Euler Simulation, Python

Contents

1 Introduction

The Euler Simulation project applies the methods used in Knowles and Yeh (2018) to simulate, in one dimension, the shoaling process of a shallow-water wave. The original paper uses such a simulation to predict the wave amplification process, that is, how a, the amplitude of the wave, relates to h, the local water depth.

This project uses the same algorithm, in python, to handle other cases, e.g. the wind's effect on a shoaling solitary wave.

2 Code Documentation

Simulation involves two python files.

2.1 simulator.py

simulator.py contains one relevant class: `Simulator1D`, which wraps necessary functionality to run a simulation.

2.1.1 `Simulator1D`: Constructor

```
sim = Simulator1D(bathymetry, dt, dx, eta0, phiS0)
```

Initializes a Simulator instance with the given parameters:

argument	description
bathymetry	numpy array of the bathymetry, must have an even number of nodes. 0 is expected to be water level.
dt	time resolution of the simulation
dx	spatial resolution of the simulation (distance between points of bathymetry)
eta0	initial free surface heights (with spatial resolution dx, 0 is expected for still water) expected to be a numpy array
phiS0	velocity potential at the free surface, expected to be a numpy array.

`Simulator1D` has the following keyword arguments:

argument	description	default
zeta_x	First derivative (gradient) of the bathymetry, if a higher order approximation of gradient is desired. If no argument for zeta_x is passed, it is instead calculated from the finite difference $$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$ where the edges are assumed to have a gradient of 0	(see description)
M	Terms in the pertubation expansion (higher number is more accurate, but requires more computation)	5
v	lowpass threshold. If v is a number, any wavenumber greater than the largest wavenumber times v is clipped off each timestep. v can also be a function that takes a wavenumber magnitude and the peak wavenumber, and returns how it should be scaled	0.7
g	acceleration due to gravity.	9.81
h0	base still water depth	bathymetry[0]
P	Wind pressure coefficient. Atmospheric pressure is set to be $P \cdot \eta_x$ to simulate the wind effect, unless P_atmos is over- written in the step() method. See Zdyrski and Feddersen (2021)	0

In the case that v is a function, the wave number is the first argument and the peak wavenumber is the second argument, so the following will result in the same low-pass filter $v = 0.7$:

```
def v(k, peak):
    if k/peak <= 0.7:
        return 1
    else:
        return 0
```

2.1.2 `Simulator1D`: step()

`sim.step("RK4")`

Steps the simulation forward using the given method. Any arguments for the method can be specified as optional arguments or keyword arguments.

argument	description
method	The method to use. This can be a string or a function. When a string is passed the method in integrator.py of the same name is used. See section (2.2) for such methods.

Optional arguments specific to a method (e.g. tolerance for implicit methods) can be passed into `step()`, which will be transferred to the method when called. In particular, `P_atmos` is common to all integration methods, and is shown below.

argument	description	default
P_atmos	Atmospheric pressure at the surface. This can be either a function with arguments `eta,phiS,eta_x,phiS_x,w` or a numpy array of length `Nx`. See the beginning of section (2.2) for more specifics on the function.	wind_pressure()

Note that wind_pressure() uses the simulation's `P` value, so by default, there is no atmospheric pressure when `P` = 0.

2.1.3 `Simulator1D`: calculate_time_derivatives()

`sim.calculate_time_derivatives(eta, phiS, zeta, zeta_x, zeta_t, P_a)`

Returns the tuple `(eta_t, phiS_t)` of time derivatives of η and Φ^S respectively. Takes the following arguments:

argument	description
eta	free surface height at the given time step; pass simulator.eta if you want the current time derivative.
phiS	free surface velocity potential at the given time step; pass simulator.phiS if you want the current time derivative
zeta	bathymetry (ζ), with 0 corresponding with a depth of -h0
zeta_x	spatial derivative of ζ
zeta_t	time derivative of ζ
P_a	atmospheric pressure at every point, should have the same samples as bathymetry. Expected to be a numpy array or a function. See the beginning of section (2.2) for more specifics on the function.

2.1.4 `Simulator1D`: volume()

`sim.volume()`

Returns the integral

$$\int \eta \; dt$$

over the bounds of the simulation, which represents the volume of water in the simulation, offset by a constant that depends only on the bathymetry. This value should be invariant in the simulation, and can give a means of measuring the accuracy of the simulation.

2.1.5 `Simulator1D`: energy()

`sim.energy()`

Calculates the total energy in the system using the formula

$$\int \eta_t \Phi^S + g\eta^2 \; dt$$

This value should be invariant in the simulation, and can give a means of measuring the accuracy of the simulation.

2.1.6 `Simulator1D`: peak_location()

`sim.peak_location()`

Returns

$$\arg\max_x \eta(x),$$

the value of x that corresponds with the heighest point of the surface. This value is equivalent to the index of the highest eta, times `dx`, and provides a means of finding the approximate position of a solitary wave.

2.1.7 `Simulator1D`: zeta_at()

`sim.zeta_at(x)`

Returns $\zeta(x)$, using a linear interpolation scheme for non-discrete points. calling `zeta_at(i*dx)` is equivalent to evaluating `zeta[i]`.

2.1.8 `Simulator1D`: run_simulation()

`sim.run_simulation(plot_dt, data_dt, directory)`

Runs a simulation, time-stepping until a stop-condition is met, and saving plots and/or data at given intervals of time.

argument	description
saveplot_dt	the timestep between saved plots. This number is rounded to the nearest multiple of the simulation dt. If this value does not round to a positive number, plots are not saved. Plots are saved as PNG files with a name corresponding to the order it is saved in. A plot with number 'i' represents the data at time 'i*saveplot_dt'
savedata_dt	the timestep between saved data. This number is rounded to the nearest multiple of the simulation dt. If this value does not round to a positive number, data is not saved. Data is saved as a json file with name 'dat.json' which is created regardless if data should be saved or not. In the case that data is not saved, only the metadata of the simulation is stored.
directory	the directory to save the files to. This can also include a prefix to the file. If `directory="~/sim/"`, then plots are saved as "[number].png" in the ~/sim/ directory. If `directory="~/sim"`, then plots are saved as "sim[number].png" in the home directory. If `directory=None`, no files are saved.

`run_simulation` has the following optional arguments:

argument	description	default
should_continue	function that determines if a simulation should stop or not. This takes the simulation as an argument and returns a boolean. The simulation is run until should_continue returns false. By default, this is the lambda function `sim: sim.t < 500`, which stops the simulation when a time of 500 is reached	(see description)
integrator	function or string that timesteps the simulation. functions should only take the simulation as an argument and return nothing, modifying the passed simulation. Strings should be the name of a method in Integrator1D, which will be called by the simulation.	"RK4"
save_eta	Parameters for how eta should be saved when data is saved to json. This should be generated using Simulator1D.data_save_params(). If None, then eta is not saved.	None
save_phi	Parameters for how phiS should be saved when data is saved to json. This should be generated using Simulator1D.data_save_params(). If None, then phiS is not saved.	None
loop_callback	this function is called after every simulation step. It should be a void function that takes the arguments `sim, step, plot, data`, where `sim` is the simulator at the step `step` is the integer multiple of dt that the simulation has run `plot` is a boolean representing if the plot was saved this step `data` is a boolean representing if the data was saved this step By default, loop_callback makes a print statement after every 100 time steps.	(see description)
plot_func	A function that is dedicated to plotting and saving the figure. The function is expected to be void and take the arguments (sim, filename).	(see below)
save_json	Whether or not to save the file to json. The metadata of the simulation is saved even if savedata_dt is not positive. Setting save_json to false prevents this.	False
save_netcdf	Whether or not to save the file to netcdf. The netCDF file ignores data truncation specifications of save_eta and save_phi.	True
save_buffer	number of datapoints to buffer in between file-writes. If 0 or 1, then every savedata_dt, the json/netCDF file is opened and written to.	10
cdf_h_invariant	Whether or not h is treated as invariant. If false, then the bathymetry is saved every frame, alongside η and ϕ^S.	True
cdf_Pderiv	The string that should populate the P_deriv field in the netcdf file. By default this will take the value "zero" if this instance's `P` value is 0 and "wind" otherwise.	(see description)
cdf_timeunits	The string that specifies the units of time for the simulation.	"seconds"
cdf_spaceunits	The string that specifies the units of x for the simulation.	"meters"

By default, plot_func is defined as the function

```
def plot_func(sim, filename):
    plt.plot(sim.x, sim.eta, "b")
    plt.plot(sim.x, sim.zeta - sim.h0, "k")
    plt.ylabel("z")
```

```
plt.xlabel("x")
plt.title(f"dx={sim.dx},dt={sim.dt},t={round(sim.t,3)}")
plt.savefig(filename)
plt.clf()
```

where `plt` is `matplotlib.pyplot`.

2.1.9 `Simulator1D`: data_save_params()

`Simulator1D.data_save_params()`

Returns a dictionary of parameters for how to save data from a simulation. The output of `data_save_params()` should be used for aguments `save_eta` and `save_phi` in `run_simulation()`.

argument	description	default
dx	The spatial resolution to save with. If None, then the resolution is the same as the simulation. This value will always be rounded to a whole number multiple of the simulation dx.	None
point_conversion	A boolean that represents if data should be coded as a vector (array), or if the vector should be converted into a list of (x,y) points. If true, then the conversion is made.	False
eps	The tolerance of the save data. The data is rounded to the nearest multiple of eps. That is, with eps=0.001, the data is saved up to the 3rd decimal place. 0 corresponds with no rounding.	0
lin_tol	Only used when point_conversion is true. Specifies a tolerance for which points should not be saved when they are close enough to a linear interpolation of the data. If the points are {(0,0),(0.5,0.5),(1,1)}, any nonnegative tolerance will discard (0.5,0.5). If no points should be discarded, a negative value should be given.	-1
zero_trunc	If a value is less than this distance from 0, the value is truncated to 0 before saving.	0

2.1.10 `Simulator1D`: init_netcdf()

`Simulator1D.init_netcdf(‘‘unforced.nc", True, ‘‘zero")`

Generates a netCDF file of the given filename and populates it with one point in time representing the simulation's current state. If the file already exists, then it is overwritten with a new file.

Returns the netCDF_File object.

argument	description
filename	The name of the file to be saved. Overwrites existing files

argument	description	default
h_invariant	Whether the simulation should be treated as if h does not vary with time	True
P_deriv	Information on how pressure is obtained. Expects "zero", "wind" or "custom". By default this will take the value "zero" if this instance's `P` value is 0 and "wind" otherwise.	(see description)
timeunits	A string representing the units for time.	"seconds"
spaceunits	A string representing the units for spatial coordinates.	"meters"
P	If P_deriv is "zero" then this does nothing. If it is "wind", then the P attribute is set to this value. If it is "custom", then P is the P_a variable at time index 0.	simulator `P`
close	Whether or not this method should close the netcdf file resource after initialization.	True

2.1.11 `Simulator1D`: soliton()

`Simulator1D.soliton(x0,a0,h0,Nx,dx)`

Returns a tuple `(eta,phiS)` corresponding to the intial conditions of η and Φ^S of a soliton at a given point in space.

argument	description
x0	The x coordinate of the soliton, where x=0 corresponds with an index of 0 in the vectorization of η and Φ^S
a0	The amplitude of the soliton
h0	The water deph beneath the soliton
Nx	The number of points in the vectorization of η and Φ^S
dx	The spatial resolution (distance between points)

argument	description	default
g	acceleration due to gravity	9.81

This can be used in the constructor of `Simulator1D` through unpacking:

```
sim = Simulator1D(bathymetry, dt, dx, *Simulator1D.soliton(x0,a0,h0,Nx,dx))
```

2.1.12 `Simulator1D`: KY_bathym()

`Simulator1D.KY_bathym()`

Produces a bathymetry profile similar to Knowles and Yeh's paper. Expects h0 = 1, but the result can be multiplied by the desired h0 if not 1.

argument	description	default
Nx	number of points	2^{14}
dx	spatial resolution (distance between each point)	0.04
s0	nominal slope of the bathymetry	0.002
d0	height of the beach plateau	0.9
gamma	smoothing parameter	0.1
X1	position where the bathymetry should start sloping up	4

2.1.13 `Simulator1D`: KY_sim()

`sim = Simulator1D.KY_sim()`

Returns a new simulator similar to Knowles and Yeh's initial conditions. The bathymetry is produced by `KY_bathym()` and the initial η and Φ^S values are produced by `soliton()`.

argument	description	default
Nx	Number of points (nodes) in the discreteized simulation	2^{14}
dx	Spatial resolution	0.04
dt	Temporal resolution (time step)	0.01
s0	Slope of the bathymetry	1/500
x0	location of the center of the starting soliton	30
a0	amplitude of the soliton	0.1
h0	depth of the water	1

2.1.14 `Simulator1D`: fields

In an instance of Simulator1D, the following fields may be of importance:

argument	description
dt	Temporal resolution (time step) of the simulation. This variable is used by an integrator when stepping the simulation. Most methods will use this time step, but an adaptive method may use a timestep that is smaller. This can be modified externally.
dx	Spatial resolution. This is the distance between two points of a vectorized function of x. This should not be modified externally.
eta	Vectorized η with a spatial resolution `dx` at the current time step. This can be modified externally, but must have the same length (Nx).
phiS	Vectorized Φ^S with a spatial resolution `dx` at the current time step. This can be modified externally, but must have the same length (Nx).
M	Terms in the pertubation expansion of Φ. Calculating the vertical velocity Φ_x scales approximately $O(M^2)$. This can be modified externally.
g	The acceleration due to gravity in this simulation. This can be modified externally.
h0	base still water depth. We approximate the bathymetry as h0 in many calculations. This can be modified externally.
zeta	Vectorized bathymetry (ζ), offset so $\zeta = 0$ corresponds with $z = -\text{h0}$. This can be modified externally, but must have the same length (Nx). Additionally, zeta_x should also be changed to the gradient of the new bathymetry.
zeta_x	Vectorized bathymetry gradient ($\nabla\zeta$). This can be modified externally, but must have the same length (Nx). Additionally, zeta should also be changed to match the new bathymetry.
Nx	Number of points used in the discretization (vectorization) of the simulation along the x-axis. This should not be modified.
sim_length	distance in x that the simulation uses. The vectorizations of η, Φ^S, and ζ have the domain [0, sim_length). This should not be modified.
x	Vectorized domain. It holds that `x[i] = dx*i`. This should not be modified.
kxdb	The double-domain of wave number. When performing an FFT on a function f with spacing dx, the values of the output correspond to the wavenumber by index. If V is such a vectorization of the function f, then $$\big(\mathcal{F}(f)\big)(\texttt{kxdb[i]}) \approx \texttt{FFT}(V)\texttt{[i]}$$ where $\mathcal{F}$ is the continuous fourier transform. This should not be modified.
kappadb	Normalized wavenumber of kxdb. This is equivalent to `abs(kxdb)`. This should not be modified
chi	Vectorized low-pass filter. Any function in the wavenumber domain can apply the filter by pointwise multiplication. This can be modified externally, but kxdb_im must also be modified accordingly.
kxdb_im	The precomputed value $ik\chi(k)$, which is computed from `complex(0,1) * kxdb * chi`. This should only be modified when chi is modified.
t	The time the simulation has run. This is only ever incremented by an integration method inside a `step()` call, and can be freely modified and accessed externally.
P	Wind pressure coefficient. The surface pressure is found as $P_a = \texttt{P} \cdot \eta_x$. This can be modified externally.

2.2 integrator.py

This python file has the class `Integrator1D` which contains only static members. Each of which is a function that takes a `Simulator1D` instance, an atmospheric pressure argument `P_atmos`, and potential optional arguments.

`P_atmos` can either be a numpy array or a function that takes in the arguments `eta,phiS,eta_x,phiS_x,w` and returns a numpy array. The array should have a length equal to the number of nodes `Nx` used in the simulator, which matches the length of the `bathymetry` array passed into the constructor. For example, when measuring the wind effect, one may consider

$$P_a = P\frac{d\eta}{dx}$$

for some constant P. Such a function can be expressed as

```
def P_atmos(eta,phiS,eta_x,phiS_x,w):
    return P * eta_x
```

2.2.1 `Integrator1D`: euler

Referenced by string `"euler"`. Makes one derivative calculation per step, using the method:

$$y_{n+1} = y_n + hf'(y_n)$$

2.2.2 `Integrator1D`: RK4

Referenced by string `"RK4"`. Makes 4 derivative calculation per step, using the classic 4 step, 4th order, Runge-Kutta method with the Butcher tableau:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

2.2.3 `Integrator1D`: implicit_midpoint

Referenced by string `"implicit_midpoint"`. Uses the implicit midpoint rule:

$$y_{n+1} = y_n + hf\left(\frac{y_n + y_{n+1}}{2}\right)$$

This equation is solved using fixed point iteration after an initial guess from euler's method.

Takes additional arguments:

argument	description	default
max_iters	The most iterations used to achieve the desired tolerance, after which, RK4 is defaulted to.	100
tol	The tolerance allowed for the iteration to stop.	10^{-10}

2.2.4 `Integrator1D`: AM1

Referenced by string `"AM1"`. Uses the one step Adams-Moulton method, the implicit trapezoidal rule:

$$y_{n+1} = y_n + h\frac{f(y_n) + f(y_{n+1})}{2}$$

This equation is solved using fixed point iteration after an initial guess from euler's method.

Takes additional arguments:

argument	description	default
max_iters	The most iterations used to achieve the desired tolerance, after which, RK4 is defaulted to.	100
tol	The tolerance allowed for the iteration to stop.	10^{-10}

2.2.5 `Integrator1D`: DIRK3

Referenced by string `"DIRK3"`. Uses Nørsett's 3 stage, 4th order diagonally implicit Runge-Kutta method withe the Butcher tableau:

$$\begin{array}{c|ccc}
x & x & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} - x & x & 0 \\
1 - x & 2x & 1 - 4x & x \\
\hline
 & \frac{1}{6(1-2x)^2} & \frac{3(1-2x)^2-1}{3(1-2x)^2} & \frac{1}{6(1-2x)^2}
\end{array}$$

where $x = 1.06858$. This equation is solved using fixed point iteration for each stage, where each stage has an initial guess of the former stage, with the first stage's initial guess as the derivative at y_n. For example, if k_1 and k_2 are the results of the first and second stage respectively, the second stage solves

$$k_2 = f\left(y_n + \left(\frac{1}{2} - x\right) hk_1 + xhk_2\right)$$

with an initial guess $k_2 = k_1$.

Takes additional arguments:

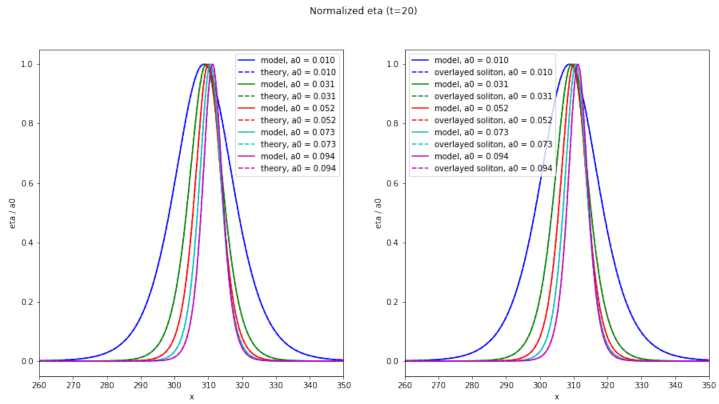
argument	description	default
max_iters	The most iterations used to achieve the desired tolerance, after which, RK4 is defaulted to.	100
tol	The tolerance allowed for the iteration to stop.	10^{-10}

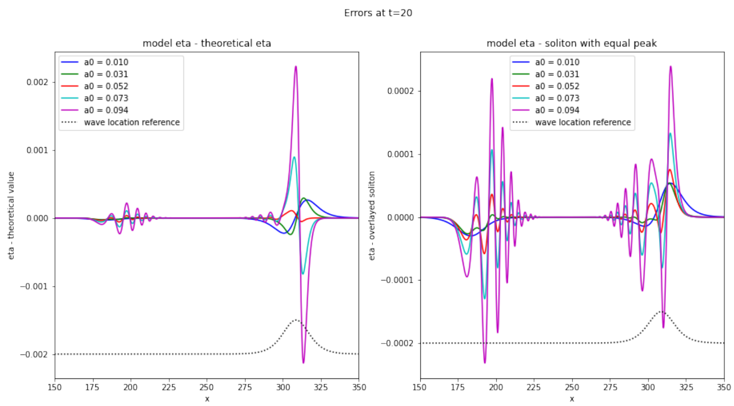
3 Validation

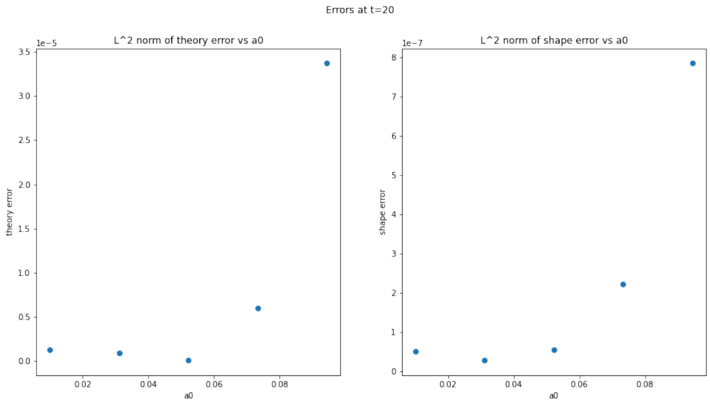
We use the properties of the soliton wave to validate our model. In the flat, unforced case, we run the model up to 20 seconds with $dx = 0.03$, $dt = 0.0025$ using the classic 4^{th} order Runge-Kutta scheme and compare the results with both a soliton shifted by

$$\Delta x = tv_p = 20\sqrt{g(h_0 - a_0)}$$

and a soliton overlayed on the model such that the peaks match.







Errors at t=20
L^2 norm of theory error vs a0
1e−5
theory error
a0
L^2 norm of shape error vs a0
1e−7
shape error
a0

4 Examples

4.1 `KY.py`

This file defines one function, `sim_KY`, which runs a simulation very similar to that of the Knowles and Yeh (2018) simulation. Run using `python KY.py`. When run in this manner, `sim_KY` is called with the default arguments. When imported in python using `import KY`, the function is not called, but can freely be called by the user. `sim_KY` has the following optional arguments:

argument	description	default
Nx	number of points (nodes) in the simulation.	2^{14}
a0	amplitude of the soliton initial condition	0.1
h0	base water depth. (depth at zeta = 0)	1.0
X0	position (not vector index) of the soliton relative to the center of the simulation space. Negative values correspond to the left half of the simulation space.	-280
d0	height of the beach plateau (on the right side)	0.9
s0	nominal slope of the bathymetry.	$\frac{1}{500}$
dx	spatial resolution (distance between points/nodes in simulation space)	0.04
dt	Time step of the simulation	0.01
Xt	Distance from the soliton wave crest to the beach toe. Positive values mean that the beach toe is in front of the wave	50
mass_err_crit	Largest deviation in mass (calculated from `volume()`) allowed before the simulation is terminated	0.5
energy_err_crit	Largest deviation in energy (calculated from `energy()`) allowed before the simulation is terminated	0.5
tmax	Largest time in simulation allowed before termination.	500
v	The low-pass filter. Takes a number to cut off all wavenumbers greater than v*max(k), or takes a function that maps k and max(k) to how much the corresponding amplitude should be scaled. See the constructor of Simulator1D.	0.7
M	The number of terms in the pertubation expansion to simulate	5
g	Acceleration due to gravity	9.81
tsave	time in between plot saves. Zero or negative corresponds to no saving. This value is rounded to a multiple of dt.	1
save_dir	the directory/file prefix for the saved plots. By default, this is "./KY_dx[dx]_dt[dt]_s[s0]_a[a0]_plot", where each bracketed value is replaced by the relevant quantity.	(see description)
gui	Whether or not to run this program with matplotlib's plotting features. Use `gui=False` when running with no GUI access.	True

5 Application to the sloped-forced case

We attempt to recreate the results in *Wind-induced changes to shoaling surface gravity wave shape*, which has the following parameters for non-dimensionalization:

$$\varepsilon_0 = \frac{H_0}{h_0}, \quad \mu_0 = \left(\frac{h_0}{L_0}\right)^2, \quad P_0 = \frac{P}{\rho_w g L_0}, \quad \gamma_0 = \frac{L_0}{L_b} \qquad (2.5)$$

where H_0 is the initial wave height (a_0 in Knowles and Yeh (2018)), L_0 is the effective half-width (as in Knowles and Yeh (2018)), defined to be $L_0 = h_0\sqrt{\frac{4h_0}{3H_0}}$ in (3.2), and $L_b = \frac{h_0}{\beta}$ is the length of the beach slope (β is the value of the slope).

In (2.6), the following nondimensionalization is used:

$$x = L_0 x', \quad h = h' h_0, \quad z = h_0 z', \quad \eta = H_0 \eta'$$

$$t = \frac{t' L_0}{\sqrt{g h_0}}, \quad \phi = \phi' H_0 L_0 \sqrt{\frac{g}{h_0}}$$

To relate to our simulation, we examine the boundary condition involving pressure:

$$0 = \varepsilon_0 P_0 \frac{\partial \eta'}{\partial x'} + \eta' + \frac{\partial \phi'}{\partial t'} + \frac{\varepsilon_0}{2}\left(\left(\frac{\partial \phi'}{\partial x'}\right)^2 + \frac{1}{\mu_0}\left(\frac{\partial \phi'}{\partial z'}\right)^2\right)$$

We notice $\partial \phi' = \partial \phi \frac{1}{H_0 L_0}\sqrt{\frac{h_0}{g}}$, $\partial t' = \partial t \frac{\sqrt{g h_0}}{L_0}$, $\partial \eta' = \frac{\partial \eta}{H_0}$, $\partial x' = \frac{\partial x}{L_0}$, so

$$\frac{\partial \phi'}{\partial t'} = \frac{\partial \phi}{\partial t} \cdot \frac{1}{H_0 g}, \qquad \frac{\partial \eta'}{\partial x'} = \frac{\partial \eta}{\partial x} \cdot \frac{L_0}{H_0}$$

$$\frac{\partial \phi}{\partial t} = -\varepsilon_0 P_0 L_0 g \frac{\partial \eta}{\partial x} - H_0 g \eta' - \frac{\varepsilon_0 H_0 g}{2}\left(\left(\frac{\partial \phi'}{\partial x'}\right)^2 - \frac{1}{\mu_0}\left(\frac{\partial \phi'}{\partial z'}\right)^2\right)$$

The parameters provided are

$$\eta_0, \quad \mu_0, \quad \frac{P}{\rho_w g L_0 \varepsilon_0} = \frac{P_0}{\varepsilon_0}, \quad \beta$$

Our P-value (simply the coefficient to the η_x term), what will be denoted as $\bar{P}$, is related to P_0/ε_0 as

$$\bar{P} = \varepsilon_0 P_0 L_0 g = \frac{H_0^2}{h_0^2}\frac{P_0}{\varepsilon_0} L_0 g = \frac{P_0}{\varepsilon_0} \cdot \frac{H_0^2}{h_0}\sqrt{\frac{4h_0}{3H_0}} g = \frac{P_0}{\varepsilon_0} \cdot 2\frac{H_0^{\frac{3}{2}}}{\sqrt{3h_0}} g$$

The bathymetry starts at $h_0 = 1$ and slopes up with slope β, up to $h = 0.1$, (corresponding to $d_0 = 0.9$). The model has a 20 unit distance between the left boundary and the start of the slope. The soliton is placed on that boundary, which we cannot do because our simulation mirrors onto the double domain and does not wrap onto it. Instead, we use $X_0 = 120$ as our soliton initial position, and $X_1 = 140$ as the location of the beach toe (exact position, not relative to X_0). This means that our parameters to put into `wind_effect` are

$$dx = 0.1, \quad dt = 0.01, \quad a_0 = 0.2 \text{ (from } \varepsilon_0 = 0.2 \text{ in Table 1, p. 9)}$$

$$s_0 = 0.01, 0.015, 0.02, 0.025 \text{ (from } \beta \text{ taking those values in Table 1)}$$

$$X_0 = 120, \;\; X_1 = 140, \;\; N_x \text{ large enough to have at least 20 units of beach plateau}$$

$$\begin{aligned}\bar{P} = \frac{P_0}{\varepsilon_0} \cdot 2\frac{0.2^{\frac{3}{2}}}{\sqrt{3}} \cdot 9.81 = 1.013\frac{P_0}{\varepsilon_0} \text{ (for each pressure value } \frac{P_0}{\varepsilon_0} \text{ in Table 1)} \\ = 0.00317, 0.00633, 0.0127, 0.0253, 0.0507\end{aligned}$$