Comparing general equilibrium modeling in Julia and GAMS: An example using CSAVE

Y.-H. Henry Chen[ψ]

This version: March 12th, 2026

Abstract

As an open-source programming language, Julia has become increasingly popular in scientific computing, due in part to its nonproprietary nature, high performance, and extensive ecosystem of packages. The goals of this paper are threefold: (1) to provide readers with an introduction to constructing a computable general equilibrium (CGE) model in Julia using MPSGE.jl, a package syntactically comparable to the MPSGE subsystem in GAMS that offers a compact framework for model formulation; (2) to compare the time required to generate and solve the same model implemented in Julia with MPSGE.jl and in GAMS/MPSGE; and (3) to demonstrate the time-saving benefits of encapsulating the CGE model within a Julia function. Under a given computer specification and model design, GTAP 9 data at various levels of aggregation are used to evaluate model runtime. The results indicate that, overall, the Julia implementation runs faster: although it requires slightly longer model generation time, it achieves substantially shorter solving time, a difference that becomes more pronounced as data resolution increases. The study also confirms that wrapping the Julia model in a function can further accelerate the solving process. Using a two-region, 56-sector setting as an example, the time-saving results are supported by statistical tests at the 1% significance level.

[ψ] Research Scientist, Center for Sustainability Science and Strategy, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

1. Introduction

Computable general equilibrium (CGE) models have been widely employed for decades to evaluate the economic impacts of alternative policies and scenarios, beginning with the seminal work of Johansen (1960). Although advances in computing power have reduced the time required to solve large-scale CGE models, the influence of programming languages on model runtime has received far less attention. Aruoba and Fernández-Villaverde (2015) compare the runtime of several programming languages and find that C^{++} and Fortran are the fastest. However, their analysis focuses on a stochastic neoclassical growth model rather than CGE models. Horridge et al. (2013) consider several versions of a CGE model implemented in GEMPACK and GAMS and find that GEMPACK runs faster. Nevertheless, implementations in other programming languages are beyond the scope of their study.

A variety of programming languages have been used in CGE modeling. Among them, GAMS (GAMS Development Corporation, 2025) and GEMPACK (Pearson and Powell, 2014) are used by around 80% of CGE modelers (Horridge et al., 2013). Within the GAMS environment, the MPSGE subsystem (Rutherford, 1999) has become one of the most widely used frameworks for specifying CGE models. While these specialized tools remain prevalent, the increasing availability of high-performance general-purpose programming languages has opened new possibilities for building and solving CGE models. A prominent example is Julia (Bezanson et al., 2017), an open-source programming language that has gained growing popularity in scientific computing due to its nonproprietary nature, extensive ecosystem of packages, and high performance, enabling code to run at speeds close to C or Fortran (Bezanson et al., 2017; Lubin and Dunning, 2015).

In this paper, I compare the performance of a CGE model implemented in Julia using the MPSGE.jl package (Anthoff et al., 2025) with an equivalent model implemented in GAMS/MPSGE. The goals of the paper are threefold: (1) to provide readers with guidance on constructing a CGE model in Julia using MPSGE.jl, a package structurally comparable to the MPSGE subsystem in GAMS, offering a concise and structured framework for model formulation; (2) to examine the time required to generate and solve the model in Julia and GAMS; and (3) to demonstrate the time-saving benefits of encapsulating the model within a Julia function. Using a fixed computer specification and model design, GTAP 9 data at various

aggregation levels are employed to measure runtime, including both model generation and solving times, and statistical tests are applied to assess differences in runtime.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the model used in this study. Section 3 describes how the model, originally written in GAMS/MPSGE, is implemented in Julia using the MPSGE.jl package. Section 4 compares model generation and solving times across different implementations. Section 5 concludes the paper and outlines directions for future research. The Julia and GAMS models developed for this study are available for download on GitHub.[1]

2. Model

The study adopts a static version of the Country-Specific Framework for Environmental Policy Analysis (CSAVE) model (Chen and Paltsev, 2025), which is motivated by the MIT EPPA model (Chen et al., 2022; Paltsev et al., 2005) and is developed based on GTAPinGAMS (Rutherford, 1997). CSAVE utilizes the build stream of GTAPinGAMS to produce the required regional and sectoral aggregation based on the GTAP9 database (Aguiar et al., 2016). The model structure can be summarized into three parts: 1) zero-profit conditions; 2) market-clearing conditions; and 3) income-balance conditions. For a producer, the economic activity is output, and for a household, the activity is utility. A zero-profit condition formulated as a Mixed Complementarity Problem (MCP) is:

$$MC - MR \geq 0; Q \geq 0 ; [MC - MR] \cdot Q = 0 \qquad (1)$$

In Condition (1), MC, MR, and Q are marginal cost, marginal revenue, and the index of relevant activity level. Other activities such as imports, exports, investment, and commodity aggregation have their own zero-profit conditions. For each market-clearing condition, the price level is

[1] The Julia model can be downloaded from https://github.com/chenyhmitedu/CSAVEinJulia. The entry point for running the model is located at ./CSAVEinJulia/main.jl. Note that the Pkg.add() lines at the top of main.jl must be uncommented and executed the first time the code is run to install the required packages. The GAMS model can be downloaded from https://github.com/chenyhmitedu/CSAVEinGAMS. Please refer to readme.txt for details on running the model.

determined based on demand and supply. The market-clearing condition in MCP format can be written as:

$$S \geq D; P \geq 0 \; ; [S - D \,] \cdot P = 0 \tag{2}$$

In Condition (2), S, D, and P are supply, demand, and price index, respectively. The income-balance condition specifies the income supporting the household expenditure and savings. For consistency, the condition is expressed in MCP format as:

$$E \geq I; E \geq 0 \; ; [E - I \,] \cdot E = 0 \tag{3}$$

In Condition (3), E represents expenditure plus savings, and so the condition states that any positive expenditure plus savings equals the income I. In the model, the price of utility is chosen as the numeraire of the model, and all other prices are measured relative to it.

The production technology (Figure 1) and consumer preference (Figure 2) are modeled by nested Constant Elasticity of Substitution (CES) function (Arrow et al., 1961). The output of each sector may be sold domestically or exported (Figure 1). The distribution of domestic sales and exports is modeled by the constant elasticity of transformation (CET) function (Powell and Gruen, 1968). Besides, the price index for aggregate consumption is used as the numeraire of the model, and all prices are measured relative to it.

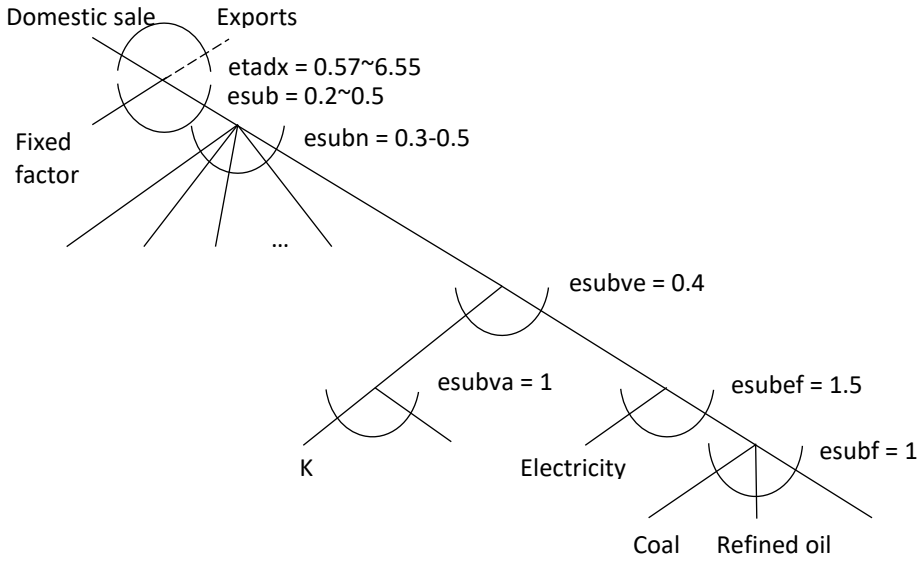

Figure 1. Cost function for a production sector

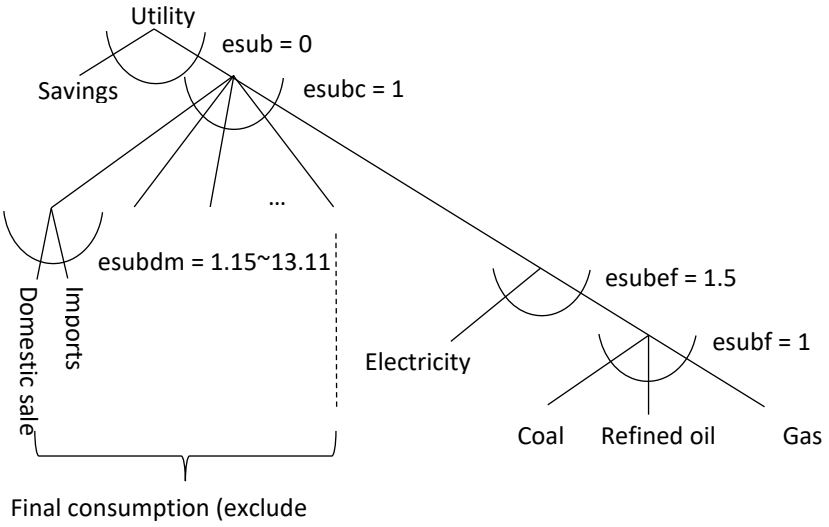

Figure 2. Expenditure function for the representative consumer

3. Julia Implementation

The study converts a static version of CSAVE written in GAMS/MPSGE into one formulated in Julia using the MPSGE.jl package. To explore how higher data resolutions may affect the required runtime, both versions of the model are adapted so they are multi-region and multi-sector CGE models sharing exactly the same settings and produce identical results.

To build the model in Julia, the package "CSAVEinJulia.jl" is created:

```
 1 module CSAVEinJulia
 2
 3   using CSV
 4   using JLD2
 5   using MPSGE
 6   using GTAPdata
 7
 8   include("load_data.jl")
 9   export load_gtap_data
10
11   include("mge.jl")
12   export MGE_model
13
14 end
```

Listing 1. An excerpt of CSAVEinJulia.jl

The package summarizes how input-output data are loaded (line 8 in Listing 1), and where the CGE model is located (line 11). The functions "load_gtap_data" and "MGE_model" defined in load_data.jl and mge.jl, respectively, are exported so that when "using CSAVEinJulia" is presented in a file, the two functions will be brought to the namespace of that file, i.e., they can be called directly.

Besides, the package defines its own environment, which is specified in Project.toml and Manifest.toml. The two files work together in defining a reproducible environment. They are generated automatically based on what packages are used. In particular, Project.toml specifies the names and identities of the direct dependencies (packages) used in CSAVEinJulia.jl. The code in Listing 2 summarizes the key components of Project.toml: "name" provides the project's name (line 1), "uuid" is the string with a universally unique identifier for CSAVEinJulia.jl (line 2), "version" offers the version number (line 3), and "authors" presents the author information (line 4). The section [deps] lists all dependencies of CSAVEinJulia.jl, with each dependency being a name-uuid pair (line 6 – 12). [sources] specifies the path to the repos of unregistered packages (line 14 – 18), and [compat] describes the compatibility constraints for the dependencies in [dep] (line 20 – 26):

```
 1 name = "CSAVEinJulia"
 2 uuid = "56894a01-51d8-47a5-a1b2-a4f35f317d29"
 3 version = "0.1.0"
 4 authors = ["Author <abc@def.edu>"]
 5
 6 [deps]
 7 JuMP = "4076af6c-e467-56ae-b986-b466b2749572"
 8 MPSGE = "d5dc2f44-7ae2-49e9-bc77-b47b6bca565d"
 9 PATHSolver = "f5f7c340-0bb3-5c69-969a-41884d311d1b"
10 CSVtoDIC = "cf815f9a-3c92-448a-b56b-77b95c7ee9f9"
11 GTAPdata = "130c15d4-ad42-4bd4-91d3-6787eb393e58"
12 …
13
14 [sources]
15 CSVtoDIC = {rev = "master", url =
16 "https://github.com/chenyhmitedu/CSVtoDIC"}
17 GTAPdata = {rev = "master", url =
18 "https://github.com/chenyhmitedu/GTAPdata"}
19
20 [compat]
21 CSVtoDIC = "0.1.0"
22 GTAPdata = "0.1.0"
23 JuMP = "1.29.3"
```

```
24 MPSGE = "0.6.4"
25 PATHSolver = "1.7.9"
26 …
```

Listing 2. An excerpt of Project.toml

Manifest.toml (Listing 3) constructs a comprehensive dependency graph (i.e., how a package depends on others) of CSAVEinJulia.jl, by providing both direct and indirect dependencies of each package used in the project. The file starts with three entries, which shows the Julia version the file was created on, the format of the manifest and a hash of the project file (Bezanson et al., 2017). Following the first three entries are details for the dependencies of CSAVEinJulia.jl. For instance, GTAPdata.jl is one of the dependencies presented, as shown in [[deps.GTAPdata]] below. As a package, GTAPdata.jl has four dependencies (i.e., it uses four packages): CSV, CSVtoDIC, DataFrames, and JLD2. "git-tree-sha1" pins the package to a specific Git commit (note that it is different from uuid and the Git commit hash)—it ensures an exact match of the package content, i.e., if any file in the package is revised, a new string for git-tree-sha1 will be automatically regenerated. "repo-rev" means the package is sourced from a branch in a Git repo, and "repo-url" provides the hyperlink of the repo.

```
 1 # This file is machine-generated - editing it directly is not advised
 2
 3 julia_version = "1.12.2"
 4 manifest_format = "2.0"
 5 project_hash = "6933b0a8f759f539c7d42917b68de419295debc6"
 6
 7 [[deps.JuMP]]
 8 deps = ["LinearAlgebra", "MacroTools", "MathOptInterface", "MutableArithmetics",
 9 "OrderedCollections", "PrecompileTools", "Printf", "SparseArrays"]
10 git-tree-sha1 = "b76f23c45d75e27e3e9cbd2ee68d8e39491052d0"
11 uuid = "4076af6c-e467-56ae-b986-b466b2749572"
12 version = "1.29.3"
13
14 [[deps.MPSGE]]
15 deps = ["DataFrames", "JuMP", "OrderedCollections", "PATHSolver"]
16 git-tree-sha1 = "c41433d82612252b090300f38af9259436394ae2"
17 uuid = "d5dc2f44-7ae2-49e9-bc77-b47b6bca565d"
18 version = "0.6.4"
19
20 [[deps.GTAPdata]]
21 deps = ["CSV", "CSVtoDIC", "DataFrames", "JLD2"]
22 git-tree-sha1 = "241154ac8dc3913f57ce3c54de22779fc4a22b00"
23 repo-rev = "master"
24 repo-url = "https://github.com/chenyhmitedu/GTAPdata"
25 uuid = "130c15d4-ad42-4bd4-91d3-6787eb393e58"
26 version = "0.1.0"
27 …
```

Listing 3. An excerpt of Manifest.toml

With information such as uuid (ensuring the uniqueness of a package's identity) and git-tree-sha1 (ensuring the exact content match), the package CSAVEinJulia.jl is version controlled by Github, which means that simulation results are reproducible for a given version of the package. Finally, for CSAVEinJulia.jl, the sequence of execution is controlled by the file main.jl, which is the entry point of running the model (see Listing 4):

```
using Pkg

Pkg.add("CSV")
Pkg.add("JLD2")
Pkg.add("MPSGE")
… …

Pkg.activate(".")               # Selects which environment you are working in
Pkg.instantiate()               # Installs the packages listed in that environment

using CSAVEinJulia
using DataFrames
… …

import PATHSolver
PATHSolver.c_api_License_SetString("1259252040&Courtesy&&&USR&GEN2035&5_1_2026
&1000&PATH&GEN&31_12_2035&0_0_0&6000&0_0")

data = load_gtap_data()

# Vectors below may be changed depending on the sectoral names and resolution
data["set_fe"]      = [:coa, :gas, :p_c]
data["set_elec"]    = [:ely]
data["set_ne"]      = setdiff(data["set_i"], union(data["set_fe"],
                       data["set_elec"]))
data["set_tr"]      = [:wtp, :atp, :otp]

# Define the Benchmark object
b = @benchmarkable MGE_model(data) samples = 1 evals = 1 seconds = 18000
model_gen = run(b)

raw_times_ns = model_gen.times
raw_times_seconds = raw_times_ns ./ 1_000_000_000
df_sol_time = DataFrame(:sec => raw_times_seconds)
path = joinpath(@__DIR__, "src/results/", "56x2_1_mg_time.csv")
CSV.write(path, df_sol_time)

MGE = MGE_model(data);

solve!(MGE, cumulative_iteration_limit = 0)

for I ∈ data["set_fe"], g ∈ data["set_g"]
  set_value!(MGE[:rtfd][I, g, :USA], data["rtfd0"][I, g, :USA]*2)
  set_value!(MGE[:rtfi][I, g, :USA], data["rtfi0"][I, g, :USA]*2)
```

```
60 end
61
62 sol = @benchmarkable solve!(MGE; cumulative_iteration_limit = 1000,
63 convergence_tolerance = 1e-8) samples = 1 evals = 1 seconds = 18000
64 model_sol = run(sol)
65 sol_times_seconds = model_sol.times ./ 1_000_000_000
66 df_sol_time = DataFrame(:sec => sol_times_seconds)
67
68 path = joinpath(@__DIR__, "src/results/", "56x2_sol_time.csv")
69 CSV.write(path, df_sol_time)
70
71 solve!(MGE, cumulative_iteration_limit = 1000)
72 df = generate_report(MGE)
73
74 df_filtered = df[df.margin .> 0.001, :]
75 println(df_filtered)
76
77 pf  = value.(MGE[:PF])  #Can just extract the one wanted variable this way
78 y   = value.(MGE[:Y])
79
 … …
85 df_y = DataFrame(Containers.rowtable(y))
86
87 output = "./src/results/output_y.csv"
88 CSV.write(output, df_y)
89
90 # Conducting bootstrap and producing figures: CGE wrapped in a function vs. CGE
91 not wrapped
 … …
95 include("./src/bootstrap.jl")
96 b = bstrap("56x2_1000_sol_time", "56x2_1000_sol_time_slow2")
97
98 include("./src/figure_histogram.jl")
99 hist_plot = fig_hist(b[1], b[2], "CGE wrapped in a function\n(sample size =
100 1000)",  "CGE not wrapped in a function\n(sample size = 1000)", 0.675, 2.1, 0.0,
101 0.22)
102 fig_hist_path = joinpath(@__DIR__, "./src/figures/", "wrapped_vs_not_wrapped.png")
103 savefig(hist_plot, fig_hist_path)
  … …
147
```

Listing 4. An excerpt of main.jl

In Listing 4, line 3 – 18 add packages used by CSAVEinJulia.jl, and they only need to be run once when constructing the environment for the first time. After that, they can be commented out, and let line 20 – 21 activate the environment, check required package versions and any missing packages based on Project.toml and Manifest.toml, and then install packages as needed. Packages are loaded by line 23 – 32. Line 34 – 41 get the GTAP data and group sectors by Julia's dictionaries. Line 44 – 45 create a benchmark object, run it, and then store results in model_gen. Line 47 – 51 get the model generation time and save it in a CSV file. Line 53 – 55 assign the object of the CGE model with the GTAP data to the variable MGE, and then solve the

model with zero iteration as a calibration check—the residual needs to be zero to ensure zero marginals for all MCP problems. Line 57 – 88 run the simulation where the US fossil fuel tax rates are doubled, and save the sectoral output results in a CSV file. Line 95 – 147 conduct nonparametric estimation for the median time differences and draw figures. The file structure of the model is summarized in Figure 1.

The CGE model formulated using the package MPSGE.jl is encapsulated in the function MGE_model() in mge.jl (line 11 in Listing 1). The structure of the function is similar to the model formulated in GAMS/MPSGE—it includes the declaration of sectors (line 22 – 31 in Listing 5), commodities (line 33 – 50), consumers (line 52 – 54), production blocks (line 56 – 114), and demand block (line 116 – 123). Production blocks are formulated by zero-profit conditions and determine relevant activity levels, while the demand block calibrates the income balance condition. Market clearing conditions are automatically handled in the background as in GAMS/MPSGE.

The model structure with different components (files and packages) are presented in Figure 1. The whole model and two packages (GTAPdata and CSVtoDIC) developed for this study are available on Github at https://github.com/chenyhmitedu.

```
 1 function MGE_model(data::Dict)
 2
 3    MGE  = MPSGEModel()
 4
 5    @parameters(MGE, begin
 6      rtfd[i=data["set_i"], g=data["set_g"], r=data["set_r"]], data["rtfd0"][i, g,
 7      r], (description = "Firms' domestic tax rates")
 …      …
18      rtf[f=data["set_f"], i=data["set_i"], r=data["set_r"]], data["rtf0"][f, i,
19      r], (description = "Primary factor tax rates")
20    end)
21
22    @sectors(MGE, begin
23      Y[g=data["set_g"], r=data["set_r"]], (description = "Supply")
 …      …
29      A[i=data["set_i"], g=data["set_g"], r=data["set_r"]], (description =
30      "Armington good")
31    end)
32
33    @commodities(MGE, begin
34      P[g=data["set_g"], r=data["set_r"]], (description = "Domestic output price")
 …      …
48      PE[i=data["set_i"], r=data["set_r"]], (description = "Price index for
49      exports (exclude subsidy and transport service)")
50    end)
```

```
51
52    @consumers(MGE, begin
53      RA[r=data["set_r"]], (description = "Representative agent")
54    end)
55
56    @production(MGE, A[i=data["set_i"], g=data["set_g"], r=data["set_r"]], [t = 0, s
57      = data["esubd"][i]], begin
58      @output(PA[i, g, r], data["vafm"][i, g, r], t)
59      @input(P[i, r], data["vdfm"][i, g, r],  s, taxes = [Tax(RA[r], rtfd[i, g,
60      r])],   reference_price = 1 + data["rtfd0"][i, g, r])
61      @input(PM[i, r],        data["vifm"][i, g, r],  s,   taxes = [Tax(RA[r],
62      rtfi[i, g, r])],   reference_price = 1 + data["rtfi0"][i, g, r])
63    end)
64
 …    …
115
116   @demand(MGE, RA[r=data["set_r"]], begin
117     @final_demand(P[:c, r],     data["vom"][:c, r])
118     @endowment(P[:c, :USA],     data["vb"][r])
119     @endowment(P[:g, r],        -data["vom"][:g, r])
120     @endowment(P[:i, r],        -data["vom"][:i, r])
121     @endowment(PF[f=data["set_mf"], r],       data["evom"][f, r])
122     @endowment(PS[f=data["set_sf"], j=data["set_i"], r],    data["vfm"][f, j, r])
123   end)
124
125   fix(P[:c, :USA], 1)
126
127   return MGE
128
129 end
```

Listing 5. An excerpt of mge.jl

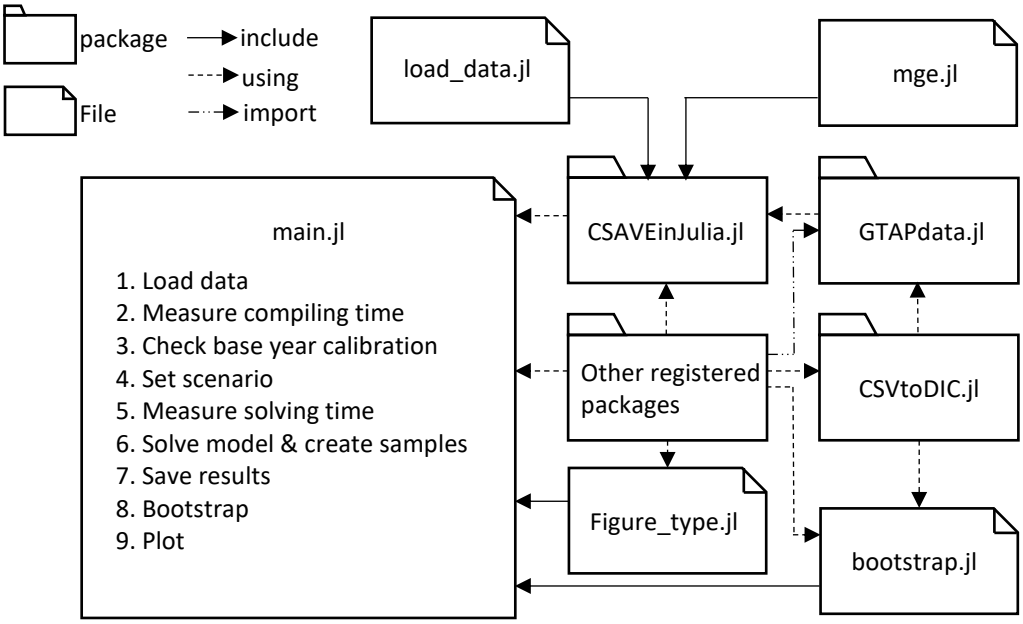

Figure 1. Model structure

4. Comparison

Before comparing the two versions of CSAVE, i.e., for convenience, let us denote the Julia version of the model by CSAVEinJulia, which refers to the complete model rather than just the package CSAVEinJulia.jl, and denote the GAMS version of the model by CSAVEinGAMS. The same counterfactual scenario is imposed on each model as a test to verify that both versions of the model produce identical outputs. The sample scenario considered is to double the base year fossil fuel tax rates in the U.S. The sectoral output indices of CSAVEinJulia match perfectly with those of CSAVEinGAMS as expected (Figure 2). With higher fossil fuel tax rates, outputs of petroleum products (p_c) and gas are reduced, and so do those for sectors that rely heavily on them as inputs, such as air transport (atp), land transport (otp), and water transport (wtp). The output of coal increases since in the base year, unlike cases for petroleum products and gas, tax rates on coal usage among sectors are essentially zero, which means that doubling them will not change the "tax-free" status of coal usage.

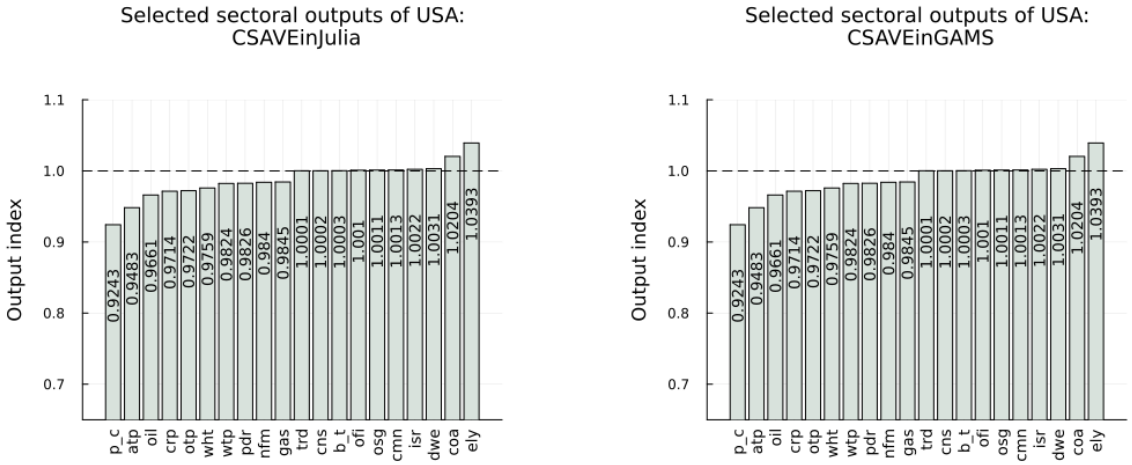

Figure 2. Selected sectoral outputs of USA

4.1 Compiling and run-time phases

A key aspect differentiating Julia and GAMS is in how the source code is handled. In Julia, it is parsed into a variety of abstract syntax trees (AST), and then subsequently converted to distinct forms of representation (IR). Each AST and IR corresponds to a particular stage of transforming the source code into machine code. Specifically, the "typed IR" in Julia is the static single assignment (SSA) form of IR produced after type inference, in which each SSA value is annotated with an inferred type (The Julia Project, 2026a)— after typed IR is generated, Julia determines which parts of the code can be fully type-specialized and compiled into optimized machine code, and which parts must retain dynamic dispatch due to incomplete type information, reducing the scope of optimization (The Julia Project, 2026b).

For example, code that can be efficiently optimized typically arises from function calls with concrete argument types; when such types are known, Julia resolves multiple dispatch at compilation time (i.e., static dispatch) for the corresponding specialization, enabling fully type-specialized and optimized machine code. Intuitively, the code is fixed and stable in this case, and no settings for retaining flexibility at the expense of slowing down execution are needed. Top-level Julia code, such as operations involving global variables, remains dynamically typed and is executed without full type specialization, limiting the compiler's ability to apply optimizations compared to code inside functions (The Julia Project, 2026a; 2026c). During run time, Julia's just-in-time (JIT) compilation is triggered when a new combination of function and argument types is encountered, producing optimized machine code before execution. The PATH solver (Ferris and Munson, 1999), which is used in solving MCP problems, then interacts with the model through Julia callback functions, which are exposed to PATH via C-compatible function pointers, enabling the solver to invoke compiled Julia code during execution (Dowson et al., 2025; The Julia Project, 2026d).

In the context of CGE modeling using MPSGE.jl, the model is specified using Julia source code, largely through macros that define the economic structure of the system. Based on this specification, MPSGE.jl formulates Conditions (1)–(3) in Section 2—by constructing a mixed complementarity problem (MCP) using JuMP.jl (the package "Julia for Mathematical Programming") (Lubin et al., 2023) and related dependencies. The residual and Jacobian evaluation functions associated with the MCP are then JIT-compiled by Julia to native machine code. Through a C-compatible function pointers, these compiled functions are exposed to the

precompiled PATH solver written in C and wrapped by PATHSolver.jl (Dowson et al., 2025). Specifically, PATH receives: 1) the number of variables; 2) the number of equations; 3) an initial guess for the endogenous variables; 4) lower and upper bounds; 5) a callback to the compiled residual function; and 6) a callback to the compiled Jacobian function. PATH iteratively calls these callbacks during the solution process until convergence tolerance is met (The Julia Project, 2026e; Julia Packages, 2026; Dirkse et al., 2026).

In GAMS, model scripts are compiled into a solver-readable problem representation rather than machine code. This representation is passed to an external solver, which is precompiled to native code (GAMS Development Corp., 2026a; 2026b; 2026c). Existing literature does not provide details about if or how solver works with some sort of "GAMS runtime evaluator" in solving the problem. Nevertheless, the GAMS documentation suggests that the model is compiled into an instance, which includes representations for residual and Jacobian functions. The solver then performs an iterative algorithm using this model instance to obtain residual and Jacobian values until the default or user specified convergence criteria are met (GAMS Development Corp., 2026d).

There is one more step for GAMS/MPSGE application: the MPSGE script embedded in GAMS code is parsed by the MPSGE preprocessor, which produces "model_name.gen" file along with variable declarations of the model. The file is included back to the GAMS program file (Rutherford, 1999), which instructs GAMS language compiler to build an Algebraic representation of the problem, before sending the model instance to PATH. The overview of Julia's and GAMS' compile and run time processes can be summarized graphically (Figure 3).

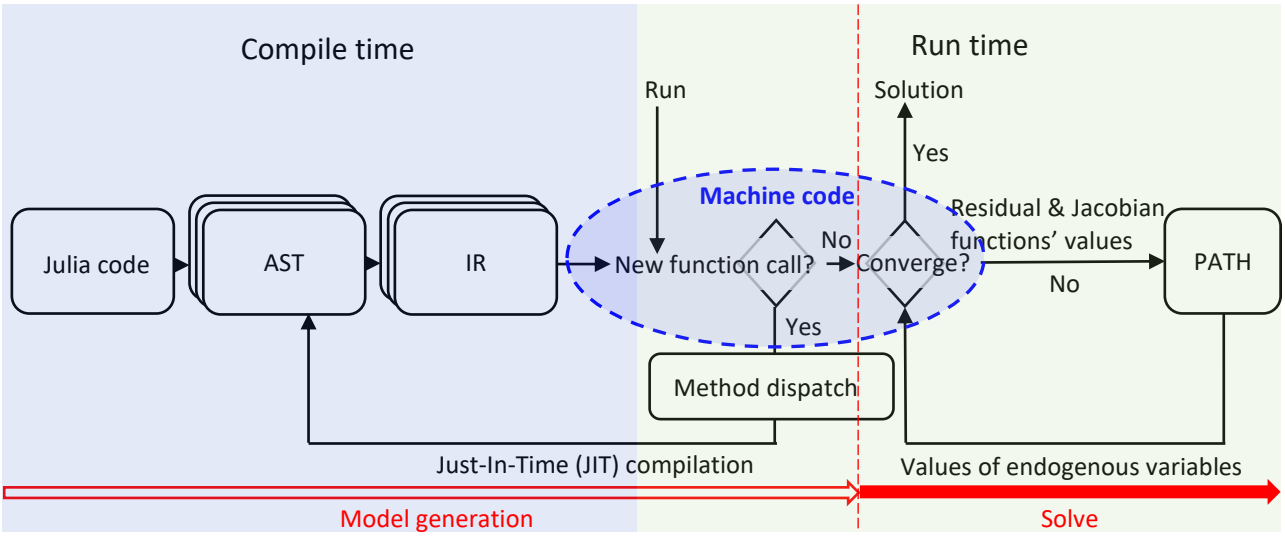

Figure 3(a). Flowchart for compiling and running CSAVEinJulia

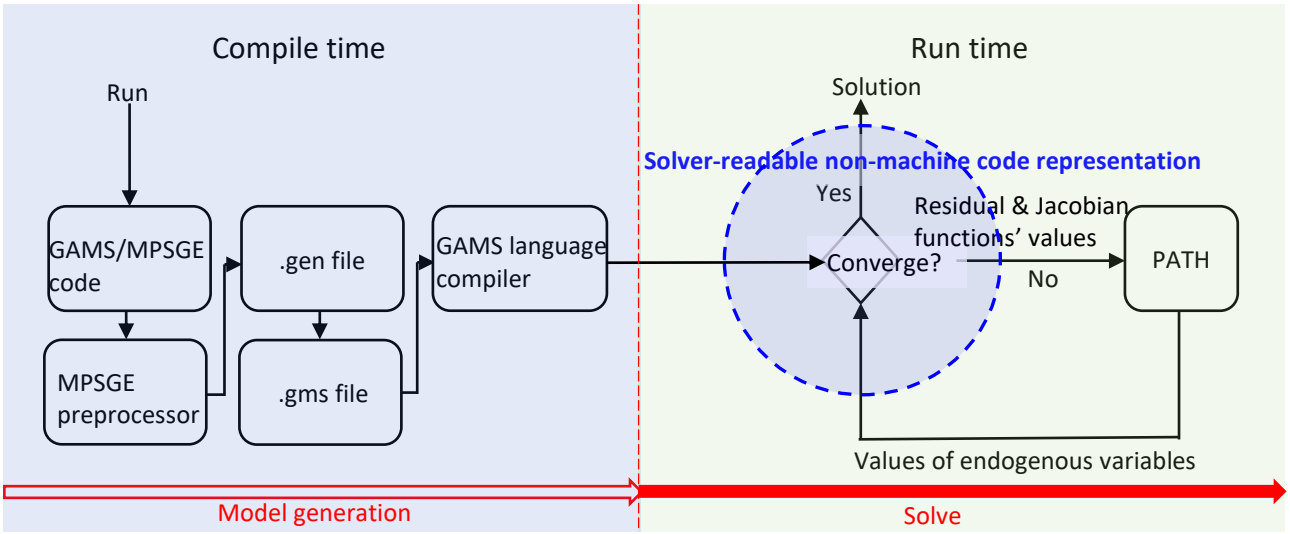

Figure 3(b). Flowchart for compiling and running CSAVEinGAMS

4.2 Model generation and solving times

The study proceeds to compare CSAVEinJulia with CSAVEinGAMS in terms of the model generation time, which is defined as the time needed when initiating the compiling process before solving the problem. For CSAVEinJulia, it includes the compilation time and also part of run time, when the machine code is optimized for function calls with concrete input types (Figure 3(a)), while for CSAVEinGAMS, the model generation time starts from language

compilation up to the point when the GAMS compiler finishes the solver-readable non-machine code representation (Figure 3(b)).

To explore how different data resolution affects the generation time, the study aggregates the GTAP9 dataset into: 1) 56 sectors and 2 regions: USA and the rest of the world—henceforth "56x2", 2) 56x4, 3) 56x8, and 4) 56x16. Details of the sectoral and regional mappings are provided in the Appendix. It is worth noting that even with the same number of sectors or regions, how sectors and regions are aggregated will have implications on data structure and therefore may affect the time needed in generating and solving the model. Besides, the time required also depends on the technical specifications of the computer. The modeling exercise is based on the PC with CPU and RAM specifications as follows: AMD Ryzen Threadripper PRO 7955WX 16-Cores 4.5 GHz (CPU), and 128 GB 4800 MT/s (RAM). The one-shot run suggests CSAVEinGAMS has shorter model generation time than CSAVEinJulia, and as the data resolution gets higher, much longer generation time is needed (Figure 4).

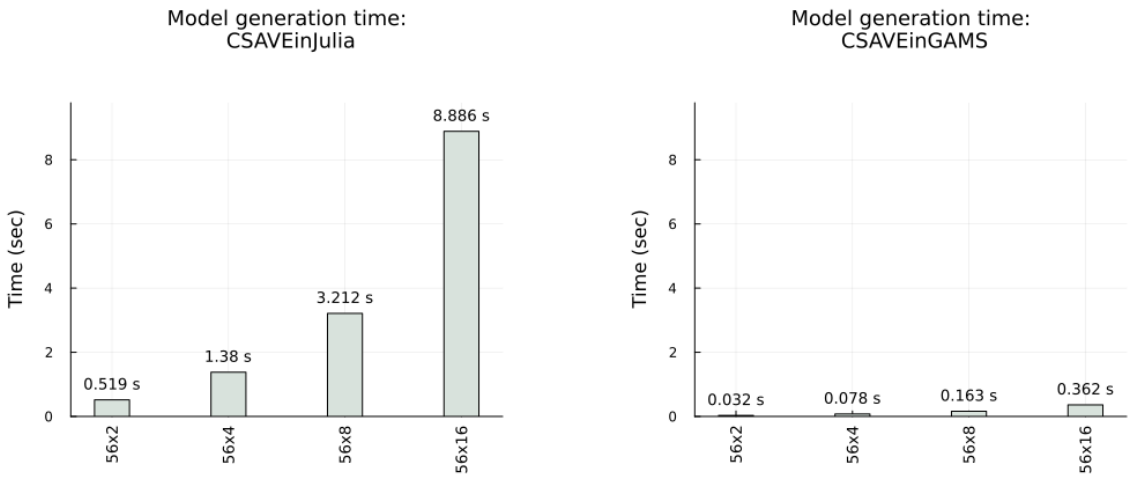

Figure 4. Model generation time comparison

The story is quite different in terms of solving time, which is defined as the time required between when the solve statement is executed and when the solution is found. CSAVEinJulia is much faster in solving the problem (Figure 5), and when both model generation and solving time

are considered, CSAVEinJulia is the speed winner. For example, to run the sample scenario under the 56x16 setting, it takes CSAVEinJulia just over one minute to find the solution, while CSAVEinGAMS needs almost 40 minutes to complete the same task (Figure 6).

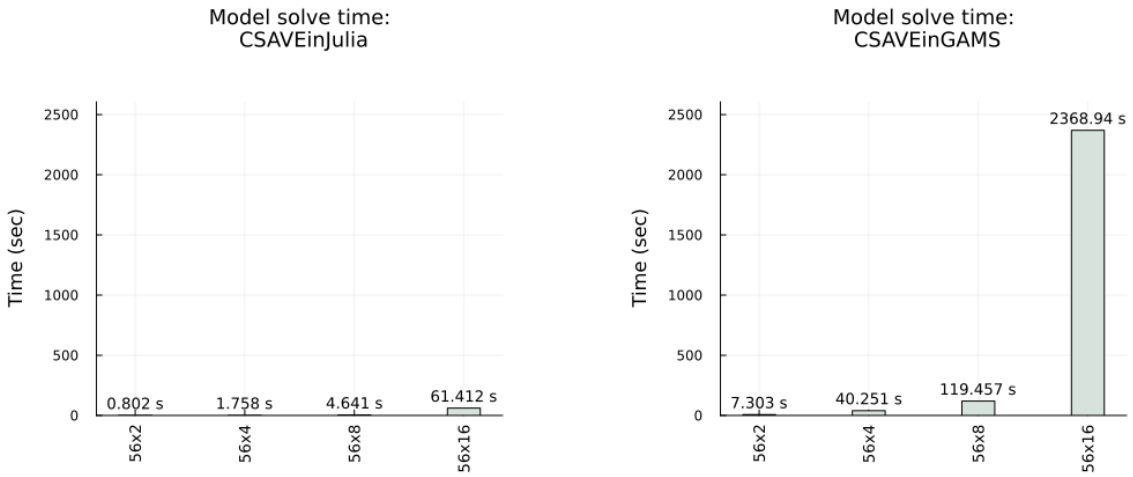

Figure 5. Model solving time comparison

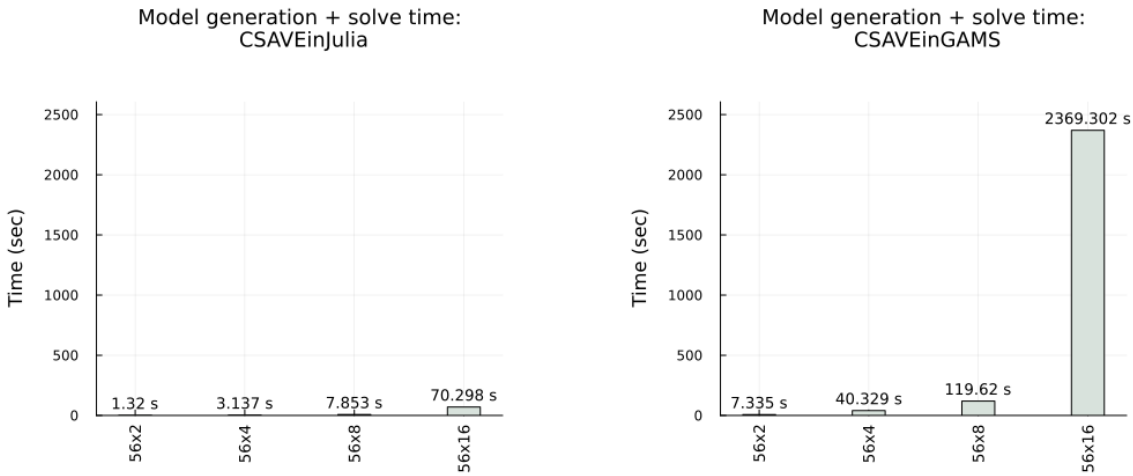

Figure 6. Model generation and solving time comparison

While the one-shot modeling exercise (Figure 4 to Figure 6) provides some hint about the two models' speed performances, the model generation time and solving time are indeed random variables, which means that each realization of the required time may be different from the previous run. Further, these variables are right skewed, as noted in the manual for Julia's package BenchmarkTools.jl (Revels, 2025):

> *"Time distributions are always right-skewed… This phenomena can be justified by considering that the machine noise affecting the benchmarking process is, in some sense, inherently positive…"*

To nullify the effect of machine noise, Chen and Revels (2016) and Revels (2025) propose using the minimum as the estimator for the location parameter of a time distribution. On the other hand, if the goal is being more "realistic" about the required time without excluding the noise effect, as it is unavoidable, one may use the median as the estimator for the central tendency of time, as the median is unaffected by outliers. The study is interested in testing if the time differences are statistically significant, and it will report results based on both estimators.

Let us denote the user time for running model i as t_i (t_i can be the model generation time or solving time, depending on the context). To test if model i runs faster than model j, the hypothesis test is:

$$H_0: t_i - t_j \leq 0$$

$$H_a: t_i - t_j > 0$$

We take the "56x2" (56-sector and 2-region) dataset as an example to conduct the test. Each of the two models (CSAVEinJulia and CSAVEinGAMS) is executed 1000 runs separately and the model generation time and solving time of each run are recorded for the individual model. Next, a bootstrap procedure (Efron, 1979) is used to construct the confidence interval for the time difference from the observations without the need of imposing the normality assumption. Specifically, the 1000 observations are resampled with replacement 1000 times to generate a bootstrap sample with 1000 observations. The process repeats until 10000 bootstrap samples are created (Efron and Tibshirani, 1994) and the statistic (minimum or median) for each sample is computed, yielding 10000 bootstrap replicates. The replicates are sorted in ascending order to derive the empirical quantiles used to construct the confidence interval (Listing 6).

```
# 3. Bootstrap Loop
 for i in 1:B
   # Resample with replacement
   X1_star = sample(X1, n1, replace=true)
   X2_star = sample(X2, n2, replace=true)

   # Calculate the statistic (difference in medians)
   #delta_star = median(X1_star) - median(X2_star)
   delta_star = minimum(X1_star) - minimum(X2_star)

   # Append "delta_star" to the end of an array ("bootstrap_differences") one
   # element at a time.
   push!(bootstrap_differences, delta_star)
 end

# 4. Construct the Confidence Interval (99% CI)
  CI_lower = quantile(bootstrap_differences, 0.01)
  CI_upper = Inf
```

Listing 6. An excerpt for bootstrap.jl

The tests confirm the findings presented in Figures 4, 5, and 6, and regardless of whether the noise effect is taken into account, overall, CSAVEinJulia still runs much faster than CSAVEinGAMS under 1% of significance level, even though it has a somewhat longer model generation time (Table 1).

Table 1. Hypothesis tests for the time difference between CSAVEinJulia and CSAVEinGAMS

Type of user time	Type of estimator t	99% C. I. for $t^j - t^g$	H_0	Reject H_0?
Model generation	min	$[0.469, \infty)$	H_0: $t^j \leq t^g$	Yes, favor H_a: $t^j > t^g$
	med	$[0.589, \infty)$	↑	↑
Solving	min	$(-\infty, -6.181]$	H_0: $t^j \geq t^g$	Yes, favor H_a: $t^j < t^g$
	med	$(-\infty, -6.365]$	↑	↑
Total	min	$(-\infty, -5.675]$	H_0: $t^j \geq t^g$	Yes, favor H_a: $t^j < t^g$
	med	$(-\infty, -5.755]$	↑	↑

* j = CSAVEinJulia; g = CSAVEinGAMS; min = minimum; med = median

4.3 Model with or without function encapsulation

In contrast to compiling lines of code with a global scope, compiling function calls is more efficient in Julia, as in the latter case concrete types become available (Section 4.1). Therefore, a coding strategy is to encapsulate the model within a function. In that case, everything inside the function is local, and to run the model will be to call the function that wraps the model, using the required data and settings as inputs.

In the following example, the study will provide a comparison of solving time between the setting where the CGE component of CSAVEinJulia is wrapped in a function and that where the model is coded at a global scope, based on the 56x2 dataset with the same policy shock (i.e., double the fossil fuel tax rates). Both settings are run 1000 times, and the solving time distribution along with some basic statistics suggest that while both settings share quite close solving time, still, wrapping the model in a function may make the solving process slightly faster, as suggested by statistics such as minimum and median (Figure 6).

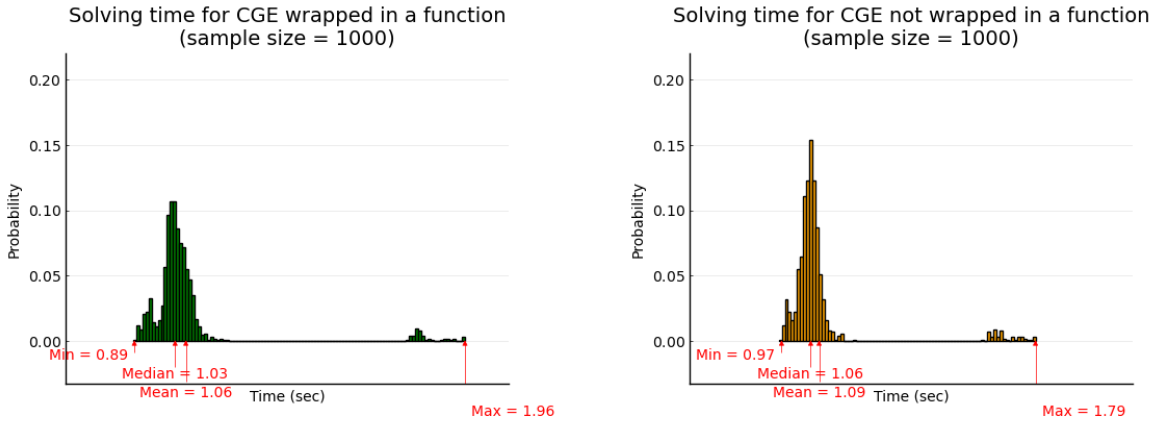

Figure 6. Solving time under different coding strategies

To test if wrapping CGE in a function really saves the solving time, following a similar bootstrapping strategy as before, a total of 10000 bootstrap replicates are created, with each replicate containing 1000 observations and generating its own statistics (minimum and median). Statistics are then ranked in ascending order to derive empirical quantiles, and confidence intervals are then constructed accordingly. The results confirm that encapsulating the model

within a function does speed up the solving process under a 1% significance level (Table 2). It is worth noting that model encapsulation has other advantages beyond time saving benefit explored here. For instance, keeping the model's scope local to the function can reduce the chance of having "namespace pollution" (The Julia Project, 2026f) and therefore makes the code more robust, especially with the presence of other components in a large-scale model.

Table 2. Time difference between CGE wrapped in a function and CGE at a global scope

Type of user time	Type of estimator t	99% C. I. for $t^j - t^g$	H_0	Reject H_0?
Solving	min	$(-\infty, -0.066]$	H_0: $t^j \geq t^g$	Yes, favor H_a: $t^j < t^g$
	med	$(-\infty, -0.032]$	↑	↑

* j = CSAVEinJulia; g = CSAVEinGAMS; min = minimum; med = median

5. Conclusions

Applied general equilibrium studies tend to pursue a higher data resolution over time by incorporating more sectors, commodities, households, factors, and regions, thereby substantially increasing model size (Horridge et al., 2013). Although advances in computational power have alleviated issues associated with the curse of dimensionality, the role of software infrastructure in accelerating the solution process of CGE models has received relatively little attention.

Although GAMS/MPSGE and GEMPACK have long dominated CGE modeling, recent developments suggest that modern open-source languages such as Julia can effectively implement large-scale models. By comparing runtimes, this study provides insights into the computational performance advantages of formulating CGE models in Julia using the MPSGE.jl package. As the time-saving performance gain becomes more pronounced at higher data resolutions, Julia represents a promising platform for large-scale CGE modeling.

With this in mind, GAMS/MPSGE will likely remain an important tool in CGE modeling exercises, particularly when model dimensionality is moderate (e.g., a single-region model with one or two dozen sectors and several primary factors, or a multi-region model with around ten or fewer regions, a dozen sectors, and a few primary factors). For instance, GAMS provides concise and expressive syntax for set operations, simplifying model formulation and allowing modelers to concentrate on the economic structure. This may result in a gentler learning curve for

performing CGE modeling. From an efficiency perspective, it is when higher data resolutions are adopted, Julia implementations begin to shine.

Future research could extend similar comparisons to alternative coding strategies or programming languages. For instance, a Julia implementation can be carried out without MPSGE.jl, as demonstrated by Ivanic (2025), albeit with substantially greater coding complexity. In this case, the performance of different Julia model formulations using alternative solvers could be assessed. More broadly, models implemented in other programming languages and solved with various solvers could be evaluated. Finally, anecdotal evidence suggests that solvers may differ in their ability to find solutions. Beyond time efficiency, the robustness of a model-solver combination is therefore another key consideration that deserves further investigation.

Acknowledgement

I would like to thank Mitch Phillipson for sharing his expertise on CGE modeling in Julia, Angelo Gurgel for providing Julia course materials, the Center for Sustainability Science and Strategy (CS3) for financial support, and participants in the MIT EPPA meeting and the CS3 Friday lunch meeting for their helpful comments. The views expressed are my own, and any remaining errors are solely my responsibility.

References

Aguiar, A., B. Narayanan, and R. McDougall (2016). An Overview of the GTAP 9 Data Base. Journal of Global Economic Analysis 1(1): 181-208. https://jgea.org/ojs/index.php/jgea/article/view/23

Anthoff, D., E. Lazarus, and M. Phillipson (2025). MPSGE for Julia. Julia-MPSGE. https://github.com/julia-mpsge/MPSGE.jl

Arrow, K. J., H. B. Chenery, B. S. Minhas and R. M. Solow (1961). Capital-Labor Substitution and Economic Efficiency. *The Review of Economics and Statistics*, Aug., 1961, Vol. 43, No. 3, 225–250.

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). Julia: A fresh approach to numerical computing. SIAM Review, Vol. 59, No. 1, 65–98. https://epubs.siam.org/doi/10.1137/141000671

Chen, J., and J. Revels (2016). Robust benchmarking in noisy environments. *arXiv e-prints*. 1608.04295. https://ui.adsabs.harvard.edu/abs/2016arXiv160804295C

Chen, Y.-H. Henry, S. Paltsev, A. Gurgel, J. Reilly and J. Morris (2022). A multisectoral dynamic model for energy, economic and climate scenario analysis. *Low Carbon Economy*, 13(2) (doi: 10.4236/lce.2022.132005)

Dowson, O., C. Kwon, R. Deits, E. Saba, J. Perla, M. Ferris, D. Anthoff, L. Peters, J. Revels, A. Tiago, and M. Phillipson (2025). PATHSolver.jl. https://github.com/chkwon/PATHSolver.jl

Dirkse, S. M. Ferris and T. Munson (2026). The Path Solver. https://pages.cs.wisc.edu/~ferris/path.html

Efron, B. (1979). Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1), 1–26. https://www.jstor.org/stable/2958830

Efron, B. and R. J. Tibshirani (1994). *An Introduction to the Bootstrap.* Chapman and Hall/CRC. New York. https://doi.org/10.1201/9780429246593

Ferris, M., and T. Munson (1999). Interfaces to PATH 3.0: Design, Implementation

and Usage. *Computational Optimization and Applications* 12, 207–227.

GAMS Development Corporation (2025). General Algebraic Modeling System (GAMS) Version 51.4.0 [Software]. https://www.gams.com/.

GAMS Development Corporation (2026a). System Overview. https://www.gams.com/products/gams/gams-language/

GAMS Development Corporation (2026b). Integrated Solvers. https://www.gams.com/products/solvers/

GAMS Development Corporation (2026c). Model and Solve Statements. https://www.gams.com/49/docs/UG_ModelSolve.html

GAMS Development Corporation (2026d). Gather-Update-Solve-Scatter (GUSS). https://www.gams.com/50/docs/S_GUSS.html

Horridge, M., A. Meeraus, K. Pearson, and T. F. Rutherford (2013). Solution Software for Computable General Equilibrium Modeling. In P. B. Dixon and D. W. Jorgenson (Eds.), *Handbook of Computable General Equilibrium Modeling*, Vol. 1B, pp. 1331–1381. North-Holland (Elsevier).

Ivanic, M. (2025). The GTAP v7 model in Julia. *Journal of Global Economic Analysis*, 10(1), 175-217. https://doi.org/10.21642/JGEA.100104AF

Johansen, L. (1960). *A Multi-Sectoral Study of Economic Growth.* North-Holland Publishing Company, Amsterdam.

Julia Packages (2026). PATHSolver.jl. (PATHSolver.jl is a wrapper for the PATH solver.) https://juliapackages.com/p/pathsolver

The Julia Project (2026a). JIT Design and Implementation. https://docs.julialang.org/en/v1/devdocs/jit/

The Julia Project (2026b). Julia SSA-form IR. https://docs.julialang.org/en/v1/devdocs/ssair/

The Julia Project (2026c). Julia Execution. https://docs.julialang.org/en/v1/devdocs/eval/#Julia-Execution

The Julia Project (2026d). Creating C-Compatible Julia Function Pointers. https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/#Creating-C-Compatible-Julia-Function-Pointers

The Julia Project (2026e). Calling C and Fortran Code. https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/

The Julia Project (2026f). Modules. https://docs.julialang.org/en/v1/manual/modules/

Lubin, M., O. Dowson, J. D. Garcia, J. Huchette, B. Legat, J. P. Vielma (2023). JuMP 1.0: recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation* 15, 581–589. https://doi.org/10.1007/s12532-023-00239-3

Lubin, M. and I. Dunning (2015). Computing in Operations Research Using Julia. *INFORMS Journal on Computing* 27(2), 238–248. https://doi.org/10.1287/ijoc.2014.0623

Paltsev S., J. Reilly, H. Jacoby, R. Eckaus and J. McFarland and M. Babiker (2005). The MIT Emissions Prediction and Policy Analysis (EPPA) Model: Version 4. MIT JPSPGC Report 125. Cambridge, MA. http://globalchange.mit.edu/files/document/MITJPSPGC_Rpt125.pdf

Pearson, K. R. and B. R. Powell (2014). *Computing Solutions for Large General Equilibrium Models Using GEMPACK*. Amsterdam: North-Holland.

Powell, A. A. and F. H. G. Gruen (1968). The Constant Elasticity of Transformation Production Frontier and Linear Supply System. *International Economic Review*, Oct., 1968, Vol. 9, No. 3, 315–328.

Revels, J. (2025). Manual. BenchmarkTools. https://juliaci.github.io/BenchmarkTools.jl/dev/manual/

Rutherford, T. (1999). Applied General Equilibrium Modeling with MPSGE as a GAMS Subsystem: An Overview of the Modeling Framework and Syntax. Computational Economics 14: 1–46.

Rutherford, T. (1997). GTAPinGAMS: The Dataset and Static Model. GTAP Conference paper No. 409. https://www.gtap.agecon.purdue.edu/resources/res_display.asp?RecordID=409

Appendix

A1. main.jl

```
using Pkg

Pkg.add("CSV")
Pkg.add("JLD2")
Pkg.add("MPSGE")
Pkg.add("JuMP")
Pkg.add(url="https://github.com/chenyhmitedu/CSVtoDIC")
Pkg.add(url="https://github.com/chenyhmitedu/GTAPdata")
Pkg.add("DataFrames")
Pkg.add("StatsBase")
Pkg.add("StatsPlots")
Pkg.add("Plots")
Pkg.add("PyPlot")
Pkg.add("Measures")
Pkg.add("Distributions")
Pkg.add("BenchmarkTools")
Pkg.add("PATHSolver")


Pkg.activate(".")               # Selects which environment you are working in
Pkg.instantiate()               # Installs the packages listed in that environment

using CSAVEinJulia
using DataFrames
using MPSGE
using CSV
using BenchmarkTools
using JuMP

import PATHSolver
PATHSolver.c_api_License_SetString("1259252040&Courtesy&&&USR&GEN2035&5_1_2026
&1000&PATH&GEN&31_12_2035&0_0_0&6000&0_0")

data = load_gtap_data()

# Vectors below may be changed depending on the sectoral names and resolution
data["set_fe"]      = [:coa, :gas, :p_c]
data["set_elec"]    = [:ely]
data["set_ne"]      = setdiff(data["set_i"], union(data["set_fe"],
data["set_elec"]))
data["set_tr"]      = [:wtp, :atp, :otp]

# Define the Benchmark object
b = @benchmarkable MGE_model(data) samples = 1 evals = 1 seconds = 18000
model_gen = run(b)

raw_times_ns = model_gen.times
raw_times_seconds = raw_times_ns ./ 1_000_000_000
df_sol_time = DataFrame(:sec => raw_times_seconds)
path = joinpath(@__DIR__, "src/results/", "56x2_1_mg_time.csv")
CSV.write(path, df_sol_time)

MGE = MGE_model(data);

solve!(MGE, cumulative_iteration_limit = 0)

for I ∈ data["set_fe"], g ∈ data["set_g"]
```

```
  set_value!(MGE[:rtfd][I, g, :USA], data["rtfd0"][I, g, :USA]*2)
  set_value!(MGE[:rtfi][I, g, :USA], data["rtfi0"][I, g, :USA]*2)
end

sol = @benchmarkable solve!(MGE; cumulative_iteration_limit = 1000,
convergence_tolerance = 1e-8) samples = 1 evals = 1 seconds = 18000
model_sol = run(sol)
sol_times_seconds = model_sol.times ./ 1_000_000_000
df_sol_time = DataFrame(:sec => sol_times_seconds)

path = joinpath(@__DIR__, "src/results/", "56x2_sol_time.csv")
CSV.write(path, df_sol_time)

solve!(MGE, cumulative_iteration_limit = 1000)
df = generate_report(MGE)

df_filtered = df[df.margin .> 0.001, :]
println(df_filtered)

pf  = value.(MGE[:PF])  #Can just extract the one wanted variable this way
y   = value.(MGE[:Y])

# Why Containers.rowtable is convenient: If one uses the built-in JuMP utility,
there's no to worry about
# the internal structure of the keys at all. It handles the "unpacking" logic for
you automatically:

df_y = DataFrame(Containers.rowtable(y))

output = "./src/results/output_y.csv"
CSV.write(output, df_y)

# Conducting bootstrap and producing figures: CGE wrapped in a function vs. CGE
not wrapped
# "CGE wrapped in a function\n(sample size = 1000)"
# "CGE not wrapped in a function\n(sample size = 1000)"

include("./src/bootstrap.jl")
b = bstrap("56x2_1000_sol_time", "56x2_1000_sol_time_slow2")

include("./src/figure_histogram.jl")
hist_plot = fig_hist(b[1], b[2], "CGE wrapped in a function\n(sample size =
1000)",  "CGE not wrapped in a function\n(sample size = 1000)", 0.675, 2.1, 0.0,
0.22)
fig_hist_path = joinpath(@__DIR__, "./src/figures/", "wrapped_vs_not_wrapped.png")
savefig(hist_plot, fig_hist_path)

include("./src/figure_CI.jl")
ci_plot = fig_ci(b[3], b[4], "95% Confidence Interval for Median Runtime
Difference", -0.04, 0.0)
fig_ci_path = joinpath(@__DIR__, "./src/figures/", "median_ci_dotplot.png")
savefig(ci_plot, fig_ci_path)

# Conducting bootstrap and producing figures: Model generation time - Julia vs.
GAMS

include("./src/bootstrap.jl")
b = bstrap("56x2_1000_mg_time", "56x2_1000_GAMS_mg_time")

include("./src/figure_histogram.jl")
hist_plot = fig_hist(b[1], b[2], "Model generation time with 56x2 setting: Julia",
"Model generation time with 56x2 setting: GAMS",0.0, 1.4, 0.0, 0.4,"","Time
(sec)")
```

```
fig_hist_path = joinpath(@__DIR__, "./src/figures/", "mg_julia_gams.png")
savefig(hist_plot, fig_hist_path)

include("./src/figure_CI.jl")
ci_plot = fig_ci(b[3], b[4], "95% Confidence Interval for Median Model Generation
Time Difference:\n Julia time minus GAMS time", 0.56, 0.6)
fig_ci_path = joinpath(@__DIR__, "./src/figures/",
"mg_julia_gams_median_ci_dotplot.png")
savefig(ci_plot, fig_ci_path)

# Conducting bootstrap and producing figures: Solve time - Julia vs. GAMS

include("./src/bootstrap.jl")
b = bstrap("56x2_1000_sol_time", "56x2_1000_GAMS_sol_time")

include("./src/figure_histogram.jl")
hist_plot = fig_hist(b[1], b[2], "Solve time with 56x2 setting: Julia", "Solve
time with 56x2 setting: GAMS",0.0, 9.0, 0.0, 0.4,"","Time (sec)")
fig_hist_path = joinpath(@__DIR__, "./src/figures/", "sol_julia_gams.png")
savefig(hist_plot, fig_hist_path)

include("./src/figure_CI.jl")
ci_plot = fig_ci(b[3], b[4], "95% Confidence Interval for Median Solve Time
Difference:\n Julia time minus GAMS time", -6.45, -6.35)
fig_ci_path = joinpath(@__DIR__, "./src/figures/",
"sol_julia_gams_median_ci_dotplot.png")
savefig(ci_plot, fig_ci_path)
```