AN INTRODUCTION TO HANS SOLVER - JULIA POWERED TOOLBOX FOR HETEROGENEOUS AGENTS STATIC EQUILIBRIUM MODELS WITH DISCRETE CHOICE AND INCOMPLETE MARKETS

MATEUSZ SZETELA

Abstract. HANS Solver is a toolbox for solving heterogeneous agents general equilibrium models with discrete choice. It has high degree of adaptability to multiple models declaration while sacrificing as little simplicity as possible. It posses built in multi-threading capabilities that do not require user intervention and allows to use hard drive, if model does not fit into operating memory of a machine. It is intended as introduction to Julia programming language for economists and alike. Further development is planned.

Introduction

Heterogeneous agents models grow on significance ever since Aiyagari [1994] published his works. A few example from the recent years are Auclert [2019], Kaplan et al. [2018]. Most of this models, despite their sophisticated microeconomic assumptions usually congregate on a few prices/variables that are crucial to determination of behavior of different agents in the model. Solution to model thus is achieved when given vector of these prices P^* equilibrium is achieved. I.e. supply meats demand, all markets clear and agents optimize their objective function, distribution of heterogeneous agents is stable, given P^* . Usually, model is solved in such a manner that there is a model function that accepts prices as parameters and returns updated prices: $M(P) = P'$. Thus, simplyfing, solution could be described as:

$$P^* : M(P^*) = P^*$$

Generally, there are two approaches to solve these models. The first one is based on discretizaition of solution space and defining model in recursive terms. This limits number of points from infinite to finite, thus creating an approximation of solution. The other modus operandi is defining model with differential equations and solving the problem in continuous time. HANS solver is designed for former approach. The later approach is much more efficient computationally, as it does not involves numerically costly random search algorithms, However, model must fulfill specific conditions as listed in Achdou et al. [2014] and differentials must be computable. The later condition can be difficult or impossible to fulfill in models

Date: 31.12.2020.

The development of HANS Solver and corresponding research is financed by National Science Centre, Poland. A project id: 2018/29/N/HS4/01225.

with discrete choice. Thus, despite its superiority in almost every field, continuous approach still has not rendered recursive method obsolete.

HANS Solver[1] was designed with a simple idea in mind: to allow easy implementation of multiple types of heterogeneous agents models with discrete choice while utilizing possibilities that Julia language provides (see Haessig and Besson [2018]). This creates a trade-of between toolbox generality (ability to service multiple types of models), simplicity (low entry cost, especially for people new to Julia language) and algorithm efficiency.

This implies that not all known methods to speed up calculations can be implemented, especially the some of the ones listed in Druedahl [2020] and Druedahl and Jørgensen [2017] as some of the algorithms are case sensitive and are difficult to put in general terms. Despite this issue the sole implementation of toolbox in Julia language makes it easy to run model using cloud computing. Even if that was not feasible in specific case, the toolbox still has high pedagogical value, as economists and analysts new to this framework can easily implement their models, solve them, modify if necessary and compare results. Furthermore, HANS Solver provides general framework and an experienced researcher can adjust it, extend or limit, to her needs.

The goal of this article is to introduce HANS Solver mechanics especially to people who have little to no experience with Julia coding language. The rest is organized as follows. In first section algorithm is described. In the second utilization is explained and performance is described.

1. Algorithm description

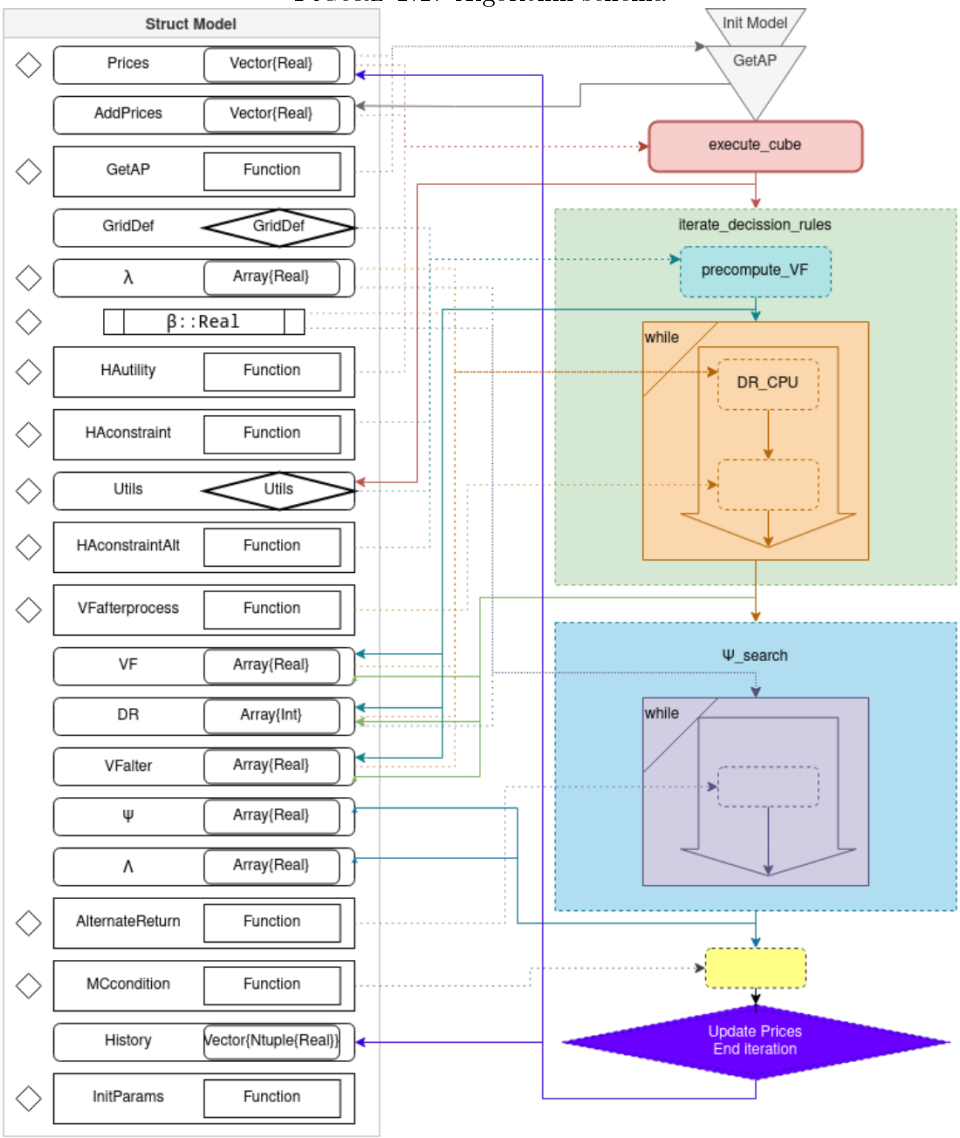
Since a picture is worth thousand words, therefore algorithm schema is presented in figure 1.1. On the left side is presented a Julia custom `struct` named `Model`. Diamonds to the left indicate which fields enter into `struct ModelInit`, a simplified version of the `Model`, that is an user defined input to HANS Solver. On the right side a simplified schema representing algorithm of a single iteration of HANS Solver. Precisely, it shows following step since when the model is calculated for given prices and until new updated prices are calculated. Dotted lines indicate inputs while continuous ones indicate outputs and algorithm flow. Rectangles indicate functions and empty ones (without text) indicate function defined by user (can be an empty function).

Most, if not all, built in functions can be called by using `?function_name` in REPL to generate help message within REPL.

The problem is solved using Optim.jl package, with simplex method. As an objective function, a sum of absolute differences between prices and updated prices is utilized.

[1] HANS: Heterogeneous Agents with Nonlinear Strategy

FIGURE 1.1. Algorithm schema



1.1. `Struct Model.` `Model` is basic element of HANS Solver. It contains all crucial elements of model: grid definition, value function matrix, probability distribution, etc. Within Julia's REPL they can be listed by using `fieldnames(Model)` command. The specific elements are described and explained below. Below, an `mymodel` is used for a name of `struct Model`.

1.1.1. *Prices.* As prices are understood variables that are input in an objective function. They are stored within a Vector, thus order is important. Within user defined functions they can be called in following way:

```
v1, v2, v3 = mymodel.Prices
```

1.1.2. *AddPrices.* This field represents additional prices. These prices do not enter ob- jective function that is used to calculate equilibrium convergence. Aside from this issue, they work exactly the same as field `Prices`.

1.1.3. *GetAP..* This field is an user defined function that return a vector of additional prices. As input it accepts Model and return a vector and stores it in AddPrices field.

1.1.4. *GridDef.* This field is a custom HANS Solver specific `struct GridDef`. This struct consists of a `Tuple` of symbols, that represent variable names, and grid definitions for state variables. The grid definitions are vectors of values that variables can take. These values must be ordered ascending. It is important to note, that it contains both present and future state variables. The future state variables must follow after present state variables. The order of variable names and grid definitions must be the same. The last field in both variable names and grid definitions is a status variable and grid must be defined as `Int`, as it is used for indexing.

Status variable is understood as variable representing (for example) employed or unemployed status. Thou if HANS Solver works with more than 2 fields, this grid definition must be vector of consecutive integers, starting at 1. The vector must also be defined as `Vector{Int}`. Within Hans Solver there is a function, `make_dimensions`, that accepts `Vector{String}` of state variables names (without differentiation between present and future) including name of status variable, and vector of grid definitions for state variables, including status variable (only number of grid points matter with regard to status variable). It returns properly defined `GridDef` struct.

Within user defined functions, `GridDef` can be referenced to get values of state variables in specific grid point:

```
v = get_vars((1,1,1,1,1),mymodel.GridDef)
```

This call generates a `NamedTuple` which fields can be accessed via v.VarName within user defined functions.

1.1.5. λ. This field represents status transition probability matrix for idiosyncratic shock. It is defined as `Matrix` - a 2 dimensional `Array`. Thus if there is more than one, together they must be represented as a matrix. For example, if there are two shocks: k, l and for each there are 2 points, then matrix would have to be:

$$\begin{bmatrix} p(k_1,l_1|k_1l_1) & p(k_1,l_1|k_1l_1) & p(k_1,l_1|k_1l_1) & p(k_1,l_1|k_1l_1) \\ p(k_1,l_2|k_1l_1) & \ddots & & \\ p(k_2,l_1|k_1l_1) & & \ddots & \\ p(k_2,l_2|k_1l_1) & & & \ddots \end{bmatrix}$$

The order of fields within matrix must correspond to order in `mymodel.GridDef.Grids[end]`.

1.1.6. β. This field represents discount factor. It is essential in calculating value function matrix and determining decision rules.

1.1.7. *HAutility.* This is a user defined function that represents heterogeneous agents utility function. It must accept the same arguments that other user defined function, `HAconstraint` returns. Generally, from the perspective of algorithm, it was possible to merge this filed with `HAconstraint`. However, it would make later (after the model is solved) calculations of inputs (like consumption) unnecessary tedious, thus this step was separated into two. The function must return single `Float` value, with `-Inf` for points that are not allowed (0 would also do if utility function is defined in R^+).

1.1.8. *HAconstraint.* This is another user defined function that is used by HANS Solver to calculate inputs to `HAutility`. It must accept three variables: Indices of grid points (as `NTuple{N,Int}`), Model, and additional arguments as single variable. The order is important and the last one is optional. As mentioned above, output must correspond to inputs of `HAutility`.

1.1.9. *Utils.* An another HANS Solver specific struct. It is automatically generated by algorithm. It is used to store utility function value for each grid point. It has two fields. The first one is a function that returns the value and is utilized within HANS Solver algorithm. The other one is a Vector of multidimensional arrays, that represent utility function value for each grid point. The number of arrays (or length of vector) corresponds to number of statuses (`length(mymodel.GridDef.Grids[end])`). Currently this struct can be of `Mmap` type or normal (RAM) type. `Mmap` means that arrays are stored on drive thus do not occupy RAM. This can be useful, as it is the largest object and if HANS Solver toolbox is used on single machine with limited RAM with modern SSD drives and motherboard this can save a lot of an operating memory while sacrificing relatively little computational speed.

1.1.10. *HAconstraintAlt.* Analogically to `HAconstraint`, this user defined function is used to calculate value function in alternate state (discrete choice). Alternate state can be represented by either a `Vector` or `Matrix` (compare to fields `DR` and `VF`).

1.1.11. *VFafterprocess.* An user defined function that is applied to `Model` after each iteration of value function matrix calculation. This step is optional - in that case an empty function should be implemented. Function must accept `Model` as argument. Operation within function must be applied directly to Model object (here called mymodel). This step is intended for nonstandard problems that require nonstandard calculations.

1.1.12. *VF.*. A `Matrix{Real}` , storing values of value function. Each column represents different status (compare to section 1.1.4). Thus values in each column are a vector which is linearized array, when referencing to `Utils`. To get values for status i to plot a heat-map (assuming there are 2 state variables), one can use following command:

```
heat_map_data = reshape(mymodel.VF[:,i], length(mymodel.GridDef.Grids[1:2])...)
```

This approach was used for multiple reasons. One is that `CartesianIndices` and `LinearIndices` within Julia are easily interchangeable and this representation makes this object compatible with λ (section 1.1.5) and thus whole user defined

model algorithm is more approachable. The next most important is that multidimensional arrays are quite RAM consuming.

1.1.13. *DR.*. A `Matrix{Int}` , storing decision rules (`LinearIndices` of `VF`). It is organized as `VF`.

1.1.14. *VFalter.* As `VF` this struct stores value function but for alternate state (discrete choice). This can be either a vector of values, where each value represents value function score for given status; or a matrix of size of `VF`. HANS Solver after calculating value function matrix (and before applying VFafterprocess) checks `VF` against `VFalter` and modifies both `VF` and `DR` if corresponding `VFalter` is greater. If `VFalter` is a vector then algorithm checks status vs position[2] status. If it is an array, it checks position and status vs position and status.

1.1.15. Ψ. Ψ[3] denotes probability distribution matrix. It is organized as both `DR` and `VF`. If model is defined correctly it should sum up with Λ to unity. Due to numerical precision this value could deviate thus scaling to unity is applied after distribution converges.

1.1.16. Λ. This struct is organized as `VFalter`. It is a brother to Ψ, as it denotes distribution of agents in alternate state.

1.1.17. *AlternateReturn.* This struct is another user defined function. It must accept as input Model object and return Ψ, Λ in this order. This function is utilized within Ψ`_search function` while loop and is applied before iterating over decision rules.

1.1.18. *MCcondition.* The last user defined function that is applied within HANS Solver algorithm. It must accept `Model` as an input and return new updated prices (section 1.1.1).

1.1.19. *InitParams.* The last user defined function that is stored in `Model struct`. It is required but can be empty function. It was added because of purely practical reason. Within Julia variables can be saved to hard drive using `JDL.jl` or `JDL2.jl` packages. After model is calculated a researcher may decide to change some parameters. Then again, after some time researcher may want to go back to previous specification. Saving struct Model alone does not allow to save constants[4] . Since it is strongly recommended to implement model parameters as constants for computational reasons, this function can contain a script that does that within `@eval begin...end` scope[5].

1.2. **Iteration Algorithm.** This subsection describes algorithm of a single iteration that starts with inputting prices for which partial equilibrium is calculated and updated prices, that solve partial equilibrium ceteris paribus are returned.

[2]Position means position on grid os state variables, excluding status variable.

[3]To get Greek letters type `\symbolname` and press Tab within Julia REPL. Here `\Psi`.

[4]It is possible but the more parameters the less convenient it becomes. Then after loading them they need to be declared as constants adding to inconvenience.

[5]In Julia defining constants within functions is not allowed (for a good reason) thus `@eval` workaround is recommended.

1.2.1. *Iteration initiation.* This part on the figure 1.1 is denoted by two downward facing triangles. The first one represents `struct Model` initiation: during first iteration `ModelInit` is transformed into `Model` with predefined default values, and during any other `Model` data fields (expect `History` and `Prices`) are cleared into default values.

The second step is responsible for calculating and loading into `Model` alternate prices which can be accessed by calling `mymodel.AddPrices` within user defined functions. After that the model is initiated and algorithm can proceed to next steps.

1.2.2. *Utility cube calculation.* In this step the most memory costly object is calculated. This step is implemented within `execute_cube` function. This HANS Solver function creates `Utils struct`, based on input parameters. In the first step, method is dete mined. It is done by selecting two specific version of functions. The first one is used within this step to create multidimensional array of utility function value for given state variables on a grid:

$$Utility_{\lambda_i}\left(s'_\iota | s_\kappa\right)$$

here κ are today state variables and ι are tomorrow state variables. Fort each status (denoted in equation above as λ_i) different array is computed and are stored as consecutive elements of vector in `mymodel.Utils.qb`. The second one is stored in `mymodel.Utils.f` and is used to access `mymodel.Utils.qb` in proper manner during value function and decision rules iteration. Such construction of `struct Utils` was decided upon for two reasons. The first one being multi device compatibility. It allows to build 'the cube' not only within RAM (CPU version) but also on hard drive (Mmap version) or, as is planned in the future, within VRAM (GPU computing is currently not supported). The other reason is limiting multidimensionality of object. Multidimensional arrays are memory costly and his steps eliminates one dimension limiting thus memory demand. Regarding future development, where GPU based computation for `Utils` is planed, it will allow store consecutive arrays for different status values on different GPUs - this point is essential since `Utils` is the largest object and VRAM of single GPU can be not enough to store it. Furthermore it will allow to fully absorb computational capabilities of multiple processing units of computer. Currently, there are two methods available:

(1) CPU: based on CPU computing and RAM memory. The whole Utils is stored in operating memory. Calculating utilizes multi-threading[6]. This method is the fastest so far but requires large amounts of free RAM which can be bottleneck on single machine.
(2) Mmap: based on `Mmap` functionality. CPU is still used with multi-threading to calculate values in RAM. However, once the cube is build it is stored on drive in working directory (files are stored as `/tmp/utils_`λ_i`.io` files where λ_i is number of status variable). Later it is accessed directly from drive. This makes calculation and later reading slower, depending on type

[6]It is necessary to start Julia REPL with multiple threads. Some IDEs have option within Julia extension to set number of threads Visual Studio Code being one of them. For Julia REPL multi-threading see https://docs.julialang.org/en/v1/manual/multi-threading/

of drive. However, for owners of modern motherboards and SSD drives this can be acceptable trade-off, especially if RAM is limited.

1.2.3. *Value function and decision rules iteration.* This part on the figure 1.1 is depicted by green rectangle. Firstly, the `VF` is precomputed by accessing `Utils`. The precomputation assumes that heterogeneous agents choose the same point ad infinitum. This does not involve status switching. The precomputation takes place simultaneously for all statuses (if sufficient number of threads is available):

$$VF'_{precomputed}(s, \lambda_i) = Utility_{\lambda_i}(s|s) + \beta VF_{precomputed}(s, \lambda_i)$$

This step not only is computed in parallel for all statuses but also is define on whole arrays. And since some array operations in Julia utilize multi-threading by default precomputation takes little time compared to value function iteration from zero to solution.

Once `VF` converges `VFalter` is calculated in similar fashion by applying user defined function `HAconstraintAlt`. Then `-Inf` in `VF` are replaced with corresponding `VFalter` values.

Once precomputation is done, old fashioned value function iteration starts. A decision rules convergence is used as criterion to finish while loop. The while loop iteration starts with calculating `VF` and `DR` utilizing multi-threading. In the next step `VFalter` is recalculated within internal while loop: `VFalter` is recursively calculated for given `VF` until values converge. It is important to note, that condition is assumed to be fulfilled after `VFalter` converges 7 times, to avoid to early convergence due to numerical artifact (this can happen especially during first iterations before `VFalter` gets close to solution). Then `VF` and `DR` are updated based on `VFalter`. If `VFalter` is a vector 0 determines choosing alternate state. If `VFalter` is a matrix negative integers determine choosing alternate state.

Afterwards, `VFafterprocess` is executed. It is an optional step. It was introduced for models with agent that acts on decision rules of heterogeneous agents and through her actions affects choices of them.

After this step convergence of `DR` is checked.

1.2.4. *Probability distribution calculation.* This part on the figure 1.1 is represented by blue rectangle. This is one of the more straightforward parts of code. Many algorithms use sparse transition matrix (Reiter [2002, 2009]) to quickly calculate this step: it is multiplied by probability distribution matrix until convergence is acquired. Such objects would be excessively large while contributing little to no computational speed. Furthermore, discrete choice furtherly complicates algorithm. Here different approach is applied.

Firstly, all agents are distributed in alternate state, for given λ[7]. This is one of two approaches that were considered. The first one was to initialize heterogeneous agents distribution on the beginning of the grid (in Julia REPL `mymodel.Ψ[1,:]= mymodel.λ^200`).

[7]For vector of alternate state it is first row of λ^{200} . For an array same values are calculated and then evenly spread out across whole vector of status.

In that case agents would get to the discrete choice points where they would chose alternate state and they would spread out from alternate state to ordinary state which required additional iteration. Thus by choosing to initialize agents in alternate state this process is omitted and algorithm accelerated.

he while loop is initiated afterwards and firstly user defined function `AlternateReturn` is applied and Ψ, Λ are updated. Then transition mechanics are considered. Since `DR` are defined as vectors gathered in a matrix (see section 1.1.13), algorithm iterates over fields of `DR` and thus probability distribution Ψ, Λ[8] is calculated. This process is done in parallel thus little to no computational speed is lost compared to sparse matrix method. In the next step probabilities are scaled to sum up to unity.

Once probability distribution converges while loop is closed.

1.2.5. *Updating Prices.* This step is fully user defined. After Ψ and `DR` are calculated it is possible to calculate aggregate stocks and flows and solve conditions for new prices that solve equilibrium ceteris paribus. After that, mymodel.Prices are changed to new ones and old ones are stored in `mymodel.History`. It is recommended to define problem as function and use either `Optim.jl` or `NLsolve.jl` to solve it.

1.3. **HANS Solver.** HANS Solver is organized around function hans_solver. It requires `ModelInit` as an input and optionally solution method for `Utils` (CPU or Mmap) as a string. If no method is applied "CPU" is selected. This function returns `Model` that contains prices that minimize objective function. Objective function is defined as:

$$\sum_{i=1}^{k} |p_i - p_i'|$$

Where p_i is i-th input price in the model, p_i' is a i-th updated price and number of prices in the model is k.

The function starts with transformation of `ModelInit` into `Model`. Then a Model struct is inputted into objective function as a function of `Prices`. Such a objective function is then implemented into optimize function from `Optim.jl` package. After that iterations described in section 1.2 are conducted until minimum of objective function is found or maximum number of iterations for optimize function is reached. Random search algorithm is selected to be a simplex method (Nelder and Mead [1965]). Other methods were considered (particle swarm, simulated annealing) but they required many more iterations for algorithmic purposes and took tenfold more time to arrive at solution. On this basis these methods were rejected, thou it is possible to easily tinker them in by manipulating `solver.base.jl` file.

[8] Ψ denotes ordinary state and Λ alternate one

2. How to use HANS Solver

The use of HANS Solver is straightforward as much as possible. Firstly, user must define `ModelInit`. As mentioned above, `ModelInit` is simplified `struct Model`. Fields required are marked with diamonds to the left on the figure 1.1. Specific instructions about what subsequent fields should contain are presented in section 1.1. Proper model declaration is obviously crucial. To test if model declaration is functional and if it produces desired effect a function `modeltest` was implemented. It accepts the same inputs as `hans_solver` but instead of finding solution it runs single iteration (section 1.2) and returns `Model`.

Once researcher decides that `ModelInit` is properly build it can be solved by typing:

```
hans_solver( mymodel, cube_method_as_string)
```

in Julia REPL. Afterwards algorithm will print a symbol in brackets, two question marks and old (inputed) prices. Symbol in brackets changes depending on a phase of algorithm. It starts with "I" for model initiation, changes to "C" for construction of Utils, then to "D" for value function iteration, afterwards to "Ψ" for probability distribution and finally to "U" for phase of updating prices.

The two question marks denote respectively value function loop (see section 1.2.3) and probability distribution loop (see section 1.2.4). If the loop is successful (finishes before reaching maximum number of iterations) it will change to an OK symbol. Otherwise it will switch to exclamation mark. In the later case, objective value function will be increased by 50% for value function failure to converge and 100% for probability distribution failure to converge. These punishment modifiers are govern by constants `DR_punish_degree` and `Ψ_punish_degree`, that are set to 0.5 and 1.0 respectively. Maximum number of iteration are ruled by constants `vf_iterator_maxiter=400` and `Ψ_maxiter=400`. A convergence criterion for probability distribution is set by `Ψ_precision = 10^-4`. This constants can be modified at user discretion.

3. Threads and performance.

Table 1. Number of threads enabled and relative speed

Number of threads	Relative speed	Share of time used by Garbage Collector
1	19.89	4.02%
4	8.22	32.55%
8	6.76	46.96%
16	4.88	56.88%
32	1.31	69.62%
64	1.0	59.17%

In a table 1 an outcome of a simple experiment is shown. For simple model, execution times of a single iteration was compared for a different numbers of threads on a machine with hyper-threading enabled. As a reference, computation time for 64 threads was taken. This was around 0.5 seconds for a single iteration. The first things that is obvious is raising share of time used by Julia's Garbage Collector.

This is mostly due to acceleration gained from using multiple threads. This is a major bottleneck if increase in performance by adding threads is considered. Reduction of Garbage Collector time from 4% to 2% would result in increase in performance on 64 threads of around 30%. This is worth taking into consideration when similar tools are developed. The test was run on a grid of $32 \times 32 \times 32 \times 32 \times 2$ points. This required nearly 6 GiB of RAM. Whole Julia process utilized at peaks 10.7 GiB of RAM on 64 threads. Using Mmap version of algorithm allowed to reduce this number to 7.8 GiB of RAM while sacrificing some performance. See the table 2 for reference. Increased time requirement for Garbage Collector is caused by much slower drive speed. Take note that 2.3 times increase in execution time is not disqualifying if four fold reduction of threads results in even greater decrease, despite bottlenecks. This highlight can make some researches consider Mmap version, especially if their model 'almost fits' into their computer operating memory.

TABLE 2. Mmap vs CPU on 64 threads

Algorithm	Relative speed	Share of time used by Garbage Collector	Peak RAM in GiB
Mmap	2.3	84.23%	7.8
CPU	1.0	59.17%	10.7

Results in the table 2 must be taken however with a grain of salt. The computation was made on modern motherboard with modern SSD drive that allows multi-access. Older machines without such innovations would preform much worse.

4. SUMMARY

The development of HANS Solver showed many challenges and opportunities regarding numerical computing in the economics. Algorithm allows for relatively convenient model declaration and allows to grasp computational capabilities of a machine with limited user intervention for non trivial economical models. Feedback from researchers is required to determine future course of project: should it aim at increasing performance, optimizing memory allocations, expand beyond CPU, improving ease of use or implement new functionalities, like Tauchen-like algorithm (Tauchen [1990]) to capture dynamics and aggregate shocks. Thou this feedback is essential for further development, HANS Solver can already be useful tool for research and pedagogical purposes. It can be an introduction to Julia environment for economists and popularize it by reducing the steepness of the learning curve for newcomers.

References

Yves Achdou, Jiequn Han, Jean-Michel Lasry, Pierre-Louis Lions, and Benjamin Moll. Heterogeneous agent models in continuous time. *Preprint*, 14, 2014.

Greg Kaplan, Benjamin Moll, and Giovanni L. Violante. Monetary policy according to hank. *American Economic Review*, 108(3):697–743, March 2018. URL https://ideas.repec.org/a/aea/aecrev/v108y2018i3p697-743.html.

John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

Michael Reiter. Computing heterogenous agent models when the distribution matters. Computing in Economics and Finance 2002 173, Society for Computational Economics, July 2002. URL https://ideas.repec.org/p/sce/scecf2/173.html.

Michael Reiter. Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control*, 33(3):649–665, March 2009. URL https://ideas.repec.org/a/eee/dyncon/v33y2009i3p649-665.html.

Adrien Auclert. Monetary policy and the redistribution channel. *American Economic Review*, 109(6):2333–2367, June 2019. URL https://ideas.repec.org/a/aea/aecrev/v109y2019i6p2333-67.html.

George Tauchen. Solving the stochastic growth model by using quadrature methods and value-function iterations. *Journal of Business & Economic Statistics*, 8(1):49–51, January 1990. URL https://ideas.repec.org/a/bes/jnlbes/v8y1990i1p49-51.html.

Pierre Haessig and Lilian Besson. Julia, my new friend for computing and optimization? 2018.

S Rao Aiyagari. Uninsured idiosyncratic risk and aggregate saving. *The Quarterly Journal of Economics*, 109(3):659–684, 1994.

Jeppe Druedahl. A guide on solving non-convex consumption-saving models. *Computational Economics*, pages 1–29, 2020.

Jeppe Druedahl and Thomas Høgholm Jørgensen. A general endogenous grid method for multi-dimensional models with non-convexities and constraints. *Journal of Economic Dynamics and Control*, 74:87–107, 2017.

Appendix A. Specification of a machine used for performance testing

Table 3. Machine specification

Element	Specification
CPU	AMD Ryzen Threadripper 2990WX 32-Core
Motherboard	X399 Taichi
RAM	3200 MHz 4×16GB
Storage	Samsung 970 PRO 1TB
Kernel	5.9.11-3-MANJARO x86_64