The partial differential equations for the production, diffusion, and membrane binding of DivIVA are as described in the supplementary note of the present work, or as Equations (1) and (2) below.

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |

where *H(m)* is the Hill’s function for the synergistic effects of membrane binding,

|  |  |
| --- | --- |
|  | (3) |

Parameter *D* is the diffusion coefficient, production and degradation rates of the diffusible DivIVA are *kp* and *kd*, respectively.  *kb* is the maximum membrane binding rate, *ks* is the corresponding rate for DivIVA to dissociate from the membrane. We further restricted the production of *u* to be at one end of *E. coli*, and the membrane binding is allowed only at the two poles. In our numerical simulation, we simplified the problem to an array of 1-dimensional grids, with a modest 10 grids in total. *kp* was set to zero except for the first grid where PopZ is localized. Similarly, *kb* is nonzero only at the first and the last grids. In this model, *m* is not diffusible, and it only exists at the two ends.

The diffusion in Eq (1) was solved using a simple finite difference scheme, and the time evolution was propagated using the Euler’s explicit method. Namely, we have

and

where

and

with unless for the first grid, and except for the first and last grids. is the matrix element of a 10×10 matrix , which describes the finite-difference propagation:

where

and we have set *D* = 0.32 μm2/s. The full length of an E. coli cell was set as 2 μm, and with 10 grids, we have = 0.2 μm. Assuming that diffusion is faster than any other protein events, we set as a small fraction of the diffusion time scales,

Starting with a zero concentration for both *u* and *m* in all grids, we propagated the system for 1 hour and record the final asymmetry ratio, defined as,

In the simulation we set nM as the concentration units. The simulation programs were developed under a recent version of Spyder (3.3.6 or newer) with both Python 3.6 and 3.7, under Mac OS 10.14.6 and CentOS Linux 7.

We include 6 short python programs with the present work.

1. scan\_k.py
2. readdata\_scan\_k.py
3. scan\_ss.py
4. readdata\_scan\_ss.py
5. single.py
6. readdata\_single.py

### scan\_k.py

scans parameters and by fixing the steady-state value

It’s for scanning parameter ( and )

input: (set in the program)

a. initial condition of *u* and *m* (all are zero)

b. set parameter: *K, n, D,* ,

c. and : scanned

d. = /, =

output:

a. result (list type)

### readdata\_scan\_k.py

reads the output of scan\_k.py and generates plots.

input:

a. result (list type)

output: heat maps of the following, as a function of the scanned and

a. M ratio result.png (the head/tail ratio for *m*)

b. U ratio result.png (the head/tail ration for *u*)

c. U + M ratio result.png (the head/tail ratio for total DivIVA)

### scan\_ss.py

scans parameters uss and mmaxss by fixing the ks and kb

It’s for scanning parameters and .

input: (set in the program)

a. initial condition of *u* and *m* (all are zero)

b. set parameter: *K, n, D,* and

c. , scanned

d. = /, =

output:

a. result (list type)

### readdata\_scan\_ss.py

reads the output of scan\_ss.py and generates plots.

input:

a. result (list type)

output: heat maps of the following, as a function of the scanned ks and kd

a. M ratio result.png (the head/tail ratio for M)

b. U ratio result.png (the head/tail ration for U)

c. U + M ratio result.png (the head/tail ratio for total DivIVA)

### single.py

produce detailed dynamics and spacial data of one selected parameter.

input: (set in the program)

a. initial condition of *u* and *m* (all are zero)

b. set parameter: *K, n, D,*, ,, .

c. = /, =

Output:

a. result (list type)

### readdata\_single.py

reads the output of single.py and generates plots.

input:

a. result (list type)

output:

a. M finial state.png

b. U finial state.png

c. M +U finial state.png

d. M time series.png

e. U time series.png

f. heat\_map\_M.png

g. heat\_map\_U.png