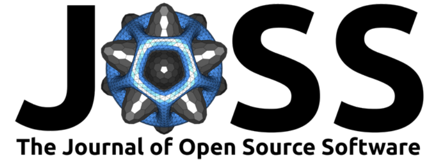


QuantizedSystemSolver: A discontinuous ODE system solver in Julia.

Elmongi Elbellili[1,2], **Daan Huybrechs**[2], **and Ben Lauwens**[1]

1 Royal Military Academy, Brussels, Belgium **2** KU Leuven, Leuven, Belgium

DOI: 10.21105/joss.07434

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @joshuaeh
- @IamBOOO

Submitted: 28 August 2024
Published: 23 January 2025

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

Contemporary engineering systems, such as electrical circuits, mechanical systems with shocks, and chemical reactions with rapid kinetics, are often characterized by dynamics that can be modeled using stiff differential equations with events. Stiffness typically arises in these systems due to the presence of both rapidly changing and slowly changing components. This stiffness requires extremely small time steps to maintain stability when using traditional numerical integration techniques. Recently, quantization-based techniques have emerged as an effective alternative for handling such complex models. Methods like the Quantized State System (QSS) and the Linearly Implicit Quantized State System (LIQSS) offer promising results, particularly for large sparse stiff models. Unlike classic numerical integration methods, which update all system variables at each time step, the quantized approach updates individual system variables independently. Specifically, in quantized methods, each variable is updated only when its value changes by a predefined quantization level. Moreover, these methods are advantageous when dealing with discontinuous events. An event is a discontinuity where the state of the system abruptly changes at a specific point. Classic methods may struggle with events: They either undergo expensive iterations to pinpoint the exact discontinuity instance or resort to interpolating its location, resulting in unreliable outcomes. Therefore, this QSS strategy can significantly reduce computational effort and improve efficiency in large sparse stiff models with frequent discontinuities (Pietro et al., 2019).

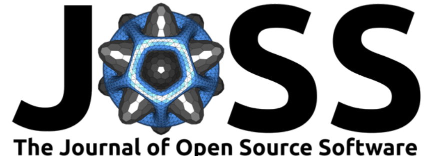
Statement of need

Traditional solvers are challenged by large sparse stiff models and systems with frequent discontinuities. The buck converter is a stiff system with frequent discontinuities that classic solvers from the DifferentialEquations.jl (Rackauckas & Nie, 2017) are currently unable to handle properly. Written in the easy-to-learn Julia programming language (Bezanson et al., 2017) and inspired by the `qss-solver` written in C (Fernández & Kofman, 2014), the QuantizedSystemSolver.jl package takes advantage of Julia features such as multiple dispatch and metaprogramming. The package shares the same interface as the DifferentialEquations.jl package and aims to efficiently solve a large set of stiff Ordinary Differential Equations (ODEs) with events by implementing the QSS and LIQSS methods. It is the first such tool to be published in the Julia ecosystem.

Quantization-based techniques

The general form of a problem composed of a set of ODEs and a set of events that QSS is able to solve is described in the following equations:

$$\dot{X} = f(X, P, t);\ \text{if } zc(X, P, t)\text{: set } x_i = H(X, P, t) \text{ and } p_j = L(X, P, t),$$



where $X = [x_1, x_2..., x_n]^T$ is the state vector, $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^+ \to \mathbb{R}^n$ is the derivative function, and t is the independent variable. $P = [p_1, p_2..., p_m]^T$ is the vector of the system discrete variables. n and m are the number of state variables and discrete variables of the system respectively. zc is an event condition, H and L are functions used in the effects of the event zc.

In QSS, besides the step size, the difference between $x_i(t_k)$ (the current value) and $x_i(t_{k+1})$ (the next value) is called the quantum Δ_i. Depending on the type of the QSS method (explicit or implicit), a new variable q_i is set to equal $x_i(t_k)$ or $x_i(t_{k+1})$ respectively. q_i is called the quantized state of x_i, and it is used in updating the derivative function (Elbellili et al., 2024). A general description of a QSS algorithm is given as follows:

Algorithm 1 QSS1

if a variable x_i needs to change **then**
  Update its value, its quantized state, and its next time of change.
  For x_j depends on x_i, update its value, its derivative, and its next time.
  For any zc depends on x_i, update its value and compute its next event time.
else if an event needs to occur **then**
  Execute the event and update the related quantized variables.
  For x_j depends on the event, update its value, its derivative, and its next time.
  For any zc depends on the event, update its value and compute its next event time.
end if

Package description

While the package is optimized to be fast, extensibility is not compromised. It is divided into three entities that can be extended separately: the `problem`, the `algorithm`, and the `solution`. The rest of the code creates these entities and glues them together. The API was designed to match the DifferentialEquations.jl interface while providing an easier way to handle events. The problem is defined inside a function, in which the user may introduce any parameters, variables, equations, and events:

```
function func(du, u, p, t)
  # parameters, helpers, differential eqs., if-statements for events; e.g.:
  du[1] = p[1] * u[1]
  if (t - 1.0 > 0.0) p[1] = -10.0 end
end
```

Then, this function is passed to an `ODEProblem` function along with the initial conditions, the time span, and any parameters or discrete variables.

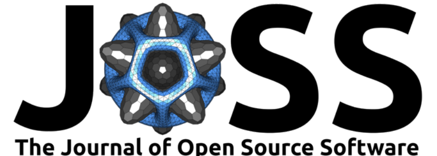
```
tspan = (0.0, 2.0)
u = [10.0] # initial conditions
p = [-1.0] # parameters and discrete variables
odeprob = ODEProblem(func, u, tspan, p)
```

The output of the previous `ODEProblem` function, which is a QSS problem, is passed to a `solve` function with other configuration arguments such as the algorithm type and the tolerance. The `solve` function dispatches on the given algorithm and starts the numerical integration.

```
sol = solve(odeprob, nmliqss2(), abstol = 1E-5, reltol = 1E-5)
```

At the end, a solution object is produced that can be queried, plotted, and analyzed for error.

```
sol(0.05, idxs = 1) # get the value of variable 1 at time 0.05
sol.stats           # get statistics about the simulation
plot(sol)           # plot the solution
```



The solver uses other packages such as `MacroTools.jl` (Innes, 2015) for user-code parsing and `SymEngine.jl` (Fernando, 2015) for Jacobian computation and dependency extraction. It also uses a modified `TaylorSeries.jl` (Benet & Sanders, 2014) that implements caching to obtain free Taylor variable operations, since the current version of TaylorSeries creates a heap allocated object for every operation. The approximation through Taylor variables transforms any complicated equations to polynomials, making root finding cheaper–a process that QSS methods rely on heavily.

The buck converter example

The buck converter decreases the voltage and increases the current with a greater power efficiency than linear regulators (Migoni et al., 2015). Its circuit is shown in Fig.1(a).

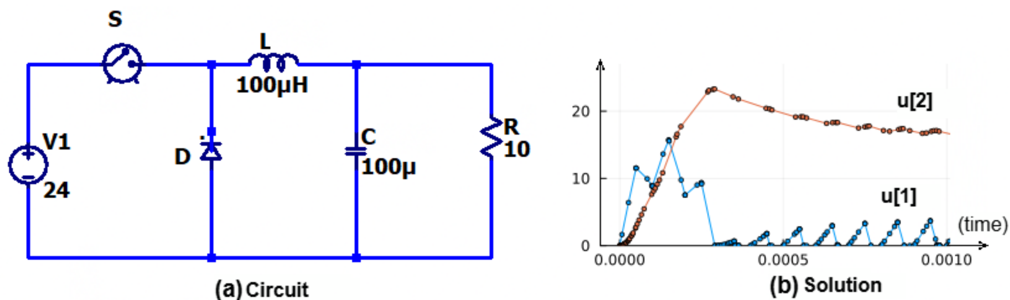


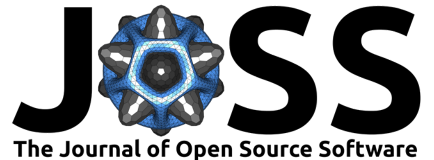
Figure 1: The buck converter

The diode D and the switch S can be modeled as two variable resistors, denoted by RD and RS. A mesh and a nodal analysis give the relationship between the different components in the circuit as follows:

$$i_d = \frac{RS.i_l - V1}{RS + RD}; \ \frac{du_c}{dt} = \frac{i_l - \frac{u_c}{R}}{C}; \ \frac{di_l}{dt} = \frac{-uc - i_d.RD}{L}$$

The buck problem contains frequent discontinuities and can be solved by the QuantizedSystemSolver.jl package using the following code, that generates the solution plot of Fig.1(b):

```
using QuantizedSystemSolver
function buck(du, u, p, t)
  # Constant parameters
  C = 1e-4; L = 1e-4; R = 10.0; V1 = 24.0; T = 1e-4; ROn = 1e-5; ROff = 1e5
  # Optional rename of the continuous and discrete variables
  RD = p[1]; RS = p[2]; nextT = p[3]; lastT = p[4]; il = u[1]; uc = u[2]
  # Equations
  id = (il * RS - V1) / (RD + RS) # diode's current
  du[1] = (-id * RD - uc)/L; du[2] = (il - uc / R) / C
  # Events
  if t - nextT > 0.0 # model when the switch is ON
    lastT = nextT; nextT = nextT + T; RS = ROn
  end
  if t - lastT - 0.5 * T > 0.0 # model when the switch is OFF
    RS = ROff
  end
  if id > 0 # model when the Diode is ON
    RD = ROn;
  else
    RD = ROff;
```



```julia
  end
end
# Initial conditions and time settings
p = [1e5, 1e-5, 1e-4, 0.0]; u0 = [0.0, 0.0]; tspan = (0.0, 0.001)
# Define the problem
QSSproblem = ODEProblem(buck, u0, tspan, p)
# solve the problem
sol = solve(QSSproblem, nmliqss2(), abstol = 1e-3, reltol = 1e-2)
# Get the value of variable 2 at time 0.0005
sol(0.0005, idxs = 2)
# plot the solution
plot(sol)
```

Conclusion

The package provides robust functionality to efficiently solve stiff ODEs with events using the quantized state method. It is well-documented, making it accessible for researchers across various domains. Additionally, users can extend its capabilities to handle a variety of problems.

Acknowledgements

This research has received no external funding.

References

Benet, L., & Sanders, D. P. (2014). *A Julia package for Taylor expansions in one or more independent variables*. https://github.com/JuliaDiff/TaylorSeries.jl. https://doi.org/10.5281/zenodo.2557003

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Elbellili, E., Lauwens, B., & Huybrechs, D. (2024). Investigation of different conditions to detect cycles in linearly implicit quantized state systems. *Proceedings of the 3rd. International Conference on Computational Modeling, Simulation and Optimization*. https://doi.org/10.1109/ICCMSO61761.2024.00095

Fernández, J., & Kofman, E. (2014). A stand-alone quantized state system solver for continuous system simulation. *SIMULATION: Transactions of the Society for Modeling and Simulation International*, *90*, 782–799. https://doi.org/10.1177/0037549714536255

Fernando, I. (2015). *SymEngine.jl*. https://github.com/symengine/SymEngine.jl

Innes, M. (2015). *MacroTools.jl*. https://github.com/FluxML/MacroTools.jl

Migoni, G., Kofman, E., Bergero, F., & Fernandez, J. (2015). Quantization-based simulation of switched mode power supplies. *SIMULATION*, *91*, 320–336. https://doi.org/10.1177/0037549715575197

Pietro, F., Migoni, G., & Kofman, E. (2019). Improving linearly implicit quantized state system methods. In *Simulation: Transactions of the Society for Modeling and Simulation International*. https://doi.org/10.1177/0037549718766689

Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl a performant and feature-rich ecosystem for solving differential equations in Julia. *The Journal of Open Research Software*, *5*, 15–24. https://doi.org/10.5334/jors.151