The Dolo Modeling Framework

James Graham [1], Spencer Lyon [1], Pablo Winant [2], and Anastasia Zhutova [3]

[1]New York University

[2]Bank of England

[3]Paris School of Economics

June 28, 2017

Introduction

- Dolo is a toolkit for *describing*, *solving*, and *simulating* economic models
- Supported by IMF, Bank of England, and QuantEcon
- Two main components:
  1. Dolang: Symbolic manipulation, equation compiler[1]
  2. Dolo: Modeling language, solution algorithms [2]
- Borrows ideas from similar software: Dynare, CompEcon, RECS, ...
- Novel contributions or extensions:
  - Model API: flexible/extensible model description
  - Solution algorithms: global, non-linear, OBC, multiple algorithms (easy comparisons)
  - Platform: 100% free and open source, scalable and performant, reproducibility (jupyter notebooks/docker), leverage community written numerical tools

[1] Main Dolang authors: Spencer & Pablo
[2] Main Dolo authors: James, Spencer, Pablo, & Anastasia

Main selling points

- Serious development
  - Version control (git)
  - Continuous integration (automatic test execution)
  - Free/open source languages
- Flexible modeling language
  - OLG with many generations
  - RBC with catastrophic events
  - NK with ZLB
- Separate model description from solution
  - Focus on specifying model
  - Apply generic implementations of algorithms
- Technology stack
  - Julia makes rapid prototyping easy with respectable performance
  - Allows for incremental optimization of algorithm performance
  - Facilitates sharing and reproducibility

dolo

Dolang 1: symbolic manipulations

```
julia> time_shift(:(a(0) + p * b(1)), 2)
:(a(2) + p * b(3))

julia> csubs(:(a + b), Dict(:b => :(c/a), :c => :(2a)))
:(a + (2a) / a)

julia> Dolang.latex(steady_state(:(a(0) + p * b(1)/a(1))))
"a+\\frac{p b}{a}"
```

dolo

Dolang 2: Function compiler

```
ff = FunctionFactory(
     [:((1-δ)*k(-1) + i(0)), :(1-(c(1)/c(0))^(-γ)*β*R)],
     [(:k, -1), (:i, 0), (:c, 0), (:c, 1)],  [:γ, :δ, :β, :R]
)
myfunc = eval(make_function(ff))[1];
```

```
julia> myfunc([0.4, 0.1, 0.35, 0.4], [2.0, 0.1, 0.95, 1.02])
2-element Array{Float64,1}:
 0.46
 0.258109

julia> myfunc(Der{1}, [0.4, 0.1, 0.35, 0.4], [2.0, 0.1, 0.95, 1.02])
2×4 Array{Float64,2}:
 0.9  1.0   0.0       0.0
 0.0  0.0  -4.27462  3.74029

julia> myfunc(Der{4}, [0.4, 0.1, 0.35, 0.4], [2.0, 0.1, 0.95, 1.02])
2-element Array{Dict{NTuple{4,Int64},Float64},1}:
 Dict{NTuple{4,Int64},Float64}()
 Dict((3, 3, 3, 4)=>-9.83217,(3, 3, 3, 3)=>-6.15761,(3, 3, 4, 4)=>-407.678,(3,
```

Model formulation

- Symbol groups: states, controls, parameters, values, ...
- Equations:
  - Transition, arbitrage, value, reward
  - Rules for which symbol groups can appear at which times
  - Structure allows us to write model-agnostic algorithms
- Calibration: parameter values and initial values for variables
- Exogenous process: IID, AR1, MarkovChain, products of previous
- Other features (all optional)
  - Complementarities – state-dependent bounds on controls
  - Domain for state variables
  - Type of grid on domain – Cartesian, Smolyak, (pseudo-)random

Example 1 – RBC Model with AR1 productivity

```
symbols:
   exogenous: [e_z]        # shortname `m`
   states: [z, k]          # shortname `s`
   controls: [n, i]        # shortname `x`
definitions:               # re-usable definitions (recursively defined)
   y: exp(z)*k^alpha*n^(1-alpha)
   c: y - i
   rk: alpha*y/k
   w: (1-alpha)*y/n
equations:
   arbitrage:              # f(m, s, x, m(1), s(1), x(1))
       - chi*n^eta*c^sigma - w                      | 0.1 <= n <= 1.0
       - 1 - beta*(c/c(1))^(sigma)*(1-delta+rk(1))  | 0.01 <= i <= inf
   transition:             # s = g(m(-1), s(-1), x(-1), m)
       - z = rho*z(-1) + e_z
       - k = (1-delta)*k(-1) + i(-1)
exogenous: !Normal         # specify process for exogenous e_z
    Sigma: [[sig_z^2]]
calibration:               # parameter values and defaults for variables
    ...
domain:                    # domain for state variables
    z: [-sig_z, sig_z]
    k: [k*0.5, k*1.5]
options:                   # Type of grid over domain
    grid: !Cartesian
        orders: [20, 50]
```

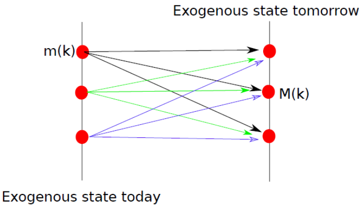
Discretization

- Generalized Discretized Process: three things
  1. time t nodes m_i
  2. time $t+1$ integration nodes m_{ij} and
  3. associated probabilities p_{ij}
- No restriction that $m_{i_1 j}$ relate to $m_{i_2 j}$
- Special cases
  - Markov Chain: $m_{ij} = \{m_i\}$, p_{ij} from transition matrix
  - Quadrature: N m_{ij} for each m_i
  - Unstructured: N_i m_{ij} for m_i
  - Bounded: m_{ij} never outside domain of m_i

dolo

Discretization

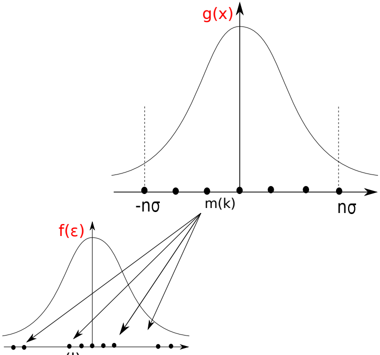
- Generalized Discretized Process: three things
  1. time t nodes m_i
  2. time $t+1$ integration nodes m_{ij} and
  3. associated probabilities p_{ij}
- No restriction that $m_{i_1 j}$ relate to $m_{i_2 j}$
- Special cases
  - Markov Chain: $m_{ij} = \{m_i\}$, p_{ij} from transition matrix
  - Quadrature: N m_{ij} for each m_i
  - Unstructured: N_i m_{ij} for m_i
  - Bounded: m_{ij} never outside domain of m_i



dolo

Discretization

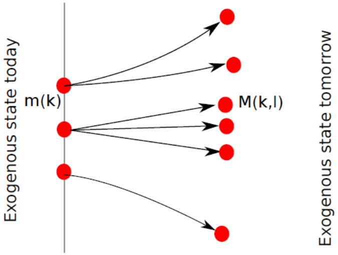
- Generalized Discretized Process: three things
  1. time t nodes m_i
  2. time $t+1$ integration nodes m_{ij} and
  3. associated probabilities p_{ij}
- No restriction that $m_{i_1 j}$ relate to $m_{i_2 j}$
- Special cases
  - Markov Chain: $m_{ij} = \{m_i\}$, p_{ij} from transition matrix
  - Quadrature: N m_{ij} for each m_i
  - Unstructured: N_i m_{ij} for m_i
  - Bounded: m_{ij} never outside domain of m_i



dolo

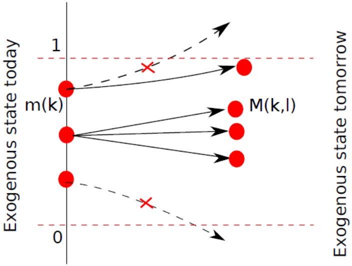
Discretization

- Generalized Discretized Process: three things
  1. time t nodes m_i
  2. time $t+1$ integration nodes m_{ij} and
  3. associated probabilities p_{ij}
- No restriction that $m_{i_1 j}$ relate to $m_{i_2 j}$
- Special cases
  - Markov Chain: $m_{ij} = \{m_i\}$, p_{ij} from transition matrix
  - Quadrature: N m_{ij} for each m_i
  - Unstructured: N_i m_{ij} for m_i
  - Bounded: m_{ij} never outside domain of m_i



Discretization

- Generalized Discretized Process: three things
  1. time t nodes m_i
  2. time $t+1$ integration nodes m_{ij} and
  3. associated probabilities p_{ij}
- No restriction that $m_{i_1 j}$ relate to $m_{i_2 j}$
- Special cases
  - Markov Chain: $m_{ij} = \{m_i\}$, p_{ij} from transition matrix
  - Quadrature: N m_{ij} for each m_i
  - Unstructured: N_i m_{ij} for m_i
  - Bounded: m_{ij} never outside domain of m_i



Algorithms

- Reference implementation of multiple algorithms
- Available algorithms for each model determined by *specification*
  - Dynamic Programming formulation:
    - Symbols needed: exogenous, state, control, reward, value
    - Equations needed: transition, felicity, value
    - Algorithms: VFI (Howard improvements)
  - Euler equation methods:
    - Symbols needed: exogenous, state, control
    - Equations needed: transition, arbitrage
    - Algorithms: perturbation, time iteration, improved time iteration, GSSA
  - Explicit expectation and response function:
    - Symbols needed: exogenous, state, control, expectations
    - Equations needed: transition, arbitrage, direct_response
    - Algorithms: direct time iteration, PEA
- Not performance optimized – but goal is better than most hand-written code
- Julia makes incremental optimizations easy

Algorithms

- Reference implementation of multiple algorithms
- Available algorithms for each model determined by *specification*
    - Dynamic Programming formulation:
        - Symbols needed: exogenous, state, control, reward, value
        - Equations needed: transition, felicity, value
        - Algorithms: VFI (Howard improvements)
    - Euler equation methods:
        - Symbols needed: exogenous, state, control
        - Equations needed: transition, arbitrage
        - Algorithms: perturbation, time iteration, improved time iteration, GSSA
    - Explicit expectation and response function:
        - Symbols needed: exogenous, state, control, expectations
        - Equations needed: transition, arbitrage, direct_response
        - Algorithms: direct time iteration, PEA
- Not performance optimized – but goal is better than most hand-written code
- Julia makes incremental optimizations easy

dolo

Algorithms

- Reference implementation of multiple algorithms
- Available algorithms for each model determined by *specification*
  - Dynamic Programming formulation:
    - Symbols needed: exogenous, state, control, reward, value
    - Equations needed: transition, felicity, value
    - Algorithms: VFI (Howard improvements)
  - Euler equation methods:
    - Symbols needed: exogenous, state, control
    - Equations needed: transition, arbitrage
    - Algorithms: perturbation, time iteration, improved time iteration, GSSA
  - Explicit expectation and response function:
    - Symbols needed: exogenous, state, control, expectations
    - Equations needed: transition, arbitrage, direct_response
    - Algorithms: direct time iteration, PEA
- Not performance optimized – but goal is better than most hand-written code
- Julia makes incremental optimizations easy

dolo

Algorithms

- Reference implementation of multiple algorithms
- Available algorithms for each model determined by *specification*
  - Dynamic Programming formulation:
    - Symbols needed: exogenous, state, control, reward, value
    - Equations needed: transition, felicity, value
    - Algorithms: VFI (Howard improvements)
  - Euler equation methods:
    - Symbols needed: exogenous, state, control
    - Equations needed: transition, arbitrage
    - Algorithms: perturbation, time iteration, improved time iteration, GSSA
  - Explicit expectation and response function:
    - Symbols needed: exogenous, state, control, expectations
    - Equations needed: transition, arbitrage, direct_response
    - Algorithms: direct time iteration, PEA
- Not performance optimized – but goal is better than most hand-written code
- Julia makes incremental optimizations easy

dolo

Example 2 – Algorithms and Discretized Processes

```
model = yaml_import(joinpath(
    Dolo.pkg_path, "examples", "models", "rbc_dtcc_ar1.yaml"
));

julia> dp = Dolo.discretize(Dolo.MarkovChain, model.exogenous);

julia> @time ti_res = time_iteration(model, dp, verbose=false);
  0.137005 seconds (1.11 M allocations: 71.930 MiB, 25.26% gc time)

julia> @time tid_res = time_iteration(model, dp,
    solver=Dict(:type => :direct), verbose=false
);
  0.027999 seconds (176.50 k allocations: 10.970 MiB, 35.58% gc time)

julia> ti_res.dr(2, [0.01])  # return (n, i) given [i_z] and [k]
2-element Array{Float64,1}:
 0.532983
 0.236385
```

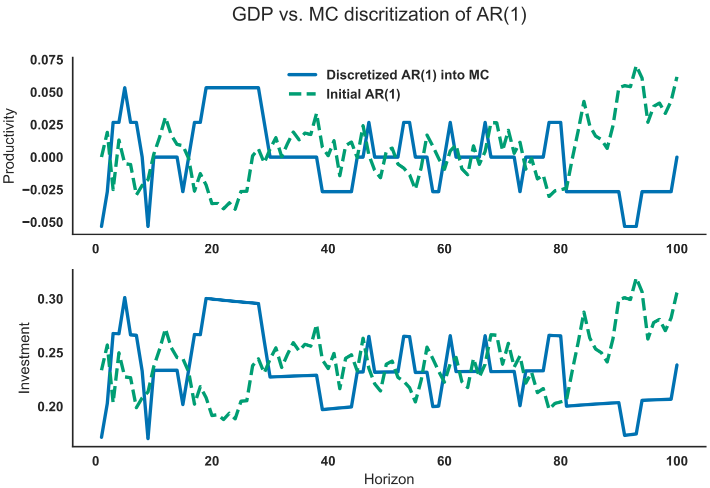
dolo

Example 2 – Algorithms and Discretized Processes

```
model = yaml_import(joinpath(
    Dolo.pkg_path, "examples", "models", "rbc_dtcc_ar1.yaml"
));

julia> dp = Dolo.discretize(Dolo.DiscretizedProcess, model.exogenous);

julia> @time ti_res = time_iteration(model, dp, verbose=false);
  0.871311 seconds (5.90 M allocations: 296.545 MiB, 15.12% gc time)

julia> @time tid_res = time_iteration(model, dp,
    solver=Dict(:type => :direct), verbose=false
);
  0.145959 seconds (941.11 k allocations: 47.970 MiB, 13.53% gc time)

julia> ti_res.dr([0.0], [0.01])  # return (n, i) given [z] and [k]
2-element Array{Float64,1}:
 0.532908
 0.236353
```

dolo

Example 2 – Algorithms and Discretized Processes



Reproducibility

- Collection of example models in dolo_models
- Jupyter notebook
  - text + latex + code + output + figures in one file
  - Easy to share, great for computational appendix to paper
- Docker:
  - Containers with pre-configured software + data
  - Others download *exact* environment and press run
  - Same environment on laptop or at scale in cloud
  - One line to open jupyter with example models and libraries:

    ```
    docker run -t -i -p8888:8888 albop/donolab
    ```
  - Then open browser (e.g. Google Chrome) to `localhost:8888`

dolo