In [1]:
```
# Turning on inline plots -- just for use in ipython notebooks.
%pylab inline
```

```
Welcome to pylab, a matplotlib-based Python environment [backend:
module://IPython.zmq.pylab.backend_inline].
For more information, type 'help(pylab)'.
```

Initialization of a numerical experiment

Before running a drifter simulation, a number of parameters need to be specified. Previous examples of this are set in `init.py`. Because these examples change over time, we'll go through a specific example here.

In [2]:
```
# Normal Python libraries
import numpy as np
from datetime import datetime, timedelta
import netCDF4 as netCDF

# tracpy modules
import inout
import tools
import run
import plotting
```

Model output

Model output from a high resolution model of the Texas-Louisiana shelf for the years 2004-2012 is stored on a thredds served at the address in loc. This is freely accessible.

In [3]:
```
# Location of TXLA model output file and grid, on a thredds server.
loc = 'http://barataria.tamu.edu:8080/thredds/dodsC/NcML/txla_nesting6.nc'
```

Time parameters

Model output is known to occur every four hours. The default test here will start at 00:00 on November 25, 2009 and run for 5 days.

In [4]:
```
# Number of days to run the drifters.
ndays = 5

# Start date in date time formatting
date = datetime(2009,11, 25, 0)

# Time between outputs
tseas = 4*3600 # 4 hours between outputs, in seconds
```

In order to smooth results, it is typical to linearly interpolate model output in time before performing particle tracking. How many interpolations steps to do between model outputs depends on the model output and how much time is between data sets, but something between 5 and 10 interpolation steps is a good place to start. A sensitivity study using this model output showed minimal overall difference in results between values of nsteps of 5, 10, and 15.

In [5]:
```
# Number of interpolation steps between model outputs.
nsteps = 5
```

After initialization, drifters can be stepped forward or backward in time. Running backward in time essentially means that we change the sign of the velocity fields and step backward in the model output files (in which case we set `ff=-1`). We'll move forward in time (`ff=1`).

In [6]:
```
# Use ff = 1 for forward in time and ff = -1 for backward in time.
ff = 1
```

Subgrid parameterization parameters

An integer flag is used to control whether or not to use subgrid parameterization in the particle tracking, and if so, which kind.

Options are:

- `doturb=0` uses no sub grid parameterization and thus the drifters are passively advected according strictly to the output velocity fields
- `doturb=1` adds to the current velocity fluxes parameterized turbulent velocity fluxes of the order of the current velocity fluxes
- `doturb=2` adds to the calculated new drifter location a slightly displaced drifter location that is randomly placed based on a circle around the drifter location
- `doturb=3` adds to the calculated new drifter location a slightly displaced drifter location that is randomly placed based on an ellipse of the bathymetry around the drifter location

The horizontal and vertical diffusivities are set by the user. These values may or may not be used in the experiment depending on whether a subgrid parameterization is used, and, if so, which is used. The horizontal diffusivity value is used by all of the horizontal subgrid parameterizations. The vertical diffusivity is not used in the two-dimensional case. Since this experiment is not using either diffusivity values, they will be set to zero to avoid confusion.

Appropriate values to use for this are currently being investigated using sensitivity studies on the Texas-Louisiana shelf. Some values have been used and compared in studies, and values can be calculated from physical drifters for a specific domain. This is on-going work! In a sensitivity study, a smaller value, like `ah=5`, leads to somewhat diffused results that are still very close to the non-diffusive case. A larger value of `ah=20` led to more diffused results that were still quite similar to the non-diffusive case.

In [7]:
```
ah = 0. # m^2/s
av = 0. # m^2/s

# turbulence/diffusion flag
doturb = 0
```

Grid information

Grid information has been stored in the `loc` array previously and the `inout` module will read necessary information from the netCDF file and place it into a dictionary.

In [8]:
```
grid = inout.readgrid(loc)
```

Drifter initialization

Horizontal

Drifters are seeded by the latitude and longitude. A simple way to do this is to set up a mesh of points within a lat/lon box. In this case, we are looking at drifters starting throughout the TX-LA shelf domain. For the `linspace` function, we can play around with the number of points to control approximately how far apart the drifters begin. For this example, the number of points are about 20 km apart.

After initializing these points, we can run them through a check script to eliminate points outside the domain (without this step, points outside the numerical domain will cause an error).

In [9]:
```
# Input starting locations as real space lon,lat locations
lon0, lat0 = np.meshgrid(np.linspace(-98.5,-87.5,55), \
                         np.linspace(22.5,31,49)) # whole domain, 20 km

# Eliminate points that are outside domain or in masked areas
lon0, lat0 = tools.check_points(lon0, lat0, grid)
```

Vertical

There are a number of options for the initial vertical placement of the drifters. The behavior is controlled by the combination of `z0` and `zpar`, and `do3d` must be set accordingly as well.

The `do3d` flag controls whether or not drifters are allowed to move vertically or not:

- `do3d=0` for two-dimensional particle tracking
- `do3d=1` for three-dimensional particle tracking

For 3D tracking, set `do3d=1` and `z0` should be an array of initial drifter depths. The array should be the same size as `lon0` and negative for under water. Currently, drifter depths need to be above the seabed for every (x, y) particle location for the script to run.

To do 3D but start at surface, use `z0 = zeros(lon0.shape)` and have either `zpar='fromMSL'` so that `z0` starting depths represent that depth below the base, time-independent sea level (or mean sea level) or choose `zpar='fromZeta'` to have `z0` starting depths represent that depth below the time-dependent sea surface. Currently only the `zpar='fromZeta'` case is coded up.

For 2D drifter movement, set `do3d=0`. Then there are the following options:

- set z0 to 's' for 2D along a terrain-following slice and zpar to be the index of s level you want to use (0 to km-1)
- set z0 to 'rho' for 2D along a density surface and zpar to be the density value you want to use. Can do the same thing with salinity ('salt') or temperature ('temp'). The model output doesn't currently have density.
- set z0 to 'z' for 2D along a depth slice and zpar to be the constant (negative) depth value you want to use
- To simulate drifters at the surface, set z0 to 's' and zpar = grid['km']-1 (whatever that value is) to put them in the upper s level. This is probably the most common option.

In [10]:
```
# for 3d flag, do3d=0 makes the run 2d and do3d=1 makes the run 3d
do3d = 0

## Choose method for vertical placement of drifters
z0 = 's' #'z' #'salt' #'s'
zpar = 29 #-10 #grid['km']-1 # 30 #grid['km']-1
```

File saving

The input `name` will be used for saving the particle tracks into a netCDF file and for the figures.

In [11]:
```
# simulation name, used for saving results into netcdf file
name = 'temp'
```

Run the numerical experiment

In [12]:
```
lonp, latp, zp, t, grid = run.run(loc, nsteps, ndays, ff, date, \
                                  tseas, ah, av, lon0, lat0, \
                                  z0, zpar, do3d, doturb, \
                                  name)
```

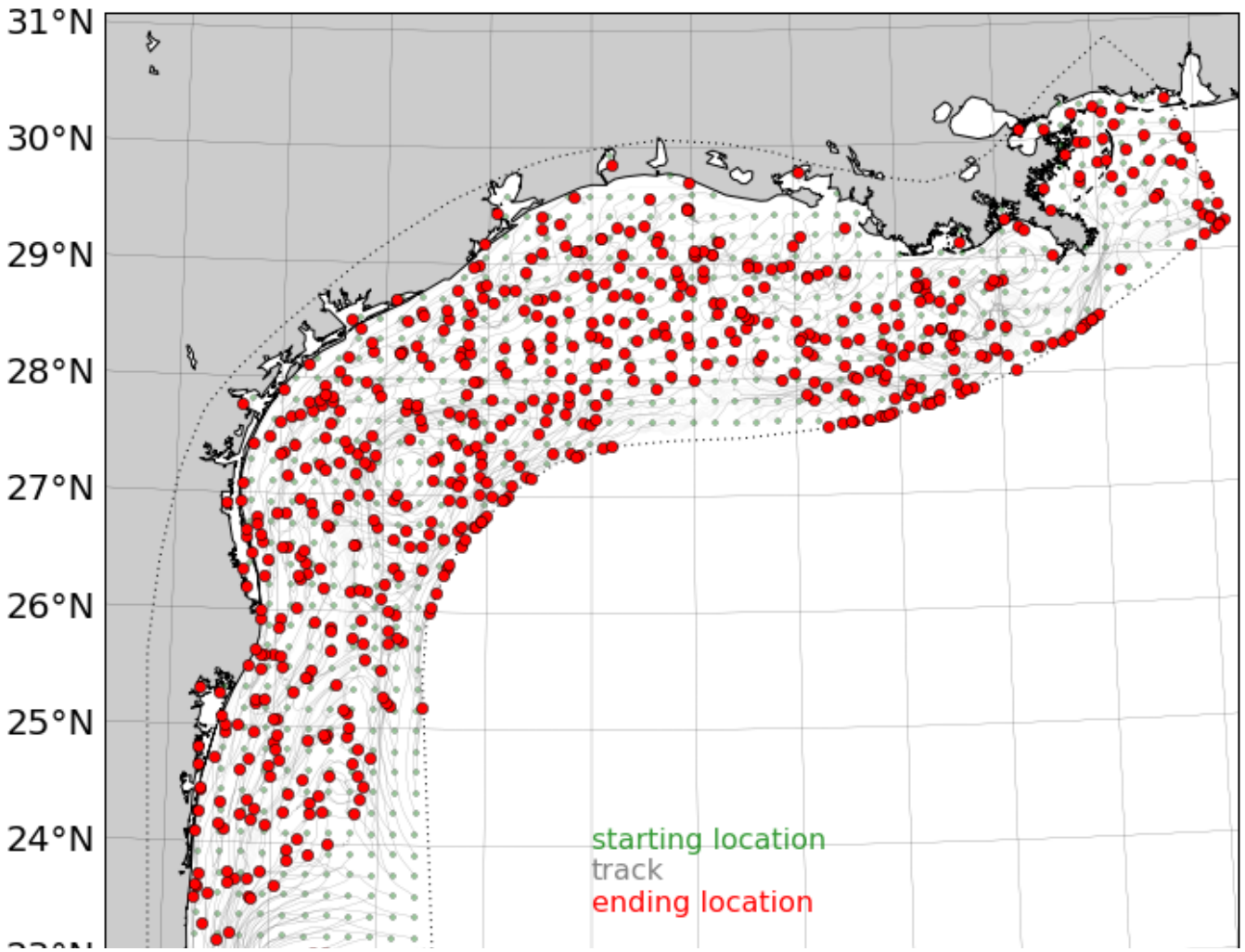
```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

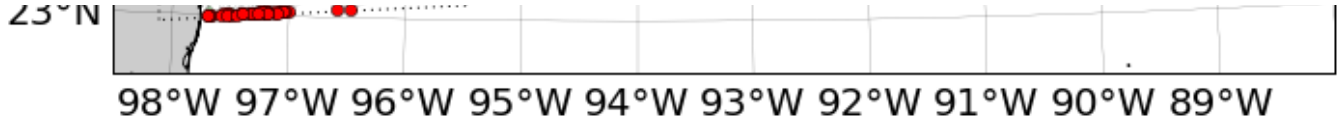
```
26
27
28
29
run time:                       58.9128530025
-----------------------------------------
Time spent on:
        Initial stuff:          17.41 (29.55%)
        Reading in fields:      40.43 (68.62%)
        Z interpolation:        0.00 (0.00%)
        Tracmass:               0.14 (0.23%)
```

Plotting the results

Plot tracks

In [13]:
```
plotting.tracks(lonp, latp, name, grid=grid)
```





Terminal/origin histograms

A histogram can quickly show where particles end up (for forward time case) or where particles originated (for backward time case).

In [14]:
```
plotting.hist(lonp, latp, name, grid=grid, which='hexbin', bins=(50,50))
```