


# Efficiency and Lazy Evaluation in Solving Schrodinger Equation by Runge-Kutta Method

*Zhang Chang-kai*

School of Physics and Astronomy  
University of Manchester

Department of Physics  
Beijing Normal University

May 2017

This document is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. 

## Abstract

This is a quick report on the performance of numerical Schrodinger equation solver written in Pure C, Haskell and Python. The programs are designed to produce the same functionality and then are used to solve the Schrodinger equation in exactly the same conditions, after which time profiling reveals the efficiency of the calculation. Moreover, the role of lazy evaluation is studied, and it is discovered that the lazy evaluation mechanism in Haskell dramatically improves the performance compared with the corresponding Python companion.

# 1 Introduction

This research concerns the efficiency investigation of the Schrodinger equation solver in Pure C, Haskell and Python. The programs are designed to solve the general Schrodinger equation using Runge-Kutta fourth-order method. The Schrodinger equation reads

$$i\frac{\partial}{\partial t}\psi = -\frac{\hbar^2}{2m}\nabla^2\psi + V\psi \quad (1)$$

In order to simplify the program, only 1D Schrodinger equation is considered. Besides, the constants in the equation are taken as  $\hbar = 1$ ,  $m = 0.5$ . Therefore, the equation becomes

$$i\frac{\partial}{\partial t}\psi = -\frac{\partial^2}{\partial x^2}\psi + V\psi \quad (2)$$

To solve the equation numerically, rewrite the equation into discrete form

$$\psi_{n+1} - \psi_n = -iH(\psi_n)\Delta t \quad (3)$$

The Hamiltonian operator is

$$\hat{H}\varphi\Big|_{x_i} = -\frac{\varphi(x_{i-1}) + \varphi(x_{i+1}) - 2\varphi(x_i)}{\Delta x^2} + V(x_i)\varphi(x_i) \quad (4)$$

The Runge-Kutta method suggests the Hamilton function  $H(\psi_n)$  to be

$$H(\psi_n) = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (5)$$

where

$$\begin{aligned} k_1 &= \hat{H}\psi_n \\ k_2 &= \hat{H}(\psi_n + \frac{h}{2}k_1) \\ k_3 &= \hat{H}(\psi_n + \frac{h}{2}k_2) \\ k_4 &= \hat{H}(\psi_n + hk_3) \end{aligned} \quad (6)$$

This is the basic principle of the calculation. This method is explicit and thus will face a convergence problem. The error of the integration will accumulate and the convergence strongly relies on the ratio

$$r = \frac{\Delta t}{\Delta x^2} \quad (7)$$

being sufficiently small.

In this research, the above algorithm is faithfully implemented in Pure C, Haskell and Python. The Pure C implementation will provide an estimation on the efficiency limit of the calculation. And the Python implementation (accelerated by numpy) is the most popular way of scientific calculation. The Haskell implementation, however, is a major role of this research. It is invoked for an analysis on how much acceleration the lazy evaluation can bring about.

All source codes mentioned in this report can be accessed through Github.

## 2 Efficiency Analysis

The programs are required to produce the following functionality

Solve the equation under

- any given initial state
- certain boundary condition
- any time-independent potential field

as well as output designated data for animation.

The efficiency analysis is performed under the following conditions

- Initial condition:

$$\psi(x, 0) = \exp\left\{-\frac{(x - 0.5)^2}{0.01}\right\} \quad (8)$$

- Boundary Condition:

$$\psi(0, t) = \psi(1, t) = 0 \quad (9)$$

- Potential:

$$V = 0 \quad (10)$$

- Temporal step size: 4e-6
- Number of recursion: 100,000
- Spacial step size: 0.02
- Number of space steps: 50
- Temporal range: (0, 0.4)
- Spacial range: (0.0, 1.0)

All the efficiency tests are taken under a 2016 MacBook running macOS Sierra 10.12.4 with 1.1 GHz Intel Core m3 dual-core processor.

The Pure C version program is compiled using GCC 6.3.0 and executed under Xcode Instrument to produce time profiling report. The summary of the report is shown in the following.

Time	Weight	Self Weight	Symbol Name
6.26 s	100.0%	0 s	engine (18755)
6.26 s	100.0%	0 s	Main Thread
6.25 s	99.9%	0 s	start
6.25 s	99.9%	325.00 ms	main
5.26 s	84.1%	880.00 ms	estimate
4.38 s	70.0%	675.00 ms	hamiltonian
3.00 ms	0.0%	3.00 ms	central
661.00 ms	10.5%	7.00 ms	printf
1.00 ms	0.0%	0 s	_dyld_start

Table 1: Time Profiling Report by Xcode Instrument

It can be seen that the total execution time is 6.26s and the major calculation `estimate` takes up most of the resources.

The Haskell version program is compiled using GHC 8.0.1 and the time profiling report is provided by GHC runtime system.

```
Prelude> :! Generator +RTS -ssderr
17,329,933,192 bytes allocated in the heap
721,864,416 bytes copied during GC
1,085,104 bytes maximum residency (551 sample(s))
115,768 bytes maximum slop
4 MB total memory in use (0 MB lost due to fragmentation)

Tot time (elapsed)  Avg pause  Max pause
Gen  0  32654 colls,  0 par  1.003s   1.038s    0.0000s   0.0008s
Gen  1     551 colls,  0 par  0.300s   0.303s    0.0006s   0.0016s

INIT      time    0.000s  ( 0.004s elapsed)
MUT       time    6.302s  ( 6.403s elapsed)
GC        time    1.303s  ( 1.342s elapsed)
EXIT      time    0.000s  ( 0.000s elapsed)
Total     time    7.608s  ( 7.748s elapsed)

%GC       time    17.1%   (17.3% elapsed)

Alloc rate 2,749,920,967 bytes per MUT second
Productivity 82.9% of total user, 81.4% of total elapsed
```

Table 2: Time Profiling Report by GHC RTS

The Python program is interpreted by standard Python 3.6.0 and the time profiling report is provided by cPython module.

```
$ python intfce.py

2599978 function calls in 15.536 seconds

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1      0.000      0.000   15.536   15.536 <string>:1(<module>)
799992      0.926      0.000      8.073      0.000 engine_ori.py:105(<lambda>)
799992      6.804      0.000      7.146      0.000 engine_ori.py:74(_genLaplace)
99999      0.127      0.000      0.127      0.000 engine_ori.py:88(_genBoundary)
      1      7.129      7.129   15.536   15.536 engine_ori.py:96(solveEq)
99999      0.207      0.000      0.207      0.000 {built-in method builtins.abs}
      1      0.000      0.000   15.536   15.536 {built-in method builtins.exec}
799992      0.343      0.000      0.343      0.000 {built-in method numpy.core
                                .multiarray.empty_like}
```

Table 3: Time Profiling Report by cPython

It is seen from the report that Haskell doubles the efficiency compared with Python. Moreover, it reaches more than 80% of the efficiency of Pure C. Therefore, the Haskell version program can be competitive especially when the required evolution time is massive.

Also, compared with Pure C, the Haskell program provides more readability and extendability. Thus, it is easier to maintain and modify.

### 3 Lazy Evaluation

One of the major factors for the great efficiency of Haskell version program is its lazy evaluation mechanism. The basic principle for the lazy evaluation is to put the actual evaluation as later as possible. What a Haskell program stores are not the actual data, but instead the *ways* to calculate the data.

As an example, suppose there is a `double` function which doubles all the elements in a list. An imperative language, e.g. Python, will have implementation as

```
double = lambda xs: [2*x for x in xs]
```

And in a functional language, e.g. Haskell, it is

```
double xs = [2*x | x <- xs]
```

There are no differences in behaviour if these two functions are called only once. However, if there are any composition of the functions, the lazy evaluation will show its advantage. To see this difference, analyse the behaviour of the following expression

```
double(double(lst))  # for Python
double double lst    -- for Haskell
```

The Python code will simply traverse the list `lst`, return the new list, pass this new list to the second `double` and traverse again. Thus, the Python will traverse the list twice in this case.

However, Haskell will do differently. The first `double` will actually do nothing but store that this is a list with each element is twice of the previous list, like

```
[x1*2, x2*2, ... xn*2]
```

And for the second `double` will do the same. Therefore, the result of the function will be

```
[x1*2*2, x2*2*2, ... xn*2*2]
```

This formulation will remain until it is required to produce output. That the evaluation will not be performed until it is required is the so-called **lazy evaluation**. This scheme will reduce the traversal from twice into once. And it will reduce more when there are more function compositions.

In Runge-Kutta fourth-order method, the Python program needs to traverse the list five times (to compute  $k_1$ ,  $k_2$ ,  $k_3$ ,  $k_4$  and  $\psi_{n+1}$ ). But with Haskell, it is supposed to reduce the traversal into once, which will significantly improve the efficiency.

Despite the great advantage shown above, the lazy evaluation will not be always better. Overuse of lazy evaluation may result in a massive waste of memory and a considerable garbage collection costs. Therefore, appropriate strictness should be added during the design of the program.

In the Haskell Schrodinger equation solver, the strictness is added at the `solve` function. The `solve` function is going to recurse 100,000 times and could result in nearly half of the execution time spending in garbage collection.

## 4 Conclusions and Remarks

From this research, it is discovered that the lazy evaluation scheme has the potential to produce a much higher calculation efficiency compared with the ordinary eager evaluation program. In solving the Schrodinger equation using Runge-Kutta fourth order method, the Haskell program with lazy evaluation reduces more than half of the computational time compared with the contemporary Python program.

Therefore, it is suggested that serious consideration should be made to introduce Haskell or lazy evaluation scheme into scientific computation, especially for those algorithms which require massive list manipulation.