o"
            graphcreator = CompleteGraph
            propintransigents::R = 0.1
            intranpositions::String = "random"
        end

        ## Information Storing Fns
        #I'm going to initialize a dataframe and update it at each time step.
        """
            create_initialcond(agent_type, σ, n_issues, size_nw,graphcreator,
                        propintransigents; intranpositions = "random")

        this fn is a helper for all other fns used in the simulation
        """
        function create_initialcond(agent_type, σ, n_issues,
                        size_nw,graphcreator,
                        propintransigents; intranpositions = "random")
            pop = createpop(agent_type, σ, n_issues, size_nw)
            g = creategraphfrompop(pop,graphcreator)
            add_neighbors!(pop,g)
            createintransigents!(pop, propintransigents,
                position = intranpositions)
            return(pop)
```

```
    end

    """
        function createstatearray(pop,time)

    Creates an array with the agents' ideal points;
    it's an alternative to saving everything in a df

    """
    function createstatearray(pop,time)
        statearray = Array{Array{Float64}}(time+1)'
        statearray[1] = pullidealpoints(pop)
        return(statearray)
    end



    """
        create_initdf(pop)

    fn to initialize the df; it should store all the info I may need later.
    """
    function create_initdf(pop)
        df = DataFrame(time = Integer[], id  = Integer[],
                       ideal_point = Real[])
        for agent in pop
            time = 0
            push!(df,[time agent.id  agent.idealpoint ])
        end
        return df
    end

    """
        update_df!(pop,df,time)

    fn to update the df with relevant information.
    """
    function update_df!(pop,df,time)
        for agent in pop
            push!(df,[time  agent.id   agent.idealpoint ])
        end
        return(df)
    end

    "self-describing... it takes a population and returns an array
    of ideal points"
    function pullidealpoints(pop)
        idealpoints = Float64[]
        for agent in pop
            push!(idealpoints,agent.idealpoint)
        end
        return(idealpoints)
```

```
end

"""
    outputfromsim(endpoints::Array)
fn to turn extracted information into system measures;
pressuposes an array with some system state (set of agents attributes)
"""
function outputfromsim(endpoints::Array)
    stdpoints = std(endpoints)
    num_points = endpoints |> countmap |> length
    return(stdpoints,num_points)
end


#= Running Functions
=#
•
"""
    agents_update!(population,p, σ, ρ)

this executes the main procedure of the model:
one pair of agents interact and another updates randomly (noise).
"""
function agents_update!(population,p, σ, ρ)
    updateibelief!(rand(population),population,p)
    ρ_update!(rand(population), ρ)
    return(population)
end


"""
    runsim!(pop,df::DataFrame,p,σ,ρ,time)
this fn runs the main procedure iteratively while updating the df;

"""
function runsim!(pop,df::DataFrame,p,σ,ρ,time)
   for step in 1:time
        agents_update!(pop,p, σ, ρ)
        update_df!(pop,df,step)
    end
    return(df)
end

"""
    runsim!(pop,p,σ,ρ,time)

runs the main procedure iteratively then returns the final population

"""
function runsim!(pop,p,σ,ρ,time)
    for step in 1:time
        agents_update!(pop, p, σ, ρ)
```

```julia
    end
    return(pop)
end


"repetition of the sim for some parameters;"
function one_run(pa::DyDeoParam)
    @unpack n_issues, size_nw, p, σ, time, ρ,
    agent_type,graphcreator, propintransigents, intranpositions = pa
    pop = create_initialcond(agent_type, σ, n_issues,
        size_nw,graphcreator, propintransigents,
        intranpositions = intranpositions)
    initdf = create_initdf(pop)
    df = runsim!(pop,df,p,σ,ρ,time)
    return(df)
end


"""this runs the simulation without using any df;
this speeds up a lot the sim, but i can't keep track of
the system state evolution;
that is, I only save the end state
"""
function simple_run(pa::DyDeoParam)
    @unpack n_issues, size_nw, p, σ, time, ρ,
    agent_type,graphcreator, propintransigents, intranpositions = pa
    pop = create_initialcond(agent_type, σ, n_issues,
        size_nw,graphcreator, propintransigents,
        intranpositions = intranpositions)
    endpop = runsim!(pop,p,σ,ρ,time)
    return(endpop)
end


"""
    simstatesvec(pa::DyDeoParam)
runs the simulation and keeps each iteration configuration (ideal points)

"""
function simstatesvec(pa::DyDeoParam)
    @unpack n_issues, size_nw, p, σ, time, ρ,
    agent_type,graphcreator, propintransigents, intranpositions = pa
    pop = create_initialcond(agent_type, σ,
        n_issues, size_nw,graphcreator, propintransigents,
        intranpositions = intranpositions)
    statearray = createstatearray(pop, pa.time)

    for step in 1:time
        pop = agents_update!(pop,p, σ, ρ)
        statearray[step+1] =  pop |> pullidealpoints
      end
    return(statearray)
```

```julia
end


"""
    statesmatrix(statearray, time, size_nw)
fn to turn the system configurations (its state) into a matrix.
I need to plot the agents' time series.
Takes a lot of time (10 min for 1.000.000 iterations and 1000 agents)
"""
function statesmatrix(pa; time = pa.time , size_nw = pa.size_nw)
    a = Array{Float64}(time+1,size_nw)
    statesvec = simstatesvec(pa)

   @showprogress 1 "Computing..." for (step,popstate) in enumerate(states
        for (agent_indx,agentstate) in enumerate(popstate)
            a[step,agent_indx] = agentstate
        end
    end
    return(a)
end


"""
    sweep_sample(param_values; time = 250_000, agent_type = "mutating o")

this fn pressuposes an array of param_values where each column is
a param and each row is a parametization;
Then it runs the sim for each parametization and pushs
system measures to another array (the output array)
"""
function sweep_sample(param_values; size_nw = 500,
        time = 250_000, agent_type = "mutating o")
    Y = []
@showprogress 1 "Computing..." for i in 1:size(param_values)[1]
    paramfromsaltelli = DyDeoParam(size_nw =  round(Int,
                                              param_values[i,1]),
                                    n_issues = round(Int,
                                               param_values[i,2]),
                                    p = param_values[i,3],
                                    σ = param_values[i,4],
                                    ρ = param_values[i,5],
                                    propintransigents = param_values[i,6]
                                    time = time,
                                    agent_type = agent_type)
        out  =  (simple_run(paramfromsaltelli) |>
                pullidealpoints |>
                outputfromsim)
        push!(Y,out)
    end
    return(Y)
end
```

```
"""
    function getsample_initcond(param_values; time = 250_000,
agent_type = "mutating o")
returns Initstd and Initnips (outputfromsim from initialcond)

"""
function getsample_initcond(param_values; time = 250_000,
        agent_type = "mutating o")

    param_values[:,1] = round.(Int,param_values[:,1])
    param_values[:,2] = round.(Int,param_values[:,2])
    Y = Tuple{Float64,Int64}[]

    @showprogress 1 "Computing..." for i in 1:size(param_values)[1]
        paramfromsaltelli = DyDeoParam(size_nw = convert(Int,
                param_values[i,1]),
                n_issues = convert(Int,
                param_values[i,2]),
                p = param_values[i,3],
                σ = param_values[i,4],
                ρ = param_values[i,5],
                propintransigents = param_values[i,6],
                time = time,
                agent_type = agent_type)
        @unpack n_issues, size_nw, p, σ,
        time, ρ, agent_type,graphcreator,
        propintransigents, intranpositions = paramfromsaltelli
        pop = create_initialcond(agent_type, σ,
            n_issues, size_nw,graphcreator, propintransigents,
            intranpositions = intranpositions)
        out = pop |> pullidealpoints |> outputfromsim
        push!(Y,out)
    end
    return(Y)
end



function extractys(Ypairs)
    Ystd = Float64[]
    Ynips =  Int64[]
    for i in Ypairs
        push!(Ystd,i[1])
        push!(Ynips,i[2])
    end
    return(Ystd,Ynips)
end


"""
function multiruns(sigmanissues::Tuple; repetitions = 50)
```

```
    helper function to plot the box plots
"""
function multiruns(sigmanissues::Tuple; repetitions = 100)
    pa = DyDeoParam(n_issues = sigmanissues[1],
                               σ = sigmanissues[2],
                               size_nw = 500,
                               time = 1_000_000,
                               p = 0.9,
                               ρ = 1e-5,
                               propintransigents = 0.0,
                               intranpositions = "random")

    repetitionsout = Array{Float64}[]

@showprogress 1 "Multiruns"    for run in 1:repetitions
        singleout = simple_run(pa) |> pullidealpoints
        push!(repetitionsout,singleout)
    end
    return(repetitionsout)
end


function mkdirs(filename)
    !(filename in readdir(pwd())) ? mkdir(filename):
    println("dir $(filename) exists... no need to create one ")
end
```

Arquivos usados na análise da simulação estão no repositório.