Supplemental Material

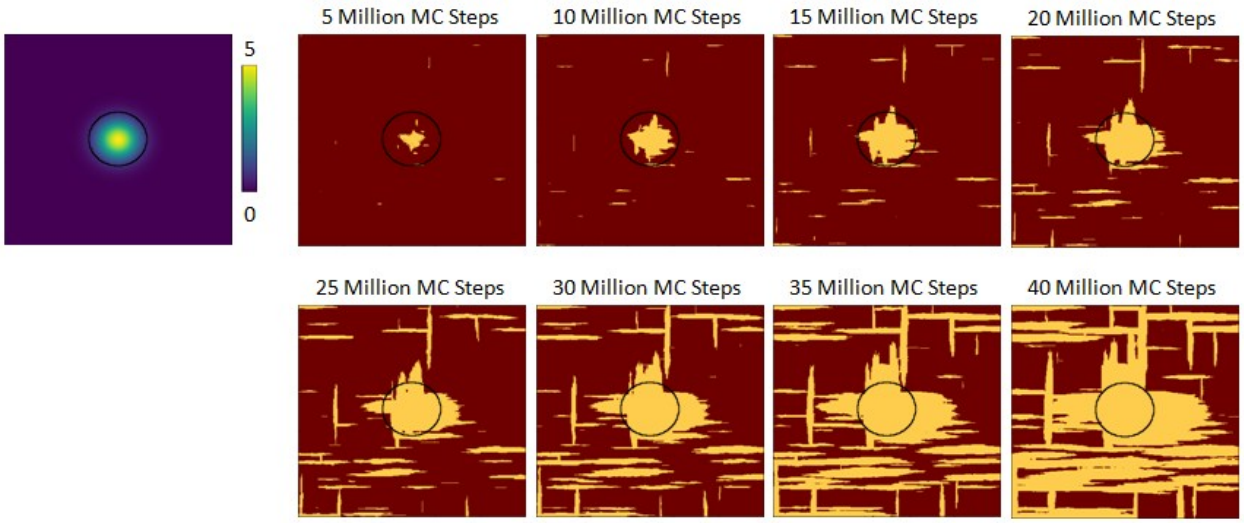

Figure S1. MC simulation employing a large 2D Gaussian function surface energy configuration and accompanying simulation snapshots. The region of influence of the high surface energy is circumscribed with a black circle that is superimposed on simulation results to serve as a visual guide.

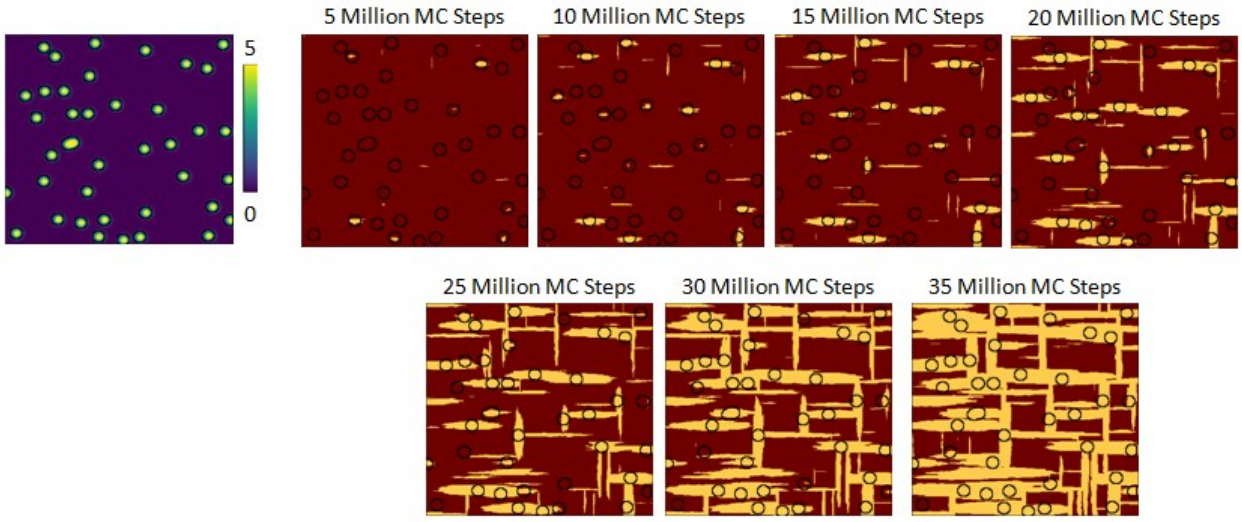

Figure S2. MC simulation employing randomly distributed small 2D Gaussian functions for the surface energy configuration and accompanying simulation snapshots. The regions of influence of the high surface energy are circumscribed with black circles that are superimposed on simulation results to serve as a visual guide.

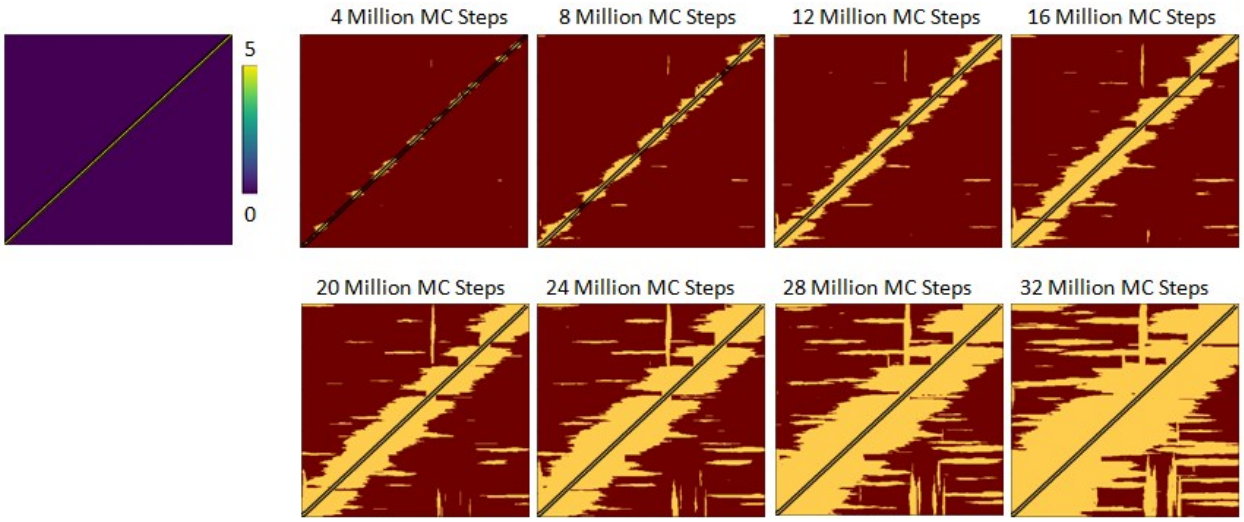

Figure S3. MC simulation employing a long diagonal surface energy configuration and accompanying simulation snapshots. The region of influence of the high surface energy is circumscribed with two black lines that are superimposed on simulation results to serve as a visual guide.

```
## ============== MC Crystallization Simulation - Version 4.0 ============== ##
# ========================================================================
#  Version 1.0 is a 0th order random crystallization simulation, applying
#  arbitrary probabilities to the relevant phenomenological events of
#  nucleation, growth, and branching. Probabilities are adjusted empirically to
#  track trends of observed experimental data.
#
#  Version 2.0 is a more sophisticated Monte Carlo style simulation, applying a
#  pseudo-Metropolis method to evaluate trial MC moves. Energy benefits given
#  for forming "bonds", and energy penalties for forming "interfaces" are
#  assigned empirically to reproduce observations.
#
#  Version 2.1 enhances simulation by only selecting vacant sites for MC step
#  consideration, to account for limitation of Version 2.0 that showed
#  rate of crystallization "slowing" as more sites became occupied.
#
#  Version 3.0 also incorporates a distribution of surface energies across the
#  surface, which controls sites of nucleation and the balance between
#  nucleation and growth rates throughout course of the simulation.
#
#  Version 4.0 incorporates branching by allowing for 90-degree rotation
#  of a new unit as it is added, which favors the attaching of new adjacent
#  units in this orientation.
# ========================================================================
import numpy as np
```

```
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize

## ======================== Initialize Parameters ======================== ##
MC_samples = 40000000 # Number of trial MC steps
Accepted = 0 # Tracks number of accepted MC moves
Rejected = 0 # Tracks number of rejected MC moves
Branches = 0 # Tracks number of times a rotated unit is added
Seed = np.random.randint(2147483646) # Generate seed
MC_seed = np.random.seed(Seed) # Seed random number generator
branching_probability = 0.25 # Probability of rotating new trial unit

cmap = 'afmhot'
norm = Normalize(-0.5, 1.8)

## ========================== Generate Grids ========================== ##
# ===================================================================================
#  Variable 'grid' stores if a given "lattice" location
#  contains a crystallite unit (1 for yes, 0 for no)
# ===================================================================================
dimensionality = 500
total_lattice_sites = int((dimensionality - 2)**2)
grid = np.zeros((dimensionality, dimensionality), dtype='int')
indices = np.indices((dimensionality, dimensionality))

# Orientation default is 1, value of 0 corresponds to 90 degree rotation
orientation = np.ones((dimensionality, dimensionality), dtype='int')

## ============================== Energetics ============================== ##
# ===================================================================================
#  Assign energy benefits and penalties associated with forming "bonds"
#  and "interfaces". The base energy (surface energy) for each site is set at
#  zero initially and adjusted to provide desired configuration. "1" corresponds
#  to uniform (trivial) configuration. "2" corresponds to a distribution where
#  each site is randomly selected from the Gaussian distribution centered
#  around zero. "3" corresponds to a single, large 2D Gaussian function in the
#  center of the lattice. "4" corresponds to many randomly distributed small 2D
#  Gaussian functions. "5" corresponds to a single long and narrow diagonal
#  strip of high energy across the middle of the lattice.
# ===================================================================================
bond = -5 # Energy benefit from adding an adjacent unit
vert_interface = 1.25 # Energy penalty from forming vertical interface
horz_interface = 5.25 # Energy penalty from forming horizontal interface

# Surface Energy Distribution
```

```
configuration = 1 # Determines which energy configuration to implement

if configuration == 1: # Uniform Distribution
surface_energy_distribution = np.zeros((dimensionality, dimensionality), \
dtype='int')

elif configuration == 2: # Random Distribution
surface_energy_mean = 0.0 # Mean of surface energy distribution
surface_energy_stdev = 1.0 # Standard deviation of surface energy distribution
surface_energy_distribution = np.random.normal( \
surface_energy_mean, surface_energy_stdev, \
                                  (dimensionality, dimensionality))

elif configuration == 3: # Large 2D Gaussian
    spread = dimensionality * 4 # Spread of effect of defect
    amp = 5 # Max surface energy at peak of defect
    center = dimensionality / 2 # Define the center coordinate of the lattice
x_diff = np.power(center - indices[1], 2) # X differences squared
y_diff = np.power(center - indices[0], 2) # Y differences squared
surface_energy_distribution = amp*np.exp(-((x_diff+y_diff)/spread))

elif configuration == 4: # Random Small 2D Gaussians
    defects = 40 # Number of defect sites in surface
    spread = dimensionality/10 # Spread of effect of defect
    amp = 5 # Max surface energy at peak of defect
surface_energy_distribution = np.zeros((dimensionality, dimensionality))
    for counter in range(0, defects):
        [x_center, y_center] = np.random.randint(1, dimensionality - 1, size=2)
x_diff = np.power(x_center - indices[1], 2)
y_diff = np.power(y_center - indices[0], 2)
surface_energy_distribution += amp*np.exp(-((x_diff+y_diff)/spread))

else: # Diagonal line
    width = 11 # Odd number giving the width of the line
    amp = 5 # Max surface energy at peak of defect
    base = amp*np.ones((dimensionality, dimensionality))
    upper = np.triu(base, (width-1)/2)
    lower = np.tril(base, -(width-1)/2)
surface_energy_distribution = np.subtract(np.subtract(base, upper), lower)

## ====================== Crystallization Simulation ====================== ##
for counter in range(MC_samples):
    # End simulation if entire surface crystallizes
    if Accepted == total_lattice_sites:
print('Surface completely crystallized')
```

```
        break

    # Save surface plot every specified number of steps
    if (counter + 1) % 5000000 == 0:
        fig = plt.figure()
        power = np.int(np.floor(np.log10(counter + 1)))
        value = (counter + 1) / pow(10, power)
file_name = 'E{:d}_{:.1f}_steps.png'.format(power, value)
plt.pcolormesh(grid, cmap=cmap, norm=norm)
plt.colorbar()
plt.title(format('%d Trial Monte Carlo Steps' % (counter + 1)))
fig.savefig(file_name)
plt.close(fig)

    # Pick an uncrystallized spot at random
    [trial_x_index, trial_y_index] = np.random.randint(
                                     1, dimensionality - 1, size=2)
    while grid[trial_y_index, trial_x_index] == 1:
        [trial_x_index, trial_y_index] = np.random.randint(
                                         1, dimensionality - 1, size=2)

    # Determine neighboring sites and corresponding orientation
up_neighbor = grid[trial_y_index - 1, trial_x_index]
up_neighbor_orient = orientation[trial_y_index - 1, trial_x_index]

down_neighbor = grid[trial_y_index + 1, trial_x_index]
down_neighbor_orient = orientation[trial_y_index + 1, trial_x_index]

left_neighbor = grid[trial_y_index, trial_x_index - 1]
left_neighbor_orient = orientation[trial_y_index, trial_x_index - 1]

right_neighbor = grid[trial_y_index, trial_x_index + 1]
right_neighbor_orient = orientation[trial_y_index, trial_x_index + 1]

all_neighbors = np.array([up_neighbor, down_neighbor,
left_neighbor, right_neighbor])
    [filled_neighbors] = np.nonzero(all_neighbors)

## ======================== Evalutate Orientation ======================== ##
# ======================================================================
#  Determine the average orientation to use for determining the definition
#  of rotate versus non-rotated unit
# ======================================================================
    # If site has no neighbors, use default value of left/right as preferred
    if len(filled_neighbors) == 0:
```

```
avg_orient = orientation[trial_y_index, trial_x_index]
    # If site has neighbors, use the average of their orientations for the
    # preferred direction. If they are equal, use left/right as preferrred
    else:
all_orient = np.array([up_neighbor_orient, down_neighbor_orient,
left_neighbor_orient, right_neighbor_orient])
avg_orient = np.round(np.average(all_orient[filled_neighbors]))

    # Determine branching or continuation
    # If the random number is less than the probability, 90 degree rotation
    # occurs relative to preferred orientation
orient_rand = np.random.random()
prob_check = int(orient_rand<branching_probability)
trial_orient = int((avg_orient and not(prob_check)) or
                   (not(avg_orient) and prob_check)) # XOR logic gate

    # Determine energy benefit/penalty to adding crystal unit in this location
delta_E = -surface_energy_distribution[trial_y_index, trial_x_index]

    # Evaluate energetics for standard orientation
    if trial_orient == 1:
        # If down site is unoccupied, penalize for new vertical interface
        if down_neighbor == 0:
delta_E += vert_interface
        # If down site is occupied and orientation matches down site,
        # benefit for removing vertical interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == down_neighbor_orient) * \
            (bond - vert_interface) + \
int(not(trial_orient == down_neighbor_orient)) * vert_interface

        # If up site is unoccupied, penalize for new vertical interface
        if up_neighbor == 0:
delta_E += vert_interface
        # If up site is occupied and orientation matches up site,
        # benefit for removing vertical interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == up_neighbor_orient) * \
            (bond - vert_interface) + \
int(not(trial_orient == up_neighbor_orient)) * vert_interface

        # If right site is unoccupied, penalize for new horizontal interface
        if right_neighbor == 0:
```

```
delta_E += horz_interface
        # If right site is occupied and orientation matches right site,
        # benefit for removing horizontal interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == right_neighbor_orient) * \
            (bond - horz_interface) + \
int(not(trial_orient == right_neighbor_orient)) * horz_interface

        # If left site is unoccupied, penalize for new horizontal interface
        if left_neighbor == 0:
delta_E += horz_interface
        # If left site is occupied and orientation matches left site,
        # benefit for removing horizontal interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == left_neighbor_orient) * \
            (bond - horz_interface) + \
int(not(trial_orient == left_neighbor_orient)) * horz_interface

    # Evaluate energetics for rotated orientation
    else:
        # If down site is unoccupied, penalize for new horizontal interface
        if down_neighbor == 0:
delta_E += horz_interface
        # If down site is occupied and orientation matches down site,
        # benefit for removing horizontal interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == down_neighbor_orient) * \
            (bond - horz_interface) + \
int(not(trial_orient == down_neighbor_orient)) * horz_interface

        # If up site is unoccupied, penalize for new horizontal interface
        if up_neighbor == 0:
delta_E += horz_interface
        # If up site is occupied and orientation matches up site,
        # benefit for removing horizontal interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == up_neighbor_orient) * \
            (bond - horz_interface) + \
int(not(trial_orient == up_neighbor_orient)) * horz_interface

        # If right site is unoccupied, penalize for new vertical interface
```

```
        if right_neighbor == 0:
delta_E += vert_interface
        # If right site is occupied and orientation matches right site,
        # benefit for removing vertical interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == right_neighbor_orient) * \
            (bond - vert_interface) + \
int(not(trial_orient == right_neighbor_orient)) * vert_interface

        # If left site is unoccupied, penalize for new vertical interface
        if left_neighbor == 0:
delta_E += vert_interface
        # If left site is occupied and orientation matches right site,
        # benefit for removing vertical interface and forming a new bond.
        # If orientation does not match, treats site as unoccupied
        else:
delta_E += int(trial_orient == left_neighbor_orient) * \
            (bond - vert_interface) + \
int(not(trial_orient == left_neighbor_orient)) * vert_interface

    # Calculate probability of adding growth unit then compare against random
    # number between 0.0 and 1.0, accepting move if probability exceeds it
exp_E = np.exp(-delta_E)
    if exp_E>= 1.0:
grid[trial_y_index, trial_x_index] = 1
orientation[trial_y_index, trial_x_index] = trial_orient
        Accepted += 1
        Branches += prob_check
    else:
        test = np.random.random()
        if exp_E> test:
grid[trial_y_index, trial_x_index] = 1
orientation[trial_y_index, trial_x_index] = trial_orient
            Accepted += 1
            Branches += prob_check
        else:
            Rejected += 1

# Surface Energy Distribution Plot
plt.figure()
lower = 0 # Lower bound of data scale
upper = 5 # Upper bound of data scale
plt.pcolormesh(surface_energy_distribution)
plt.colorbar()
```

```
plt.clim(lower, upper)
plt.title('Surface Energy Distribution')
```