Adaptive Order Radau Methods

Shreyas Ekanathan

Mentor: Dr. Chris Rackauckas

MIT PRIMES October Conference
10/12/2024

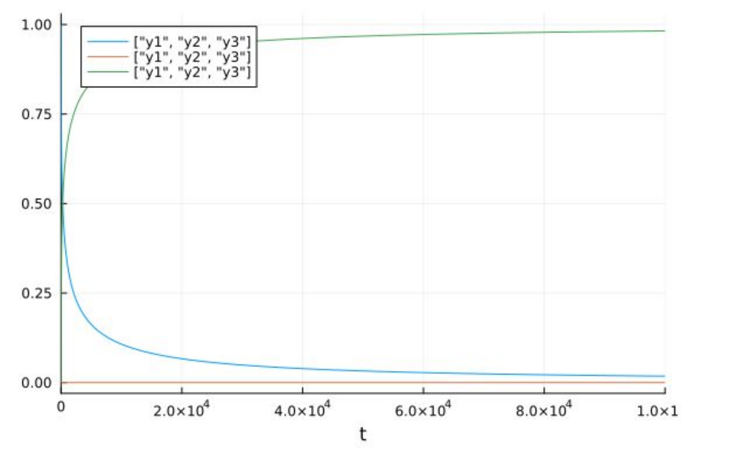
Table of Contents

Problem

- Stiff Ordinary Differential Equations

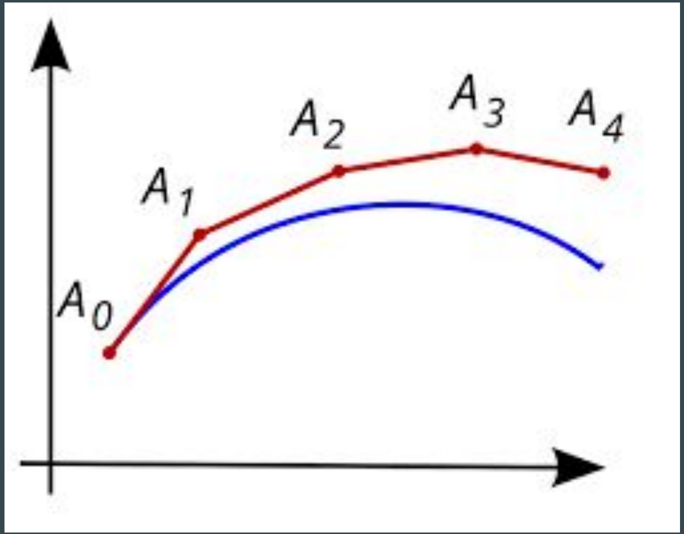
$$
\begin{array}{cccl}
A & \xrightarrow{0.04} & B & \text{(slow)} \\
B+B & \xrightarrow{3\cdot 10^7} & C+B & \text{(very fast)} \\
B+C & \xrightarrow{10^4} & A+C & \text{(fast)}
\end{array}
$$

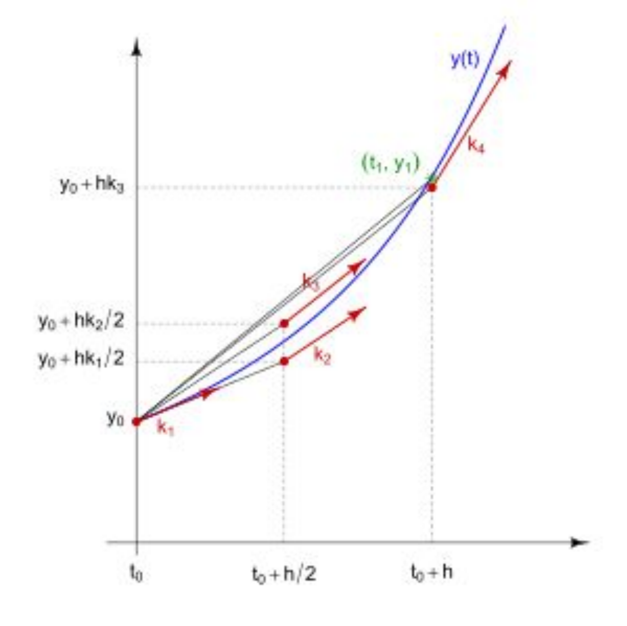
$$
\begin{array}{lll}
\text{A:} & y_1' = -0.04y_1 + 10^4 y_2 y_3 & y_1(0) = 1 \\
\text{B:} & y_2' = \quad 0.04y_1 - 10^4 y_2 y_3 - 3\cdot 10^7 y_2^2 & y_2(0) = 0 \\
\text{C:} & y_3' = \qquad\qquad\qquad\qquad\qquad 3\cdot 10^7 y_2^2 & y_3(0) = 0.
\end{array}
$$



What Exactly is a Runge-Kutta Method?

- Generalized collocation methods to numerically approximate solutions to first order differential equations





Mathematical Formulation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t)$$

$$y(t_0) = y_0$$

Approximate solution at time $t = t_0 + dt$ as:

$$y(t) = y_0 + dt\Sigma_{i=1}^{s} b_i k_i$$

Where k_p is defined as:

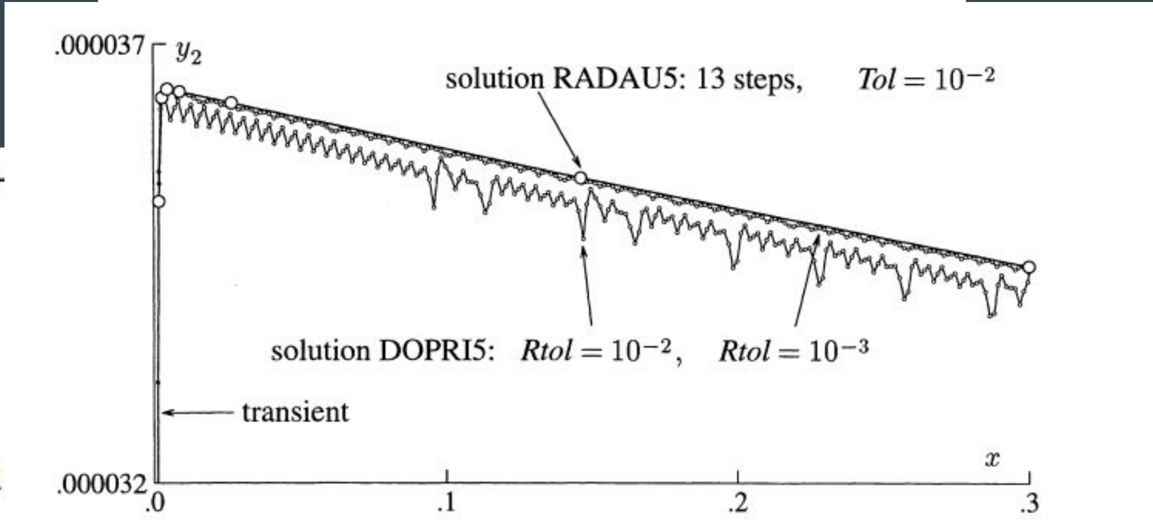
$$k_p = f(y_n + \Sigma_{i=1}^{p-1} A_{pi} k_i, t_n + c_p dt)$$

$$\begin{array}{c|cccc}
c_1 & A_{11} & A_{12} & \cdots & A_{1,s} \\
c_2 & A_{21} & A_{22} & \cdots & A_{2,s} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
1 & A_{s,1} & A_{s,2} & \cdots & A_{s,s} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}$$

$$\begin{array}{c|ccc}
\frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{350} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\
\frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{350} & \frac{-2-3\sqrt{6}}{225} \\
1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\
\hline
 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9}
\end{array}$$

Benefits of Radau

$$y(t_0 + dt) = y_0 + dt \cdot f(y_0, t_0 + dt)$$



- A-

  $dt \to 0$

- L-
  - A-stable and robust to oscillation
  - $\lim_{w \to \infty} g(w) = 0$
  - Important for solving stiff equations and DAEs

Building a Radau Method

- Tableau: Evaluate constants for our method.
- Time-stepping: Simulate one time-step of the method, determining the value of u at $t = t_0+dt$

Linear System

- When performing a time-step in a Radau method, we need to evaluate the solution to a costly linear system involving A^{-1}.

$$(I - dt \cdot A)\Delta z = -z + k \cdot dt \cdot A$$

- Optimize: Find a rigid structure for A^{-1}!
- Goal: Find a transformation matrix T that sends A^{-1} into a nice form.

What can we do better than a naive implementation?

Transformation of the solver to use the complex eigenbasis to simplify the most costly part of the computation!

This means that a solver for real-valued ODEs can be accelerated by using computations in the complex plane!

A^{-1}

- A^{-1} is a square matrix that has 1 real eigenvalue and several complex conjugate pairs of eigenvalues.

$$\begin{pmatrix} 0.196815 & -0.0655354 & 0.023771 \\ 0.394424 & 0.292073 & -0.0415488 \\ 0.376403 & 0.512486 & 0.111111 \end{pmatrix}$$

$$\begin{matrix} 2.6811 - 3.05043i \\ 2.6811 + 3.05043i \\ 3.63783 \end{matrix}$$

$$\begin{pmatrix} -0.128458 + 0.0273087i & -0.128458 - 0.0273087i & 0.0912324 \\ 0.185636 - 0.348249i & 0.185636 + 0.348249i & 0.241718 \\ 0.909404 & 0.909404 & 0.966048 \end{pmatrix}$$

Transformation Matrix

Take a basis (r, u, v)!

$$\begin{pmatrix} -0.128458 + 0.0273087i & -0.128458 - 0.0273087i & 0.0912324 \\ 0.185636 - 0.348249i & 0.185636 + 0.348249i & 0.241718 \\ 0.909404 & 0.909404 & 0.966048 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 0.0912324 & -0.128458 & 0.0273087 \\ 0.241718 & 0.185636 & 0.348249 \\ 0.966048 & 0.909404 & 0.909404 \end{pmatrix}$$

$$\begin{pmatrix} 0.196815 & -0.0655354 & 0.023771 \\ 0.394424 & 0.292073 & -0.0415488 \\ 0.376403 & 0.512486 & 0.111111 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} 3.637 & 0 & 0 \\ 0 & 2.6811 & -3.0504 \\ 0 & 3.0504 & 2.6811 \end{pmatrix}$$

Solving the System

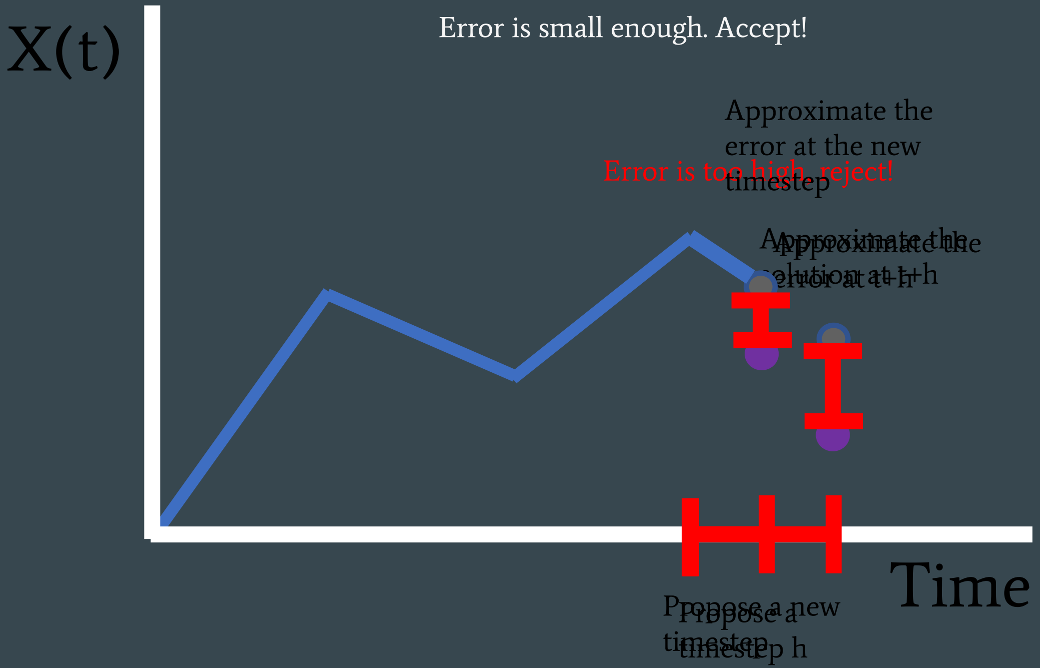
- Now, instead of explicitly multiplying to solve the linear system, we can utilize our rigid structure of A^{-1}
- Each block is represented as:

$$\begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}$$

- Multiplying by the function evaluations gives:

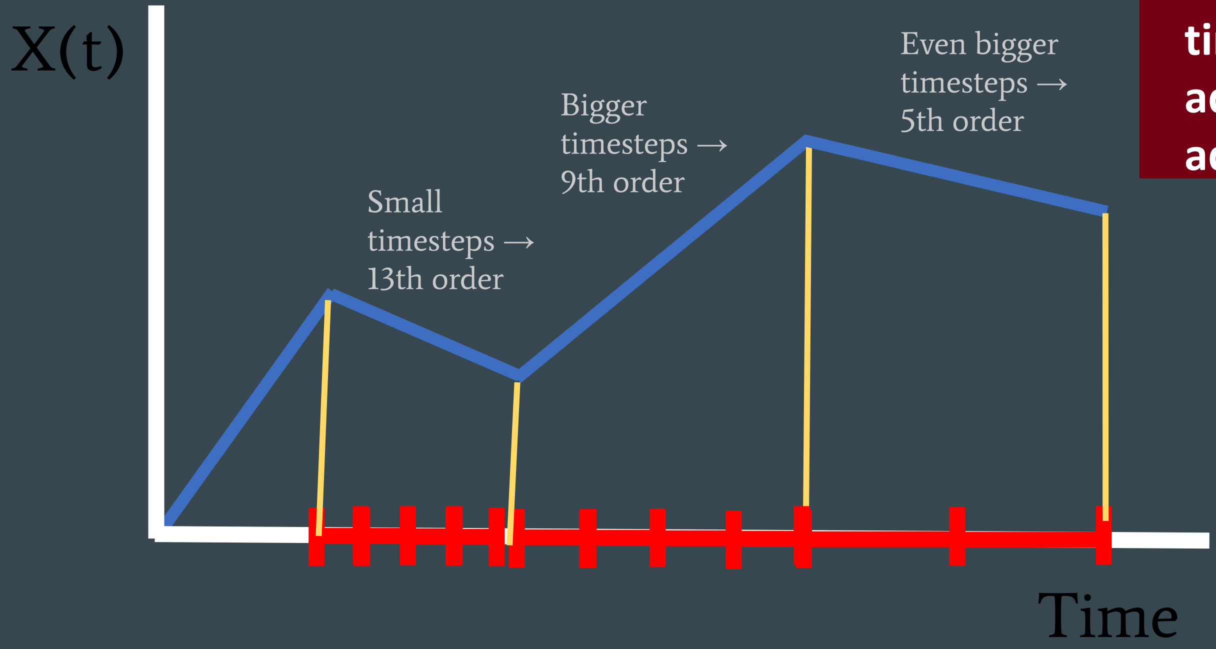
$$\texttt{rhs[i]} = \texttt{fw[i]} - \alpha[i] * fw[i] + \beta[i] * fw[i+1]$$
$$\texttt{rhs[i+1]} = \texttt{fw[i+1]} - \beta[i] * fw[i] - \alpha[i] * fw[i+1]$$

Radau's Step Size Adaptivity



Idea: use different orders of Radau to estimate error and adapt steps on the fly

Radau's Order Adaptivity



Idea: high order methods are only more efficient for smaller time steps, so mix order adaptivity with time step adaptivity

Building on Existing Work

- No hard-coded coefficients
  - The methods can generate the coefficients (A, b, c, T, etc) on the fly.
- Full Adaptivity
  - Existing methods are constrained to orders 5, 9, and 13, but our implementation will eventually span 1 - as high as you want!

Tableau Computation

```julia
tmp = Vector{BigFloat}(undef, num_stages - 1)
for i in 1:(num_stages - 1)
    tmp[i] = 0
end
tmp2 = Vector{BigFloat}(undef, num_stages + 1)
for i in 1:(num_stages + 1)
   tmp2[i]=(-1)^(num_stages + 1 - i) * binomial(num_stages , num_stages + 1 - i)
end
radau_p = Polynomial{BigFloat}([tmp; tmp2])
for i in 1:(num_stages - 1)
    radau_p = derivative(radau_p)
end
c = real(roots(radau_p))
c[num_stages] = 1
c_powers = Matrix{BigFloat}(undef, num_stages, num_stages)
for i in 1 : num_stages
    for j in 1 : num_stages
        c_powers[i,j] = c[i]^(j - 1)
    end
end
inverse_c_powers = inv(c_powers)
c_q = Matrix{BigFloat}(undef, num_stages, num_stages)
for i in 1 : num_stages
    for j in 1 : num_stages
        c_q[i,j] = c[i]^(j) / j
    end
end
a = c_q * inverse_c_powers
a_inverse = inv(a)
b = Vector{BigFloat}(undef, num_stages)
for i in 1 : num_stages
    b[i] = a[num_stages, i]
end
```

```
julia> c
9-element Vector{BigFloat}:
 0.01777991514736345181320510103767906126648839823850043366560783167875924759076611
 0.09132360789979395600374145807454135310704047574456766876687017263479530047321594
 0.2143084793956307583575412675816703226744297175224765908137326191408580981556082
 0.3719321645832723024308539604826292446681000633774806778676385412497303512437736
 0.5451866848034266490322722299532130551329800560537052249374829413005927638033035
 0.7131752428555694810513137602509073414468837909454265589730704929652137979293597
 0.8556337429578544285147814797717850302864781605395751604095670933496281720817967
 0.9553660447100301492668789781416922384764228669900323904483832488568928590110162
 1.0
```

```
julia> a
9×9 Matrix{BigFloat}:
 0.0227884  -0.00858964   0.00645103  -0.00525753   0.00438883  -0.00365122   0.00294049  -0.00214927   0.000858843
 0.048908    0.0507021   -0.0135238    0.00920937  -0.00715571   0.00574725  -0.00454258   0.00328816  -0.00130907
 0.0437428   0.108302     0.0729196   -0.0168799    0.0107046   -0.00790195   0.00599141  -0.00424802   0.00167815
 0.0462492   0.0965607    0.154299     0.0867194   -0.0184516    0.0110367   -0.00767328   0.00522822  -0.00203591
 0.0448344   0.102307     0.138218     0.181264     0.0904336   -0.0180851    0.0101934   -0.00640527   0.00242717
 0.0456588   0.0991455    0.145747     0.163648     0.185945     0.0836133   -0.0158099    0.00813825  -0.00291047
 0.0452006   0.100854     0.141942     0.171189     0.169783     0.167768     0.067079    -0.0117922    0.00360925
 0.0454165   0.10006      0.143653     0.168019     0.175561     0.155886     0.128894     0.0428108   -0.00493457
 0.0453573   0.100277     0.143193     0.168847     0.174137     0.158422     0.123595     0.073827     0.0123457
```

```
julia> b
9-element Vector{BigFloat}:
 0.04535725246164145850644674920795090340044910382256608423236014041761080722192241
 0.1002766490122759787105825452084585374837100645139951717640162083073491265415234
 0.1431933481786155855733528188760054645853142508187793084336149756313973617687846
 0.1688469834879647929018621198957803825501226794932222463313217885924663669033451
 0.1741365013864832970359955155930094385249082238316676536703664502234506531674301
 0.1584218878352189891690004248211828904282803901135082813370203191442722378911759
 0.1235946891022965261806198974844998620884284109101835335540040960997860641428332
 0.07382700952315769290979424990076684192644119748373204166495034257132170740227098
 0.01234567901234567901234567901234567901234567901234567901234567901234567464013971
```

Acknowledgements

I would like to thank:

- My mentor, Dr. Chris Rackauckas
- MIT PRIMES
- My family

Thank You!

•••

Questions?

References

The current state of the art work has been done by Ernst Hairer, who developed most of the techniques used in this presentation.

Much of the theory is cited from *Solving Ordinary Differential Equations II*, by Ernst Hairer and Gerhard Wanner.

In addition, the paper Stiff differential equations solved by Radau methods was very helpful.

Hairer's scripts can be found online at here, while my scripts can be found here.