# Design: RKF45 Generator

December 8, 2017

# 1 Revision History

| Date | Version | Notes |
|---|---|---|
| November 3 | 1.0 | Initial Draft |
| November 8 | 1.1 | Processed feedback, continued writing |
| November 29 | 1.2 | Processed feedback, added interface specifications |
| December 7 | 1.3 | Processed Paul's feedback |

# 2 Symbols, Abbreviations & Acronyms

See SRS at https://github.com/aschaap/cas741.

# Contents

# 3 Introduction

[Reusing this introduction does not make any sense for your project. Please replace with an introduction suitable for your design document. —SS] [I have adapted it to illustrate the need for describing binding times. —AS]

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored. However, for this particular project, a decomposition into modules is not really useful because it is essentially a single module to begin with.

For multi-stage programming (MSP), there is another dimension in which decomposition can occur: binding times. While traditional software gives the option of binding variables to values at either compile time or runtime, MSP introduces the concept of staging, or delaying execution of code annotated as such. This gives the programmer more possible binding times. Delaying execution allows one to combine fragments of code into larger ones, which will be called generation time for clarity. This occurs after compilation time. Runtime is the moment the generated code is executed. Generators-in-generators are not considered because they are overkill for the problem of solving ODEs.

An analysis of anticipated and unlikely changes is still relevant; however, this will not translate to modules. It will affect the individual module decomposition (into functions), which merits consideration on its own, as illustrated by VanHilst and Notkin (1996). Decomposing by binding time is the way this single-module program is decomposed. Therefore, the following anticipated changes have been identified:

**AC1:** The algorithm used to solve the given IVP.

**AC2:** The implementation of the algorithms used to solve the IVP.

**AC3:** Adding interpolation of the solution provided by the specified algorithm.

As well as the following unlikely changes:

**UC1:** Output format (generated code).

**UC2:** The way the driver program invokes the software.

The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.

- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.

- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 describes the notation used throughout this document. Section 5 illustrates the alternative criterion used to decompose this program. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

# 4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1|c_2 \Rightarrow r_2|...|c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by the RKF45 Generator.

| Data Type | Notation | Description |
|---|---|---|
| character | char | a single symbol or digit |
| natural number | $\mathbb{N}$ | a number without a fractional component in [1, $\infty$) |
| natural number | $\mathbb{N}^0$ | a number without a fractional component in [0, $\infty$) |
| real | $\mathbb{R}$ | any number in (-$\infty$, $\infty$) |

[I see complex numbers later. You should put this type in your table too. —SS]

The specification of the RKF45 Generator uses some derived data types: strings, sequences and functions. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. In addition, the RKF45 Generator uses functions, which are defined by the data types of their inputs and outputs. Functions can also be arguments to other functions (e.g. the ODE is provided to the generator). Local functions are described by giving their type signature followed by their specification.

[These ideas still apply, but rather than separate sections, maybe you could just summarize this information in your revised introduction. —SS] [I've shortened and moved anticipated and unlikely changes to the intro. —AS]

# 5 Binding Times

This section will demonstrate the difference between binding times via the oft-used power example. Rather than dividing the program into modules, a decomposition by binding times is given. Such a decomposition is more appropriate when multi-stage programming (MSP) is employed, something Parnas (1972) was unaware of at that time, due to Multi-stage programming being much newer.

## 5.1 Example

Consider the following example[1] for calculating any $x^n$:

```
let square x = x * x
let rec power n x =
  if n = 0 then 1
  else if n mod 2 = 0 then square (power (n/2) x)
  else x * (power (n-1) x)
(* val power : int -> int -> int = <fun> *)
```

(The comment below the code shows the reply of the MetaOCaml interactive prompt, called the "toplevel", which contains the types of the power function.)

Assuming $x^7$ is a common operation, it would be useful to define

```
let power7 x = power 7 x
```

to give the *partial application* a name and use it throughout the code. Partial application means that when giving a function some but not all the arguments it expects, one creates a new function that only expects the remaining arguments. In the case of power7, the resulting function type will be int -> int.

In MetaOCaml, one can also create a specialized instance of the power function for a particular value of $n$. However, in the case of MetaOCaml, one can generate code which accepts any $x$ and computes $x^n$. To do this, the power function must be annotated such that computations which can be performed at generation time are distinguished from those that must be performed at run time. Because information such as $n$ is available at right away and certain not to change in the future, any computation that only depends on $n$ may be performed when the code is being generated. In contrast, information such as $x$ is not known

[1] Adapted from http://okmij.org/ftp/ML/MetaOCaml.html

at generation time and subject to change, so any computation that needs it will have to be postponed until the generated code is actually run (when x is given).

```
let rec spower n x =
  if n = 0 then .<1>.
  else if n mod 2 = 0 then .<square .~(spower (n/2) x)>.
  else .<.~x * .~(spower (n-1) x)>.;;
(* val spower : int -> int code -> int code = <fun> *)
```

Two annotations, or staging constructs, specific to MetaOCaml appear in the code above. These are brackets .< e >. and escape .~e. Brackets .< e >. 'quasi-quote' the expression e, annotating it as computed later. Escape .~e, which must be used within brackets, tells that e is computed now but produces the result for later. That result, the code, is inserted (spliced-in) into the containing bracket. The inferred type of spower is different. The result is no longer an int, but int code – the code of expressions that compute an int. The type of spower spells out which argument is received now, and which later: the future-stage argument has the code type. To specialize spower to 7, one can define

```
let spower7_code = .<fun x -> .~(spower 7 .<x>.)>.;;
(*
val spower7_code : (int -> int) code = .<
  fun x_1 -> x_1 * ((* CSP square *) (x_1 * ((* CSP square *) (x_1 *
    1))))>.
*)
```

and obtain the code of a function that expects any $x$ and returns $x^7$. Code, even of functions, can be printed, which is what MetaOCaml toplevel (interactive prompt) just did. The printout contains so-called cross-stage persistent values, or CSP, which 'quote' present-stage values such as square to be used later. One may think of CSP as references to 'external libraries' – only in our case the program acts as a library for the code it generates. If square were defined in a separate file such as `sq.ml`, then its occurrence in spower7_code would have been printed as Sq.square.

To use the specialized $x^7$ now, in our code, we should 'run' spower7_code, applying the function Runcode.run : 'a code -> 'a. The function compiles the code and links it back to our program. Runcode.run is aliased to the prefix operation (!.):

```
open Runcode
let spower7 = !. spower7_code
(* val spower7 : int -> int = <fun> *)
```

The specialized spower7 has the same type as the partially applied power7 above. They behave identically. However, power7 x will do the recursion on $n$, checking $n$'s parity, etc. In contrast, the specialized spower7 has no recursion (as can be seen from spower7_code). All

operations on n have been done when the spower7_code was computed, producing the straight-line code spower7 that only multiplies $x$.

(MetaOCaml supports an arbitrary number of later stages, letting one write not only code generators but also generators of code generators, etc.)

[I like the above summary. I would like you to use the same systematic steps for presenting your code. —SS] [I'm afraid this is challenging. —AS]

## 5.2 Compile-Time

As the name implies, this is when the code is compiled. Code generation could occur at this time, but this will not happen for the RK generator. Instead, generation will occur when the odesolve entry-point function is called. This is the only function exposed to the driver program initially. (Generated code will be available after calling odesolve)

The generator and its RK methods will be completely defined at this time. There are no inputs to it at this time, because it is effectively a library. However, the interface should satisfy the following:

### 5.2.1 Syntax

Syntax of the access programs available after this binding time.

**Exported Access Programs**

| Name | In | Out | Exceptions |
|---|---|---|---|
| odesolve | $[t_0, t_N] : \mathbb{R} \times \mathbb{R}, k : \mathbb{N}, \mathbf{x_0} : \mathbb{C}^n, \mathbf{f} : \mathbb{R} \times \mathbb{C}^n \to \mathbb{C}^n$ | $(\mathbb{N}^0 \times \mathbb{R} \to \mathbb{R})$ `code` | LESS_THAN_2_KNOTS |

[You have complex numbers for your ode? This is fine, but please make sure that your SRS documentation reflects this decision. (I haven't checked to see what the SRS says. —SS] [What does the type of the output code imply? I'm not sure what the natural number is for? I would have thought the output would be $\mathbb{R} \to \mathbb{C}^n$ (a function from time to the complex numbers defining the current values of the state variables). —SS] For the sake of clarity and easier reference from the semantics portion, the identifiers match most of the data definitions from the SRS. Note that $k$ is the exception; this is the number of knots (to divide the interval into to determine the stepsize $h$ from the SRS). [Have you updated the SRS to mention knots? The use of splines to obtain values between the ode solver points is something that wasn't previously clear in your documentation. —SS]

### 5.2.2 Semantics

Semantics of the access program(s) available after this binding time.

**State Variables** none

**Access Routine Semantics** odesolve():

- transition: N/A
- output: $out := (\mathbb{N}^0 \times \mathbb{R} \to \mathbb{R})$ `code`
  - Code that contains a function that will provide a solution for points such that point $p = t_0 + m \times h$ based on precomputed knowledge embedded in it by the generator. [This doesn't actually make things any clearer. I guess the natural number is $m$? What happened to the complex numbers? Your initial values were a sequence of values, corresponding to state variables $x_1$, $x_2$, ... $x_n$. What happened to this sequence? —SS]
- exception: $exc := (k < 2 \Rightarrow$ LESS_THAN_2_KNOTS)
  - The number of knots should be at least two, which would correspond to one at each end of the interval, and the step size $h$ being equal to the size of the interval. Behaviour for less than two knots is undefined.

### 5.2.3 Local Functions

[I don't see any complex numbers here. Was there earlier introduction a mistake, or am I missing something? —SS]

These are other functions utilized by odesolve to generate its output. They also need to be implemented by the programmer(s).

evalrk45: $\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n) \to \mathbb{R}^n$
evalrk45($t_k$,$h$, $\mathbf{x}_k$, $\mathbf{f}$) $\equiv$ IM1

This function corresponds to Instance Model 1 in the SRS.

evalrk23: $\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n) \to \mathbb{R}^n$
evalrk23($t_k$,$h$, $\mathbf{x}_k$, $\mathbf{f}$) $\equiv$ IM2

This function corresponds to Instance Model 2 in the SRS.

rk: $\mathbb{R}^n \times \mathbb{R}^n \times (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n) \times (\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n) \to \mathbb{R}^n) \to (\mathbb{R}^n)^k$
rk($karray$, $\mathbf{x}_k$, $\mathbf{f}$, $evalrk$) $\equiv result : (\forall k | k \in karray : result_i = evalrk(k_i, h, \mathbf{x}_k, \mathbf{f}))$

This function calls evalrk45 or evalrk23 on every point in the interval that happens to be a multiple of the step size ($h$). $karray$ is a sequence of size $k$ containing all the points for which to run the chosen RK method. $result$ is a two-dimensional sequence of size $n \times k$ containing the result of running the RK method on each point. $evalrk$ is either the evalrk45 or evalrk23 function.

construct: $\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \times (\mathbb{R}^n)^k \times \mathbb{N} \to (\mathbb{N} \times \mathbb{R} \to (string \vee \mathbb{R}))$ code
construct($t_0$,$t_N$,$karray$,$result$) $\equiv$ anonymous function(i,t) $\equiv (result_t)_i \vee$ OUT_OF_RANGE

This function takes the result of rk and produces a function that will return specific result values for specific ipnuts [spell check —SS] within the interval. *karray* and *result* are the same as above.

## 5.3 Generation Time

This is part of the run-time of the driver program, but from the point of view of this library, separating the moment the code for a particular IVP is generated and the moment(s) it is executed is useful. Since there is only one access function available to the driver program, this is what happens when this function is called.

The ODE, interval, step size, desired RK method and initial value are known at generation time. Since these will not change when looking for solutions given a particular point on the interval for which the algorithm is solving, the IVP can be solved at this time using the entry-point function created at compile-time.

Note that the generated code consists of a function that is not explicitly named; this is up to the driver program. Therefore, the function name is represented by "anonymous function".

## 5.4 Syntax

### 5.4.1 Exported Access Programs

| Name | In | Out | Exceptions |
|---|---|---|---|
| N/A - anonymous function | i : $\mathbb{N}_0, t : \mathbb{R}$ | $x_i(t) : \mathbb{R}$ | NOT_IN_RANGE |

## 5.5 Semantics

Semantics of the access program(s) available after this binding time.

### 5.5.1 State Variables

None.

### 5.5.2 Access Routine Semantics

"anonymous function"():

- transition: none
- output: $out := x_i(t) : \mathbb{R}$
  - A numerical value for the particular point and ODE given as input
- exception: $exc := (t < t_0 \vee t_N < t \Rightarrow \text{NOT\_IN\_RANGE})$

### 5.6 Run-Time

This is also part of the driver program's run-time, but from the point of view of the RK generator, this is (one of) the moment(s) when the generated code is run.

The function that will return values for any input within the specified interval (exported at generation time) will execute at run-time.

## 6 Connection Between Requirements and Design

N/A. While the commonality analysis has clear implications, the requirements are not explicitly recorded anywhere.

## 7 Module Decomposition

N/A. The program is broken down by binding times, rather than modules, due to the use of multi-stage programming via MetaOCaml.

## 8 Traceability Matrix

N/A.

## 9 Use Hierarchy Between Modules

N/A.

[I had to delete the Parnas1972a to get bibtex to work. —SS]

## References


Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.

Michael VanHilst and David Notkin. Decoupling change from design. *SIGSOFT Softw. Eng. Notes*, 21(6):58–69, October 1996. ISSN 0163-5948. doi: 10.1145/250707.239109. URL http://doi.acm.org/10.1145/250707.239109.

[What I was hoping to see was a transition from the unstaged code to the staged version, like you did for the square function, and like what was done for the vector norm in the ggk paper. At the very least showing the unstaged code shows the RK algorithm. —SS] [Do you think someone, who knows MetaOcaml, could implement the ode solver generator from the design specification. I feel like information is missing. How do the local functions fit with the odesolve function? The reader has to piece that together for themselves? —SS] [Maybe you could include more code snippets in your design. As we briefly discussed a literate document would be a helpful approach here. —SS]