边值微分代数方程的配点法求解器
&
边值问题求解器的最新进展

曲庆宇

2024.11.3

Master student at Zhejiang University

2023 GSoC SciML student

2024 NumFOCUS supported SciML developer

with research interest in numerical methods, differential equations, and machine learning

Outline

1. Research Overview
2. Recent progress in BVP solving
3. Collocation solvers for BVDAE

1 Research Overview

Research background

Comparison of differential equations solvers suite

Popular packages

- **FORTRAN**: MIRKDC, BVP_SOLVER, COLSYS, COLNEW, COLDAE…
- **MATLAB**: bvp4c, bvp5c
- **Python**: scipy.solve_bvp
- **R**: bvpSolve
- **Julia**: BoundaryValueDiffEq.jl: Shooting, MultipleShooting, MIRK2~MIRK6, RadauIIA1,2,3,5,7, LobattoIIIA2,3,4,5, LobattoIIIB2,3,4,5, LobattoIIIC2,3,4,5, MIRKN2-MIRKN6….

Comparison Of Differential Equation Solver Software

Subject/Item	MATLAB	SciPy	deSolve	DifferentialEquations.jl	undials	Hairer	ODEPACK/Netlib /NAG	JitCODE	PyDSTool	FATODE	GSL	BOOST	Mathematica	Maple
Language	MATLAB	Python	R	Julia	C++ and Fortran	Fortran	Fortran	Python	Python	Fortran	C	C++	Mathematica	Maple
Selection of Methods for ODEs	Fair	Poor	Fair	Excellent	Good	Fair	Good	Poor	Poor	Fair	Poor	Fair	Fair	Fair
Efficiency*	Poor	Poor****	Poor***	Excellent	Excellent	Good	Good	Good	Good	Good	Fair	Fair	Fair	Good
Tweakability	Fair	Poor	Good	Excellent	Excellent	Good	Good	Fair	Fair	Fair	Fair	Fair	Good	Fair
Event Handling	Good	Good	Fair	Excellent	Good**	None	Good**	None	Fair	None	None	None	Good	Good
Symbolic Calculation of Jacobians and Autodifferentiation	None	None	None	Excellent	None	None	None	None	None	None	None	None	Excellent	Excellent
Complex Numbers	Excellent	Good	Fair	Good	None	None	None	None	None	None	None	Good	Excellent	Excellent
Arbitrary Precision Numbers	None	None	None	Excellent	None	None	None	None	None	None	None	Excellent	Excellent	Excellent
Control Over Linear/Nonlinear Solvers	None	Poor	None	Excellent	Excellent	Good	Depends on the solver	None	None	None	None	None	Fair	None
Built-in Parallelism	None	None	None	Excellent	Excellent	None	None	None	None	None	None	Fair	None	None
Differential-Algebraic Equation (DAE) Solvers	Good	None	Good	Excellent	Good	Excellent	Good	None	Fair	Good	None	None	Good	Good
Implicitly-Defined DAE Solvers	Good	None	Excellent	Fair	Excellent	None	Excellent	None	None	None	None	None	Good	None
Constant-Lag Delay Differential Equation (DDE) Solvers	Fair	None	Poor	Excellent	None	Good	Fair (via DDVERK)	Fair	None	None	None	None	Good	Excellent
State-Dependent DDE Solvers	Poor	None	Poor	Excellent	None	Excellent	Good	None	None	None	None	None	None	Excellent
Stochastic Differential Equation (SDE) Solvers	Poor	None	None	Excellent	None	None	None	Good	None	None	None	None	Fair	Poor
Specialized Methods for 2nd Order ODEs and Hamiltonians (and Symplectic Integrators)	None	None	None	Excellent	None	Good	None	None	None	None	None	Fair	Good	None
Boundary Value Problem (BVP) Solvers	Good	Fair	None	Good	None	None	Good	None	None	None	None	None	Good	Fair
GPU Compatibility	None	None	None	Excellent	Good	None	None	None	None	None	None	Good	None	None
Analysis Addons (Sensitivity Analysis, Parameter Estimation, etc.)	None	None	None	Excellent	Excellent	None	Good (for some methods like DASPK)	None	Poor	Good	None	None	Excellent	None

* Efficiency takes into account not only the efficiency of the implementation, but the features of the implemented methods (advanced timestepping controls, existence of methods which are known to be more efficient, Jacobian handling)

** Event handling needs to be implemented yourself using basic rootfinding functionality

*** There is a way to write your own C/Fortran code for the derivative, in which case it nearly matches Julia's speed

**** This timing includes JIT compilation with Numba, see https://github.com/JuliaDiffEq/SciPyDiffEq.jl for timing details

For more detailed explainations and comparisons, see the following blog post:

http://www.stochasticlifestyle.com/a-comparison-between-differential-equation-solver-suites-in-matlab-r-julia-python-c-and-fortran

Scale	None	Poor	Fair	Good	Excellent
Explanation	Functionality does not exist	Functionality exists, but is feature-incomplete	The basic features exist	The basic features exist and some extra tweakability exists. May include extra methods for efficiency.	Has all of the basic features and more. Extra features for flexibility and efficiency.

https://docs.sciml.ai/DiffEqDocs/stable/solvers/bvp_solve/

1 Recent progress in BVP solving

Research background

BVP has the form of:

$$\frac{du}{dt} = f(u,p,t), \quad a < t < b$$

$$g(u(a),u(b)) = 0$$

- Swirling flow problem

$$\begin{cases} \epsilon f'''' + f f''' + g g' = 0 \\ \epsilon g'' + f g' - f' g = 0 \end{cases}$$

$$f(0) = f(1) = f'(0) = f'(1)$$
$$g(0) = \Omega_0, \ \ g(1) = \Omega_1$$

- The best launching time problem(Optimal Control)

$$\begin{cases} x' = \frac{v_x v_c}{h} \\ y' = \frac{v_y v_c}{h} \\ v_x' = \frac{F}{M|v_c|\sqrt{1+\lambda^2}} \\ v_y' = \frac{F}{M|v_c|\sqrt{1+\lambda^2}} - \frac{g}{v_c} \\ \lambda_2' = 0 \\ \lambda' = -\lambda_2 \frac{v_c}{h} \\ t_f' = 0 \end{cases}$$

$$t_0 = 0, \ x_0 = 0, \ y_0 = 0,$$
$$v_{x0} = 0, \ v_{y0} = 0,$$
$$y_f = h, \ v_x(t_f) = v_c, \ v_y(t_f) = v_c$$

BVP solving is vital in modeling and understanding complex dynamical system

Methods overview

Classical Runge-Kutta methods

- Explicit Runge-Kutta methods

$$
\begin{array}{c|cccc}
0 & & & & \\
c_2 & a_{2,1} & & & \\
c_3 & a_{3,1} & a_{3,2} & & \\
\vdots & \vdots & & \ddots & \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1}
\end{array}
$$

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + c_2 h, y_n + (a_{2,1} k_1) h) \\
k_3 &= f(t_n + c_3 h, y_n + (a_{3,1} k_1 + a_{3,2} k_2) h) \\
&\vdots \\
k_s &= f(t_n + c_s h, y_n + (a_{s,1} k_1 + a_{s,2} k_2 + \cdots + a_{s,s-1} k_{s-1}) h)
\end{aligned}
$$

$$
y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i
$$

- Implicit Runge-Kutta methods

$$
\begin{array}{c|cccc}
c_1 & a_{1,1} & a_{1,2} & \cdots & a_{1,s} \\
c_2 & a_{2,1} & a_{2,2} & \cdots & a_{2,s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s,1} & a_{s,2} & \cdots & a_{s,s-1} \\
\hline
 & b_1 & b_2 & \cdots & b_s \\
 & b_1^* & b_2^* & \cdots & b_s^*
\end{array}
$$

$$
\begin{aligned}
k_1 &= f(t_n + c_1 h, y_n + h \textstyle\sum_{l=1}^{s} a_{1,l} k_l) \\
k_2 &= f(t_n + c_2 h, y_n + h \textstyle\sum_{l=1}^{s} a_{2,l} k_l) \\
k_3 &= f(t_n + c_3 h, y_n + h \textstyle\sum_{l=1}^{s} a_{3,l} k_l) \\
&\vdots \\
k_s &= f(t_n + c_s h, y_n + h \textstyle\sum_{l=1}^{s} a_{s,l} k_l)
\end{aligned}
$$

$$
y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i
$$

Methods overview

Monotonic Implicit Runge-Kutta(MIRK) methods

$$\frac{du}{dt} = f(u,p,t), \quad t \in [a,b]$$

$$g_a(y(a)) = 0, \quad g_b(y(b)) = 0$$

$$\{t_i\}_{i=0}^{N}, \quad t_0 = a, \ t_N = b$$

- MIRK methods has Butcher tableau coefficients

$$\begin{array}{c|c|ccccc}
c_1 & v_1 & 0 & 0 & \cdots & 0 & 0 \\
c_2 & v_2 & x_{2,1} & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c_{s-1} & v_{s-1} & x_{s-1,1} & x_{s-1,2} & \cdots & 0 & 0 \\
c_s & v_s & x_{s,1} & x_{s,2} & \cdots & x_{s,s-1} & 0 \\
\hline
 & & b_1(\theta) & b_2(\theta) & \cdots & b_{s-1}(\theta) & b_s(\theta)
\end{array}$$

satisfying $c_j = v_j + \sum_{k=1}^{j-1} a_{j,k}$

- Computation of discrete stages

$$K_j = f(t_i + c_j h_i, (1 - v_j) y_i + v_j y_{i+1} + h_i \sum_{r=1}^{j-1} x_{jr} K_r)$$

- Collocation equations in subinterval

$$\phi_i(y_i, y_{i+1}) = y_{i+1} - y_i - h_i \sum_{j=1}^{s} b_j K_j = 0$$

- Nonlinear system in the whole time span

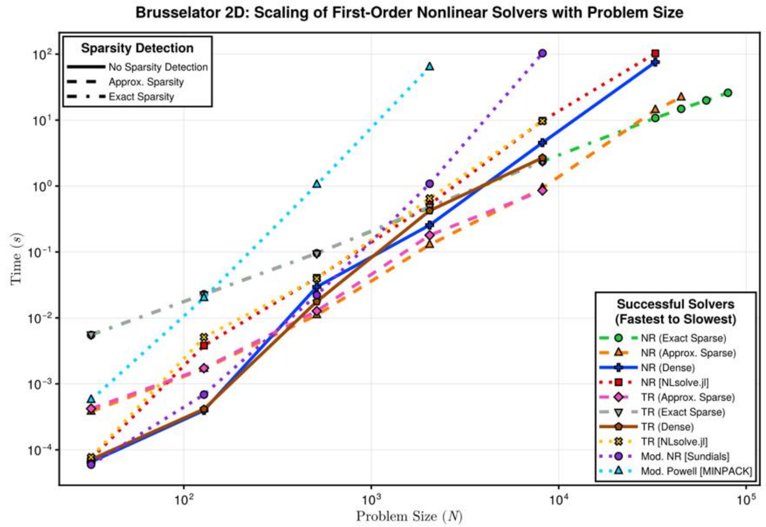
$$\Phi(Y) = \begin{pmatrix} g_a(y_0) \\ \phi_1(y_0, y_1) \\ \vdots \\ \phi_N(y_{N-1}, y_N) \\ g_b(y_N) \end{pmatrix} = 0$$

2 Recent progress in BVP solving

Methods overview — MIRK methods

- Using **NonlinearSolve.jl** to solve the nonlinear system, forming the Newton iteration with a banded matrix

$$
\begin{pmatrix}
\frac{\partial g_0^{(m)}}{\partial y_0} & 0 & 0 & \cdots & 0 & 0 \\
\frac{\partial \phi_0^{(m)}}{\partial y_0} & \frac{\partial \phi_0^{(m)}}{\partial y_1} & 0 & \cdots & 0 & 0 \\
0 & \ddots & \ddots & \cdots & 0 & 0 \\
0 & 0 & 0 & \cdots & \frac{\partial \phi_{N-1}^{(m)}}{\partial y_{N-1}} & \frac{\partial \phi_{N-1}^{(m)}}{\partial y_N} \\
0 & 0 & 0 & \cdots & 0 & \frac{\partial g_1^{(m)}}{\partial y_N}
\end{pmatrix}
\begin{pmatrix}
\Delta y_0^{(m)} \\ \vdots \\ \Delta y_N^{(m)}
\end{pmatrix}
= -
\begin{pmatrix}
g_0(y_0^{(m)}) \\ \phi_0^{(m)} \\ \vdots \\ \phi_{N-1}^{(m)} \\ g_1(y_N^{(m)})
\end{pmatrix}
$$



- Utilize SparseDiffTools.jl to exploit the sparse pattern and matrix coloring to accelerate the nonlinear solving

- Using automatical jacobian selection algorithm to solve the nonlinear system, then we get the initial discrete solution and the discrete stages for the defect estimation

$$\phi_i(y_i, y_{i+1}) = y_{i+1} - y_i - h_i \sum_{j=1}^{s} b_j K_j = 0 \qquad K_j = f(t_i + c_j h_i, (1 - v_j) y_i + v_j y_{i+1} + h_i \sum_{r=1}^{j-1} x_{jr} K_r)$$

[NonlinearSolve.jl: High-Performance and Robust Solvers for Systems of Nonlinear Equations in Julia]

2 Recent progress in BVP solving

Defect control adaptivity

Adaptivity in MIRK and FIRK

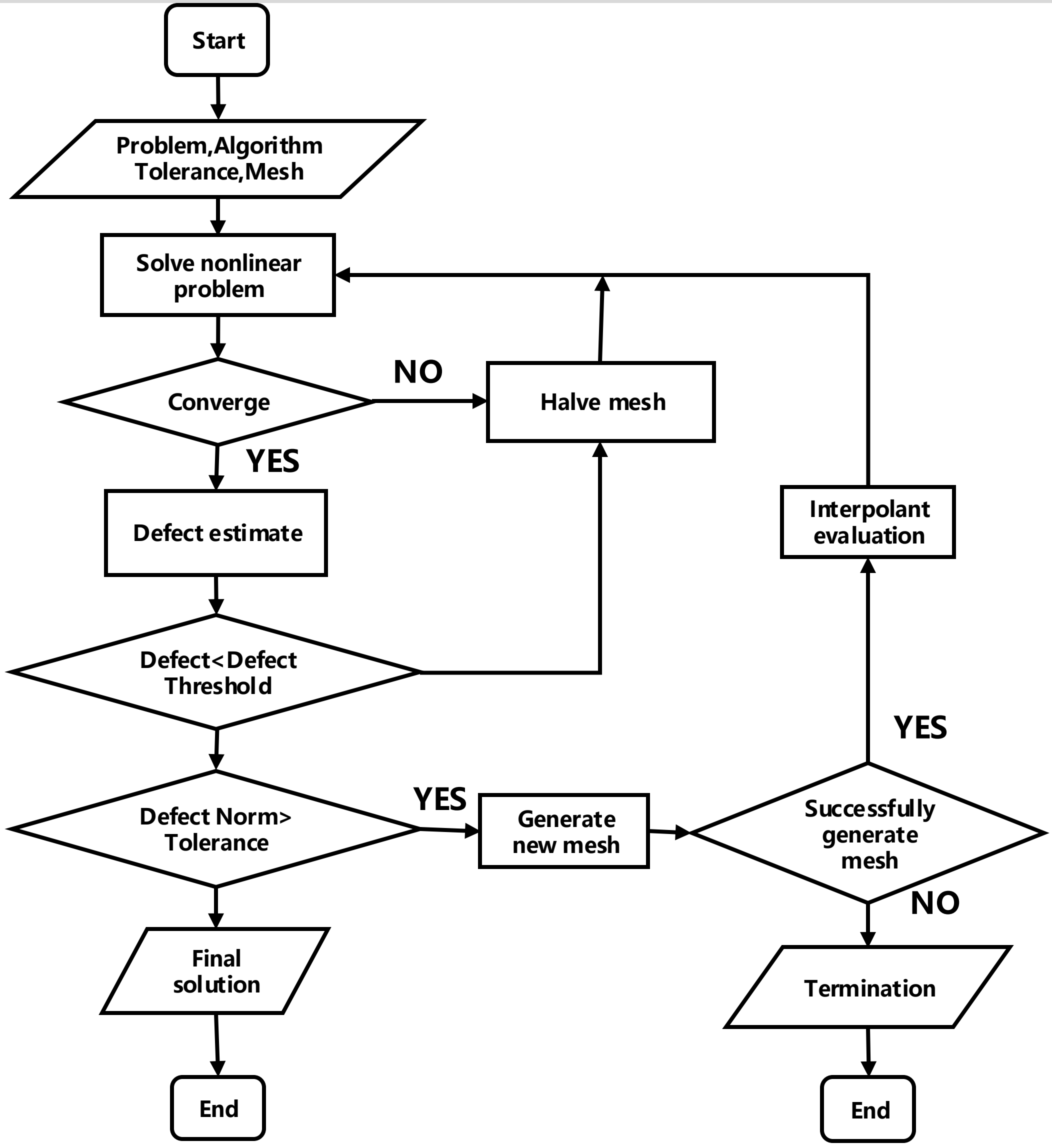
Improve accuracy→Refine mesh→Low efficiency

Improve efficiency→Coarsen mesh→Low accuracy

The balance between **accuracy** and **efficiency**

The core of the defect control adaptivity: According to the computing defect refine the grid to achieve more accurate numerical solutions

[Numerical Solution of Boundary Value Problems for Ordinary Differential Equations]



2 Recent progress in BVP solving

Collocation methods

Computing example

A simple boundary value problem

$$z'' = z$$

Satisfying boundary conditions

$$z(0) = 1, \quad z(1) = 0$$

Transforming to first-order system

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ \frac{1}{\lambda} f(y_1) \end{pmatrix}$$

With boundary conditions

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y_1(1) \\ y_2(1) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Setting **`adaptive`** kwargs as **`true`**, we can turn on defect control adaptivity(on by default)

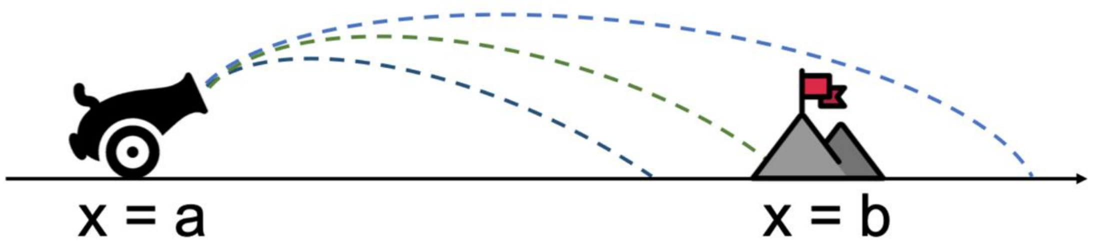
```julia
using BoundaryValueDiffEq, Plots

function f!(du, u, p, t)
    du[1] = u[2]
    du[2] = 1/p * u[1]
end
function bc!(resid, u, p, t)
    resid[1] = u[1][1] - 1
    resid[2] = u[1][end]
end
u0 = [0, 0]
tspan = (0.0, 1.0)
prob = BVProblem(f!, bc!, u0, tspan)
sol = solve(prob, MIRK4(), adaptive=true)
```

2 Recent progress in BVP solving

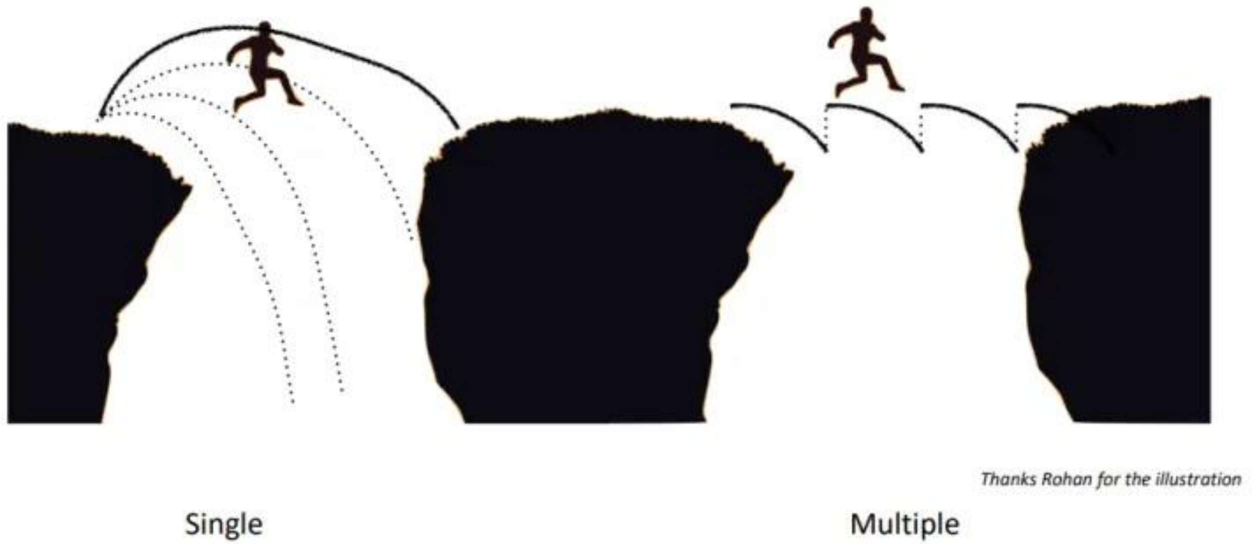
Shooting methods

Basic idea of shooting methods

`Shooting` method convert BVP to IVP and adjusting its numerical solution to satisfy the boundary conditions



`MultipleShooting` method divides the interval into several subintervals and forms several IVPs, then uses a similar idea of Shooting to solve the BVP



Usage example

```
using BoundaryValueDiffEq, OrdinaryDiffEq
sol = solve(prob, Shooting(Tsit5()), dt=0.01)
```

```
using BoundaryValueDiffEq, OrdinaryDiffEq
sol = solve(prob, MultipleShooting(5, Tsit5()), dt=0.01)
```

2 Recent progress in BVP solving

Problem constructor — Boundary value problem construction

Normal Boundary value problem
BVProblem
Has two-point constraints or
multi-points constraints

```julia
using BoundaryValueDiffEq, Plots

function f!(du, u, p, t)
    du[1] = u[2]
    du[2] = 1/p * u[1]
end
function bc!(resid, u, p, t)
    resid[1] = u[1][1] - 1
    resid[2] = u[1][end]
end
u0 = [0, 0]
tspan = (0.0, 1.0)
prob = BVProblem(f!, bc!, u0, tspan)
sol = solve(prob, MIRK4(), adaptive=true)
```

Two-point boundary value problems
TwoPointBVProblem
Constraints at start and end

```julia
using BoundaryValueDiffEq, Plots

function f!(du, u, p, t)
    du[1] = u[2]
    du[2] = 1/p * u[1]
end
function bca!(resid, u_a, p)
    resid[1] = u_a[1] - 1
end
function bcb!(resid, u_b, p)
    resid[1] = u_b[1]
end
u0 = [0.0, 0.0]
tspan = (0.0, 1.0)
prob = TwoPointBVProblem(f!, (bca!, bcb!), u0, tspan;
        bcresid_prototype = (zeros(1), zeros(1)))
sol = solve(prob, MIRK4(), adaptive=true)
```

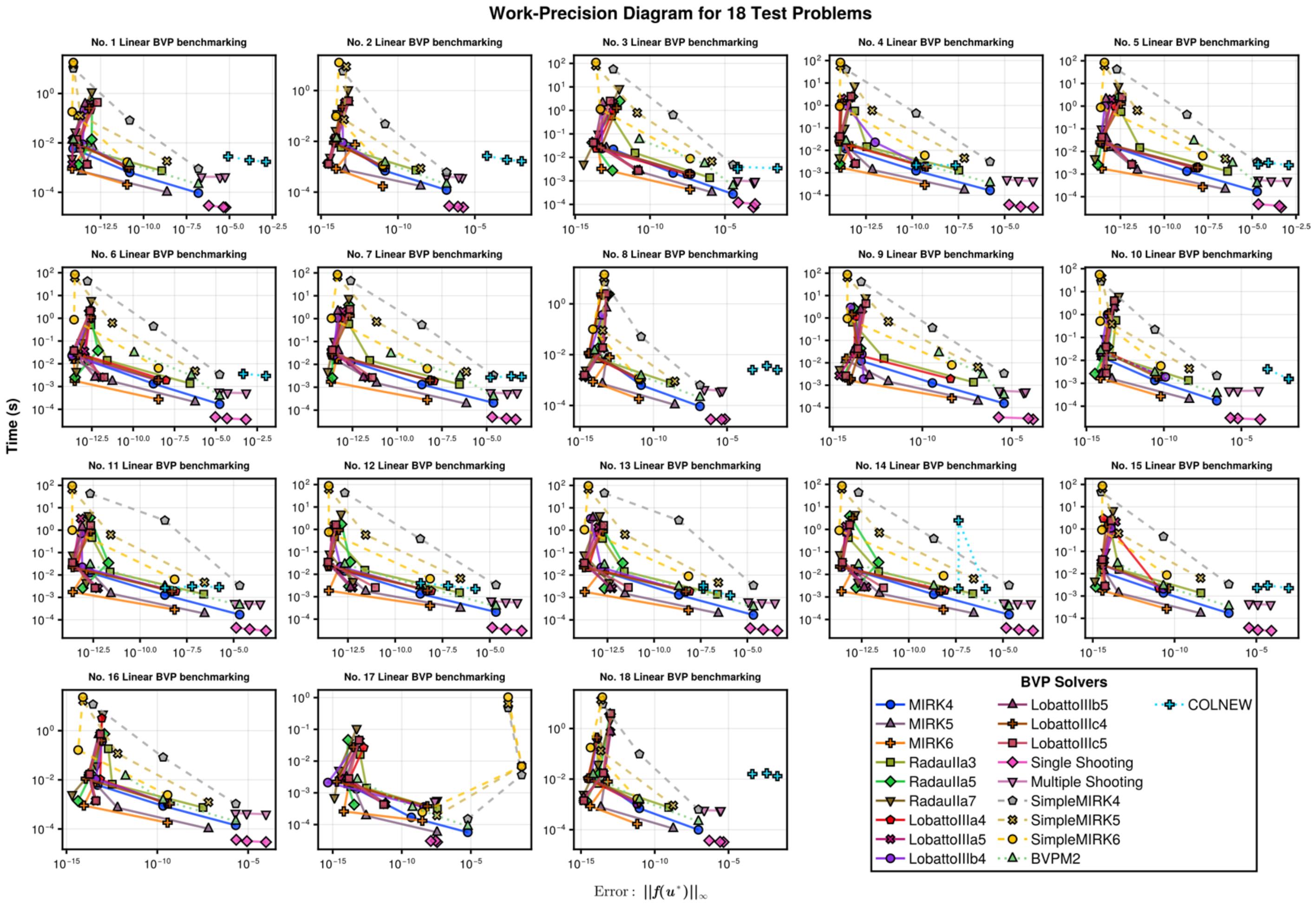
2 Recent progress in BVP solving

Methods evaluation

Benchmarks

Use standard testing problem set to benchmark algorithms in BoundaryValueDiffEq.jl:

- MIRK2~MIRK6 methods
- RadauIIA methods
- LobattoIIA, LobattoIIB, LobattoIIC methods
- SingleShooting methods
- MultipleShooting methods
- BVPM2(FORTRAN BVP solver)
- COLNEW(FORTRAN BVP solver)
- BVPSOL(FORTRAN BVP solver)



https://docs.sciml.ai/SciMLBenchmarksOutput/dev/NonStiffBVP/linear_wpd/

https://archimede.uniba.it/~bvpsolvers/testsetbvpsolvers/?page_id=29

Methods overview

Runge-Kutta-Nystrom(RKN) methods

$$y''(t) = f(t, y(t), y'(t))$$

$$y(t_0) = y_0, \quad y'(t_0) = y'_0$$

- RKN methods has Butcher table formation

$$\begin{array}{c|ccc|ccc}
c_1 & a_{1,1} & \cdots & a_{1,s} & \bar{a}_{1,1} & \cdots & \bar{a}_{1,s} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s,1} & \cdots & a_{s,s} & \bar{a}_{s,1} & \cdots & \bar{a}_{s,s} \\
\hline
 & b_1 & \cdots & b_s & \bar{b}_1 & \cdots & \bar{b}_s
\end{array}$$

satisfying $c_j = \sum_{i=1}^{s} a_{i,j}$

- Computation of discrete stages

$$K_i = f\Big(t_0 + c_i h, \; y_0 + c_i h y'_0 + h^2 \sum_{j=1}^{i} \bar{a}_{i,j} K_j, \; y'_0 + h \sum_{j=1}^{i} a_{i,j} K_j\Big)$$

- Iterations in the whole interval

$$y_{n+1} = y_n + h y'_n + h^2 \sum_{i=1}^{s} \bar{b}_i K_i$$

$$y'_{n+1} = y'_n + h \sum_{i=1}^{s} b_i K_i$$

Methods overview

Monotonic Implicit Runge-Kutta-Nystrom(MIRKN) methods

$$y''(t) = f(t, y(t), y'(t))$$

$$g(y(a), y'(a), y(b), y'(b)) = 0$$

$$\{t_i\}_{i=0}^{N}, \quad t_0 = a, \ t_N = b$$

- MIRKN methods has Butcher table formation

$$\begin{array}{c|c|c|ccccc|c|ccccc}
c_1 & v_1 & w_1 & 0 & 0 & \cdots & 0 & 0 & v_1' & 0 & 0 & \cdots & 0 & 0 \\
c_2 & v_2 & w_2 & x_{2,1} & 0 & \cdots & 0 & 0 & v_2' & x_{2,1}' & 0 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c_{s-1} & v_{s-1} & w_{s-1} & x_{s-1,1} & x_{s-1,2} & \cdots & 0 & 0 & v_{s-1}' & x_{s-1,1}' & x_{s-1,2}' & \cdots & 0 & 0 \\
c_s & v_s & w_s & x_{s,1} & x_{s,2} & \cdots & x_{s,s-1} & 0 & v_s' & x_{s,1}' & x_{s,2}' & \cdots & x_{s,s-1}' & 0 \\
\hline
 & & & b_1(\theta) & b_2(\theta) & \cdots & b_{s-1}(\theta) & b_s(\theta) & & b_1'(\theta) & b_2'(\theta) & \cdots & b_{s-1}'(\theta) & b_s'(\theta)
\end{array}$$

satisfying $\quad c_r = v_r' + \sum_{j=1}^{s} x_{rj}'$

- Computation of discrete stages

$$K_r = f\Big(t_{i-1} + c_r h,$$
$$(1 - v_r)y_{i-1} + v_r y_i + h((c_r - v_r - w_r)y_{i-1}' + w_r y_i') + h^2 \sum_{j=1}^{r-1} x_{rj} K_j,$$
$$(1 - v_r')y_{i-1}' + v_r' y_i' + h \sum_{j=1}^{r-1} x_{rj}' K_j\Big)$$

- Nonlinear equation in the whole interval

$$\begin{bmatrix} y_i - y_{i-1} - h y_{i-1}' - h^2 \sum_{r=1}^{s} b_r k_r \\ y_i' - y_{i-1}' - h \sum_{r=1}^{s} b_r' k_r \end{bmatrix} = 0$$

2 Recent progress in BVP solving

Methods overview — Use MIRKN methods

Example second-order BVP

$$\begin{cases} y_1''(x) = y_2(x) \\ \epsilon y_2''(x) = -y_1(x)y_2'(x) - y_3(x)y_3'(x) \\ \epsilon y_3''(x) = y_1'(x)y_3(x) - y_1(x)y_3'(x) \end{cases}$$

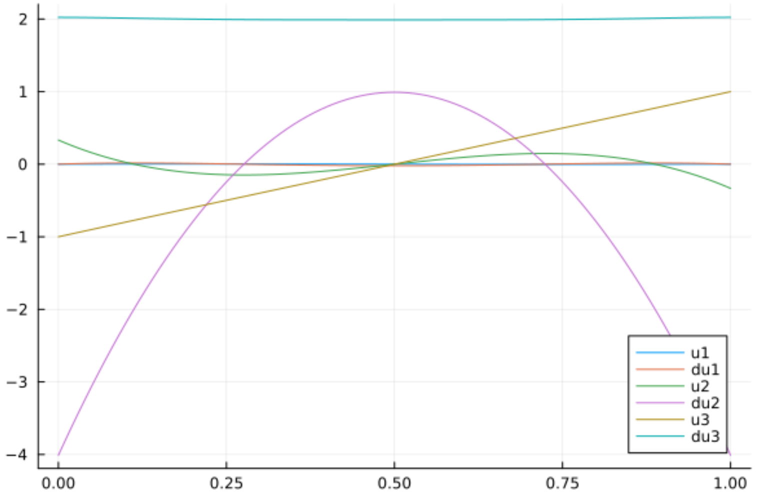
With boundary conditions

$$y_1(0) = y_1'(0) = y_1(1) = y_1'(1) = 0$$

$$y_3(0) = -1, \quad y_3(1) = 1$$

Successful solving:

```
using BoundaryValueDiffEq, Plots
function test!(ddu, du, u, p, t)
    ϵ = 0.1
    ddu[1] = u[2]
    ddu[2] = (-u[1]*du[2] - u[3]*du[3])/ϵ
    ddu[3] = (du[1]*u[3] - u[1]*du[3])/ϵ
end
function bc!(res, du, u, p, t)
    res[1] = u[1][1]
    res[2] = u[end][1]
    res[3] = u[1][3] + 1
    res[4] = u[end][3] - 1
    res[5] = du[1][1]
    res[6] = du[end][1]
end
u0 = [1.0, 1.0, 1.0]
tspan = (0.0, 1.0)
prob = SecondOrderBVProblem(test!, bc!, u0, tspan)
sol = solve(prob, MIRKN4(), dt=0.01)
```



Comparison between MIRK and MIRKN:

ϵ	MIRK methods	MIRKN methods
0.1	8.11×10^{-2}	4.36×10^{-2}
0.01	2.89×10^{-1}	1.50×10^{-1}
0.001	1.14	4.40×10^{-1}
0.0001	5.77	3.33
10^{-5}	72.8	24.0
10^{-6}	143.3	59.5
10^{-7}	515.7	210.8

[MONO-IMPLICIT RUNGE–KUTTA–NYSTROM METHODS WITH APPLICATION TO BOUNDARY VALUE ORDINARY DIFFERENTIAL EQUATIONS]

2 Recent progress in BVP solving

Problem constructor

Second order boundary value problem construction

Normal Boundary value problem
SecondOrderBVProblem
Has two-point constraints or multi-points constraints

```
using BoundaryValueDiffEq, Plots
function f!(ddu, du, u, p, t)
    ϵ = 0.1
    ddu[1] = u[2]
    ddu[2] = (-u[1]*du[2] - u[3]*du[3])/ϵ
    ddu[3] = (du[1]*u[3] - u[1]*du[3])/ϵ
end
function bc!(res, du, u, p, t)
    res[1] = u[1][1]
    res[2] = u[end][1]
    res[3] = u[1][3] + 1
    res[4] = u[end][3] - 1
    res[5] = du[1][1]
    res[6] = du[end][1]
end
u0 = [1.0, 1.0, 1.0]
tspan = (0.0, 1.0)
prob = SecondOrderBVProblem(f!, bc!, u0, tspan)
sol = solve(prob, MIRKN4(), dt=0.01)
```

Two-point boundary value problems
TwoPointSecondOrderBVProblem
Constraints at start and end

```
using BoundaryValueDiffEq, Plots
function f!(ddu, du, u, p, t)
    ϵ = 0.1
    ddu[1] = u[2]
    ddu[2] = (-u[1]*du[2] - u[3]*du[3])/ϵ
    ddu[3] = (du[1]*u[3] - u[1]*du[3])/ϵ
end
function bca!(res, du, u, p)
    res[1] = u[1]
    res[3] = u[3] + 1
    res[5] = du[1]
end
function bcb!(res, du, u, p)
    res[1] = u[1]
    res[2] = u[3] - 1
    res[3] = du[1]
end
u0 = [1.0, 1.0, 1.0]
tspan = (0.0, 1.0)
prob=TwoPointSecondOrderBVProblem(f!,(bca!, bcb!),u0,tspan,
        bcresid_prototype=(zeros(3),zeros(3)))
sol = solve(prob, MIRKN4(), dt=0.01)
```

Boundary Value Differential-Algebraic Equations has the form of

$$
\begin{cases}
D^{m_1}u_1 = f_1(t,\mathbf{z}(\mathbf{u}),\mathbf{y}) \\
D^{m_2}u_2 = f_2(t,\mathbf{z}(\mathbf{u}),\mathbf{y}) \\
\quad\cdots \\
D^{m_d}u_d = f_d(t,\mathbf{z}(\mathbf{u}),\mathbf{y}) \\
0 = f_{d+1}(t,\mathbf{z}(\mathbf{u}),\mathbf{y}) \\
\quad\cdots \\
0 = f_{d+m}(t,\mathbf{z}(\mathbf{u}),\mathbf{y})
\end{cases}
\qquad
\begin{cases}
g_1(\beta_1,\mathbf{z}(\mathbf{u})(\beta_1)) = 0 \\
g_2(\beta_2,\mathbf{z}(\mathbf{u})(\beta_2)) = 0 \\
\quad\cdots \\
g_{m^\star}(\beta_{m^\star},\mathbf{z}(\mathbf{u})(\beta_{m^\star})) = 0
\end{cases}
$$

- Hydrodynamic semiconductor

$$
\begin{cases}
x_1' = x_2 y - \alpha J \\
x_2' = y - 1 \\
0 = x_1 - (\frac{J^2}{y} + y)
\end{cases}
$$

$$y(0) = y(\beta) = \bar{n}$$

Numerical Solution of Boundary Value Problems for Ordinary Differential Equation

Methods overview

Dilemma of Boundary value differential-algebraic equations

BDF and implicit Runge-Kutta behave well on initial value DAE, but cause instability when solving boundary value DAE

The possible occurrence of increasing solution modes for BVPs render robust schemes such as BDF unstable

We need native BVDAE solvers in the SciML universe!!

Research background

	BVDAE solvers
MATLAB	BVPSUITE
Mathematica	None
Fortran	COLDAE
Python	Interfacing COLDAE(solve_bvp)
R	Interfacing COLDAE(bvpSolve)
Julia	**SciML BoundaryValueDiffEq.jl**

Methods overview

BVDAE Gauss-Legendre collocation

1.Build collocation equations with Implicit RK tableau

- In Gauss-Legendre collocation points: $t_i = t_{n-1} + \rho_j h_n, \quad j = 1, \cdots, k$

Monomial representation for piecewise approximation

$$u_i^\pi = \sum_{l=0}^{m_i-1} \frac{(t-t_{n-1})^l}{l!} z_{n-1,p+l} + (h_n)^{m_i} \sum_{l=1}^{k} \psi_l\left(\frac{t-t_{n-1}}{h_n}\right)\omega_{il}$$

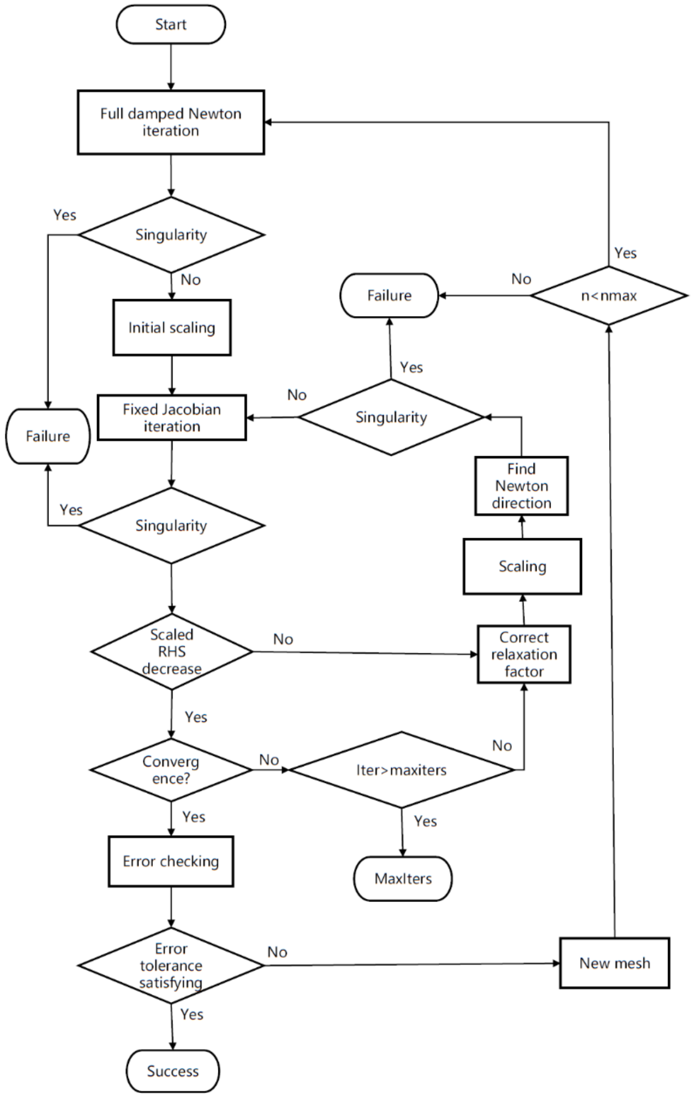
Which satisfying

$$D^j \psi_l(0) = 0, \;\; j = 0, \cdots, m_i - 1$$

$$D^{m_i} \psi_l(\rho_i) = \delta_{j,l} \;\; j, l = 1, \cdots, k$$

- Substitute to the collocation equations:

$$\begin{aligned} D^{m_i} u_i^\pi(t_j) &= f_i(t_j, z^\pi(t_j), y^\pi(t_j)), \\ 0 &= f_i(t_j, z^\pi(t_j), y^\pi(t_j)) \end{aligned}$$



Methods overview

BVDAE Gauss-Legendre collocation

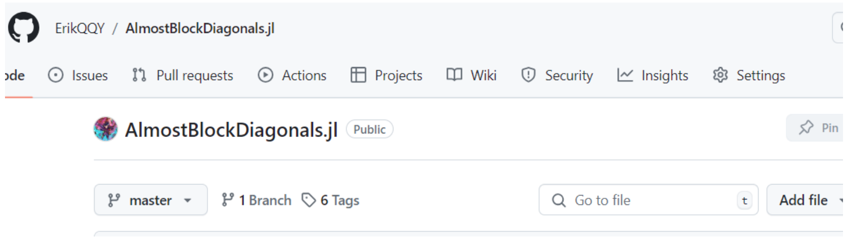
2. Construct a nonlinear system

$$\begin{aligned} \mathcal{D}^{m_i} u_i^\pi(t_j) &= f_i(t_j, \mathbf{z}^\pi(t_j), \mathbf{y}^\pi(t_j)), \\ 0 &= f_i(t_j, \mathbf{z}^\pi(t_j), \mathbf{y}^\pi(t_j)), \end{aligned} \quad \xrightarrow{\text{Residual control}} \quad f(u, p) = 0$$

Discretized BVDAE has the underlying geometry in the form of

Almost Block Diagonal Matrices

$$\begin{array}{cccccccccc}
x & x & x & x & & & & & & \\
x & x & x & x & & & & & & \\
x & x & x & x & & & & & & \\
 & & x & x & x & x & & & & \\
 & & x & x & x & x & & & & \\
 & & & & x & x & x & x & & \\
 & & & & x & x & x & x & & \\
 & & & & & & x & x & x & x \\
 & & & & & & x & x & x & x \\
 & & & & & & x & x & x & x
\end{array}$$

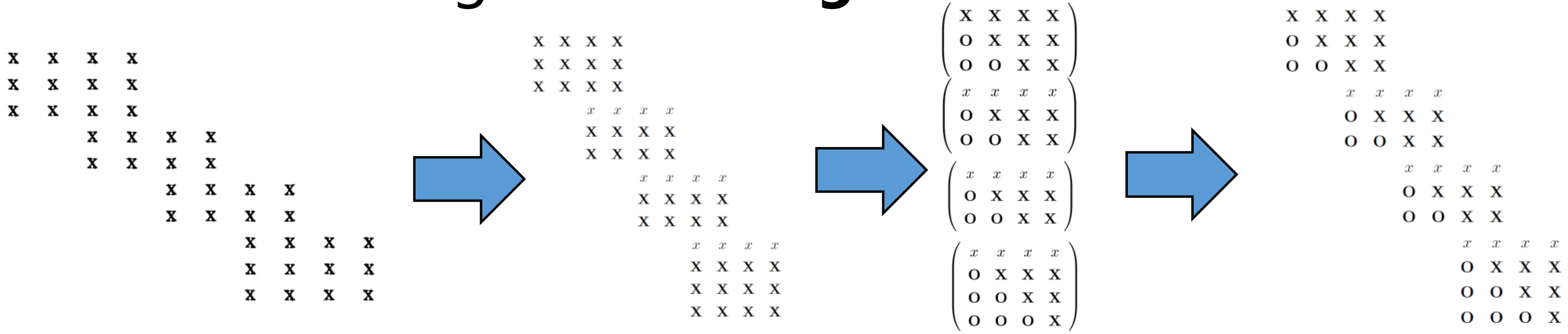


https://github.com/ErikQQY/AlmostBlockDiagonals.jl

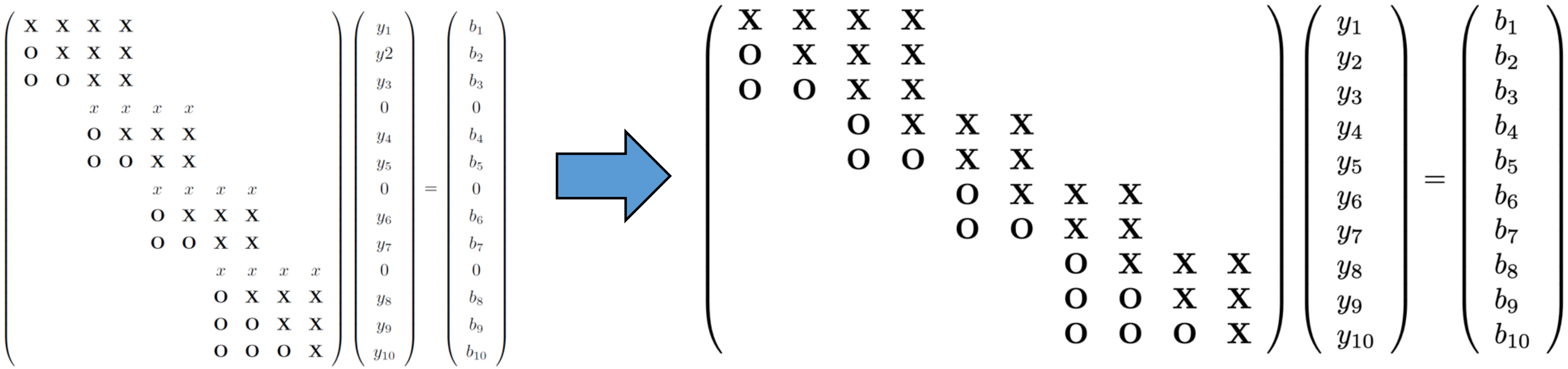
[SOLVEBLOK: A Package for Solving Almost Block Diagonal Linear Systems]()

BVDAE Gauss-Legendre collocation

Almost block diagonal matrix **gaussian elimination**



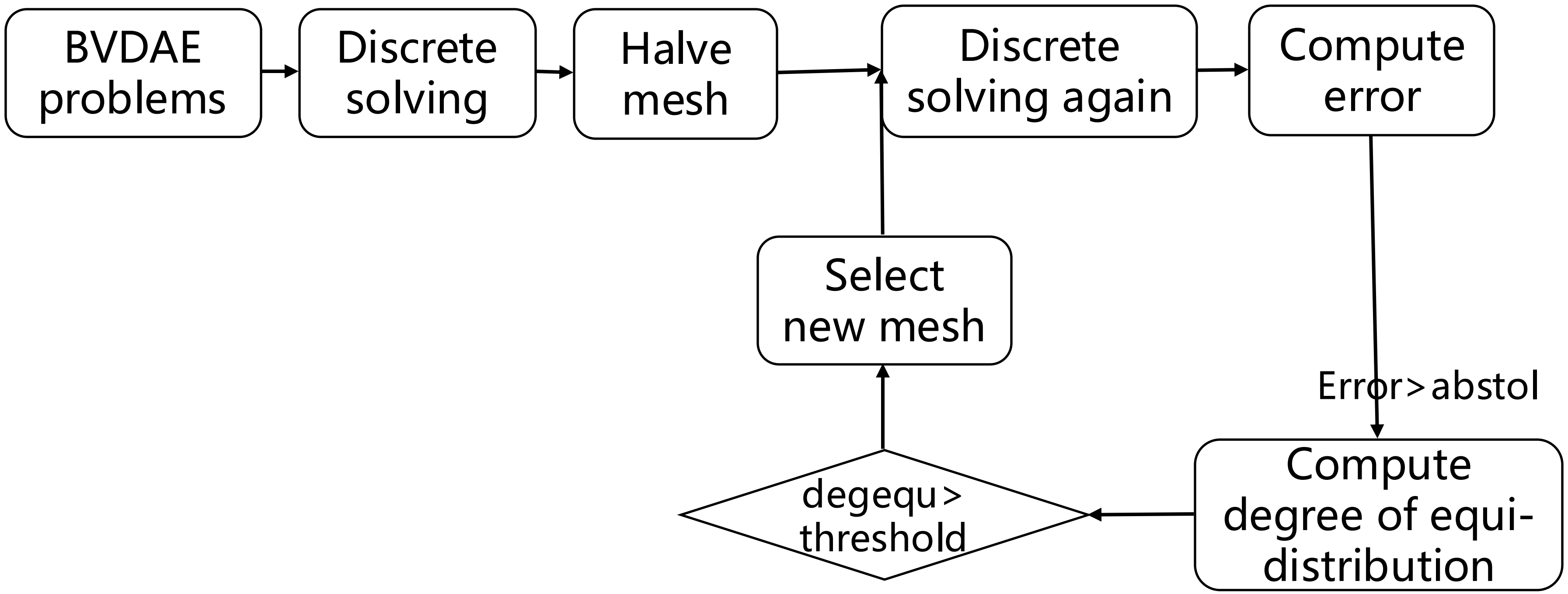
Factorized almost block diagonal matrix **substitution**

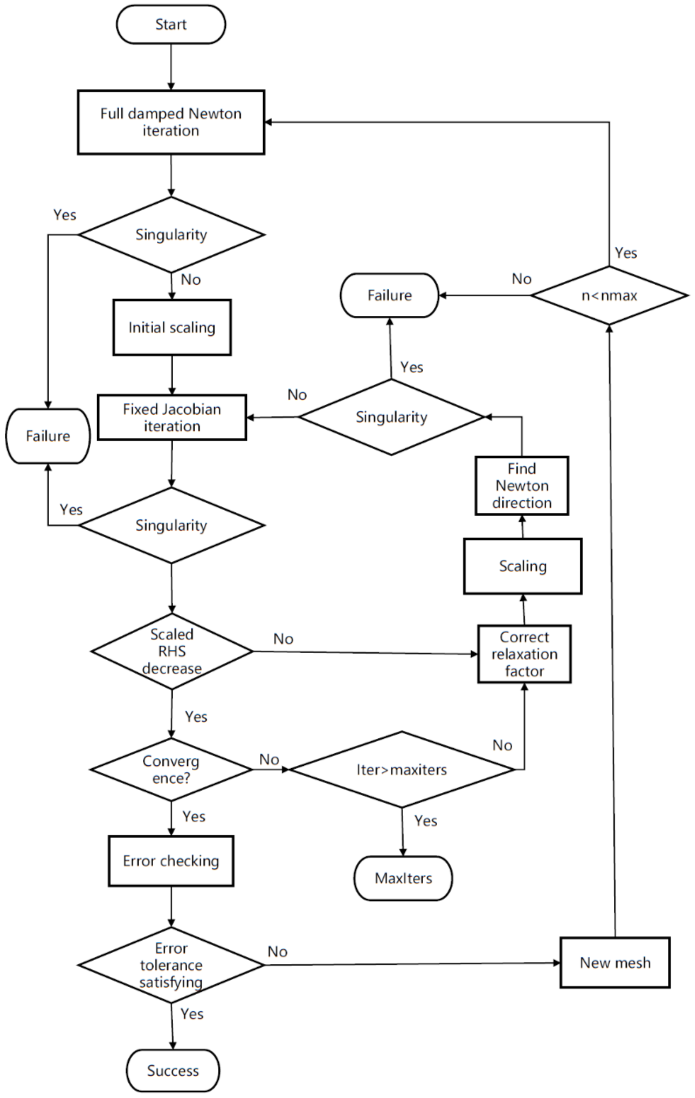


Methods overview

BVDAE Gauss-Legendre collocation

3. Error control and mesh refinement





“A Collocation Solver for Mixed Order Systems of Boundary Value Problems” by U. Ascher et al

Numerical Example

Numerical index-1 BVDAE example:

$$\begin{cases} x_1' = (\epsilon + x_2 - p_2(t))y + p_1'(t) \\ x_2' = p_2'(t) \\ x_3' = y \\ 0 = (x_1 - p_1(t))(y - e^t) \end{cases}$$

With boundary conditions

$$\begin{cases} x_1(0) = p_1(0) \\ x_2(1) = p_2(0) \\ x_3(0) = 1 \end{cases}$$

Successful solving:

```julia
function f!(du, u, p, t)
    du[1] = (1 + u[2] - sin(t)) * u[4] + cos(t)
    du[2] = cos(t)
    du[3] = u[4]
    du[4] = (u[1] - sin(t)) * (u[4] - e^t)
end
function bc!(res, u, p, t)
    res[1] = u[1]
    res[2] = u[3] - 1
    res[3] = u[2] - sin(1.0)
end
u0 = [0.0, 0.0, 0.0, 0.0]
tspan = (0.0, 1.0)
zeta = [0.0, 0.0, 1.0]
fun = ODEFunction(f!, bc!;
    differential_vars=[true,true,true,false])
prob = BVProblem(fun, bc!, u0, tspan, 2.7)
sol = solve(prob, Ascher4(zeta = zeta), dt = 0.01)
```

```
julia> sol = solve(prob, Ascher4(zeta=zeta1), dt=0.05)
retcode: Success
Interpolation: 1st order linear
t: 21-element Vector{Float64}:
 0.0
 0.05
 0.1
 0.15
 0.2
 0.25
 0.3
 0.35
 0.4
 0.45
 0.5
 0.55
 0.6
 0.65
 0.7
 0.75
 0.8
 0.85
 0.9
 0.95
 1.0
u: 21-element Vector{Vector{Float64}}:
 [0.0, -6.787465755374971e-16, 1.0, 2.9754690790284775e-9]
 [0.04997916923348861, 0.0499791692706777, 0.9999999999628102, 1.7844546770893853e-8]
 [0.09983341657254163, 0.09983341664682757, 0.9999999999257134, 3.266901932284832 4e-8]
 [0.1494381323624016, 0.1494381324735987, 0.9999999998888023, 4.7411838894060543e-8]
 [0.19866933064723027, 0.19866933079506074, 0.999999999852169, 6.203622843168631e-8]
 [0.2474039590704283, 0.2474039592545225, 0.9999999998159054, 7.650549023079141e-8]
 [0.29552020644144134, 0.29552020666133916, 0.9999999997801016, 9.078355482885126e-8]
 [0.34289780720029905, 0.34289780745545095, 0.9999999997448475, 1.0483470975828086e-7]
 [0.3894183420188165, 0.389418342308650 2, 0.9999999997102313, 1.1862401354588583e-7]
 [0.4349655337875697, 0.4349655341112299, 0.9999999996763393, 1.3211656521215883e-7]
 [0.4794255382474592, 0.4794255386042028, 0.9999999996432563, 1.452789894530286e-7]
 [0.522687228541724, 0.522687228930659, 0.9999999996110649, 1.580784593297911 8e-7]
 [0.5646424729748812, 0.5646424733950351, 0.9999999995798458, 1.7048243029711375e-7]
 [0.6051864052857161, 0.6051864057360393, 0.9999999995496763, 1.8246053849142003e-7]
 [0.6442176867583241, 0.6442176872376908, 0.999999999520633, 1.9398259720996858e-7]
 [0.6816387595161225, 0.681638760023334, 0.9999999994927878, 2.0501953594308267e-7]
 [0.7173560903657336, 0.7173560908995227, 0.9999999994662103, 2.1554426766372893e-7]
 [0.7512804045812602, 0.7512804051402926, 0.9999999994409671, 2.2553023152986318e-7]
 [0.7833269090446044, 0.7833269096274833, 0.9999999994171208, 2.3495264461264017e-7]
 [0.8134155041841057, 0.8134155047893736, 0.9999999993947313, 2.4378756662330396e-7]
 [0.8414709841817527, 0.8414709848078965, 0.9999999993738558, 0.0]
```

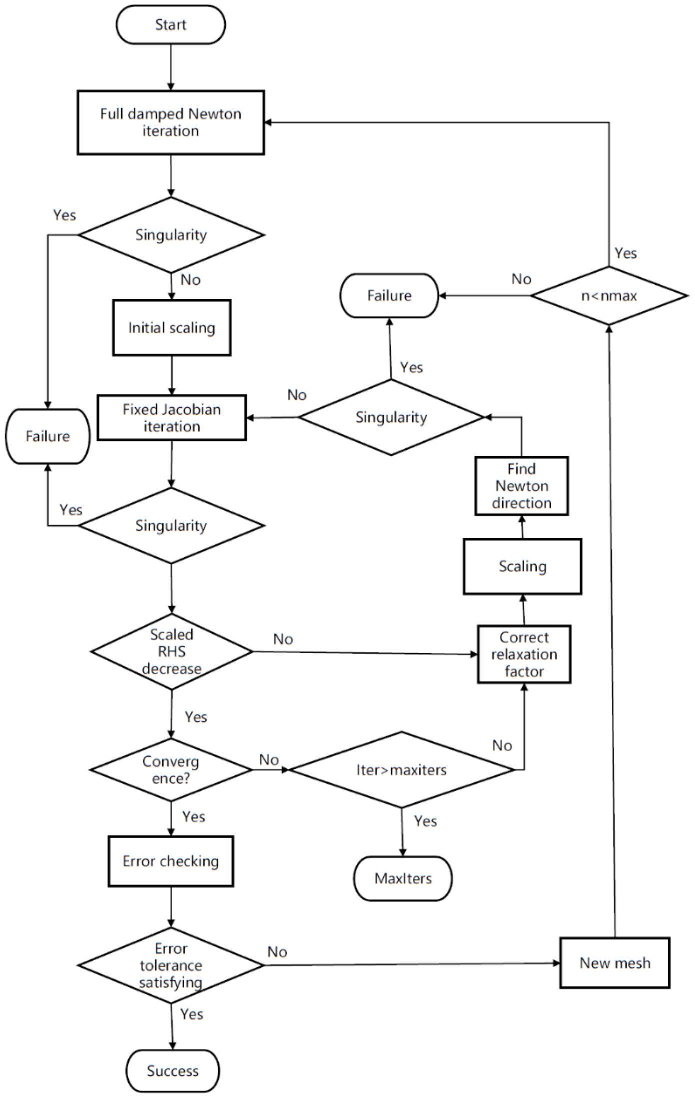
The final solver

➢ The **expected final implemented solver**:

- Collocation for semi-explicit index-1 BVDAE
- Projected collocation methods

➢ **Future plans of the project**:

- Refactor the current implementation to the SciML style
- Support higher-index and mixed-order BVDAE problem solving
- Benchmarks against Fortran solvers



How to get involved?

Google Summer of Code

Apply

Interested contributors propose a project to work on.

Code

Accepted GSoC contributors spend the summer coding with guidance from a mentor.

Share

Submit your code for the world to use!

开源之夏

1. 03/11 - 03/31 社区报名与审核
2. 04/04 - 04/29 社区提交项目与审核
3. 04/30 - 06/04 学生注册、与导师沟通项目、提交项目申请
4. 06/05 - 06/25 项目申请审核
5. 06/26 中选公示
6. 06/26 - 06/30 项目开发预热
7. 07/01 - 09/30 项目开发
8. 10/01 - 10/31 结项审核-PR/MR 合并与导师审核
9. 当前 11/01 - 11/08 结项审核-组委会成果审核
10. 11/09 结项公示
11. 11月 年度优秀学生评选

SciML Fellowship

SciML Small Grants Program:
Funded Open Source Contributions to Julia's SciML

Process:

- Pick a project from the project list
- Send a short application to sciml@julialang.org
- Solidify acceptance criteria with the steering council and reviewer
- Once accepted, you have 1 month to complete the issue, no other applications accepted for the project in that time frame.
- Make pull requests, submit code
- Get code reviewed by experts
- When merged, profit! $$$

For details, see the projects list:
https://sciml.ai/small_grants/

Help SciML Improve! Projects include:

- Improving compilation latency and startup times
- Improvements to the package structure for designing new solver algorithms
- Developing and maintaining benchmarks

And many more to come

Interested in helping SciML improve faster? Consider donating to the SciML Open Source Organization to help fund more small grants projects! Earmarking funds for specific projects is accepted and we'd be happy to help craft the project to ensure success!

Thanks for Watching!!!