RK-Opt User Manual

Release 1.0.0

David I. Ketcheson, Matteo Parsani, Aron J. Ahmadia, and Hendrik

Nov 02, 2020

CONTENTS

$RK - Opt$ is a software package for designing numerical ODE solvers with coefficients optimally chosen to provide desired properties. It is available from https://github.com/ketch/RK-Opt, with documentation at http://rk-opt.readthedocs.io/en/latest/. The primary focus of the package is on the design of Runge-Kutta methods (including both stability polynomials and full Butcher tableaus), but some routines for designing other classes of methods such as multistep Runge-Kutta and general linear methods are also included. Supported objective functions include the principal error norm and the SSP coefficient. Supported constraints include stability polynomial coefficients, low-storage formulations, and structural constraints (explicit, diagonally implicit, etc.) RK-Opt uses CVX as well as MATLAB's Optimization Toolbox and Global Optimization Toolbox.

The RK-Opt package consists of the following packages:

- ***RK-coeff-opt*:** Find optimal Runge-Kutta method coefficients for a prescribed order of accuracy and number of stages. The objective function can be chosen as either the **SSP coefficient** or the **leading truncation error coefficient**. The method may be constrained to have a **low-storage implementation** and/or a prescribed **stability polynomial**. Implicit and diagonally implicit methods can also be optimized.
- ***am_radius-opt*:** Find stability functions with optimal radius of absolute monotonicity. This includes codes for optimizing stability functions of multistep, multistage methods and even methods with downwinding. The optimization of rational functions is experimental.
- ***polyopt*:** Given a spectrum (typically corresponding to a spatial semi-discretization of a PDE), find an optimal stability polynomial in terms of its coefficients. These polynomial coefficients can then be used as input to **RK-coeff-opt** to find a corresponding Runge-Kutta method.
- ***RKtools*:** Some general utilities for analyzing Runge-Kutta methods.

RK-Opt has been developed by David Ketcheson (primary developer and maintainer), Matteo Parsani, Aron Ahmadia, Zack Grant, and Hendrik Ranocha.

RK-Opt is released under a modified BSD License. If you use RK-Opt in published work, please cite it; see *Citing RK-Opt*.

CHAPTER

ONE

RK-OPT

1.1 Automated design of Runge-Kutta methods

An s-stage Runge-Kutta method has roughly s^2 coefficients (roughly $s^2/2$ for explicit methods), which can be chosen so as to provide high accuracy, stability, or other properties. Historically, most interest in Runge-Kutta methods has focused on methods using the minimum number of stages for a given order of accuracy. However, in the past few decades there has been increasing recognition that using *extra* stages can be worthwhile in order to improve other method properties. Some areas where this is particularly useful are in the enhancement of linear and nonlinear stability properties, the reduction of storage requirements, and the design of embedded pairs. Methods with dozens or even hundreds of stages are not unheard of.

At the same time, most existing Runge-Kutta methods have been designed by hand, by researchers laboriously solving the order conditions. When using extra stages, the number of available parameters makes the selection of a near-optimal choice by hand impossible, and one resorts to computational optimization. This leads to a different paradigm of numerical method design, in which we use sophisticated numerical (optimization) algorithms to design sophisticated numerical (integration) algorithms. It can be expected that this trend will accelerate in the future, and perhaps one day simple manually-constructed algorithms will be the exception.

RK-Opt contains a set of tools for designing Runge-Kutta methods in this paradigm. It has been constructed mostly in the direct line of our research, but we have made some effort to help others easily understand and use it. We hope that you find it useful, and that you will contribute any enhancements you may develop back to the project by sending us a pull request on GitHub.

1.2 Installation

This section describes how to obtain RK-Opt and test that it is working correctly.

1.2.1 Dependencies

- MATLAB (relatively recent versions; tested with R2018a and later)
- MATLAB Optimization Toolbox
- MATLAB Global Optimization Toolbox
- CVX (http://cvxr.com/cvx/)

1.2.2 Obtaining RK-Opt

- Download: https://github.com/ketch/RK-Opt/
- Or clone:

```
$ git clone https://github.com/ketch/RK-Opt.git
```

After unzipping/cloning, add the subdirectory `RK-Opt/RKtools` to your MATLAB path (see https://www.mathworks.com/help/matlab/ref/addpath.html).

1.3 Testing your installation

You can test your RK-Opt installation by running the MATLAB script `test.m`.

1.3.1 Running the tests

To run the tests, do the following in MATLAB:

```
>> cd /path/to/RK-Opt/
>> test
```

If everything is set up correctly, this will run several tests, and inform you that the tests passed.

1.4 Citing RK-Opt

Are you using RK-Opt in research work to be published? If so, please include explicit mention of our work in your publication. We suggest language such as this:

> "To solve problem (17), we used RK-Opt, a package for the design of numerical ODE solvers [1],[2]"

with the following entry in your bibliography:

> [1] **RK-Opt: A package for the design of numerical ODE solvers**, version X.Y.Z. David I. Ketcheson, Matteo Parsani, Zachary J. Grant, Aron J. Ahmadia, and Hendrik Ranocha, *Journal of Open Source Software*, 5(54):2514 (2020)

with the appropriate version number inserted. It may also be appropriate to cite one of the following:

- if you use the **RK-coeff-opt** package to optimize SSP coefficients, please reference [2].
- If you use the **RK-coeff-opt** package to develop low-storage methods, please reference [3].
- **If you use the RK-coeff-opt package to optimize for accuracy, and/or enforce a given** stability function, please reference [4].
- If you use the **RK-coeff-opt** package to develop general linear methods, please reference [7].
- If you use the **polyopt** package, please reference [5].
- If you use the **am_rad-opt** package, please reference [6].

> [2] **Highly Efficient Strong Stability Preserving Runge-Kutta Methods with Low-Storage Implementations**. David I. Ketcheson, *SIAM Journal on Scientific Computing*, 30(4):2113-2136 (2008)
>
> [3] **Runge–Kutta methods with minimum storage implementations**. David I. Ketcheson, *Journal of Computational Physics*, 229(5):1763-1773 (2010)

[4] **Optimized explicit Runge-Kutta schemes for the spectral difference method applied to wave propagation problems**. Matteo Parsani, David I. Ketcheson, and W. Deconinck, *SIAM Journal on Scientific Computing*, 35(2):A957-A986 (2013)

[5] **Optimal stability polynomials for numerical integration of initial value problems**. David I. Ketcheson and Aron J. Ahmadia, *Communications in Applied Mathematics and Computational Science*, 7(2):247-271 (2012)

[6] **Computation of optimal monotonicity preserving general linear methods**. David I. Ketcheson, *Mathematics of Computation*, 78(267):1497-1513 (2009)

[7] **Strong Stability Preserving Two-step Runge–Kutta Methods.** David I. Ketcheson, Sigal Gottlieb, CB Macdonald, *SIAM Journal on Numerical Analysis*, 2011;49(6):2618 (2011).

Also, please do let us know if you are using this software so we can add your work to our Applications section.

1.5 References

CHAPTER

TWO

REFERENCE

This section contains a compilation of the documentation of each function, organized by subpackage.

2.1 RK-coeff-opt

This subpackage contains routines for finding optimal Runge-Kutta method coefficients, given a prescribed order of accuracy, number of stages, and an objective function. Constraints on the stability polynomial (possibly obtained using **polyopt** or **am_radius-opt**) can optionally be provided.

To run the tests, execute the MATLAB commands ` `results_rkopt = runtests('test_rkopt.m'); table(results_rkopt)` `

Contents

2.1.1 oc_butcher

```
function coneq=oc_butcher(A,b,c,p)
```

Order conditions for Runge-Kutta methods. This version is based on Butcher's approach.

Assumes $p > 1$.

2.1.2 rk_opt

```
function rk = rk_opt(s,p,class,objective,varargin)
```

Find optimal RK and multistep RK methods. The meaning of the arguments is as follows:

- s number of stages.
- k number of steps (1 for RK methods)
- p order of the Runge-Kutta (RK) scheme.
- class: class of method to search. Available classes:
  - 'erk' : Explicit Runge-Kutta methods
  - 'irk' : Implicit Runge-Kutta methods
  - 'dirk' : Diagonally implicit Runge-Kutta methods
  - 'sdirk' : Singly diagonally implicit Runge-Kutta methods
  - '2S', etc. : Low-storage explicit methods; see *Ketcheson, "Runge-Kutta methods with minimum storage implementations". J. Comput. Phys. 229(5):1763 - 1773, 2010*)
  - 'emsrk1/2' : Explicit multistep-Runge-Kutta methods
  - 'imsrk1/2' : Implicit multistep-Runge-Kutta methods
  - 'dimsrk1/2' : Diagonally implicit multistep-Runge-Kutta methods
- objective: objective function ('ssp' = maximize SSP coefficient; 'acc' = minimize leading truncation error coefficient) Accuracy optimization is not currently supported for multistep RK methods
- poly_coeff_ind: index of the polynomial coefficients to constrain (β_j) for $j > p$ (j denotes the index of the stage). The default value is an empty array. Note that one should not include any indices $i \leq p$, since those are determined by the order conditions.
- poly_coeff_val: constrained values of the polynomial coefficients (β_j) for $j > p$ (tall-tree elementary weights). The default value is an empty array.
- startvec: vector of the initial guess ('random' = random approach; 'smart' = smart approach; alternatively, the user can provide the startvec array. By default startvec is initialized with random numbers.
- solveorderconditions: if set to 1, solve the order conditions first before trying to optimize. The default value is 0.
- np: number of processor to use. If np > 1 the MATLAB global optimization toolbox *Multistart* is used. The default value is 1 (just one core).
- num_starting_points: Number of starting points for the global optimization per processor. The default value is 10.

- writeToFile: whether to write to a file. If set to 1 write the RK coefficients to a file called "ERK-p-s.txt". The default value is 1.
- append_time: whether a timestamp should be added to the output file name
- constrain_emb_stability: a vector of complex points where the embedded method should be stable. Sometimes, fmincon cannot find solutions if emb_poly_coeff_ind,emb_poly_coeff_val are given. In these situations, there are a few parameter combinations where it can be advantageous to ask fmincon to directly constraint the value of the embedded stability function at a few points. In general, the existing approach using polyopt and emb_poly_coeff_ind,emb_poly_coeff_val seems to be better for most problems.
- algorithm: which algorithm to use in fmincon: 'sqp','interior-point', or 'active-set'. By default sqp is used.
- suppress_warnings: whether to suppress all warnings

Note: numerical experiments have shown that when the objective function is the minimization of the leading truncation error coefficient, the interior-point algorithm performs much better than the sqp one.

- display: level of display of fmincon solver ('off', 'iter', 'notify' or 'final'). The default value is 'notify'.
- problem_class: class of problems for which the RK is designed ('linear' or 'nonlinear' problems). This option changes the type of order conditions check, i.e. linear or nonlinear order conditions control. The default value is 'nonlinear'.

Note: Only s , p , class and objective are required inputs. All the other arguments are **parameter name - value arguments to the input parser scheme**. Therefore they can be specified in any order.

Example:

```
>> rk=rk_opt(4,3,'erk','acc','num_starting_points',2,'np',1,
↪'solveorderconditions',1)
>> rk=rk_opt(4,3,'erk','acc','num_starting_points',2,'np',1,
↪'solveorderconditions',1,'np',feature('numcores'))
```

The fmincon options are set through the optimset that creates/alters optimization options structure. By default the following ad

- MaxFunEvals = 1000000
- TolCon = 1.e-13
- TolFun = 1.e-13
- TolX = 1.e-13
- MaxIter = 10000
- Diagnostics = off
- DerivativeCheck = off
- GradObj = on, if the objective is set equal to 'ssp'

2.1.3 unpack_lsrk

```
function [A,b,bhat,c,alpha,beta,gamma1,gamma2,gamma3,delta]=unpack_lsrk(X,class)
```

Extracts the coefficient arrays from the optimization vector.

This function also returns the low-storage coefficients.

2.1.4 check_RK_order

```
function p = check_RK_order(A,b,c)
```

Determines order of a RK method, up to sixth order.

For an s-stage method, input A should be a $s \times s$ matrix; b and c should be column vectors of length s.

2.1.5 unpack_msrk

```
function [A,Ahat,b,bhat,D,theta] =  unpack_msrk(X,s,k,class)
```

Extract the coefficient arrays from the optimization vector

2.1.6 errcoeff

```
function D = errcoeff(A,b,c,p)
```

Inputs:

- A, b, c – Butcher tableau
- p – order of accuracy of the method

Computes the norm of the vector of truncation error coefficients for the terms of order $p + 1$: (elementary weight - 1/(density of the tree)/(symmetry of the tree)

For now we just use Butcher's approach. We could alternatively use Albrecht's.

2.1.7 linear_constraints

```
function [Aeq,beq,lb,ub] = linear_constraints(s,class,objective,k)
```

This sets up:

- The linear constraints, corresponding to the consistency conditions $\sum_j b_j = 1$ and $\sum_j a_{ij} = c_j$.
- The upper and lower bounds on the unknowns. These are chosen somewhat arbitrarily, but usually aren't important as long as they're not too restrictive.

2.1.8 set_n

```
function n=set_n(s,class)
```

Set total number of decision variables

2.1.9 order_conditions

```
function tau = order_conditions(x,class,s,p,Aeq,beq)
```

This is just a small wrapper, used when solveorderconditions=1.

2.1.10 oc_ksrk

```
function coneq= oc_ksrk(A,b,D,theta,p)
```

Order conditions for multistep-RK methods.

..warning:

```
Here we assume a certain minimum stage order,
which is necessarily true for methods with
strictly positive abscissae (b>0).
This assumption dramatically reduces the
number of order conditions that must be
considered for high-order methods.
For methods that do not satisfy b>0, this
assumption may be unnecessarily restrictive.
```

2.1.11 write_field(writeFid,name,value)

```
function write_field(writeFid,name,value)
```

Utility function to write a single parameter and value.

2.1.12 oc_albrecht

```
function coneq=oc_albrecht(A,b,c,p)
```

Order conditions for SSP RK methods.

This version is based on Albrecht's approach.

2.1.13 unpack_rk

```
unction [A,b,c,Ahat,bhat,chat]=unpack_rk(X,s,class)
```

Extracts the coefficient arrays from the optimization vector.

The coefficients are stored in a single vector x as:

```
x=[A b' c']
```

A is stored row-by-row.

Low-storage methods are stored in other ways as detailed inline below.

2.1.14 shuosher2butcher

```
function [A,b,c]=shuosher2butcher(alpha,beta);
```

Generate Butcher form of a Runge-Kutta method, given its Shu-Osher or modified Shu-Osher form.

For an m-stage method, α and β should be matrices of dimension $(m+1) \times m$.

2.1.15 nonlinear_constraints

```
function [con,coneq]=nonlinear_constraints(x,class,s,p,objective,poly_coeff_ind,poly_
↪coeff_val,k,emb_poly_coeff_ind,emb_poly_coeff_val,constrain_emb_stability)
```

Impose nonlinear constraints:

- if objective = 'ssp' : both order conditions and absolute monotonicity conditions
- if objective = 'acc' : order conditions

The input arguments are:

- x: vector of the decision variables. See unpack_rk.m for details about the order in which they are stored.
- *class*: class of method to search ('erk' = explicit RK; 'irk' = implicit RK; 'dirk' = diagonally implicit RK; 'sdirk' = singly diagonally implicit RK; '2S', '3S', '2S*', '3S*' = low-storage formulations).
- s:number of stages.
- p: order of the RK scheme.
- *objective*: objective function ('ssp' = maximize SSP coefficient; 'acc' = minimize leading truncation error coefficient).
- *poly_coeff_ind*: index of the polynomial coefficients (β_j) for $j > p$.
- *poly_coeff_val*: values of the polynomial coefficients (β_j) for $j > p$ (tall-tree elementary weights).
- k: Number of steps for multi-step, mlti-stage schemes.
- *emb_poly_coeff_ind*: index of the polynomial coefficients of the embedded scheme (β_j) for $j > p$.
- *emb_poly_coeff_val*: values of the polynomial coefficients of the embedded scheme (β_j) for $j > p$ (tall-tree elementary weights).

The outputs are:

- *con*: inequality constraints, i.e. absolute monotonicity conditions if objective = 'ssp' or nothing if objective = 'acc'
- *coneq*: order conditions plus stability function coefficients constraints (tall-tree elementary weights)

Two forms of the order conditions are implemented: one based on **Butcher's approach**, and one based on **Albrecht's approach**. One or the other may lead to a more tractable optimization problem in some cases, but this has not been explored carefully. The Albrecht order conditions are implemented up to order 9, assuming a certain stage order, while the Butcher order conditions are implemented up to order 9 but do not assume anything about the stage order. Albrecht's approach is used by default.

2.1.16 rk_obj

```
function [r,g]=rk_obj(x,class,s,p,objective)
```

Objective function for RK optimization.

The meaning of the input arguments is as follows:

- x: vector of the unknowns.
- class: class of method to search ('erk' = explicit RK; 'irk' = implicit RK; 'dirk' = diagonally implicit RK; 'sdirk' = singly diagonally implicit RK; '2S', '3S', '2S*', '3S*' = low-storage formulations).
- s:number of stages.
- p: order of the RK scheme.
- objective: objective function ('ssp' = maximize SSP coefficient; 'acc' = minimize leading truncation error coefficient).

The meaning of the output arguments is as follows:

- r: it is a scalar containing the radius of absolute monotonicity if objective = 'ssp' or the value of the leading truncation error coefficient if objective = 'acc'.
- g: a vector containing the gradient of the objective function respect to the unknowns. It is an array with all zero elements except for the last component which is equal to one if objective = 'ssp' or it is an empty array if objective = 'acc'.

2.2 am_radius-opt

Find stability functions with optimal radius of absolute monotonicity. This includes codes for optimizing stability functions of multistep, multistage methods and even methods with downwinding.

Generally, the optimization problem is phrased as a sequence of linear programming feasibility problems. For details, see [Ket09].

The optimization of rational functions is experimental.

Contents

- *radimpfast*
- *Rskp*
- *Rkp_dw*
- *Rkp_imp*
- *Rkp_imp_dw*

2.2.1 multi_R_opt

```
function multi_R = multi_R_opt(k,p,class,varargin)
```

This function is a script to run the routines Rskp, Rkp_dw, Rkp_imp, or Rkp_imp_dw several times with different inputs, in order to construct tables of optimal values like those that appear in [Ket09].

The inputs k, p, and (optionally) s should be vectors containing the numbers of steps, orders of accuracy, and numbers of stages to be considered, respectively. The output includes results for all combinations of values from the input vectors.

The family of methods to be considered is specified in the string 'class'. Valid values are:

- **'skp': find optimal general linear methods (multistep, multistage).** In this case the vector s must be included in the inputs.
- 'kp_imp': find optimal implicit linear multistep methods.
- 'kp_dw': find optimal explicit downwind linear multistep methods.
- 'kp_imp_dw': find optimal implicit downwind linear multistep methods.

2.2.2 Rkp

```
function [R,alpha,beta]=Rkp(k,p)
```

Find the optimal SSP k-step explicit LMM with order of accuracy p.

Inputs:

- k = # of steps
- p = order of accuracy

Outputs:

- R = the SSP coefficient
- alpha, beta = the coefficients of the method

Requires MATLAB's optimization toolbox for the LP solver.

2.2.3 radimpfast

```
function rad=radimpfast(p,q)
```

Compute the radius of absolute monotonicity of the rational function whose numerator has coefficients p and denominator has coefficients q. The coefficients are ordered in ascending powers.

This function is outdated and needs to be fixed.

Uses van de Griend's algorithm [vdGK86], assuming multiplicity one for all roots. Uses high precision arithmetic.

2.2.4 Rskp

```
function [R,gamma]=Rskp(s,k,p)
```

Finds the optimal contractive k-step, s-stage GLM with order of accuracy p for linear problems.

Inputs:

- s = # of stages
- k = # of steps
- p = order of accuracy

Outputs:

- R = threshold factor
- gamma = coefficients of the polynomials

  for k=1, the resulting polynomial is $\sum_{j=0}^{m}(1+z/R)^j$

  For details on the general case, see [Ket09].

This routine requires MATLAB's optimization toolbox for the LP solver.

2.2.5 Rkp_dw

```
function [R,alpha,beta,tbeta]=Rkp_dw(k,p)
```

Finds the optimal SSP k-step explicit LMM with order of accuracy p allowing downwind operators

Inputs:

- k = # of steps
- p = order of accuracy

Outputs:

- R = the SSP coefficient
- alpha, beta, tbeta = the coefficients of the method

The method is given by $u_n = \sum_{j=0}^{k-1}(\alpha[j] + \beta[j]F(u_{n-k+j}) + tbeta[j]tF(u_{n-k+j}))$ where tF(u) is the negated downwind operator.

Depends on MATLAB's optimization toolbox for the LP solver

2.2.6 Rkp_imp

```
function [R,alpha,beta]=Rkp_imp(k,p)
```

Find the optimal SSP k-step implicit LMM with order of accuracy p

Inputs:

- k = # of steps
- p = order of accuracy

Outputs:

- R = the SSP coefficient
- alpha, beta = the coefficients of the method

Depends on MATLAB's optimization toolbox for the LP solver

2.2.7 Rkp_imp_dw

```
function [R,alpha,beta]=Rkp_imp_dw(k,p)
```

Finds the optimal k-step implicit LMM with order of accuracy p allowing downwinding

Inputs:

- k = # of steps
- p = order of accuracy

Outputs:

- R = the SSP coefficient
- alpha, beta, tbeta = the coefficients of the method

Depends on MATLAB's optimization toolbox for the LP solver

2.3 polyopt

Given a spectrum (typically corresponding to a spatial semi-discretization of a PDE), finds an optimal stability polynomial. The polynomial coefficients can then be used as input to $RK-coeff-opt$ to find a corresponding Runge-Kutta method.

This is the implementation of the algorithm described in [KA12]. The code was written by Aron Ahmadia and David Ketcheson.

To run the tests, execute the MATLAB commands ` `results_polyopt = runtests('test_polyopt.m'); table(results_polyopt)` `

Contents

2.3.1 spectrum

```
function lamda = spectrum(name,N,kappa,beta)
```

Return N discretely sampled values from certain sets in the complex plane.

Acceptable values for name:

- 'realaxis': $[-1, 0]$
- 'imagaxis': $[-i, i]$
- 'disk': $z : |z + 1| = 1$
- 'rectangle': $x + iy : -\beta \leq y \leq \beta, -\kappa \leq x \leq 0$
- 'Niegemann-ellipse' and 'Niegemann-circle': See [NDB11]
- 'gap': Spectrum with a gap; see [KA12]

kappa and beta are used only if name == 'rectangle'

2.3.2 opt_poly_bisect

```
function [h,poly_coeff,diag_bisect] = opt_poly_bisect(lam,s,p,basis,varargin)
```

Finds an optimally stable polynomial of degree s and order p for the spectrum lam in the interval (h_min,h_max) to precision eps.

Optional arguments:

solvers: A cell array of cvx solver names that should be used to solve the convex problem in the inner loop. Defaults to {'sdpt3', 'sedumi'}. You can also add 'mosek' and 'gurobi' if you have obtained (free academic) licences for these and installed them in cvx.

lam_func: A function used to generate the appropriate spectrum at each bisection step, instead of using a fixed (scaled) spectrum. Used for instance to find the longest rectangle of a fixed height (see Figure 10 of the CAMCoS paper).

Examples:

- To find negative real axis inclusion:

```
lam = spectrum('realaxis',500);
s = 10; p = 2;
[h,poly_coeff] = opt_poly_bisect(lam,s,p,'chebyshev')
```

- To reproduce figure 10 of [KA12]

```
lam_func = @(kappa) spectrum('rectangle',100,kappa,10)
[h,poly_coeff] = opt_poly_bisect(lam,20,1,'chebyshev','lam_func',lam_func)
plotstabreg_func(poly_coeff,[1])
```

2.4 RKtools

Some general utilities for analyzing Runge-Kutta methods.

Some of the routines expect as input a structured array rk. This structure must have the fields A, b, c, containing its Butcher coefficients. Optionally, it may represent an additive Runge-Kutta method or an embedded pair in which case it should also have $Ahat$, $bhat$, $chat$ containing the coefficients of the secondary method.

Contents

2.4.1 internal_stab_explicit_butcher

```
function [stability] = internal_stab_explicit_butcher(A,b,c,spectrum,one_step_dt,p)
```

This function computes and plots both intermediate and one-step internal stability vector of an explicit Runge-Kutta scheme given its Butcher tableau.

Note that for an explicit Runge-Kutta scheme the stability functions are polynomials in the complex variable z.

Construct the intermediate stability functions psi_j (where j is the index of the stage).

Note that for an explicit scheme the intermediate stability polynomial associated to the first stage is always 1, i.e. psi_1 = 1. Therefore we just compute and plot the remaining (s-1) intermediate stability polynomials plus the one-step stability polynomial of the Runge-Kuatta method.

2.4.2 plotstabreg_func

```
function [contour_matrix] = plotstabreg_func(p,q,bounds,ls,lw)
```

plot the absolute stability region of a one-step method, given the stability function

Inputs:

- p: coefficients of the numerator of the stability function
- q: coefficients of the denominator of the stability function

if q is omitted, it is assumed that the function is a polynomial

Remaining inputs are optional:

- bounds: bounds for region to compute and plot (default [-9 1 -5 5])
- ls: line style (default '-r')
- lw: line width (default 2)

2.4.3 plotstabreg

```
function [contour_matrix] = plotstabreg(rk,plotbounds,ls,lw)
```

Plots the absolute stability region of a Runge-Kutta method, given the Butcher array

2.4.4 optimal_shuosher_form

```
function [v,alpha,beta] = optimal_shuosher_form(A,b,c)
```

2.4.5 L2_timestep_poly

```
function c = L2_timestep_poly(sdisc,p,q,eps,tol)
```

Find the absolutely timestep for a given combination of linear spatial discretization and stability function.

Also (optionally) plots the region of absolute stability and the eigenvalues.

The timestep is determined to within accuracy eps (default 10^-4).

The spectral stability condition is checked to within tol (default 10^-13).

2.4.6 semispectrum

```
function L = semispectrum(method,order,doplot,nx,cfl)
```

Plot spectra of various semi-discretizations of the advection equation

Current choices for method:

- 'fourier': Fourier spectral method
- 'chebyshev': Chebyshev spectral method
- 'updiff': Upwind difference operators (linearized WENO)
- 'DG': Discontinuous Galerkin method

The value of order matters only for the 'updiff' and 'DG' methods and selects the order of accuracy in those cases.

2.4.7 rk_stabfun

```
function [p,q] = rk_stabfun(rk)
```

Outputs the stability function of a RK method. The Butcher coefficients should be stored in rk.A, rk.b.

p contains the coefficients of the numerator

q contains the coefficients of the denominator

$$\phi(z) = \frac{\sum_j p_j z^j}{\sum_j q_j z^j} = \frac{\det(I - z(A + eb^T))}{\det(I - zA)}.$$

2.4.8 am_radius

```
function r = am_radius(A,b,c,eps,rmax)
```

Evaluates the Radius of absolute monotonicity of a Runge-Kutta method, given the Butcher array.

For an m-stage method, A should be an $m \times m$ matrix and b and c should be column vectors of length m.

Accuracy can be changed by modifying the value of eps (default 10^{-10}) Methods with very large radii of a.m. (>50) will require the default value of rmax to be increased.

The radius of absolute monotonicity is the largest value of r such that

\begin{align*} K(I+rA)^{-1} \ge & 0 \\ rK(I+rA)^{-1}e_m \le & e_{m+1} \end{align*}

where $$ K = \left(\begin{array}{c} A \\ b^T \end{array}\right) $$

CHAPTER

THREE

CONTRIBUTING

If you wish to contribute, we recommend that you fork the RK-Opt GitHub repository, implement your additions, and issue a pull request. You may also simply e-mail a patch to us.

BIBLIOGRAPHY

[Ket09] David I. Ketcheson. Computation of optimal monotonicity preserving general linear methods. *Mathematics of Computation*, 78(267):1497–1513, September 2009.

[KA12] David I. Ketcheson and Aron J. Ahmadia. Optimal stability polynomials for numerical integration of initial value problems. *Communications in Applied Mathematics and Computational Science*, 7(2):247–271, 2012.

[NDB11] Jens Niegemann, Richard Diehl, and Kurt Busch. Efficient low-storage Runge–Kutta schemes with optimized stability regions. *Journal of Computational Physics*, 231(2):372–364, September 2011.

[vdGK86] J A van de Griend and J. F. B. M. Kraaijevanger. Absolute Monotonicity of Rational Functions Occurring in the Numerical Solution of Initial Value Problems. *Numerische Mathematik*, 49:413–424, 1986.