SUNTANS User Guide

Stanford Unstructured Nonhydrostatic Terrain-following Adaptive Navier-Stokes Simulator

O. B. Fringer
Environmental Fluid Mechanics Laboratory,
Stanford University
Stanford, CA 94305-4020

Nov 16, 2011

Contents

1 Downloading and installing SUNTANS

1.1 Downloading and installing the latest source

To obtain a copy of SUNTANS, you can either download the latest tarball `suntans.tgz` from the suntans web page, or download the latest version from the cvs server using the directions in Section 1.2. If you have a working key, you should be able to download the latest source with

```
cvs checkout suntans
```

which will download the following three directories (not including the `CVS` directory):

- `suntans/main`: Contains the main source code and some examples in `main/examples`.
- `suntans/mfiles`: Some useful m-files for use with matlab.
- `suntans/papers`: Contains this user guide.

These same directories will be created if you download and unpack `suntans.tgz` with `tar xzvf suntans.tgz`.

If you have the GNU C compiler installed (`gcc`) you should be able to enter `suntans/main` and compile SUNTANS with

```
make
```

Otherwise, you will need to specify the correct C compiler with the variable `CC` in `suntans/main/Makefile`. With an appropriate C compiler, the example in `suntans/main/examples/iwaves` should also compile without any further installation (after changing `CC` in `suntans/main/examples/iwaves/Makefile` if necessary). To run the example, enter that directory and type

```
make test
```

This will run an internal waves example on a one-dimensional grid of equilateral triangles. To view the results, return to `suntans/main` and compile the graphical user interface with

```
make sunplot
```

This will create the `sunplot` executable, which can then be used to view the results of the internal waves example with

```
./sunplot --datadir=examples/iwaves/data
```

Note: This GUI requires the existence of the `Xlib` libraries and it is assumed that these are located in `/usr/X11R6`. Make sure the `XINC` and `XLIBDIR` variables are specified correctly in the `suntans/main/Makefile` if `sunplot` does not compile.

In order to run the other examples (as described in Section 6), the grid generation package Triangle [5] must be installed, and to run them in parallel, the message-passing interface (MPI) and the parallel graph partitioning libraries (ParMetis [4]) must be installed. Instructions for downloading and installing these packages are available from the individual websites for each package:

MPI	`http://www-unix.mcs.anl.gov/mpi/mpich/`
ParMetis	`http://www-users.cs.umn.edu/~karypis/metis/parmetis/`
Triangle	`http://www-2.cs.cmu.edu/~quake/triangle.html`

Note that you must compile the triangle libraries as object files by making them with `make trilibrary`.

1.1.1 Notes on Parmetis

Currently SUNTANS compiles and runs with both Parmetis-2 and Parmetis-3. If you are using Parmetis-3 you will need to edit suntans/main/partition.c to include the correct parmetis header file. For Parmetis-2, the header file is parmetis.h, while for version 3 it is parmetislib.h.

1.1.2 Notes on mpich

SUNTANS works with both mpich-1 and mpich-2 and macro definitions are used to automatically select the applicable version, although profiles.c-mpich1 and profiles.c-mpich2 are files known to work for these respective versions for mpich.

After installing the above software, edit `suntans/main/Makefile.in` so that the directories containing the appropriate packages are correctly specified as follows:

- `MPIHOME` should contain the base directory of the mpich distribution. For example, `MPIHOME=/usr/local/mpich-1.2.7`
- `PARMETISHOME` should contain the base directory of the ParMetis distribution.
- `TRIANGLEHOME` should contain the base directory of the Triangle libraries.

Note that there cannot be any spaces between the "=" sign and the value. As an example, the `Makefile.in` file might look like

```
MPIHOME=/usr/local/mpich-1.2.7
PARMETISHOME=/usr/local/packages/ParMetis-2.0
TRIANGLEHOME=/usr/local/packages/triangle
```

Once these locations are properly specified, compile the SUNTANS executable in `suntans/main` and link it with the software that has been installed with

```
make
```

This will create the main executable `sun`. To remove the object files, use

```
make clean
```

To clean up the distribution and return it to the original state it was in upon downloading, use

```
make clobber
```

The original source for SUNTANS contains empty macro definitions in `suntans/main/Makefile.in`, i.e.

```
MPIHOME=
PARMETISHOME=
TRIANGLEHOME=
```

Undefined macros in this file imply that the software is not installed, and suntans will compile accordingly. Note that you cannot run SUNTANS in parallel unless both MPICH and ParMETIS are installed, since ParMETIS performs the parallel grid partitioning. However, SUNTANS does not require the Triangle libraries to run in its serial or parallel modes. Omission of the Triangle libraries requires the generation of grid files using an alternate grid generation package, as described in Section 3.2.

1.2 Keeping up to date with CVS

You can keep an up to date copy of the SUNTANS distribution on your machine by using the cvs repository. To do so, you need to send your ssh public key to the suntans server administrator so that it can be installed in the authorized keys file on the suntans server. To obtain your public key, use

```
ssh-keygen -t rsa
```

This will create a public key in the file `~/.ssh/id_rsa.pub`, which you should send via email to the administrator. You may use an empty passphrase but it is not recommended. Once you send your public key to the suntans cvs administrator, in the `bash` shell, type

```
export CVSROOT=:ext:cvsuser@suntans:/home/cvs
export CVS_RSH=ssh
```

This sets the `CVSROOT` environmental variable so that when you use the cvs commands they look for the cvs repository on the suntans server as user cvsuser. You will be granted read-only access to the cvs repository, so you can keep an up to date copy of the latest source on your machine. To check out the latest copy, type

```
cvs checkout suntans
```

This will create the `suntans` directory and all of the subdirectories on the server. If you would like to sync your copy of SUNTANS with the copy on the server, use

```
cvs update suntans
```

Note that you are only allowed read access to the cvs server. Any changes you make to the SUNTANS files on your machine will not be updated on the server unless you are added to the group list on the server. You will also not be able to ssh into the server if you try, as this will freeze your screen as only cvs access is allowed via ssh.

2 Running SUNTANS

Before running SUNTANS, you must have a valid triangulation and a valid parameter file `suntans.dat`. For details on the triangulation, see Section 3, and for details on the `suntans.dat` parameter file, see Section 7. If you have a valid grid and parameter file in the same directory as the `sun` executable, SUNTANS can be executed as a single processor job by running the main executable `sun` with

```
mpirun sun -t -g -s
```

Running with the input and output data files in the same directory as the source and executable is in general not a good idea because SUNTANS creates many input and output files that will clutter up the local directory. It is best to create a new directory and specify that directory as the working data directory on the command line and place the grid and parameter files in that directory. For example, if the input files are in the local `data` directory, then SUNTANS would be run with

```
mpirun sun -t -g -s --datadir=./data
```

To run SUNTANS on multiple processors, use the `-np` flag with `mpirun`. For example, to run SUNTANS on 64 processors, use

```
mpirun -np 64 sun -t -g -s --datadir=./data
```

The following is a list of flags that determine the behavior of the `sun` executable upon running:

- `-t` Create a triangulation from a planar straight line graph. For details see Section 3.
- `-g` Partition the triangulation among the given number of processors and compute grid geometry and cell and processor connectivity. For details see Section 3.
- `-s` Run the SUNTANS solver.
- `-v[vvv]` Output information about the progress of the run. The more `v`s, the more verbose the output (maximum 4).
- `-w` Print out warnings that may lead to crashes or erroneous results to the screen (independent of `-v`).
- `-r` Restart SUNTANS from a previous run. For details see Section 5.

A typical SUNTANS run with, for example, 4 processors, will proceed as follows:

1. Place the parameter file `suntans.dat` and planar straight line graph file in a directory, say the `data` directory.
2. Create the triangulation and grid information with

   ```
   mpirun -np 4 sun -t -g --datadir=./data
   ```

This stores the grid information in files (see Section 3) for later reading later. Since this process may take some time it is a good idea to run this once for large grids and read in the grid data from a file.

3. Read the grid data from the files and run the solver and output information about the run with

   ```
   mpirun -np 4 sun -s -vv --datadir=./data
   ```

4. Once this run is finished it is possible to restart the run (for details see Section 5) with the data in `data` using

   ```
   mpirun -np 4 sun -s -r -vv --datadir=./data
   ```

Note that the reason behind being able to specify the data directory at the command line is that it enables the executable to be run from the same directory but to use data from different directories that may contain different parameters. For example, if the `data1` directory and `data2` directory contain different grid data (from previous calls to SUNTANS with the `-t` and `-g` flags), then SUNTANS can be run either with

```
mpirun -np 4 sun -s --datadir=./data1
```

or with

```
mpirun -np 4 sun -s --datadir=./data2
```

3 Creating or Reading in Triangular Grids

3.1 Using the triangle libraries

Triangular grids can be created from a simple planar straight line graph (pslg) which is specified as the `pslg` file in `suntans.dat`. The format of this file is similar to the format of the planar straight line graph file for use in triangle. The planar straight line graph file is a listing of points and edges that make up a closed contour. These edges will comprise the boundaries of the triangulation that are created by the triangle libraries. The simplest PSLG file is one which specifies a box with sides of length 1. This requires four points and four edges, as shown in the following file `box.dat`:

```
# Number of points
4
# List of points (x,y,marker)
0.0 0.0 0
0.0 1.0 1
1.0 1.0 2
1.0 0.0 3
# Number of segments
4
# List of segments and boundary markers (point #, point #, marker)
0 1 1
1 2 1
2 3 1
3 0 1
# Number of holes
0
# Minimum area
.005
```

The minimum area causes the triangle program to continue to add triangles until the minimum area of one of the triangles satisfies this area. Setting the `pslg` variable to `box.dat` in `suntans.dat` and running SUNTANS with

```
mpirun sun -t -g --datadir=./data
```

This creates a triangulation composed of 256 right triangles within a square with sides of length 1. The three columns after the number of points specification correspond to the x-y coordinates of each point, along with a marker for each point. Following the number of segments specification, the first two columns specify the indices of the points that make up the end points of each segment in C-style numbering, such that the indices go from 0 to one less than the number of points. The third column in the segment list specifies the marker for each segment. This is how the boundary conditions are specified, and is discussed in more detail in Section 4.2. The number of holes is currently not used.

Degenerate triangulations may arise when neighboring triangles are close to having right angles. In this case the distance between the Voronoi points may be close to zero and hence

may severely limit the time step. This is remedied by setting the `CorrectVoronoi` variable to 1 in `suntans.dat`. If this parameter is set, then the Voronoi points are corrected when the distance between Voronoi points is less than `VoronoiRatio` times the distance between the cell centroids. For example, for two neighboring right triangles, the Voronoi points will be coincident and hence the distance between the Voronoi points of these neighboring triangles will be zero. If `VoronoiRatio` is set to `0.5`, then the Voronoi points are adjusted so that the distance between them is half the distance between the cell centroids. If `VoronoiRatio` is set to `1`, then the Voronoi points correspond to the cell centroids.

A useful m-file `checkgrid.m` is provided in the suntans/mfiles directory that determines the shortest distance between Voronoi points on a grid as well as a histogram of the Voronoi distances on the entire grid. It is important to remember that most grid generation packages, including triangle, are meant for finite-element calculations in which calculations are usually node-based (i.e. everything is stored at the Delaunay points). Because SUNTANS is Voronoi-based, this places a strong constraint on the distance between the Voronoi points and as a result it can be quite difficult to generate good grids in highly complex domains. We have found a great deal of success with the grid generation package GAMBIT from Fluent, Inc. and use it for most of our production-scale SUNTANS runs.

3.2 Reading triangular grids from a file

Use of the `-t` flag creates three files specified in `suntans.dat`: `points`, `cells`, and `edges`. By default, these are specified to be

```
points    points.dat
edges     edges.dat
cells     cells.dat
```

The `points` file contains a listing of the x-y coordinates of the Delaunay points in the full triangulation before being subdivided among different processors. This file contains three columns although the last column is never used. The total number of lines in this file is N_p, the number of triangle vertics in the triangulation. The `edges` file contains N_e rows each of which defines an edge in the triangulation, and five columns in the following format:

```
Point1 Point2 Marker Voronoi1 Voronoi2
```

`Point1` and `Point2` contain indices to points in the `points` file and make up the end points of the Delaunay edges. Because SUNTANS uses C-style indexing, then $0 \leq$`Point1,Point2`$< N_p$. `Marker` specifies the type of edge. If `Marker=0`, then it is a computational edge, otherwise, it is a boundary edge, and the boundary condition is specified in Section 4.2. The last two entries, `Voronoi1` and `Voronoi2`, are the indices to the Voronoi points which make up the end points of the Voronoi edge which intersects this Delaunay edge. As such, we must have $0 \leq$`Voronoi1,Voronoi2`$< N_c$. These Voronoi points correspond to triangles defined in the file `cells`. Voronoi points which are ghost points are indicated by a -1. The `cells` file contains N_c rows each of which corresponds to a triangle in the triangulation, and 8 columns in the following format:

```
xv yv Point1 Point2 Point3 Neigh1 Neigh2 Neigh3
```

The xv and yv points correspond to the x-y coordinates of the Voronoi points of each triangle and Point1, Point2, and Point3 correspond to indices to points in the points file which make up the vertices of the triangle. These indices must satisfy
$0 \leq$Point1,Point2,Point3$< N_p$.
Neigh1, Neigh2, and Neigh3 correspond to indices to neighboring triangles. Neighboring triangles which correspond to ghost points are represented by a -1. For neighbors not lying outside boundaries, we must have
$0 \leq$Neigh1,Neigh2,Neigh3$< N_c$.
Because SUNTANS determines the number of triangle vertices N_p, edges N_e, and cells N_c by the number of rows in the points, cells, and edges files, respectively, it is important not to have extra carriage returns at the end of these files.

These three files are generated each time the -t flag is used with SUNTANS. If the -t flag is not used, then when called with -g, SUNTANS reads these three files and computes grid geometry and, if desired, partitions it among several processors. The -g flag outputs the following data files, which are specified in suntans.dat. One file associated with each of these descriptors is created for each processor in a partitioned grid. For example, if the file name specified after cells in suntans.dat is given by cells.dat, then when called with 2 processors, the -g flag would output two files names cells.dat.0 and cells.dat.1, each corresponding to the cells file of each processor.

- cells Same as the output when using -t, except on a per-processor basis. The indices to the triangle vertices still correspond to indices in the global points file, which is not distributed among the processors. All other indices are local to the specific processor.

- edges Same as the output when using -t, except on a per-processor basis. The indices to the end points of the edges still correspond to indices in the global points file, which is not distributed among the processors. All other indices are local to the specific processor.

- celldata Contains the grid data associated with the Voronoi points of each cell and contains N_c rows, where N_c is the number of cells on each processor (including inter-processor ghost points). Each row contains the following entries:

  xv yv Ac dv Nk Edge{1-3} Neigh{1-3} N{1-3} def{1-3}

  - xv yv are the Voronoi coordinates
  - Ac is the cell area
  - dv is the depth at the point xv,yv. This is the depth of the bottom-most face of the column beneath this cell and is not the actual depth, which is always greater than dv.
  - Nk is the number of vertical levels in the water column.
  - Edge{1-3} are indices to the three edges that correspond to the faces of the cell.
  - Neigh{1-3} are the indices to the three neighboring cells.
  - N{1-3} is the dot product of the unique normal with the outward normal on each face.

– `def{1-3}` is the distance from the Voronoi point to the three faces.

- `edgedata` Contains the grid data associated with the Delaunay edges and contains N_e rows, where N_e is the number of edges on each processor (including interprocessor edges). Each row contains the following entries:

  `df dg n1 n2 xe ye Nke Nkc grad{1,2} gradf{1,2} mark xi{1,2,3,4} eneigh{1,2,3,4}`

  – `df` is the length of the edge.

  – `dg` is the distance between the Voronoi points on either side of the edge. If this is a boundary edge then `dg` is twice the distance between the edge and the Voronoi point on the inside of the boundary.

  – `n1,n2` are the components of the normal direction of the edge. These correspond to the *unique* normals of each edge. The outward normal for this edge corresponding to a particular cell is given by `n1*N, n2*N`, where `N` is the dot product of the unique normal with the outward normal and is specified in the `celldata` file.

  – `xe, ye` are the coordinates of the intersection of the edge with the Delaunay edge.

  – `Nke` is the number of active edges in the vertical (see `Nkc`).

  – `Nkc` is the maximum number of active cells in the vertical which neighbor a given edge. `Nkc` is always at least `Nke`. See Figure 1 for a graphical depiction. item

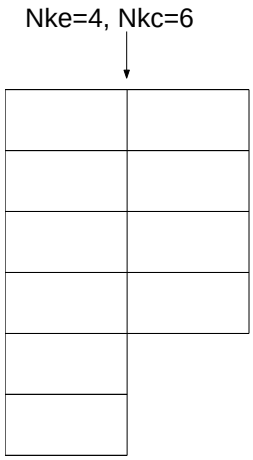

Figure 1: Depiction of an edge in which Nke=4 and Nkc=6.

`grad{1,2}` are indices to the Voronoi points defined in the `celldata` file. If N_c is the number of cells on a processor, then $0 \leq$`grad{1,2}`$< N_c$.

– `gradf{1,2}` are indices that determine the location of the edge in the ordering of the `Edge{1-3}` or `def{1-3}` arrays. Each cell contains a pointer to this edge in its list of `Edge{1-3}` pointers. The `gradf{1,2}` index is a number from 0 to 2 which determines which face number this edge is of a particular cell.

– `mark` Contains the marker type for this edge. All edges with the value 0 are computational edges, while other values are described in Section 4.2.

- `topology`

  ```
  Np Nneighs neighbor{0,1,2,...,Np-1}\n
  neigh0: num_cells_send num_cells_recv num_edges_send num_edges_recv
  cell_send_indices ...
  cell_receive_indices ...
  edge_send_indices ...
  edge_recv_indices ...
  neigh1: num_cells_send num_cells_recv num_edges_send num_edges_recv
  cell_send_indices ...
  cell_receive_indices ...
  edge_send_indices ...
  edge_recv_indices ...
  .
  .
  .
  neigh{Numneighs-1}: num_cells_send num_cells_recv num_edges_send num_edges_recv
  cell_send_indices ...
  cell_receive_indices ...
  edge_send_indices ...
  edge_recv_indices ...
  celldist[0] celldist[1] celldist[2] ... celldist[MAXBCTYPES-1]
  edgedist[0] edgedist[1] edgedist[2] ... edgedist[MAXBCTYPES-1]
  cellp[0],...,cellp[Nc-1]
  edgep[0],...,edgep[Ne-1]
  ```

- `vertspace` Contains the vertical grid spacings. This file has `Nkmax` rows, where `Nkmax` is the number of z-levels.

4 Specifying initial and boundary conditions

Initial and boundary conditions are specified by editing the files `initialization.c` and `boundaries.c`. When you edit either one of these files, you must recompile the SUNTANS executable in order for it to reflect changes made to the initial or boundary conditions.

4.1 Initial conditions

Initial conditions are set by altering the functions in the file `initialization.c` so that they return the desired initial distributions. There are five functions that specify the initial depth, free-surface, salinity, temperature, and velocity fields. Each function takes as its argument the x,y,z coordinates of the desired initial condition. You need to edit the function to return the desired initial value based on the given x,y,z coordinates. The five functions are

- `ReturnDepth` This returns the depth as a function of the given x,y coordinate. As an example, to set a linear slope that rises from 10 m depth to 1 m depth in 1000 m, the function would look like

  ```
  REAL ReturnDepth(REAL x, REAL y) {
    return 10 - 9*x/1000;
  }
  ```

  Note that the depth is specified as a positive quantity. This function is only used when the `IntDepth` variable is set to `0` in `suntans.dat`. Otherwise, when `IntDepth` is set to 1, the depth is interpolated from the file specified by the `depth` file in `suntans.dat`. This file must contain the depth data in three columns given by x, y, depth, where depth is specified as a negative quantity and elevation is positive.

- `ReturnFreeSurface` This returns the initial free surface distribution as a function of the x,y coordinate. As an example, to initialize the free surface with a 1 m cosine seiche in a 1000 m domain, the function would look like

  ```
  REAL ReturnFreeSurface(REAL x, REAL y, REAL d) {
    return cos(PI*x/1000);
  }
  ```

  Note that `PI` is a global variable defined in the file `suntans.h` as `3.141592654`. The depth is also supplied to the `ReturnFreeSurface` function in case the free surface is a function of the depth.

- `ReturnSalinity` and `ReturnTemperature` These are similar functions in that they return the specified scalar field as a function of the x, y, and z coordinates. Note that if `beta` is 0 in `suntans.dat`, then salinity transport is not computed, likewise if `gamma` is 0, then temperature transport is not computed. As it is now, the code computes the

temperature as a passive scalar while the density anomaly ρ is computed solely as a function of the salinity anomaly s, such that

$$\frac{\rho}{\rho_0} = \beta s \, .$$

This equation of state is specified in the file `state.c`.

- `ReturnHorizontalVelocity` This returns the horizontal velocity defined at the faces of each cell. The initial vertical velocity is computed by continuity from this initial velocity field. Since this function returns the velocity normal to a cell face, then you must specify both velocity components `u` and `v` and then return the component normal to the face, which is defined by the vector with components `n1` and `n2`. As an example, to return an irrotational vortex with maximum velocity of 1, centered about (x,y)=(5,5), the function would appear as

```
REAL ReturnHorizontalVelocity(REAL x, REAL y, REAL n1, REAL n2, REAL z) {
  REAL u, v, umag=1;

  u = -umag*(y-5)/5;
  v = umag*(x-5)/5;

  return u*n1+v*n2;
}
```

4.2 Specifying boundary conditions

SUNTANS allows the specification of the velocity at the open boundaries in the file `boundaries.c`. Boundary condition types on the velocity field can be specified in the `pslg` file which sets the marker as described in Sections 3.1 and 3.2. The marker type can be one of the following:

- `1` For closed boundaries.
- `2` For open or velocity-specified boundaries.
- `3` For open or stage-specified boundaries.
- `4` For no-slip boundary conditions.

If the marker is set to `1`, then the velocity is set to `0` at those edges. If it is set to `2`, then the velocity is specified in the file `boundaries.c` by iterating over edges with marker=`2`. If it is set to `3` then stage can be specified at the Voronoi cell center for the cell adjacent to this edge. No-slip boundaries are specified by setting the marker to `4`, where the velocity on the edge is specified by updating `phys->boundary_u`, `phys->boundary_v`, and `phys->boundary_w`. No-slip conditions on top and bottom cells are specified by setting `CdB=-1` or `CdT=-1`, respectively. An example covering usage of type `2` boundary conditions, type `3` boundary conditions, and type `4` boundary conditions will be covered in turn.

4.2.1 Open or velocity-specified boundary conditions

This file loops through the edges whose edge markers are 2. As an example, consider the flow in a channel that is 1000 m long in which two boundaries are specified as inflow and outflow and the other two are specified as solid walls, as shown in Figure 2. Suppose you would like

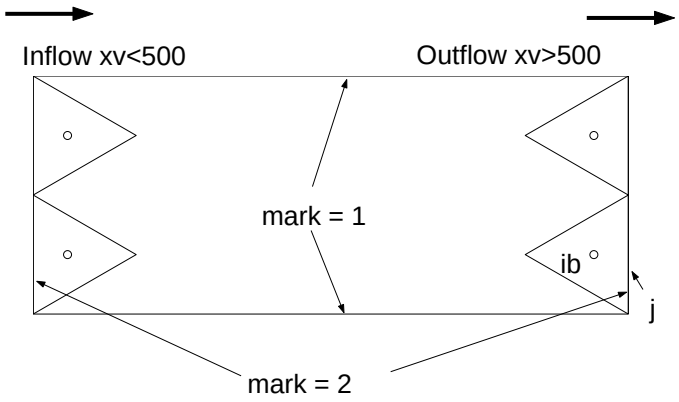

Figure 2: Depiction of a channel with an inflow at $x = 0$ m and outflow at $x = 1000$ m with the inflow/outflow boundaries specified with 2 markers and the solid walls with 1 markers. The indices of the cells adjacent to the boundaries are denoted by ib while the boundary faces have indices j.

to specify an incoming velocity that oscillates at a frequency ω due to an incoming wave at $x = 0$ m and that you would like to use the linearized open boundary condition to specify the outflow at $x = 1000$ m. Since the edge markers at the inflow and outflow are the same, you need to set the boundary condition based on the x,y location of the specified edge. To do so, you would set up the functions in the `boundaries.c` file as follows

- Function OpenBoundaryFluxes:

```
for(jptr=grid->edgedist[2];jptr<grid->edgedist[3];jptr++) {
  j = grid->edgep[jptr];

  ib = grid->grad[2*j];

  if(grid->xv[ib]>500) {
    for(k=grid->etop[j];k<grid->Nke[j];k++)
       ub[j][k] = -phys->h[ib]*sqrt(prop->grav/(grid->dv[ib]));
  } else {
    for(k=grid->etop[j];k<grid->Nke[j];k++)
       ub[j][k]=phys->boundary_u[jptr-grid->edgedist[2]][k]*grid->n1[j]+
                phys->boundary_v[jptr-grid->edgedist[2]][k]*grid->n2[j];
  }
}
```

- BoundaryVelocities Function:

```
for(jptr=grid->edgedist[2];jptr<grid->edgedist[3];jptr++)
  j = grid->edgep[jptr];

  ib = grid->grad[2*j];

  if(grid->xv[ib]>500) {
    for(k=grid->etop[j];k<grid->Nke[j];k++) {
      phys->boundary_u[jptr-grid->edgedist[2]][k]=phys->uc[ib][k];
      phys->boundary_v[jptr-grid->edgedist[2]][k]=phys->vc[ib][k];
      phys->boundary_w[jptr-grid->edgedist[2]][k]=
          0.5*(phys->w[ib][k]+phys->w[ib][k+1]);
    }
  } else {
    for(k=grid->etop[j];k<grid->Nke[j];k++) {
      phys->boundary_u[jptr-grid->edgedist[2]][k]=
          prop->amp*cos(prop->omega*prop->rtime);
      phys->boundary_v[jptr-grid->edgedist[2]][k]=0;
      phys->boundary_w[jptr-grid->edgedist[2]][k]=0;
      }
  }
}
```

The outermost `jptr` loop loops over the edges which have boundary markers specified as `2`. The `j` index is an index to an edge along the open boundary, and the `ib` index is an index to the cell adjacent to that boundary edge. Since the Voronoi points are specified at the `ib` indices, then we need to specify the boundary condition based on the location of the Voronoi points of the adjacent cell. In this case the open boundary exists at x=1000, but since we know that this boundary exists for boundary edges for $x > 500$, we use the if statement `if(grid->xv[ib]>500)` to specify these boundary edges. The first part of the if statement sets the flux at the open boundary over all the vertical levels with

```
for(k=grid->etop[j];k<grid->Nke[j];k++)
  ub[j][k] = -phys->h[ib]*sqrt(prop->grav/(grid->dv[ib]));
```

which is identical to

$$u_b = -h_b\sqrt{\frac{g}{d}},$$

and this is the linearized shallow-water free-surface boundary condition (Note that the upwind depth and free-surface height are being used to approximate the values at the open boundary). This loop uses the `Nke[j]` variable, which is the number of vertical levels at face `j`, and `etop[j]`, which is the index of the top level at the boundary. The variable `etop[j]` is usually `0` unless filling and emptying occur at the boundary edges.

In addition to setting the boundary flux in the OpenBoundaryFluxes function, the user must also set the Cartesian components of velocity in the BoundaryVelocities function. These are used to compute the advection of momentum. They can also be used to specify boundary velocities for no-slip type 4 boundary conditions. At the open outflow boundary, the boundary velocity components are set to the upwind value of the velocity component. For the u-component of velocity, for example, the boundary value is specified with

```
phys->boundary_u[jptr-grid->edgedist[2]][k]=phys->uc[ib][k];
```

Here, `uc` is the u-component of velocity at the Voronoi point with index `ib`.

At the inlet, only the Cartesian velocity components need to be specified. In this example, the u-component of velocity at the inlet is set with the code

```
phys->boundary_u[jptr-grid->edgedist[2]][k]=
                prop->amp*cos(prop->omega*prop->rtime);
```

and the other components are set to 0. This function uses the variables `amp` and `omega` which are set in `suntans.dat`, and `rtime` is the physical time in the simulation. The Cartesian velocities which are specified at the inlet boundary are then used to compute the flux in the OpenBoundaryFluxes function with

```
ub[j][k]=phys->boundary_u[jptr-grid->edgedist[2]][k]*grid->n1[j]+
          phys->boundary_v[jptr-grid->edgedist[2]][k]*grid->n2[j];
```

This is just the dot product of the velocity field specified at the boundary with the normal vector at the boundary edge, which points into the domain by definition. Examples of how to employ velocity boundary conditions are described in Sections 6.5 and 6.7. Section 6.5 also demonstrates how to specify salinity and temperature at the boundaries.

4.2.2 Open or stage-specified boundary conditions

A good example of specifying stage boundary conditions (type 3) is the `suntans/main/examples/estuary` example, where a stage boundary condition is specified at the ocean boundary to the west of the domain. A simplified tidal signal is specified via the following code contained within the function BoundaryVelocities() of `suntans/main/examples/estuary/boundaries.c`.

```
for(iptr=grid->celldist[1];iptr<grid->celldist[2];iptr++) {
  i = grid->cellp[iptr];
  phys->h[i]=-prop->amp*sin(prop->omega*prop->rtime);
}
```

In this example, type 2 boundary conditions result in indicies to computational cells adjacent to type 2 boundary condition being stored in [`celldist[1]`, `celldist[2]`). Stage information for these cells can then be specified by setting the cell stage via the cell pointer `i=grid->cellp[iptr]`. In this example, `prop->amp` specifies tidal amplitude A, `prop->omega` specifies the frequency ω, and `prop->rtime` the simulation time t within the prototype function $h = -A\sin(\omega t)$.

4.2.3 No-slip boundary conditions

No-slip boundary conditions in the plane XZ are specified via type `4` boundary conditions in `edges.dat`. Setting `CdT=-1` and `CdB=-1` within `suntans.dat` specifies no-slip boundary conditions for the top and bottom of the domain within the plane XY. As an example, consider the `testXZ` case in `suntans/main/examples/cavity`. The parameter file `suntans/main/examples/cavity/rundata/suntansXZ.dat` has

```
CdT                -1   # Drag coefficient at surface
CdB                -1   # Drag coefficient at bottom
```

and the function BoundaryVelocities() in `suntans/main/examples/cavity/boundariesXZ.c` has

```
  for(jptr=grid->edgedist[4];jptr<grid->edgedist[5];jptr++) {

    j = grid->edgep[jptr];
    ib=grid->grad[2*j];
    boundary_index = jptr-grid->edgedist[2];

    for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
      if(grid->xe[j]<0.5)
        phys->boundary_w[boundary_index][k]= -1.0;
      else
        phys->boundary_w[boundary_index][k]=0.0;
      phys->boundary_v[boundary_index][k] = 0.0;
      phys->boundary_u[boundary_index][k] = 0;
    }
  }
```

where the no-slip boundary condition is applied on the west side of the boundary directed downward via `phys->boundary_w[boundary_index]= -1.0` for edges with markers `jptr` in [`grid->edgedist[4]grid->edgedist[5]`).

4.3 Specifying heat flux

To specify source terms in the temperature equation, you will need to edit the HeatSource() function in suntans/main/sources.c. The implementation assumes adiabatic surface and bottom boundary conditions, and so all heat fluxes are implemented as source terms in the temperature equation of the form

$$\frac{\partial T}{\partial t} + \mathbf{u}\cdot\nabla T \;=\; \frac{\partial}{\partial z}\left[\left(\kappa_T + \kappa_{tv}\right)\frac{\partial T}{\partial z}\right] + Q(T)\,,$$

where κ_T is the molecular diffusivity (defined in suntans.dat) and κ_{tv} is the vertical turbulent eddy-diffusivity, computed in turbulence.c. terms to simplify the discussion, the temperature equation is given by

$$\frac{\partial T}{\partial t} = Q(T)\,. \tag{1}$$

The source term can be linearized about some temperature T_0 so that, defining $T = T_0 + \Delta T$, we have

$$\begin{aligned} Q(T) &= Q(T_0 + \Delta T)\,, \\ &= Q(T_0) + \left(\frac{\partial Q}{\partial T}\right)_{T_0} \Delta T + O(\Delta T^2)\,, \\ &= Q(T_0) + \left(\frac{\partial Q}{\partial T}\right)_{T_0} (T - T_0) + O(\Delta T^2)\,, \\ &= Q(T_0) - \left(\frac{\partial Q}{\partial T}\right)_{T_0} T_0 + \left(\frac{\partial Q}{\partial T}\right)_{T_0} T + O(\Delta T^2)\,, \\ &= A(T_0) + B(T_0)T + O(\Delta T^2)\,, \end{aligned}$$

where

$$A(T_0) = Q(T_0) - \left(\frac{\partial Q}{\partial T}\right)_{T_0} T_0\,,$$

$$B(T_0) = \left(\frac{\partial Q}{\partial T}\right)_{T_0}\,.$$

The heat equation is discretized with the theta method so that equation (1) becomes

$$\frac{T^{n+1} - T^n}{\Delta t} = A(T_0) + B(T_0)\left[\theta T^{n+1} + (1-\theta)T^n\right]\,,$$

where θ is specified by the value of `theta` in suntans.dat. After rearranging, the temperature at the new time step is obtained with

$$T^{n+1} = \left[1 - \theta \Delta t B(T_0)\right]^{-1}\left[T^n + \Delta t A(T_0) + (1-\theta)\Delta t B(T_0) T^n\right]\,.$$

Typically the best linearization is to linearize about the temperature at time step n so that $T_0 = T^n$. A simple implementation of heat flux is provided in the windstress example in `suntans/main/examples/windstress` and the code is in the HeatSource() function in `suntans/main/examples/windstress/sources.c`.

5 Restarting Runs

It is always a good idea to design production-scale runs so that at any point in time if your job crashes you can restart it without having lost too much data. At the end of a run, SUNTANS outputs the restart file specified by `StoreFile` in `suntans.dat`. To restart a run using this restart file, you need to copy this file into the file specified by `StartFile` in `suntans.dat`. This reads the data required to continue a run. For example, at the end of a two-processor run in which the `StoreFile` variable is set to `store.dat`, two restart files are created for each processor, namely `store.dat.0` and `store.dat.1`. To restart this run, set the entry for `StartFile` in `suntans.dat` to `start.dat` and copy `StoreFile` to `StartFile` for each processor, i.e. copy the contents of `store.dat.0` to `start.dat.0` and `store.dat.1` to `start.dat.1`.

Since the new output data from the restart run will overwrite any existing data from a previous run, it is a good idea to create a new directory for each restart run and copy the restart files into that directory. Each run requires the `suntans.dat` file as well as other files for the grid as well, but rather than copy these over to the new directory, it is best to create links to the already existing files in order to save disk space. In order for the restart run to proceed, links must be created for each of these files (or copies):

- suntans.dat
- celldata.dat.np
- edgedata.dat.np
- topology.dat.np
- vertspace.dat

where `np` corresponds to each processor id. Considering the previous example, to restart a run one might create a new directory called `run2`. Then links could be made to the existing files with

```
ln -s suntans.dat run2/suntans.dat
ln -s celldata.dat.0 run2/celldata.dat.0
ln -s celldata.dat.1 run2/celldata.dat.1
ln -s edgedata.dat.0 run2/edgedata.dat.0
ln -s edgedata.dat.1 run2/edgedata.dat.1
ln -s topology.dat.0 run2/topology.dat.0
ln -s topology.dat.1 run2/topology.dat.1
ln -s vertspace.dat run2/vertspace.dat
```

and then restarting SUNTANS (after copying the store files into run2) with

```
mpirun -np 2 sun -s -r --datadir=./run2
```

where the `r` flag indicates a restart run. An alternative method to perform a restart is to create a new directory and then alter the entries in `suntans.dat` so that the output

files are specified by their full rather than relative paths. For example, to set the output directory so that SUNTANS outputs the free surface to the new directory `run2`, the entry for `FreeSurfaceFile` in `suntans.dat` could be changed from

```
FreeSurfaceFile    fs.dat
```

to

```
FreeSurfaceFile    run2/fs.dat
```

6 SUNTANS Examples

6.1 Running the examples

Each example is located in one of the directories in the `examples` directory in the main source directory. As an example, consider the internal waves example located in `examples/iwaves`. The internal waves simulation in this directory is defined by the following files in the `examples/iwaves` directory:

- `initialization.c`: Initial conditions
- `boundaries.c`: Boundary conditions
- `rundata/suntans.dat`: Parameters
- `rundata/pslg.dat`: Planar straight line graph

All examples are run in the same way. The makefile in the `iwaves` directory compiles and links the initial and boundary condition files with the main sun executable in the source directory. Then it copies the parameter file (`suntans.dat`) and the planar straight line graph file to the directory `data` in the particular example directory. Then the main sun executable is run with

```
../../sun -t -g -s -vv --datadir=data
```

where the particular flags may depend on the example, but the executable is run from the main source directory and the data is local to the particular examples directory. In order to compile and run an example, enter that example directory and type `make test`. This will run the example simulation and place the output data into the data directory in the examples directory. While the simulation is running, the data can be viewed from another command line from the main source directory with

```
./sunplot --datadir=examples/iwaves/data
```

It is important that you run `make clobber` between successive tests in other test directories to ensure that you are recompiling the `../../sun` executable with the correct initial and boundary condition files each time. Without clobbering, rerunning `make test` will run the test case while only reflecting changes in `suntans.dat`. Therefore, it is not necessary to clobber if the same test is being run with different parameters in `suntans.dat`. The following test cases exist in the examples directory. The first 5 are discussed in detail in what follows. Note that many examples directories also contain an `mfile` directory that has useful mfiles for analyzing output.

1. tides: Example tidal forcing in Monterey Bay
2. accuracy: Demonstrates second-order time accuracy using an internal waves seiche
3. boundaries: River plume example
4. cavity: Cavity flow example with no-slip boundary conditions

5. lockexchange: Lock exchange flow with closed boundaries
6. iwaves: Internal tide generation over a simplified continental shelf
7. channel: Simple channel flow demonstrating use of the turbulence model
8. cylinder: Formation of vortex street in lee of a cylinder
9. estuary: Tidal flow in a simple estuary with specified river inflow
10. tides-restart: Same as tides, but shows how to use restart filed
11. wetdry: Example of wetting and drying
12. windstress: Constant wind and heat forcing over a parabolic lake

6.2 Tidal forcing

This example is in the `examples/tides` directory and contains an example of how to force SUNTANS with tidal constituents at the boundaries. The domain consists of Monterey Bay and is a simplified example of the case run by Jachec *et al.* [3], which focuses on internal wave generation in the region. This example consists only of the barotropic flow and the user should be warned that because of the coarse resolution of the grid and of the boundary conditions that the results have not been validated. The important feature of this example is that it can be used as a starting point from which to set up a more complex simulation containing tidal forcing. Note that all files and directories are relative to the directory `suntans/main/examples/tides` unless otherwise noted. A very brief README file also exists in `suntans/main/examples/tides/README` for the impatient.

6.2.1 Grid and bathymetry

The grid for this example was generated with GAMBIT and is shown in Figure 3 (for tips on using GAMBIT for SUNTANS please see
`http://suntans.stanford.edu/documentation/suntans_tutorial.pdf`). Grids like this can be generated by obtaining a coastline from the NOAA coastline extractor
`http://rimmer.ngdc.noaa.gov/coast/`
and reading it into a grid generation program **after** converting the coordinates to a Cartesian grid. Most grids in SUNTANS are obtained with a UTM projection, but for larger grids that cross over multiple UTM zones, the Mercator projection is also suitable. Upon running SUNTANS (with `-g -vv`), Voronoi distance statistics will be output as follows:

```
Voronoi statistics:
        Minimum distance: 1.97e+02
        Maximum distance: 4.43e+03
        Mean distance: 2.85e+03
        Standard deviation: 4.14e+02
```

Note that while the minimum distance in this example is acceptable, the Voronoi distances do not indicate whether there are degenerate cells in the domain (i.e. obtuse triangles). In order to correct for any degenerate triangles, the parameter `CorrectVoronoi` is set to `-1`, and the parameter `VoronoiRatio` is set to `85` degrees (these parameters are in `suntans.dat`). This corrects any triangles with angles greater than 85 degrees by placing the Voronoi points at the triangle centroids. After correction, the statistics are displayed as

```
Corrected 11 of 1534 cells with angles > 85.0 degrees (0.72%).
Voronoi statistics after correction:
        Minimum distance: 9.49e+02
        Maximum distance: 4.34e+03
        Mean distance: 2.85e+03
        Standard deviation: 3.95e+02
```

This highlights the fact that 11 cells in the grid were obtuse and were corrected. Also note that as a result of the correction the minimum distance between Voronoi points increased from 197 m to 949 m. This substantially raises the minimum allowable time step for stability of the nonlinear advection terms.

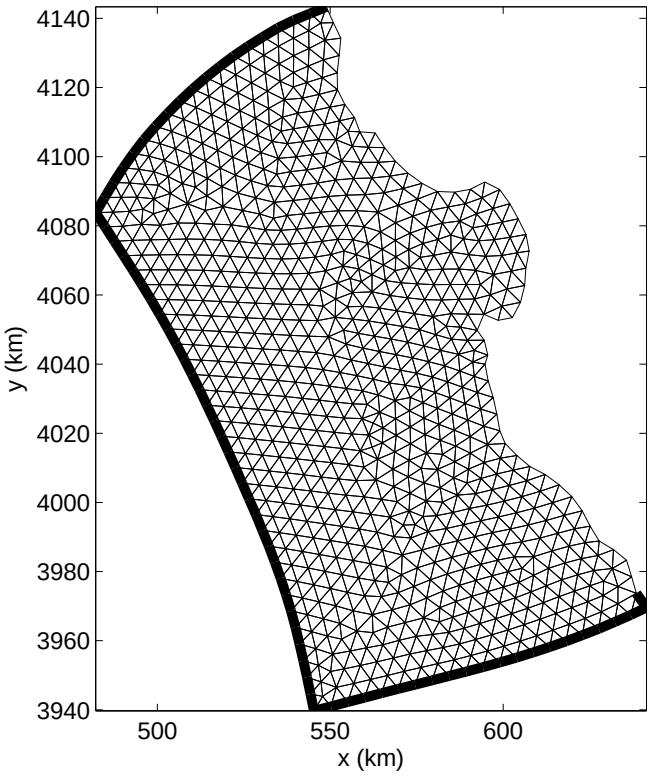

Figure 3: Monterey Bay grid generated with GAMBIT. The boundary edges with markers of type 2 are highlighted in bold. The other boundary edges are type 1 (closed).

For the bathymetry, 250 m resolution bathymetry is available in the file `rundata/mbay_bathy.dat` which was obtained from the MBARI multibeam survey `http://www.mbari.org/data/mapping/monterey/default.htm`

This file is specified by the `depth` variable in `suntans.dat`. The interpolated bathymetry is depicted in Figure 4. Upon running SUNTANS, bathymetry from this file is interpolated onto the (possibly corrected) Voronoi points. Because this procedure can take quite some time, it is a good idea to only run it when necessary and use the pre-interpolated bathymetry for subsequent runs. Each time SUNTANS interpolates bathymetry, it outputs the interpolated bathymetry into the file `mbay_bathy.dat-voro` (it appends "-voro" to the specified bathymetry file). This file is created when the variable `IntDepth` is set to 1 in `suntans.dat`. Otherwise, if this variable is set to 2, data is not interpolated but instead is read in from the `-voro` file. Note that any time changes are made to the Voronoi points, the bathymetry should be re-interpolated.

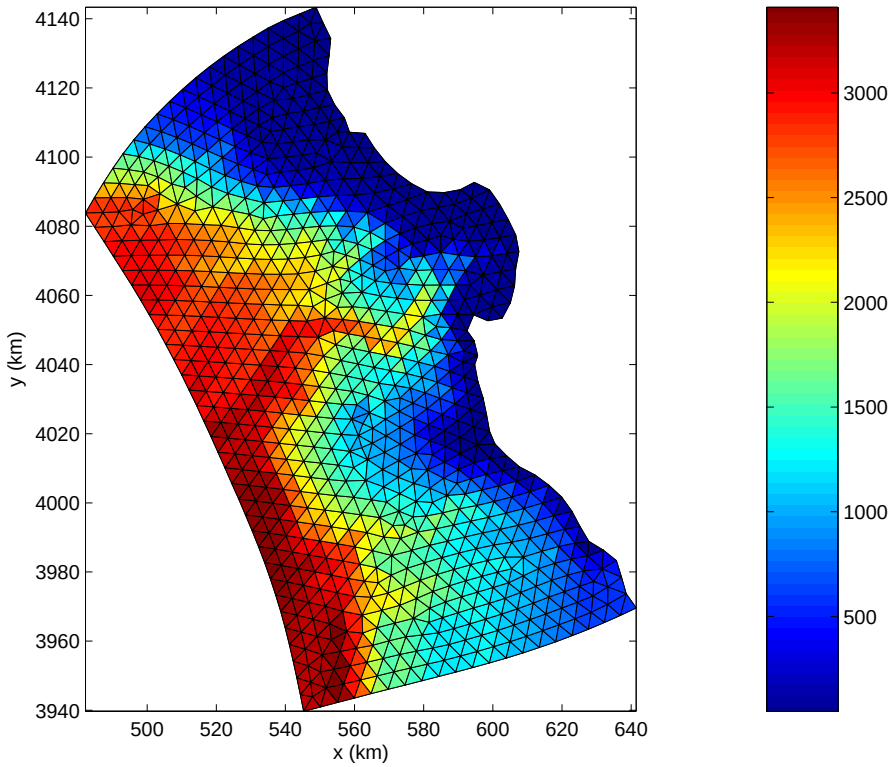

Figure 4: MBARI bathymetry (in m) interpolated onto the Monterey Bay grid.

6.2.2 Initial conditions

Initial conditions for this problem are a quiescent free surface and velocity field, and no stratification. Stratification can be turned off by setting `beta` to zero in `suntans.dat`. This is because the equation of state, as given in `suntans/main/state.c` is linear and only requires `beta` to convert from salinity to density. Note that when `beta` is zero, advection of salinity is also not computed, which can save on computation time.

6.2.3 Boundary conditions

Boundary conditions are given by the first eight tidal constituents as computed by OTIS which can be obtained from
`http://www.coas.oregonstate.edu/research/po/research/tide/`
Tidal constituent data is read into SUNTANS in the function `BoundaryVelocities` in the file `boundaries.c`, and the data is stored in the files specified by the variable `TideInput` in `suntans.dat` (the default is `tidecomponents.dat`). Each processor will require a file with the name `tidecomponents.dat.proc`, where `proc` is the processor number. This file contains the tidal constituents for the horizontal velocity components and the free surface height (in cm s^{-1} and cm) at the locations of the centers of each boundary edge of type 2. These locations are given in the files specified by the variable `TideOutput` in `suntans.dat` (the default is `tidexy.dat`). The `TideInput` file has the following binary format for each processor:

```
numtides (1 X int) number of tidal constituents.
numboundaryedges (1 X int) number of boundary edges.
omegas (numtides X REAL) frequencies of tidal components.
u_amp[0] (numtides X REAL) amplitude of easting velocity at location 0.
u_phase[0] (numtides X REAL) phase of easting velocity at location 0.
v_amp[0] (numtides X REAL) amplitude of northing velocity at location 0.
v_phase[0] (numtides X REAL) phase of northing velocity at location 0.
h_amp[0] (numtides X REAL) amplitude of free surface at location 0.
h_phase[0] (numtides X REAL) phase of free surface at location 0.
u_amp[1] (numtides X REAL) amplitude of easting velocity at location 1.
u_phase[1] (numtides X REAL) phase of easting velocity at location 1.
v_amp[1] (numtides X REAL) amplitude of northing velocity at location 1.
v_phase[1] (numtides X REAL) phase of northing velocity at location 1.
h_amp[1] (numtides X REAL) amplitude of free surface at location 1.
h_phase[1] (numtides X REAL) phase of free surface at location 1.
...
u_amp[numboundaryedges-1] (numtides X REAL) amplitude of easting velocity at last location.
u_phase[numboundaryedges-1] (numtides X REAL) phase of easting velocity at last location.
v_amp[numboundaryedges-1] (numtides X REAL) amplitude of northing velocity at last location.
v_phase[numboundaryedges-1] (numtides X REAL) phase of northing velocity at last location.
h_amp[numboundaryedges-1] (numtides X REAL) amplitude of free surface at last location.
h_phase[numboundaryedges-1] (numtides X REAL) phase of free surface at last location
```

The user can create this file or it can be created using the `suntides` matlab package, (located in `suntans/mfiles/suntides`) which is an adaptation of the `tidegui` package created by Brian Dushaw[1]. The `suntides.m` file reads in the x-y locations of the boundary points for each processor and outputs the associated tidal components in binary form. The matlab script `tides.m` in the tides example directory performs this action. The x-y location data files are created by running SUNTANS, which will complain if no `tidecomponents.dat.proc` files are found in the data directory, i.e the directory specified in the command line call in suntans with `--datadir=`. In the present example, this directory is given by `data`. Upon running the test, SUNTANS will complain with

```
Error opening data/tidecomponents.dat.0!
Writing x-y boundary locations to data/tidexy.dat.0 instead.
Error opening data/tidecomponents.dat.1!
```

[1] http://909ers.apl.washington.edu/ dushaw/tidegui/tidegui.html

```
Writing x-y boundary locations to data/tidexy.dat.1 instead.
```

This indicates that the tidal component data files were missing, so SUNTANS writes the locations of the markers of type 2 (the boundary edges) to the `tidexy.dat.proc` files. Once these files have been created, the `tides.m` script can be run to create the `tidecomponents.dat.proc` files. This can be run from the command line in unix with `matlab < tides.m`.

In order to prevent transient oscillations associated with the impulsive starting of the tidal boundary conditions, it is always a good idea to spin-up the boundary conditions over a day or so with

$$F_{actual} = F_{imposed}\left[1 - \exp(t/\tau_{ramp})\right] ,$$

so that the forcing approaches the imposed forcing over a time scale of τ_{ramp}. This is implemented in the file `boundaries.c` in the function `BoundaryVelocities`, where the tidal data is input. For example, the u-velocity component is imposed with the line

```
phys->boundary_u[jind][k]=u*(1-exp(-prop->rtime/prop->thetaramptime))/100.0;
```

Note that the value of `u` is multiplied by the ramping function as well as divided by 100.0 in order to convert from cm s^{-1} to m s^{-1}. The ramping function ramps up over a time scale given by the variable `thetaramptime`, which is defined as `86400` (one day in seconds) in `suntans.dat`. This variable also implies more damping during this initial transient period by ramping the value of `theta` in the simulation from 1 (fully implicit barotropic time-stepping) to the value specified in `suntans.dat` by the variable `theta`. Note that $\theta > 0.5$ for stability, and the value of $\theta = 0.55$ is typically used. In the above expression for the damping transient, for the time, the variable `rtime` is used, which represents the time, in seconds, from the beginning of the simulation, which starts at `rtime=0`. Using the `suntides` example, all tidal phases are with reference to the beginning of the year specified when running the `suntides.m` script. For the present example, the year is given by the line `year = 2006` in `setup_tides.m`. In order to start the simulation from a specific day in 2006, a time offset is specified in `suntans.dat` as the variable `toffSet`, which is in days. This offset is read into `boundaries.c` and converted to seconds with the line

```
toffSet = MPI_GetValue(DATAFILE,"toffSet","BoundaryVelocities",myproc)*secondsPerDay;
```

6.2.4 Running the example

This example is run by typing `make test` in the `suntans/main/examples/tides` directory. This will compile the boundary conditions specified in `boundaries.c` and link them with the main `sun` executable, and then run the test. Data is copied from the `rundata` directory to the `data` directory, which becomes the working directory. Typing `make test` again only copies `rundata/suntans.dat` to `data/suntans.dat` and runs the example again. Therefore, if any changes that are made to the grid or any associated parameters, the directory should be flushed with `make clobber`. This ensures that typing `make test` will recompute the grid.

Upon running the tidal example, the simulation will complain that the tidal input files (as specified by `TideInput` in `suntans.dat`) are not present, with the error message

```
Error opening data/tidecomponents.dat.0!
Writing x-y boundary locations to data/tidexy.dat.0 instead.
Error opening data/tidecomponents.dat.1!
Writing x-y boundary locations to data/tidexy.dat.1 instead.
```

In order to create these files (for details see Section 6.2.3), run the matlab script `tides.m`. This script will read in the locations of the boundary points in the `tidexy.dat.*` files and output the required `tidecomponents.dat.*` files. In order to run this script, the paths to the required m-files must be specified in the file `setup_tides.m`. These are given by

```
addpath ../../../mfiles
addpath ../../../mfiles/suntides
addpath /home/fringer/research/SUNTANS/tides/m_map
```

The first two paths should not need to be changed since these represent the default locations of the suntans mfiles and suntides directories. However, the user must set the location of the path to the m_map directory, which contains routines to convert from lon/lat coordinates to Cartesian coordinates using the UTM projection. This package can be downloaded from `http://www.eos.ubc.ca/~rich/map.html` Note that the data directory in the `suntides` directory must also be unpacked in order to run the `suntides.m` script. This can be done by typing `make` in the `suntans/mfiles/suntides` directory. Running the `tides.m` script will generate the `tidecomponents.dat` files and will display the following message:

```
File ./data/tidexy.dat.0, found 40 boundary points.
File ./data/tidexy.dat.1, found 32 boundary points.
```

This indicates that the locations of the boundary edges were found in the files `tidexy.dat.*` and these were output into the `tidecomponents.dat.*` files in the `data` directory. Once these files have been created, the example can be run with `make test`, which will proceed as follows:

```
Running suntans...
Processor 0,  Total memory: 1.77 Mb, 3865 cells
All processors: 3.58 Mb, 7719 cells (474 bytes/cell)
Outputting data at step 1 of 13440
5% Complete. 5.11e-02 s/step; 652.11 s remaining.
Outputting data at step 1344 of 13440
10% Complete. 4.85e-02 s/step; 587.19 s remaining.
15% Complete. 4.68e-02 s/step; 534.33 s remaining.
Outputting data at step 2688 of 13440
.
.
.
```

This indicates that the simulation (which by default runs on two processors) takes 0.05 seconds per time step to run, which is 1800 times faster than real time (since the time step, as specified by the `dt` variable in `suntans.dat` is 90 s). Data is output every 1344 time steps as indicated by the `ntout` variable in `suntans.dat`. The simulation will run for a total of `nsteps=13440` time steps or a total simulation time of `nsteps*dt`=14 days. Note that the values of the horizontal (`nu_H`) and vertical (`nu`) eddy-viscosities (called "laminar" because they are constant) are large in order to prevent the buildup of grid-scale energy over the course of the simulation on a relatively coarse grid.

Output of the results can be viewed with the `compare_to_otis.m` matlab script (which requires the correct path locations in `setup_tides.m`). This script compares the results of SUNTANS to the OTIS results at the location specified in the file `data/dataxy.dat` (in UTM coordinates). For details on how to specify these points and analyze data from these points, see the header in the file `suntans/main/profiles.c` and the associated mfile `suntans/mfiles/profplot.m`. The associated lines in the present example in `suntans.dat` are given by

```
########################################################################
#
# For output of data
#
########################################################################
ProfileVariables        hu    # Only output free surface and currents
DataLocations   dataxy.dat    # dataxy.dat contains column x-y data
ProfileDataFile profdata.dat  # Information about profiles is in profdata.dat
ntoutProfs              1     # Output profile data every 1 time step
NkmaxProfs              1     # Only output the top 1 z-level
numInterpPoints         1     # Output data at the nearest neighbor.
```

These indicate that free surface and velocity data at the locations specified in the file `dataxy.dat` will be output every time step (`ntoutProfs`). Because `NkmaxProfs` is set to 1, only the top cell is output. Otherwise, if `NkmaxProfs` is zero, all layers are output (this example has `Nkmax=10` layers). The `numInterpPoints` variable indicates how many of the points nearest to the desired point are output. SUNTANS does not interpolate output data; this is left up to the user. In the present example it suffices to output nearest-neighbor data. The output data can be compared to the predictions of OTIS with the script `compare_to_otis.m`. Over the 14-day period, the results are given in Figure 5. It should be noted that because the OTIS data used for this example is quite coarse (1 degree) and because the interpolation used to obtain the constituents near the coastline can lead to inaccurate predictions of the tidal boundary conditions in the shallow areas of the domain, this comparison should not be regarded as a measure of the accuracy of SUNTANS tidal simulations. The user is encouraged to obtain higher-resolution tidal data for more accurate simulations. For more detail please visit Brian Dushaw's web page[2]. To highlight the differences associated with the coarse tidal data, Figure 6 compares the results of SUNTANS when employing three different interpolation techniques to obtain the tidal data at the SUNTANS boundaries. The particular interpolation method can be changed in the function `suntans/mfiles/suntides/get_tides.m`, and the results are strikingly different because the interpolation method has a significant effect on forcing values near the coastline, since it is assumed that land values from OTIS are identically zero.

6.3 Time accuracy

This example is in the `examples/accuracy` directory and it serves to verify the time accuracy of SUNTANS. Details of the time accuracy of SUNTANS and this particular test case can

[2]http://909ers.apl.washington.edu/ dushaw/tidegui/tidegui.html

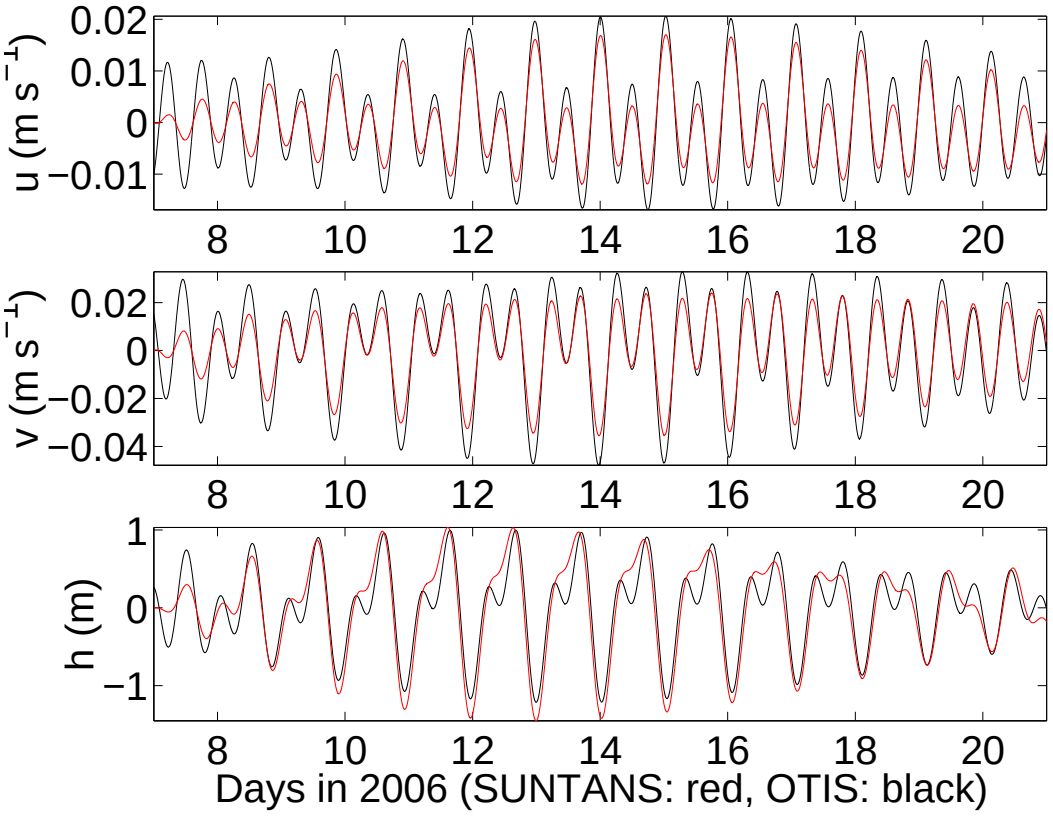

Figure 5: Comparison of SUNTANS to OTIS at the location specified in the dataxy.dat file.

be found in Fringer *et al.* [2]. The parameter file is given by `suntans.in`, which lacks the parameters `dt` and `nsteps`. These are copied to the actual parameter file `data/suntans.dat` upon running the script `accuracy.sh`, which runs SUNTANS with different time steps and computes accuracy criteria using the `accuracy.c` code. This code is a good example of how to read in and analyze SUNTANS data using the C programming language. More details are given below.

6.3.1 Grid

The domain is 100 m long and 100 m deep, and it uses 100 vertical levels and 100 equilateral triangles, as defined in the planar straight line graph file `rundata/oned.dat`, which was created with the `onedgrid.m` m-file which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

All boundary edges are of type 1 (closed) boundaries. The depth is constant and is defined in `initialization.c` in the `ReturnDepth` function with the line

```
return 100;
```

Note that in order for this depth to be specified, the `IntDepth` parameter must be set to 0 in the `suntans.dat` parameter file.

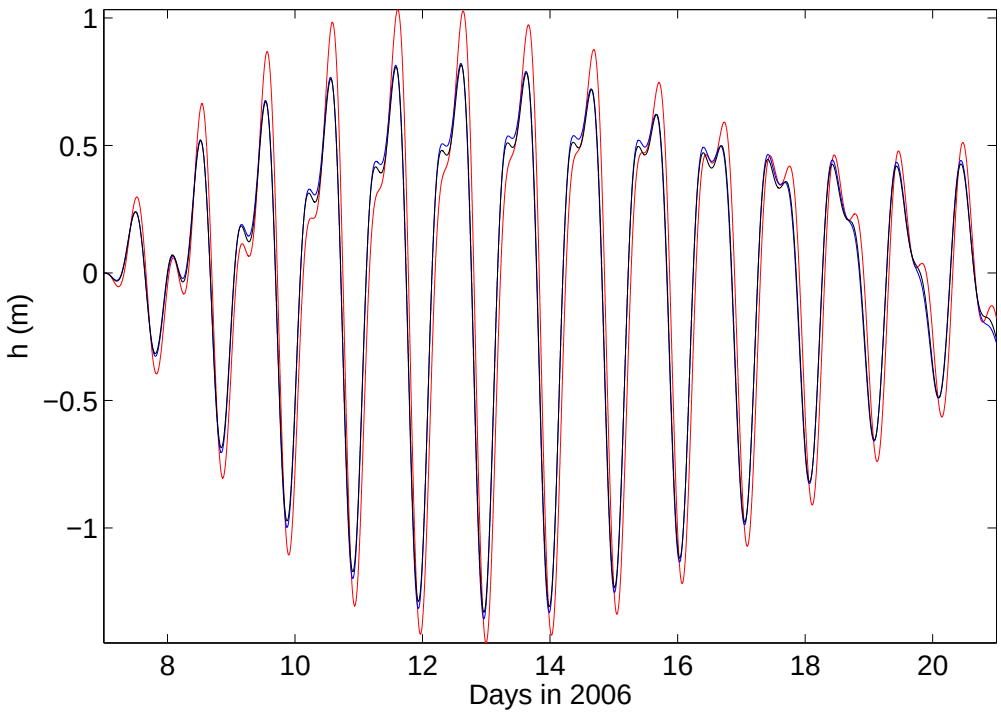

Figure 6: Comparison of SUNTANS results when using different matlab interpolation techniques to interpolate the tidal constituents from OTIS onto the SUNTANS boundary points. Legend: red=linear (same as Figure 5), blue=cubic, black=spline.

6.3.2 Initial conditions

The initial velocity field is quiescent, the initial free surface height is 0, and the initial density distribution is defined in the function `ReturnSalinity` in `initialization.c` as

```
return -.03*tanh(2.0*2.6467/20*(z+50-cos(PI*x/100)));
```

Because $\beta = 1$ in `suntans.dat` (the parameter `beta`), this is also the initial density distribution, since $(\rho - \rho_0)/\rho_0 = \beta(s - s_0)$ More generally, then, this density distribution is given by

$$\frac{\rho}{\rho_0} = -\frac{1}{2}\frac{\Delta\rho}{\rho_0}\tanh\left[\frac{2\tanh^{-1}(\alpha)}{\delta}\left(z + \frac{D}{2} - \zeta\right)\right],$$

where

- $\Delta\rho/\rho_0 = 0.06$: Density difference between layers.
- $\alpha = 0.99$: Parameter that defines interface extent.
- $\delta = 20$ m: Interface thickness.
- z: Vertical coordinate
- $D = 100$ m: Depth

- $\zeta = \cos(\pi x/L)$: Interface profile, where $L = 100$ m is the domain length.

and is shown in Figure 7.

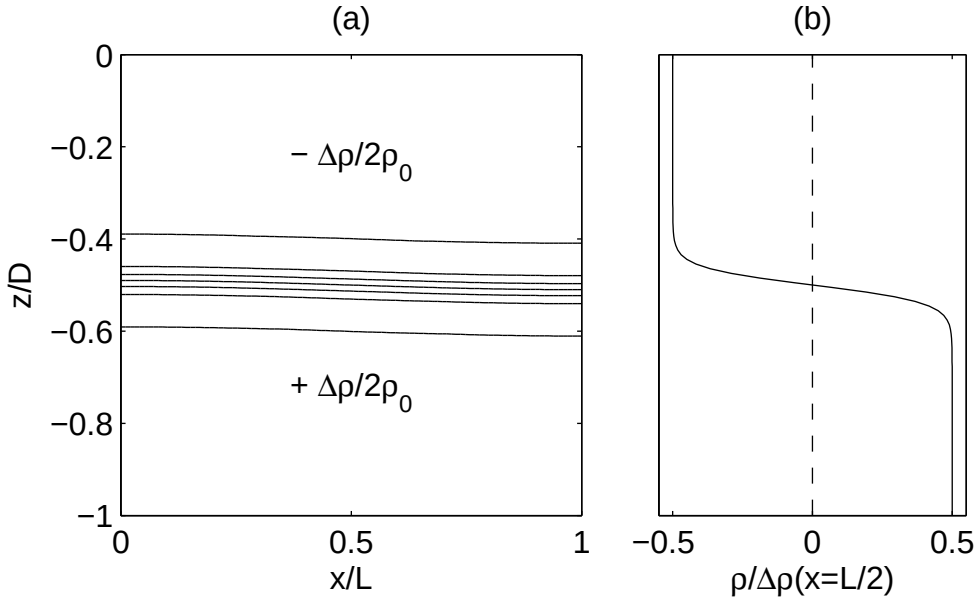

Figure 7: Initial density distribution in x-z (a) and as a profile at $x = L/2$ (b) for the time accuracy test case.

6.3.3 Boundary conditions

Because all boundaries of the planar straight line graph (`rundata/oned.dat`) are of type 1, the functions in the `boundaries.c` file are not used in this example.

6.3.4 Running the test

The accuracy test case is run at the command line with

```
make test
```

This will compile the local initial and boundary condition files and link them with the executable `../../sun`. The script `accuracy.sh` loops through different time step sizes given by $\Delta t_n = \Delta t_0/2^n$, with $n = 0, 1, \ldots, 5$, where $\Delta t_0 = 0.1$ s. If the reference solution is given by the solution ϕ^{ref} with a time step size of $\Delta t_0/32$, then the error between a solution using $\Delta t = \Delta t_0/2^n$ and the reference solution can be calculated with

$$Error(n)^2 = \frac{\sum_{i=0}^{Nc-1}\sum_{k=0}^{N_{kmax}-1}\left(\phi_{i,k} - \phi_{i,k}^{ref}\right)^2}{\sum_{i=0}^{Nc-1}\sum_{k=0}^{N_{kmax}-1}\left(\phi_{i,k}^{ref}\right)^2}.$$

Since this is a second-order method, we must have $Error(n)/Error(n-1) = 4$ since the time step size for $Error(n)$ is double that of $Error(n-1)$. Therefore, $Error(n)/Error(0) = 4^n =$

$(\Delta t_0/\Delta t_n)^2$, $n = 0, 1, \ldots, 4$. After the `accuracy.sh` script finishes, the results are analyzed with the code in `accuracy.c`, which computes $Error(n)$. The output of this code is given by

```
Error results (Error(n)/Error(0)):

Your results:
-------------------------------------------------
dt0/dt  U       W       S       Q       Q0      h
-------------------------------------------------
1       1.0     1.0     1.0     1.0     1.0     1.0
2       3.9     3.9     4.0     4.1     1.2     4.0
4       15.6    15.5    16.2    16.4    2.1     16.4
8       65.1    64.8    68.1    68.9    4.4     69.1
16      324.7   323.5   340.6   344.8   12.8    345.9

Reference results:
1       1.0     1.0     1.0     1.0     1.0     1.0
2       3.9     3.9     4.0     4.1     1.2     4.0
4       15.6    15.5    16.2    16.4    2.1     16.4
8       65.1    64.8    68.1    68.9    4.4     69.1
16      324.7   323.5   340.6   344.8   12.8    345.9

Difference (relative):
1       0.00    0.00    0.00    0.00    0.00    0.00
2       -0.00   0.00    0.00    0.00    0.00    -0.00
4       -0.00   0.00    0.00    0.00    0.00    -0.00
8       -0.00   0.00    0.00    0.00    0.00    -0.00
16      -0.00   0.00    0.00    0.00    0.00    -0.00
```

The code outputs the results from the current run, then displays the results that have been saved from a working version of this simulation, and then plots the relative difference between your results and the stored results as

$$\text{Difference} = \frac{\text{Your}(Error(n)/Error(0)) - \text{Stored}(Error(n)/Error(0))}{\text{Stored}(Error(n)/Error(0))}.$$

Each column represents:

- `dt0/dt`: $\Delta t_0/\Delta t^n$, $n = 0, 1, \ldots 4$
- `U`: Horizontal velocity.
- `W`: Vertical velocity
- `S`: Salinity (or density, since $\beta = 1$).
- `Q`: Nonhydrostatic pressure using the second-order Adams-Bashforth extrapolation using the last two time steps (for details see Fringer *et al.* [2]).

- `Q0`: Nonhydrostatic pressure without the extrapolation of `Q`.
- `h`: Free surface.

The first time you download and run the examples, the results you obtain should be identical to the stored results since they were obtained with the same code, and the relative difference should be identically zero. Changing parts of the code may or may not affect the accuracy, but if it does then this will be reflected by nonzero values for the relative difference.

6.4 Lock exchange

This example is in the `examples/lockexchange` directory and it demonstrates the nonhydrostatic internal lock exchange problem. Due to the coarseness of the grid being used, numerical diffusion acts to limit the formation of the KH billows along the interface. These billows can easily be obtained by increasing the resolution, following the simulation outlined by Fringer *et al.* [2].

6.4.1 Grid

The grid in this example is obtained with the `onedgrid.m` m-file which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

In this case, the grid contains 200 cells in the horizontal and 20 in the vertical, with a length of 100 m and a depth of 20 m. This constant depth is specified in `initialization.c` in the function `ReturnDepth` with the line

```
return 20;
```

Note that in order for this depth to be specified, the `IntDepth` parameter must be set to 0 in the `suntans.dat` parameter file. As shown in Figure 8, this grid is stretched in the vertical in order to provide extra resolution at the bottom boundary. This is done by specifying a negative stretching factor of $r = -1.025$ (`suntans.dat: rstretch = -1.025`), which causes each grid layer to be 1.025 times thicker than the layer beneath it.

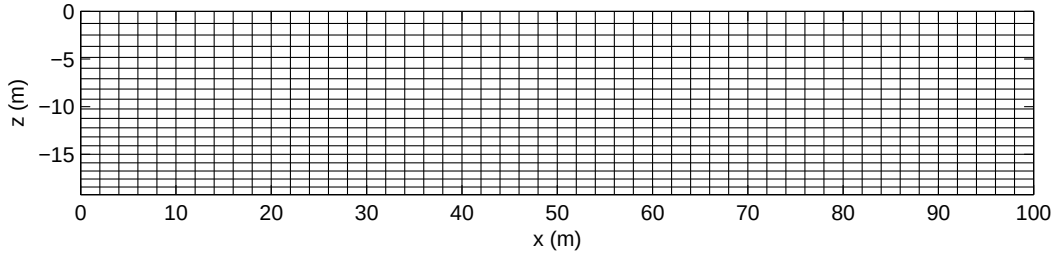

Figure 8: Depiction of the vertically stretched grid for the lock exchange problem. Every fourth vertical cell face plotted for clarity.

6.4.2 Initial conditions

The initial velocity field is quiescent, the initial free surface height is 0, and the initial density distribution is defined in the function `ReturnSalinity` in `initialization.c` as

```
return -.001*tanh(2.0*2.6467/5*(x-50.0));
```

Because $\beta = 1$ in `suntans.dat` (the parameter `beta`), this is also the initial density distribution, since $(\rho - \rho_0)/\rho_0 = \beta(s - s_0)$ More generally, then, this density distribution is given by

$$\frac{\rho}{\rho_0} = -\frac{1}{2}\frac{\Delta\rho}{\rho_0}\tanh\left[\frac{2\tanh^{-1}(\alpha)}{\delta}\left(x - \frac{L}{2}\right)\right],$$

where

- $\Delta\rho/\rho_0 = 2 \times 10^{-3}$: Density difference.
- $\alpha = 0.99$: Parameter that defines interface extent.
- $\delta = 5$ m: Interface thickness.
- x: Horizontal coordinate.
- $L = 100$ m: Length

and is shown in Figure 9.

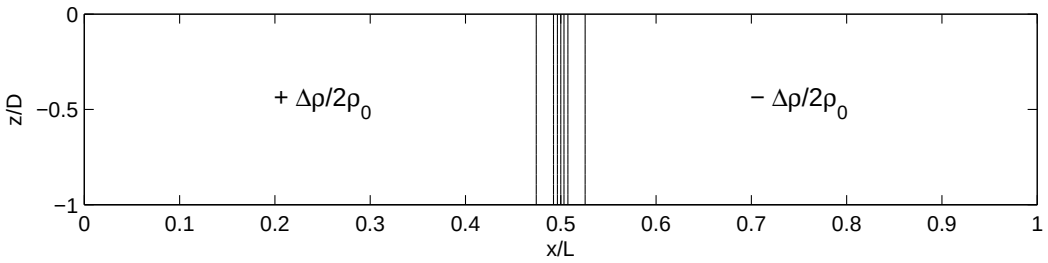

Figure 9: Initial density distribution in x-z (a) and as a profile at $x = L/2$ (b) for the lock exchange test case.

6.4.3 Boundary conditions

Because all boundaries of the planar straight line graph (`rundata/oned.dat`) are of type 1, the functions in the `boundaries.c` file are not used in this example.

6.4.4 Running the test

This test case is run with the command

```
make test
```

The simulation runs for 500 time steps (`suntans.dat: nsteps = 500`) with a time step size of 0.2 s (`suntans.dat: dt = 0.2`). For this grid, given that the Voronoi distance for the equilateral triangles with sides of length $\Delta x = L/200$ is $D_g = \Delta x/\sqrt{3} = 0.2887$ m, and the maximum velocity is roughly $u_{max} = 0.5$ m s^{-1} (it actually never exceeds 0.44 m s^{-1}), the horizontal Courant number is $C_x = u_{max}\Delta t/D_g = 0.35$, and the vertical Courant number is $C_z = w_{max}\Delta t/\Delta z_{min} = 0.2$m s^{-1}0.2s/0.78$m = 0.05$. Because central differencing is employed for advection of momentum (`suntans.dat: nonlinear = 2`), the horizontal grid Peclet number must satisfy the one-dimensional stability criterion which requires that $Pe_{\Delta x} < 2/C_x$. This is an approximation and is not as restrictive as it should be for multidimensional, unstructured-grid problems, but it serves as a good approximation (See Fletcher [1] for details). For the horizontal stability, then, this requires that $\nu_H \geq u_{max}^2\Delta t/2 = 0.025$ m^2 s^{-1} (`suntans.dat: nuH = 0.025`). Likewise, for vertical stability, $\nu = 0.016$ m^2 s^{-1} (`suntans.dat: nu = 0.016`).

The results can be viewed with the `sunplot` gui from the main source directory with

```
./sunplot --datadir=examples/lockexchange/data
```

This will open up a planview of the one-dimensional grid of equilateral triangles, showing that there are 26 (1+`nsteps`/`ntout`) time steps to plot. To view the profile plot, depress the Profile button with the middle mouse button. The last time step of the velocity vectors along with the salinity field can be viewed with the following buttons:

1. Right mouse on $-->$: To get to last time step.
2. Left mouse on Vectors : Turn on velocity vectors.
3. 3× Right mouse on Vectors : Increase vector length by a factor of 8.
4. 2× Left mouse on $>$: Increase `iskip` to 3 to plot every third vector for clarity.

The display should appear as it does in Figure 10.

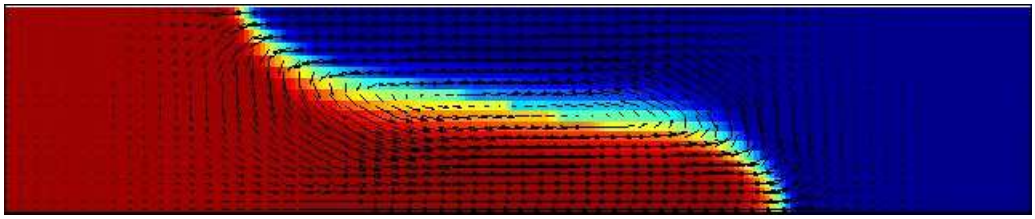

Figure 10: Sunplot display of the lock exchange example after 500 time steps.

6.5 Boundary condition example

This example is in the `examples/boundaries` directory and it simulates a simplified river plume in order to demonstrate the use of velocity as well as scalar boundary conditions at the inflow and the outflow.

6.5.1 Grid

The two-dimensional grid of equilateral triangles is created with the m-file `twodgrid.m`, which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

As shown in Figure 11, it is 3 km long by 1 km wide with a total of 1215 equilateral triangles with edge lengths of 75 m. The eastern and southern boundaries are marked as type 2 edges. These will be specified in `boundaries.c`. The depth is 10 m and is specified in `initialization.c` as

```
return 10;
```

There are 10 vertical levels (`suntans.dat: Nkmax = 10`).

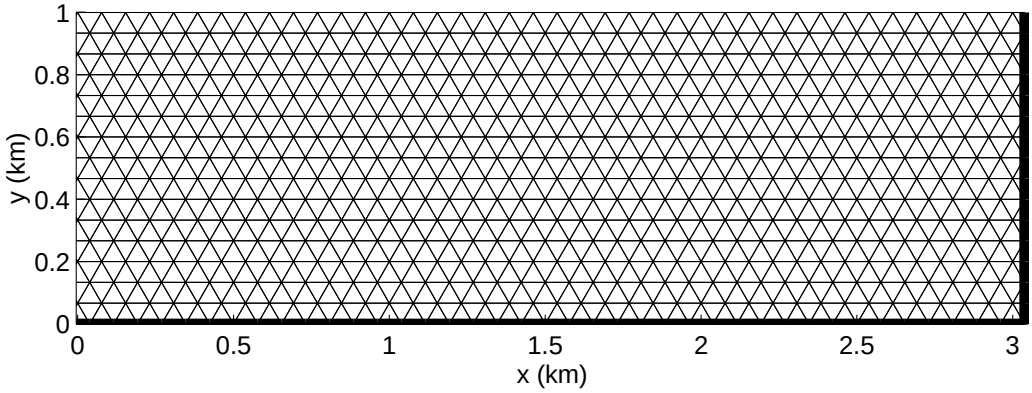

Figure 11: Two-dimensional planview of the river plume grid, showing the boundary edges of type 2 in bold.

6.5.2 Initial conditions

As specified in `initialization.c`, the initial conditions are straightforward in that all quantities are initialized to zero except the temperature field, which is set to 1 everywhere. It is treated as a passive tracer when $\gamma = 1$ (`suntans.dat: gamma = 1`).

6.5.3 Boundary conditions

Eastern Boundary: open
The eastern boundary fluxes normal to the faces are specified while values used for advection of momentum and scalars are upwinded

Boundary fluxes
As specified in `boundaries.c`, the eastern boundary is an open boundary and employs the linearized boundary condition on the velocity with

$$u_b = -h_b\sqrt{\frac{g}{d}},.$$

where h_b is the free-surface height, g is gravitational acceleration, and d is the depth, and the minus sign implies flux out of the domain when $h_b > 0$. This is enforced in the `OpenBoundaryFluxes` function with the lines

```
if(grid->yv[ib]>50) {
   for(k=grid->etop[j];k<grid->Nke[j];k++)
     ub[j][k]=-phys->h[ib]*sqrt(prop->grav/grid->dv[ib]);
...
```

The `ib` index is the index of the cell adjacent to the boundary, and the if-statement is required to distinguish this eastern outflow boundary from the southern boundary. Note that because the edge lengths are $\Delta x = 75$ m, this implies that the distance between the Voronoi points and the boundary edges is $\Delta x/(2\sqrt{3}) = 21.7$ m. Therefore, we know that if the location of the boundary Voronoi point `ib` adjacent to a boundary edge `j` is greater than 50 m, it must be the eastern boundary.

Boundary velocities
At the eastern open boundary, the boundary velocities which are used for advection of momentum out of the domain are given by the upwind velocity components. This is specified in the `BoundaryVelocities` function with the lines

```
if(grid->yv[ib]>50) {
   for(k=grid->etop[j];k<grid->Nke[j];k++) {
      phys->boundary_u[jptr-grid->edgedist[2]][k]=
                                   phys->uc[ib][k];
      phys->boundary_v[jptr-grid->edgedist[2]][k]=
                                   phys->vc[ib][k];
      phys->boundary_w[jptr-grid->edgedist[2]][k]=
                             0.5*(phys->w[ib][k]+phys->w[ib][k+1]);
   }
```

Because the flux of momentum is calculated at the vertical centers of the vertical faces, the upwind vertical velocity is interpolated from the upper (k) and lower (k+1) faces of the kth cell.

Boundary scalars
The outflow conditions on the scalars are imposed by specified the upwind scalar quantity at the outflow face. This is set in the function `BoundaryScalars` with the lines

```
for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
   phys->boundary_T[jptr-grid->edgedist[2]][k]=phys->T[ib][k];
```

```
    phys->boundary_s[jptr-grid->edgedist[2]][k]=phys->s[ib][k];
}
```

Southern boundary: specified
All quantities at the southern boundary are specified. Since this example simulates a river inflow, the velocity and scalar fields are specified along only part of the southern boundary, while the rest of the boundary is kept at zero inflow to represent a solid boundary. This requires if-statements to determine which of the southern boundary edges fall between the extent of the inflowing river. An alternative is to set the markers of the southern boundary edges that fall within the boundaries of the river to 2, and the rest to 1.

Boundary fluxes Because the velocity components at the southern boundary are specified, the incoming boundary flux is computed in `OpenBoundaryFluxes` using the velocity components specified in the `BoundaryVelocities` function, but only within the extent of the 300 m wide river, which is defined for $900 < x < 1200$ m (4 cell faces), as in:

```
if(grid->xv[ib]>900&&grid->xv[ib]<1200)
  for(k=grid->etop[j];k<grid->Nke[j];k++)
    ub[j][k]=
      phys->boundary_u[jptr-grid->edgedist[2]][k]*grid->n1[j]+
      phys->boundary_v[jptr-grid->edgedist[2]][k]*grid->n2[j];
```

Since `boundary_u` and `boundary_v` store the inflow velocity components, then this is setting the flux at the inflow to the normal component of the velocity at the inflow, i.e.

$$U_{inflow} = \mathbf{u}_{inflow} \cdot \mathbf{n}_{inflow} \,,$$

where $\mathbf{n}_{inflow}$ is, by convention, the inward pointing normal at the boundary face.

Boundary velocities The north-south velocity is specified at the inflow for $900 < x < 1200$ over a specified depth, which represents the river inflow, such that

$$\begin{aligned} u_{inflow} &= 0\,, \\ v_{inflow} &= \begin{cases} \text{amp} & z > -D_r \\ 0 & \text{otherwise}\,, \end{cases} \\ w_{inflow} &= 0\,. \end{aligned}$$

where amp is specified in `suntans.dat` as 0.01 m s^{-1}, and $D_r = 3$ m is the inflow depth. In `boundaries.c`, this inflow is implemented with

```
if(grid->xv[ib]>900 && grid->xv[ib]<1200) {
  z=0;
  for(k=grid->etop[j];k<grid->Nke[j];k++) {
    z-=0.5*grid->dzz[ib][k];
    if(z>-3.0)
      phys->boundary_v[jptr-grid->edgedist[2]][k]=prop->amp;
```

```
    z-=0.5*grid->dzz[ib][k];
  }
}
```

All other components are set to zero at the beginning of the main loop in the function `BoundaryVelocities` with

```
for(k=grid->etop[j];k<grid->Nke[j];k++) {
  phys->boundary_u[jptr-grid->edgedist[2]][k]=0;
  phys->boundary_v[jptr-grid->edgedist[2]][k]=0;
  phys->boundary_w[jptr-grid->edgedist[2]][k]=0;
}
```

Boundary scalars The inflow boundary condition for the salinity (or density, when $\beta = 1$) field represents an inflow of fresh water with a density anomaly of $\Delta\rho/\rho_0 = -10^{-4}$, with a surface depth of $D_r = 3$ m, such that

$$\rho_{inflow} = \begin{cases} \Delta\rho & z > -D_r \\ 0 & \text{otherwise}\,. \end{cases}$$

The inflow boundary condition for temperature is a no-gradient condition, and as a result the temperature just outside the boundary is equal to that just inside the boundary. These boundary conditions for salinity (density) and temperature are given in `boundaries.c` as

```
if(grid->yv[ib]<50)
  if(grid->xv[ib]>1000 && grid->xv[ib]<1200)
    for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
      z-=0.5*grid->dzz[ib][k];
      if(z>-3.0) {
        phys->boundary_T[jptr-grid->edgedist[2]][k]=phys->T[ib][k];
        phys->boundary_s[jptr-grid->edgedist[2]][k]=-0.0001;
      } else {
        phys->boundary_T[jptr-grid->edgedist[2]][k]=phys->T[ib][k];
        phys->boundary_s[jptr-grid->edgedist[2]][k]=0;
      }
      z-=0.5*grid->dzz[ib][k];
    }
```

6.5.4 Running the test

Upon typing `make test`, the simulation runs for a total of 3000 time steps (`suntans.dat: nsteps = 3000`) with a time step of 60 s (`suntans.dat: dt = 60`), outputting data every 120 time steps (`suntans.dat: ntout = 120`). While the data is running you can view the results from the main source directory with

```
./sunplot --datadir=./examples/boundaries/data
```

As can be seen from the results, the river plume forms a bulge as well as a coastal current resulting from the nonzero Coriolis parameter of $f = 5\times10^{-4}$ (`suntans.dat: Coriolis_f = 5e-4`). After 3000 time steps, the density anomaly at the upper layer is depicted in Figure 12. This figure was obtained using the m-file `sunsurf.m`, which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

If you are running this m-file from the `examples/boundaries` directory, then the plot in Figure 12 was obtained with the command

```
timestep = 26;
klevel = 1;
processor = 0;
sunsurf('s','.../data',timestep,klevel,processor);
```

Note that `sunsurf.m` requires `unsurf.m`.

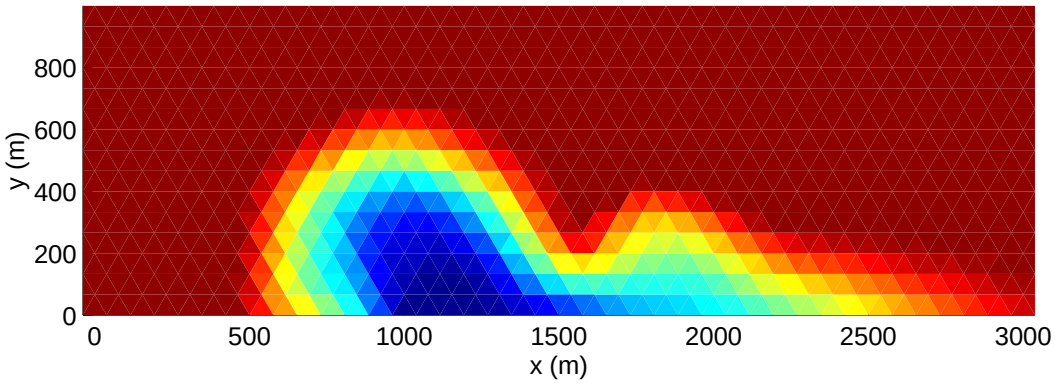

Figure 12: River plume example showing the salinity (density) field after 3000 time steps.

6.6 Cavity flow

The cavity flow example (`examples/cavity`) provides a means to test advection of momentum, no-slip boundary conditions, and rigid lid conditions. There are two examples, `testXY` and `testXZ` which comprise a cavity flow simulation in the XY and XZ planes, respectively. The `testXY` example demonstrates use of a stretched grid and no-slip boundary conditions. The `testXZ` example shows usage of the no-slip boundary condition as a means to force flows within the domain and highlights the utility of a rigid lid computation. These cases provide a starting point for examination of the effect of numerical methods employed on balances between the nonlinear advection of momentum, the pressure gradient, and viscous dissipation.

6.6.1 Grid

The grid for the `testXY` case was generated with Rusty Holleman's TOM gridding solver with $\Delta x \approx 0.01$ on the outer boundary and $\Delta x \approx 0.1$ on the inner part of the domain

for a total of 4016 cells with a single layer (`suntans.dat: Nkmax = 1`). The `testXZ` grid was derived from a 1D row of equilateral grid cells with right-angle triangles included on both sides so that the enforced boundary condition would be normal to the flow from the prescribed driven lid in the z-direction on the left side of the domain. There are 128 cells in the horizontal and 128 cells in the vertical (`suntans.dat: Nkmax = 128`). Note that due to the coarseness of the grids, there is some deviation from the results in Zang *et al.* [6].

6.6.2 Initial conditions

Flows within the domain are intially quiescent with depths of 1.0 and free surface heights of 0. The simulation is for pure water with constant density and viscosity.

6.6.3 Boundary conditions

Boundary conditions for the `testXY` case are specified by ensuring that all boundary markers are no-slip boundary conditions of type `4`. The `testXZ` case is somewhat more complicated, as the western and eastern-most edge is of type `4` with the channel sides parallel to the flow closed with type `1`. No-slip top and bottom boundary conditions are specified by setting `CdT -1` and `CdB -1` in `suntans.dat`. Deflection of the free surface, as would be expected to be developed in response to flows, is controlled via the rigid lid approximation by setting `grav=9.81e6` in `suntans.dat`.

For the `testXY` case all boundary velocities are 0 except the lid as specified in BoundaryVelocities() of `boundariesXY.c` via

```
for(jptr=grid->edgedist[4];jptr<grid->edgedist[5];jptr++) {
  j = grid->edgep[jptr];
  ib=grid->grad[2*j];
  boundary_index = jptr-grid->edgedist[2];

  for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
    if(grid->ye[j]<0.000010)
      phys->boundary_u[boundary_index][k]= 1.0;
    else
     phys->boundary_u[boundary_index][k] = 0.0;
     phys->boundary_v[boundary_index][k] = 0.0;
     phys->boundary_w[boundary_index][k] = 0.0;
  }
}
```

Observe that this function specifies the boundary value for each edge of type `4`, where the lid (located at the southern end of the domain) is driven by a constant flow towards the east.

The boundary conditions for `testXZ` are somewhat similar, but have the driven lid on the western most end of the domain with flow towards the south. Their specification in BoundaryVelocities() in `boundariesXZ.c` follows.

```
for(jptr=grid->edgedist[4];jptr<grid->edgedist[5];jptr++) {
```

```
    j = grid->edgep[jptr];
    ib=grid->grad[2*j];
    boundary_index = jptr-grid->edgedist[2];

    for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
      if(grid->xe[j]<0.5)
        phys->boundary_w[boundary_index][k]= -1.0;
      else
        phys->boundary_w[boundary_index][k]=0.0;
      phys->boundary_v[boundary_index][k] = 0.0;
      phys->boundary_u[boundary_index][k] = 0.0;
    }
  }
```

The lid velocity for both cases is unity to allow easy specification of Reynolds numbers via ν.

6.6.4 Running the test

This test case is run with the command

```
make test
```

which runs the `testXY` and `testXZ` cases in turn. These cases can be individually run with the commands

```
make testXY
```

and

```
make testXZ
```

Central differencing for advection of momentum (`suntans.dat: nonlinear = 2`) is employed and nonhydrostatic pressure used for the `testXZ` case (`suntans.dat: nonhydrostatic = 1`). Small time steps of `0.001` and `0.005` are employed to resolve eddies over a total time of `40`, corresponding to a rough approximation for steady state. Viscosity controls the Reynolds number and is set to yield $Re = 3200$ (`suntans.dat: nu = 0.0003125` and `suntans.dat: nu_H = 0.0003125`) for comparisons with Zang *et al.* [6]. Results from the simulation can be compared with these literature data values via the Matlab scripts `CompareResultXY.m` and `CompareResultXZ.m`. A more resolved case than the examples (13,500 cells) is shown in Figure 13, where $L = 1$ is the square domain's length, $U = 1.0$ is lid velocity, and u and v are the zonal and meridional velocities, respectively.

6.7 Internal waves

This example is found in the `examples/iwaves` directory, and it demonstrates the formation of internal wave beams at an idealized coastal shelf break in two dimensions x-z. The barotropic M2 tide is forced at the western boundary, forcing tidal currents over the break and thus generating internal wave beams.

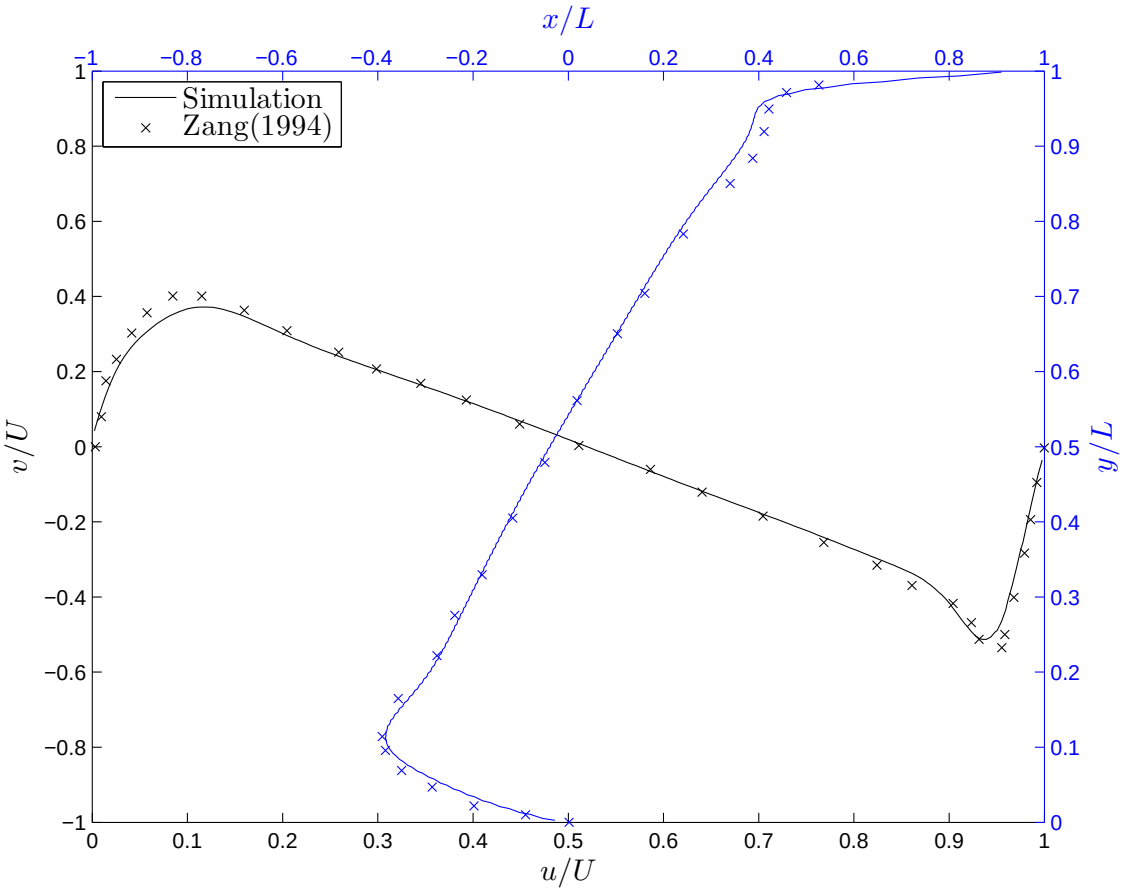

Figure 13: Velocity profiles for refined XY cavity flow (lines) compared with literature data from Zang *et al.* [6](squares)

6.7.1 Grid

The grid in this example is obtained with the `onedgrid.m` m-file which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

In this case, the one-dimensional grid contains 100 cells in the horizontal and 100 in the vertical, and the domain is 100 km long with a maximum depth of 3000 m. All edges are of type 1 except for the western boundary edge, which is of type 2. The shelf slope geometry is given by

$$d(x) = \begin{cases} D_0 & x \leq x_{mid} - L_s/2 \\ D_s & x > x_{mid} + L_s/2 \\ D_0 - (D_0 - D_s)\left[(x - x_{mid})/L_s + \frac{1}{2}\right] & \text{otherwise}\,. \end{cases}$$

where

$L_s = 20$ km: Horizontal extent of slope.

$x_{mid} = 65$ km: Center of slope.

$D_0 = 3000$ m: Maximum depth.

$D_s = 500$ m: Shelf depth.

This depth profile is specified in `boundaries.c` in the function `ReturnDepth` with

```
Ls = 20000;
xmid = 65000;
D0 = 3000;
Ds = 500;
if(x<=xmid-Ls/2)
  return D0;
else if(x>xmid+Ls/2)
  return Ds;
else
  return D0-(D0-Ds)*((x-xmid)/Ls+0.5);
```

As with all the present examples, in order for this depth to be specified, the `IntDepth` parameter must be set to 0 in the `suntans.dat` parameter file, otherwise, depth data must be supplied in the file specified by `depth` in `suntans.dat`. As shown in Figure 14, this grid is stretched in the vertical in order to provide extra resolution at the top boundary. This is done by specifying a positive stretching factor (as opposed to a negative stretching factor for the bottom boundary layer, as in Section 6.4) of $r = 1.025$ (`suntans.dat: rstretch = 1.025`), which causes each grid layer to be 1.025 times thicker than the layer **above** it.

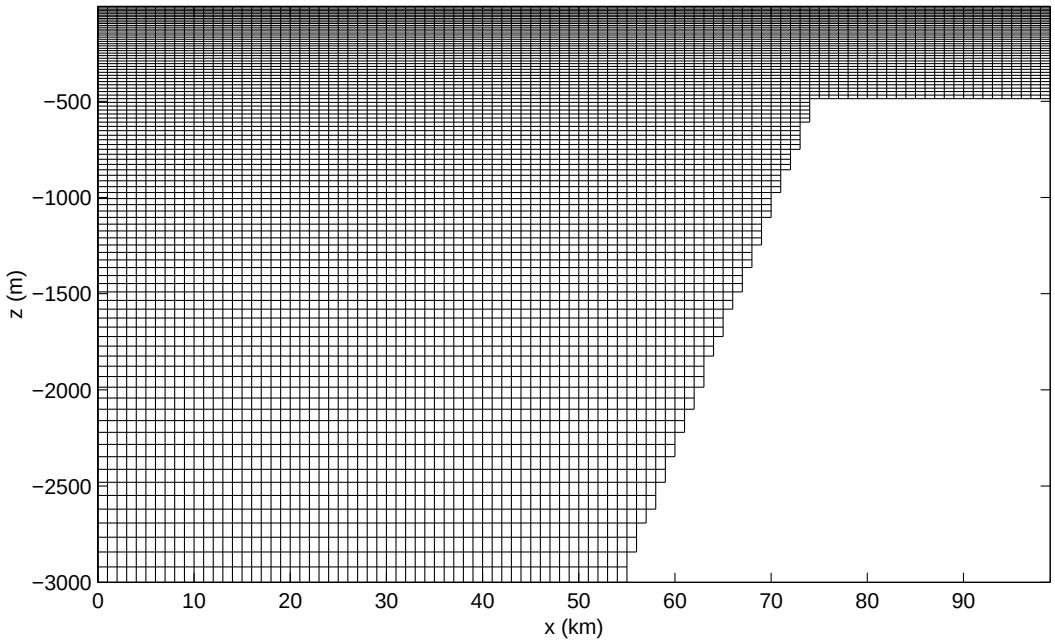

Figure 14: Depiction of the bathymetry and vertically stretched grid for the internal waves problem. Minimum grid spacing: 6.94 m, maximum: 79.94 m.

6.7.2 Initial conditions

The initial conditions for the internal wave problem, as specified in `initialization.c`, are quiescent velocity field, zero free-surface, and constant temperature field (the passive tracer) of $T = 1$, with an initial salinity field (or density, since `beta=1` in `suntans.dat`) of

$$s(z) = \begin{cases} \Delta s\,(-z)^{\alpha_s} & z < -D_{pycnocline}\,, \\ \Delta s\,(D_{pycnocline})^{\alpha_s} & \text{otherwise}\,, \end{cases}$$

where $\Delta s = 0.024$, $\alpha_s = 0.0187$, and $D_{pycnocline} = 20$. These parameters have been chosen to create a typical U.S. west coast density field. This salinity (density) profile is implemented in the function `ReturnSalinity` in `initialization.c` as

```
deltaS = 0.024;
alphaS = 0.0187;
D_pycnocline = 20;

if(z<-D_pycnocline)
  return deltaS*pow(-z,0.0187);
else
  return deltaS*pow(D_pycnocline,0.0187);
```

6.7.3 Boundary conditions

Boundary conditions are gradient-free on the temperature and salinity fields at the western boundary. This is implemented in the function `BoundaryScalars` in the file `boundaries.c` as

```
for(k=grid->ctop[ib];k<grid->Nk[ib];k++) {
  phys->boundary_T[jptr-grid->edgedist[2]][k]=phys->T[ib][k];
  phys->boundary_s[jptr-grid->edgedist[2]][k]=phys->s[ib][k];
}
```

where `ib` is the index of the Voronoi point adjacent to the boundary edge `j`.

For the velocity field, the x-direction velocity is specified at the western boundary as a sinusoidally varying barotropic velocity with a M_2 tidal period of 12.42 hours, and the other components are zero. This is implemented in the function `BoundaryVelocities` in the file `boundaries.c` as

```
for(k=grid->etop[j];k<grid->Nke[j];k++) {
  phys->boundary_u[jptr-grid->edgedist[2]][k]=
        prop->amp*sin(prop->omega*prop->rtime);
  phys->boundary_v[jptr-grid->edgedist[2]][k]=0;
  phys->boundary_w[jptr-grid->edgedist[2]][k]=0;
}
```

The variables `amp` and `omega` are set in `suntans.dat` (`amp = 4` mm s^{-1} and `omega = 1.4026e-4` rad s^{-1}), and `rtime` is the simulation time, in seconds. These velocities are then used in the `OpenBoundaryFluxes` function to compute the velocity normal to the boundary face with

```
for(k=grid->etop[j];k<grid->Nke[j];k++)
  ub[j][k]=phys->boundary_u[jptr-grid->edgedist[2]][k]*grid->n1[j]+
           phys->boundary_v[jptr-grid->edgedist[2]][k]*grid->n2[j];
```

Since there is only one boundary, no if-statements are required to determine if this is an open or closed boundary condition, such as in Section 6.5. More complex partially-clamped boundary conditions can be implemented that allow the internal wave field to exit the domain while still forcing the barotropic velocity field, but this is beyond the scope of this simplified example.

6.7.4 Running the test

The test case is run at the command line with the command

```
make test
```

which runs the simulation for 3000 time steps (`suntans.dat: nsteps = 3000`) with a time step size of 29.808 s (`suntans.dat: dt = 29.808`) for a total of 2 M_2 tidal periods. This test case uses the sponge layer which is hardwired into SUNTANS. The parameters are set to `sponge_distance = 5000` and `sponge_decay = 7200` (for details see Section 7).

The results can be viewed with the `sunplot` gui from the main source directory with

```
./sunplot --datadir=examples/iwaves/data
```

This brings up a planview of the one-dimensional grid of equilateral triangles. To display the profile plot of the results, press the Profile button with the middle mouse button and then the Axis button with the left mouse button so that the axes fill the plot window. This will display the salinity (density) field at the first time step, as shown in Figure 15. Plotting the u-velocity at data steps 11 and 21 will display the horizontal velocity field after one and two periods of forcing, respectively, as shown in Figures 16 and 17. Note that even with the use of the sponge layer, some internal wave energy is reflected back from the left boundary and affects the internal wave field shown in Figure 17. In addition to using `sunplot`, printouts for these plots can be obtained with the `plotslice.m` m-file which can be downloaded from

http://suntans.stanford.edu/downloads_stanford.

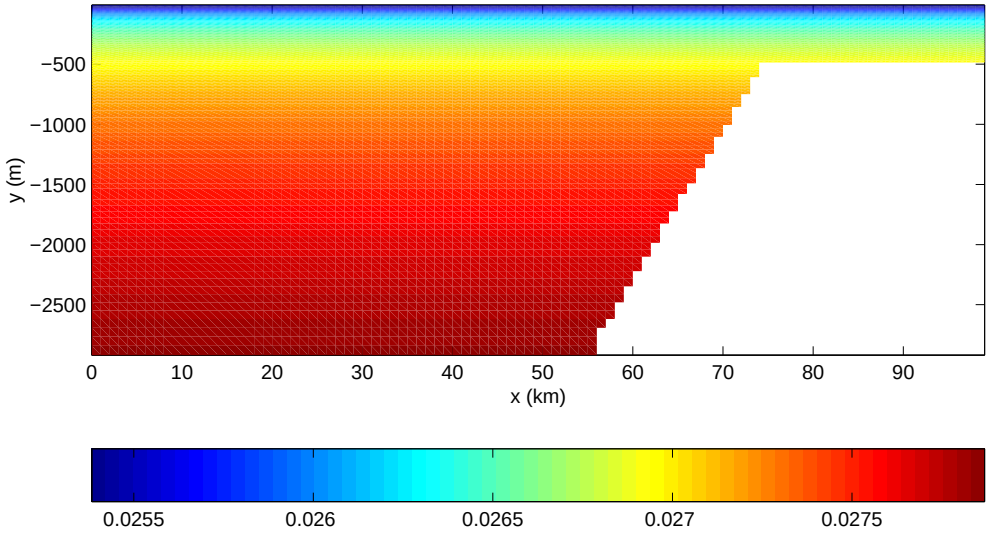

Figure 15: The initial salinity (density) field for the internal wave test case.

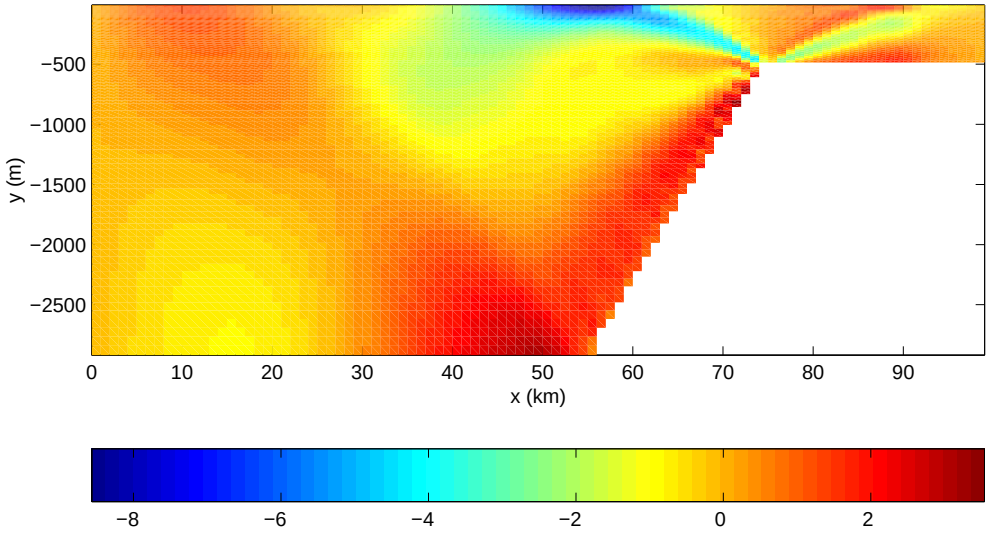

Figure 16: The horizontal velocity field (mm s^{-1}) after one M_2 tidal period for the internal wave test case.

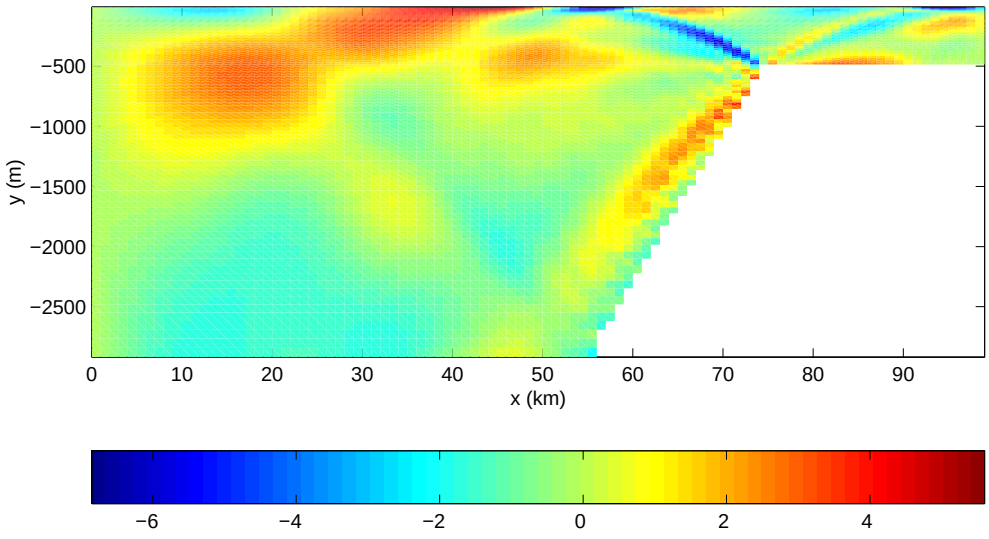

Figure 17: The horizontal velocity field (mm s^{-1}) after two M_2 tidal periods for the internal wave test case.

7 Parameter listing in suntans.dat

The main SUNTANS input file is called `suntans.dat` and contains three columns. The first column contains the name of the variable used in the program while the second column contains its value, and the information after that value is ignored. Lines beginning with `#` are ignored, while **empty lines will cause the program to crash**. The SUNTANS executable looks for this file in the directory specified by `--datadir=`. If this is not specified, then it looks for it in the same directory as the `sun` executable.

7.1 Physical/Computational parameters

These parameters define values used in SUNTANS. The syntax of the line in `suntans.dat` is given by

```
variablename  variablevalue   # Description of variable
```

When this file is read, the effect is equivalent to `variablename=variablevalue`. This file cannot have empty lines, but the spacing between variablename and value is arbitrary, as long as it does not contain a new line. Only the first variable is read before the `#` sign. That is, old variable values can exist to the right of the variable values being used and they are neglected. The following line is therefore legal:

```
dt    0.025  0.05 (unstable)  0.0125 (10/12/04)  # Time step size
```

This will have the effect of `dt=0.025` and neglecting the rest of the line after `0.025`.

7.1.1 Nkmax: $N_{kmax} \geq 1$

This is the number of cells in the vertical and is used when the -g flag is specified in order to build the vertical grid levels. It must be greater than or equal to 1.

7.1.2 stairstep: Boolean

If stairstep is 1 then the vertical grid spacing does not change in the horizontal. If stairstep is 0, then partial-stepping is employed and the height of the bottom cell is set such that the bottom face of the cell coincides with the actual value of the depth. Otherwise (when stairstep=1), the bottom face of of the lowermost cell is always greater than the actual depth.

7.1.3 rstretch: $-1.1 < r < -1$ or $1 < r < +1.1$

Employ vertical grid stretching such that the distribution of vertical grid spacings is given by

$$\Delta z_k = r\, \Delta z_{k+1}\,,$$

where $0 \leq$ k $<$ Nkmax. If $r < 0$ then the grid is refined near the bottom, otherwise it is refined at the free-surface.

7.1.4 CorrectVoronoi: Boolean

Typical triangulations will always contain degenerate triangles in which one of the angles are close to or greater than 90 degrees. For near-right neighboring triangles, the distance between their Voronoi points can be quite small (zero when two right triangles abut one another), while for obtuse triangles, the Voronoi point lies outside the triangle. This can be corrected in one of two ways.

In the first method, CorrectVoronoi is 1, and the Voronoi points are corrected if they are too close to each other relative to the distance between the centroids of the neighboring triangles. If triangle 1 and 2 have centroids defined by (xc_1, yc_1) and (xc_2, yc_2) and Voronoi points defined by (xv_1, yv_1) and (xv_2, yv_2), then the distance between the centroids is given by

$$D_c^2 = (xc_2 - xc_1)^2 + (yc_2 - yc_1)^2 \,,$$

and the distance between the Voronoi points is given by

$$D_v^2 = (xv_2 - xv_1)^2 + (yv_2 - yv_1)^2 \,.$$

The Voronoi points are corrected if $D_g/D_c < V_r$, where V_r is the `VoronoiRatio` parameter, and if this is the case, they are updated with

$$\begin{aligned} xv_1 &= xc + V_r(xc_1 - xc) \,, \\ yv_1 &= yc + V_r(yc_1 - yc) \,, \\ xv_2 &= xc + V_r(xc_2 - xc) \,, \\ yv_2 &= yc + V_r(yc_2 - yc) \,, \end{aligned}$$

where $xc = (xc_1 + xc_2)/2$ and $yc = (yc_1 + yc_2)/2$. Using this correction methodology, setting $V_r = 1$ moves the Voronoi points to the centroids of the cells.

In the second method, CorrectVoronoi is set to -1 and triangles with angles greater than or equal to the angle defined by VoronoiRatio are corrected by moving the Voronoi point to the triangle centroid. While the centroid is guaranteed to be inside the triangle and the distance between centroids is on the same order as the triangle edge lengths, the disadvantage is that the Voronoi edge connecting the Voronoi points is no longer perpendicular to the Delaunay edges. This reduces the accuracy of the gradients defined by differences between values at Voronoi points because it is assumed that the Voronoi-Delaunay intersection is orthogonal.

7.1.5 VoronoiRatio

Adjustment factor for degenerate triangles. Use of this parameter depends on the value of CorrectVoronoi defined above:

if CorrectVoronoi=1, $0 \leq$ VoronoiRatio ≤ 1: Voronoi points are moved by this amount towards the triangle centroids.

if CorrectVoronoi=-1 $0 \leq$ VoronoiRatio ≤ 90: Move Voronoi points to centroids for triangles with angles greater than or equal to VoronoiRatio.

if CorrectVoronoi=0:no correction is performed.

7.1.6 vertgridcorrect: Boolean

For an equispaced vertical grid, the vertical grid spacing is constant and is given by $\Delta z = D_{max}/N_{kmax}$, where D_{max} is the maximum depth in the domain. In the rare case for which the minimum depth, D_{min} in the domain is less than Δz, there are not enough grid points to resolve the region with the smallest depth. If this is the case and vertgridcorrect is set to 1, then the number of vertical grid cells is adjusted such that $\Delta z = D_{min}$, such that $N_{kmax} = D_{max}/D_{min}$.

7.1.7 IntDepth: Boolean

If set to 1, then the depths at the Voronoi points are interpolated from the depth specified by the depth file (see below). Otherwise, the depths are specified in the file `initialization.c`.

7.1.8 dzsmall: No longer used

7.1.9 scaledepth: Boolean

For debugging purposes. If set to 1, scales the depth by scaledepthfactor (see below).

7.1.10 scaledepthfactor

If scaledepth=1, this scales the depth by the given amount and hence can be used to reduce the depth and hence the surface gravity wave speed by a desired amount.

7.1.11 thetaramptime: $\tau_\theta > 0$

This is used to damp out transient barotropic motions by initializing the value of θ in the free-surface solver to 1 and ramping it down to the value of θ specified below. The τ_θ parameter determines the timescale, in seconds, over which this is damped, such that, if the value of θ specified below is given by θ_0, then the value of θ that is used in the calculations is given by

$$\theta(t) \;=\; \theta_0 + (1-\theta_0)\exp\left(-\frac{t}{\tau_\theta}\right) .$$

7.1.12 beta: $\beta \geq 0$

Expansivity of salt. This is used in the gravity term on the right hand side of the Navier-Stokes equations in the form

$$-g\beta\,(s - s_0) \ ,$$

where $s(x, y, z, t)$ is the salinity and $s_0(z)$ is the initial salinity. If $\beta = 0$ advection of salinity is not computed.

7.1.13 theta: $0 < \theta \leq 1$

Implicitness parameter for the free-surface solver and the vertial diffusion terms.

- $\theta = 0$: Fully explicit (unstable)
- $\theta = 0.5$: Crank-Nicolson (neutrally stable in the linear sense)
- $\theta = 1.0$: Fully implicit
- $\theta = 0.55$: For practical simulations

7.1.14 thetaS: $0 \leq \theta_S \leq 1$

Implicitness parameter for vertical scalar advection and diffusion. Note that this value of thetaS does not necessarily share the same stability properties as the free-surface solver since stability for scalar transport is governed by the horizontal Courant number even when $\theta_S > 0.5$.

- $\theta_S = 0$: Fully explicit
- $\theta_S = 0.5$: Crank-Nicolson (Ensures no vertical Courant limitation for advection or diffusion)
- $\theta_S = 1.0$: Fully implicit (same as above but first-order accurate)
- $\theta_S = \theta$: To guarantee consistency with continuity

7.1.15 thetaB: Not used

7.1.16 kappa_s: $\kappa_s \geq 0$

Laminar vertical diffusivity of salt in units of $\mathrm{m}^2\ \mathrm{s}^{-1}$. Does not affect stability.

7.1.17 kappa_sH: $\kappa_{sH} \geq 0$

Laminar horizontal diffusivity of salt in units of $\mathrm{m}^2\ \mathrm{s}^{-1}$. For stability

$$\frac{\kappa_{sH}\Delta t}{\min(D_g^2)} < \frac{1}{2},$$

where D_g is the distance between Voronoi points.

7.1.18 gamma: γ: Boolean

Temperature is implemented as a passive scalar. If gamma is set to 0 then scalar transport of T is not computed.

7.1.19 kappa_T: $\kappa_T \geq 0$

Same as *kappa_s* but for T.

7.1.20 kappa_TH: $\nu_{TH} \geq 0$

Same as *kappa_TH* but for T.

7.1.21 nu: $\nu \geq 0$

Vertical diffusivity of momentum in units of $m^2\ s^{-1}$. Does not affect stability unless central-differencing is employed for momentum advection (see Section 7.1.51).

7.1.22 nu_H: $\nu_H \geq 0$

Horizontal diffusivity of momentum in units of $m^2\ s^{-1}$. See 7.1.30 and 7.1.51 for details on stability.

7.1.23 tau_T: τ_T

Parameter used in boundaries.c to employ a surface wind stress in units of $m^2\ s^{-2}$, i.e. tau_T $= (u^*)^2$. Note that if the surface wind stress due to a wind speed at 10 m is given by

$$\tau_{\text{wind}} = \rho_{\text{air}} C_d U_{10}^2 \,,$$

then to obtain the value for tau_T, one must use

$$\text{tau\_T} = \frac{\tau_{\text{wind}}}{\rho_0} \,.$$

7.1.24 z0T: z_{0T}

Lid roughness used to calculate drag at the lid (in meters). See CdT.

7.1.25 z0B: z_{0B}

Bottom roughness used to calculate drag at the bottom boundary layer (in meters). See CdB.

7.1.26 CdT: C_{dT}

If the roughness at the lid is zero (i.e. z0T=0) then the drag coefficient at the lid is given by this constant, otherwise the drag coefficient is calculated with the rough wall relationship. Use of no-slip boundary conditions for the top cell is specified by setting CdT=-1.

$$C_{dT} = \left[\frac{1}{\kappa} \ln \left(\frac{1}{2} \frac{\Delta z_{top}}{z_{0T}} \right) \right]^{-2} \,.$$

where Δz_{top} is the thickness of the top cell (in meters) and $\kappa = 0.4$ is the von Karman constant.

7.1.27 CdB: C_{dB}

If the roughness at the bottom is zero (i.e. z_{0B}=0) then the drag coefficient at the bottom is given by this constant, otherwise the drag coefficient is calculated with the rough wall relationship. Use of no-slip boundary conditions for the bottom cell is specified by setting CdB=-1.

$$C_{dB} = \left[\ln \left(\frac{1}{2} \frac{\Delta z_{bot}}{z_{0B}} \right) \right]^{-2} ,$$

where Δz_{top} is the thickness of the bottom cell (in meters) and $\kappa = 0.4$ is the von Karman constant.

7.1.28 CdW: C_{dW}

Constant drag coefficient of sidewalls. Drag on sidewalls is only computed if the wall does not extend up to the free surface. For example, in a compound channel, sidewall drag is only computed within the channel and not on the edges of the floodplain, as shown in Figure 18.

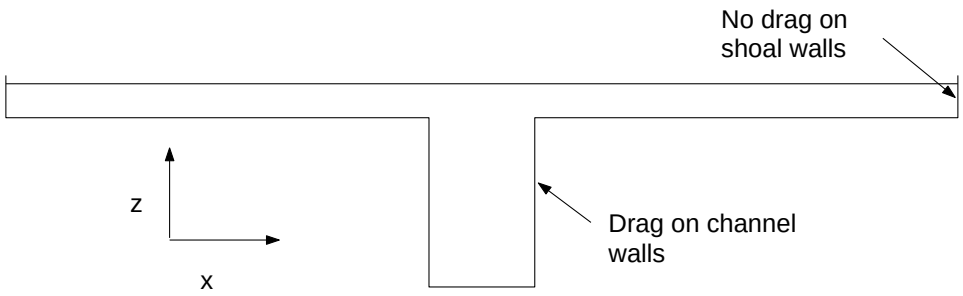

Figure 18: Example of a compound channel, for which sidewall drag is only computed in the channel and not on the shoals.

7.1.29 turbmodel: Boolean

If turbmodel is 1 then the Mellor-Yamada level 2.5 turbulence model is used, otherwise no turbulence model is employed.

7.1.30 dt: Δt

Time step. Assuming $\theta > 0.5$, stability is limited by

$$\Delta t < C \left[\frac{\min(D_g)}{\max(|U|)}, \frac{\min(\Delta z)}{\max(|w|)}, \frac{\min(D_g)}{\max(c_i)}, \frac{1}{2} \frac{\min(D_g)^2}{\max(\kappa_{sH})}, \frac{1}{2} \frac{\min(D_g)^2}{\max(\kappa_{TH})}, \frac{1}{2} \frac{\min(D_g)^2}{\max(\nu_H)} \right] ,$$

where c_i is the internal wave speed, C is the relevant Courant number, and D_g is the minimum Voronoi distance. Note that this assumes first-order upwind is used for advection of momentum. If central-differencing is used, stability is further constrained by the grid Peclet number (see Section 7.1.51 for details).

7.1.31 Cmax: $C_{max} > 0$

Maximum permissible Courant number for advection of momentum.

7.1.32 nsteps: $N_{steps} \geq 0$

Total number of time steps.

7.1.33 ntout: $N_{tout} > 0$

How often to output data. Since the first time step is always output, the total number of time steps that are output is then given by

$$1 + \text{floor}\left(\frac{N_{steps}}{N_{tout}}\right) .$$

7.1.34 ntprog: $0 \leq N_{tprog} \leq 100$

Report progress at $N_{tprog}\%$ intervals if $N_{tprog} > 0$, otherwise do not report progress. Either way, progress is reported only if the verbose flag is at least `-v`.

7.1.35 ntconserve: $0 \leq N_{tconserve} \leq N_{steps}$

How often to output conservative data, such as mass, volume, and potential energy, into the file specified by ConserveFile. Output should be edited to suit user's needs in the `OutputData` function in `phys.c`.

7.1.36 nonhydrostatic: Boolean

(1): Compute the nonhydrostatic pressure, (0): neglect the nonhydrostatic pressure and calculate the vertical velocity with continuity.

7.1.37 cgsolver: No longer used

7.1.38 maxiters: $I_{max,H} > 0$

Maximum number of iterations allowed of the free-surface conjugate-gradient solver before reporting nonconvergence. If the free-surface solver is not converging in less than 200 iterations then the problem is poorly conditioned.

7.1.39 qmaxiters: $I_{max,Q} > 0$

Maximum number of iterations allowed of the nonhydrostatic pressure conjugate-gradient solver before reporting nonconvergence.

7.1.40 qprecond: 0,1, or 2

Type of preconditioner to use for the nonhydrostatic pressure solver. For a typical environmental flow with an aspect ratio of $\mathcal{O}(10^2)$:

0 No preconditioner - 1000s of iterations to converge

1 Diagonal preconditioner - 100s of iterations to converge

2 Block-diagonal preconditioner - 10s of iterations to converge.

There is no significant difference among the preconditioners when the aspect ratio is roughly 1.

7.1.41 epsilon: $\epsilon_H > 0$

Tolerance for free-surface conjugate gradient solver. Default: $\epsilon_H = 10^{-10}$.

7.1.42 qepsilon: $\epsilon_Q > 0$

Tolerance for nonhydrostatic pressure conjugate gradient solver. Default: $\epsilon_Q = 10^{-5}$.

7.1.43 resnorm: Boolean

Whether or not to normalize the residual when computing tolerance criteria for the conjugate gradient solvers. At a given iteration, if the residual is given by $r = b - Ax$, then tolerance is evaluated with:

0 : error=$r^T r$

1 : error=$\frac{r^T r}{b^T b}$

7.1.44 relax: No longer used

7.1.45 amp

Amplitude parameter for use in `boundaries.c` in the form `prop->amp`.

7.1.46 omega

Frequency parameter for use in `boundaries.c` in the form `prop->omega`.

7.1.47 flux

Flux parameter for use in `boundaries.c` in the form `prop->flux`.

7.1.48 timescale

Timescale parameter for use in `boundaries.c` in the form `prop->timescale`.

7.1.49 volcheck: Boolean

(0): Do not check for volume conservation, (1): check for volume conservation and print out errors if volume is not conserved to within the tolerance defined by `CONSERVED` in `suntans.h`. Prints regardless of verbosity setting at command line.

7.1.50 masscheck: Boolean

(0): Do not check for mass conservation, (1): check for mass conservation and print out errors if volume is not conserved to within the tolerance defined by `CONSERVED` in `suntans.h`. Prints regardless of verbosity setting at command line.

7.1.51 nonlinear: 0,1, or 2

Advection scheme for momentum:

0 : No advection of momentum, i.e. $\mathbf{u} \cdot \nabla \mathbf{u} = 0$.

1 : First-order upwind for momentum.

2 : Second-order central for momentum. Note that diffusion must be present for stability. While it is difficult to perform an exact stability analysis for the unstructured-grid equations, the 1-d stability limitation requires that grid Peclet number according to

$$Pe_\Delta < \frac{2}{C},$$

where $Pe_\Delta = \frac{\max(|U|)\max(D_g)}{\min(\nu_H)}$ and $C = \frac{\max(|U|)\Delta t}{\max(D_g)}$, which translates into the requirement that the horizontal diffusion be restricted to

$$\nu_H > \frac{\max(|U|)^2 \Delta t}{2}.$$

Equivalently, the vertical diffusion is restricted by

$$\nu > \frac{\max(|w|)^2 \Delta t}{2}.$$

7.1.52 newcells: Boolean

Since the advection of momentum is not conservative in the upper cells, the velocity in the upper cells can be adjusted to ensure that

$$U_{top}^{n+1} \Delta z_{top}^{n+1} = U_{top}^{n} \Delta z_{top}^{n}.$$

This should be used with caution and results should be checked to make sure this does not alter them significantly. Otherwise the upper layer depth is too small relative ot the free surface motions.
(0): Do not adjust upper layer cells, (1): Adjust upper layer cells.

7.1.53 wetdry: Boolean

Different time-stepping schemes are used for horizontal scalar advection when wetting and drying occur (vertical advection always uses the θ method). To guarantee consistency with continuity, the θ method must be used for scalar advection. This is formally first-order accurate in time, so results are more accurate when second-order Adams-Bashforth is used if there is no wetting and drying. Therefore, the wetdry Boolean is set according to:
(0): No wetting and drying, employ AB2 for horizontal scalar advection
(1): Wetting and drying: employ θ-method for horizontal scalar advection.
Depending on the stratification at the surface and the grid size it is often desirable to employ wetdry = 1 so as to ensure consistency with continuity and avoid spurious oscillations in the temperature and salinity fields.

7.1.54 Coriolis_f: $f \geq 0$

Coriolis frequency $f = 2\Omega \sin(\phi)$, where ϕ is the latitude.

7.1.55 sponge_distance: $D_{sponge} \geq 0$

An example of how to implement a sponge layer is shown in `phys.c` in the form

$$\frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{\tau_{sponge}} \exp\left(-\frac{x}{D_{sponge}}\right),$$

where the τ_{sponge} timescale is defined by the value of sponge_decay. If $D_{sponge} = 0$ then no sponge layer is employed. **Note that this sponge layer assumes the boundary is located at $x = 0$ and that the sponge layer is roughly D_{sponge} m thick and extends in the positive x direction!**

7.1.56 sponge_decay: $\tau_{sponge} > 0$

Decay timescale for sponge layer (in seconds).

7.1.57 readSalinity: Boolean

(0): Vertical salinity distribution is specified in `initilization.c`
(1): Read vertical salinity distribution from the file specified by the InitSalinityFile parameter. This file must be in binary form and must contain N_{kmax} double precision values in top-to-bottom order.

7.1.58 readTemperature: Boolean

(0): Vertical temperature distribution is specified in `initilization.c`
(1): Read vertical temperature distribution from the file specified by the InitTemperatureFile parameter. This file must be in binary form and must contain N_{kmax} double precision values in top-to-bottom order.

7.2 Input/Output files

These parameters define the names of the input files. The syntax of the line in `suntans.dat` is given by

```
filespecifier  fileprefix   # Description of parameter
```

Global files do not contain the processor suffix, while per-processor files contain the processor number as the suffix. For example, in a four-processor run, there will be four i/o files relating to this specifier, i.e. fileprefix.0, fileprefix.1, fileprefix.2, and fileprefix.3. The file called fileprefix (with no suffix) contains information for all processors.

7.2.1 pslg: input

Planar straight line graph file. Does not contain the processor number suffix. See Section 3 for details.

7.2.2 points: input/output

Delaunay points file (N_p total points). Does not contain the processor number suffix. For details see Section 3.

Size: $N_p \times 3$

Type: ASCII

Format: (float)xp, (float)yp, (int)marker

7.2.3 edges: input/output

Edge connectivity file (N_e total edges). For details see Section 3.

Size: $N_e \times 5$

Type: ASCII

Format:

(int$\times$ 2)Indices to Delaunay points that make up endpoints

(int)Edge marker for boundary condition

(int$\times$ 2)Indices to Voronoi points on either side of edge

7.2.4 cells: input/output

Cell connectivity file (N_c total cells). For details see Section 3.

Size: $N_c \times 8$

Type: ASCII

Format:

(float× 2) Voronoi coordinates

(int × 3) Indices to points that make up vertics

(int × 3) Indices to neighboring cells

7.2.5 depth: input

Contains bathymetry data used to interpolate the depth onto the Voronoi points only if the IntDepth parameter is set to 1. Otherwise the depth is set in `initialization.c`. The spacing of the data is arbitrary since the interpolation routine searches for the nearest neighbors to perform the interpolations. Note that the absolute value of the depth is read in from this file, so SUNTANS does not distinguish between elevation and depression.

Size: Number of bathymetry data points × 3

Type: ASCII

Format:

- (float)x
- (float)y
- (float)depth

7.2.6 celldata: input/output

Contains cell-centered data for each processor. (N_c total cells)

Size: $N_c \times 17$

Type: ASCII

Format:

(float)Voronoi point x

(float)Voronoi point y

(float)Cell Area

(float)Voronoi depth

(int)Number of vertical levels

(3×int)Index to edge faces

(3×int)Index to cell neighbors

(3×int)Dot-product with outward normal on faces

(3×float)Distance from Voronoi point to normal on faces

7.2.7 edgedata: input/output

Contains edge-centered data for each processor. (N_e total edges)

Size: $N_e \times 13$
Type: ASCII

Format:

(float)Edge length

(float)Voronoi length

(float× 2)x-y components of unique normal

(float× 2)x-y coordinates of center of edge

(int)Number of edges in vertical

(int)Maximum number of neighboring cells in vertical

(int)Upwind cell neighbor (if $U > 0$)

(int)Downwind cell neighbor (if $U > 0$)

(int× 2) gradf pointers (see Section 3)

(int)Edge marker

7.2.8 vertspace: input/output

Contains vertical grid spacing in top-to-bottom ordering. Same for all processors.

Size: $N_{kmax} \times 1$
Type: ASCII

Format: (float)Grid spacing

7.2.9 topology: input/output

Contains topology information per processor. (N_c = number of cells, N_e = number of edges)

Size: Depends on processor topology
Type: ASCII

Format:

(int)Number of processors

(int)Number of neighboring processors (Numneighs)

(int×Numneighs)Processor neighbor list

For each neighboring processor:

(int)Number of cells to send (cellsend)

(int)Number of cells to receive (cellrecv)

(int)Number of edges to send (edgesend)
(int)Number of edges to receive (edgerecv)
(int× cellsend)Indices of cells to send
(int× cellrecv)Indices of cells to recv
(int× edgesend)Indices of edges to send
(int× edgerecv)Indices of edges to recv
(int×3)Indices for celldist array
(int×6)Indices for edgedist array
(int×N_c)Indices for cellp array
(int×N_e)Indices for edgep array

7.2.10 FreeSurfaceFile: output

Contains the free-surface data. (N_c = number of cells)

Size: $\left[1+\text{floor}\left(\frac{N_{steps}}{N_{tout}}\right)\right]\times N_c$
Type: binary

Format: (double×N_c) for each time step.

Matlab equivalent: h(1:Nc) for each time step.

7.2.11 HorizontalVelocityFile: output

Contains the three components of velocity at the Voronoi points. (N_c = number of cells)

Size: $\left[1+\text{floor}\left(\frac{N_{steps}}{N_{tout}}\right)\right]\times N_{kmax}\times 3\times N_c$
Type: binary

Format: (double×$N_{kmax}\times 3\times N_c$) for each time step.

Matlab equivalent: u(1:Nkmax,1:3,1:Nc) for each time step.

Note: empty cells (beneath z=-d) contain value of `EMPTY` defined in `suntans.h`.

7.2.12 VerticalVelocityFile: output

Contains the vertical velocity at the horizontal faces. (N_c = number of cells)

Size: $\left[1+\text{floor}\left(\frac{N_{steps}}{N_{tout}}\right)\right]\times(1+N_{kmax})\times N_c$
Type: binary

Format: (double×$(1+N_{kmax})\times N_c$) for each time step.

Matlab equivalent: w(1:Nkmax+1,1:Nc) for each time step.

Note: empty cells (beneath z=-d) contain value of `EMPTY` defined in `suntans.h`.

7.2.13 SalinityFile,BGSAlinityFile,TemperatureFile, PressureFile,VerticalGridFile,EddyViscosityFile, ScalarDiffusivityFile: outputs

Contain data at Voronoi points for the given quantity. (N_c = number of cells)

Size: $\left[1 + \text{floor}\left(\frac{N_{steps}}{N_{tout}}\right)\right] \times N_{kmax} \times N_c$

Type: binary

Format: (double$\times N_{kmax} \times N_c$) for each time step.

Matlab equivalent: s(1:Nkmax,1:Nc) for each time step.

Note: empty cells (beneath z=-d) contain value of `EMPTY` defined in `suntans.h`.

7.2.14 ConserveFile: output

File used to report conservative variables such as volume, mass, and energy. The format for this file is user-implemented in the `OutputData` function in `phys.c`.

7.2.15 ProgressFile: output

Short file containing a printout of the run progress in the form

```
On (int) of (int),t=(float) ((int)% Complete, (int) output)
```

to show total number of time steps complete, the current simulation time (t=...), the percentage completion, and the number of steps that have been output for viewing.

7.2.16 StoreFile: output

Saves data to be loaded for a restart run or to analyze data in existence at the time of a blow up.

Size: Depends on grid geometry since cells beneath the depth are not stored (unlike other individual data files).

Type: binary

Format:

(int)last time step

(double$\times N_c$)Free surface

(double$\times N_e \times N_{ke}$)AB2 term at n-1 for horizontal momentum

(double$\times N_c \times N_k$)AB2 term at n-1 for vertical momentum

(double$\times N_c \times N_k$)AB2 term at n-1 for salinity

(double$\times N_c \times N_k$)AB2 term at n-1 for temperature

(double×$N_c \times N_k$)AB2 term at n-1 for Mellory-Yamada 2.5 q when turbulence=1

(double×$N_c \times N_k$)AB2 term at n-1 for Mellory-Yamada 2.5 L when turbulence=1

(double×$N_c \times N_k$)Mellor-Yamada 2.5 q when turbulence=1

(double×$N_c \times N_k$)Mellor-Yamada 2.5 L when turbulence=1

(double×$N_c \times N_k$)Turbulent eddy-viscosity

(double×$N_c \times N_k$)Turbulent scalar-diffusivity

(double×$N_e \times N_{ke}$)Horizontal velocity

(double×$N_c \times (N_k + 1)$)Vertical velocity

(double×$N_c \times N_k$)Nonhydrostatic pressure

(double×$N_c \times N_k$)Salinity

(double×$N_c \times N_k$)Temperature

(double×$N_c \times N_k$)Background salinity

7.2.17 StartFile: input

Used to read in data for a restart run. See Section 5 for details.

7.2.18 InitSalinityFile: input

Contains initial vertical top-to-bottom salinity distribution. Read only when readSalinity=1.

Size: N_{kmax}

Type: binary

Format: (double×N_{kmax})

7.2.19 InitTemperatureFile: input

Contains initial vertical top-to-bottom temperature distribution. Read only when readTemperature=1.

Size: N_{kmax}

Type: binary

Format: (double×N_{kmax})

8 Using the sunplot GUI

The sunplot GUI is meant to be a debugging tool and hence it does not have the ability to output results into a figure file, such as postscript, jpeg, or eps. It is, however, a handy tool to quickly view results without having to execute third-party software. Sunplot has the ability to view all of the data output from SUNTANS on several processors, and it only requires the X11 libraries.

This howto describes how to view the results presented in the two-dimensional river plume example and assumes that the data has been created in the respective data directory in `examples/boundaries/data`. Details on how to create this data is given in Section 6.5, and details on how to compile `sunplot` are described in Section 1.1.

8.1 Starting up the GUI

The `sunplot` GUI can be started while a simulation is running as long as the data files are not empty. To view the data with the default settings in a given directory, use

```
./sunplot --datadir=examples/boundaries/data
```

This will bring up a planview of the salinity data at the first time step $n = 1$ in the upper layer $k = 1$ (note that the GUI does not use C-style indexing, so the first index starts at k=1). If more than one processor is being viewed, that should be specified at the command line, as in

```
./sunplot -np 8 --datadir=examples/boundaries/data
```

For multiprocessor output, the processors being displayed will start at 0 and end at `np-1`. For example, for a 64 processor job, the command

```
./sunplot -np 3 --datadir=examples/boundaries/data
```

will show data on processors 0, 1, and 2.

8.2 Moving around in time

The three buttons beneath `''Step: 1 of 41''` on the upper left of the GUI control the time stepping.

< −−

Left button: Moves one step backward in time.

Middle button: No effect

Right button: Moves to time step 1.

M :

Left button: Animates forward in time from begining to end.

Middle button: Asks the user to input a desired time step at the command line.

Right button: Animates backward in time from the end to the begining.

$|--\ >|$;

Left button: Moves one step forward in time.

Middle button: No effect

Right button: Moves to the last step (in this case step 41).

8.3 Displaying different vertical levels

The two buttons beneath `''Level: 1 of 20''` on the upper left of the GUI control the vertical level being shown. Level 1 is the top level, while level 20 is level N_{kmax}.

$|<\ --|$

Left button: Displays level up (decreasing k).

Middle button: No effect

Right button: Displays level 1.

$|--\ >|$:

Left button: Displays one level down (increasing k).

Middle button: No effect

Right button: Displays level N_{kmax}.

8.4 Displaying different processors

The three buttons beneath `''Showing all 1 Procs''` on the upper left of the GUI control the processor being shown. The default is to show all processors.

$|<\ --|$

Left button: Decreases the processor being displayed. Loops around if on processor 1.

Middle button: No effect

Right button: Displays processor 1 (no C-style indexing).

|All| :

Left button: Displays all processors.

Middle button: No effect.

Right button: No effect.

$|--\ >|$:

Left button: Increases the processor number being displayed. Loops around if on the last processor.

Middle button: No effect

Right button: Displays the last processor.

8.5 Surface plots of data

The default quantity to display is the salinity, or s, in planview, at level 1 and time step 1. Color axes are scaled to the minimum and maximum of the profile being shown. They may also be specified by parameters in `suntans.dat`, as described in Section 8.10. The following buttons control their respective plots:

S : Salinity

Left button: Displays the salinity s.

Middle button: Displays the salinity anomoly $s - s_0$.

Right button: Displays the background salinity s_0.

T : Temperature

Left button: Displays the temperature T.

Middle button: No effect.

Right button: No effect.

q : Nonhydrostatic pressure

Left button: Displays the nonhydrostatic pressure q.

Middle button: No effect.

Right button: No effect.

h : Free surface

Left button: Displays the free surface h.

Middle button: No effect.

Right button: No effect.

n : Eddy viscosity

Left button: Displays the eddy viscosity ν_t at the center of the cell.

Middle button: No effect.

Right button: No effect.

k : Scalar diffusivity

Left button: Displays the scalar diffusivity κ_t at the center of the cell.

Middle button: No effect.

Right button: No effect.

U :

Left button: Displays the horizontal cartesian velocity component u, at the center of the cell.

Middle button: No effect.

Right button: Displays the horizontal cartesian baroclinic velocity component u_b, at the center of the cell. This is calculated with

$$u_b = u - \frac{1}{D}\int_{-D}^{h} u\, dz\,.$$

V :

Left button: Displays the horizontal cartesian velocity component v, at the center of the cell.

Middle button: No effect.

Right button: Displays the horizontal cartesian baroclinic velocity component v_b, at the center of the cell. This is calculated with

$$v_b = v - \frac{1}{D}\int_{-D}^{h} v\, dz\,.$$

W :

Left button: Displays the vertical cartesian velocity component w, at the center of the cell.

Middle button: No effect.

Right button: No effect.

Depth :

Left button: Displays the depth d beneath the 0 datum.

Middle button: No effect.

Right button: Displays the water depth $h + d$.

None :

Left button: No surface plot is shown. Useful for only showing vector field.

Middle button: No effect.

Right button: No effect.

8.6 Vector plots

Vector plotting is controlled with the Vectors, iskip, and kskip buttons.

Vectors :

Left button: Toggle vectors on/off.

Middle button: Shorten the vectors by half their length.

Right button: Double the length of the vectors.

`iskip/kskip`: Number of indices to skip to clarify the vector plot. When in planview mode of the data, only every `iskip` vector on the unstructured grid will be shown. Changing `kskip` has no effect on the planview plot. In profile mode, every `iskip` vector in the x-direction is shown and every `kskip` vector in the z-direction is shown. The value of `iskip` and `kskip` are changed with their respective arrow buttons:

< : Decrease skip amount.

Left button: Decrease skip by 1.

Middle button: No effect.

Right button: Set skip amount to 1.

> : Increase skip amount.

Left button: Increase skip by 1.

Middle button: No effect.

Right button: Planview mode: set skip amount to 5. In profile mode, set skip amount to maximum index size.

8.7 Displaying the grid

The grid can be viewed with the following buttons:

Edges :

Left button: Toggle display of Delaunay edges.

Middle button: No effect.

Right button: No effect.

Voro :

Left button: Toggle display of Voronoi points.

Middle button: No effect.

Right button: No effect.

Dela :

Left button: Toggle display of Delaunay points.

Middle button: No effect.

Right button: No effect.

8.8 Zooming in on data and obtaining profile plots

By default, zooming is on. The effect of using the mouse and selecting a point on the plotted data depends on whether zooming is on or off or if you are viewing a profile plot.

ZOOM

Left button: Toggle turning zoom on/off.

When zooming is **on**:

Left button: Can be used either to select an area in the plot to zoom into, or it can be used to zoom into an area by a factor of 2, centered about the clicking point.

Middle button: Returns to the default in which the entire domain is displayed provided that there is no mouse movement. If there is mouse movement, the view will pan corresponding to the mouse motion on the button release. Alternatively, the user can toggle a panning mode via pressing the 'm' key.

Right button: Zooms out by a factor of 2.

When zooming is **off**:

Left button: Used to select a line along which a 2-d x-z transect of the data is displayed. Note that the transect uses a simple nearest-neighbor interpolation to display the data in the transect. If you press the mouse button once in the domain while zooming is off, you may get a message that says `''Cannot plot this slice''`. This occurs because the length of the transect you have chosen is effectively zero by clicking once on the plot window when zooming is off.

Middle button: No effect.

Right button: Displays the processor number and the value of the data at the selected point.

Middle button: No effect.

Right button: No effect.

Profile

Left button: This will toggle back and forth between the surface plot and the previous transect that was displayed. In the event that no transect has been displayed, the error message `Need two points for a profile first!` will be displayed. In this case, turn zooming off and select two points for a profile.

Middle button: Display the default west-east transect along the middle of the data set. This is useful when you have a channel flow in the x-direction and want to see the data in the x-z plane.

Right button: Display the default north-south transect along the middle of the data set. This is useful when you have a channel flow in the x-direction and want to see the cross-channel data in the x-y plane.

8.9 Changing the axes scaling

By default, the x-y axes are scaled equally in "Image" mode (If the `-2d` flag is supplied at the command line, however, the default mode is "Normal"). This makes profile plots in x-z difficult to view especially for high aspect ratio grids. Hitting the Aspect button with the left mouse button will toggle between "image" mode and "normal" mode, in which the data is scaled to fit the plot area. You can also do this by hitting the "a" key. The middle and right mouse buttons have no effect here.

8.10 Changing the color axes

By default, the color axes are scaled with the maximum and minimum values of the data. This can be changed with the Caxis button as follows:

Caxis

Left button: Toggle between scaling with the maximum and minimum data values and using the values specified in `suntans.dat`. Specifically, if you add the following lines to your `suntans.dat` file, the data can be scaled with these color axes values when pressing the Caxis button:

```
caxismin   -0.25      # Minimum value for color axes
caxismax    0.25      # Maximum value for color axes
```

If these values are not in `suntans.dat`, an error will be indicated each time data is plotted until the Caxis button is hit again to return to the default scaling mode.

Middle button: No effect.

Right button: Enter color axes values at the command line. The format must be in the form `cmin cmax` followed by a carriage return. Carriage returns may also exist between the entries of `cmin` and `cmax`.

8.11 Quitting out of sunplot

You can either hit the Quit button or the "q" key to quit out of sunplot.

8.12 Command line options

Options can be passed to sunplot at the command line to reduce the number of key strokes required to obtain a desired view of data. The following is a list of command line arguments:

`--datadir=path`: This is the path to the directory containing the sunplot.dat data file. If this is not specified, then it is assumed that the data file is in the local directory.

`-np 2`: Specify the number of processors included in the data. For a multi-processor data set, if this is not specified then only the data from processor `0` will be displayed.

`-2d`: Display an x-z profile plot of the data in axis Normal mode. This is equivalent to starting sunplot with no command line options and hitting the Axes button with the left mouse button and the Profile button with the middle mouse button.

`-g`: Only display the grid. This is useful for debugging the grid without loading in any other data.

`-m`: Step forward through the data as an animation.

References

[1] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics, Volume I.* Springer-Verlag, 1997.

[2] O. B. Fringer, M. Gerritsen, and R. L. Street. An unstructured grid, finite-volume, nonhydrostatic, parallel coastal ocean simulator. *Ocean Modeling*, 14(3-4):139–278, 2006.

[3] S. M. Jachec, O. B. Fringer, M. G. Gerritsen, and R. L. Street. Numerical simulation of internal tides and the resulting energetics within Monterey Bay and the surrounding area. *Geophys. Res. Lett.*, 33:L12605, 2006.

[4] G. Karypis, K. Schloegel, and V. Kumar. *Parmetis*: Parallel graph partitioning and sparse matrix ordering library, 1998.

[5] J. R. Shewchuck. *Triangle*: A two-dimensional quality mesh generator and delaunay triangulator. version 1.3, 1996.

[6] Y. Zang, R. L. Street, and J. R. Koseff. A non-staggered grid, fractional step method for time-dependent incompressible Navier-Stokes equations in curvilinear coordinates. *J. Comput. Phys.*, 114:18–33, 1994.