

Numerical continuation for the spatial model of vegetation-water-herbivore system

Joydeep Singha¹, Hannes Uecker², and Ehud Meron¹

¹The Swiss Institute for Dryland Environmental and Energy Research, BIDR, Ben-Gurion University of the Negev, Sede Boqer Campus, Israel

²Institut für Mathematik, Universität Oldenburg, Germany

Corresponding authors: joydeepsingha105@gmail.com

Abstract

We describes the direct numerical simulation codes for the manuscript "*Traveling vegetation-herbivore waves may sustain ecosystems threatened by droughts and population growth*" by Singha et al.

1 Brief description of the model

In [SUM24] we consider a spatially explicit model that describes the coupled dynamics of vegetation, soil water, and herbivores in dryland ecosystems. The model captures two critical stressors—limited water availability and herbivory—and incorporates feedback mechanisms that can give rise to spatial self-organization. In particular, the model accounts for a behavioral component of herbivore movement, referred to as “vegetaxis,” in which herbivores are attracted to denser vegetation patches. This feature plays a crucial role in shaping vegetation–herbivore patterns.

Let $B(x, t)$, $W(x, t)$, and $H(x, t)$ represent the vegetation biomass density, soil water content, and herbivore biomass density, respectively, at spatial location x and time t . The governing equations are:

$$\begin{aligned}\partial_t B &= \Lambda BW(1 + EB)^2 \left(1 - \frac{B}{K_B}\right) - M_B B + D_B \nabla^2 B - G(B)H, \\ \partial_t W &= P - \frac{NW}{1 + RB/K_B} - \Gamma BW(1 + EB)^2 + D_W \nabla^2 W, \\ \partial_t H &= -M_H H + AG(B)H \left(1 - \frac{H}{K_H}\right) - \nabla \cdot \mathbf{J}_H.\end{aligned}$$

Vegetation growth depends on local water availability and is enhanced by root development, modeled through the nonlinear term $\Lambda BW(1 + EB)^2$. Growth is also limited by saturation effects and competition, captured by the factor $1 - B/K_B$. Vegetation biomass decreases due to natural mortality at rate M_B and through consumption by herbivores, where the consumption rate $G(B)$ follows a saturating function:

$$G(B) = \frac{\alpha B}{\beta + B},$$

with α representing the maximum per capita consumption rate and β the biomass at which half-maximum consumption occurs.

The soil water balance includes a constant precipitation input P , reduced by evaporation, which is itself mitigated by shading from vegetation ($NW/(1 + RB/K_B)$). Plants extract water from the

soil via a nonlinear uptake term proportional to $BW(1 + EB)^2$, while water diffuses laterally at a rate D_W .

Herbivores grow by consuming vegetation, reproducing at a rate proportional to the product $G(B)H$, moderated by density dependence via the factor $1 - H/K_H$. Herbivore losses are due to natural mortality at rate M_H and spatial redistribution. Movement is governed by the flux term:

$$\mathbf{J}_H = -D_R(B)\nabla H + HD_V(B)\nabla B,$$

where $D_R(B)$ describes biomass-dependent random movement:

$$D_R(B) = D_{HH} \frac{H\xi^2}{\xi^2 + B^2},$$

and $D_V(B)$ describes biased movement up vegetation gradients (vegetaxis):

$$D_V(B) = D_{HB} \frac{B}{\kappa + B}.$$

In this framework, herbivores tend to move more rapidly in bare-soil regions and slow down as they approach vegetation, while also being drawn toward denser patches. This behavior mimics an exploitation strategy, allowing herbivores to efficiently locate and graze on vegetation.

This system of equations can give rise to a rich variety of spatial and temporal patterns, including stationary vegetation stripes, localized herbivore aggregations, and traveling vegetation–herbivore waves. These emergent behaviors are critical for understanding the resilience of dryland ecosystems under increasing stress due to climate change and population-driven grazing pressure.

We use **pyPDE** package in python to produce all the results related to the direct numerical simulations (DNS) in Figures 8, 11, 12, 13 in [SUM24]. Before proceeding it is advised that the user should visit the official webpage for this python package for step by step instruction for installing it, all of its dependancies. We advise that the user should go through the example problems, specially the construction of the custom PDE class, given in there and run these examples using the installed package to get familer with the way it works.

The folder **Direct numerical simulation** contains all the necessary files (see Table 1 for an overview).

Table 1: Scripts and functions in **bwhcont/**

file	purpose, remarks
DNSpypde1D2D.py	used for DNS in one dimensional domain in Figs. 8, 11, and in two dimensional domain in Fig. 13 in [SUM24]
DNSpypde2D.py	used for DNS two dimensional domain for Fig. 12 in [SUM24]
DNS_plot.py	post processing for h5py data files
BW.mp4, BH.mp4	animation of evolution of B, W, H travelling wave
out_data_t30.h5py	example output data from DNSpypde1D2D.py
DNSdedalus.py	dedalus implementation of the model

In **Direct numerical simulation/DNSpypde1D2D.py**, the class **BWH** contains all the details of the model. We keep the parameters of the system as global variable in the beginning (see listing 1).

```

31 #defining the PDE class object for the BWH system containing the paramters
32 #B = vegetation, W = soil water, H = herbivore
33 class BWH(PDEBase):
34     def __init__(self, P = 120, bc = "auto_periodic_neumann"):
35         super().__init__()
36         self.lamda = 0.5           #vegetation growth rate
37         self.E = 10.0             #root to shoot ratio
38         self.K = 0.9              #maximum vegetation per unit area

```

```

39     self.M = 11.4                #vegetation mortality rate
40     self.DB = 1.2
41
42     self.N = 20.0                #evaporation rate
43     self.R = 0.01                #reduced evaporation due to vegetation
44     self.GamW = 10.0             #soil water uptake rate
45     self.DW = 150.0              #lateral soil water diffusion coefficient
46
47     self.mH = 0.06                #herbivore mortality rate
48     self.GamH = 0.3              #herbivore proliferation rate
49     self.AlphaH = 0.6            #vegetation consumption rate
50     self.BetaH = 0.82            #satiation biomass
51     self.K_H = 150.0             #maximum herbivore per unit area
52
53     self.DHH = 400                #diff. coeff. for random motility
54     self.Zeta = 0.001            #reference biomass for 50% random motility drop
55     self.DHB = 700                #diff. coeff. for vegetaxis motility
56     self.k = 0.0001              #reference biomass for 50% vegetaxis motility drop
57
58     self.P = P                    #precipitation
59     self.bc = bc                  #boundary condition

```

Listing 1: Direct numerical simulation/DNSpypde1D2D.py parameters of the model ;

The initial condition is specified in `get_initial_state` function (see listing 2).

```

59     #definition of the function for initial condition
60     #if a special initial condition is used it can be input through Bini, Wini, Hini
    during call
61     #def get_initial_state(self, grid, storage, Bini, Wini, Hini):
62
63     #if random initial condition is used then no need input of Bini, Wini, Hini
64     def get_initial_state(self, grid, storage, Lx, mesh):
65         # Random initial condition
66         B = ScalarField.random_uniform(grid)
67         W = ScalarField.random_uniform(grid)
68         H = ScalarField.random_uniform(grid)
69
70         #if initial condition is input during call, modify B, W, H from Bini, Wini, Hini
        respectively
71         #one can also modify the random initial condition by accessing B, W, H data as
        below
72         #B.data = Bini
73         #W.data = Wini
74         #H.data = Hini
75
76         #example of changing initial condition
77         x = np.linspace(0, Lx, mesh)
78         B.data = 0.5 + (0.01 * np.sin(2 * np.pi * 4 * x/Lx))
79         W.data = 1.2
80         H.data = 0.25
81
82         #this labels are useful when output data file is analysed later
83         B.label = "Plants"
84         W.label = "Water"
85         H.label = "Herbivore"
86
87
88     return FieldCollection([B, W, H])

```

Listing 2: Direct numerical simulation/DNSpypde1D2D.py initial condition;

The rate equation is specified in `evolution_rate` function (see listing 3).

```

95     #rate equations for B, W, H
96     def evolution_rate(self, state, t = 0):

```

```

97     B, W, H = state
98
99
100     #rate equations implemented in one dimension
101     B_t = (self.lamda * B * W * ((1.0 + (self.E * B))**2) * (1.0 - (B/self.K))) - (
self.M * B) - (self.AlphaH * H * B/(self.BetaH + B)) + (self.DB * B.laplace(self.bc)
)
102     W_t = self.P - (self.N * W/(1.0 + (self.R * B/self.K))) - (self.GamW * B * W *
(1.0 + (self.E * B))**2) + (self.DW * W.laplace(self.bc))
103     H_t = (-self.mH * H) + (self.GamH * self.AlphaH * H * B * (1.0 - (H/self.K_H))/(
self.BetaH + B)) - ((self.DHB * self.k/(self.k + B)) * ((H.gradient(self.bc) * B.
gradient(self.bc)))) + ((self.DHB * H * self.k/(self.k + B)**2) * B.gradient(self.bc
)**2) - ((self.DHB * H * self.k/(self.k + B)) * B.laplace(self.bc)) + ((self.DHH *
self.Zeta**2/(self.Zeta**2 + B**2)) * H.laplace(self.bc)) - ((self.DHH * 2 * self.
Zeta**2 * B/(self.Zeta**2 + B**2)**2) * ((H.gradient(self.bc) * B.gradient(self.bc))
))
104
105     #rate equations implemented in two dimension
106     #B_t = (self.lamda * B * W * ((1.0 + (self.E * B))**2) * (1.0 - (B/self.K))) - (
self.M * B) - (self.AlphaH * H * B/(self.BetaH + B)) + (self.DB * B.laplace(self.bc)
)
107     #W_t = self.P - (self.N * W/(1.0 + (self.R * B/self.K))) - (self.GamW * B * W *
(1.0 + (self.E * B))**2) + (self.DW * W.laplace(self.bc))
108     #H_t = (-self.mH * H) + (self.GamH * self.AlphaH * H * B * (1.0 - (H/self.K_H))
/(self.BetaH + B)) - ((self.DHB * self.k/(self.k + B)) * ((H.gradient(self.bc) @ B.
gradient(self.bc)))) + ((self.DHB * H * self.k/(self.k + B)**2) * (B.gradient(self.
bc) @ B.gradient(self.bc))) - ((self.DHB * H * self.k/(self.k + B)) * B.laplace(self
.bc)) + ((self.DHH * self.Zeta**2/(self.Zeta**2 + B**2)) * H.laplace(self.bc)) - ((
self.DHH * 2 * self.Zeta**2 * B/(self.Zeta**2 + B**2)**2) * ((H.gradient(self.bc) @
B.gradient(self.bc))))
109
110     return FieldCollection([B_t, W_t, H_t])

```

Listing 3: Direct numerical simulation/DNSpydde1D2D.py initial condition;

. We also add a numba implementation of the model so that it can be run in parallel where the domain is divided in multiple cores (see listing 5). We advise the user to visit the relevant webpage for py-pde.

```

108     #numba implementation to run this code parallely. Python packages such as Numba, Numba
-mpi, mpi4py are required to
109     #run it parallely. We suggest to consult the py-pde documentation page mentioned in
the beginning.
110     def _make_pde_rhs_numba(self, state):
111         """numba implementation of the PDE"""
112         #parameter values as defined before
113         lamda = self.lamda
114         E = self.E
115         K = self.K
116         M = self.M
117         DB = self.DB
118
119         N = self.N
120         R = self.R
121         GamW = self.GamW
122         DW = self.DW
123
124         mH = self.mH
125         GamH = self.GamH
126         AlphaH = self.AlphaH
127         BetaH = self.BetaH
128         K_H = self.K_H
129
130         DHH = self.DHH
131         Zeta = self.Zeta

```

```

132     DHB = self.DHB
133     k = self.k
134
135     P = self.P
136
137     #defining the gradient, square of gradient and laplace operation
138     #these are defined according to py-pde methods. Please consult the py-pde
documentation to get familer.
139     gradient = state.grid.make_operator("gradient", bc = self.bc)
140     gradient_squared = state.grid.make_operator("gradient_squared", bc = self.bc)
141     laplace = state.grid.make_operator("laplace", bc = self.bc)
142
143     #defining dot operation, required for two dimensional implementation
144     #dot = VectorField(state.grid).make_dot_operator()
145
146     #numba implementation of the rate equations
147     @nb.jit(nopython=True)
148     def pde_rhs(state_data, t):
149         B = state_data[0]
150         W = state_data[1]
151         H = state_data[2]
152
153         rate = np.empty_like(state_data)
154
155
156         #rate equations implemented in one dimension
157         rate[0] = (lamda * B * W * ((1.0 + (E * B))**2) * (1.0 - (B/K))) - (M * B) -
(AlphaH * H * B/(BetaH + B)) + (DB * laplace(B))
158         rate[1] = P - (N * W/(1.0 + (R * B/K))) - (GamW * B * W * (1.0 + (E * B))
**2) + (DW * laplace(W))
159         rate[2] = (-mH * H) + (GamH * AlphaH * H * B * (1 - (H/K_H))/(BetaH + B)) -
((DHB * k/(k + B)) * ((gradient(H) * gradient(B)))) + ((DHB * H * k/(k + B)**2) *
gradient_squared(B)) - ((DHB * H * k/(k + B)) * laplace(B)) + (((DHH * Zeta**2)/((
Zeta**2) + (B**2))) * laplace(H)) - ((DHH * 2 * (Zeta**2) * B/((Zeta**2) + (B**2))
**2) * (gradient(H) * gradient(B)))
160
161
162         #rate equations implemented in two dimension
163         #rate[0] = (lamda * B * W * ((1.0 + (E * B))**2) * (1.0 - (B/K))) - (M * B)
- (AlphaH * H * B/(BetaH + B)) + (DB * laplace(B))
164         #rate[1] = P - (N * W/(1.0 + (R * B/K))) - (GamW * B * W * (1.0 + (E * B))
**2) + (DW * laplace(W))
165         #rate[2] = (-mH * H) + (GamH * AlphaH * H * B * (1 - (H/K_H))/(BetaH + B)) -
((DHB * k/(k + B)) * (dot(gradient(H), gradient(B)))) + ((DHB * H * k/(k + B)**2) *
gradient_squared(B)) - ((DHB * H * k/(k + B)) * laplace(B)) + (((DHH * Zeta**2)/((
Zeta**2) + (B**2))) * laplace(H)) - ((DHH * 2 * (Zeta**2) * B/((Zeta**2) + (B**2))
**2) * dot(gradient(H), gradient(B)))
166         return rate
167
168     return pde_rhs

```

Listing 4: Direct numerical simulation/DNSpypde1D2D.py numba implementation for optimisation;

. The length of the domain, mesh size and grid construction with periodic boundary condition after the PDE class is completely defined (see listing ??).

```

171 #domain size in one dimension
172 #to implement in two dimension, use Ly for domain length in one dimension
173 Lx = 4*np.pi
174 mesh = 1024
175
176 #defining the grid with periodic boundary condition in one dimension
177 grid = CartesianGrid([[0,Lx]], [mesh], periodic = True)

```

Listing 5: Direct numerical simulation/DNSpypde1D2D.py domain details and grid;

. We then enter value(s) of precipitation in list. For the value(s), P , the PDE class is initialised and the simulation is run in the final step (see listing 6).

```

202 P_list = [120]          #precipitation (use 70mm/y for the two dimensional case)
203 #storage = MemoryStorage() #useful when using jupyter notebook for the analysis of data
    #without saving a data file
204
205 for p in P_list:
206     eq = BWH()           #initialising the PDE class
207     eq.P = p             #P input
208     storage = FileStorage("out_data_t30.h5")      #output data file
209     trackers = ["progress", storage.tracker(0.5)] #tracker to see the duration of
    simulation
210
211     #defining the solver : Runge-Kutta with adaptive time stepping with domain divided
    into 4 cores
212     solver_mpi = ExplicitMPISolver(eq, scheme = 'rk', decomposition=2, backend = 'auto',
    adaptive = True)
213
214     #controller for parallel computation specifying the total time duration
215     controller_mpi = Controller(solver_mpi, t_range = 30, tracker = trackers)
216
217     #when Bini, Hini, Wini is provided from data or prepared specially as above then the
    function call to "get_initial_state" with Bini, Wini, Hini as input
218     #state = eq.get_initial_state(grid, storage, Bini, Wini, Hini)
219
220     #generally if random initial condition is used, then no need for Bini, Wini or Hini
    as input
221     state = eq.get_initial_state(grid, storage, Lx, mesh)
222
223     #to run the model !
224     sol = controller_mpi.run(state)

```

Listing 6: Direct numerical simulation/DNSpypde1D2D.py simulation run for a given precipitation(s);

. The DNS code generates data files in **h5py**. We have provided a **jupyter notebook**, **DNS_plot.ipynb** where methods for loading reading **h5py** is given. With the current parameter values given in the listings, we can obtain the patterned solution at $P = 120\text{mm/y}$. We have also provided the "out_data.h5py" and animations, **BW.mp4** and **BH.mp4** obtained using the plot commands in **DNS_plot.ipynb**. The user is advised to run the codes as background task or in a cluster server as the iterations are slow.

2 Instructions for generating the DNS figures in the Manuscript

Figure 8 in [SUM24] is obtained from simulation of one dimensional implementation where the initial condition is homogeneous with an added perturbation. The steps to obtain Figure 11 and 12 in [SUM24] are similar with the difference that Fig. 11 is for an one dimensional domain while Fig. 12 is in two dimension. For Figure 11, At first the precipitation range is identified where a single patch of vegetation spot is stable. The conventional way to do it is to first obtain a patterned solution in the " B, W " system without " H ". The code **DNSpypde1D2D.py** can be easily changed to a simulate " B, W " model by commenting out the " H " equation and removing the consumption term from the " B " rate equations. From the output data file, a domain length of size of a wave length containing one wave is chosen and kept as it is while the data for the rest of the domain is changed to zero. This modified data file is used as initial condition for " B, W " system with a slightly lower P . This process is repeated until the single patch initial condition was stable as it is. During this process we reduce P by a value of 2 mm/y in each trial step until the desired solution of stable patch of B is obtained. The single spot solution used as an initial condition for the full " B, W, H " system. Initial condition for the " H " is given as a Gaussian function. We have provided all the necessary parts to perform this task in the code as comments. Figure 13 of [SUM24] is obtained in similar step as. However here we

first obtain the spot pattern in the " B, W " model. In the same way as before we isolate a spot and identify P value where only this spot is stable. We then introduce " H " through a Gaussian function for the complete " B, W, H " where the stable single spot solution was used as an initial condition for B and W . The steps to obtain Figure 12 of [SUM24] is simpler. We provide `DNSpypde2D.py` in the same folder where a full two dimensional implementation is provided for convenience. The model is simply simulated from a homogeneous initial condition with an added perturbation of random noise to a sufficiently large amount of time when desired solutions become stable. In order to run the system within reasonable time the user advised to use a cluster server where multiple cores are available. The particular pyplot commands to generate these are available from the corresponding author of [SUM24] upon request.

The system can also be simulated using the `dedalus` package in python but we have not used it explicitly to generate the figures in [SUM24]. We have only used it initially in order to verify a few results related to the bifurcation diagrams. In one dimension we could not make a parallel implementation resulting in a slow simulation time. For this reason we did not pursue this package and switched to `pypde` for all DNS needs for [SUM24]. However we have provided a basic `dedalus` implementation in `DNSdedalus.py` in case the user is interested.

References

- [SUM24] J. Singha, H. Uecker, and E. Meron. Traveling vegetation–herbivore waves may sustain ecosystems threatened by droughts and population growth, 2024. Preprint.