DYALOG

Elsinore 2023

# Grain Growth and Array Programming

jgl@dyalog.com
jesus.galanlopez@ugent.be

GHENT UNIVERSITY

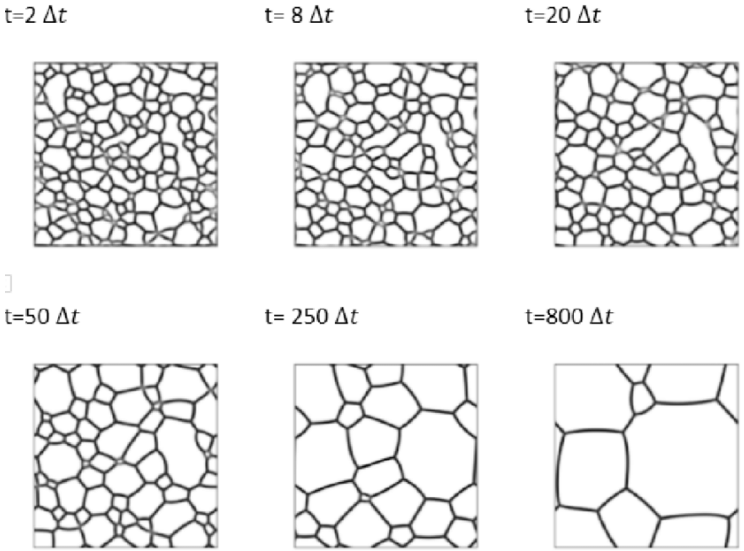
*Based on a true story*

# Problem statement

- Materials science students
- Choose modelling problem
- Scientific literature
- Reproduce model

# Grain Growth

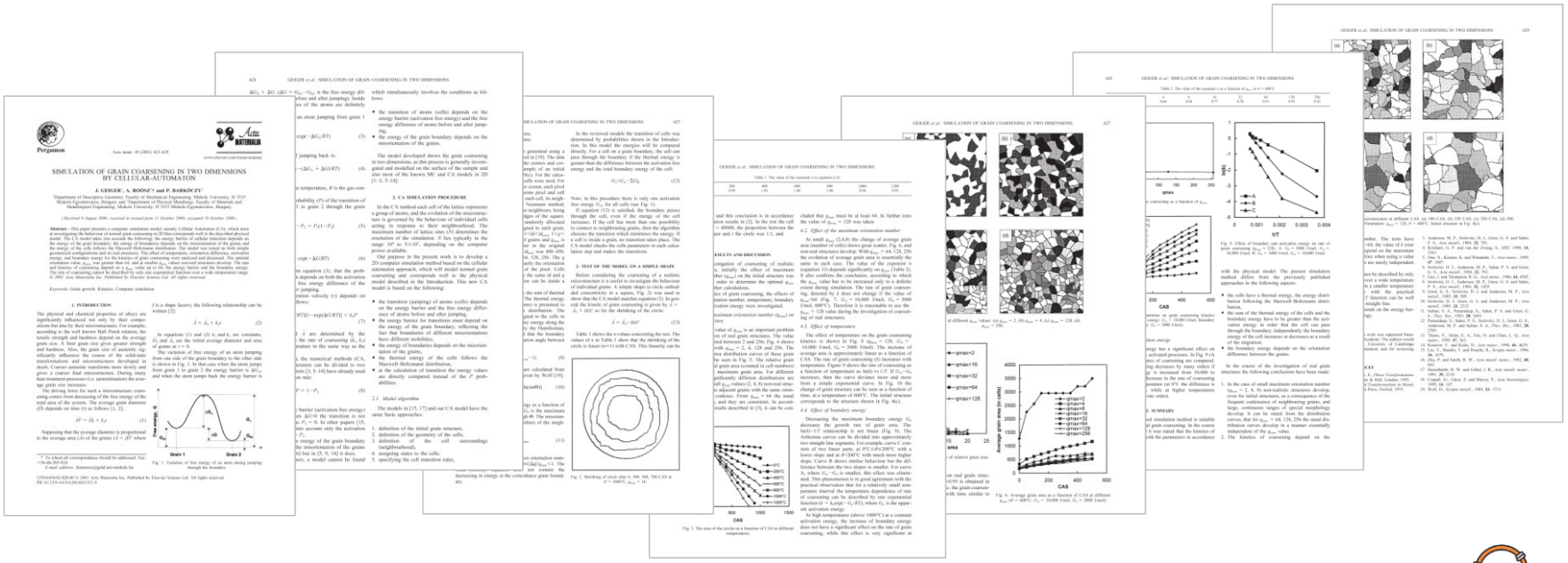
- Increment of size of grains in a polycrystal subjected to high temperatures




Dimokrati, A., & Benyoucef, M. (2016, September)
Phase Field Simulation of Normal Grain Growth
CONAT 2016

# Grain Growth (Geiger, 2001)

https://doi.org/10.1016/S1359-6454(00)00352-9



# Grain Growth (Geiger, 2001)

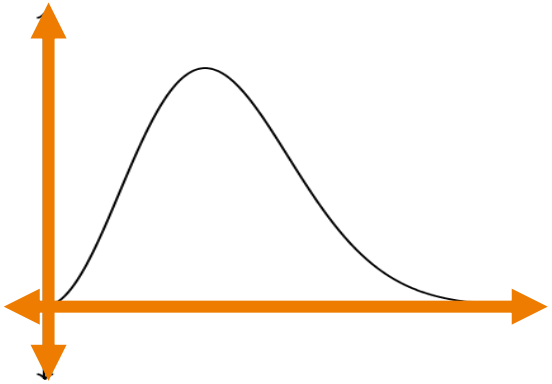
https://doi.org/10.1016/S1359-6454(00)00352-9

- Thermal energy
- Boundary energy
- Activation energy

# Grain Growth (Geiger, 2001)

https://doi.org/10.1016/S1359-6454(00)00352-9

- Thermal energy
- Boundary energy
- Activation energy



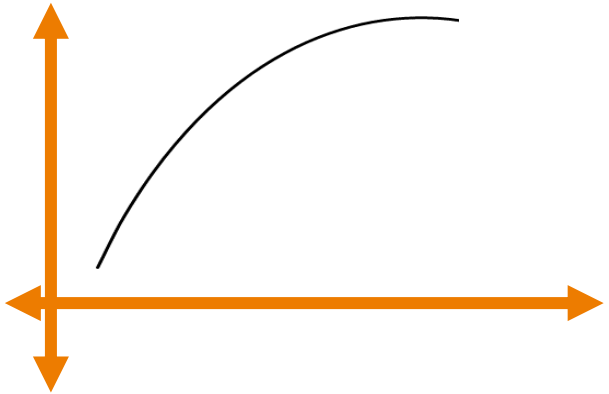
Maxwell-Boltzmann distribution

$$G_T(T) = -R\ T\ \log(x)$$

$x \in U(0,1)$

# Grain Growth (Geiger, 2001)

https://doi.org/10.1016/S1359-6454(00)00352-9



- Thermal energy
- Boundary energy
- Activation energy

Read-Shockley equation $$G_{B_{ij}}(\Delta\theta_{ij}) = G_0 \sin(\Delta\theta_{ij})(1 - \log(\sin(\Delta\theta_{ij})))$$

Misorientation $$\Delta\theta_{ij} = \frac{\pi}{2}\frac{(q_i - q_j)}{q_{max}}$$

# Grain Growth (Geiger, 2001)

https://doi.org/10.1016/S1359-6454(00)00352-9

- Thermal energy
- Boundary energy
- Activation energy

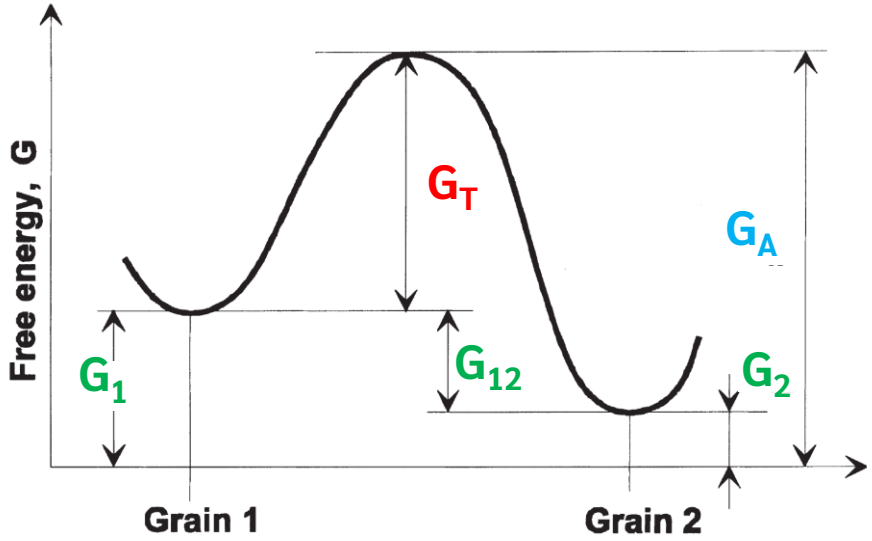
$$G_A = 10000\ J/mol$$

# Grain Growth (Geiger, 2001)

https://doi.org/10.1016/S1359-6454(00)00352-9

- Thermal energy
- Boundary energy
- Activation energy

Transform into neighbour with minimum new boundary energy




Geiger, J., Roósz, A., & Barkoczy, P. (2001). Simulation of grain coarsening in two dimensions by cellular-automaton. Acta materialia, 49(4), 623-629.

# First solution (Python)

- The students know some Python
- Python is easy!
- Python is there

# First solution (Python)

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.measure import regionprops
from skimage.morphology import label
from timeit import timeit

R = 8.314   # ideal gas constant
GA = 10000  # activation energy
G0 = 3000   # maximum boundary energy

# thermal energy according to Maxwell-Boltzmann distribution at temperature T (kelvin)
def thermal(T):
    return -R*T*np.log(1-np.random.random())

# misorientation angle (in radians) between q and q1
def misorientation(q, q1, qmax):
    return (np.pi/2)*np.abs(q-q1)/qmax

# total boundary energy of cell with orientation q and neighbours with orientations q1
def cell_boundary(q, q1, qmax):
    gb = 0
    for qi in q1:
        dg = misorientation(q, qi, qmax)
        sin = np.sin(dg)
        if sin > 0:
            gb += G0*sin*(1-np.log(sin)) if sin > 0 else 0  # Read-Shockley equation
    return gb

def solve(q, T, f=0, n=0):
    i, d = 0, []
    qmax = np.max(q)
    while True:
        di = diameter(q)
        d.append(di)
        if f > 0 and i%f == 0:
            plot(q, qmax, "%d (%.2f)"%(i,di))
        i, (q, changed) = i+1, step(q, qmax, T)
        if (not changed and n == 0) or i == n:
            break
    # final microstructure
    if f > 0:
        plot(q, qmax, "%d (%.2f)"%(i,diameter(q)))
    return q, d
```

```
# return next orientation map q, after grain growth step at temperature T,
# and a boolean indicating if there was any change
def step(q, qmax, T):
    nr, nc = q.shape                # number of rows and columns
    p, changed = np.copy(q), False  # p is the map of product orientations
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[i-1 if i>0 else nr-1,j]
            s = q[i+1 if i<nr-1 else 0,j]
            e = q[i,j+1 if j<nc-1 else 0]
            w = q[i,j-1 if j>0 else nc-1]
            q1 = [n, s, e, w]
            # check if a cell will transform into some neighbour
            if (q1.count(q[i,j]) == len(q1)) or (thermal(T) + cell_boundary(q[i,j], q1, qmax) < GA):
                continue
            # calculate new boundary energy when transforming into each neighbour
            gbn = cell_boundary(n, q1, qmax)
            gbs = cell_boundary(s, q1, qmax)
            gbe = cell_boundary(e, q1, qmax)
            gbw = cell_boundary(w, q1, qmax)
            gb1 = [gbn, gbs, gbe, gbw]
            # pick product for which the boundary energy is the lowest
            p[i,j] = q1[np.argmin(gb1)]
            # check if the orientation changed
            if p[i,j] != q[i,j]:
                changed = True
    return p, changed
```

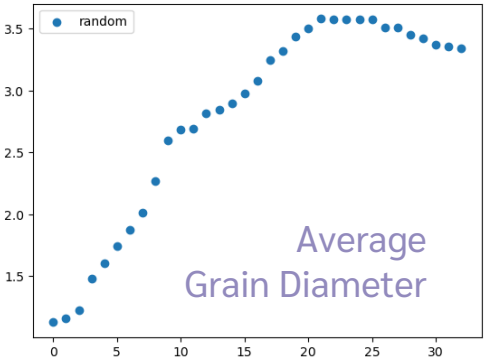
# First solution (Python)

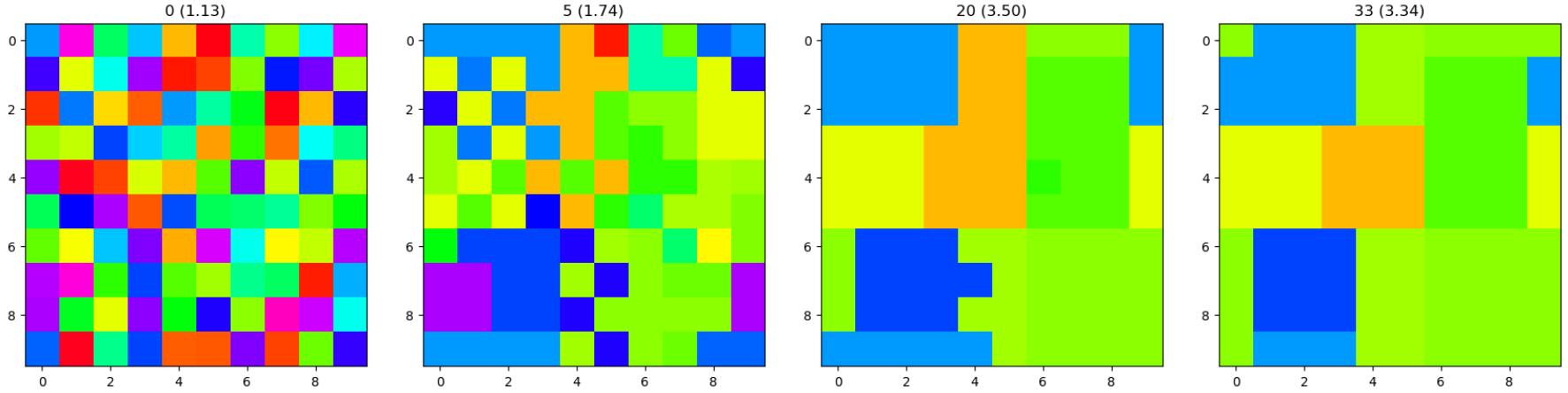
Loops

```
# return next orientation map q, after grain growth step at temperature T,
# and a boolean indicating if there was any change
def step(q, qmax, T):
    nr, nc = q.shape                 # number of rows and columns
    p, changed = np.copy(q), False  # p is the map of product orientations
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[i-1 if i>0 else nr-1,j]
            s = q[i+1 if i<nr-1 else 0,j]
            e = q[i,j+1 if j<nc-1 else 0]
            w = q[i,j-1 if j>0 else nc-1]
            q1 = [n, s, e, w]
            # check if a cell will transform into some neighbour
            if (q1.count(q[i,j]) == len(q1)) or (thermal(T) + cell_boundary(q[i,j], q1, qmax) < GA):
                continue
            # calculate new boundary energy when transforming into each neighbour
            gbn = cell_boundary(n, q1, qmax)
            gbs = cell_boundary(s, q1, qmax)
            gbe = cell_boundary(e, q1, qmax)
            gbw = cell_boundary(w, q1, qmax)
            gb1 = [gbn, gbs, gbe, gbw]
            # pick product for which the boundary energy is the lowest
            p[i,j] = q1[np.argmin(gb1)]
            # check if the orientation changed
            if p[i,j] != q[i,j]:
                changed = True
    return p, changed
```

# First solution (Python)

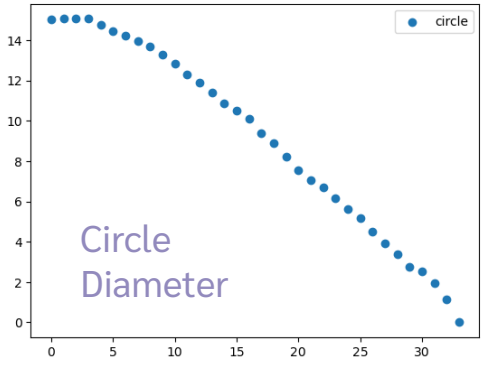
## Random microstructure

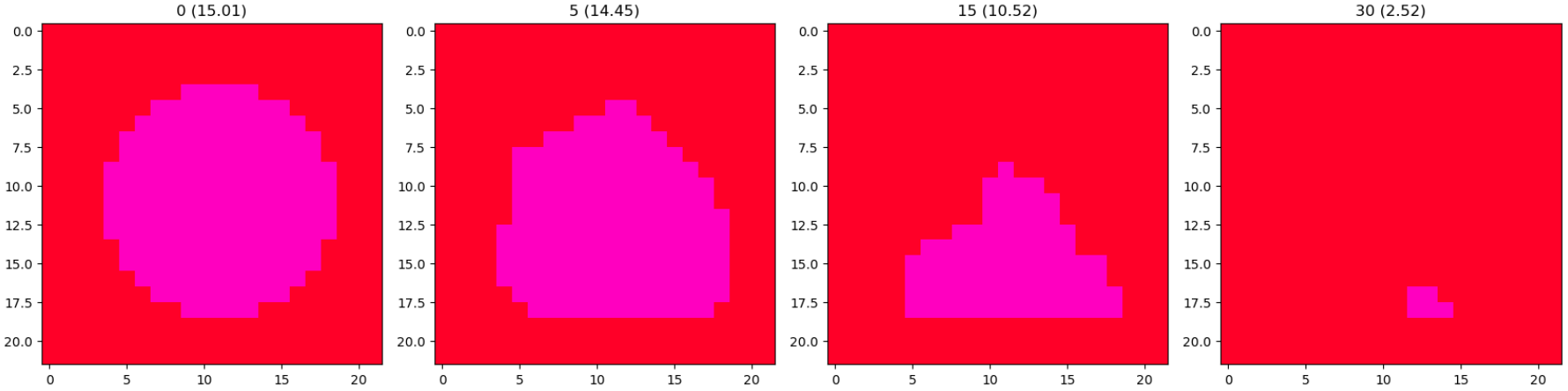


# First solution (Python)

## Single circular grain




# First solution (Python)

```
# return next orientation map q, after grain growth step at temperature T,
# and a boolean indicating if there was any change
def step(q, qmax, T):
    nr, nc = q.shape                 # n                olumns
    p, changed = np.copy(q), False  # p                t orientations
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east
            n = q[i-1 if i>0 else nr-1,j]
            s = q[i+1 if i<nr-1 else 0,j]
            e = q[i,j+1 if j<nc-1 else 0]
            w = q[i,j-1 if j>0 else nc-1]
            q1 = [n, s, e, w]
            # check if a cell will transform int          eighbour
            if (q1.count(q[i,j]) == len(q1)) o          al(T) + cell_boundary(q[i,j], q1, qmax) < GA):
                continue
            # calculate new boundary energy v          ansforming into each neighbour
            gbn = cell_boundary(n, q1, qmax)
            gbs = cell_boundary(s, q1, qmax)
            gbe = cell_boundary(e, q1, qmax)
            gbw = cell_boundary(w, q1, qmax)
            gb1 = [gbn, gbs, gbe, gbw]
            # pick product for which the bou          energy is the lowest
            p[i,j] = q1[np.argmin(gb1)]
            # check if the orientation chan
            if p[i,j] != q[i,j]:
                changed = True
    return p, changed
```

# Partial Results

- Thermal energy
- Current boundary energy
- Boundary energy when transforming into each neighbour
- New orientations (transformed cells)

# First solution (Python)

```
# return next orientation map q, after grain growth step at temperature T,
# and a boolean indicating if there was any change
def step(q, qmax, T):
    nr, nc = q.shape                 # number of rows and columns
    p, changed = np.copy(q), False  # p is the map of product orientations
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[i-1 if i>0 else nr-1,j]
            s = q[i+1 if i<nr-1 else 0,j]
            e = q[i,j+1 if j<nc-1 else 0]
            w = q[i,j-1 if j>0 else nc-1]
            q1 = [n, s, e, w]
            # check if a cell will transform into some neighbour
            if (q1.count(q[i,j]) == len(q1)) or (thermal(T) + cell_boundary(q[i,j], q1, qmax) < GA):
                continue
            # calculate new boundary energy when transforming into each neighbour
            gbn = cell_boundary(n, q1, qmax)
            gbs = cell_boundary(s, q1, qmax)
            gbe = cell_boundary(e, q1, qmax)
            gbw = cell_boundary(w, q1, qmax)
            gb1 = [gbn, gbs, gbe, gbw]
            # pick product for which the boundary energy is the lowest
            p[i,j] = q1[np.argmin(gb1)]
            # check if the orientation changed
            if p[i,j] != q[i,j]:
                changed = True
    return p, changed
```

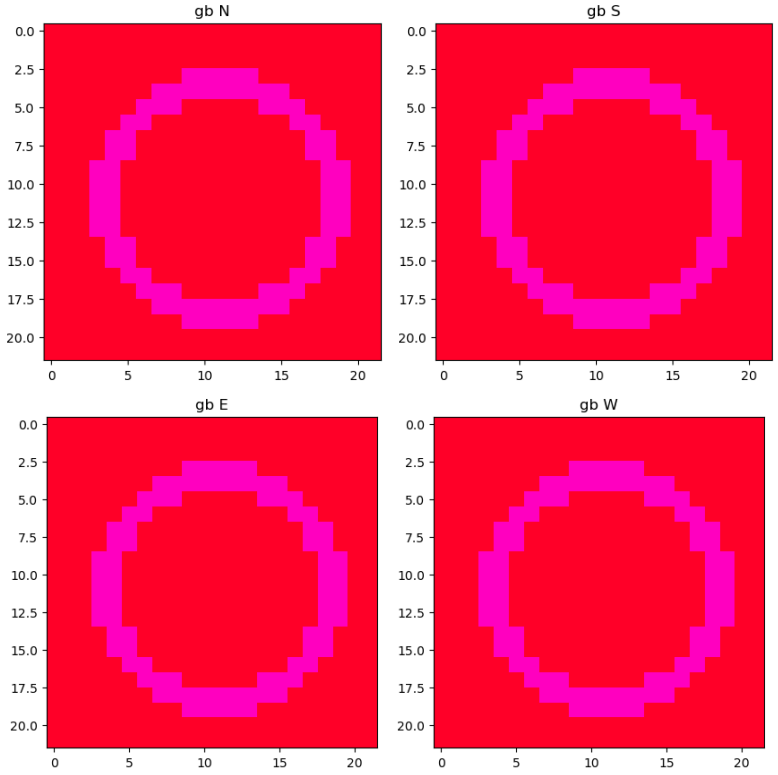
# Partial Results (Python)

```
# new version of step that plots partial results if plt is True
def step_partial(q, qmax, T, plt):
    nr, nc = q.shape                    # number of rows and columns
    p, changed = np.copy(q), False      # p is the map of product orientations
    gt = np.zeros(q.shape)              # thermal energy
    gb = np.zeros(q.shape)              # current grain boundary energy
    gbn = np.zeros(q.shape)             # new grain boundary energy if transforms into north neighbour
    gbs = np.zeros(q.shape)             # new grain boundary energy if transforms into south neighbour
    gbe = np.zeros(q.shape)             # new grain boundary energy if transforms into east neighbour
    gbw = np.zeros(q.shape)             # new grain boundary energy if transforms into west neighbour
    m = np.zeros(q.shape, dtype=int)    # neighbour for which new grain boundary energy is minimum
    t = np.zeros(q.shape, dtype=bool)   # cells that will transform
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[(i-1 if i>0 else nr-1),j]
            s = q[(i+1 if i<nr-1 else 0),j]
            e = q[i,(j+1 if j<nc-1 else 0)]
            w = q[i,(j-1 if j>0 else nc-1)]
            q1 = [n, s, e, w]
            # calculate current boundary energy
            gb[i,j] = cell_boundary(q[i,j], q1, qmax)
            # and when transforming into each neighbour
            gbn[i,j] = cell_boundary(n, q1, qmax)
            gbs[i,j] = cell_boundary(s, q1, qmax)
            gbe[i,j] = cell_boundary(e, q1, qmax)
            gbw[i,j] = cell_boundary(w, q1, qmax)
            gb1 = [gbn[i,j], gbs[i,j], gbe[i,j], gbw[i,j]]
            # find neighbour for which new boundary energy is minimum
            m[i,j] = np.argmin(gb1)
            # thermal energy
            gt[i,j] = thermal(T)
            # check if a cell will transform into some neighbour
            t[i,j] = (q1.count(q[i,j]) != len(q1)) and (gt[i,j] + gb[i,j] >= GA)
            if not t[i,j]:
                continue
            # pick orientation of neighbour with minimum new boundary energy
            p[i,j] = q1[m[i,j]]
            # check if the cell changed of orientation
            if p[i,j] != q[i,j]:
                changed = True
    if plt:
        plot_partial(gt, gb, [gbn, gbs, gbe, gbw], m, t)
    return p, changed
```

Define arrays

Fill arrays inside loop

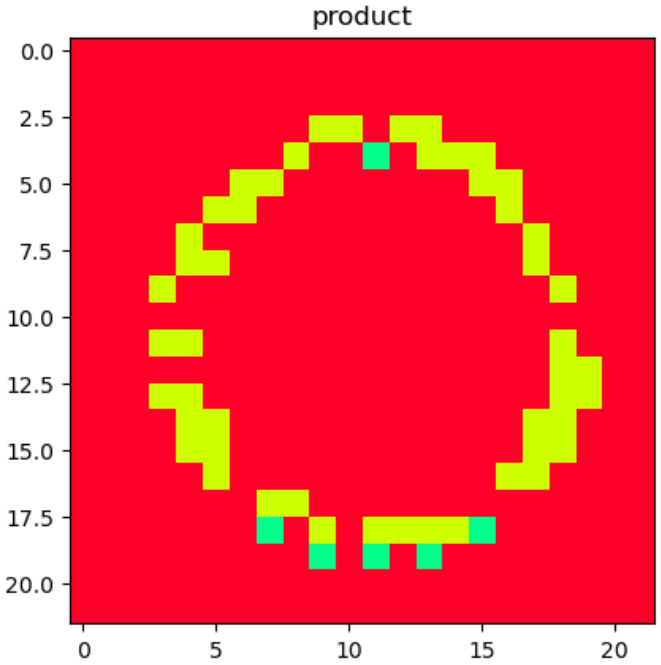
# Partial Results (Python)




New boundary energy

Same change of energy for all neighbours

# Partial Results (Python)




Cells transform only into north and south neighbours

Randomize choice of neighbour with minimum energy

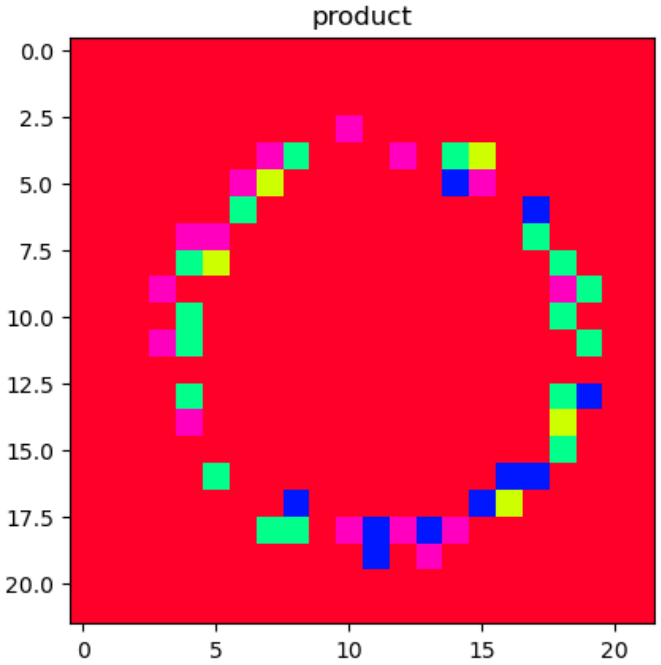
# Random Perturbation (Python)

```
# new version of step_partial which applies a random perturbation to
# new boundary energies before searching the minimum
def step_rnd(q, qmax, T, plt):
    nr, nc = q.shape                    # number of rows and columns
    p, changed = np.copy(q), False      # p is the map of product orientations
    gt = np.zeros(q.shape)              # thermal energy
    gb = np.zeros(q.shape)              # current grain boundary energy
    gbn = np.zeros(q.shape)             # new grain boundary energy if transforms into north neighbour
    gbs = np.zeros(q.shape)             # new grain boundary energy if transforms into south neighbour
    gbe = np.zeros(q.shape)             # new grain boundary energy if transforms into east neighbour
    gbw = np.zeros(q.shape)             # new grain boundary energy if transforms into west neighbour
    m = np.zeros(q.shape, dtype=int)    # neighbour for which new grain boundary energy is minimum
    t = np.zeros(q.shape, dtype=bool)   # cells that will transform
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[(i-1 if i>0 else nr-1),j]
            s = q[(i+1 if i<nr-1 else 0),j]
            e = q[i,(j+1 if j<nc-1 else 0)]
            w = q[i,(j-1 if j>0 else nc-1)]
            q1 = [n, s, e, w]
            # calculate current boundary energy
            gb[i,j] = cell_boundary(q[i,j], q1, qmax)
            # and when transforming into each neighbour
            gbn[i,j] = cell_boundary(n, q1, qmax) * (1 + RP * np.random.random())
            gbs[i,j] = cell_boundary(s, q1, qmax) * (1 + RP * np.random.random())
            gbe[i,j] = cell_boundary(e, q1, qmax) * (1 + RP * np.random.random())
            gbw[i,j] = cell_boundary(w, q1, qmax) * (1 + RP * np.random.random())
            gb1 = [gbn[i,j], gbs[i,j], gbe[i,j], gbw[i,j]]
            # find neighbour for which new boundary energy is minimum
            m[i,j] = np.argmin(gb1)
            # thermal energy
            gt[i,j] = thermal(T)
            # check if a cell will transform into some neighbour
            t[i,j] = (q1.count(q[i,j]) != len(q1)) and (gt[i,j] + gb[i,j] >= GA)
            if not t[i,j]:
                continue
            # pick orientation of neighbour with minimum new boundary energy
            p[i,j] = q1[m[i,j]]
            # check if the cell changed of orientation
            if p[i,j] != q[i,j]:
                changed = True
    if plt:
        plot_partial(gt, gb, [gbn, gbs, gbe, gbw], m, t)
    return p, changed
```

Apply small random perturbation to new boundary energies

```
RP = 0.01  # random perturbation
```

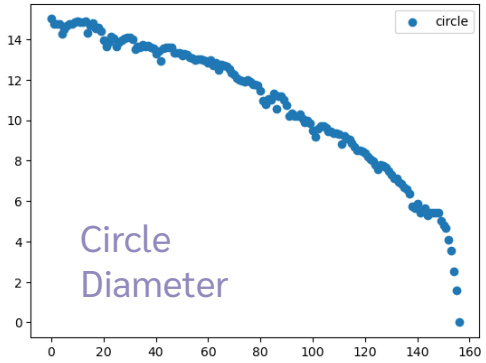
# Random Perturbation (Python)

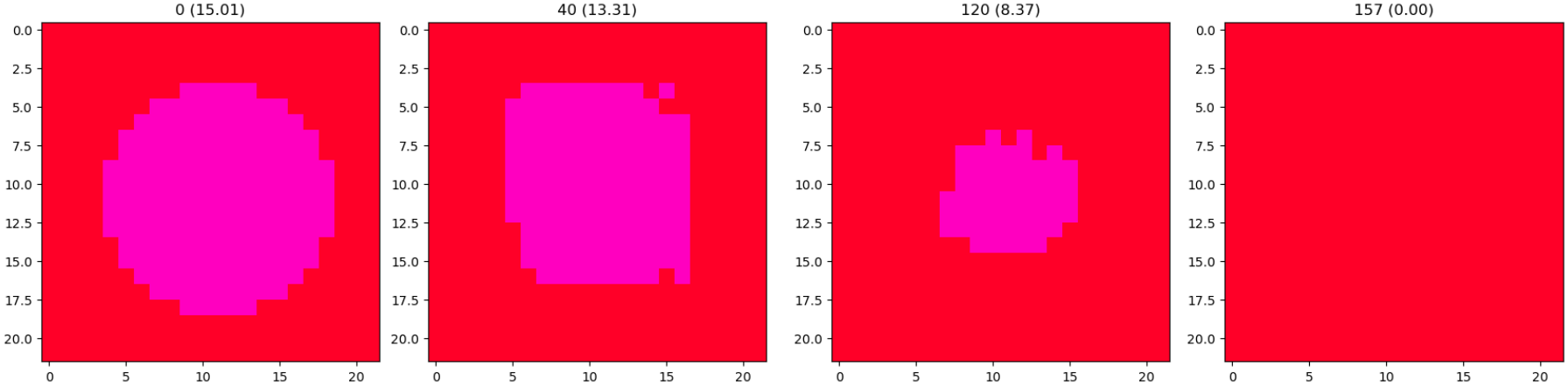



Neighbour into which to transform is randomly chosen

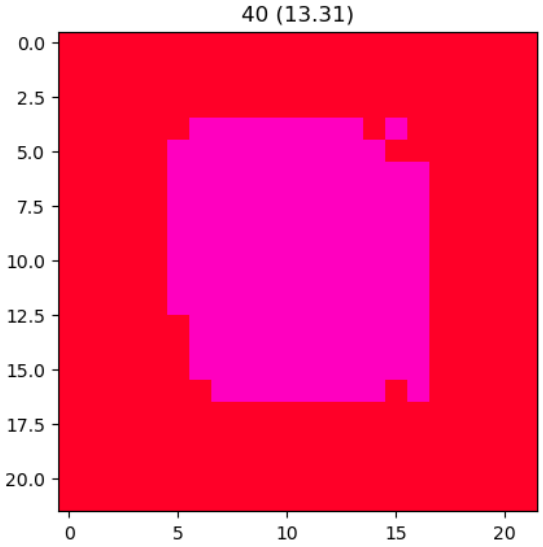
# Random Perturbation

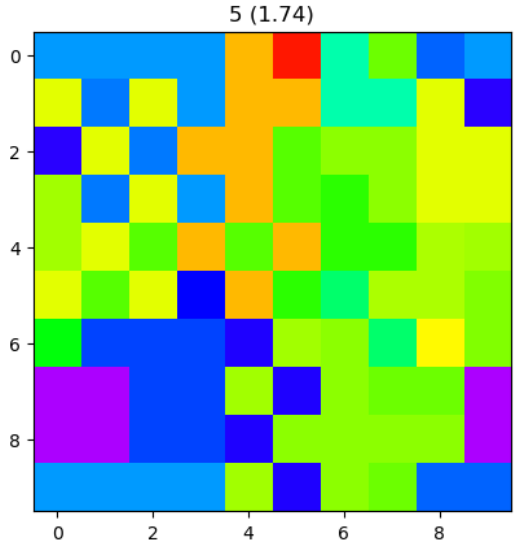
## Single circular grain



# Other problems




Unrealistic
“checkboard pattern”

Consider second
order neighbours

# Random Perturbation (Python)

```
# new version of step_partial which applies a random perturbation to
# new boundary energies before searching the minimum
def step_rnd(q, qmax, T, plt):
    nr, nc = q.shape                    # number of rows and columns
    p, changed = np.copy(q), False      # p is the map of product orientations
    gt = np.zeros(q.shape)              # thermal energy
    gb = np.zeros(q.shape)              # current grain boundary energy
    gbn = np.zeros(q.shape)             # new grain boundary energy if transforms into north neighbour
    gbs = np.zeros(q.shape)             # new grain boundary energy if transforms into south neighbour
    gbe = np.zeros(q.shape)             # new grain boundary energy if transforms into east neighbour
    gbw = np.zeros(q.shape)             # new grain boundary energy if transforms into west neighbour
    m = np.zeros(q.shape, dtype=int)    # neighbour for which new grain boundary energy is minimum
    t = np.zeros(q.shape, dtype=bool)   # cells that will transform
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[(i-1 if i>0 else nr-1),j]
            s = q[(i+1 if i<nr-1 else 0),j]
            e = q[i,(j+1 if j<nc-1 else 0)]
            w = q[i,(j-1 if j>0 else nc-1)]
            q1 = [n, s, e, w]
            # calculate current boundary energy
            gb[i,j] = cell_boundary(q[i,j], q1, qmax)
            # and when transforming into each neighbour
            gbn[i,j] = cell_boundary(n, q1, qmax) * (1 + RP * np.random.random())
            gbs[i,j] = cell_boundary(s, q1, qmax) * (1 + RP * np.random.random())
            gbe[i,j] = cell_boundary(e, q1, qmax) * (1 + RP * np.random.random())
            gbw[i,j] = cell_boundary(w, q1, qmax) * (1 + RP * np.random.random())
            gb1 = [gbn[i,j], gbs[i,j], gbe[i,j], gbw[i,j]]
            # find neighbour for which new boundary energy is minimum
            m[i,j] = np.argmin(gb1)
            # thermal energy
            gt[i,j] = thermal(T)
            # check if a cell will transform into some neighbour
            t[i,j] = (q1.count(q[i,j]) != len(q1)) and (gt[i,j] + gb[i,j] >= GA)
            if not t[i,j]:
                continue
            # pick orientation of neighbour with minimum new boundary energy
            p[i,j] = q1[m[i,j]]
            # check if the cell changed of orientation
            if p[i,j] != q[i,j]:
                changed = True
    if plt:
        plot_partial(gt, gb, [gbn, gbs, gbe, gbw], m, t)
    return p, changed
```

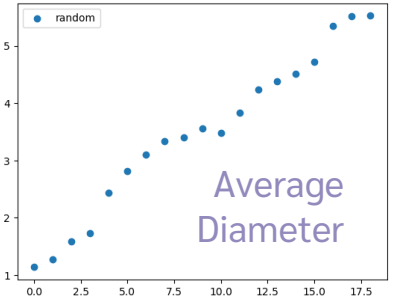
```
RP = 0.01  # random perturbation
```

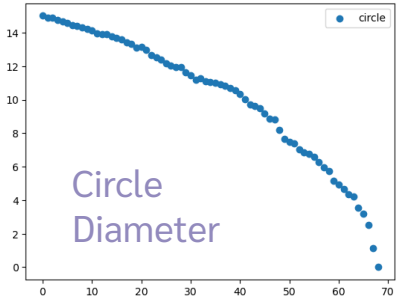
# 2nd order neighbours (Python)

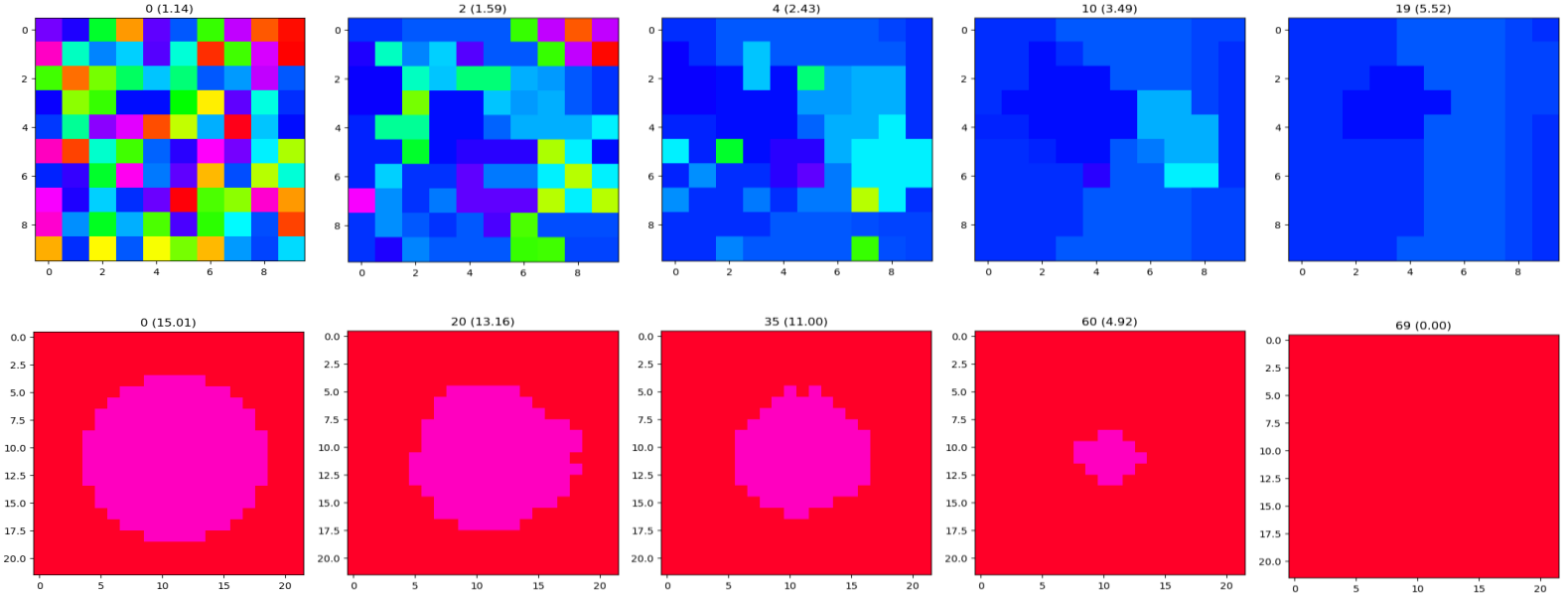
```python
# new version of step_rnd which takes into account the influence of secon-order neighbours
def step_rnd2(q, qmax, T, plt):
    nr, nc = q.shape                    # number of rows and columns
    p, changed = np.copy(q), False      # p is the map of product orientations
    gt = np.zeros(q.shape)              # thermal energy
    gb = np.zeros(q.shape)              # current grain boundary energy
    gbn = np.zeros(q.shape)             # new grain boundary energy if transforms into north neighbour
    gbs = np.zeros(q.shape)             # new grain boundary energy if transforms into south neighbour
    gbe = np.zeros(q.shape)             # new grain boundary energy if transforms into east neighbour
    gbw = np.zeros(q.shape)             # new grain boundary energy if transforms into west neighbour
    m = np.zeros(q.shape, dtype=int)    # neighbour for which new grain boundary energy is minimum
    t = np.zeros(q.shape, dtype=bool)   # cells that will transform
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[(i-1 if i>0 else nr-1),j]
            s = q[(i+1 if i<nr-1 else 0),j]
            e = q[i,(j+1 if j<nc-1 else 0)]
            w = q[i,(j-1 if j>0 else nc-1)]
            q1 = [n, s, e, w]
            # calculate current boundary energy
            gb[i,j] = cell_boundary(q[i,j], q1, qmax)
            # and when transforming into each neighbour
            gbn[i,j] = cell_boundary(n, q1, qmax)
            gbs[i,j] = cell_boundary(s, q1, qmax)
            gbe[i,j] = cell_boundary(e, q1, qmax)
            gbw[i,j] = cell_boundary(w, q1, qmax)
            if SF > 0:
                nw = q[(i-1 if i>0 else nr-1),(j-1 if j>0 else nc-1)]
                se = q[(i+1 if i<nr-1 else 0),(j+1 if j<nc-1 else 0)]
                ne = q[(i-1 if i>0 else nr-1),(j+1 if j<nc-1 else 0)]
                sw = q[(i+1 if i<nr-1 else 0),(j-1 if j>0 else nc-1)]
                q2 = [nw, se, ne, sw]
                gbn[i,j] += SF * cell_boundary(n, q2, qmax)
                gbs[i,j] += SF * cell_boundary(s, q2, qmax)
                gbe[i,j] += SF * cell_boundary(e, q2, qmax)
                gbw[i,j] += SF * cell_boundary(w, q2, qmax)
            gbn[i,j] *= 1 + RP * np.random.random()
            gbs[i,j] *= 1 + RP * np.random.random()
            gbe[i,j] *= 1 + RP * np.random.random()
            gbw[i,j] *= 1 + RP * np.random.random()
            gb1 = [gbn[i,j], gbs[i,j], gbe[i,j], gbw[i,j]]
            # find neighbour for which new boundary energy is minimum
            m[i,j] = np.argmin(gb1)
            # thermal energy
            gt[i,j] = thermal(T)
            # check if a cell will transform into some neighbour
            t[i,j] = (q1.count(q[i,j]) != len(q1)) and (gt[i,j] + gb[i,j] >= GA)
            if not t[i,j]:
                continue
            # pick orientation of neighbour with minimum new boundary energy
            p[i,j] = q1[m[i,j]]
            # check if the cell changed of orientation
            if p[i,j] != q[i,j]:
                changed = True
    if plt:
        plot_partial(gt, gb, [gbn, gbs, gbe, gbw], 'NSEW', m, t)
    return p, changed
```

```python
RP = 0.01  # random perturbation
SF = 0.25  # second-order-neighbours factor
```

# 2nd order


random
Average
Diameter


circle
Circle
Diameter


0 (1.14)
2 (1.59)
4 (2.43)
10 (3.49)
19 (5.52)
0 (15.01)
20 (13.16)
35 (11.00)
60 (4.92)
69 (0.00)

# 2nd order neighbours (Python)

```python
# new version of step_rnd which takes into account the influence of secon-order neighbours
def step_rnd2(q, qmax, T, plt):
    nr, nc = q.shape                # number of rows and columns
    p, changed = np.copy(q), False  # p is the map of product orientations
    gt = np.zeros(q.shape)          # thermal energy
    gb = np.zeros(q.shape)          # current grain boundary energy
    gbn = np.zeros(q.shape)         # new grain boundary energy if transforms into north neighbour
    gbs = np.zeros(q.shape)         # new grain boundary energy if transforms into south neighbour
    gbe = np.zeros(q.shape)         # new grain boundary energy if transforms into east neighbour
    gbw = np.zeros(q.shape)         # new grain boundary energy if transforms into west neighbour
    m = np.zeros(q.shape, dtype=int)   # neighbour for which new grain boundary energy is minimum
    t = np.zeros(q.shape, dtype=bool)  # cells that will transform
    for i in range(nr):
        for j in range(nc):
            # 1st-order neighbours: north, south, east, west
            n = q[(i-1 if i>0 else nr-1),j]
            s = q[(i+1 if i<nr-1 else 0),j]
            e = q[i,(j+1 if j<nc-1 else 0)]
            w = q[i,(j-1 if j>0 else nc-1)]
            q1 = [n, s, e, w]
            # calculate current boundary energy
            gb[i,j] = cell_boundary(q[i,j], q1, qmax)
            # and when transforming into each neighbour
            gbn[i,j] = cell_boundary(n, q1, qmax)
            gbs[i,j] = cell_boundary(s, q1, qmax)
            gbe[i,j] = cell_boundary(e, q1, qmax)
            gbw[i,j] = cell_boundary(w, q1, qmax)
            if SF > 0:
                nw = q[(i-1 if i>0 else nr-1),(j-1 if j>0 else nc-1)]
                se = q[(i+1 if i<nr-1 else 0),(j+1 if j<nc-1 else 0)]
                ne = q[(i-1 if i>0 else nr-1),(j+1 if j<nc-1 else 0)]
                sw = q[(i+1 if i<nr-1 else 0),(j-1 if j>0 else nc-1)]
                q2 = [nw, se, ne, sw]
                gbn[i,j] += SF * cell_boundary(n, q2, qmax)
                gbs[i,j] += SF * cell_boundary(s, q2, qmax)
                gbe[i,j] += SF * cell_boundary(e, q2, qmax)
                gbw[i,j] += SF * cell_boundary(w, q2, qmax)
            gbn[i,j] *= 1 + RP * np.random.random()
            gbs[i,j] *= 1 + RP * np.random.random()
            gbe[i,j] *= 1 + RP * np.random.random()
            gbw[i,j] *= 1 + RP * np.random.random()
            gb1 = [gbn[i,j], gbs[i,j], gbe[i,j], gbw[i,j]]
            # find neighbour for which new boundary energy is minimum
            m[i,j] = np.argmin(gb1)
            # thermal energy
            gt[i,j] = thermal(T)
            # check if a cell will transform into some neighbour
            t[i,j] = (q1.count(q[i,j]) != len(q1)) and (gt[i,j] + gb[i,j] >= GA)
            if not t[i,j]:
                continue
            # pick orientation of neighbour with minimum new boundary energy
            p[i,j] = q1[m[i,j]]
            # check if the cell changed of orientation
            if p[i,j] != q[i,j]:
                changed = True
    if plt:
        plot_partial(gt, gb, [gbn, gbs, gbe, gbw], 'NSEW', m, t)
    return p, changed
```

Too much repetitive code

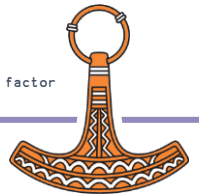
Hard to understand what the model is doing

Easy to make mistakes

Slow

ARRAY PROGRAMMING

```
RP = 0.01  # random perturbation
SF = 0.25  # second-order-neighbours factor
```



# 2nd order neighbours (NumPy)

```
# new version of step_rnd2 using numpy functions
def step_numpy2(q, qmax, T, plt):
    q1 = [np.roll(q, i, axis=a) for i, a in zip([-1,1,1,-1], [0,0,1,1])]      # 1st-order neighbours
    gt, gb = -R*T*np.log(1-np.random.random(q.shape)), boundary(q, q1, qmax)  # thermal and boundary energies
    gb1 = np.array([boundary(qi, q1, qmax) for qi in q1])                     # new boundary energies
    if SF > 0:
        q2 = [np.roll(qi, i, axis=a) for qi, i, a in zip(q1, [-1,1,-1,1], [1,1,0,0])]  # 2nd-order neighbours
        gb1 += SF * np.array([boundary(qi, q2, qmax) for qi in q1])                    # their boundary energies
    gb1 *= 1 + RP * np.random.random(gb1.shape)                               # apply random perturbation
    t, m = (gt + gb >= GA), np.argmin(gb1, axis=0)                            # cells to transform and minimum
    p = np.where(t, np.choose(m, q1), q)                                      # product orientations
    if plt:
        plot_partial(gt, gb, gb1, 'NSEW', m, t)
    return p, not np.array_equal(q, p)
```

Less code, less mistakes, more performance

Less likely to be written by students

```
RP = 0.01  # random perturbation
SF = 0.25  # second-order-neighbours factor
```

# 2nd order neighbours (NumPy)

```
# new version of step_rnd2 using numpy functions
def step_numpy2(q, qmax, T, plt):
    q1 = [np.roll(q, i, axis=a) for i, a in zip([-1,1,1,-1], [0,0,1,1])]      # 1st-order neighbours
    gt, gb = -R*T*np.log(1-np.random.random(q.shape)), boundary(q, q1, qmax)  # thermal and boundary energies
    gb1 = np.array([boundary(qi, q1, qmax) for qi in q1])                     # new boundary energies
    if SF > 0:
        q2 = [np.roll(qi, i, axis=a) for qi, i, a in zip(q1, [-1,1,-1,1], [1,1,0,0])]  # 2nd-order neighbours
        gb1 += SF * np.array([boundary(qi, q2, qmax) for qi in q1])                    # their boundary energies
    gb1 *= 1 + RP * np.random.random(gb1.shape)                               # apply random perturbation
    t, m = (gt + gb >= GA), np.argmin(gb1, axis=0)                            # cells to transform and minimum
    p = np.where(t, np.choose(m, q1), q)                                      # product orientations
    if plt:
        plot_partial(gt, gb, gb1, 'NSEW', m, t)
    return p, not np.array_equal(q, p)

# return boundary energies for orientations q with neighbours q1
def boundary(q, q1, qmax):
    dg = np.array([misorientation(q, qi, qmax) for qi in q1])
    sin = np.sin(dg)
    gbi = G0 * sin * (1 - np.log(sin, out=np.ones_like(dg), where=dg>0))
    return np.sum(gbi, axis=0)
```

```
RP = 0.01  # random perturbation
SF = 0.25  # second-order-neighbours factor
```

# NumPy

- Array objects
  - The N-dimensional array (**ndarray**)
  - Scalars
  - Data type objects (**dtype**)
  - Indexing routines
  - Iterating over arrays
  - Standard array subclasses
  - Masked arrays
  - The array interface protocol
  - Datetimes and Timedeltas
- Array API Standard Compatibility
  - Table of Differences between **numpy.array_api** and **numpy**
- Constants
  - **Inf**
  - **Infinity**
  - **NAN**
  - **NINF**
  - **NZERO**
  - **NaN**
  - **PINF**
  - **PZERO**
  - **e**
  - **euler_gamma**
  - **inf**
  - **infty**
  - **nan**
  - **newaxis**
  - **pi**
- Universal functions (**ufunc**)
  - **ufunc**
  - Available ufuncs
- Routines
  - Array creation routines
  - Array manipulation routines
  - Binary operations
  - String operations
  - C-Types foreign function interface (**numpy.ctypeslib**)
  - Datetime support functions
  - Data type routines
  - Mathematical functions with automatic domain
  - Floating point error handling
  - Discrete Fourier Transform (**numpy.fft**)
  - Functional programming
  - NumPy-specific help functions
  - Input and output
  - Linear algebra (**numpy.linalg**)
  - Logic functions
  - Masked array operations
  - Mathematical functions
  - Matrix library (**numpy.matlib**)
  - Miscellaneous routines
  - Padding Arrays
  - Polynomials
  - Random sampling (**numpy.random**)
  - Set routines
  - Sorting, searching, and counting
  - Statistics
  - Test Support (**numpy.testing**)
  - Support for testing overrides (**numpy.testing.overrides**)
  - Window functions
- Typing (**numpy.typing**)
  - Mypy plugin
  - Differences from the runtime NumPy API
  - API
- Global state
  - Performance-related options
  - Debugging-related options
  - Testing planned future behavior
- Packaging (**numpy.distutils**)
  - Modules in **numpy.distutils**
  - Configuration class
  - Building Installable C libraries
  - Conversion of **.src** files
- NumPy distutils - users guide
  - SciPy structure
  - Requirements for SciPy packages
  - The **setup.py** file
  - The **__init__.py** file
  - Extra features in NumPy Distutils
- Status of **numpy.distutils** and migration advice
  - Migration advice
  - Interaction of **numpy.distutils** with **setuptools**
- NumPy C-API
  - Python Types and C-Structures
  - System configuration
  - Data Type API
  - Array API
  - Array Iterator API
  - UFunc API
  - Generalized Universal Function API
  - NumPy core libraries
  - C API Deprecations
  - Memory management in NumPy
- CPU/SIMD Optimizations
  - CPU build options
  - How does the CPU dispatcher work?
- NumPy security
  - Advice for using NumPy on untrusted data
- NumPy and SWIG
  - numpy.i: a SWIG Interface File for NumPy
  - Testing the numpy.i Typemaps

# NumPy

## Available ufuncs

### Math operations

add(x1, x2, /[, out, where, casting, order, ...])
subtract(x1, x2, /[, out, where, casting, ...])
multiply(x1, x2, /[, out, where, casting, ...])
matmul(x1, x2, /[, out, casting, order, ...])
divide(x1, x2, /[, out, where, casting, ...])
logaddexp(x1, x2, /[, out, where, casting, ...])
logaddexp2(x1, x2, /[, out, where, casting, ...])
true_divide(x1, x2, /[, out, where, ...])
floor_divide(x1, x2, /[, out, where, ...])
negative(x, /[, out, where, casting, order, ...])
positive(x, /[, out, where, casting, order, ...])
power(x1, x2, /[, out, where, casting, ...])
float_power(x1, x2, /[, out, where, ...])
remainder(x1, x2, /[, out, where, casting, ...])
mod(x1, x2, /[, out, where, casting, order, ...])
fmod(x1, x2, /[, out, where, casting, ...])
divmod(x1, x2[, out1, out2], / [[, out, ...])
absolute(x, /[, out, where, casting, order, ...])
fabs(x, /[, out, where, casting, order, ...])
rint(x, /[, out, where, casting, order, ...])
sign(x, /[, out, where, casting, order, ...])
heaviside(x1, x2, /[, out, where, casting, ...])
conj(x, /[, out, where, casting, order, ...])
conjugate(x, /[, out, where, casting, ...])
exp(x, /[, out, where, casting, order, ...])
exp2(x, /[, out, where, casting, order, ...])
log(x, /[, out, where, casting, order, ...])
log2(x, /[, out, where, casting, order, ...])
log10(x, /[, out, where, casting, order, ...])
expm1(x, /[, out, where, casting, order, ...])
log1p(x, /[, out, where, casting, order, ...])
sqrt(x, /[, out, where, casting, order, ...])
square(x, /[, out, where, casting, order, ...])
cbrt(x, /[, out, where, casting, order, ...])
reciprocal(x, /[, out, where, casting, ...])
gcd(x1, x2, /[, out, where, casting, order, ...])
lcm(x1, x2, /[, out, where, casting, order, ...])

### Trigonometric functions

sin(x, /[, out, where, casting, order, ...])
cos(x, /[, out, where, casting, order, ...])
tan(x, /[, out, where, casting, order, ...])
arcsin(x, /[, out, where, casting, order, ...])
arccos(x, /[, out, where, casting, order, ...])
arctan(x, /[, out, where, casting, order, ...])
arctan2(x1, x2, /[, out, where, casting, ...])
hypot(x1, x2, /[, out, where, casting, ...])
sinh(x, /[, out, where, casting, order, ...])
cosh(x, /[, out, where, casting, order, ...])
tanh(x, /[, out, where, casting, order, ...])
arcsinh(x, /[, out, where, casting, order, ...])
arccosh(x, /[, out, where, casting, order, ...])
arctanh(x, /[, out, where, casting, order, ...])
degrees(x, /[, out, where, casting, order, ...])
radians(x, /[, out, where, casting, order, ...])
deg2rad(x, /[, out, where, casting, order, ...])
rad2deg(x, /[, out, where, casting, order, ...])

### Bit-twiddling functions

bitwise_and(x1, x2, /[, out, where, ...])
bitwise_or(x1, x2, /[, out, where, casting, ...])
bitwise_xor(x1, x2, /[, out, where, ...])
invert(x, /[, out, where, casting, order, ...])
left_shift(x1, x2, /[, out, where, casting, ...])
right_shift(x1, x2, /[, out, where, ...])

### Comparison functions

greater(x1, x2, /[, out, where, casting, ...])
greater_equal(x1, x2, /[, out, where, ...])
less(x1, x2, /[, out, where, casting, ...])
less_equal(x1, x2, /[, out, where, casting, ...])
not_equal(x1, x2, /[, out, where, casting, ...])
equal(x1, x2, /[, out, where, casting, ...])
logical_and(x1, x2, /[, out, where, ...])
logical_or(x1, x2, /[, out, where, casting, ...])
logical_xor(x1, x2, /[, out, where, ...])
logical_not(x, /[, out, where, casting, ...])
maximum(x1, x2, /[, out, where, casting, ...])
minimum(x1, x2, /[, out, where, casting, ...])
fmax(x1, x2, /[, out, where, casting, ...])
fmin(x1, x2, /[, out, where, casting, ...])

### Floating functions

isfinite(x, /[, out, where, casting, order, ...])
isinf(x, /[, out, where, casting, order, ...])
isnan(x, /[, out, where, casting, order, ...])
isnat(x, /[, out, where, casting, order, ...])
fabs(x, /[, out, where, casting, order, ...])
signbit(x, /[, out, where, casting, order, ...])
copysign(x1, x2, /[, out, where, casting, ...])
nextafter(x1, x2, /[, out, where, casting, ...])
spacing(x, /[, out, where, casting, order, ...])
modf(x[, out1, out2], / [[, out, where, ...])
ldexp(x1, x2, /[, out, where, casting, ...])
frexp(x[, out1, out2], / [[, out, where, ...])
fmod(x1, x2, /[, out, where, casting, ...])
floor(x, /[, out, where, casting, order, ...])
ceil(x, /[, out, where, casting, order, ...])
trunc(x, /[, out, where, casting, order, ...])

- Array c
- Array A
- Consta
- Univer

API
?y
vork?
or

Array objects


The N-dimensional array (ndarray)
- Constructing arrays
- Indexing arrays
- Internal memory layout of an ndarray
- Array attributes
- Array methods
- Arithmetic, matrix multiplication, and comparison operations
- Special methods

Scalars
- Built-in scalar types
- Attributes
- Indexing
- Methods
- Defining new types

Data type objects (dtype)
- Specifying and constructing data types
- dtype

Indexing routines
- Generating index arrays
- Indexing-like operations
- Inserting data into arrays
- Iterating over arrays

Iterating over arrays
- Single array iteration
- Broadcasting array iteration
- Putting the Inner Loop in Cython

Standard array subclasses
- Special attributes and methods
- Matrix objects
- Memory-mapped file arrays
- Character arrays (numpy.char)
- Record arrays (numpy.rec)
- Masked arrays (numpy.ma)
- Standard container class
- Array Iterators

Masked arrays
- The numpy.ma module
- Using numpy.ma
- Examples
- Constants of the numpy.ma module
- The MaskedArray class
- MaskedArray methods
- Masked array operations

The array interface protocol
- Python side
- C-struct access
- Type description examples
- Differences with Array interface (Version 2)

Datetimes and Timedeltas
- Datetime64 Conventions and Assumptions
- Basic Datetimes
- Datetime and Timedelta Arithmetic
- Datetime Units
- Business Day Functionality
- Datetime64 shortcomings


nctions

ut, where, casting, ...])
2, /[, out, where, ...])
where, casting, ...])
, out, where, casting, ...])
out, where, casting, ...])
where, casting, ...])
[, out, where, ...])
out, where, casting, ...])
, out, where, ...])
, where, casting, ...])
out, where, casting, ...])
out, where, casting, ...])
where, casting, ...])
where, casting, ...])

API

y

work?

or

remainder(x1, x2, /[, out, where, casting, ...])
mod(x1, x2, /[, out, where, casting, order, ...])
fmod(x1, x2, /[, out, where, casting, ...])
divmod(x1, x2[, out1, out2], / [[, out, ...])
absolute(x, /[, out, where, casting, order, ...])
fabs(x, /[, out, where, casting, order, ...])
rint(x, /[, out, where, casting, order, ...])
sign(x, /[, out, where, casting, order, ...])
heaviside(x1, x2, /[, out, where, casting, ...])
conj(x, /[, out, where, casting, order, ...])
conjugate(x, /[, out, where, casting, ...])
exp(x, /[, out, where, casting, order, ...])
exp2(x, /[, out, where, casting, order, ...])
log(x, /[, out, where, casting, order, ...])
log2(x, /[, out, where, casting, order, ...])
log10(x, /[, out, where, casting, order, ...])
expm1(x, /[, out, where, casting, order, ...])
log1p(x, /[, out, where, casting, order, ...])

sinh(x, /[, out, where, casting, order, ...])
cosh(x, /[, out, where, casting, order, ...])
tanh(x, /[, out, where, casting, order, ...])
arcsinh(x, /[, out, where, casting, order, ...])
arccosh(x, /[, out, where, casting, order, ...])
arctanh(x, /[, out, where, casting, order, ...])
degrees(x, /[, out, where, casting, order, ...])
radians(x, /[, out, where, casting, order, ...])
deg2rad(x, /[, out, where, casting, order, ...])
rad2deg(x, /[, out, where, casting, order, ...])

## Bit-twiddling functions

bitwise_and(x1, x2, /[, out, where, ...])
bitwise_or(x1, x2, /[, out, where, casting, ...])
bitwise_xor(x1, x2, /[, out, where, ...])
invert(x, /[, out, where, casting, order, ...])
left_shift(x1, x2, /[, out, where, casting, ...])
right_shift(x1, x2, /[, out, where, ...])

## Floating functions

isfinite(x, /[, out, where, casting, order, ...])
isinf(x, /[, out, where, casting, order, ...])
isnan(x, /[, out, where, casting, order, ...])
isnat(x, /[, out, where, casting, order, ...])
fabs(x, /[, out, where, casting, order, ...])
signbit(x, /[, out, where, casting, order, ...])
copysign(x1, x2, /[, out, where, casting, ...])
nextafter(x1, x2, /[, out, where, casting, ...])
spacing(x, /[, out, where, casting, order, ...])
modf(x[, out1, out2], / [[, out, where, ...])
ldexp(x1, x2, /[, out, where, casting, ...])
frexp(x[, out1, out2], / [[, out, where, ...])
fmod(x1, x2, /[, out, where, casting, ...])
floor(x, /[, out, where, casting, order, ...])
ceil(x, /[, out, where, casting, order, ...])
trunc(x, /[, out, where, casting, order, ...])

Array objects

The N-dimensional array (ndarray)
- Constructing arrays
- Indexing
- Internal
- Array attr
- Array me
- Arithmeti
- comparis
- Special m

Scalars
- Built-in s
- Attribute
- Indexing
- Methods
- Defining

Data type objects (d
- Specifyin
- dtype

Indexing routines
- Generating index arrays
- Indexing-like operations
- Inserting data into arrays
- Iterating over arrays

Examples
Constants of the numpy.ma module
The MaskedArray class
MaskedArray methods
Masked array operations

Routines

Array creation routines
- From shape or value
- From existing data
- Creating record arrays (numpy.rec)
- Creating character arrays (numpy.char)
- Numerical ranges
- Building matrices
- The Matrix class

Array manipulation routines
- Basic operations
- Changing array shape
- Transpose-like operations
- Changing number of dimensions
- Changing kind of array
- Joining arrays
- Splitting arrays
- Tiling arrays
- Adding and removing elements
- Rearranging elements

Binary operations
- Elementwise bit operations
- Bit packing
- Output formatting

String operations
- String operations
- Comparison
- String information
- Convenience class

C-Types foreign function interface (numpy.ctypeslib)
- as_array
- as_ctypes
- as_ctypes_type
- load_library
- ndpointer
- c_intp

Datetime support functions
- numpy.datetime_as_string
- numpy.datetime_data
- Business day functions

Data type routines
- numpy.can_cast
- numpy.promote_types
- numpy.min_scalar_type
- numpy.result_type
- numpy.common_type
- numpy.obj2sctype
- Creating data types
- Data type information
- Data type testing
- Miscellaneous

Mathematical functions with automatic domain
- Functions

Floating point error handling
- Setting and getting error handling
- Internal functions

Discrete Fourier Transform (numpy.fft)
- Standard FFTs
- Real FFTs
- Hermitian FFTs
- Helper routines
- Background information
- Implementation details
- Type Promotion
- Normalization
- Real and Hermitian transforms
- Higher dimensions
- References
- Examples

Functional programming
- numpy.apply_along_axis
- numpy.apply_over_axes
- numpy.vectorize
- numpy.frompyfunc
- numpy.piecewise

NumPy-specific help functions
- Finding help
- Reading help

Input and output
- NumPy binary files (NPY, NPZ)
- Text files
- Raw binary files
- String formatting
- Memory mapping files
- Text formatting options
- Base-n representations
- Data sources
- Binary format description

Linear algebra (numpy.linalg)
- The @ operator
- Matrix and vector products
- Decompositions
- Matrix eigenvalues
- Norms and other numbers
- Solving equations and inverting matrices
- Exceptions
- Linear algebra on several matrices at once

Logic functions
- Truth value testing
- Array contents
- Array type testing
- Logical operations
- Comparison

Masked array operations
- Constants
- Creation
- Inspecting the array
- Manipulating a MaskedArray
- Operations on masks
- Conversion operations
- Masked arrays arithmetic

Mathematical functions
- Trigonometric functions
- Hyperbolic functions
- Rounding
- Sums, products, differences
- Exponents and logarithms
- Other special functions
- Floating point routines
- Rational routines
- Arithmetic operations
- Handling complex numbers
- Extrema Finding
- Miscellaneous

Matrix library (numpy.matlib)
- numpy.matlib.empty
- numpy.matlib.zeros
- numpy.matlib.ones
- numpy.matlib.eye
- numpy.matlib.identity
- numpy.matlib.repmat
- numpy.matlib.rand
- numpy.matlib.randn

Miscellaneous routines
- Performance tuning
- Memory ranges
- Array mixins
- NumPy version comparison
- Utility
- Matlab-like Functions

Padding Arrays
- numpy.pad

Polynomials
- Transitioning
- from numpy.poly1d to numpy.polynomial
- Documentation for the polynomial Package
- Documentation for Legacy Polynomials

Random sampling (numpy.random)
- Quick Start
- Design
- Concepts
- Features

Set routines
- numpy.lib.arraysetops
- Making proper sets
- Boolean operations

Sorting, searching, and counting
- Sorting
- Searching
- Counting

Statistics
- Order statistics
- Averages and variances
- Correlating
- Histograms

Test Support (numpy.testing)
- Asserts
- Asserts (not recommended)
- Decorators
- Test Running
- Guidelines

Support for testing overrides (numpy.testing.override
- Utility Functions

Window functions
- Various windows

Learning Python might be easy

Mastering NumPy is hard


| | |
|---|---|
| Mathematics | + - × ÷ ⌈ ⌊ * ! \| ⊛ ○ ⌸ ⊥ ⊤ ? |
| Logic and Comparison | ~ ∧ ∨ ⍲ ⍱ < > ≤ ≥ = ≠ ≡ ≢ |
| Structural | ⍴ , ⍪ ⌽ ⊖ ⍉ ↑ ↓ ⊂ ⊆ ≡ ≢ ∈ |
| Selection and Sets | [] ⌷ ⊃ / ⌿ \ ⍀ ~ ≠ ∪ ∩ ⊣ ⊢ |
| Search and Ordering | ⍳ ⍸ ∈ ⍷ ⍋ ⍒ |
| Operators | ¨ ⍨ ⍣ . ∘ / \ ⌿ ⍀ ⌸ ⍤ ⍥ ⍠ @ ⌺ [] |
| Dfns and dops | ∇ ⍺ ⍵ ∇∇ ⍺⍺ ⍵⍵ {} |
| Miscellaneous | ¯ ← → ⍬ ⍎ ⍕ ⋄ ⍝ ⍞ ⎕ ⍼ I & # () |

# APL Grain Growth Model

```
gg←{
    area←{0=⊃ω:+/0≠,ω ⋄ (⍳2)×.((≢,ω){αα÷1⌈+/,ω≠1⌽[α]ω})⊂ω}
    boundary←(⌈/,ω){+⌿G0×s×1-⊛@(0∘<)s←1oo|↑(ω-⊂α)÷2×αα}
    step←{gt←-R×α×⊛?≠⍨ω ⋄ gb←ω boundary⊢q1←1 ¯1(⊖⍤,⌽⍤)⊂ω ⋄ t←(⊃(⊂ω)∨.≠q1)∧GA≤gt+gb
        gb1←(1+RP×?≠⍨q1)×{ω+SF×q1 boundary⍤⊂1 ¯1(⊖⍤∘(2↓⊢),⌽⍤∘(2↓⌽))q1}⍣(SF>0)boundary⍤∘⊂⍨q1
        m←⊃∘⍋⍤1⍉gb1 ⋄ (t/⍤,⍉m⌷⍤0 1⍉q1)@(t⍨)ω}
    R←8.314 ⋄ (GA G0 SF RP)←αα ⋄ ⊃{α(2×(ω÷o1)*÷2)}/α{q((⊃⌽ω),area⊢q←α step⊃ω)}⍣ωω⊢ω(area ω)
}
```

Whole solution: step calculation, solve, calculate average diameter

Only APL primitives, no external libraries or system commands

Partial results available in the interpreter

# APL Gr

```
gg←{
    area←{0=⊃ω:
    boundary←(⌈
    step←{gt←-R
        gb1←(1+
        m←⊃∘⍋⍤1
    R←8.314 ⋄ (
}
```

Whole solut

Only APL pr

Partial resul

```
gg←{  ⍝ ⍺⍺: parameters (GA, G0, SF, RP), ⍵⍵: repetitions, ⍺: temperature, ⍵: initial orientations

    R←8.314                                 ⍝ ideal gas constant
    (GA G0 SF RP)←⍺⍺                        ⍝ simulation parameters
    qmax←⌈/,⍵                               ⍝ maximum orientation

    misorientations←{○|↑(⍵-⊂⍺)÷2×qmax}      ⍝ ⍺: orientations, ⍵: neighbour orientations
                                            ⍝   return misorientations in radians

    boundary←{                              ⍝ ⍺: orientations, ⍵: list of neighbours orientations
        s←1○⍺ misorientations ⍵             ⍝   sin of misorientation angle
        +⌿G0×s×1-⍵⍵{0<⍵}s                   ⍝   return sum of all boundary energies (Read-Shockley)
    }

    boundary2←{                             ⍝ ⍵: list of 1st-order neighbours orientations
        SF≤0:0                              ⍝   return zero if SF is not positive
        q2←(1 ¯1⍥2↑⍵),1 ¯1⌽⍥2↑⌽⍵             ⍝   orientations of 2nd-order neighbours
        SF×⍵ boundary⍤⊂q2                   ⍝   return boundary energy of 2nd-order neighbours
    }

    step←{                                  ⍝ ⍺: temperature, ⍵: orientations
        gt←-R×⍺×⍵⍵?≠⍤⍨⍵                     ⍝   thermal energy
        q1←(1 ¯1⍥⊂⍵),1 ¯1⌽⍥⊂⍵               ⍝   orientations of 1st-order neighbours
        gb←⍵ boundary q1                    ⍝   boundary energy
        t←(∨⌿↑(⊂⍵)≠⍤⊃q1)∧GA≤gt+gb           ⍝   cells that will transform
        gb1←q1 boundary⍤⊂q1                 ⍝   boundary energy when transforming into each neighbour
        gb1←gb1+boundary2 q1                ⍝   add contribution of 2nd-order neighbours
        gb1←gb1×1+RP×?≠⍤⍨q1                 ⍝   apply random perturbation
        m←⊃⍒⌿1⍋⍤⍒1⌽↑gb1                     ⍝   index of neighbour for minimum energy
        p←⌽⍵m[]⍤⍒0 1⌽↑q1                    ⍝   product orientation for all cells
        ((,t)/,p)@(t⍤⍨)⍵                    ⍝   return new orientations
    }

    area←{                                  ⍝ ⍵: orientations
        0=⊃⍵:+/0≠,⍵                         ⍝   return non-zero cells for circle problem
        w←(≠,⍵)÷1⌈+/,⍵≠1⌽⍵                  ⍝   average grain width
        h←(≠,⍵)÷1⌈+/,⍵≠1⊖⍵                  ⍝   average grain height
        w×h                                 ⍝   return average grain area
    }

    next←{(q a)←⍵ ⋄ n←⍺ step q ⋄ n(a,area n)}  ⍝ ⍺: temperature, ⍵: orientations and list of areas
                                            ⍝   return new orientations and updated list of areas

                                            ⍝ solve:
    (q a)←⍺(next⍣⍵⍵)⍵(area ⍵)               ⍝ get final orientations and list of areas
    q(2×(a÷○1)*÷2)                          ⍝ return final orientations and list of diameters
}
```

# APL Grain Growth Model

```
gg←{
    area←{0=⊃⍵:+/0≠,⍵ ⋄ (⍳2)×.((≠,⍵){⍺⍺÷1⌈+/,⍵≠1⌽[⍺]⍵})⊂⍵}
    boundary←(⌈/,⍵){+⌿G0×s×1-⊗@(0∘<)s←1oo|↑(⍵-⊂⍺)÷2×⍺⍺}
    step←{gt←-R×⍺×⊗?≠⍤¨⍵ ⋄ gb←⍵ boundary⊢q1←1 ⍢1(⊖⍤¨,⌽⍤¨)⊂⍵ ⋄ t←(⊃(⊂⍵)∨.≠q1)∧GA≤gt+gb
        gb1←(1+RP×?≠⍤¨q1)×{⍵+SF×q1 boundary⍤¨⊂1 ⍢1(⊖⍤¨∘(2↓⊢),⌽⍤¨∘(2↓⌽))q1}⍣(SF>0)boundary⍤¨∘⊂⍤¨q1
        m←⊃∘⍋⍤¨1⌽↑gb1 ⋄ (t/⍤¨,⌽m[]⍤¨0 1⌽↑q1)@(t⍤¨)⍵}
    R←8.314 ⋄ (GA G0 SF RP)←⍺⍺ ⋄ ⊃{⍺(2×(⍵÷⍥1)*÷2)}/⍺{q((⊃⌽⍵),area⊢q←⍺ step⊃⍵)}⍣⍵⍵⊢⍵(area ⍵)
}
```

# Py'N'APL Grain Growth Model

```
from pynapl import APL

apl = APL.APL()  # APL session

gg = apl.op('''{
        area←{0=⊃ω:+/0≠,ω ⋄ (⍳2)×.((≠,ω){αα÷1⌈+/,ω≠1⌽[α]ω})⊂ω}
        boundary←(⌈/,ω){+⌿G0×s×1-⊛@(0∘<)s←1⍥⌽|↑(ω-⊂α)÷2×αα}
        step←{gt←-R×α×⊛?≠⍤⊃ω ⋄ gb←ω boundary⊢q1←1 ¯1(⍣¨,⌽¨)⊂ω ⋄ t←(⊃(⊂ω)∨.≠q1)∧GA≤gt+gb
            gb1←(1+RP×?≠⍤⊃q1)×{ω+SF×q1 boundary¨⊂1 ¯1(⍣¨∘(2↓⊢),⌽¨∘(2↓⌽))q1}⍨(SF>0)boundary¨∘⊂⍤⊃q1
            m←⊃∘⍋⍤⍒1⌽↑gb1 ⋄ (t/⍤⍒,⌽m[]⍤⍒0 1⌽↑q1)@(t⍤⊃)ω}
        R←8.314 ⋄ (GA G0 SF RP)←αα ⋄ ⊃{α(2×(ω÷⍥1)*÷2)}/α{q((⊃⌽ω),area⊢q←α step⊃ω)}⍨ωω⊢ω(area ω)
}''')

def solve_apl(q, T, n=0):
    return gg((GA, G0, SF, RP), apl.fn('≡⍤⊃') if n == 0 else n)(T, q)
```

# Benchmarks

Circle problem (radious 25, random perturbation, 2nd order)

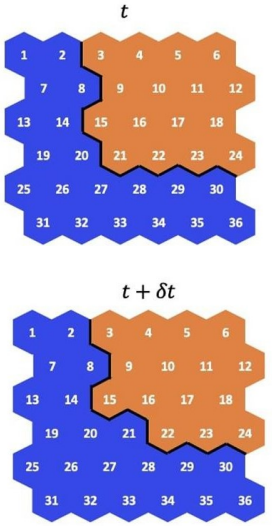
```
Python 11.506s
NumPy   0.214s
APL     0.160s
```

The APL version runs faster in the interpreter, and it could be faster

The NumPy version could be faster too

# Going further

- Crystallographic orientations
- Growth velocity (boundary mobility)
- Influence of 2nd order neighbours in formation of new boundaries




Traka, Konstantina, Karo Sedighiani, Cornelis Bos, Jesus Galan Lopez, Katja Angenendt, Dierk Raabe, and Jilt Sietsma. "Topological aspects responsible for recrystallization evolution in an IF-steel sheet–Investigation with cellular-automaton simulations." Computational Materials Science 198 (2021): 110643.

# Conclusions

- Array programming offers clear advantages in code clarity, size and performance
- The availability of intermediate results is an often-overlooked advantage
- Python is a widely known, easy to learn, nice little language, but NumPy is another story
- APL can be a serious contender

# What APL does great

- Powerful array programming with a few  primitives
- Easy (and fast) to develop, once you learn
- High performance “out of the box”
- Dyalog (IDE, documentation, support)
- Community

# Room for improvement

- Unknown in some circles
- Introduction for domain experts
- Specialized “libraries”
- Community and ecosystem

# To sum up

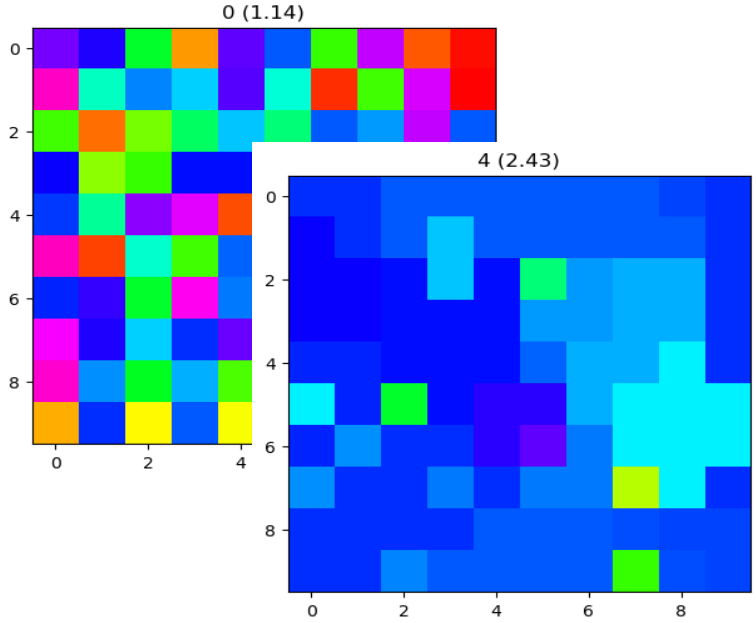
- APL is great
- We need:
  - More people
  - More code

# Thank you

DYALOG

Elsinore 2023

# Grain Growth and Array Programming




jgl@dyalog.com
jesus.galanlopez@ugent.be

GHENT UNIVERSITY