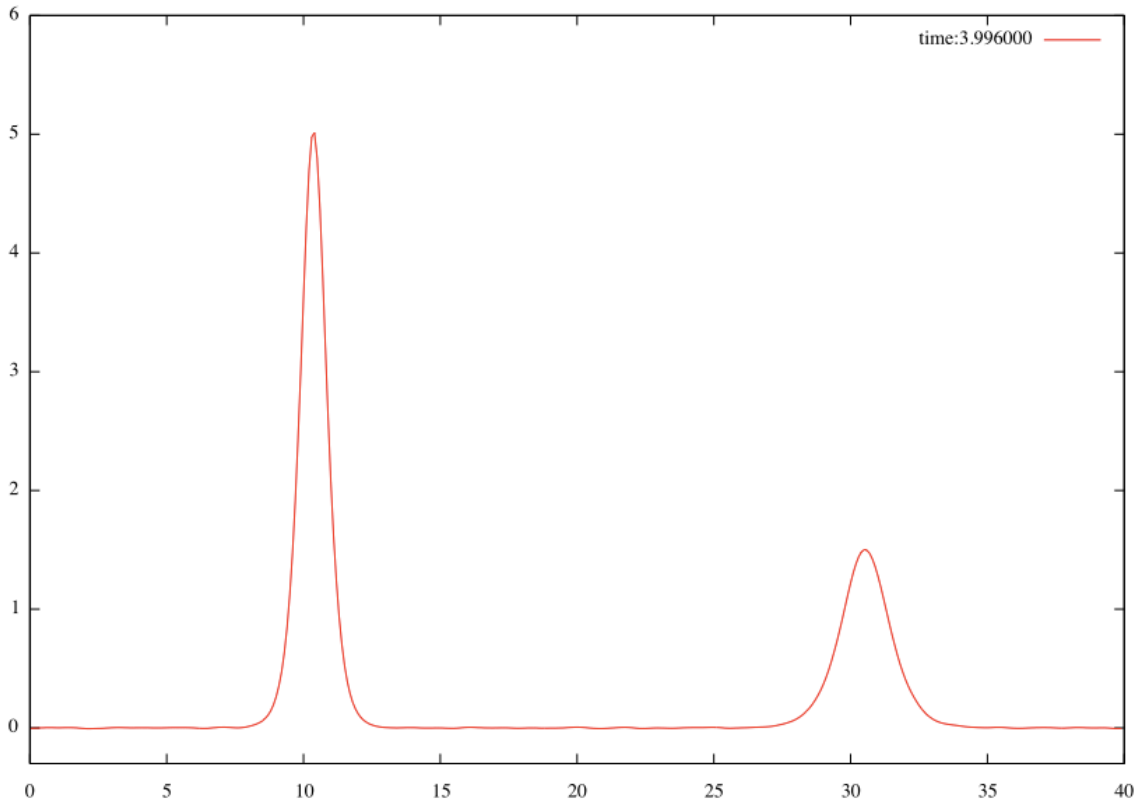
Numerical Solution for Korteweg-de-Vries (KdV) Equation

Xiaobo Sun

Collaborated with Can and Jason



1. Introduction

Korteweg-de Vries (KdV) equation is a nonlinear partial differential equation, which is initially introduced to describe the propagation of shallow water waves. We consider the KdV equation as:

$$U_t + U_{xxx} + 6UU_x = 0$$

In this project, we will use numerical method to solve the equation.

2. Project Team

Our team includes Can, Jason, and me (Xiaobo). We discussed about the KdV equation, Runge-Kutta, and other details of the project. We also shared sample codes for Runge-Kutta algorithm and gnuplot script.

3. Version Management

We use git and github to manage our code. Every team member has a private git repository, and we also share a public repository using github. The repository can be found at the following URL:

Team repository: https://github.com/sxbwjm/IAM851-PS.git
My repository: https://github.com/sxbwjm/IAM851-project1.git

4. Implementation

The project is implemented by C, which includes the following parts:

4.1 Discretization in Space (kdv.c)

We first discretize the equation in space, so that the computer can calculate the derivatives of x. The equation can be rewritten as:

$$U_t = -U_{xxx} - 6UU_x$$

We use [0:40] as x boundary, and split x into 400 grids, then calculate the value of right hand side expression at each x. This is implemented in “kdv.c”.

There are five functions in this file:

. DxU(u1, u2, deltaX):
Calculate the first derivative of x location in the middle of u1 and u2.
It returns (u2 – u1) / deltaX.

. DxxU(u_1, u, u1, deltaX):
Calculate the second derivative of x location at u, using DxU function.

. DxxxU(u_2, u_1, u, u1, u2, deltaX):
Calculate the third derivative of x location at u, using both DxU and DxxU functions

. rhs(u_2, u_1, u, u1, u2, deltaX):
Calculate the value of the right hand side expression in the equation at u.

. rhsFunc(U, newU):
Calculate the values of the right hand side expression for all x locations. U and newU are arrays of u values.

4.2 Timestepping (Runge-Kutta.c)

Runge Kutta algorithm is a method of numerically integrating ordinary differential equations.

Given equation: u' = f(t, u) with initial condition: u(0) = u0. Suppose that u_n is the value of the variable at time t_n. The Runge-Kutta formula takes u_n and t_n and calculates an approximation for u_{n+1} at a brief time later, t_{n+h}. It uses a weighted average of approximated values of f (t, u) at several times within the interval (t_n, t_n+h). The 4th order formula (also known as RK4) is given by:

$k1 = f(t_n, u_n)$
$k2 = f(t_n + h/2, u_n + k1 * h / 2)$
$k3 = f(t_n + h/2, u_n + k2 * h / 2)$
$k4 = f(t_n + h, u_n + k3)$

$u_{n+1} = u_n + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6$

Since we have the function (rhsFunc) for calculating the right hand side expression in the equation of:

$$Ut = -Uxxx - 6UUx$$

If the initial value U0 is know at t = 0, it is easy to get U values at any time t using Runge-Kutta 4th formula. It is worth pointing out there is no "t" variable in the right hand side function in the above equation.

The Runge-Kutta algorithm is implemented in “Runge-Kutta.c”.

4.3 Initial Conditions (init_conditions.c)

To use Runge-Kutta method to get U values at any time, the initial condition at t = 0 is necessary:

$$Ut0 = f(0, x)$$

Theoretically, f(0, x) can be any function. However, we need to choose some special functions as initial conditions to verify our implementation. Four functions are used:

1) $u(x,0) = N(N+1)/cosh^2(x)$ N = 0.25

2) $u(x,0) = N(N+1)/cosh^2(x)$ N = 1

3) $u(x,0) = (\frac{c}{2})/cosh^2(\frac{\sqrt{c}}{2})$ c = 5

4) $u(x,0) = (\frac{c1}{2})/cosh^2\left(\frac{\sqrt{c1}}{2}\right) + (\frac{c2}{2})/cosh^2\left(\frac{\sqrt{c2}}{2}\right)$ c1 = 10, c2 = 3

2) & 3) are one-soliton solutions, 4) is a two-soliton solution.

In the program, these functions are saved in a function pointer array, and user can choose which function to use by entering a number:

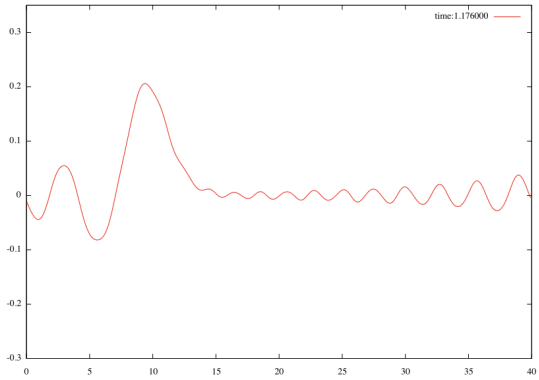
```
choose inital function:
************************************************************
0  u(x,0) = N(N+1)/cosh(x-20)^2          N = 0.25
1  u(x,0) = N(N+1)/cosh(x-20)^2          N = 1
2  u(x,0) = (c/2)/cosh2(√c/2 * (x-20))   c = 10
3  u(x,0) = (c1/2)/cosh2(√c1/2 * (x-10)+(c2/2)/cosh2(√c2/2 * (x-20))
************************************************************
Enter number:0
processing:100%
process time: 19.000 secs
Please run "gnuplot animate.plt" to show the result
```

4.4 GNUPlot Script (plot_script.c)

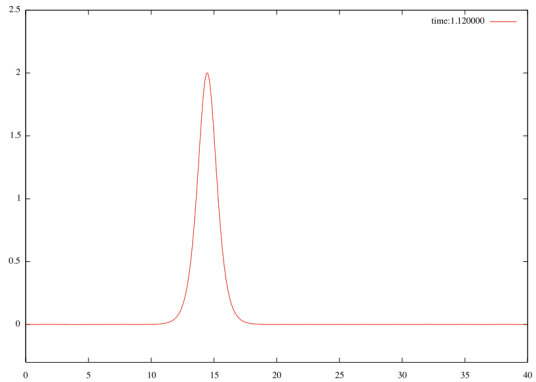
x and u value pairs are saved in text files (one file for one time frame), and can be plotted as animation in GNUPlot. We use GNUPlot script to iterate all text files to generate the amination. Since different initial function results in different max u values, to get the best animation, it is necessary to specify different range for u in the script. “plot_script.c” generate the script dynamically based on the user’s selection

on the initial function. After all data is processed, user can run “gnuplot animate.plt” (in the project directory) from the command line to watch the result.

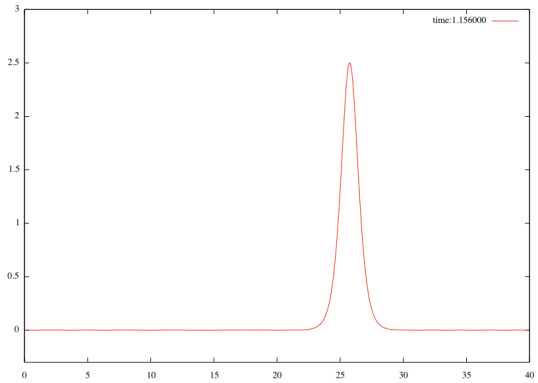
4.5 Output



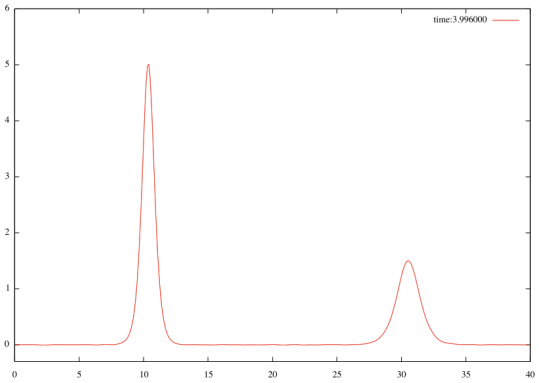
1. $u(x,0) = N(N+1)/cosh^2(x)$ N = 0.25



2. $u(x,0) = N(N+1)/cosh^2(x)$ N = 1



3. $u(x,0) = (\frac{c}{2})/cosh^2(\frac{\sqrt{c}}{2})$



4. $u(x,0) = (\frac{c1}{2})/cosh^2\left(\frac{\sqrt{c1}}{2}\right) + (\frac{c2}{2})/cosh^2\left(\frac{\sqrt{c2}}{2}\right)$

graph 2 and graph 3 are one-soliton; graph 4 is two-soliton.

* These output are recorded as gif animate files and saved in “gifs” directory.

4.6 Parallel Computation

To get good result, the delta t and delta x must be small enough to remove noise from the wave. This results in a very heavy computation in the program. There's no reason why this couldn't be done in parallel.

OpenMP is a simple C/C++/Fortran compiler extension that allows programmer to add parallelism into existing source code without significantly having to rewrite it.

Apparently, parallel is not applicable for timestepping loop since the computation is based on the values of the previous time frame. We tested the performance by parallelizing the vector computation and right hand side function.

Table 1: processing time (seconds)

Parrellel Pattern	**Threads=1**	**Threads=2**	**Threads=4**	**Threads = 8**
No parallelization	26	-	-	-
Parallelizing vector	-	32	35	295
Parallelizing Rhs function	-	20	13	321

As we can see from the table, parallelizing the rhs function can dramatically improve the performance. However, it becomes extremely slower when threads number is increased to 8.

On the other side, parallelizing vector computation will make performance worse since the process in the "for" loop is not heavy. More time will be spent on scheduling rather than real computation.

5. References

[1] İdris Dağ, Numerical solutions of KdV equation using radial basis functions
[2] KdV Solitions: http://lie.math.brocku.ca/~sanco/solitons/kdv_solitons.php