RK Generator

Alexander Schaap

October 25, 2017

1 Revision History

Date	Version	Notes
October 16	1.0	Initial draft for presentation
October 25	1.1	Processed feedback and completed report

2 Symbols, Abbreviations and Acronyms

symbol	**description**
ODE	Ordinary Differential Equation
T	Test

Contents

List of Tables

List of Figures

3 General Information

This document is a test plan for the RK Generator family of ODE solvers.

3.1 Purpose

This document specifies the (black-box) verification and validation tests for the RK Generator. The intended audience are testers aiming to test the system and developers maintaining the software. This document is expected to be updated when development of the system proceeds.

3.2 Scope

The scope of the testing of the RK Generator is restricted to correctness, in some cases within an expected margin of error. Performance, while implied, should also be measured. The implementation could be compared to an unstaged version of the same code (plain OCaml rather than MetaOCaml; no generation). Alternatively, a comparison against Matlab would give an indication of performance compared to the competition.

3.3 Overview of Document

The structure of this documents is based on Smith (2006). The next section will outline the software to test, the testing team and the proposed approach. Subsequently, system tests are covered, and ultimately unit tests will be added in the section after that.

4 Plan

This section begins with a brief description of the software being tested, followed by a succinct description of the test team before discussing the automated approach proposed, the tools expected to accomplish this, and the manual testing necessary.

4.1 Software Description

The RK Generator is a metaprogramming approach to a library of ODE solvers. It generates a function that will provide values for input on the given

interval. This function will be unique to the inputs provided. During the creation of this function, metaprogramming is also employed for performance reasons.

4.2 Test Team

The test "team" will be comprise Alexander Schaap.

4.3 Automated Testing Approach

Automated tests exist primarily in the form of unit tests, though some "system" tests will do black-box verification.

4.4 Verification Tools

For unit tests, OUnit seems an obvious choice. For automating compilation and running unit tests as a primitive form of continuous integration, Make is sufficient. Unit tests give us regression testing for free. A tool exists to measure code coverage. It is called bisect_ppx and appears actively maintained. It remains to be seen how useful it is for MetaOCaml code. While code coverage has its weaknesses, it should suffice because there is little branching in this software.

4.5 Non-Testing Based Verification

Ideally, unit tests capture all the desired behaviour. However, manual inspection of the generated code will be required. The manual assessment should determine whether modifications to the generator are beneficial to either making the code more readable or improving the results in terms of accuracy. While readability isn't strictly a requirement, simple (and therefore more readable) code is often a decent indicator of correctness in metaprogramming.

5 System Test Description

Also known as black-box tests, system tests verify the software as a whole. This is done without assumed knowledge of the internal workings. This section covers the system tests along with non-functional tests.

5.1 Tests for Functional Requirements

The intent is to test using problems with known solutions to get around oracle problem. Results of other ODE solvers such as Matlab or Octave can be used for comparison purposes in case of unknown solutions. Currently, all tests have known solutions.

It is assumed that these tests test both the generator and the generated code. The inputs therefore are for the generator, but inputs are also needed for the generated code. Since the generator creates a function that takes one input, the input for generated code is provided one at a time (making the generated code run multiple times, which should be the typical use case).

5.1.1 RK4 & RK2

These tests are to be repeated for each method. The expected results for each method are described separately.

Solution Correctness Tests

1. Low-order-ODE[1]

   Type: Functional, Dynamic

   Initial State: N/A

   Input (generator): $x'(t) = 5x - 3 \quad x(2) = 1, \quad 2 \leq t \leq 3, \quad h = 0.1$

   Input (generated): 2, 3, 0, -1024, 2.2, 2.5, 23

   Output (RK4): values that match[2] the output of the exact solution $x(t) = \frac{2}{5}e^{5(t-2)} + \frac{3}{5}$ for the same inputs

   Output (RK2): similar to RK4, but with some margin of error that is within $O(h^3)$.

   How test will be performed: via OUnit

2. High-order-ODE[3]

[1] Adapted from: http://mathinsight.org/ordinary_differential_equation_introduction_examples

[2] Taking potential floating-point rounding errors into account

[3] Adapted from: http://mathinsight.org/ordinary_differential_equation_introduction_examples

Type: Functional, Dynamic

Initial State: N/A

Input (generator): $x'(t) = 7x^2x^3 \quad x(2) = 3, \quad 2 \leq t \leq 4, \quad h = 0.1$

Input (generator): 2, 3, 0, 2.5, -89, 8192

Output (RK4): values close to the exact solution $x(t) = \frac{-1}{\frac{7}{4}x^4 - \frac{85}{3}}$; error should be within $O(h^5)$. A plot of the error could help indicate this in a way that gives more confidence.

Output (RK2): similar to that of RK4, but with the larger margin of error of $O(h^3)$.

How test will be performed: via OUnit

3. Stiff-ODE

Type: Functional, Dynamic

Initial State: N/A

Input: $y'(t) = -15y(t), \quad y(0) = 1, \quad 0 \leq t \leq 1, \quad h = 0.1.$

Output (RK4 & RK2): An inaccurate numeric result that deviates significantly from the exact solution of $y(t) = e^{-15t}$ with $y(t) \to 0$ as $t \to \infty$ (or even as $t \to 1$).

How test will be performed: via OUnit

Test via diffing the output of the generator with previous results and decide whether any change is desired or not

Inspect code to see whether it matches the algorithm; aim to write code that reflects it.

...

5.2 Tests for Nonfunctional Requirements

The implied nonfunctional requirement to be tested is performance. There is no "wrong" result though. The goal is to provide insight into the performance in comparison to obvious alternatives.

5.2.1 Performance Tests

Calculation Speed Tests

1. Comparison against unstaged (plain OCaml) code

   Type: Non-Functional, Dynamic

   Initial State: N/A

   Input/Condition: Input to any one of the functional tests will be provided to both versions of the code, but the objective is to compare running times

   Output/Result: identical outputs that correspond to the expected output of the respective test chosen, but the staged (MetaOCaml) code should be faster

   How test will be performed: via Make and OUnit

2. Comparison against Matlab

   Type: Non-Functional, Dynamic

   Initial State: N/A

   Input: Input to any one of the functional tests above will be provided to both Matlab and the generator (note that Matlab could require a different input format)

   Output: Similar output but with the expectation that Matlab may be more accurate. Matlab will also likely be faster (excluding start-up time).

   How test will be performed: via Make

...

5.3 Traceability Between Test Cases and Requirements

N/A - No requirements, just a commonality analysis (which implies requirements).

6 Unit Testing Plan

Unit tests can be divided into two distinct parts. The first set verifies portions of the generator, while the second part has to verify the generated code.

One could easily assume that verifying a generator is similar to verifying other programs, and that tools can be reused for this purpose. However, a common approach at the moment is to write generated code to a file and use "diff" to compare the two. By comparing the generator's output to previous generator output for identical inputs the tester can manually determine by careful examination whether any differences are appropriate.

These tests will have to be defined after implementation.

References

W. Spencer Smith. Systematic development of requirements documentation for general purpose scientific computing software. In *Proceedings of the 14th IEEE International Requirements Engineering Conference, RE 2006*, pages 209–218, Minneapolis / St. Paul, Minnesota, 2006. URL http://www.ifi.unizh.ch/req/events/RE06/.

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

7.2 Usability Survey Questions?

This is a section that would be appropriate for some teams.