MPO 662 – Problem Set 6

Abstract

The aim of this homework is to perform an in-depth analysis of the energy conserving discretization of the shallow water equation presented in class. The analysis centers on the continuous and discrete equations, and a working code should be produced at the end. This code must be validated against a known test problem. The reference to consult for this problem is: Sadourny (1975) and which can be downloaded from the course web site.

1 Introduction

The inviscid shallow water equations in vector form are:

\[
\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + f \mathbf{k} \times \mathbf{u} + g \nabla \eta = 0 \tag{1}
\]

\[
\frac{\partial \eta}{\partial t} + \nabla \cdot [(H + \eta) \mathbf{u}] = 0. \tag{2}
\]

where \( \mathbf{u} \) is the velocity vector, \( \eta \) the surface displacement, \( H \) the resting depth of the fluid, \( f \) is the Coriolis parameter, \( \mathbf{k} \) the unit vertical vector, \( H \) the resting depth of the fluid, and \( g \) the gravitational acceleration. The instantaneous layer thickness is \( h = H + \eta \). The resting depth can be a function of space, but not time; the continuity equation can thus be written as \( \frac{\partial h}{\partial t} + \nabla \cdot h \mathbf{u} = 0 \)

2 The continuous equations

Defining the vorticity: \( \zeta = \nabla \times \mathbf{u} = v_x - u_y \), and potential vorticity \( q = \frac{f + \zeta}{h} \), show that the momentum equation can be re-written in the equivalent form:

\[
\frac{\partial \mathbf{u}}{\partial t} + q \mathbf{k} \times h \mathbf{u} + \nabla \left( \frac{\mathbf{u} \cdot \mathbf{u}}{2} + g \eta \right) = 0 \tag{3}
\]

Show that this system of equation conserves the following quantities

- Total Volume: \( \int_A \eta \, dA \) \tag{4}
- Total Energy: \( \int_A \left[ \frac{g\eta^2}{2} + h \frac{\mathbf{u} \cdot \mathbf{u}}{2} \right] \, dA \) \tag{5}
- Potential Enstrophy: \( \int_A h q^2 \, dA, \quad q = \frac{f + \zeta}{h} \) \tag{6}

Hints: Taking the inner product of the momentum equation with \( h \mathbf{u} \) to derive the energy equation. To derive the enstrophy conservation equation, first derive the equation for potential vorticity, multiply it by \( hq \) and derive the conservation equation for potential enstrophy. Use the vector identities in the appendix, 6 in your derivations.
The depth and pressure are defined at the center of cell \((j, k)\). The x-component of the velocity is defined on vertical cell edges \((j + \frac{1}{2}, k)\) and the y-components at the centers of horizontal edges \((j, k + \frac{1}{2})\). The vorticity, coriolis force, and enstrophy are defined on cell corners \((j + \frac{1}{2}, k + \frac{1}{2})\).

### 3 The discrete equations

The equations are discretized using finite differences, and the variables are staggered according to the Arakawa C-grid shown in figure 1. Show that the following finite difference scheme conserves energy discretely:

\[
\begin{align*}
\frac{\partial u}{\partial t} - qV_{xy} + \delta_x \Phi &= 0 \\
\frac{\partial v}{\partial t} + qU_{yx} - \delta_y \Phi &= 0 \\
\frac{\partial \eta}{\partial t} + \delta_x U + \delta_y V &= 0
\end{align*}
\]

The new variables are defined on the Arakawa C-grid as follows:

- x-Mass Fluxes \( U = \overline{h^x u} \) \( u\)-point
- y-Mass Fluxes \( V = \overline{h^y v} \) \( v\)-point
- Total Head \( \Phi = g\eta + \frac{\overline{u^2} + \overline{v^2}}{2} \) \( \eta\)-point
- Potential vorticity \( q = \frac{\delta_x v - \delta_y u + f}{h} \) \( \zeta\)-point

The operator \( \delta_x a \) and \( \overline{ax} \) are those defined in your notes. You can either follow the outline of Sadourny’s derivation to prove the discrete conservation laws, or you can follow the technique outlined in class of grouping terms in flux form. If you follow the latter technique the derivation of the energy equation requires that you form the following operation to bring the energy definition to the pressure points:

\[
\frac{\partial E}{\partial t} = h\overline{uu_t} + h\overline{vv_t} + \Phi\eta_t
\]

### 4 An energy conserving code

Write a code to solve the shallow water equation using the above energy conserving scheme. When developing large codes it is useful to adopt an incremental strategy: divide the code into small
useful chunks, verify that these chunks work as expected, and adopt them in the larger code as working properly. The validation strategy can also proceed by incrementally increasing the tests complexity. Here we outline a strategy that can help develop the entire code step-by-step and reliably. The incremental coding comes about by simplifications: either dropping terms from the equations (linearizing) or setting some parameters to zero.

4.1 Linearized 2D shallow water equations

One way to achieve that for the shallow water equation is not deal with the nonlinear terms nor rotational terms in the beginning. You are then solving a linearized version of the equation. By doing that you defer dealing with the nonlinear and rotational terms to a later stage, and you can have access to an exact solution. The linearized equations without rotation are:

\[
\begin{align*}
\frac{\partial u}{\partial t} + \delta_x \Phi^l &= 0 \\
\frac{\partial v}{\partial t} + \delta_y \Phi^l &= 0 \\
\frac{\partial \eta}{\partial t} + \delta_x U + \delta_y V &= 0
\end{align*}
\]

where \( \Phi^l = g \eta \), and \((U, V) = (\mathcal{T}^x u, \mathcal{T}^y v)\). Write a code to solve the above equations starting from the 1D code you have written for homework 5. The task is then to figure out how to deal with the 2D arrays, and validate the code against an analytical solution.

The analytical solution for a standing wave in a closed rectangular basin of dimension \( a \times b \) is:

\[
\begin{align*}
u &= U \sin \frac{m \pi x}{a} \cos \frac{n \pi y}{b} \sin \sigma t \\
v &= V \cos \frac{m \pi x}{a} \sin \frac{n \pi y}{b} \sin \sigma t \\
\eta &= A \cos \frac{m \pi x}{a} \cos \frac{n \pi y}{b} \cos \sigma t
\end{align*}
\]

\[
U = \frac{gA m \pi}{\sigma a}, \quad V = \frac{gA n \pi}{\sigma b}, \quad \sigma^2 = gH \left[ \left( \frac{m \pi}{a} \right)^2 + \left( \frac{n \pi}{b} \right)^2 \right]
\]

where \( m, n \) are mode numbers in the \( x \) and \( y \) directions, respectively. Use this solution to validate your code. You may set \( g = H = A = a = b = 1 \) in your code and make sure you run the following cases:

1. \( n = 0 \) and \( m = 1 \) which corresponds essentially to a 1D SWE in \( x \).
2. \( m = 0 \) and \( n = 1 \) which corresponds essentially to a 1D SWE in \( y \).
3. \( m = 4 \) and \( n = 3 \) which is a fully 2D case.

Verify in each case that the mass is conserved to machine accuracy. Double check the order of the scheme for the 2D case by carrying out a systematic convergence study. Integrate for at least 2 wave cycles: \( T > 2\pi / \sigma \). The initialization for this test is in tlinwave.f90: copy this file to shinit.f90 before compiling.
4.2 Nonlinear 2D shallow water equations

Extend the code you have developed for the linearized SWE equation to cover the non-linear and rotating case.

Use it to simulate the propagation of an equatorial Rossby soliton. The Rossby soliton dynamics hinges on the interplay between the wave-steepening caused by the non-linear terms and the dispersion caused by the Coriolis term; the two are exactly balanced and the soliton propagates without change of shape for long distances. The codes to initialize the Rossby soliton and to perform the time-integration will be provided. What is required of you is to provide all the tendency terms, i.e. those others than $\frac{\partial u}{\partial t}$. The solution of the equatorial Rossby soliton has been studied in Boyd (1980) where an asymptotic solution in normalized variables is provided.

The domain is an equatorial $\beta$-plane where $-24 \leq x \leq 24$, and $-8 \leq y \leq 8$. The Coriolis variation is given by $f = y$. The depth of the fluid is $H = 1$ and the gravitational acceleration is $g = 1$. The final integration time is $t = 40$. A stable time step for the coarsest resolution is $\Delta t = 0.02$ (stable time-step); please explain what sets it. The boundaries are closed: so that the boundary conditions are: $u(\pm 24, y) = 0$, $v(x, \pm 8) = 0$; use symmetry when updating variables associated with the vorticity. The initial conditions are:

\begin{align*}
    u &= \frac{6y^2 - 9}{4} \phi(x)e^{-\frac{y^2}{2}} \\
    v &= 2y \frac{\partial \phi}{\partial x} e^{-\frac{y^2}{2}} \\
    \eta &= \frac{6y^2 + 3}{4} \phi(x)e^{-\frac{y^2}{2}} \\
    \phi(x) &= 0.771B^2 \text{sech}^2 Bx \\
    \frac{\partial \phi}{\partial x} &= -2B \tanh Bx \phi \\
    B &= 0.395
\end{align*}

You should monitor the evolution of the energy, mass and potential enstrophy during the course of the simulation. Provide contour lines of the initial, mid-, and final states. Try 3 spatial resolutions: $\Delta x = \Delta y = 0.5, 0.25, 0.125$.

The code to initialize for the Rossby soliton problem is in the tarred file. See the attached documentation on how to untar the files and the function of the different subroutines.
Figure 2: Equatorial Rossby Soliton initial conditions, from top to bottom: $u$, $v$ and $\zeta$, respectively.
5 Code Documentation and Installation

The codes provided here are for your convenience. Feel free to use any, or none of the subroutines, particularly if you feel it is too cumbersome to read or that you have better code. However, should a particular section of code (beside the coding of \texttt{swhrs} routine) prove buggy, here is a fall back that I have tested. The second purpose is to present some code development practices like global variables, makefiles, code validation, etc... This section gives a summary description of the fortran routine and the matlab script included in the tar file.

5.1 Installation

To untar the files, do the following:

1. Save the file to your disk

2. unpack using the tar command:
   \texttt{tar xvf ers.tar}
   this will create a directory called ERS with all the files in it.

3. Modify the \texttt{Makefile} to suit your system. It currently contains the compiler and associated options for 4 popular compilers: the GNU compiler gfortran, the intel compiler ifort, the Portland Group pgf90, and the IBM xlf90 compiler. The compiler name is stored in a variable called \texttt{FC} and its options are in \texttt{FFLAGS}. Several of these definitions exist for each compiler. Comment out the ones you don’t need and leave the ones you need, that is a single definition of \texttt{FC} and a single definition of \texttt{FFLAGS}. A makefile comment line starts with the \# symbol. Finally, there are two definitions of \texttt{FFLAGS} per compiler, one includes options for debugging, array bounds checking, invalid operations, . . . , and the other one contains options for code optimization. You should ALWAYS turn debugging ON and optimization off when you are developing the code. Once you are done with checking the validity of your code and that you have weeded out the bugs, turn off debugging (because it slows a code to crawl) and turn optimization on.

4. To make the executable, just type “\texttt{make}” or “\texttt{make shallow}”, and an executable called \texttt{shallow} will be created. To start fresh if you made a mistake in the compilation, say “\texttt{make clean}”, and start over. If you find the make utility and the multiple files too cumbersome feel free to merge them into a single file. Since module must be declared before being used, you want to merge the files in a specific order. The order should be \texttt{kinds.f90}, \texttt{grid.f90}, \texttt{shparams.f90}, \texttt{shinit.f90}, \texttt{shrhs.f90}, \texttt{shtstep.f90} and \texttt{shallow.f90}.

5. To run the code say “./\texttt{shallow} < \texttt{shallow.in}”. The initial set-up is missing the correct subroutine \texttt{swhrs} that calculates the right hand side and your job is to write it. What is there simply sets the time tendencies to 0, so the solution will stay frozen to the IC no matter how many time-steps you take.

5.2 Code documentation

The directory contains several files. Here is an explanation for the content of each one of them:

1. \texttt{swe.m} is a simple matlab script to visualize the results. It requires the script \texttt{fortread.m} to read fortran binary files.
2. **shallow.f90**: is the main program, it calls the initialization, the time stepping routines, does I/O, and calculates diagnostics.

3. **shallow.in** contains run-time parameters that the code reads. These are easy to understand and are documented in the file itself.

4. **shparams.f90**: is a module that declares a few global variables, and a subroutine to read the input problem parameters. The variables declared in the data portion of the module (before the `contains` statement) are the global variables, most notable among them are the number of cells in the \( x, y \) directions: \( Nx, Ny \), and the grid size \( dx \) and \( dy \). Modules (or subroutines) that need access to these variables in `shparams` must contain the line `use shparams`. For example, `shtstep` accesses only \( Nx \) and \( Ny \) from `shparams` whereas `shrhs` would need to access: `gravity`, `depth`, `bdrag`, `viscosity`, `dx`, `dy`, `Nx`, `Ny`, `Nx1`, `Ny1`.

5. **grid.f90** Initializes the grid coordinates. There are \( Nx \times Ny \) cells in the \( x \times y \) directions. The coordinates of the cell edges are defined in arrays `xe(1:Nx+1)` and `ye(1:Ny+1)`. Here `xe(1), ye(1)` is the lower left corner and `xe(Nx+1), ye(Ny+1)` is the upper right corner.

6. **shinit.f90** contains the initial conditions for \( u, v, \eta \). Different problems can be set by changing `shinit.f90`. The initialization of the Equatorial Rossby Soliton is duplicated in `trossby2.f90`, and that for the linear standing wave in `tlinwave.f90`. Copy the proper file to `shinit.f90` before building the executable. The initialization call has the form:
   
   ```fortran
   call uvh_initial(u,v,zt, xe,ye, nx,ny, time)
   ```

   `u,v,zt` are the \( x/y \) velocities, and `zt` is the pressure. They are double precision arrays of size:
   
   ```fortran
   real*8 :: u(nx+1,ny), v(nx,ny+1), zt(nx,ny)
   ```

   `time` is an optional argument and can be omitted in which case it defaults to 0.

7. **shtstep.f90** contains subroutines for time-stepping: `RK4Step`, `RK3Step` or `AB3Step`, choose which ever one you want. The call is as follows:
   
   ```fortran
   call RK4Step(u,v,zt,nx,ny,dt)
   ```

   where `dt` is the time-step. On input `u,v,zt` contain the variable at the current time step, and on output the variables at the next time step.

8. **kinds.f90**: is a module that encapsulates the double real-type.

9. **shrhs.f90** contains the subroutine to calculate the time tendencies for the shallow water equations, namely `swerhs`. It also has subroutines to initialize the Coriolis parameter and the wind-stress (if any). Both variables are local to the subroutine.

### 5.3 Programming task

You have to code a subroutine called `swerhs` to calculate the right hand side of the equation using the FD scheme shown above. The call should have the following format:

```fortran
subroutine swerhs(u,v,zt,ru,rv,rzt)
```

where `u,v,zt` are the variables at the current time level, and `ru,rv,rzt`, are the right hand side of the \( x-y \) momentum equations, and the continuity equations. They are double precision real variables of the same size as the corresponding variables. If you put the subroutine in a file called `shrhs.f90`, you would not need to change the file names in the Makefile.
5.4 Programming hints

5.4.1 Global variables

Global variables become necessary as the program grows since, firstly, it simplifies passing variables between different parts of the code, and secondly, it shortens the list of dummy arguments in a subroutine call. The longer this list the greater the chances for coding mistakes. The global variables in a code are visible to all subroutines. There are several ways to declare global variables, the way it is done in this code is via modules. The file shparams.f90 is a good example of a module. It is divided into two parts: the data section where variables are declared such as gravity and depth, and the subroutine section. The two are separated by the statement contains. The variables declared in the data section are all visible to the two subroutines in the module; these module variables must not be redeclared. To access module variables from another subroutine or module, one needs to insert the statement use shparams; the data and the subroutines in module shparams become available. Again, those variables should not be redeclared otherwise the new declarations will mask/override the imported variables.

5.4.2 Data types

Although most of you have used single precision variables; there are instances where double precision variables are desirable. Fortran 90 provides a simple way to switch between types (in the old Fortran 77 you needed to go through the entire code); all it requires is a little coding discipline. The module kinds.f90 declares the double precision type, and assigns it to the variable r8; now a variable of that type can be declared as follows:

```fortran
real(r8) :: depth

depth = 10.35_r8 ! 10.3 meter depth
depth = 5.e3_r8 ! 5000 meter
depth = 5_r8 ! 5 meter
```

By tagging the _r8 suffix you declare that variable to be of the desired type. If you need to change the precision to single, all you have to do now is change the declaration in kinds.f90 to something like

```fortran
integer, parameter :: r8 = selected_real_kind(7, 32)
```

5.4.3 Programming

You should first run the code as is without taking a single time step. The code will write out the initial condition for the Equatorial Rossby Soliton. You should try to plot (draw contours or elevation) of the initial conditions to see what they look like. I have some matlab scripts that can help you. The code outputs the data in fortran binary format which is not immediately readable by matlab. You can either change the format to ASCII or use my matlab scripts. The diagnostics subroutine where the energy, potential vorticity and potential enstrophy budgets are computed should be filled in. This will let you get familiar with the variable staggering on a C-grid.
It is best to write small subroutine to compute the different quantities needed, such as interpolating from the $\eta$-points to the $u$-points. By writing small, easily tested routine you simplify the development of large codes; most of the coding time consumed is not in the initial coding but in code validation. By writing small and clear subroutines you will simplify substantially the validation part.

References


SEE ME IF YOU HAVE ANY QUESTIONS
6 Vector calculus identities

The following vector identities are useful:

- The curl of a gradient is always zero:
  \[ \nabla \times \nabla \alpha = 0 \]  \hspace{1cm} (28)

- The divergence of the curl of any vector \( \mathbf{u} \) is always zero
  \[ \nabla \cdot \nabla \times \mathbf{u} = 0 \]  \hspace{1cm} (29)

- The Laplacian of a scalar field \( \psi \) is the divergence of the gradient:
  \[ \nabla^2 \psi = \nabla \cdot (\nabla \psi) \]  \hspace{1cm} (30)

- Laplacian of a vector
  \[ \nabla \times \nabla \times \mathbf{u} = \nabla (\nabla \cdot \mathbf{u}) - \nabla^2 \mathbf{u} \]  \hspace{1cm} (31)

- Gradient of an inner product
  \[ \nabla (\mathbf{u} \cdot \mathbf{v}) = (\mathbf{u} \cdot \nabla) \mathbf{v} + (\mathbf{v} \cdot \nabla) \mathbf{u} + \mathbf{u} \times (\nabla \times \mathbf{v}) + \mathbf{v} \times (\nabla \times \mathbf{u}) \]  \hspace{1cm} (32)

In particular when \( \mathbf{v} = \mathbf{u} \) we have:
  \[ \nabla \left( \frac{\mathbf{u} \cdot \mathbf{u}}{2} \right) = (\mathbf{u} \cdot \nabla) \mathbf{u} + \mathbf{u} \times (\nabla \times \mathbf{u}) \]  \hspace{1cm} (33)

which gives rise to the so-called rotational form of the non-linear term.

- Divergence of cross product
  \[ \nabla \cdot (\mathbf{u} \times \mathbf{v}) = \mathbf{v} \cdot \nabla \times \mathbf{u} - \mathbf{u} \cdot \nabla \times \mathbf{v} \]  \hspace{1cm} (34)

- Curl of cross product
  \[ \nabla \times (\mathbf{u} \times \mathbf{v}) = \mathbf{u} (\nabla \cdot \mathbf{v}) - \mathbf{v} (\nabla \cdot \mathbf{u}) + (\mathbf{v} \cdot \nabla) \mathbf{u} - (\mathbf{u} \cdot \nabla) \mathbf{v} \]  \hspace{1cm} (35)

- Product of scalar and vector
  \[ \nabla \cdot (\psi \mathbf{u}) = \mathbf{u} \cdot \nabla \psi + \psi \nabla \cdot \mathbf{u} \]  \hspace{1cm} (36)
  \[ \nabla \times (\psi \mathbf{u}) = \nabla \psi \times \mathbf{u} + \psi \nabla \times \mathbf{u} \]  \hspace{1cm} (37)