Reconsideration and new perspective about WYF equation

Tokyo university of science, Sawado research group
6217049 Yamato Suda

March 24, 2021

Abstract

This paper constructs of three chapters, Fundamentals, Main and Appendix.

In Fundamentals, we explain mathematical contents and numerical calculations. We hope these help readers.

In Main chapter, we show our ideas and fidings for WYF equation. Concretely speaking, we examined previous research by Iwasaki et al. [2]. Then, we find new mechanism between nonlinear term and dispersion term. Therefore, We verify the role of these two terms. Finally, we obtain new perspectives. We consider that these results indicate more bloader insight for WYF equation.

In Appendix, we show some contents which are couldn't explain in the last two sections. We believe that some of these will help readers. Please read from the part you would like to. If reader use this paper, it is the best our pleasure.

Contents

List of Figures

List of Tables

1 Fundamentals

1.1 KdV dynamics

1.1.1 KdV equation and solitons in 1 dimension [14]

In 1834, J.Scott-Russell (1808-1882) observed a strange wave on canal. That wave traveled with uniform velocity and no distortion in shape. He was very interested in that phenomena and examined it. And he named the wave that are different from nomal oscillating wave '*solitary wave*'.

In 1895, Korteweg and de Vries submitted a equation. It discribes the wave traveling one direction on surface of shallow water. It is called that '*Korteweg-de Vries equation*'(KdV equation):

$$\frac{\partial \eta}{\partial t} + \frac{3c_0}{2h}\eta\frac{\partial \eta}{\partial \xi} + \frac{c_0 h^2}{6}\frac{\partial^3 \eta}{\partial \xi^3} = 0, \tag{1.1.1}$$

where η is height of water surface from mean of it, $\xi = x - c_0 t$ is the coordinate moving with velocity $c_0 = \sqrt{gh}$. The solitary wave is given as a solution of the partial differential equaiton:

$$\eta = \eta_0 \text{ sech}\left(\frac{1}{2}\sqrt{\frac{3\eta_0}{h^3}(x - ct)}\right), \tag{1.1.2}$$

$$c = c_0\left(a + \frac{\eta_0}{2h}\right) = \sqrt{gh}\left(a + \frac{\eta_0}{2h}\right), \tag{1.1.3}$$

where sech $x = 1/\cosh x = 2/(e^x + e^{-x})$. This research was not very got attention.

N.J.Zabusky and M.D.Kruskal were interested in '*Fermi-Pasta-Ulam recurrence phenomena*'. It is observed in lattice system combined with nonlinear springs. This system shows excitation few modes in time evolution. Then, these modes return to initial one mode. Zabusky and Kruskal derived KdV equation as continuous model of lattice with harmonic and cubic nonlinear terms. Note that it is clear that Zabusky, Kruskal are related to KdV equation. They found that initial condition : $u(x, 0) = \cos \pi x$ makes few peaks as a shape of wave. Further these peaks travel and collide each other. At that time, their shapes are restored and moving independently. He named these waves '*soliton*'. It means that each solitary waves behave like a particle.

Solitons have some feratures below:

1. The energy density is localised.
2. The energy density has a space-time dependence of the form

$$\varepsilon(\boldsymbol{x}, t) = \varepsilon(x - \boldsymbol{u}t), \tag{1.1.4}$$

   where $\boldsymbol{u}$ is some velocity vector.
3. Solitons are those solitary waves whose energy density profiles are asymptotically (as $t \to \infty$) restored to their original shapes and velocities.

$$\varepsilon(\boldsymbol{x}, t) \to \sum_{i=1}^{N} \varepsilon_0(\boldsymbol{x} - \boldsymbol{a}_i - \boldsymbol{u}_i t) \qquad \text{as } t \to -\infty, \tag{1.1.5}$$

$$\varepsilon(\boldsymbol{x}, t) \to \sum_{i=1}^{N} \varepsilon_0(\boldsymbol{x} - \boldsymbol{a}_i - \boldsymbol{u}_i t + \boldsymbol{\delta}_i) \qquad \text{as } t \to +\infty. \tag{1.1.6}$$

That is, soliton is solitary wave whose velocity and shapes are restored after collision.

We will show one of the examples of soliton. KdV equation is described below

$$u_t + auu_x + bu_{xxx} = 0, \qquad (a, b : const.), \tag{1.1.7}$$

where $u(x, t)$ is wave field. The constants a, b are arbitrary number. We choose parameters a, b and obtain the equation as follows

$$u_t + 6uu_x + u_{xxx} = 0. \tag{1.1.8}$$

Each term has meanings. The second term $6uu_x$ is called 'nonlinear term'. It distorts the shape of wave. The third term u_{xxx} is called 'dispersion term'. It dispreads the waves per different wavenumbers. When these two effects are balance, the solitary wave propagates with no distortion.

This equation is integrable. A little algebra shows that equation (1.1.8) gives the analytical solution of the form

$$u(x, t) = \frac{1}{2}\lambda \operatorname{sech}^2\left(\frac{1}{2}\sqrt{\lambda}(x - \lambda t - x_0)\right), \tag{1.1.9}$$

where λ is parameter which decides the size of solution. This solution is called *one-soliton solution*. Generalised *N-solitons solution* are also known as an analytical form.

1.1.2 KdV equation in 2 dimensions.

Next, we will extend KdV equation from space one dimension to two dimension. This extension is known many types. We show two of them.

KP equation

The first equation is *Kadomatsev-Petviashili equation* (KP equation) [14]

$$(u_t + 6uu_x + u_{xxx})_x \pm u_{yy} = 0. \tag{1.1.10}$$

This equation has soliton solutions. Now, we set $u(x, y, t)$ as below:

$$u(x, y, t) = 2\frac{\partial^2}{\partial x^2} \log f(x, y, t). \tag{1.1.11}$$

Using Hirota's method, we can get the solution as follows

$$f = 1 + e^{\eta}, \tag{1.1.12}$$
$$\eta = k(x + py - \omega t), \quad \omega = k^2 \pm p^2. \tag{1.1.13}$$

This is called one-soliton solution and propagate on (x, y) plane.

ZK equation

The second example is called *Zakharov-Kuznetsov equation* (ZK equation) which form is given by

$$u_t + 6uu_x + u_{xxx} + u_{xyy} = 0. \tag{1.1.14}$$

This equation was proposed by Zakharov and Kuznetsov [10]. They derived it as the equation to a three-dimensional soliton solution. Restrict to two dimension, there are many works [2, 3, 6]. Melkonian obtained the analytical solution by using Hirota's method [6]. However, it is equivalent to the 1-soliton solution of KdV equation in one dimension.

We show some properties of ZK equation acoording to Iwasaki and Klein [2, 3]. We can rewrite ZK equation by using arbitrary real constant α like

$$u_t + \alpha uu_x + \nabla^2 u_x = 0, \tag{1.1.15}$$

where we use $\nabla^2 = \partial_{xx} + \partial_{yy}$. For simplicity, we choose $\alpha = 2$ and obtain the ZK equation of the form

$$u_t + 2uu_x + \nabla^2 u_x = 0. \tag{1.1.16}$$

We explain two features of ZK equation :

1. It has *solitary wave solution.*
2. It has a few number of *conserved quantities.*

First, we derive the solitary wave solution and show the numerical results. Second, we give the expression of conserved quantities.

To obtain solitary wave solution, we suppose that it propagates along x-axis with velocity c

$$u(x, y, t) = Q(x - ct, y). \tag{1.1.17}$$

Substitute eq.(1.1.17) to eq.(1.1.16) and integrate it, we get the equation of the form

$$-cQ + Q_{xx} + Q_{yy} + Q^2 = 0. \tag{1.1.18}$$

proof

We define the new variable $s \equiv x - ct$ and rewrite $Q \to Q(s, y)$. Then we substitute it to eq.(1.1.16) and obtain

$$-cQ' + 2QQ' + Q''' + Q'_{yy} = 0, \tag{1.1.19}$$

where we write $\partial Q/\partial s \Rightarrow Q'$. The s derivative is equal to x one, so we rewrite eq.(1.1.19) to

$$-c\frac{\partial Q}{\partial x} + 2Q\frac{\partial Q}{\partial x} + \frac{\partial^3 Q}{\partial x^3} + \frac{\partial^3 Q}{\partial x \partial y^2} = 0. \tag{1.1.20}$$

We can integrate this equation by x and finally obtain the equation for $Q(x, y)$ as

$$-cQ + Q^2 + Q_{xx} + Q_{yy} = 0. \tag{1.1.21}$$

This is just eq.(1.1.18).

Moreover we suppose axial-symmetry, i.e. Q has only dependence of $r \equiv \sqrt{x^2 + y^2}$, we can reduce eq.(1.1.18) to the form

$$\frac{d^2 Q}{dr^2} + \frac{1}{r}\frac{dQ}{dr} - cQ + Q^2 = 0. \tag{1.1.22}$$

We can solve eq.(1.1.22) numerically by using shooting method [2], Figure1.1. Iwasaki et al. called it "bell-shaped solitary wave solution" and they investigated the stability of pulse and collision of two pulses. From their work, the one pulse travels along to x-axis without deformation. In the collision of two pulses, if their amplitudes are similar, the pulses behave approximately soliton, i.e. their collision is elastic. If their amplitudes are dissimilar, the strong pulse becomes stronger and the weak one becomes weaker with generations of ripples. We verify that the bell-shaped solution propagates with no deformation. We show these results in section3.

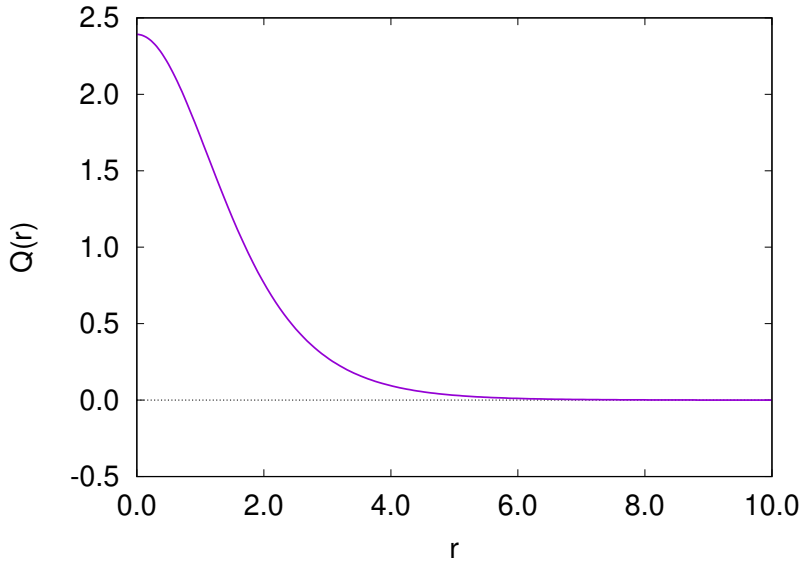


Figure1.1: Numerical solution to eq.(1.1.22) for $c = 1.0$. We get the value as $Q(0) = 2.39196$ by shooting method with boundary condition $|Q_r| \to 0 \quad (r \to \infty)$.

Next, we show the conserved quantities for ZK equation [2, 3]. They are expressed in the form as following

$$M = \iint u \, dS, \tag{1.1.23}$$

$$P = \iint \frac{1}{2} u^2 \, dS, \tag{1.1.24}$$

$$E = \iint \left(\frac{1}{2} ({u_x}^2 + {u_y}^2) - \frac{1}{3} u^3 \right) dS, \tag{1.1.25}$$

where interval of integration is defined as $x \times y = [-\infty, \infty] \times [-\infty, \infty]$. The proofs are given in section3. By using the scale invariance of eq.(1.1.22) under the transformation $F = cQ(\sqrt{c}r)$, we obtain

$$M = m, \tag{1.1.26}$$

$$P = \frac{1}{2} mc, \tag{1.1.27}$$

$$E = -\frac{1}{3} mc^2. \tag{1.1.28}$$

These are basic properties of ZK equation.

proof

- Scale invariance of eq.(1.1.22).

  Consider the transformation $Q = cF(\sqrt{c}r)$, eq.(1.1.22) change to the form

$$\frac{d^2F}{dR^2} + \frac{1}{R}\frac{dF}{dR} - F + F^2 = 0, \tag{1.1.29}$$

  where $R \equiv \sqrt{c}r$.

- The integrated values.

  We substitute $Q = cF$ into the eq.(1.1.23) – (1.1.25). By using relation $R = \sqrt{X^2 + Y^2} = \sqrt{c}r, X = \sqrt{c}x, Y = \sqrt{c}y$, we get

$$\begin{aligned} M &= \iint Q\, dxdy \\ &= \iint cF \frac{dX}{\sqrt{c}} \frac{dY}{\sqrt{c}} \\ &= \iint F\, dXdY \equiv m = \text{const.} \quad (\because F \to 0, \quad r \to \infty) \end{aligned} \tag{1.1.30}$$

$$\begin{aligned} P &= \iint \frac{1}{2} Q^2\, dxdy \\ &= \iint \frac{1}{2} cF^2\, dXdY \\ &= \frac{c}{2} \iint \left[F - \frac{1}{R}\frac{d}{dR}\left(R\frac{dF}{dR} \right) \right] dXdY \quad (\because \text{eq.(1.1.29)}) \\ &= \frac{1}{2}mc, \quad (\because F \to 0, \quad r \to \infty) \end{aligned} \tag{1.1.31}$$

$$\begin{aligned} E &= \iint \left(\frac{1}{2}({u_x}^2 + {u_y}^2) - \frac{1}{3}u^3 \right) dS \\ &= c^2 \iint \left[\frac{1}{2}\left(\frac{dF}{dR} \right)^2 - \frac{1}{3}F^3 \right] dXdY = -\frac{1}{3}mc^2. \end{aligned} \tag{1.1.32}$$

1.1.3 Integrable systems.

We call the system '*integrable*' when we can solve an initial value problems, i.e. we can get value of function $u(x,t)$ by using initial data $u(x,0)$. This property leads the existence of conserved quantity and conservation low. In particular, the amount of it is infinite in integrable system. We show that KdV equaiton[*1] has infinite number of conserved quantities. In order to show that, we introduce Bäcklund transformation. The Bäcklund transformation for the KdV equation $u_t - 6uu_x + u_{xxx} = 0$ is given by following equations

$$u(x,t) = w_x(x,t), \qquad u'(x,t) = w'_x(x,t), \tag{1.1.33}$$

$$w_x + w'_x = -2\eta^2 + \frac{1}{2}(w - w')^2, \tag{1.1.34}$$

$$w_t + w'_t = 2({w_x}^2 + w_x w'_x + {w'_x}^2) - (w - w')(w_{xx} - w'_{xx}), \tag{1.1.35}$$

where η is constant. If w_x is solution of KdV equation, w'_x is also solution of it[*2]. In order to obtain the conservation low, we rewrite equations following forms

$$w_x + w'_x = -2\eta^2 + \frac{1}{2}(w - w')^2, \tag{1.1.36}$$

$$w_t - w'_t = -[2w_{xx} - 2w_x(w - w') + 4\eta^2(w - w')]_x. \tag{1.1.37}$$

Further, we expand $w - w'$ in terms of $1/\eta$ as follows

$$w - w' = 2\eta + \sum_{n=1}^{\infty} f_n \eta^{-n}. \tag{1.1.38}$$

Substitute eq.(1.1.38) to eq.(1.1.37) and compare the coefficients of same order $1/\eta$. Finally, we get recurrence relation about f_n below

$$f_{n+1} = u\delta_{n,0} - \frac{1}{2}f_{n,x} - \frac{1}{4}\sum_{m=1}^{n-1} f_n \eta^{-n}, \tag{1.1.39}$$

where f_n is conserved density. It is because that we can get the equation of continuity by substituting eq.(1.1.38) into eq.(1.1.37)

$$\frac{\partial}{\partial t}f_n + \frac{\partial}{\partial x}(-2uf_n + 4f_{n+2}) = 0, \qquad (n = 1, 2, \cdots). \tag{1.1.40}$$

[*1] We deal with the form of

$$u_t - 6uu_x + u_{xxx} = 0.$$

It is obtained by changing dependent variable $u \to -u$ in

$$u_t + 6uu_x + u_{xx} = 0.$$

[*2] Refer the section3.

From eq.(1.1.39), we can constract f_n as follows

$$f_1 = u, \qquad f_2 = -\frac{1}{2}u_x, \qquad f_3 = -\left(\frac{1}{2}\right)^2 (u^2 - u_{xx}), \tag{1.1.41}$$

$$f_4 = \left(\frac{1}{2}\right)^3 (2u^2 - u_{xx})_x, \qquad f_5 = \left(\frac{1}{2}\right)^4 [2u^3 + u_x^2 - (6uu_x - u_{xxx})_x]. \tag{1.1.42}$$

As noted above, we can obtain the f_n sequentially. Thus, KdV equation has infinite number of conserved quantities. This property leads the stability of its solutions.

1.2 Fluid dynamics.

In this section, we explain the funadamentals of fluid dynamics. In paticular, *inviscid incompressive* fluid, i.e. fulid have no viscosity and constant density of mass and is often called *perfect fluid*. This fluid often be used to research fulid dynamics. It is because that it is easy to deal with. If you want to learn more, you sholud refer to some literatures [13, 12] and so on.

1.2.1 Governing equations

In order to describe the fulid motion, we often use field of velocity $\vec{v} = (u, v, w)$ where u, v, w are function of space and time, i.e. (x, y, z, t). There are two governing equations in fluid dynamics. The first equation is *Navier-Stokes equation* and the second one is called *Equation of continuity*. These equations are given as following forms

$$\rho\frac{\partial \vec{v}}{\partial t} + \rho(\vec{v}\cdot\nabla)\vec{v} = -\nabla p + \vec{F}, \tag{1.2.1}$$

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\vec{v}) = 0, \tag{1.2.2}$$

where $\rho(\vec{r}, t)$ is density of fluid, $p(\vec{r}, t)$ is pressure of fluid, $\vec{F}$ is external force to fulid per volume and ∇ is ordinary diferential operator : $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$. These two equations are valid for inviscid fluid. Moreover, we restrict it to incompressive fluid. This assumption means that density ρ is constant. Finally we get governing equations for the perfect fluid as followings

$$\frac{\partial \vec{v}}{\partial t} + (\vec{v}\cdot\nabla)\vec{v} = -\frac{1}{\rho}\nabla p + \vec{F}, \tag{1.2.3}$$

$$\nabla\cdot\vec{v} = 0, \tag{1.2.4}$$

where we rewrite external force $\vec{F}' \equiv \vec{F}/\rho$ as $\vec{F}$ because $\vec{F}$ is almost propotional to the density ρ.

1.2.2 The derivation of WYF equation [7]

We consider the motion of the perfect fluid at the planetary surface. It is often called the β-plane model. In order to obtain its form, we consider the rotating frame on the tangent plane at the surface with angular velocity Ω (Figre1.2). According to the classical mechanics, we have to treat the Coriolis force. Consequently, we get the equations from eq.(1.2.3) of the forms

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + 2\Omega v + x\Omega^2, \tag{1.2.5}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + -2\Omega u + y\Omega^2, \tag{1.2.6}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g, \tag{1.2.7}$$

where g is the gravitational accelerartion on its planet. For small angular velocity $\Omega \ll 1$, we can neglect the terms Ω^2 and obtain

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + 2\Omega v, \tag{1.2.8}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial y} - 2\Omega u, \tag{1.2.9}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g. \tag{1.2.10}$$

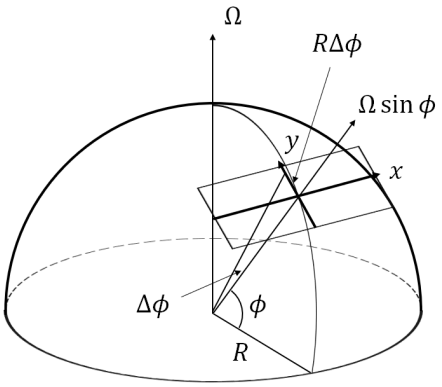


Figure1.2: The β-plane on the planet.

Next, we consider the surface at the latitude ϕ and small difference $\phi \to \phi + \Delta\phi$. When we consider the Coriolis force at this surface, it becomes

$$(2\Omega v, -2\Omega u, 0) \to (2\Omega v \sin\phi, -2\Omega u \sin\phi, 0). \tag{1.2.11}$$

Moreover, for the angular difference $\Delta\phi$, we can approximate the Coriolis force of the form

$$2\ \Omega v \sin(\phi + \Delta\phi) \sim 2\Omega v \sin\phi + 2\Omega v\, \Delta\phi \cos\phi = (f + \beta y)v, \tag{1.2.12}$$

$$-2\Omega u \sin(\phi + \Delta\phi) \sim -2\Omega v \sin\phi - 2\Omega u\, \Delta\phi \cos\phi = -(f + \beta y)u, \tag{1.2.13}$$

$$y \equiv R\Delta\phi, \quad f \equiv 2\Omega \sin\phi, \quad \beta = 2\Omega \cos\phi/R, \tag{1.2.14}$$

where R is the radius of the planet and y is displacement along the direction of latitude. Finally, we obtain the equations as following

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + (f + \beta y)v, \tag{1.2.15}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial y} - (f + \beta y)u, \tag{1.2.16}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - g. \tag{1.2.17}$$

This is called *β-plane approximation*. Next, in order to deal with the shallow water system[*3], we consider to rescale these equations. If the static fluid has the depth H with pressure p_s, the velocity of the fluid is zero, such that the equations eq.(1.2.15)-(1.2.17) become

$$0 = -\frac{1}{\rho}\frac{\partial p_s}{\partial x}, \quad 0 = -\frac{1}{\rho}\frac{\partial p_s}{\partial y}, \quad 0 = -\frac{1}{\rho}\frac{\partial p_s}{\partial z} - g. \tag{1.2.18}$$

From these equations, we can write $p_s = p_s(z)$ of the form

$$p_s(z) = -\rho g(z - H) + p_0, \tag{1.2.19}$$

where p_0 is the pressure at the surface of fluid. In the comoving frame with the fluid, we rewrite the pressure as

$$p = p_s + (p - p_s) = p_s(z) + P. \tag{1.2.20}$$

By substituing eq.(1.2.20) to eq.(1.2.15)-(1.2.17), we obtain the equations

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{1}{\rho}\frac{\partial P}{\partial x} + (f + \beta y)v, \tag{1.2.21}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{1}{\rho}\frac{\partial P}{\partial y} - (f + \beta y)u, \tag{1.2.22}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial P}{\partial z}. \tag{1.2.23}$$

Then, in order to rescale these equations, we employ the physical scale factors

the vertical scale H and the velocity W,
the horizontal scale L and the velocity V.

From using these scale factors[*4], the variables rewrite as following

$$x = Lx',\ y = Ly',\ z = Hz',\ t = \frac{L}{V}t',\ u = Vu',\ v = Vv',\ w = Ww', \tag{1.2.24}$$

and dimensionless variables x' etc. are order $\mathcal{O}(1)$

$$x', y', z', u', v', w' \sim \mathcal{O}(1). \tag{1.2.25}$$

[*3] The system which has the sufficient scale of horizontal length to neglect the scale of depth.

[*4] These scale factors have dimension. We define the new dimensionless variables by eq.(1.2.24).

In terms of new variables, the condition of incompressive fluid eq.(1.2.4) becomes

$$\frac{V}{L}\left(\frac{\partial u'}{\partial x'}+\frac{\partial v'}{\partial y'}\right)+\frac{W}{H}\frac{\partial W'}{\partial z}=0. \tag{1.2.26}$$

Thus, the scale factors are related with each other $\frac{V}{L}=\frac{W}{H}$. The eq.(1.2.21)-(1.2.23) are rewritten to following forms

$$\frac{V^2}{L}\left(\frac{\partial u'}{\partial t'}+u'\frac{\partial u'}{\partial y'}+w'\frac{\partial u'}{\partial z'}\right)=-\frac{1}{\rho L}\frac{\partial P}{\partial x'}+fV\left(1+\frac{\beta L}{f}y'\right)v', \tag{1.2.27}$$

$$\frac{V^2}{L}\left(\frac{\partial v'}{\partial t'}+u'\frac{\partial v'}{\partial y'}+w'\frac{\partial v'}{\partial z'}\right)=-\frac{1}{\rho L}\frac{\partial P}{\partial y'}-fV\left(1+\frac{\beta L}{f}y'\right)u', \tag{1.2.28}$$

$$\frac{V^2H}{L^2}\left(\frac{\partial w'}{\partial t'}+u'\frac{\partial w'}{\partial x'}+v'\frac{\partial w'}{\partial y'}+w'\frac{\partial w'}{\partial z'}\right)=-\frac{1}{\rho H}\frac{\partial P}{\partial z'}. \tag{1.2.29}$$

We are intrested in the explicit value of the physical scale. In paticular, we examine for the Earth and the Jupiter as below :

$$\underline{\text{The Earth}}: f = 2\Omega\sin\phi\sim\frac{2\pi}{1\,\text{day}}\sim 10^{-4}\,\text{s} \tag{1.2.30}$$

$$\beta = 2\Omega\cos\phi/R\sim\frac{2\pi}{1\,\text{day}}\frac{1}{R}\sim 10^{-11}\,\text{m}^{-1}\text{s}^{-1} \tag{1.2.31}$$

$$L\sim 10^6\,\text{m},\quad H\sim 10^4\,\text{m},\quad V=10\,\text{m}\,\text{s}^{-1} \tag{1.2.32}$$

$$\Rightarrow\frac{V^2}{L}\sim 10^{-4},\quad fV\sim 10^{-3},\quad fV\frac{\beta L}{f}\sim 10^{-4} \tag{1.2.33}$$

$$\underline{\text{The Jupiter}}: f = 2\Omega\sin\phi\sim\frac{2\pi}{0.4\,\text{day}}\sim 10^{-4}\,\text{s} \tag{1.2.34}$$

$$\beta = 2\Omega\cos\phi/R\sim\frac{2\pi}{0.4\,\text{day}}\frac{1}{R}\sim 10^{-12}\,\text{m}^{-1}\text{s}^{-1} \tag{1.2.35}$$

$$L\sim 10^7\,\text{m},\quad H\sim 10^5\,\text{m},\quad V=100\,\text{m}\,\text{s}^{-1} \tag{1.2.36}$$

$$\Rightarrow\frac{V^2}{L}\sim 10^{-3},\quad fV\sim 10^{-2},\quad fV\frac{\beta L}{f}\sim 10^{-3} \tag{1.2.37}$$

From these estimations, we find that the order of l.h.s. in eq.(1.2.27),(1.2.28) are different from them of the r.h.s. by 10^{-1}. Thus, we require the relations for the first and the second term in the r.h.s. of eq.(1.2.27),(1.2.28) as following

$$-\frac{1}{\rho L}\frac{\partial P}{\partial x'}\sim -fVv',\qquad -\frac{1}{\rho L}\frac{\partial P}{\partial y'}\sim fVu'. \tag{1.2.38}$$

These relations lead to the results

$$P=f\rho LVp',\quad p'\sim\mathcal{O}(1), \tag{1.2.39}$$

where p' is dimensionless variable. By using these estimations, we finally obtain the equations

in terms of rescaled variables

$$\frac{V}{fL}\left(\frac{\partial u'}{\partial t'}+u'\frac{\partial u'}{\partial x'}+v'\frac{\partial u'}{\partial y'}+w'\frac{\partial u'}{\partial z'}\right)=-\frac{\partial p'}{\partial x'}+(1+\frac{\beta L}{f}y')v', \tag{1.2.40}$$

$$\frac{V}{fL}\left(\frac{\partial v'}{\partial t'}+u'\frac{\partial v'}{\partial x'}+v'\frac{\partial v'}{\partial y'}+w'\frac{\partial v'}{\partial z'}\right)=-\frac{\partial p'}{\partial y'}-(1+\frac{\beta L}{f}y')u', \tag{1.2.41}$$

$$\frac{V}{fL}\frac{H^2}{L^2}\left(\frac{\partial w'}{\partial t'}+u'\frac{\partial w'}{\partial x'}+v'\frac{\partial w'}{\partial y'}+w'\frac{\partial w'}{\partial z'}\right)=-\frac{\partial p'}{\partial z'}. \tag{1.2.42}$$

Here, we return to the order of equations again. From estimations above, we directly find

$$\underline{\text{The Earth}}: \frac{V}{fL}\sim 10^{-1}, \quad \frac{H}{L}\sim 10^{-2}, \quad \frac{\beta L}{f}\sim 10^{-1}. \tag{1.2.43}$$

$$\underline{\text{The Jupiter}}: \frac{V}{fL}\sim 10^{-1}, \quad \frac{H}{L}\sim 10^{-2}, \quad \frac{\beta L}{f}\sim 10^{-1}. \tag{1.2.44}$$

As a result of the factor H^2/L^2 in the l.h.s. of eq.(1.2.44), we can regard the l.h.s. of eq.(1.2.44) as zero, i.e. $0=-\partial p'/\partial z'$. From eq.(1.2.24),(1.2.39), we finally obtain the equations in terms of original variables as following

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}-(f+\beta y)v=-\frac{1}{\rho}\frac{\partial P}{\partial x}, \tag{1.2.45}$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}+(f+\beta y)u=-\frac{1}{\rho}\frac{\partial P}{\partial y}, \tag{1.2.46}$$

$$0=-\frac{\partial P}{\partial z}, \tag{1.2.47}$$

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}=0. \tag{1.2.48}$$

We consider the two-dimensional perfect fluid in β-plane model and estimated it by using the value of physical scale. As a result, we obtain a set of partial differential equations given by eq.(1.2.45)-(1.2.48).

Next, we consider the shallow water system : the fluid has the mean depth H and the pressure is p_0 at the surface of it. For this boundary condition, the pressure of fluid $P(x,y,t)$ defined by eq.(1.2.20) is given by[*5]

$$P(x,y,t)=\rho g(h(x,y,t)-H), \tag{1.2.49}$$

where $h(x,y,t)$ is the depth of fluid (fig.1.3). Substituting eq.(1.2.49) to eq.(1.2.45) and (1.2.46), we obtain two equaitons as follows

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+w\frac{\partial u}{\partial z}-(f+\beta y)v=-g\frac{\partial h}{\partial x}, \tag{1.2.50}$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+w\frac{\partial v}{\partial z}+(f+\beta y)u=-g\frac{\partial y}{\partial y}. \tag{1.2.51}$$

[*5] If the pressure p is p_0 at $z=h$, the expression of p becomes

$$p_0=-\rho g(h-H)+p_0+P(x,y,t).$$

Thus, we get the relation $P(x,y,t)=\rho g(h-H)$.

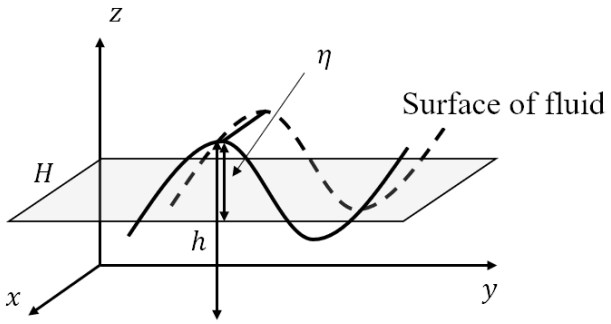


Figure1.3: The definitions of H, h and η. H is the mean depth of fluid, h is the depth of fluid and η is the displacement of fluid from H.

Moreover, we suppose that u, v is independent of the coordinate z. If $\partial u/\partial z, \partial v/\partial z = 0$ at initial time, this conditon is satisfied in time evolution because eq.(1.2.50),(1.2.51) have no z dependence without these terms. Therefore, we can rewrite eq.(1.2.50) and (1.2.51) to the forms

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - (f + \beta y)v = -g\frac{\partial h}{\partial x}, \tag{1.2.52}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + (f + \beta y)u = -g\frac{\partial y}{\partial y}. \tag{1.2.53}$$

Next we integrate eq.(1.2.48) by z

$$\int_0^h dz \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + \int_0^h dx \frac{\partial w}{\partial z} = 0, \tag{1.2.54}$$

$$h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + [w]_0^h = 0, \tag{1.2.55}$$

where $w(z = 0) = 0$, because we are interested in the motion of surface of fluid. At the surface of fluid, $h(x, y, t)$ will change as following

$$t : (x, y, h(x, y)) \to t + \delta t : (x + u\delta t, y + v\delta t, h + w\delta t),$$

$$\begin{aligned} h(x, y, t) + w\delta t &= h(x + u\delta t, y + v\delta t, +\delta t) \\ &= h(x, y, t) + \left(\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y}\right)\delta t + \mathcal{O}(\delta t^2), \\ w(z = h)\delta t &\sim \left(\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y}\right)\delta t, \\ \therefore w(z = h) &= \frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y}. \end{aligned}$$

Therefore, we can rewrite equation eq.(1.2.55) to the form

$$\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} + h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = 0. \tag{1.2.56}$$

Finally we obtain the equations of the shallow water system

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - (f+\beta y)v = -g\frac{\partial h}{\partial x}, \tag{1.2.57}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + (f+\beta y)u = -g\frac{\partial y}{\partial y}, \tag{1.2.58}$$

$$\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} + h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = 0. \tag{1.2.59}$$

In order to derive the WYF equation, we introduce the new variable $\eta(x, y, t)$ which is defined by the displacement from the mean depth of fluid (fig.1.3) :

$$\eta(x, y, t) \equiv h(x, y, t) - H. \tag{1.2.60}$$

By using η and the rescaled variables x', y', t', u', v' eq.(1.2.24), we rewrite the shallow water equaitons to the forms

$$\frac{V^2}{L}\left(\frac{\partial u'}{\partial t} + u'\frac{\partial u'}{\partial x'} + v'\frac{\partial u'}{\partial y'}\right) - fV\left(1 + \frac{\beta L}{f}y'\right)v' = -\frac{g}{L}\frac{\partial \eta}{\partial x'}, \tag{1.2.61}$$

$$\frac{V^2}{L}\left(\frac{\partial v'}{\partial t} + u'\frac{\partial v'}{\partial x'} + v'\frac{\partial v'}{\partial y'}\right) + fV\left(1 + \frac{\beta L}{f}y'\right)u' = -\frac{g}{L}\frac{\partial \eta}{\partial y'}, \tag{1.2.62}$$

$$\frac{V}{L}\left(\frac{\partial \eta}{\partial t} + u'\frac{\partial \eta}{\partial x'} + v'\frac{\partial \eta}{\partial y'}\right) + \frac{V}{L}(H+\eta)\left(\frac{\partial u'}{\partial x'} + \frac{\partial v'}{\partial y'}\right) = 0. \tag{1.2.63}$$

Here, we show some calculations below :
The rescaled variables are given by

$$x = Lx',\ y = Ly',\ z = Hz',\ t = \frac{L}{V}t',\ u = Vu',\ v = Vv',\ w = Ww'.$$

Therefore, we rewrite eq.(1.2.57) in terms of new variables as following

$$\frac{\partial u}{\partial t} \to \frac{\partial t'}{\partial t}\frac{\partial u}{\partial t'} = \frac{V}{L}\frac{\partial}{\partial t'}(Vu') = \frac{V^2}{L}\frac{\partial u'}{\partial t'}, \tag{1.2.64}$$

$$u\frac{\partial u}{\partial x} \to Vu'\frac{\partial x'}{\partial x}\frac{\partial}{\partial x'}(Vu') = \frac{V^2}{L}u'\frac{\partial u'}{\partial x'}, \tag{1.2.65}$$

$$v\frac{\partial u}{\partial y} \to Vv'\frac{\partial y'}{\partial y}\frac{\partial}{\partial y'}(Vu') = \frac{V^2}{L}v'\frac{\partial u'}{\partial y'}, \tag{1.2.66}$$

$$-(f+\beta y)v \to -(f+\beta L y')Vv' = -fV\left(1 + \frac{\beta L}{f}y'\right)v', \tag{1.2.67}$$

$$-g\frac{\partial h}{\partial x} \to -g\frac{\partial x'}{\partial x}\frac{\partial \eta}{\partial x'} = -\frac{g}{L}\frac{\partial \eta}{\partial x'}. \tag{1.2.68}$$

Therefore, we obtain the equation eq.(1.2.61) above. par We remember the estimations for the scale factors eq.(1.2.33) and (1.2.37) as follows

$$\underline{\text{The Earth}} : \frac{V^2}{L} \sim 10^{-4}, \quad fV \sim 10^{-3}, \quad fV\frac{\beta L}{f} \sim 10^{-4}. \tag{1.2.69}$$

$$\underline{\text{The Jupiter}} : \frac{V^2}{L} \sim 10^{-3}, \quad fV \sim 10^{-2}, \quad fV\frac{\beta L}{f} \sim 10^{-3}. \tag{1.2.70}$$

Therfore, we require some terms in the equations to be satisfied the conditions given by

$$\underline{\text{The Earth}} : -\frac{g}{L}\frac{\partial \eta}{\partial x'} \sim fVv' \sim 10^{-3}, \quad -\frac{g}{L}\frac{\partial \eta}{\partial y'} \sim fVu' \sim 10^{-3}. \tag{1.2.71}$$

$$\underline{\text{The Jupiter}} : -\frac{g}{L}\frac{\partial \eta}{\partial x'} \sim fVv' \sim 10^{-2}, \quad -\frac{g}{L}\frac{\partial \eta}{\partial y'} \sim fVu' \sim 10^{-2}. \tag{1.2.72}$$

Moreover, we introduce order $\mathcal{O}(1)$ variable η' as

$$\eta \equiv \frac{fLV}{g}\eta'. \tag{1.2.73}$$

Finally we obtain the equations in terms of the rescaled variables :

$$\frac{V}{fL}\left(\frac{\partial u'}{\partial t'} + u'\frac{\partial u'}{\partial x'} + v'\frac{\partial u'}{\partial y'}\right) - \left(1 + \frac{\beta L}{f}y'\right)v' = -\frac{\partial \eta'}{\partial x'}, \tag{1.2.74}$$

$$\frac{V}{fL}\left(\frac{\partial v'}{\partial t'} + u'\frac{\partial v'}{\partial x'} + v'\frac{\partial v'}{\partial y'}\right) + \left(1 + \frac{\beta L}{f}y'\right)u' = -\frac{\partial \eta'}{\partial y'}, \tag{1.2.75}$$

$$\frac{V}{fL}\frac{f^2L^2}{gH}\left(\frac{\partial \eta'}{\partial t'} + u'\frac{\partial \eta'}{\partial x'} + v'\frac{\partial \eta'}{\partial y'}\right) + \left(1 + \frac{V}{fL}\frac{f^2L^2}{gH}\eta'\right)\left(\frac{\partial u'}{\partial x'} + \frac{\partial v'}{\partial y'}\right) = 0. \tag{1.2.76}$$

In order to consider the shallow water system, we introduced the displacement of fluid $\eta(x, y, t)$ and rescaled the equations. As a result of that, we obtained the new set of partial differential equations eq.(1.2.74)-(1.2.76) which include some scale parameters. Next our work is applying the perturbation to these equations by using the appropriate parameters. Consequently, we obtain the WYF equation.

Here we introduce the scale parameters :

$$\hat{\epsilon} \equiv \frac{V}{fL}, \quad \hat{\beta} \equiv \frac{\beta L}{f}, \quad \hat{s} \equiv \frac{gH}{f^2L^2}. \tag{1.2.77}$$

In particular, $\hat{s}$ can be expressed as

$$\hat{s} = \frac{\lambda_{\mathrm{R}}^2}{L^2}, \tag{1.2.78}$$

where

$$\lambda_{\mathrm{R}} \equiv \frac{c_g}{f}, \quad c_g \equiv \sqrt{gH}. \tag{1.2.79}$$

λ_R is called the Rossby radious of deformation and c_g is called the internal or external velocity of gravitational wave. The deformation radius is very important factor to determine the regime. We can regard λ_{R} as a distance which gravitational wave with velocity c_g propagates during time $1/f$. Qualitatively, λ_{R} represents the degree of effect between gravity and Coriolis force.

In the case of the Jupiter, the parameters $\hat{\epsilon}, \hat{\beta}$ are estimated as

$$\hat{\epsilon} = \frac{V}{fL} \sim \frac{100}{10^{-4} \cdot 10^7} \sim 10^{-1}, \tag{1.2.80}$$

$$\hat{\beta} = \frac{\beta L}{f} \sim \frac{10^{-12} \cdot 10^7}{10^{-4}} \sim 10^{-1}, \tag{1.2.81}$$

so we employ $\hat{\beta}$ as the parameter of perturbation. By using $\hat{\epsilon}, \hat{\beta}$, we can rewrite the equaitons eq.(1.2.74-(1.2.76) to the forms

$$\hat{\epsilon}\frac{Du}{Dt} - (1+\hat{\beta}y)v = -\eta_x, \tag{1.2.82}$$

$$\hat{\epsilon}\frac{Dv}{Dt} + (1+\hat{\beta}y)u = -\eta_y, \tag{1.2.83}$$

$$\frac{D\eta}{Dt} + (\eta + \frac{\hat{s}}{\hat{\epsilon}})(u_x + u_y) = 0, \tag{1.2.84}$$

where we omit $'$ for simplicity and D/Dt is two-dimensional Lagrange derivative given by

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y}. \tag{1.2.85}$$

Now we consider the scaling regime :

$$\hat{\epsilon} \sim \hat{\beta}^2, \quad \hat{s} \sim \hat{\beta}, \quad \hat{\beta} \ll 1. \tag{1.2.86}$$

That imposes that scale of the Red spot is larger than scale of flow of the atmosphere, i.e. $L \gg \lambda_{\mathrm{R}}$, from eq.(1.2.78).

We expand the variables via $\hat{\beta}$, i.e.

$$u \sim u^{(0)} + \hat{\beta}u^{(1)} + \hat{\beta}^2 u^{(2)} + \mathcal{O}(\hat{\beta}^3), \tag{1.2.87}$$

$$v \sim v^{(0)} + \hat{\beta}v^{(1)} + \hat{\beta}^2 v^{(2)} + \mathcal{O}(\hat{\beta}^3), \tag{1.2.88}$$

$$\eta \sim \eta^{(0)} + \hat{\beta}\eta^{(1)} + \hat{\beta}^2 \eta^{(2)} + \mathcal{O}(\hat{\beta}^3), \tag{1.2.89}$$

and substitute into the eq.(1.2.82)-(1.2.84). Notice that the order of $\hat{\epsilon}$ is same to $\hat{\beta}^2$, we find the leading order is

$$\mathcal{O}(\hat{\beta}^0) : u^{(0)} = -\eta_y^{(0)}, \quad v^{(0)} = \eta_x^{(0)}, \tag{1.2.90}$$

$$\frac{D^{(0)}\eta^{(0)}}{Dt} + \eta^{(0)}(u_x^{(0)} + v_y^{(0)}) + \frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}(u_x^{(1)} + v_y^{(1)}) = 0, \tag{1.2.91}$$

where $D^{(0)}/Dt$ means

$$\frac{D^{(0)}}{Dt} = \frac{\partial}{\partial t} + u^{(0)}\frac{\partial}{\partial x} + v^{(0)}\frac{\partial}{\partial y}.$$

The relation eq.(1.2.90) expresses the geostrophic balance and we reduce these relations to

$$u_x^{(0)} + v_y^{(0)} = 0. \tag{1.2.92}$$

Using this equation, eq.(1.2.91) becomes

$$\frac{D^{(0)}\eta^{(0)}}{Dt} + \frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}(u_x^{(1)} + v_y^{(1)}) = 0. \tag{1.2.93}$$

The next order is

$$\mathcal{O}(\hat{\beta}^1) : -\hat{\beta}v^{(1)} - \hat{\beta}yv^{(0)} = -\hat{\beta}\eta_x^{(1)}, \quad \hat{\beta}u^{(1)} + \hat{\beta}yu^{(0)} = -\hat{\beta}\eta_y^{(1)}, \tag{1.2.94}$$

$$\hat{\beta}\frac{D^{(0)}\eta^{(1)}}{Dt} + \frac{D^{(1)}\eta^{(0)}}{Dt} + \frac{\hat{s}\hat{\beta}^2}{\hat{\epsilon}}(u_x^{(2)} + v_y^{(2)}) + \eta^{(0)}\hat{\beta}(u_x^{(1)} + v_y^{(1)}) = 0, \tag{1.2.95}$$

where we use eq.(1.2.92). From the eq.(1.2.94), we obtain

$$u_x^{(1)} + v_y^{(1)} = -v^{(0)}. \tag{1.2.96}$$

The continuous equation eq.(1.2.91) becomes

$$\frac{D^{(0)}\eta^{(0)}}{Dt} + \frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}(-v^{(0)}) = 0, \tag{1.2.97}$$

$$\Rightarrow \left(\frac{\partial}{\partial t} - \frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}\frac{\partial}{\partial x}\right)\eta^{(0)} = 0, \tag{1.2.98}$$

where we use eq.(1.2.90). Since we are interested in the long scale phenomena, we introduce a new variable which describes the long time scale T :

$$T = \hat{\beta}t, \quad \hat{\beta} \ll 1. \tag{1.2.99}$$

By using this variable, the equation eq.(1.2.95) becomes to the form

$$\frac{D^{(0)}\eta^{(1)}}{Dt} + \left(\frac{\partial\eta^{(0)}}{\partial T} + u^{(1)}\eta_x^{(0)} + v^{(1)}\eta_y^{(0)}\right) + \frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}(u_x^{(2)} + v_y^{(2)}) + \eta^{(0)}(u_x^{(1)} + v_y^{(1)}) = 0. \tag{1.2.100}$$

The second order is

$$\mathcal{O}(\hat{\beta}^2) : \hat{\epsilon}\frac{D^{(0)}u^{(0)}}{Dt} - \hat{\beta}^2 y v^{(1)} - \hat{\beta}^2 v^{(2)} = -\hat{\beta}^2\eta_x^{(2)}, \tag{1.2.101}$$

$$\hat{\epsilon}\frac{D^{(0)}v^{(0)}}{Dt} + \hat{\beta}^2 y u^{(1)} + \hat{\beta}^2 u^{(2)} = -\hat{\beta}^2\eta_y^{(2)}. \tag{1.2.102}$$

From these two equations, we obtain

$$u_x^{(2)} + v_y^{(2)} = \frac{\partial}{\partial t}\nabla^2\eta^{(0)} + J[\eta^{(0)}, \nabla^2\eta^{(0)}] + 2yv^{(0)} - \eta_x^{(1)}, \tag{1.2.103}$$

where we defined ∇^2 and $J[A, B]$ as follows

$$\nabla^2 \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}, \tag{1.2.104}$$

$$J[A, B] \equiv \frac{\partial A}{\partial x}\frac{\partial B}{\partial y} - \frac{\partial A}{\partial y}\frac{\partial B}{\partial x}. \tag{1.2.105}$$

By using eq.(1.2.90) and (1.2.94), we can reduce the first and the second term of eq.(1.2.100) to the form

$$\begin{aligned}
&\frac{D^{(0)}\eta^{(1)}}{Dt} + \left(\frac{\partial\eta^{(0)}}{\partial T} + u^{(1)}\eta_x^{(0)} + v^{(1)}\eta_y^{(0)}\right) \\
&= \frac{\partial\eta^{(1)}}{\partial t} + u^{(0)}\eta_x^{(1)} + v^{(1)}\eta_y^{(1)} + \frac{\partial\eta^{(0)}}{\partial T}u^{(1)}\eta_x^{(0)} + v^{(1)}\eta_y^{(0)} \\
&= \frac{\partial\eta^{(1)}}{\partial t} - \eta_y^{(0)}\eta_x^{(1)} + \eta_x^{(0)}\eta_y^{(1)} + \frac{\partial\eta^{(0)}}{\partial T} + (-\eta_y^{(1)} - yu^{(0)})\eta_x^{(0)} + (\eta_x^{(1)} - yv^{(0)})\eta_y^{(0)} \\
&= \frac{\partial\eta^{(1)}}{\partial t} + \frac{\partial\eta^{(0)}}{\partial T}.
\end{aligned} \tag{1.2.106}$$

Therefore, eq.(1.2.100) finally becomes

$$\underbrace{\frac{D^{(0)}\eta^{(1)}}{Dt}+\left(\frac{\partial\eta^{(0)}}{\partial T}+u^{(1)}\eta_x^{(0)}+v^{(1)}\eta_y^{(0)}\right)}_{\text{eq.(1.2.106)}}+\frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}\underbrace{(u_x^{(2)}+v_y^{(2)})}_{\text{eq.(1.2.103)}}+\eta^{(0)}\underbrace{(u_x^{(1)}+v_y^{(1)})}_{=0}=0,$$

$$\Rightarrow\frac{\hat{\beta}}{\hat{s}}\frac{\partial\eta^{(1)}}{\partial t}-\frac{\hat{\beta}^2}{\hat{\epsilon}}\eta_x^{(1)}+\frac{\hat{\beta}}{\hat{s}}\frac{\partial\eta^{(0)}}{\partial T}-\frac{\partial}{\partial t}\nabla^2\eta^{(0)}-J[\eta^{(0)},\nabla^2\eta^{(0)}]+\frac{2\hat{\beta}^2}{\hat{\epsilon}}y\eta_x^{(0)}-\frac{\hat{\beta}}{\hat{s}}\eta^{(0)}\eta_x^{(0)}=0. \tag{1.2.107}$$

In this equation, $\eta^{(1)}$ is the higher order variable than $\eta^{(0)}$ so we neglect the first and the second term and replace the 4th term by eq.(1.2.99). In other words, we remove the fast variable t from the equation. We finally obtain the governing equation of the form

$$\frac{\hat{\beta}}{\hat{s}}\frac{\partial\eta^{(0)}}{\partial T}-\frac{\hat{\beta}}{\hat{s}}\eta^{(0)}\eta_x^{(0)}-\frac{\hat{s}\hat{\beta}}{\hat{\epsilon}}\frac{\partial}{\partial x}\nabla^2\eta^{(0)}+\frac{2\hat{\beta}^2}{\hat{\epsilon}}y\eta_x^{(0)}-J[\eta^{(0)},\nabla^2\eta^{(0)}]=0. \tag{1.2.108}$$

By introducing the new scale factors given by

$$\hat{\epsilon}\equiv S\hat{\beta}^2,\quad \hat{s}\equiv E\hat{\beta},\quad T\to\frac{S}{E}T,\quad S\sim E\sim\mathcal{O}(1), \tag{1.2.109}$$

we can rewrite the equation

$$\frac{\partial\eta}{\partial T}-\frac{E}{S}\eta\eta_x-S\frac{\partial}{\partial x}\nabla^2\eta+2y\eta_x+EJ[\nabla^2\eta,\eta]=0, \tag{1.2.110}$$

where we omit the subscript $^{(0)}$ from the equation (1.2.108). This is the Williams-Yamagata-Flierl equation. We are interested in this equation and its mathematical mechanism.

1.3 Numerical calculation

In this section, we show you the methods of numerical calculation. It is necessary to learn these skills because they often give us consideration beyond calculation by hand. However, we always should have distrust and check its result by ourself. As long as we keep that in mind, these techniques are useful.

As listed in contents above, we explain the five number of methods as follows.

1. 4th order Runge-Kutta
2. Simpson method
3. Fast fourier transform
4. Shooting method
5. About the Jacobian $J[A, B]$

Since they are basic schemes, we recommend that you carry out these programs by yourself. Let's begin.

1.3.1 4th order Runge-Kutta method

In order to solve a problem in Physics, we often face to the differential equations. There are various types and they have various properties. However, we finally solve the ordinary differential equation. In this section, we explain *the Runge-kutta method* which is employed in many situations. Before explanation, we note some rules. In order to deal with continuos function as discrete data, we have to express the function other form. One of the way is that we choose the sample points and calculate the value of function on it. For example,

$$\text{sample points} : t_0, t_1, t_2, \dots \qquad (t_{n+1} = t_n + \Delta t), \tag{1.3.1}$$

$$\text{value of function} : f(t_0), f(t_1), f(t_2), \dots, \tag{1.3.2}$$

where $f(t_n)$ means value of function in time $t = t_n$. In particular, we write function value in $t = t_n$ as f_n, i.e. $f_n = f(t_n)$. In general, ordinary differrential equation is written in the form as follows

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}, t), \quad \text{initial conditon} : \vec{x}(0) = \vec{x}_0. \tag{1.3.3}$$

Solving the ordinary differential equation means that we decide the value of $\vec{x}(t)$ from $\vec{x}(0)$. In the case of numerical method, we divide t into discrete values $t_0, t_1, t_2, \dots$. We use them as sample points and value of function f_n. The basic idea of the Runge-Kutta method is Taylor expansion given by

$$x_{n+1} = x_n + hx'_n + \cdots \frac{h^q}{q!} x_n^{(q)}, \tag{1.3.4}$$

where $h = \Delta t$ and $f^{(q)}$ is n-th derivative of f. Using chain rule, we obtain

$$\frac{d}{dt} = \frac{\partial}{\partial t} + \frac{dx}{dt}\frac{\partial}{\partial t} = \frac{\partial f}{\partial t} + f\frac{\partial}{\partial x}. \tag{1.3.5}$$

Therefore, we can calculate x_{n+1} in eq.(1.3.4) by using following relations

$$x' = \frac{dx}{dt} = f, \quad x'' = \frac{dx'}{dt} = \frac{df}{dt} = f_t + ff_x, \quad x^{(q)} = \left(\frac{\partial}{\partial t} + f\frac{\partial}{\partial x}\right)^{q-1} f. \tag{1.3.6}$$

However, it is too cost to calculate higher derivatives. Then, we calculate $f(x,t)$ variously and make a linear combination as to match it to Taylor expansion in higher order.

In Runge-Kutta method, we calculate value x_{n+1}, i.e. the value of x in $t_{n+1} = t_n + h$, by following formula

$$x_{n+1} = x_n + \sum_{i=1}^{p} w_i k_i, \tag{1.3.7}$$

$$k_i = hf\left(x_n + \sum_{j=1}^{p} \alpha_{ij} k_j, t_n + c_i h\right), c_i = \sum_{j=1}^{p} \alpha_{ij}, \tag{1.3.8}$$

where w_i is weight for each k_i. From this formula, we can expect x_{n+1} by p terms of k_i multiplied by w_i and this formula is implicit. To restrict a explicit formula, we employ a condition to α given by

$$\alpha_{ij} = 0 \quad (i \leq j). \tag{1.3.9}$$

This condition reduce the formula to useful form which need only previous value f_n to calculate f_{n+1}. That is k_i are given by

$$k_1 = hf(x,t), \tag{1.3.10}$$
$$k_2 = hf(x + \alpha_{21}k_1, t + c_2 h), \tag{1.3.11}$$
$$k_3 = hf(x + \alpha_{31}k_1 + \alpha_{32}k_2, t + c_3 h), \tag{1.3.12}$$
$$k_4 = hf(x + \alpha_{41}k_1 + \alpha_{42}k_2 + \alpha_{43}k_3, t + c_4 h), \tag{1.3.13}$$
$$\vdots$$

We expand these relations by h and require them matching to Taylor expansion for 4th order. After considerably algebra[*6], we finally obtain the 4th order Runge-Kutta formula as followings

[*6] Firstly, we expand the relations eq.(1.3.10)-(1.3.13). Second step, we compare coefficients at same order. Then, we obtain the coupled equations and solve them. Details of derivation are shown in reference [11].

$$k_1 = hf(x_n, t_n), \tag{1.3.14}$$
$$k_2 = hf\left(x_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right), \tag{1.3.15}$$
$$k_3 = hf\left(x_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right), \tag{1.3.16}$$
$$k_4 = hf\left(x_n + k_3, t_n + h\right), \tag{1.3.17}$$
$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{1.3.18}$$

This formula is standard in numerical calculation. There are two reasons. First, if we require the higher order formula, i.e. 5th or more, it is very complex and not all of them are more precise than 4th formula. Second, higher formula is suppoted by the differentiability of function. So we cann't secure the precision of it. Therefore, we often use 4th formula and it is sufficient in almost cases.

We show sample code of 4th order Runge-Kutta method in space two dimensions. This code is written by explicit space-differencing expression. If you would to improve accuracy of its numerical results, you should try to rewrite this code and make it better (e.g. rewrite to implicit expression or employ the Fourier spectral method). The functions which we don't show in this code are shown in section3, so please refer to that.

Listing 1.1: The Runge-Kutta algorithm

```
#include "Myheader.h"
#include <iostream>
#include <chrono>

void judgetypeofgauss(int typeofgauss, int indexofa, double *a, double *b);
void iniparameterset(int nofgauss, double *a, double *b, int *indexofgauss);
int numberofcx(double cx,double dx);
int numberofcy(double cy, double dy);

void geoadvortex(int *xnum, int *ynum, double *chx, double *chy,
  double **eta, double **ddeta, double **jj);
void equation(int *xnum, int *ynum, double *alpha, double *alpha2,
  double *epsilon_Jac, double *chx, double *chy, double *E, double *S,
  double **eta, double **dxeta, double **ddeta, double **jj, double **equ);
void xbcset(double **eta, int j, int xnum);
void ybcset(double **eta, int i, int ynum, int bc, double epsilon_shear,
  double alpha, double alpha2, double dy, double dy2);

void detectpeak(int ii, int jj ,double **eta, double *x, double *y, double dx,
  double dy, double *etamax, double *xofmax, double *yofmax, int cyclontype);

int main(){

  int i,j,k,l,ii,kk,xnum,ynum,ite,itenumber,bc,nofgauss,cyclontype;
  int *indexofgauss;
  double alpha,alpha2,epsilon_shear,epsilon_Jac,chx,chy,E,S;
```

```
27  double dx, dy, dy2, dt,t,time,argx,argy,ex,func;
28  double *cx, *cy, *a, *b, *offsetx, *offsety;
29  double **eta,**dxeta,**ddeta,**jj,**equ,**eta0,**eta1,**eta2,**eta3,**eta4;
30  double b1,b2,b3,b4,b5,b6,b7;
31  double etamax1,xofmax1,yofmax1,etamax2,xofmax2,yofmax2;
32
33  /*definition of parameters*/
34
35  // Parameters1
36  /*
37        xnum = 200; ynum = 100;
38        chx = 20; chy = 10;
39        alpha = 10.0; alpha2 = 10.0;
40  */
41  // Parameters
42  xnum=NX;ynum=NY;
43  chx=20;chy=10;
44  //alpha = 10.0; alpha2 = 10.0;
45  alpha = 9.6875; alpha2 = 9.6875; //2021/02/28 add
46
47  // Parameters2
48  /*
49        xnum = 400; ynum = 200;
50        chx = 20; chy = 10;
51        alpha = 10.0; alpha2 = 10.0;
52        smthnum = 1;
53  */
54
55  // E = 0.05; S = 0.01; /* Big vortex */
56      E = 2.0; S = 0.5; /* Medium vortex */
57  // E = 25.0; S = 20.0;/* Small vortex */
58
59
60        // Coordinates
61        for (i = 0; i <= xnum + 4; ++i) {
62              cx[i] = -chx + dx * (i - 2); // -20.3125 < x < 20.3125
63        }
64        for (j = 0; j <= ynum; ++j) {
65              cy[j] = -chy + dy * j; // -10.0000 < y < 10.0000
66        }
67
68      /*
69      --parameters--
70      a[]...hight of Gaussian
71      b[]...width of Gaussian
72      offsetx[]...offset in x-direction
73      offsety[]...offset in y-direction
74
75      -----initialize the parameters-----
76      nofgauss : number of lump solutions
77      indexofgauss[1..nofgauss]:select the types of lump solutions
78      1 --> c=1.0
79      2 --> c=2.0 3 --> c=2.2
80      4 --> c=3.0 5 --> c=3.3
81      6 --> c=4.0 7 --> c=4.4
```

```
82      ex) If you set indexofgauss[1]=1 and indexofgauss[2]=4, first lump solution
83       is c=1.0 and second one is c=3.0. Total number of lump solution is two.
84       If you would to set one lump solution, you have to set nofgauss=1.
85
86      (*)If you would set three or more lump solutions, you have to do followings
87      a)set nofgauss=N(N is number of lump solution),
88      b)add the function:detectpeak() to main code,
89      c)change the format which write down to FILE:peak-trajectory.
90      */
91      itenumber=100000;
92
93      nofgauss=1;
94      indexofgauss=ivector(1,nofgauss);
95      offsetx=dvector(1,nofgauss);
96      offsety=dvector(1,nofgauss);
97      a=dvector(1,2*nofgauss);
98      b=dvector(1,2*nofgauss);
99
100     //set the type of lump solutions
101     indexofgauss[1]=1;
102     indexofgauss[2]=1;
103     //indexofgauss[3]=3;
104
105     iniparameterset(nofgauss,a,b,indexofgauss);
106     // refer to section3 to find definition of function.
107
108     //set the offset in x and y directions
109     /*
110     offsetx[k]:offset of x direction for k-lump solution
111     offsety[k]:offset of xydirection for k-lump solution
112     b1-b7:these parameters are obtained by fitting (LM method).
113     These decide the width of lump solutions.
114     */
115     b1=2.63475; b2=1.863548; b3=1.776844; b4=1.521279;
116     b5=1.45066; b6=1.317375; b7=1.25608;
117
118     offsetx[1]=0.0; offsety[1]=0.0;
119     //offsetx[2]=-10.0; offsety[2]=0.0;
120     //offsetx[3]=0.0; offsety[2]=0.0;
121
122     /*
123     Set initial condition eta[i][j].
124
125     Boundary conditions
126     x-direction:periodic
127     y-direction:
128     bc=0->free boundary
129     bc=1->with shear flow
130
131     Cyclontype of lump solution
132     cyclontype=1 -> anticyclonic (amplitude is positive)
133     cyclontype=-1 -> cyclonic (amplitude is negative)
134
135     Shear flow type
136     epsilon_shear:
```

```
    epsilon_shear=-1 -> anticyclonic (amplitude is positive)
    epsilon_shear=1 -> cyclonic (amplitude is negative)
    this parameter represents type of shear flow.
    (*)the boundary conditions are calculated with shear flow,
    so you don't have to change it.
    */

    bc=0;
    //bc=1;

    cyclontype=1;
    //cyclontype=-1;

    epsilon_shear=-1.0;

    for (i = 0; i <= xnum+4; i++) {
        eta[i][j]=0.0;
          for (j = 0; j <= ynum; j++) {
                for (k=1,l=1;k<=nofgauss;k++) {
                      argx=(cx[i]-offsetx[k])/b[l];
                      argy=(cy[j]-offsety[k])/b[l];
                      ex=exp(-argx*argx-argy*argy);
                      func=cyclontype*a[l]*ex;
                      eta[i][j]+=func;
                      l++;
                      argx=(cx[i]-offsetx[k])/b[l];
                      argy=(cy[j]-offsety[k])/b[l];
                      ex=exp(-argx*argx-argy*argy);
                      func=cyclontype*a[l]*ex;
                      eta[i][j]+=func;
                      l++;
                }
                      eta[i][j]+=0.5*epsilon_shear*bc*cy[j]*cy[j]; //shear flow
          }
    }
    fclose(data02);
    free_dvector(b,1,2*nofgauss);
    free_dvector(a,1,2*nofgauss);
    free_dvector(offsety,1,nofgauss);
    free_dvector(offsetx,1,nofgauss);
    free_ivector(indexofgauss,1,nofgauss);

for (j = 2; j <= ynum - 2; ++j) { xbcset(eta,j,xnum); }
for (i = 0; i <= xnum + 4; ++i) { ybcset(eta,i,ynum,bc,epsilon_shear,alpha,
  alpha2,dy,dy2); }
  // refer to section3 to find definition of function.









  // detect peak
  //1st peak
```

```
192     ii=numberofcx(offsetx[1],dx);
193     kk=numberofcy(offsety[1],dy);
194     printf("␣␣1st␣peak\n");
195     detectpeak(ii,kk,eta,cx,cy,dx,dy,&etamax1,&xofmax1,&yofmax1,cyclontype);
196     // refer to section3 to find definition of function.
197     printf("␣␣Local␣maximum␣:␣(%6.4lf␣,␣%6.4lf␣;␣%6.4lf)\n",xofmax1,yofmax1,etamax1
            );
198     //2nd peak
199     ii=numberofcx(offsetx[2],dx);
200     kk=numberofcy(offsety[2],dy);
201     printf("␣␣2nd␣peak\n");
202     detectpeak(ii,kk,eta,cx,cy,dx,dy,&etamax2,&xofmax2,&yofmax2,cyclontype);
203     printf("␣␣Local␣maximum␣:␣(%6.4lf␣,␣%6.4lf␣;␣%6.4lf)\n",xofmax2,yofmax2,etamax2
            );
204     printf("\n");
205
206     ////////////////////////////////////////////////////////////////////////
207     //Main integral
208         ite = 0;
209         while (ite <= itenumber) {
210         // Runge-Kutta method
211         /*-------------------------------------------------------------------*/
212         // step 1
213         equation(&xnum, &ynum, &alpha, &alpha2, &epsilon_Jac, &chx, &chy,
214           &E, &S, eta, dxeta, ddeta, jj, equ);
215
216            for (i = 2; i <= xnum + 2; ++i) {
217               for (j = 2; j <= ynum - 2; ++j) {
218                  eta1[i][j] = dt * equ[i][j];
219                  eta0[i][j] = eta[i][j] + 0.5*eta1[i][j];
220                }
221              }
222           //Boundary conditions
223           //x-direction:periodic
224           //y-direction: bc=0->shearflow bc=1->free boundary
225            for (j = 2; j <= ynum - 2; ++j) { xbcset(eta0, j, xnum); }
226            for (i = 0; i <= xnum + 4; ++i) { ybcset(eta0,i,ynum,bc,epsilon_shear,
227              alpha,alpha2,dy,dy2); }
228         /*-------------------------------------------------------------------*/
229         /*-------------------------------------------------------------------*/
230         // step 2
231         equation(&xnum, &ynum, &alpha, &alpha2, &epsilon_Jac, &chx, &chy,
232           &E, &S, eta0, dxeta, ddeta, jj, equ);
233
234            for (i = 2; i <= xnum + 2; ++i) {
235               for (j = 2; j <= ynum - 2; ++j) {
236                  eta2[i][j] = dt * equ[i][j];
237                  eta0[i][j] = eta[i][j] + 0.5*eta2[i][j];
238                }
239              }
240           //Boundary conditions
241           //x-direction:periodic
242           //y-direction: bc=0->shearflow bc=1->free boundary
243            for (j = 2; j <= ynum - 2; ++j) { xbcset(eta0, j, xnum); }
244            for (i = 0; i <= xnum + 4; ++i) { ybcset(eta0,i,ynum,bc,epsilon_shear,
```

```
245             alpha,alpha2,dy,dy2); }
246         /*-------------------------------------------------------------------*/
247         /*-------------------------------------------------------------------*/
248         // step 3
249         equation(&xnum, &ynum, &alpha, &alpha2, &epsilon_Jac, &chx, &chy,
250           &E, &S, eta0, dxeta, ddeta, jj, equ);
251
252           for (i = 2; i <= xnum + 2; ++i) {
253              for (j = 2; j <= ynum - 2; ++j) {
254                 eta3[i][j] = dt * equ[i][j];
255                 eta0[i][j] = eta[i][j] + eta3[i][j];
256               }
257             }
258           //Boundary conditions
259           //x-direction:periodic
260           //y-direction: bc=0->shearflow bc=1->free boundary
261            for (j = 2; j <= ynum - 2; ++j) { xbcset(eta0, j, xnum); }
262            for (i = 0; i <= xnum + 4; ++i) { ybcset(eta0,i,ynum,bc,epsilon_shear,
263              alpha,alpha2,dy,dy2); }
264         /*-------------------------------------------------------------------*/
265         /*-------------------------------------------------------------------*/
266         // step 4
267         equation(&xnum, &ynum, &alpha, &alpha2, &epsilon_Jac, &chx, &chy,
268           &E, &S, eta0, dxeta, ddeta, jj, equ);
269
270            for (i = 2; i <= xnum + 2; ++i) {
271              for (j =2; j <= ynum - 2; ++j) {
272                 eta4[i][j] = dt * equ[i][j];
273               }
274             }
275
276             for (i = 2; i <= xnum + 2; ++i) {
277               for (j = 2; j <= ynum - 2; ++j) {
278                  //oneosix=1/6
279                  eta[i][j] += oneosix *(eta1[i][j]+2.0*eta2[i][j]+2.0*eta3[i][j]+
                         eta4[i][j]);
280               }
281             }
282           //Boundary conditions
283           //x-direction:periodic
284           //y-direction: bc=0->shearflow bc=1->free boundary
285            for (j = 2; j <= ynum - 2; ++j) { xbcset(eta, j, xnum); }
286            for (i = 0; i <= xnum + 4; ++i) { ybcset(eta,i,ynum,bc,epsilon_shear,
287              alpha,alpha2,dy,dy2); }
288         /*-------------------------------------------------------------------*/
289         // Plot
290                if (ite % 500 == 0) { /*file export*/ } //plot
291                ++ite;
292                       }//while{}
293 }
294
295 void geoadvortex(int *xnum, int *ynum, double *chx, double *chy, double **eta,
296   double **ddeta, double **jj) {
297         int i, j;
298         double dx, dy, dum,dumx,dumy;
```

```
299
300         dx=2*(*chx)/(*xnum); //=0.15625
301         dy=2*(*chy)/(*ynum); //=0.15625
302         dumx=1.0/dx;
303         dumy=1.0/dy;
304         dum=1.0/12.0;
305
306   for (i = 1; i <= *xnum + 3; ++i){
307     for (j = 1; j <= *ynum - 1; ++j){
308        /* dumx=1/dx dumy=1/dy */
309           ddeta[i][j]=(eta[i+1][j]-2.0*eta[i][j]+eta[i-1][j])*dumx*dumx
310                       +(eta[i][j+1]-2.0*eta[i][j]+eta[i][j-1])*dumy*dumy;
311                }
312         }
313   for (i = 2; i <= *xnum + 2; ++i){
314     for (j = 2; j <= *ynum - 2; ++j){
315               /*J[\nabla^2 \eta, \eta]:vorticity advection term.
316               J=(J++ + J+* + J*+)/3
317               dum=1/12, 1/dx=dumx, 1/dy=dumy
318               */
319      jj[i][j] = dum*( ( (ddeta[i+1][j]-ddeta[i-1][j])*(eta[i][j+1]-eta[i][j-1])
320               -(ddeta[i][j+1]-ddeta[i][j-1])*(eta[i+1][j]-eta[i-1][j]) )
321               +( ddeta[i+1][j]*(eta[i+1][j+1]-eta[i+1][j-1])
322               -ddeta[i-1][j]*(eta[i-1][j+1]-eta[i-1][j-1])
323               -ddeta[i][j+1]*(eta[i+1][j+1]-eta[i-1][j+1])
324               +ddeta[i][j-1]*(eta[i+1][j-1]-eta[i-1][j-1]) )
325               +( ddeta[i+1][j+1]*(eta[i][j+1]-eta[i+1][j])
326               -ddeta[i-1][j-1]*(eta[i-1][j]-eta[i][j-1])
327               -ddeta[i-1][j+1]*(eta[i][j+1]-eta[i-1][j])
328               +ddeta[i+1][j-1]*(eta[i+1][j]-eta[i][j-1]) ) )*dumx*dumy;
329                }
330         }
331   return;
332 }
333
334 /*the r.h.s. of WYF equation*/
335 void equation(int *xnum, int *ynum, double *alpha, double *alpha2,
336   double *epsilon_Jac, double *chx, double *chy, double *E, double *S,
337   double **eta, double **dxeta, double **ddeta, double **jj, double **equ) {
338
339         int i, j;
340         double dx, dy, ddetax, etacor;
341         double dum1,dum2,dum3,dum4,dum5,dum6;
342         double *cy;
343
344         cy=dvector(0, *ynum + 1);
345
346         dx=2*(*chx)/(*xnum);
347         dy=2*(*chy)/(*ynum);
348         dum1=0.5/dx;
349         dum2=1.0/dy;
350         dum3=4.0*dum1*dum1;
351         dum4=dum2*dum2;
352         dum5=1./9.;
353         dum6=(*E)/(*S);
```

```
  //dum1=0.5/dx
  //dum2=1.0/dy
  //dum3=1.0/dx/dx=4.0*0.5/dx*0.5/dx
  //dum4=1.0/dy/dy
  //dum5=1/9
  //dum6=E/S

  for (j=0;j<=(*ynum);++j) { cy[j]=-1.0*(*chy)+dy*j; }
  for (i=1;i<=(*xnum)+3;++i) {
      for (j=1;j<=(*ynum)-1;++j) {
        //dum1=0.5/dx
        dxeta[i][j]=dum1*(eta[i+1][j]-eta[i-1][j]);
      }
    }

        geoadvortex(xnum, ynum, chx, chy, eta, ddeta, jj);

  for (i=2;i<=(*xnum)+2;++i) {
      for (j=2;j<=(*ynum)-2;++j) {
        //dum3=1.0/dx/dx=4.0*0.5/dx*0.5/dx
        ddetax=dum3*(dxeta[i+1][j]-2.0*dxeta[i][j]+dxeta[i-1][j])
             +dum4*(dxeta[i][j+1]-2.0*dxeta[i][j]+dxeta[i][j-1]);
              //dum4=1.0/dy/dy
              //dum5=1/9
        etacor=dum5*(eta[i+1][j+1]+eta[i][j+1]+eta[i-1][j+1]
               +eta[i+1][j]+eta[i][j]+eta[i-1][j]
               +eta[i+1][j-1]+eta[i][j-1]+eta[i-1][j-1]);
        equ[i][j]=dum6*etacor*dxeta[i][j]-2.0*cy[j]*dxeta[i][j]
                 +(*S)*ddetax-(*E)*jj[i][j];
               }
        }
        free_dvector(cy, 0, *ynum + 1);
  return;
}
```

1.3.2 Simpson method

We often get a situation that we desire a integral of function. Further, it is hardly to integrate exactly. In this section, we will show you the method to obtain integrals numerically. Using this formula, we can integrate most of functions which are well defined.

The problem is given by

$$I = \int_{x_0}^{x_2} f(x)dx. \tag{1.3.19}$$

Our goal is finding value of integral I numerically. Also in this section, we divide cotinuous variable into discrete values, i.e. we divide x into $x_0, x_1, x_2, \cdots$.

The basic idea of Simpson formula is aprroximation by quadratic polynominal. In figure1.4, we plot $f(x)$ and quadratic polynominal $g(x)$. If we move x_2 closer to x_0, the precision of approximation is getting better. When the interval of coordinate $\Delta x = x_2 - x_0$ is small enough, $f(x)$ can be replaced by quadratic polynominal $g(x)$ as follows

$$I = \int_{x_0}^{x_2} f(x)dx \simeq \int_{x_0}^{x_2} g(x)dx = \int_{x_0}^{x_2} \left(Ax^2 + Bx + C\right) dx. \tag{1.3.20}$$

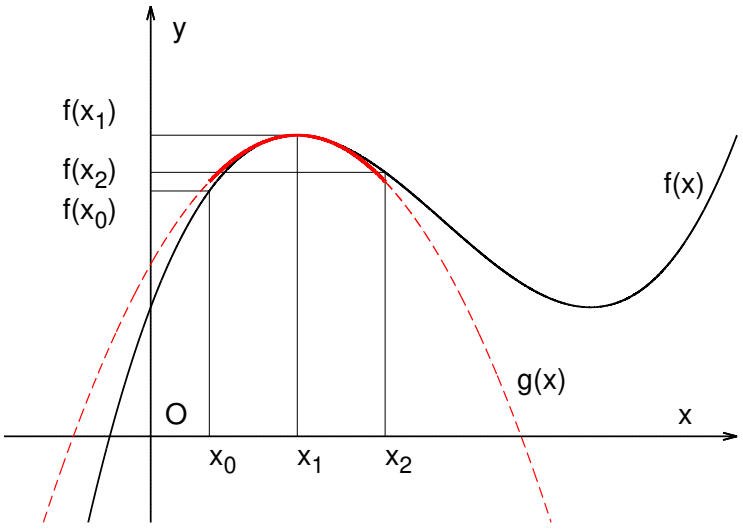


Figure1.4: Approximation by quadratic polynominal $g(x)$.

Calculate this integral and express its result by using x_0 and x_2, we obtain equation as folloings

$$\begin{aligned}
I &= \int_{x_0}^{x_2} f(x)dx \\
&\simeq \int_{x_0}^{x_2} \left(Ax^2 + Bx^{+}C\right) dx \\
&= \left[\frac{A}{3}x^3 + \frac{B}{2}x^2 + Cx\right]_{x_0}^{x_2} \\
&= \frac{A}{3}({x_2}^3 - {x_0}^3) + \frac{B}{2}({x_2}^2 - {x_0}^2) + C(x_2 - x_0) \\
&= \frac{A}{3}(x_2 - x_0)({x_2}^2 + x_2x_0 + {x_0}^2) + \frac{B}{2}(x_2 - x_0)(x_2 + x_0) + C(x_2 - x_0) \\
&= \frac{1}{6}(x_2 - x_0)[2A({x_2}^2 + x_2x_0 + {x_0}^2) + 3B(x_2 + x_0) + 6C]
\end{aligned}$$

$$\therefore \int_{x_0}^{x_2} f(x)dx \simeq \frac{1}{6}(x_2 - x_0)[2A({x_2}^2 + x_2x_0 + {x_0}^2) + 3B(x_2 + x_0) + 6C]. \tag{1.3.21}$$

Moreover, we introduce the new variable x_1 by

$$x_1 \equiv \frac{x_0 + x_2}{2}, \tag{1.3.22}$$

and calculate $f(x_0) + 4f(x_1) + f(x_2)$, then we obtain the relation as follows

$$\begin{aligned}
&f(x_0) + 4f(x_1) + f(x_2) \\
&= A{x_0}^2 + Bx_0 + C + 4\left[A\left(\frac{x_0 + x_2}{2}\right)^2 + B\left(\frac{x_0 + x_2}{2}\right) + C\right] + A{x_2}^2 + Bx_2 + C \\
&= A{x_0}^2 + Bx_0 + C + A(x_0 + x_2)^2 + 2B(x_0 + x_2) + 4C + A{x_2}^2 + Bx_2 + C \\
&= 2A({x_0}^2 + x_0x_2 + {x_2}^2) + 3B(x_0 + x_2) + 6C
\end{aligned}$$

$$\therefore f(x_0) + 4f(x_1) + f(x_2) = 2A({x_0}^2 + x_0x_2 + {x_2}^2) + 3B(x_0 + x_2) + 6C. \tag{1.3.23}$$

From eq.(1.3.21) and eq.(1.3.23), we finally obtain the simpson formula given by

$$I = \int_{x_0}^{x_2} f(x)dx \simeq \frac{1}{6}(x_2 - x_0)[f(x_0) + 4f(x_1) + f(x_2)]. \tag{1.3.24}$$

The precision of this formula is $\mathcal{O}(\Delta x^3),\ \Delta x \equiv x_2 - x_0$. This formula only needs $f(x_0), f(x_1)$ and $f(x_2)$, so it is easy to calculate the integral. However, it is restricted to the case of small Δx, so we generalize this formula to long range version.

In order to get a such expression, we consider the integral I given by

$$I = \int_a^b f(x)dx, \tag{1.3.25}$$

where $L = b - a$ is not necessarily small. We divide $[a, b]$ into $2m$ sample points and rewrite $a \equiv x_0,\ b \equiv x_{2m}\ \ (m : integer)$. Then, we apply simpson formula eq.(1.3.24) to x_n, x_{n+1} and

x_{n+2}. That is,

$$\int_{x_0}^{x_2} f(x)dx \simeq \frac{1}{6}(x_2 - x_0)[f(x_0) + 4f(x_1) + f(x_2)], \tag{1.3.26}$$

$$\int_{x_2}^{x_4} f(x)dx \simeq \frac{1}{6}(x_4 - x_2)[f(x_2) + 4f(x_3) + f(x_4)], \tag{1.3.27}$$

$$\vdots$$

$$\int_{x_{2m-2}}^{x_{2m}} f(x)dx \simeq \frac{1}{6}(x_{2m} - x_{2m2})[f(x_{2m-2}) + 4f(x_{2m-1}) + f(x_{2m})], \tag{1.3.28}$$

and make summations all of avobe per sides. Finally we obtain the expression of integral as follows

$$\begin{aligned}\left(\int_{x_0}^{x_2} + \int_{x_2}^{x_4} + \cdots + \int_{x_{2m-2}}^{x_{2m}}\right) f(x)dx = &\frac{1}{6}(\underline{x_0 + 2h - x_0})[f(x_0) + 4f(x_0 + h) + f(x_0 + 2h)] \\ &+ \frac{1}{6}(\underline{x_0 + 4h - (x_0 + 2h)})[f(x_0 + 2h) + 4f(x_0 + 3h) + f(x_0 + 4h)] \\ &+ \cdots \\ &+ \frac{1}{6}(\underline{x_0 + 2mh - (x_0 + 2(m-2)h)}) \\ &\times [f(x_0 + (2m-2)h) + 4f(x_0 + (2m-1)h) + f(x_0 + 2mh)],\end{aligned} \tag{1.3.29}$$

$$\therefore \int_{x_0}^{x_{2m}} f(x)dx = \frac{h}{3}\left[\sum_{k=1}^{m} f(x_0 + (2k-2)h) + \sum_{k=1}^{m} 4f(x_0 + (2k-1)h) + \sum_{k=1}^{m} f(x_0 + 2kh)\right], \tag{1.3.30}$$

where $h = (b-a)/2m$. This relation eq.(1.3.30) is also called *simpson formula*.

We show the sample code as below. It is not so difficult to calculate this integral. However, you should only pay attention to the index of arrays. In particular, on the border of range (or area), you should monitor how the index behave in algorithm.

Listing 1.2: The simpson formula algorithm

```c
#include <math.h>

void Simpson2dim(double **func, double *x, double *y, int xnum, int ynum,
     double *integral)
/*
When we set func[][], this function carries out integral and substitute
its results to integral. The size of matrix is func[0...xnum][0..ynum].
We suppose x[0]=xMin,x[xnum]=xMax,y[0]=yMin,y[ynum]=yMax,
dx=(xMax-xMin)/xnum and dy=(yMax-yMin)/ynum.
By using these dx,dy, we set x[i]=xMin+i*dx,y[j]=yMin+j*dy.
The value of func[][] is calculated on as x[i],y[j] as func[i][j].
If you would to change the integral range, you have to set the range of index
in i,j roop.
*/
{
    int i,j;
    double dx,dy;
    dx=(x[xnum]-x[0])/xnum;
    dy=(y[ynum]-y[0])/ynum;

    for (i=0;i<=(xnum-2)/2;i++) {
        for (j=0;j<=(ynum-2)/2; j++) {
          integral+=(func[2*i][2*j]+4.0*func[2*i+1][2*j]+func[2*i+2][2*j])
              +4.0*(func[2*i][2*j+1]+4.0*func[2*i+1][2*j+1]+func[2*i+2][2*j+1])
              +(func[2*i][2*j+2]+4.0*func[2*i+1][2*j+2]+func[2*i+2][2*j+2]);
        }
    }
    integral *= dx*dy/9.0;
}
```

1.3.3 FFT (Fast Fourier Transform)

In this section, we explain *Fourier Transform*. This transformation is used in many cases. For example, frequency analysis of sound wave, analysis of X-ray spectrum, removing the noize and so on. In order to use this techniques practically, we have to deel with discrete data again. We show this method as *Disctrete Fourier Transform* and *Fast Fourier Transform*.

First of all, the fourier transform and inverse are defined by

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt, \tag{1.3.31}$$

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} df, \tag{1.3.32}$$

where i is imaginary unit, f is frequency and $H(f)$ is called *the Fourier transform* of $h(t)$. In the fourier transform, there are many useful formula. However, we omit its explanation here. If you are interested in that, please refer to some literatures. We consider the problem : How can we get the fourier transform $H(f)$ from discrete data $h(t_k)$? In order to answer this question, we make discrete data from function $h(t)$ as followings

$$h_k = h(t_k), \quad t_k = k\Delta, \quad k = 0, 1, 2, \ldots, N-1, \tag{1.3.33}$$

where Δ is sampling time and N is any even number. From these data, we want to get the fourier transform $H(f_n)$ defined by

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t} dt, \tag{1.3.34}$$

where f_n are given by

$$f_n = \frac{n}{N\Delta}, \quad n = -\frac{N}{2}, \ldots, \frac{N}{2}. \tag{1.3.35}$$

Our work is to find $H(f_n)$ per each f_n. Using relations eq.(1.3.33) and eq.(1.3.35), we rewrite eq.(1.3.34) as discrete form below

$$H(f_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f_n t}\, dt \approx \sum_{k=0}^{N-1} h_k\, e^{2\pi i f_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N}. \tag{1.3.36}$$

Then, we rewrite final summation in eq.(1.3.36) to H_n

$$H_n = \sum_{k=0}^{N-1} h_k\, e^{2\pi i k n/N}. \tag{1.3.37}$$

We call H_n *discrete fourier transform* (DFT) and our goal is finding H_n from h_k. It is because that if we find H_n per each f_n, we can immediately get the fourier transform $H(f_n)$ as approximation form by

$$H(f_n) \approx \Delta H_n. \tag{1.3.38}$$

We will show *the Danielson and Lanczos remma* that is algorithm to calculate H_n very fast. Before considering that algorithm, we estimate the order of calculation roughly. In eq.(1.3.37), we introduce complex number W as

$$W \equiv e^{2\pi i/N}. \tag{1.3.39}$$

Using W, we rewrite eq.(1.3.37) to

$$H_n = \sum_{k=0}^{N-1} W^{nk}\, h_k. \tag{1.3.40}$$

In eq.(1.3.40), we calculate the product of matrix and vector. The matrix has $N \times N$ components and vector has N components, so the cost of calculation becomes $\mathcal{O}(N^2)$ roughly. However, the Danielson and Lanczos remma reduces its cost to $\mathcal{O}(N \log_2 N)$. Therefore, this algorithm is called *Fast Fourier Transform* (FFT). The basic idea of it is to divide the DFT of its length N into a pair of the DFT of its length $N/2$. The proof is given by followings

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ijk/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ijk/(N/2)} f_{2j+1} \\ &= F_k^e + W^k F_k^o, \end{aligned} \tag{1.3.41}$$

where W is the same to eq.(1.3.39) and subscript of F^e, F^o means *even* and *odd*. That is F_k^e (F_k^o) is k component of DFT which obtained from the even (odd) number components of f_j. This relation divide the length of DFT N into $N/2$. We can use eq.(1.3.41) inductively and get one component DFT finally. For example, we consider this algorithm for $N = 4$. First step, we divide F_k into F_k^e and F_k^o by using eq.(1.3.41)

$$F_k = F_k^e + W^k F_k^o, \tag{1.3.42}$$

where $k = 0, 1, 2, 3$. Second step, we divide F_k^e and F_k^o into half of length again

$$F_k^e = F_k^{ee} + W^k F_k^{eo}, \tag{1.3.43}$$
$$F_k^o = F_k^{oe} + W^k F_k^{oo}. \tag{1.3.44}$$

From eq.(1.3.41), if $N = 4$, we can finally write down the r.h.s of eq.(1.3.43) and eq.(1.3.44) to the followings

$$F_k^{ee} = f_0, \quad F_k^{eo} = f_2, \tag{1.3.45}$$
$$F_k^{oe} = f_1, \quad F_k^{oo} = f_3. \tag{1.3.46}$$

Using these relations oppositely, we can get the DFT in the case of $N = 4$. We show this process in Figure1.5.

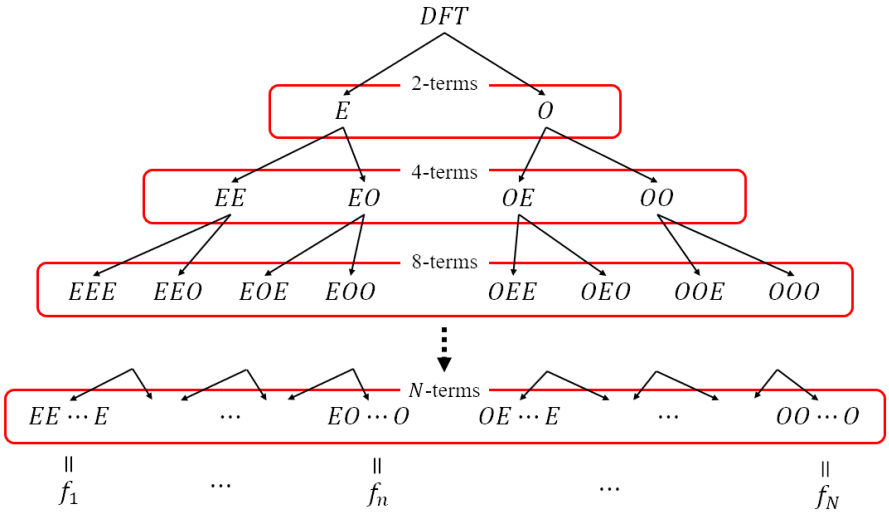


Figure1.5: The Danielson and Lanczos remma. The DFT of its length N is divided into the DFT of half length pairs. Finally we get the one component DFT and it is identical transform, i.e. $F_k^{eeo\cdots o} = f_n$.

The most important step is to decide the f_n in the final one component DFT as eq.(1.3.45) and eq.(1.3.46). It is easy to get f_n. We rearrange patterns of e, o in reverse and substitute $e = 0$, $o = 1$. We obtain the number n of f_n in binary number. For example, in eq.(1.3.45) and eq.(1.3.46), we can obtain the number n as following way

$$\begin{aligned} \text{corresponding} &\Rightarrow ee = 00,\ eo = 01,\ oe = 10,\ oo = 11 \\ \text{bit reverse} &\Rightarrow ee = 00,\ oe = 10,\ eo = 01,\ oo = 11 \\ \text{binary num.} &\Rightarrow f_0,\ f_2,\ f_1,\ f_3 \end{aligned}$$

$$\therefore F_k^{ee} = f_0,\ F_k^{eo} = f_2,\ F_k^{oe} = f_1,\ F_k^{oo} = f_3. \tag{1.3.47}$$

By using these techniques, we can obtain the algorithm for *Fast Fourier Transform* (FFT). This algorithm calculate the H_n in eq.(1.3.37) so we can find the components of wave number contained in input data.

The FFT routine given below is based on one originally written by N.M. Brenner. The input quantities are the number of complex data points **nn**, the data array **data[1,2,...,2*nn]**, and **isign** which should be set to either ± 1 and it is the sign of i in the exponential of eq.(1.3.37). When **isign** is set to -1, the routine calculates the inverse transform as follows

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n\, e^{-2\pi i k n/N}, \tag{1.3.48}$$

where it does not multiply by the normalizing factor $1/N$. Notice that the argument **nn** is the number of *complex* data points. The actual length of the real array **data[1,2,...,2*nn]** is two times **nn**, with each complex value occupying two consecutive location. In other words, **data[1]** is the real part of f_0, **data[2]** is the imaginary part of f_0, and so on up to **data[2*nn-1]**, which is the real part of f_{N-1}, and **data[2*nn]**, which is the imaginary part of f_{N-1}. The FFT routine gives back the F_n packed in exactly the same fashion, as **nn** complex numbers.

The real and imaginary parts of the zero frequency component F_0 are in **data[1]** and **data[2]** ; the smallest non-zero positive frequency has real and imaginary parts in **data[3]** and **data[4]** ; the smallest (in absolute value) non-zero negative frequency has real and imaginary parts in **data[2*-1]** and **data[2*nn]**. Positive frequencies increasing in magnitude are stored in the real-imaginary pairs **data[5]**, **data[6]** up to **data[nn-1]**, **data[nn]**. Negative frequencies of increasing magnitude are stored in **data[2*nn-3]**, **data[2*nn-2]** down to **data[nn+3]**, **data[nn+4]**. Finally, the pair **data[nn+1]**, **data[nn+2]** contain the real and imaginary parts of the one aliased point that contains the most positive and the most negative frequency.

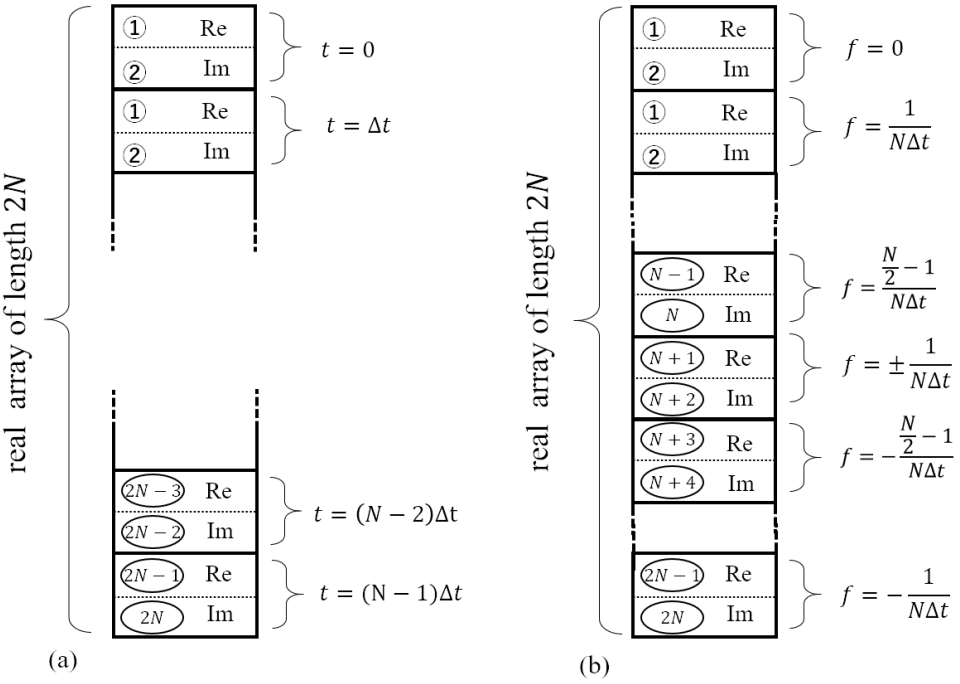


Figure1.6: Input and output arrays for FFT. (a) The input array contains N (a power of 2) complex time samples in a real array of length $2N$, with real and imaginary parts alternating. (b) The output array contains the complex Fourier spectrum at N values of frequency. Real and imaginary parts again alternate. The array stars with zero frequency, works up to the most positive frequency (which is ambiguous with the most negative frequency). Negative frequencies follow, from the second-most negative up to the frequency just below zero.

Listing 1.3: The fast fourier transform algorithm

```
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
#include <math.h>

void FFT1dim(float data[], unsigned long nn,int isign){
  unsigned long n,mmax,m,j,istep,i;
  double wtemp,wr,wpr,wpi,wi,theta;
  float tempr,tempi;
  //Algorithm for bit reversal
  n=nn << 1;
  j=1;
  for (i = 1; i < n; i+=2) {
    if (j>i) {
      SWAP(data[j],data[i]); //change two complex numbers
      SWAP(data[j+1],data[i+1]);
    }
    m=n >> 1;
    while (m>=2 && j>m) {
      j -= m;
      m >>=1;
    }
    j+=m;
  }
  mmax=2; //Danielson-Lanczos remma
  while (n>mmax) {
    istep=mmax << 1;
    theta=isign*(6.28318530717959/mmax);
    wtemp=sin(0.5*theta);
    wpr=-2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m = 1; m < mmax; m+=2) {
        for (i = m; i <= n; i+=istep) {
          j=i+mmax;
          tempr=wr*data[j]-wi*data[j+1];
          tempi=wr*data[j+1]+wi*data[j];
          data[j]=data[i]-tempr;
          data[j+1]=data[i+1]-tempi;
          data[i] += tempr;
          data[i+1] += tempi;
        }
          wr=(wtemp=wr)*wpr-wi*wpi+wr;
          wi=wi*wpr+wtemp*wpi+wi;
    }
    mmax=istep;
  }
}

void FFTndim(float data[],unsigned long nn[],int ndim,int isign){
  int idim;
  unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
  unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
  float tempi,tempr;
  double theta,wi,wpi,wpr,wr,wtemp;
```

```
  //count number of complex numbers
  for (ntot=1,idim=1; idim <= ndim; idim++)
    ntot *= nn[idim];
  nprev=1;
  for (idim=ndim; idim >= 1; idim--) {
    n=nn[idim];
    nrem=ntot/(n*nprev);
    ip1=nprev << 1;
    ip2=ip1*n;
    ip3=ip2*nrem;
    i2rev=1;
    for (i2=1; i2 <= ip2; i2+=ip1) {
      if (i2 < i2rev) {
        for (i1=i2; i1 <= i2+ip1-2; i1+=2) {
          for (i3=i1; i3 <= ip3; i3+=ip2) {
            i3rev=i2rev+i3-i2;
            SWAP(data[i3],data[i3rev]);
            SWAP(data[i3+1],data[i3rev+1]);
          }}
      }
      ibit=ip2 >> 1;
      while (ibit >= ip1 && i2rev > ibit) {
        i2rev -= ibit;
        ibit >>= 1;
      }
      i2rev += ibit;
    }
    ifp1=ip1;
    while (ifp1 < ip2) {
      ifp2=ifp1 << 1;
      theta=isign*6.28318530717959/(ifp2/ip1);
      wtemp=sin(0.5*theta);
      wpr=-2.0*wtemp*wtemp;
      wpi=sin(theta);
      wr=1.0; wi=0.0;
      for (i3=1; i3 <= ifp1; i3+=ip1) {
        for (i1=i3; i1 <= i3+ip1-2; i1+=2) {
          for (i2=i1; i2 <= ip3; i2+=ifp2) {
            k1=i2;
            k2=k1+ifp1;
            tempr=(float)wr*data[k2]-(float)wi*data[k2+1];
            tempi=(float)wr*data[k2+1]+(float)wi*data[k2];
            data[k2]=data[k1]-tempr;
            data[k2+1]=data[k1+1]-tempi;
            data[k1]+=tempr;
            data[k1+1]+=tempi;
          }}
        wr=(wtemp=wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
      }
      ifp1=ifp2;
    }
    nprev *= n;
  }
}
```

1.3.4 Shooting method

This method can solve two point boundary value problems. We explain that problems. When we solve the ordinary differential equation, we need the initial conditions as many as the order of the equation. However, we often face to the situation that we require the solution of differential equation to satisfy the additional condition at boundaries. These problems are typically called *two point boundary value problems*. The phrase "two point boundary problems" is used widely for the more complex problems. In this section, we deal with one of the simplist problem. If you want more information, please refer to some literatures.

In order to obtain the solution of ordinary differential equation, we start integral with any initial condition from starting point (which is one of the boundary) to final point (which is the other boundary)[*7]. However, it is very hard to obtain the solution which satisfy the boundary condition at final point. It is because that the choices of initial condition are infinite. Thus, we consider to adjust the initial condition repeatedly untill the boundary condition is satisfied. This is the idea of the shooting method.

For the standard two point boundary problem, we desire the solution to a set of N coupled 1st order ordinary differential equations[*8]. We write them as

$$\frac{dy_i(x)}{dx} = g_i(x, y_1, \ldots, y_N), \quad i = 1, 2, \ldots, N, \tag{1.3.49}$$

and the solution y_i satisfy the boundary condition on x_1, x_2[*9] :

$$B_{1j}(x_1, y_1, y_2, \ldots, y_N) = 0, \quad j = 1, \ldots, n_1, \tag{1.3.50}$$
$$B_{2k}(x_2, y_1, y_2, \ldots, y_N) = 0, \quad k = 1, \ldots, n_2. \tag{1.3.51}$$

This solution is satisfying n_1 boundary conditions at the starting point x_1 and $n_2 = N - n_1$ boundary conditions at the final point x_2. Therefore, in this case, we can choice freely n_2 values. Then, we set these to the components of vector $\vec{V}$. We can regard the boundary conditions eq.(1.3.50) as functions which output $y_i(x_1)$ from input vector $\vec{V}$:

$$y_i(x_1) = y_i(x_1; V_1, V_2, \ldots, V_{n_2}), \quad i = 1, \ldots, N. \tag{1.3.52}$$

If arbitrary vector $\vec{V}$ is given in some way, we can obtain the initial values $\vec{y}(x_1) = (y_1(x_1), y_2(x_1), \ldots, y_N(x_1))$. With these initial values, we can integrate from x_1 to x_2 by using any method to solve the initial value problem, e.g. Runge-Kutta method. Then, we get $\vec{y}(x_2)$ on the final point x_2. Here, we define error vector $\vec{F}$ which represents the difference from the

[*7] We restrict the explanation to one dimensional problem, i.e. the function $f(x)$ depends only on one variable x.

[*8] We can reduce the ordinary differential equation which has two or higher order to a set of first-order ordinary differential equations.

[*9] Here, we write x_1, x_2 as the coordinate of boundaries.

boundary condition eq. (1.3.51). The most simplist case is the form

$$F_k = B_{2k}(x_2, \vec{y}(x_2)), \qquad k = 1, 2, \ldots, n_2. \tag{1.3.53}$$

We employ this expression, but you can choose other form as vector $\vec{F}$. In this time, it should be equivalent that satisfying boundary condition on x_2 to become vector $\vec{F} = 0$. We are ready to begin the shooting method. Our goal is to obtain the vector $\vec{V}$ which makes vector $\vec{F} = 0$. In order to modify $\vec{F}$, we solve the linear coupled equations of n_2 variables as the form[*10]

$$\boldsymbol{A} \cdot \delta\vec{V} = -\vec{F}. \tag{1.3.57}$$

Then, we add $\delta\vec{V}$ to $\vec{V}$ as

$$\vec{V}^{new} = \vec{V}^{old} + \delta\vec{V}. \tag{1.3.58}$$

We modify the vector $\vec{V}$ repeatedly. Where matrix $\boldsymbol{A}$ has components defined by

$$A_{ij} = \frac{\partial F_i}{\partial V_j}. \tag{1.3.59}$$

However, it is almost impossible to calculate eq.(1.3.59) exactly, so we use the differencing formula

$$\frac{\partial F_i}{\partial V_j} \approx \frac{F_i(V_1, \ldots, V_j + \Delta V_j, \ldots) - F_i(V_1, \ldots, V_j, \ldots)}{\Delta V_j}. \tag{1.3.60}$$

It is important to choose ΔV_j appropriately. Moreover, the choice of initial condition is also important for shooting method.

We explain how to use the code of shooting method. Function **shoot** carries out shooting method one cycle. It requires the vector **v**[**1**...**n2**] which represents n_2 free parameters on x_1 and we calculate initial condition in **load( )** by using **v**[ ]. The vector **dv**[**1**...**n2**] contains increment of **v**[ ]. Also we need the vector **f**[**1**...**n2**] which has error at x_2 and it should be zero for the exact solution.

[*10] This idea is based on the *Newton-Raphson method*. If $\vec{V}$ is given in some way but it dosen't satisfy $\vec{F}(\vec{V}) = 0$, we have to modify the vector $\vec{F}$ and would make to be $\vec{F} = 0$. Then, we require the condition as

$$\vec{F}(\vec{V} + \delta\vec{V}) = 0. \tag{1.3.54}$$

This vector $\delta\vec{V}$ is the best modification. In order to obtain $\delta\vec{V}$, we expand eq.(1.3.54)

$$\vec{F}(\vec{V} + \delta\vec{V}) \simeq \vec{F}(\vec{V}) + \delta\vec{V} \cdot \frac{\partial\vec{F}}{\partial\vec{V}}. \tag{1.3.55}$$

We require the r.h.s. of eq.(1.3.55) is equal to 0, so we finally get the condition to $\delta\vec{V}$ as following form

$$\delta\vec{V} \cdot \frac{\partial\vec{F}}{\partial\vec{V}} = -\vec{F}(\vec{V}). \tag{1.3.56}$$

shoot

1. **load(x1,v,y)**: When free parameters **v** is given, it calculates **y**[**1**...**n**] at x_1.
2. **score(x2,y,f)**: When we obtain the vector **y**[**1**...**n**], we calculate error from boundary condition at x_2 and we substitute its rerult to **f**[**1**...**n**].
3. **del[1...n]** : This vector contains incrementals ΔV_j eq.(1.3.60).
4. **v[1...n2]** : Initial vector.
5. **derivs( )**: This function solves the ordinary differential equation.

This algorithm requires some other functions. They are listed in section3.

Listing 1.4: The shooting method algorithm

```
#include "Myheader.h"

void shoot(int nvar, double v[], double delv[],int n2, double x1, double x2,
  double eps, double h1, double hmin, double f[], double dv[], double cx[],
  double cy[], double cdydx[])
/* This function carries out shooting method from x1 to x2.
Initial condition : Function load() makes y(x1) by using input vector v[1...n2].
This function use odeint() as solver to ordinary differential equation.
On x2, this routine call the function score() and calculate f[1...n] which
should be 0 when the boundary condition are satisfied.
Solve the linear matrix equation to find dv[1...n2] by Newton-Raphson method.
User has to define the function derivs(x,y,dydx) to solve the ODE. */
{
  int nok,nbad,iv,i,*indx,kfinal,kmax;
  double sav,det,*y,**dfdv,*xp,**yp,dxsav,dum1,dum2,dum3;
  void odeint(double ystart[], int nvar, double x1, double x2,
      double eps, double h1, double hmin, int *nok, int *nbad,
      int kmax, double *xp, double **yp, double dxsav, int *kfinal,
      void (*derivs)(double,double [],double []),void (*rkqs)(double [],
          double [],int, double *,double, double,double [], double *,
          double *, void (*)(double, double [], double[])));
  void ludcmp(double **a, int n, int *indx, double *d);
  void lubksb(double **a, int n, int *indx, double b[]);
  void derivs(double x, double y[], double dydx[]);
  void rkqc(double y[], double dydx[], int n, double *x,
      double htry, double eps, double yscal[], double *hdid, double *hnext,
      void (*derivs)(double, double [], double []));
  void load(double x1, double v[], double y[]);
  void score(double xf, double y[], double f[]);

  kmax=3000;
  y=dvector(1,nvar);
  indx=ivector(1,nvar);
  dfdv=dmatrix(1,n2,1,n2);
  xp=dvector(1,kmax+2);
  yp=dmatrix(1,nvar+1,1,kmax+1);
  dxsav=0.0;

  //Solve the ODE from x1
  load(x1,v,y);
  odeint(y,nvar,x1,x2,eps,h1,hmin,&nok,&nbad,kmax, xp, yp, dxsav, &kfinal, derivs,
```

```
      rkqc);
  score(x2,y,f);
  for (iv = 1; iv <= n2; iv++) { //improve the inisial condition on x1
      sav=v[iv];
      v[iv] += delv[iv]; //increment parameters
      load(x1,v,y);
      odeint(y,nvar,x1,x2,eps,h1,hmin,&nok,&nbad, kmax, xp, yp, dxsav, &kfinal,
          derivs,rkqc);
      score(x2,y,dv);
      for (i = 1; i <= n2; i++) { //calculate partial derivative numericaly
          dfdv[i][iv]=(dv[i]-f[i])/delv[iv];
      }
      v[iv]=sav;
  }
  for (iv = 1; iv <= n2; iv++) {dv[iv] = -f[iv];}
  ludcmp(dfdv,n2,indx,&det);
  lubksb(dfdv,n2,indx,dv);
  for (iv = 1; iv <= n2; iv++) v[iv] += dv[iv];
  for (i = 1; i <= kfinal+1; i++) {
      dum1=xp[i],dum2=yp[1][i],dum3=yp[2][i];
      cx[i]=dum1,cy[i]=dum2,cdydx[i]=dum3;
  }
  free_dmatrix(dfdv,1,n2,1,n2);
  free_ivector(indx,1,nvar);
  free_dvector(y,1,nvar);
  free_dvector(xp,1,kmax+2);
  free_dmatrix(yp,1,nvar+1,1,kmax+1);
}

void derivs(double x, double y[], double dydx[])
/* Solitary wave solution to ZK equation. */
{
    double dum,c,y1,y2;
    y1=y[1],y2=y[2];
    dum=1.0/x;
    c=1.0;
    dydx[1] = y2;
    dydx[2] = -1.0*dum*y2+c*y1-y1*y1;
}

void load(double x1, double v[], double y[])
/* Function : free parameters to initial condition. */
{
      y[1]=v[1]; // <---initial value 1, we improve this value
      y[2]=0.0; // <---initial value 2
}

void score(double x2, double y[], double f[])
/* This function calculate error at x2(=boundary).*/
{
      double dum1,dum2,MSR,c,epsilon;
      epsilon=1.0e-10;
      c=1.0;
      f[1]=y[1];
}
```

1.3.5 About non-linear terms:$J[A, B]$ [1]

In the fluid dynamics, we often have to deal with a Jacobian $J[A, B]$ defined by

$$J[A, B] \equiv \frac{\partial A}{\partial x}\frac{\partial B}{\partial y} - \frac{\partial A}{\partial y}\frac{\partial B}{\partial x}. \tag{1.3.61}$$

This term contains *nonlinearity*. Therefore, it leads numerical instability. In this section, we explain that how to deal with the Jacobian $J[A, B]$. These explanation are based on Arakawa [1].

The most simplist space differencing way is the form

$$\begin{aligned} J_{i,j}[A, B] = \frac{1}{4h^2}[&(A_{i+1,j} - A_{i-1,j})(B_{i,j+1} - B_{i,j-1}) \\ &- (A_{i,j+1} - A_{i,j-1})(B_{i+1,j} - B_{i-1,j})], \end{aligned} \tag{1.3.62}$$

where i, j are the grid index number of x, y and d is the grid interval. Here, we omit discussion about its details. According to that paper, the differencing of the Jacobian term are proposed as

$$J_{i,j}[A, B] = \frac{1}{3}(J^{++}_{i,j} + J^{+\times}_{i,j} + J^{\times+}_{i,j}), \tag{1.3.63}$$

where three types Jacobian $J_{i,j}$ are given as following respectively

$$\begin{aligned} J^{++}_{i,j}[A, B] = \frac{1}{4h^2}[&(A_{i+1,j} - A_{i-1,j})(B_{i,j+1} - B_{i,j-1}) \\ &- (A_{i,j+1} - A_{i,j-1})(B_{i+1,j} - B_{i-1,j})], \end{aligned} \tag{1.3.64}$$

$$\begin{aligned} J^{+\times}_{i,j}[A, B] = \frac{1}{4h^2}[&A_{i+1,j}(B_{i+1,j+1} - B_{i+1,j-1}) - A_{i-1,j}(B_{i-1,j+1} - B_{i-1,j-1}) \\ &- A_{i,j+1}(B_{i+1,j+1} - B_{i-1,j+1}) + A_{i.j-1}(B_{i+1,j-1} - B_{i-1,j-1})], \end{aligned} \tag{1.3.65}$$

$$\begin{aligned} J^{\times+}_{i,j}[A, B] = \frac{1}{4h^2}[&A_{i+1,j+1}(B_{i,j+1} - B_{i+1,j}) - A_{i-1,j-1}(B_{i-1,j} - B_{i,j-1}) \\ &- A_{i-1,j+1}(B_{i,j+1} - B_{i-1,j}) + A_{i+1,j-1}(B_{i+1,j} - B_{i,j-1})]. \end{aligned} \tag{1.3.66}$$

We employ this differencing form eq.(1.3.63). For example, we show the code which calculate $J[\nabla^2\eta, \eta]$ numerically by differencing eq.(1.3.63). In order to calculate space-derivative $\nabla^2\eta$, we employ central differrence $\mathcal{O}(h^2)$. This calculation is used in main calculation of WYF equation.

Listing 1.5: The numerical treatment of Jacobian

```
void geoadvortex(int *xnum, int *ynum, double *chx, double *chy,
     double **eta, double **ddeta, double **jj) {
       int i, j;
       double dx, dy, dum,dumx,dumy;

       dx=2*(*chx)/(*xnum);
       dy=2*(*chy)/(*ynum);
       dumx=1.0/dx;
       dumy=1.0/dy;
       dum =1.0/12.0;

  for (i = 1; i <= *xnum + 3; ++i){
    for (j = 1; j <= *ynum - 1; ++j){
       /* dumx=1/dx dumy=1/dy */
         ddeta[i][j]=(eta[i+1][j]-2.0*eta[i][j]+eta[i-1][j])*dumx*dumx
                    +(eta[i][j+1]-2.0*eta[i][j]+eta[i][j-1])*dumy*dumy;
               }
       }
  for (i = 2; i <= *xnum + 2; ++i){
    for (j = 2; j <= *ynum - 2; ++j){
              /*J[\nabla^2 \eta, \eta]:vorticity advection term.
              J=(J++ + J+* + J*+)/3
              dum=1/12, 1/dx=dumx, 1/dy=dumy
              */
     jj[i][j] = dum*( ( (ddeta[i+1][j]-ddeta[i-1][j])*(eta[i][j+1]-eta[i][j-1])
              -(ddeta[i][j+1]-ddeta[i][j-1])*(eta[i+1][j]-eta[i-1][j]) )
              +( ddeta[i+1][j]*(eta[i+1][j+1]-eta[i+1][j-1])
              -ddeta[i-1][j]*(eta[i-1][j+1]-eta[i-1][j-1])
              -ddeta[i][j+1]*(eta[i+1][j+1]-eta[i-1][j+1])
              +ddeta[i][j-1]*(eta[i+1][j-1]-eta[i-1][j-1]) )
              +( ddeta[i+1][j+1]*(eta[i][j+1]-eta[i+1][j])
              -ddeta[i-1][j-1]*(eta[i-1][j]-eta[i][j-1])
              -ddeta[i-1][j+1]*(eta[i][j+1]-eta[i-1][j])
              +ddeta[i+1][j-1]*(eta[i+1][j]-eta[i][j-1]) ) )*dumx*dumy;
               }
       }
  return;
}
```

2 Main

In this part, a previous research and our findings are shown. We begin, section2.1 and section2.2, with explanation about the previous study and our model. Then, in section2.3 and section2.4, we show numerical methods we employing and its results. In section2.5, we discuss these results. Finally, in section2.6 and section2.7, we conclude this part and show remaining works and new perspectives.

2.1 Previous research

In 1984, G. P. Williams and T. Yamagta proposed the equation of the form

$$\underset{(1)}{\frac{\partial \eta}{\partial t}} \underset{(2)}{-\frac{E}{S}\eta\frac{\partial \eta}{\partial x}} \underset{(3)}{- S\frac{\partial}{\partial x}\nabla^2\eta} \underset{(4)}{+ 2y\eta_x} \underset{(5)}{+ EJ[\nabla^2\eta, \eta]} = 0. \tag{2.1.1}$$

This equation describes the two dimensional fluid motion in the intermediate-geostrophic regime [9]. The dependent variable $\eta(x, y, t)$ represents displacement from the mean depth of fluid H[*1] and parameters E and S are real constants which are decided by observed data about the GRS on Jupiter. We employ $E = 2$, $S = 0.5$ as value of these parameters. It follows previous research [9]. Each term included the equation is called as following respectively : (2) the nonlinear term, (3) the dispersion term, (4) the meridional twisting term and (5) the vorticity advection term or the Jacobian term. Williams and Yamagata explained the longevity of the GRS as the stability of IG vortices. That mechanism is understood as the balance between nonlinear term and dispersion term in eq.(2.1.1). Moreover, the existence of shear flow enhances that stability. Here, the shear flow means the strong east-west winds observed on the surface of Jupiter's atmosphere. They also stated that the Jacobian causes a merging vortices.

Especially, they pointed out the asymmetric behavior between anticyclonic and cyclonic vortex, the former corresponds to the GRS in Jupiter. Concreately speaking, for eq.(2.1.1), the anticyclonic vortex is long-lived but cyclonic one is not. It is because that the nonlinear effect balances to the dispersion effect for anticyclonic vortex, but it is not true for the cyclonic one. In addition to that, in order to examine that asymmetric behavior in the shear flow, they

[*1] The detail of derivation of eq.(2.1.1), please refer to section1.

introduced the term of the form $u^0(y)$ defined by

$$\xi = \eta + \int_0^y u^0(y')dy', \tag{2.1.2}$$

where ξ is new dependent variable which describes the motion of vortex in the shear flow $u^0(y)$[*2]. By substituting new variable ξ to eq.(2.1.1), we obtain the equation in terms of ξ as following form

$$\xi_t - \frac{E}{S}\xi\xi_x - \left[S + Eu^0(y)\right]\nabla^2\xi_x + \left[2y + Eu^0_{yy} + \frac{E}{S}\int_0^y u^0(y')dy'\right]\xi_x + EJ[\nabla^2\xi, \xi] = 0. \tag{2.1.4}$$

Three types of solution can occur in accordance with sign of the third term in eq.(2.1.4) [9]. It is because that if sign of the third term is same as eq.(2.1.1), the variable ξ would behave like η, i.e. in the case of $u^0(y) = 0$.

According to these previous studies, the stability of IG vortex is understood as the balance between nolinear term and dispersion term and the Jacobian term can cause the merging of vortices. We are intrested in this mathematical mechanism. In this paper, we would like to give the new perspective for this system.

2.2 Our model

Firstly, for convenience, we employ the transformation given by

$$\eta \longrightarrow \frac{\alpha S^{4/3}}{E}\eta, \quad x \longrightarrow S^{1/3}x, \quad y \longrightarrow S^{1/3}y, \tag{2.2.1}$$

where α is a real positive constant. This tranformation eliminates the parameters E and S from eq.(2.1.1). Consequently we obtain the equation of the form given by

$$\eta_t - \alpha\eta\eta_x - \nabla^2\eta_x + 2y\eta_x + \alpha J[\nabla^2\eta, \eta] = 0. \tag{2.2.2}$$

In particular, we set parameter $\alpha = 2$. Finally, the model equation is given by the form as

$$\eta_t - 2\eta\eta_x - \nabla^2\eta_x + 2y\eta + 2J[\nabla^2\eta, \eta] = 0. \tag{2.2.3}$$

According to our choice about E, S and α, $\Delta x = \Delta y = 1$ corresponds to dimensional value about 4700 km and $\Delta t = 1$ is equal to 64 days. By using this transformation eq.(2.2.1), the equaiton including shear flow $u^0(y)$ eq.(2.1.4) becomes to the form

$$\eta_t - 2\eta\eta_x - (1 + 2u^0)\nabla^2\eta_x + 2\left[y + u^0_{yy} + \int_0^y u^0(y')dy'\right]\eta_x + 2J[\nabla^2\eta, \eta] = 0. \tag{2.2.4}$$

[*2] The variable η and the vector of velocity $\vec{v} = (u, v)$ is related each other by

$$u = -\frac{\partial\eta}{\partial y}, \quad v = \frac{\partial\eta}{\partial x}. \tag{2.1.3}$$

Therefore, the term $u^0(y)$ represents the shear flow , i.e. the east-west winds.

Through the equation eq.(2.2.3) and eq.(2.2.4), we verify the balance between nonlinear term and dispersion term.

2.3 Numerical methods

In order to solve the patial differential equation as eq.(2.2.3), we employ the standard centered differencing and Arakawa scheme [1] for the Jacobian term $J[\nabla^2\eta, \eta]$. We carry out the integral by 4th Runge-Kutta scheme. To continue calculation stably and avoid boundary effect, we choose $x = \pm 20$, $y = \pm 10$ and periodic and freeslip condition for x-direction and y-direction respectively. The resolution is 256×128 and the time interval is $\Delta t = 1.0 \times 10^{-4}$. By using these setups, we solve the equation

$$\eta_t - 2\eta\eta_x - \nabla^2\eta_x + 2y\eta + 2J[\nabla^2\eta, \eta] = 0. \tag{2.3.1}$$

This is the same as eq.(2.2.4). In order to examine the effect of shear flow, we especially focus on the initial condition for eq.(2.3.1). According to the previous research [9], they carried out the numerical studies with initial condition such as

$$\eta(x, y, 0) = e^{-x^2-y^2} - \frac{1}{2}y^2. \tag{2.3.2}$$

The first term represents anticyclonic vortex. If its sign is negative, it is ragarded as cyclonic one. The second term means the anticyclonic shear flow and its inverse sign is the cyclonic one. We consider to these points from the next section.

2.4 Numerical studies

In the section 2.1, we cited that the stability of IG vortex is understood by the balance between nonlinear term and dispersion term in eq.(2.1.1). Further, if we change the variable to ξ, its balance would be kept when the sign of $[S + Eu^0(y)]$ is positive in eq.(2.1.4). We use this idea for eq.(2.2.3) and eq.(2.2.4) to verify the stability fo vortex. For two equations, eq.(2.2.3) and eq.(2.2.4), the balance condition becomes to the form

$$1 + 2u^0(y) > 0. \tag{2.4.1}$$

This condition leads an inequality for y defining the blance area of two effects : nonlinear term and dispersion term. Here, in order to match the physical requirement on Jobian atmosphere, we restrict the form of $u^0(y)$ to polynominal of y up to the first order, i.e. we suppose the form of $u^0(y)$ as following

$$u^0(y) = a + by, \tag{2.4.2}$$

where a, b are real constants. For example, we show the case of (1) $u^0(y) = -y$ and (2) $u^0(y) = -1 - y$. From these shear flow, we obtain (1) $y < 1/2$ and (2) $y < -1/2$ as the balance area shown in figure2.1 as following

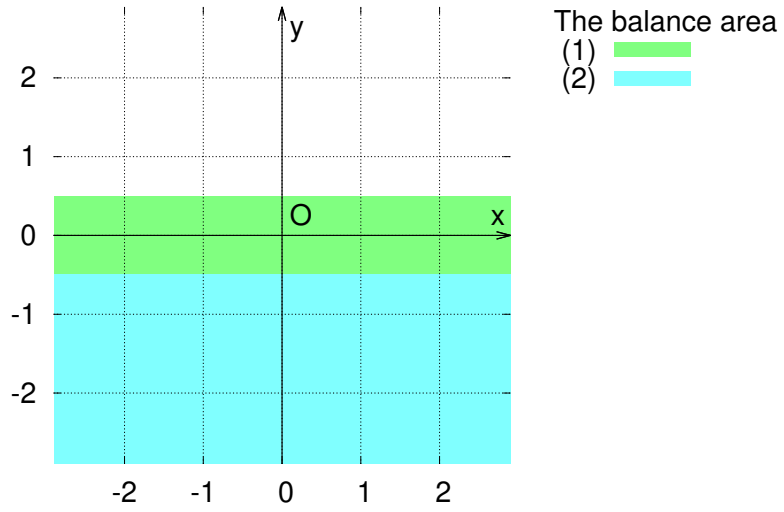


Figure2.1: This figure shows the balance area in the case of (1) $u^0(y) = -y$ and (2) $u^0(y) = -1-y$ which correspond to green and blue area respectively.

In green and blue area, the coefficient of the third term in eq.(2.2.4) is positive when we choose $u^0(y) = -y$ as shear flow. Therefore, nonlinear term and dispersion term are balance in there. In blue area, its balance should be held when we choose $u^0(y) = -1-y$. In other words, we expect the stability of IG vortex in these areas. It is because the balance between nonlinear term and dispersion term are kept in there when we choose the specific form as the shear flow respectively. We carry out the calculation for the two cases above. Further, that result indicates there is a new mechanism for stability of the IG vortex. Concretely speaking, the Jacobian and the shear flow play the important roles even if there is an only single vortex with shear flow. According to Williams et al. [9], the Jacobian plays its role during vortex collisions. However, our results show that the Jacobian prevents a single vortex deforming and the shear flow supports its work.

Firstly, in section 2.4.1, we show the case of a single anticyclonic vortex in the anticyclonic shear flow $u^0(y) = -y$. We expect that this vortex sustains its form without dispersion. Secondly, in section 2.4.2, we verify the case of $u^0(y) = -1-y$. In this case, the balance area is shown by blue area in figure2.1. If the vortex locates on the out of its area, how dose the vortex behave? We show the results for these two cases in the next two subsections.

2.4.1 The shear flow $u^0(y) = -y$

In order to examine the behavior of anticyclonic vortex in anticyclonic shear flow, we solve eq.(2.3.1) with initial condition

$$\eta(x, y, 0) = (vortex) - \frac{1}{2}y^2 \equiv \eta_0 - \frac{1}{2}y^2, \tag{2.4.3}$$

where η_0 represents the single vortex which are obtained from ZK equation as solitary wave solution [2, 3], section1.1.2. Firstly, we show the numerical results in figure2.2. We can find that the vortex keeps its form with a little deformation along east-west direction. The amplitude of η is varying in the range about +6%, but it doesn't become smaller than initial value in figure2.3a.

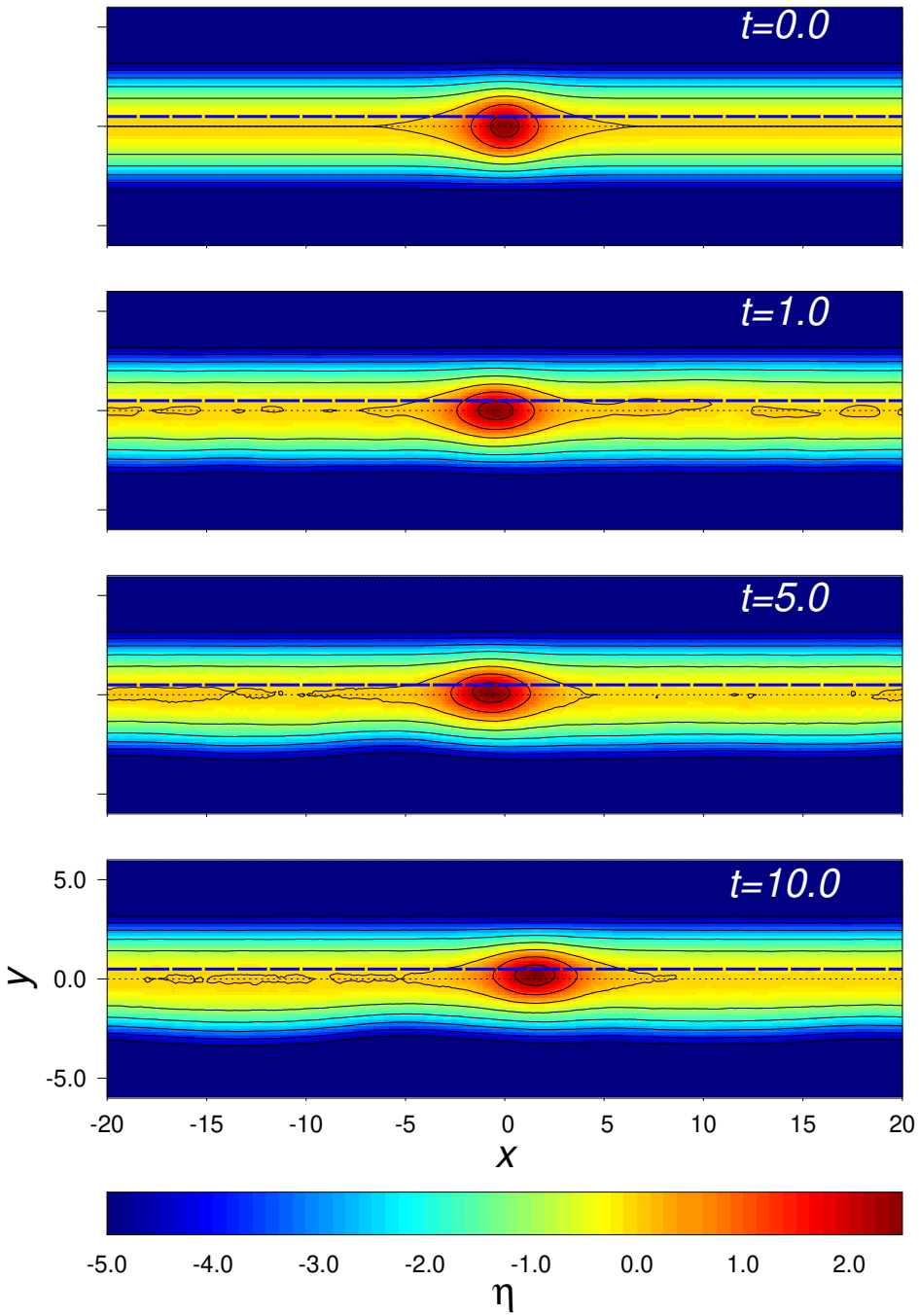
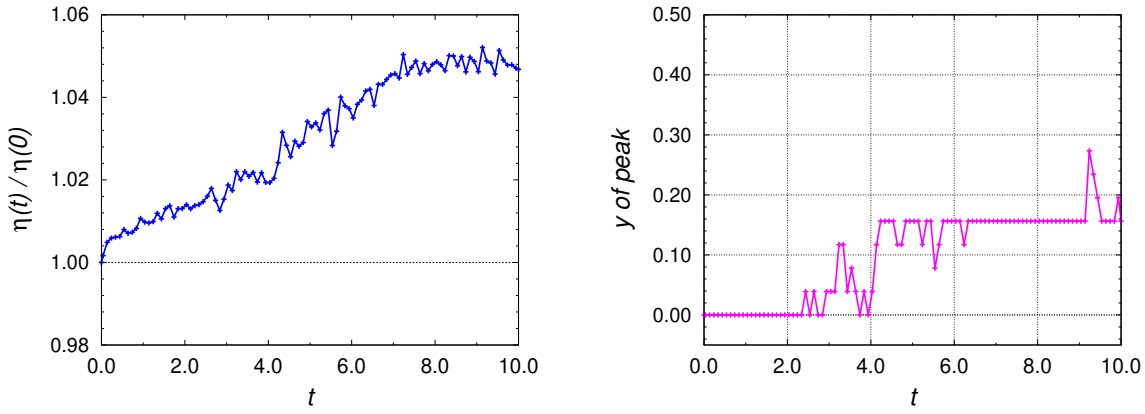


Figure2.2: Dashed blue lines indicate $y = 1/2$ lines. Contour lines are drawn at $-5, -4, -3, -2, -1, 0, 1, 2, 2.5$. $\Delta t = 1$ is corresponds to real time interval about 64 days. The IG vortex keeps its amplitude and location on y direction.



(a) The ratio of peak amplitude $\eta(x, y, t)$ to the initial amplitude $\eta(x, y, 0)$ depending on time.

(b) The y coordinate of peak $\eta(x, y, t)$ depending on time.

Figure2.3: In the case of $u^0(y) = -y$.

Moreover, the peak of vortex keeps locating on the balance area which indicated by dashed blue lines in figure2.2 and figure2.3*b*. We can understand that stability comes from the balance between nonlinear term and dispersion term.

2.4.2 The shear flow $u^0(y) = -1 - y$

Secondly, we show the results in the case of shear flow $u^0(y) = -1 - y$. We solve the eq.(2.3.1) with the initial condition given by

$$\eta(x, y, 0) = \eta_0 - y - \frac{1}{2}y^2, \tag{2.4.4}$$

where η_0 is the same as before. This shear flow reduces the balance area from $y < 1/2$ to $y < -1/2$. Therefore, we expect the different behavior from the result with $u^0(y) = -y$. The figure2.4 shows the time evolution of vortex in the shear flow eq.(2.4.4). We can find that the vortex sustains its form with a little deformation and it moves to westward direction in figure2.4. Finally, amplitude of vortex becomes smaller than the initial value by 4% in figure2.5*a*. Although the amplitude becomes smaller, the vortex keeps its form in time evolution. For the peak of the vortex, at first, it locates in the out of the balance area, i.e. $y > -1/2$ then the vortex moves to southward direction and enters the balance area in figure2.5*b*. We can also find this behavior in figure2.4.

However, the north side of vortex remains in the no-balance area and that part keeps its form as the vortex. In other words, if there is no balance between nonlinear term and dispersion term, the vortex still sustains its form. Therefore, we consider that there is another mechanism sustaining the form of the vortex. We focus on this point and examine it from next section. This is the new findings and our purpose in this chapter.

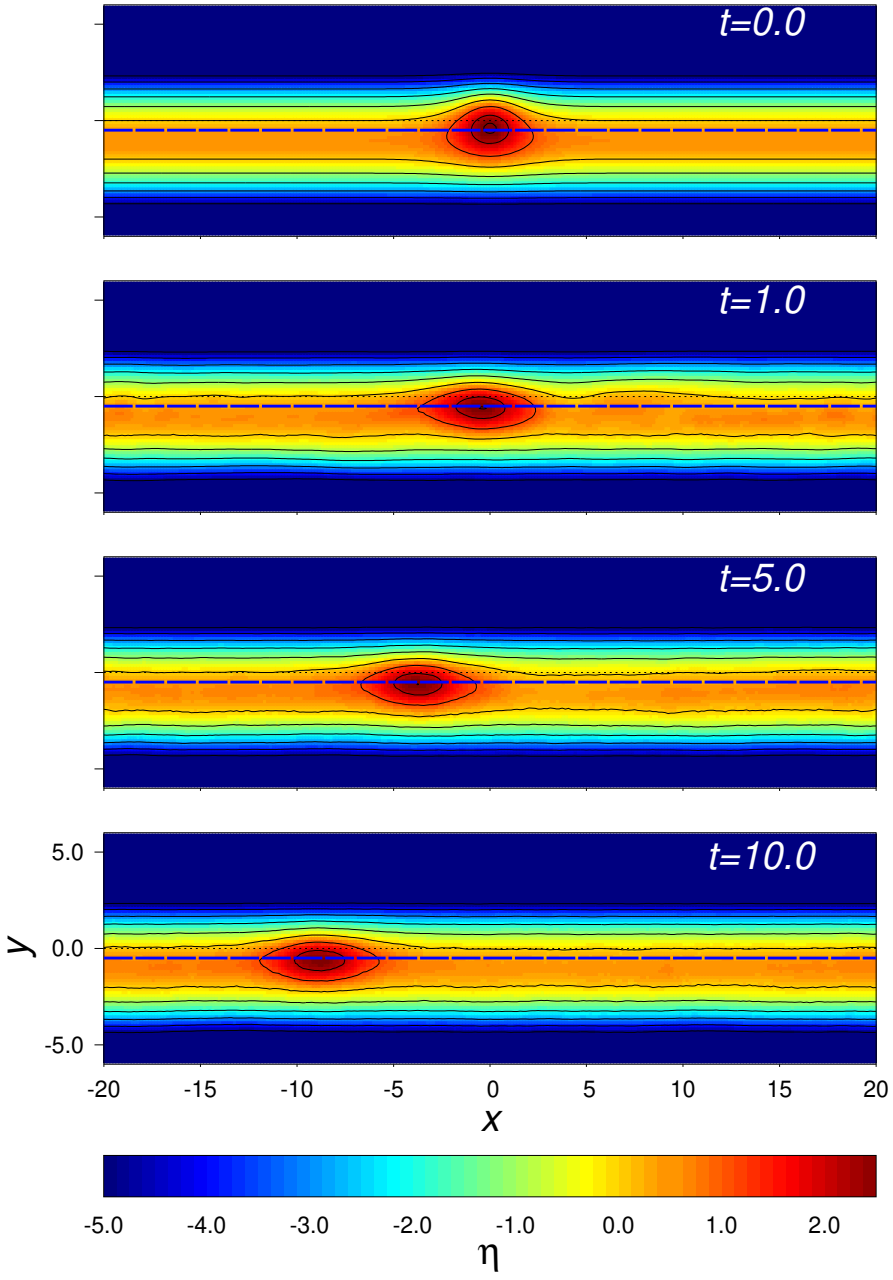
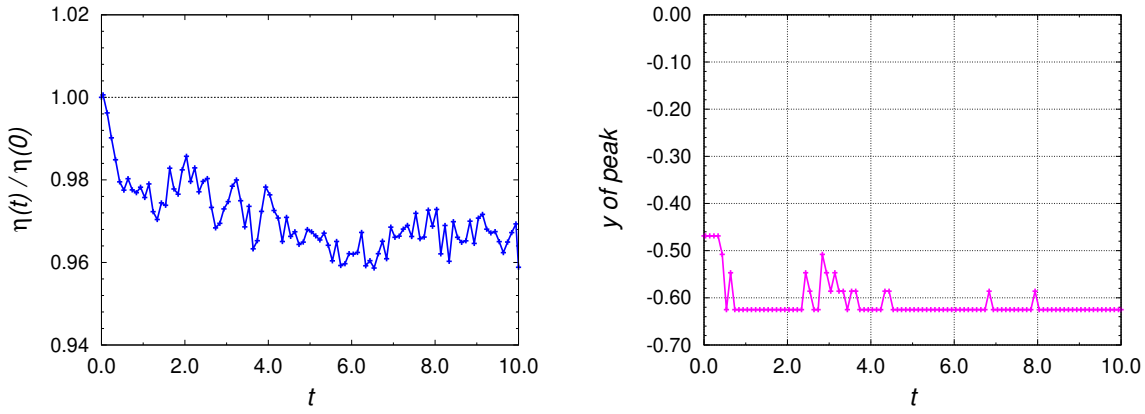


Figure2.4: Dashed blue lines indicate $y = -1/2$ lines. Contour lines are the same as figure2.1. The IG vortex sustains its form but it moves westward direction.



(a) The ratio of peak amplitude to the initial amplitude depending on time.

(b) The y coordinate of peak $\eta(x, y, t)$ depending on time.

Figure2.5: In the case of $u^0(y) = -1 - y$.

2.4.3 Examination

In the last section, we pointed out that the vortex can sustain its form in the out of the balance area. In order to examine what mechanism causes this stability, we introduce the tranformation given by

$$\xi \equiv \eta - y. \tag{2.4.5}$$

This tranformation corresponds to employing the shear flow of the form

$$u^0(y) = -1. \tag{2.4.6}$$

By substituting eq.(2.4.6) to eq.(2.3.1), we obtain the equation for ξ given by

$$\xi_t - 2\xi\xi_x + \nabla^2\xi_x + 2J[\nabla^2\xi, \xi] = 0. \tag{2.4.7}$$

We can clearly find that nonlinear term and dispersion term are not balance for the anticyclonic vortex in eq.(2.4.7). However, in order to describe the vortex like the GRS, we require anticyclonic one. Therefore, we consider the vortex which has positive amplitude. From our results, we find that the Jacobian and the shear flow contribute to the stability of anticyclonic vortex. We show these results by numerical studies in four cases.

Firstly, we consider the equation with no Jacobian :

$$\xi_t - 2\xi\xi_x + \nabla^2\xi_x = 0. \tag{2.4.8}$$

In particular, we show the two results solving eq.(2.4.8) with initial condition, (1) $\xi(x, y, 0) = \eta_0$, (2) $\xi(x, y, 0) = \eta_0 - y^2/2$. At figure2.6, the anticyclonic vortex develops following eq.(2.4.8) with no shear flow, i.e. (1). Since nonlinear term and dispersion term are not balance each other for anticyclonic vortex, these effects tend to disperse the vortex. At $t = 1.0$, the vortex already

deformed along westward direction. After that time, the vortex spreaded over the area and did not sustain itself. At figure2.7, the anticyclonic vortex develops following eq.(2.4.8) with shear flow, i.e. (2). Also in this case, nonlinear effect and dispersion effect tend to disperse the vortex. The shear flow doesn't sustain the form of vortex.

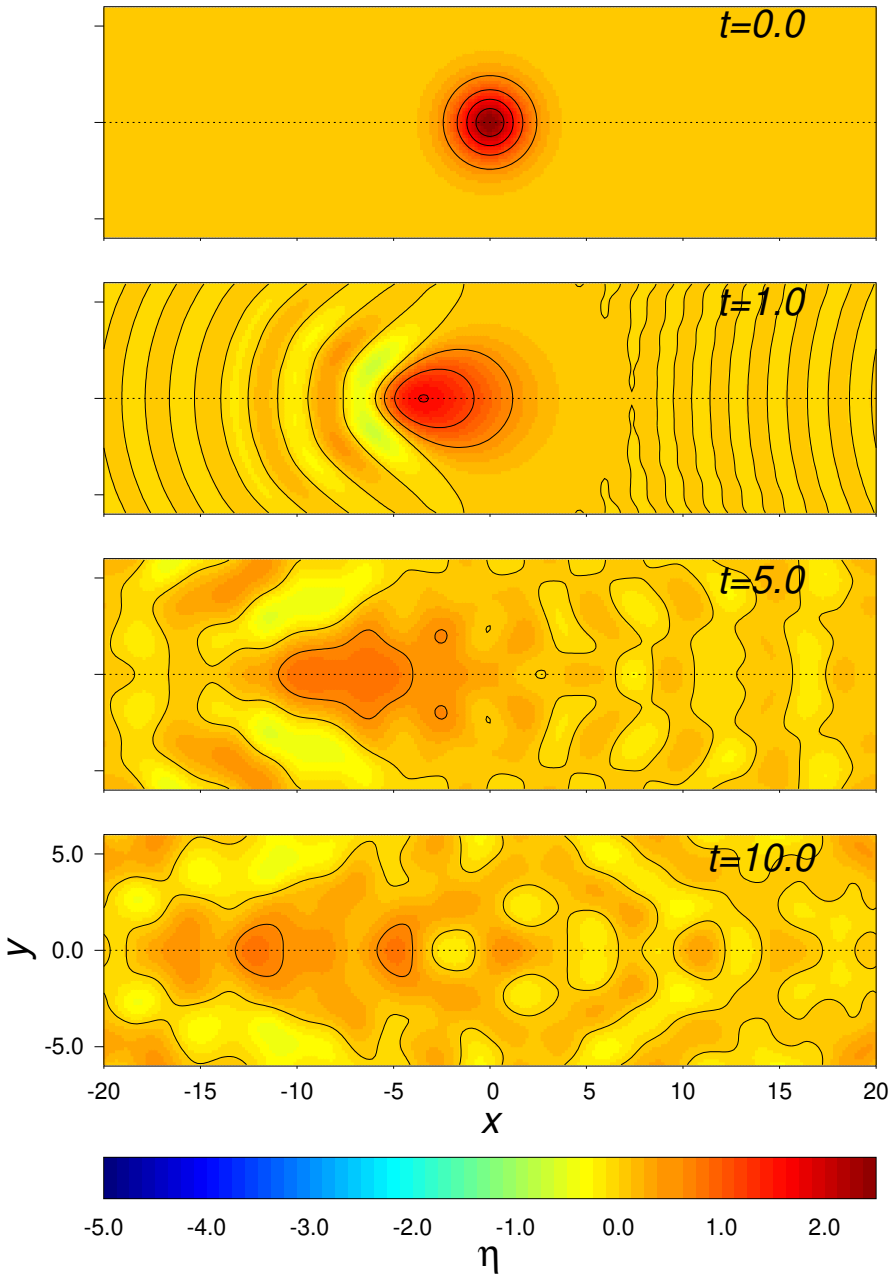


Figure2.6: Result of the case (1). Contour lines are drawn at $-3, -2, -1, 0, 0.5, 1.0, 1.5, 2.0, 2.5$. Nonlinear effect and dispersion effect disperse the vortex. Only the part $y = \pm 6$ of a computational domain of $y = \pm 10$ is shown.

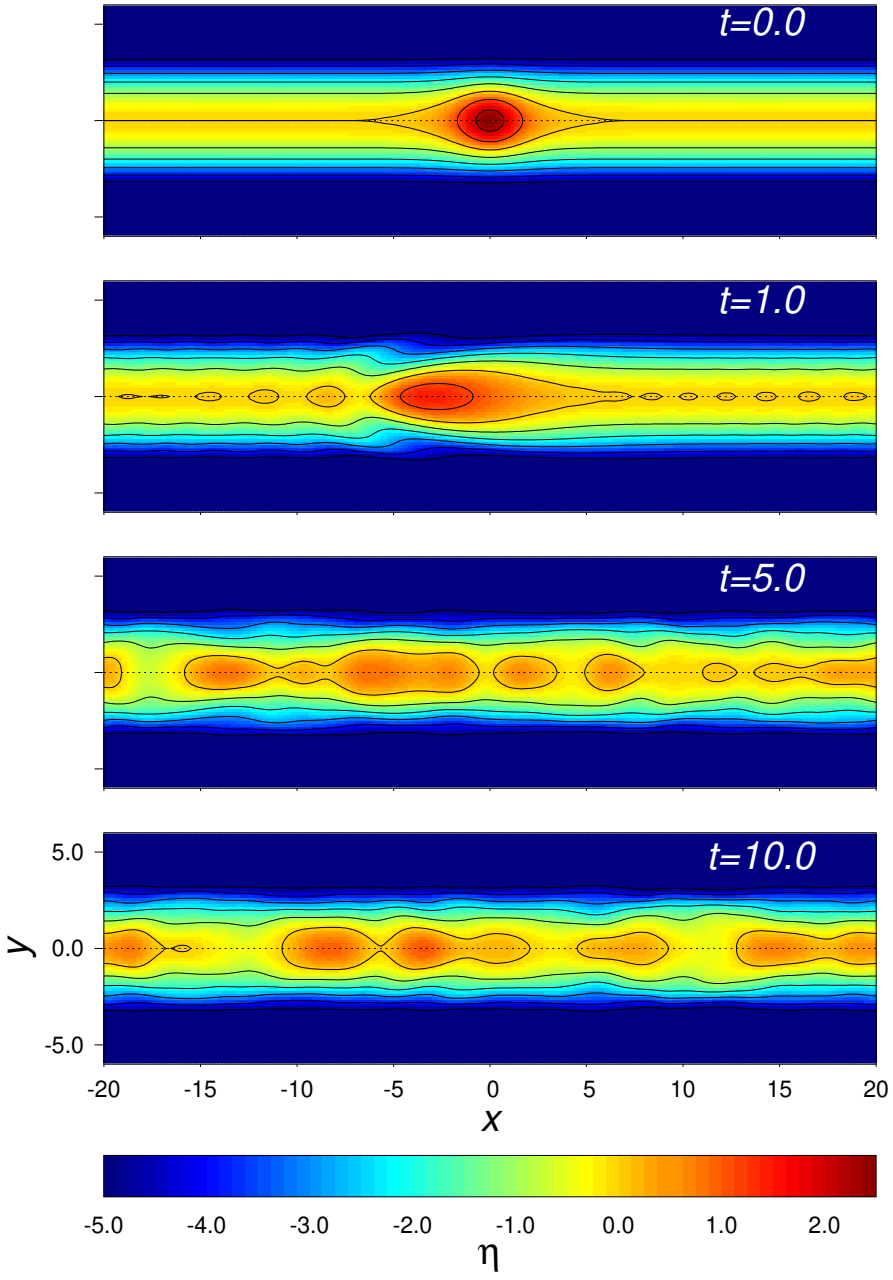


Figure2.7: Result of the case (2). Contour lines are drawn at -5,-4,-3,-2,-1,0,1,2,2.5. Also in this case, vortex does not sustain itself.

Secondly, we consider the equation with the Jacobian

$$\xi_t - 2\xi\xi_x + \nabla^2\xi_x + 2J[\nabla^2\xi, \xi] = 0. \tag{2.4.9}$$

Similarly, we show the two results solving eq.(2.4.9) with initial condition, (3) $\xi(x, y, 0) = \eta_0$, (4) $\xi(x, y, 0) = \eta_0 - y^2/2$. We show the case (3) at figure2.8. The nonlinear effect and dispersion effect disperse the vortex. Further, the Jacobian works on the deformed vortex. That makes some computational noises and leads instability of simulation. From our results, computational simulation blows up at $t = 2.0$ (figure2.8).

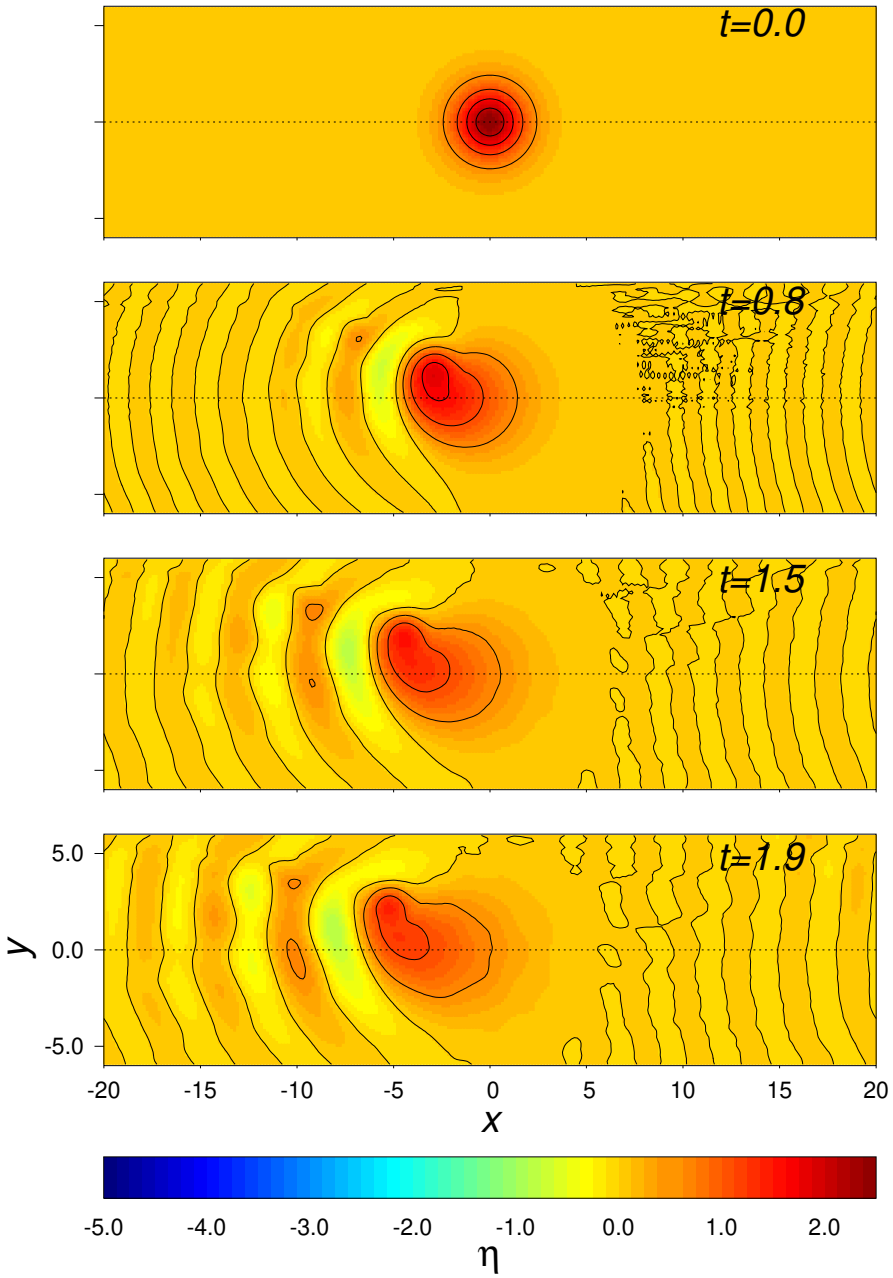


Figure2.8: Result of the case (3). Since calculation blows up after $t = 1.9$, we show the results untill that time. Contour lines are drawn at $-3, -2, -1, 0, 0.5, 1.0, 1.5, 2.0, 2.5$.

Next, we show the case (4) in figure2.9 and figure2.10 which are color-map of $\xi(x, y, t)$ and profiles respectively. Only this case, the anticyclonic vortex sustains its form during time evolution. However, the amplitude of vortex falls to nearly 84%. This reduction is bigger than the case of figure2.5a. The y coordinate of peak moves to northward direction.

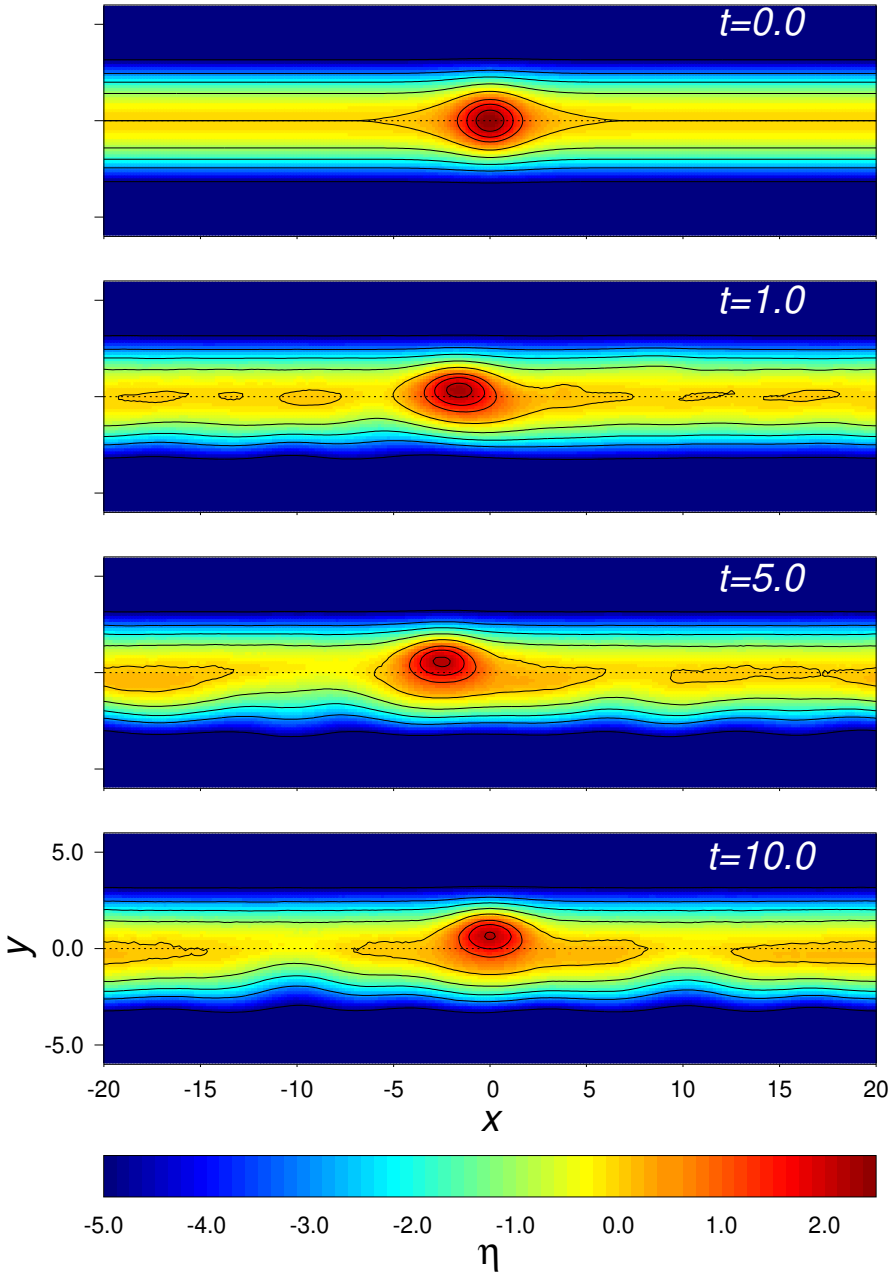
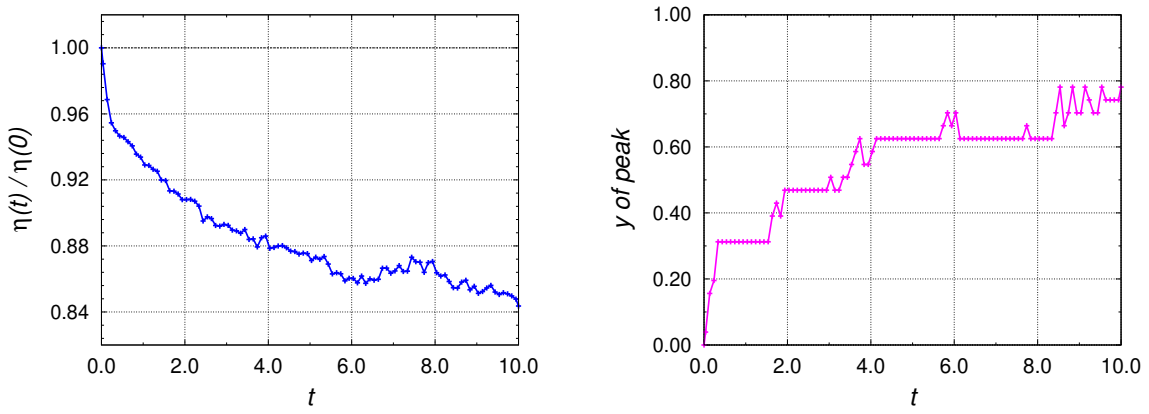


Figure2.9: Result of the case (4). Contour lines are drawn at -5,-4,-3,-2,-1,0,1,2,2.5. Only in this case, the vortex can sustain itself.



(a) The ratio of peak amplitude to the initial amplitude. Amplitude falls to nearly 84%.

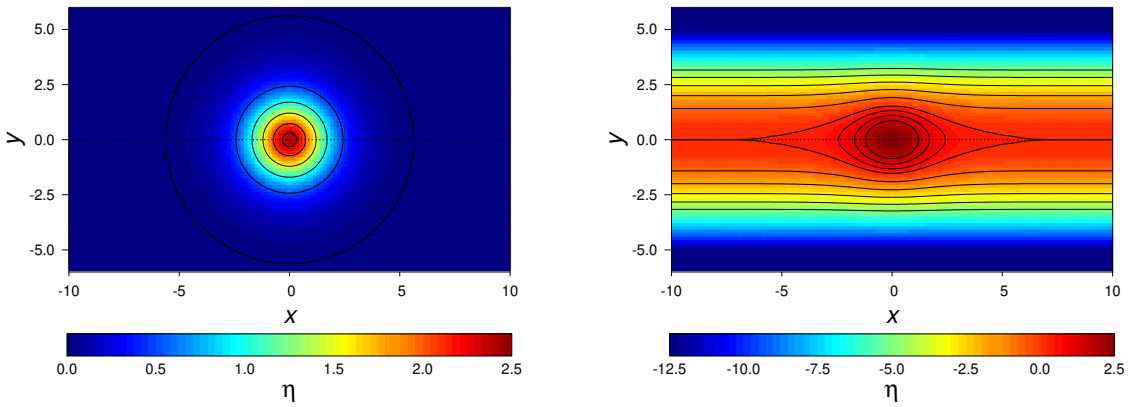
(b) The y coordinate of peak $\eta(x, y, t)$ depending on time.

Figure2.10: In the case of eq.(2.4.9) with shear flow.

2.5 Discussion

From these results in the last section, we find that if there is no balance between nolinear term and dispersion term, the Jacobian and shear flow sustain the form of vortex during time evolution. That role of the Jacobian and shear flow are not mentioned by Williams and Yamagata [9]. Therefore, these results give us many comprehensions and perspectives. Before discussion such findings, we give a qualitative explanation about the mechanism of the Jacobian and shear flow.

Firstly, the shear flow deforms the vortex like figure2.11. It is natural to cause these effects because the shear flow has the form of $-y^2/2$ and this gives a convex upward function.



(a) Anticyclonic vortex with no shear flow. Contour lines are drawn at $0, 0.01, 0.5, 1, 1.5, 2, 2.3, 2.5$.

(b) Anticyclonic vortex with shear flow. Contour lines are drawn at from -5 to 2.5.

Figure2.11: The color map and contour lines for anticyclonic vortex.

Secondly, the Jacobian only works for asymmetric vortex or configuration. In other words, when $\nabla^2\xi$ and ξ have only dependence of r[*3], the Jacobian $J[\nabla^2\xi, \xi]$ becomes to zero. In order to recognize that fact, we show a little algebra and numerical results. We calculate the Jacobian term of the form $J[A, B]$ with $A = A(r)$ and $B = B(r)$. By introducing polar coordinates in two dimensional space (r, θ), we can rewrite derivatives as follows

$$\partial_x = \cos\theta\partial_r - \frac{\sin\theta}{r}\partial_\theta, \tag{2.5.1}$$

$$\partial_y = \sin\theta\partial_r + \frac{\cos\theta}{r}\partial_\theta, \tag{2.5.2}$$

where ∂_r denotes $\partial/\partial r$. Consequently, we obtain the result given by

$$J[A, B] = A_x B_y - A_y B_x = \cos\theta A_r \cdot \sin\theta B_r - \sin\theta A_r \cdot \cos\theta B_r = 0. \tag{2.5.3}$$

Eq.(2.5.3) indicates that the Jacobian doesn't work for symmetric functions. Next, we show the effect of Jacobian between two vortices. This is an example of asymmetric configuration. In figure2.12, we show a color map. This is the solution to an equation of the form

$$\xi_t + 2J[\nabla^2\xi, \xi] = 0, \tag{2.5.4}$$

with initial condition given by

$$\xi(x, y, 0) = \eta_0|_{x=-3} + \eta_0|_{x=+3}\,. \tag{2.5.5}$$

We can find that two vortices attract each other. It is well known the fact that Jacobian has a such effect for ansymmetric vortex and configuration.

For the existence of shear flow and the Jacobian, we can understand the stability of anticyclonic vortex in the case of (4) in the last section. Shear flow deforms the vortex like figure2.11*b*. Besides, the Jacobian has effect when the vortex is deformed and the configuration is asymmetric. Therefore, if it is impossible to expect the balance between nolinear term and dispersion term, the Jacobian and shear flow prevent the anticyclonic vortex from dispersing.

Williams and Yamagata mentioned as the balance between nonlinear term and dispersion term causes stability of anticyclonic vortex. However, our results indicate that not only such a balance but also the effect of the Jacobian and shear flow sustains the form of vortex in the original equation eq.(2.3.1).

[*3] where r means a distance from the center of vortex. If vortex moves away from initial position, this result is also valid.

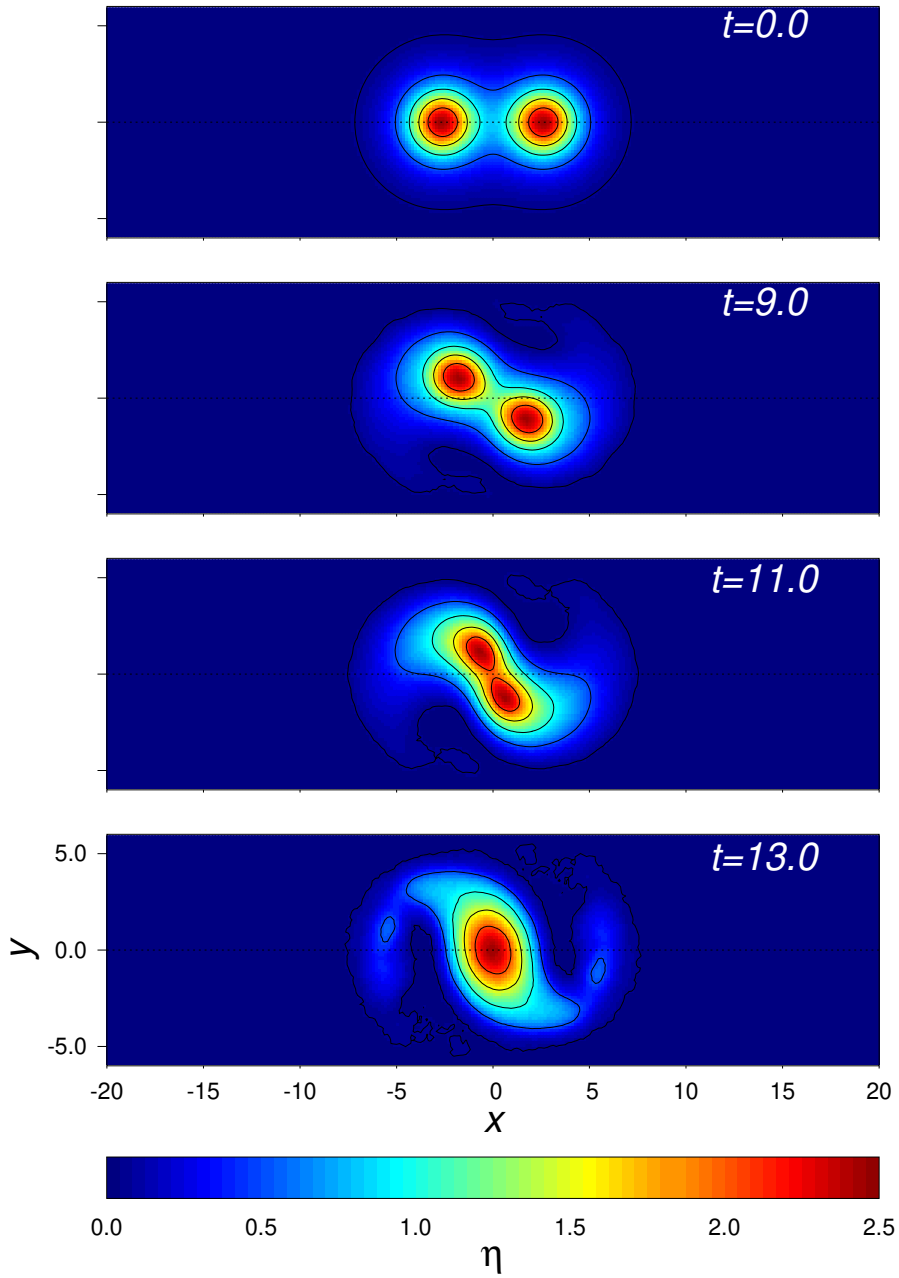


Figure2.12: Solution to eq.(2.5.4) with two vortices. Contour lines are drawn at $0.05, 0.5, 1, 1.5, 2, 2.5$. Since there is the Jacobian term, two vortices are attracted each other. Further, that effect causes the merging of two vortices.

2.6 Conclusion

We examine the stability of vortex caused by the Jacobian and shear flow in eq.(2.6.1)[*4]

$$\xi_t - 2\xi\xi_x + \nabla^2\xi + 2J[\nabla^2\xi, \xi] = 0. \tag{2.6.1}$$

Without shear flow, it is impossible to sustain the form of anticyclonic vortex in eq.(2.6.1). The reason why we look for anticyclonic vortex is that it corresponds to the GRS on Jupiter. Therefore, we seek such solution. In order to obtain the stable anticyclonic vortex, we employ the initial condition of the form

$$\xi(x, y, 0) = \eta_0 - \frac{1}{2}y^2 \tag{2.6.2}$$

in eq.(2.6.1). This form of solution is the just stable anticyclonic vortex we desired (section2.4). Moreover, our results reveal the fact that the Jacobian and shear flow work behind the balance between nonlinear effect and dispersion effect (section2.5).

2.7 Future works

In section2.4.3, we introduced the tranformation of the form

$$\xi = \eta - y. \tag{2.7.1}$$

Our purpose of introducing this transformation is removing the balance between nonlinear term and dispersion term and to observe the role of the Jacobian and shear flow. However, we can get the other perspective from this transformation. We substitute transformation eq.(2.7.1) to the original equation[*5]

$$\eta_t - 2\eta\eta_x - \nabla^2\eta_x + 2y\eta_x + 2J[\nabla^2\eta, \eta] = 0, \tag{2.7.2}$$

and obtain the equation eq.(2.6.1). In this step, we can consider the more generalized form of transformation as follows :

$$\zeta \equiv \eta + a + by + cy^2, \tag{2.7.3}$$

where a, b and c are real parameters. Substituing eq.(2.7.3) to eq.(2.7.2) , we obtain the equation for ζ :

$$\zeta_t - 2\zeta\zeta_x - (1 + 2b + 4cy)\nabla^2\zeta_x + 2\left[a + (1 + b)y + cy^2\right]\zeta_x + 2J[\nabla^2\zeta, \zeta] = 0 \tag{2.7.4}$$

Eq.(2.7.4) contains eq.(2.3.1) , eq.(2.4.7) and so on as a special case.

[*4] This is the same as eq.(2.4.7).
[*5] This is the same as eq.(2.3.1).

We show some exmaples as follows :

1. (1) $a = b = c = 0$
$$\zeta_t - 2\zeta\zeta_x - \nabla^2\zeta_x + 2y\zeta_x + 2J[\nabla^2\zeta, \zeta] = 0 \tag{2.7.5}$$

2. (2) $a = c = 0, b = -1$
$$\zeta_t - 2\zeta\zeta_x + \nabla^2\zeta_x + 2J[\nabla^2\zeta, \zeta] = 0 \tag{2.7.6}$$

Moreover, from this viewpoint, we can deal with the shear flow within a choice of parameters. In order to reveal any findings, these ideas have to be studied more quantitatively. For our lack of ability, we couldn't study more at present. However, we hope that the elegant fact will appear in front of our sight.

3 Appendix

3.1 KdV dynamics

3.1.1 Bäcklund transform

We show the Bäcklund transform for KdV equation of the form

$$u_t - 6uu_x + u_{xxx} = 0. \tag{3.1.1}$$

The Bäcklund tranform is given as follows

$$u(x,t) = w_x(x,t), \qquad u'(x,t) = w'_x(x,t), \tag{3.1.2}$$

$$w_t + w'_t = 2({w_x}^2 + w_x w'_x + {w'_x}^2) - (w - w')(w_{xx} - w'_{xx}), \tag{3.1.3}$$

$$w_x + w'_x = -2\eta^2 + \frac{1}{2}(w - w')^2, \tag{3.1.4}$$

where η is constant. If w_x is solution of KdV equation eq.(3.1.1), w'_x is also solution of it. We will show that.

At first, since $w_x = u$, the equation is followed of the form

$$w_{xt} - 6w_x w_{xx} + w_{xxxx} = 0. \tag{3.1.5}$$

It is because u satisfy the equation eq.(3.1.1). Then, we calculate derivatives eq.(3.1.3) and (3.1.4) by variable x

$$\begin{aligned}
(3.1.3) \to w_{xt} + w'_{xt} &= 2(2w_x w_{xx} + w_{xx}w'_x + w_x w'_{xx} + 2w'_x w'_{xx}) \\
&\quad -(w_x - w'_x)(w_{xx} - w'_{xx}) - (w - w')(w_{xxx} - w'_{xxx}), && (3.1.6)\\
(3.1.4) \to w_{xx} + w'_{xx} &= (w - w')(w_x - w'_x), && (3.1.7)\\
w_{xxx} + w'_{xxx} &= (w_x - w'_x)^2 + (w - w')(w_{xx} - w'_{xx}), && (3.1.8)\\
w_{xxxx} + w'_{xxxx} &= 3(w_x - w'_x)(w_{xx} - w'_{xx}) + (w - w')(w_{xxx} - w'_{xxx}). && (3.1.9)
\end{aligned}$$

Moreover, by using eq.(3.1.6)-(3.1.7), we make the l.h.s. of equation eq.(3.1.1) as follows

$$\begin{aligned}
&(\text{1st term}) \to w_{xt} + w'_{xt} = (3.1.6) && (3.1.10)\\
&(\text{2nd term}) \to -6(w_x + w'_x)(w_{xx} + w'_{xx}) = -6(w_x w_{xx} + w'_x w'_{xx}) - 6(w_x w'_{xx} + w'_x w_{xx}) && (3.1.11)\\
&(\text{3rd term}) \to w_{xxxx} + w'_{xxxx} = (3.1.9) && (3.1.12)
\end{aligned}$$

By using these equations, we obtain the expression of the l.h.s. of eq.(3.1.1) as follows

$$
\begin{aligned}
(l.h.s.) &= \underbrace{w_{xt} + w'_{xt}}_{(a)} - \underbrace{6(w_x + w'_x)(w_{xx} + w'_{xx})}_{(b)} + \underbrace{(w_{xxxx} + w'_{xxxx})}_{(c)} \\
&= \underbrace{2(2w_x w_{xx} + w_{xx}w'_x + w_x w'_{xx} + 2w'_x w'_{xx}) - (w_x - w'_x)(w_{xx} - w'_{xx}) - (w - w')(w_{xxx} - w'_{xxx})}_{(a)} \\
&\quad \underbrace{-6(w_x w_{xx} + w'_x w'_{xx}) - 6(w_x w'_{xx} + w'_x w_{xx})}_{(b)} + \underbrace{3(w_x - w'_x)(w_{xx} - w'_{xx}) + (w - w')(w_{xxx} - w'_{xxx})}_{(c)} \\
&= 4(w_x w_{xx} + w'_x w'_{xx}) + 2(w_x w'_{xx} + w'_x w_{xx}) - (w_x w_{xx} + w'_x w'_{xx}) + (w_x w'_{xx} + w'_x w_{xx}) \\
&\quad -6(w_x w_{xx} + w'_x w'_{xx}) - 6(w_x w'_{xx} + w'_x w_{xx}) + 3(w_x w_{xx} + w'_x w'_{xx}) - 3(w_x w'_{xx} + w'_x w_{xx}) \\
&= -6(w_x w'_{xx} + w'_x w_{xx}).
\end{aligned} \tag{3.1.13}
$$

Therefore, we obtain the identity of the form

$$
\begin{aligned}
& w_{xt} + w'_{xt} - 6(w_x + w'_x)(w_{xx} + w'_{xx}) + (w_{xxxx} + w'_{xxxx}) = -6(w_x w'_{xx} + w'_x w_{xx}) \\
\Leftrightarrow\ & w_{xt} + w'_{xt} - 6(w_x w_{xx} + w'_x w'_{xx}) + (w_{xxxx} + w'_{xxxx}) = 0
\end{aligned} \tag{3.1.14}
$$

When the equation eq.(3.1.48) is satisfied, w' is also the solution to eq.(3.1.48), i.e. w' satisfies the equation as follows

$$
w'_{xt} - 6w'_x w'_{xx} + w'_{xxxx} = 0. \tag{3.1.15}
$$

Therefore, if w_x is solution of KdV equation eq.(3.1.1), w'_x is also solution of it. This is the fact mentioned in section1.1.3.

3.1.2 Conserved quantities of $u_t - 2uu_x + \nabla^2 u_x + 2J[\nabla^2 u, u] = 0$

In this section, we give a proof that there are some conserved quantities for the equation as follows

$$u_t - 2uu_x + \nabla^2 u_x + 2J[\nabla^2 u, u] = 0. \tag{3.1.16}$$

Since we have already explained this equation in section2, we don't mention its details here. This equation has the conserved quantities given by following forms

$$I_n \equiv \int D_n \, dxdy, \tag{3.1.17}$$

$$D_1 = u, \tag{3.1.18}$$

$$D_2 = \frac{1}{2}u^2, \tag{3.1.19}$$

$$D_3 = \frac{1}{3}u^3 + \frac{1}{2}(u_x^2 + u_y^2), \tag{3.1.20}$$

where we employ interval of integration like $x \times y : [-M, M] \times [-L, L]$. In order to conserve these quantities, we need some conditions. In other words, these quantities are *not always* conserverd. We will explain these points later.

Before considering I_n, for convenience, we show an identity for $J[\nabla^2 u, u]$. We make a product of u^p and $J[\nabla^2 u, u]$ as following

$$\begin{aligned}
u^p J[\nabla^2 u, u] &= u^p \frac{\partial \nabla^2 u}{\partial x}\frac{\partial u}{\partial y} - u^p \frac{\partial \nabla^2 u}{\partial y}\frac{\partial u}{\partial x} \\
&= \frac{\partial \nabla^2 u}{\partial x} u^p \frac{\partial u}{\partial y} - \frac{\partial \nabla^2 u}{\partial y} u^p \frac{\partial u}{\partial x} \\
&= \frac{\partial \nabla^2 u}{\partial x}\frac{\partial}{\partial y}\left(\frac{u^{p+1}}{p+1}\right) - \frac{\partial \nabla^2 u}{\partial y}\frac{\partial}{\partial x}\left(\frac{u^{p+1}}{p+1}\right) \\
&= \frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}\frac{u^{p+1}}{p+1}\right) - \frac{\partial^2 \nabla^2 u}{\partial x \partial y}\frac{u^{p+1}}{p+1} - \frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}\frac{u^{p+1}}{p+1}\right) + \frac{\partial^2 \nabla^2 u}{\partial x \partial y}\frac{u^{p+1}}{p+1} \\
&= \frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}\frac{u^{p+1}}{p+1}\right) - \frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}\frac{u^{p+1}}{p+1}\right)
\end{aligned} \tag{3.1.21}$$

Therefore, we obtain the identity

$$u^p J[\nabla^2 u, u] = \frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}\frac{u^{p+1}}{p+1}\right) - \frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}\frac{u^{p+1}}{p+1}\right). \tag{3.1.22}$$

We use this identity in proof below.

At first, we consider $D_1 = u$ and I_1 of the form given by

$$I_1 = \int u \, dxdy. \tag{3.1.23}$$

Then, we calculate derivative by time t

$$\frac{dI_1}{dt} = \frac{d}{dt}\int u \, dxdy = \int \frac{d}{dt} u \, dxdy = \int u_t \, dxdy. \tag{3.1.24}$$

In order to obtain u_t, we use eq.(3.1.16) as following

$$\begin{aligned}(3.1.16) &\Rightarrow u_t - 2uu_x + \nabla^2 u_x + 2J[\nabla^2 u, u] = 0 \\ &\Rightarrow u_t = 2uu_x - \nabla^2 u_x - 2J[\nabla^2 u, u] \\ &= \frac{\partial}{\partial x}\left(u^2 - \nabla^2 u\right) - 2\frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}u\right) + 2\frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}u\right) \\ &= \frac{\partial}{\partial x}\left(u^2 - \nabla^2 u + 2u\frac{\partial \nabla^2 u}{\partial y}\right) - \frac{\partial}{\partial y}\left(2u\frac{\partial \nabla^2 u}{\partial x}\right). \end{aligned} \tag{3.1.25}$$

Therefore, we get the derivative of I_1 as follows

$$\begin{aligned}\frac{dI_1}{dt} &= \int u_t \, dxdy \\ &= \int dy \left[u^2 - \nabla^2 u + 2u\frac{\partial \nabla^2 u}{\partial y}\right]_{-M}^{M} + \int dx \left[2u\frac{\partial \nabla^2 u}{\partial x}\right]_{-L}^{L}. \end{aligned} \tag{3.1.26}$$

When we employ the periodic bounbary condition on x-direction, we find

$$(\text{the 1st term of r.h.s. in eq.(3.1.26)}) = \int dy \left[u^2 - \nabla^2 u + 2u\frac{\partial \nabla^2 u}{\partial y}\right]_{-M}^{M} = 0. \tag{3.1.27}$$

In order to conserve I_1, we require the equation as following

$$(\text{the 2nd term of r.h.s. in eq.(3.1.26)}) = \int dx \left[2u\frac{\partial \nabla^2 u}{\partial x}\right]_{-L}^{L} \equiv 0. \tag{3.1.28}$$

That means

$$\left[2u\frac{\partial \nabla^2 u}{\partial x}\right]_{-L}^{L} = 0. \tag{3.1.29}$$

If the function $u(x, y, t)$ or $\nabla^2 u_x(x, y, t)$ decrease rapidly when $y \to \pm L$, this condition is satisfied. In this meaning, I_1 is *not always* conserved. In other words, for the localized solution, e.g. the Gaussian and lump solution, I_1 is conserved quantities.

Secondly, we consider $D_2 = u^2/2$ and I_2 of the form given by

$$I_2 = \int \frac{1}{2}u^2 \, dxdy. \tag{3.1.30}$$

Similarly, we calculate derivative by t as follows

$$\frac{dI_2}{dt} = \frac{d}{dt}\int \frac{1}{2}u^2 \, dxdy = \int \frac{d}{dt}\left(\frac{1}{2}u^2\right) dxdy = \int uu_t \, dxdy. \tag{3.1.31}$$

In order to obtain uu_t, we multiply eq.(3.1.16) by u and obtain

$$\begin{aligned}u \times (3.1.16) &\Rightarrow uu_t - 2u^2 u_x + u\nabla^2 u_x + 2uJ[\nabla^2 u, u] = 0 \\ &\Rightarrow uu_t = 2u^2 u_x - u\nabla^2 u_x - 2uJ[\nabla^2 u, u] \\ &= 2u^2 u_x - uu_{xxx} - uu_{xyy} - 2uJ[\nabla^2 u, u] \\ &= \frac{\partial}{\partial x}\left(\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left(u_x^2 + u_y^2\right)\right) + \frac{\partial}{\partial y}\left(-uu_{xy}\right) - 2uJ[\nabla^2 u, u]. \end{aligned} \tag{3.1.32}$$

By using indentify eq.(3.1.22) for $p = 1$, we obtain

$$uJ[\nabla^2 u, u] = \frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}\frac{u^2}{2}\right) - \frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}\frac{u^2}{2}\right). \tag{3.1.33}$$

Therefore, uu_t can be written of the form

$$uu_t = \frac{\partial}{\partial x}\left(\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left(u_x^2 + u_y^2\right) + u^2\frac{\partial \nabla^2 u}{\partial y}\right) + \frac{\partial}{\partial y}\left(-uu_{xy} - u^2\frac{\partial \nabla^2 u}{\partial x}\right). \tag{3.1.34}$$

Finally, we obtain dI_2/dt as following

$$\begin{aligned}\frac{dI_2}{dt} &= \int uu_t\, dxdy \\ &= \int dy\, \left[\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left(u_x^2 + u_y^2\right) + u^2\frac{\partial \nabla^2 u}{\partial y}\right]_{-M}^{M} - \int dx\, \left[uu_{xy} + u^2\frac{\partial \nabla^2 u}{\partial x}\right]_{-L}^{L}.\end{aligned} \tag{3.1.35}$$

We can easily find that eq.(3.1.35) requires the conditions given by

$$\text{1.periodic boundary condition on } x\text{-direction.} \tag{3.1.36}$$

$$\text{2.}u(x,y,t)|_{\pm L} = 0 \text{ or } \partial_x\partial_y u(x,y,t)|_{y=\pm L} = 0 \text{ and } \nabla^2 u(x,y,t)|_{y=\pm L} = C. \tag{3.1.37}$$

where C is constant. The functions decreasing rapidly satisfy 2nd condition eq.(3.1.37). When these conditions are satisfied, $dI_2/dt = 0$, i.e. I_2 is the conserved quantity.

Finally, we consider D_3 given by

$$D_3 = \frac{1}{3}u^3 + \frac{1}{2}(u_x^2 + u_y^2), \tag{3.1.38}$$

and I_3 is the form

$$I_3 = \int \left(\frac{1}{3}u^3 + \frac{1}{2}(u_x^2 + u_y^2)\right) dxdy. \tag{3.1.39}$$

We can verify similarly for I_3 as following

$$\frac{dI_3}{dt} = \frac{d}{dt}\int D_3\, dxdy = \int \frac{d}{dt}\left(\frac{1}{3}u^3 + \frac{1}{2}(u_x^2 + u_y^2)\right) dxdy. \tag{3.1.40}$$

The 1st term of r.h.s. of eq.(3.1.40) becomes following form

$$(\text{ the 1st term of eq.(3.1.40) }) \Rightarrow \frac{d}{dt}\left(\frac{1}{3}u^3\right) = u^2 u_t. \tag{3.1.41}$$

Then, we multiply eq.(3.1.16) by u^2 and use it

$$\begin{aligned}\Rightarrow u^2 u_t &= 2u^3 u_x - u^2\nabla^2 u_x - 2u^2 J[\nabla^2 u, u] \\ &= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 - u^2\nabla^2 u\right) + 2uu_x\nabla^2 u - 2u^2 J[\nabla^2 u, u].\end{aligned} \tag{3.1.42}$$

Next, the second term of r.h.s. of eq.(3.1.40) becomes following form

$$\begin{aligned}(\text{ the 2nd term of eq.(3.1.40) }) \Rightarrow \frac{d}{dt}\left(\frac{1}{2}(u_x^2+u_y^2)\right) \\ = u_x u_t + u_y u_t \\ = \frac{\partial}{\partial x}(u_x u_t) + \frac{\partial}{\partial y}(u_y u_t) - u_t \nabla^2 u. \end{aligned} \tag{3.1.43}$$

By using these two relations, we make eq.(3.1.42) + eq.(3.1.43) and obtain following results

$$\begin{aligned}
&\Rightarrow \frac{d}{dt}\left(\frac{1}{3}u^3 + \frac{1}{2}(u_x^2+u_y^2)\right) \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 - u^2\nabla^2 u\right) + \frac{\partial}{\partial x}(u_x u_t) + \frac{\partial}{\partial y}(u_y u_t) + 2uu_x\nabla^2 u - u_t\nabla^2 u - 2u^2 J[\nabla^2 u, u] \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 - u^2\nabla^2 u + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) - 2u^2 J[\nabla^2 u, u] - \underline{(u_t - 2uu_x)}\nabla^2 u \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 - u^2\nabla^2 u + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) - 2u^2 J[\nabla^2 u, u] + \underbrace{(\nabla^2 u_x + 2J[\nabla^2 u, u])}_{\because \text{eq.(3.1.16)}}\nabla^2 u \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 - u^2\nabla^2 u + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) + \frac{\partial}{\partial x}\left(\frac{1}{2}(\nabla^2 u)^2\right) + \underbrace{2\,\nabla^2 u\, J[\nabla^2 u, u]}_{(a)} - \underbrace{2u^2 J[\nabla^2 u, u]}_{(b)},
\end{aligned} \tag{3.1.44}$$

$(a) : 2\,\nabla^2 u\, J[\nabla^2 u, u]$

$$\begin{aligned}
&\nabla^2 u\, J[\nabla^2 u, u] \\
&= \nabla^2 u \frac{\partial \nabla^2 u}{\partial x}\frac{\partial u}{\partial y} - \nabla^2 u \frac{\partial \nabla^2 u}{\partial y}\frac{\partial u}{\partial x} \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}(\nabla^2 u)^2\right)\frac{\partial u}{\partial y} - \frac{\partial}{\partial y}\left(\frac{1}{2}(\nabla^2 u)^2\right)\frac{\partial u}{\partial x} \\
&= \frac{\partial}{\partial y}\left\{\frac{\partial}{\partial x}\left(\frac{1}{2}(\nabla^2 u)^2\right)u\right\} - \frac{\partial^2}{\partial x \partial y}\left(\frac{1}{2}(\nabla^2 u)^2\right)u - \frac{\partial}{\partial x}\left\{\frac{\partial}{\partial y}\left(\frac{1}{2}(\nabla^2 u)^2\right)u\right\} + \frac{\partial^2}{\partial y \partial y}\left(\frac{1}{2}(\nabla^2 u)^2\right)u \\
&= \frac{\partial}{\partial y}\left\{\frac{\partial}{\partial x}\left(\frac{1}{2}(\nabla^2 u)^2\right)u\right\} - \frac{\partial}{\partial x}\left\{\frac{\partial}{\partial y}\left(\frac{1}{2}(\nabla^2 u)^2\right)u\right\},
\end{aligned}$$

$$\therefore 2\,\nabla^2 u\, J[\nabla^2 u, u] = \frac{\partial}{\partial y}\left\{\frac{\partial}{\partial x}\left((\nabla^2 u)^2\right)u\right\} - \frac{\partial}{\partial x}\left\{\frac{\partial}{\partial y}\left((\nabla^2 u)^2\right)u\right\}. \tag{3.1.45}$$

$(b) : -2u^2\, J[\nabla^2 u, u]$

$$-2u^2\, J[\nabla^2 u, u] = -\frac{\partial}{\partial y}\left(\frac{\partial \nabla^2 u}{\partial x}\frac{2}{3}u^3\right) + \frac{\partial}{\partial x}\left(\frac{\partial \nabla^2 u}{\partial y}\frac{2}{3}u^3\right), \tag{3.1.46}$$

where we use identify eq.(3.1.22) for $p = 2$. Therefore, we finally obtain the form dI_3/dt as follows

$$\begin{aligned}
\frac{dI_3}{dt} = &\int dy\, \left[\frac{1}{2}u^4 - u^2\nabla^2 u + u_x u_t + \frac{1}{2}(\nabla^2 u)^2 - u\frac{\partial}{\partial y}\left((\nabla^2 u)^2\right) + \frac{2}{3}u^3\frac{\partial \nabla^2 u}{\partial y}\right]_{-M}^{M} \\
&+ \int dx\, \left[u_y u_t + u\frac{\partial}{\partial x}\left((\nabla^2 u)^2\right) - \frac{2}{3}u^3\frac{\partial \nabla^2 u}{\partial x}\right]_{-L}^{L}.
\end{aligned} \tag{3.1.47}$$

When we employ periodic condition on x-direction, the 1st term of r.h.s. of eq.(3.1.47) becomes to zero. Consider the 2nd term of r.h.s. of eq.(3.1.47), we can expand it to this

$$(\text{the 2nd term}) = u_y(2uu_x - \nabla^2 u_x - 2J[\nabla^2 u, u]) + 2u(\nabla^2 u)(\nabla^2 u_x) - \frac{2}{3}u^3\nabla^2 u_x.$$

Therefore, the 2nd term of r.h.s. of eq.(3.1.47) consists of $u, u_x, u_y, u_{xx}, u_{yy}, u_{xxx}, u_{yyy}, u_{xyy}$ and u_{xxy}. In order to get the conserved quantity I_3, we have to impose the boundary conditions to these functions on $y = \pm L$. Concretely speaking, when $u(x, y, t)$ has the form decreasing rapidly to border of region, these conditions are satisfied. In other words, in this time, I_3 is also conseved quantity.

3.1.3 Exact solution to KdV equation in one dimension

In this section, we verify the exact solution to KdV equation in one dimension. Before explanation, we show some mathematical relations as follows

$$(\sinh x)' = \cosh x, \tag{3.1.48}$$
$$(\cosh x)' = \sinh x, \tag{3.1.49}$$
$$(\tanh x)' = \frac{1}{\cosh^2 x} = \operatorname{sech}^2 x, \tag{3.1.50}$$
$$(\operatorname{sech} x)' = \left(\frac{1}{\cosh x}\right)' = -\tanh x \operatorname{sech} x, \tag{3.1.51}$$
$$\cosh^2 x - \sinh^2 x = 1. \tag{3.1.52}$$

Using mathematical relations above, we substitute directly

$$u(x,t) = \frac{1}{2}\lambda \operatorname{sech}^2\left(\frac{1}{2}\sqrt{\lambda}(x - \lambda t - x_0)\right) \tag{3.1.53}$$

into KdV equation in one dimention given by

$$u_t + 6uu_x + u_{xxx} = 0. \tag{3.1.54}$$

We rewrite $u(x,t)$ eq.(3.1.54) by using new variable $s(x,t)$ as

$$u(s(x,t)) = \frac{\lambda}{2}\operatorname{sech}^2 s, \qquad s = \frac{1}{2}\sqrt{\lambda}(x - \lambda t - x_0). \tag{3.1.55}$$

Consequently, we obtain the derivatives by each variable as follows

$$\frac{\partial s}{\partial t} = -\frac{\sqrt{\lambda}}{2}\lambda, \qquad \frac{\partial s}{\partial x} = \frac{\sqrt{\lambda}}{2} \tag{3.1.56}$$

We calculate each term of eq.(3.1.54) as following :

1. u_t

$$u_t \Rightarrow \frac{\partial u}{\partial t} = \frac{\partial s}{\partial t}\frac{\partial u}{\partial s}, \tag{3.1.57}$$
$$\frac{\partial u}{\partial s} = \frac{\lambda}{2}\frac{\partial}{\partial s}\operatorname{sech}^2 s = \lambda \operatorname{sech} s \frac{\partial}{\partial s}\operatorname{sech} s = -\lambda \tanh s \operatorname{sech}^2 s, \tag{3.1.58}$$
$$\therefore u_t = -\frac{\sqrt{\lambda}}{2}\lambda \cdot \left(-\lambda \tanh s \operatorname{sech}^2 s\right) = \frac{\lambda^2}{2}\sqrt{\lambda}\tanh s \operatorname{sech}^2 s. \tag{3.1.59}$$

2. $6uu_x$

$$u_x \Rightarrow \frac{\partial u}{\partial x} = \frac{\partial s}{\partial x}\frac{\partial u}{\partial s} = \frac{\sqrt{\lambda}}{2}\left(-\lambda \tanh s \operatorname{sech}^2 s\right) = -\frac{\lambda}{2}\sqrt{\lambda}\tanh s \operatorname{sech}^2 s, \tag{3.1.60}$$
$$\therefore 6uu_x = 6 \cdot \frac{\lambda}{2}\operatorname{sech}^2 s \cdot \left(-\frac{\lambda}{2}\sqrt{\lambda}\tanh s \operatorname{sech}^2 s\right) = -\frac{3}{2}\lambda^2\sqrt{\lambda}\tanh s \operatorname{sech}^4 s. \tag{3.1.61}$$

3. u_{xxx}

$$
\begin{aligned}
u_{xxx} \Rightarrow \frac{\partial^3 u}{\partial x^3} &= \frac{\partial^2}{\partial x^2} u_x \\
&= \frac{\partial^2}{\partial x^2}\left(-\frac{\lambda}{2}\sqrt{\lambda}\tanh s \operatorname{sech}^2 s\right) \\
&= -\frac{\lambda}{2}\sqrt{\lambda}\frac{\partial}{\partial x}\left\{\underline{\left(\frac{\partial}{\partial x}\tanh s\right)}\operatorname{sech}^2 s + \tanh s \underline{\left(\frac{\partial}{\partial x}\operatorname{sech}^2 s\right)}\right\}, \quad (3.1.62)
\end{aligned}
$$

$$
\underline{\frac{\partial}{\partial x}\tanh s} = \frac{\partial s}{\partial x}\frac{\partial}{\partial s}\tanh s = \frac{\sqrt{\lambda}}{2}\operatorname{sech}^2 s, \quad (3.1.63)
$$

$$
\underline{\frac{\partial}{\partial x}\operatorname{sech}^2 s} = \frac{\partial s}{\partial x}\frac{\partial}{\partial s}\operatorname{sech}^2 s = \frac{\sqrt{\lambda}}{2}\cdot 2\operatorname{sech} s\,\frac{\partial}{\partial s}\operatorname{sech} s = -\sqrt{\lambda}\tanh s \operatorname{sech}^2 s, \quad (3.1.64)
$$

$$
\begin{aligned}
\Rightarrow \frac{\partial^3 u}{\partial x^3} &= -\frac{\lambda}{2}\sqrt{\lambda}\frac{\partial}{\partial x}\left\{\frac{\sqrt{\lambda}}{2}\operatorname{sech}^4 s - \sqrt{\lambda}\tanh^2 s \operatorname{sech}^2 s\right\} \\
&= -\frac{\lambda^2}{2}\left\{\underline{\frac{\partial}{\partial x}\left(\frac{1}{2}\operatorname{sech}^4 s\right)} - \underline{\frac{\partial}{\partial x}\left(\tanh^2 s \operatorname{sech}^2 s\right)}\right\}, \quad (3.1.65)
\end{aligned}
$$

$$
\begin{aligned}
\underline{\frac{\partial}{\partial x}\left(\frac{1}{2}\operatorname{sech}^4 s\right)} &= \frac{1}{2}\frac{\partial s}{\partial x}\frac{\partial}{\partial s}\left(\operatorname{sech}^4 s\right) \\
&= \frac{1}{2}\cdot\frac{\sqrt{\lambda}}{2}\cdot 4\operatorname{sech}^3 s\frac{\partial}{\partial s}\operatorname{sech} s \\
&= -\sqrt{\lambda}\operatorname{sech}^3 s\cdot\tanh s\operatorname{sech} s \\
&= -\sqrt{\lambda}\tanh s\operatorname{sech}^4 s, \quad (3.1.66)
\end{aligned}
$$

$$
\begin{aligned}
\underline{\frac{\partial}{\partial x}\left(\tanh^2 s\operatorname{sech}^2 s\right)} &= \left(\frac{\partial}{\partial x}\tanh^2 s\right)\operatorname{sech}^2 s + \tanh^2 s\frac{\partial}{\partial x}\left(\operatorname{sech}^2 s\right) \\
&= \frac{\partial s}{\partial x}\frac{\partial}{\partial s}(\tanh^2 s)\operatorname{sech}^2 s + \tanh^2 s\frac{\partial s}{\partial x}\frac{\partial}{\partial s}\left(\operatorname{sech}^2 s\right) \\
&= \frac{\sqrt{\lambda}}{2}\cdot 2\tanh s\operatorname{sech}^4 s + \frac{\sqrt{\lambda}}{2}\tanh^2 s\cdot 2\operatorname{sech} s(-\tanh s\operatorname{sech} s) \\
&= \sqrt{\lambda}\tanh s\operatorname{sech}^4 s - \sqrt{\lambda}\tanh^3 s\operatorname{sech}^2 s, \quad (3.1.67)
\end{aligned}
$$

$$
\begin{aligned}
\therefore u_{xxx} \Rightarrow \frac{\partial^3 u}{\partial x^3} &= -\frac{\lambda^2}{2}\left(-\sqrt{\lambda}\tanh s\operatorname{sech}^4 s - \sqrt{\lambda}\tanh s\operatorname{sech}^4 s + \sqrt{\lambda}\tanh^3 s\operatorname{sech}^2 s\right) \\
&= -\frac{\lambda^2}{2}\sqrt{\lambda}\left(-2\tanh s\operatorname{sech}^4 s + \tanh^3 s\operatorname{sech}^2 s\right) \quad (3.1.68)
\end{aligned}
$$

From these relations, the l.h.s. of the eq.(3.1.54) becomes

$$
\begin{aligned}
& u_t + 6uu_x + u_{xxx} \\
&= \frac{\lambda^2}{2}\sqrt{\lambda}\tanh s\,\mathrm{sech}^2 s - \frac{3}{2}\lambda^2\sqrt{\lambda}\tanh s\,\mathrm{sech}^4 s - \frac{\lambda^2}{2}\sqrt{\lambda}\left(-2\tanh s\,\mathrm{sech}^4 s + \tanh^3\,\mathrm{sech}^2 s\right) \\
&= \lambda^2\sqrt{\lambda}\left(\frac{1}{2}\tanh s\,\mathrm{sech}^2 s - \frac{1}{2}\tanh\,\mathrm{sech}^4 s - \frac{1}{2}\tanh^3\,\mathrm{sech}^2 s\right) \\
&= \frac{\lambda^2\sqrt{\lambda}}{2}\tanh s\,\mathrm{sech}^2 s\left\{\left(1 - \mathrm{sech}^2 s\right) - \tanh^2 s\right\} \\
&= \frac{\lambda^2\sqrt{\lambda}}{2}\tanh s\,\mathrm{sech}^2 s\left(\frac{\cosh^2 s - 1}{\cosh^2 s} - \frac{\sinh^2 s}{\cosh^2 s}\right) \\
&= \frac{\lambda^2\sqrt{\lambda}}{2}\tanh s\,\mathrm{sech}^2 s\left(\frac{\cosh^2 s - \sinh^2 s - 1}{\cosh^2 s}\right) \\
&= 0. \qquad (\because \cosh^2 x - \sinh^2 x = 1)
\end{aligned}
\tag{3.1.69}
$$

Therefore, we find that $u(x,t) = \lambda/2\,\mathrm{sech}^2(\sqrt{\lambda}(x - \lambda t - x_0)/2)$ is solution to KdV equation in one dimension : $u_t + 6uu_x + u_{xxx} = 0$.

3.2 Zakharov-Kuznetsov equation

In this section, we show some properties of ZK equation. Many of them are seen in Iwasaki et al. and Klein et al [2, 3]. However, we would like to propose the different approach by using approximation of two gaussian as initial condition. If it is necessary for you, we hope these help you.

3.2.1 Exact solution to ZK equation

In section1.1.2, we explained that ZK equation has solitary wave solution which obtained by numerical calculation. On the other hand, ZK equation has also analytical solution which are generalized from 1-soliton solution in KdV equation (shown in section3.1.3). We show that in this section.

We rewrite KdV equation eq.(3.1.54) in terms of other variables ξ, τ and $v(\xi, \tau)$ as follows

$$v_\tau + 6vv_\xi + v_{\xi\xi\xi} = 0. \tag{3.2.1}$$

This equation eq.(3.2.1) is obtained from ZK equation of the form[*1] [6]

$$u_t + 2uu_x + u_{xxx} + u_{xyy} = 0, \tag{3.2.2}$$

where transformation during v, ξ, τ, u, x, y, t are given by

$$\xi = x + qy, \quad \tau = (1+q^2)t, \quad v = \frac{u}{3(1+q^2)}, \quad (q : constant). \tag{3.2.3}$$

Using these transformations, we can obtain exact solution of eq.(3.2.2) from the solution to eq.(3.2.1) as follows

$$\begin{aligned} u(x,y,t) &= 3(1+q^2)v(\xi,\tau) \\ &= 3(1+q^2)\cdot\frac{1}{2}\lambda\,\mathrm{sech}^2\left(\frac{1}{2}\sqrt{\lambda}(x+qy-\lambda(1+q^2)t-\xi_0)\right) \\ &= \frac{3(1+q^2)\lambda}{2}\,\mathrm{sech}^2\left(\frac{1}{2}\sqrt{\lambda}(x+qy-\lambda(1+q^2)t-\xi_0)\right). \end{aligned} \tag{3.2.4}$$

We can verify similarly as eq.(3.1.53) that eq.(3.2.4) is solution to eq.(3.2.2). Since the solution eq.(3.2.4) is *not localized* solution, however, it has not always the conserved quantities. From in this point, we don't employ this solution as initial condition because we desire the vortex which localized solution. Moreover, we show the proof of eq.(3.2.1)-(3.2.3) following. If you need, please refer to that.

[*1] We write $\nabla^2 u_x$ to explicit form $u_{xxx} + u_{xyy}$.

proof of eq.(3.2.1)-(3.2.3)

1. u_t

$$\begin{aligned} u_t = \frac{\partial u}{\partial t} &= \frac{\partial}{\partial t} 3(1+q^2)v \\ &= \frac{\partial \tau}{\partial t}\frac{\partial}{\partial \tau} 3(1+q^2)v \\ &= 3(1+q^2)^2 \frac{\partial v}{\partial \tau}. \end{aligned} \tag{3.2.5}$$

2. $2uu_x$

$$\begin{aligned} 2uu_x = 2u\frac{\partial u}{\partial x} &= 6(1+q^2)v\frac{\partial}{\partial x} 3(1+q^2)v \\ &= 6\cdot 3(1+q^2)^2 v \frac{\partial \xi}{\partial x}\frac{\partial v}{\partial \xi} \\ &= 6\cdot 3(1+q^2)^2 v \frac{\partial v}{\partial \xi}. \end{aligned} \tag{3.2.6}$$

3. u_{xxx}

$$\begin{aligned} u_{xxx} = \frac{\partial^3 u}{\partial x^3} &= \left(\frac{\partial \xi}{\partial x}\right)^3 \frac{\partial^3}{\partial x^3} 3(1+q^2)v \\ &= 3(1+q^2)\frac{\partial^3 v}{\partial \xi^3}. \end{aligned} \tag{3.2.7}$$

4. u_{xyy}

$$\begin{aligned} u_{xyy} = \frac{\partial^3 u}{\partial x \partial y^2} &= \frac{\partial \xi}{\partial x}\frac{\partial}{\partial \xi}\cdot\left(\frac{\partial \xi}{\partial y}\right)^2 \frac{\partial^2}{\partial \xi^2} 3(1+q^2)v \\ &= 3q^2(1+q^2)\frac{\partial^3 v}{\partial \xi^3}. \end{aligned} \tag{3.2.8}$$

5. $u_t + 2uu_x + u_{xxx} + u_{xyy} = 0$

$$\begin{aligned} &\Rightarrow 3(1+q^2)^2\frac{\partial v}{\partial \tau} + 6\cdot 3(1+q^2)^2 v\frac{\partial v}{\partial \xi} + 3(1+q^2)\frac{\partial^3 v}{\partial \xi^3} + 3q^2(1+q^2)\frac{\partial^3 v}{\partial \xi^3} = 0, \\ &\Rightarrow \frac{\partial v}{\partial \tau} + 6v\frac{\partial v}{\partial \xi} + \frac{\partial^3 v}{\partial \xi^3} = 0. \end{aligned} \tag{3.2.9}$$

This is the same as to equation eq.(3.2.1).

3.2.2 Conserved quantities of ZK equation [2, 4, 10]

For the ZK equation of the form

$$u_t + 2uu_x + \nabla^2 u_x = 0, \tag{3.2.10}$$

under *the certain conditions*, it has the conserved quantities given by

$$M[u(t)] = \iint u\, dS, \tag{3.2.11}$$

$$P[u(t)] = \iint \frac{1}{2}u^2\, dS, \tag{3.2.12}$$

$$E[u(t)] = \iint \left(\frac{1}{2}({u_x}^2 + {u_y}^2) - \frac{1}{3}u^3\right) dS, \tag{3.2.13}$$

where we consider $x \times y = [-M, M] \times [-L, L]$ as interval of integral (which is the same as section3.1.2). In this section, we verify that eq.(3.2.11)-(3.2.13) are the conserved quantities of ZK equation eq.(3.2.10). Similarly to section3.1.2, the conserved quantities are defined with D_n and they are given by following forms

$$M[u(t)] : D_1 = u, \tag{3.2.14}$$

$$P[u(t)] : D_2 = \frac{1}{2}u^2, \tag{3.2.15}$$

$$E[u(t)] : D_3 = \frac{1}{2}({u_x}^2 + {u_y}^2) - \frac{1}{3}u^3. \tag{3.2.16}$$

In order to verify eq.(3.2.11)-(3.2.13), we calculate derivatives of M , P and E by variable t. We show these results as following.

1. $M[u]$ $(D_1 = u)$

$$\begin{aligned}\frac{dM[u(t)]}{dt} &= \iint \frac{\partial u}{\partial t}\, dxdy \\ &= \iint (-2uu_x - u_{xxx} - u_{xyy})\, dxdy \\ &= \iint \frac{\partial}{\partial x}\left(-u^2 - u_{xx} - u_{yy}\right) dxdy \\ &= \int dy \left[-u^2 - u_{xx} - u_{yy}\right]_{-M}^{M}. \end{aligned} \tag{3.2.17}$$

   When we employ the periodic condition on x-direction, the r.h.s. of eq.(3.2.17) becomes to zero, i.e. $M[u]$ is conserved quantity.

2. $P[u]$ $(D_2 = u^2/2)$

$$\begin{aligned}\frac{dP[u(t)]}{dt} &= \frac{1}{2}\iint \frac{\partial u^2}{\partial t}\, dxdy \\ &= \iint uu_t\, dxdy. \end{aligned} \tag{3.2.18}$$

Then, we multiply both sides eq.(3.2.10) by u and obtain

$$\begin{aligned}(3.2.10)\times u &\Rightarrow uu_t + 2u^2u_x + uu_{xxx} + uu_{xyy} = 0\\ &\Rightarrow uu_t = -2u^2u_x - uu_{xxx} - uu_{xyy}\\ &\Rightarrow uu_t = \frac{\partial}{\partial x}\left(-\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left({u_x}^2 + {u_y}^2\right)\right) + \frac{\partial}{\partial y}\left(-uu_{xy}\right).\end{aligned} \tag{3.2.19}$$

By substituting eq.(3.2.19) into eq.(3.2.18), we finally obtain the equation of the form

$$\begin{aligned}\frac{dP[u(t)]}{dt} &= \iint\left\{\frac{\partial}{\partial x}\left(-\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left({u_x}^2 + {u_y}^2\right)\right) + \frac{\partial}{\partial y}\left(-uu_{xy}\right)\right\}dxdy\\ &= \int dy\left[-\frac{2}{3}u^3 - uu_{xx} + \frac{1}{2}\left({u_x}^2 + {u_y}^2\right)\right]_{-M}^{M} + \int dx\left[-uu_{xy}\right]_{-L}^{L}.\end{aligned} \tag{3.2.20}$$

For the 1st term of eq.(3.2.20), it becomes to zero when we employ the periodic condition on x-direction. On the other hand, for the 2nd term of eq.(3.2.20), this part requires condition given by

$$\left[-uu_{xy}\right]_{-L}^{L} = 0. \tag{3.2.21}$$

Therefore, $P[u]$ is not always conserved. In other words, for localized solution, $P[u]$ is also conserved quantity.

3. $E[u]$ $(D_3 = ({u_x}^2 + {u_y}^2)/2 - u^3/3)$

$$\begin{aligned}\frac{dE[u(t)]}{dt} &= \iint\frac{\partial}{\partial t}\left(\frac{1}{2}({u_x}^2 + {u_y}^2) - \frac{1}{3}u^3\right)dxdy\\ &= \iint\left(u_xu_{xt} + u_yu_{yt} - u^2u_t\right)dxdy.\end{aligned} \tag{3.2.22}$$

Again we multiply both sides eq.(3.2.10) by $-u^2$ and obtain

$$\begin{aligned}(3.2.10)\times(-u^2) &\Rightarrow -u^2u_t - 2u^3u_x - u^2u_{xxx} - u^2u_{xyy} = 0\\ &\Rightarrow -u^2u_t = 2u^3u_x + u^2u_{xxx} + u^2u_{xyy}\\ &\Rightarrow -u^2u_t = \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2u_{xx} + u^2u_{yy}\right) - 2uu_x(u_{xx} + u_{yy}).\end{aligned} \tag{3.2.23}$$

Next, we calculate $u_xu_{xt} + u_yu_{yt}$ as follows

$$\begin{aligned}u_xu_{xt} + u_yu_{yt} &= u_x\frac{\partial}{\partial x}(u_t) + u_y\frac{\partial}{\partial y}(u_t)\\ &= \frac{\partial}{\partial x}\left(u_xu_t\right) - u_{xx}u_t + \frac{\partial}{\partial y}(u_yu_t) - u_{yy}u_t\\ &= \frac{\partial}{\partial x}(u_xu_t) + \frac{\partial}{\partial y}(u_yu_t) - u_t(u_{xx} + u_{yy}).\end{aligned} \tag{3.2.24}$$

From eq.(3.2.23) and eq.(3.2.24), finally we get the r.h.s. of eq.(3.2.22) of the form

$$
\begin{aligned}
& u_x u_{xt} + u_y u_{yt} - u^2 u_t \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy}\right) - 2uu_x(u_{xx}+u_{yy}) + \frac{\partial}{\partial x}(u_x u_t) + \frac{\partial}{\partial y}(u_y u_t) - u_t(u_{xx}+u_{yy}) \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy} + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) - (u_{xx}+u_{yy})\underline{(u_t + 2uu_x)} \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy} + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) + (u_{xx}+u_{yy})\underbrace{(u_{xxx}+u_{xyy})}_{\because \text{eq.(3.2.10)}} \\
&= \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy} + u_x u_t\right) + \frac{\partial}{\partial y}(u_y u_t) + \frac{\partial}{\partial x}\left(\frac{1}{2}(u_{xx}+u_{yy})^2\right). \qquad (3.2.25)
\end{aligned}
$$

$$
\begin{aligned}
\frac{dE[u(t)]}{dt} &= \iint \left\{ \frac{\partial}{\partial x}\left(\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy} + u_x u_t + \frac{1}{2}(u_{xx}+u_{yy})^2\right) + \frac{\partial}{\partial y}(u_y u_t)\right\} dxdy \\
&= \int dy \left[\frac{1}{2}u^4 + u^2 u_{xx} + u^2 u_{yy} + u_x u_t + \frac{1}{2}(u_{xx}+u_{yy})^2\right]_{-M}^{M} + \int dx \left[u_y u_t\right]_{-L}^{L}. \qquad (3.2.26)
\end{aligned}
$$

From the expression of eq.(3.2.26), in order to obtain the conserved quantity, we require $u(x,y,t)$ to satisfy the conditions as follows

(a) The periodic condition on x-direction.

(b) On y-direction,

$$
\underline{u_y \to 0 \ \ (|y| \to L)}, \text{ or } \underline{u, u_x, u_{xxx} \text{ and } u_{xyy} \to 0 \ \ (|y| \to L)}, \qquad (3.2.27)
$$

because u_t can be rewritten by eq.(3.2.10) as $u_t = -2uu_x - u_{xxx} - u_{xyy}$.

When these conditions are satisfied, $E[u]$ becomes to the conserved quantity.

According to these results, the quantities M, P and E are not always conserved and that depends on the boundary conditions for solution. Therefore, we don't employ the exact solution to ZK equation (shown in section3.2.1). If we choose the localized solution as initial condition of eq.(3.2.10), these quantities are conserved. This is the reason why we construct lump solution. The behavior of these conserved quantities are shown in next section.

3.2.3 Behavior of one lump solution in ZK equation [2]

In this section, we show the behavior of one lump solution in ZK equation. They are shown by Iwasaki et al. [2]. In addition to that, we make new solutions which consist of two gaussian. We compare these solutions and verify the accuracy of the new solutions.

We show again ZK equation as follows

$$u_t + 2uu_x + \nabla^2 u_x = 0. \tag{3.2.28}$$

In section1.1.2, we explained that equation eq.(3.2.28) has solitary wave solution of the form

$$u(x, y, t) = Q(x - ct, y), \tag{3.2.29}$$

where c is velocity of solitary wave. By substituting eq.(3.2.29) into the equation eq.(3.2.28), we obtain the equation which decides the form of $Q(x - ct, y)$ as following

$$-cQ + Q_{xx} + Q_{yy} + Q^2 = 0. \tag{3.2.30}$$

Moreover, we suppose $Q = Q(r),\ r \equiv \sqrt{x^2 + y^2}$ and obtain the equation as follows

$$\frac{d^2Q}{dr^2} + \frac{1}{r}\frac{dQ}{dr} - cQ + Q^2 = 0. \tag{3.2.31}$$

They are mentioned in section1.1.2. We obtain the solution of eq.(3.2.31) by *shooting method* and show this result in figure3.1. In terms of its appearance, it is called "bell-shaped solitary wave" solution by Iwasaki et al [2]. However, we call it "lump solution" for generality.

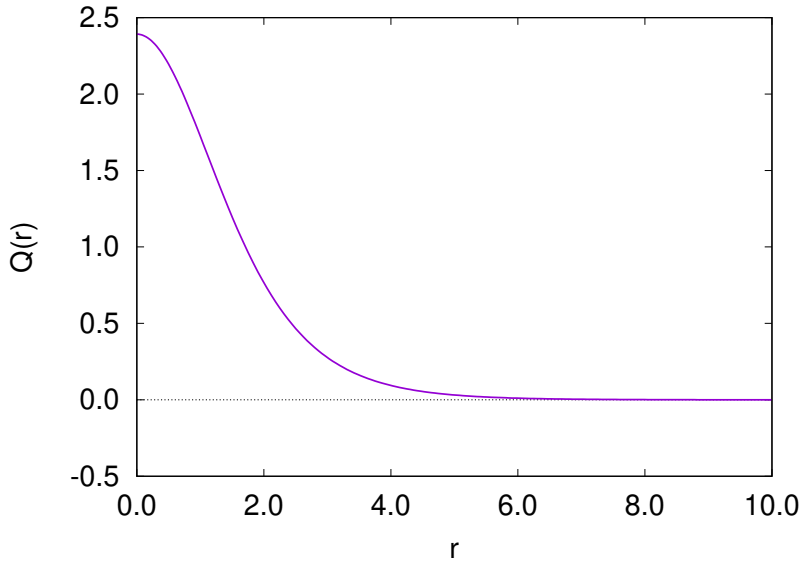


Figure3.1: Result of shooting method for $c = 1.0$. We get the value as $Q(0) = 2.39196$ by shooting method with boundary condition $|Q_r| \to 0 \ \ (r \to \infty)$.

In the previous research, they investigate the stability of pulse and collision of two pulses. From their work, the one pulse travel along to x-axis without deformation. In the collision of two pulses, if their amplitudes are similar, the pulses behave approximately soliton, i.e. their collision is elastic. If their amplitudes are dissimilar, the strong pulse becomes stronger and the weak one becomes weaker with generation of ripples.

In this section, we verify that both original lump solution and our lump solution propagate with no deformation. Before explanation about that, we show that how we construct new lump solution. Our idea is to use numerical lump solution of eq.(3.2.31) easily, i.e. we would like to use lump solutions without shooting method. In order to achieve our purpose, we reproduce the numerical lump solution by the superposition of gaussian as follows

$$f_N(r) = \sum_{k=1}^{N} A_k \exp\left[-\left(\frac{r}{B_k}\right)^2\right], \tag{3.2.32}$$

where A_k, B_k are parameters fitting via numerical results of eq.(3.2.31). For simplicity, we employ $N = 2$ in eq.(3.2.32) and we use Levenberg-Marquardt method [5] to decide parameters A_k and B_k.

We show numerical results, fitting results and residual of them in the case of $c = 1.0, c = 4.0$ and $c = 4.4$ in Figure3.2-Figure3.7. For classfication, we call numerical results "numerical lump (solution)" and fitting results "fitting lump (solution)".

- In Figure3.2 and Figure3.3, we show the case of $c = 1.0$. Plotted points are numerical lump and solid line is fitting lump in Figure3.2. In Figure3.3, residual of numerical lump and fitting lump defined by
$$\Delta Q \equiv Q_{num} - Q_{fit}, \tag{3.2.33}$$
are shown.
- In Figure3.4 and Figure3.5, the case of $c = 4.0$ are shown.
- In Figure3.6 and Figure3.7, the case of $c = 4.4$ are shown.

We find that superposition of two gaussian can reproduce numerical results within following accuracy. Concretely speaking, for $c = 1.0$, magnitude of residuals of numerical lump and fitting lump are less than 0.8×10^{-3}. In the case of $c = 4.0$, it is smaller than 3.0×10^{-2}. For $c = 4.4$, magnitude of residuals is less than 4.0×10^{-2} in Figure3.7.

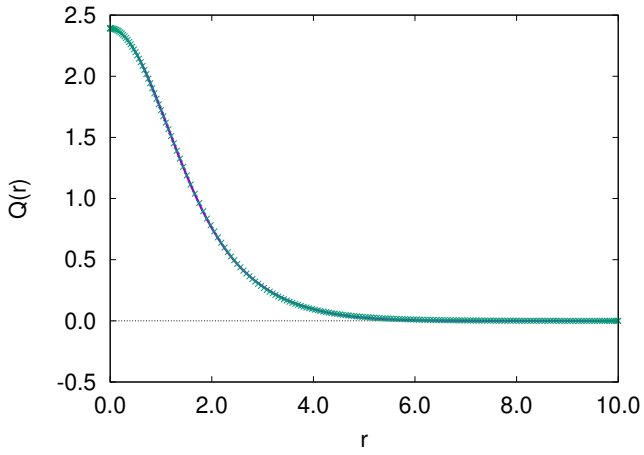


Figure3.2: The plotted points are numerical lump and solid line is fitting lump ($c = 1.0$).

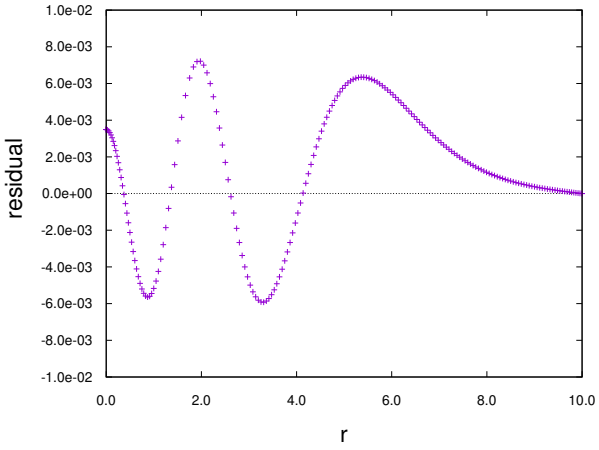


Figure3.3: Residual are plotted. Their magnitudes are no less than 0.8×10^{-3} ($c = 1.0$).

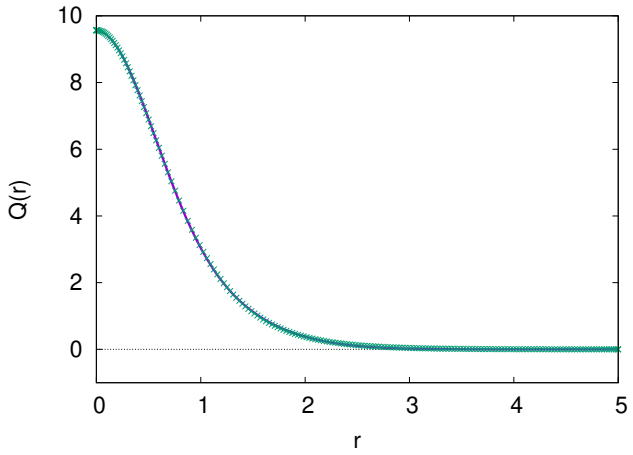


Figure3.4: Numerical lump and fitting lump in the case of $c = 4.0$.

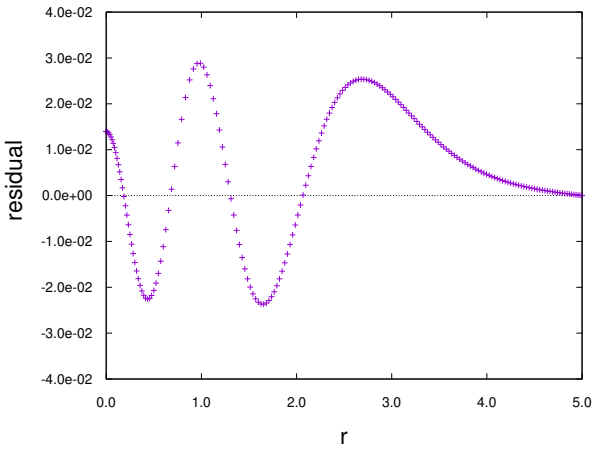


Figure3.5: Residual in the case of $c = 4.0$.

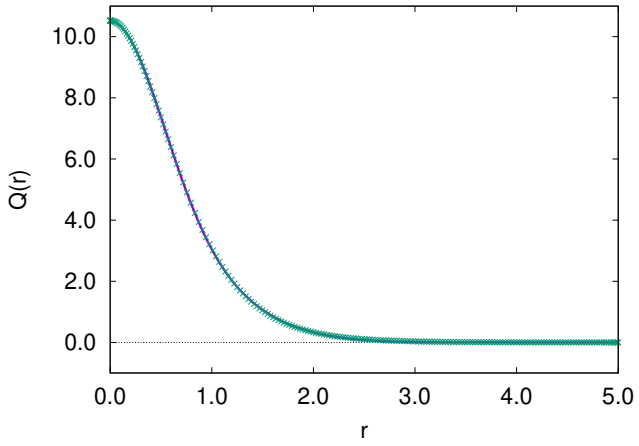


Figure3.6: Numerical lump and fitting lump in the case of $c = 4.4$.

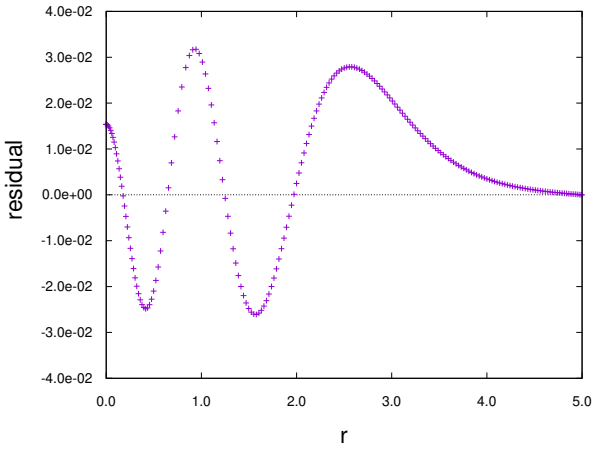


Figure3.7: Residual in the case of $c = 4.4$.

Table3.1: Parameters A_k and B_k fitted by LM method. Fitting accuracy is given by magnitude of sum sq.

c	k	A_k	B_k	sum sq. [Before fitting]
$c = 1.0$	1	0.9468	2.6348	0.00273
	2	1.4417	1.4711	[34.9972]
$c = 2.0$	1	1.8915	1.8635	0.01087
	2	2.8851	1.0406	[4.4052]
$c = 2.2$	1	2.0805	1.7768	0.01316
	2	3.1737	0.9922	[2.8352]
$c = 3.0$	1	2.8389	1.5213	0.02412
	2	4.3260	0.8496	[0.7476]
$c = 3.3$	1	3.1216	1.4507	0.02942
	2	4.7598	0.8101	[1.38811]
$c = 4.0$	1	3.7872	1.3174	0.04367
	2	5.7667	0.7356	[24.2994]
$c = 4.4$	1	6.3435	0.7013	0.05291
	2	4.1657	1.2561	[82.7507]

The fitting parameters A_k, B_k are shown in table3.1. Its accuracy is given by magnitude of sum square (sum sq.) defined by the sum of square of residual error as follows

$$\text{sum sq.} \equiv \sum_{m=1}^{M} \left[f(x_m) - f_m^{opt} \right]^2 , \tag{3.2.34}$$

where $f(x_m)$ is numerical results and f_m^{opt} is value of fitting function on sample point x_m.

Next, we show that fitting lump is sufficient to examine the previous research. In order to check its accuracy, we use the conserved quantities for ZK equation given by

$$M = \iint u \, dS, \tag{3.2.35}$$

$$P = \iint \frac{1}{2}u^2 \, dS, \tag{3.2.36}$$

$$E = \iint \left(\frac{1}{2}({u_x}^2 + {u_y}^2) - \frac{1}{3}u^3 \right) dS. \tag{3.2.37}$$

In the last section3.2.2, we gave a proof about these quantities. In particular, we estimate these solutions by relative error defined by

$$\Delta I \equiv \left| \frac{I(t) - I(0)}{I(0)} \right|, \tag{3.2.38}$$

where $I(t)$ is any conserved quantity. In the case of one lump solution, our numerical calculations show that numerical lump solutions satisfy the conservation low with relative error $\Delta I \leq 1.0 \times 10^{-2}$ and fitting lump solution does it with $\Delta I \leq 1.0 \times 10^{-2}$.

These results and their contents are shown in Figure3.8-Figure3.19 and Table3.2. In these figures, we shown time evolution of lump solutions (both numerical and fitting) in ZK equation

$$u_t + 2uu_x + \nabla^2 u_x = 0. \tag{3.2.39}$$

The time evolution of lump solutions are shown in "Left figure" respectively. On the other hand, in "Right figure", (A)Path of lump peak, (B)Variation of amplitude of lump and (C)Relative error of conserved quantities eq.(3.2.38) are shown. Moreover, relative error of conserved quantities M, P and E are expressed by solid line (magenta), broken line (green) and chain line (blue) respectively.

Table3.2: Contents of figures in section3.2.3

	Value of c	Left figure	Right figure
Numerical lump	$c = 1.0$	Figure3.8	Figure3.9
	$c = 2.0$	Figure3.10	Figure3.11
	$c = 4.0$	Figure3.12	Figure3.13
Fitting lump	$c = 1.0$	Figure3.14	Figure3.15
	$c = 2.0$	Figure3.16	Figure3.17
	$c = 4.0$	Figure3.18	Figure3.19

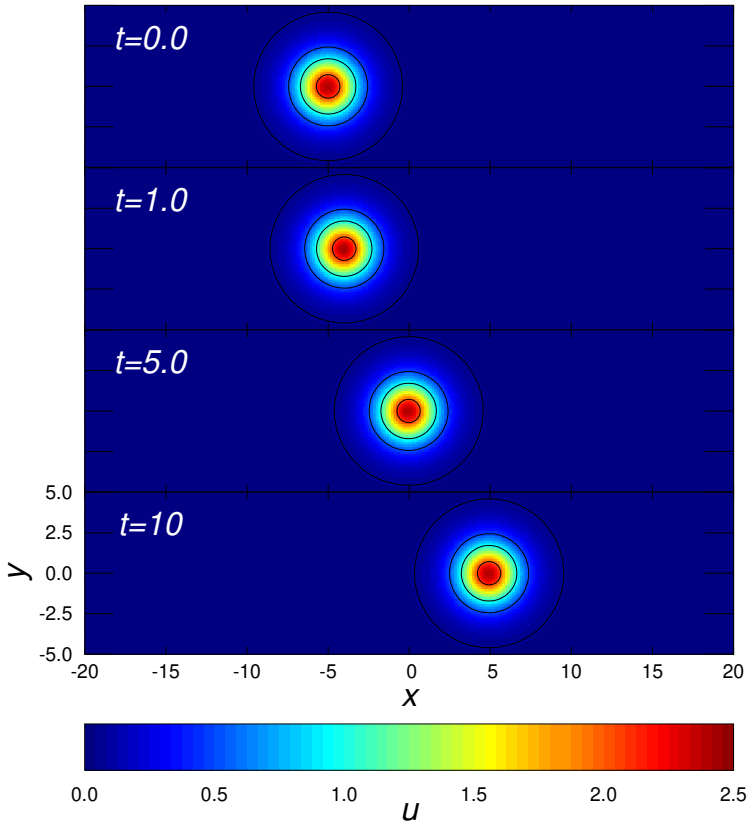


Figure3.8: Time evolution of numerical lump solution ($c = 1.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 1, 2$.

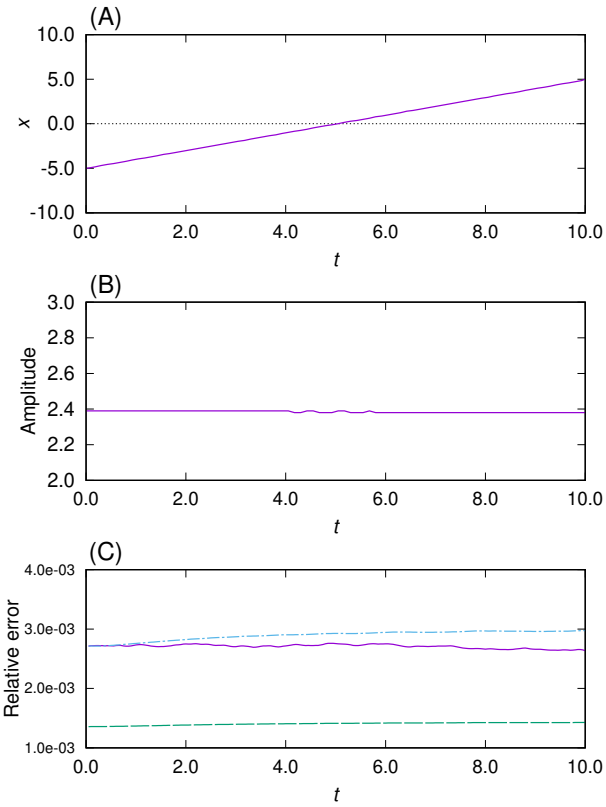


Figure3.9: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.8.

From figure3.8 and Figure3.9, we can find that numerical lump solution travels along x-direction with velocity $c = 1.0$. Further, its amplitude and conserved qunatities sustain themselves.

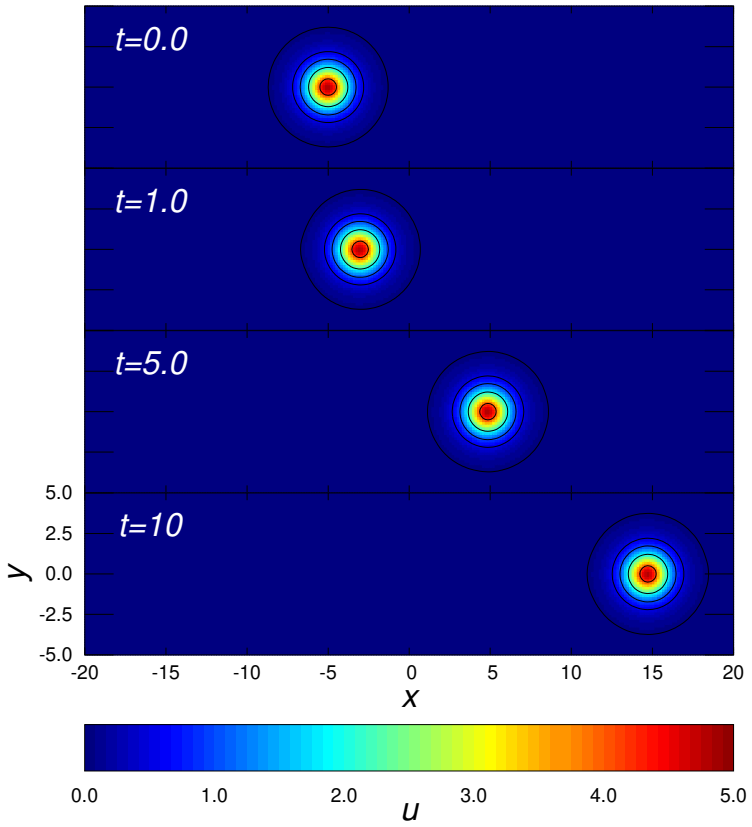


Figure3.10: Time evolution of numerical lump solution ($c = 2.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 1, 2, 4$.

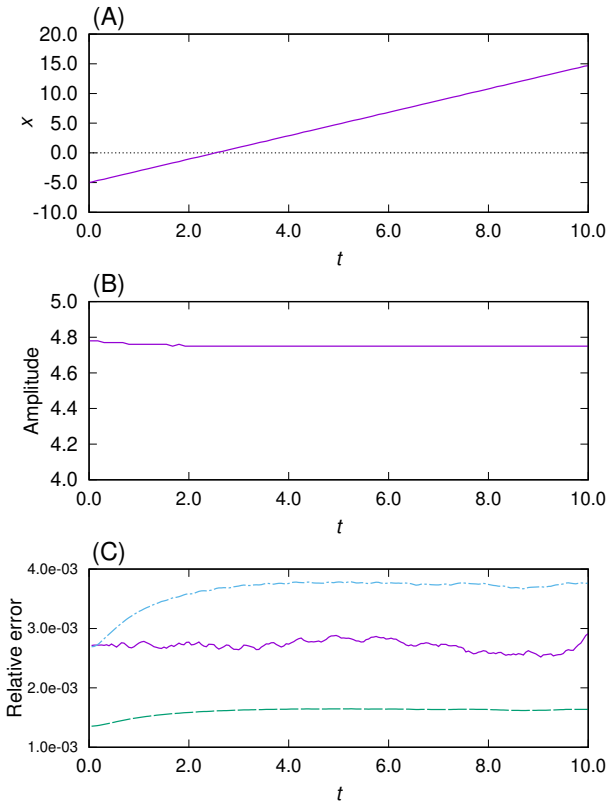


Figure3.11: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.10.

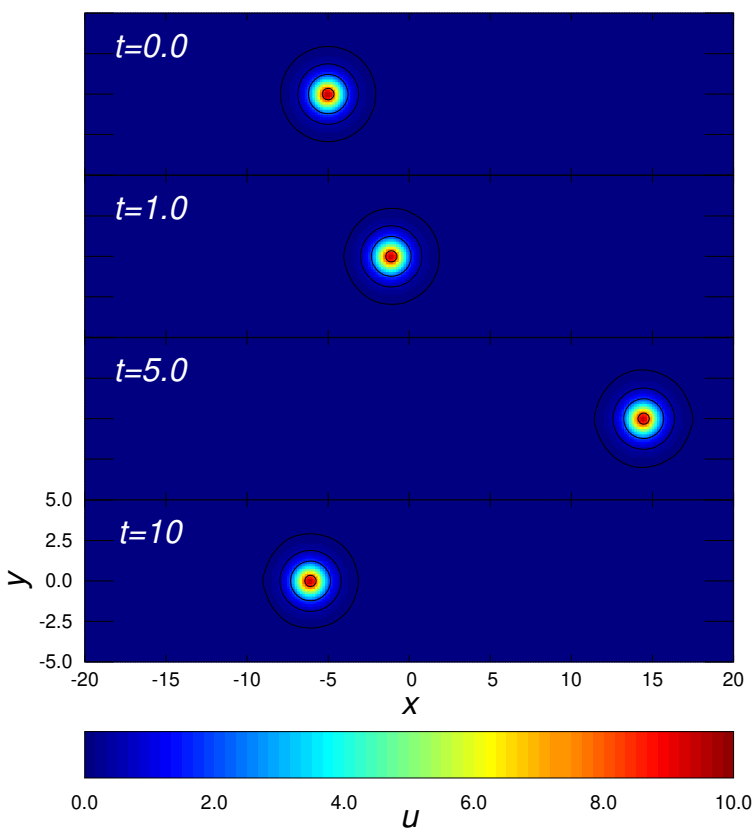


Figure3.12: Time evolution of numerical lump solution ($c = 4.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 2, 8$.

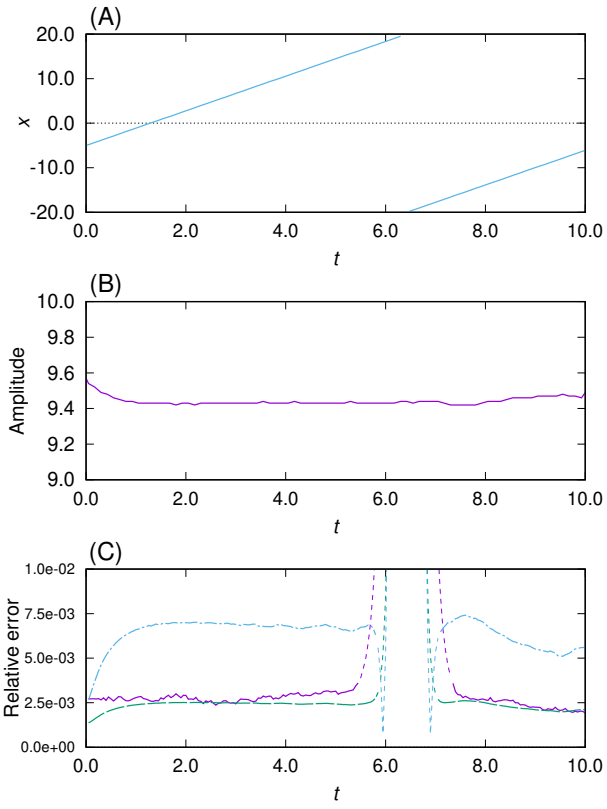


Figure3.13: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.12.

In Figure3.12 and Figure3.13, numerical lump solution in the case of $c = 4.0$ is shown. This solution sustains its form in time evolution. We can find that conserved quantities blow up at $t \approx 6.0$. It is because lump solution reaches to borders of domain. Conserved quantities depned on boundary values, so these behavior are caused. However, since it is temporary action, it doesn't matter for time evolution of lump solution.

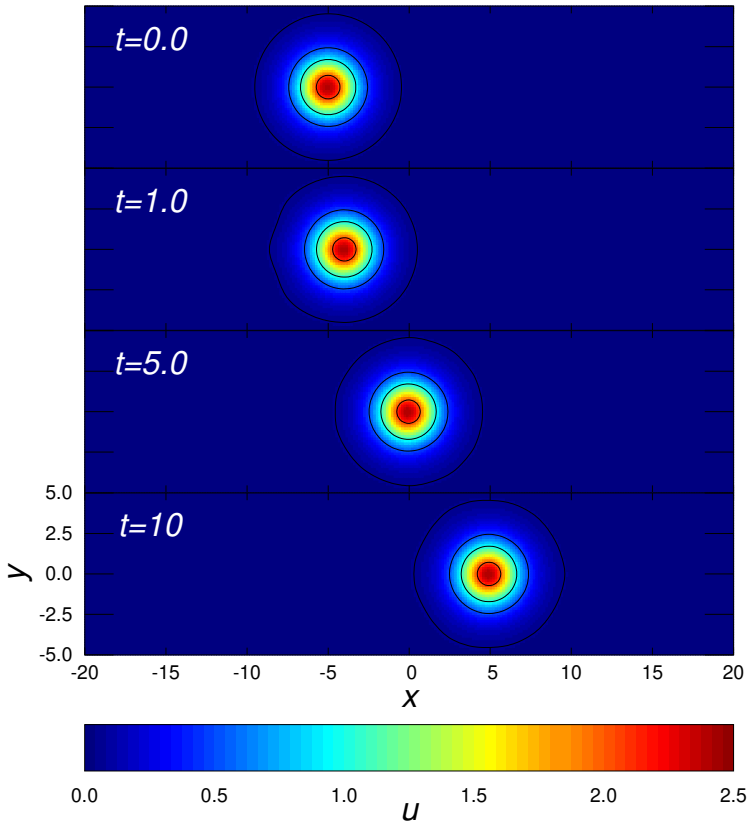


Figure3.14: Time evolution of fitting lump solution ($c = 1.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 1, 2$.

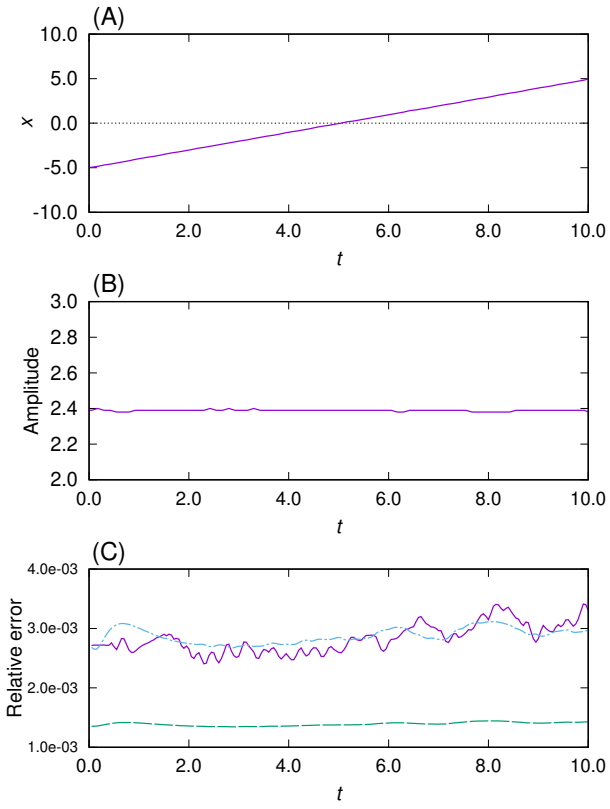


Figure3.15: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.14.

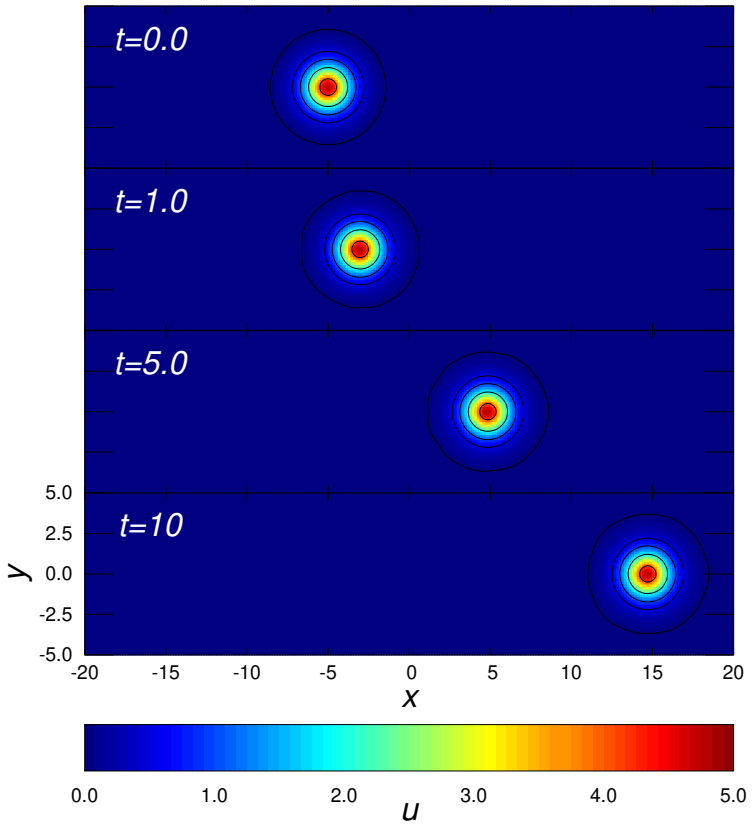


Figure3.16: Time evolution of fitting lump solution ($c = 2.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 1, 2, 4$.

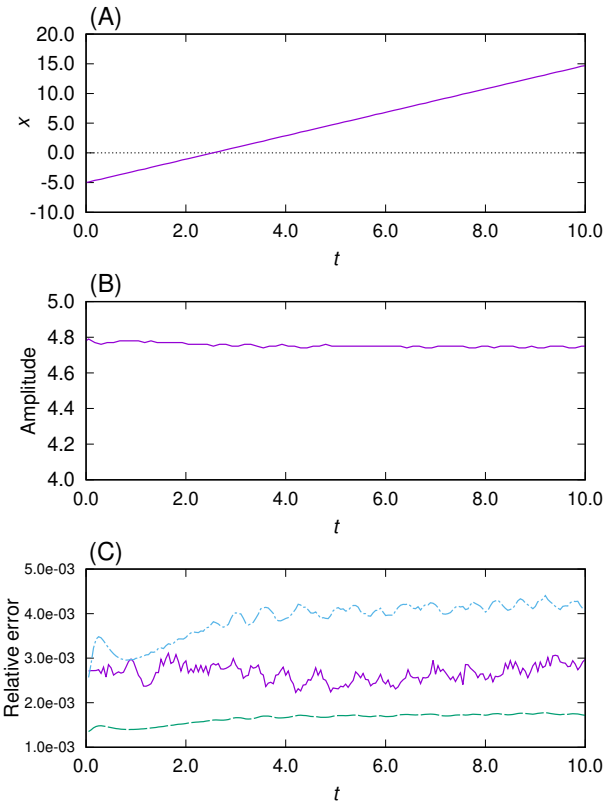


Figure3.17: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.16.

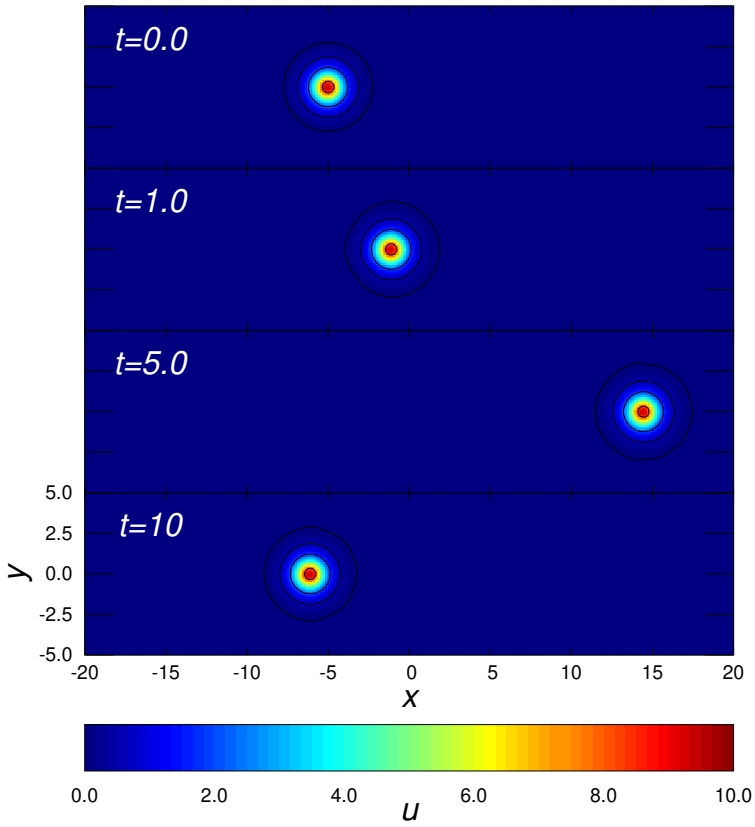


Figure3.18: Time evolution of fitting lump solution ($c = 4.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.05, 0.5, 2, 8$.

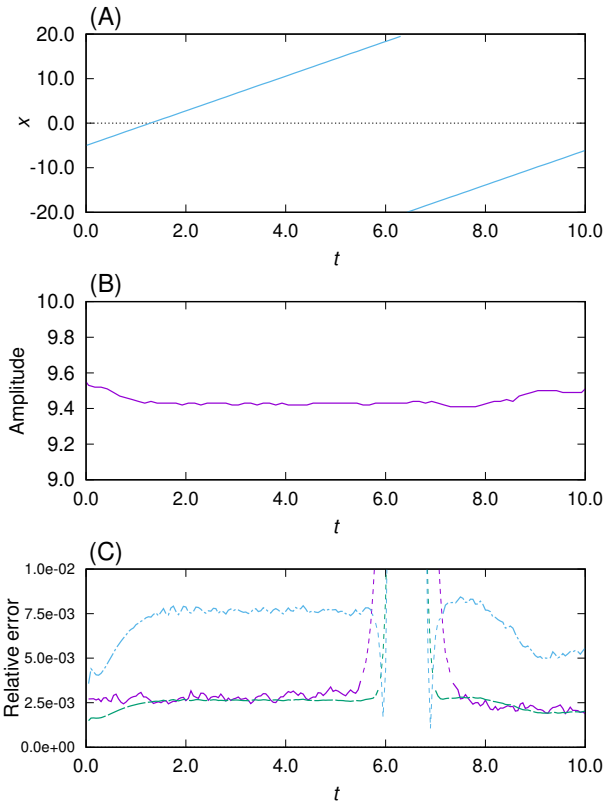


Figure3.19: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.18.

3.2.4 Behavior of two lump solution in ZK equation

In this section, we show the cases of two lump solution in ZK equation. We show these contents in table3.3 as follows

Table3.3: Contents of figures in section3.2.4

	Value of c	Left figure	Right figure
Numerical lump	$c = 4.0$ and $c = 4.4$	Figure3.20	Figure3.21
	$c = 1.0$ and $c = 4.0$	Figure3.22	Figure3.23
Fitting lump	$c = 4.0$ and $c = 4.4$	Figure3.24	Figure3.25
	$c = 1.0$ and $c = 4.0$	Figure3.26	Figure3.27

Left figures and right figures are the same as section3.2.3, i.e. "Left figure" is time evolution and "Right figure" is its profiles. When we choose two lump solution as initial condition of ZK equation, as we explained in section1.1.2, that behavior depends on amplitude of two lumps. In order to show that fact, we calculate four patterns above. At first, we show numerical results in the case of "similar" and "dissimlar" amplitude in Figure3.20-Figure3.23. They indicate the results of previous research [2]. Then, we show the results using fitting lump solution in Figure3.24-Figure3.27. We find that they behave like as numerical lumps. It shows that fitting lump solution is useful as substitute for numerical lump solution. Moreover, we verify its validity by relative error of conserved quantities. They support fitting lump solution. For the condition $c = 4.0$ and $c = 4.4$, numerical lumps have relative error $\Delta I \approx 1.0 \times 10^{-2}$ (Figure3.21) and fitting lumps have also its value as $\Delta I \approx 1.0 \times 10^{-2}$ (Figure3.25). For the condition $c = 1.0$ and $c = 4.4$, numerical lumps have relative error $\Delta I \leq 1.5 \times 10^{-2}$ (Figure3.23) and fitting lumps have also its value as $\Delta I \leq 1.5 \times 10^{-2}$ (Figure3.27).

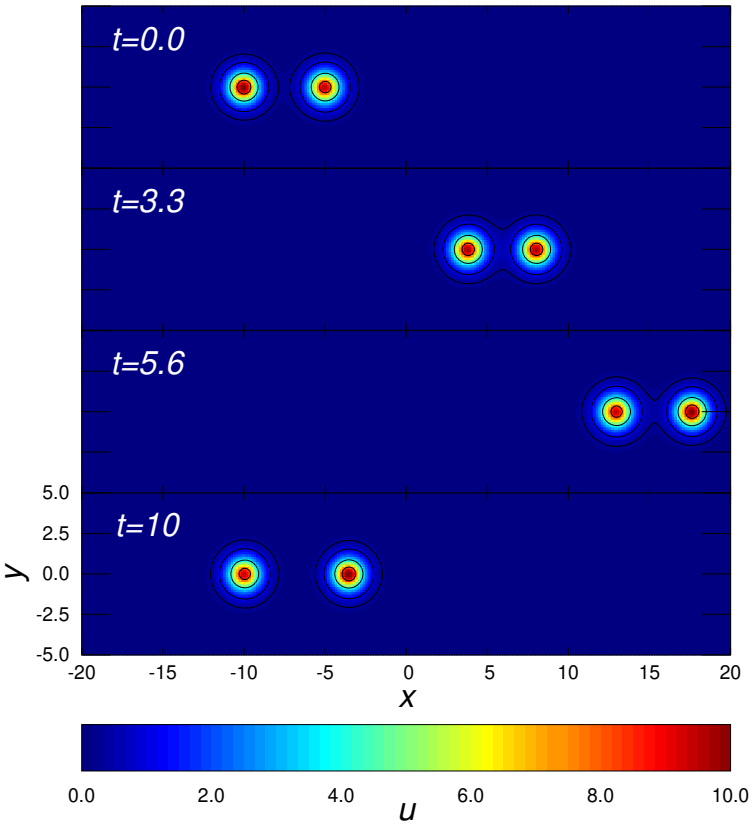


Figure3.20: Time evolution of numerical lump solution ($c = 4.0, 4.4$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.3, 1, 4, 8$.

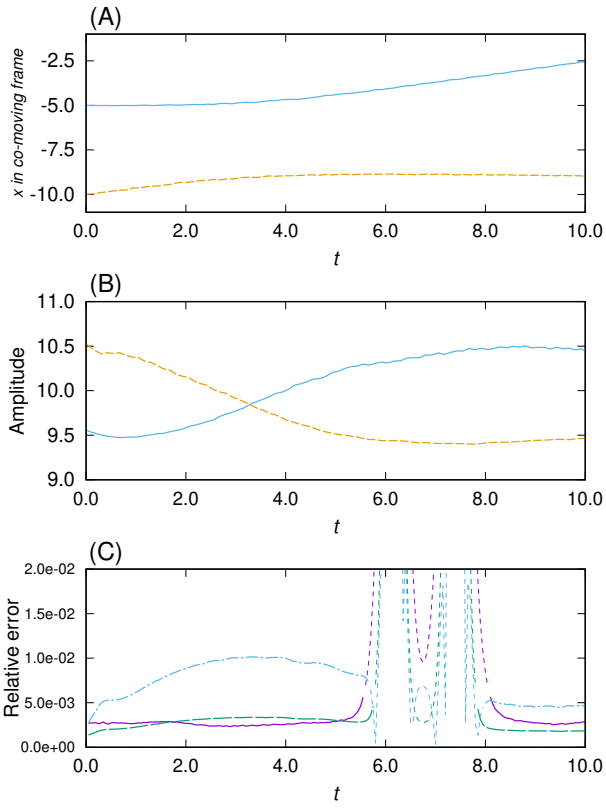


Figure3.21: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.20.

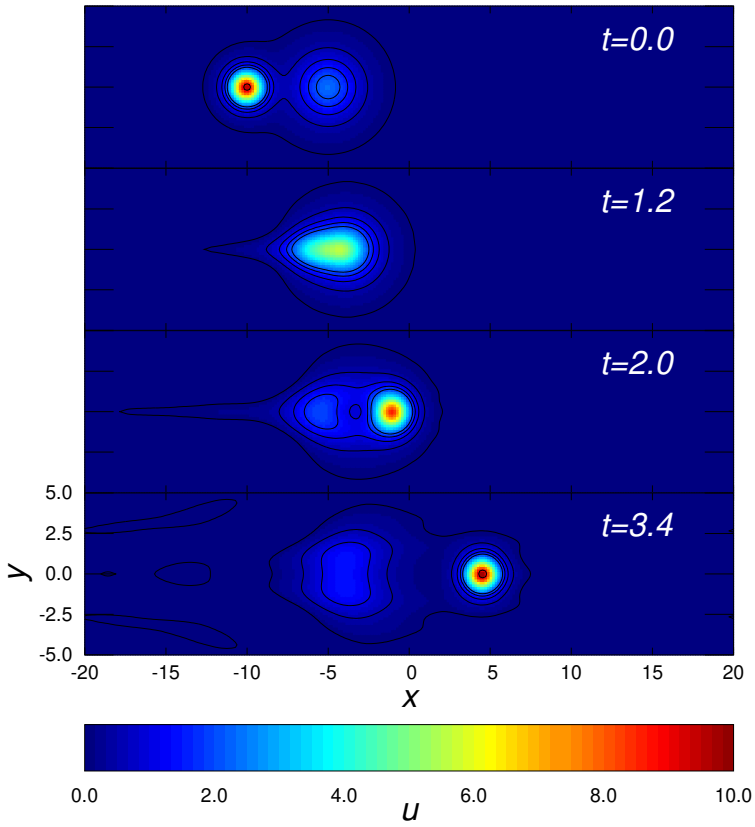


Figure3.22: Time evolution of numerical lump solution ($c = 1.0, 4.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

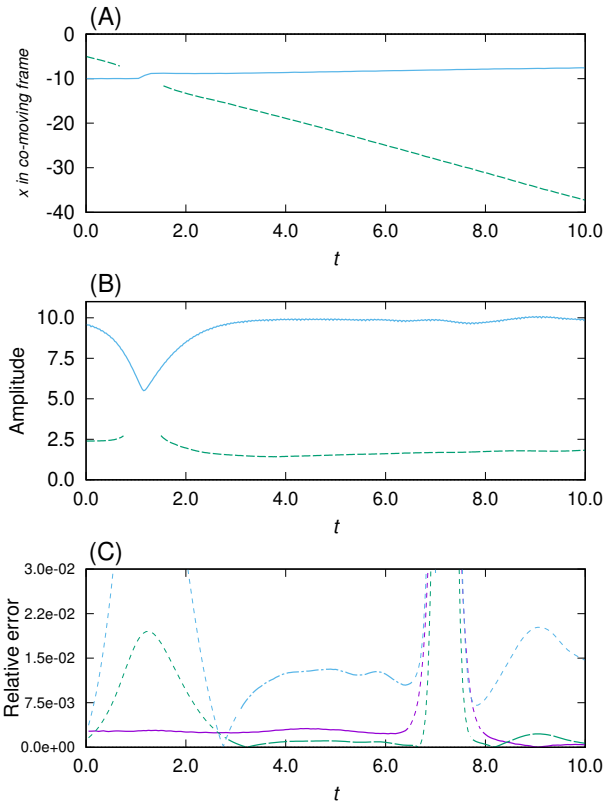


Figure3.23: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.22.

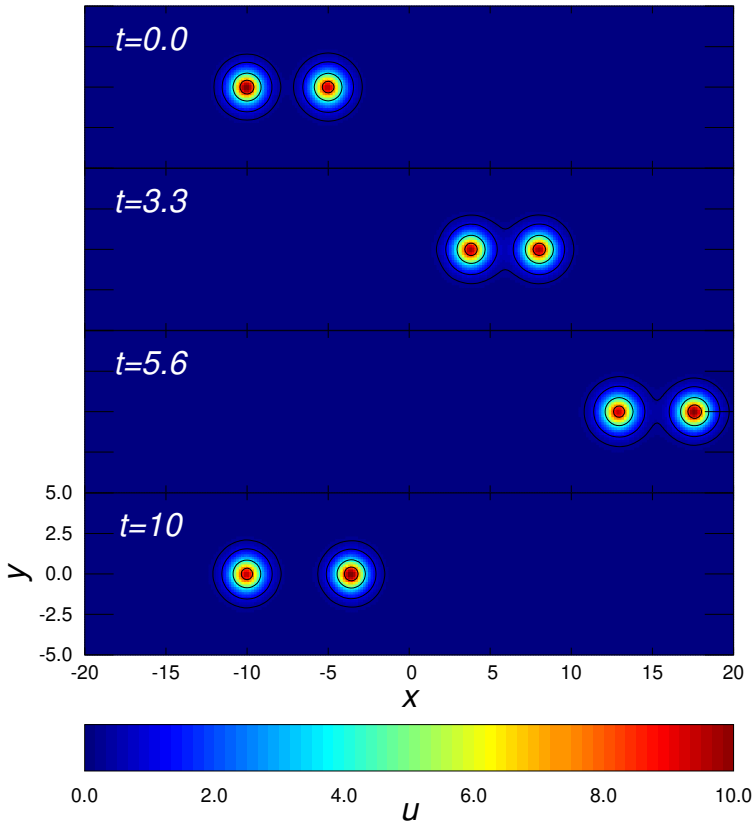


Figure3.24: Time evolution of fitting lump solution ($c = 4.0, 4.4$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.3, 1, 4, 8$.

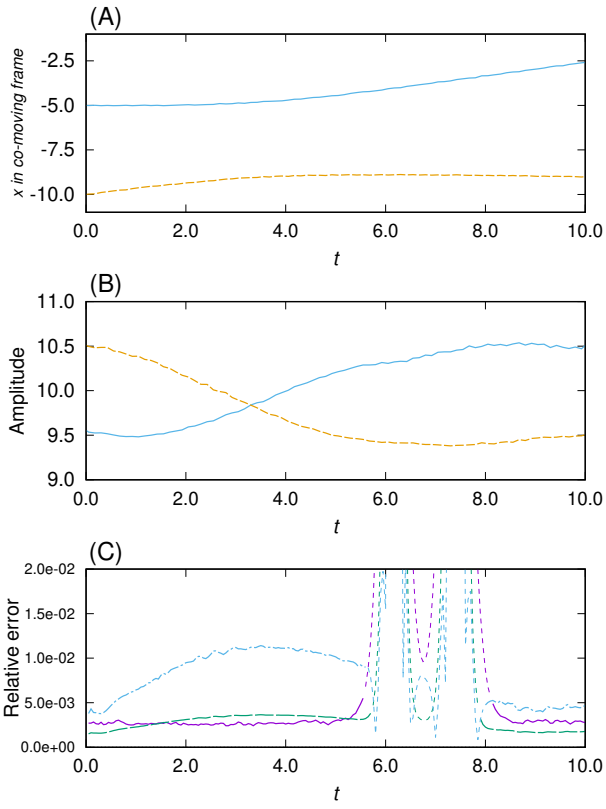


Figure3.25: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.24.

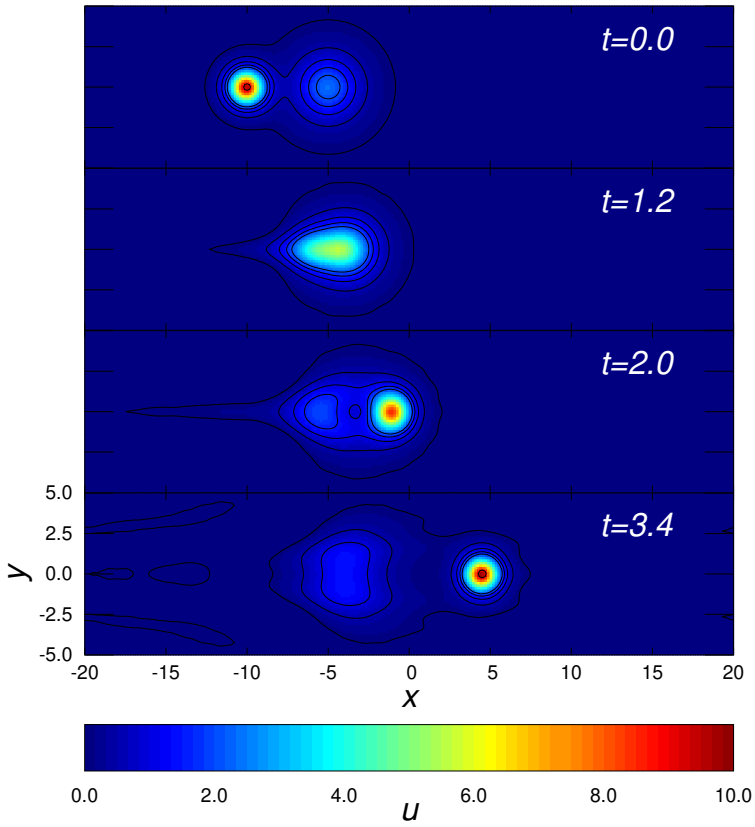


Figure3.26: Time evolution of fitting lump solution ($c = 1.0, 4.0$) for $t = 0.0$ - 10.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

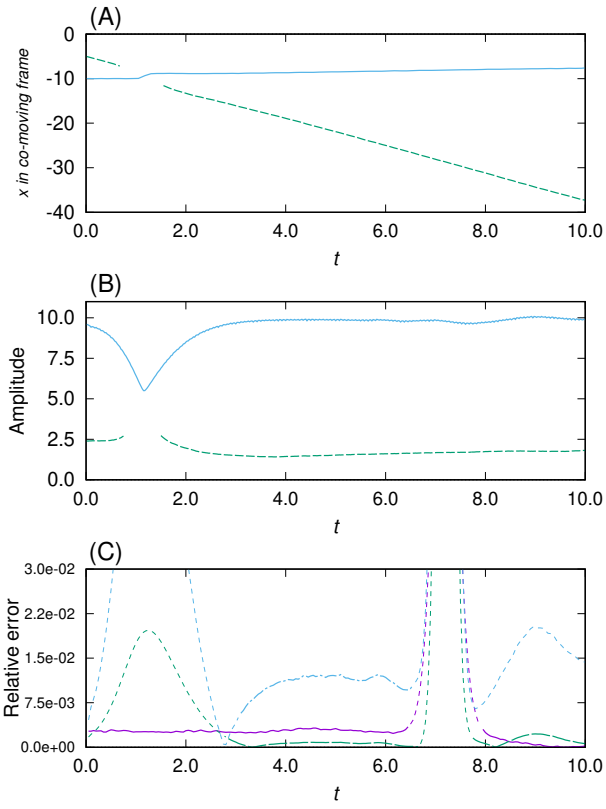


Figure3.27: (A)Path of lump peak, (B)Variation of amplitude of lump, (C)Relative error of conserved quantities in the case of Figure3.26.

3.2.5 Behavior of multi lump solution in ZK equation

In this section, we show the results of multi fitting lump solution. Concretely speaking, three of fitting lump ($c = 1.0$) and one of fitting lump solution ($c = 4.0$) are set as initial condition of ZK equation. According to the previous research [2], they exmained up to two of numerical solutions, however four lump solutions are not mentioned by them. We show that following.

In Figure3.28-Figure3.31, the time evolution of four fitting lumps are shown. Similarly to the case of two lumps, when their amplitude are dissimilar, they collide each other and a strong lump becomes stronger. We can see that process in these figures.

In Figure3.32, amplitude of each lump are shown. Solid line (blue) represents the fitting lump $c = 4.0$ and other broken lines (green, magenta and red) are fitting lump $c = 1.0$. For visibility, amplitude of lumps $c = 1.0$ are offset in Figure3.32. In Figure3.33, we show the paths of fitting lump solutions. In figure (A), solid (red) lines are path of $c = 1.0$ and broken (blue) line is path of $c = 4.0$ with co-moving frame $x = t$ which means with velocity $c = 1.0$. In figure (B), solid (blue) line is $c = 4.0$ and broken (red) lines are $c = 1.0$ respectively. Also, in this case, we show these results with co-moving frame $x = 3.9\,t$, i.e. we employ co-moving frame with velocity $c = 3.9$.

We find that if there are many lump solutions, the strong lump becomes stronger and weak lump becomes weaker through collision of lumps.

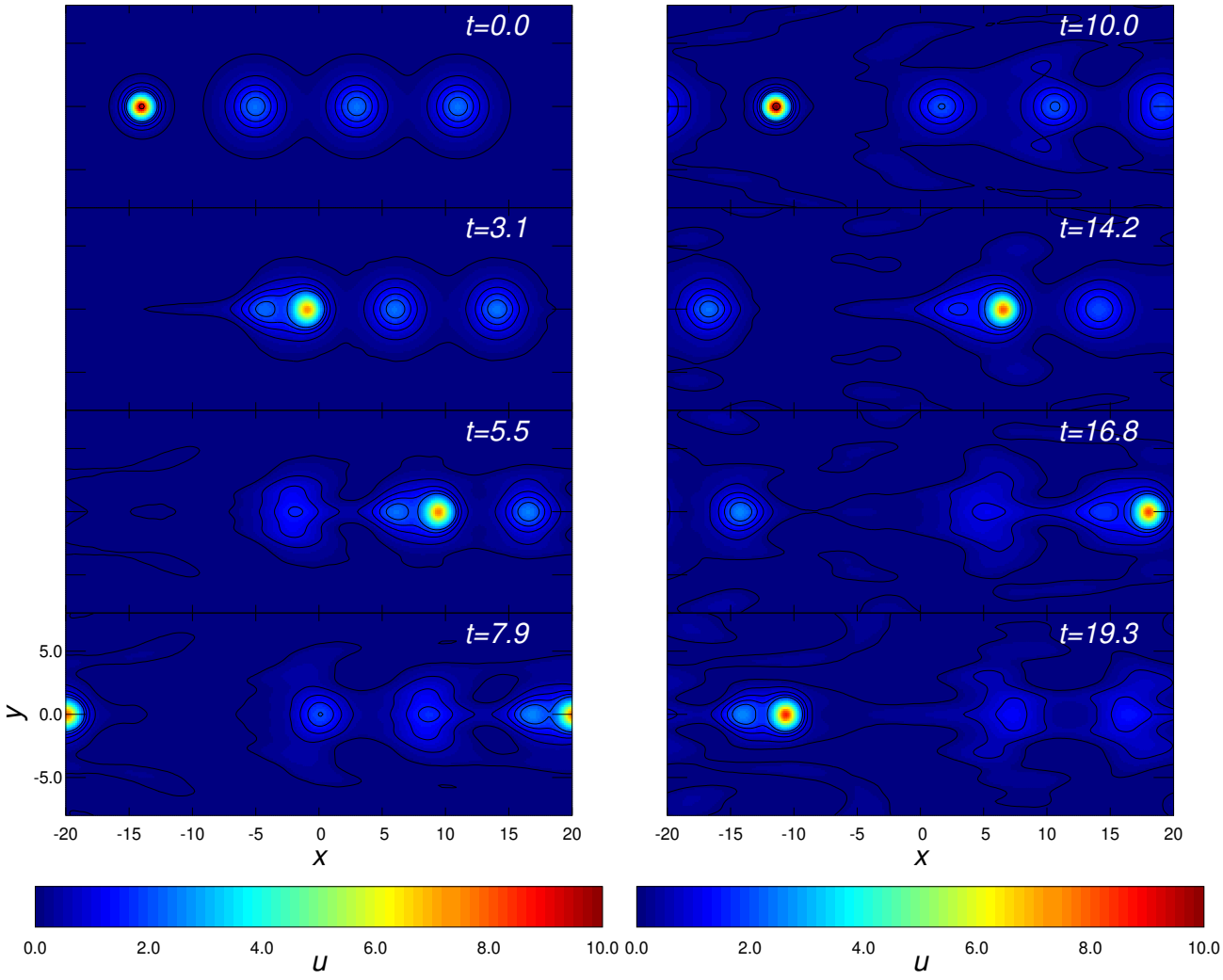


Figure3.28: During $t = 0.0$ - 10.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

Figure3.29: During $t = 10.0$ - 20.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

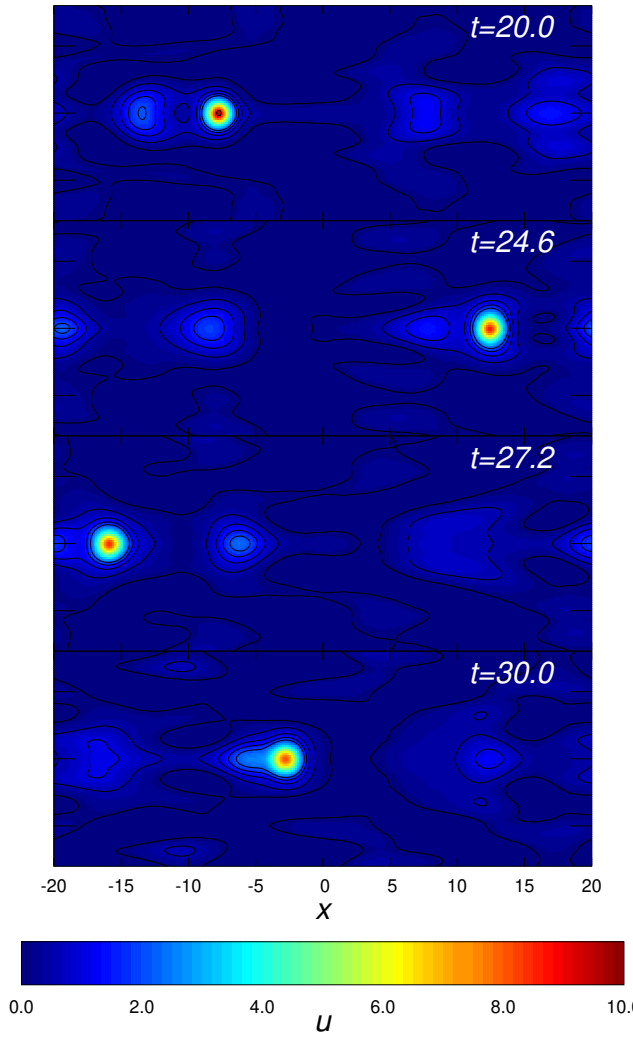


Figure3.30: During $t = 20.0$ - 30.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

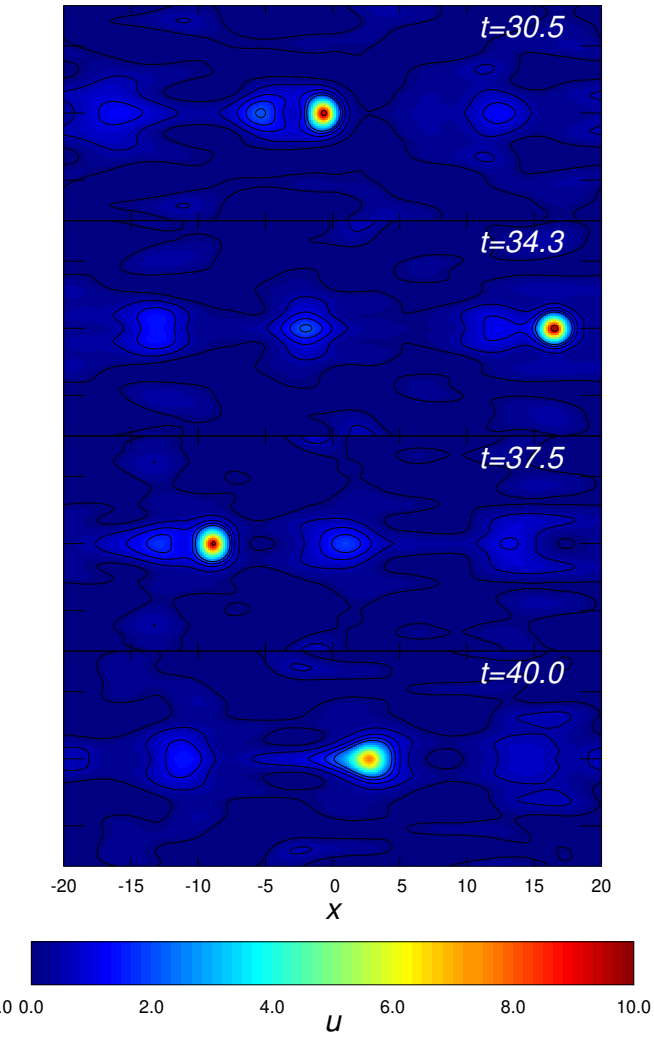


Figure3.31: During $t = 30.0$ - 40.0. Contour lines are drawn at $0.08, 0.5, 1, 1.5, 2, 9$.

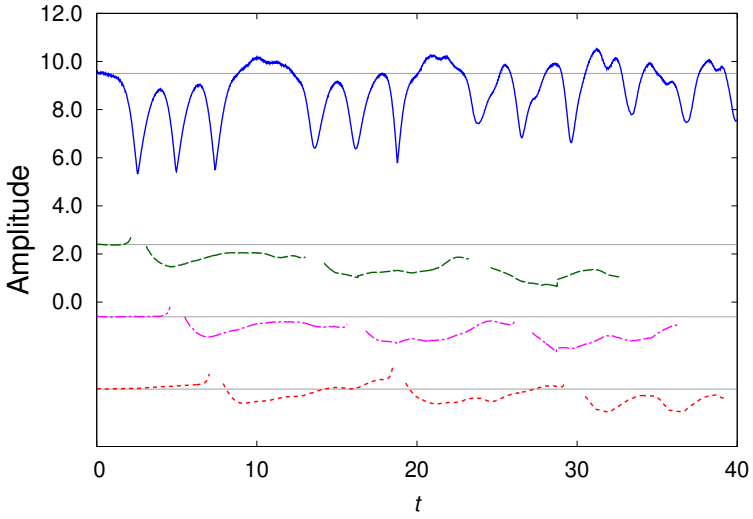


Figure3.32: Time evolution of amplitudes for $t =$ 0.0-40.0.

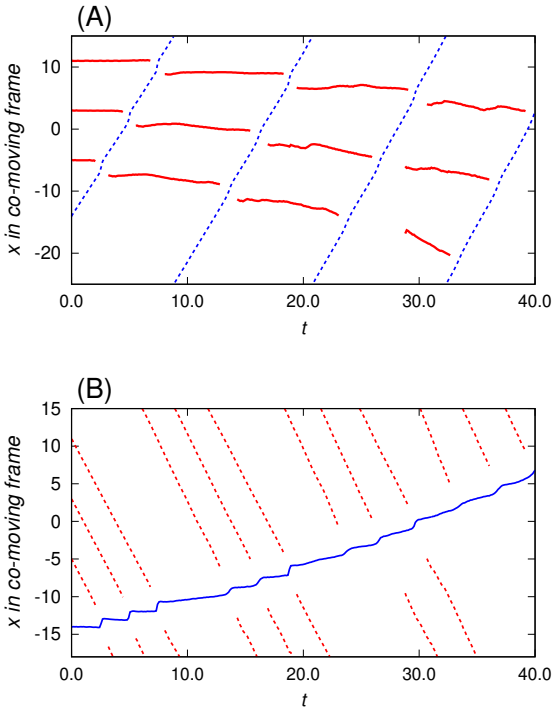


Figure3.33: Paths of fitting lumps ($c = 1.0, c =$ 4.0) for $t = 0.0$-40.0.

3.3 Numerical calculation

We use the code in Numerical Recipes [8]. We did operation checks for these funcitons, but you should check them again by yourself if you use. "Myheader.h", "Functions.cpp" and "Malloc-functions.cpp" are able to use in either IDE or command line. If you would like to use other functions including the functions we show in section1, you should add them to "Functions.cpp" and main file.

3.3.1 My header file

In order to use macros and "Malloc functions" which is defined in Malloc-functions.cpp, you have to compile this header file. Further, if you would like to use this header file, you have to include "Myheader.h" in main file and other file defining some functions.

Listing 3.1: My header file

```
/*include header files*/
/*if you need another header file, write here*/
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
/*definition of macros*/
#define SWAP(a,b) {tempr=(a);(a)=(b);(b)=tempr;}
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
#define PAI (3.14159265358979323846264338327950 29)
#define NR_END 1
#define TINY 1.0e-20;
#define TINY2 1.0e-30
#define FREE_ARG char*
#define PGROW -0.20
#define PSHRNK -0.25
#define FCOR 0.06666666666666666 /*1.0/15.0*/
#define SAFETY 0.9
#define ERRCON 6.0e-4 /*value of ERRCON is (4/SAFETY) to the power of (1/PGROW)*/
#define MAXSTEP 10000
#define NX 256
#define NY 128
#define NMAX 5000
#define GET_PSUM \
                    for (j=1;j<=ndim;j++) {\
                    for (sum=0.0,i=1;i<=mpts;i++) sum += p[i][j];\
                    psum[j]=sum;}
/*prototype declaration of Malloc-function.cpp*/
void nrerror(const char error_text[]);
double ***dtensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh);
void free_dtensor(double ***t, long nrl, long nrh, long ncl, long nch, long ndl,
    long ndh);
double **dmatrix(long nrl, long nrh, long ncl, long nch);
void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch);
```

```
34 double *dvector(unsigned long nl, unsigned long nh);
35 void free_dvector(double *v, unsigned long nl, unsigned long nh);
36 int ***itensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh);
37 void free_itensor(int ***t, long nrl, long nrh, long ncl, long nch, long ndl, long
       ndh);
38 int **imatrix(long nrl, long nrh, long ncl, long nch);
39 void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch);
40 int *ivector(unsigned long nl, unsigned long nh);
41 void free_ivector(int *v, unsigned long nl, unsigned long nh);
```

3.3.2 Malloc functions

In order to use dynamic memory allocation, you have to need this file "Malloc-functions.cpp". The dynamic memory allocation is useful when we want to use the array which has variable size. It is beause that the dynamic memory allocation can change the size of array while running code. For exmaple, when the program reads the data file which has unknown size and store them, the program should determine the size of array which is stored data in that program.

When you use these functions, you should declare pointer data type, e.g.$*$**data**. Further, you should allocate array which has variable size, e.g. $\mathbf{data} = \mathbf{dvector}(\mathbf{0}, \mathbf{n})$. This array behaves as double type vector which has size $\mathbf{n}$ $(= \mathbf{data}[\mathbf{0}, \ldots, \mathbf{n}-\mathbf{1}])$. At the end of program, you should release the memory by using command **free**, e.g. $\mathbf{free\_dvector}(\mathbf{data}, \mathbf{0}, \mathbf{n})$. We use these command in the code shown in section1 and section3. Therefore, you can use it easily.

Listing 3.2: The file of definition of Malloc functuons

```
1  #include "Myheader.h"
2
3  void nrerror(const char error_text[])
4  /* Numerical Recipes standard error handler */
5  {
6    fprintf(stderr, "Numerical␣Recipes␣run-time␣error...\n");
7    fprintf(stderr, "%s\n",error_text);
8    fprintf(stderr, "...now␣exiting␣to␣system...\n");
9    exit(1);
10 }
11
12 double ***dtensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
13        {
14        long i, j;
15        unsigned long nrow = nrh - nrl + 1, ncol = nch - ncl + 1, ndep = ndh - ndl
              + 1;
16        double ***t;
17
18        t = (double ***)calloc(nrow + 1, sizeof(double**));
19        if(NULL == t)
20                {
21                printf("\7******␣No␣memories␣1␣in␣dtensor******\n");
22                exit(0);
23                }
24        t += 1;
```

```
25          t -= nrl;
26
27          t[nrl] = (double **)calloc(nrow * ncol + 1, sizeof(double*));
28          if(NULL == t[nrl])
29                  {
30                  printf("\7****** No memories 2 in dtensor******\n");
31                  exit(0);
32                  }
33          t[nrl] += 1;
34          t[nrl] -= ncl;
35
36          t[nrl][ncl] = (double *)calloc(nrow * ncol * ndep + 1, sizeof(double));
37          if(NULL == t[nrl][ncl])
38                  {
39                  printf("\7****** No memories 3 in dtensor******\n");
40                  exit(0);
41                  }
42          t[nrl][ncl] += 1;
43          t[nrl][ncl] -= ndl;
44
45          for(j = ncl + 1 ; j <= nch ; j++)
46                  {
47                  t[nrl][j] = t[nrl][j - 1] + ndep;
48                  }
49          for(i = nrl + 1 ; i <= nrh ; i++)
50                  {
51                  t[i] = t[i - 1] + ncol;
52                  t[i][ncl] = t[i - 1][ncl] + ncol * ndep;
53                  for(j = ncl + 1 ; j <= nch ; ++j)
54                          {
55                          t[i][j] = t[i][j -1] + ndep;
56                          }
57                  }
58
59          return t;
60          }
61
62  #define FREE_ARG char*
63
64  void free_dtensor(double ***t, long nrl, long nrh, long ncl, long nch, long ndl,
        long ndh)
65          {
66          free((FREE_ARG) (t[nrl][ncl] + ndl - 1));
67          free((FREE_ARG) (t[nrl] + ncl - 1));
68          free((FREE_ARG) (t + nrl - 1));
69          }
70
71
72  double **dmatrix(long nrl, long nrh, long ncl, long nch)
73          {
74          long i;
75          unsigned long nrow = nrh - nrl + 1, ncol = nch - ncl + 1;
76          double **m;
77
78          m = (double **)calloc(nrow + 1, sizeof(double*));
```

```
79          if(NULL == m)
80                  {
81                  printf("\7****** No memories 1 in dhmatrix ******\n");
82                  exit(0);
83                  }
84          m += 1;
85          m -= nrl;
86
87          m[nrl] = (double *)calloc(nrow * ncol + 1, sizeof(double));
88          if(NULL == m[nrl])
89                  {
90                  printf("\7****** No memories 2 in dhmatrix ******\n");
91                  exit(0);
92                  }
93          m[nrl] += 1;
94          m[nrl] -= ncl;
95
96          for(i = nrl + 1 ; i <= nrh ; ++i)
97                  {
98                  m[i] = m[i - 1] + ncol;
99                  }
100
101         return m;
102         }
103
104 #define FREE_ARG char*
105
106 void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
107         {
108         free((FREE_ARG) (m[nrl] + ncl - 1));
109         free((FREE_ARG) (m + nrl - 1));
110         }
111
112
113 double *dvector(unsigned long nl, unsigned long nh)
114         {
115         double *v;
116
117         v = (double *)calloc(nh - nl + 2, sizeof(double));
118         if(NULL == v)
119                 {
120                 printf("\7****** No memories in dvector ******\n");
121                 exit(0);
122                 }
123         return v - nl + 1;
124         }
125
126 #define FREE_ARG char*
127
128 void free_dvector(double *v, unsigned long nl, unsigned long nh)
129         {
130         free((FREE_ARG) (v + nl - 1));
131         }
132 int ***itensor(long nrl, long nrh, long ncl, long nch, long ndl, long ndh)
133         {
```

```
134         long i, j;
135         unsigned long nrow = nrh - nrl + 1, ncol = nch - ncl + 1, ndep = ndh - ndl
                + 1;
136         int ***t;
137 
138         t = (int ***)calloc(nrow + 1, sizeof(int**));
139         if(NULL == t)
140                 {
141                 printf("\7****** No memories 1 in dtensor******\n");
142                 exit(0);
143                 }
144         t += 1;
145         t -= nrl;
146 
147         t[nrl] = (int **)calloc(nrow * ncol + 1, sizeof(int*));
148         if(NULL == t[nrl])
149                 {
150                 printf("\7****** No memories 2 in dtensor******\n");
151                 exit(0);
152                 }
153         t[nrl] += 1;
154         t[nrl] -= ncl;
155 
156         t[nrl][ncl] = (int *)calloc(nrow * ncol * ndep + 1, sizeof(int));
157         if(NULL == t[nrl][ncl])
158                 {
159                 printf("\7****** No memories 3 in dtensor******\n");
160                 exit(0);
161                 }
162         t[nrl][ncl] += 1;
163         t[nrl][ncl] -= ndl;
164 
165         for(j = ncl + 1 ; j <= nch ; j++)
166                 {
167                 t[nrl][j] = t[nrl][j - 1] + ndep;
168                 }
169         for(i = nrl + 1 ; i <= nrh ; i++)
170                 {
171                 t[i] = t[i - 1] + ncol;
172                 t[i][ncl] = t[i - 1][ncl] + ncol * ndep;
173                 for(j = ncl + 1 ; j <= nch ; ++j)
174                         {
175             t[i][j] = t[i][j -1] + ndep;
176                         }
177                 }
178 
179         return t;
180         }
181 
182 #define FREE_ARG char*
183 
184 void free_itensor(int ***t, long nrl, long nrh, long ncl, long nch, long ndl, long
        ndh)
185         {
186         free((FREE_ARG) (t[nrl][ncl] + ndl - 1));
```

```
187         free((FREE_ARG) (t[nrl] + ncl - 1));
188         free((FREE_ARG) (t + nrl - 1));
189         }
190
191
192 int **imatrix(long nrl, long nrh, long ncl, long nch)
193         {
194         long i;
195         unsigned long nrow = nrh - nrl + 1, ncol = nch - ncl + 1;
196         int **m;
197
198         m = (int **)calloc(nrow + 1, sizeof(int*));
199         if(NULL == m)
200                 {
201                 printf("\7******␣No␣memories␣1␣in␣dhmatrix␣******\n");
202                 exit(0);
203                 }
204         m += 1;
205         m -= nrl;
206
207         m[nrl] = (int *)calloc(nrow * ncol + 1, sizeof(int));
208         if(NULL == m[nrl])
209                 {
210                 printf("\7******␣No␣memories␣2␣in␣dhmatrix␣******\n");
211                 exit(0);
212                 }
213         m[nrl] += 1;
214         m[nrl] -= ncl;
215
216         for(i = nrl + 1 ; i <= nrh ; ++i)
217                 {
218                 m[i] = m[i - 1] + ncol;
219                 }
220
221         return m;
222         }
223
224 #define FREE_ARG char*
225
226 void free_imatrix(int **m, long nrl, long nrh, long ncl, long nch)
227         {
228         free((FREE_ARG) (m[nrl] + ncl - 1));
229         free((FREE_ARG) (m + nrl - 1));
230         }
231
232
233 int *ivector(unsigned long nl, unsigned long nh)
234         {
235         int *v;
236
237         v = (int *)calloc(nh - nl + 2, sizeof(int));
238         if(NULL == v)
239                 {
240                 printf("\7******␣No␣memories␣in␣dvector␣******\n");
241                 exit(0);
```

```
242                 }
243         return v - nl + 1;
244         }
245
246 #define FREE_ARG char*
247
248 void free_ivector(int *v, unsigned long nl, unsigned long nh)
249         {
250         free((FREE_ARG) (v + nl - 1));
251     }
```

3.3.3 The definition of other functions [8]

In this section, the definition of functions which we couldn't explain in before section are shown. In order to use these functions, you have to include header file "Myheader.h"[*2]. We have already shown that file in the last section, so you can check it. If you learn more details of these functions, please refer to the Numerical Recipes in C [8].

Gauss Jordan elimination

This algorithm expect to solve linear matrix equation as follows[*3]

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \left[\begin{pmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \end{pmatrix} \begin{pmatrix} x_{12} \\ x_{22} \\ x_{32} \\ x_{42} \end{pmatrix} \begin{pmatrix} x_{13} \\ x_{23} \\ x_{33} \\ x_{43} \end{pmatrix} \right] = \left[\begin{pmatrix} b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{pmatrix} \begin{pmatrix} b_{12} \\ b_{22} \\ b_{32} \\ b_{42} \end{pmatrix} \begin{pmatrix} b_{13} \\ b_{23} \\ b_{33} \\ b_{43} \end{pmatrix} \right] . \quad (3.3.1)$$

For this function, **a[1...n][1...n]** is $n \times n$ matrix, **b[1...n][1...m]** is $n \times m$ matrix. They are coefficient matrix and m vectors in the r.h.s respectively. If we input these information, **a** is rewritten by its inverse matrix and **b** is done by solution vectors. **n,m** are dimension of **a** and **m**.

This algorithm finds the pivot then, if necessary, exchange row and set pivot to diagonal line. Columns are not exchange but replace its label. That is **indxc[i]** (i pivot) is i culumn with which deal and **indxr[i]** is initial place of that pivot. If **indxc[i]** is not equal to **indxr[i]**, that means there was exchanging columns. Consequently, solution vector **b** is in order correct and inverse matrix becomes not in order.

[*2] This header file contains some important definition of macros. Since this dependence, we recommend you to include it.

[*3] For simplicity, we write down explicitly in the case of four dimention. Algorithm can deal with n dimension.

Listing 3.3: Gauss Jordan elimination

```
#include "Myheader.h"
void gaussj(double **a, int n, double **b, int m)
{
    int *indxc, *indxr, *ipiv;
/* Array of integer type : inpiv[1...n],indxc[1...n],indxr[1...n] are used
    to record the pivot choices */
    int i,icol,irow,j,k,l,ll;
    double big,dum,pivinv,tempr;
    indxc=ivector(1,n);
    indxr=ivector(1,n);
    ipiv=ivector(1,n);
    for (j=1;j<=n;j++) ipiv[j]=0;
    for (i = 1; i <= n; i++) { //main roop
        big=0.0;
        for (j = 1; j <= n; j++)
             if (ipiv[j] != 1)
                 for (k = 1; k <= n; k++) {
                      if (ipiv[k] == 0) {
                          if (fabs(a[j][k]) >= big){
                              big=fabs(a[j][k]);
                              irow=j;
                              icol=k;
                          }
                      } else if (ipiv[k] > 1) nrerror("gaussj: Singular Matrix-1");
                 }
        ++(ipiv[icol]);
        if (irow != icol){
            for (l=1;l<=n;l++) SWAP(a[irow][l],a[icol][l])
            for (l=1;l<=m;l++) SWAP(b[irow][l],b[icol][l])
        }
        indxr[i]=irow; /*devide pivot row by pivot element*/
        indxc[i]=icol;
        if (a[icol][icol] == 0.0) nrerror("gaussj: Singular Matrix-2");
        pivinv=1.0/a[icol][icol];
        a[icol][icol]=1.0;
        for (l=1;l<=n;l++) a[icol][l] *= pivinv;
        for (l=1;l<=m;l++) b[icol][l] *= pivinv;
        for (ll=1;ll<=n;ll++)
            if (ll != icol) {
                dum=a[ll][icol];
                a[ll][icol]=0.0;
                for (l=1;l<=n;l++) a[ll][l] -= a[icol][l]*dum;
                for (l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;
            }
    }
    for (l=n;l>=1;l--) {
        if (indxr[l] != indxc[l])
            for (k=1;k<=n;k++)
                SWAP(a[k][indxr[l]],a[k][indxc[l]]);
    }
    free_ivector(ipiv,1,n);
    free_ivector(indxr,1,n);
    free_ivector(indxc,1,n);
}
```

LU decomposition

In general, one can rewrite matrix $\boldsymbol{A}$ to the form of products of two matrices as follows

$$\boldsymbol{L} \cdot \boldsymbol{U} = \boldsymbol{A}, \tag{3.3.2}$$

where $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{U}$ is a upper triangular matrix. For exmaple, in four dimension case, this relation is written as

$$\begin{pmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{pmatrix} \cdot \begin{pmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}. \tag{3.3.3}$$

By using this LU decomposition, we can easily solve the linear equation like

$$\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}. \tag{3.3.4}$$

First step, we rewrite eq.(3.3.4) to the form

$$\boldsymbol{A} \cdot \boldsymbol{x} = (\boldsymbol{L} \cdot \boldsymbol{U}) \cdot \boldsymbol{x} = \boldsymbol{L} \cdot (\boldsymbol{U} \cdot \boldsymbol{x}) = \boldsymbol{b}, \tag{3.3.5}$$

where we use eq.(3.3.2). Then, we solve

$$\boldsymbol{L} \cdot \boldsymbol{y} = \boldsymbol{b} \tag{3.3.6}$$

and obtain $\boldsymbol{y}$. Second step, since $\boldsymbol{y}$ is equal to $(\boldsymbol{U} \cdot \boldsymbol{x})$, we solve the equation

$$\boldsymbol{U} \cdot \boldsymbol{x} = \boldsymbol{y}, \tag{3.3.7}$$

and obtain $\boldsymbol{x}$. This is the just solution vector we desired. This process to obtain vector $\boldsymbol{x}$ is carried out in function **lubksb** as follows. In order to complete these scheme, we need the LU decomposition of matrix **A**. This algorithm is shown in function **ludcmp** as follows. The details of LU decomposition, please refer to Numerical recipes in C [8][*4].

Next, we explain how to use following function. If we input $n \times n$ matrix **a[1...n][1 ...n]**, function **ludcmp** rewrites **a** by its LU decomposition. This function needs matrix **a** and dimension of matrix **n** which is integer. Output vector **indx[1..n]** is used to record exchanging row. The value of **d** is either 1 or -1, e.g. if number of exchanging row is even, $d = 1$ and if it is odd, $d = -1$. When we calculate determinant, we use the value of **d**. This routine is used with **lubksb** to solve linear-equations and find inverse matrix.

[*4] We employ *Crout's algorithm* to carry out LU decomposition. It is not so difficult.

Listing 3.4: LU decomposition

```
void ludcmp(double **a, int n, int *indx, double *d)
{
  int i,imax,j,k;
  float big,dum,sum,temp;
  double *vv;

  vv=dvector(1,n);
  *d=1.0;
  for (i = 1; i <= n; i++) {
      big=0.0;
      for (j = 1; j <= n; j++)
          if ((temp=fabs(a[i][j]))>big) big=temp;
      if (big==0.0) nrerror("Singular matrix in routine ludcmp");
      //If maximum of compornents is 0, this matrix is singular one.
      vv[i]=1.0/big;
  }
  for (j = 1; j <= n; j++) {
      for (i = 1; i < j; i++) {
          sum=a[i][j];
          for (k = 1; k < i; k++) sum -= a[i][k]*a[k][j];
          a[i][j]=sum;
      }
      big=0.0;
      for (i = j; i <= n; i++) {
          sum=a[i][j];
          for (k = 1; k < j; k++)
              sum -= a[i][k]*a[k][j];
          a[i][j]=sum;
          if ((dum=vv[i]+fabs(sum)) >= big) {
            big=dum;
            imax=i;
          }
      }
      if (j != imax) {
          for (k = 1; k <= n; k++) {
            dum=a[imax][k];
            a[imax][k]=a[j][k];
            a[j][k]=dum;
          }
          *d = -(*d);
          vv[imax]=vv[j];
      }
      indx[j]=imax;
      if (a[j][i] == 0.0) a[i][j]=TINY;
      /*
      If pivot is 0, the matrix is singular.
      */
      if (j != n) {
          dum=1.0/(a[j][j]);
          for (i = j+1; i <= n; i++) a[i][j] *= dum;
      }
  }
  free_dvector(vv,1,n);
}
```

Solver of linear equation using backward substituting

This function solves the equation eq.(3.3.4). The explanation of its algorithm is shown above. Function **lubksb** needs matrix **a**[**1**...**n**], its dimension **n**, integral vector **indx**[**1**...**n**] and vector **b**[**1**...**n**] where matrix **a** and double vector **indx** are LU decomposition and its record derived from **ludcmp**. Input vector **b** is the r.h.s. of eq.(3.3.4). Output is solution vector $\boldsymbol{x}$ which is put in vector **b**.

The function **lubksb** can solve the equation like eq.(3.3.4) with using **ludcmp**. We show the example code in lines from 21 to 25.

Listing 3.5: Solver of linear equation using backward substituting

```
1  void lubksb(double **a, int n, int *indx, double b[])
2  {
3      int i,ii=0,ip,j;
4      double sum;
5
6      for (i = 1; i <= n; i++) {
7          ip=indx[i];
8          sum=b[ip];
9          b[ip]=b[i];
10         if (ii)
11             for (j = ii; j <= i-1; j++) sum -= a[i][j]*b[j];
12         else if (sum) ii=i;
13         b[i]=sum;
14     }
15     for (i = n; i >= 1; i--) {
16         sum=b[i];
17         for (j = i+1; j <= n; j++) sum -= a[i][j]*b[j];
18         b[i]=sum/(a[i][i]);
19     }
20 }
21 //Example code to solve Ax=b
22 {
23   ludcmp(a,n,indx,&d);
24   lubksb(a,n,indx,b);
25 }
```

Runge-Kutta stepper and driver

When n variables **y[1...n]** and its derivatives **dydx[1...n]** are given at x, this routine **rk4** puts forward time interval h and substitutes new variables into **yout[1...n]** by using 4th order Runge-Kutta method. We should define the routine **derivs(x,y,dydx)** which calculates derivatives on x and puts into **dydx**. **derivs** calculates F on $dy/dx = F$, so we have to define by ourself.

For example, we show the harmonic oscillator given by

$$m\frac{d^2x}{dt^2} = -kx. \tag{3.3.8}$$

Since this is second order ordinary differential equation, we can rewrite to the form of coupled equations like

$$\frac{dx}{dt} = v, \tag{3.3.9}$$

$$\frac{dv}{dt} = -\frac{k}{m}x. \tag{3.3.10}$$

From these forms, we can regard eq.(3.1.53) and eq.(3.3.10) as

$$x \to \mathrm{y[1]}, \qquad v \to \mathrm{y[2]}, \tag{3.3.11}$$

$$v \to \mathrm{dydx[1]}, \qquad -\frac{k}{m}x \to \mathrm{dydx[2]}. \tag{3.3.12}$$

Therefore, we can define the function **derivs** as lines from 29 to 34. Similarly, we can define **derivs** in other cases.

Listing 3.6: Runge-Kutta stepper 1

```
void rk4(double y[], double dydx[], int n, double x, double h, double yout[],
    void (*derivs)(double, double [], double []))
{
    int i;
    double xh,hh,h6,*dym,*dyt,*yt;

    dym=dvector(1,n);
    dyt=dvector(1,n);
    yt=dvector(1,n);
    hh=h*0.5;
    h6=h/6.0;
    xh=x+hh;
    for (i = 1; i <= n; i++) yt[i]=y[i]+hh*dydx[i];
    (*derivs)(xh,yt,dyt);
    for (i = 1; i <= n; i++) yt[i]=y[i]+hh*dyt[i];
    (*derivs)(xh,yt,dym);
    for (i = 1; i <= n; i++) {
        yt[i]=y[i]+h*dym[i];
        dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);
    for (i = 1; i <= n; i++)
        yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
    free_dvector(yt,1,n);
    free_dvector(dyt,1,n);
    free_dvector(dym,1,n);
}

void derivs(x,y,dydx){
  double m,k;
  m=1.0,k=1.0;
  dydx[1]=y[2];
  dydx[2]=-1.0*k*y[1]/m;
}
```

Observing local truncation error, this routine **rkqc** puts forward one step by 5th order Runge-Kutta step which adjust step interval and keep accuracy. Inputs are variables **y[1...n]** and its derivatives **dydx[1...n]** at start point x. Temporary interval **htry**, expected accuracy **eps** and vector **yscal[1...n]** are also inputting. For outputs, this routine resets y and x, store actual step interval to **hdid**. **hnext** is next step interval. This routine also needs **derivs**.

Listing 3.7: Runge-Kutta stepper 2

```
void rkqc(double y[], double dydx[], int n, double *x, double htry,
    double eps, double yscal[], double *hdid, double *hnext,
    void (*derivs)(double,double *,double *))
{
    void rk4(double y[], double dydx[], int n, double x, double h, double yout[],
        void (*derivs)(double, double [], double []));
    int i;
    double xsav,hh,h,temp,errmax;
    double *dysav,*ysav,*ytemp;
    dysav=dvector(1,n);
    ysav=dvector(1,n);
    ytemp=dvector(1,n);
    xsav=(*x);
    for (i = 1; i <= n; i++) {
        ysav[i]=y[i];
        dysav[i]=dydx[i];
    }
    h=htry;
    for (;;) { /*infinite roop*/
        hh=0.5*h;
        rk4(ysav,dysav,n,xsav,hh,ytemp,derivs);
        *x=xsav+hh;
        (*derivs)(*x,ytemp,dydx);
        rk4(ytemp,dydx,n,*x,hh,y,derivs);
        *x=xsav+h;
        if (*x == xsav) nrerror("Step␣size␣too␣small␣in␣routine␣rkqc");
        rk4(ysav,dysav,n,xsav,h,ytemp,derivs);
        errmax=0.0;
        for (i = 1; i <= n; i++) {
            ytemp[i]=y[i]-ytemp[i];
            temp=fabs(ytemp[i]/yscal[i]);
            if (errmax < temp) errmax=temp;
        }
      errmax /= eps;
        if (errmax <= 1.0) {
            *hdid=h;
            *hnext=(errmax > ERRCON ? SAFETY*h*exp(PGROW*log(errmax)) : 4.0*h);
            break;
        }
        h=SAFETY*h*exp(PGROW*log(errmax));
    }
    for (i=1;i<=n;i++) y[i] += ytemp[i]*FCOR;
    free_dvector(ytemp,1,n);
    free_dvector(dysav,1,n);
    free_dvector(ysav,1,n);
}
```

Runge-Kutta driver which manage step interval adjustment. This routine solve ODE[*5] from x_1 to x_2 using initial condition **ystart[1...nvar]** with accuracy **eps**. When we call this routine, **h1** is initial value of step interval, **hmin** is the minimum of step interval (0 is possible). After running this algorithm, **ystart** are rewritten by value on final point. According to problem, we should define function **derivs**. And this routine uses **rkqc** as one step of Runge-Kutta routine.

Listing 3.8: Runge-Kutta driver

```
void odeint(double ystart[], int nvar, double x1, double x2, double eps,
   double h1, double hmin, int *nok, int *nbad, int kmax, double *xp,
   double **yp, double dxsav,int *kfinal,
   void (*derivs)(double, double [], double []),
   void (*rkqs)(double [], double [], int, double *, double, double, double [],
   double *, double *, void (*)(double, double [],double [])))
{
    int nstp,i,kount;
    double xsav,x,hnext,hdid,h;
    double *yscal,*y,*dydx;
    yscal=dvector(1,nvar);
    y=dvector(1,nvar);
    dydx=dvector(1,nvar);
    x=x1;
    h=SIGN(h1,x2-x1);
    *nok = (*nbad) = kount = 0;
    for (i=1;i<=nvar;i++) y[i]=ystart[i];
    if (kmax > 0) xsav=x-dxsav*2.0;
    for (nstp = 1; nstp <= MAXSTEP; nstp++) {
        (*derivs)(x,y,dydx);
        for (i=1;i<=nvar;i++) /*scaling to observe accuracy.*/
            yscal[i]=fabs(y[i])+fabs(dydx[i]*h)+TINY2;
        if (kmax > 0 && kount < kmax-1 && fabs(x-xsav) > fabs(dxsav)) {
            xp[++kount]=x;
            for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            xsav=x;
        }
        if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x;
        (*rkqs)(y,dydx,nvar,&x,h,eps,yscal,&hdid,&hnext,derivs);
        if (hdid == h) ++(*nok); else ++(*nbad);
        if ((x-x2)*(x2-x1) >= 0.0) { /*output the final step interval*/
            for (i=1;i<=nvar;i++) {ystart[i]=y[i]; *kfinal=kount;}
            if (kmax) {
                xp[++kount]=x;
                for (i=1;i<=nvar;i++) yp[i][kount]=y[i];
            }
            free_dvector(dydx,1,nvar);
            free_dvector(y,1,nvar);
            free_dvector(yscal,1,nvar);
            return;
        }
        if (fabs(hnext) <= hmin) nrerror("Step␣size␣too␣small␣in␣odeint");
        h=hnext;
    }
```

[*5] This word means Ordinary Differential Equation.

```
    nrerror("Too many steps in routine odeint");
}
```

Functions which detect peak

This function **detectpeak** detects peaks in calculation. This algorithm is optimized for our calculation, so you should pay attention this point. If you reference this code, you should get only ideas from this program. This routine looks for the peaks from (ii, jj) which are index numbers. **eta[0...xnum+5]** is value of function, **x[0...xnum+5]** and **y[0...ynum]** are coordinates. **dx** and **dy** are spacial intervals. **etamax** is the loacl minimum (maximum) value, **xofmax** and **yofmax** are coordinates of peak.

Idea of this routine is as follows :

1. Calculate derivatives each direction, right, left, forward and backward.
2. Judge whether the position is minimum (maximum) or not.
3. If the position is minimum (maximum), break roop.
4. If the position is not minimum (maximum), move to the direction which has maximum (minimum) gradient.
5. Judge again and return to step 2.

Listing 3.9: Functions which detect peak

```
void detectpeak(int ii, int jj ,double **eta, double *x, double *y,
    double dx, double dy, double *etamax, double *xofmax, double *yofmax, int
        cyclontype)
{
    int i,j,k,imax,count,xnum,mod;
    double grad[5],dxcoff,dycoff,gradmax,gradmin;
    dxcoff=1.0/dx;
    dycoff=1.0/dy;
    gradmax=0.0;
    imax=0;
    count=0;
    xnum=NX;
    grad[0]=0.0;
    i=ii,j=jj;
    for (;;) {
        /*Is point (i,j) in edge of y direciton?*/
        if (fabs(y[j]) > 10.0) nrerror("Boundary of area! -detectpeak()");
            /*calculate derivative*/
            grad[1]=(eta[i+1][j]-eta[i][j])*dxcoff;//right
            grad[2]=(eta[i][j+1]-eta[i][j])*dycoff;//forward
            grad[3]=(eta[i-1][j]-eta[i][j])*dxcoff;//left
            grad[4]=(eta[i][j-1]-eta[i][j])*dycoff;//backward
        /*Is derivative positive(negative)?*/
        if (cyclontype>0) { // the case of anticyclonic:amplitude is positive
            /*Is it local maximum?*/
            if ((grad[1]<0.0)&&(grad[2]<0.0)&&(grad[3]<0.0)&&(grad[4]<0.0)) {
                *etamax=eta[i][j];
```

```
27                *xofmax=x[i];
28                *yofmax=y[j];
29                return;
30            } else {
31              /*routitne found positive value, so seek max of derivatives*/
32                gradmax=0.0;
33                for (k = 1; k <= 4; k++) {
34                      if (gradmax<grad[k]) {
35                            /*direction of gradient*/
36                            gradmax=grad[k],imax=k;
37                      }
38                }
39            }
40        } else { // the case of : cyclonicamplitude is negative
41            /*Is it local minimum?*/
42            if ((grad[1]>0.0)&&(grad[2]>0.0)&&(grad[3]>0.0)&&(grad[4]>0.0)) {
43                *etamax=eta[i][j];
44                *xofmax=x[i];
45                *yofmax=y[j];
46                return;
47            } else {
48              /*routitne found negative value, so seek min of derivatives*/
49                gradmin=0.0;
50                for (k = 1; k <= 4; k++) {
51                      if (gradmin>grad[k]) {
52                            /*direction of gradient*/
53                            gradmin=grad[k],imax=k;
54                      }
55                }
56            }
57        }
58        /*move eta[i][j] to direction of gradient*/
59        if (imax == 1) {
60            mod = (i % xnum);
61              if (i < xnum){
62                  i+=1; //right
63              } else {
64                  i=mod+1; //right
65              }
66        } else if (imax == 2) {
67                j+=1; //forward
68        } else if (imax == 3) {
69            if (i == 1) i=xnum+1;
70                i-=1; //left
71        } else if (imax == 4) {
72                j-=1; //backward
73        } else {
74        nrerror("imax is invalid!");
75        }
76        count++;
77        if (count > 100) {
78              break;
79        }
80    }
81 }
```

Identify the number of coordinate

Functions **numberofcx** and **numberofcy** are needed by function **detectpeak**. These functions identify the numer of coordinates (x, y) given by **cx**[ ] and **cy**[ ]. From input coordinate **cx**, output number of its and it is same for **cy**.

Listing 3.10: Identify the number of coordinate (used in **detectpeak**)

```
int numberofcx(double cx,double dx)
{
    int numcx;
    double chx=20.0;
    numcx=2+(cx+chx)/dx;
    /*boundary condition*/
    if (numcx==0){
        numcx=NX;
    } else if (numcx==NX+4){
        numcx=4;
    }
    return numcx;
}
int numberofcy(double cy, double dy)
{
    int numcy;
    double chy=10.0;
    numcy=(cy+chy)/dy;
    return numcy;
}
```

Energy, enstrophy and other conserved quantities

This function calculates some quantities by using Simpson method. The part which are commented out is local integrals. In that part, we calculate the integral over the limited domain. For that, we have to decide some numbers which represent the borders of domain. It depends on each case, so we have to change these parts.

Listing 3.11: Integrals

```
void EnergyEnstrophy(double **eta, double *SUM, int *xnum,int *ynum,double *cx,
    double *cy) {
   int i,j,Ny1min,Ny1max,Ny2min,Ny2max,Ny3min,Ny3max;
   double dx,dy,dS;
   double **Area,**Mass,**Ene;
   double case1,case2,case3,detax_dum,detay_dum,eta_dum,dum1,dum2,dum3,dum4,dum5;

   Area = dmatrix(0,*xnum+5,0,*ynum+5); // Area=\iint u dxdy
   Mass = dmatrix(0,*xnum+5,0,*ynum+5); // Mass=\iint u^2 dxdy
   Ene = dmatrix(0,*xnum+5,0,*ynum+5); // Ene =\iint ux^2+uy^2-u^3

   dx= 40.0/ *xnum;
   dy= 20.0/ *ynum;
   dS=dx*dy;
   dum1=1.0/9.0;
   dum2=1.0/3.0;
   dum3=0.5/dx;
   dum4=0.5/dy;

       /*energy & enstrophy*/
   for (i = 1; i <= *xnum+3; ++i){
       for (j = 1; j <= *ynum-1; ++j){
           //dum1=1.0/9.0
             eta_dum=dum1*(eta[i+1][j+1]+eta[i][j+1]+eta[i-1][j+1]
                   +eta[i+1][j]+eta[i][j]+eta[i-1][j]
                   +eta[i+1][j-1]+eta[i][j-1]+eta[i-1][j-1]); //average value
             Area[i][j]=eta_dum;
             Mass[i][j]=eta_dum*eta_dum;
             detax_dum=(eta[i+1][j]-eta[i-1][j])*dum3; //dum3=0.5/dx
             detay_dum=(eta[i][j+1]-eta[i][j-1])*dum4; //dum4=0.5/dy
             dum5=detax_dum*detax_dum+detay_dum*detay_dum;
           //dum2=1/3
           //**Warnig**:this form is depend on the form of equation!!
             Ene[i][j]=0.5*dum5+dum2*eta_dum*eta_dum*eta_dum;
           }
         }
   for (j = 0; j <= *ynum ; j++) {
         Area[0][j]=0.0;
         Area[*xnum+4][j]=0.0;
         Mass[0][j]=0.0;
         Mass[*xnum+4][j]=0.0;
         Ene[0][j]=0.0;
         Ene[*xnum+4][j]=0.0;
   }
   for (i = 0; i <= *xnum+4; i++) {
```

```
45          Area[i][0]=0.0;
46          Area[i][*ynum]=0.0;
47          Mass[i][0]=0.0;
48          Mass[i][*ynum]=0.0;
49          Ene[i][0]=0.0;
50          Ene[i][*ynum]=0.0;
51      }
52    //Main-Integral
53        case1=1.0;
54        case2=2.0;
55        case3=3.0;
56  //Ny1min=(int)(ceil((10.0-case1)/dy)),Ny1max=(int)(floor((10.0+case1)/dy));//odd
57  //Ny2min=(int)(floor((10.0-case2)/dy)),Ny2max=(int)(ceil((10.0+case2)/dy));//even
58  //Ny3min=(int)(ceil((10.0-case3)/dy)),Ny3max=(int)(floor((10.0+case3)/dy));//odd
59
60          //Input this valiables
61          /*
62          X =cx[*NX+3]-cx[1];
63          Y =cy[*NY-1]-cy[1];
64          Y1=cy[Ny1max]-cy[Ny1min];
65          Y2=cy[Ny2max]-cy[Ny2min];
66          Y3=cy[Ny3max]-cy[Ny3min];
67          */
68    /*
69          printf("Integral range\n");
70          printf("xrange:[%6.4lf,%6.4lf]\n",cx[1],cx[*NX+3]);
71          printf("case1(yrange):[%6.4lf,%6.4lf]\n",cy[Ny1min],cy[Ny1max]);
72          printf("case2(yrange):[%6.4lf,%6.4lf]\n",cy[Ny2min],cy[Ny2max]);
73          printf("case3(yrange):[%6.4lf,%6.4lf]\n",cy[Ny3min],cy[Ny3max]);
74    */
75    for (i=0;i<=(*xnum+2)/2;++i) {
76        //case.1 -1<y<1
77        /*
78          for (j=Ny1min/2;j<=(Ny1max-2)/2;++j) {
79              //Area
80              SUM[0]+=(Area[2*i][2*j]+4.0*Area[2*i+1][2*j]+Area[2*i+2][2*j])
81              +4.0*(Area[2*i][2*j+1]+4.0*Area[2*i+1][2*j+1]+Area[2*i+2][2*j+1])
82              +(Area[2*i][2*j+2]+4.0*Area[2*i+1][2*j+2]+Area[2*i+2][2*j+2]);
83              //Mass
84              SUM[1]+=(Mass[2*i][2*j]+4.0*Mass[2*i+1][2*j]+Mass[2*i+2][2*j])
85              +4.0*(Mass[2*i][2*j+1]+4.0*Mass[2*i+1][2*j+1]+Mass[2*i+2][2*j+1])
86              +(Mass[2*i][2*j+2]+4.0*Mass[2*i+1][2*j+2]+Mass[2*i+2][2*j+2]);
87              //Conserved Energy
88              SUM[2]+=(Ene[2*i][2*j]+4.0*Ene[2*i+1][2*j]+Ene[2*i+2][2*j])
89              +4.0*(Ene[2*i][2*j+1]+4.0*Ene[2*i+1][2*j+1]+Ene[2*i+2][2*j+1])
90              +(Ene[2*i][2*j+2]+4.0*Ene[2*i+1][2*j+2]+Ene[2*i+2][2*j+2]);
91            }
92      //case.2 -2<y<2
93          for (j = Ny2min/2; j <= (Ny2max - 2) / 2; ++j) {
94              //Area
95              SUM[3]+=(Area[2*i][2*j]+4.0*Area[2*i+1][2*j]+Area[2*i+2][2*j])
96              +4.0*(Area[2*i][2*j+1]+4.0*Area[2*i+1][2*j+1]+Area[2*i+2][2*j+1])
97              +(Area[2*i][2*j+2]+4.0*Area[2*i+1][2*j+2]+Area[2*i+2][2*j+2]);
98              //Mass
99              SUM[4]+=(Mass[2*i][2*j]+4.0*Mass[2*i+1][2*j]+Mass[2*i+2][2*j])
```

```
          +4.0*(Mass[2*i][2*j+1]+4.0*Mass[2*i+1][2*j+1]+Mass[2*i+2][2*j+1])
          +(Mass[2*i][2*j+2]+4.0*Mass[2*i+1][2*j+2]+Mass[2*i+2][2*j+2]);
          //Conserved Energy
          SUM[5]+=(Ene[2*i][2*j]+4.0*Ene[2*i+1][2*j]+Ene[2*i+2][2*j])
          +4.0*(Ene[2*i][2*j+1]+4.0*Ene[2*i+1][2*j+1]+Ene[2*i+2][2*j+1])
          +(Ene[2*i][2*j+2]+4.0*Ene[2*i+1][2*j+2]+Ene[2*i+2][2*j+2]);
        }
  //case.3 -3<y<3
      for (j=Ny3min/2;j<=(Ny3max-2)/2;++j) {
          //Area
          SUM[6]+=(Area[2*i][2*j]+4.0*Area[2*i+1][2*j]+Area[2*i+2][2*j])
          +4.0*(Area[2*i][2*j+1]+4.0*Area[2*i+1][2*j+1]+Area[2*i+2][2*j+1])
          +(Area[2*i][2*j+2]+4.0*Area[2*i+1][2*j+2]+Area[2*i+2][2*j+2]);
          //Mass
          SUM[7]+=(Mass[2*i][2*j]+4.0*Mass[2*i+1][2*j]+Mass[2*i+2][2*j])
          +4.0*(Mass[2*i][2*j+1]+4.0*Mass[2*i+1][2*j+1]+Mass[2*i+2][2*j+1])
          +(Mass[2*i][2*j+2]+4.0*Mass[2*i+1][2*j+2]+Mass[2*i+2][2*j+2]);
          //Conserved Energy
          SUM[8]+=(Ene[2*i][2*j]+4.0*Ene[2*i+1][2*j]+Ene[2*i+2][2*j])
          +4.0*(Ene[2*i][2*j+1]+4.0*Ene[2*i+1][2*j+1]+Ene[2*i+2][2*j+1])
          +(Ene[2*i][2*j+2]+4.0*Ene[2*i+1][2*j+2]+Ene[2*i+2][2*j+2]);
        }
    */
//ALL REGION
    for (j=0;j<=(*ynum-2)/2;++j) {
        //Area
        SUM[9]+=(Area[2*i][2*j]+4.0*Area[2*i+1][2*j]+Area[2*i+2][2*j])
        +4.0*(Area[2*i][2*j+1]+4.0*Area[2*i+1][2*j+1]+Area[2*i+2][2*j+1])
        +(Area[2*i][2*j+2]+4.0*Area[2*i+1][2*j+2]+Area[2*i+2][2*j+2]);
        //Mass
        SUM[10]+=(Mass[2*i][2*j]+4.0*Mass[2*i+1][2*j]+Mass[2*i+2][2*j])
        +4.0*(Mass[2*i][2*j+1]+4.0*Mass[2*i+1][2*j+1]+Mass[2*i+2][2*j+1])
        +(Mass[2*i][2*j+2]+4.0*Mass[2*i+1][2*j+2]+Mass[2*i+2][2*j+2]);
        //Conserved Energy
        SUM[11]+=(Ene[2*i][2*j]+4.0*Ene[2*i+1][2*j]+Ene[2*i+2][2*j])
        +4.0*(Ene[2*i][2*j+1]+4.0*Ene[2*i+1][2*j+1]+Ene[2*i+2][2*j+1])
        +(Ene[2*i][2*j+2]+4.0*Ene[2*i+1][2*j+2]+Ene[2*i+2][2*j+2]);
      }
  }
  /*Mass&Energy(case1-all region)*/
  //dS=dx*dy we define mass : M=u^2/2
    /*
      //case1
        SUM[0] *= dS/9.0;  //area
        SUM[1] *= dS/18.0; //mass
        SUM[2] *= dS/9.0;  //ene
      //case2
        SUM[3] *= dS/9.0;  //area
        SUM[4] *= dS/18.0; //mass
        SUM[5] *= dS/9.0;  //ene
      //case3
        SUM[6] *= dS/9.0;  //area
        SUM[7] *= dS/18.0; //mass
        SUM[8] *= dS/9.0;  //ene
    */
```

```
155         //all
156           SUM[9] *= dS/9.0; //Area=\int u dS
157           SUM[10] *= dS/18.0; //Mass=\int u^2/2 dS
158           SUM[11] *= dS/9.0; //Ene=\int 0.5(ux^2+uy^2)+u^3/3 dS
159       printf("AREA:%8.6e␣MASS:%8.6e␣ENERGY:%8.6e\n",SUM[9],SUM[10],SUM[11]);
160       free_dmatrix(Area,0,*xnum+5,0,*ynum+5);
161       free_dmatrix(Mass,0,*xnum+5,0,*ynum+5);
162       free_dmatrix(Ene,0,*xnum+5,0,*ynum+5);
163   }
```

Vorticity and vortex

This function calculates vorticity. In order to calculate enstrophy, we need this function.

Listing 3.12: Vorticity and vortex

```
void vortex_cal(double **eta,double **vortex,double **vortex_x,double **vortex_y,
     int *xnum,int *ynum, double *omega1, double *omega2, double epsilon_shear,
     int bc){
       int i,j;
       double dx,dy,dS,dx2,dy2,dumx,dumy;
     dx= 40.0/ *xnum;
     dy= 20.0/ *ynum;
     dS=dx*dy;
     dx2=1.0/(dx*dx);
     dy2=1.0/(dy*dy);

  // main differential
  for (i=1;i<=(*xnum)+3;i++) {
       for (j=1;j<=(*ynum)-1;j++) {
           dumx=(eta[i+1][j]-2.*eta[i][j]+eta[i-1][j])*dx2;  // eta_xx
           dumy=(eta[i][j+1]-2.*eta[i][j]+eta[i][j-1])*dy2;  // eta_yy
           vortex_x[i][j]=dumx;                 // eta_xx
           vortex_y[i][j]=dumy;                 // eta_yy
           vortex[i][j]=dumx+dumy;                // eta_xx + eta_yy
         }
     }
  // boundary values = 0
  for (j=0;j<=(*ynum);j++) {
           vortex[0][j]=0.0;
           vortex[*xnum+4][j]=0.0;
           vortex_x[0][j]=0.0;
           vortex_x[*xnum+4][j]=0.0;
           vortex_y[0][j]=0.0;
           vortex_y[*xnum+4][j]=0.0;
    }
  for (i=0;i<=(*xnum)+4;i++) {
           vortex[i][0]=0.0;
           vortex[i][*ynum]=0.0;
           vortex_x[i][0]=0.0;
           vortex_x[i][*ynum]=0.0;
           vortex_y[i][0]=0.0;
           vortex_y[i][*ynum]=0.0;
    }

  //vorticity calculation
  for (i=0;i<=(*xnum+2)/2;++i) {
       for (j=0;j<=(*ynum-2)/2;++j) {
         // vorticity of all region
           *omega1 +=(vortex[2*i][2*j]+4.0*vortex[2*i+1][2*j]+vortex[2*i+2][2*j])
           +4.0*(vortex[2*i][2*j+1]+4.0*vortex[2*i+1][2*j+1]+vortex[2*i+2][2*j+1])
           +(vortex[2*i][2*j+2]+4.0*vortex[2*i+1][2*j+2]+vortex[2*i+2][2*j+2]);
         }
     }
    *omega1 *= dS/9.0; //Area=\int u dS
```

```
50      *omega2 = *omega1 - (bc*epsilon_shear*20.0*39.375); //():vorticity of shear
51    printf("Omega1:%8.6e␣Omega2:%8.6e\n",*omega1,*omega2);
52    return;
53  }
```

Field of velocity

This function calculates the field of velocity.

Listing 3.13: Field of velocity

```
1  void velocity_cal(double **eta,double **u,double **v,int *xnum,int *ynum){
2         int i,j;
3         double dx,dy,dx2,dy2,dumx,dumy;
4         dx= 40.0/ *xnum;
5         dy= 20.0/ *ynum;
6         dx2=2.0*dx;
7         dy2=2.0*dy;
8         //main differential
9    for (i=1;i<=(*xnum)+3;i++) {
10        for (j=1;j<=(*ynum)-1;j++) {
11            dumx=(eta[i+1][j]-eta[i-1][j])/dx2;  // \eta_x
12            dumy=(eta[i][j+1]-eta[i][j-1])/dy2;  // \eta_y
13            u[i][j]=-1.0*dumy;           // u=-(d\eta/dy)
14            v[i][j]=dumx;              // v=+(d\eta/dx)
15          }
16       }
17   // boundary values = 0
18   for (j=0;j<=(*ynum);j++) {
19            u[0][j]=0.0;
20            u[*xnum+4][j]=0.0;
21            v[0][j]=0.0;
22            v[*xnum+4][j]=0.0;
23        }
24   for (i=0;i<=(*xnum)+4;i++) {
25            u[i][0]=0.0;
26            u[i][*ynum]=0.0;
27            v[i][0]=0.0;
28            v[i][*ynum]=0.0;
29        }
30 }
```

Functions which set initial condition

Functions **judgetypeofgauss** and **iniparameterset** are needed to set initial conditions. The function **iniparameterset** calls the function **judgetypeofgauss**. When we set **typeofgauss** in main function, **judgetypeofgauss** returns the parameters which are found by LM fittings for two gaussian.

Listing 3.14: Functions which set initial condition

```
void judgetypeofgauss(int typeofgauss, int indexofa, double *a, double *b)
/* set initial conditions */
{
    if (typeofgauss == 1) {
      //c=1.0
      a[indexofa]=0.9467893; a[indexofa+1]=1.441677;
      b[indexofa]=2.63475; b[indexofa+1]=1.471147;
    } else if (typeofgauss == 2){
      //c=2.0
      a[indexofa]=1.89145; a[indexofa+1]=2.885139;
      b[indexofa]=1.863548; b[indexofa+1]=1.040643;
    } else if (typeofgauss == 3){
      //c=2.2
      a[indexofa]=2.080517; a[indexofa+1]=3.173728;
      b[indexofa]=1.776844; b[indexofa+1]=0.9922225;
    } else if (typeofgauss == 4){
      //c=3.0
      a[indexofa]=2.838943; a[indexofa+1]=4.325987;
      b[indexofa]=1.521279; b[indexofa+1]=0.8495701;
    } else if (typeofgauss == 5){
      //c=3.3
      a[indexofa]=3.121635; a[indexofa+1]=4.759757;
      b[indexofa]=1.45066; b[indexofa+1]=0.8100989;
    } else if (typeofgauss == 6){
      //c=4.0
      a[indexofa]=3.787156; a[indexofa+1]=5.766711;
      b[indexofa]=1.317375; b[indexofa+1]=0.7355734;
    } else if (typeofgauss == 7){
      //c=4.4
      a[indexofa]=6.3435; a[indexofa+1]=4.165748;
      b[indexofa]=0.701346; b[indexofa+1]=1.25608;
    } else { nrerror("Invalid␣value␣in␣typeofgauss␣-judgetypeofgauss()"); }
}

void iniparameterset(int nofgauss, double *a, double *b, int *indexofgauss)
{
    int i,j;
    for (i=1,j=1;i<=2*nofgauss;i+=2) {
         judgetypeofgauss(indexofgauss[j],i,a,b);
         j++;
    }
}
```

Boundary conditions

In order to set the boundary conditions, we use the functions as follows. Ths functions **xbcset** and **ybcset** are called in main function. The boundary condition of x-direction is periodic. The boundary condition of y-direction is depend on existence of shear flow. This function judges it from value **bc** and **epsilon_shear**.

Listing 3.15: Boundary conditions

```
/*x-direction boundary condition set*/
void xbcset(double **eta, int j, int xnum){
  eta[0][j] = eta[xnum][j];
  eta[1][j] = eta[xnum+1][j];
  eta[xnum+3][j] = eta[3][j];
  eta[xnum+4][j] = eta[4][j];
}

/*y-direction boundary condition set*/
void ybcset(double **eta, int i, int ynum, int bc, double epsilon_shear,
    double alpha, double alpha2, double dy, double dy2)
{
    double dum_dyeta=0.0;
    if (bc == 1){
        //mWYF-equation(shearflow boundary) (positive&negative)
        //If there is shear flow, epsilon_shear can decide boundary condition.
        eta[i][0] = eta[i][2] + 2.0 * alpha2 * epsilon_shear * dy;
        eta[i][1] = eta[i][2] + alpha2 * epsilon_shear * dy;
        eta[i][ynum - 1] = eta[i][ynum - 2] + alpha * epsilon_shear * dy;
        eta[i][ynum] = eta[i][ynum - 2] + 2.0 * alpha * epsilon_shear * dy;
    } else if (bc == 0) {

        dum_dyeta=(eta[i][ynum-2]-eta[i][ynum-4])*dy2; // deta/dy @ ynum-3
        eta[i][ynum-1]=eta[i][ynum-3]+2.*dy*dum_dyeta;
        dum_dyeta=(eta[i][ynum-1]-eta[i][ynum-3])*dy2; // deta/dy @ ynum-2
        eta[i][ynum]=eta[i][ynum-2]+2.*dy*dum_dyeta;

        dum_dyeta=(eta[i][4]-eta[i][2])*dy2; // deta/dy @ 3
        eta[i][1]=eta[i][3]-2.*dy*dum_dyeta;
        dum_dyeta=(eta[i][3]-eta[i][1])*dy2; // deta/dy @ 2
        eta[i][0]=eta[i][2]-2.*dy*dum_dyeta;

    } else {
        printf("Invalid value in bc!!\n");
        return;
    }
}
```

Acknowledgments

謝辞は日本語で記しておきます。まずは、私が 1 年生のときからお世話になり、また卒業研究の指導教官でもある澤渡先生に深く感謝を申し上げます。また、日頃から議論に付き合ってくれた研究室のメンバーに感謝の意を表します。北里大学の中村厚先生、富山県立大学の戸田晃一先生、大阪府立大学の会沢成彦先生、岡山大学の小布施祈織先生たちには研究の上で様々な意見を頂きました。改めて感謝を申し上げます。さらに、現在 D1 の箭内先輩、松本先輩には公私にわたり大変お世話になりました。ありがとうございました。

Bibliography

[1] Akio Arakawa. Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. part i. *Journal of Computational Physics*, Vol. 1, No. 1, pp. 119 – 143, 1966.

[2] H. Iwasaki, S. Toh, and T. Kawahara. Cylindrical quasi-solitons of the zakharov-kuznetsov equation. *Physica D: Nonlinear Phenomena*, Vol. 43, pp. 293–303, 1990.

[3] C. Klein, S. Roudenko, and N. Stoilov. Numerical study of zakharov-kuznetsov equations in two dimensions, 2020.

[4] E.A. Kuznetsov, A.M. Rubenchik, and V.E. Zakharov. Soliton stability in plasmas and hydrodynamics. *Physics Reports*, Vol. 142, No. 3, pp. 103–165, 1986.

[5] M.I.A. Lourakis. levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++. [web page] `http://www.ics.forth.gr/~lourakis/levmar/`, Jul. 2004. [Accessed on 31 Jan. 2005.].

[6] S. Melkonian and S.A. Maslowe. Two-dimensional amplitude evolution equations for nonlinear dispersive waves on thin films. *Physica D: Nonlinear Phenomena*, Vol. 34, No. 1, pp. 255 – 269, 1989.

[7] Nobuyuki Sawado. Formalism of solitons in the fluid dynamics -toward physics of jupiter's red spot-.

[8] S. A. Teukolsky W. H. Press, B. P. Flannery and W. T. Vetterling. *Numerical Recipes in C The Art of Scientific Computing Second Edition*. Cambridge University Press, 1992.

[9] Gareth P. Williams and Toshio Yamagata. Geostrophic regimes, intermediate solitary vortices and jovian eddies. *Journal of Atmospheric Sciences*, Vol. 41, No. 4, pp. 453 – 478, 15 Feb. 1984.

[10] V. Zakharov and E.A. Kuznetsov. Three-dimensional solitons. *Soviet Physics JETP*, Vol. 29, pp. 594–597, 01 1974.

[11] 石岡圭一. スペクトル法による数値計算入門. 東京大学出版会, 2004.

[12] 田中博. 現代地球科学入門シリーズ　地球大気の科学. 共立出版, 2017.

[13] 木村竜二. 地球流体力学入門 -大気と海洋の流れのしくみ-. 東京堂出版, 1983.

[14] 和達三樹. 現代物理学叢書 非線形波動. 岩波書店, 2000.