Evolutionary Optimization of Runge-Kutta Coefficients for Specific Differential Equations

THESIS

Presented to the Faculty of the Department of Physics and Astronomy
in Partial Fulfillment of the Major Requirements
for the Degree of

BACHELOR OF SCIENCE IN
PHYSICS

Benjamin Kulas

May 2019

© 2019 Middle Tennessee State University
All rights reserved.

The author hereby grants to MTSU permission to reproduce
and to distribute publicly paper and electronic
copies of this thesis document in whole or in part
in any medium now known or hereafter created.

Evolutionary Optimization of Runge-Kutta Coefficients for Specific Differential Equations

Benjamin Kulas

Signature of Author:

_____

Department of Physics and Astronomy
May 2019

Certified by:

_____

Dr. Eric Klumpe
Professor of Physics & Astronomy
Thesis Supervisor

Accepted by:

_____

Dr. Ronald Henderson
Professor of Physics & Astronomy
Chair, Physics & Astronomy

ABSTRACT

Methods of numerical integration in the Runge-Kutta family are parametrized by several arrays of coefficients used in the integration process. These coefficients can take any value and still form a valid integration method, so long as they are properly normalized. This makes them well-suited to optimization by evolutionary algorithms. A program named deltaRK is introduced which uses an evolutionary algorithm to produce an integrator specialized for simulating a single differential equation. deltaRK currently only supports differential equations with an associated conserved quantity such as energy or angular momentum. Integrators produced by deltaRK are more accurate on the trained system than standard Runge-Kutta integrators with no loss in speed.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. Computer Modeling

Modeling, including computer modeling, is a powerful tool for physicists to understand the universe. Programming a computer to simulate a physical system using what is known about that system is a convenient and useful way of learning about phenomena which are difficult or impossible to observe directly, but for which a solid theoretical basis exists. Computer modeling is used in nearly all branches of physics, including fluid dynamics, solid-state physics, and particle physics, but one discipline in which it is very prominently used is astrophysics.

By the nature of astrophysics, we cannot perform experiments on our objects of study, and for all but a handful of celestial objects, our ability to gather data is limited to the power of our telescopes. For example, we have only recently become able to discover exoplanets which are *not* gas giants. Hence we turn to simulation, and in particular numerical integration. Numerical integration is a method of approximating the solution to a differential equation, which essentially works by splitting the independent variable into discrete chunks rather than continuous values. It is especially useful for differential equations which have no closed-form solution. This work will consider only numerical integration with time as the independent variable, but this restriction is not universal.

There are many methods for numerical integration, or integrators, all of which have the effect of advancing state variables such as position and velocity forward in time by one discrete time step. The error associated with this step, known as truncation

error, is larger for larger time steps, but some integrators have smaller error than others for a given time step. As such, an area of applied mathematics has come into existence which concerns itself with developing integrators which minimize truncation error. It should be noted that truncation error is distinct from the round-off error that results from storing numbers in a digital system.

B. Runge-Kutta Methods

One set of integrators is called the Runge-Kutta methods. This group includes many of the most commonly-known integrators, such as Euler's method, Heun's method, and the midpoint method. The integrator most commonly known as 'Runge-Kutta' was developed by German mathematicians Carl Runge and Martin Kutta circa 1900. All the information needed to apply a specific Runge-Kutta method can be represented in a graphical format called a Butcher tableau [1].

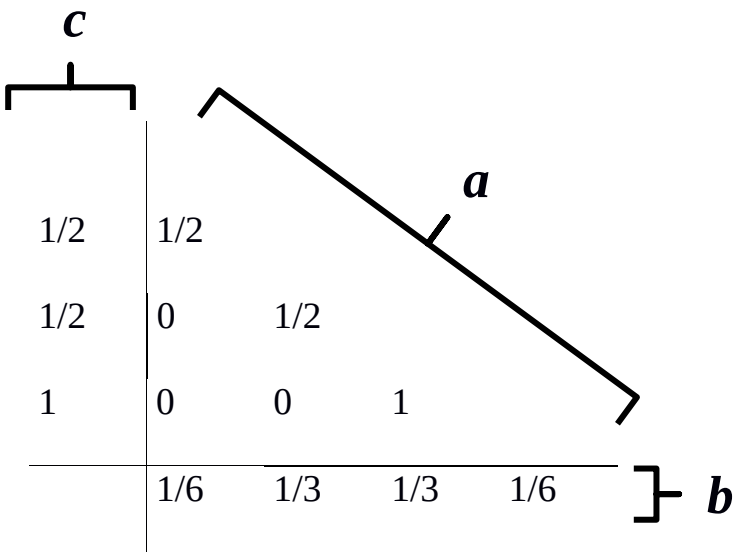

TABLE 1: The Butcher tableau representing the original fourth-order Runge-Kutta method. Blank spaces are elided zeros.

The Butcher tableau is composed of three elements. The large triangular matrix is ***a***, or the Runge-Kutta matrix. The row vector along the bottom is ***b***, also called the weights, and the column vector on the left is ***c***, also called the nodes. A single Runge-Kutta method is parametrized by these three sets of coefficients [1]. These coefficients must obey two normalization conditions. The sum of each row in ***a*** must equal the corresponding element of ***c***, and the sum of ***b*** must equal 1. That is,

$$\sum_j a_{ij} = c_i \text{ and } \sum_i b_i = 1.$$

Any set of coefficients that satisfies these two conditions represents a valid Runge-Kutta method.

C. Evolutionary Algorithms

Evolutionary algorithms are a form of heuristic optimization which apply evolution through artificial selection on an array of potential solutions to a problem. The algorithm is analogous to real-life artificial selection. From a starting population, the algorithm chooses the best candidates using some criterion, called a *fitness function.* It then combines the candidates such that their desirable properties are preserved, analogous to reproduction, and deletes the less-fit individuals. The new population then undergoes the same process [2]. This iterative algorithm is a form of machine learning which, given sufficient time, can develop unexpected and creative solutions to problems. While evolutionary algorithms are most often applied to engineering problems such as the development of walking robots, they can be applied to any problem involving optimizing the inputs to a function. That function simply becomes the fitness function, and potential inputs become the population. Evolutionary algorithms are best suited to high-dimensional optimization problems; that is, problems with many independent variables [2]. However, there are tradeoffs to be considered. By choosing an evolutionary algorithm over more mathematically rigorous means, such as gradient descent, one sacrifices reproducibility and the guarantee of a good solution for improved speed and the ability to optimize many variables simultaneously.

D. Evolving Integrators

Given that evolutionary algorithms are best used to optimize systems of many variables, it follows that these algorithms could optimize the coefficients of a Runge-Kutta integrator. To this end, Martino and Nicosia created EVO-RUNGE-KUTTA [5], which uses a genetic algorithm to optimize for numerical stability, i.e. the ability to simulate chaotic systems without losing accuracy. This created integrators with favorable overall properties. Conversely, my project optimizes for the ability to simulate a single target differential equation without losing accuracy, creating highly specialized integrators for one application.

II. DELTARK

A. Concept

I developed a Python program, called deltaRK, so named because it modifies Runge-Kutta integrators. deltaRK uses an evolutionary algorithm to optimize the coefficients of a Runge-Kutta integrator for integrating a certain differential equation. The program generates a random initial population of Butcher tableaux of a given size, then applies one of several evolutionary algorithms to the population. deltaRK features a generalized Runge-Kutta integrator for second-order differential equations, support for storing and retrieving tableaux in the commonly-used JSON data interchange format, and a modular, structured configuration mechanism.

B. Supported Evolutionary Algorithms

There are currently three evolutionary algorithms available in deltaRK: simple, mu comma lambda (μ , λ), and mu plus lambda $(\mu + \lambda)$. In the following descriptions, 'best' and 'worst' refer to evaluation via the fitness function, and 'individual' refers to a single candidate integrator.

The simple algorithm first replaces the worst individuals in the population with clones of the best individuals. This increases the overall fitness of the population. It then varies the population by applying a probability to either combine two individuals (crossover) or modify one individual randomly (mutation). This process repeats for each generation [3].

The remaining two algorithms, (μ , λ) and $(\mu + \lambda)$, each take two integer parameters, μ and λ. Both start by varying the entire population, only applying crossover *or* mutation, never both, to an individual. This yields a pool of offspring of size λ. The (μ , λ) algorithm then selects μ of the best individuals from the offspring, while $(\mu + \lambda)$ selects from both the offspring and the original population. This process repeats for each generation [3].

C. Fitness Functions

Evaluating the performance of an integrator is not trivial. There are many potential metrics for accuracy and robustness, much like an athlete's strength, speed, and stamina are all separate but desirable qualities. Any of these metrics can be used as a fitness function, and as such several fitness functions are available in deltaRK.

The deltaRK fitness function which has produced the most useful results when tested against systems with known exact solutions is one which calculates the standard deviation over time of a conserved quantity, named `ConsStdev`. That is, at every point during the simulation, the function calculates the value of the conserved quantity; then when the run is finished, it calculates the standard deviation of those stored values. A perfect simulation would of course have zero variation in any conserved quantity, so a lower standard deviation implies being closer to reality and yields a higher fitness score.

When the conserved quantity provided is energy, deltaRK will produce *pseudo-symplectic* integrators, so named because the energy error of true symplectic integrators

is bounded [9]. One drawback is that this fitness function applies only to systems which *have* a conserved quantity, i.e. closed systems. Fortunately, many physical systems conserve energy or angular momentum, including n-body systems where several objects all interact with each other.

Another fitness function is `PseudoOrder`, which takes an array of time steps as a parameter. It then runs an entire simulation for each time step and curve fits

$$\mathrm{ConsStdev}(\Delta t) = a(\Delta t)^b,$$

where $\mathrm{ConsStdev}(\Delta t)$ represents evaluating the given tableau using Δt as the time step. *b* then becomes the fitness value. This process does not calculate the actual order of the integrator, but it does yield a number which governs how error changes as the time step changes, hence *pseudo-order*.

D. Robustness

Yet another desirable quality for integrators is robustness, a measure of an integrator's ability to use a larger time step without introducing much additional error. A larger time step means fewer steps taken overall and thus a simulation that takes less real time. It is possible to use deltaRK to create integrators which retain accuracy with very large time steps. Just like real evolution, the population will adapt to its environment. Thus, by changing the training parameters so that Δt is large, the integrators produced will naturally be able to handle large time steps. There is of course a limit to the size of the time step. For periodic systems, e.g. planets orbiting a star,

once the time step comes within an order of magnitude of the period of motion, even true symplectic integrators struggle to remain stable.

E. Performance

In theory, deltaRK supports any number of objects in any number of dimensions with any equation of motion. However, the time required to advance the simulation by one time step increases with the number of objects, since the acceleration and energy functions become slower with every object and dimension added.

For a simple system such as an undriven undamped pendulum, both acceleration and energy are simple formulae which can be calculated in constant time. But for gravitational interaction between N objects, computing the acceleration on every object and computing the total energy are both $O(N^2)$ operations; that is, the time taken to compute these two quantities increases proportionally with N^2. This is because the energy between every possible pair of objects must be calculated. For example, consider adding a tenth object to a 9-body system. N increases by a factor of 10/9 or 11%, while the number of calculations to compute acceleration and energy increases by a factor of 90/72 or 25%. Since there are many steps in a single n-body simulation, many simulations in a single generation of evolution, and many generations in the overall evolutionary algorithm, this small slowdown will be greatly magnified.

III. RESULTS

A. Comparison to Other Integrators

Integrators produced by deltaRK accumulate a smaller energy error on the trained system than the classic Runge-Kutta method. Unlike truly symplectic integrators, however, deltaRK's pseudo-symplectic integrators still accumulate error over time. This is because the evolutionary algorithm is not an exact process and is governed mostly by random chance, making it statistically impossible to produce a perfect symplectic integrator.

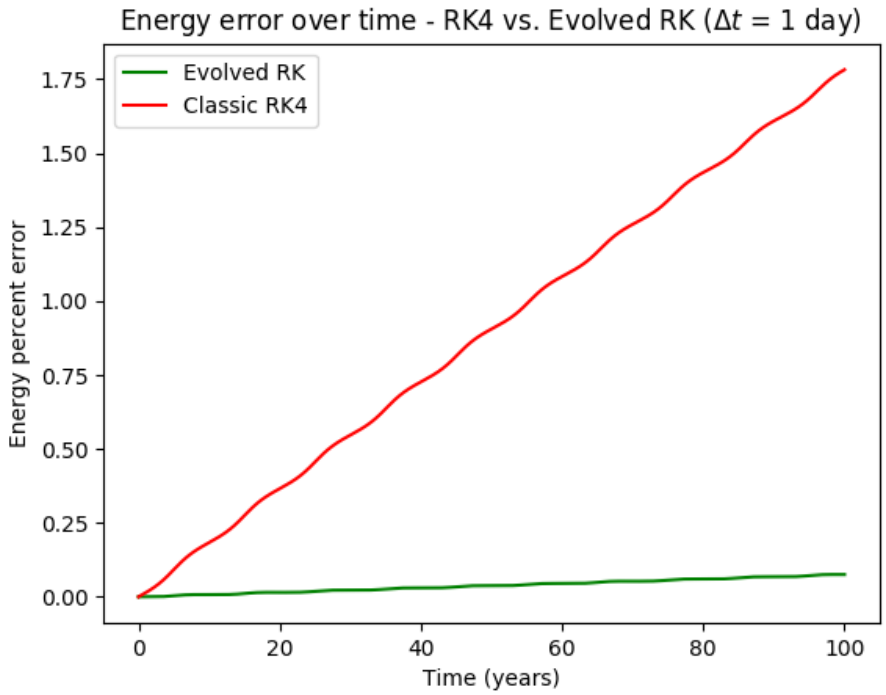

FIGURE 1: Energy drift for both the classic Runge-Kutta method and a pseudo-symplectic integrator produced by deltaRK, using a two-dimensional 2-body simulation of Jupiter and the Sun as a test. The time step is set to 1 day. The evolved integrator was trained on a time period of 1 year, while the final simulation was run for 100 years.

Using the robustness training method detailed in section 2.D, deltaRK also produced integrators which remained stable even with large time steps.

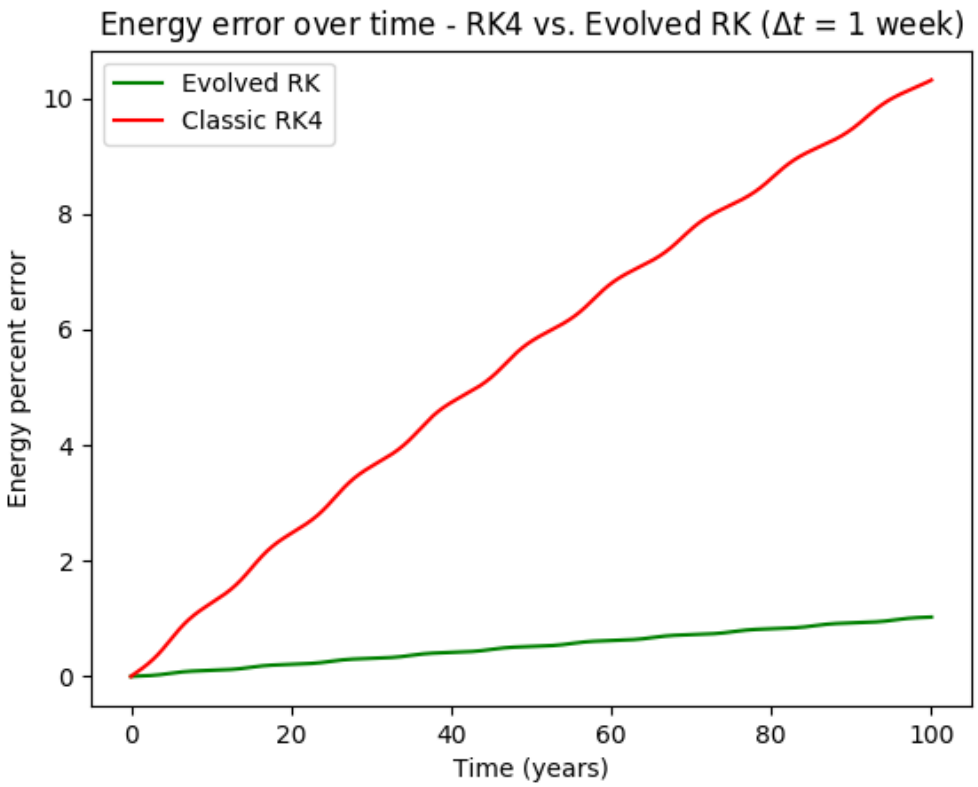

FIGURE 2: Energy drift for both the classic Runge-Kutta method and another pseudo-symplectic integrator produced by deltaRK. The DE and parameters used to produce the integrator are the same as in Figure 1, with the exception that the time step is set to 7 days as opposed to 1.

This robustness training produces integrators which perform better than standard Runge-Kutta at any time step, but which paradoxically perform better with larger time steps.

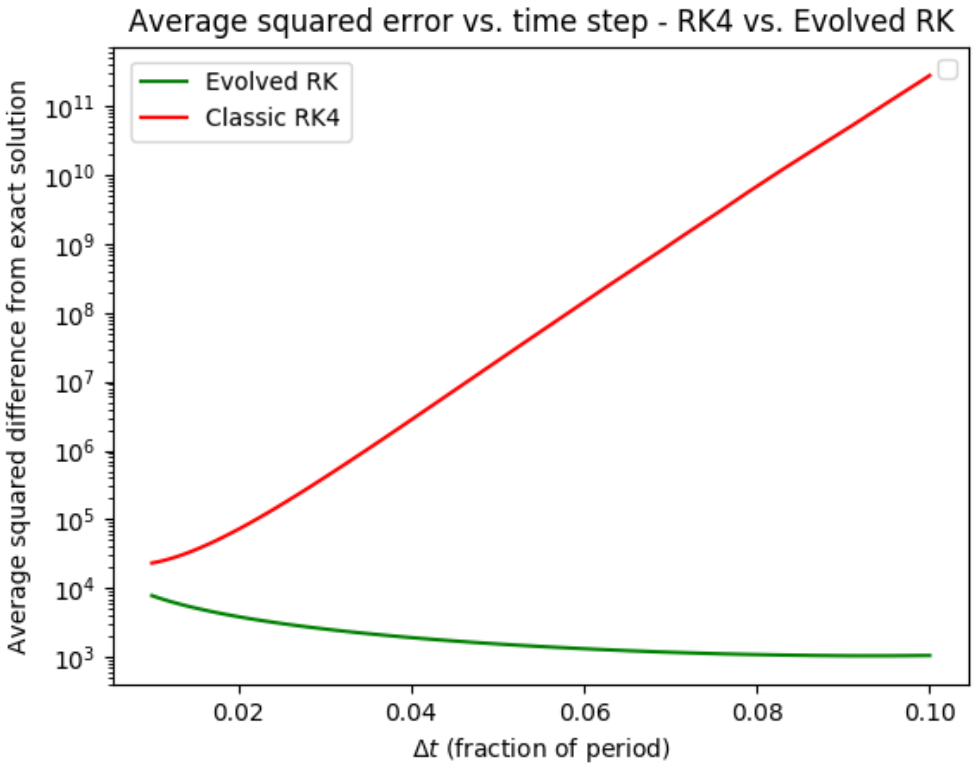

FIGURE 3: The squared error averaged over simulation time for varying time steps. Each data point represents a simulation. The DE being simulated is a simple harmonic oscillator, allowing exact error measurements. Note the log scale on the vertical axis.

While deltaRK-produced integrators outmatch classic Runge-Kutta methods, they are still Runge-Kutta methods. As such, they have a large amount of intrinsic error, and cannot compete with state-of-the-art methods such as the 15th-order IAS15 or the 11th-order symplectic WHFast algorithms [6, 7, 8] in accuracy. However, there likely exists a range of desired accuracy for which deltaRK is the most performant option.

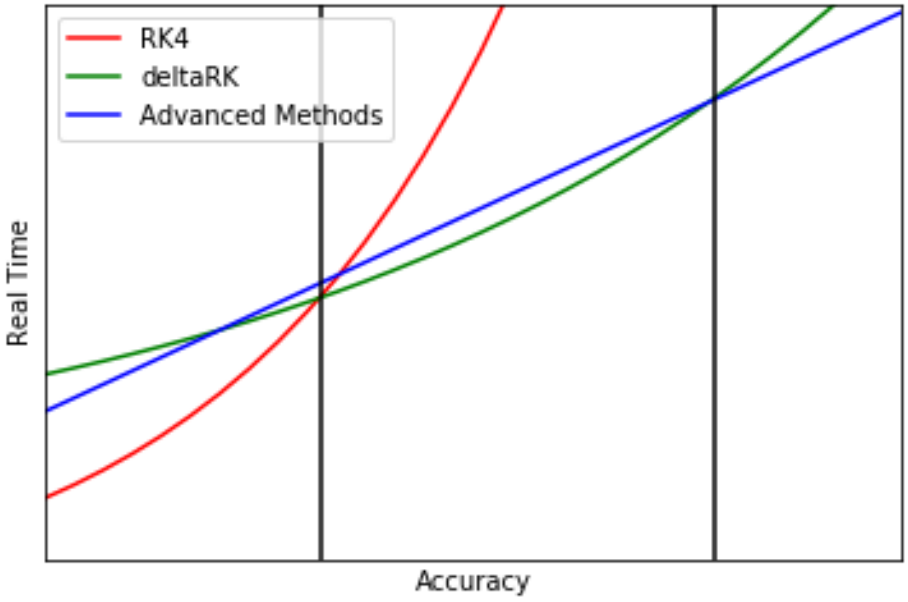

FIGURE 4: A hypothetical plot of real time required vs. accuracy desired for various integration methods. The black vertical lines denote the range of accuracy for which deltaRK would be optimal. This plot is not based on real data, but is meant to convey the tradeoffs associated with choosing an integration method.

B. Limitations

Evolutionary algorithms can only optimize so many variables at a time. For integrators with more than 6 or 7 stages, there are too many coefficients to optimize, and the algorithm becomes unstable and produces nonsensical results, often overflowing the floating-point numbers used to store the coefficients. This can be solved by limiting the number of stages and thus the size of the Butcher tableau.

In addition, the `ConsStdev` fitness function does not capture every aspect of the system. In particular, the frequency of periodic systems will erroneously and nonphysically slow over time, though the overall motion remains the same.

C. Conclusions

deltaRK creates Runge-Kutta methods to simulate a single physical system. Measurements of accuracy and robustness show that these generated methods outperform standard RK methods. While the program is limited by tableau size, number of generations, and performance, results suggest that optimizing for a specific system is a viable way of producing new Runge-Kutta methods. While not as accurate as modern methods, deltaRK may still be a reasonable option for certain applications.

D. Future Work

The fitness functions introduced in this work can be combined [3] with previous [5] and future fitness functions to create a hybrid method. There are also more potential fitness functions which optimize for a specific system. It should also be possible to evolve other types of integrators, including a class of truly symplectic methods [4].

IV. APPENDIX: CONFIGURATION

Configuration is accomplished through a user-edited Python file. This file uses nested Python classes to allow users to specify the differential equation, conserved quantity, initial conditions, training parameters, and evolution parameters in an intuitive and structured way.

```
from cfg_classes import *
from math import pi

de, cons = Gravity(
    G = 4*pi**2,
    masses = [1, 1e-3]
)

rk_steps = 4

training = NbodyStdev(
    q0 = [ [0, 0],
           [5, 0] ],
    v0 = [ [0, 0],
           [0, 2.7] ],
    dt = 1/365,
    tf = 1,
)

evolution = MuCommaLambda(
    popsize = 50,
    ngen = 100,
    mu = 50,
    lambda_ = 75,
    crossover = Crossover(
        cxpb = .75,
        alpha = .04
    ),
    mutation = Mutation(
        outerpb = .01,
        innerpb = .08,
        stdev = .04
    ),
    selection = Tournament(tournsize=4)
)
```

LISTING 1: An example `rkconfig.py`*, which sets up a simulation of Jupiter orbiting the Sun with a time step of 1 day, with a population of 50 to be evolved for 100 generations using the* (μ , λ) *algorithm. Any units can be used; this example uses Keplerian units (A.U. / years / solar masses).*

VI. REFERENCES

1. J.C. Butcher, Appl. Num. Math. 20, 247 (1996).
2. K. De Jong, Mach. Learn. 3, 121 (1988).
3. F-A. Fortin, F-M. De Rainville, M-A. Gardner, M. Parizeau, and C. Gagné, J. Mach. Learn. Res. 13, 2171 (2012).
4. H. Kinoshita, H. Yoshida, and H. Nakai, Celest. Mech. Dyn. Astr. 50, 59 (1991).
5. I. Martino and G. Nicosia, in: Learning and Intelligent Optimization, LION 2012, Lecture Notes in Computer Science vol. 7219, edited by Hamadi, Y., and Schoenauer, M. (Springer Berlin Heidelberg, Berlin, 2012).
6. H. Rein, and S-F. Liu, Astron. Astrophys. 537, A128 (2012).
7. H. Rein and D. Spiegel, Mon. Not. R. Astron. Soc. 446 [2], 1424 (2015).
8. H. Rein. and D. Tamayo, Mon. Not. R. Astron. Soc. 452 [1], 376 (2015).
9. H. Yoshida, IAU Symp. 152, 407 (1992).