LTRANS

Lagrangian **TRANS**port model (LTRANS) v.2

User's Guide

Authors:
Zachary R. Schlag
Elizabeth W. North

January 6, 2012

University of Maryland Center for Environmental Science
Horn Point Laboratory
Cambridge, Maryland 21613
USA

Developers

The Lagrangian TRANSport (LTRANS v.2) model is based upon LTRANS v.1 (formerly the Larval TRANSport Lagrangian model). Zachary Schlag completed signigicant updates to the code in LTRANS v.2 with input from Elizabeth North, Chris Sherwood, and Scott Peckham. LTRANS v.1 was built by Elizabeth North and Zachary Schlag of University of Maryland Center for Environmental Science Horn Point Laboratory.

User's Guide

This User's Guide is based on the v.1 User's Guide (Schlag et al. 2008) and was updated by Zachary Schlag and Elizabeth North.

Acknowledgements

For LTRANS v.2, we thank Christopher Sherwood, Scott Peckham and Rich Signell for their advice and guideance on the updates to the code, and for users like Cheryl Harrison who reported bugs that we had the opportunity to fix. For LTRANS v.1, we thank Thomas Gross, Charles Hannah, Raleigh Hood, Ming Li, Richard Signell, Uffe Thygesen, Liejun Zhong, the members of the International Council for the Exploration of the Sea (ICES) Working Group on Modelling Physical-Biological Interactions, and 2008 LTRANS Workshop participants for their helpful advice and discussions. Funding for LTRANS v.1 and v.2 was provided by the National Science Foundation Biological Oceanography Program (OCE-0424932, OCE-0453905, OCE-0829512), National Science Foundation Physical Oceanography Program (OCE-1048630), Maryland Department of Natural Resources (K00P4200981), NOAA Chesapeake Bay Studies (NA04NMF457038), and NOAA-funded UMCP Advanced Study Institute for the Environment (Z759502 NA06NES4280016).

Citation Information

Schlag, Z. R., and E. W. North. 2012. Lagrangian TRANSport (LTRANS) v.2 model User's Guide. Technical Report of the University of Maryland Center for Environmental Science Horn Point Laboratory. Cambridge, MD. 183 p.

Table of Contents

I. Overview

The **L**agrangian **TRANS**port model (LTRANS) is an off-line particle-tracking model that runs with the stored predictions of a 3D hydrodynamic model, specifically the Regional Ocean Modeling System (ROMS). LTRANS is intended to simulate passive particles, particles with sinking or floating behavior like sediment or oil droplets and planktonic organisms like oyster larvae. LTRANS is written in Fortran 90 and is designed to track the trajectories of particles in three dimensions. It includes a 4th order Runge-Kutta scheme for particle advection and a random displacement model for vertical turbulent particle motion. Reflective boundary conditions, particle behavior, and settlement routines are also included. LTRANS v.1 was built by Elizabeth North and Zachary Schlag of University of Maryland Center for Environmental Science Horn Point Laboratory. Modifications for LTRANS v.2 were undertaken by Zachary Schlag. Funding was provided by the National Science Foundation Biological Oceanography Program and Physical Oceanography Program, Maryland Department of Natural Resources, NOAA Chesapeake Bay Office, and NOAA-funded UMCP Advanced Study Institute for the Environment. Components of LTRANS have been in development since 2002 and are described in the following publications: North et al. (2005, 2006a, 2006b, 2008, 2011).

Model structure

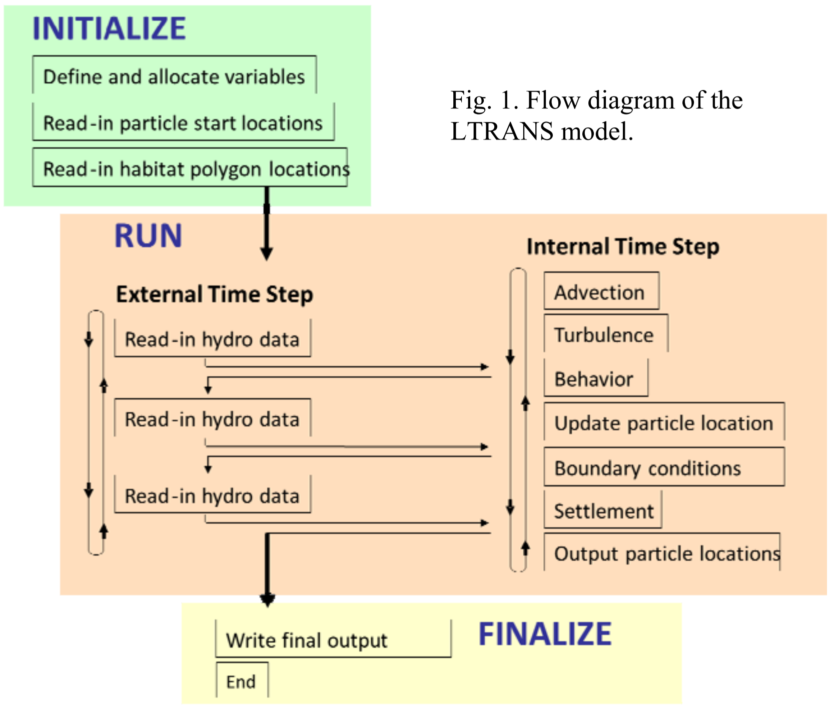

Fig. 1. Flow diagram of the LTRANS model.

LTRANS is designed to predict the movement of particles based on advection, turbulence and behavior. It has an external and internal time step (Fig. 1) and boundary condition algorithms that keep particles from leaving the model domain. The external time step is the time step of hydrodynamic model output (e.g., 10 min). The internal time step is the time interval during which particle movement is calculated (e.g., 120 s). The internal time step is smaller than the external time step so that particles do not move in large jumps that could cause inconsistencies between predictions of the hydrodynamic model and the particle tracking model. At each internal time step ofLTRANS, particle motion is calculated as the sum of movement due to advection, turbulence and behavior. The model contains sub-models for each of these components. The turbulence and behavior routines can be turned off so that particle movement is based solely on advection. LTRANS also includes sub-models for boundary conditions and particle settlement (i.e., target areas, habitats) as well as specially designed search algorithms that significantly increase the speed of model computations.

Interpolation scheme

Hydrodynamic model predictions (stored in NetCDF format) are read in and interpolated in space and time to the particle location. The first step in the process of interpolating the water properties (e.g., current velocities, salinity, temperature, sea surface height, and vertical and horizontal diffusivities) to the particle location is to determine the grid cell in which the particle is located. For this, we use the 'crossings' point-in-polygon approach coupled with a search algorithm for computational efficiency. Once the particle is located in a grid cell, water properties are interpolated in space to the particle location. All water properties are interpolated

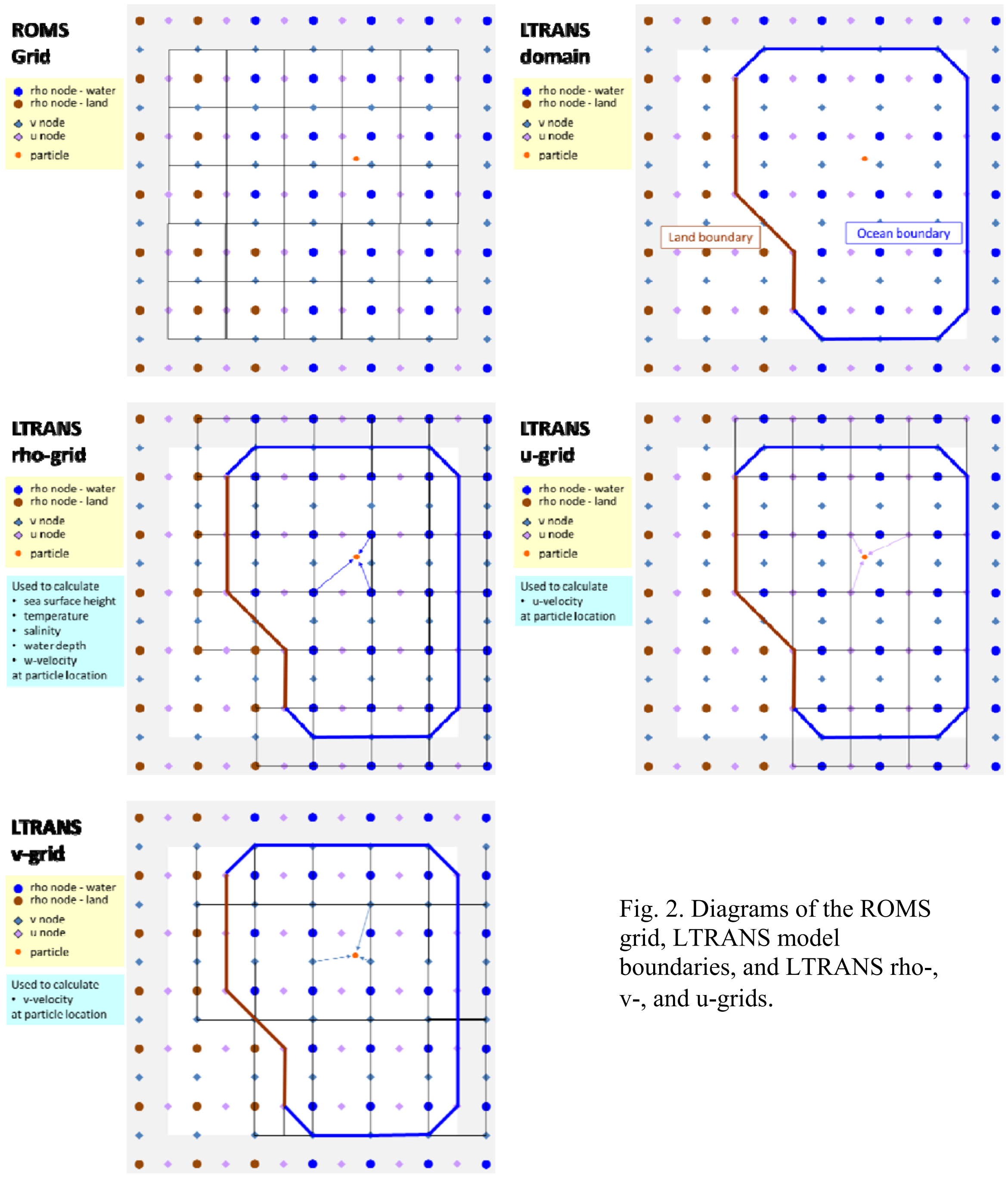

Fig. 2. Diagrams of the ROMS grid, LTRANS model boundaries, and LTRANS rho-, v-, and u-grids.

from the native ROMS grid points (i.e., u grid points are used to calculate *u*-velocity at the particle location, v grid points are used for *v*-velocity, and rho grid points are used for sea surface height, *w*-velocity, salinity, and diffusivity calculations) (Fig. 2). For two-dimensional water properties (e.g., sea surface height, water depth) bilinear interpolation is used. For three-dimensional water properties (e.g., current velocities, diffusivities, salinity), a water-column profile scheme is applied (North et al. 2006a). In this scheme, values are interpolated along each s-level to create a vertical profile of values at the x-y particle location (Fig. 3). A tension spline curve is then fit to the vertical profile and used to estimate the water property at the particle location. The interpolation scheme was adapted from North et al. (2006a), streamlined to increase computational speed, and enhanced to handle model domains with irregular bottoms and non-rectangular grid geometries. It should be noted that this interpolation scheme likely assumes that the underlying hydrodynamic model grid is orthogonal (Rich Signell, pers. comm.).

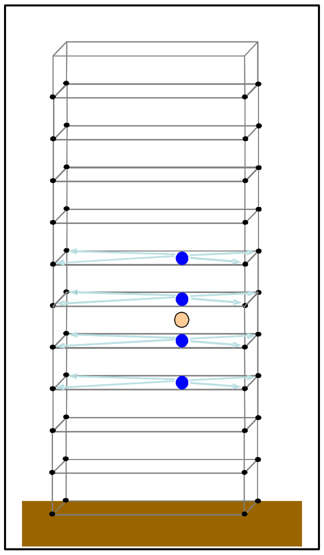
Fig. 3. Schematic of ROMS model grid and 'water column' interpolation scheme. Hydrodynamic model predictions are interpolated along s -levels to the x-y locations (blue circles) above and below a particle (orange circle). Then a tension spline is fit to the values at the x-y locations to determine the water property at the particle location.

Although there are several available methods for interpolating to the particle location (e.g., linear interpolation, cubic splines) we chose to use a sophisticated tension spline curve fitting routine. Both cubic and simple tension splines cause 'offshoots'. Offshoots occur when the interpolated line does not preserve the monotonicity and concavity of the original data. Offshoots can easily be seen with the cubic spline interpolation technique (Fig. 3). LTRANS was originally developed with the Tension Spline Curve Fitting Package (TSPACK). TSPACK (TOMS/716) was created by Robert J. Renka (renka@cs.unt.edu, Department of Computer Science and Engineering, University of North Texas) and is available for download from http://www. netlib.org and http://portal.acm.org/citation.cfm?id=151277. TSPACK fits tension splines to data that preserve the concavity and monotonicity of the data (Fig. 4). The routines in TSPACK are highly articulate and produce excellent profiles, although they are computationally demanding because an individual tension factor is estimated for each segment of the profile. The tests of the random displacement model for vertical sub-grid scale turbulence (North et al. 2006a) were undertaken with TSPACK. Occasionally, the curve fitting method would fail to converge. In the North et al. (2006a) simulations, this occurred 0.0004% of the time, or once in 244,500 calls to TSPACK. In these rare cases, simple linear interpolation of the vertical profile was used .

TSPACK is copyrighted by the Association for Computing Machinery (ACM). With the permission of Dr. Renka and ACM, TSPACK was modified for use in LTRANS by removing unused code and call variables and updating it to Fortran 90. The modified version of TSPACK is included in the LTRANS source code in the Tension Spline Module (tension_module.f90). If you would like to use LTRANS with the modified TSPACK software, please read and respect the ACM Software Copyright and License Agreement (http://www.acm.org/publications/policies/softwarecrnotice). For noncommercial use, ACM grants "a royalty-free, nonexclusive right to execute, copy, modify and distribute both the binary and source code solely for academic, research and other similar noncommercial uses" subject to the conditions noted in the license agreement. Note that if you plan commercial use of LTRANS with the modified TSPACK software, you must contact ACM at permissions@acm.org to arrange an appropriate license. It may require payment of a license fee for commercial use.

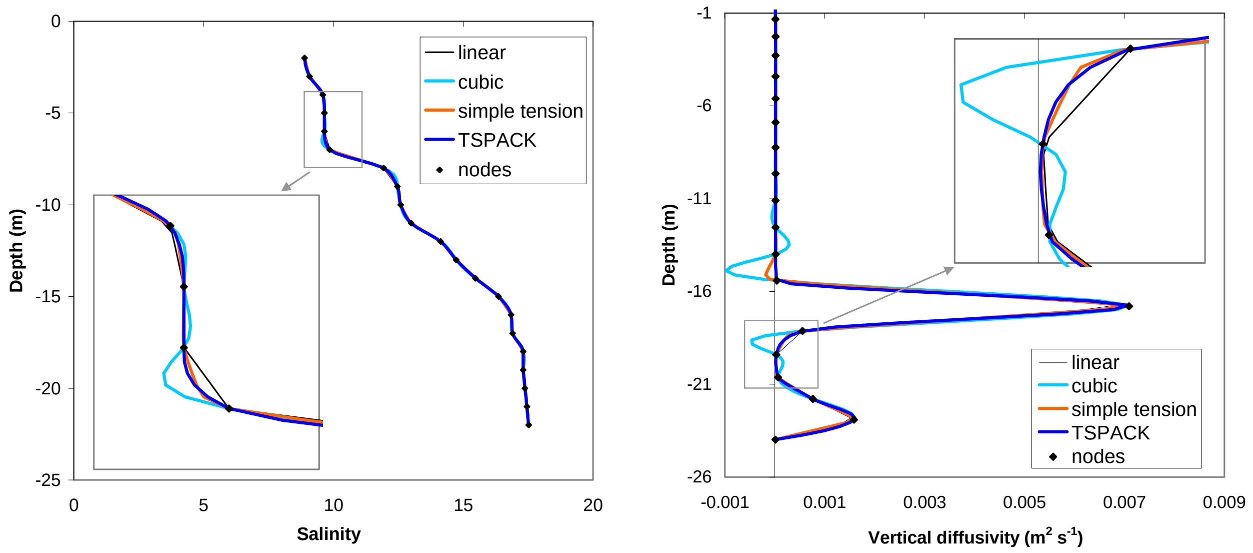

Fig. 4. Fit of linear, cubic spline, simple tension spline (tension factor = 10) and the TSPACK tension spline to profiles of salinity (left) and vertical diffusivity (right). The former is field data, the later is derived from ROMS. Note that linear interpolation does not preserve what one would expect to be a smooth profile. The cubic and simple tension splines create overshoots. These overshoots are especially problematic in the random displacement model (for vertical sub-grid scale turbulence) because they create artificial inflection points in the diffusivity profile which cause particles to move away from these points.

For particle tracking, it is necessary to interpolate in time as well as space because the duration between successive outputs of the hydrodynamic models (i.e., the external time step) is longer than the time step of particle motion (i.e., the internal time step). To do this, water properties are estimated at the particle location (as above) at three time points that correspond to the hydrodynamic model output (i.e., at the 10-min intervals of the external time step). Then a polynomial curve is fit to the water properties at three time points and used to calculate the water properties at the time of particle motion (i.e. for the internal time step). The advection, turbulence and behavior sub-models incorporate these spatial and temporal interpolation techniques; specifics associated with each sub-model are discussed below.

Advection sub-model. A 4th order Runge-Kutta scheme in space and time is used to calculate particle movement due to advection. This scheme solves for the *u*-, *v*-, and *w*- current velocities (representing the x-, y-, and z-directions) at the particle location using an iterative process that incorporates velocities at previous and future times to provide the most robust estimate of the trajectory of particle motion in water bodies with complex fronts and eddy fields (Dippner 2004) like Chesapeake Bay. Current velocities (m s^{-1}) provided by the Runge-Kutta scheme are multiplied by the duration of the internal time step (δt) to calculate the displacement of the particle in each component direction. Displacements (m) are then added to the original location of the particle (x_n, y_n, z_n) in order to calculate the new location of the particle ($x_{n+1}, y_{n+1},$

z_{n+1}):

(1) $x_{n+1} = x_n + u\delta t$

(2) $y_{n+1} = y_n + v\delta t$

(3) $z_{n+1} = z_n + w\delta t$

The *u* and *v* current velocities are separated into north and east component directions before particle motion is estimated. Law-of-the-wall (a log layer calculation) is applied to the current velocities within one s-level of bottom to simulate reduction in current velocities near bottom. A freeslip condition is applied near land boundaries.

Turbulence sub-model

Hydrodynamic models do not simulate turbulent motion at scales smaller than the grid resolution of the model. In particle-tracking models, particles can be moved in millimeter to centimeter steps -- much less than the hydrodynamic model grid scale. A random component must be added to particle motion in order to reproduce turbulent diffusion that occurs at the scale of particle motion (Hunter et al. 1993, Visser 1997, Brickman and Smith 2002). A random displacement model (Visser 1997) is implemented within the LTRANS to simulate sub-grid scale turbulent particle motion in the vertical (z) direction:

(4) $$z_{n+1} = z_n + K_v'\delta t + R\left[2r^{-1}K_v\delta t\right]^{1/2}$$

where z_n = initial particle location, K_v = vertical diffusivity evaluated at ($z_n + 0.5K_v'\delta t$), δt = time step of the random displacement model, $K_v' = \partial K_v/\partial z$ evaluated at z_n, and *R* is a random number generator with mean = 0 and standard deviation *r* = 1. Unlike random walk models, random displacement models do not result in numerical artifacts if the vertical resolution is adequate to resolve sharp variations in vertical diffusivity (Visser 1997; Brickman and Smith 2002). In LTRANS, the turbulent particle motion sub-model uses the same approach for determining K_v and K_v' at the particle location as that used in the advection model, except that 1) a smoothing algorithm is applied to the water column profile of K_v to prevent artificial aggregation of particles in regions of sharp gradients in diffusivity (North et al. 2006a), and 2) a 4th order Runge-Kutta was applied in time but not in space due to computational constraints.

A random walk model is used to simulate turbulent particle motion in the horizontal direction (x- or y- directions). When K_h is constant, the random displacement model defaults to a random walk model (Visser 1997):

(5) $$x_{n+1} = x_n + R\left[2r^{-1}K_h\delta t\right]^{1/2}$$

where K_h = horizontal diffusivity evaluated at (x_n). This was suitable for the ROMS model for which LTRANS was developed (Li et al. 2005, 2006, Zhong and Li 2006) because it was implemented with a constant value for K_h (1 m^2 s^{-1}). The model output was interpolated to the particle location using the same approach as was used for advection (described above), except that a 4th order Runge-Kutta was applied in time only (not space) due to the computational constraints. Note that it is likely that a random displacement model should be used if horizontal diffusivity is not constant in the hydrodynamic model.

Behavior sub-model

The behavior sub-model assigns vertical sinking, floating, or swimming velocities to particles. This velocity includes a speed component and an orientation (up or down) component that can depend upon particle characteristics such as species and developmental stage. The speed component controls the speed of particle motion due to sinking, floating or swimming. The speed can be set as constant or as a function of particle age. The orientation component regulates the direction of particle movement (i.e., sink, float, swim down in the presence of light, etc.). To simulate random variation in the movements of individual larvae, the direction of particle motion is assigned a random component that can be weighted so that particles have a tendency to move up or down depending on species and/or age of particle.

Settlement sub-model

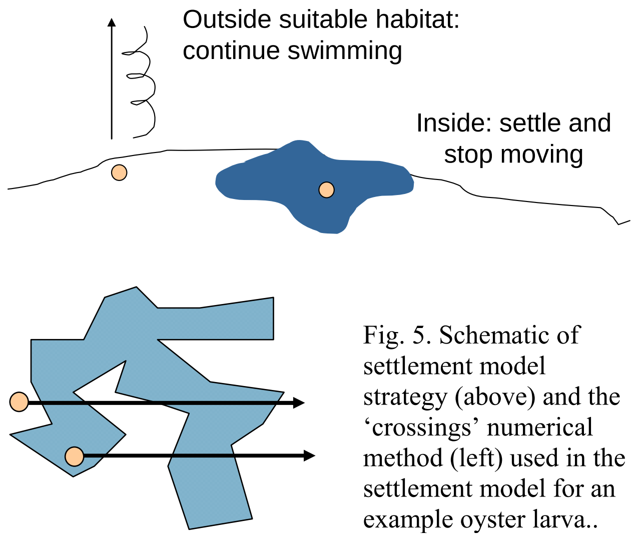

Fig. 5. Schematic of settlement model strategy (above) and the ‘crossings’ numerical method (left) used in the settlement model for an example oyster larva..

The purpose of the settlement sub-model is to determine if a particle is inside or outside an irregularly shaped polygon such as suitable habitat (e.g., marine reserve, seagrass bed, oyster reef). Once a particle reaches a specified age, the Settlement Module tests the location of settlement-stage particles at each internal time step (e.g., every 2 min) to determine if they are within the boundaries of a habitat polygon. If so, they stop moving (Fig. 5). To determine if the particle is inside or outside an irregularly shaped polygon, the ‘crossings method’, a ‘point-in-polygon’ technique, is applied. A ray, parallel to the x-coordinate axis, is shot from the particle (a point) to the east. The number of times the ray intersects with the line segments of each polygon is calculated. If the number of intersections is odd, then the particle is within the polygon. If the number is even, then the particle is outside the polygon boundaries. A search restriction algorithm ensures that the locations of particles are tested only for nearby polygons to reduce computation time.

Boundary conditions

Before particles settle or die (i.e., between the time they are released and the time they stop moving), the location of each particle is tested every internal time step to ensure that it remains within the model boundaries. If the motion of the particle causes it to exceed the boundaries, the particle is placed within the model domain as specified below.

Vertical boundaries (surface and bottom) are specified for each particle by interpolating sea surface height and bottom depth to the x-y location of the particle. If a particle passes through the surface or bottom boundary due to turbulence or vertical advection, the particle is placed back in the model domain at a distance that is equal to the distance that the particle exceeds the boundary (i.e., it is reflected vertically). If a particles passes through the surface or bottom due to particle behavior, the particle is placed just below the surface or above the bottom (i.e., it stops near the boundary).

Reflective horizontal boundary condition routines keep particles within the model domain. For ROMS, boundaries are taken to be halfway between water and land grid points. Boundary points of the main land/sea boundary and each individual island are ordered to create closed polygons. The 'crossings' point-in-polygon approach is used to determine if a particle is inside or outside the model boundaries. If the particle is on land or on an island, the particle is reflected off the boundary with an angle of reflection that equals the angle of approach to the boundary. The distance that the particle is reflected is equal to the distance that the particle exceeded the boundary. The horizontal boundary condition routine allows multiple reflections within one time step. At open ocean boundaries, the user may specify either reflection or 'sticking'. For 'sticking', if the particle intersects the open ocean boundary, it would stop moving at the boundary and remain there until the end of the simulation.

User's Guide

Our objective in writing this User's Guide is to provide the necessary information for users to 1) set up and run LTRANS v.2, and 2) be able modify LTRANS v.2 to adapt it to their needs. We have tried to define every variable in the model. If you search the document (Ctrl F) and cannot find the definition of a variable used in LTRANS, please report this to enorth@umces.edu and we will correct it. Your suggestions on how to make this document more useful also would be appreciated.

Updates to LTRANS v.2

Substantial revisions to the code were undertaken as part of the update to LTRANS v.2. Notable improvements include: organized into Initialize-Run-Finalize (IRF) format, rotating indices were added to speed up computation, limited read-in of hydrodynamic model information speeds up computations, spherical projection conversion was added, separate grid generator program and grid.data input file were eliminated, the ability to allow particle death without invoking settlement was implemented, horizontal swimming was added with the new tidal stream transport behavior, and multiple changes were made to increase robustness of input and output. Changes are listed here:

Global Changes
- replaced .inc files with .data files and .h file for a more dynamic model
- dynamic allocation of variables
- made real values double precision, and ensured double precision values were used in equations by using the DBLE() coercion function

LTRANS.f90
- new Initialize-Run-Finalize (IRF) format
- new input/output formats (including NetCDF)
- set advection to 0.0 for particles found below the roughness height (z0).

- **setEle** for all elements before initHydro, for read-in of hydrodynamic data in region of particles
- **polintd** to get values for current **Zpar** and **P_zeta**
- removed **idum_call_count**
- new code to handle behavior 7 (tidal stream transport)
- track collisions with the bottom or land boundaries
- added the ability to time processes
- added output of header information for .csv output files
- printing moved to internal time loop
- new logical variable **newPart** for new particles instead of testing for first iteration
- new subroutine **writeOutput** that handles output to csv or nc files

Behavior Module (behavior_module.f90)
- removed **status** variable
- added **dead** and **oob** logical variables to track if particle is dead or out of bounds
- added tidal stream transport (which includes horizontal swimming)
- new procedures **isDead** and **Die** to handle particle death
- new procedures **setOut** and **isOut** to handle out of bound particles
- new **finBehave** subroutine to deallocate variables and call **finSettlement** if necessary

Boundary Module (boundary_module.f90)
- distinguishes between land & water boundaries
- optional logical argument **isWater** was added to intersect_reflect to identify an open ocean boundary
- writes OpenOceanBoundaryMidpoints.csv file if **BoundaryBLNs** = TRUE so the open ocean boundaries are written to a file
- replaced '==' with .EQV. to check equivalence of logical arguments in **createBounds**
- in **iBounds** replaced if(i == **maxisland** .OR. **hid**(i+1) /= isle) with an if/else to ensure that (i == **maxisland**) is tested before (**hid**(i+1) /= **isle**) because if i == **maxisland**, then **h**(i+1) is out of bounds as h only goes up to i

Conversion Module (conversion_module.f90)
- New Spherical Projection equations added
- Old mercator projection is still available
- If **SphericalProjection** == TRUE, uses spherical, else uses mercator
- **RCF** is now calculated in conversion module using **PI** value from parameter module

Hydrodynamic Module (hydrodynamic_module.f90)
- output error messages from NF90_STRERROR
- added flexibility in finding **s_rho** or **sc_r**, and **s_w** or **sc_w**
- read in lat & long of grid, then convert
- new mask editing to handle areas not covered by u or v grid cells
- finding adjacent elements restricted for speed enhancement
- restricted (i,j) rho,u,v read-in (new **t_ijruv** variable)

- rotating indices
- multiply by mask to ensure land values have 0.0 value
- in **setEle**, optional argument debug, to produce output to Problems.txt if an error occurs
- new subroutine **setEle_all** to find initial elements for all the initial particles
- new s-level and w-level equations based on **Vtransform**. VTransform is a value in LTRANS.data set to 1, 2, or 3 to specify which equation to use to calculate the depths of the s-levels.
- new subroutine **setijruv** to find i,j bounds for restricted read-in
- new subroutine **finHydro**
- new subroutines **initNetCDF**, **createNetCDF**, & **writeNetCDF** to create NetCDF output
- removed **modanum** for nonsequential netcdf numbering

Norm Module (norm_module.f90)
- use **genrand_real3 (0,1)** instead of **genrand_real1 [0,1]** to suppress return of the value 0 which can result in NANs

Parameter Module (parameter_module.f90)
- new Subroutines **LTRANS_inpar**, **Error_Check**, and **gridData**
- read in LTRANS.data
- generate information on the model grid data (no longer read in from a separate file)
- subtract 1 from **lonmin** & **latmin** to eliminate potential roundoff error

Settlement Module (settlement_module.f90)
- changed **settle**(n) from an integer value to a logical
- moved particle death (including **DIE** and **DEAD** subroutines) to behavior module
- added **finSettlement** subroutine
- started to make input more flexible, but needs to be further improved
- in **CreatePolySpecs** replaced if(**count**.NE.0 .AND. **polys**(j,1).EQ.**polynums**(count)) cycle with nested ifs to ensure **count** /=0 before checking **polynums**(**count**)

Vertical Turbulence Module (ver_turb_module.f90)
- **p2** is now calculated as **ws***4 in the module, rather than read in

Modules with no notable changes:
- gridcell module (gridcell_module.f90)
- horizontal turbulence (hor_turb_module.f90) (*double precision changes)
- interpolation (interpolation_module.f90)
- random number module (random_module.f90) (***genrand_real3** made public)
- point_in_polygon (point_in_polygon_module.f90)
- tension (tension_module.for)

Concluding thoughts

The LTRANS model is designed to maintain fidelity with hydrodynamic model predictions. All interpolation occurs from the original staggered grid of the u, v, and rho grid points directly to the particle location. In addition, horizontal interpolation occurs along s-levels in an attempt to follow the structure of the hydrodynamic model in regions of changing bathymetry. These interpolation schemes may be costly in computation time compared to less accurate schemes; the benefits have not been quantified. The LTRANS model was developed to simulate oyster larvae in Chesapeake Bay, a region with complex bathymetry and horizontal and vertical current shears. It is not known whether the LTRANS interpolation schemes would be appropriate in other systems, and, if so, in what conditions they should be used. We invite the particle tracking community to participate in cross-system comparisons to help develop standardized methods for interpolation, turbulence and time stepping for different systems.

Open Source License

LTRANS v.2 is an open-source model and licensed under the MIT/X License. This license is similar to the ROMS model license. Here is a copy of the LTRANS v.2 model license file:

II. Setting up LTRANS in a new model domain

Overview. This section provides step-by-step instructions for setting up and running LTRANS v.2 in both the Windows and Linux environments. More details about the input file types and formats can be found in this User's Guide Input Files section (p. 27). Sample input files can be found at the "LTRANS Example Input and Output Files" section of the LTRANS website (http://northweb.hpl.umces.edu/LTRANS.htm). The 'release configuration' of LTRANS v.2 is designed to run with these example input files.

0. Note that two modules that are released with LTRANS were not created by LTRANS developers and have different license files. They are the Mersenne Twister and TSPACK programs found in the Random Number Module (Random_module.f90) and the Tension Spline Module (Tension_module.f90), respectively. Please review and respect the permissions of these programs. The information on these programs can found in the appropriate module sections of this User's Guide as well as on the "External Dependencies and Programs" section of the LTRANS web site (http://northweb.hpl.umces.edu/LTRANS.htm).

1. Install NetCDF Libraries

Because LTRANS reads in ROMS-generated NetCDF (.nc) files, LTRANS requires that the appropriate NetCDF libraries be installed on your computer. Linux users will likely have to build their own libraries using the source code/binaries on the Unidata website (http://www.unidata.ucar.edu/software/netcdf/). In addition, Linux users will need to determine if their version of NetCFD was compiled with HDF5 and curl, which would mean that LTRANS should be compiled with the options '-hdf5' and '-curl'.

In Windows Visual Fortran environment, the following pre-built binaries may be used. The enclosed pre-built NetCDF library files were downloaded from (see URL) and should be placed in (see path) the following locations on your computer:

http://www.unidata.ucar.edu/software/netcdf/binaries.html
netcdf.dll, place in C:\Program Files\Microsoft Visual Studio\DF98\BIN
netcdf.inc, place in C:\Program Files\Microsoft Visual Studio\DF98\INCLUDE
netcdf.lib, place in C:\Program Files\Microsoft Visual Studio\DF98\LIB.

http://www.unidata.ucar.edu/software/netcdf/docs/other-builds.html#windows_ifort_f90
netcdf90.lib, place in C:\Program Files\Microsoft Visual Studio\DF98\LIB
netcdf90.mod, place in C:\Program Files\Microsoft Visual Studio\DF98\INCLUDE
typesizes.f90, place in C:\Program Files\Microsoft Visual Studio\DF98\INCLUDE
typesizes.mod, place in C:\Program Files\Microsoft Visual Studio\DF98\INCLUDE

These files can be found in VF-NetCDF.zip file in the "External Dependencies and Programs" section of the LTRANS web site (http://northweb.hpl.umces.edu/LTRANS.htm). Note that the paths above reflect the default installation location of Microsoft Visual Studio; if you installed it in a different location, your path will need to be different. Also, note that the "netcdf.lib" file needs to be added to the LTRANS Visual Fortran project before building LTRANS.

2. Make sure the ROMS NetCDF files contain the appropriate variables. The LTRANS model uses hydrodynamic data from ROMS NetCDF files. It uses two types of files, a file that contains information about the model grid and the output files that contain the hydrodynamic model predictions. Often there are multiple sequential hydrodynamic output files. The following variables should be in the file that contains the ROMS model grid information:

Netcdf ID	Description
angle	angle between x-coordinate and true east direction
h	depths of rho nodes
lat_rho	latitude of rho nodes
lat_u	latitude of u nodes
lat_v	latitude of v nodes
lon_rho	longitude of rho nodes
lon_u	longitude of u nodes
lon_v	longitude of v nodes
mask_rho	rho node mask value
mask_u	u node mask value
mask_v	v node mask value

The following variables should be in the ROMS output files that contain the hydrodynamic model predictions. Note that the variables **Cs_r**, **Cs_w**, **s_rho**, and **s_w** must be in the first output file used by LTRANS. In old versions of ROMS the variables **s_rho** and **s_w** were called **sc_r** and **sc_w** respectively. If the model fails to find the variables **s_rho** or **s_w**, it will then look for **sc_r** or **sc_w**. The other variables should be in all of the output files used by LTRANS.

Netcdf ID	Description
Aks	vertical diffusivity of salinity at rho nodes
Cs_r	value used to adjust rho node depths
Cs_w	value used to adjust w node depths
salt	rho node salinity
s_rho	value used to convert s-levels to rho node depths
s_w	value used to convert s-levels to w node depths
temp	rho node temperature
u	u-direction velocity
v	v-direction velocity
w	w-direction velocity
zeta	zeta levels at rho nodes

3. Update path to ROMS NetCDF files
These ROMS NetCDF files can either be placed in the same directory as the program (this is the way the LTRANS.data file is currently configured) or placed in a separate folder. The names and location of the files should be updated in the LTRANS.data file using the following parameters: **NCgridfile** (for the grid file) and **prefix**, **filenum**, , **numdigits**, **suffix** for the first output file

(only the first output file need be specified). If the first output file has one additional time step, the variable **startfile** must be set to .TRUE., otherwise it should be set .FALSE.. The ROMS NetCDF files are generally large so you may choose to keep them in a separate folder. In this case, the path to the folder with the NetCDF files must be specified in the parameters **NCgridfile** (for the grid file) and **prefix** (for the ROMS output files) found in the LTRANS.data file. The length of **prefix** in the variable declaration section must remain greater than or equal to the length of the path stored in it. Also note that the length of variable **filenm** must remain greater than or equal to the length of the full file name. (These lengths are set in the 'LTRANS.h' file and are currently 100 or more characters long.)

4. Create particle locations file
The particle locations are read in from a .csv file which contains either four or five columns: longitude, latitude, depth (in meters), date of birth (delay in seconds from beginning of model until particle should be released (i.e., start moving)) and, if settlement is turned on, the identification number (id) of the habitat polygon the particle starts on. This file must have at least as many rows as the number of particles in the parameter **numpar**. All of the particle start locations should be within the model boundaries. See the Input Files section of the User's Guide (p. 27) for more information. Place the particle locations file in the same folder as the code and specify the filename in the LTRANS.data file using the **parfile** parameter. If you would like the file to be in a separate folder, add the file's path to the **parfile** parameter.

5. Update 'User specified' parameters and variables in LTRANS.data file to turn on or off turbulent particle motion, specify particle behavior (or lack thereof), and calculate salinity and temperature at the particle location, in addition to selecting other options. See User's Guide Include Data (Initialization) File section (p.20) for more information.

6. If you would like to use the Settlement Module:

- **a. Turn settlementon = .TRUE.** in LTRANS.data include file.
- **b. Make habitat location files:** In order for the model to run with settlement, it must read in habitat location data from .csv files. There are two types of habitat location files: habitat boundary files and habitat hole boundary files. See User's Guide Input section (p. 27) for more information. Place the habitat files in the same folder as the code and specify the file names in the LTRANS.data file using the **habitatfile** and **holefile** parameter. If you would like the file to be in a separate folder, add the file's path to the **habitatfile** and **holefile** parameters.
- **c. Update Settlement Module parameters** in LTRANS.data include file.

7. Compile and run LTRANS. This section includes instructions for compiling and running LTRANS in the Windows and Linux environments.

a. In the Linux environment:

First, create a directory and place the following in that directory: all of the LTRANS .f90 files, the makefile, the ROMS NetCDF grid and history files (unless an alternate path is specified in the LTRANS.data file), and the .csv input file for particle locations and for habitat (optional) (unless alternate paths are specified in the LTRANS.data file). The commands found below are called from inside this new directory. (For the LTRANS v.2 testcase, simply place all .f90, .data, .nc, and .csv files in the directory).

Next, compile and run LTRANS. A makefile has been provided in LTRANS.zip. This makefile is initially set up to compile the model using ifort on Linux with –O2 –fp-model precise flags and the NetCDF include and library files having been installed in /usr/local/include and /usr/local/lib. If using a different compiler, different flags, or NetCDF files in a different location, there is a "USER-DEFINED OPTIONS" section at the top of the file where these options may be easily changed. If using a different compiler, change the line "FC = ifort" to reflect the correct compiler. If the NetCDF files have been installed in a different location, then the values of NETCDF_INCDIR and NETCDF_LIBDIR will need to be altered so that they include the correct path to the NetCDF files. Also, if NetCDF was compiled with HDF5 set the value of HDF5 to on (HDF5 := on), otherwise leave its value blank (HDF5 :=). Lastly, set the value of FFLAGS to the desired compiler flags. The makefile should be in the same directory as the .f90 files. Call it using the command:

```
make
```

The makefile will compile all the modules and the main program into the executable file LTRANS.exe. The makefile simply carries out the steps detailed below with echo commands to give updates on its progress.

To compile the model without the makefile, begin by compiling the Fortran modules without linking. This will create .o and .mod files that are necessary to compile and link the whole program. The following commands will compile the modules without linking using the ifort Linux compiler:

```
ifort -c gridcell_module.f90
ifort -c interpolation_module.f90
ifort -c parameter_module.f90
ifort -c point_in_polygon_module.f90
ifort -c random_module.f90
ifort -c tension_module.for
ifort -c conversion_module.f90
ifort -c –I/usr/local/include hydrodynamic_module.f90
ifort -c norm_module.f90
ifort -c boundary_module.f90
ifort -c hor_turb_module.f90
ifort -c settlement_module.f90
ifort -c ver_turb_module.f90
ifort -c behavior_module.f90
```

If the NetCDF include files were installed in a directory other than /usr/local/include then the command to compile the Hydrodynamic Module will need to be modified to reflect the actual location of the files. It is also recommended to include the compiler flags –O2 and –fp-model precise. The flag –fp-model precise disables optimizations that can change the result of floating-point calculations. Although these flags ensure the accuracy of floating-point computations, they may slow performance.

Now the executable can be created using the .o files created in the previous step. The following command will compile and link the code and create the executable file LTRANS.exe (to give the executable file a different name, replace 'LTRANS.exe' with the desired name):

```
   ifort -o LTRANS.exe LTRANS.f90 gridcell_module.o interpolation_module.o
parameter_module.o point_in_polygon_module.o random_module.o
tension_module.o conversion_module.o hydrodynamic_module.o norm_module.o
boundary_module.o hor_turb_module.o settlement_module.o ver_turb_module.o
behavior_module.o -L/usr/local/lib -lnetcdf
```

If the NetCDF library files were installed in a directory other than /usr/local/lib, then this command will need to be modified to reflect the actual location of the files. If NetCDF was compiled with HDF5 and curl, the following libraries must be added to the end of the command after –lnetcdf: -lhdf5_hl -lhdf5 –lcurl. Again, it is recommended that the –O2 and –fp-model precise flags be used to preserve floating-point accuracy.

Now that the executable has been created, the program can be run by simply calling the executable. If 'LTRANS.exe' is the executable name then the command to call the executable looks like this:

```
./LTRANS.exe
```

b. In the Visual Fortran (for Windows) environment:

A. Change reference from 'netcdf' to 'netcdf90' in the LTRANS Hydrodynamic Module and Parameter Module. To use LTRANS in Visual Fortran, a small change to the code in the Hydrodynamic and Parameter Modules needs to be made. The line "USE netcdf" will need to be changed to "USE netcdf90" in the three Hydrodynamic Module subroutines **initGrid**, **initHydro**, **updateHydro**, **createNetCDF**, and **writeNetCDF**, as well as the Parameter Module subroutine **gridData**.

B. Create a Visual Fortran project:
   1) Start up Visual Fortran
   2) Click on File -> New (or use shortcut Ctrl + N):
      i. Select 'Fortran Console Application'
      ii. Type in the desired project name into the 'Project name:' box
      iii. Select the location you want the project in the 'Location:' box
      iv. Click 'OK' button. This creates a project folder in the specified location that has the same name as the project.

v. In the subsequent dialogue window, ensure that ‘An empty project’ is selected, and click the ‘Finish’ button

vi. In the ‘New Project Information’ window that pops up, click ‘OK’

C. Add all of the .f90 files found in the “LTRANS” folder, as well as the NetCDF library file (netcdf.lib), to the project (Project -> Add To Project -> Files).

D. Compile the source files in the following stages:

1) Stage 1:
   - i. gridcell_module.f90
   - ii. interpolation_module.f90
   - iii. random_module.f90
   - iv. parameter_module.f90
   - v. point_in_polygon_module.f90
   - vi. tension_module.for
2) Stage 2:
   - i. conversion_module.f90
   - ii. norm_module.f90
   - iii. hydrodynamic_module.f90
3) Stage 3:
   - i. boundary_module.f90
   - ii. hor_turb_module.f90
   - iii. settlement_module.f90
   - iv. ver_turb_module.f90
4) Stage 4:
   - i. behavior_module.f90
5) Stage 5:
   - i. LTRANS.f90

E. Link (‘build’) the project

F. Make sure the ROMS model grid and output NetCDF output files, and the LTRANS input .csv files (particle locations, habitat and hole files) are located in the project folder (unless alternate paths are specified in the LTRANS.data file).

G. Run the program

8. Check to make sure LTRANS is running correctly. The following is written to the screen when LTRANS compiles and runs successfully. It is a good idea to check that the initial particle and habitat polygon latitude and longitude values are read in correctly (otherwise multiple errors can occur).

```
 ******************** Model Info ********************

Run Name:               = LTRANS v.2 test case
Executable Directory:   = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/
Output Directory:       = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/
Run By:                 = Elizabeth North
Institution:            = UMCES: HPL
Started On:             = 6 Jan 2012

Days:                   = 4.500
Particles:              = 608
Particle File:          = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/Initial_particle_locations.csv
```

```
  Behavior:              = C.virginica oyster larvae
  Particle Mortality:    = On
  Settlement:            = On
  Habitat File:          = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/End_polygons.csv
  Hole File:             = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/End_holes.csv

  Horizontal Turbulence: = On
  Vertical Turbulence:   = On
  Projection:            = Spherical
  Ocean Boundary:        = Open
  Salt & Temp Output:    = Off
  Track Collisions:      = No
  Track Model Timing:    = Yes

  Grid File:             = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/baymouth_grid_macroms.nc


  First Hydro File:      = /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/clipped_macroms_his_0003.nc

  Seed:                  = 9


  *************** LTRANS INITIALIZATION **************
 read in particle locations         608
   Particle n=5 Latitude=   37.2761608977490      Longitude=
  -76.2108564098010
   Particle n=5 Depth= -0.250000000000000
   Particle n=5 X=   5118201.07743135      Y=   921278.227723400
   Particle n=5 Start Polygon=      101001
 read-in grid information
 create elements
 find adjacent elements
  - rho
  - u
  - v
 prepare boundary arrays
 output lat/long blanking file
 output metric blanking file
 initialize behavior
 read in habitat polygon locations
   Edge i=5 Polygon ID=   101001.000000000
   Edge i=5 Center Lat=   37.1290000000000      Long=  -76.2400000000000
   Edge i=5   Edge Lat=   37.0610000000000      Long=  -76.2530000000000
   Hole i=5 Center Lat=   37.1270000000000      Long=  -76.0270000000000
   Hole i=5   Edge Lat=   37.1570000000000      Long=  -76.0440000000000
 find polygons in elements
 finding each particle's initial element
 /share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/clipped_macroms_his_0003.nc
Time to initialize model =  0.204 seconds.

****** BEGIN ITERATIONS *******
 write output to file, day =   4.166666666666666E-002
 write output to file, day =   8.333333333333333E-002
 existing matrix,stepf=           4
 write output to file, day =   0.125000000000000
 existing matrix,stepf=           5
 write output to file, day =   0.166666666666667
 existing matrix,stepf=           6
 write output to file, day =   0.208333333333333
 existing matrix,stepf=           7
 write output to file, day =   0.250000000000000
 existing matrix,stepf=           8
 write output to file, day =   0.291666666666667
 existing matrix,stepf=           9
 write output to file, day =   0.333333333333333
 existing matrix,stepf=          10
 write output to file, day =   0.375000000000000
 existing matrix,stepf=          11
.
```

```
 .
 .
 .
 .
 write output to file, day =    4.29166666666667
 existing matrix,stepf=          33
 write output to file, day =    4.33333333333333
 existing matrix,stepf=          34
 write output to file, day =    4.37500000000000
 existing matrix,stepf=          35
 write output to file, day =    4.41666666666667
 existing matrix,stepf=          36
 write output to file, day =    4.45833333333333
 existing matrix,stepf=          37
 write output to file, day =    4.50000000000000
 write endfile.csv
Time to run model =  4 minutes and 58.983 seconds.

****** END LTRANS *******
```

If you run LTRANS with the sample input files that can be found at the LTRANS website (http://northweb.hpl.umces.edu/LTRANS.htm), you could compare the endfile.csv output file that you generate with the one posted on the website. If you would like to help us determine the uniformity of LTRANS calculations across platforms, please send us your endfile.csv output file along with information about your platform (computer, operating system, compiler) to enorth@umces.edu. We would appreciate it.

III. Include Data File (Initialization)

Overview: The include file, LTRANS.data, contains the parameters that are used to adapt LTRANS to different ROMS hydrodynamic model domains, change particle attributes (e.g., turn on/off behavior and turbulence), and set input/output file paths. All initialization variables are placed in this file so that the code does not need to be modified to run LTRANS in different model domains or with different particle characteristics. Everything that the user may need to change can be found in LTRANS.data.

LTRANS.data

The variables in the LTRANS.data include file need to be changed manually. The definition of each parameter is specified within the file. Instructions for updating the parameters are also included in the file where appropriate. Below is the text of LTRANS.data file that is included with the LTRANS v2 release configuration. For more information on the parameters, see the module sections of this Users Guide.

```
! ******************************* LTRANS Include Data File *******************************

!----  This is the file that contains input values for LTRANS with parameters grouped ---
!----  (Previously LTRANS.inc)


!*** NUMBER OF PARTICLES ***
$numparticles

  numpar = 608               ! Number of particles (total number for whole simulation)
                             ! numpar should equal the number of rows in the particle
                             ! locations input file
$end



!*** TIME PARAMETERS ***
$timeparam

  days   = 4.5               ! Number of days to run the model
  iprint = 3600              ! Print interval for LTRANS output (s); 3600 = every hour
  dt     = 3600              ! External time step (duration between hydro model predictions) (s)
  idt    = 120               ! Internal (particle tracking) time step (s)

$end



!*** ROMS HYDRODYNAMIC MODULE PARAMETERS ***
$hydroparam

  us         = 20            ! Number of Rho grid s-levels in ROMS hydro model
  ws         = 21            ! Number of W grid s-levels in ROMS hydro model
  tdim       = 71            ! Number of time steps per ROMS hydro predictions file
  hc         = 0.2           ! Min Depth - used in ROMS S-level transformations
  z0         = 0.0005        ! ROMS bottom roughness parameter (Zob)
  Vtransform = 1             ! 1-WikiRoms Eq. 1, 2-WikiRoms Eq. 2, 3-Song/Haidvogel 1994 Eq.
                             !
  readZeta   = .TRUE.        ! If .TRUE. read in sea-surface height   (zeta) from NetCDF file, else use constZeta
  constZeta  = 0.0           ! Constant value for Zeta if readZeta is .FALSE.
  readSalt   = .TRUE.        ! If .TRUE. read in salinity             (salt) from NetCDF file, else use constSalt
  constSalt  = 0.0           ! Constant value for Salt if readSalt is .FALSE.
  readTemp   = .TRUE.        ! If .TRUE. read in temperature          (temp) from NetCDF file, else use constTemp
  constTemp  = 0.0           ! Constant value for Temp if readTemp is .FALSE.
  readU      = .TRUE.        ! If .TRUE. read in u-momentum component (U   ) from NetCDF file, else use constU
  constU     = 0.0           ! Constant value for U if readU is .FALSE.
  readV      = .TRUE.        ! If .TRUE. read in v-momentum component (V   ) from NetCDF file, else use constV
  constV     = 0.0           ! Constant value for V if readV is .FALSE.
  readW      = .TRUE.        ! If .TRUE. read in w-momentum component (W   ) from NetCDF file, else use constW
  constW     = 0.0           ! Constant value for W if readW is .FALSE.
  readAks    = .TRUE.        ! If .TRUE. read in salinity vertical diffusion coefficient (Aks ) from NetCDF file, else
                             ! use constAks
  constAks   = 0.0           ! Constant value for Aks if readAks is .FALSE.

$end



!*** TURBULENCE MODULE PARAMETERS ***
$turbparam

  HTurbOn       = .TRUE.     ! Horizontal Turbulence on (.TRUE.) or off (.FALSE.)
  VTurbOn       = .TRUE.     ! Vertical   Turbulence on (.TRUE.) or off (.FALSE.)
  ConstantHTurb = 1.0        ! Constant value of horizontal turbulence (m2/s)

$end



!*** BEHAVIOR MODULE PARAMETERS ***
$behavparam

  Behavior = 4               ! Behavior type (specify a number)
                             !   Note: The behavior types numbers are:
                             !     0 Passive, 1 near-surface, 2 near-bottom, 3 DVM,
                             !     4 C.virginica oyster larvae, 5 C.ariakensis oyster larvae,
                             !     6 constant, 7 Tidal Stream Transport
  OpenOceanBoundary = .TRUE. ! Note: If you want to allow particles to "escape" via open ocean
                             !   boundaries, set this to TRUE; Escape means that the particle
                             !   will stick to the boundary and stop moving
  mortality = .TRUE.         ! TRUE if particles can die; else FALSE
  deadage = 367200           ! Age at which a particle stops moving (i.e., dies) (s)
                             !   Note: deadage stops particle motion for all behavior types
  pediage = 302400           ! Age when particle reaches max swim speed and can settle (s)
                                ! Note: for oyster larvae behavior types (4 & 5),
                             !     pediage = age at which a particle becomes a pediveliger
                             !   Note: pediage does not cause particles to settle if
                             !     the Settlement module is not on
  swimstart = 0.0            ! Age that swimming or sinking begins (s) 1 day = 1.*24.*3600.
  swimslow  = 0.005          ! Swimming speed when particle begins to swim (m/s)
  swimfast  = 0.005          ! Maximum swimming speed (m/s)  0.05 m/s for 5 mm/s
                             !   Note: for constant swimming speed for behavior types 1,2 & 3,
                             !     set swimslow = swimfast = constant speed
  Sgradient = 1.0            ! Salinity gradient threshold that cues larval behavior (psu/m)
                             !   Note: This parameter is only used if Behavior = 4 or 5.
  sink      = -0.0003        ! Sinking velocity for behavior type 6
                             !   Note: This parameter is only used if Behavior = 6.
! Tidal Stream Transport behavior type:
  Hswimspeed = 0.9           ! Horizontal swimming speed (m/s)
  Swimdepth  = 2             ! Depth at which fish swims during flood time
                             ! in meters above bottom (this should be a positive value
                             ! Note: this formulation may need some work

$end



!*** DVM. The following are parameters for the Diurnal Vertical Migration (DVM) behavior type ***
!  Note: These values were calculated for September 1 at the latitude of 37.0 (Chesapeake Bay mouth)
!  Note: Variables marked with ** were calculated with light_v2BlueCrab.f (not included in LTRANS yet)
!  Note: These parameters are only used if Behavior = 3
$dvmparam

  twistart  = 4.801821       ! Time of twilight start (hr) **
  twiend    = 19.19956       ! Time of twilight end (hr) **
  daylength = 14.39774       ! Length of day (hr) **
  Em        = 1814.328       ! Irradiance at solar noon (microE m^-2 s^-1) **
  Kd        = 1.07           ! Vertical attenuation coefficient
  thresh    = 0.0166         ! Light threshold that cues behavior (microE m^-2 s^-1)

$end



!*** SETTLEMENT MODULE PARAMETERS ***
$settleparam

 settlementon = .TRUE.       ! settlement module on (.TRUE.) or off (.FALSE.)
                             ! Note: If settlement is off: set minholeid, maxholeid, minpolyid,
                             !   maxpolyid, pedges, & hedges to 1
                             !   to avoid both wasted variable space and errors due to arrays of size 0.
                             ! If settlement is on and there are no holes: set minholeid,
                             !   maxholeid, and hedges to 1
 holesExist = .TRUE.         ! Are there holes in habitat? yes(TRUE) no(FALSE)
 minpolyid  = 101001         ! Lowest habitat polygon id number
 maxpolyid  = 101004         ! Highest habitat polygon id number
 minholeid  = 100201         ! Lowest hole id number
 maxholeid  = 100401         ! Highest hole id number
 pedges     = 67             ! Number of habitat polygon edge points (# of rows in habitat polygon file)
 hedges     = 32             ! Number of hole edge points (number of rows in holes file)

$end



!*** CONVERSION MODULE PARAMETERS ***
```

```
$convparam

  PI = 3.14159265358979      ! Pi
  Earth_Radius = 6378000     ! Equatorial radius of Earth (m)
  SphericalProjection = .TRUE.  ! Spherical Projection from ROMS if TRUE. If FALSE, mercator projection is used.
  latmin = 30                ! Minimum longitude value, only used if SphericalProjection is .TRUE.
  lonmin = -134              ! Minimum  latitude value, only used if SphericalProjection is .TRUE.
$end



!*** INPUT FILE NAME AND LOCATION PARAMETERS ***;
! ** ROMS NetCDF Model Grid file **
  !Note: the path to the file is necessary if the file is not in the same folder as the code
  !Note: if .nc file in separate folder in Linux, then include path. For example:
  !      NCgridfile = '/share/enorth/CPB_GRID_wUV.nc'
  !Note: if .nc file in separate folder in Windows, then include path. For example:
  !      NCgridfile = 'D:\ROMS\CPB_GRID_wUV.nc'
$romsgrid
  NCgridfile='/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/baymouth_grid_macroms.nc'

$end


! ** ROMS Predictions NetCDF Input (History) File **
  !Filename = prefix + filenum + suffix
  !Note: the path to the file is necessary if the file is not in the same folder as the code
  !Note: if .nc file in separate folder in Windows, then include path in prefix. For example:
  !      prefix='D:\ROMS\y95hdr_'
  !      if .nc file in separate folder in Linux, then include path in prefix. For example:
  !      prefix='/share/lzhong/1995/y95hdr_'
$romsoutput
  prefix='/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/clipped_macroms_his_'   ! NetCDF Input Filename prefix
  suffix='.nc'               ! NetCDF Input Filename suffix
  filenum = 3                ! Number in first NetCDF input filename
  numdigits = 4              ! Number of digits in number portion of file name (with leading zeros)
  startfile = .TRUE.         ! Is it the first file, i.e. does the file have an additional time step?
$end


! ** Particle Location Input File **
  !Note: the path to the file is necessary if the file is not in the same folder as the code
$parloc

  parfile  = '/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/Initial_particle_locations.csv'    !Particle locations

$end
```

```
! ** Habitat Polygon Location Input Files **
!Note: the path to the file is necessary if the file is not in the same folder as the code
$habpolyloc

  habitatfile = '/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/End_polygons.csv'  !Habitat polygons
  holefile    = '/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/End_holes.csv'     !Holes in habitat polygons

$end


! ** Output Related Variables **
$output

  !NOTE: Full path must already exist.  Model can create files, but not directories.
  outpath = './output/'      ! Location to write output .csv and/or .nc files
                             ! Use outpath = './' to write in same folder as the executable
  NCOutFile = 'output'       ! Name of the NetCDF output files (do not include .nc)
  outpathGiven = .FALSE.     ! If TRUE files are written to the path given in outpath
  writeCSV     = .TRUE.      ! If TRUE write CSV output files
  writeNC      = .FALSE.     ! If TRUE write .NC output files
  NCtime       = 0           ! Time interval between creation of new NetCDF output files (seconds)
                             ! Note: setting this to 0 will result in just one large output file

  !NetCDF Model Metadata:
  SVN_Version  = 'https://svn1.hosted-projects.com/cmgsoft/LTRANS/trunk  Version: 39'
  RunName      = 'LTRANS v.2 test case'
  ExeDir       = '/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/'
  OutDir       = '/share/enorth/LTRANS_v2_testing/LTRANS_v2_FINAL/'
  RunBy        = 'Elizabeth North'
  Institution  = 'UMCES: HPL'
  StartedOn    = '6 Jan 2012'
$end



!*** OTHER PARAMETERS ***
$other

  seed         = 9           ! Seed value for random number generator (Mersenne Twister)
  ErrorFlag    = 0           ! What to do if an error is encountered: 0=stop, 1=return particle to previous location,
                             ! 2=kill particle & stop tracking that particle, 3=set particle out of bounds &
                             ! stop tracking that particle
                             ! Note: Options 1-3 will output information to ErrorLog.txt
                             ! Note: This is only for particles that travel out of bounds illegally
  BoundaryBLNs = .TRUE.      ! Create Surfer Blanking Files of boundaries? .TRUE.=yes, .FALSE.=no
  SaltTempOn   = .FALSE.     ! Calculate salinity and temperature at particle
                          ! location: yes (.TRUE.) or no (.FALSE.)
  TrackCollisions  = .FALSE. ! Write Bottom and Land Hit Files? .TRUE.=yes, .FALSE.=no
  WriteHeaders     = .TRUE.  ! Write .txt files with column headers? .TRUE.=yes, .FALSE.=no
  WriteModelTiming = .TRUE.  ! Write .csv file with model timing data? .TRUE.=yes, .FALSE.=no

  ijbuff = 4                 ! number of extra elements to read in on every side of the particles

$end
```

IV. Input Files

This section includes information on the input files needed to run LTRANS: 1) the NetCDF files from the ROMS hydrodynamic model, 2) a comma delimited file that contains the particle locations, and 3) comma delimited files that contain habitat boundaries for the Settlement Module. The latter is only needed if the Settlement Module is turned on.

A. ROMS NetCDF files

Overview: The LTRANS model uses hydrodynamic data from ROMS NetCDF files. It uses two types of files, a file that contains information about the model grid, and the output files that contain the hydrodynamic model predictions (history files). If the model predictions files also contain grid information, they can be used as the model grid file. Often there are multiple sequential output files that contain hydrodynamic model predictions. LTRANS assumes that the sequential ROMS output files contain the same number of time steps in each file (e.g., if the first file contains predictions at 144 discrete times, then all files should contain predictions at 144 discrete times). There is one exception to this. Sometimes the first hydrodynamic model predictions file generated by a ROMS run contains one additional time step (e.g., the first file has 145 discrete times, the remaining files have 144). If LTRANSbegins with this file then the variable **startfile** in LTRANS.data must be set to .TRUE., otherwise it must be set .FALSE. In this situation the variable **tdim** in LTRANS.data should be set to the number of time steps in the subsequent hydrodynamic files, i.e. one less than the number of time steps in the first file.

The following variables should be in the file that contains the ROMS model grid information:

Netcdf ID	**Description**
angle	angle between x-coordinate and true east direction
h	depths of rho nodes
lat_rho	latitude of rho nodes
lat_u	latitude of u nodes
lat_v	latitude of v nodes
lon_rho	longitude of rho nodes
lon_u	longitude of u nodes
lon_v	longitude of v nodes
mask_rho	rho node mask value
mask_u	u node mask value
mask_v	v node mask value

The following variables should be in the sequential ROMS files that contain the hydrodynamic model predictions. Note that the variables **Cs_r**, **Cs_w**, **s_rho**, and **s_w** must be in the first ROMS predictions file used by LTRANS. The other variables should be all of the files.

Netcdf ID	**Description**
Aks	vertical diffusivity of salinity at rho nodes
Cs_r	value used to adjust rho node depths
Cs_w	value used to adjust w node depths
salt	rho node salinity

s_rho	value used to convert s-levels to rho node depths
s_w	value used to convert s-levels to w node depths
temp	rho node temperature
u	u-direction velocity
v	v-direction velocity
w	w-direction velocity
zeta	zeta levels at rho nodes

There are three sections in which the ROMS NetCDF files are read in to LTRANS v.2. Information from the ROMS grid file is read in in subroutine **initGrid** in the Hydrodynamic Module and in **gridData** in the Parameter Module. The call to **initGrid** is located near the beginning of LTRANS.f90. The data read in includes the x and y coordinates of the nodes in the rho, u, and v grids, depth at the rho nodes, the angle between x-coordinate and true east, masks of the rho, u, and v grid nodes that specify whether the nodes are on land or in water, and the variables necessary to calculate s-levels: **SC**, **CS**, **SCW**, and **CSW**. This data is read in once and does not change.

The final section in which NetCDF files are read into the program occurs when information is read in from sequential output files of ROMS model predictions. This is done at the beginning of the external time step in LTRANS.f90 by calling the subroutines **initHydro** or **updateHydro** found in the Hydrodynamic Module. The current version of LTRANS uses files that contain 1 day of ROMS model output. When the program reaches a new day, it opens that day's NetCDF file and reads in the needed data. This data includes U, V, and W velocities, salinity, temperature, zeta, and vertical diffusivity. LTRANS stores in memory data needed for three external time steps (not the whole day's worth of data) to avoid overloading the computer's memory.

Input File: A single input file is used that contains the ROMS model grid data, and sequential input files are used that contain ROMS model predictions. This is the same grid file used by ROMS to specify the ROMS model grid. Again, it is possible that the grid information may have been written into the model predictions (history) files, in which case a single history file can be specified instead of a grid file in LTRANS.data.

Initialization: In order to run the model with NetCDF input, NetCDF libraries must exist on the computer on which LTRANS is compiled. Also, before linking the program, the file "netcdf.lib" should be added to the project (if compiling using Windows Visual Fortran). Finally, the correct name of the ROMS NetCDF files must be specified within the LTRANS.data include file so the appropriate data can be accessed. If these files are not located in the source code folder then the correct path to the files must be specified.

Numerical Method: Before data can be read in from a NetCDF file, the file must be opened by calling the function NF90_OPEN. For example, a NetCDF file might be opened with the line "**STATUS** = NF90_OPEN(filename, NF90_NOWRITE, **NCID**)", where "filename" can be a hard-coded filename such as "CPB_GRID_wUV.nc" or a character array containing the file name. The advantage of the character array, as seen in this program, is that the array can be altered and reused again in a loop, while hard-coding is not as flexible. "NF90_NOWRITE" in the above NF90_OPEN statement is a flag indicating that the file will be open for reading but not for writing. **NCID** is the returned NetCDF ID used in following statements in order to retrieve the data within the file. The function returns an integer that stands for a particular status (whether it succeeded, failed, etc.) and that value is stored in the variable **STATUS** to be tested to see if opening the file occurred without error. The line following the open statement should have "if (STATUS .NE. NF90_NOERR)" to test if there was an error, followed by an appropriate action such as writing "Problem NF90_OPEN" to output as is done in LTRANS.

The variable "**filenm"** is a character array that contains the name of the file that is to be opened. It is pieced together from other character arrays as well as integers (**filenm** = **prefix** + **counter** + **suffix)**. With the ROMS predictions files used to create LTRANS, **prefix** = "y95hdr_", **suffix** = ".nc", and **counter** was used to increment the name of sequential input files by one day (each file contains one day of ROMS model predictions). Note that the prefix should also have the path to the file if the file is not located in the same directory as the code. The **counter** in the middle of the file name is created by adding **iint**, the current day of the model (0 for the first day of the model, 1 for the second, etc.), to the day of the year on which the model starts. Therefore, if the model is on the third day of a run that starts on the 174th day of the year, the day of the year will be calculated as 174 + 2 = 176. This value is stored in **counter**. Then **prefix**, **counter**, and **suffix** are all written to the character array **filenm** which is used to open the appropriate NetCDF file. This allows the program simply to increment **iint**, recalculate **counter**, and remake **filenm** without excessive code, making it superior to hard-coding.

Once the file is open, the program must read the data from it. There are many functions that can be used to read a NetCDF file. This program uses two: NF90_INQ_VARID and NF90_GET_VAR. The function NF90_INQ_VARID is used to get the variable ID of a certain variable within the NetCDF file. This requires the exact name of the variable in the file. If this is not known, there are other functions that can help you find it. Additional functions and NetCDF information can be found at the links at the end of this section. Because the ROMS variables names are known, we use NF90_INQ_VARID. The form of the function is "STATUS = NF90_INQ_VARID(**NCID**, 'varname', **VID**)", where **STATUS** serves the same purpose as in the open function, **NCID** is the NetCDF ID returned from the open function, 'varname' is the specific variable name the program is looking for, and **VID** is the variable ID returned from the function.

Now that the program knows the NetCDF ID (**NCID**) and the variable ID (**VID**), it can get the data for that specific variable in that particular NetCDF file. This is done using the NF90_GET_VAR function. There are several different formats in which different variables can be passed to this function, changing how the output is returned. In LTRANS, we use two different formats. The first format is "**STATUS** = NF90_GET_VAR (**NCID**, **VID**, Var)", where **STATUS** once again serves the same purpose, **NCID** is the NetCDF ID, **VID** is the variable ID, and Var is the variable into which the data is being read. This only works properly if the variable has the same dimensions as the data. After NF90_GET_VAR is called, the variable **STATUS** is tested again to ensure that the data has been read in properly.

The format above is only useful for reading in an entire array from a NetCDF file. To read in only part of an array the second format is used. The second format of a call to this function used in LTRANS is "**STATUS** = NF90_GET_VAR (**NCID**, **VID**, Var, START, COUNT)", where

STATUS is used to check that the function worked properly, **NCID** is the NetCDF ID, **VID** is the variable ID, Var is the variable that the data is being read into, START is the position in the array from which to start reading, and COUNT is the number of positions to read in from each dimension. For this to work properly, the dimensions of Var must be the same as the dimensions of the variable COUNT. Again, after the function call, the variable **STATUS** is tested to ensure that the data was read in without error.

The following is a list of the variable IDs, the variables they are read into, and the description of what they are:

Netcdf ID	LTRANS variable	Description
Aks	**KHb (c, f)**	vertical diffusivity of salinity at rho nodes
angle	**rho_angle**	angle between x-coordinate and true east direction
Cs_r	**CS**	value used to adjust rho node depths
Cs_w	**CSW**	value used to adjust w node depths
h	**depth**	depths of rho nodes
lat_rho	**lat_rho**	latitude of rho nodes
lat_u	**lat_u**	latitude of u nodes
lat_v	**lat_v**	latitude of v nodes
lon_rho	**lon_rho**	longitude of rho nodes
lon_u	**lon_u**	longitude of u nodes
lon_v	**lon_v**	longitude of v nodes
mask_rho	**rho_mask**	rho node mask value
mask_u	**u_mask**	rho node mask value
mask_v	**v_mask**	rho node mask value
salt	**saltb (c, f)**	rho node salinity
s_rho	**SC**	value used to convert s-levels to rho node depths
s_w	**SCW**	value used to convert s-levels to w node depths
temp	**tempb (c, f)**	rho node temperature
u	**Uvelb (c, f)**	u-direction velocity
v	**Vvelb (c, f)**	v-direction velocity
w	**Wvelb (c, f)**	w-direction velocity
zeta	**zetab (c, f)**	zeta levels at rho nodes

After everything has been properly read into the program, the function NF90_CLOSE is called. It has the format "**STATUS** = NF90_CLOSE(**NCID**)" and simply takes the NetCDF ID (**NCID**) and disassociates it from the NetCDF file it was associated to. This makes it free to be used with the next NetCDF file.

The main structure of LTRANS is based on the assignment of a unique number to each ROMS model grid point (referred to as a node). Each grid cell (referred to as an 'element') is comprised of a set of 4 nodes. After the hydrodynamic data is read from the NetCDF files into the variables listed above, it is reorganized so that each data point is assigned the appropriate node number. The data points are assigned node numbers in the subroutine **initGrid** in the Hydrodynamic Module after the grid variables are read in.

Further information regarding how to input data from a NetCDF file can be found at the following NetCDF Fortran 77 and Fortran 90 interface guide websites:
http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-f77/
http://www.unidata.ucar.edu/software/netcdf/docs/netcdf-f90/

B. Particle location file

Overview: The starting locations of all particles are read in from an external file which is in comma-delimited format. The code for reading in this file is found in the main LTRANS.f90 program.

Input File: Depending on whether or not the Settlement Module is turned on, this file contains either four or five columns. In either case, the first column contains each particle's latitudinal coordinate, the second contains its longitudinal coordinate, the third column contains the particle's depth (in meters from surface, e.g., -35.55), and the fourth column contains the "date of birth" which is delay in seconds from the beginning of the model run until the time that particle will start being tracked (i.e., start to move). This is used to release batches of particles at different times. If all particles start moving at the start of the model simulation, then the value for their "date of birth" should be zero. If the Settlement Module is turned on, a fifth column must contain an identification number, usually the identification number of the habitat polygon from which each particle starts. In the example LTRANS model, the file is called "initial_part_location.csv".

Initialization: The filename and path (if needed) to the file must be specified correctly in the variable **parfile** in the LTRANS.data include file. The variable **numpar** should be equal to the number of rows in the particle locations file. (**numpar** equals the total number of particles tracked).

Numerical Method: The file is opened into unit 1 and then read into the variables **pLon, pLat, par(n,pZ), and par(n,pDOB)**, and, if settlement is on, **startpoly,** using a loop that iterates through **numpar** particles. For each particle **n**, **pLon**(**n**) contains the particle's longitude, **pLat**(**n**) contains the particle's latitude, **par(n,pZ)** contains the particle's depth, **par(n,pDOB)** contains the particles date of birth, and **startpoly**(**n**) contains a habitat polygon identification number.

After the data is read in, it must be converted from latitude and longitude to meters in order to be used in the model. This conversion is done in the Conversion Module.The particle start latitude and longitude locations are converted to meters and stored in the variables **par(n,pX)** and **par(n,pY),** respectively.

Variable Definitions: The following variables are used when reading in the particle locations file:

numpar – integer – number of particles
parfile – character array – path (if needed) and file name of the input file
pLat – dp – initial latitude of particles

pLon – dp – initial longitude of particles
par – dp – array containing particle x, y, and z locations (in meters), among other values
startpoly – integer – identification number for particles
settlementon – logical – .TRUE. if settlement is on, else .FALSE.

C. Habitat Polygon location files for Settlement Module

Overview: If the Settlement Module will be used, then the locations of habitat polygons must be read in from external files. These comma delimited files contain polygon identification numbers and latitude and longitude coordinates for the center and edges of the polygons. If the polygons have holes in them, two files must be read in, one containing the coordinates for the polygons and the second containing the coordinates for the holes.

Input File: Data regarding habitat polygon locations is contained in two separate files. The first contains the locations of the edges of all the polygons, while the second contains the locations of the edges of any holes that exist in the polygons. The file containing the polygon edge data has five columns: identification number, center point longitude, center point latitude, edge point longitude, and edge point latitude. Each polygon has one identification number and one center latitude and longitude that do not change, and a series of different edge points that encircle the center point. Thus, a file will use several rows to define a single polygon, repeating the identification number and center latitude and longitude with different edge latitudes and longitudes, going around the polygon's outline and ending with the edge point it started on to close the shape. See "End_polygons.csv" for an example.

The file containing the holes in the polygons is set up exactly like the polygon file but with a sixth column. The columns are the hole identification number, hole center longitude, hole center latitude, hole edge longitude, hole edge latitude, and the polygon identification number. The sixth column indicates the identification number of the polygon in which the hole is located. In the example LTRANS model, the file with hole information is called "End_holes.csv".

Initialization: The filename and path (if needed) to the file must be specified correctly in the variables **habitatfile** and **holefile** in LTRANS.inc. Also, a number of additional parameters in LTRANS.inc must be initialized. The parameter **pedges** must equal the number of rows in the habitat polygon file and the parameter **hedges** must equal the number of rows in the hole file. The parameters **minholeid**, **maxholeid**, **minpolyid**, and **maxpolyid** must contain the minimum and maximum id numbers used in both the habitat polygon and hole input files.

Numerical Method: The settlement input files are read in by the subroutine initSettlement found in the Settlement Module. The habitat polygon edge file is opened into unit 1 and then read into the variables **curpoly** and **P_lonlat** using a loop that iterates **pedges** number of times. For each edge **i**, **P_lonlat**(**i**,1) contains the edge's polygon identification number, **P_lonlat**(**i**,2) contains its center longitude, **P_lonlat**(**i**,3) contains its center latitude, **P_lonlat**(**i**,4) contains its edge longitude, and **P_lonlat** (**i**,5) contains its edge latitude. The polygon identification number is an integer, and must be read into an integer variable, **curpoly**, before being transferred to double precision variable **P_lonlat**.

The hole edge file is opened into unit 2 and then read into the variables **curpoly**, **curpoly2**, and **H_lonlat** using a loop that iterates **hedges** number of times. For each hedge **i**, **H_lonlat** (**i**,1) contains the current hedge's hole identification number, **H_lonlat** (**i**,2) contains its center longitude, **H_lonlat** (**i**,3) contains its center latitude, **H_lonlat** (**i**,4) contains its edge longitude, **H_lonlat** (**i**,5) contains its edge latitude, and **H_lonlat** (**i**,6) contains the identification number of the habitat polygon in which this hole is located. The hole identification number and habitat polygon identification numbers are integers, and are read into integer variables **curpoly** and **curpoly2**, before being transferred to double precision variable **H_lonlat**.

After the data is read in, it must be converted from latitude and longitude to meters in order to be used in the model. This is completed using the Conversion Module. The habitat polygon boundary locations are converted from the variable **P_latlon** and stored in the variable **polys**. The hole boundary locations are converted from the variable **H_lonlat** and stored in the variable **holes**.

Variable Definitions: The following variables are used when reading in habitat polygon files:

- **curpoly** – integer – temporarily holds integer input before transferring its value to double precision variables
- **curpoly2** – integer – temporarily holds integer input before transferring its value to double precision variables
- **habitatfile** – character array, parameter – the file and path (if needed) of the habitat polygon data
- **hedges** – integer, parameter – total number of hole edges
- **H_lonlat –** dp – latitude and longitude hole data read in from **holefile**
- **holefile** – character array, parameter – the file and path (if needed) of the hole data
- **holes** – dp – id number, center longitude, center latitude, edge longitude, edge latitude for each habitat polygon, habitat polygon id number
- **minholeid** – integer, parameter – lowest hole id number
- **minpolyid** – integer, parameter – lowest habitat polygon id number
- **maxholeid** – integer, parameter – highest hole id number
- **maxpolyid** – integer, parameter – highest habitat polygon id number
- **pedges** – integer, parameter – number of habitat polygon edge points
- **P_lonlat –** dp – latitude and longitude habitat polygon data read in from **habitatfile**
- **polys** – dp – id number, center longitude, center latitude, edge longitude, and edge latitude for each habitat polygon

V. Execution (LTRANS.f90, main program)

LTRANS.f90 contains the main structure of the particle-tracking program. It initializes the model, executes the external time step, internal time step, and particle loops, advects particles, writes output, and then finalizes the model. It calls the modules that read in hydrodynamic model information, move particles due to turbulence and behavior, test if particles are in habitat polygons, and apply boundary conditions to keep particles in the model domain. All this is done with three subroutine calls, ini_LTRANS, run_LTRANS, and fin_LTRANS, which initialize, run, and finalize the model.

Private Variables: The module contains twenty nine variables accessible only in this module and the subroutines and functions within it:

endpoly – integer – id number of the habitat polygon the particle ended on
ex – dp – back, center, and forward external times (s)
hitBottom – integer – number of times a particle struck bottom (i.e., intersected with bottom) in the last print interval time step
hitLand – integer – number of times a particle struck land (i.e., intersected with land boundaries) in the last print interval time step
it – integer – iteration variable for internal time step
ix – dp – back, center, and forward internal times (s)
nAttrib – integer, parameter – number of particle attributes in the **par** array

1	=	Particle X-coordinate (pX)
2	=	Particle Y-coordinate (pY)
3	=	Particle Z-coordinate (pZ)
4	=	Particle new X-coordinate (pnX)
5	=	Particle new Y-coordinate (pnY)
6	=	Particle new Z-coordinate (pnZ)
7	=	Particle previous X-coordinate (ppX)
8	=	Particle previous Y-coordinate (ppY)
9	=	Particle previous Z-coordinate (ppZ)
10	=	Status of particle (pStatus)
11	=	Particle Date Of Birth (pDOB)
12	=	Particle Age (s) (pAge)
13	=	Age at which particle settled or died (pLifespan)

p – integer – iteration variable for external time step
P_Salt – dp – salinity at the particle's location
P_Temp – dp – temperature at the particle's location
pAge – integer – index for particle age (s) in the **par** array
par – dp – array that stores attributes of each particle – **par**(numpar,nAttrib)
pDOB – integer – index for particle Date Of Birth in the **par** array
pLifespan – integer – index for age at which particle settled or died in the **par** array
pnX – integer – index for new x-location of particle in the **par** array
pnY – integer – index for new y-location of particle in the **par** array
pnZ – integer – index for new z-location of particle in the **par** array
ppX – integer – index for previous x-location of particle in the **par** array
ppY – integer – index for previous y-location of particle in the **par** array

ppZ – integer – index for previous z-location of particle in the **par** array
prcount – integer – print counter
printdt – integer – seconds elapsed in model time since last *para* file printed
pStatus – integer – index for status of particle in the **par** array (previously Color in LTRANS v.1)
pX – integer – index for current x-location of particle in the **par** array
pY – integer – index for current y-location of particle in the **par** array
pZ – integer – index for current z-location of particle in the **par** array
startpoly – integer – id number of the habitat polygon the particle started on
timeCounts – real – time counter for model time
times – real – time counter for real time (used for assessing speed of model)

A. LTRANS (run_LTRANS)

Overview: This subroutine initializes, runs, and finalizes the entire model. The following sections of the User's Guide contain explanations of the remaining code in the main program LTRANS.f90: initialization, the external time step, internal time step and particle loops as well as boundary condition tests, advection, print statements (output), finalization, and the subroutine find_currents.

Initialization: The length of the external time step (in seconds) is set in LTRANS.data with the variable **dt**. The value **dt** should be equal to the duration of the hydrodynamic data output intervals. The variable **stepT,** the total number of external time steps in the model, is initialized to **seconds** divided by **dt**, where **seconds** is the total number of seconds that the model will run and **dt** is the duration, in seconds, of the external time step.

Numerical Method: The subroutine begins by calling ini_LTRANS to initialize the model. Once the initialization is complete, the external time step loop begins as well as the internal time step and particle loops that are nested within it. The external time step consists of a loop from 1 to **stepT** using the variable **p** to iterate. Once the external time loop finishes, fin_LTRANS is called to finalize the model.

Variable Definitions: The following variables are used in this section:
days – real – number of days to run the model
dt – integer, parameter – duration of the external time step (s)
p – integer – iteration variable for external time step
seconds – integer – total number of seconds that the LTRANS model will run
stepT – integer – total number of external time steps

B. Initialization (ini_LTRANS)

Overview: Before the iterative loops that comprise the heart of the particle tracking model structure, LTRANS.f90 starts with an initialization section. Parameters are read in, variable arrays are allocated and initialized, and the particle locations are read in and their latitude and longitude coordinates are converted to meters. If output is being written to NetCDF files, the output files need to be created and initialized by calling initNetCDF and createNetCDF.
In addition, information about the ROMS hydrodynamic model domain is read in and used to create the LTRANS model domain and grid element structure. In LTRANS, an element is defined as a set of four adjacent rho, u or v nodes that form a quadrilateral. Each element is assigned a unique identification number. These numbers are used to store previous, and efficiently search for new, particle locations. Two subroutines are called to initialize the LTRANS domain and element structure. Subroutine initGrid is used to read the coordinates of the nodes in the rho, u, and v grids, depth at the rho nodes, the angle between x-coordinate and true east, masks of the rho, u, and v grid nodes that specify whether the nodes are on land or in water, and the variables necessary to calculate s-levels. It also assigns unique identification numbers to rho-, u- and v elements to create the LTRANS grid element structure.

Subroutine createBounds defines the LTRANS model boundaries based on the land/sea masking of the rho grid. Subroutine initBehave is used to initialize the matrices that contain information on particle attributes for the Behavior Module. Depending on how the model is set to handle problem particles (if **ErrorFlag** from LTRANS.data is 1-3) the file 'ErrorLog.txt' may be created to log all the particles that encounter problems. The model checks if the particles are in bounds by calling mbounds and ibounds, and determines which element each particle begins within by calling setEle_all.

Subroutine initHydro is called to read in the initial hydrodynamic data (u-, v-, and w-velocities, salinity, temperature, zeta, and vertical diffusivity) for the back, center, and forward time steps from the first ROMS sequential output file. The next sections create output files for boundary collisions, model timing, and header files describing the other output files, if they are specified to do so by the variables **TrackCollisions**, **WriteModelTiming**, and **WriteHeaders** from LTRANS.data. Lastly, the amount of time it took to initialize the model is written to the screen.

C. External time step loop (run_External_Timestep)

Overview: The loop which iterates for each external time step contains the majority of the execution code of the program. The execution of the external time step loop can be broken down into two major sections: updating the hydrodynamic data and the internal time step loop. The internal time step loop and the print statements will be covered in the following sections. The main purpose of the external time step loop is to update hydrodynamic data. The hydrodynamic data comes from ROMS NetCDF files which contain information about u velocity, v velocity, w velocity, salinity, sea surface height, and other attributes.

To calculate water properties at the particle location, LTRANS uses hydrodynamic model output from the current ('center') time step, the previous ('back') time step, and the future ('forward') time step. On the first iteration of the external time step the attributes of the back, center, and forward times are taken directly from the first netcdf file. However, on every subsequent iteration the back and center time steps' attributes are transferred from the previous center and forward time steps, respectively, and data from the netcdf files is only read in for the forward time step.

Initialization: The length of the external and internal time steps (in seconds) are set in LTRANS.data with the variables **dt** and **idt**. The value **dt** should be equal to the duration of the hydrodynamic data output intervals andthe value **idt** should be a factor of **dt**. The variable **stepIT** (the number of internal time steps per external time step) is then initialized as the value of **dt** (the external time step) divided by **idt** (the internal time step).

Numerical Methods: The external time step exists within a loop from 1 to **stepT** using the variable **p** to iterate found in run_LTRANS. The first two iterations use the same data, so the hydrodynamic data was initialized before the first iteration by calling subroutine **initHydro** and is not updated again until **p** is greater than 2. On all other iterations, the program updates hydrodynamic data by calling subroutine **updateHydro**. Both initHydro and updateHydro can be found in the Hydrodynamic Module. In **updateHydro**, the 'forward' variables are updated with the most recent hydrodynamic data and the 'back' and 'center' variables are replaced with the 'center' and 'forward' variables from the previous time step, respectively using rotating indices.

Following the update hydrodynamic data section is a short section used to update the external time step values in **ex**. The variable **ex** is an array of three values used to store the back time, center time, and forward time in seconds. These values are calculated by using multiples of **dt**, the size of the external time step in seconds. The external time step ends by running the internal time steps. The internal time step consists of a loop from 1 to **stepIT** using the variable **it** to iterate.

Variable Definitions: The following variables are used in this section:
- **dt** – integer, parameter – duration of the external time step (s)
- **ex** – dp – back, center, and forward external times (s)
- **p** – integer – iteration variable for external time step
- **seconds** – real – total number of seconds that the LTRANS model will run
- **stepT** – integer – total number of external time steps
- **tdim** – integer, parameter – total number of external time steps within each hydrodynamic model output file. Set in LTRANS.inc

D. Internal time step loop

Overview: The internal time step loop is the loop in which the particle tracking occurs. The internal time step is shorter than the external time step to allow particles to move in smaller intervals than the hydrodynamic model output intervals. Within each iteration of the internal time step loop, the time and internal time step values are updated. After this, the program enters the particle loop where particle movement over the time step is calculated (see next section for a description of the particle loop). Once this is complete, particle locations are updated. These events occur every iteration of the internal time step.

Initialization: The initial values of several variables must be set in LTRANS.data to dictate when output of the *para* files occurs. The variable **printdt**, which keep track of time passed since last print, is initialized to zero. The variable **idt** is initialized to the number of seconds in the internal (particle tracking) time step. Lastly, the variable **iprint** is initialized to the number of seconds between each time that data is to be written to a *para* file. Note that **iprint** should be a multiple of **idt**, since the output code is read at intervals of **idt**.

Numerical Method: First, the values of **ix**, the internal time step values, are calculated. **ix** is an array with three values, so it can hold the internal 'back', 'center', and 'forward' times. Once this is completed, update_particles is called to update the location of each particle for the current internal time step. Lastly, the print counter **printdt** is updated and if it has reached the value in **iprint**, printOutput is called to write model output.

Variable Definitions: The following variables are used in this section:

- **daytime** – real – model time in days
- **idt** – integer, parameter – duration of the internal time step
- **it** – integer – iteration variable for internal time step
- **ix** – dp – back, center, and forward internal times (s)
- **n** – integer – iteration variable for particle loops
- **newP_xyz** – dp – new particle x,y,z locations after particle loop
- **numpar** – integer, parameter – total number of particles
- **P_xyz** – dp – particle x,y,z locations before particle loop
- **stepIT** – integer – number of internal time steps per external time step
- **time** – integer – model time in seconds

E. Particle Loop

Overview: The particle loop is a loop that iterates through all of the particles, updating each particle's position for the current internal time step. It calculates advection at the particle location and conducts vertical and horizontal boundary tests. It is within this loop that the optional turbulence, behavior and settlement modules can be called. For this section, it is useful to be reminded that the main structure of LTRANS is based on the assignment of a unique number to each ROMS model grid point (referred to as nodes). Each grid cell (referred to an 'element') is comprised of a set of 4 nodes.

Numerical Method: First, the age of the particle is updated. The particle's age is incremented by the internal time step, **idt**. The particle's updated age and its previous status can be used to determine how it will behave in this iteration. For example, for oyster larvae, the program can determine if the particle is pre-pediveliger stage, pediveliger stage, or so old that it dies. The updated particle age is passed to **updateStatus** in the Behavior Module to update the current particle's settled or dead status. If the particle is dead or has settled, the program passes to the next particle because the current particle needs no further computation. It also does this when the particle has travelled outside the model domain via open ocean boundaries.

Next, the subroutine determines in which rho, u, and v grid elements each particle is located. To find the grid element in which the particle is located, the program calls the subroutine **setEle** from the Hydrodynamic Module without the optional final argument, indicating that the subroutine will cycle through a predetermined list of all the elements adjacent to the element in which the particle was last known to be. If the particle is not in the element in which it was in during the last time step or in any of its adjacent elements, the particle has jumped across an element and is considered to have violated the Courant-Friedrichs-Levy condition (note this is an informal application of formal condition which was derived for hydrodynamic models, not particle tracking models). In this case, the program either stops (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3). If this occurs frequently, the model should be restarted with a smaller internal time step (**idt**). If the program does not stop, it determines the rho grid, u grid, and v grid elements in which the particle is located and stores the node numbers of these elements for later computations.

Next, the program calls subroutine **setInterp** from the Hydrodynamic Module, which determines interpolation values for the particle's current location. The subroutine uses bilinear interpolation. In the unusual event that the bilinear interpolation fails, an inverse weighted distance technique is employed. The program then checks to ensure that the particle is within the vertical boundaries. The program then creates a matrix of z-coordinates at the particle's location for each s-level. This is done for the back, center, and forward external times steps for each rho and w s-level. This is necessary because sea surface height changes over time and therefore the vertical position of the s-levels also change.

The next four blocks of code are for advection (described below), horizontal turbulence (see Horizontal Turbulence Module), vertical turbulence (see Vertical Turbulence Module), and behavior (see Behavior Module). In addition, salinity and temperature at the particle location is calculated if the **SaltTempOn** parameter is set to .TRUE. in the LTRANS.data file. The displacement (m) of the particle due to advection in the x, y, and z directions are stored in the variables **AdvectX**, **AdvectY**, and **AdvectZ** (see *Advection* section below). The displacements of the particle due to horizontal turbulence in the x- and y-directions are stored in **TurbHx** and **TurbHy**. **TurbV** holds the result of the displacement of the particle in the z-direction due to vertical turbulence. The variables **XBehav**, **YBehav**, and **ZBehav** store the displacement in the x-, y-, and z-direction due to particle behavior, though behavior only affects the x- and y-directions if the behavior is set to Tidal Stream Transport.

Once those values have been calculated, they are applied to alter the location of the particle. **AdvectX** and **TurbHx** are added to the x-position of the particle, **AdvectY** and **TurbHy** are added to the y-position of the particle, and **AdvectZ** and **TurbV** are added to the z-position of the particle. If the particle is not within vertical boundaries after those updates it is reflected off the surface or bottom (see *Vertical boundaries test* section below). **ZBehav** is then added to the new z position and the vertical bounds are tested again. If the particle location is outside of a boundary, it is placed just within the boundary rather than being reflected. This is slightly different if the particle's behavior is Tidal Stream Transport. If the particle is exhibiting bottom oriented behavior during tidal stream transport, then z-position is set to the bottom depth at that location and the x- and y- position doesn't change. If the particle is currently riding tidal stream transport then **XBehav** is added to the x-position of the particle and **YBehav** is added to the y-position of the particle and the z-position is set to the specified swimming depth.

The next section of the program checks that the trajectory of the particle from its old location to its new location does not pass through any horizontal boundaries by calling the subroutine **intersect_reflect** that is located in the Boundary Module (see *Horizontal boundaries tests* section below). If it does pass through a boundary, it is reflected back into the model domain. The particle can reflect a maximum of three times before the program will print an error message to the screen and discontinue (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3). After the horizontal boundaries are tested, the program calls subroutine **setEle** as a final test to make sure that the particle is still within a rho-, u-, and v-grid element.

The last step in the particle loop is to determine if the particle can currently settle (if the Settlement Module is turned on (**settlementon** = .TRUE.). This is done by calling the subroutine **testSettlement** in the Settlement Module. The subroutine first checks if the particle is of the right age to settle, has not already settled, and if there are habitat polygons to settle on within the element in which the particle is located. If it passes those three tests, it continues by determining if the particle is within the boundaries of any habitat polygon in that element and not within the boundaries of any holes in that habitat polygon. If the particle is within a habitat polygon and not within a hole, then it settles. Its ending habitat polygon number is stored in the variable **endpoly** and its depth is set to equal the depth of the bottom at that location. If it is not within the boundaries of a habitat polygon, or if it is within the boundaries of a hole, then the particle can not settle and nothing is done. Regardless of whether the particle settles or not, this ends the particle loop. Upon the completion of the particle loop, there is a short section that iterates through all the particles from 1 to **numpar**, using the variable **n**, and updates the particle's position in the variable **par**. The particles' previous locations (**ppX**,**ppY**,**ppZ**) are updated to their current locations (**pX**,**pY**,**pZ**), and their current locations to their new locations (**pnX**,**pnY**,**pnZ**).

Variable Definitions: The following variables are used in this section:

AdvectX – dp – the distance that a particle moves in one internal time step due to advection in the x-direction (m)
AdvectY – dp – the distance that a particle moves in one internal time step due to advection in the y-direction (m)
AdvectZ – dp – the distance that a particle moves in one internal time step due to advection in the z-direction (m)
Behav – dp – the distance that a particle moves in one internal time step due to behavior (m)
idt – integer, parameter – duration of the internal time step

TurbHx – dp – the distance that a particle moves in one internal time step due to subgrid scale turbulence in the x-direction (m)
TurbHy – dp – the distance that a particle moves in one internal time step due to subgrid scale turbulence in the y-direction (m)
TurbV – dp – the distance that a particle moves in one internal time step due to subgrid scale turbulence in the z-direction (m)

1. Vertical boundaries test

Overview: At each time step the water surface levels change and the particles move. This section checks that the particles are not moved above the surface or below the bottom (i.e., this keeps them in the water).

Numerical Method: Vertical boundaries (surface and bottom) are specified for each particle by interpolating sea surface height and bottom depth to the x-y location of the particle. The values are interpolated by the subroutine **getInterp** from the Hydrodynamic Module. Once the vertical boundaries have been calculated they are stored in **P_depth**, **P_zetab**, **P_zetac**, and **P_zetaf**. The bottom boundary is stored in **P_depth**, and the surface boundaries due to changing sea levels are stored in **P_zetab**, **P_zetac**, and **P_zetaf** for the back, center, and forward external times. There are two sections where the vertical boundaries are tested, one at the beginning and one at the end of the particle loop.

If the particle is out of bounds at the beginning of the particle loop, then it is simply placed just within the boundaries. This is typically only needed if the sea surface has lowered since the last time step, leaving the particle just above water. The vertical boundaries are checked twice at the end of the particle loop, once after the particle's location is updated due to advection and turbulence, and one final time after the particle's location is updated due to behavior. If a particle passes through the surface or bottom boundary due to turbulence or vertical advection, the particle is placed back in the model domain at a distance that is equal to the distance that the particle has exceeded the boundary (i.e., it is reflected vertically). If a particles passes through the surface or bottom due to particle behavior, the particle is placed just below the surface or above the bottom (i.e., it stops near the boundary).

Variable Definitions: The following variables are used in this section:
newZpos – dp – new z coordinate after advection and turbulence
P_depth – dp – sea floor depth at the particle location
P_xyz – dp – particle's x,y,z coordinates before the current time step
P_zb(c, f) – dp – particle's depth at back, center, and forward internal time
P_zetab(c, f) – dp – surface height at the particle location for back, center, and forward time
reflect – dp – distance to reflect particle from surface or bottom if advection and turbulence moved the particle out of bounds

2. Advection

Overview: This model determines the displacement (m) of a particle due to advection in the x-, y-, and z- directions for each internal time step. Current velocities are estimated using a 4th order Runge-Kutta technique. 'Law of the wall' (log-layer calculation) is applied to particles near bottom.

Numerical Method: The advection model is based on a 4th order Runge-Kutta numerical method. A prototype of 4th order Runge-Kutta looks like this:

$$y_{n+1} = y_n + (^h/_6)*(k_{n1} + 2k_{n2} + 2k_{n3} + k_{n4})$$

Where:

$$k_{n1} = f(t_n, y_n)$$
$$k_{n2} = f(t_n + ½h, y_n + ½h* k_{n1})$$
$$k_{n3} = f(t_n + ½h, y_n + ½h* k_{n2})$$
$$k_{n4} = f(t_n + h, y_n + h* k_{n3})$$

These prototypes are implemented by calculating the k_n values using the subroutine find_currents subroutine in LTRANS.f90. First the k_{n1} values for the u-, v-, and w- directions are determined by passing in the values for the current particle location. find_currents returns the k_{n1} values. New location coordinates are calculated using k_{n1} values and passed back to find_currents to calculate k_{n2} values. This process is repeated until k_{n4} values are determined. The k_n values then are plugged into the main function to calculate advection values in the u-, v-, and w- directions. Finally, the u- and v-values are rotated using **P_angle** to the x- and y- component directions. Velocities at the particle location are multiplied by the internal time step **idt** to calculate displacement (m).

Variable Definitions: The following variables are used in the advection section:

- **AdvectX**, **AdvectY**, **AdvectZ** – dp – distance moved in the x, y, and z directions (m) due to advection
- **ex** – dp – back, center, and forward external times (s)
- **idt** – integer, parameter – duration of the internal time step (s)
- **ix** – dp – back, center, and forward internal times (s)
- **kn1_u(v, w)**, **kn2_u(v, w)**, **kn3_u(v, w)**, **kn4_u(v, w)** – dp – u, v, and w-component advection currents at the 1st, 2nd, 3rd, and 4th Runga-Kutta position
- **maxpartdepth**, **minpartdepth** – dp – to ensure the depth used to calculate 2nd, 3rd, and 4th Runga-Kutta positions is not outside vertical bounds
- **p** – integer – iteration variable for external time step
- **P_angle** – dp– angle between x-coordinate and true east at particle location
- **P_U**, **P_V**, **P_W** – dp – final advection values in U, V, and W directions
- **P_zb(c, f)** – dp – particle's depth at back, center, and forward time
- **Pwc_wzb(c, f)** – dp – w-coordinate depths at particle location
- **Pwc_zb(c, f)** – dp – rho-coordinate depths at particle location
- **Uad**, **Vad**, **Wad** – dp – U, V, and W advection values returned from FIND_CURRENTS
- **x1**, **x2**, **x3** – dp – x-coordinate used to calculate 2nd, 3rd, and 4th Runga-Kutta positions
- **Xpar** – dp – x-coordinate of the particle before the current time step
- **y1**, **y2**, **y3** – dp – y-coordinate used to calculate 2nd, 3rd, and 4th Runga-Kutta positions
- **Ypar** – dp – y-coordinate of the particle before the current time step
- **z1, z2, z3** – dp – z-coordinate used to calculate 2nd, 3rd, and 4th Runga-Kutta positions

Zpar – dp – z-coordinate of the particle

3. Horizontal boundaries test

Overview: After particle positions have been updated due to horizontal advection and turbulence, the particle locations are tested to ensure that they have not left the model domain.

Standard methods for dealing with horizontal boundaries in particle-tracking models (e.g., remove particle from routine, stick particle to boundary, place particle back at previous location, etc.) could not be applied because of the complicated nature of Chesapeake Bay shorelines and narrow tributaries. We developed reflective horizontal boundary condition routines to keep particles within the domain. For the boundary condition routines, boundary points of the mainland/sea boundary and each individual island are ordered to create closed polygons. Boundaries are taken to be halfway between water and land rho grid points. The trajectory of the particle is tested using the subroutine **intersect_reflect** from the Boundary Module. If the particle crosses over a horizontal boundary, it is reflected off the boundary it crosses with an angle of reflection that equals the angle of approach to the boundary. The distance that the particle is reflected is equal to the distance that the particle exceeded the boundary. This is done a maximum of three times before the program writes an error message and discontinues (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3). If the trajectory is found to no longer cross any boundaries before three bounces, the “crossings” point-in-polygon approach is used to ensure that the particle is inside the mainland/sea and outside island boundaries by using the subroutines **mbounds** and **ibounds** from the Boundary Module.

Initialization: The model boundaries are created by the subroutine **createBounds** in the Boundary Module, which is called in LTRANS.f90 after calling **initGrid** from the Hydrodynamic Module. **createBounds** must be called after **initGrid** because it needs the values in rho_mask which are read in by **initGrid**. Once **createBounds** has been called and the boundaries have been successfully created, the subroutines **mbounds**, **ibounds**, and **intersect_reflect** are used to determine if the particle is in the main boundaries, island boundaries, or must reflect off of any boundaries.

Numerical Method: The horizontal boundaries test uses three subroutines from the Boundary Module: **mbounds**, **ibounds**, and **intersect_reflect** (see the Boundary Module section for details on these subroutines). The subroutine **mbounds** determines whether or not the particle location is within the main boundaries of the model. The subroutine **ibounds** determines if the particle is within an island boundary. The subroutine **intersect_reflect** determines where an intersection took place and where the particle will be once reflected off of the boundary. Details on these subroutines can be found in the subroutine sections (for mbounds, ibounds, and intersect_reflect).

The horizontal boundaries are tested to ensure that the particle is within the model domain in two different sections of the code. The first test occurs during initialization. It determines if the starting location for each particle is within the model boundaries. The particle's location is tested with the subroutines **mbounds** and **ibounds**. If the particle is found to be either outside of the main boundaries or within island boundaries, the program writes "outside main bounds", returns the particle number, and stops (when ErrorFlag = 0 during initialization).

Horizontal boundaries are tested for each particle at the end of the particle loop after the particle's location has been updated due to advection, turbulence, and behavior to ensure that its new location is within the model boundaries.

First, the subroutine **intersect_reflect** is called to determine whether the particle trajectory crosses model boundaries. If an intersection occurs then it is called again using the intersection location and reflection location as the new particle start and end points. If after being reflected three times the particle is still outside of the boundaries, the program prints "still out after 3rd reflection" and stops (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3). If the particle ends in bounds with three or fewer reflections, **mbounds** is called to determine if the particle's new position is within main model boundaries. If the particle's new location is outside of the boundaries, the program prints an error message and stops (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3). If the particle is found to be within the main boundaries by **mbounds**, the program calls **ibounds**. If **ibounds** finds that the particle is not in an island, the particle passes the test and the program moves on. If the particle is in an island, the program prints an error message and stops (if ErrorFlag = 0) or writes the particle number to ErrorLog.txt and continues by putting the particle in its previous location, changing its status to dead, or changing its status to out of bounds (ErrorFlag = 1, 2, or 3).

When **ibounds** and **mbounds** return that the particle is within the model boundaries, the program moves on with the final reflected location as the new location of the particle.

Variable Definitions: The following variables are used in this section:

fintersectX, **fintersectY** – dp – x,y coordinates of intersection returned by intersect_reflect
freflectX, **freflectY** – dp – x,y coordinates of reflected location returned by intersect_reflect
in_island – integer – return variable of ibounds (1 – in and island, 0 – not in an island)
inbounds – integer – return variable of mbounds (1 – in bounds, 0 – not in bounds)
intersectf – integer – return variable of intersect_reflect (1 – intersection found, 0 – none)
island – dp – return variable of ibounds; id of the island the particle is in
newXpos, **newYpos** – dp – particles new x,y coordinates after advection and turbulence
nXpos, **nYpos** – dp – x,y coordinates of the location the particle is heading to, passed to intersect_reflect
P_xyz – dp – particle's x,y,z coordinates before the current time step
Reflects – integer – counter for the number of reflections made
skipbound – integer – input/output variable of intersect_reflect used to ensure that the particle does not reflect two consecutive times off the same boundary
Xpar, **Ypar** – dp – particle's x,y coordinates before the current time step
Xpos, **Ypos** – dp – input for intersect_reflect, containing x,y coordinates of the particle

F. Output

Overview: The output can ultimately be used to plot and view the particles and compare the outcomes of different model runs. There are three types of comma-delimited output files: *boundary collision*, *para* and *endbars*. The *boundary collisions* and *para* files are created periodically at set intervals throughout the running of the program and contain the particle locations at the current time. The *endbars* file is created only at the end of the program and contains information regarding each particles' start location, end location, and ending status. LTRANSv2 adds the additional option of writing output to NetCDF files as an alternative to the .csv files. There are two subroutines in LTRANS.f90 that handle output to the NetCDF and/or *boundary collisions* and *para* files. These will be described below. The *endbars* file is written in fin_LTRANS, which will be described in the next section.

Initialization: To write NetCDF files and/or *para* and *endbars* files, **writeNC** and/or **writeCSV** need to be set .TRUE. in LTRANS.data. Nothing else has to be initialized for the *endbars* file. For the *para* and NetCDF files, the initial values of several variables must be set to dictate when output occurs. The variable **printdt**, which keeps track of time passed since the last time output was written, is initialized to zero. The variable **idt** is initialized to the number of seconds in the internal (particle tracking) time step. Lastly, the variable **iprint** is initialized to the number of seconds between each time that data is to be written to a *para* or NetCDF file. Note that **iprint** should be a multiple of **idt**, since the output code is read at intervals of **idt**.

Numerical Method: The model prints *para* files and/or writes to a NetCDF file on the first iteration of the external time step and every interval of **iprint** seconds after the initial print. Each time the program reads the output section, the variable **printdt** is incremented by **idt** seconds (the internal time step). When **printdt** is equal to **iprint**, output is written with the latest data and **printdt** is reset to zero. This is all done in run_Internal_Timestep after the particle loop has completed. Also,the variable **prcount** keeps track of the number of times that the program has written output and is used to number the output files.

printOutput

Overview: This subroutine determines what values need to be written to either *para* or NetCDF files based on settings specified in LTRANS.data, then calls writeOutput with those values. If boundary collisions are being tracked, then the latest collision data is written to the *boundary collision* files. Lastly, if timing data is written out, if model timing data is being tracked.

Input Variables: This subroutine has no variables used for input.

Output Variables: This subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **numpar**, **SaltTempOn**, **TrackCollisions**, and **WriteModelTiming** from PARAM_MOD.

Module procedures used: The subroutine uses the functions x2lon and y2lat from the Conversion Module.

Numerical Method: This subroutine first checks the variables **SaltTempOn** and **TrackCollisions** from LTRANS.data to determine whether it should include output for salinity, temperature, or land/bottom hits. writeOutput is then called with the particles' x-, y-, and z-coordinates, and age, as well as the print counter **prcount**. If the model is tracking collisions, bottom hits and land hits are added to the list. If salinity and temperature at the particle location is being calculated, that information is added as well.

If the model is tracking boundary collisions, the latest collision data is written to .csv files. Each particle's x- and y- coordinates are converted to latitude and longitude and if it has collided with land or bottom since the last print, that data is written to LandHits.csv or BottomHits.csv. Once this is completed, the model resets the values in hitBottom and hitLand to 0. The last section is for model timing. If model timing is being tracked, then the latest timing information is appended to the Timing.csv file.

Variable Definitions: The following variables are used in this subroutine:
  variable name - type – explanation

writeOutput

Overview: This subroutine writes output to *para* files and/or calls writeNetCDF to write output to NetCDF files based on settings specified in LTRANS.data.

Input Variables: This subroutine has five required variables used for input, and four optional variables. The subroutine requires the x-, y-, and z- coordinates for each particle (**x**, **y**, **z**), the age of each particle (**P_age**), and the print counter (**prcount**) used in creating the *para* filename. The optional variables are **hitBottom** and **hitLand** which are included if boundary collisions are being tracked, and **P_Salt** and **P_Temp** which are included if salinity and temperature at the particle location is being calculated.

Output Variables: This subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **numpar**, **outpath**, **outpathGiven**, **SaltTempOn**, **TrackCollisions**, **writeCSV**, and **writeNC** from PARAM_MOD.

Module procedures used: The subroutine uses the function getStatus from the Behavior Module, the functions x2lon and y2lat from the Conversion Module, and the subroutine writeNetCDF from the Hyrodynamics Module.

Numerical Method: This subroutine first converts the particles' x- and y- coordinates (**x**, **y**) to latitude and longitude (**pLat**, **pLon**) and gets the status identification number for each particle from the behavior module by calling getStatus. If the model is writing output to .csv files (**writeCSV** = .TRUE.) then a para output file's name is assembled using **prefix2**, **counter2**, and **suffix2**, which are written to **buffer2** which is then read into **filenm2**. **Prefix2** is a four character array containing "para", **counter2** is an eight digit integer equal to 10,000,000 plus **prcount**, and **suffix2** is a four character array containing ".csv". **filenm2** can then be used in open statements to create para files.

When a para file is created it contains the current location of each particle and its status identification number. It also contains salinity and temperature at the particle's location from the previous internal time step (idt) if the variable **SaltTempOn** = .TRUE. in the LTRANS.data include file. The program can then cycle through and print each particle's current depth, status ID number, longitude, and latitude.

If the model is writing output to NetCDF files (**writeNC** = .TRUE.), then the subroutine writeNetCDF is called from the Hydrodynamic Module. The subroutine is passed the current model time (**ix**(3)), as well as the age (**P_age**), longitude and latitude (**pLon** and **pLat**), depth (**z**), and status identification number (**statuses**) of each particle. If the model is tracking collisions, bottom hits and land hits (**hitBottom** and **hitLand**) are also passed to the subroutine. If salinity and temperature at the particle location is being calculated, that information is passed along as well (**P_Salt** and **P_Temp**).

Variable Definitions: The following variables are used in this section:

- **buffer2** – char. array – temporary holder of complete *para* filename
- **color** – integer – color code for Surfer/Scripter
- **counter2** – integer – center (number) portion of the *para* output filename
- **dt** – integer, parameter – duration of the external time step (s)
- **endpoly** – integer – id number of the habitat polygon the particle ended on
- **filenm2** – char. array – complete *para* output filename 'para' + counter2 + '.csv'
- **ii** – integer – iteration variable for writing particle data to *para* files
- **iprint** – integer, parameter – interval between each print of *para* files (s)
- **n** – integer – iteration variable for particle loops
- **numpar** – integer, parameter – total number of particles
- **p** – integer – iteration variable for external time step
- **prcount** – integer – print counter
- **prefix2** – char. array – first part of the *para* output filename: 'para'
- **printdt** – integer – seconds elapsed in model time since last *para* file printed
- **P_nlatlon** – dp – particle's new location in latitude and longitude
- **P_xyz** – dp – particle's x,y,z coordinates before the current time step
- **startpoly** – integer – id number of the habitat polygon the particle started on
- **settle** – integer – settlement status (0 = did not settle, 1 = settled, 2 = dead)
- **suffix2** – char. array – end part of the *para* output filename: '.csv'
- **time** – integer – amount of time that has passed in the model (s)

G. Finalization (fin_LTRANS)

Overview: This subroutine writes out the *endfile* final locations file if output is being written to .csv files, deallocates all the dynamically allocated variables in the model, and concludes by writing to the screen how long the model ran.

Input Variables: This subroutine has no variables used for input.

Output Variables: This subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **numpar**, **outpath**, **outpathGiven**, **settlementon**, and **writeCSV** from PARAM_MOD.

Module procedures used: The subroutine uses the functions interp and WCTS_ITPI from the Hydrodynamic Module, the subroutine finBehave and function getStatus from the Behavior Module, x2lon and y2lat from the Conversion Module, and the subroutine finHydro from the Hydrodynamics Module.

Numerical Method: This subroutine first checks the variable **writeCSV** from LTRANS.data to determine whether it should create a endfile.csv file. If the model is writing to .csv files, each particle's x- and y- coordinates are converted to latitude and longitude. If settlement is turned on, the program will write each particle's starting habitat polygon identification number, ending habitat polygon number, particle status identification number, latitude, longitude, and lifespan. If settlement is turned off, the program only prints the particle status identification number, latitude, longitude, and lifespan.

Next, the subroutine deallocates all the dynamically allocated variables used by LTRANS.f90. It also calls finBehave and finHydro so that the dynamically allocated variables in those two modules will be deallocated as well. Note that if settlement is turned on, finBehave calls finSettlement to deallocate the Settlement Module's variables. The subroutine concludes by determining how long it took for the model to run, and writing that information to the screen in an easy to read format.

Variable Definitions: The following variables are used in this subroutine:
variable name - type – explanation

E. Variable definitions for the main program

AdvectX – dp – the distance that a particle moves in one internal time step due to advection in the x-direction (m)
AdvectY – dp – the distance that a particle moves in one internal time step due to advection in the y-direction (m)
AdvectZ – dp – the distance that a particle moves in one internal time step due to advection in the z-direction (m)

anykey – character – for error state read statement ‘Press Any Key’
Behav – dp – the distance that a particle moves in one internal time step due to behavior (m)
Behavior – integer – particle starting behavior (0 = passive, 1 = near-surface, 2 = near-bottom, 3 = DVM, 4 = *C. virginica* oyster larvae, 5 = *C. ariakensis* oyster larvae, 6 = constant sinking velocity)
buffer2 – char. array – temporary holder of complete *para* filename
color – integer – color code for Surfer/Scripter
counter2 – integer – center (number) portion of the *para* output filename
days – real – number of days to run the model
daytime – real – model time in days
Delay – dp – time to delay particle release (s)
deplvl – integer – lowest of the four consecutive s-levels closest to particle depth
dt – integer, parameter – duration of the external time step (s)
ele_err – integer – error ID returned from setEle
endpoly – integer – id number of the habitat polygon the particle ended on
ex – dp – back, center, and forward external times (s)
filenm2 – char. array – complete para output filename ‘para’ + counter2 + ‘.csv’
fintersectX – dp – x- coordinate of intersection returned by intersect_reflect
fintersectY – dp – y- coordinate of intersection returned by intersect_reflect
freflectX – dp – x- coordinate of reflected location returned by intersect_reflect
freflectY – dp – y- coordinate of reflected location returned by intersect_reflect
HTurbOn – logical – .TRUE. if Horizontal Turbulence is to be turned on, else .FALSE.
i – integer – iteration variable
idt – integer, parameter – duration of the internal time step
idum_call_count – dp – counter for number of times random number generator is called
ii – integer – iteration variable for writing particle data to para files
in_island – integer – return variable of ibounds (1 – in and island, 0 – not in an island)
inbounds – integer – return variable of mbounds (1 – in bounds, 0 – not in bounds)
inpoly – integer – return variable of settlement; returns 0 if particle does not settle and the habitat polygon id that it settles in if it does settle
intersectf – integer – return variable of intersect_reflect (1 – intersection found, 0 – none)
iprint – integer, parameter – interval between each print of para files (s)
island – dp – return variable of ibounds; id of the island the particle is in
it – integer – iteration variable for internal time step
ix – dp – back, center, and forward internal times (s)
j – integer – iteration variable
k – integer – iteration variable
kn1_u – dp – u-component advection currents at the 1st Runga-Kutta position
kn1_v – dp – v-component advection currents at the 1st Runga-Kutta position
kn1_w – dp – w-component advection currents at the 1st Runga-Kutta position
kn2_u – dp – u-component advection currents at the 2nd Runga-Kutta position
kn2_v – dp – v-component advection currents at the 2nd Runga-Kutta position
kn2_w – dp – w-component advection currents at the 2nd Runga-Kutta position
kn3_u – dp – u-component advection currents at the 3rd Runga-Kutta position
kn3_v – dp – v-component advection currents at the 3rd Runga-Kutta position
kn3_w – dp – w-component advection currents at the 3rd Runga-Kutta position

kn4_u – dp – u-component advection currents at the 4th Runga-Kutta position
kn4_v – dp – v-component advection currents at the 4th Runga-Kutta position
kn4_w – dp – w-component advection currents at the 4th Runga-Kutta position
maxpartdepth – dp – to ensure the depth used to calculate 2nd, 3rd, and 4th Runga-Kutta positions is not above vertical bounds
minpartdepth – dp – to ensure the depth used to calculate 2nd, 3rd, and 4th Runga-Kutta positions is not below vertical bounds
n – integer – iteration variable for particle loops
newP_xyz – dp – new particle x,y,z locations after particle loop
newXpos – dp – particles new x coordinates after advection and turbulence
newYpos – dp – particles new y coordinates after advection and turbulence
newZpos – dp – new z coordinate after advection and turbulence
numpar – integer, parameter – total number of particles
nXpos – dp – x coordinate of the location the particle is heading to, passed to intersect_reflect
nYpos – dp – y coordinate of the location the particle is heading to, passed to intersect_reflect
p – integer – iteration variable for external time step
P_age – dp – the time at which the particle starts movement, the current age of the particle, and the age at which the particle stops movement (via 'death' or settlement)
P_angle – dp– angle between x-coordinate and true east at particle location
P_depth – dp – sea floor depth at the particle location
P_latlon - dp – particle's start location in latitude and longitude
P_nlatlon – dp – particle's new location in latitude and longitude
P_Salt – dp – salinity at the particle's location
P_Temp – dp – temperature at the particle's location
P_U – dp – final advection value in U direction
P_V – dp – final advection value in V direction
P_W – dp – final advection value in W direction
P_xyz – dp – particle's x,y,z coordinates before the current time step
P_zb – dp – particle's depth at back internal time
P_zc – dp – particle's depth at center internal time
P_zetab – dp – surface height at the particle location for back time
P_zetac – dp – surface height at the particle location for center time
P_zetaf – dp – surface height at the particle location for forward time
P_zf – dp – particle's depth at forward internal time
parfile – character array – name and path (if needed) of the particle start location file
prcount – integer – print counter
prefix2 – char. array – first part of the para output filename: 'para'
printdt – integer – seconds elapsed in model time since last para file printed
Pwc_wzb – dp – w-coordinate depths at particle location for back time
Pwc_wzc – dp – w-coordinate depths at particle location for center time
Pwc_wzf – dp – w-coordinate depths at particle location for forward time
Pwc_zb – dp – rho-coordinate depths at particle location for back time
Pwc_zc – dp – rho-coordinate depths at particle location for center time
Pwc_zf – dp – rho-coordinate depths at particle location for forward time

reflect – dp – distance to reflect particle from surface or bottom if advection and turbulence moved the particle out of bounds
reflects – integer – counter for the number of reflections made
SaltTempOn – logical – .TRUE. if calculate salinity and temperature at particle location, else .FALSE.
seconds – real – total number of seconds that the LTRANS model will run
seed – integer – number used to initialize the random number generator Mersenne Twister
settle – integer – settlement status (0 = did not settle, 1 = settled, 2 = dead)
skipbound – integer – input/output variable of intersect_reflect used to ensure that the particle does not reflect two consecutive times off the same boundary
startpoly – integer – id number of the habitat polygon the particle started on
stepIT – integer – number of internal time steps per external time step
stepT – integer – total number of external time steps
suffix2 – char. array – end part of the para output filename: '.csv'
time – integer – amount of time that has passed in the model (s)
TurbHx – dp – the distance that a particle moves in one internal time step due to sub grid scale turbulence in the x-direction (m)
TurbHy – dp – the distance that a particle moves in one internal time step due to sub grid scale turbulence in the y-direction (m)
TurbV – dp – the distance that a particle moves in one internal time step due to sub grid scale turbulence in the z-direction (m)
Uad – dp – U advection value returned from FIND_CURRENTS
us – integer – number of depth levels in the rho, u, and v grids
VAD – dp – V advection value returned from FIND_CURRENTS
VTurbOn – logical – .TRUE. if Vertical Turbulence is to be turned on, else .FALSE.
WAD – dp – W advection value returned from FIND_CURRENTS
ws – integer – number of depth levels in the w grid
x1 – dp – x-coordinate used to calculate 2nd Runga-Kutta position
x2 – dp – x-coordinate used to calculate 3rd Runga-Kutta position
x3 – dp – x-coordinate used to calculate 4th Runga-Kutta position
Xpar – dp – particle's x- coordinate before the current time step
Xpos – dp – input for intersect_reflect, containing x coordinate of the particle
y1 – dp – y-coordinate used to calculate 2nd Runga-Kutta position
y2 – dp – y-coordinate used to calculate 3rd Runga-Kutta position
y3 – dp – y-coordinate used to calculate 4th Runga-Kutta position
Ypar – dp – particle's y- coordinate before the current time step
Ypos – dp – input for intersect_reflect, containing y coordinate of the particle
z1 – dp – z-coordinate used to calculate 2nd Runga-Kutta position
z2 – dp – z-coordinate used to calculate 3rd Runga-Kutta position
z3 – dp – z-coordinate used to calculate 4th Runga-Kutta position
Zpar – dp – particle's z- coordinate before the current time step

H. Subroutine FIND_CURRENTS

Overview: Subroutine FIND_CURRENTS calculates the current velocities in the u-, v-, and w-directions at the particle location at a specific moment in time.

Input Variables: This subroutine has 16 variables used for input. The variables **Xpar**, **Ypar**, and **Zpar** are the particle's xyz coordinates. **Pwc_zb**, **Pwc_zc**, **Pwc_zf**, **Pwc_wzb**, **Pwc_wzc**, and **Pwc_wzf** are the depths at the particle location on the rho and w grids for the external back, center, and forward times. **P_zb**, **P_zc**, and **P_zf** contain the depth of the particle at the external back, center, and forward times. The variables **ex** and **ix** contain the external and internal time step values in seconds. The variable **p** contains the number of the current iteration of the external time step; the subroutine uses **p** to determine if it is the first iteration, which is treated differently from subsequent iterations. Lastly, the variable **version** contains the value 1, 2, or 3, which indicates which results—the back, center, or forward results—should be returned.

Output Variables: This subroutine has three output variables. **Uad**, **Vad**, and **Wad** return the current velocity values for the u-, v-, and w- directions.

Module parameters used: The subroutine uses the parameters **us**, **ws** and **z0** from PARAM_MOD.

Module procedures used: The subroutine uses the functions interp and WCTS_ITPI from the Hydrodynamic Module, the subroutine TSPSI and function HVAL from the Tension Module, and the function polintd from the Interpolation Module.

Numerical Method: This subroutine interpolates to the x-y particle location along s-levels near the particle (the four closest to the particle) to create a profile of current velocities at the particle location. It then fits a tension spline and which it uses to determine the current velocities at the particle location for back, center, and forward times of the external time step. Polynomial interpolation in time is then used to calculate current velocities at the back, center and forward times of the internal time step.

The subroutine first determines which four s-levels are the closest to the particle for both the rho and the w s-levels. The subroutine checks if the particle is below each s-level, starting with the lowest and moving up through each s-level. The first s-level below which the particle is found is the closest level above the particle. This level, the one above it, and the two below it are taken as the four closest to the particle (two above and two below). The two exceptions to this are if the particle is between the first two s-levels or the last two s-levels, in which case there will only be one s-level on one side, and three on the other.

If the particle is between the lowest two s-levels, its advection values are affected by the bottom and are determined using log-layer calculation values. At this point the advection values for back, center, and forward times of the external time step are calculated by log values for the particles that are within the lowest s-level. If the particle's distance from the bottom is less than the roughness coefficient **z0**, the advection values are set to 0.0. Using these values, current velocities at the internal time steps are calculated using polynomial interpolation. Depending on the value of **version**, either the back, center, or forward internal time step is calculated. If **version** is 1, the back internal time step current velocity values will be calculated. If **version** is

2, then the center current velocities are calculated, and if **version** is 3, the forward current velocities are calculated.

If the particle is not between the lowest two s-levels, the advection values can be calculated using the subroutine WCTS_ITPI from the Hydrodynamic Module. This subroutine uses the four closest s-levels and creates tension splines of the water column at back, center, and forward time using TSPACK (found in tension_module.f90). The tension spline is then evaluated at the particle location. If TSPACK fails to make a tension spline, then linear interpolation is used instead. Once the advection values for back, center, and forward times have been calculated, the current velocities at the internal time steps are calculated using polynomial interpolation. As with the method shown above for particles affected by the bottom, the value of **version** determines whether the subroutine will return either the back, center, or forward internal time step.

The subroutine FIND_CURRENTS is then completed, and the current velocity values are returned in the variables **UAD**, **VAD**, and **WAD**.

Variable Definitions: The following variables are used in this subroutine:

ex(3) - dp – external time step values in seconds for back, center, and forward
ey(3) - dp – advection velocities at external time steps
i - integer – used for iteration through do loops
ii - integer – lowest of the four consecutive s-levels closest to particle depth on the rho grid
iii - integer – lowest of the four consecutive s-levels closest to particle depth on the w grid
ix(3) – dp - internal time step values in seconds for back, center, and forward times
nN – integer, parameter – number of s-levels used in tension splines
p - integer – external time step do loop iteration variable
P_Ub - dp – u-velocity at particle location at back time
P_Uc - dp – u-velocity at particle location at center time
P_Uf - dp – u-velocity at particle location at forward time
P_Vb - dp – v-velocity at particle location at back time
P_Vc - dp – v-velocity at particle location at center time
P_Vf - dp – v-velocity at particle location at forward time
P_Wb - dp – w-velocity at particle location at back time
P_Wc - dp – w-velocity at particle location at center time
P_Wf - dp – w-velocity at particle location at forward time
P_zb - dp – depth of particle at back time
P_zc - dp – depth of particle at center time
P_zf - dp – depth of particle at forward time
Pwc_ub - dp – u-velocity at the lowest rho s-level at particle location at back time
Pwc_uc - dp – u-velocity at the lowest rho s-level at particle location at center time
Pwc_uf - dp – u-velocity at the lowest rho s-level at particle location at forward time
Pwc_vb - dp – v-velocity at the lowest rho s-level at particle location at back time
Pwc_vc - dp – v-velocity at the lowest rho s-level at particle location at center time
Pwc_vf - dp – v-velocity at the lowest rho s-level at particle location at forward time
Pwc_wb - dp – w-velocity at the second lowest w s-level at particle location at back time
Pwc_wc - dp – w-velocity at the second lowest w s-level at particle location at center time

Pwc_wf - dp – w-velocity at the second lowest w s-level at particle location at forward time
Pwc_wzb(ws) - dp – z-coordinates of each w s-level at particle location at back time
Pwc_wzc(ws) - dp – z-coordinates of each w s-level at particle location at center time
Pwc_wzf(ws) - dp – z-coordinates of each w s-level at particle location at forward time
Pwc_zb(us) - dp – z-coordinates of each rho s-level at particle location at back time
Pwc_zc(us) - dp – z-coordinates of each rho s-level at particle location at center time
Pwc_zf(us) - dp – z-coordinates of each rho s-level at particle location at forward time
Uad - dp – u-direction advection return value
us – integer, parameter – total number of rho s-levels
Vad - dp – v-direction advection return value
version - integer – input variable to determine what the subroutine returns
version = 1 : return advection at back time
version = 2 : return advection at center time
version = 3 : return advection at forward time
Wad - dp – w-direction advection return value
ws – integer, parameter – total number of w grid s-levels
Xpar - dp – x-coordinate of particle
Ypar - dp – y-coordinate of particle
z0 - dp – roughness height of bay floor
Zpar - dp – z-coordinate of particle

H. Subroutine writeModelInfo

Overview: Subroutine simply writes all the model run information out to the screen.

Input Variables: This subroutine has no variables used for input.

Output Variables: This subroutine has no output variables.

Module parameters used: The subroutine uses several parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Numerical Method: This subroutine writes to the screen every major user-defined option from LTRANS.data.

Variable Definitions: The following variables are used in this subroutine:
variable name - type – explanation

VI. Behavior Module (behavior_module.f90, BEHAVIOR_MOD)

Overview: The Behavior Module is used to assign biological or physical characteristics to particles. The module is called for each particle for each internal time step and returns the distance that a particle moves due to behavior in that time step. In LTRANS v.2, particle characteristics can include a swimming/sinking speed component and a behavioral cue component that can depend upon particle age. The swimming/sinking speed component controls the speed of particle motion and can be constant or set with a function. The behavioral cue component regulates the vertical direction of particle movement. For biological behaviors, a random component is added to the swimming speed and direction to simulate random variation in the movements of individuals (in behavior types 1 – 5, see list below). Physical characteristics, such as constant sinking velocity, can also be assigned to particles without the additional random movements (behavior type 6). Horizontal swimming is included in tidal stream transport (behavior type 7). The following behavior types are currently available in LTRANS and are specified using the **Behavior** parameter in the LTRANS.data file:

- Passive (no behavior): **Behavior** = 0. In this case, the Behavior Module is not executed. Particle motion is based on advection and, if turned on, horizontal and vertical turbulence.
- Near-surface orientation: **Behavior** = 1. Particles swim up if they are deeper than 1 m from the surface.
- Near-bottom orientation: **Behavior** = 2. Particles swim down if they are shallower than 1 m from the bottom.
- Diurnal vertical migration: **Behavior** = 3. Particles swim down if light levels at the particle location exceed a predefined threshold value.
- *Crassostrea virginica* oyster larvae: **Behavior** = 4. Swimming speeds and direction of motion vary depending upon age (stage) according to field and laboratory observations (see North et al. 2008).
- *C. ariakensis* oyster larvae: **Behavior** = 5. Swimming speeds and direction of motion vary depending upon age (stage) according to field and laboratory observations (see North et al. 2008).
- Sinking velocity: **Behavior** = 6. Particles move up or down with constant sinking (or floating) speeds without individual random motion. Code that calculates salinity and temperature at the particle location is included (but commented out) as a basis for calculating density-dependent sinking velocities.
- Tidal Stream Transport: **Behavior** = 7. Particles undergo tidal stream transport in the flood tide direction by swimming horizontally with the flooding tide at a predetermined depth. Particles remain on bottom during slack and ebb tide. This code is still in development.

Private Variables: The module contains ten variables accessible only to this module. Double precision array **timer** acts as the timer for *C. ariakensis* downward swimming behavior. Integer array **P_behave** holds the integer specifying the behavior for each particle. Double precision arrays **P_pediage**, **P_deadage**, **P_Sprev**, **P_zprev**, and **P_swim** contain the age at which the particle will settle, the age at which the particle will die (stop moving), the salinity at the particle's previous location, the depth at the particle's previous location, and the particle's swimming speed, respectively, for each particle. Logical arrays **bottom**, **dead**, and **oob** contain

.TRUE. for each particle if they are on the bottom (for behavior 7), dead, or out of bounds, respectively, or .FALSE. if they are not.

Public Procedures: The following are the public subroutines and functions contained within the Behavior Module: subroutines **behave**, **die**, **finBehave**, **initBehave**, **setOut**,**updateStatus**, and functions **getStatus**, **isDead**, **isOut**.

Private Procedures: This module has no private subroutines or functions.

A. Subroutine behave

Overview: This subroutine returns the values **XBehav**, **YBehav**, **ZBehav**, and **bott**, the distance (m) that a particle moves in one internal time step (**idt**) in the x, y, and z directions, and if the particle is on the bottom. Most behaviors only utilize the output variable **ZBehav**. Only Tidal Stream Transport uses **XBehav**, **YBehav**, and **bott**.

Input Variables: The subroutine takes 21 variables as input: **Xpar** (x-coordinate of the particle), **Ypar** (y-coordinate of the particle), **Zpar** (z-coordinate of the particle), **Pwc_zb** (z-coordinates of each rho s-level at particle location at back time), **Pwc_zc** (z-coordinates of each rho s-level at particle location at center time), **Pwc_zf** (z-coordinates of each rho s-level at particle location at forward time), **P_zb** (depth of particle at back time), **P_zc** (depth of particle at center time), **P_zf** (depth of particle at forward time), **P_zetac** (sea surface height at particle location), **P_age** (current age of the particle), **P_depth** (depth of the particle), **P_U** (u- component velocity at the particle location), **P_V** (v- component velocity at the particle location), **P_angle** (angle between x- coordinate and true east at the particle location), , **n** (current particle number), **it** (iteration number of the internal time step), **ex** (back, center, and forward external times), **ix** (back, center, and forward internal times), **daytime** (time since the beginning of the model run), and **p** (iteration variable for external time step).

Output Variables: This subroutine outputs the variables **XBehav**, **YBehav**, **ZBehav**, and **bott**, the distance (m) that a particle moves in one internal time step (**idt**) in the x, y, and z directions, and if the particle is on the bottom.

Module parameters used: The subroutine uses 16 parameters from PARAM_MOD: **us** (number of depth levels in the rho, u, and v grids), **dt** (length of external time step), **idt** (length of internal time step), **twistart** (time of twilight when the sun started rising), **twiend** (time of twilight when the sun finished sinking), **Em** (irradiance at solar noon), **pi** (value of mathematical constant PI), **daylength** (length of day), **Kd** (vertical light attenuation coefficient), **thresh** (light threshold that cues diurnal vertical migration behavior), **Sgradient** (salinity gradient threshold that cues larval behavior for oyster larvae behavior types), **swimfast** (maximum swimming speed), **swimstart** (age when swimming or sinking begins), **sink** (sinking velocity for constant behavior type), **Hswimspeed** (horizontal swimming speed), and **Swimdepth** (depth particles rise to during flood tide)..

Module procedures used: This subroutine uses the functions WCTS_ITPI from HYDRO_MOD and genrand_real1 from RANDOM_MOD.

Private Variables Used: This subroutine uses the variables **timer**, **P_behave**, **P_pediage**, **P_deadage**, **P_Sprev**, **P_zprev**, **P_swim,** and **bottom**, which are private variables accessible only to the Behavior Module.

Numerical Method: First, the subroutine calculates swimming speeds and values needed for the oyster larvae behaviors, and then the distance of transport via particle behavior is calculated depending upon the specified behavior type (**P_behave**).

Initially, the particle swimming speed (**P_swim(n,3)**) is updated for each time step. If the age of the particle (**P_age**) is greater than or equal to the age when the swimming behavior starts (**swimstart**), then (**P_swim(n,3)**) is set to a number that is a function of particle age (if **swimslow** does not equal **swimfast** in the LTRANS.data file) or a constant value (if **swimslow** equals **swimfast**). If the age of the particle is greater than or equal to the age at which particles reach maximum swimming speed and can settle (**P_pediage**), then the swimming speed is set equal to the maximum swimming speed (**swimfast**).

Additional routines are executed if particle behavior simulates oyster larvae (**P_behave** = 4 or 5). If a particle is of pediveliger age and not dead, then the behavior code (**P_behave**) is set to 2 so that the particle has pediveliger-like bottom-oriented behavior. Also, the **timer** variable is decremented (see explanation below) and salinity at the particle location (**P_S**) is calculated. Salinity at the particle location (P_S) is also calculated for Tidal Stream Transport behavior.

Next the velocity (speed and direction) of particle motion in the vertical direction due to behavior (**parBehav**) is calculated depending upon the particle behavior type (only true for behaviors 1-6, as described below). Finally, velocity is multiplied by the duration of the internal time step (**idt**) to derive the distance (**ZBehav**) that the particle moves in that time step. The value for **ZBehav** includes distance as well as direction (with negative indicating movement down and positive indicating movement up). Behavior 7 not only calculates movement in the vertical directions (**ZBehav**), but it also calculates movement in the horizontal (**XBehav** and **YBehav**). The following paragraphs contain explanations of the eight behavior types available in LTRANS:

***1. Passive** (no behavior): **Behavior** = 0.*
In this case, the Behavior Module is not executed and no motion due to behavior is included in particle motion. Therefore, particle motion is based on advection and, if turned on, horizontal and vertical turbulence. To make particles simulate water motion with no behavior, specify the following parameters in the LTRANS.data file: **HTurbOn** = .TRUE., **VTurbOn** = .TRUE., and **Behavior** = 0.

***2. Near-surface orientation**: **Behavior** = 1.*
A particle 'swims' up if it is deeper than 1 m from the surface and 'swims' randomly if it is within 1 m of the surface. Particle velocity (**parBehav**, m s^{-1}) is determined by the particle

swimming speed (**P_swim(n,3)**) multiplied by 1) a value that sets its direction (**negpos** = 1 for up, **negpos** = -1 for down), and 2) a random deviate (**devB**) between 0 and 1 which is used to simulate random variation in swimming speeds between individuals. The direction of motion (**negpos**) also includes a random component to simulate differences in swimming directions between individuals. A random deviate (**dev1**) between 0 and 1 is drawn and compared to the value of **switch**, which is set to be a number between 0 and 1. If the value of **dev1** exceeds the value of **switch**, then particle direction is down (**negpos** = -1). If the depth of the particle is deeper than 1 m from the surface, **switch** = 0.8 so that the particle has an 80% chance of swimming up and a 20% chance of swimming down in that time step. If the depth of the particle is within 1 m of the surface, **switch** = 0.5 so that the particle swims in random directions (with a 50% chance of swimming up and a 50% chance of swimming down).

***3. Near-bottom orientation**: **Behavior** = 2.*

A particle 'swims' down if it is shallower than 1 m from the bottom and 'swims' randomly if it is within 1 m of the bottom. Particle velocity (**parBehav**, m s^{-1}) is determined by the particle swimming speed (**P_swim(n,3)**) multiplied by 1) a value that sets its direction (**negpos** = 1 for up, **negpos** = -1 for down), and 2) a random deviate (**devB**) between 0 and 1 which is used to simulate random variation in swimming speeds between individuals. The direction of motion (**negpos**) also includes a random component to simulate differences in swimming directions between individuals. A random deviate (**dev1**) between 0 and 1 is drawn and compared to the value of **switch**, which is set to be a number between 0 and 1. If the value of **dev1** exceeds the value of **switch**, then particle direction is down (**negpos** = -1). If the depth of the particle is deeper than 1 m from the surface, **switch** = 0.2 so that the particle has a 20% chance of swimming up and an 80% chance of swimming down in that time step. If the depth of the particle is within 1 m of the bottom, **switch** = 0.5 so that the particle swims in random directions (with a 50% chance of swimming up and a 50% chance of swimming down).

***4. Diurnal vertical migration (DVM)**: **Behavior** = 3.*

A particle swims down if light levels at the particle location exceed a predefined threshold value. NOTE: this section of code is not universal! The equation for irradiance at the water's surface (**E0**) has not been published, was fit to light data from the Chesapeake Bay region, and initialized with values for September 1 at the latitude of 37.00 degrees. If you would like to use the DVM behavior, the calculation of light at the particle location must be adjusted for underwater irradiance at the location and time of the model runs, which depends upon both the available sunlight (e.g., angle of the sun) as well as the clarity of the water (e.g., the attenuation coefficient).

The DVM section of the Behavior Module first calculates irradiance just below the water's surface (**E0**) using estimates of daylength (**daylength**), time since the sun started rising (**tst**) and irradiance at solar noon (**Em**). These values, which are derived from values set in the LTRANS.data file, depend on the day of the year and latitude and are calculated with equations available in Kirk (1994) and Meeus (1998). It should be noted that these calculations assume that the LTRANS model simulations start at midnight. Once surface light levels are calculated, irradiance at the depth of the particle location (**P_light**, μE m^{-2} s^{-1}) is calculated using **E0** and an attenuation coefficient (**Kd**).

Next the particle velocity is calculated by comparing the light at the particle location (**P_light**) to a user-defined threshold value that cues behavior (**thresh**, µE m^{-2} s^{-1}). Particle velocity (**parBehav**, m s^{-1}) is determined by the particle swimming speed (**P_swim(n,3)**) multiplied by 1) a value that sets its direction (**negpos** = 1 for up, **negpos** = -1 for down), and 2) a random deviate (**devB**) between 0 and 1 which is used to simulate random variation in swimming speeds between individuals. The direction of motion (**negpos**) also includes a random component to simulate differences in swimming directions between individuals. A random deviate (**dev1**) between 0 and 1 is drawn and compared to the value of **switch**, which is set to be a number between 0 and 1. If the value of **dev1** exceeds the value of **switch**, then particle direction is down (**negpos** = -1).

If the irradiance at the particle location (**P_light**) is greater than the threshold light value (**thresh**), then **switch** = 0.2 so that the particle has a 20% chance of swimming up and an 80% chance of swimming down in that time step. If irradiance at the particle location is less than the threshold value, then **switch** = 0.5 so that the particle swims in random directions (with a 50% chance of swimming up and a 50% chance of swimming down).

5. Oyster larvae (two species): Behavior *= 4 and 5.*
Particle motion due to behavior depends upon the species of oyster larvae (4 = *Crassostrea virginica* and 5 = *C. ariakensis*) as well as the age (stage) of particles. LTRANS model simulations with these behaviors are described in North et al. (2008) and animated at http://northweb.hpl.umces.edu/videos_animations/Oyster_Larvae_Animations.htm.

The oyster larvae behavior sub-model is parameterized with larval behaviors discerned in preliminary analysis of laboratory studies (Newell et al. 2005). All particles begin swimming when they are 0.5 days old (as trocophores) and continue swimming during the veliger stage. In the model runs of North et al. (2008), each particle is assigned a different veliger and pediveliger stage duration to simulate individual variation. The veliger stage ends at the time of P_pediage and the pediveliger stage ends at the time of P_deadage. In North et al. (2008), particles enter the pediveliger stage after ~13.5 days, and remain competent to settle for another ~7 days during which they search for suitable substrate. If they do not find suitable substrate within this time period, they are considered 'dead'. The code that assigns each particle a different stage duration occurs in the subroutine **initBehave** but is commented out so that all particles have the same **P_pediage** and **P_deadage** for use with other behavior types.

Swimming speeds of *C. virginica* and *C. ariakensis* larvae vary from 0 to ~3.0 mm s^{-1} over the course of the 2-3 week development from fertilized eggs to pediveligers ready for settlement (Mann and Rainer 1990, Kennedy 1996, Newell et al. 2005). In LTRANS v.2, the swimming speed of a particle is determined by its age. For particles from 0 to 0.5 days old, particles are assumed to be fertilized gametes and early trocophores that do not swim (i.e., swimming speed = 0). After 0.5 days, particles enter the late trocophore and veliger stages and begin to swim. From day 0.5 to the end of the veliger stage, their maximum swimming speed increases linearly from 0.5 mm s^{-1} to 3 mm s^{-1}. To simulate random variation in the movements of individual oyster larvae, the maximum swimming speed is multiplied by a number drawn from a uniform random distribution between 0 and 1 so that particle swimming speed varies in each time step. During the pediveliger stage, the swimming speed is 3 mm s^{-1} and no random component is added (although there is a random component to the direction as explained below). The swimming speeds of *C. virginica* and *C. ariakensis* are treated with the same model formulation because laboratory results indicated that their speeds did not significantly differ (Newell et al. 2005).

The behavioral cue component of the behavior sub-model regulates the direction of particle movement. Preliminary analysis of laboratory studies (Newell et al. 2005) indicates that *C. virginica* larvae generally swim up in the presence of a halocline whereas *C. ariakensis* larvae generally swim down and remain near bottom. Laboratory results of Hidu and Haskin (1978) also indicate that *C. virginica* oyster larvae change behavior in response to salinity gradients. This plus information from discussions with R. Newell, J. Manuel, and V. Kennedy are used to assign the stage-dependent behaviors to *C. virginica* and *C. ariakensis* particles. Although oyster

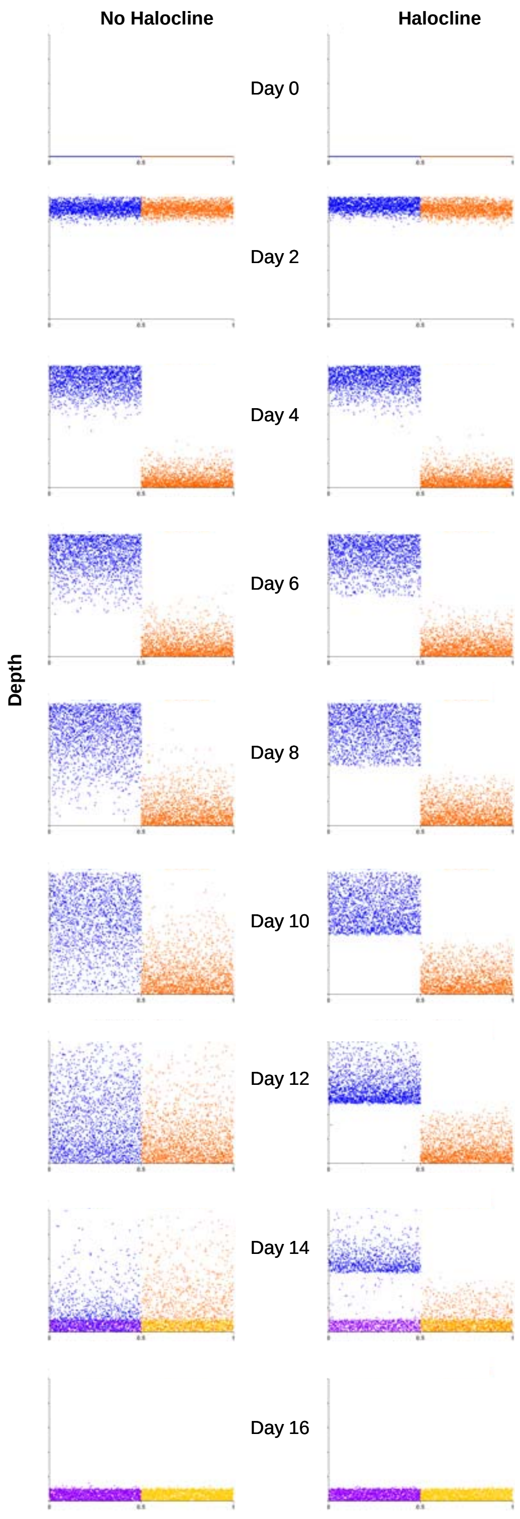

Fig. 6. Depth distribution of *C. virginica* veligers (blue) and pediveligers (purple), and *C. ariakensis* veligers (orange) and pediveligers (yellow) over time in the absence (left) or presence (right) of halocline.

larvae tend to swim in a helical trajectory, behavioral motion of these particles is limited to the vertical direction and is considered an integration of a helical swimming pattern. In addition, the randomly assigned upward and downward motion of particles is considered to be an integration of observed swimming and sinking behaviors.

To simulate random variation in the movements of individual oyster larvae, the direction of particle motion is assigned a random component, which is weighted so the particles have a tendency to move up or down depending on species and age of particle. Particle velocity (**parBehav**, m s^{-1}) is determined by the particle swimming speed (**P_swim(n,3)**) multiplied by 1) a value that sets its direction (**negpos** = 1 for up, **negpos** = -1 for down), and 2) a random deviate (**devB**) between 0 and 1 which is used to simulate random variation in swimming speeds between individuals. The direction of motion (**negpos**) also includes a random component to simulate differences in swimming directions between individuals. A random deviate (**dev1**) between 0 and 1 is drawn and compared to the value of **switch**, which is set to be a number between 0 and 1. If the value of **dev1** exceeds the value of **switch**, then particle direction is down (**negpos** = -1). The stage- and species-specific values of **switch** are described below.

The change in depth distribution of *C. virginica* and *C. ariakensis* particles with development and in response to haloclines is summarized in Fig. 6. In the late trocophore and early veliger stage (0.5 to 1.5 d), particles of both species have a 90% chance of swimming up (**switch** = 0.1) to simulate the initial near-surface distribution of larvae observed by Newell and Manuel (pers. comm.). Once in the veliger stage, the behaviors differ between species and in the presence or absence of a halocline. In the absence of a halocline, *C. virginica* veliger-stage particles are assigned probabilities that shift their distribution from the upper layer to the lower layer as they increase in age, from a 51% chance of swimming up in each time step to a 51.7% chance of swimming down (**switch** is a linear function of particle age). This results in a gradual shift in the depth distribution of older particles, as has been observed (Andrews 1983, Baker 2003) and modeled in previous studies (Dekshenieks et al. 1996). In the absence of a halocline, *C. ariakensis* veliger-stage particles between 2.5 and 3.0 days old are assigned probabilities of swimming down that decrease from 70% to 50.05% (**switch** is a linear function of particle age) to gradually shift their distribution toward bottom (Fig. 6). After 3.5 days of age, particles are assigned 50.05% probability of swimming down (**switch** = 0.495) to simulate broadly dispersed, but weakly bottom-oriented, distributions in well-mixed conditions as observed by J. Manuel and R. Newell (pers. comm.).

In the presence of a halocline, the veliger-stage particles of the two species respond differently to the same salinity cue (Fig. 6). The presence of a halocline is determined by the change in the vertical gradient in salinity (ΔS) experienced by the particle. This is computed as the change in salinity (s) at the particle location divided by the change in depth of the particle (z) between the previous ($n-1$) and the present (n) time step:

(6) $$\Delta S = \frac{\left(s_{n-1} - s_n\right)}{\left(z_{n-1} - z_n\right)}$$

If this gradient in salinity is greater than a threshold value, then *C. virginica* veliger-stage particles are cued to swim up with an 80% probability in that time step (**switch** = 0.80). This response, combined with the slight bottom-oriented shift as particles increased in age, results in aggregation of particles above the halocline (Fig. 6). Aggregation of *C. virginica* larvae above a halocline has been observed in several field studies (summarized by Kennedy 1996). If *C. ariakensis* veliger-stage particles pass through a salinity gradient, they are cued to swim down with 80% probability (**switch** = 0.20) for 2 hrs (using the **timer** variable) if they are not within 1 m of the bottom. If the particles come within 1 m of the bottom within the 2-hr time period, the probability of swimming up or down is 50% (**switch** = 0.50). This simulates the strong bottom-oriented behavior of *C. ariakensis* in the presence of a halocline as observed by Newell et al. 2005.

Once the age of a particle (**P_age**) is greater than or equal to its **P_pediage**, the behavior type of the particle is changed to bottom-oriented (**P_behave** = 2). Pediveliger-age particles of both species have the same behavior: they swim down with 100% probability until within 1 m of bottom. Within 1 m of bottom, pediveliger particles have randomly directed motions with a 50% probability of swimming up or down (Fig. 6). Particles remain in the pediveliger stage until they either settle on a simulated oyster bar (if the Settlement Module is turned on) or reach the age at which they are no longer competent to settle (i.e., they die). At this point, particles stop moving to conserve computational resources.

***6. Sinking velocity**: **Behavior** = 6.*
Particles move up or down with constant sinking (or floating) speeds without individual random motion. The velocity is specified using the variable **sink** in the LTRANS.data file. Code that calculates salinity and temperature at the particle location is included in this section (but is commented out) in case the user would like to calculate density at the particle location and density-dependent sinking velocities (e.g., Stokes velocity).

***7. Tidal Stream Transport: Behavior** = 7.*
Particle move up on flood tide to a pre-determined depth **Swimdepth** and swim horizontally in the direction of flood tide with a swimming speed of **Hswimspeed** (both variables are set in LTRANS.data). If current speed at the particle location is less than 0.05 m s^{-1} (considered slack tide), the particle moves to the bottom where it remains until salinity at the particle location is increasing, indicating flood tide. If current speeds are greater than 0.05 m/s, the particle will swim in the direction of the flooding current at the specified depth. If current speeds drop below this threshold, the particle will assume slack tide and swim to the bottom.

Variable Definitions: The following variables are used in this subroutine:

bott – logical – output variable used for Tidal Stream Transport, returns .TRUE. if the particle is currently waiting on bottom for the next flood tide

bottom – logical – module array used for Tidal Steam Transport, for each particle it contains .TRUE. if the particle is currently waiting on bottom for the next flood tide, .FALSE. if it is currently swimming with the flood tide

btest – integer – used to initiate random swimming (if btest = 0)

currentspeed – dp – current speed at the particle location, used to determine Tidal Stream Transport

daylength – dp – length of day (hr), set in LTRANS.data

daytime – real – time since the beginning of the model run (days)

deltaS – dp – change in salinity at particle location between previous and current time step
deltaz – dp – change in particle depth between previous and current time step
deplvl – integer – depth level
dev1 – real – random deviate used to add individual variation to the direction of particle motion
devB – real – random deviate used to add individual variation to the swimming speed of particles
dt – integer – length of external time step; interval between hydrodynamic inputs (s)
dtime – dp – time of day in hrs since midnight
E0 – dp – irradiance just below the water's surface (μE m^{-2} s^{-1})
Em – dp – irradiance at solar noon (μE m^{-2} s^{-1})
ex – dp – back, center, and forward external times (s)
Hdistance – dp – horizontal distance (m) particle will travel in this time step at its current swimming speed; used in Tidal Stream Transport
Hswimspeed – dp – horizontal swimming speed (m/s) used in Tidal Stream Transport
i – integer – variable used to iterate through do loops
idt – integer – length of internal (particle tracking) time step (s)
it – integer – iteration number of the internal time step
ix – dp – back, center, and forward internal times (s)
Kd – dp – vertical light attenuation coefficient
n – integer – the current particle number
negpos – real – sets direction of particle motion (1 for up, -1 for down)
p – integer – iteration variable for external time step
P_age – dp – the current age of the particle (s)
P_angle – dp– angle between x-coordinate and true east at particle location
P_behave – integer – Behavior type of each particle
P_deadage – dp – array of ages at which particles stop moving (i.e., dies)
P_depth – dp – depth of the particle (m)
P_light – dp – irradiance at the particle location (μE m^{-2} s^{-1})
P_pediage – dp – array of ages at which particles reach maximum swimming speed and can settle (i.e., becomes a pediveliger for oyster larvae behavior types)
P_S – dp – salinity at the particle location
P_Sprev – dp – Salinity at the previous location of the particle (for calculating salinity gradient experience by particle)
P_swim – dp – matrix used to calculate linear change in swimming speed and store swimming speed value for each particle. (n,1) = slope, (n,2) = intercept, (n,3) is swimming speed (m/s).
P_T – dp – temperature at the particle location
P_U – dp – u- component velocity at the particle location
P_V – dp – v- component velocity at the particle location
P_zb - dp – depth of particle at back time (m)
P_zc - dp – depth of particle at center time (m)
P_zf - dp – depth of particle at forward time (m)
P_zetac – dp - sea surface height at particle location (m)
P_zprev – dp – Depth at which particle was previously located (for calculating salinity gradient experience by particle)

parBehav – dp – particle velocity (m s^{-1})
pi – dp – the mathematical constant π
Pwc_zb(us) - dp – z-coordinates of each rho s-level at particle location at back time
Pwc_zc(us) - dp – z-coordinates of each rho s-level at particle location at center time
Pwc_zf(us) - dp – z-coordinates of each rho s-level at particle location at forward time
Sgradient – dp – the salinity gradient threshold that cues larval behavior (psu m^{-1}) for oyster larvae behavior types. Specified in LTRANS.data file.
sink – dp – sinking velocity for constant behavior type
Sslope – dp – salinity gradient that larvae experiences between the previous and current time step (psu m^{-1})
Swimdepth – dp – distance above bottom (m) at which particle swims during flood time
swimfast – dp – maximum swimming speed (m s^{-1}). Specified in LTRANS.data file.
swimstart – dp – age (time) when swimming or sinking begins (s). Specified in LTRANS.data file.
switch – real – a number between 0 and 1 which controls the probability that a particle will swim up or down in any given time step
switchslope – real – used to calculate **switch** with a linear function that depends on the particle age
theta – dp – angle used to determine the direction of currents at the particle location for tidal stream transport behavior
thresh – dp – light threshold that cues diurnal vertical migration behavior (μE m^{-2} s^{-1})
timer – real – timer used to count how long particles swim down for *C. ariakensis* behavior type
tst – dp – time since twilight when the sun started rising (the beginning of the day)
twiEnd – dp – the time of twilight after sunset (hr)
twiStart – dp – the time of twilight before sunrise (hr)
us – integer – number of depth levels in the rho, u, and v grids
X – dp – the current velocity in the x- component direction; used in Tidal Stream Transport behavior
XBehav – dp – the distance that a particle moves in the x direction in one internal time step due to behavior (m)
Xpar – dp – x-coordinate of the particle
Y – dp –current velocity in the y- component direction; used in Tidal Stream Transport behavior
YBehav – dp – the distance that a particle moves in the y direction in one internal time step due to behavior (m)
Ypar – dp – y-coordinate of the particle
ZBehav – dp – the distance that a particle moves in the z direction in one internal time step due to behavior (m)
Zpar – dp – z-coordinate of the particle

B. Subroutine die

Overview: This subroutine changes the value in array **dead** associated with particle **n**, to .TRUE., indicating that particle **n** is now deceased.

Input Variables: This function only has the input variable, **n**, which contains the particle number which has died.

Output Variables: The subroutine has no return variables.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The function edits the array **dead**.

Numerical Method: The value in array **dead** at location **n** is set to .TRUE..

Variable Definitions: The following variables are used in this function:
- **n** – integer – the number of the current particle for which the mortality status is being changed
- **dead** – logical – array containing the mortality status of each particle, .TRUE. = deceased, .FALSE. = alive.

C. Subroutine finBehave

Overview: This subroutine deallocates all the arrays allocated for the behavior module, and if settlement is turned on, calls finSettlement.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameter **settlementon** from PARAM_MOD.

Module procedures used: The subroutine uses the subroutine finSettlement from SETTLEMENT_MOD.

Private Variables Used: The subroutine uses the variables **timer**, **P_behave**, **P_pediage**, **P_deadage**, **P_Sprev**, **P_zprev**, **P_swim**, **dead**, and **oob** which are private variables accessible only to the Behavior Module.

Numerical Method: The subroutine deallocates the matrices that contain behavior information for each particle: **timer**, **P_behave**, **P_pediage**, **P_deadage**, **P_Sprev**, **P_zprev**, **P_swim**, **dead**, and **oob**. Then, if the Settlement Module is turned on, this subroutine calls finSettlement so that the variables allocated within it may be deallocated as well.

Variable Definitions: The following variables are used in this subroutine:

dead – logical – array containing the mortality status of each particle, .TRUE. = deceased, .FALSE. = alive.

oob – logical – array containing the out of bounds status of each particle in regards to contact with open ocean boundaries, .TRUE. = left the model via "open ocean", .FALSE. = never encountered "open ocean" boundaries.

P_behave – integer – Behavior type of each particle

P_deadage – dp – array of ages at which particles stop moving (i.e., die)

P_pediage – dp – array of ages at which particles reach maximum swimming speed and can settle (i.e., becomes a pediveliger for oyster larvae behavior types)

P_Sprev – dp – Salinity at the previous location of the particle (for calculating salinity gradient experience by particle)

P_swim – dp – matrix used to calculate linear change in swimming speed and store swimming speed value for each particle. (n,1) = slope, (n,2) = intercept, (n,3) is swimming speed (m/s).

P_zprev – dp – Depth at which particle was previously located (for calculating salinity gradient experience by particle)

settlementon – logical – turns Settlement Module on (.TRUE.) or off (.FALSE.). Specified in LTRANS.data file.

timer – real – timer used to count how long particles swim down for *C. ariakensis* behavior type

D. Function getStatus

Overview: This function returns a status identification number that describes a particle's behavior type (0-7) or settled (-2) or dead (-1) or out of bounds (-3) status. These numbers are subsequently written to output files. The **getStatus** function was developed to provide output that can be used to assign colors to particles in visualization routines such as Surfer/Scripter.

Input Variables: This function only has the input variable, **n**, which contains the current particle number. The function determines the status of the particle specified by **n**.

Output Variables: The function returns an integer identification number for the particle indicating its behavior type or status.

Module parameters used: The function uses **settlementon** and **openoceanboundary** from PARAM_MOD.

Module procedures used: The function uses function **isSettled** from SETTLEMENT_MOD.

Private Variables Used: The function uses the variables **P_behave**, **dead**, and **oob** which are private variables accessible only to the Behavior Module.

Numerical Method: The status identification number is assigned to the behavior type (0-7). Then it checks the particle's value in array **dead**, and if the particle has died, it is set to the value -1. If settlement is turned on, the function **isSettled** is called from the Settlement Module to determine if the particle has settled. If it has settled then the status identification number is updated to -2. Lastly, if "open ocean" boundaries are turned on, it checks the particle's value in the array **oob**, and if the particle is out of bounds, sets the value to -3. The function then returns the final updated status identification number.

Variable Definitions: The following variables are used in this function:

dead – logical – array containing the mortality status of each particle, .TRUE. = deceased, .FALSE. = alive.
n – integer – the number of the current particle for which the status identification number is being determined
openoceanboundary – logical – contains .TRUE. if open ocean boundaries are turned on, else .FALSE.
oob – logical – array containing the out of bounds status of each particle in regards to contact with open ocean boundaries, .TRUE. = left the model via "open ocean", .FALSE. = never encountered "open ocean" boundaries.
P_behave – integer – Behavior type of each particle
settlementon – logical – turns Settlement Module on (.TRUE.) or off (.FALSE.). Specified in LTRANS.data file.

E. Subroutine initBehave

Overview: This subroutine allocates and initializes the matrices that contain information on particle attributes for the Behavior Module.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **numpar** (total number of particles), **Behavior** (initial behavior type), **swimfast** (maximum swimming speed), **swimslow** (swimming speed when particle begins to swim), **swimstart** (age when swimming or sinking begins), **pediage** (age at which particle reaches maximum swimming speed and can settle), **deadage** (age at which particle stops moving/dies), **Sgradient** (salinity gradient threshold that cues larval behavior for oyster larvae behavior types), and **settlementon** (logical indicating whether the Settlement Module is on or not) from PARAM_MOD.

Module procedures used: The subroutine uses the subroutine **initSettlement** from SETTLEMENT_MOD.

Private Variables Used: The subroutine uses the variables **timer**, **P_behave**, **P_pediage**, **P_deadage**, **P_Sprev**, **P_zprev**, **P_swim**, **dead**, **bottom** and **oob**, which are private variables accessible only to the Behavior Module.

Numerical Method: The subroutine allocates and initializes the matrices that contain information on the particle attributes, including behavior type for each particle (**P_behave**), the age at which a particle reaches maximum swimming speed and can settle if the Settlement Module is on (**P_pediage**), and the age at which a particle stops moving (**P_deadage**). If the particles simulate oyster larvae, **P_pediage** refers to the age at which a particle becomes a pediveliger and **P_deadage** refers to the age at which the particle dies. If you would like to assign a different **P_pediage** and **P_deadage** to each particle, the code can be added in this subroutine (see commented code for an example). Note that **P_deadage** can be used to stop particle motion for all behavior types and that **P_pediage** does not cause particles to settle if the Settlement Module is not on. Finally, **P_deadage** is used to stop calculating particle motion due to advection, turbulence and behavior in order to conserve computational resources if the particle position no longer needs to be tracked.

The matrix **P_swim** is also populated in subroutine initBehave. Initially, swimming speed (**P_swim(n,3)**) is set equal to zero in this subroutine. The other arrays in **P_swim** are used to calculate a linear change in swimming speed that depends upon particle age by specifying a slope (**P_swim(n,1)**) and intercept (**P_swim(n,2)**). The slope and intercept are defined by the parameters **swimfast**, **swimslow**, and **swimstart** which are specified in the LTRANS.data file. To implement a constant swimming speed (for behavior types 1-3 that include random components to individual particle motions), set both **swimslow** and **swimfast** to the desired constant speed. To implement a constant sinking (or floating) velocity without individual variation, set the parameters **Behavior** = 6 and **sink** equal to the desired velocity in the LTRANS.data file.

Additional variables are initialized for the oyster larvae behavior routines (**timer**, **P_Srev**, **P_zprev**, **dead**, **oob**). Note that the array **bottom** is only allocated and initialized for Tidal Stream Transport (**Behavior** = 7).Finally, if the Settlement Module is turned on, this subroutine passes the age at which particles can settle (**P_pediage**) to the Settlement Module.

Variable Definitions: The following variables are used in this subroutine:

Behavior – integer – initial behavior type which is set in the LTRANS.data file (0 = passive, 1 = near-surface, 2 = near-bottom, 3 = diurnal vertical migration (DVM), 4 = *C. virginica* oyster larvae, 5 = *C.* ariakensis oyster larvae, 6 = sinking velocity, 7 = Tidal Stream Transport)

bottom – logical – module array used for Tidal Steam Transport, for each particle it contains .TRUE. if the particle is currently waiting on bottom for the next flood tide, .FALSE. if it is currently swimming with the flood tide

dead – logical – array containing the mortality status of each particle, .TRUE. = deceased, .FALSE. = alive.

deadage – dp – age at which particle stops moving, set in the LTRANS.data file

n – integer – used to iterate through each particle

numpar – integer – total number of particles

oob – logical – array containing the out of bounds status of each particle in regards to contact with open ocean boundaries, .TRUE. = left the model via “open ocean”, .FALSE. = never encountered “open ocean” boundaries.
P_behave – integer – Behavior type of each particle
P_deadage – dp – array of ages at which particles stop moving (i.e., die)
P_pediage – dp – array of ages at which particles reach maximum swimming speed and can settle (i.e., becomes a pediveliger for oyster larvae behavior types)
P_Sprev – dp – Salinity at the previous location of the particle (for calculating salinity gradient experience by particle)
P_swim – dp – matrix used to calculate linear change in swimming speed and store swimming speed value for each particle. (n,1) = slope, (n,2) = intercept, (n,3) is swimming speed (m/s).
P_zprev – dp – Depth at which particle was previously located (for calculating salinity gradient experience by particle)
pediage – dp – age at which particles can settle, set in the LTRANS.data file
settlementon – logical – turns Settlement Module on (.TRUE.) or off (.FALSE.). Specified in LTRANS.data file.
Sgradient – dp – the salinity gradient threshold that cues larval behavior (psu/m) for oyster larvae behavior types. Specified in LTRANS.data file.
swimfast – dp – maximum swimming speed (m/s). Specified in LTRANS.data file.
swimslow – dp – swimming speed when particle begins to swim (m/s). Specified in LTRANS.data file.
swimstart – dp – age (time) when swimming or sinking begins (s). Specified in LTRANS.data file.
timer – real – timer used to count how long particles swim down for *C. ariakensis* behavior type

F. Function isDead

Overview: This function checks the status of the current particle to see if the particle has died.

Input Variables: The function has just one input variable. It is passed the particle id number of the current particle (**n**).

Output: The function returns the logical value .TRUE. if the current particle has died and .FALSE. if it has not.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variable **dead** which is a private variable accessible only to the procedures in the Behavior Module.

Numerical Method: This function simply returns the value in array **dead** at the location of the specified particle, **n**, which is .TRUE. if the particle has died or .FALSE. if it is still alive.

Variables Definitions: The following variables are used in this section:

- **n** – integer – id number of the current particle
- **dead** – logical – array containing the mortality status of each particle, .TRUE. = deceased, .FALSE. = alive.
- **isDead** – logical – the function output variable

G. Function isOut

Overview: This function checks the status of the current particle to see if the particle has left the model domain.

Input Variables: The function has just one input variable. It is passed the particle id number of the current particle (**n**).

Output: The function returns the logical value .TRUE. if the current particle has left the model domain and .FALSE. if it has not.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variable **oob** which is a private variable accessible only to the procedures in the Behavior Module.

Numerical Method: This function simply returns the value in array **oob** at the location of the specified particle, **n**, which is .TRUE. if the particle has exited the model domain or .FALSE. if it is still in the model domain.

Variables Definitions: The following variables are used in this section:

- **n** – integer – id number of the current particle
- **oob** – logical – array containing the out of bounds status of each particle in regards to contact with open ocean boundaries, .TRUE. = left the model via "open ocean", .FALSE. = never encountered "open ocean" boundaries.
- **isOut** – logical – the function output variable

H. Subroutine setout

Overview: This subroutine changes the status of the current particle to indicate that the particle has left the model domain.

Input Variables: The function has just one input variable. It is passed the particle id number of the particle to set out(**n**).

Output Variables: The subroutine has no output variables..

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variable **oob** which is a private variable accessible only to the procedures in the Behavior Module.

Numerical Method: This function simply sets the value in array **oob** at the location of the specified particle, **n**, to .TRUE. indicating the particle has exited the model domain.

Variables Definitions: The following variables are used in this section:
- **n** – integer – id number of the current particle
- **oob** – logical – array containing the out of bounds status of each particle in regards to contact with open ocean boundaries, .TRUE. = left the model via “open ocean”, .FALSE. = never encountered “open ocean” boundaries.

I. Subroutine updateStatus

Overview: This subroutine determines whether a particle has died and updates its status accordingly.

Input Variables: The subroutine takes two variables as input: **P_age** (the current age of the particle) and **n** (the number identifying the particle).

Output Variables: The subroutine has no output variables.

Module parameters used: The function uses **settlementon** and **mortality** from PARAM_MOD.

Module procedures used: The subroutine uses the function isSettled from SETTLEMENT_MOD.

Private Variables Used: The subroutine uses the variable **P_deadage**, which is a private variable accessible only to the Behavior Module.

Numerical Method: The subroutine determines if a particle dies and updates the variable **dead** (via the subroutine die). A particle is considered dead (and will stop moving) if its age (**P_age**) exceeds **P_deadage** and it has not yet been able to “settle” on suitable habitat. A particle can only die if **mortality** is set to .TRUE. in LTRANS.data. If settlement is turned on a call is made to function **isSettled** found in the Settlement Module to make sure the particle has not settled.

Variable Definitions: The following variables are used in this subroutine:

mortality – logical – turns particle mortality on (.TRUE.) or off (.FALSE.)
n – integer – the number identifying the current particle
P_age – dp – the current age of the particle (s)
P_deadage – dp – age at which particle stops moving (i.e., dies)
settlementon – logical – turns Settlement Module on (.TRUE.) or off (.FALSE.).

VII. Boundary Module (boundary_module.f90, BOUNDARY_MOD)

Overview: The Boundary Module contains all the variables and procedures necessary to create the model boundaries and to test if a particle has traveled outside the boundaries.

Private Variables: The module contains fifteen variables and one derived data type accessible only in this module. The variable **bnds** is a one-dimensional allocatable array of the derived data type **boundary**. In subroutine createBounds, **bnds** is allocated to the total number of boundary points. For each boundary point, **bnds** contains data concerning whether the point lies on the U grid (**onU**) (if not on the U grid, the boundary point is implied to be on the V grid), the number of nodes from the left side (**ii**) and the bottom (**jj**) specifying the point's location, and the number assigned to the polygon (**poly**), where 1 indicates the main water boundary polygon and all subsequent values indicate island boundary polygons. The module also has variables for island hole identification numbers (**hid**), island hole edge point x- and y- coordinates (**hx**, **hy**), and habitat polygon edge point x- and y- coordinates (**bx**, **by**), which are one dimensional allocatable arrays. A pair of two-dimensional allocatable arrays contains the x- and y- coordinates of the start and end points of every boundary segment in the model (**bnd_x**, **bnd_y**). Another one dimensional array called **land** contains a logical value for each boundary segment, .TRUE. if it is a land boundary, .FALSE if it is an open ocean boundary. The module has five integer variables: the total number of boundary points (**bnum**), the number of boundary points that make up the water boundary polygon (**maxbound**), the number of boundary points that make up all of the island boundary polygons (**maxisland**), the total number of islands (**numislands**), and the total number of boundary segments that surround both the water and the islands (**nbounds**). Lastly, the Boundary Module contains the logical variable **BND_SET** which is initialized to .FALSE. but switches to .TRUE. once the boundaries have been set.

Public Procedures: The following are the public subroutines and functions contained within the Boundary Module: **subroutines createBounds**, **ibounds**, **intersect_reflect**, **mbounds**, and **function isBndSet**.

Private Procedures: The following are the private subroutines and functions accessible only to the other procedures in the Boundary Module: subroutines **add**, **getNext, output_llBounds**, and **output_xyBounds**.

A. Subroutine add

Overview: This subroutine is called by createBounds to add a boundary point to **bnds**, the array containing the boundary point information.

Input Variables: The subroutine has three input variables: **isU**, **iii**, and **jjj**. The variable **isU** is logical and will be .TRUE. if the boundary point being added is on the U grid and .FALSE. if it is on the V grid. The variables **iii** and **jjj** contain the i and j locations of the added point on the U or V grid.

Input/Output Variables: The subroutine has two variables used for input and output: **polydone** and **done**. **polydone** is a flag to indicate whether the current polygon has been completed, and **done** is a flag to indicate that all the boundaries have been used.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variables **bnds** and **BND_SET**, which are private variables accessible only to the procedures in the Boundary Module.

Numerical Method: This subroutine first checks if the point to be added is identical to the first point added for the current polygon. If this is the case, the point is not added. Instead, **polydone** is set to .TRUE. to indicate the current polygon has been completed, **polynum** is incremented, and **polystart** is reset to reflect what will be the first position of the next polygon, if there is a subsequent polygon. If the total number of boundary points added (**b**) is equal to the total number of boundary points (**bnum**) then **done** is set to .TRUE. to indicate that all the boundary points have been added and **BND_SET** is switched to .TRUE. to indicate that the boundaries have been completed.

If the boundary point passed in to **add** is not identical to the first point added for the current polygon, then **b** is incremented to the next boundary point location and **bnds** is updated at the **b** array location with the new point information.

Variable Definitions: The following variables are used in this subroutine:

b – integer – number of boundary points added so far
BND_SET – logical – .TRUE. if the boundaries have been created, else .FALSE.
bnds – derived data type **boundary** – boundary point data; whether each point lies on the U grid (**onU**), as opposed to the V grid, its location on the respective grid in terms of nodes from the left side (**ii**) and nodes from the bottom (**jj**), and the id number of the polygon (**poly**) that the boundary point is a part of
done – logical – .TRUE. if all the boundary points have been used, else .FALSE.
iii – integer – i position on the grid of the point passed in
isU – logical – .TRUE. if point passed in is on the U grid, .FALSE. if it is on the V grid
jjj – integer – j position on the grid of the point passed in
polydone – logical – .TRUE. if the current polygon has been finished, else .FALSE.
polynum – integer – number of the current boundary polygon
polystart – integer – location in **bnds** of the current polygon's first boundary point

B. Subroutine createBounds

Overview: This subroutine creates the model boundaries based on the land/sea masking of the rho grid and stores them in the variables **bnd_x**, **bnd_y**, **bx**, **by**, **hid**, **hx**, and **hy**.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **ui**, **uj**, **vi**, and **vj** from the Parameter Module, which contain the dimensions of the rho, u, and v grids.

Module procedures used: The subroutine uses the subroutines **getMask_Rho** and **getUVxy** from the Hydrodynamic Module.

Private Variables Used: This function uses the variables **bnds**, **bnum**, **BND_SET**, **bnd_x**, **bnd_y**, **bx**, **by**, **hid**, **hx**, **hy**, **land**, **maxbound**, **maxisland**, **nbounds**, and **numislands**, which are private variables accessible only to the procedures in the Boundary Module.

Numerical Method: This subroutine creates the land/sea boundaries based on the masking of the LTRANS Rho grid (see Fig. 2). Boundary points are located at U and V grid points, which lie directly between the Rho grid points. Any time adjacent Rho grid points are masked differently, i.e., one as water and one as land, the U or V grid point between those two points will be a boundary point. At open ocean boundaries near the edge of the model grid, the U or V grid point between the two outermost Rho grid points will be used for the LTRANS boundary point.

The subroutine assumes that there is only one body of water in the given masked Rho grid, i.e., land never separates two areas of water. On the other hand, multiple islands may exist within the body of water. When making boundaries, the subroutine traces around the body of water and then traces around any islands.

The subroutine first determines the number of boundary points in the given masked Rho grid. It does this by counting all the locations where either a land grid point and water grid point are adjacent (a land–water boundary) or the first and second points from the edge of the model grid are both masked as water (an open ocean boundary).

The subroutine then determines the ***element form*** of each Rho element. A Rho element is defined as a set of four adjacent Rho nodes that form a quadrilateral, as shown in the following illustration (Fig. 7):

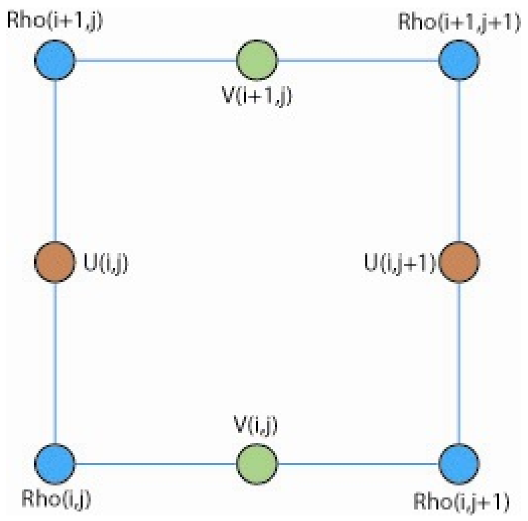

Fig. 7. This depicts an LTRANS Rho element. The four corners, in blue, are the four Rho nodes that comprise the Rho element. The U and V nodes, in orange and green respectively, lie directly between the Rho nodes. Their location is given in terms of i, position in the up/down direction, and j, position in the left/right direction.

The element form is determined based on which of its four nodes are masked as water and which are masked as land. The series of element forms is illustrated in Fig. 8.

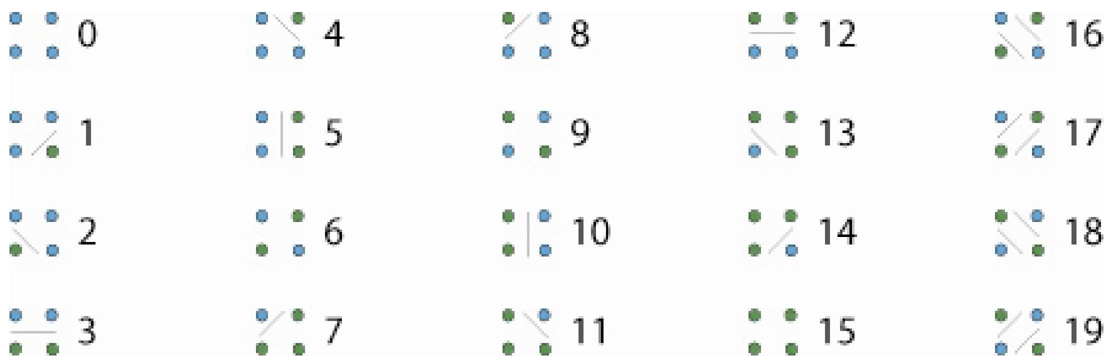

Fig. 8. Nineteen element forms used to create LTRANS model boundaries.

The blue circles represent nodes masked as water; the green circles represent nodes masked as land. The lines represent where the boundary lines will pass through each form. Forms 6 and 9 do not have lines because there are two possible ways for the boundaries to pass through them, as shown in forms 16 through 19. They are referred to as 'crosses' because either water crosses through land or land crosses through water in these elements. If an element with form 6 or 9 exists, the subroutine solves for the direction of the boundary to reclassify the cross to the appropriate form 16 though 19. It does this by testing which boundary direction makes a continuous boundary instead of leading to multiple unconnected islands.

Once all the forms are determined and crosses solved, the boundaries can be made. The subroutine searches through the elements starting in the lower left corner and takes the first element with water as the beginning of the boundary. Because the form and direction of entry of each boundary element have already been determined, the subroutine is able to trace around the edge of the water, adding boundary points in the order they are encountered, until it returns to the initial boundary point.

When the water boundary has been completed, the subroutine checks if all the boundary points have been included in the boundary. If they have not all been used, it means the grid contains islands, so the subroutine finds an element that contains an unused boundary and begins following the edge of the island in the same way it followed the water boundary. When the subroutine returns to the starting element of the island boundary, it again checks if all the boundary points have been included. The process is repeated until all the boundaries have been used (i.e., all islands have been created).

The subroutine must then link the boundaries that it has stored in **bnds** with the x- and y- coordinate locations of the U and V grid nodes and create the variables **bnd_x**, **bnd_y**, **bx**, **by**, **hid**, **hx**, and **hy**. To do this, the subroutine calls the function getUVxy, which gets the x- and y- coordinates of the U and V grids from the Hydrodynamic Module and stores them in the variables **x_u**, **x_v**, **y_u**, and **y_v**. To get the x- and y- coordinates of each boundary point, the subroutine converts the (i,j) location contained in **bnds** using either **x_u** and **y_u** or **x_v** and **y_v**, depending on whether the point is on the U or V grid (specified in **bnds**). The locations are stored in **bx** and **by** if they are the boundary points of the first polygon (the water boundaries) and **hx** and **hy** if they are boundary points of additional polygons (island boundaries). At the same time, the values in **hid** are filled with island id numbers, which start at 1001 for the first island. Additionally, while the boundary points are being specified in **bx** and **by**, they are also being assigned a logical value in the array **oo**, which contains .TRUE. for the boundary points that are classified as "open ocean" or .FALSE. if they're classified as land. The boundary checking subroutines expect closed polygons, so the last boundary point of each polygon is made identical to the first. Lastly, the variables **bnd_x** and **bnd_y** are filled with the x- and y- coordinates of the start and end points of every boundary segment in the model, and the values in **land** are specified; .TRUE. if either point is land, .FALSE. if both points are "open ocean."

Variables Definitions: The following variables are used in this section:

BND_SET – logical – set .TRUE. after the boundaries have been created

bnd_x – dp – x- coordinate of the start and end points of every boundary segment

bnd_y – dp – y- coordinate of the start and end points of every boundary segment

bnds – derived data type **boundary** – boundary point data; whether each point lies on the U grid (**onU**) (as opposed to the V grid), the point's location on the respective grid in terms of nodes from the left side (**ii**) and nodes from the bottom (**jj**), and the id number of the polygon (**poly**) that the boundary point is a part of

bnum – integer – total number of boundary points in **bnds**

bx – dp – x- coordinate of the main water boundary edge points

by – dp – y- coordinate of the main water boundary edge points

c – integer – counter used in filling **bx**, **by**, **hid**, **hx**, and **hy**

crossnum – integer – counter for number of unsolved crosses

deadend1 – logical – used when solving crosses, if the path that exited the cross from the left has reached a dead end **deadend1** is set to .TRUE. to make the subroutine stop searching the left path

deadend2 – logical – used when solving crosses, if the path that exited the cross from the right has reached a dead end **deadend2** is set to .TRUE. to make the subroutine stop searching the right path

dir – integer – direction the boundary left the previous element based on the numeric keypad (8-up, 6-right, 4-left, 2-down)

dir1 – integer – when solving crosses, the direction from which the path that exited the cross from the left exited the previous element; based on the numeric keypad

dir2 – integer – when solving crosses, the direction from which the path that exited the cross from the right exited the previous element; based on the numeric keypad

done – logical – .TRUE. if all the boundaries have been completed, else .FALSE.

ele – derived data type **element** – for each element, the form of that element (**form**) and whether or not the element has been used (**used** and **unused**). The variable **unused** is for elements that are 'cross' elements and have two boundaries that pass through them; it is .TRUE. if the cross element has never been used and .FALSE. if it has been used at least once. The variable **used** is for both regular and 'cross' elements; it is .FALSE. until an element has been completely used, i.e. used once by regular elements and used twice by 'cross' elements.

found – logical – set .TRUE. to indicate a starting point has been located

hid – dp – island id number assigned to each island edge point

hx – dp – x- coordinate location of island edge points

hy – dp – y- coordinate location of island edge points

i – integer – iteration variable

ipos – integer – position of current element in terms of elements from the left when making boundaries

ipos1 – integer – when solving crosses, current position of the path that exited left in terms of elements from the left

ipos2 – integer – when solving crosses, current position of the path that exited right in terms of elements from the left

j – integer – iteration variable

jpos – integer – position of current element in terms of elements from the bottom when making boundaries

jpos1 – integer – when solving crosses, current position of the path that exited left in terms of elements from the bottom

jpos2 – integer – when solving crosses, current position of the path that exited right in terms of elements from the bottom

k – integer – iteration variable

land – logical – for each boundary: .TRUE. if land boundary, .FALSE. if "open ocean"

m – integer – iteration variable

mask_rho – real – rho grid land/sea masking used to create boundaries

maxbound – integer – total number of boundary points surrounding the main body of water

maxisland – integer – total number of boundary points surrounding all the islands

nbounds – integer – total number of boundary segments

numislands – integer – total number of islands

numpoly – integer – total number of boundary polygons (water boundaries polygon plus the total number of island polygons)

oldcrossnum – integer – used when solving crosses to ensure that an endless loop is not encountered
oo – logical – for each boundary point: .TRUE. if "open ocean," .FALSE. if land
pend – integer – when filling the variables **bx**, **by**, **hid**, **hx**, and **hy**, location in **bnds** of the last boundary point of the current polygon
polydone – logical – .TRUE. if the current polygon has been finished, else .FALSE.
polysizes – integer – number of boundary points in each polygon
pstart – integer – when filling the variables **bx**, **by**, **hid**, **hx**, and **hy**, location in **bnds** of the first boundary point of the current polygon
STATUS – integer – used to test if allocation of variables was successful
U – logical – sent to subroutine .ADD. with the value .TRUE. to indicate the boundary point is on the U grid
ui – integer – number of nodes across u grid
uj – integer – number of nodes down rho and u grids
V – logical – sent to subroutine .ADD. with the value .FALSE. to indicate the boundary point is on the V grid
vi – integer – number of nodes across rho and v grids
vj – integer – number of nodes across u grid
wf – logical – short for 'waterfall'; indicates the boundary path is going through elements on the edge of the rho grid, i.e. the 'open ocean' boundary points
wf1 – logical – when solving crosses indicates the path that exited the cross from the left is going through elements on the edge of the rho grid
wf2 – logical – when solving crosses indicates the path that exited the cross from the right is going through elements on the edge of the rho grid
x_u – real – x- coordinates of u grid nodes
x_v – real – x- coordinates of v grid nodes
y_u – real – y- coordinates of u grid nodes
y_v – real – y- coordinates of v grid nodes

C. Subroutine getNext

Overview: This subroutine is called by createBounds to find the next element along the boundary when trying to determine the form of a 'cross' element.

Input Variables: The subroutine has just one variable that is used only for input, **form**, which contains an integer that represents the form of the current element.

Input/Output Variables: The subroutine has four variables used for input and output. The variables **i** and **j** contain the location of the Rho element in which the boundary exists. The variable **wf** is a logical variable that is .TRUE. if the boundary is currently an open ocean boundary. Lastly, the variable **dir** contains an integer that indicates the direction from which the boundary exited the previous element.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **uj** and **vi** from the Parameter Module, which contain the dimensions of the rho grid.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses no private variables.

Numerical Method: This subroutine finds the next element, following the bounds clockwise around water. This is accomplished based on the element's location (**i**, **j**), whether or not the element is on the edge of the rho grid (**wf**), the direction the path is coming from (**dir**), and the form of the current element (**form**). The subroutine simply finds the code associated with the given combination of **wf**, **dir**, and **form** and updates the element's location (**i**, **j**), **wf**, and **dir** values appropriately.

Variable Definitions: The following variables are used in this subroutine:

- **dir** – integer – direction the boundary left the previous element based on the numeric keypad (8-up, 6-right, 4-left, 2-down)
- **form** – integer – value from 0 to 19 that represents the form of the current element, based on which of its four nodes are masked as water and which are masked as land
- **i** – integer – position of current element in terms of elements from the left
- **j** – integer – position of current element in terms of elements from the bottom
- **wf** – logical – .TRUE. if the element is on the edge of the Rho grid, else .FALSE.

D. Subroutine ibounds

Overview: Subroutine ibounds uses the point-in-polygon approach to determine if a particle is inside one of the islands within the model domain.

Input Variables: This subroutine has two input variables: the particle's x- and y- location (**clongx**, **claty**).

Output Variables: This subroutine has two output variables. The variable **in_island** contains 1 if the particle was found within island boundaries and 0 if it was not. The variable **island** contains the id number of the island that contains the particle, if applicable.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses the function **inpoly** from the Point-in-Polygon Module.

Private Variables Used: This function uses the variables **hid**, **hx**, **hy**, **maxisland**, and **numislands**, which are private variables accessible only to procedures in the Boundary Module.

Initialization: The variable **island** is initialized to 0.0 to avoid returning an id accidentally passed to the subroutine. The variable **in_island** is also initialized to 0 to prevent it from being passed a 1 accidentally when the particle is not on an island. The variable **i** is initialized to 1 to be used as a counter to iterate through the island boundaries. The variable **isle** is initialized to the island id of the first island. Lastly, **start** is initialized to 0, because it contains the array location of the first island boundary point of the current island minus 1.

Numerical Method: This subroutine first checks if there are any islands in the model. If there are no islands in the model, the subroutine skips to the end and returns the initialized zero values of **in_island** and **island**. Otherwise, the subroutine iterates through the island boundary points until it reaches the end of an island. At this point the array **isbnds** is allocated to the number of boundary points in that island and the location of the boundary points is read into **isbnds**. The particle location and island boundaries are passed to the Point-in-Polygon Module subroutine **inpoly** to determine if the particle is inside the island's boundaries. As long as **inpoly** returns .FALSE., indicating that the point is not in the polygon, the subroutine will continue until it tests all of the islands. If at any point **inpoly** returns .TRUE., the island id is stored in the output variable **island** and **in_island** is set to 1, to indicate that the particle is in an island. Otherwise, the initial zero values of **in_island** and **island** are returned.

Variable Definitions: The following variables are used in this subroutine:

claty – dp – particle's y- coordinate location
clongx – cp – particle's x- coordinate location
count – integer – number of boundary points in the current island
hid – dp – island id number of each island edge point
hx – dp – x- coordinate location of island edge points
hy – dp – y- coordinate location of island edge points
i – integer – iteration variable
in_island – integer – return variable; 1 if particle is in an island, else 0
isbnds – dp – array allocated to the number of edge points in the current island and filled with the x- and y- coordinates of the island edge points; to be passed to **inpoly**
island – dp – return variable; island id number if particle is in an island, else 0
isle – dp – island id number of current island being checked
j – integer – iteration variable
maxisland – integer – total number of island edge points
numislands – integer – total number of islands
start – integer – keeps track of the first location in **hid**, **hx**, and **hy** of the current island

E. Subroutine intersect_reflect

Overview: Subroutine **intersect_reflect** calculates the intersection between the particle trajectory and the boundary line in a grid cell and then calculates the reflection, returning the new particle location.

Input Variables: This subroutine has four variables used only as input: the location of the particle before movement (**Xpos**, **Ypos**) and the location of the particle after movement (**nXpos**, **nYpos**).

Output Variables: The subroutine has five variables used solely for output. The variables **fintersectX** and **fintersectY** contain the location at which the particle's trajectory intersects the edge boundary. The variables **freflectX** and **freflectY** contain the location of the particle after it reflects off the boundary. Lastly, **intersectf** contains a 1 if an intersection occurs or a 0 if it does not.

Input/Output Variables: The variable **skipbound** is used for both input and output. The value input through **skipbound** indicates that a certain boundary that should be skipped when the subroutine loops through the boundaries searching for an intersection. The subroutine can also output a value in **skipbound** to be used as input for the subroutine if it is not reset before its next call. Lastly, there is the optional output variable **isWater** which contains the value .TRUE. if the intersected boundary is an "open ocean" boundary, or the value .FALSE. if the intersected boundary is a land boundary.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variables **bnd_x**, **bnd_y**, **land**, and **nbounds**, which are private variables accessible only to the procedures in the Boundary Module.

Initialization: The variables **distBC**, **Mbc**, **Bbc**, **Mp**, and **Bp** are initialized to 0.0. The variables **intersect** and **intersectf** are initialized to 0. Lastly, **fintersectX**, **fintersectY**, **freflectX**, **freflectY**, and **dtest** are all initialized to -999999.0.

Numerical Method: The subroutine first determines the x and y limits of the particle trajectory. The higher of the two values in **Xpos** and **nXpos** is stored in **xhigh**, and the lower is stored in **xlow**. The same is done for **Ypos** and **nYpos**, storing the limits in the variables **yhigh** and **ylow**. This allows the program to check that the intersection, if one occurs, is between the limits in the x and y directions.

The subroutine then enters a loop that iterates through each individual boundary segment of the model domain. To save time, the subroutine first checks if the boundary segment end points are both to the north, south, east, or west of the particle trajectory start and end points. If so, the boundary is skipped because the particle cannot possibly cross the boundary. Otherwise, **bxhigh**, **bxlow**, **byhigh**, and **bylow** are determined for the boundary segment in the same way that **xhigh**, **xlow**, **yhigh**, and **ylow** were determined for the particle trajectory.

Since vertical lines have an undefined slope, they must be handled separately. The subroutine therefore checks if either the particle trajectory or boundary segment is vertical. If either is vertical, one of four scenarios takes place: 1) the particle trajectory is vertical and the boundary is horizontal, 2) the boundary is vertical and the particle trajectory is horizontal, 3) the particle trajectory is vertical and the boundary is not horizontal, or 4) the boundary is vertical and the particle trajectory is not horizontal.

For the scenarios in which either the boundary or the trajectory is vertical and the other is horizontal, the place where the lines will intersect is already known. If the intersection takes place between the endpoints of the boundary segment and within the trajectory of the particle, the reflection goes straight back in the direction from which it came. If the intersection does not take place within those requirements, the subroutine moves on to check the next boundary line.

In the situations where either the boundary or the trajectory is vertical and the other is not horizontal, an extra step is involved. The x-coordinate of the point of intersection is known (it is the x-coordinate of the vertical line), but the y-coordinate must be calculated. The slope and intercept are calculated, and then the x-coordinate of the vertical line is used to solve for y. The subroutine then checks if the point of intersection lies between the endpoints of the trajectory and boundary segment and, if it is, calculates the reflection. The reflection is on the line perpendicular to the boundary line that goes through the point (**nXpos**,**nYpos**) and is the same distance from the boundary line as that point but on the other side of the boundary line (see illustration below). If the intersection does not take place within the requirements, the subroutine moves on to check the next boundary line.

If neither the boundary segment nor particle trajectory is vertical, the subroutine calculates the slope and x-intercept of the particle trajectory and boundary segment and the point at which they intersect. If this point of intersection occurs between the boundary segment endpoints, the reflected location is calculated. The reflection lies on the line perpendicular to the boundary line that passes through (**nXpos**,**nYpos**), at the same distance from the boundary line as the original projected particle endpoint (see Fig. 7). The line perpendicular to the boundary line through the point (**nXpos**,**nYpos**) and the distance from the point (**nXpos**,**nYpos**) to the boundary line are calculated. Because the reflected point is the same distance from the boundary line, it is calculated as the point two times that distance from the point (**nXpos**,**nYpos**). Since this description yields two points, the one closest to the boundary line is used.

If the boundary line is horizontal (east-west), the line perpendicular to it will be vertical and have an undefined slope. It is therefore handled separately, and in fact it has a simple solution because the x-location of the reflection is the same as that of **nXpos**.

For all of the boundary lines for which an intersection did not occur within the segment endpoints, the subroutine loops back to check the next boundary segment. However, if an intersection did occur, the distance from the particle's start location to its point of intersection is calculated. If there have been no other boundary segments with an intersection closer, the coordinates of the intersection and the reflected location are stored in the appropriate output variables along with whether or not the boundary is a land or "open ocean" boundary if the output variable **isWater** is present. Also, **intersectf** is set equal to 1 to indicate that an intersection has occurred, and **skipbound** is set equal to the identification number of the boundary that was intersected so that it will be ignored when the next call to the subroutine checks that the reflected trajectory has not intersected another boundary.

Even if an intersection is found, the subroutine continues to check the other boundaries in case there is a closer point of intersection, since it is possible for a particle's trajectory to pass through a segment to go out of bounds and then through another one to come back in. Once the proper reflected location has been calculated, the subroutine returns the values **fintersectX**, **fintersectY**, **freflectX**, and **freflectY** shown in Fig. 9. The new particle trajectory, from the intersect point to the reflected location, is then tested to make sure it does not intersect with another boundary.

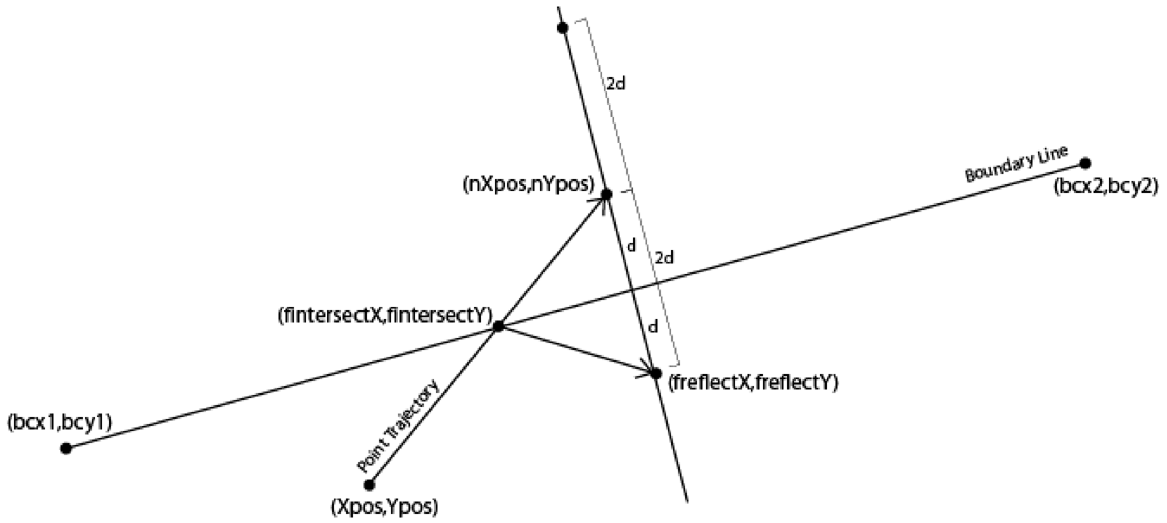

Fig. 9. Schematic of line segments used to calculate the intersection and reflection of particle trajectories when the boundary and/or particle trajectory lines are not horizontal or vertical.

Variable Definitions: The following variables are used in this subroutine:

Bbc - dp – x intercept of current boundary line
bBCperp - dp – x intercept of the line perpendicular to the current boundary line, going through the projected location of the particle (nXpos, nYpos)
bcx1 - dp – x-position of 1st endpoint of current boundary segment
bcx2 - dp – x-position of 2nd endpoint of current boundary segment
bcy1 - dp – y-position of 1st endpoint of current boundary segment
bcy2 - dp – y-position of 2nd endpoint of current boundary segment
bnd_x(2,nbounds) - dp – x-coordinates of the boundary points
bnd_y(2,nbounds) - dp – y-coordinates of the boundary points
Bp - dp – x intercept of the particle trajectory line
bxhigh - dp – contains the higher x-coordinate of the two endpoints of the current segment
bxlow - dp – contains the lower x-coordinate of the two endpoints of the current segment
byhigh - dp – contains the higher y-coordinate of the two endpoints of the current segment
bylow - dp – contains the lower y-coordinate of the two endpoints of the current segment
crossk - dp – cross product for determining distance of particle from boundary
d_Pinter - dp – distance from particle start location to the point of intersection with the current boundary line
dist1 - dp – distance from the point of intersection to the first of two possible reflection locations

dist2 - dp – distance from the point of intersection to the second of two possible reflection locations
distBC - dp – length of current boundary segment
dPBC - dp – distance from projected particle location (nXpos, nYpos) to boundary line
dtest - dp – distance from particle start location to the nearest encountered point of intersection
fintersectX - dp – x-position of the nearest encountered point of intersection
fintersectY - dp – y-position of the nearest encountered point of intersection
freflectX - dp – x-position of reflected location resulting from the nearest encountered point of intersection
freflectY - dp – y-position of reflected location resulting from the nearest encountered point of intersection
i - integer – used to iterate through the boundary segments
intersctx - dp – x-position at which particle trajectory intersects current boundary segment
interscty - dp – y-position at which particle trajectory intersects current boundary segment
intersect - integer – flag integer; 0 if particle trajectory does not intersect current boundary segment, 1 if it does
intersectf - integer – return variable; returns 1 if an intersection occurred, 0 if not
isWater – logical – optional return variable; .TRUE. if intersected boundary is "open ocean", .FALSE. if it is land
Mbc - dp – slope of current boundary segment
mBCperp - dp – slope of line perpendicular to current boundary segment
Mp - dp – slope of particle trajectory
nbounds - integer – total number of boundary segments
nXpos - dp – projected new Xpos if no intersections occur
nYpos - dp – projected new Ypos if no intersections occur
rPxyzX - dp – reflected x-position after intersecting current boundary segment
rPxyzY - dp – reflected y-position after intersecting current boundary segment
rx1 - dp – x-position of first of two possible reflection locations
rx2 - dp – x-position of second of two possible reflection locations
ry1 - dp – y-position of first of two possible reflection locations
ry2 - dp – y-position of second of two possible reflection locations
skipbound - integer – outputs the number of the boundary segment intersected, so that it can be used as input on successive calls to the subroutine to skip that boundary segment
skipboundi - integer – boundary number with the closest encountered point of intersection
xhigh - dp – the higher x-position between the particles old position and projected new position
xlow - dp – the lower x-position between the particles old position and projected new position
Xpos - dp – x-position of particle before movement
yhigh - dp – the higher y-position between the particle's old position and projected new position
ylow - dp – the lower y-position between the particle's old position and projected new position
Ypos - dp – y-position of particle before movement

F. Function isBndSet

Overview: The function isBndSet returns the value of **BND_SET**, which is .TRUE. if the boundaries have been created and .FALSE. if they have not yet been created..

Input Variables: The function has no variables used for input.

Output: The function returns .TRUE. if the boundaries have been created and .FALSE. if they have not yet been created.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variable **BND_SET**, which is a private variable accessible only to the procedures in the Boundary Module.

Numerical Method: The function simply sets the return variable **isBndSet** to the logical value of **BND_SET**.

Variable Definitions: The following variables are used in this function:
BND_SET – logical – .TRUE. if the boundaries have been created, else .FALSE.

G. Subroutine mbounds

Overview: Subroutine mbounds uses the point-in-polygon approach to determine if a particle is inside the model domain. The basic concept behind the point-in-polygon approach is that a ray shot out from a point in a polygon will cross an odd number of edges if it is within the polygon and an even number if it is not.

Input Variables: This subroutine has two variables used only as input: the x- and y-coordinate location of the particle (**Xpos**, **Ypos**).

Output Variables: The output variable **inbounds** returns a 1 if the particle is found to be within the domain boundaries and a 0 if it is not.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses the function **inpoly** from the Point-in-Polygon Module.

Private Variables Used: This function uses the variables **bx**, **by**, and **maxbound**, which are private variables accessible only to the procedures in the Boundary Module.

Numerical Method: This subroutine basically reformats the main water boundary edge point coordinates into one variable (**blatlon**), which can be passed to the function **inpoly** along with the particle's x- and y- coordinates (**Xpos**, **Ypos**). The output variable **inbounds** is initialized to zero, and if **inpoly** returns .TRUE., indicating that the particle is inside the water boundaries, the value in **inbounds** is updated to 1.

Variable Definitions: The following variables are used in this subroutine:

bx – dp – x- coordinate of the water edge points
by – dp – y- coordinate of the water edge points
blatlon – dp – x- and y- coordinates of the water edge points, formatted for **inpoly**
i – integer – iteration variable
inbounds – integer – output variable; returns 1 if particle is in the water, else 0
maxbound – integer – total number of water edge points
Xpos – dp – particle's x- coordinate
Ypos – dp – particle's y- coordinate

H. Subroutine output_llBounds

Overview: Subroutine **output_llBounds** takes the boundaries stored in the variable **bnds** and converts them into blanking files (.bln) for Surfer/Scripter using latitude and longitude coordinates.

Input Variables: This subroutine has no variables used as input.

Output Variables: This subroutine has no variables used as output

Module parameters used: The subroutine uses the parameters **ui**, **uj**, **vi** and **vj** from the Parameter Module, which contain the dimensions of the u and v grids.

Module procedures used: The subroutine uses the subroutine **getUVxy** from the Hydrodynamic Module. It also uses the functions **x2lon** and **y2lat** from the Conversion Module.

Private Variables Used: This subroutine uses the variables **bnds** and **bnum**, which are private variables accessible only to the procedures in the Boundary Module.

Numerical Method: This subroutine first calls **getUVxy** to get the x- and y- coordinates of the u and v grid nodes. It then calculates the number of boundary edge points in each boundary polygon. If there are no islands in the model domain, then there will only be one boundary polygon. The program then iterates through the boundary polygons and for each polygon writes a row containing the number of edge points in that polygon (plus one to close the polygon) and the number 1 to indicate a closed polygon, followed by rows containing the longitude and latitude of each boundary edge point in that polygon, ending with the longitude and latitude of the first edge point again in order to close the polygon.

Variable Definitions: The following variables are used in this subroutine:
- **i** – integer – iteration variable
- **j** – integer – iteration variable
- **k** – integer – iteration variable
- **m** – integer – iteration variable
- **numpoly** – integer – number of polygons
- **pend** – integer – location in **bnds** of the current polygon's last edge point
- **polysizes** – integer – the number of boundary edge points in each polygon
- **pstart** – integer – location in **bnds** of the current polygon's first edge point
- **STATUS** – integer – status ID returned from intrinsic function allocate
- **x_u** – real – x- coordinates of the u grid nodes
- **x_v** – real – x- coordinates of the v grid nodes
- **y_u** – real – y- coordinates of the u grid nodes
- **y_v** – real – y- coordinates of the v grid nodes

I. Subroutine output_xyBounds

Overview: Subroutine **output_xyBounds** takes the boundaries stored in the variable **bnds** and converts them into blanking files (.bln) for Surfer/Scripter using x- and y- coordinates.

Input Variables: This subroutine has no variables used as input.

Output Variables: This subroutine has no variables used as output

Module parameters used: The subroutine uses the parameters **ui**, **uj**, **vi** and **vj** from the Parameter Module, which contain the dimensions of the u and v grids.

Module procedures used: The subroutine uses the subroutine **getUVxy** from the Hydrodynamic Module.

Private Variables Used: This subroutine uses the variables **bnds** and **bnum**, which are private variables accessible only to the procedures in the Boundary Module.

Numerical Method: This subroutine first calls **getUVxy** to get the x- and y- coordinates of the u and v grid nodes. It then calculates the number of boundary edge points in each boundary polygon. If there are no islands in the model domain, then there will only be one boundary polygon. The program then iterates through the boundary polygons and for each polygon writes a row containing the number of edge points in that polygon (plus one to close the polygon) and the number 1 to indicate a closed polygon, followed by rows containing the x- and y- coordinates of each boundary edge point in that polygon, ending with the x- and y- coordinates of the first edge point again in order to close the polygon.

Variable Definitions: The following variables are used in this subroutine:

i – integer – iteration variable
j – integer – iteration variable
k – integer – iteration variable
m – integer – iteration variable
numpoly – integer – number of polygons
pend – integer – location in **bnds** of the current polygon's last edge point
polysizes – integer – the number of boundary edge points in each polygon
pstart – integer – location in **bnds** of the current polygon's first edge point
STATUS – integer – status ID returned from intrinsic function allocate
x_u – real – x- coordinates of the u grid nodes
x_v – real – x- coordinates of the v grid nodes
y_u – real – y- coordinates of the u grid nodes
y_v – real – y- coordinates of the v grid nodes

VIII. Conversion Module (conversion_module.f90, CONVERT_MOD)

Overview: The Conversion Module contains all the procedures necessary to convert locations between latitude and longitude coordinates and metric (x and y) coordinates. There are two types of conversions available in LTRANS: spherical and mercator. Both conversions are calculated using code based on equations from Seagrid which is used to generate ROMS model grids (Seagrid was developed by Rich Signell, USGS. See http://woodshole.er.usgs.gov/operations/modeling/seagrid/seagrid.html for more information). The equations used in the spherical conversion are from earthdist.m. The mercator conversions are calculated using equations from sg_mercator.m and seagrid2roms.m. Note that when ROMS is run in spherical mode, it assumes that the distance between points is calculated on a spherical earth, but it does not do calculations in projected coordinates. So, although LTRANS uses the same assumption about the shape of the earth as ROMS, LTRANS is not doing the same calculations as ROMS (Rich Signell, pers. com.).

The Conversion Module contains four interface blocks with two functions each. One of the two functions accepts input of type real, and the other accepts input of type double precision. Both functions return double precision output. The four interface blocks cover the four necessary conversions: longitude to x- coordinate, latitude to y- coordinate, x- coordinate to longitude, and y- coordinate to latitude.

Module Parameters Used: The Conversion Module uses five parameters that are set in the LTRANS.data file. These include the Earth's equatorial radius (**Earth_Radius**), minimum latitude and longitude values (**latmin** and **lonmin**) to base spherical projections off of, the value of the mathematical constant π (**PI**), and the logical variable **SphericalProjection** which specifies which conversions the model will use (.TRUE. = Spherical; .FALSE. = Mercator).

Public Procedures: The following are the public interfaces contained within the Conversion Module: **lat2y, lon2x, x2lon, y2lat.**

A. Interface lat2y

Overview: Interface **lat2y** contains two functions, **rlat2y** and **dlat2y**, which both convert latitude to y-coordinates. **rlat2y** accepts latitude in type real and **dlat2y** accepts latitude in type double precision, though both return the y- coordinate in the type double precision.

Input Variables: This interface has one input variable, the latitude to be converted (**lat**).

Output: The interface outputs the converted y- coordinate.

Numerical Method: If spherical projection is specified, y is calculated as the arclength of the central angle between the two vectors [ax,ay,az] and [bx,by,bz] multiplied by the Earth's radius. The interface uses the following function to convert latitude to metric y- coordinates and then returns the value of y:

$$y = a\cos(ax * bx + ay * by + az * bz) * Earth\_Radius$$

If mercator projection is specified, the interface uses the following function to convert latitude to metric y- coordinates and then returns the value of y:

$$y = \log\left(\tan\left(\frac{\pi}{4} + \frac{lat}{2RCF}\right)\right) * Earth\_Radus$$

Variable Definitions: The following variables are used in this interface:

ax – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the x-coordinate of the point at (alon,alat)
ay – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the y-coordinate of the point at (alon,alat)
az – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the z-coordinate of the point at (alon,alat)
bx – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the x-coordinate of the point at (blon,blat)
by – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the y-coordinate of the point at (blon,blat)
bz – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the z-coordinate of the point at (blon,blat)
c – dp – if the Earth were a sphere with a radius of 1, this value contains the radius of the circle that would be created by cutting the sphere at the given latitude
alat – dp – latitude in radians of the point (lonmin,lat)
alon – dp – longitude in radians of the point (lonmin,lat)
blat – dp – latitude in radians of the point (lonmin,latmin)
blon – dp – longitude in radians of the point (lonmin,latmin)
Earth_Radius – dp – Earth's equatorial radius
lat – dp – latitude that needs to be converted
PI – dp – the mathematical constant π
RCF – dp – radian conversion factor

B. Interface lon2x

Overview: Interface lon2x contains two functions, rlon2x and dlon2x, which both convert longitude to x- coordinates. rlon2x accepts longitude in type real and dlon2x accepts longitude in type double precision, though both functions return the x- coordinate in the type double precision.

Input Variables: This interface has one required input variable, the longitude to be converted (**lon**), and one optional input variable necessary for spherical projection, the latitude at which the longitude is to be converted (**lat**).

Output: The interface outputs the converted x- coordinate.

Numerical Method: If spherical projection is specified, x is calculated as the arclength of the central angle between the two vectors [ax,ay,az] and [bx,by,bz] multiplied by the Earth's radius. The interface uses the following function to convert longitude and latitude to metric x-coordinates and then returns the value of x:

$$x = a\cos(ax * bx + ay * by + az * bz) * Earth\_Radius$$

If mercator projection is specified, the interface uses the following function to convert longitude to metric x- coordinates and then returns the value of x:

$$x = \frac{lon}{RCF} * Earth\_Radus$$

Variable Definitions: The following variables are used in this interface:

ax – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the x-coordinate of the point at (alon,alat)
ay – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the y-coordinate of the point at (alon,alat)
az – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the z-coordinate of the point at (alon,alat)
bx – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the x-coordinate of the point at (blon,blat)
by – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the y-coordinate of the point at (blon,blat)
bz – dp – if the Earth were a sphere with a radius of 1 and centered at [0,0,0], the z-coordinate of the point at (blon,blat)
c – dp – if the Earth were a sphere with a radius of 1, this value contains the radius of the circle that would be created by cutting the sphere at the given latitude
alat – dp – latitude in radians of the point (lon,lat)
alon – dp – longitude in radians of the point (lon,lat)
blat – dp – latitude in radians of the point (lonmin,lat)
blon – dp – longitude in radians of the point (lonmin,lat)
Earth_Radius – dp – Earth's equatorial radius
lon – dp – longitude that needs to be converted
PI – dp – the mathematical constant π
RCF – dp – radian conversion factor

C. Interface x2lon

Overview: Interface x2lon contains two functions, rx2lon and dx2lon, which both convert x-coordinates to longitude. rx2lon accepts the x- coordinate in type real and dx2lon accepts the x-coordinate in type double precision, though both functions return the longitude in the type double precision.

Input Variables: This interface has one input variable, the x- coordinate to be converted (**x**).

Output: The interface outputs the converted longitude.

Numerical Method: If spherical projection is specified, latitude is calculated as specified in the interface y2lat using the metric y- coordinate and stored in the variable **lat**. Then the interface uses the following function to convert the metric x- coordinate to longitude and then returns the value of lon

$$lon = lon\min + RCF * a\cos\left(\frac{\cos(x / Earth\_Radius) - \sin(lat)^2}{\cos(lat)^2}\right)$$

If mercator projection is specified, the interface uses the following function to convert x- coordinates to longitude and then returns the value of lon:

$$lon = \frac{x}{Earth\_Radus} * RCF$$

Variable Definitions: The following variables are used in this interface:

- **Earth_Radius** – dp – Earth's equatorial radius
- **lat** – dp – latitude of the y- coordinate used in spherical projection
- **RCF** – dp – radian conversion factor
- **x** – dp – x- coordinate that needs to be converted
- **y** – dp – y- coordinate associated with the x- coordinate that needs to be converted

D. Interface y2lat

Overview: Interface y2lat contains two functions, ry2lat and dy2lat, which both convert y- coordinates to latitude. ry2lat accepts y- coordinates in type real and dy2lat accepts y- coordinates in type double precision, though both functions return the latitude in the type double precision.

Input Variables: This interface has one input variable, the y- coordinate to be converted (**y**).

Output: The interface outputs the converted latitude.

Numerical Method: If spherical projection is specified, the interface uses the following function to convert y- coordinates to latitude and then returns the value of lat:

$$lat = lat\min + \frac{y * RCF}{Earth\_Radius}$$

If mercator projection is specified, the interface uses the following function to convert y-coordinates to latitude and then returns the value of lat:

$$lat = 2RCF * \left(\arctan\left(e^{\frac{y}{Earth\_Radius}} \right) - \frac{\pi}{4} \right)$$

Variable Definitions: The following variables are used in this interface:

Earth_Radius – dp – Earth's equatorial radius
y – dp – y- coordinate that needs to be converted
PI – dp – the mathematical constant π
RCF – dp – radian conversion factor

IX. Gridcell Module (gridcell_module.f90, GRIDCELL_MOD)

Overview: The Gridcell Module contains the subroutine **gridcell** which is used for determining if a point lies within an element.

Public Procedures: The following are the public subroutines and functions contained within the Gridcell Module: subroutine **Gridcell**.

A. Subroutine Gridcell

Overview: Subroutine gridcell serves two purposes. When passed an element number through the optional argument **checkele**, it determines if a particle is in that particular element. When the optional argument is omitted, the subroutine determines in which element the particle is currently located.

Input Variables: This subroutine requires five parameters as input variables: the total number of wet elements in the given grid (**elements**), the x- and y- locations of the four nodes in each of the elements (**ele_x**, **ele_y**), and the x- and y- position of the particle (**Xpos**, **Ypos**). If only one element needs to be checked, the subroutine takes one additional parameter, **checkele**, which contains the element number to be checked.

Output Variables: The subroutine has two output parameters. **P_ele** returns the element number in which the particle is found, if the particle is found. The variable **triangle** returns a 1 if the particle was found in an element and a 0 if it was not.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses no private variables.

Numerical Method: The subroutine first checks whether or not the optional argument **checkele** was present in the function call. If it is present, then the variables **elestart** and **eleend** are both initialized to the value in **checkele**. If it is not present, **elestart** is initialized to 1 and **eleend** is initialized to the value in **elements**. The main loop in the subroutine iterates from **elestart** to **eleend**, so if **checkele** is present the subroutine will only check the element specified in **checkele**; otherwise, all the elements are checked.

In each iteration, the subroutine first checks if the particle is completely north, south, east, or west of every node of the element. If it is, the particle is not in the element and the program moves on to the next element. If it is not, the subroutine then checks if the particle is directly on the element's boundary points or on a horizontal boundary segment. If it is, the particle is considered in the element, and the subroutine ends the loop and returns the current element number.

If the element does not pass either of those checks, the subroutine uses a point-in-polygon approach to determine if the particle is in the element. To do this, it loops through each of the element's four boundary segments. If a ray shot east from the particle goes through the current boundary segment, the corresponding position in the array **counter** is changed to 1 (**counter** has four positions, one associated with each boundary segment, initialized to 0). After the subroutine has iterated through each boundary segment, the values in **counter** are summed and stored in **total**. In this case, if the particle is in the element, the ray will have passed through only one of the boundary segments; if the particle is not in the element, the ray will have passed through either two or none of the segments. Thus, if **total** contains an odd number, the program considers the particle in the element and ends the loop, returning the element number. If **total** contains an even number, the particle is not in the element and the subroutine continues to the next element.

Variable Definitions: The following variables are used in this subroutine:

- **bhighy** - dp – y-position of the highest boundary point
- **blowy** - dp – y-position of the lowest boundary point
- **bx1** - dp – x-position of the 1st end point of the current boundary segment
- **bx2** - dp – x-position of the 2nd end point of the current boundary segment
- **by1** - dp – y-position of the 1st end point of the current boundary segment
- **by2** - dp – y-position of the 2nd end point of the current boundary segment
- **checkele** – integer – optional input argument that contains an element number and, when present, prompts the subroutine to check only that one element
- **counter**(4) - integer – array with a position associated with each boundary segment; each position contains 1 if a ray shot east from the particle passes through its associated boundary segment or 0 if it does not
- **ele_x**(4,elements) - dp – x-positions of all four nodes in every element
- **ele_y**(4,elements) - dp – y-positions of all four nodes in every element
- **eleend** – integer – number of the last element to check
- **elements** - integer – total number of elements
- **elestart** – integer – number of the first element to check
- **i** - integer – used to iterate through the elements
- **p** - integer – used to iterate through the four line segments of the element
- **P_ele** - integer – returns the ID number of the element that contains the particle
- **slope** - dp – slope of the current line segment
- **total** - integer – sum of the values in **counter**; if it is odd then the particle is in the element
- **triangle** - integer – return variable; 1 if particle is found to be in an element, 0 if not
- **xintersect** - dp – x intersect of current line segment
- **Xpos** - dp – x-position of the particle
- **Ypos** - dp – y-position of the particle

X. Horizontal turbulence Module (hor_turb_module.f90, HTURB_MOD)

Overview: Hydrodynamic models do not simulate turbulent motion at scales smaller than the grid resolution of the model (e.g., 1 km). In particle-tracking models, however, particles are moved in millimeter to centimeter steps—much smaller than the hydrodynamic model grid scale. It is necessary to add a random component to particle motion in order to reproduce turbulent diffusion that occurs at the scale of particle motion (Visser 1997, Brickman and Smith 2001). A random walk model is used to simulate turbulent particle motion in the horizontal direction (x- and y- directions) because LTRANS was developed to use output from a hydrodynamic model with constant horizontal diffusivity (Li et al. 2005). For hydrodynamic models with variable horizontal diffusivity, a random displacement model (Visser 1997) should be used. See Vertical Turbulence Module section for an example of a random displacement model.

Public Procedures: The following are the public subroutines and functions contained within the Horizontal Turbulence Module: **subroutine HTurb.**

A. Subroutine HTurb

Overview: This subroutine calculates the horizontal turbulence in the x- and y- directions.

Input Variables: The subroutine HTurb has no input variables.

Output Variables: The subroutine returns the horizontal displacement (m) in the x-and y- directions during one internal time step through the variables **TurbHx** and **TurbHy**.

Module parameters used: The subroutine uses the parameters **ConstantHTurb** and **idt** from the Parameter Module, which contain the value of constant horizontal diffusivity and the size of the internal time step respectively.

Module procedures used: The subroutine uses the function **norm** from the Norm Module.

Private Variables Used: The subroutine uses no private variables.

Initialization: This module must be 'turned on' in the LTRANS.data include file by setting the parameter **HTurbOn** = .TRUE. In addition, the constant value of horizontal diffusivity must be set in LTRANS.inc in the parameter **ConstantHTurb**.

Numerical Method: When horizontal diffusivity is constant, the random displacement model defaults to a random walk model (Visser 1997):

$$x_{n+1} = x_n + R\left[2r^{-1}K\delta t\right]^{1/2}$$

where K = horizontal diffusivity evaluated at (x_n). For the LTRANS development, K was equal to 1 m^2 s^{-1}, the same constant horizontal diffusivity that was used in the ROMS model of Chesapeake Bay (Li et al. 2005).

Variable Definitions: The following variables are used in this section:

- **ConstantHTurb** – dp, parameter – constant horizontal diffusivity
- **devX** – dp - the random deviate in the x-direction
- **devY** – dp - the random deviate in the y-direction
- **idt** – integer – length of internal (particle tracking) time step (s)
- **r** – dp – the standard deviation of the random deviate
- **TurbHx** – dp - displacement in x-direction due to horizontal turbulence during internal time step
- **TurbHy** – dp - displacement in y-direction due to horizontal turbulence during internal time step

XI. Hydrodynamic Module (hydrodynamic_module.f90, HYDRO_MOD)

Overview: The Hydrodynamic Module handles code related to the NetCDF hydrodynamic model input files, as well as the Rho, U, and V grid elements created based on the input from those files.

Private Variables: The module contains seventy seven variables accessible only in this module and the subroutines and functions within it:

- **COUNTr** – integer – array of integers specifying the number of indices to read in along each dimension; used when reading in one time step worth of data (excludes zeta data)
- **COUNTz** – integer – array of integers specifying the number of indices to read in along each dimension; used when reading in one time step worth of zeta data
- **CS** – real – s-level stretching curves for the Rho grid
- **CSW** – real – s-level stretching curves for the W grid
- **depth** – real – sea floor depth at each Rho node location
- **filenm** – character array – concatenated hydrodynamic input file name
- **GRD_SET** – logical – set .TRUE. when the grid has been read in, else .FALSE.
- **iint** – integer – keeps track of which input file to open (0 = file 1, 1 = file 2, etc.)
- **m_r** – dp – land/sea masking of the Rho grid in (i,j) location format
- **m_u** – dp – land/sea masking of the U grid in (i,j) location format
- **m_v** – dp – land/sea masking of the V grid in (i,j) location format
- **mask_rho** – real – land/sea masking of the Rho grid in (i,j) location format
- **NCcount** – integer – when writing to multiple NetCDF files, this stores the current NetCDF file number used for the filename
- **NCstart** – integer – the time in the model that the current NetCDF output file was created; needed to determine if **NCtime** seconds have passed and a new output file needs to be created
- **P_r_element** – integer – Rho element that each particle is in
- **P_u_element** – integer – U element that each particle is in
- **P_v_element** – integer – V element that each particle is in
- **prcount** – integer – time step within the current NetCDF output file
- **r_Adjacent** – integer – array containing, for each element, its own Rho element id followed by the element ids of all the Rho elements that share a node with that element
- **r_ele_x** – dp – x-coordinate location of the four nodes in each wet Rho element
- **r_ele_y** – dp – y-coordinate location of the four nodes in each wet Rho element
- **RE** – integer – the four Rho node numbers that make up each wet Rho element
- **rho_angle** – dp – angle between Rho node's x-coordinate and true east direction (radian)
- **rnode1** – integer – 1st of 4 Rho nodes that make up the Rho element containing the particle
- **rnode2** – integer – 2nd of 4 Rho nodes that make up the Rho element containing the particle
- **rnode3** – integer – 3rd of 4 Rho nodes that make up the Rho element containing the particle
- **rnode4** – integer – 4th of 4 Rho nodes that make up the Rho element containing the particle
- **rx** – dp – x- coordinate location of all the Rho nodes
- **ry** – dp – y- coordinate location of all the Rho nodes
- **SC** – real – s-level coordinates for the Rho grid
- **SCW** – real – s-level coordinates for the W grid

STARTr – integer – array specifying the index in a variable from which the first data values will be read; used when reading in one time step worth of data (excludes zeta data)
STARTz – integer – array specifying the index in the zeta variable from which the first data values will be read; used when reading in one time step worth of zeta data
stepf – integer – keeps track of the location in the current hydrodynamic file of the forward time step
t – dp – binary interpolation variable
t_b – integer – the position in the rotating indices of **t_Kh**, **t_salt**, **t_temp**, **t_Uvel**, **t_Vvel**, **t_Wvel,** and **t_zeta** in which hydrodynamic back time is stored
t_c – integer – the position in the rotating indices of **t_Kh**, **t_salt**, **t_temp**, **t_Uvel**, **t_Vvel**, **t_Wvel,** and **t_zeta** in which hydrodynamic center time is stored
t_f – integer – the position in the rotating indices of **t_Kh**, **t_salt**, **t_temp**, **t_Uvel**, **t_Vvel**, **t_Wvel,** and **t_zeta** in which hydrodynamic forward time is stored
t_ijruv – integer – the limits in terms of i,j grid space of the current particle distribution plus **ijbuff** nodes in each direction
t_Kh – real – vertical diffusivity at hydrodynamic back, center and forward time
t_salt – real – salinity at hydrodynamic back, center and forward time
t_temp – real – temperature at hydrodynamic back, center and forward time
t_Uvel – real – u- component velocity at hydrodynamic back, center and forward time
t_Vvel – real – v- component velocity at hydrodynamic back, center and forward time
t_Wvel – real – w- component velocity at hydrodynamic back, center and forward time
t_zeta – real – sea surface height at hydrodynamic back, center and forward time
tOK – integer – stores method of interpolation for current particle (1 = binary interpolation of 1st triangle, 2 = binary interpolation of 2nd triangle, 3 = inverse weighted distance)
u – dp – binary interpolation variable
u_Adjacent – integer – array containing, for each element, its own U element id followed by the element ids of all the U elements that share a node with that element
u_ele_x – dp – x-coordinate location of the four nodes in each wet U element
u_ele_y – dp – y-coordinate location of the four nodes in each wet U element
UE – integer – the four U node numbers that make up each wet U element
unode1 – integer – 1st of 4 U nodes that make up the U element containing the particle
unode2 – integer – 2nd of 4 U nodes that make up the U element containing the particle
unode3 – integer – 3rd of 4 U nodes that make up the U element containing the particle
unode4 – integer – 4th of 4 U nodes that make up the U element containing the particle
ux – dp – x- coordinate location of all the U nodes
uy – dp – y- coordinate location of all the U nodes
v_Adjacent – integer – array containing, for each element, its own V element id followed by the element ids of all the V elements that share a node with that element
v_ele_x – dp – x- coordinate location of the four nodes in each wet V element
v_ele_y – dp – y- coordinate location of the four nodes in each wet V element
VE – integer – the four V node numbers that make up each wet V element
vnode1 – integer – 1st of 4 V nodes that make up the V element containing the particle
vnode2 – integer – 2nd of 4 V nodes that make up the V element containing the particle
vnode3 – integer – 3rd of 4 V nodes that make up the V element containing the particle
vnode4 – integer – 4th of 4 V nodes that make up the V element containing the particle
vx – dp – x- coordinate location of all the V nodes

vy – dp – y- coordinate location of all the V nodes
Wgt1 – dp – weight of 1st node when interpolating via inverse weighted distance
Wgt2 – dp – weight of 2nd node when interpolating via inverse weighted distance
Wgt3 – dp – weight of 3rd node when interpolating via inverse weighted distance
Wgt4 – dp – weight of 4th node when interpolating via inverse weighted distance
x_u – real – x- coordinate location of the U grid in (i,j) location format
x_v – real – x- coordinate location of the V grid in (i,j) location format
y_u – real – y- coordinate location of the U grid in (i,j) location format
y_v – real – y- coordinate location of the V grid in (i,j) location format

Public Procedures: The following are the public subroutines and functions contained within the Settlement Module: Subroutines **createNetCDF**, **finHydro**, **getMask_Rho**, **getR_ele**, **getUVxy**, **initGrid**, **initHydro**, **initNetCDF**, **setEle**, **setEle_all**, **setInterp**, **updateHydro**, **writeNetCDF**, and Functions **getInterp**, **getP_r_element**, **getSlevel**, **getWlevel**, **interp**, and **WCTS_ITPI**.

A. Subroutine createNetCDF

Overview: This subroutine creates a NetCDF file and prepares it for model output.

Input Variables: The subroutine has one optional input variable. If the subroutine is passed the variable **dob** then the particles' date of birth, i.e. the delay in seconds before they are released, is written to the NetCDF file.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **ExeDir, Institution, NCOutFile, NCtime, numpar, OutDir, outpath, outpathGiven, RunBy, RunName, SaltTempOn, StartedOn, SVN_Version,** and **TrackCollisions** from the Parameter Module.

Module procedures used: The subroutine uses no functions or subroutines from other LTRANS modules. It does use NetCDF 90 functions.

Private Variables Used: The subroutine uses the variables **NCcount** and **prcount** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first constructs the NetCDF file name based on the values stored in **NCOutFile, NCtime, outpath,** and **outpathGiven** and creates the output file by calling NF90_CREATE. Then the dimensions are created by making calls to NF90_DEF_DIM. Next, all the variables are created using calls to NF90_DEF_VAR. Additional variables are defined depending upon if **dob** is present, or if **TrackCollisions** or **SaltTempOn** are set to .TRUE. in LTRANS.data. Then attributes are defined for each variable, as well as globally, to provide information about the variables and the model run using NF90_PUT_ATT. Once all the dimension, variable, and attribute definitions are complete, the model calls NF90_ENDDEF to exit define mode. If **dob** is present, the particle date of birth (time in seconds to delay release) is written to the file. Finally the NetCDF file is closed using NF90_CLOSE.

Variables Definitions: The following variables are used in this section:

colorID – integer – id assigned to the particle status variable
depthID – integer – id assigned to the particle depth variable
dob – dp – particle date of birth (time in seconds to delay each particle's release)
dobID – integer – id assigned to the particle date of birth variable
ExeDir – character array – path to the model run executable; for documentation purposes
hitBID – integer – id assigned to the bottom collisions variable
hitLID – integer – id assigned to the land collisions variable
Institution – character array – organization that is running the model; for documentation purposes
latID – integer – id assigned to the particle latitude variable
lonID – integer – id assigned to the particle longitude variable
modtimeID – integer – id assigned to the model time variable
ncFile – character array – concatenated hydrodynamic output file name
NCcount – integer – when writing to multiple NetCDF files, this stores the current NetCDF file number used for the filename
NCID – integer – NetCDF ID used in NetCDF functions
NCOutFile – character array – name given to the NetCDF output file(s) if outputting to NetCDF files (i.e. writeNetCDF = .true.)
NCtime – integer – time interval, in seconds, between the creation of new NetCDF output files
numpar – integer – total number of particles
numparID – integer – id assigned to the numpar dimension
OutDir – character array – location of the output files from this model run; for documentation purposes
outpath – character array – if **outpathGiven** = .true., this specifies the directory path in which to write the output files
outpathGiven – logical – if .TRUE. files are written to the path given in **outpath**
pageID – integer – id assigned to the particle age variable
prcount – integer – time step within the current NetCDF output file
RunBy – character array – name of the person who setup/run the model; for documentation purposes
RunName – character array – Name given to this specific model run to identify it; for documentation purposes
saltID – integer – id assigned to the salinity variable
SaltTempOn – logical – .TRUE. if calculate salinity and temperature at particle location, else .FALSE.
StartedOn – character array – date in which the model run was began; for documentation purposes
STATUS – integer – status ID returned from NetCDF functions
SVN_Version – character array – SVN Repository and version number; for documentation purposes
tempID – integer – id assigned to the temperature variable
timeID – integer – id assigned to the time dimension

TrackCollisions – logical – write files containing bottom and land collision information?
.TRUE. = yes, .FALSE. = no

B. Subroutine finHydro

Overview: This subroutine deallocates each of the private array variables for the Hydrodynamic Module.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has just no output variables.

Module parameters used: The subroutine uses no parameters from the Parameter Module.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **CS, CSW, depth, mask_rho, m_r, m_u, m_v, P_r_element, P_u_element, P_v_element, r_Adjacent, r_ele_x, r_ele_y, RE, rho_angle, rx, ry, SC, SCW, t_den, t_Kh, t_salt, t_temp, t_Uvel, t_Vvel, t_Wvel, t_zeta, u_Adjacent, u_ele_x, u_ele_y, UE, ux, uy, v_Adjacent, v_ele_x, v_ele_y, VE, vx, vy, x_u, x_v, y_u,** and **y_v** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine makes several calls to DEALLOCATE() in order to deallocate each of the array variables that had been allocated to contain data private to the Hydrodynamic Module.

Variables Definitions: A subset of the private variables defined on p. 99 are used in this section.

C. Function getInterp

Overview: The function getInterp returns the interpolated value at the particle's location using the interpolation variables **t**, **tOK**, **u**, **Wgt1**, **Wgt2**, **Wgt3**, and **Wgt4**, stored from a call to subroutine setInterp, and the hydrodynamic variables that have been read in by initHydro and updateHydro.

Input Variables: The function has one required input variable and one optional input variable. It must be passed a character array containing the variable name to interpolate (**var**). For variables with different s-levels, the optional variable **i** must be present to indicate which s-level to interpolate from.

Output: The function returns the interpolated value at the particle's location of the given data type.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variables **depth, rho_angle, rnode1, rnode2, rnode3, rnode4, t, t_b, t_c, t_den, t_f, t_Kh, t_salt, t_temp, t_zeta, tOK, u, Wgt1, Wgt2, Wgt3,** and **Wgt4** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: This subroutine begins by checking **var** to determine which data values to use for interpolation: **depth**, **t_KH**, **rho_angle**, **t_salt**, **t_temp**, or **t_zeta**. The appropriate data values at the four rho node locations that make up the rho element containing the particle are assigned to the variables **v1**, **v2**, **v3**, and **v4**. The method of interpolation and the values necessary to use that method of interpolation (having already been determined by setInterp) are then used with these variables to determine the interpolated value and return it.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

- **i** – integer – optional input variable; s-level to interpolate to
- **v1** – dp – value at **rnode1** to interpolate from
- **v2** – dp – value at **rnode2** to interpolate from
- **v3** – dp – value at **rnode3** to interpolate from
- **v4** – dp – value at **rnode4** to interpolate from
- **var** – character array – variable name to interpolate

D. Subroutine getMask_Rho

Overview: This subroutine returns the values in the Hydrodynamic Module private variable **mask_rho**. The subroutine allows createBounds in the Boundary Module to make the boundaries based on the rho grid land/sea masking.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has just one output variable, **mask**, which returns the rho masking.

Module parameters used: The subroutine uses no parameters from PARAM_MOD .

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **mask_rho** and **GRD_SET** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first checks if the grid data has been read in. If it has, the values in **mask_rho** are transferred to the output variable **mask** and the subroutine returns. If the grid data has not been read in, error messages are printed to the screen saying the program cannot continue. The program then stops.

Variables Definitions: The following variables are used in this section:
- **mask** – real – return variable; copy of **mask_rho**
- **mask_rho** – real – land/sea masking of the Rho grid in (i,j) location format

E. Function getP_r_element

Overview: This function returns the id of the rho element in which the particle is currently located. The subroutine allows the settlement subroutine in the Settlement Module to narrow its search to only the habitat polygons in the same element as the particle. The subroutine setEle, also located in the Hydrodynamic Module, must be called prior to calling this function at each iteration to ensure that the correct value is stored in **P_r_element**.

Input Variables: The function has just one input variable, **n**, which contains the number of the particle whose rho element is to be returned.

Output: The function outputs the id of the rho element the particle is currently in.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variable **P_r_element** which is a private variable accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: This function finds the rho element id for the given particle number stored in **P_r_element** and returns it.

Variables Definitions: The following variables are used in this section:
- **n** – integer – particle number whose rho element is to be returned
- **P_r_element** – integer – Rho element that each particle is in

F. Subroutine getR_ele

Overview: This subroutine returns the values in the variables **r_ele_x** and **r_ele_y**. The subroutine allows the subroutine createPolySpecs in the Settlement Module to determine which habitat polygons are in each element.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has two output variables, **ele_x** and **ele_y**, which return the x- and y- locations of the four nodes in each of the wet rho elements.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **GRD_SET**, **r_ele_x**, and **r_ele_y**, which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first checks if the grid data has been read in. If it has, the values in **r_ele_x** and **r_ele_y** are transferred to the output variables **ele_x** and **ele_y**, and the subroutine returns. If the grid data has not been read in, error messages are printed to the screen saying the program cannot continue. The program then stops.

Variables Definitions: The following variables are used in this section:
ele_x – dp – return variable; copy of **r_ele_x**
ele_y – dp – return variable; copy of **r_ele_y**
GRD_SET – logical – set .TRUE. when the grid has been read in, else .FALSE.
r_ele_x – dp – x- coordinate location of the four nodes in each wet Rho element
r_ele_y – dp – y- coordinate location of the four nodes in each wet Rho element

G. Function getSlevel

Overview: This function returns the depth of the given s-level.

Input Variables: The function has three input variables: the zeta and depth values at the location where the s-level depth is needed (**zeta**, **depth**) and the number of the s-level of which the depth is needed (**i**).

Output: The function returns the depth of the given s-level.

Module parameters used: The function uses the parameters **hc** and **Vtransform** from the Parameter Module, which contain the minimum hydrodynamic model depth and the transformation method used to determine depth.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variables **SC** and **CS** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The function uses one of three different equations, based on the value given to **Vtransform** in LTRANS.data. When **Vtransform** contains either 1 or 2 the subroutine uses equations found at https://www.myroms.org/wiki/index.php/Vertical_S-coordinate:

$$z(x,y,\sigma,t) = S(x,y,\sigma) + \zeta(x,y,t)\left[1 + \frac{S(x,y,\sigma)}{h(x,y)}\right], \qquad (1)$$

$$S(x,y,\sigma) - h_c\sigma + \left[h(x,y) - h_c\right] C(\sigma)$$

(2)

$$z(x,y,\sigma,t) = \zeta(x,y,t) - \left[\zeta(x,g,t) + h(x,y)\right] S(x,y,\sigma),$$

$$S(x,y,\sigma) = \frac{h_c\sigma + h(x,y)\,C(\sigma)}{h_c + h(x,y)}$$

Where z is depth, S is a nonlinear vertical transformation functional, ζ is the time-varying free-surface, h is the unperturbed water column thickness, σ is a fractional vertical stretching coordinate, C is a nondimensional, monotonic, vertical stretching function, and h_c is a positive thickness controlling the stretching.

If Vtransform contains 3, then the equation 2.16 from Song and Haidvogel (1994) is used to convert s-level coordinates to z-coordinates.

Variables Definitions: The following variables are used in this section:

CS – real – s-level stretching curves for the Rho grid
depth – dp – sea floor depth at the location where the s-level depth is to be calculated
h – dp – sea floor depth at the location where the s-level depth is to be calculated as a negative value
hc – real, parameter – minimum hydrodynamic model depth
i – integer – s-level at which to calculate depth
S – dp – non-linear vertical transformation functional
SC – real – s-level coordinates for the Rho grid
Vtransform – integer – flag to indicate which transform to use in generating depth levels: 1-WikiRoms Eq. 1, 2-WikiRoms Eq. 2, 3-Song/Haidvogel 1994 Eq.
zeta – dp – zeta value at the location where the s-level depth is to be calculated

H. Subroutine getUVxy

Overview: This subroutine returns the values in the variables **x_u**, **y_u**, **x_v**, and **y_v**. The subroutine allows createBounds in the Boundary Module to make the boundaries at the U and V grid node locations.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has four output variables: the x- and y- coordinate locations of the U and V grid nodes (**ux**, **uy**, **vx**, **vy**).

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **GRD_SET**, **x_u**, **x_v**, **y_u**, and **y_v**, which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first checks if the grid data has been read in. If it has, the values in **x_u**, **x_v**, **y_u** and **y_v** are transferred to the output variables **ux**, **vx**, **uy** and **vy**, and the subroutine returns. If the grid data has not been read in, error messages are printed to the screen saying the program cannot continue. The program then stops.

Variables Definitions: The following variables are used in this section:

- **GRD_SET** – logical – set .TRUE. when the grid has been read in, else .FALSE.
- **ux** – real – return variable; copy of **x_u**
- **uy** – real – return variable; copy of **y_u**
- **vx** – real – return variable; copy of **x_v**
- **vy** – real – return variable; copy of **y_v**
- **x_u** – real – x- coordinate location of the U grid in (i,j) location format
- **x_v** – real – x- coordinate location of the V grid in (i,j) location format
- **y_u** – real – y- coordinate location of the U grid in (i,j) location format
- **y_v** – real – y- coordinate location of the V grid in (i,j) location format

I. Function getWlevel

Overview: This function returns the depth of the given w grid s-level.

Input Variables: The function has three input variables: the zeta and depth values at the location where the w grid s-level depth is needed (**zeta**, **depth**) and the number of the w grid s-level of which the depth is needed (**i**).

Output: The function returns the depth of the given w grid s-level.

Module parameters used: The function uses the parameters **hc** and **Vtransform** from the Parameter Module, which contain the minimum hydrodynamic model depth and the transformation method used to determine depth.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variables **SCW** and **CSW**, which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The function uses one of three different equations, based on the value given to Vtransform in LTRANS.data. When Vtransform contains either 1 or 2 the subroutine uses equations found at https://www.myroms.org/wiki/index.php/Vertical_S-coordinate. If Vtransform contains 3, then the it uses equation 2.16 from Song and Haidvogel (1994) to convert s-level coordinates to z-coordinates.

Variables Definitions: The following variables are used in this section:

- **CSW** – real – s-level stretching curves for the W grid
- **depth –** dp – sea floor depth at the location where the w grid s-level depth is to be calculated
- **h –** dp – sea floor depth at the location where the s-level depth is to be calculated as a negative value
- **hc** – real, parameter – minimum hydrodynamic model depth
- **i** – integer – w grid s-level at which to calculate depth
- **S –** dp – non-linear vertical transformation functional
- **SCW** – real – s-level coordinates for the W grid
- **Vtransform** – integer – flag to indicate which transform to use in generating depth levels: 1-WikiRoms Eq. 1, 2-WikiRoms Eq. 2, 3-Song/Haidvogel 1994 Eq.
- **zeta** – dp – zeta value at the location where the w grid s-level depth is to be calculated

J. Subroutine initGrid

Overview: This subroutine reads in the grid information to create all the element variables.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **filenum**, **max_rho_elements**, **max_u_elements**, **max_v_elements**, **NCgridfile**, **numdigits**, **numpar**, **prefix**, **rho_elements**, **rho_nodes**, **suffix**, **u_elements**, **u_nodes**, **ui**, **uj**, **us**, **v_elements**, **v_nodes**, **vi**, **vj**, **ws** from the Parameter Module. **NCgridfile**, **prefix**, **suffix**, and **filenum** contain the path (if needed) and file name of the NetCDF model grid input file, the path (if needed) and first part of the file name of the hydrodynamic input files, the number in the first hydrodynamic input file, and the final part of the file name of the hydrodynamic input files.

Module procedures used: The subroutine uses the interfaces lat2y and lon2x from CONVERT_MOD (the Conversion Module). It also uses NetCDF 90 functions.

Private Variables Used: The function uses the variables **CS, CSW, depth, filenm, GRD_SET, m_r, m_u, m_v, mask_rho, P_r_element, P_u_element, P_v_element, r_Adjacent, r_ele_x, r_ele_y, RE, rho_angle, rx, ry, SC, SCW, u_Adjacent, u_ele_x, u_ele_y, UE, ux, uy, v_Adjacent, v_ele_x, v_ele_y, VE, vx, vy, x_u, x_v, y_u,** and **y_v** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine begins by allocating most of the variables that are private to the hydrodynamic module, as well as all the variables being used just by this subroutine. Next, it opens the NetCDF model grid input file (**NCgridfile**) and reads in the grid information that does not change throughout the run of the model: depth, latitude and longitude coordinates of the Rho, U, and V grids, land/sea masking of the Rho, U, and V grids, and the angle between the x- coordinate and true east direction in radians at the rho nodes. Next, the subroutine opens the first hydrodynamic input file and reads in the s-level variables: **CS**, **CSW**, **SC**, and **SCW**.

The remainder of the subroutine creates the grid nodes and elements used throughout LTRANS. First, the node locations are converted from latitude and longitude (**lon_rho**, **lat_rho**, **lon_u**, **lat_u**, **lon_v**, **lat_v**) to x- and y- coordinates(**x_rho**, **y_rho**, **x_u**, **y_u**, **x_v**, **y_v**). Next, the angle and mask variables (**angle**, **mask_rho**, **mask_u**, and **mask_v**) are converted from the two-dimensional format with which they were read in to a one-dimensional format (**rho_angle**, **rho_mask**, **u_mask**, and **v_mask**), giving each grid node a single node number rather than its previous (i,j) location. Next, the variables **r_ele**, **u_ele**, and **v_ele** are created with the node numbers that make up the four corners of each element. Then the elements with no nodes masked as water are removed and the remaining 'wet' elements are stored in the variables **RE**, **UE**, and **VE**. The x- and y- coordinate variables for the Rho, U and V grids are then converted to the new node number format and stored in the variables **rx**, **ry**, **ux**, **uy**, **vx**, and **vy**. Finally, variables that hold the x- and y- coordinates of all four nodes in each wet element are created: **r_ele_x**, **r_ele_y**, **u_ele_x**, **u_ele_y**, **v_ele_x**, **v_ele_y**. The last section of the subroutine creates variables that hold, for each element, the element's own element number followed by the element numbers of all the elements that share a node with that element (**r_Adjacent**, **u_Adjacent**, and **v_Adjacent**). These variables are used to restrict search algorithms in subroutine setEle when finding where a particle may be located after one time step. The subroutine finishes by setting the value of GRD_SET to .TRUE. indicating that the grid has been set, and then deallocating all the variables local to this subroutine.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

- **angle** – real – angle between Rho node's x-coordinate and true east direction (radian) in (i,j) location format
- **count** – integer – used in conversions from (i,j) location formats to node number formats
- **filenum** – integer, parameter – number in the first hydrodynamic input file name
- **i** – integer – iteration variable
- **inele** – integer – when finding 'wet' elements, initialized to zero, but switched to one if any of an element's four nodes are masked as water
- **j** – integer – iteration variable
- **lat_rho** – dp – latitude of the Rho grid in (i,j) location format
- **lat_u** – dp – latitude of the U grid in (i,j) location format
- **lat_v** – dp – latitude of the V grid in (i,j) location format
- **lon_rho** – dp – longitude of the Rho grid in (i,j) location format
- **lon_u** – dp – longitude of the U grid in (i,j) location format
- **lon_v** – dp – longitude of the V grid in (i,j) location format
- **m** – integer – used to count adjacent elements when finding adjacent elements
- **mask_u** – real – land/sea masking of the U grid in (i,j) location format

mask_v – real – land/sea masking of the V grid in (i,j) location format
max_rho_elements – integer, parameter – maximum number of rho grid elements
max_u_elements – integer, parameter – maximum number of u grid elements
max_v_elements – integer, parameter – maximum number of v grid elements
NCgridfile – character array, parameter – name and path (if needed) of the NetCDF grid file
NCID – integer – NetCDF ID used in NetCDF functions
numdigits – integer – the number of digits in the Hydrodynamic Input Files' names, including leading zeros
numpar – integer – total number of particles
prefix – character array, parameter – first part of hydrodynamic input file name (and path if needed)
r_ele – integer – node numbers for each rho element
rho_elements – integer, parameter – total number of wet rho elements
rho_mask – integer – land/sea masking of the Rho grid in node number format
rho_nodes – integer – total number of rho nodes
romdepth – real – sea floor depth of the rho grid in (i,j) location format
STATUS – integer – status ID returned from NetCDF functions
suffix – character array, parameter – final part of the hydrodynamic input file name
u_ele – integer – node numbers for each u element
u_elements – integer, parameter – total number of wet u elements
u_mask – integer – land/sea masking of the u grid in node number format
u_nodes – integer - total number of u nodes
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the rho and u grids
us – integer – number of depth levels in the rho, u, and v grids
v_ele – integer – node numbers for each v element
v_elements – integer, parameter – total number of wet rho elements
v_mask – integer – land/sea masking of the v grid in node number format
v_nodes – integer - total number of v nodes
vi – integer, parameter – number of nodes across the rho and v grids
VID – integer – variable ID used in NetCDF functions
vj – integer, parameter – number of nodes down the v grid
ws – integer – number of depth levels in the w grid
x_rho – real – x- coordinate location of the Rho grid in (i,j) location format
y_rho – real – y- coordinate location of the Rho grid in (i,j) location format

K. Subroutine initHydro

Overview: This subroutine is called prior to the first iteration through the external time step loop of LTRANS.f90. It reads in the initial hydrodynamic data for the back, center, and forward time steps from the first three time steps of the first ROMS sequential output file. This data includes u, v, and w velocities, salinity, temperature, zeta, and vertical diffusivity.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **constAks**, **constDens**, **constSalt**, **constTemp**, **constU**, **constV**, **constW**, **constZeta**, **filenum**, **numdigits**, **numpar**, **prefix**, **readAks**, **readDens**, **readSalt**, **readTemp**, **readU**, **readV**, **readW**, **readZeta**, **rho_nodes**, **suffix**, **u_nodes**, **ui**, **uj**, **us**, **v_nodes**, **vi**, **vj**, and **ws** from the Parameter Module.

Module procedures used: The subroutine uses no functions or subroutines from other modules. It does use NetCDF 90 functions.

Private Variables Used: The subroutine uses the variables **COUNTr, COUNTz, filenm, iint, m_r, m_u, m_v, STARTr, STARTz, stepf, t_b, t_c, t_den, t_f, t_ijruv, t_Kh, t_salt, t_temp, t_Uvel, t_Vvel, t_Wvel,** and **t_zeta** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine begins by allocating the variables that are private to the hydrodynamic module that contain the changing hydrodynamic values, as well as all the variables being used just by this subroutine. Next, it initializes the hydrodynamic input file counting variable (**iint**) to zero, indicating that the model is using input from the first ROMS sequential output file. It then stores the file name of the first ROMS sequential output file in the variable **filenm** by combining **prefix**, the current file number (**iint** + **filenum**), and **suffix**. Next, the variables **t_b**, **t_c**, and **t_f** are initialized to 1, 2, and 3 respectively to indicate which positions in the variable arrays contain the back, center, and forward time steps. Also, **stepf** is initialized to 3 to indicate that the forward time step is the third time step of the file. Note that **stepf** is a private variable of the Hydrodynamic Module to be shared with updateHydro.

Next, the subroutine **setijruv** is called to find the range of the rho, u, and v grids that contain particles, so that only a limited portion of each grid needs to be read in. Using the NF90_OPEN command the first ROMS sequential output file is opened and a subset of the first three time steps of data are read in. Once the zeta, salinity, temperature, vertical diffusivity, and v-, v- and w- component velocities have been read in, they must be converted from (i,j) location format to node numbers and stored in the private variable equivalents of the local variables into which they were first read.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

- **constAks** – dp – if **readAks** = .false, the constant salinity vertical diffusion coefficient (Aks)
- **constSalt** – dp – if **readSalt** = .false, the constant value of salinity
- **constTemp** – dp – if **readTemp** = .false, the constant value of temperature
- **constU** – dp – if **readU** = .false, the constant value of u- component velocity
- **constV** – dp – if **readV** = .false, the constant value of v- component velocity
- **constW** – dp – if **readW** = .false, the constant value of w- component velocity
- **constZeta** – dp – if **readZeta** = .false, the constant value of surface height
- **count** – integer – used in conversions from (i,j) location formats to node number formats

counter – integer – number in the concatenated ROMS sequential output file name
filenum – integer, parameter – number in the first hydrodynamic input file name
i – integer – iteration variable
j – integer – iteration variable
k – integer – iteration variable
NCID – integer – NetCDF ID used in NetCDF functions
numdigits – integer – the number of digits in the Hydrodynamic Input Files' names, including leading zeros
numpar – integer – total number of particles
prefix – character array, parameter – first part of hydrodynamic input file name (and path if needed)
readAks – logical – If .TRUE. read in salinity vertical diffusion coefficient (Aks) from NetCDF file, else use **constAks**
readSalt – logical – If .TRUE. read in salinity from NetCDF file, else use **constSalt**
readTemp – logical – If .TRUE. read in temp from NetCDF file, else use **constTemp**
readU – logical – If .TRUE. read in u- component velocity from NetCDF file, else use **constU**
readV – logical – If .TRUE. read in v- component velocity from NetCDF file, else use **constV**
readW – logical – If .TRUE. read in w- component velocity from NetCDF file, else use **constW**
readZeta – logical – If .TRUE. read in sea surface height from NetCDF file, else use **constZeta**
rho_nodes – integer – total number of rho nodes
romKH – real – vertical diffusivity at the first three hydrodynamic time steps in (i,j) location format
romS – real – salinity at the first three hydrodynamic time steps in (i,j) location format
romT – real – temperature at the first three hydrodynamic time steps in (i,j) location format
romU – real – u- component velocity at the first three hydrodynamic time steps in (i,j) location format
romV – real – v- component velocity at the first three hydrodynamic time steps in (i,j) location format
romW – real – w- component velocity at the first three hydrodynamic time steps in (i,j) location format
romZ – real – zeta at the first three hydrodynamic time steps in (i,j) location format
STATUS – integer – status ID returned from NetCDF functions
suffix – character array, parameter – final part of the hydrodynamic input file name
u_nodes – integer - total number of u nodes
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the u grid
us – integer – number of depth levels in the rho, u, and v grids
v_nodes – integer - total number of v nodes
vi – integer, parameter – number of nodes across the rho and v grids
VID – integer – variable ID used in NetCDF functions
vj – integer, parameter – number of nodes down the v grid
ws – integer, parameter – number of depth levels in the w grid

L. Subroutine initNetCDF

Overview: This subroutine initializes variables necessary if output is being written to NetCDF.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variables **NCcount** and **NCstart** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine initializes the values of NCcount and NCstart to 0.

Variables Definitions: A subset of the private variables defined on p. 99 are used in this section.

M. Function interp

Overview: This function determines the method of interpolation and returns the interpolated value at the particle's location using the hydrodynamic variables read in by initHydro and updateHydro. The function uses bilinear interpolation to interpolate values at the nodes of a quadrilateral to a point located within the quadrilateral. In the rare case that the t/u values of bilinear interpolation are undefined, the inverse weighted difference is employed. This function is completely independent of setInterp and getInterp, and in no way affects either procedure.

Input Variables: The function has three required input variables and one optional input variable. It must be passed the x- and y- coordinates of the particle location that is being interpolated to and a character array containing the name of the variable to interpolate (**var**). For variables with different s-levels, the optional variable **i** must be present to indicate which s-level to interpolate from.

Output: The function returns the interpolated value at the particle's location of the given data type.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses the variables **depth, rho_angle, rnode1, rnode2, rnode3, rnode4, rx, ry, t_b, t_c, t_den, t_f, t_Kh, t_salt, t_temp, t_Uvel, t_Vvel, t_Wvel, t_zeta, unode1, unode2, unode3, unode4, ux, uy, vnode1, vnode2, vnode3, vnode4, vx,** and **vy** , which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: This subroutine begins by checking **var** to determine which data values to use for interpolation: **depth**, **t_KH**, **rho_angle**, **t_salt**, **t_temp**, **t_Uvel**, **t_Vvel**, **t_Wvel**, or **t_zeta**. The appropriate data values at the four nodes that make up the element containing the particle are assigned to the variables **v1**, **v2**, **v3**, and **v4**. Next, the x- and y- locations of the nodes must be set in **x1**, **x2**, **x3**, **x4**, **y1**, **y2**, **y3**, and **y4**. Depending on which value is being interpolated, these are the locations of the nodes of either the Rho, U or V grid element that the particle is in, previously determined by a call to setEle. The subroutine then determines the interpolation method, interpolates, and returns.

Bilinear interpolation is the best way to interpolate a value, but it is used for triangles, and the subroutine starts with a quadrilateral. The quadrilateral is therefore divided into two triangles, where nodes 1, 2, and 3 compose the first triangle and nodes 1, 3, and 4 compose the second triangle. Bilinear interpolation values for the first triangle are calculated and stored in the variables **t** and **u**, and the result of interpolating to the particle is stored in **vp**. If the values in **t** and **u** are both above zero and their sum is below 1, the subroutine is complete and it returns.

However, if either **t** or **u** is below zero, or their sum is above one, bilinear interpolation of the first triangle failed (i.e., the particle is not in that triangle or the result is undefined). If this is the case, the values in **t** and **u** are replaced by bilinear interpolation values for the second triangle, and the result in **vp** is recalculated. Once again, the values in **t** and **u** are tested to ensure that they are both above zero and their sum is below 1. If this is true the subroutine is complete and it returns.

However, if the bilinear interpolation fails for the second triangle, the subroutine resorts to using inverse weighted distance. The subroutine finds the distance from the particle to each of the four nodes of the element. Each of these distances is divided by the sum of all four distances to determine the weight of each node. These weights are then used to calculate the interpolated value at the particle, **vp**, and the subroutine returns.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

- **Dis1** – dp – distance from the particle to the element's 1st node
- **Dis2** – dp – distance from the particle to the element's 2nd node
- **Dis3** – dp – distance from the particle to the element's 3rd node
- **Dis4** – dp – distance from the particle to the element's 4th node
- **i** – integer – optional input variable; s-level to interpolate to
- **RUV** – integer – variable to indicate which grid to use (1 = Rho, 2 = U, 3 = V)
- **TDis** – dp – sum of the distances from the particle to each of the four nodes

tt – dp – binary interpolation variable
uu – dp – binary interpolation variable
v1 – dp – value at 1st element node to interpolate from
v2 – dp – value at 2nd element node to interpolate from
v3 – dp – value at 3rd element node to interpolate from
v4 – dp – value at 4th element node to interpolate from
var – character array – data type to interpolate from
vp – dp – interpolated value at the particle's location
Wt1 – dp – weight of 1st node when interpolating via inverse weighted distance
Wt2 – dp – weight of 2nd node when interpolating via inverse weighted distance
Wt3 – dp – weight of 3rd node when interpolating via inverse weighted distance
Wt4 – dp – weight of 4th node when interpolating via inverse weighted distance
x1 – dp – x- coordinate of 1st element node to interpolate from
x2 – dp – x- coordinate of 2nd element node to interpolate from
x3 – dp – x- coordinate of 3rd element node to interpolate from
x4 – dp – x- coordinate of 4th element node to interpolate from
xp – dp – x- coordinate of the particle being interpolated to
y1 – dp – y- coordinate of 1st element node to interpolate from
y2 – dp – y- coordinate of 2nd element node to interpolate from
y3 – dp – y- coordinate of 3rd element node to interpolate from
y4 – dp – y- coordinate of 4th element node to interpolate from
yp – dp – y- coordinate of the particle being interpolated to

N. Subroutine setEle

Overview: This subroutine determines the Rho, U and V grid elements in which a particle is located. When passed the optional argument **first** with the value .TRUE., the subroutine iterates through all the wet elements of each grid and finds the elements containing the particle. If **first** is not present or has the value .FALSE. then the subroutine only checks the elements adjacent to the element the particle was in during the last time step.

Input Variables: The subroutine has three required input variables and one optional input variable. It requires the x- and y- coordinates of the particle (**Xpar**, **Ypar**) and the particle number (**n**). The optional argument (**first**) is a logical variable that when .TRUE. indicates that it is the first iteration and all the elements must be searched and when .FALSE. indicates that it is a subsequent iteration and the search can be restricted to elements adjacent to the particle's last known element locations. If not present, the subroutine defaults to the latter process.

Output Variables: The subroutine has one optional output variable. The integer variable **err**, when present, returns the error status of the subroutine. The error status value of zero indicates no error, while the values one, two and three indicate an error occurred finding the Rho, U, or V grid element, respectively, when searching all the elements, and values of four, five and six indicate an error occurred finding the Rho, U, or V grid element, respectively, when just searching adjacent elements.

Module parameters used: The subroutine uses the parameters **rho_elements**, **u_elements,** and **v_elements** from the Parameter Module, which contain the total numbers of wet rho, u and v grid elements.

Module procedures used: The subroutine uses the subroutine gridcell from the Gridcell Module.

Private Variables Used: The subroutine uses the variables **P_r_element**, **P_u_element**, **P_v_element**, **r_Adjacent**, **r_ele_x**, **r_ele_y**, **RE**, **rnode1**, **rnode2**, **rnode3**, **rnode4**, **u_Adjacent**, **u_ele_x**, **u_ele_y**, **UE**, **unode1**, **unode2**, **unode3**, **unode4**, **v_Adjacent**, **v_ele_x**, **v_ele_y**, **VE**, **vnode1**, **vnode2**, **vnode3**, and **vnode4**, which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first initializes **error** to zero, indicating that no errors have occurred. Then, if the variable **first** was present in the function call and contained the value .TRUE., the subroutine calls gridcell for the rho, u and v grids to search all the elements and determine which elements the particle is in. If gridcell cannot find an element the particle is in then the value in **error** is changed to reflect which grid that gridcell had problems with (1 = Rho, 2 = U, 3 = V). If there were errors on multiple grids the highest error number will be saved. If **first** was not present in the function call or it contained the value .FALSE., the subroutine checks if the particle is still in the element it was last in. If not, it checks the elements adjacent to the one it was last in. These checks are made with calls to gridcell for the rho, u and v grids with the additional optional argument **checkele** present to indicate the one element to search. If the particle was not found in the same element as before or in any of its adjacent elements, then the value in **error** is changed to reflect with which grid the subroutine had problems (4 = Rho, 5 = U, 6 = V). If there were errors on multiple grids the highest numbered error number will be saved. Once the Rho, U and V grid elements have been found, the values in **rnode1**, **rnode2**, **rnode3**, **rnode4**, **unode1**, **unode2**, **unode3**, **unode4**, **vnode1**, **vnode2**, **vnode3**, and **vnode4** are updated to reflect the node numbers of the four nodes that make up the Rho, U and V grid elements.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

checkele – integer – used when checking adjacent elements; prompts gridcell to check only the element number it contains when included in the call to the subroutine
err – integer – optional output variable; returns the error status id of the subroutine
error – integer – keeps track of the error status id within the subroutine
first – logical – optional input variable; when present passed to **fst**
fst – logical – when .TRUE. indicates all the elements must be search, and when .FALSE. indicates the search can be restricted to adjacent elements. Set equal to **first** when present and when not defaults to .FALSE.
i – integer – iteration variable
n – integer – particle number whose elements are being found

oP_ele – integer – holds the current element number (rho, u, or v) when cycling through adjacent elements in r_Adjacent, u_Adjacent, or v_Adjacent
P_ele – integer – holds the element number returned from gridcell when cycling through adjacent elements in r_Adjacent, u_Adjacent, or v_Adjacent
P_r_ele – integer – holds the rho element number returned from gridcell when cycling through all the rho elements
P_u_ele – integer – holds the u element number returned from gridcell when cycling through all the u elements
P_v_ele – integer – holds the v element number returned from gridcell when cycling through all the v elements
rho_elements – integer, parameter – total number of wet rho elements
triangle – return variable from gridcell; 0 = not in an element, 1 = in an element
u_elements – integer, parameter – total number of wet u elements
v_elements – integer, parameter – total number of wet v elements
Xpar – dp – x- coordinate of the particle
Ypar – dp – y- coordinate of the particle

O. setEle_all

Overview: This subroutine determines the Rho, U and V grid elements in which each of the particles is located.

Input Variables: The subroutine has three required input variables: the x- and y- coordinates of the particle (**Xpar**, **Ypar**) and the particle number (**n**).

Output Variables: The subroutine has one optional output variable. The integer variable **err**, when present, returns the error status of the subroutine. The error status value of zero indicates no error, while the values one, two and three indicate an error occurred finding the Rho, U, or V grid element, respectively, when searching all the elements, and values of four, five and six indicate an error occurred finding the Rho, U, or V grid element, respectively, when just searching adjacent elements.

Module parameters used: The subroutine uses the parameters **rho_elements**, **u_elements,** and **v_elements** from the Parameter Module.

Module procedures used: The subroutine uses the subroutine gridcell from the Gridcell Module.

Private Variables Used: The subroutine uses the variables **P_r_element**, **P_u_element**, **P_v_element**, **r_Adjacent**, **r_ele_x**, **r_ele_y**, **u_Adjacent**, **u_ele_x**, **u_ele_y**, **v_Adjacent**, **v_ele_x**, and **v_ele_y** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first initializes **error** to zero, indicating that no errors have occurred. Then, the subroutine calls gridcell for the rho, u and v grids to search all the elements and determine which elements the first particle is in. For each of the remaining particles, the model checks the element and adjacent elements of the previous particle first, and if is not found, checks all the elements. This is done to speed up initialization, as particles are usually released near each other. If gridcell cannot find an element the particle is in then the value in **error** is changed to reflect which grid that gridcell had problems with (1 = Rho, 2 = U, 3 = V) and **p** is changed to reflect the id of the particle with an error. If there were errors on multiple grids or with multiple particles the highest error number for the last particle with an error will be saved.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

checkele – integer – used when checking adjacent elements; prompts gridcell to check only the element number it contains when included in the call to the subroutine
err – integer – optional output variable; returns the error status id of the subroutine
error – integer – keeps track of the error status id within the subroutine
fst – logical – when .TRUE. indicates all the elements must be search, and when .FALSE. indicates the search can be restricted to adjacent elements. Set equal to **first** when present and when not defaults to .FALSE.
i – integer – iteration variable
n – integer – particle number whose elements are being found
oP_ele – integer – holds the current element number (rho, u, or v) when cycling through adjacent elements in r_Adjacent, u_Adjacent, or v_Adjacent
p – integer – last particle number which could not be found in an element
P_ele – integer – holds the element number returned from gridcell when cycling through adjacent elements in r_Adjacent, u_Adjacent, or v_Adjacent
P_r_ele – integer – holds the rho element number returned from gridcell when cycling through all the rho elements
P_u_ele – integer – holds the u element number returned from gridcell when cycling through all the u elements
P_v_ele – integer – holds the v element number returned from gridcell when cycling through all the v elements
par – integer – optional output variable; returns the id of last particle that wasn't found in an element, or 0 if this doesn't occur
rho_elements – integer, parameter – total number of wet rho elements
triangle – return variable from gridcell; 0 = not in an element, 1 = in an element
u_elements – integer, parameter – total number of wet u elements
v_elements – integer, parameter – total number of wet v elements
Xpar – dp – x- coordinate of the particle
Ypar – dp – y- coordinate of the particle

P. Subroutine setijruv

Overview: This subroutine determines the ranges, in terms of i and j, that particles can be found on the rho, u, and v grids, expands the ranges by the value in **ijbuff**, and stores them in **t_ijruv**.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has just no output variables.

Module parameters used: The subroutine uses **ijbuff**, **numpar**, **ui**, **uj**, **vi**, and **vj** from the Parameter Module.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **P_r_element, P_u_element, P_v_element, RE, t_ijruv, UE,** and **VE** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine iterates through each particle checking which element that particle is in. It then takes the i and j coordinates of the four nodes comprising that element, and determines if the range of particles, in term of i and j, needs to be increased. It does this for the rho, u, and v grids, storing the ranges in **t_ijruv**. Because particles may move out of this range before the next time step, a buffer is added to the range to make sure data is available if this happens. The buffer, whose size is specified in **ijbuff** in LTRANS.data, is added to each side of the ranges. If this buffer increases the range beyond the grid size, the edge of the grid is used.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

i – integer – i- coordinate of the bottom left corner of the element containing the current particle
ijbuff – integer – the model only reads in hydrodynamic data for nodes in the viscinity of the particles' locations; this contains the size of the buffer around the particles' locations of which to read. For instance if ijbuff = 4, then the model reads in an additional four nodes in each direction around the area that contains the particles.
j – integer – j- coordinate of the bottom left corner of the element containing the current particle
n – integer – used to iterate through the particles
numpar – integer – total number of particles
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the u grid
vi – integer – number of nodes across the v grid
vj – integer – number of nodes down the v grid

Q. Subroutine setInterp

Overview: This subroutine determines the best method of interpolation at the particle's location on the rho grid and stores that method, along with the values necessary to use that method, for later interpolation by the function getInterp. Since the same particle location and rho node locations are used to interpolate several variables, the interpolation values can be saved by this subroutine and used repeatedly by getInterp. The subroutine setEle must be called prior to calling setInterp so that the correct element will be used for interpolation.

Input Variables: The subroutine has three input variables: the x- and y- coordinates of the particle (**xp**, **yp**) and the particle number (**n**).

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses the variables **rnode1, rnode2, rnode3, rnode4,** **rx**, **ry**, **t**, **tOK**, **u**, **Wgt1**, **Wgt2**, **Wgt3**, and **Wgt4**, which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine begins by setting the x- and y- locations of the rho nodes in **x1**, **x2**, **x3**, **x4**, **y1**, **y2**, **y3**, and **y4**. The subroutine then determines the interpolation method, stores the interpolation values, and returns.

Bilinear interpolation is the best way to interpolate a value, but it is used for triangles, and the subroutine starts with a quadrilateral. The quadrilateral is therefore divided into two triangles, where nodes 1, 2, and 3 compose the first triangle and nodes 1, 3, and 4 compose the second triangle. Bilinear interpolation values for the first triangle are calculated and stored in the variables **t** and **u**, and **tOK** is set to 1 to indicate that the current method is bilinear interpolation of the first triangle. If the values in **t** and **u** are both above zero and their sum is below 1, the subroutine is complete and it returns.

However, if either **t** or **u** is below zero, or their sum is above one, bilinear interpolation of the first triangle failed (i.e., the particle was not in that triangle or the result is undefined). If this is the case, the values in **t** and **u** are replaced by bilinear interpolation values for the second triangle and **tOK** is updated to 2, indicating that the current method is bilinear interpolation of the second triangle. Once again, the values in **t** and **u** are tested to ensure that they are both above zero and their sum is below 1. If this is true the subroutine is complete and it returns.

However, if the bilinear interpolation fails for the second triangle, the subroutine resorts to using inverse weighted distance. The subroutine finds the distance from the particle to each of the four nodes of the element. Each of these distances is divided by the sum of all four distances to determine the weight of each node. These weights are then stored in the variables **Wgt1**, **Wgt2**, **Wgt3**, and **Wgt4**, **tOK** is set equal to 3 to indicate that inverse weighted distance was used, and the subroutine returns.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

Dis1 – dp – distance from the particle to the element's 1st node
Dis2 – dp – distance from the particle to the element's 2nd node
Dis3 – dp – distance from the particle to the element's 3rd node
Dis4 – dp – distance from the particle to the element's 4th node
n – integer – particle number being interpolated to
TDis – dp – sum of the distances from the particle to each of the four nodes
x1 – dp – x- coordinate of 1st element node to interpolate from
x2 – dp – x- coordinate of 2nd element node to interpolate from
x3 – dp – x- coordinate of 3rd element node to interpolate from
x4 – dp – x- coordinate of 4th element node to interpolate from
xp – dp – x- coordinate of the particle being interpolated to
y1 – dp – y- coordinate of 1st element node to interpolate from
y2 – dp – y- coordinate of 2nd element node to interpolate from
y3 – dp – y- coordinate of 3rd element node to interpolate from
y4 – dp – y- coordinate of 4th element node to interpolate from
yp – dp – y- coordinate of the particle being interpolated to

R. Subroutine updateHydro

Overview: This subroutine is called at the beginning of all but the first two iterations through the external time step loop of LTRANS.f90. It updates the hydrodynamic data for the back, center, and forward time steps by rotating indices such that the previous center values are now referred to by the back variables, the previous forward values are now referred to by the center variables, and the previous back values are now referred to by the forward variables. Lastly the new forward values are read in from a ROMS sequential output file. If the end of a ROMS sequential output file is reached, the subroutine will open the next file and begin reading from it. The data read in includes U, V, and W velocities, salinity, temperature, zeta, and vertical diffusivity.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables.

Module parameters used: The subroutine uses the parameters **constAks**, **constDens**, **constSalt**, **constTemp**, **constU**, **constV**, **constW**, **constZeta**, **filenum**, **numdigits**, **prefix**, **readAks**, **readDens**, **readSalt**, **readTemp**, **readU**, **readV**, **readW**, **readZeta**, **rho_nodes**, **startfile**, **suffix**, **tdim**, **u_nodes**, **ui**, **uj**, **us**, **v_nodes**, **vi**, **vj**, and **ws** from the Parameter Module.

Private Variables Used: The subroutine uses the variables **COUNTr, COUNTz, filenm, iint, m_r, m_u, m_v, STARTr, STARTz, stepf, t_b, t_c, t_den, t_f, t_ijruv, t_Kh, t_salt, t_temp, t_Uvel, t_Vvel, t_Wvel,** and **t_zeta** , which are private variables accessible only to the procedures in the Hydrodynamic Module.

Module procedures used: The subroutine uses no functions or subroutines from other modules. It does use NetCDF 90 functions.

Numerical Method: The subroutine first rotates the indices such that back time now refers to the previous center time, center time now refers to the previous forward time, and forward time can be read in to the previous back time which is no longer needed. Next, the subroutine compares **stepf**, which keeps track of the 'forward' time step, to **tdim**, the total number of time steps in each hydrodynamic model output file. If **stepf** is less than **tdim**, the forward time step has not yet passed the final time step of the output file, so **stepf** is merely incremented. An exception to this is if the first file is being read and it happens to be a "startfile" which has an additional time step. If this is the case and **stepf** equals **tdim**, **stepf** is incremented one more time. However, if **stepf** is not less than **tdim** (or equal as in the exception above), the netcdf file for the next time period must be opened. The filename for the next netcdf file is found and written to the variable **filenm**. Once the correct filename is stored in **filenm**, it can be used to open the NetCDF file and read in data from the next hydrodynamic model output file.

Next, the subroutine **setijruv** is called to find the range of the rho, u, and v grids that contain particles, so that only a limited portion of each grid needs to be read in. The program then reads in a subset, specified in **t_ijruv**, of the new forward variables from the netcdf file using **stepf** to extract data from the correct time step. Since the subroutine is only read in one time step at a time, the START and COUNT variables are prepared for each read in. Once the zeta, salinity, temperature, vertical diffusivity, and U- V- and W- component velocities have been read in, they must be converted from (i,j) location format to node numbers and stored in the private variable equivalents of the local variables into which they were first read.

Variables Definitions: In addition to a subset of the private variables defined on p. 99, the following variables are used in this section:

- **constAks** – dp – if **readAks** = .false, the constant salinity vertical diffusion coefficient (Aks)
- **constSalt** – dp – if **readSalt** = .false, the constant value of salinity
- **constTemp** – dp – if **readTemp** = .false, the constant value of temperature
- **constU** – dp – if **readU** = .false, the constant value of u- component velocity
- **constV** – dp – if **readV** = .false, the constant value of v- component velocity
- **constW** – dp – if **readW** = .false, the constant value of w- component velocity
- **constZeta** – dp – if **readZeta** = .false, the constant value of surface height
- **count** – integer – used in conversions from (i,j) location formats to node number formats
- **counter** – integer – number in the concatenated ROMS sequential output file name
- **filenum** – integer, parameter – number in the first hydrodynamic input file name
- **i** – integer – iteration variable
- **j** – integer – iteration variable
- **k** – integer – iteration variable
- **NCID** – integer – NetCDF ID used in NetCDF functions
- **numdigits** – integer – the number of digits in the Hydrodynamic Input Files' names, including leading zeros
- **prefix** – character array, parameter – first part of hydrodynamic input file name (and path if needed)

readAks – logical – If .TRUE. read in salinity vertical diffusion coefficient (Aks) from NetCDF file, else use **constAks**
readSalt – logical – If .TRUE. read in salinity from NetCDF file, else use **constSalt**
readTemp – logical – If .TRUE. read in temp from NetCDF file, else use **constTemp**
readU – logical – If .TRUE. read in u- component velocity from NetCDF file, else use **constU**
readV – logical – If .TRUE. read in v- component velocity from NetCDF file, else use **constV**
readW – logical – If .TRUE. read in w- component velocity from NetCDF file, else use **constW**
readZeta – logical – If .TRUE. read in sea surface height from NetCDF file, else use **constZeta**
rho_nodes – integer – total number of rho nodes
romKHf – real – vertical diffusivity at the hydrodynamic forward time step in (i,j) location format
romSf – real – salinity at the hydrodynamic forward time step in (i,j) location format
romTf – real – temperature at the hydrodynamic forward time step in (i,j) location format
romUf – real – u- component velocity at the hydrodynamic forward time step in (i,j) location format
romVf – real – v- component velocity at the hydrodynamic forward time step in (i,j) location format
romWf – real – w- component velocity at the hydrodynamic forward time step in (i,j) location format
romZf – real – zeta at the hydrodynamic forward time step in (i,j) location format
startfile – logical – does the first NetCDF hydrodynamic input file begin at 12:00am and contain an additional time step? .TRUE. = yes, .FALSE. = no
STATUS – integer – status ID returned from NetCDF functions
suffix – character array, parameter – final part of the hydrodynamic input file name
tdim – integer, parameter – size of the time dimension used in the ROMS sequential hydrodynamic input files
u_nodes – integer - total number of u nodes
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the u grid
us – integer – number of depth levels in the rho, u, and v grids
v_nodes – integer - total number of v nodes
vi – integer, parameter – number of nodes across the rho and v grids
VID – integer – variable ID used in NetCDF functions
vj – integer, parameter – number of nodes down the v grid
ws – integer, parameter – number of depth levels in the w grid

S. Function WCTS_ITPI

Overview: This function creates a water column tension spline at back, center, and forward hydrodynamic time, then uses polynomial interpolation to determine internal time values to get the final value of the particle in space and time. The name of this function is derived from the initials of Water Column Tension Spline, Internal Time Polynomial Interpolation. The return value can be the value at back time, the value at center time, the value at forward time, or a weighted average of the three with center weighing four times as much as back and forward. This is dependent on the value passed into the function through the variable **v** which contains an integer from one to four indicating the version of output to return (1 = back, 2 = center, 3 = forward, 4 = weighted average).

Input Variables: The function has fifteen input variables. It is passed a character array containing the variable name (without b, c, or f) to interpolate (**var**), the x- and y- coordinates of the particle (**Xpos**, **Ypos**), the lowest number of the four s-levels closest to the particle's depth (**deplvl**), the z-coordinates of each rho s-level at the particle location at back, center and forward time (**Pwc_zb**, **Pwc_zc**, **Pwc_zf**), the total number of s-levels (**slvls**), the depth of the particle at back, center and forward time (**P_zb**, **P_zc**, **P_zf**), the external time step values in seconds for back, center, and forward time (**ex**), the internal time step values in seconds for back, center, and forward time (**ix**), the current iteration of the external time loop (**p**), and the version of output to return (**v**). Note that depending on the variable that is being interpolated, s-levels in the above descriptions could instead refer to w grid s-levels.

Output: The function returns the interpolated value at the particle's location at back time, center time, forward time, or a weighted average of the three with center time weighing four times as much as back and forward times. Which value is returned depends on the value passed to the function through the variable **v** (1 = back, 2 = center, 3 = forward, 4 = weighted average).

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses the subroutine TSPSI and function HVAL from TSPACK in the Tension Spline Module, as well as linint and polintd from the Interpolation Module.

Private Variables Used: The function uses no private variables.

Numerical Method: The function begins by taking the variable name passed in through **var**, concatenating b, c, or f on the end and storing it in the variables **varb**, **varc**, and **varf**. The function then interpolates the values along the four s-levels closest to the x-y location of the particle. Next, TSPACK fits a tension spline to the four points in the water column and uses it to calculate the value at the particle's location. This occurs for the each of the water column profiles from the back, center, and forward times of the external time step. These values are stored in the variable **ey** which is passed to the function polintd. If it is the first iteration of the external time step, the three values stored in **ey** are back time, back time (again), and center time, rather than back time, center time, and forward time. Next, a polynomial is used to interpolate the external time step salinity values to the particle's location at back, center, and forward time of the internal time step. The current value at the particle's location is then determined using a weighted average of these three values, with center time being weighted four times as heavily as back or forward time. The function then returns the value at back time, center time, forward time, or the weighted average, depending on the value of **v**.

Variables Definitions: The following variables are used in this section:

abb_vb – dp – abridged version of particle water column variables at the back time step (e.g. **Pwc_Sb**), containing the data for the 4 sigma levels closest to the particle's depth
abb_vc – dp – abridged version of particle water column variables at the center time step (e.g. **Pwc_Sc**), containing the data for the 4 sigma levels closest to the particle's depth
abb_vf – dp – abridged version of particle water column variables at the forward time step (e.g. **Pwc_Sf**), containing the data for the 4 sigma levels closest to the particle's depth
abb_zb – dp – abridged version of **Pwc_zb**, containing the data for just the 4 sigma levels closest to the particle's depth
abb_zc – dp – abridged version of **Pwc_zc**, containing the data for just the 4 sigma levels closest to the particle's depth
abb_zf – dp – abridged version of **Pwc_zf**, containing the data for just the 4 sigma levels closest to the particle's depth
anykey – character – for error state read statement 'Press Any Key'
deplvl – integer – lowest of the four consecutive s-levels (or w s-levels) closest to particle depth
ex – dp – external time step values in seconds for back, center, and forward
ey – dp – value at external time steps
i – integer – iteration variable
IER – integer – error indicator or iteration count (for TSPACK)
ix – dp – internal time step values in seconds for back, center, and forward times
nN – integer, parameter – number of s-levels used in tension splines
p – integer – external time step do loop iteration variable
P_V – dp – weighted average of the values at the particle's location at back, center, and forward internal time, with center time being weighted four times more heavily than back of forward time
P_vb – dp – value at the particle's location in the water column at back external time
P_vc – dp – value at the particle's location in the water column at center external time
P_vf – dp – value at the particle's location in the water column at forward external time
P_zb – dp – z- coordinate of the particle at back time
P_zc – dp – z- coordinate of the particle at center time
P_zf – dp – z- coordinate of the particle at forward time
Pwc_zb – dp – z-coordinates of each rho s-level at particle location at back time
Pwc_zc – dp – z-coordinates of each rho s-level at particle location at center time
Pwc_zf – dp – z-coordinates of each rho s-level at particle location at forward time
SigErr – integer – indicates error that TSPACK failed to converge
SIGM – dp – array containing tension factors from TSPSI
slope – dp – return variable of linint, not used in this function
slvls – integer – number of s-levels of the grid being used
v – integer – version of output (1 = back, 2 = center, 3 = forward, 4 = weighted average)
var – character array – input variable; data type to interpolate from
varb – character array – data type to interpolate from, specific for back time step
varc – character array – data type to interpolate from, specific for center time step
varf – character array – data type to interpolate from, specific for forward time step
vb – dp – value at the particle's location in the water column at the back internal time step

vc – dp – value at the particle's location in the water column at the center internal time step
vf – dp – value at the particle's location in the water column at the forward internal time step
Xpos – dp – x- coordinate of the particle
YP – dp – array containing derivatives from TSPSI
Ypos – dp – y- coordinate of the particle

T. Subroutine writeNetCDF

Overview: This subroutine writes output to a NetCDF file.

Input Variables: The subroutine has six required and four optional input variables. The subroutine requires the current model time (**time**) as well as the age, longitude and latitude coordinates, depth, and status of each particle (**age**, **lon**, **lat**, **depth**, **colors**). If salinity and temperature at each particle's location is being output, then the optional variables **Salt** and **Temp** may be included. If the model is tracking particle collision with bottom and land boundaries, the optional variables **hitB** and **hitL** may be included.

Output Variables: The subroutine has just no output variables.

Module parameters used: The subroutine uses **NCOutFile**, **NCtime**, **numpar**, **oilRun**, **outpath**, **outpathGiven**, **SaltTempOn**, and **TrackCollisions** from the Parameter Module.

Module procedures used: The subroutine uses no functions or subroutines from other modules. It does use NetCDF_90 functions.

Private Variables Used: The subroutine uses the variables **NCcount, NCstart,** and **prcount** which are private variables accessible only to the procedures in the Hydrodynamic Module.

Numerical Method: The subroutine first determines if the model is writing to multiple NetCDF files, and if so, whether or not a new file must be created. If **NCtime** is set to 0 in LTRANS.data then only one NetCDF output file is being created. If **NCtime** is greater than zero a new file is created by calling **createNetCDF** when the time elapsed is greater than the value in **NCtime**. In either case, the NetCDF file name is constructed and stored in the variable **ncFile**. This file is then opened using NF90_OPEN. Once opened, the required input variables **time**, **age**, **lon**, **lat**, **depth**, and **colors** are written to the file in the next file time step. If the optional variables **Salt**, **Temp**, **hitB**, or **hitL** are present, they too are written to the NetCDF output file. The subroutine then closes the file using NF_CLOSE and returns.

Variables Definitions: The following variables are used in this section:
age – dp – age of each particle
colorID – integer – id assigned to the particle status variable
colors – dp – status id of each particle
depth – dp – depth of each particle (m)
depthID – integer – id assigned to the particle depth variable

hitB – integer – the number of times each particle has hit bottom since the last model output
hitBID – integer – id assigned to the bottom collisions variable
hitL – integer – the number of times each particle has hit land since the last model output
hitLID – integer – id assigned to the bottom collisions variable
lat – dp – latitudinal position of each particle
latID – integer – id assigned to the particle latitude variable
lon – dp – longitudinal position of each particle
lonID – integer – id assigned to the particle longitude variable
modtimeID – integer – id assigned to the model time variable
NCcount – integer – when writing to multiple NetCDF files, this stores the current NetCDF file number used for the filename
NCelapsed – integer – model time elapsed since the creation of the last NetCDF output file
ncFile – character array – concatenated hydrodynamic output file name
NCID – integer – NetCDF ID used in NetCDF functions
NCOutFile – character array – name given to the NetCDF output file(s) if outputting to NetCDF files (i.e. writeNetCDF = .true.)
NCstart – integer – the time in the model that the current NetCDF output file was created; needed to determine if **NCtime** seconds have passed and a new output file needs to be created
NCtime – integer – time interval, in seconds, between the creation of new NetCDF output files
numpar – integer – total number of particles
outpath – character array – if **outpathGiven** = .true., this specifies the directory path in which to write the output files
outpathGiven – logical – if .TRUE. files are written to the path given in **outpath**
pageID – integer – id assigned to the particle age variable
prcount – integer – time step within the current NetCDF output file
Salt – dp – salinity at each particle's location
saltID – integer – id assigned to the particle salinity variable
SaltTempOn – logical – .TRUE. if calculate salinity and temperature at particle location, else .FALSE.
STATUS – integer – status ID returned from NetCDF functions
Temp – dp – temperature at each particle's location
tempID – integer – id assigned to the particle temperature variable
time – integer – current model time
TrackCollisions – logical – write files containing bottom and land collision information? .TRUE. = yes, .FALSE. = no

XII. Interpolation Module (interpolation_module.f90, INT_MOD)

Overview: The Interpolation Module contains two procedures that interpolate data. Subroutine linint uses linear interpolation, while subroutine polintd uses polynomial interpolation.

Public Procedures: The following are the public subroutines and functions contained within the Interpolation Module: **subroutine linint** and **function polintd**.

A. Subroutine linint

Overview: This subroutine uses linear interpolation to estimate a value (y) at a specified location (x-coordinate) based on two arrays of the same size that contain a series of x-y pairs. The array with x-values must be in strictly increasing order.

Input Variables: The subroutine has four input variables: an array of data (**ya**), the array containing the strictly increasing locations of those data (**xa**), the size of the two arrays (**n**), and the point location (**x**) within the range of the locations in **xa** array.

Output Variables: The subroutine has two output variables: the interpolated value (**y**) and the slope of the line used for linear interpolation (**m**).

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private variables used: The subroutine uses no private variables.

Numerical Method: The subroutine first uses a binary search algorithm to find the **xa** array point locations directly above and below the point that is being interpolated to (**x**). Then, the equation of the line that passes through these two points (paired with their corresponding **ya** values) is calculated and used to interpolate **x** to its corresponding **y** value. Once the slope is stored in **m** and the interpolated value stored in **y**, the subroutine returns.

Variables Definitions: The following variables are used in this section:

b – dp – x intercept of the line used for linear interpolation
jhi – integer – used in binary search algorithm; once the algorithm is finished, holds the array location directly above the point being interpolated to
jlo – integer – used in binary search algorithm; once the algorithm is finished, holds the array location directly below the point being interpolated to
k – integer – used in binary search algorithm, holds the midpoint location to be checked next
m – dp – slope of the line used for linear interpolation
n – integer – number of array locations in **xa** and **ya**
x – dp – location to be interpolated to
xa – dp – locations of the data in **ya** to be interpolated from
y – dp – linearly interpolated value at the location **x**

ya – dp – data at the locations in **xa** to be interpolated from

B. Function polintd

Overview: This function creates a polynomial using three points (in increasing order) and interpolates to a given location that lies within those three points.

Input Variables: The function has four input variables: the array of y-coordinates (**ya**), the array containing the strictly increasing x-coordinates (**xa**), the size of the two arrays (**n**), and the location to be interpolated to (**x**).

Output: The function returns the double precision value calculated by polynomial interpolation at the given location **x**.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: The function uses no private variables.

Numerical Method: This subroutine first determines which of the locations in the array **xa** are closest to the point being interpolated to. Next, the value of the variable **c** is calculated to be used in the final equation for polynomial interpolation. Then the values of **a** and **b** are calculated, dependent on which location in **xa** was closest to **x**. The **a**, **b**, and **c** values are then used in the final polynomial interpolation equation, along with the location to be interpolated to (**x**) and the values in **xa** and **ya** at the closest array location. The value returned from the final polynomial equation is then returned from the function.

Variables Definitions: The following variables are used in this section:
a – dp – calculated value for polynomial interpolation
b – dp – calculated value for polynomial interpolation
c – dp – calculated value for polynomial interpolation
dif – dp – distance from **x** to the closest **xa** location
dift – dp – when finding closest **xa** location; distance from **x** to the current **xa** location
i – integer – iteration variable
n – integer – number of array locations in **xa** and **ya**
ns – integer – index of **xa** that is closest to **x**
x – dp – location to be interpolated to
xa – dp – locations of the data in **ya** to be interpolated from
ya – dp – data at the locations in **xa** to be interpolated from

XIII. Norm Module (norm_module.f90, NORM_MOD)

Overview: The Norm Module contains the function **norm**, which returns a random number (a 'deviate') drawn from a normal distribution with zero mean and unit variance (i.e., standard deviation = 1).

Public Procedures: The following are the public subroutines and functions contained within the module: Function **norm**.

A. Function norm

Overview: This function returns a random number (a 'deviate') drawn from a normal distribution with zero mean and unit variance (i.e., standard deviation = 1).

Input Variables: The function has no input variables.

Output: The function returns the random deviate.

Module parameters used: The subroutine uses the parameter **PI** from the Parameter Module, which contains the value of the mathematical constant π.

Module procedures used: The subroutine uses the function **genrand_real3** from the Mersenne Twister program in the Random Number Generator Module (random_module.f90).

Private Variables Used: The subroutine uses no private variables.

Numerical Method: For a description of the basic equation, see the Box-Muller transform section in Wikipedia (http://en.wikipedia.org/wiki/Box-Muller_transform). Note that the function gasdev from Press et al. (1992) is based on the polar version of the Box-Muller transform and is more computationally efficient (but is not strictly open source as is the function Norm).

Output from the function Norm passed the following tests for normal distributions: Kolmogorov-Smirnov, Cramer-von Mises and Anderson-Darling (SAS 9.1., n = 1,000,000). Fig. 10 contains a histogram of the deviates used in these tests.

Variable Definitions: The following variables are used in this function:

- **dev1** - dp – a random deviate drawn from a uniform distribution between 0 and 1
- **dev2** - dp – a random deviate drawn from a uniform distribution between 0 and 1
- **pi** - dp – the value of the mathematical constant π

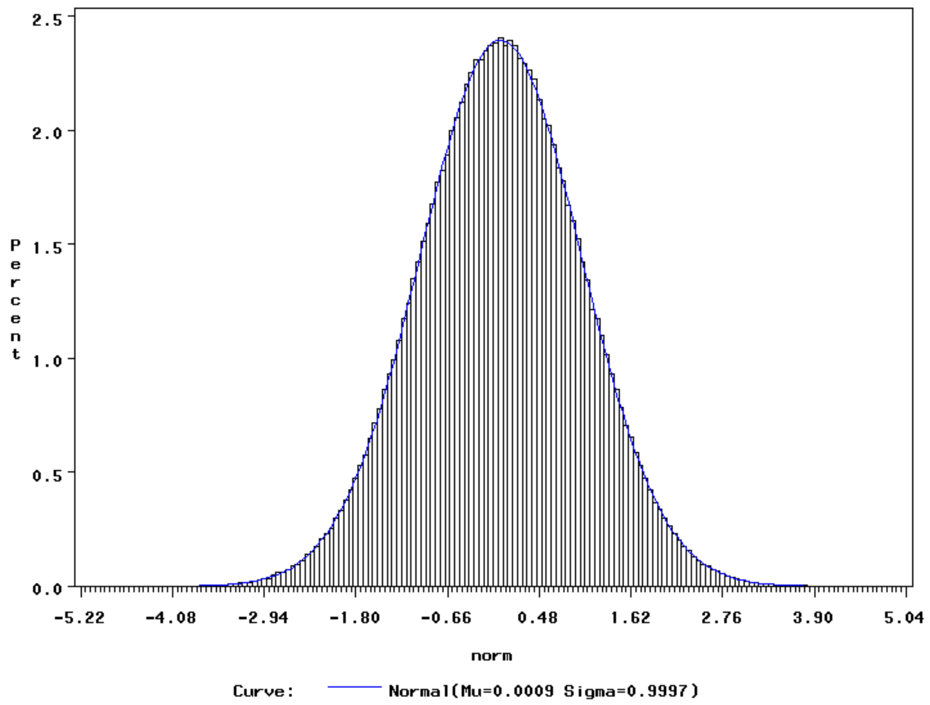

Fig. 10. Histogram of deviates derived from 1,000,000 calls of the function **norm**. The blue line indicates the expected value based on the formula for the normal curve.

XIV. Parameter Module (parameter_module.f90, PARAM_MOD)

Overview: The Parameter Module declares all the user specified parameters at compile time by reading in the header file LTRANS.h, making the parameters declared within the header file available to all the other modules. Then during run time, the module reads in the include file LTRANS.data which contains all the values of the user specified parameters. The user therefore only needs to change parameter values in LTRANS.data before rerunning a model, rather than having to recompile. Once the module has read in all the user specified values, it opens the file specified in the parameter **NCgridfile**, and determines the remainder of the necessary parameter values.

User Specified Parameters: The following are the parameters read in from LTRANS.data:

Behavior – integer – particle starting behavior (0 = passive, 1 = near-surface, 2 = near-bottom, 3 = DVM, 4 = *C. virginica* oyster larvae, 5 = *C. ariakensis* oyster larvae, 6 = constant sinking velocity)
BoundaryBLNs – logical – should the model output Surfer blanking files?; .TRUE. = yes, .FALSE. = no
constAks – dp – if **readAks** = .false, the constant salinity vertical diffusion coefficient (Aks)
ConstantHTurb – dp – value of constant horizontal diffusivity (m2/s)
constSalt – dp – if **readSalt** = .false, the constant value of salinity
constTemp – dp – if **readTemp** = .false, the constant value of temperature
constU – dp – if **readU** = .false, the constant value of u- component velocity
constV – dp – if **readV** = .false, the constant value of v- component velocity
constW – dp – if **readW** = .false, the constant value of w- component velocity
constZeta – dp – if **readZeta** = .false, the constant value of surface height
daylength – dp – length of daytime (hr); for diurnal vertical migration behavior type
days – real – number of days to run the model
deadage – dp – age in seconds at which a particle stops moving (i.e., dies)
dt – integer – length of external time step; interval between hydrodynamic inputs (s)
Earth_Radius – dp – equatorial radius
Em – dp – irradiance at solar noon
ErrorFlag – integer – flag to indicate what the model should do if an error is encountered; 0=stop model, 1=return particle to previous location and continue, 2=kill particle and continue, 3=set particle out of bounds and continue
ExeDir – character array – path to the model run executable; for documentation purposes
filenum – integer – number in the first hydrodynamic input file name
habitatfile – character array – name and path (if needed) of habitat polygon input file
hc – real – minimum hydrodynamic model depth; used in s-level transformations
hedges – integer – number of hole edge points in **holefile**
holefile – character array – name and path (if needed) of input file containing hole data
holesExist – logical – .TRUE. if holes exist in any habitat and will need to be read in
Hswimspeed – dp – horizontal swimming speed (m/s) for tidal stream transport behavior
HTurbOn – logical – .TRUE. if Horizontal Turbulence is to be turned on, else .FALSE.
idt – integer – length of internal (particle tracking) time step (s)
ijbuff – integer – the model only reads in hydrodynamic data for nodes in the vicinity of the particles' locations; this contains the size of the buffer around the particles' locations of

which to read. For instance if ijbuff = 4, then the model reads in an additional four nodes in each direction around the area that contains the particles.

Institution – character array – organization that is running the model; for documentation purposes

iprint – integer – interval in model time to wait between each output file (s)

Kd – dp – vertical attenuation coefficient

latmin – dp – if **SphericalProjection** = .TRUE., the latitude value of which metric conversions are based from; should be lower than any latitude values appearing within the model domain

lonmin – dp – if **SphericalProjection** = .TRUE., the longitude value of which metric conversions are based from; should be lower than any longitude values appearing within the model domain

maxholeid – integer – highest hole id number used

maxpolyid – integer – highest habitat polygon id number used

minholeid – integer – lowest hole id number used

minpolyid – integer – lowest habitat polygon id number used

mortality – logical – can the particles die?; .TRUE. = yes, .FALSE. = no

NCgridfile – character array – name and path (if needed) of the NetCDF grid file

NCOutFile – character array – name given to the NetCDF output file(s) if outputting to NetCDF files (i.e. writeNetCDF = .true.)

NCtime – integer – time interval, in seconds, between the creation of new NetCDF output files

Numdigits – integer – the number of digits in the Hydrodynamic Input Files' names, including leading zeros

numpar – integer – total number of particles

OpenOceanBoundary – logical – If particles can "escape" via open ocean boundaries, set this to TRUE; Escape means that the particle will stick to open ocean boundaries and stop moving

OutDir – character array – location of the output files from this model run; for documentation purposes

outpath – character array – if **outpathGiven** = .true., this specifies the directory path in which to write the output files

outpathGiven – logical – if .TRUE. files are written to the path given in **outpath**

parfile – character array – name and path (if needed) of the particle start location file

pedges – integer – number of habitat polygon edge points in **habitatfile**

pediage – dp – age at which particles reach their maximum swimming speed and are competent to settle (if settlement is turned on)

PI – dp – the mathematical constant π

prefix – character array – first part of hydrodynamic input file name (and path if needed)

readAks – logical – If .TRUE. read in salinity vertical diffusion coefficient (Aks) from NetCDF file, else use **constAks**

readSalt – logical – If .TRUE. read in salinity from NetCDF file, else use **constSalt**

readTemp – logical – If .TRUE. read in temp from NetCDF file, else use **constTemp**

readU – logical – If .TRUE. read in u- component velocity from NetCDF file, else use **constU**

readV – logical – If .TRUE. read in v- component velocity from NetCDF file, else use **constV**
readW – logical – If .TRUE. read in w- component velocity from NetCDF file, else use **constW**
readZeta – logical – If .TRUE. read in sea surface height from NetCDF file, else use **constZeta**
RunBy – character array – name of the person who setup/run the model; for documentation purposes
RunName – character array – Name given to this specific model run to identify it; for documentation purposes
SaltTempOn – logical – .TRUE. if calculate salinity and temperature at particle location, else .FALSE.
seed – integer – number used to initialize the random number generator Mersenne Twister
settlementon – logical - .TRUE. if the model is to use the Settlement Module, else .FALSE.
Sgradient – dp – salinity gradient threshold to cue larval behavior (psu/m)
sink – dp – constant sinking velocity for behavior type 6
SphericalProjection – logical – if .TRUE. use spherical projection, else use mercator
StartedOn – character array – date in which the model run was began; for documentation purposes
startfile – logical – does the first NetCDF hydrodynamic input file begin at 12:00am and contain an additional time step? .TRUE. = yes, .FALSE. = no
suffix – character array – final part of the hydrodynamic input file name
SVN_Version – character array – SVN Repository and version number; for documentation purposes
Swimdepth – dp – if tidal stream transport behavior is being used, depth maintained by the particle during flood time
swimfast – dp – maximum particle swimming speed (m/s)
swimslow – dp – when particle first begins swimming, initial speed (m/s)
swimstart – dp – age that particle swimming or sinking begins (s)
tdim – integer – size of the time dimension used in the hydrodynamic input files
thresh – dp – light threshold that cues behavior
TrackCollisions – logical – write files containing bottom and land collision information? .TRUE. = yes, .FALSE. = no
twiend – dp – time of twilight end (hr)
twistart – dp – time of twilight start (hr)
us – integer – number of depth levels in the rho, u, and v grids
Vtransform – integer – flag to indicate which transform to use in generating depth levels: 1-WikiRoms Eq. 1, 2-WikiRoms Eq. 2, 3-Song/Haidvogel 1994 Eq.
VTurbOn – logical – .TRUE. if Vertical Turbulence is to be turned on, else .FALSE.
writeCSV – logical – write particle location output to .csv files? .TRUE.=yes, .FALSE.=no
WriteHeaders – logical – write .txt files with column headers? .TRUE.=yes, .FALSE.=no
WriteModelTiming – logical – Write a .csv file with model timing data? .TRUE.=yes, .FALSE.=no
writeNC – logical – write particle location output to NetCDF files? .TRUE.=yes, .FALSE.=no
ws – integer – number of depth levels in the w grid

z0 – dp – ROMS roughness parameter

Grid File Generated Parameters: The following are the parameters whose values are obtained from the NetCDF Grid File:

max_rho_element – integer – maximum number of rho grid elements
max_u_element – integer – maximum number of u grid elements
max_v_element – integer – maximum number of v grid elements
rho_elements – integer – total number of wet rho elements (i.e. elements with at least one node masked as water)
rho_nodes – integer – total number of rho nodes
u_elements – integer – total number of wet u elements (i.e. elements with at least one node masked as water)
u_nodes – integer - total number of u nodes
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the u grid
v_elements – integer – total number of wet v elements (i.e. elements with at least one node masked as water)
v_nodes – integer - total number of v nodes
vi – integer – number of nodes across the v grid
vj – integer – number of nodes down the v grid

Public Procedures: The following are the public subroutines contained within the Parameter Module: errorHandler, getParams, and gridData.

A. Subroutine errorHandler

Overview: This subroutine handles any errors that occur within the Parameter Module.

Input Variables: This subroutine has two input variables: **header**, a character array containing a message related to the error, and **flag**, an integer which contains the value -1 if the error is fatal and requires the program to stop.

Output Variables: This subroutine has no output variables.

Module parameters used: This subroutine uses no variables from other modules.

Module procedures used: This subroutine uses no functions or subroutines from other modules.

Private Variables Used: This subroutine uses no variables private to this module.

Numerical Method: If the value of **flag** is -1, the error message in **header** is printed to the screen and the model is stopped. If **flag** is not -1, "('***** WARNING *****')" is printed to the screen followed by the error message in **header**, and the model continues.

Variable Definitions:

flag – integer – flag to control model behavior; if it contains -1 the model stops
header – character array – message associated with the encountered error

B. Subroutine getParams

Overview: This subroutine retrieves the parameter values from LTRANS.data then calls gridData to generate the remaining parameter values.

Input Variables: This subroutine has no input variables.

Output Variables: This subroutine has no output variables.

Module parameters used: This subroutine uses no variables from other modules.

Module procedures used: This subroutine uses no functions or subroutines from other modules.

Private Variables Used: This subroutine uses no variables private to this module.

Numerical Method: First, the subroutine initializes **err** to 0, to indicate that no errors have occurred. Next, LTRANS.data is opened and all the parameter values are read in through their respective namespaces (e.g. numparticles, timeparam, hydroparam, etc). Once this is complete, gridData is called to generate the remaining parameter values based on data from the NetCDF grid file specified in **NCgridfile**. If there were any problems encountered in the previous steps, the value of **err** was changed to reflect that particular problem, a message is saved in **header** to reflect that error, and errorHandler is called. If no problem was encountered the subroutine then decrements **lonmin** and **latmin** by 1 to reduce the chance of a rounding error, and the subroutine ends.

Variable Definitions:
See User Specified Parameters section above.

C. Subroutine gridData

Overview: This subroutine opens the NetCDF grid file and retrieves the values necessary to calculate the values for all the grid file generated parameters listed above.

Input Variables: This subroutine has no input variables.

Output Variables: This subroutine has one optional output variables. If **IOSTAT** is present, it returns an error code.

Module parameters used: This subroutine uses no variables from other modules.

Module procedures used: This subroutine uses no functions or subroutines from other modules.

Private Variables Used: This subroutine uses no variables private to this module.

Numerical Method: This subroutine opens up the NetCDF grid file specified in the variable **NCgridfile** in LTRANS.data and reads in the dimensions of the rho, u, and v grids. It then allocates the variables **mask_rho**, **mask_u**, and **mask_v** so that the masks may be read in from the grid file. With this information, the number of nodes and elements for each grid can be calculated. The mask for each grid is used to determine the total number of "wet" elements, which are elements which contain at least one node masked as water. The values calculated are stored in the variables listed above under "Grid File Generated Parameters." If any problems are encountered in the subroutine, the value of **err** is changed to reflect that particular problem. At the end of the subroutine, if **IOSTAT** is present, it is set to the value in **err**.

Variable Definitions:

dimcount – integer – dimension size returned from calls to NF90_INQUIRE_DIMENSION
dimid – integer – dimension id returned from calls to NF90_INQ_DIMID
err – integer – 0 if no errors are encountered, otherwise a specific error code
eta_rho – integer – number of nodes down the rho grid
eta_u – integer – number of nodes down the u grid
eta_v – integer – number of nodes down the v grid
GF_ID – integer – grid file id returned from call to NF90_OPEN
i – integer – iteration variable
IOSTAT – integer – optional return variable containing an error code if errors are encountered or 0 if not
j – integer – iteration variable
mask_rho – real – land/sea masking of the Rho grid in (i,j) location format
mask_u – real – land/sea masking of the U grid in (i,j) location format
mask_v – real – land/sea masking of the V grid in (i,j) location format
max_rho_element – integer – maximum number of rho grid elements
max_u_element – integer – maximum number of u grid elements
max_v_element – integer – maximum number of v grid elements
maxR – integer – total number of rho grid elements
maxU – integer – total number of u grid elements
maxV – integer – total number of v grid elements
nR – integer – total number of rho nodes
nU – integer – total number of u nodes
nV – integer – total number of v nodes
rho_elements – integer – total number of wet rho elements (i.e. elements with at least one node masked as water)
rho_nodes – integer – total number of rho nodes
STATUS – integer – status ID returned from NetCDF functions
u_elements – integer – total number of wet u elements (i.e. elements with at least one node masked as water)
u_nodes – integer - total number of u nodes
ui – integer – number of nodes across the u grid
uj – integer – number of grids down the u grid

v_elements – integer – total number of wet v elements (i.e. elements with at least one node masked as water)
v_nodes – integer - total number of v nodes
vi – integer – number of nodes across the v grid
VID – integer – variable ID used in NetCDF functions
vj – integer – number of nodes down the v grid
wetR – integer – total number of "wet" rho grid elements
wetU – integer – total number of "wet" u grid elements
wetV – integer – total number of "wet" v grid elements
xi_rho – integer – number of nodes across the rho grid
xi_u – integer – number of nodes across the u grid
xi_v – integer – number of nodes across the v grid

XV. Point-in-Polygon Module (point_in_polygon_module.f90, PIP_MOD)

Overview: The Point-in-Polygon Module contains one function, INPOLY, which determines if a point is inside or outside an irregularly shaped polygon using the 'crossings method', a 'point-in-polygon' technique. A ray, parallel to the x-coordinate axis, is shot from the point to the east. The number of times the ray intersects with the line segments of the polygon is calculated. If the number of intersections is odd, then the particle is within the polygon. If the number is even, then the particle is outside the polygon boundaries.

Public Procedures: The following are the public subroutines and functions contained within the Point-in-Polygon Module: Function **INPOLY**.

A. Function INPOLY

Overview: This function checks if a point is within the boundaries of an irregularly shaped polygon.

Input Variables: The function **INPOLY** has four required input variables and one optional input variable. It is passed the x- and y- locations (**x**,**y**) of the current particle, the number of edge points of the irregular polygon (**n**), and the x- and y- locations of the edge points (**e**). It may also be passed the logical variable **onin** which if .TRUE. indicates that a particle on an edge is considered to be in the polygon and if .FALSE. indicates that a particle on an edge is considered to be outside the polygon. If **onin** is not included, the function defaults to treating a particle on an edge as being inside the polygon.

Output: The function returns the logical value .TRUE. if the point is found to be inside the polygon and .FALSE. if it is found to be outside.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: The subroutine uses no functions or subroutines from other modules.

Private Variables Used: The subroutine uses no private variables.

Numerical Method: The first thing this function does is determine the value of **onout**, which if .TRUE. indicates that if the point is on an edge, it is out of bounds, and if .FALSE. indicates that a point on an edge is in bounds. If **onin** is present in the function call, then **onout** is the opposite of **onin**. If it is not present, the default value is .FALSE.. Next, the output variable **inpoly** is initialized to .TRUE. and the variable **crossed**, which counts the number of times the ray shot east from the point crosses polygon boundaries, is initialized to zero. The function is set up such that the output variable is initialized to .TRUE. and remains .TRUE. until proven .FALSE.

The first loop in function INPOLY determines whether each edge point is above, below, or shares the same y- coordinate with the point. If any edge point lies directly on the ray east from the point then **on** is set to .TRUE., indicating that an additional loop is going to be needed (see next paragraph). If any edge point shares the same x- and y- coordinate with the point then the function will return. The value returned is dependent on **onout**; if **onout** is .TRUE. then it returns .FALSE., else it returns .TRUE..

The next section of code covers the situation where one or more of the edge points lands on the ray shot east from the point. The function must check the edge points before and after the edge point that lies on the ray. If one is above and one is below then **crossed** is incremented; if they are both above or both below, **crossed** is not incremented as the ray does not actually cross the edge. The code can handle situations in which the edge point is the first or last in the polygon or multiple consecutive edge points lie on the ray. This section also handles the situation in which one edge point is on the ray and either the edge point before or the edge point after is not on, above, or below the ray, meaning that the edge crosses directly over the point. If this occurs, then the function returns dependent on **onout**; if **onout** is .TRUE. then it returns .FALSE., else it returns .TRUE..

The last section of code iterates through each of the edge segments. If the edge points that make up an edge segment are either both above, both below, or both to the left of the point, then the edge segment cannot cross the ray. If both are to the right of the point, with one above and one below then the edge segment crosses the ray and **crossed** is incremented. If the two edge points are on opposite sides of the point, both horizontally and vertically, then the equation of the line segment must be calculated and the equation solved for the point's y-coordinate. If the intersection occurs on the ray then **crossed** is incremented.

Lastly, INPOLY uses the mod function to determine if an odd or even number of crosses was counted. If the number of crosses was odd, the point is not in the polygon so the output variable **inpoly** is set to .FALSE. and returned. Otherwise, the function returns the initial output variable value of .TRUE..

Variables Definitions: The following variables are used in this section:

- **b** – dp – x intercept of linear equation for edge segment
- **crossed** – integer – counter for number of times ray crosses boundaries
- **e** – dp – x- and y- coordinates of edge points
- **first** – logical – used for rare situation where the first edge point is on the ray, stays .TRUE. until an edge point is found that is not on the ray
- **hilo** – integer – keeps track of whether each edge point is above (1), below (-1), or equal to (0) the y- coordinate of the point; used when an edge point lies on the ray
- **i** – integer – iteration variable
- **ix** – dp – x- coordinate of the intersection between the ray and a boundary segment
- **j** – integer – iteration variable
- **m** – dp – slope of linear equation for edge segment
- **n** – integer – number of edge points in the polygon passed in through **e**
- **on** – logical – initialized to .FALSE., set .TRUE. if an edge point is on the ray, to indicate that the second block of code needs to be run
- **onin** – logical – optional input variable to tell function whether or not a point on an edge segment is in bounds

onout – logical – actual variable used to determine output if the point is on an edge; .FALSE. indicates that a particle on an edge is considered in bounds and .TRUE. indicates that such a particle is out of bounds. If **onin** is present in the function call, **onout** is the opposite logical value. If **onin** is not present, then **onout** defaults to .FALSE..

x – dp – x- coordinate of the point

y – dp – y- coordinate of the point

XVI. Random Number Module (random_module.f90, RANDOM_MOD)

Overview: The following Mersenne Twister (MT) program, mt19937ar.f, is used to generate random numbers between 0 and 1 from a uniform distribution. The Mersenne Twister is a fast random number generator with a period of 2^{19937}-1. It was downloaded from the following website:
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/FORTRAN/mt19937ar.f

See the Mersenne Twister Home Page for more information:
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

The license web page for the Mersenne Twister (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/elicense.html) indicates that "Until 2001/4/6, MT had been distributed under GNU Public License, but after 2001/4/6, we decided to let MT be used for any purpose, including commercial use. 2002-versions mt19937ar.c, mt19937ar-cok.c are considered to be usable freely."

This program was converted to F90 by Zachary Schlag for use in LTRANS. The header text from the program is below. The program is first initialized in LTRANS.f90 with subroutine **init_genrand** and then the functions **genrand_real1** and **genrand_real3** are used to generate random deviates in the Behavior, Horizontal Turbulence and Vertical Turbulence Modules.

Input Variable: This program has just one input variable, **seed**, which is the number used to initialize the Mersenne Twister. The value of seed is set in the LTRANS.data file.

Output: The functions **genrand_real1** and **genrand_real3** return a random number drawn from a uniform distribution between 0 and 1, where **genrand_real1** may return the values 0 and 1, and **genrand_real3** may not.

Variables Definitions: The following variable is used in this section:
seed – integer – is the number used to initialize the Mersenne Twister.

```
! ************* Mersenne Twister ****************
!
!  A C-program for MT19937, with initialization improved 2002/1/26.
!  Coded by Takuji Nishimura and Makoto Matsumoto.
!
!  Before using, initialize the state by using init_genrand(seed)
!  or init_by_array(init_key, key_length).
!
!  Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
!  All rights reserved.
!  Copyright (C) 2005, Mutsuo Saito,
!  All rights reserved.
!
!  Redistribution and use in source and binary forms, with or without
!  modification, are permitted provided that the following conditions
!  are met:
!    1. Redistributions of source code must retain the above copyright
!       notice, this list of conditions and the following disclaimer.
```

```
!
!    2. Redistributions in binary form must reproduce the above copyright
!       notice, this list of conditions and the following disclaimer in the
!       documentation and/or other materials provided with the distribution.
!
!    3. The names of its contributors may not be used to ENDorse or promote
!       products derived from this software without specific prior written
!       permission.
!
!  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
!  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
!  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
!  A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
!  CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
!  EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
!  PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
!  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
!  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
!  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
!  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
!
!
!  Any feedback is very welcome.
!  http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
!  email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
!
!-----------------------------------------------------------------------
!  FORTRAN77 translation by Tsuyoshi TADA. (2005/12/19)
!
!  FORTRAN90 translation by Zachary Schlag (2008/08/29)
!
!     ---------- initialize routines ----------
!  SUBROUTINE init_genrand(seed): initialize with a seed
!  SUBROUTINE init_by_array(init_key,key_length): initialize by an array
!
!     ---------- generate FUNCTIONs ----------
!  INTEGER FUNCTION genrand_int32(): signed 32-bit INTEGER
!  INTEGER FUNCTION genrand_int31(): unsigned 31-bit INTEGER
!  DOUBLE PRECISION FUNCTION genrand_real1(): [0,1] with 32-bit resolution
!  DOUBLE PRECISION FUNCTION genrand_real2(): [0,1) with 32-bit resolution
!  DOUBLE PRECISION FUNCTION genrand_real3(): (0,1) with 32-bit resolution
!  DOUBLE PRECISION FUNCTION genrand_res53(): (0,1) with 53-bit resolution
!
!  This program uses the following non-standard intrinsics.
!    ishft(i,n): If n>0, shifts bits in i by n positions to left.
!                If n<0, shifts bits in i by n positions to right.
!    iand (i,j): Performs logical AND on corresponding bits of i and j.
!    ior  (i,j): Performs inclusive OR on corresponding bits of i and j.
!    ieor (i,j): Performs exclusive OR on corresponding bits of i and j.
```

XVII. Settlement Module (settlement_module.f90, SETTLEMENT_MOD)

Overview: The Settlement Module handles all code related to the settlement routine. This includes reading in the habitat polygons and holes, creating variables containing the specifications of the habitat polygons and holes, keeping track of the settlement status of every particle, and checking if the particle is within a habitat polygon and can settle. The module uses a point-in-polygon approach to determine if a particle is within the boundaries of any of the habitat polygons or holes.

Private Variables: The module contains ten variables and one derived data type accessible only in this module. The habitat polygon array, **polys**, contains attributes for each habitat polygon including id number, center longitude, center latitude, edge longitude, and edge latitude. The hole array, **holes**, has similar attributes for each hole, but it also includes a sixth attribute to keep track of in which habitat polygon the hole is located. The variables **maxbdis** and **maxhdis** contain the maximum distance from the center of each habitat polygon and hole to its furthest edge point. The array **settle** contains the settlement status of each particle (.FALSE. = not settled, .TRUE. = settled). The array **settletime** contains the age at which the particles are competent to settle. The variables **polyspecs** and **holespecs** contain the position in the **polys** and **holes** arrays at which the attributes for each habitat polygon or hole, respectively, begin to be listed, along with the number of edge points that make up that particular habitat polygon or hole. The variables **elepolys** and **polyholes** are both of derived data type **polyPerEle** and consist of an integer **numpoly** and an allocatable integer array **poly** of size **numpoly** when allocated. The variable **elepolys** contains, in **numpoly**, the number of habitat polygons within each element, and for elements where **numpoly** > 0, it contains in the array **poly** the id numbers of all the habitat polygons within that element. The variable **polyholes** contains, in **numpoly**, the number of holes within each habitat polygon, and for habitat polygons where **numpoly** > 0, it contains in the array **poly** the id numbers of all the holes within that habitat polygon.

Initialization: LTRANS is set up so that the Settlement Module must be ‘turned on’ in the LTRANS.data include file by setting the parameter **settlementon** = .TRUE.

Private Procedures: The following are the private subroutines and functions accessible only to the other procedures in the Settlement Module: subroutines **createPolySpecs, getHabitat, hsettle, and psettle.**

Public Procedures: The following are the public subroutines and functions contained within the Settlement Module: function **isSettled**, and subroutines **finSettlement**, **initSettlement**, and **testSettlement**.

A. Subroutine createPolySpecs

Overview: This subroutine fills the variables **polyspecs**, **elepolys**, **holespecs**, and **polyholes** with information to allow the program to check settlement more quickly. The variable **polyspecs** contains, for every habitat polygon, the location in array **polys** of the first edge point and the number of edge points. The variable **holespecs** contains the same information for hole edge points in the variable **holes**. The variables **elepolys** and **polyholes** are both of derived data type

polyPerEle, consisting of an integer **numpoly** and an allocatable integer array **poly** of size **numpoly** when allocated. The variable **elepolys** will contain, in **numpoly**, the number of habitat polygons within each element, and for elements where **numpoly** > 0, it will contain the id numbers of all the habitat polygons within that element in **poly**. The variable **polyholes** will contain, in **numpoly**, the number of holes within each habitat polygon**,** and for habitat polygons where **numpoly** > 0, it will contain the id numbers of all the holes within that habitat polygon in **poly**.

Input Variables: The subroutine createPolySpecs has no input variables.

Output Variables: The subroutine createPolySpecs has no output variables.

Module parameters used: This subroutine uses the parameters **hedges, holesExist, maxpolyid, minpolyid, pedges,** and **rho_elements** from the Parameter Module. The parameters **pedges** and **hedges** contain the number of polygon edges and hole edges in the **habitatfile** and **holefile** respectively. The logical parameter **holesExist** contains the value .TRUE. if holes exist in the habitat polygons, and .FALSE. if there are no holes in the habitat. The parameters **maxpolyid** and **minpolyid** contain the highest and lowest polygon identification numbers from the **habitatfile**. The parameter **rho_elements** contains the total number of wet rho elements (rho elements that contain water) in the model.

Module procedures used: This subroutine calls getR_ele from the Hydrodynamic Module, gridcell from the Gridcell Module, and inpoly from the Point-in-Polygon Module.

Private Variables Used: The subroutine uses the private variables **polys**, **maxbdis**, **holes**, **polyspecs**, **holespecs**, **elepolys**, and **polyholes** which are accessible only to this module.

Numerical Method: This subroutine first calls getR_ele to get the x- and y- locations of the rho elements in the variables **r_ele_x**, and **r_ele_y**, which will be used to fill **elepolys**. Next, **polyspecs** is filled by iterating through the variable **polys** and storing the array position of the first edge of each habitat polygon and the number of edges in each habitat polygon.

The subroutine must then iterate through each of the elements in order to fill **elepolys**. Since the number of habitat polygons is unknown for each element this is done using a dynamically allocated linked list. The variable **polyhead** points to the head of the list, **polytail** points to the tail end of the list, and the integer **count** is used to keep track of the number of habitat polygon ids in the list. The concept of a linked list is that **polyhead** will point to a derived object containing a link to the next object as well as any data necessary for the current object, in this case the data would be the habitat polygon id which is stored in the variable **num**. In this way a chain of objects can be formed of any length. For each element the variables **count** is initialized to 0 and the pointers **polyhead** and **polytail** are completely deallocated. The subroutine then iterates through all the habitat polygon edges. If any of the edges of a habitat polygon are in the element, **count** is incremented and the habitat polygon's id is stored in the linked list. If this is the first object in the list it is pointed to directly by **polyhead**, otherwise it is added to the tail end of the list which is pointed to by **polytail**. If none of the edges of a habitat polygon are in that element, the subroutine checks if any of the four element nodes are in that habitat polygon. This is done in case either an entire element lies within a habitat polygon or a habitat polygon edge crosses through an element without actually having an edge point in the element. If an element edge point is in the habitat polygon then **count** is incremented and the habitat polygon's id is added to the linked list. After all the habitat polygons have been tested, the number in **count** can be transferred to **numpoly** in **elepolys** for the current element. If there were any habitat polygons in the current element, **poly** in **elepolys** is allocated to the value of **count** and the habitat polygon ids are transferred from the linked list to **poly** in **elepolys**, and the linked list is deallocated.

If there are any holes that exist in the habitat polygons then **holespecs** and **polyholes** need to be filled as well. The variable **holespecs** is filled in the same way as **polyspecs** by iterating through **holes** and storing the array position of each hole's first edge point and the total number of edge points in each hole. Since the habitat polygon id that contains a given hole is read in with each hole edge point, the subroutine can iterate through the hole edge points for each habitat polygon and, if a hole is in the polygon, increment **count** and store the hole id in a linked list using **polyhead** and **polytail** in the same way it was done for **elepolys**. After all the holes have been tested, the number in **count** is transferred to **numpoly** in **polyholes** for the current habitat polygon. If there were any holes in the current habitat polygon then **poly** in **polyholes** is allocated as length **count** and the hole ids are transferred from the linked list and the list is deallocated.

Variables Definitions: The following variables are used in this section:

- **check** – logical – increases efficiency in checking if element nodes are in the habitat polygon or hole; if none of the nodes are within range of the polygon then **check** is .FALSE. and the test is skipped, else **check** is .TRUE. and the normal test occurs
- **checkele** – integer – rho element id passed to gridcell when filling **elepolys**
- **count** – integer – used to count polygons when filling **elepolys** and **polyholes**
- **dis** – dp – distance from element node to polygon center, tested against the maximum distance for the current habitat polygon in **maxbdis,** used with **check**
- **elepolys** – derived data type **polyPerEle -** the number of habitat polygons within each element is stored in the variable **numpoly.** For elements where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the habitat polygons within that element.
- **hedges** – integer – number of hole edge points in **holefile**
- **holes** – dp – id number, center longitude, center latitude, edge longitude, edge latitude, and associated habitat polygon id number for each hole
- **holesExist** – logical, parameter – .TRUE. if there are holes in habitat, .FALSE. if not
- **holespecs** – integer – the starting location in **holes** of each hole along with the number of edge points that make up that particular hole
- **i** – integer – iteration variable
- **j** – integer – iteration variable
- **k** – integer – iteration variable
- **maxbdis** – dp – maximum distance from the center of each habitat polygon to its farthest edge point
- **maxpolyid** – integer – highest habitat polygon id number used
- **minpolyid** – integer – lowest habitat polygon id number used

P_ele – integer – return variable from gridcell that is unused by createPolySpecs
pedges – integer – number of habitat polygon edge points in **habitatfile**
poly – dp – allocatable array, allocated to the number of edge points in the current habitat polygon when checking if the element nodes are in the polygon
polyhead – pointer of derived data type **polynum** – points to the head of a dynamically linked list that exists to temporarily contain the id numbers of all the habitat polygons in the current element, or all the holes in the current habitat polygon, before this data is transferred into **elepolys** or **polyholes**
polyholes – derived data type **polyPerEle** – number of holes within each habitat polygon is stored in the variable **numpoly.** For habitat polygons where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the holes within that habitat polygon
polys – dp – id number, center longitude, center latitude, edge longitude, and edge latitude for each habitat polygon
polyspecs – integer – the starting location in **polys** of each habitat polygon along with the number of edge points that make up that particular polygon
polytail – pointer of derived data type **polynum** – points to the tail of the dynamically linked list that is described above in **polyhead**
ptr – pointer of derived data type **polynum** – points to a node in the dynamically linked list that is described above in **polyhead**
r_ele_x – dp – x-location of the four nodes in each element
r_ele_y – dp – y-location of the four nodes in each element
rho_elements – integer, parameter – number of wet rho elements
triangle – integer – return variable from gridcell, 1 if in the element, 0 if not

B. Subroutine finSettlement

Overview: This subroutine deallocates all the dynamic arrays allocated by **initSettlement**.

Input Variables: The subroutine has just no input variables.

Output: The subroutine has just no output variables.

Module parameters used: The function uses no parameters from PARAM_MOD.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: This subroutine deallocates the variables **elepolys, holes, holespecs, maxbdis, maxhdis, polyholes, polys, polyspecs, settle,** and **settletime** which are private variables accessible only to the procedures in the Settlement Module.

Numerical Method: This function deallocates the variables **elepolys, holes, holespecs, maxbdis, maxhdis, polyholes, polys, polyspecs, settle,** and **settletime** which are private variables accessible only to the procedures in the Settlement Module.

Variables Definitions: The following variables are used in this section:

elepolys – derived data type **polyPerEle** - the number of habitat polygons within each element is stored in the variable **numpoly.** For elements where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the habitat polygons within that element.

holes – dp – id number, center longitude, center latitude, edge longitude, edge latitude, and associated habitat polygon id number for each hole

holespecs – integer – the starting location in **holes** of each hole along with the number of edge points that make up that particular hole

maxbdis – dp – maximum distance from the center of each habitat polygon to its farthest edge point

maxhdis – dp – maximum distance from the center of each hole polygon to its farthest edge point

polyholes – derived data type **polyPerEle** – number of holes within each habitat polygon is stored in the variable **numpoly.** For habitat polygons where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the holes within that habitat polygon

polys – dp – id number, center longitude, center latitude, edge longitude, and edge latitude for each habitat polygon

polyspecs – integer – the starting location in **polys** of each habitat polygon along with the number of edge points that make up that particular polygon

settle – logical – whether each particle has settled (.TRUE.) or not settled (.FALSE.)

settletime – dp – age at which the particles are competent to settle

C. Subroutine getHabitat

Overview: This subroutine reads in the habitat polygon and hole locations.

Input Variables: The subroutine has no input variables.

Output Variables: The subroutine has no output variables

Module parameters used: This subroutine uses the logical parameter **holesExist** from the Parameter Module, which contains the value .TRUE. if holes exist in the habitat polygons, and .FALSE. if there are no holes in habitat. It also borrows from the Parameter Module the parameters **habitatfile** and **holefile** which contain in character arrays the name and path (if needed) of the habitat polygon and hole input files. The parameters **pedges** and **hedges** contain the number of polygon edges and hole edges in the **habitatfile** and **holefile** respectively.

Module procedures used: This subroutine calls lon2x and lat2y from the Conversion Module.

Private Variables Used: The subroutine uses private variables **polys**, **holes**, **maxbdis**, and **maxhdis**, which are accessible only to this module.

Numerical Method: The subroutine starts by opening the habitat polygon input file, **habitatfile**. It iterates through each habitat polygon, reading the habitat polygon information into the variable **P_lonlat**. The longitude and latitude that were just read in are then converted to x- and y- coordinates, using the functions **lon2x** and **lat2y**, and saved in the variable **polys**. This is followed by a loop that determines the distance from the center of each habitat polygon to its farthest edge point and saves that information in **maxbdis** to increase the efficiency of other search routines. If **holesExist** is .TRUE., indicating that holes exist in habitat, then the same process is repeated for holes. The hole information is initially read from the hole file, **holefile**, into **H_lonlat** and then converted and stored in **holes**. The distance from the center to the farthest edge point for each hole is then calculated and stored in the variable **maxhdis**.

Variables Definitions: The following variables are used in this section:

- **curpoly** – integer – id of the current polygon, used when calculating **maxbdis** and **maxhdis**
- **dise** – dp – distance from the center of the current polygon to the current polygon edge point, used when calculating **maxbdis** and **maxhdis**
- **H_lonlat –** dp – latitude and longitude hole data read in from **holefile**
- **habitatfile** – character array, parameter – the file and path (if needed) of the habitat polygon data
- **hedges** – integer, parameter – total number of hole edges
- **holefile** – character array, parameter – the file and path (if needed) of the hole data
- **holes** – dp – id number, center longitude, center latitude, edge longitude, edge latitude, and associated habitat polygon id number for each hole
- **holesExist** – logical, parameter – .TRUE. if there are holes in the habitat, else .FALSE.
- **i** – integer – iteration variable
- **maxbdis** – dp – maximum distance from the center of each habitat polygon to its farthest edge point
- **maxhdis** – dp – maximum distance from the center of each hole to its farthest edge point
- **P_lonlat –** dp – latitude and longitude habitat polygon data read in from **habitatfile**
- **pedges** – integer, parameter – total number of habitat polygon edges
- **polys** – dp – id number, center longitude, center latitude, edge longitude, and edge latitude for each habitat polygon

D. Subroutine hsettle

Overview: This subroutine checks if the current particle is within the boundaries of any holes in a particular habitat polygon.

Input Variables: The subroutine hsettle has two input variables, the x- and y- locations (**Px**,**Py**) of the current particle.

Input/Output Variables: The subroutine has one variable that is used for both input and output: **holein**. It is passed the id of the habitat polygon to check for holes. The value it returns is 0 if the particle is not in a hole or the id of the hole if it is in one.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: This subroutine calls INPOLY from the Point-in-Polygon Module.

Private Variables Used: The subroutine uses the private variables **polyholes**, **holespecs**, **poly**, **holes**, and **maxhdis**, which are accessible only to this module.

Numerical Method: The subroutine first initializes **polyin** to the value passed in through **holein,** then gives **holein** a value of zero to prepare it for use as output. The subroutine then checks **polyholes** to see if there are any holes in the current habitat polygon. If there are not, the subroutine ends. If holes do exist in the current habitat polygon, then the subroutine iterates through the holes in that polygon, whose ids are stored in **polyholes**. For each hole, the location and number of edge points of that hole are retrieved from **holespecs**, the array **polybnds** is allocated to the number of edge points, and the edge point x- and y- coordinates are read in from **holes**. Once that is done the subroutine can check if the particle is inside the hole by calling INPOLY with optional argument **onin** set to .FALSE. so that, if the particle is on an edge, it is not considered to be inside the hole. If the particle is found to be inside a hole, the subroutine exits, returning the id of the hole it is in through the variable **holein**. If all the holes are checked and the particle is not in any hole, then the subroutine returns with a zero in **holein**.

Variables Definitions: The following variables are used in this section:

- **dis** – dp – distance from the particle's location to the hole's center location, tested against the maximum distance for the current hole in **maxhdis**
- **holein** – integer – input/output variable; inputs the habitat polygon to check; outputs the id of the hole if the particle is in a hole, or zero if it is not in a hole
- **holes** – dp – id number, center longitude, center latitude, edge longitude, edge latitude, and associated habitat polygon id number for each hole
- **holespecs** – integer – the starting location in **holes** of each hole along with the number of edge points that make up that particular hole
- **i** – integer – iteration variable
- **j** – integer – iteration variable
- **maxhdis** – dp – maximum distance from the center of each hole to its farthest edge point
- **polybnds** – dp – allocatable array, allocated to the number of edge points in the hole that is currently being checked
- **polyholes** – derived data type **polyPerEle** – number of holes within each habitat polygon is stored in the variable **numpoly.** For habitat polygons where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the holes within that habitat polygon
- **polyin** – integer – holds the id of the habitat polygon passed in by **holein**
- **Px** – dp – x- location of the particle
- **Py** – dp – y- location of the particle
- **size** – integer – the number of edge points that make up the hole currently being checked, obtained from **holespecs**

start – integer – the location in **holes** of the first edge point of the hole currently being checked, obtained from **holespecs**

E. Subroutine initSettlement

Overview: This subroutine initializes the Settlement Module.

Input Variables: The subroutine initSettlement has one input variable, the double precision array **P_pediage**, which indicates at what age the particles are competent to settle. The values in **P_pediage** are transferred to the private Settlement Module variable **settletime**.

Output Variables: The module has no output variables.

Module parameters used: This subroutine uses the parameters **hedges, maxholeid, maxpolyid, minholeid, minpolyid, numpar, pedges,** and **rho_elements** from the Parameter Module.

Module procedures used: This subroutine calls getHabitat and createPolySpecs which are both private subroutines also located in the Settlement Module.

Private Variables Used: This subroutine allocates the variables **elepolys, holes, holespecs, maxbdis, maxhdis, polyholes, polys, polyspecs, settle,** and **settletime** which are private variables accessible only to the procedures in the Settlement Module.

Numerical Method: initSettlement starts by allocating the variables **elepolys, holes, holespecs, maxbdis, maxhdis, polyholes, polys, polyspecs, settle,** and **settletime** using values from the Parameter Module. Then it initializes the variable **settle** to .FALSE., meaning that all of the particles begin not settled. Next, it initializes the values in **settletime** by transferring the values from the input variable **P_pediage**. Lastly, it calls the two private subroutines getHabitat and createPolySpecs which read in the habitat polygons and holes and create special variables containing details about the habitat and holes to speed up the settlement routine.

Variables Definitions: The following variables are used in this section:

elepolys – derived data type **polyPerEle** - the number of habitat polygons within each element is stored in the variable **numpoly.** For elements where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the habitat polygons within that element.

hedges – integer – number of hole edge points in **holefile**

holes – dp – id number, center longitude, center latitude, edge longitude, edge latitude, and associated habitat polygon id number for each hole

holespecs – integer – the starting location in **holes** of each hole along with the number of edge points that make up that particular hole

maxbdis – dp – maximum distance from the center of each habitat polygon to its farthest edge point

maxhdis – dp – maximum distance from the center of each hole polygon to its farthest edge point
maxholeid – integer – highest hole id number used
maxpolyid – integer – highest habitat polygon id number used
minholeid – integer – lowest hole id number used
minpolyid – integer – lowest habitat polygon id number used
n – integer – iteration variable
numpar – integer, parameter – total number of particles
P_pediage – dp – age at which particles become pediveligers and are competent to settle
pedges – integer – number of habitat polygon edge points in **habitatfile**
polyholes – derived data type **polyPerEle** – number of holes within each habitat polygon is stored in the variable **numpoly.** For habitat polygons where **numpoly** > 0, the array **poly** is allocated to size **numpoly** and contains the id numbers of all the holes within that habitat polygon
polys – dp – id number, center longitude, center latitude, edge longitude, and edge latitude for each habitat polygon
polyspecs – integer – the starting location in **polys** of each habitat polygon along with the number of edge points that make up that particular polygon
settle – logical – whether each particle has settled (.TRUE.) or not settled (.FALSE.)
settletime – dp – age at which the particles are competent to settle

F. Function isSettled

Overview: This function checks if the settlement status of the current particle is set to .TRUE., meaning the particle has settled.

Input Variables: The function has one input variable, the particle id number of the current particle (**n**).

Output: The function returns the logical value .TRUE. if the current particle has settled, and .FALSE. if it has not.

Module parameters used: This function uses the parameter **settlementon** from the Parameter Module.

Module procedures used: The function uses no functions or subroutines from other modules.

Private Variables Used: This function uses the variable **settle** which is a private variable accessible only to the procedures in the Settlement Module.

Numerical Method: If the settlement has been turned on by setting **settlementon** to .TRUE. in LTRANS.data, then the value returned is equal to the given particle's settlement value in settle (.TRUE. if settled, .FALSE. if not settled). Otherwise, the return value is set to .FALSE., as particles cannot settle if settlement is not turned on.

Variables Definitions: The following variables are used in this section:

n – integer – id number of the current particle

settle – integer – array containing the settlement status of each particle

isSettled – logical – the output variable of the Settled function (.TRUE. if the particle has "settled", and .FALSE. if not)

G. Subroutine psettle

Overview: This subroutine checks if the current particle is within the boundaries of any habitat polygons in a particular rho element.

Input Variables: Subroutine psettle has three input variables: the x- and y- locations of the current particle (**Px**,**Py**) and the rho element in which to search (**R_ele**).

Output Variables: The subroutine has one output variable. It returns the variable **polyin** which contains the id of the habitat polygon in which the particle lies or, if it is not in any habitat polygon, the value zero.

Module parameters used: The subroutine uses no parameters from PARAM_MOD.

Module procedures used: This subroutine calls inpoly from the Point-in-Polygon Module.

Private Variables Used: The subroutine uses the private variables **elepolys**, **polyspecs**, **polys**, and **maxbdis**, which are accessible only to this module.

Numerical Method: The subroutine begins by initializing **polyin** to zero. It then checks **elepolys** to see if there are any habitat polygons in the current rho element. If there are not, the subroutine ends. If habitat polygons do exist in the current rho element, the subroutine iterates through the habitat polygons in that element whose ids are stored in **elepolys**. For each habitat polygon, the location and number of edge points of that polygon are retrieved from **polyspecs**, the array **polybnds** is allocated to the number of edge points, and the edge point x- and y-coordinates are read in from **polys**. The subroutine can then check if the particle is inside the habitat polygon by calling inpoly. If the particle is found to be inside a habitat polygon, the subroutine exits, returning the id of the polygon it is in through the variable **polyin**. If all the habitat polygons are checked and the particle is not in a habitat polygon, the subroutine returns with **polyin** still set to its initial zero.

Variables Definitions: The following variables are used in this section:

dis – dp – distance from particle's location to the current habitat polygon's center location, tested against the maximum distance for the current polygon in **maxbdis**

elepolys – derived data type **polyPerEle** – number of habitat polygons within each rho element is stored in the variable **numpoly.** For elements where **numpoly** > 0, the array

poly is allocated to size **numpoly** and contains the id numbers of all the habitat polygons within that rho element

i – integer – iteration variable

j – integer – iteration variable

maxbdis – dp – maximum distance from the center of each habitat polygon to its farthest edge point

polybnds – dp – allocatable array, allocated to the number of edge points in the habitat polygon that is currently being checked

polyin – integer – output variable; holds the id of the habitat polygon containing the particle, or zero if the particle is not within a habitat polygon

polys – dp – id number, center longitude, center latitude, edge longitude, and edge latitude, for each habitat polygon

polyspecs – integer – the starting location in **polys** of each habitat polygon along with the number of edge points that make up that particular polygon

Px – dp – x- location of the particle

Py – dp – y- location of the particle

size – integer – the number of edge points that make up the habitat polygon currently being checked, obtained from **polyspecs**

start – integer – the location in **polys** of the first edge point of the habitat polygon currently being checked, obtained from **polyspecs**

H. Subroutine testSettlement

Overview: This subroutine checks if the current particle is able to settle at its present age and location.

Input Variables: The subroutine settlement has four input variables: the age (**P_age**), number (**n**), and x- and y- locations (**Px**,**Py**) of the current particle.

Output Variables: The subroutine has one output variable. It returns the variable **inpoly** which contains 0 if the particle cannot settle or, if the particle can settle, the id of the habitat polygon in which it settles.

Module parameters used: This subroutine uses the logical parameter **holesExist** from the Parameter Module, which contains the value .TRUE. if holes exist in the habitat polygons and .FALSE. if there are no holes in the habitat.

Module procedures used: This subroutine calls getP_r_element from the Hydrodynamic Module. It also calls psettle and hsettle which are both private subroutines also located in the Settlement Module.

Private Variables Used: This subroutine uses the private variable **settle**, which is accessible only to this module.

Numerical Method: The first thing settlement does is call getP_r_element to find out which rho element the current particle is in. **inpoly** is initialized to 0 to indicate that the particle has not settled. Next, the particle's age is checked to see if it is greater than the age at which the particle is competent to settle. If the particle is not old enough to settle, the subroutine exits. Otherwise, it calls psettle, which checks if the particle is in any of the habitat polygons in the same element as the particle. If it is not in a habitat polygon then the subroutine exits. If it is in a habitat polygon and holes exist in habitat polygons, hsettle is called to find out if it is in a hole in the habitat polygon. If it is found to be in a hole, then **inpoly** is reset to 0; otherwise, **inpoly** is set to the id of the habitat polygon in which it is settling, and settle is set to 1 for the current particle, to change the settlement status of this particle to settled.

Variables Definitions: The following variables are used in this section:

n – integer – iteration variable

inpoly – integer – output variable; returns 0 if particle does not settle and the habitat polygon id that it settles in if it does settle

P_age – dp – input variable; the current age of the particle (s)

polyin – integer – used for output from psettle and hsettle

Px – dp – particle's x- coordinate

Py – dp – particle's y- coordinate

R_ele – integer – stores the id number of the rho element the current particle is in after it is returned from getP_r_element

XVIII. Tension Spline Module (tension_module.for, TENSION_MOD)

Overview: The Tension Spline Module is use to fit a tension spline curve to a water column profile of water properties at the particle location. This module uses a modified version of **Tension Spline Curve Fitting Package (TSPACK).** TSPACK (TOMS/716) was created by Robert J. Renka (renka@cs.unt.edu, Department of Computer Science and Engineering, University of North Texas) and is available for download from http://www. netlib.org and http://portal.acm.org/citation.cfm?id=151277. TSPACK fits tension splines to data that preserve the concavity and monotonicity of the data (Fig. 3). The routines in TSPACK are highly articulate and produce excellent profiles, although they may be somewhat computationally demanding because an individual tension factor is estimated for each segment of the profile. The tests of the random displacement model for vertical sub-grid scale turbulence (North et al. 2006a) were undertaken with TSPACK. Occasionally, the curve fitting method would fail to converge. In the North et al. (2006a) simulations, this occurred 0.0004% of the time, or once in 244,500 calls to TSPACK. In these rare cases, simple linear interpolation of the vertical profile was used to avoid program termination. LTRANS also uses simple interpolation to avoid program termination if TSPACK fails to converge.

TSPACK is copyrighted by the Association for Computing Machinery (ACM). With the permission of Dr. Renka and ACM, TSPACK was modified for use in LTRANS by removing unused code and call variables and updating it to Fortran 90. If you would like to use LTRANS with the modified TSPACK software, please read and respect the ACM Software Copyright and License Agreement (http://www.acm.org/publications/policies/softwarecrnotice). For noncommercial use, ACM grants "a royalty-free, nonexclusive right to execute, copy, modify and distribute both the binary and source code solely for academic, research and other similar noncommercial uses" subject to the conditions noted in the license agreement. Note that if you plan commercial use of LTRANS with the modified TSPACK software, you must contact ACM at permissions@acm.org to arrange an appropriate license. It may require payment of a license fee for commercial use.

This program was modified by Zachary Schlag for use in LTRANS. The following subroutines and functions from TSPACK are used in LTRANS: subroutines **TSPSI**, **SIGS**, **SNHCSH**, **YPC1**, and functions **HVAL**, **HPVAL**, **STORE**, **INTRVL**. The header text within TSPACK provides extensive documentation for the subroutines, functions and variables used within this module. This text is reproduced below.

**************************** TSPACK DOCUMENTATION **********************************

TSPACK: Tension Spline Curve Fitting Package

Robert J. Renka
05/27/91

I. INTRODUCTION

The primary purpose of TSPACK is to construct a smooth function which interpolates a discrete set of data points. The function may be required to have either one or two continuous derivatives, and, in the C-2 case, several options are provided for selecting end conditions. If the accuracy of the data does not warrant interpolation, a smoothing function (which does not pass through the data points) may be constructed instead. The fitting method is designed to avoid extraneous inflection points (associated with rapidly varying data values) and preserve local shape properties of the data (monotonicity and convexity), or to satisfy the more general constraints of bounds on function values or first derivatives. The package also provides a parametric representation for constructing general planar curves and space curves.

The fitting function h(x) (or each component h(t) in the case of a parametric curve) is defined locally, on each interval associated with a pair of adjacent abscissae (knots), by its values and first derivatives at the endpoints of the interval, along with a nonnegative tension factor SIGMA associated with the interval (h is a Hermite interpolatory tension spline). With SIGMA = 0, h is the cubic function defined by the endpoint values and derivatives, and, as SIGMA increases, h approaches the linear interpolant of the endpoint values. Since the linear interpolant preserves positivity, monotonicity, and convexity of the data, h can be forced to preserve these properties by choosing SIGMA sufficiently large. Also, since SIGMA varies with intervals, no more tension than necessary is used in each interval, resulting in a better fit and greater efficiency than is achieved with a single constant tension factor.

II. USAGE

TSPACK must be linked to a driver program which reserves storage, reads a data set, and calls the appropriate subprograms selected from those described below in section III.B. Header comments in the software modules provide details regarding the specification of input parameters and the work space requirements. It is recommended that curves be plotted in order to assess their appropriateness for the application. This requires a user-supplied graphics package.

III. SOFTWARE

A) Code

The code is written in 1977 ANSI Standard Fortran. All variable and array names conform to the default typing convention: I-N for type INTEGER and A-H or O-Z for type REAL. (There are no DOUBLE PRECISION variables.) There are 32 modules, and they are ordered alphabetically in the package. Each consists of the following sections:

1) the module name and parameter list with spaces separating the parameters into one to three subsets: input parameters, I/O parameters, and output parameters (in that order);
2) type statements in which all parameters are typed

and arrays are dimensioned;

3) a heading with the name of the package, identification of the author, and date of the most recent modification to the module;
4) a description of the module's purpose and other relevant information for the user;
5) input parameter descriptions and output parameter descriptions in the same order as the parameter list;
6) a list of other modules required (called either directly or indirectly);
7) a list of intrinsic functions called, if any; and
8) the code, including comments.

Note that it is assumed that floating point underflow results in assignment of the value zero. If not the default, this may be specified as either a compiler option or an operating system option. Also, overflow is avoided by restricting arguments to the exponential function EXP to have value at most SBIG=85. SBIG, which appears in DATA statements in the evaluation functions, HVAL, HPVAL, HPPVAL, and TSINTL, must be decreased if it is necessary to accomodate a floating point number system with fewer than 8 bits in the exponent. No other system dependencies are present in the code.

The modules that solve nonlinear equations, SIGS, SIGBP, SIG0, SIG1, SIG2, and SMCRV, include diagnostic print capability which allows the iteration to be traced. This can be enabled by altering logical unit number LUN in a DATA statement in the relevant module.

B) Module Descriptions

The software modules are divided into three categories, referred to as level 1, level 2, and level 3, corresponding to the hierarchy of calling sequences: level 1 modules call level 2 modules which call level 3 modules. For most applications, the driver need only call two level 1 modules -- one from each of groups (a) and (b). However, additional control over various options can be obtained by directly calling level 2 modules. Also, additional fitting methods, such as parametric smoothing, can be obtained by calling level 2 modules. Note that, in the case of smoothing or C-2 interpolation with automatically selected tension, the use of level 2 modules requires that an iteration be placed around the computation of knot derivatives and tension factors.

1) Level 1 modules

These are divided into two groups.

a) The following modules return knots (in the parametric case), knot derivatives, tension factors, and, in the case of smoothing, knot function values, which define the fitting function (or functions in the parametric case). The naming convention should be evident from the descriptions.

TSPSI   Subroutine which constructs a shape-preserving or unconstrained interpolatory function. Refer to TSVAL1.

TSPSS   Subroutine which constructs a shape-preserving or unconstrained smoothing spline. Refer to TSVAL1.

TSPSP   Subroutine which constructs a shape-preserving or unconstrained planar curve or space curve. Refer to TSVAL2 and TSVAL3.

TSPBI   Subroutine which constructs a bounds-constrained interpolatory function. The constraints are defined by a user-supplied array containing upper and lower bounds on function values and first derivatives, along with required signs for the second derivative, for each interval. Refer to TSVAL1.

TSPBP   Subroutine which constructs a bounds-constrained parametric planar curve. The constraints are defined by user-supplied arrays containing upper and lower bounds on the signed perpendicular distance between the smooth curve segment and the polygonal line segment associated with each knot interval. The bounds might, for example, be chosen to avoid intersections between smooth contour curves. Refer to TSVAL2.

b) The following modules return values, derivatives, or integrals of the fitting function(s).

TSVAL1 Subroutine which evaluates a Hermite interpolatory tension spline or its first or second derivative at a user-specified set of points. Note that a smoothing curve constructed by TSPSS is the interpolant of the computed knot function values. The evaluation points need not lie in the interval defined by the knots, but care must be exercised in assessing the accuracy of extrapolation.

TSVAL2 Subroutine which returns values or derivatives of a pair of Hermite interpolatory tension splines which form the components of a parametric planar curve. The output values may be used to construct unit tangent vectors, curvature vectors, etc.

TSVAL3 Subroutine which returns values or derivatives of three Hermite interpolatory tension splines which form the components of a parametric space curve.

TSINTL Function which returns the integral over a specified domain of a Hermite interpolatory tension spline. This provides an effective means of quadrature for a function defined only by a discrete set of values.

2) Level 2 modules

These are divided into four groups.

a) The following modules are called by TSPSP and TSPBP to obtain a sequence of knots (parameter values) associated with a parametric curve. For some data sets, it might be advantageous to replace these with routines that implement an alternative method of parameterization.

ARCL2D Subroutine which computes the sequence of cumulative arc lengths associated with a sequence of points in the plane.

ARCL3D Subroutine which computes the sequence of cumulative arc lengths associated with a sequence of points in

3-space.

b) The following modules are called by the level 1, group (a) modules to obtain knot derivatives (and values in the case of SMCRV).

YPC1	Subroutine which employs a monotonicity-constrained quadratic interpolation method to compute locally defined derivative estimates, resulting in a C-1 fit.
YPC1P	Subroutine similar to YPC1 for the case of periodic end conditions. In the case of a parametric curve fit, periodic end conditions are necessary to obtain a closed curve.
YPC2	Subroutine which determines a set of knot-derivative estimates which result in a tension spline with two continuous derivatives and satisfying specified end conditions.
YPC2P	Subroutine similar to YPC2 for the case of periodic end conditions.
SMCRV	Subroutine which, given a sequence of abscissae with associated data values and tension factors, returns a set of function values and first derivative values defining a twice-continuously differentiable tension spline which smoothes the data and satisfies either natural or periodic end conditions.

c) The following modules are called by the level 1, group (a) modules to obtain tension factors associated with knot intervals.

SIGS	Subroutine which, given a sequence of abscissae, function values, and first derivative values, determines the set of minimum tension factors for which the Hermite interpolatory tension spline preserves local shape properties (monotonicity and convexity) of the data. SIGS is called by TSPSI, TSPSS, and TSPSP.
SIGBI	Subroutine which, given a sequence of abscissae, function values, and first derivative values, determines the set of minimum tension factors for which the Hermite interpolatory tension spline satisfies specified bounds constraints. SIGBI is called by TSPBI.
SIGBP	Subroutine which, given a sequence of points in the plane with associated derivative vectors, determines the set of minimum tension factors for which the parametric planar tension spline curve defined by the data satisfies specified bounds on the signed orthogonal distance between the parametric curve and the polygonal curve defined by the points. SIGBP is called by TSPBP.

d) The following functions are called by the level 1, group (b) modules to obtain values and derivatives. These provide a more convenient alternative to the level 1 routines when a single value is needed.

HVAL   Function which evaluates a Hermite interpolatory tension spline at a specified point.

HPVAL   Function which evaluates the first derivative of a Hermite interpolatory tension spline at a specified point.

HPPVAL Function which evaluates the second derivative of a Hermite interpolatory tension spline at a specified point.

3) Level 3 modules

These are divided into three groups.

a) The following functions are called by SIGBI to compute tension factors, and are convenient for obtaining an optimal tension factor associated with a single interval.

SIG0   Function which, given a pair of abscissae, along with associated endpoint values and derivatives, determines the smallest tension factor for which the corresponding Hermite interpolatory tension spline satisfies a specified bound on function values in the interval.

SIG1   Function which, given a pair of abscissae, along with associated endpoint data, determines the smallest tension factor for which the corresponding Hermite interpolatory tension spline satisfies a specified bound on first derivative values in the interval.

SIG2   Function which, given a pair of abscissae, along with associated endpoint data, determines the smallest tension factor for which the corresponding Hermite interpolatory tension spline preserves convexity (or concavity) of the data.

b) The following modules are of general utility.

INTRVL Function which, given an increasing sequence of abscissae, returns the index of an interval containing a specified point. INTRVL is called by the evaluation functions TSINTL, HVAL, HPVAL, and HPPVAL.

SNHCSH Subroutine called by several modules to compute accurate approximations to the modified hyperbolic functions which form a basis for exponential tension splines.

STORE   Function used by SIGBP, SIGS, SIG0, SIG1, and SIG2 in computing the machine precision. STORE forces a value to be stored in main memory so that the precision of floating point numbers in memory locations rather than registers is computed.

c) The remaining modules are listed below.

B2TRI   Subroutine called by SMCRV to solve the symmetric positive-definite block tridiagonal linear system associated with the gradient of the quadratic

```
                functional whose minimum corresponds to a smooth-
                ing curve with nonperiodic end conditions.

        B2TRIP  Subroutine similar to B2TRI for periodic end
                conditions.

        ENDSLP  Function which returns the derivative at X1 of a
                tension spline h(x) which interpolates three
                specified data points and has third derivative
                equal to zero at X1.   ENDSLP is called by YPC2
                when this choice of end conditions is selected
                by an input parameter.

        YPCOEF  Subroutine called by SMCRV, YPC2, and YPC2P to com-
                pute coefficients defining the linear system.
```

IV. REFERENCE

For the theoretical background, consult the following:

RENKA, R. J. Interpolatory tension splines with automatic selection of tension factors. SIAM J. Sci. Stat. Comput. 8 (1987), pp. 393-415.

SUBROUTINE TSPSI

```
C***********************************************************
C                                              From TSPACK
C                                          Robert J. Renka
C                                Dept. of Computer Science
C                                     Univ. of North Texas
C                                        renka@cs.unt.edu
C                                                 07/08/92
C   This subroutine computes a set of parameter values which
C define a Hermite interpolatory tension spline H(x).  The
C parameters consist of knot derivative values YP computed
C by Subroutine YPC1, and tension factors SIGMA computed by
C Subroutine SIGS.  Alternative methods for computing SIGMA
C are provided by Subroutine TSPBI and Functions SIG0, SIG1,
C and SIG2.
C   Refer to Subroutine TSPSS for a means of computing
C parameters which define a smoothing curve rather than an
C interpolatory curve.
C   The tension spline may be evaluated by Subroutine TSVAL1
C or Functions HVAL (values), HPVAL (first derivatives),
C HPPVAL (second derivatives), and TSINTL (integrals).
C On input:
C       N = Number of data points.  N .GE. 2 and N .GE. 3 if
C           PER = TRUE.
C       X = Array of length N containing a strictly in-
C           creasing sequence of abscissae:  X(I) < X(I+1)
C           for I = 1,...,N-1.
C       Y = Array of length N containing data values asso-
C           ciated with the abscissae.  H(X(I)) = Y(I) for
C           I = 1,...,N.
C       YP = Array of length N containing first derivatives
C           of H at the abscissae.  Refer to Subroutine YPC1
C On output:
C       YP = Array containing derivatives of H at the
C            abscissae.  YP is not altered if -4 < IER < 0,
C            and YP is only partially defined if IER = -4.
C       SIGMA = Array containing tension factors.  SIGMA(I)
C               is associated with interval (X(I),X(I+1))
C               for I = 1,...,N-1.  SIGMA is not altered if
C               -4 < IER < 0 (unless IENDC is invalid), and
```

```
C               SIGMA is constant (not optimal) if IER = -4
C               or IENDC (if used) is invalid.
C       IER = Error indicator or iteration count:
C             IER = IC .GE. 0 if no errors were encountered
C                      and IC calls to SIGS and IC+1 calls
C                      to YPC1, YPC1P, YPC2 or YPC2P were
C                      employed.  (IC = 0 if NCD = 1).
C             IER = -1 if N, NCD, or IENDC is outside its
C                      valid range.
C             IER = -2 if LWK is too small.
C             IER = -3 if UNIFRM = TRUE and SIGMA(1) is out-
C                      side its valid range.
C             IER = -4 if the abscissae X are not strictly
C                      increasing.
C Modules required by TSPSI:  SIGS, SNHCSH, STORE, YPC1,
C Intrinsic functions called by TSPSI:  ABS, MAX
C************************************************************
```

SUBROUTINE SIGS

```
C************************************************************
C                                                 From TSPACK
C                                             Robert J. Renka
C                                   Dept. of Computer Science
C                                        Univ. of North Texas
C                                           renka@cs.unt.edu
C                                                    11/17/96
C   Given a set of abscissae X with associated data values Y
C and derivatives YP, this subroutine determines the small-
C est (nonnegative) tension factors SIGMA such that the Her-
C mite interpolatory tension spline H(x) preserves local
C shape properties of the data.  In an interval (X1,X2) with
C data values Y1,Y2 and derivatives YP1,YP2, the properties
C of the data are
C       Monotonicity:  S, YP1, and YP2 are nonnegative or
C                        nonpositive,
C  and
C       Convexity:     YP1 .LE. S .LE. YP2  or  YP1 .GE. S
C                        .GE. YP2,
C where S = (Y2-Y1)/(X2-X1).  The corresponding properties
C of H are constant sign of the first and second deriva-
C tives, respectively.  Note that, unless YP1 = S = YP2, in-
C finite tension is required (and H is linear on the inter-
C val) if S = 0 in the case of monotonicity, or if YP1 = S
C or YP2 = S in the case of convexity.
C   SIGS may be used in conjunction with Subroutine YPC2
C (or YPC2P) in order to produce a C-2 interpolant which
C preserves the shape properties of the data.  This is
C achieved by calling YPC2 with SIGMA initialized to the
C zero vector, and then alternating calls to SIGS with
C calls to YPC2 until the change in SIGMA is small (refer to
C the parameter descriptions for SIGMA, DSMAX and IER), or
C the maximum relative change in YP is bounded by a toler-
C ance (a reasonable value is .01).  A similar procedure may
C be used to produce a C-2 shape-preserving smoothing curve
C (Subroutine SMCRV).
C   Refer to Subroutine SIGBI for a means of selecting mini-
C mum tension factors to satisfy more general constraints.
C On input:
C       N = Number of data points.  N .GE. 2.
C       X = Array of length N containing a strictly in-
C           creasing sequence of abscissae:  X(I) < X(I+1)
C           for I = 1,...,N-1.
C       Y = Array of length N containing data values (or
C           function values computed by SMCRV) associated
C           with the abscissae.  H(X(I)) = Y(I) for I =
C           1,...,N.
C       YP = Array of length N containing first derivatives
C            of H at the abscissae.  Refer to Subroutines
C            YPC1, YPC1P, YPC2, YPC2P, and SMCRV.
```

```
C The above parameters are not altered by this routine.
C On output:
C       SIGMA = Array containing tension factors for which
C               H(x) preserves the properties of the data,
C               with the restriction that SIGMA(I) .LE. 85
C               for all I (unless the input value is larger).
C               The factors are as small as possible (within
C               the tolerance), but not less than their
C               input values.  If infinite tension is re-
C               quired in interval (X(I),X(I+1)), then
C               SIGMA(I) = 85 (and H is an approximation to
C               the linear interpolant on the interval),
C               and if neither property is satisfied by the
C               data, then SIGMA(I) = 0 (unless the input
C               value is positive), and thus H is cubic in
C               the interval.
C       IER = Error indicator and information flag:
C             IER = I if no errors were encountered and I
C                     components of SIGMA were altered from
C                     their input values for 0 .LE. I .LE.
C                     N-1.
C             IER = -1 if N < 2.  SIGMA is not altered in
C                      this case.
C             IER = -I if X(I) .LE. X(I-1) for some I in the
C                      range 2,...,N.  SIGMA(J-1) is unal-
C                      tered for J = I,...,N in this case.
C Modules required by SIGS:  SNHCSH, STORE
C Intrinsic functions called by SIGS:  ABS, EXP, MAX, MIN,
C                                         SIGN, SQRT
C
C***********************************************************
```

SUBROUTINE SNHCSH

```
C***********************************************************
C                                               From TSPACK
C                                           Robert J. Renka
C                                 Dept. of Computer Science
C                                      Univ. of North Texas
C                                          renka@cs.unt.edu
C                                                  11/20/96
C   This subroutine computes approximations to the modified
C hyperbolic functions defined below with relative error
C bounded by 3.4E-20 for a floating point number system with
C sufficient precision.
C   Note that the 21-digit constants in the data statements
C below may not be acceptable to all compilers.
C On input:
C       X = Point at which the functions are to be
C           evaluated.
C X is not altered by this routine.
C On output:
C       SINHM = sinh(X) - X.
C       COSHM = cosh(X) - 1.
C       COSHMM = cosh(X) - 1 - X*X/2.
C Modules required by SNHCSH:  None
C Intrinsic functions called by SNHCSH:  ABS, EXP
C**********************************************************
```

SUBROUTINE YPC1

```
C***********************************************************
C                                               From TSPACK
C                                           Robert J. Renka
C                                 Dept. of Computer Science
C                                      Univ. of North Texas
C                                          renka@cs.unt.edu
C                                                  06/10/92
C   This subroutine employs a three-point quadratic interpo-
C lation method to compute local derivative estimates YP
```

```
C associated with a set of data points.  The interpolation
C formula is the monotonicity-constrained parabolic method
C described in the reference cited below.  A Hermite int-
C erpolant of the data values and derivative estimates pre-
C serves monotonicity of the data.  Linear interpolation is
C used if N = 2.  The method is invariant under a linear
C scaling of the coordinates but is not additive.
C On input:
C       N = Number of data points.  N .GE. 2.
C       X = Array of length N containing a strictly in-
C           creasing sequence of abscissae:  X(I) < X(I+1)
C           for I = 1,...,N-1.
C       Y = Array of length N containing data values asso-
C           ciated with the abscissae.
C Input parameters are not altered by this routine.
C On output:
C       YP = Array of length N containing estimated deriv-
C            atives at the abscissae unless IER .NE. 0.
C            YP is not altered if IER = 1, and is only par-
C            tially defined if IER > 1.
C       IER = Error indicator:
C             IER = 0 if no errors were encountered.
C             IER = 1 if N < 2.
C             IER = I if X(I) .LE. X(I-1) for some I in the
C                     range 2,...,N.
C Reference:  J. M. Hyman, "Accurate Monotonicity-preserving
C               Cubic Interpolation",  LA-8796-MS, Los
C               Alamos National Lab, Feb. 1982.
C Modules required by YPC1:  None
C Intrinsic functions called by YPC1:  ABS, MAX, MIN, SIGN
C***********************************************************
```

FUNCTION HVAL

```
C***********************************************************
C                                               From TSPACK
C                                           Robert J. Renka
C                                 Dept. of Computer Science
C                                      Univ. of North Texas
C                                          renka@cs.unt.edu
C                                                  11/17/96
C   This function evaluates a Hermite interpolatory tension
C spline H at a point T.  Note that a large value of SIGMA
C may cause underflow.  The result is assumed to be zero.
C   Given arrays X, Y, YP, and SIGMA of length NN, if T is
C known to lie in the interval (X(I),X(J)) for some I < J,
C a gain in efficiency can be achieved by calling this
C function with N = J+1-I (rather than NN) and the I-th
C components of the arrays (rather than the first) as par-
C ameters.
C On input:
C       T = Point at which H is to be evaluated.  Extrapo-
C           lation is performed if T < X(1) or T > X(N).
C       N = Number of data points.  N .GE. 2.
C       X = Array of length N containing the abscissae.
C           These must be in strictly increasing order:
C           X(I) < X(I+1) for I = 1,...,N-1.
C       Y = Array of length N containing data values.
C           H(X(I)) = Y(I) for I = 1,...,N.
C       YP = Array of length N containing first deriva-
C            tives.  HP(X(I)) = YP(I) for I = 1,...,N, where
C            HP denotes the derivative of H.
C       SIGMA = Array of length N-1 containing tension fac-
C               tors whose absolute values determine the
C               balance between cubic and linear in each
C               interval.  SIGMA(I) is associated with int-
C               erval (I,I+1) for I = 1,...,N-1.
C Input parameters are not altered by this function.
C On output:
C       IER = Error indicator:
```

```
C             IER = 0  if no errors were encountered and
C                      X(1) .LE. T .LE. X(N).
C             IER = 1  if no errors were encountered and
C                      extrapolation was necessary.
C             IER = -1 if N < 2.
C             IER = -2 if the abscissae are not in strictly
C                      increasing order.  (This error will
C                      not necessarily be detected.)
C       HVAL = Function value H(T), or zero if IER < 0.
C Modules required by HVAL:  INTRVL, SNHCSH
C Intrinsic functions called by HVAL:  ABS, EXP
C***********************************************************
```

FUNCTION HPVAL

```
C***********************************************************
C                                                From TSPACK
C                                            Robert J. Renka
C                                  Dept. of Computer Science
C                                       Univ. of North Texas
C                                           renka@cs.unt.edu
C                                                   11/17/96
C   This function evaluates the first derivative HP of a
C Hermite interpolatory tension spline H at a point T.
C On input:
C       T = Point at which HP is to be evaluated.  Extrapo-
C           lation is performed if T < X(1) or T > X(N).
C       N = Number of data points.  N .GE. 2.
C       X = Array of length N containing the abscissae.
C           These must be in strictly increasing order:
C           X(I) < X(I+1) for I = 1,...,N-1.
C       Y = Array of length N containing data values.
C           H(X(I)) = Y(I) for I = 1,...,N.
C       YP = Array of length N containing first deriva-
C            tives.  HP(X(I)) = YP(I) for I = 1,...,N.
C       SIGMA = Array of length N-1 containing tension fac-
C               tors whose absolute values determine the
C               balance between cubic and linear in each
C               interval.  SIGMA(I) is associated with int-
C               erval (I,I+1) for I = 1,...,N-1.
C Input parameters are not altered by this function.
C On output:
C       IER = Error indicator:
C             IER = 0  if no errors were encountered and
C                      X(1) .LE. T .LE. X(N).
C             IER = 1  if no errors were encountered and
C                      extrapolation was necessary.
C             IER = -1 if N < 2.
C             IER = -2 if the abscissae are not in strictly
C                      increasing order.  (This error will
C                      not necessarily be detected.)
C       HPVAL = Derivative value HP(T), or zero if IER < 0.
C Modules required by HPVAL:  INTRVL, SNHCSH
C Intrinsic functions called by HPVAL:  ABS, EXP
C***********************************************************
```

FUNCTION STORE

```
C***********************************************************
C                                                From TSPACK
C                                            Robert J. Renka
C                                  Dept. of Computer Science
C                                       Univ. of North Texas
C                                           renka@cs.unt.edu
C                                                   06/10/92
C   This function forces its argument X to be stored in a
C memory location, thus providing a means of determining
C floating point number characteristics (such as the machine
C precision) when it is necessary to avoid computation in
C high precision registers.
C On input:
```

```
C       X = Value to be stored.
C X is not altered by this function.
C On output:
C       STORE = Value of X after it has been stored and
C               possibly truncated or rounded to the single
C               precision word length.
C Modules required by STORE:  None
C************************************************************
```

INTEGER FUNCTION INTRVL

```
C************************************************************
C                                                From TSPACK
C                                            Robert J. Renka
C                                  Dept. of Computer Science
C                                       Univ. of North Texas
C                                           renka@cs.unt.edu
C                                                   06/10/92
C   This function returns the index of the left end of an
C interval (defined by an increasing sequence X) which
C contains the value T.  The method consists of first test-
C ing the interval returned by a previous call, if any, and
C then using a binary search if necessary.
C On input:
C       T = Point to be located.
C       N = Length of X.  N .GE. 2.
C       X = Array of length N assumed (without a test) to
C           contain a strictly increasing sequence of
C           values.
C Input parameters are not altered by this function.
C On output:
C       INTRVL = Index I defined as follows:
C                  I = 1    if  T .LT. X(2) or N .LE. 2,
C                  I = N-1  if  T .GE. X(N-1), and
C                  X(I) .LE. T .LT. X(I+1) otherwise.
C Modules required by INTRVL:  None
C************************************************************
```

XIX. Vertical Turbulence Module **(ver_turb_module.f90, VTURB_MOD)**

Overview: Hydrodynamic models do not simulate turbulent motion at scales smaller than the grid resolution of the model (e.g., 1 km). In particle-tracking models, however, particles are moved in millimeter to centimeter steps—much less than the hydrodynamic model grid scale. It is necessary to add a random component to particle motion in order to reproduce turbulent diffusion that occurs at the scale of particle motion (Visser 1997, Brickman and Smith 2001). Without turbulent particle motion, particle-tracking models do not reproduce the hydrodynamic model predictions for the spread of tracer concentrations (North et al. 2006a). In LTRANS, a random displacement model (Visser 1997) is implemented within the larval transport model to simulate sub-grid scale turbulent particle motion in the vertical (z) direction. Because there are vertical gradients in diffusivity, a random displacement model is used instead of a simple random walk model (see page 97) to avoid artificial aggregation of particles in regions of low diffusivity (Visser 1997, Brickman and Smith 2001, North et al. 2006a).

This Vertical Turbulence Module is based on work presented in North et al. (2006a) in which the random displacement model was embedded in an on-line particle tracking model. The model was tested to determine if it could maintain the Well Mixed Condition, "an initially uniform concentration of [neutrally buoyant] particles uniform for all time" (Brickman and Smith 2002). Here is an excerpt from the abstract of the North et al. (2006a) paper:

> "A new interpolation scheme, the 'water column profile' scheme, was developed and used to implement a random displacement model (Visser 1997) for turbulent particle motions. A new interpolation scheme was necessary because linear interpolation schemes caused artificial aggregation of particles where abrupt changes in vertical diffusivity occurred. The new 'water column profile' scheme was used to fit a continuous function (a tension spline) to a smoothed profile of vertical diffusivities at the x-y particle location. The new implementation scheme was checked for artifacts and compared with a standard random walk model using 1) Well Mixed Condition tests, and 2) dye-release experiments. The Well Mixed Condition tests confirmed that the use of the 'water column profile' interpolation scheme for implementing the random displacement model significantly reduced numerical artifacts. In dye-release experiments, high concentrations of Eulerian tracer and Lagrangian particles were released at the same location up-estuary of the salt front and tracked for 4 days. After small differences in initial dispersal rates, tracer and particle distributions remained highly correlated (r = 0.84 to 0.99) when a random displacement model was implemented in the particle-tracking model. In contrast, correlation coefficients were substantially lower (r = 0.07 to 0.58) when a random walk model was implemented. In general, model performance tests indicated that the 'water column interpolation' scheme was an effective technique for implementing a random displacement model within a hydrodynamic model, and both could be used to accurately simulate diffusion in a highly baroclinic frontal region."

Public Procedures: The following are the public subroutines and functions contained within the Vertical Turbulence Module: **Subroutine VTurb**.

A. Subroutine VTurb

Overview: This subroutine calculates the sub-grid scale particle turbulence in the z- direction.

Input Variables: The subroutine VTurb has nine input variables. The input variables are the vertical location of the particle (**P_zc**), the depth (**P_depth**) and surface height (**P_zetac**) at the particle location, the external (**ex**) and internal (**ix**) time variables, the current iteration of the external time loop (**p**), and the depths of s-levels (**Pwc_wzb**, **Pwc_wzb**, **Pwc_wzb**).

Output Variables: The module returns the displacement (m) in the z-direction during one internal time step in the variable **Turbv**.

Module parameters used: This subroutine uses the parameters **ws** and **idt** from the Parameter Module, which contain the number of w s-levels and the duration of the internal time step in seconds respectively.

Module procedures used: This subroutine uses the function **getInterp** from the Hydrodynamic Module, the function **norm** from the Norm Module, the functions **linint** and **polintd** from the Interpolation Module, and the procedures **TSPSI**, **HVAL**, and **HPVAL** from TSPACK in the Tension Spline Module.

Private variables used: The subroutine uses no private variables.

Initialization: This module must be 'turned on' in LTRANS.data by setting the parameter VTurbOn = .TRUE.

Numerical Method: The random displacement model takes the form of:

$$z_{n+1} = z_n + K_v'\delta t + R\left[2r^{-1}K_v\delta t\right]^{1/2}$$

where z_n = initial particle location, K_v = vertical diffusivity evaluated at ($z_n + 0.5K_v'\delta t$), δt = time step of the random displacement model, $K_v' = \partial K_v/\partial z$ evaluated at z_n, and R is a random number generator with mean = 0 and standard deviation r = 1. Note that the term with the gradient in vertical diffusivity (K_v') gives the particle a kick away from regions of low diffusivity. This prevents artificial aggregation of particles in these regions. The turbulent particle motion sub-model uses the same approach for determining K_v and K_v' at the particle location as that used in the advection model, except that 1) a profile of the entire water column is created, 2) a smoothing algorithm is applied to the water column profile of K_v to prevent artificial aggregation of particles in regions of sharp gradients in diffusivity (North et al. 2006a), and 3) a 4th order Runge-Kutta is applied in time but not in space due to computational constraints.

To prepare the smoothed profile of vertical diffusivity, first the number of vertical points is proliferated, and a value of K_v is assigned to each point by linear interpolation (the number of points is set as 4 times the number of s-levels). The profile is then smoothed with an 8-point moving average. An 8-point moving average was found to cause the least number of failures of the Well Mixed Condition test (North et al. 2006a). After smoothing, the surface and bottom values of the profile are restored to original values in the hydrodynamic model output (which are likely the background vertical diffusivity of the hydrodynamic model). Finally, a tension spline is fit to the profile and used to calculate K_v and K_v' at the particle location using TSPACK (Fig. 11) in the Tension Spline Module. The time step of the random displacement model (e.g., 2 s) is set to much shorter than the internal time step (e.g., 120 s). This avoids unrealistically large jumps in particle motion that could occur if times steps are large and gradients in diffusivity are steep.

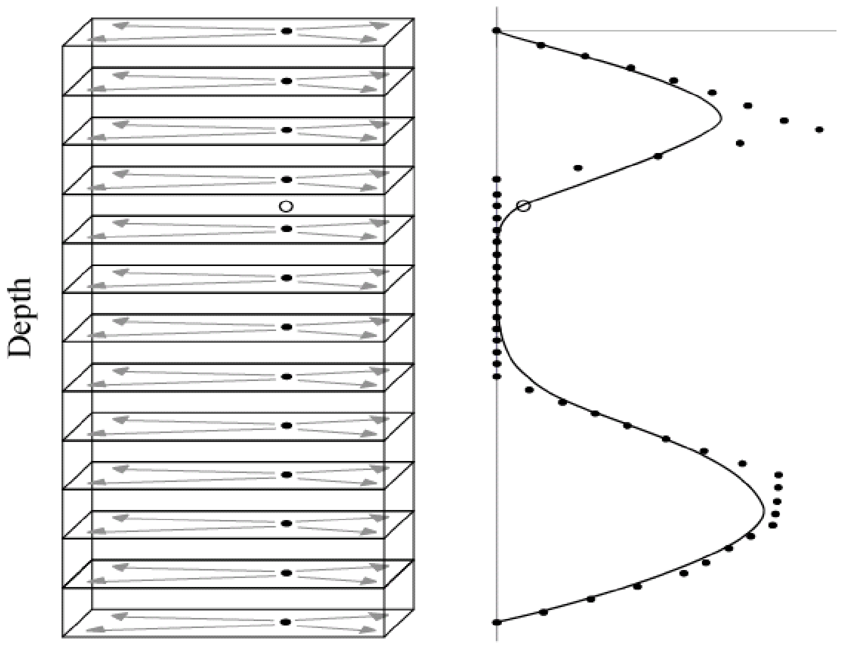

Fig. 11. Schematic of interpolation scheme for vertical turbulence module. Left panel: Subset of model grid. Hydrodynamic model output was interpolated at each s-level to create a vertical profile (filled circles) at the x-y particle location. Right panel: Profile of vertical diffusivity. Data points were proliferated with linear interpolation (filled circles), smoothed with an 8-pt moving average, and fit with a tension spline (line) in order to estimate vertical diffusivity at the particle location (open circle). After Fig. 2 of North et al. 2006a.

It should be noted that the time step of both the particle-tracking model and the random displacement model likely influence the ability of the Vertical turbulence model to pass the Well Mixed Condition test (i.e., maintain the uniformity of an initially uniform concentration of particles over time). Moreover, the degree of stratification in the hydrodynamic model, and hence the magnitude of the gradient in vertical diffusivity, likely influences its ability to maintain the Well Mixed Condition. It is not known what the appropriate time step should be for a given degree of stratification. Further analyses are required.

Many of the variables used in this module refer to x and y coordinates but actually represent vertical (z) coordinates and horizontal diffusivities. This convention was chosen to match the input values of the tension spline interpolation package, in which z-coordinates are treated as x-values and diffusivities are treated as y-values.

Variable Definitions: The following variables are used in this section:

background – dp – background vertical diffusivity from ROMS
deltat – real - time step of random displacement model
DEV – real - the random deviate drawn from a normal distribution
ex – dp – x-values (from external time step) for polynomial interpolation in time (s)
ey – dp - y-values (from external time step) for polynomial interpolation in time
i – integer – iteration variable
idt – integer, parameter - internal time step of particle tracking model
IER – integer – error indicator or iteration count (for TSPACK)
ifitx – dp - vertical coordinates for the profile of vertical diffusivity at the particle location for use in random displacement model
ifitxb – dp - vertical coordinates for the profile of vertical diffusivity at the particle location from previous (‘back’) internal time step
ifitxc – dp - vertical coordinates for the profile of vertical diffusivity at the particle location from current (‘center’) internal time step
ifitxf – dp - vertical coordinates for the profile of vertical diffusivity at the particle location from next (‘forward’) internal time step
ifity – dp - profile of vertical diffusivity at particle location for use in random displacement model
ifityb – dp - profile of vertical diffusivity at the particle location from previous (‘back’) internal time step
ifityc – dp - profile of vertical diffusivity at the particle location from current (‘center’) internal time step
ifityf – dp - profile of vertical diffusivity at the particle location from next (‘forward’) internal time step
interceptb – dp – intercept used for linear interpolation of vertical diffusivity (from the previous (‘back’) external time step) to proliferated points
interceptc – dp - intercept used for linear interpolation of vertical diffusivity (from the current (‘center) external time step) to proliferated points
interceptf – dp - intercept used for linear interpolation of vertical diffusivity (from the next (‘forward’) external time step) to proliferated points
ix – dp – x-values (from internal time step) for polynomial interpolation in time (s)
j – integer – iteration variable
jlo – integer – contains the nearest s-level below the proliferated points, to be used when linearly interpolating values to the proliferated points
k – integer – iteration variable
KH3rdc – dp - vertical diffusivity (K_v) evaluated at ($z_n + 0.5K'_v\delta t$)
Kprimec – dp - gradient in vertical diffusivity (i.e., slope) at particle location
KprimeZc – dp - second term in RDM equation ($K'_v\delta t$)
loop – integer - number of iterations of the random displacement model loop
movexb – dp – vertical coordinates for smoothed profile at the particle location from previous (‘back’) external time step
movexc – dp – vertical coordinates for smoothed profile at the particle location from current (‘center’) external time step
movexf – dp – vertical coordinates for smoothed profile at the particle location from future (‘forward’) external time step

moveyb – dp – smoothed profile of vertical diffusivity at the particle location from previous (‘back’) external time step
moveyc – dp - smoothed profile of vertical diffusivity at the particle location from current (‘center’) external time step
moveyf – dp - smoothed profile of vertical diffusivity at the particle location from future (‘forward’) external time step
newxb – dp – vertical coordinates for proliferated diffusivity values from previous (‘back’) external time step
newxc – dp - vertical coordinates for proliferated diffusivity values from current (‘center’) external time step
newxf – dp - vertical coordinates for proliferated diffusivity values from future (‘forward’) external time step
newyb – dp – diffusivity values from previous (‘back’) external time step
newyc – dp – diffusivity values from current (‘center’) external time step
newyf – dp – diffusivity values from future (‘forward’) external time step
newZc – dp - new particle depth (z-coordinate) after each time step of the random displacement model
p – integer - external time step do loop iteration variable
p2 - integer, parameter – number of vertical coordinates to proliferate to
P_depth – dp - water depth at particle location (m)
P_zc – dp - particle depth (m)
P_zetac – dp - sea surface height at particle location (m)
ParZc – dp - particle depth (m)
Pwc_KHb – dp - vertical coordinates for diffusivity values from ROMS model from previous (‘back’) external time step
Pwc_KHc – dp - vertical coordinates for diffusivity values from ROMS model from current (‘center’) external time step
Pwc_KHf – dp – vertical coordinates for diffusivity values from ROMS model from future (‘forward’) external time step
Pwc_wzb - dp – z-coordinates of each w s-level at particle location at back time
Pwc_wzc - dp – z-coordinates of each w s-level at particle location at center time
Pwc_wzf - dp – z-coordinates of each w s-level at particle location at forward time
r – real - the standard deviation of the random deviate
SigErr - integer – indicates error that TSPACK failed to converge
SIGMAKc – dp – tension factors computed by TSPACK
slopekb – dp – slope used for linear interpolation of vertical diffusivity (from the previous (‘back’) external time step) to proliferated points
slopekc – dp - slope used for linear interpolation of vertical diffusivity (from the current (‘center) external time step) to proliferated points
slopekf – dp - slope used for linear interpolation of vertical diffusivity (from the next (‘forward’) external time step) to proliferated points
slopem – dp – slope at particle location calculated by linear interpolation
thisyc – dp – a dummy variable used to fill the call line of linint
TurbV – dp - displacement in z-direction due to vertical turbulence during internal time step
ws – integer, parameter – number of w s-levels
YPKc – dp – derivatives at nodes computed by, and used within, TSPACK

Z3rdc – dp – vertical position at which to compute diffusivity for use in the random displacement model (Z3rdc = $z_n + 0.5K_v'\delta t$)

XX. Literature Cited

Brickman, D., and P. C. Smith, 2002. Lagrangian stochastic modeling in coastal oceanography. Journal of Atmospheric and Ocean Technology 19: 83–99.

Dippner, J. W. 2004. Mathematical modelling of the transport of pollution in water, in *Mathematical Models*, edited by J. A. Filar and J. B. Krawczyk in *Encyclopedia of Life Support Systems* (EOLSS), UNESCO, Eolss Publishers, Oxford,UK, [http://www.eolss.net].

Hunter, J., P. Craig, and H. Phillips. 1993. On the use of random-walk models with spatially-variable diffusivity. Journal of Computational Physics 106:366-376.

Kirk, J. T. O. 1994. Light and photosynthesis in aquatic ecosystems, 2nd edition. Cambridge University Press. Cambridge, UK. 509 p.

Li, M., L. Zhong, and W. C. Boicourt. 2005. Simulations of Chesapeake Bay estuary: Sensitivity to turbulence mixing parameterizations and comparison with observations, Journal of Geophysical Research, 110, C12004, doi:10.1029/2004JC002585.

Li, M., L. Zhong, W. C. Boicourt, S. Zhang and D.-L. Zhang. 2006. Hurricane-induced stormsurges, currents and destratification in a semi-enclosed bay. Geophysical Research Letters 33: L02604, doi:10.1029/2005GL024992.

Meeus, J. 1998. Astronomical algorithms, 2nd edition. Willmann-Bell Inc. Richmond, VA. 477 p.

North, E. W., E. E. Adams, S. Schlag, C. R. Sherwood, R. He, S. Socolofsky. 2011. Simulating oil droplet dispersal from the Deepwater Horizon spill with a Lagrangian approach. AGU Book Series: Monitoring and Modeling the Deepwater Horizon Oil Spill: A Record Breaking Enterprise.

North, E. W., R. R. Hood, S.-Y. Chao, and L. P. Sanford. 2005. The influence of episodic events on transport of striped bass eggs to an estuarine nursery area. Estuaries 28(1): 106-121.

North, E. W., R. R. Hood, S.-Y. Chao, and L. P. Sanford. 2006a. Using a random displacement model to simulate turbulent particle motion in a baroclinic frontal zone: a new implementation scheme and model performance tests. Journal of Marine Systems 60: 365-380.

North, E. W., Z. Schlag, R. R. Hood, L. Zhong, M. Li, and T. Gross. 2006b. Modeling dispersal of *Crassostrea ariakensis* oyster larvae in Chesapeake Bay. Final Report to Maryland Department of Natural Resources, July 31, 2006. 55 p.

North, E. W., Z. Schlag, R. R. Hood, M. Li, L. Zhong, T. Gross, and V. S. Kennedy. 2008. Vertical swimming behavior influences the dispersal of simulated oyster larvae in a coupled particle-tracking and hydrodynamic model of Chesapeake Bay. Marine Ecology Progress Series 359: 99-115.

Schlag, Z. R., E. W. North, and K. A. Smith. 2008. Larval TRANSport Lagrangian model (LTRANS) User's Guide. University of Maryland Center for Environmental Science, Horn Point Laboratory. Cambridge, MD. 146 pp.

Song, Y., and D. B. Haidvogel. 1994. A semi-implicit ocean circulation model using a generalized topography-following coordinate system, Journal of Computational Physics 115 (1): 228-244.

Visser, A.W., 1997. Using random walk models to simulate the vertical distribution of particles in a turbulent water column. Marine Ecology Progress Series 158: 275–281.

Zhong, L. and M. Li. 2006. Tidal energy fluxes and dissipation in the Chesapeake Bay. Continental Shelf Research 26: 752-770.

XI. Appendix: List of Modules, Functions and Subroutines

Overview: The following is a list of all the modules used by LTRANS as well as the subroutines and functions contained within them.

<u>behavior_module.f90</u> (BEHAVIOR_MOD)

Subroutines within the module:

- initBehave
  - uses parameters Behavior, deadage, numpar, pediage, settlementon, Sgradient, swimfast, swimslow, swimstart from PARAM_MOD
  - uses subroutine initSettlement from SETTLEMENT_MOD
- updateStatus
  - uses parameters mortality, settlementon from PARAM_MOD
  - uses function isSettled from SETTLEMENT_MOD
- behave
  - uses parameters daylength, diameterAgeRate, dissolve, dt, Em, Hswimspeed, idt, interfacial, Kd, minimumDiameter, oilAging, pi, Sgradient, sink, Swimdepth, swimfast, swimstart, us, thresh, twiend, twistart from PARAM_MOD
  - uses function WCTS_ITPI from HYDRO_MOD
  - uses function genrand_real1 from RANDOM_MOD
- die
- setOut
- finBehave
  - uses parameter settlementon from PARAM_MOD
  - uses subroutine finSettlement from SETTLEMENT_MOD

Functions within the module:

- getStatus
  - uses parameters OpenOceanBoundary, settlementon from PARAM_MOD
  - uses function isSettled from SETTLEMENT_MOD
- isDead
- isOut

<u>boundary_module.f90</u> (BOUNDARY_MOD)

Subroutines within the module:

- add
- createBounds
  - uses parameters BoundaryBLNs, ui, uj, vi, vj from PARAM_MOD
  - uses subroutines getMask_Rho, getUVxy from HYDRO_MOD
- getNext
  - uses parameters uj, vi from PARAM_MOD
- ibounds
  - uses function inpoly from PIP_MOD
- intersect_reflect

mbounds
uses function inpoly from PIP_MOD
output_llBounds
uses parameters ui, uj, vi, vj from PARAM_MOD
uses interfaces x2lon, y2lat from CONVERT_MOD
uses subroutine getUVxy from HYDRO_MOD
output_xyBounds
uses parameters ui, uj, vi, vj from PARAM_MOD
uses subroutine getUVxy from HYDRO_MOD
Functions within the module:
isBndSet

conversion_module.f90 (CONVERT_MOD)
uses parameters Earth_Radius, latmin, lonmin, PI, SphericalProjection from PARAM_MOD
Subroutines within the module:
(None)
Functions within the module:
dlat2y
dlon2x
dx2lon
dy2lat
rlat2y
rlon2x
rx2lon
ry2lat

gridcell_module.f90 (GRIDCELL_MOD)
Subroutines within the module:
gridcell
Functions within the module:
(None)

hor_turb_module.f90 (HTURB_MOD)
Subroutines within the module:
HTurb
uses parameters ConstantHTurb, idt from PARAM_MOD
uses function norm from NORM_MOD
Functions within the module:
(None)

hydrodynamic_module.f90 (HYDRO_MOD)

uses parameters numpar, ui, vi, uj, vj, us, ws, tdim, rho_nodes, u_nodes, v_nodes,
max_rho_elements, max_u_elements, max_v_elements, rho_elements,
u_elements, v_elements from PARAM_MOD

Subroutines within the module:

- createNetCDF
  - uses parameters ExeDir, Institution, NCOutFile, NCtime, numpar, oilRun, OutDir, outpath, outpathGiven, RunBy, RunName, SaltTempOn, StartedOn, SVN_Version, TrackCollisions from PARAM_MOD
  - uses netcdf90
- finHydro
- getMask_Rho
- getR_ele
- getUVxy
- initGrid
  - uses parameters filenum, max_rho_elements, max_u_elements, max_v_elements, NCgridfile, numdigits, numpar, prefix, rho_elements, rho_nodes, suffix, u_elements, u_nodes, ui, uj, us, v_elements, v_nodes, vi, vj, ws from PARAM_MOD
  - uses interfaces lat2y, lon2x from CONVERT_MOD
  - uses netcdf90
- initHydro
  - uses parameters constAks, constDens, constSalt, constTemp, constU, constV, constW, constZeta, filenum, numdigits, numpar, prefix, readAks, readDens, readSalt, readTemp, readU, readV, readW, readZeta, rho_nodes, suffix, u_nodes, ui, uj, us, v_nodes, vi, vj, ws from PARAM_MOD
  - uses netcdf90
- initNetCDF
- setEle
  - uses parameters rho_elements, u_elements, v_elements from PARAM_MOD
  - uses subroutine gridcell from GRIDCELL_MOD
- setEle_all
  - uses parameters rho_elements, u_elements, v_elements from PARAM_MOD
  - uses subroutine gridcell from GRIDCELL_MOD
- setijruv
  - uses parameters ijbuff, numpar, ui, uj, vi, vj from PARAM_MOD
- setInterp
- updateHydro
  - uses parameters constAks, constDens, constSalt, constTemp, constU, constV, constW, constZeta, filenum, numdigits, prefix, readAks, readDens, readSalt, readTemp, readU, readV, readW, readZeta, rho_nodes, startfile, suffix, tdim, u_nodes, ui, uj, us, v_nodes, vi, vj, ws from PARAM_MOD
  - uses netcdf90
- writeNetCDF
  - uses parameters NCOutFile, NCtime, numpar, oilRun, outpath, outpathGiven, SaltTempOn, TrackCollisions from PARAM_MOD

uses netcdf90

Functions within the module:

getInterp

getP_r_element

getSlevel

uses parameters hc, Vtransform from PARAM_MOD

getWlevel

uses parameters hc, Vtransform from PARAM_MOD

interp

WCTS_ITPI

uses subroutine TSPSI, function HVAL from TENSION_MOD

uses subroutine linint, function polintd from INT_MOD

interpolation_module.f90 (INT_MOD)

Subroutines within the module:

linint

Functions within the module:

polintd

LTRANS.f90 (main program)

Subroutines within the main LTRANS program:

ini_LTRANS

uses parameters Behavior, days, dt, ErrorFlag, getParams, idt, numpar, oilRun, parfile, SaltTempOn, seed, settlementon, TrackCollisions, WriteHeaders, WriteModelTiming, writeNC from PARAM_MOD

uses subroutines die, initBehave, setOut from BEHAVIOR_MOD

uses subroutines createBounds, ibounds, mbounds from BOUNDARY_MOD

uses interfaces lat2y, lon2x from CONVERT_MOD

uses subroutine init_genrand from RANDOM_MOD

uses subroutines createNetCDF, initGrid, initHydro, initNetCDF, setEle_all, writeNetCDF from HYDRO_MOD

fin_LTRANS

uses parameters numpar, outpath, outpathGiven, settlementon, writeCSV from PARAM_MOD

uses subroutine finBehave and function getStatus from BEHAVIOR_MOD

uses interfaces x2lon, y2lat from CONVERT_MOD

uses subroutine finHydro from HYDRO_MOD

find_currents

uses parameters us, ws, z0 from PARAM_MOD

uses functions interp, WCTS_ITPI from HYDRO_MOD

uses function polintd from INT_MOD

uses subroutine TSPSI, function HVAL from TENSION_MOD

printOutput

uses parameters numpar, oilRun, SaltTempOn, TrackCollisions,
WriteModelTiming from PARAM_MOD
uses interfaces x2lon, y2lat from CONVERT_MOD

run_LTRANS
uses parameters days,dt,idt,iPrint,WriteModelTiming from PARAM_MOD
uses subroutines updateHydro from HYDRO_MOD

update_particles
uses parameters Behavior, dissolve, ErrorFlag, HTurbOn, idt, mortality, numpar,
OpenOceanBoundary, SaltTempOn, settlementon, Swimdepth,
TrackCollisions, us, VTurbOn, WriteModelTiming, ws from
PARAM_MOD
uses subroutines behave, die, setOut, updateStatus and functions isDead, isOut
from BEHAVIOR_MOD
uses subroutines ibounds, intersect_reflect, mbounds from BOUNDARY_MOD
uses interfaces x2lon, y2lat from CONVERT_MOD
uses subroutine HTurb from HTURB_MOD
uses subroutines setEle, setInterp and functions getInterp, getSlevel, getWlevel,
WCTS_ITPI from HYDRO_MOD
uses function polintd from INT_MOD
uses subroutine testSettlement and function isSettled from SETTLEMENT_MOD
uses subroutine VTurb from VTURB_MOD

writeOutput
uses parameters numpar, oilRun, outpath, outpathGiven, SaltTempOn,
TrackCollisions, writeCSV, writeNC from PARAM_MOD
uses function getStatus from BEHAVIOR_MOD
uses interfaces x2lon, y2lat from CONVERT_MOD
uses subroutine writeNetCDF from HYDRO_MOD

writeModelInfo
uses PARAM_MOD

Functions within the main LTRANS program:
(None)

norm_module.f90 (NORM_MOD)

Subroutines within the module:
(None)

Functions within the module:
norm
uses parameter PI from PARAM_MOD
uses function genrand_real3 from RANDOM_MOD

random_module.f90 (RANDOM_MOD)

Subroutines used in the module:
init_genrand

Functions used in the module:
genrand_real1
genrand_real3

parameter_module.f90 (PARAM_MOD)

Subroutines within the module:
getParams
gridData
Functions within the module:
(None)

point_in_polygon_module.f90 (PIP_MOD)

Subroutines within the module:
(None)
Functions within the module:
inpoly

settlement_module.f90 (SETTLEMENT_MOD)

uses parameters **numpar**, **rho_elements**, **minholeid**, **maxholeid**, **minpolyid**, **maxpolyid**, **pedges**, **hedges** from PARAM_MOD
Subroutines within the module:
createPolySpecs
uses parameters **hedges, holesExist, maxpolyid, minpolyid, pedges, rho_elements** from PARAM_MOD
uses subroutine getR_ele from HYDRO_MOD
uses subroutine gridcell from GRIDCELL_MOD
uses function INPOLY from PIP_MOD
finSettlement
getHabitat
uses parameters **habitatfile**, **hedges**, **holefile**, **holesExist, pedges** from PARAM_MOD
uses interfaces lon2x, lat2y from CONVERT_MOD
hsettle
uses function INPOLY from PIP_MOD
initSettlement
uses parameters hedges, maxholeid, maxpolyid, minholeid, minpolyid, numpar, pedges, rho_elements from PARAM_MOD
psettle
uses function INPOLY from PIP_MOD
testSettlement
uses parameter **holesExist** from PARAM_MOD
uses function getP_r_element from HYDRO_MOD
Functions within the module:

isSettled

uses parameter settlementon from PARAM_MOD

<u>tension_module.for</u> (TENSION_MOD)

Subroutines within the module:

TSPSI
SIGS
SNHCSH
YPC1

Functions within the module:

HVAL
HPVAL
STORE
INTRVL

<u>ver_turb_module.f90</u> (VTURB_MOD)

Subroutines within the module:

VTurb

uses parameters **ws**, **idt** from PARAM_MOD
uses function getInterp from HYDRO_MOD
uses function **norm** from NORM_MOD
uses subroutine linint, function polintd from INT_MOD
uses subroutine TSPSI, functions HVAL, HPVAL from TENSION_MOD

Functions within the module:

(None)