11.1 Introduction

Translating algorithms into code takes art, patience, and hard work. The process can be greatly shortened by following a few rules to implement the algorithm. The sources of the algorithm are not of concern here although in practice we should be familiar with the intent and limitations of any algorithm, particularly when it relates to numerical computations. We will leave these issues for later as we are mainly concerned with programming.

11.2 Rules of thumbs

This section lists a few rules of thumbs to improve the coding of algorithms, and help in the homework problem.

11.2.1 Define the problem

Understand what is required in the computations, define the input, the output and the computations to be performed. Making sure the problem is well-defined goes a long way in determining what kind of input data and what sort of output is required.

11.2.2 Divide and conquer

Dividing the problem into functionally separate tasks helps in making large coding projects manageable. Identify the repetitive tasks and consider isolating them into well-designed subroutines and functions. This subdivision is critical for the remainder of the programming tasks as it: divides into manageable units, allows
validation to proceed in stages, and simplify code reuse. The details of the computations will then be boxed into compartments and a higher level abstraction about the algorithm is then possible. Most numerical tasks follow the plan:

1. data input
2. perform computations
3. output results

Each of these tasks may contain a long list of subtasks but the divide and conquer strategy can be applied to those recursively.

11.2.3 Use top-down design

Start designing the problem from the top starting with the main program and filling in the subroutine as you go. This will help you clarify the flow of the code to yourself. Write shell routines in the initial go and start filling them and validating them in groups. Postpone the details of each routine as long as possible.

11.2.4 Use pseudo-code

Describe in pseudo code (i.e. programming language independent) the steps needed in the algorithms.

11.2.5 Decide on optimal data structure

In most numerical computations this will simply arrays as they are the most efficient data structures for many types of computations.

11.2.6 Conceptual simplicity

Avoid using obfuscated code even though it may seem great at the moment. Obfuscated code can be hard to read, and even the implementors would have a hard time remembering what it does 3 months after they coded it.

11.2.7 Validation

It is important to gain assurances that parts of the code are working properly in order to catch coding and (more ominously) algorithmic mistakes. It is then imperative that subroutines are validated as one is developing the code. If an error is introduced later it will useful to restrict its tracking to the untested portions of the code. Validation can take many forms:
11.3 ORDINARY DIFFERENTIAL EQUATIONS

1. Run a problem with a known (trivial) answer and see if your code replicates it.

2. Insert print statements at intermediate stages to see how far the program goes before crashing or starting to give non-sense.

3. Use a debugger to go through the code.

We will attempt to illustrate these principles by looking at the problem of integrating ordinary differential equations.

11.3 Ordinary Differential Equations

In science and engineering we often encounter equations of the sort:

\[
\frac{d\vec{w}}{dt} = \vec{F}(\vec{w}, t)
\]  

subject to the initial condition \( \vec{w}(t = 0) = \vec{w}_0 \). Here \( \vec{w} \) is a vector of unknowns and \( \vec{F} \) is the right hand side of the ordinary differential equation that may depend on \( \vec{w} \) and time. A simple example of the above problem is the problem of inertial oscillations on an \( f \)-plane (one where the Coriolis parameter is constant):

\[
\frac{du}{dt} = fv \\
\frac{dv}{dt} = -fu
\]

where \( f \) is the Coriolis parameter assumed constant here, and \( u \) and \( v \) are the zonal and meridional velocity components. In this case we have

\[
\vec{w} = \begin{pmatrix} u \\ v \end{pmatrix} \quad \vec{F} = \begin{pmatrix} fv \\ -fu \end{pmatrix}
\]

For the simple case where \( f \) is constant the solution is trivial to obtain. It is

\[
u = u_0 \cos ft + v_0 \sin ft \\
v = v_0 \cos ft - u_0 \sin ft
\]

where \( u_0 \) and \( v_0 \) are the components of \( \vec{w}_0 \).

11.4 Fourth Order Runge Kutta Method

One robust algorithm that integrates ordinary differential equations in time is the Runge Kutta method of order 4, or RK4 for short. The algorithm proceeds in four
stages to predict the value of the unknown function at the next time level starting from the known initial conditions. All that is needed is the repeated evaluation of the function $\vec{F}$ on the right hand side of equation 11.1. The steps of the algorithms can be described as follows: Given the solution at $\vec{w}^m$ and the time step $\Delta t$:

1. Compute the gradient of the function at time $t_n$ and predict a first temporary value at $t_n + \frac{\Delta t}{2}$

$$\vec{F}^{(1)} = \vec{F} \left( \vec{w}^m, t_n \right), \quad \vec{w}^{(1)} = \vec{w}^m + \frac{\Delta t}{2} \vec{F}^{(1)} \quad (11.7)$$

2. Compute the gradient of the function at time $t_n + \frac{\Delta t}{2}$ using the previous temporary value of the solution $\vec{w}^{(1)}$ and compute a second temporary value at $t_n + \frac{\Delta t}{2}$

$$\vec{F}^{(2)} = \vec{F} \left( \vec{w}^{(1)}, t_n + \frac{\Delta t}{2} \right), \quad \vec{w}^{(2)} = \vec{w}^m + \frac{\Delta t}{2} \vec{F}^{(2)} \quad (11.8)$$

3. Compute the gradient of the function at time $t_n + \frac{\Delta t}{2}$ using $\vec{w}^{(2)}$ and compute a first prediction of the solution at time $t_{n+1}$

$$\vec{F}^{(3)} = \vec{F} \left( \vec{w}^{(2)}, t_n + \frac{\Delta t}{2} \right), \quad \vec{w}^{(3)} = \vec{w}^m + \Delta t \vec{F}^{(3)} \quad (11.9)$$

4. Compute the gradient of the function at time $t_n + \Delta t$ and compute the correct solution at time $t_{n+1}$ using a linear combinations of the slopes computed so far

$$\vec{F}^{(4)} = \vec{F} \left( \vec{w}^{(3)}, t_n + \Delta t \right), \quad \vec{w}^{n+1} = \vec{w}^m + \Delta t \frac{\vec{F}^{(1)} + 2\vec{F}^{(2)} + 2\vec{F}^{(3)} + \vec{F}^{(4)}}{6} \quad (11.10)$$

### 11.5 Analysis

Here we will attempt to define the problem and analyzing the algorithm so as to start sketching out the design of our code.

### 11.6 Problem definition

We need to integrate the function $\vec{w}$ in time given its initial conditions $\vec{w}_0$. 
11.6. PROBLEM DEFINITION

11.6.1 Physical and Numerical Input Parameters

The parameters of the problem we wish to compute consist of

1. the Coriolis parameter $f$
2. the initial condition $\vec{w}_0$

These parameters are intrinsic to the problem and do not depend on the numerical scheme. The numerical parameters consist of:

1. The time step size $\Delta t$
2. and the number of time steps to take say $M$. The final integration time would then be $t_M = M \times \Delta t$ assuming we start our timer at 0.
3. We probably need to save the solution at intermediate steps to plot its variation in time. We will thus need a parameter to determine the frequency of writing out the solution.

11.6.2 Output Parameters

The solution $\vec{w}$ need to be saved as the computations proceed. Diagnostics about the computations should be helpful in telling us whether we are proceeding on the right track or not. One possible diagnostics is the energy contained in the system. Multiplying the equations by $u$ and $v$ respectively and adding them up it is evident that

$$u \frac{du}{dt} + v \frac{dv}{dt} = \frac{d}{dt} \left( \frac{u^2 + v^2}{2} \right) = 0$$

and hence the energy $(u^2 + v^2)/2$ should remain constant at all times. It is thus a good idea to monitor the energy as the calculation proceeds.

11.6.3 Analysis of the RK4 algorithm

One can make the following remarks about the algorithm:

1. The algorithm requires 4 steps that are similar in appearance.
2. At each stage the slope is computed and a temporary value for the solution is computed.
3. The final slope is a linear combinations of the slopes computed so far.
4. The algorithm requires the repeated evaluation of the slope with different input parameters
5. All components of the solutions are treated identically, so arrays are the
optimal data structure to store the solution.

6. The initial input state $\vec{w}^n$ is required for each stage and hence must be
preserved. It can only be overwritten at the last stage.

7. The RK4 subroutine does not care what the function $\vec{F}$ is, it is thus a prime
candidate to put it in a subroutine.

8. I need to store the temporary intermediate stages $\vec{w}^{(1,2,3,4)}$ at least until the
next slope is computed.

11.6.4 Design

- The main program would need to initialize the data and orchestrate the
integration and evaluation of the solution in time.

- A subroutine is needed to initialize the data if it is not read from a file (in
  case the data is coming from an experiment, for example).

- The repeated evaluation of $\vec{F}$ calls for it to be isolated in a subroutine that
  returns the slope given $\vec{w}$ and time $t$.

- The integration should be isolated in a subroutine also. It is easiest to design
  this subroutine if it is meant to take a single step only. So its input would be
  the value of the solution at time $t_n$ and its output would be the solution
  at time $t_{n+1}$.

- The RK4 routine does not need to know about the Coriolis parameters so
  it is probably best to set this parameters in the routine that computes the
  function $\vec{F}$.

A pseudo-code that reflect the above ideas is the following:

mainprogram
  Read Parameters dt, Ntimes, isnap
  Initialize Solution at t=0, w0
  save initial conditions
  Time-Loop for integration
    call rk4 for a single step
    once every isnap time step
      save solution
      compute and print a diagnostics
  end time loop
end mainprogram
11.7. The Coding and Incremental Validation

A first implementation of the main program, may look as follows

```fortran
program inert
    implicit none
    integer, parameter :: nd=2 ! dimension of solution vector
    real :: w(nd) ! solution vector
    integer :: Ntimes, isnap ! number of time steps and save frequency

    print *, 'Enter dt, Ntimes, isnap'
    read *, dt, Ntimes, isnap

    call Initialize(w,nd)
    call PrintOut(w,nd)
    call Diagnostics(w,nd)
    do it = 1,Ntimes
        call Rk4(w,time,dt,nd)
        if (mod(it,isnap)==0) then
            call Printout(w,nd)
            call Diagnostics(w,nd)
        endif
    enddo

    stop
end program inert
```

There are a few variable declarations missing but the compiler will help us find those pretty easily because the `implicit none` will require that all variables be declared. Now we begin the task of writing the subroutines. A first cut at the initialize subroutine would look like:
subroutine Initialize(w,nd)
    implicit none
    integer, intent(in) :: nd
    real, intent(out) :: w(nd)
    w(1) = 1.0
    w(2) = 0.0
    return
end subroutine Initialize

The subroutine is quite simple and it is unlikely that we have introduced a bug. Nevertheless, we could probably produce an executable and run the code to check it out. To do that however, we would need to provide the remainder of the subroutines. At this point all we need is the declaration for the computer to produce the executables. These would look like:

!***********************************************************************
subroutine PrintOut(w,nd)
    implicit none
    integer, intent(in) :: nd
    real, intent(in) :: w(nd)
    return
end subroutine PrintOut
!***********************************************************************
subroutine Diagnostics(w,nd)
    implicit none
    integer, intent(in) :: nd
    real, intent(in) :: w(nd)
    return
end subroutine Diagnostics
!***********************************************************************
subroutine Rk4(w,time,dt,nd)
    implicit none
    integer, intent(in) :: nd  ! dimension of solution vector
    real, intent(in) :: time,dt  ! time level and time step
    real, intent(inout) :: w(nd)  ! solution vector
    return
end subroutine Rk4

These routines contain only variable declarations and no executable statements at this stage. They are enough now to create an executable and run the code to initialize the vector \( w \). At this point we can ascertain that the code runs but we have no way of verifying the output; we need to write out the solution and the routine PrintOut is then necessary. It is probably best if we write the solution to a file that we can examine after the run is done. Here it is:
11.7. THE CODING AND INCREMENTAL VALIDATION

subroutine PrintOut(w, nd)
    implicit none
    integer, intent(in) :: nd
    real, intent(in) :: w(nd)
    logical, save :: first=.true. ! save value between calls

    if (first) then ! open file on first call only
        open(8, file='output.dat', status='unknown', form='formatted', &
             action='write')
        first = .false. ! done with first call
    endif
    write(8,*) w(1), w(2) ! write solution in two column

    return
end subroutine PrintOut

The subroutine imbeds all the necessary output operations. On first entry to the subroutine, the logical first is true, and the file output.dat is opened for writing; first's value is then changed to .false. so that a subsequent call would only append the new values of the solution to the end of the file. Note that the save attribute in the type declaration of the local variable first guarantees that its value is retained across multiple invocation of the subroutine. At this point we can run the code and check if the output returns the expected answer.

The next step is to insert the code for the Diagnostic subroutine and validate its outcome. Only the following line need be inserted

```
print *, 'Energy= ', 0.5*(w(1)**2+w(2)**2)
```

Again executing the new code and checking its output will validate its correctness.

We are now ready to tackle the bulk of the Rk4 and rhs routines. It is clear that the intermediate values of the solution could be stored in a single temporary arrays that can be overwritten once the slopes for that stage have been computed. Moreover, we need 4 separate arrays (although it is possible to rewrite the algorithm to use fewer arrays) to store the intermediate slopes.

subroutine Rk4(w, time, dt, nd)
    implicit none
    integer, intent(in) :: nd
    real, intent(in) :: time, dt
    real, intent(inout) :: w(nd)

    real :: r1(nd), r2(nd), r3(nd), r4(nd) ! temp slopes
    real :: wt(nd) ! intermediate solution

    return
end subroutine Rk4
CHAPTER 11. CODING & NUMERICAL INTEGRATION OF ODES

```
call rhs(r1,w,time,nd) ! stage 1
wt = w + 0.5*dt*r1
call rhs(r2, wt, time+0.5*dt,nd) ! stage 2
wt = w + 0.5*dt*r2
call rhs(r3, wt, time+0.5*dt,nd) ! stage 3
wt = w + dt*r3
call rhs(r4, wt, time+dt, nd) ! stage 4
w = w + dt * ( r1 + 2.0*(r2+r3) + r4 ) / 6.0
return
end subroutine Rk4
```

Validating this code is relatively simple provided we give it a trivial forcing function. The solution should not change for example if the right hand side is zero. The right hand side subroutine for our case however would be as follows:

```
subroutine rhs(r,w,time,nd)
  implicit none
  integer, intent(in) :: nd
  real, intent(in) :: time, w(nd)
  real, intent(out) :: r(nd)
  real, parameter :: f = 1
  r(1) = f * w(2)
  r(2) =-f * w(1)
return
end subroutine rhs
```

Notice that we chose to declare the Coriolis parameter locally since it is not needed by any of the other routines.

### 11.8 Run Plotting and Validation

Once the bugs and compiler errors are corrected, the code can be run for a short run. Initial runs should proceed with optimization parameters off, and debuggers parameters on. These should include

1. `-g` to generate debug operation and inhibit optimization
2. `-C` to check array bounds at run-time. The code stops with an informational message if the array bounds are trespassed.
3. Turn floating point exception on to catch overflows, divide by zero, and other floating point exceptions. Underflow exception is not generally a serious problem and can be overlooked.
4. Trap uninitialized variables. Some compilers have the options to insert a nonsensical values in all variables declared in the program. The code crashes if a variable is used before its value is set by the programmer. Unfortunately it does not seem like the Portland Group compiler has this feature while the Intel "ifort" compiler does; It is -trapuv

5. Another common source of errors is the mismatch between the dummy argument list in a subroutine declaration and the calling statement. This can take the form of either wrong data type, wrong size, or wrong sequence. We will introduce an interface construct that help us catch these sort of problems at compile time.

Once the code has been validated for a small test run, the optimization can be removed since they tend to slow execution severely.

11.8.1 Plotting

The plotting in matlab is usually a breeze. Simply launch matlab and read in the 2 columns with the load matlab command. The two columns will contain the evolution of the solution caught at every isnap time steps.