Project assignment: Particle-based simulation of transport by ocean currents

Tor Nordam, Jonas Blomberg Ghini, Jon Andreas Støvneng

1 Introduction

It is probably well known that the Norwegian Meteorological Institute [1] produces a weather forecast, predicting the weather several days into the future. This weather forecast includes a vector field of air velocities at 10 m elevation above the ground, commonly known as "the wind"[2]. Less well known is that they also produce a forecast for the ocean, which includes a velocity field describing the ocean currents about 2 days into the future. In the event of an oil spill at sea, this information can be used to predict where the oil will end up, which in turn can be used to direct response operations to try to minimise the damage. During for example the Deepwater Horizon oil spill in the Gulf of Mexico in 2010, numerical simulations were used daily to predict what would happen over the next few days.

The goal of the project is to simulate transport of matter by ocean currents. While oil spills at sea are relatively well known from media coverage, it is by no means an easy task to simulate what happens. Oil spilled at sea displays quite complex behaviour: it can form droplets submerged in the water, continuous slicks at the surface, it will partially dissolve, partially evaporate and partially biodegrade, and it can form stable oil-in-water emulsions, all of which will significantly alter its properties. For these reasons, we will study the simpler case of transport of a dissolved chemical. A dissolved chemical will move in the same way as the surrounding water, without sinking or rising due to differences in density.

In this project, we will study the transport of dissolved chemicals in the ocean using a particle method. You will learn to write programs that can read data which are provided by the Meteorological Institute, and how to interpolate these data and use them to calculate trajectories. You will also learn to plot positions, trajectories and concentration fields on a map.

[1] For some reason also known as met.no, pronounced "met no".

[2] See `http://www.yr.no/kart/#laga=vind`.

2 Background and Theory

The point of this exercise is to look at how we can calculate the transport of a chemical in the ocean. In our simulations, we will represent the dissolved matter as numerical particles, also called "Lagrangian elements". The idea is that a numerical particle will represent a given amount of dissolved matter. This is not to be interpreted as an actual, physical particle, but simply as a numerical approximation. If we have large numbers of numerical particles, they can be used to calculate concentration, which is proportional to the density of particles (that is, the number of particles in a volume).

The particles will, due to the velocity of the water. We assume that the presence of the particles don't affect the motion of the water, so we can take the velocity of the water as a given. This is an excellent approximation for any release of a chemical into the ocean, as the volume of the water will be far greater than the volume of the chemical.

We will consider two different models for how the particles move with the water. In the first model, the particles have some inertia. When the water moves, the particles experience a force, due to friction, that tends to make the particles move along with the water. This can be expressed by the Ordinary Differential Equation (ODE)

$$\ddot{\mathbf{x}} = \frac{\alpha}{m}(\mathbf{v}_w - \dot{\mathbf{x}}), \tag{1}$$

which is simply Newton's second law, with $\mathbf{F} = \alpha(\mathbf{v}_w - \dot{\mathbf{x}})$. Here, $\mathbf{v}_w$ is the velocity of the water, $\dot{\mathbf{x}}$ is the velocity of the particle, and α is a drag coefficient. Note that if $\mathbf{v}_w = \dot{\mathbf{x}}$, then there is no force, as there is no relative motion between water and particle, and thus no friction.

In the other model, we will simply assume that the particle always moves with the same velocity as the water. This is essentially the same as saying the particle has no inertia. In this model, if you give the particle some velocity, and release it, the motion will immediately be damped, due to friction against the water. It is therefore called the overdamped limit. This model is described by the ODE

$$\dot{\mathbf{x}} = \mathbf{v}_w(\mathbf{x}, t). \tag{2}$$

We will see that if the drag coefficient is large, then the overdamped limit is a good approximation, and the two equations will give very similar trajectories for the particles. We will also see that there are some subtle issues that require extra care when handling Eq. (1), while Eq. (2) is more straightforward.

2.1 Oceanographic data

In this project, we will use pre-calculated ocean current data to tell us the velocity of the water, $\mathbf{v}_w$, as a function of $\mathbf{x}$ and t. The oceanographic data are produced by met.no, and they are the results of running a numerical simulation engine known as ROMS[3], on a model domain known as NorKyst800m[4]. It provides information about current velocities, water temperature and salinity for an area covering the entire Norwegian coast, at 800 m $\times$ 800 m horizontal resolution, with 12 vertical layers[5], and with a time resolution of 1 hour.

The data are available for download as NetCDF[6] files, or the data can be accessed via OPeNDAP[7]. NetCDF is a file format for storing data in arrays. It is quite well suited for medium to large amounts of data (up to 100s of gigabytes in a single file works fine), and it is very commonly used for geographical data such as ocean or atmosphere data. In order to access the data, we will use the python library `xarray`.

The data files contain the x and y components of the velocity field, stored in two variables named `u` and `v` (it is quite common to store only the horizontal components of the current velocity). These variables are stored as rank 4 arrays, which give their values as a function of time, depth, y position and x position (note that the order of the dimensions in the files is (t, z, y, x)). The coordinates of the grid points along these dimensions are also stored in the data files, in the variables `time`, `depth`, `Y`, and `X`. For simplicity, we will ignore the depth dimension, dealing only with movement in the horizontal plane, and time. In Fig. 1, temperature data for the surface water is shown, as an example to illustrate the extent of the available data. In Fig. 2, the same data are shown in the coordinate system used in the file, with the x coordinates on the horizontal axis, and the y coordinates on the vertical. The origin of the coordinates system is at the North Pole.

The dimensions x and y, as shown in Fig.2, are the coordinate axes in what is known as a polar steregraphic projection of the Earth's surface onto a plane. For the purpose of this project, we will deal with motion in the xy plane, with coordinates as shown. As the vector components of the velocity field are aligned with these coordinate axes, we can use the components directly to calculate motion in the xy plane. This means that for the transport simulations, we will ignore the curvature of the Earth. In the end, we will see how to transform from xy coordinates to longitude and latitude, and plot the particle positions on a map.

[3] `www.myroms.org`

[4] `http://www.imr.no/temasider/modeller/kystmodellen/`

[5] From 0 to 300 m depth, with higher resolution in the upper layers

[6] Network Common Data Form, see `www.unidata.ucar.edu/software/netcdf/`.

[7] Open-source Project for a Network Data Access Protocol, see `www.opendap.org/`.

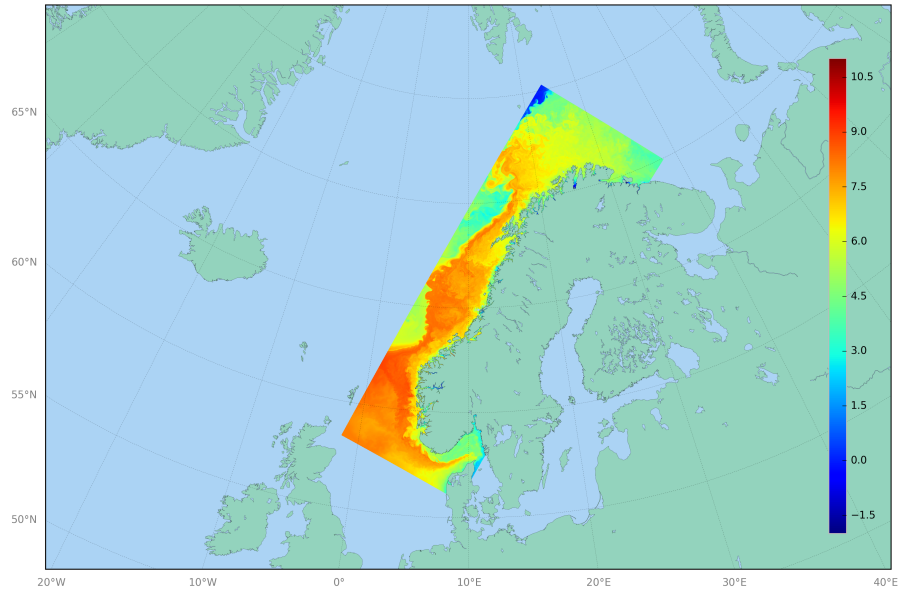

Figure 1: The domain of the NorKyst800m model, showing surface water temperatures on February 4, 2017.

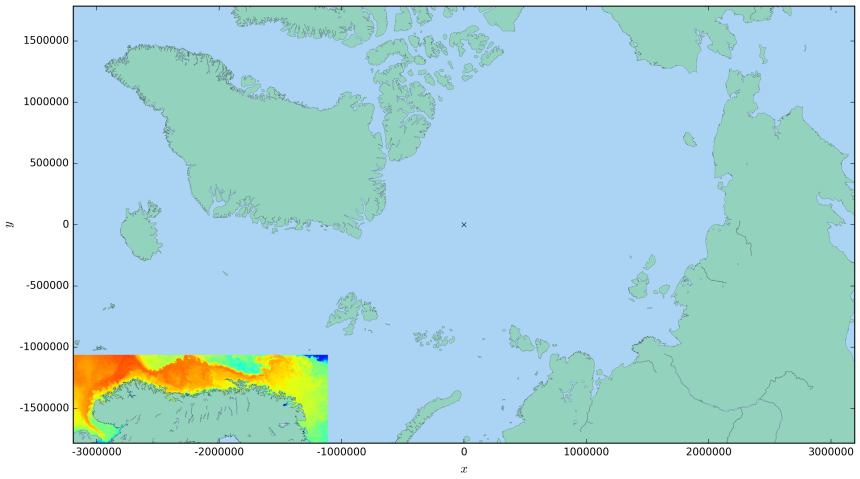

Figure 2: The domain of the NorKyst800m model, shown in the coordinate system used to store the data. The origin of the coordinate system is at the North pole (marked with $\times$), and the distances are in meters.

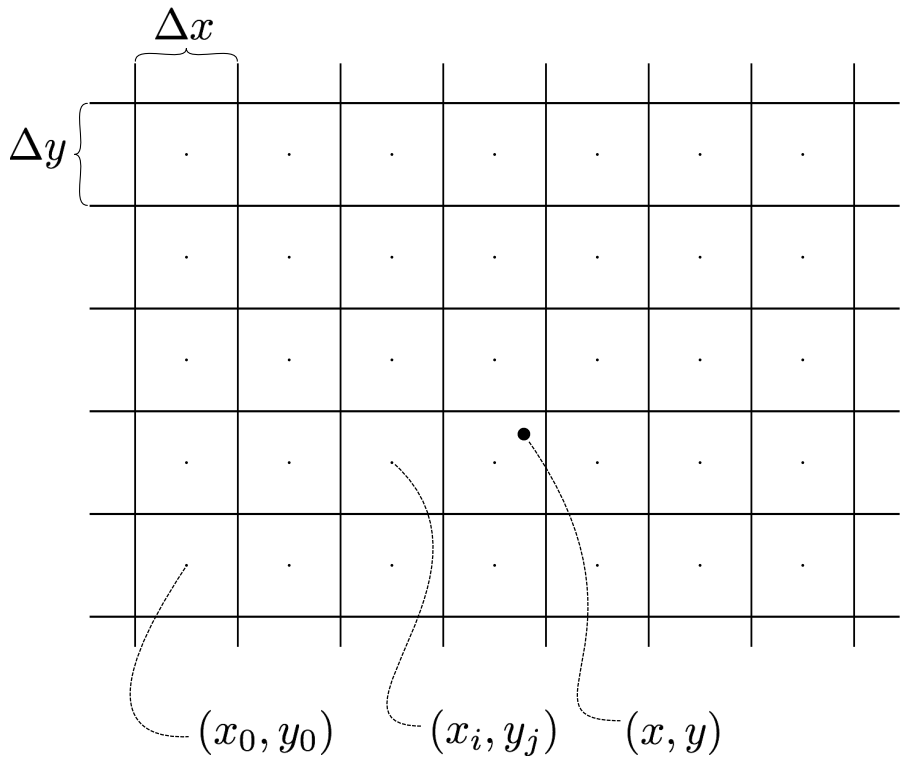

Figure 3: Velocity vector components are given only on discrete points, $(x_i, y_j) = (x_0 + i\Delta x, y_0 + j\Delta y)$. In order to calculate trajectories, we need to evaluate the velocity at arbitrary positions (x, y).

2.2 Interpolation

The oceanographic data we will use here are provided on a grid of discrete positions and times, (t_n, z_l, y_j, x_i). As mentioned, we will consider motion in the xy plane only, *i.e.*, we will consider a constant depth $z = 0$. The available data then represents a time-variable 2D vector field of two-component vectors. The vectors of this field are given on a regular quadratic grid, with cells of size 800 m $\times$ 800 m. We say that the spatial resolution of the data is 800 m. Additionally, there is a temporal resolution: the data are provided at one-hourly intervals.

For each cell in the grid, and for each interval of one hour, there is one vector. The vector at time and location (t_n, y_j, x_i) gives the average velocity of water in an 800 m $\times$ 800 m cell centered at (x_i, y_j), and in a time interval of length 1 hour, centered at time t_n.

In order to calculate the trajectory of a particle that moves in the velocity field defined by these data, we will have to evaluate the vector field at arbitrary locations (see Fig. 3). To evaluate the velocity at a location (x, y), one possible

option is to identify which cell that location is in, and use the vector defined at the center of that cell. This is sometimes known as nearest-neighbour interpolation. The advantages of this approach include simple implementation, and the fact that it makes a certain intuitive sense, as the vector is supposed to define the average velocity inside a cell. The major disadvantage is that the vector field is then discontinuous at the boundaries of cells, which is not realistic.

A more advanced approach is the technique known as spline interpolation, which produces piecewise polynomial results. The idea of spline interpolation is to ensure that not only the interpolated variable is continuous, but also its first $k-1$ derivatives, where k is the order, or degree, of the spline interpolation. An example of spline interpolation in one dimension is shown in Fig. 4. We have some given data points, marked with blue dots, and the goal is to find a function that passes through all of these points. First order spline interpolation is equivalent to simple linear interpolation (the blue line in the top panel). In this case, the interpolated values are continuous, but the derivative (the blue line in the bottom panel) will change discontinuously at each datapoint. For second order splines, the function is now continuous and smooth, the derivative is also continous, but piecewise linear, which means the second derivative will be discontinous. For third order splines, also known as cubic splines, both the function and the derivative are continuous and smooth. The same ideas also extend to two and more dimensions[8].

In this project, we will use the class called `RectBivariateSpline` from `scipy.interpolate` to carry out the interpolation to arbitrary positions in the xy plane. The procedure is as follows:

- For each new timestep, get the rank 2 arrays holding the variables `u` and `v` for that time, and for the surface layer ($z = 0$).
- Pass `u`, along with the coordinate arrays `Y` and `X` to `RectBivariateSpline`. This returns a function, which can be used to obtain the interpolated value of `u` at any point.
- Do the same for `v`.
- Use these to calculate the motion of the particles, step forward in time, and repeat.

See the attached code examples for some help on how to get started.

[8] See Section 3.4 of the textbook for further details.

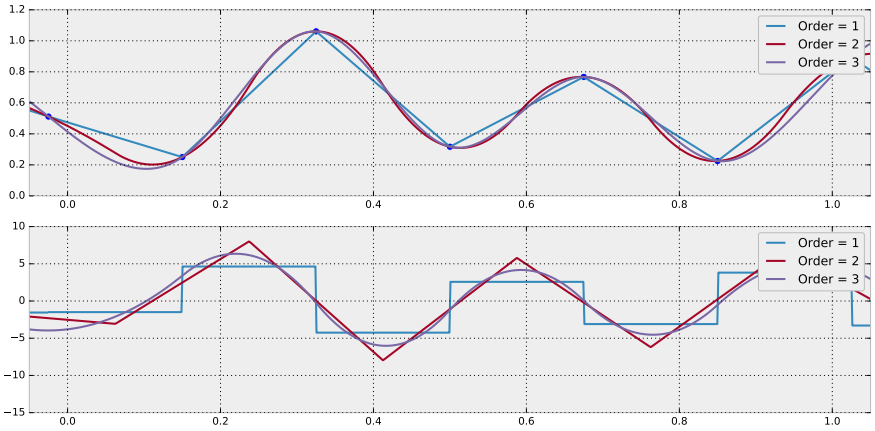

Figure 4: Example of one-dimensional spline interpolation (top) with derivatives (bottom).

3 Plotting on maps with python

After calculating how the particles are transported with the ocean currents, we will plot their trajectories and positions on a map. There are two main libraries in use to plot data on maps with python, `basemap` and `cartopy` (both can be installed with conda). In this project, we will use `cartopy`, which is slightly easier to use, and seems to be more actively developed (though it currently lacks some features found in `basemap`).

When moving particles around with the water velocity, we will use the coordinate system shown in Fig. 2. For plotting trajectories, it is straightforward to just use those coordinates directly, which will show distances in meters. However, if we want to show the trajectories on a map, we need to convert from the xy coordinate system of the polar stereographic projection, to longitude and latitude. We will use a library called `pyproj` for this purpose. This requires a little bit of work, so code examples are provided.

Problems

1 - Simple test case

In this problem, we will start simple. Instead of using actual ocean current data, we will use the following analytical expression to represent the velocity of the water:

$$\mathbf{v}_w(\mathbf{x}, t) = \frac{2\pi R}{T}[-\frac{y}{R}, \frac{x}{R}], \tag{3}$$

where $R = \sqrt{x^2 + y^2}$, and $T = 24$ hours. This equation describes a rotating velocity field where the speed at each radius is exactly correct to make a full circle in 24 hours. Note that while this is not actually a function of t, you should still define the function to take t as an argument, in order to make it easy to swap the test function for the real velocity field later on, and you should write f to take the x argument as a vector containing position, for example like this:

```
def f(X, t):
    x = X[0]
    y = X[1]
    R = np.sqrt(x**2 + y**2)
    ...
```

In each of the tasks below, we will consider a particle starting from an initial position $\mathbf{x}_0 = [0, L]$, where $L = 100$ m, (and in tasks **1c** and **1d** with velocity $\dot{\mathbf{x}} = [0, 0]$) at time $t = 0$.

a Let the trajectory, $\mathbf{x}(t)$, of the particle be controlled by Eq. (2). Calculate the trajectory as the particle moves for 48 hours, using the Forward Euler method (Section 6.1.1 in the textbook).

The exact solution is that the particle should end up back where it started, after completing two complete circles. Calculate the global error as the distance from the exact end position, to the numerically obtained end position. Find the trajectory using several different timesteps (pick at least 10 different values), h, and make a plot showing the error as a function of h, on a log-log scale.

Finally, find the longest timestep you can use, while still getting an error of less than 10 meters.

b Repeat the assignment in task **1a**, but now using the Explicit Trapezoid method (Section 6.2.2 in the textbook). This is a 2nd-order method of the Runge-Kutta family, and is also known as Heun's method. Compare the time the simulation takes to run in the two cases (Euler and Explicit Trapezoid), for the timstep you found that will give an error less than 10 meters.

c Let the trajectory, $\mathbf{x}(t)$, of the particle be controlled by Eq. (1), with $m = 10^{-2}$ kg and $\alpha = 5 \cdot 10^{-5}$ Ns/m. Calculate the trajectory as the particle moves for 48 hours, using the Explicit Trapezoid method.

Also in this case, it is possible to find an analytic expression for the solution, which we can evalute numerically. See the Appendix for the details, and a short piece of code that will give you the numerical value of the exact solution.

Do the numerical integration of the trajectory with several different timesteps (pick at least 10 different values between 100 s and 1000 s), and for each one, calculate the distance from the end position, to the end position of the exact simulation. Make a plot showing the global error as a function of h, on a log-log scale.

Finally, make a plot which shows the entire trajectory (use a timestep you feel is sufficiently accurate, based on the results above), and compare to a trajectory from task **1a**. Discuss why you need a much shorter timestep to get accurate results in this case (compared to task **1b**, when the trajectories look fairly similar.

d Finally, use the Euler and Explicit Trapezoid methods (which make up a pair of Runge-Kutta methods of orders 1 and 2) to construct a variable timestep integrator (see Section 6.5.1 in the textbook), and apply that to the problem of task **1c**. Plot the resulting trajectory, and plot the development of the timestep over time. Discuss the results.

2 - Transport of a particle with ocean currents

We will now turn to using actual ocean current data to calculate particle trajectories, starting out with a single particle. For the tasks below, you will consider a particle starting at a position $\mathbf{x}_0$, at time $t = 0$. Let the trajectory of the particle be governed by Eq. (2), where $\mathbf{v}_w$ is taken from the ocean current data.

In principle, it is possible to run with data directly from the server at `http://thredds.met.no`, however you may find that it takes a long time to run simulations. Instead, we have prepared a datafile containing 20 days of data spanning from February 1 to February 20, 2017. It is available for download here:

- `http://folk.ntnu.no/nordam/data/Norkyst-800m.nc`

In Problems **2** and **3** you will use the Explicit Trapezoid method, with a timestep $h = 3600$ s. Since this integrator evaluates the velocity field only at integer multiples of h, and since the data are provided at intervals of 1 hour, no interpolation in time is needed. In the provided examples, you will find a class

called `Interpolator`, which contains code to read and interpolate data from the files. See the examples for how to use it.

a Let $\mathbf{x}_0 = (-3000000, -1200000)$, and transport the particle with the ocean current for a time of 10 days. Plot your results in xy coordinates. Try different start dates between February 1 and February 10. Compare your calculated trajectory to results of others. If you start at the same time and position, you should get exactly the same results.

b Use the attached example code to plot your trajectory on a map. Using Figs. 1 and 2 as a guide, try selecting three other initial positions, and repeat the simulation.

3 - Concentration of dissolved chemical

In this problem, we will simulate the transport of a large number of particles, in order to represent the transport of a dissolved chemical in the ocean. Let each particle's trajectory be controlled by Eq. (2), and note that the motion of any one particle is not affected by the presence of other particles.

Start out a collection of N_p particles, randomly placed in the square defined by $-3010000 < x < -2990000$, $-1210000 < y < -1190000$. Transport all particles with the ocean current for a time of 10 days. You can try different values of N_p, but if you make good use of numpy array operations, this should not take much longer for $N_p = 10000$ than it did for one particle, as most of the time is spent fetching data from disk.

a Plot the positions of the particles on a map, at $t = 0$, and after 2, 4, 6, 8, and 10 days.

b In this task, you will calculate concentrations by defining a grid, with cells of size 800 m $\times$ 800 m, in the xy coordinate system defined by the projection, and simply counting the number of particles in each cell. Make sure your grid is large enough to cover all (or at least most) of the particles. Plot the concentration grid on a map, at $t = 0$, and after 2, 4, 6, 8 and 10 days.

Appendix

This appendix shows how to find the analytic solution in subproblem **1c**, and evaluate it numerically. You should compare the result of your numerically integrated solution with the exact solution found here.

First, let us define $k = \alpha/m$, $\omega = 2k\pi/T$. Then, combining the velocity field in Eq. (3) with the equation of motion given by Eq. (1), we get

$$\ddot{x} + k\dot{x} + \omega y = 0, \tag{4a}$$
$$\ddot{y} + k\dot{y} - \omega x = 0. \tag{4b}$$

Rewrite this as a first order system of equations by introducing the unknowns $(u, v) = (\dot{x}, \dot{y})$:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{u} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} u \\ v \\ -ku - \omega y \\ -kv + \omega x \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\omega & -k & 0 \\ \omega & 0 & 0 & k \end{pmatrix} \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}. \tag{5}$$

The initial condition specified in the problem text is $[x_0, y_0, u_0, v_0] = [L, 0, 0, 0]$. The solution to this initial value problem is written as

$$\begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix} = \sum_{i=0}^{3} v_i C_i \mathrm{e}^{\lambda_i t}, \tag{6}$$

where (λ_i, v_i) are eigenvalues and eigenvectors of the 4×4 matrix in Eq. (5). These we can find numerically by using the function `eig` from `numpy.linalg`. The constant C_i must be such that the solution is equal to the initial condition when $t = 0$, hence

$$\begin{pmatrix} x_0 \\ y_0 \\ u_0 \\ v_0 \end{pmatrix} = \sum_{i=0}^{3} v_i C_i, \tag{7}$$

and we can find C_i by solving Eq. (7). We can solve it numerically with the function `solve`, also from `numpy.linalg` (this is a wrapped version of the fortran library `lapack`, which solves the equation using LU-decomposition).

Putting together all of the above, and the numerical values of α, m and T, we can write down the following piece of python code, which will print the position of the particle after 48 hours:

```
L     = 1.0E+02
alpha = 5.0E-05
m     = 1.0E-02
k     = alpha/m
w     = 2*np.pi/(24*60*60)*k

A = np.array([
    [ 0, 0, 1, 0],
    [ 0, 0, 0, 1],
    [ 0,-w,-k, 0],
    [ w, 0, 0,-k]
])

lams , V = np.linalg.eig(A)
X0       = np.array([L,0.,0.,0.])
C        = np.linalg.solve(V,X0)

t = 2*24*3600
X = V.dot(C*np.exp(lams*t)) # pun
print('Position after 48 hours = ', X[0].real, X[1].real)
```