# scratchpad

December 22, 2021

## 0.1 Checklist

- Pseudocode, well-written
  - [+] Synthesize paper
  - [+] Demo flow
  - [+] Test on larger raster
- Get Rainfall Distributions
  - Local areas (circles)
  - Amounts over time
- Runoff [+] Precipitation [+] Uniform [+] Circular (API in progress) [+] Border [+] Time-varying
- Infiltration (much smaller scale), so ruled out.

### 0.1.1 Testing

[+] a funcanim of pouring over time, and if things balance out. [+] Is mass conserved? [+] Water level on cell/1D slice over time [+] Write results to disk (params, time) [+] Quiver plot

### 0.1.2 Strategies

[+] Metrics + Water height/cell + mean flow rate (m^3/s) [ ] % chance of inundation

[ ] Sites. Sites of interest (e.g. population centers, rivers, etc) [ ] Damages caused by flood vs water level + Blocking Flow (high Walls : what gets flooded in turn?) + block based on the water levels we observe

### 0.1.3 Plotting

[~+] Set rcParams [+] 3D viz of a DEM [+] Flowline of a point using D8, perhaps alternatives (then do salmon algo)

[+] test to see if a flood barrier works

### 0.1.4 Theoretical Baselines

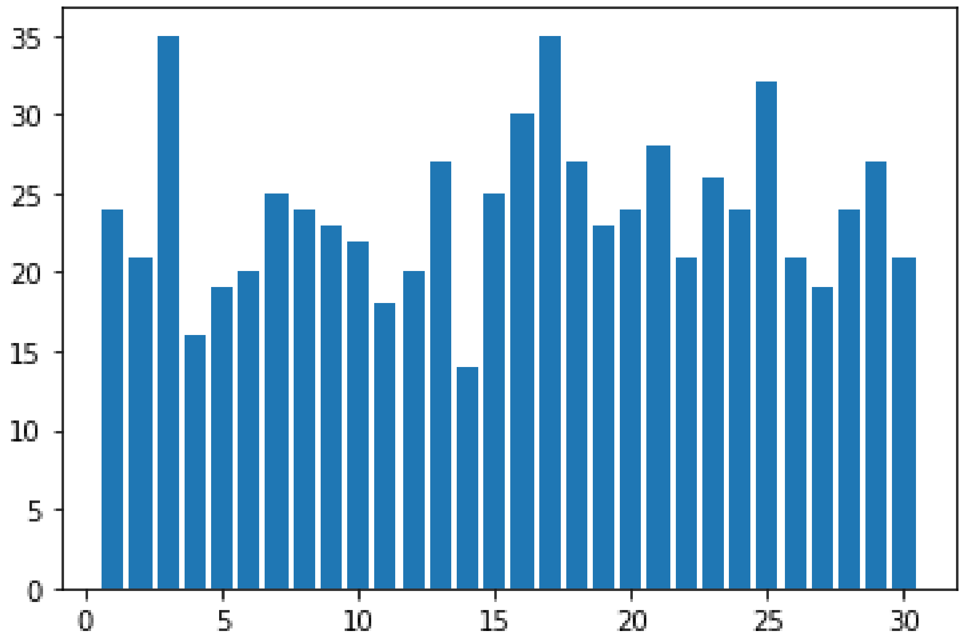
[+] Flow accumulation matrices + "Swimming Upstream" over gradient/slope fields [+] Maximum flow velocity (mgh -> 1/2 m v^2)

### 0.1.5 Upkeep

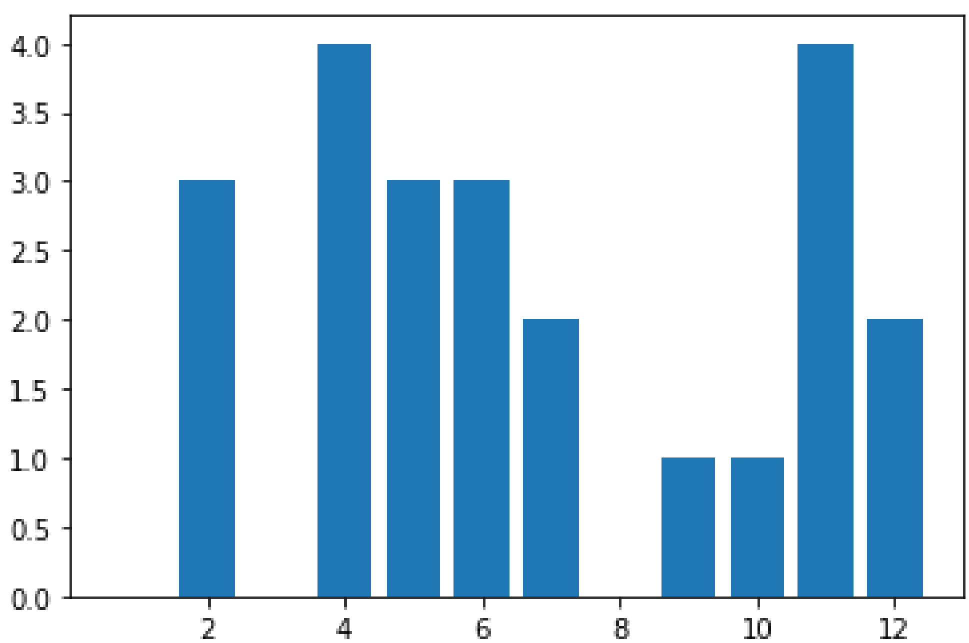
[+] Save figures [+] Save data [+] Refactor code

[ ]:
```
# input: curr_time(s) returns: rain (in m)
def s_to_h(s):
    second_of_day = s % (3600*24)

    return second_of_day/3600

# A Poisson-Cascade Model

# poisson distribution for days in a month
monthly_rainfall = np.random.poisson(lam=700/30, size=30)

# sample a day from monthly_rainfall
day = np.random.choice(monthly_rainfall, size=1)
# poisson distribution for rainfall each hour
daily_rainfall = np.random.poisson(lam=day/12, size=12)

# sample an hour from daily_rainfall
hour = np.random.choice(daily_rainfall, size=1)
# poisson distribution for rainfall each minute
hourly_rainfall = np.random.poisson(lam=hour/60, size=60)

# bar plot of monthly rainfall
plt.bar(np.arange(1,31), monthly_rainfall)
print(np.sum(monthly_rainfall))
plt.show()

# clear plot
plt.bar(np.arange(1,13), daily_rainfall)
print(np.sum(daily_rainfall))
plt.show()

plt.bar(np.arange(1,61), hourly_rainfall)
print(np.sum(hourly_rainfall))
plt.show()

# plot rainfall over all plots
```

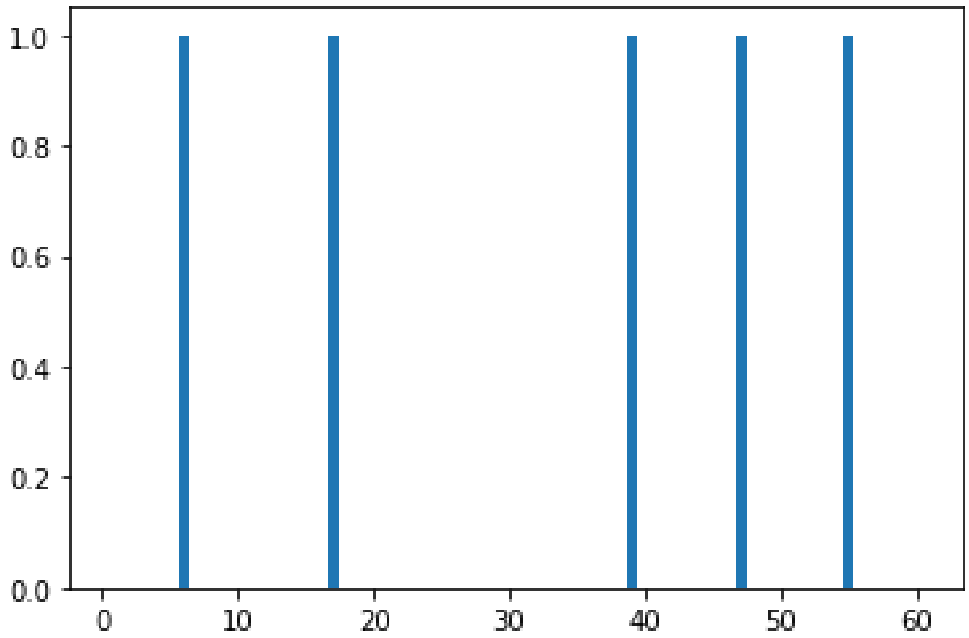
```
715
```


35
30
25
20
15
10
5
0
0
5
10
15
20
25
30

23


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2
4
6
8
10
12

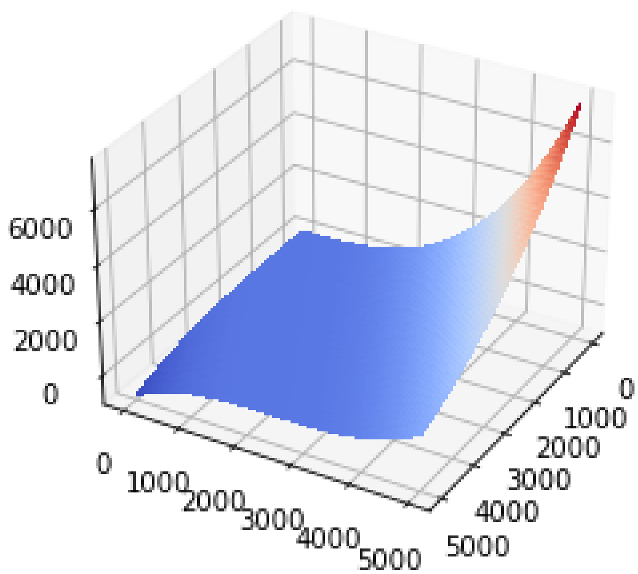
5




[ ]:
```
import scipy.stats as sts
import matplotlib.pyplot as plt
from scipy.ndimage import generic_filter

import numpy as np
import sklearn.datasets as ds
from matplotlib import cm
import matplotlib.animation as animation

import richdem as rd
import tqdm

import time
from numba import jit
```

[ ]:
```
def hunter(x, t):
    C = 0.1
    u = 2
    n = 0.02
    return (7/3 * ( C - n**2 * u**2 * (x - u*t)))**3/7

x = np.linspace(0, 5000, 100)
y = hunter(x, 20)
```

```python
# plot in 3d
fig = plt.figure()

X, Y = np.meshgrid(x, x)

z = hunter(X, Y)

# CONTOUR3D
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, z, cmap=cm.coolwarm, linewidth=0, antialiased=False)

# rotate
ax.view_init(elev=30, azim=30)
```




[ ]:

```python

```

[ ]:

```python
# set rcparams
#set linewidth
plt.rcParams['axes.linewidth'] = 0.8
```

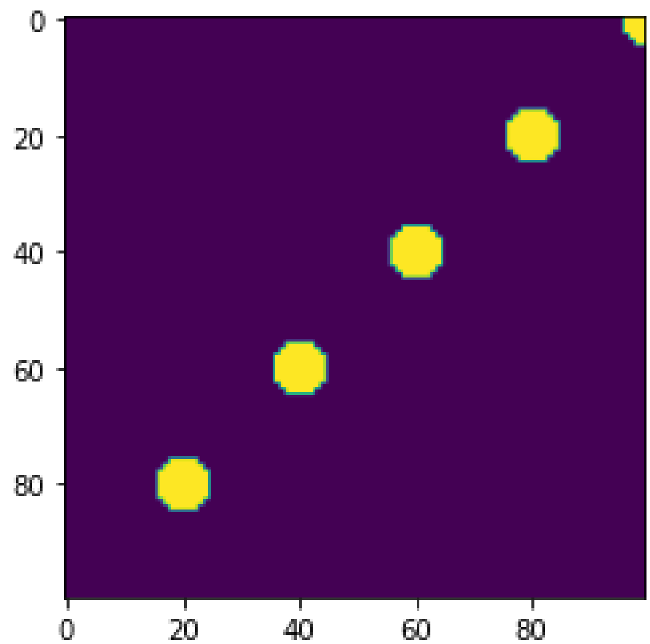
[ ]:

```python
from skimage import draw

placements = range(0,100,20)
test = np.zeros((100,100))

for p in placements:
```

```
    ro, co = draw.circle(p,100 - p,5,test.shape)

    test[ro,co] = 1

plt.imshow(test)
```

```
<ipython-input-4-f044cdaa8e5a>:7: FutureWarning: circle is deprecated in favor
of disk.circle will be removed in version 0.19
  ro, co = draw.circle(p,100 - p,5,test.shape)
```

[ ]: <matplotlib.image.AxesImage at 0x7fb6d0eee5e0>




[ ]:
```
def init_water(layer, fill = 1, kind = 'border'):
    # later is slice of CA with water heights

    water_layer = layer.copy()

    if kind == 'border':

        water_layer[0,:], water_layer[-1,:] = fill, fill
        water_layer[:,0], water_layer[:, -1] = fill, fill

    elif kind == 'everywhere':
        water_layer[:] = fill

    elif kind == 'circle':
```

```
        # generate a circle of rainfall in center
        pass


    return water_layer

directions = {
    0: 1,
    1: 2,
    2: 4,
    3: 8,
    5: 16,
    6: 32,
    7: 64,
    8: 128}

def find_direction(window, method  = 'Dinf'):
    # alias: Dinf in literature. Direct flow to lowest neighbor(s)

    # Keys for neighbor positions relative to kernel
    # 0 1 2
    # 3 4 5
    # 6 7 8

    # window has flat array positions
    center = window[4]
    lower_cells = np.where(window < center, window, float('inf'))

    # Get indices of downstream cells
    idxs = np.where(lower_cells < float('inf'))
    idxs = list(*idxs)

    return np.sum([directions[i] for i in idxs]) if len(idxs) > 0 else 0

def init_directions(layer, method = 'Dinf'):
    # idx for elevation values
    moore_kernel = np.ones((3,3))

    # TODO: better cval for constant mode
    # maybe repeat?
    directions = generic_filter(
                    layer,
                    find_direction,
                    footprint = moore_kernel,
                    mode = 'constant',
                    cval = np.nan)
```

```
    # Set border of directions to 0
    directions[0,:] = 0
    directions[:,0] = 0
    directions[-1,:] = 0
    directions[:,-1] = 0

    return directions

def calculate_slope(window, d = 1, degrees = True):
    # d is width of a cell

    # 0 1 2 3 [4] 5 6 7 8
    # 0 1 2 3     4 5 6 7

    # if central cell is no_data, return itself
    if window[4] == np.nan or window[4] < 0:
        return window[4]

    df_dx = (np.sum(window[[2, 5, 5, 8]])  - np.sum(window[[0, 3, 3, 6]]))/8*d
    df_dy = (np.sum(window[[6, 7, 7, 8]])  - np.sum(window[[0, 1, 1, 2]]))/8*d

    rise_run = np.sqrt(df_dx**2 + df_dy**2)

    if degrees:
        # rise/run -> value in degrees
        # 57.29578 ~ 180/pi (acceptable precision)
        return np.arctan(rise_run) * 57.29578

    else:
        #return absolute value of rise/run
        return rise_run

def init_slope(dem_layer):
    # Fill out gradients (degrees) for each cell in grid
    # idx for elevation values

    moore_kernel = np.ones((3,3))
    slopes = generic_filter(
                dem_layer,
                calculate_slope,
                footprint = moore_kernel,
                mode = 'nearest',
                cval = 0)

    return slopes
```

[ ]:

```python
def create_basin(N = 5, layers = 5, seed = 1):
    # Return a toy elevation model
    # layers can be :[DEM, WaterLevels, Slope, Direction, ICVols]

    #set seed for reproducibility
    np.random.seed(seed)

    grid = np.zeros((N,N,layers))

    for i in range(N):
        for j in range(N):
            # Use small scale as tally is easier for testing
            grid[i,j,0] -= 50**2*sts.norm.pdf(i*2, loc = N/2, scale = N) + \
                        50**2*sts.norm.pdf(j*2, loc = N/2, scale = N) - 500

    '''
    dem = np.array(
    [
        [10,10,10,10,10],
        [10,8 ,8 ,8 ,10],
        [10,8 ,5,8 ,10],
        [10,8 ,8 ,8 ,10],
        [10,10,10,10,10],
    ]
    )

    grid[...,0] = dem'''

    return init_grid(grid)



def init_grid(grid, **kwargs):
    # Initialise each layer of the grid
    # Check number of features in last column
    # if 1, then it is a DEM

    if len(grid.shape) == 2:
        # Add additional columns for water column, direction, and slope
        dem  = grid
        grid = np.zeros((grid.shape[0], grid.shape[1], 4))
        grid[...,0] = dem

    grid = grid.copy()

    fill = kwargs.get('fill', 1)
    kind = kwargs.get('kind', 'border')
```

```
    grid[...,1] = init_water(grid[...,1], fill = fill, kind = kind)
    grid[...,2] = init_slope(grid[...,0])
    grid[...,3] = init_directions(grid[...,0])

    return grid
```

[ ]:
```
def dist(
    layer,
    benchmark = None,

    ax = None,
    title = "",
    bins=20,
    color='w',
    edgecolor='k',
    figsize=(5,3),
    ):

    if ax == None:
        fig = plt.figure(figsize=figsize)
        ax = fig.add_subplot(1,1,1)

    ax.hist(
        layer.flatten(),
        bins = bins,
        color = color,
        hatch = '///',
        edgecolor = edgecolor)

    ax.set_title(
        title)

    if benchmark:
        ax.axvline(
            benchmark,
            color = 'red',
            label = f''
        )

        ax.plot(
            0,0,',',
            label = f'')
        ax.legend(loc = "upper left")

    adjust_spines(ax, ['bottom'])
    ax.set_xlabel(f'')
```

```
    plt.tight_layout()
    return ax

def adjust_spines(ax, spines, offset = 0):
        for loc, spine in ax.spines.items():
            if loc in spines:
                spine.set_position(('outward', offset))  # outward by offset␣
 ↪points
                #spine.set_smart_bounds(True)
            else:
                spine.set_color('none')
        # turn off ticks where there is no spine
        if 'left' in spines:
            ax.yaxis.set_ticks_position('left')
        else:
            # no yaxis ticks
            ax.yaxis.set_ticks([])

        if 'bottom' in spines:
            ax.xaxis.set_ticks_position('bottom')
        else:
            # no xaxis ticks
            ax.xaxis.set_ticks([])


def plot_dem(dem, rotation = 30,  cmap = 'binary', ax = None):
    # A function that plots a DEM (or any 2d array) in 3d

    bins = dem.shape[0]
    dem = dem.flatten()

    if not ax:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')

    hist, xedges, yedges = np.histogram2d(dem, dem, bins=bins, range=[[0,␣
 ↪bins], [0, bins]])

    # Figure out anchors for each bar
    xpos, ypos = np.meshgrid(xedges[:-1] + 0.1, yedges[:-1] + 0.1,␣
 ↪indexing="ij")
    xpos = xpos.ravel()
    ypos = ypos.ravel()
    zpos = 0

    # Construct arrays with the dimensions for each bar
```

```
    dx = dy = 1 * np.ones_like(zpos)
    dz = dem

    cmap = cm.get_cmap(cmap) # discrete colormap
    max_height = np.max(dz)    # max height
    min_height = np.min(dz)
    # normalize each z to [0,1], and get their rgb values
    rgba = [cmap((k-min_height)/max_height) for k in dz]

    lc = ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color = rgba, zsort='average')

    # show from side
    ax.view_init(elev=rotation, azim= -90 + rotation)
    # remove axes and ticks
    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_zticks([])

    return lc

# A function that plots a DEM heightmap in 4 angles
def orbit_dem(dem, n = 4, cmap = 'Greys_r'):
    # Plot a DEM from different n angles
    # init 3d subplots
    # A figure with a grid of subplots, no margin
    fig = plt.figure(figsize=(15,15))
    for i in range(n):
        ax = fig.add_subplot(n, 4, i+1, projection='3d')
        rot = 90 * i/n
        lc = plot_dem(dem, rot, cmap, ax)

    # Make layout compact
    fig.colorbar(lc, ax = ax, shrink = 0.8)
    fig.tight_layout()

    return fig


def plot_water(dem, water, rotation = 45, ax = None):
    # A function that plots a DEM (or any 2d array) in 3d

    bins = dem.shape[0]
    dem = dem.flatten()

    if not ax:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')
```

```
    hist, xedges, yedges = np.histogram2d(dem, dem, bins=bins, range=[[0,␣
↪bins], [0, bins]])

    # Figure out anchors for each bar
    xpos, ypos = np.meshgrid(xedges[:-1] + 0.1, yedges[:-1] + 0.1,␣
↪indexing="ij")
    xpos = xpos.ravel()
    ypos = ypos.ravel()
    zpos = 0

    # Construct arrays with the dimensions for each bar
    dx = dy = 1 * np.ones_like(zpos)
    dz = dem
    # so we can see the water better
    dz = dem - dem.min()

    cmap = cm.get_cmap("Greys_r") # discrete colormap
    max_height = np.max(dz)    # max height
    min_height = np.min(dz)
    # normalize each z to [0,1], and get their rgb values
    rgba = [cmap((k-min_height)/max_height) for k in dz]

    lc1 = ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color = rgba, alpha = 0.01,␣
↪zsort='average')

    # Now stack the water map
    bins = water.shape[0]
    water = water.flatten()

    dz1 = water

    cmap = cm.get_cmap("Greys") # discrete colormap
    max_height = np.max(dz1)    # max height
    min_height = np.min(dz1)
    #normalize each z to [0,1], and get their rgb values
    rgba = ["blue" if k >= 1e-6 else cmap((k-min_height)/max_height) for k in␣
↪dz1]

    # stack over previous 3d barplot
    lc2 = ax.bar3d(xpos, ypos, dz, dx, dy, dz1, color = rgba, alpha = 0.5,␣
↪zsort='average')

    # show from side
    ax.view_init(elev=rotation, azim= -90 + rotation)
    # remove axes and ticks
    ax.set_xticks([])
```

```
    ax.set_yticks([])
    ax.set_zticks([])

    return lc2
```

```
[ ]: def get_direction_keys(num):
         # get direction key(s) of lowest neighbor(s)
         binary = np.binary_repr(num, width=8)
         # invert binary convert to list
         binary = list(map(int, binary[::-1]))
         # find all 1s
         indices = np.where(np.array(binary) == 1)

         return list(indices[0]) if len(indices[0]) > 0 else None

     def get_direction_idxs(key):
         # unpack ij_dict to displacement indices
         return ij_dict[key][0], ij_dict[key][1]

     def make_direction_dict():
             # 3x3 array 0 to 8
         i = np.array([-1,-1,-1, 0, 0, 0, 1, 1, 1])
         j = np.array([-1, 0, 1,-1, 0, 1,-1, 0, 1])

         # stack i and j
         ij = np.stack((i,j), axis = 1)

         # save each row as value in dict
         ij_dict = {k:list(v) for k,v in enumerate(ij)}

         return ij_dict

     def generate_flow_acc(dir_grid, n_iters = 10000, max_visits = 5):

         # Take a direction grid and generate flow accumulation grid
         N = dir_grid.shape[0]
         # Init flow accumulation matrix
         flow_acc = np.zeros((N,N))

         global ij_dict

         ij_dict = make_direction_dict()

         for i in range(n_iters):

             # pick a random cell
             x = np.random.randint(0, N)
```

```python
        y = np.random.randint(0, N)

        curr_cell = [x,y]

        lim = 0
        while lim < max_visits:
            i,j  = curr_cell[0], curr_cell[1]
            direction = int(dir_grid[i,j])
            # Get directions of flow
            dir_keys = get_direction_keys(direction)

            if dir_keys:
                # performance: choose a neighbor to flow to first
                key_idx = np.random.choice(len(dir_keys))
                key = dir_keys[key_idx]
                dx, dy = get_direction_idxs(key)

                downstream_neighbor = [i+dx, j+dy]

                curr_cell = downstream_neighbor
                flow_acc[curr_cell[0], curr_cell[1]] += 1
                lim += 1
            else:
                # reached boundary/outlet, break
                break

    return flow_acc
```

[ ]:
```python
# Optimization
@jit(nopython=True)
def calc_diffs(i,j, water_heights, central_height, tau, area, N):
    # store downstream neighbors (position, height diff)
    v = []

    # calculate height difference between central cell and neighbors
    for dx in range(-1,2):
        for dy in range(-1,2):
            # if neighbor is in bounds
            if 0 <= i+dx < N and 0 <= j+dy < N:
                neighbor_height = water_heights[i+dx,j+dy]
                # if neighbor is not no_data
                if neighbor_height > 0:
                    # calculate difference
                    diff = central_height - neighbor_height
                    # exclude self
                    if diff > 0:
                        v.append(((i + dx, j + dy), diff * area))
```

```python
    return v

def run_sim(basin, **kwargs):

    #### Parameters #########
    # difference threshold to limit oscillations
    tau = kwargs.get('tau', 0.1)
    t = kwargs.get('t', 60)

    edgel = kwargs.get('edgel', 10)
    area = edgel*edgel
    dist = edgel

    g = 10
    # Manning's roughness coefficient
    n = kwargs.get('n', 0.02)
    iter  = kwargs.get('iter', 60)
    thresholds = kwargs.get('thresh', [0.1, 0.5, 1., 5.])

    #### Variables ##########
    tot_mass = np.zeros(iter)
    target_cell = kwargs.get('target_cell', [0,0])
    cell_water = np.zeros(iter)
    flow_rate = np.zeros(iter)
    frac_flooded = np.zeros((iter, len(thresholds)))
    tally = 0
    its = 0

    plot = kwargs.get('plot', False)

    if plot:
        interval = kwargs.get('interval', 10)
        frames = []
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        frames.append([plot_water(basin[...,0],basin[...,1], ax = ax)])

    else:
        frames  = None
        fig = None

    start = time.time()
    curr_time = 0

    N = basin[...,0].shape[0]
    for it in tqdm.tqdm(range(iter)):
```

```python
    # dummy constant rainfall
    #rain = 0.001
    #if curr_time < 60*20:
        #basin[...,1] += rain

    water_heights = basin[...,0] + basin[...,1]
    # make a copy of water levels layer
    water_levels = basin[...,1].copy()
    # alias for previous intercellular transfers
    intercellular_transfers = basin[...,4]

    it_flows = []

    for i in range(N):
        for j in range(N):

            central_height = water_heights[i,j]

            # store downstream neighbors (position, height-diff*area)
            v = calc_diffs(i,j, water_heights, central_height, tau, area, N)

            # sum up differences to find total available volume
            vols = [x[1] for x in v]
            v_tot_avail = np.sum(vols)

            # minimum in v
            try:
                v_min = min(vols)
            except:
                v_min = 1e-4
            try:
                v_max = max(vols)
            except:
                # possibly np.inf
                v_max = 1e4

            # calculate weight for each downstream neighbor
            v_tot_avail += v_min
            weights = [(x[0], x[1]/(v_tot_avail)) for x in v]
            w_min = v_min/(v_tot_avail)
            weights.append(((i,j),  w_min))
            w_max = max([x[1] for x in weights])

            # do weights sum to 1
            #assert 1 - np.array([x[1] for x in weights]).sum()) <=  0.01
```

```
                central_depth = basin[i,j,1]
                manning = 1/n * central_depth**(2/3) * np.sqrt(v_max / dist)
                # maximum permissible velocity
                vm = min(np.sqrt(central_depth*g), manning)
                inter_cell_max = vm * central_depth * t * edgel

                v_incell = central_depth * area
                ic_prev = intercellular_transfers[i,j]

                # total amount flowing out of central cell (m^3/t)
                ic_vol = min(v_incell, inter_cell_max/w_max, v_min + ic_prev)

                if ic_vol == v_min + ic_prev:
                    tally += 1
                its += 1
                # update intercellular transfer
                intercellular_transfers[i,j] = ic_vol
                it_flows.append(ic_vol)

                # update water column in neighbors
                for x in weights:
                    ii,jj = x[0]
                    if i == ii and j == jj:
                        # update water column in central cell
                        water_levels[ii,jj] -= ic_vol/area
                    # update water column in neighbors
                    water_levels[ii,jj] += ic_vol * x[1] / area

        # merge copy into basin
        basin[...,1] = water_levels
        # clear water levels from memory
        water_levels = None
        # update time
        curr_time += t

        ##### For Analysis #####
        if plot:
            if it % interval == 0:
                frames.append([plot_water(basin[...,0],basin[...,1], ax = ax)])
                plt.close()

        tot_mass[it]  = basin[...,1].sum()
        cell_water[it] = basin[target_cell[0], target_cell[1], 1]
        flow_rate[it] = np.mean(it_flows)
        frac_flooded[it, :] = np.transpose([np.sum(basin[...,1] > thresh)/
↪basin[...,1].size for thresh in thresholds])
```

```
    stop = time.time()
    duration = stop - start


    # write results to new line in results.txt
    with open('perf_results.txt', 'a') as f:
        f.write((f'{duration},{N},{iter},{tot_mass[0]},{tau},{t},{area} \n'))

    return {
        'tot_mass': tot_mass,
        'cell_water': cell_water,
        'flow_rate': flow_rate,
        'duration': duration,
        'N': N,
        'iter': iter,
        'tau': tau,
        't': t,
        'area': area,
        'frames': frames,
        'fig': fig,
        'frac_flooded': frac_flooded,
        'thresholds': thresholds,
        'tally': tally,
        'its': its
    }

# a batch runner that run_sim for a range of parameters
def run_batch(basin, **kwargs):
    #### Parameters ####

    tau_range = kwargs.get('tau_range', [0.001])
    t_range = kwargs.get('t_range', [60])
    n_range = kwargs.get('n_range', [0.02])
    iter_range = kwargs.get('iter_range', [60])
    # Threshold for "flooded" in (m)
    thresh_range = kwargs.get('thresh_range', [[0.1, 0.5, 1, 2]])

    # create a list of dictionaries to store results
    results = []

    # run simulations for each parameter combination
    for tau in tau_range:
        for t in t_range:
            for n in n_range:
                for iter in iter_range:
                    for thresh in thresh_range:
                        basin = create_basin(10)
```

```
                        kwargs = {
                            'tau': tau,
                            't': t,
                            'n': n,
                            'iter': iter,
                            'thresh': thresh,
                            'target_cell': (1,1)
                        }

                        results.append(run_sim(basin, **kwargs))

    return results

#basin = create_basin(100)

#batch_results  = run_batch(basin)

#res = run_sim(basin, iter = 60*24)
```

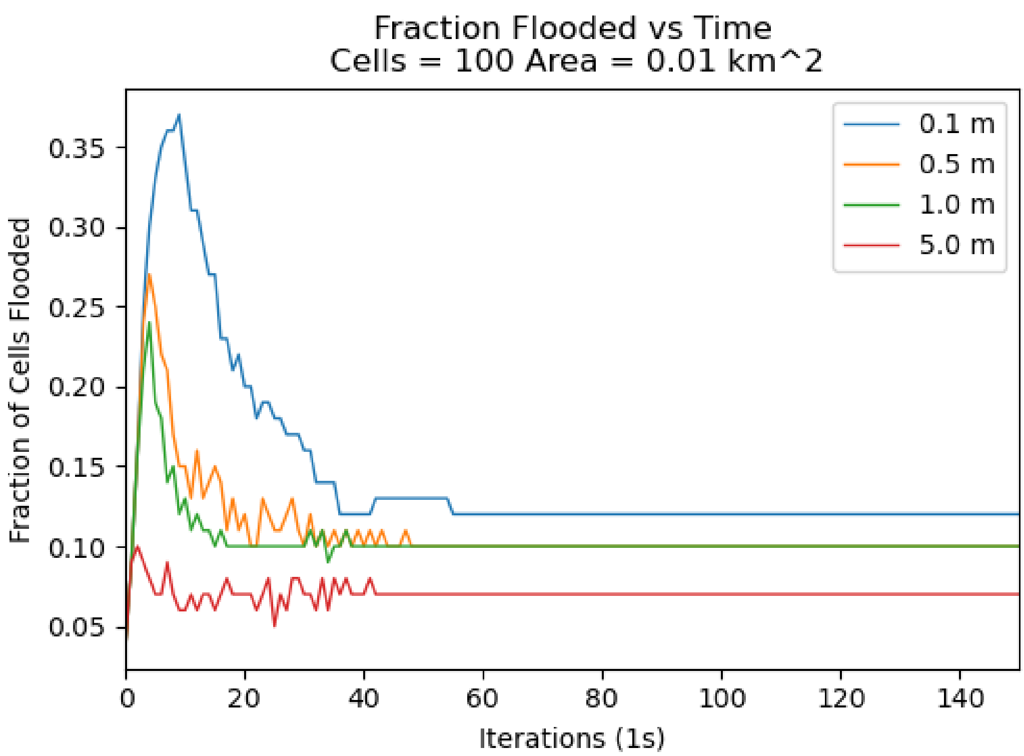
[ ]:
```
dam_basin = create_basin(10)

# create a tall dam
dam_basin[:, 4, 0] = dam_basin[...,0].max()
# reset water layer, put water only to the right of the dam
dam_basin[:, :, 1] = 0
dam_basin[9,9,1] = 100


# run sim
results = run_sim(dam_basin, iter = 1200, t = 1, plot = True, interval = 2)
```

```
100%|      | 1200/1200 [00:16<00:00, 74.50it/s]
```

[ ]:
```
# plot each column in res['frac_flooded']
def flood_plot(results):
    # init figure
    fig, ax = plt.subplots(1,1, figsize = (6,4), dpi = 100)
    # for each column in frac flooded, plot
    for i in range(results['frac_flooded'].shape[1]):
        ax.plot(
            results['frac_flooded'][:,i],
            '-',
            linewidth = 1,
            label = f'{results["thresholds"][i]} m')

    ax.legend()
    # add labels
    t = results['t']
```

```
    ax.set_xlabel(f'Iterations ({ t }s)')
    ax.set_ylabel('Fraction of Cells Flooded')

    cells = results["N"] * results["N"]
    area = results["area"] * cells / 1e6
    # add commas to area
    area = f'{area:,}'
    cells = f'{cells:,}'
    sim_params = f' Fraction Flooded vs Time \n  Cells = { cells  } Area =␣
↪{area} km^2 '

    ax.set_title(f'{sim_params}')
    plt.close()
    return fig

flood = flood_plot(results)

# set xlim in flood
ax = flood.get_axes()[0]
ax.set_xlim(0, 150)

flood
```

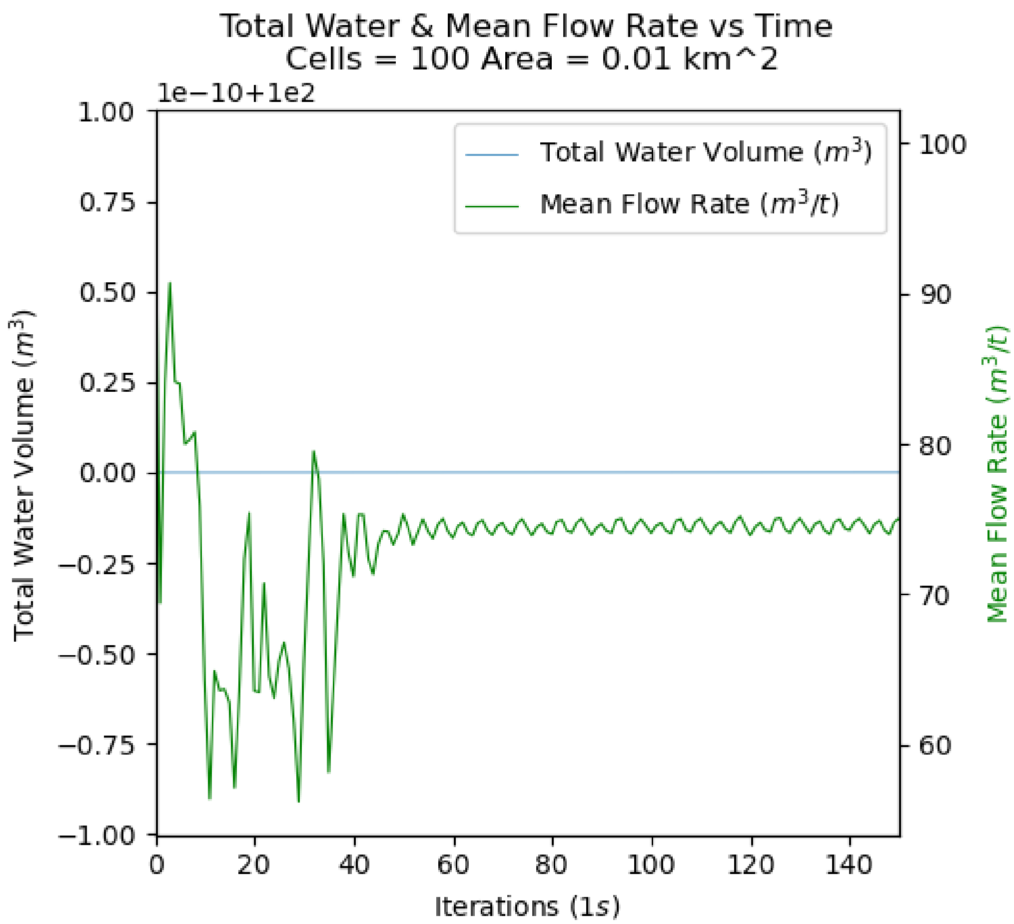
[ ]:



[ ]:
```python
# plot with 2 y axis
def diagnostic_plot(results):
    # plot mean flow late and totmass (should be conserved)
    # results is dict of sim variables
    fig, ax = plt.subplots(1,1, figsize = (5,5), dpi = 100)
    p1 = ax.plot(
        results['tot_mass'],
        '-',
        linewidth = 0.5,
        label = 'Total Water Volume $(m^3)$')
    # on different y axis plot flow_rate
    twin = ax.twinx()

    p2 = twin.plot(
        results['flow_rate'],
        '-g',
        linewidth = 0.8,
        label = 'Mean Flow Rate $(m^3/t)$',
    )
    # make left yaxis labels red
    ax.yaxis.label.set_color('red')
    ax.legend(handles = [p1[0], p2[0]])


    # set twin ylabel
    ax.set_ylabel('Total Water Volume $(m^3)$', color = 'black')
    twin.set_ylabel('Mean Flow Rate $(m^3/t)$', color = 'green')

    t = results['t']
    # set xlabel
    ax.set_xlabel(f'Iterations $({ t } s)$')

    cells = results["N"] * results["N"]
    area = results["area"] * cells / 1e6
    # add commas to area
    area = f'{area:,}'
    cells = f'{cells:,}'
    sim_params = f' Total Water & Mean Flow Rate vs Time \n  Cells = { cells  }␣
↪Area = {area} km^2 '

    # pad title
    ax.set_title(f'{sim_params}', pad = 15)

    # do not show figure
    plt.close(fig)

    return fig
```

```
fig = diagnostic_plot(results)


# set xlim in flood
ax = fig.get_axes()[0]
ax.set_xlim(0, 150)

fig

# Minor Flood Stage
# Moderate Flood Stage
# Major Flood Stage
```

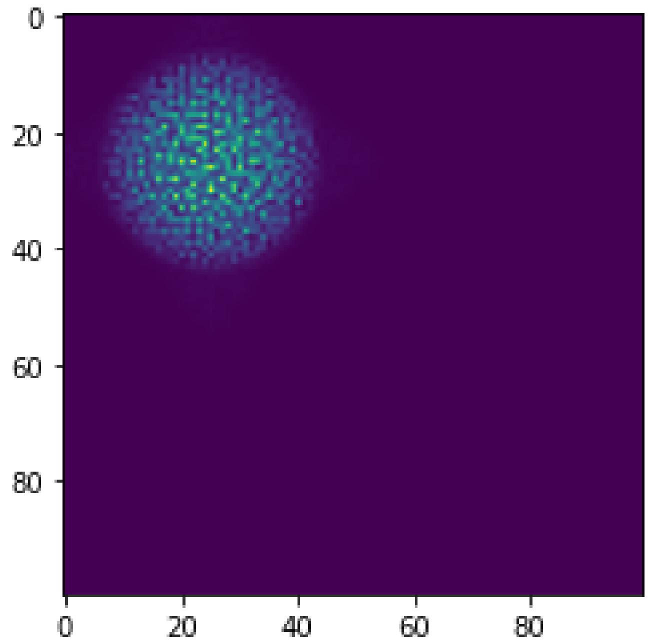
[ ]:




[ ]:

```
# init figure with high DPI
plt.imshow(basin[...,0], alpha = 0.5)
plt.imshow(basin[...,1], alpha = 1)
```

[ ]: <matplotlib.image.AxesImage at 0x7ff000ce5e80>



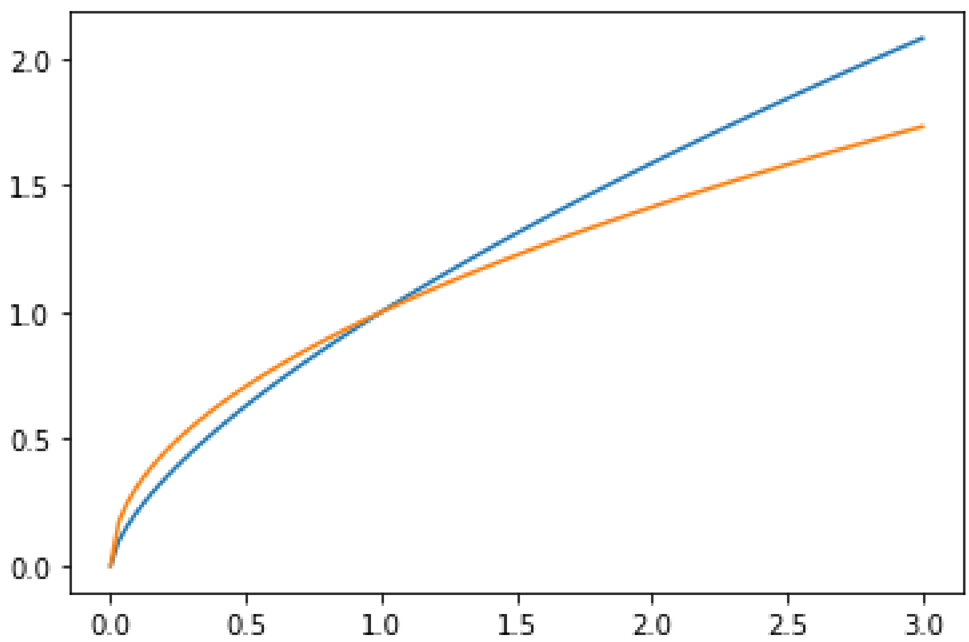
[ ]:
```
# 1mm -> 1mm over 1m^2

# 30*30 = 900 m^2
# 900


# convert mm to m
def mm_to_m(mm):
    return mm/1000




# generate a rainfall sample for square from normal
# distribution with mean = 0.1 and std = 0.01
```

[ ]:
```
# put batch results into dataframe
import pandas as pd
df = pd.DataFrame(batch_results)

# plot cell water
fig, ax = plt.subplots(1,1, figsize = (5,5))
```

```
# group by tau on cell water
a = df.groupby('thresholds').aggregate({'frac_flooded': 'mean'})


for i,e in enumerate(a["fraction_flooded"]):
    plt.plot(e, label = f'{a.index[i]} m')

# label axes
plt.xlabel('Time (s)')
plt.ylabel('Fraction of Cells Flooded')
plt.legend()
```

[ ]:
```
#cube root of water column
column = np.linspace(0,3,100)

plt.plot(column, column**(2/3))
plt.plot(column, column**(1/2))
```

[ ]: [<matplotlib.lines.Line2D at 0x7fc1cc9599a0>]



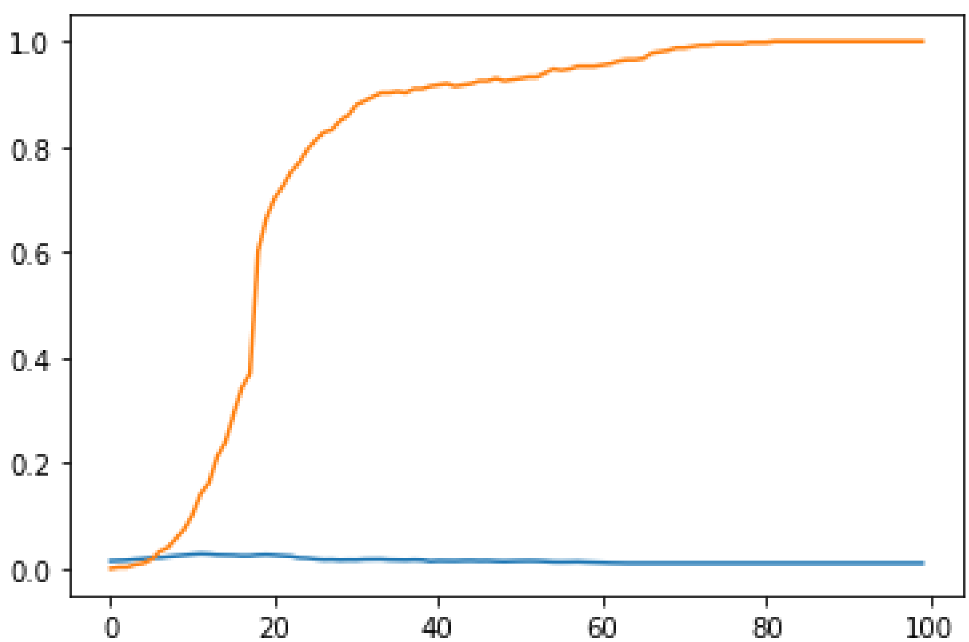

[ ]:
```
path = './media/beauford.npz'
with np.load(path) as data:
    dem = data['beauford']

# cachement size
```

```
c = 20
dem = dem[500:500+c, 500:500+c]

beauford = init_grid(dem, kind = 'everywhere', fill = 0.01)

params = {
    'tau': 0.1,
    'iter': 100,
    'target_cell': [5,5],
    'edgel': 1,
    'n' : 0.02,
    't': 60,
    'plot': False,
    'interval': 2
}

res = run_sim(beauford, **params)
```

```
100%|      | 100/100 [00:00<00:00, 101.03it/s]
```

[ ]:
```
## System level diagnostic plots

plt.plot(res['flow_rate'])

plt.plot(res['fraction_flooded'])
```

[ ]:
```
[<matplotlib.lines.Line2D at 0x7fb904371bb0>]
```

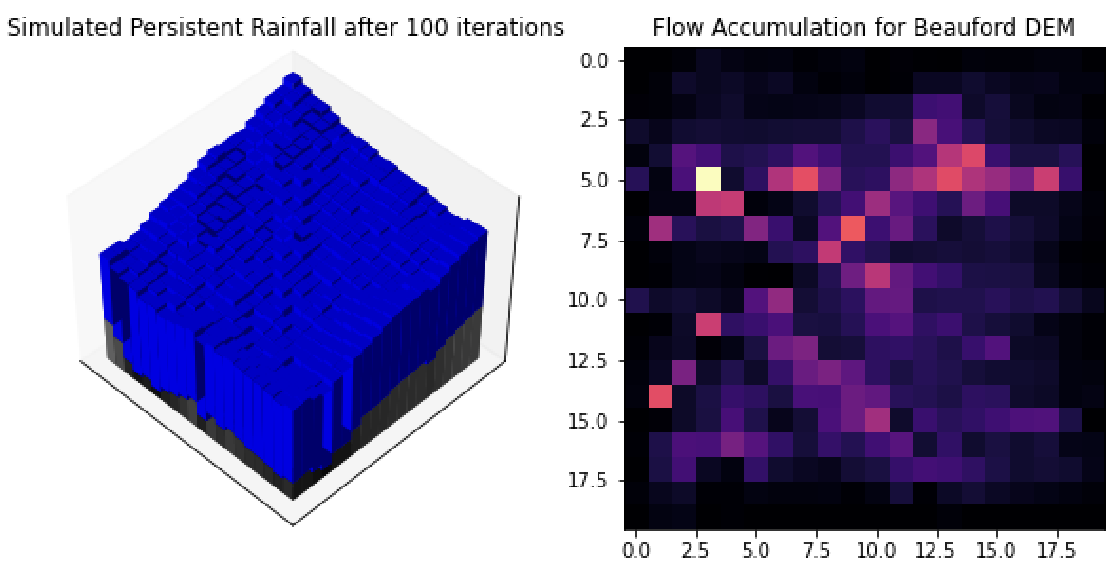


[ ]:
```
# Save the animation (will take a while, see ./media/column_anim.gif)
anim = animation.ArtistAnimation(cell, mass, interval=50, blit=False,␣
↪repeat_delay=1000)
anim.save(f"./media/beauford_uniform{np.random.randint(1,10)}.gif",␣
↪writer='ffmpeg', fps=10)
```

```
MovieWriter ffmpeg unavailable; using Pillow instead.
```

[ ]:
```
# Add column subplots
fig = plt.figure(figsize = (10,5), )
ax = fig.add_subplot(121, projection='3d')
plot_water(beauford[...,0], beauford[...,1], ax = ax)
# set ax title
ax.set_title(f'Simulated Persistent Rainfall after {res["iter"]} iterations')

# Add a smaller subplot to right
fig.add_subplot(122)
plt.imshow(generate_flow_acc(beauford[...,3]), cmap = 'magma')
# set title
plt.title('Flow Accumulation for Beauford DEM')

# save figure
#plt.savefig('./media/flow_acc_beau.png')
```

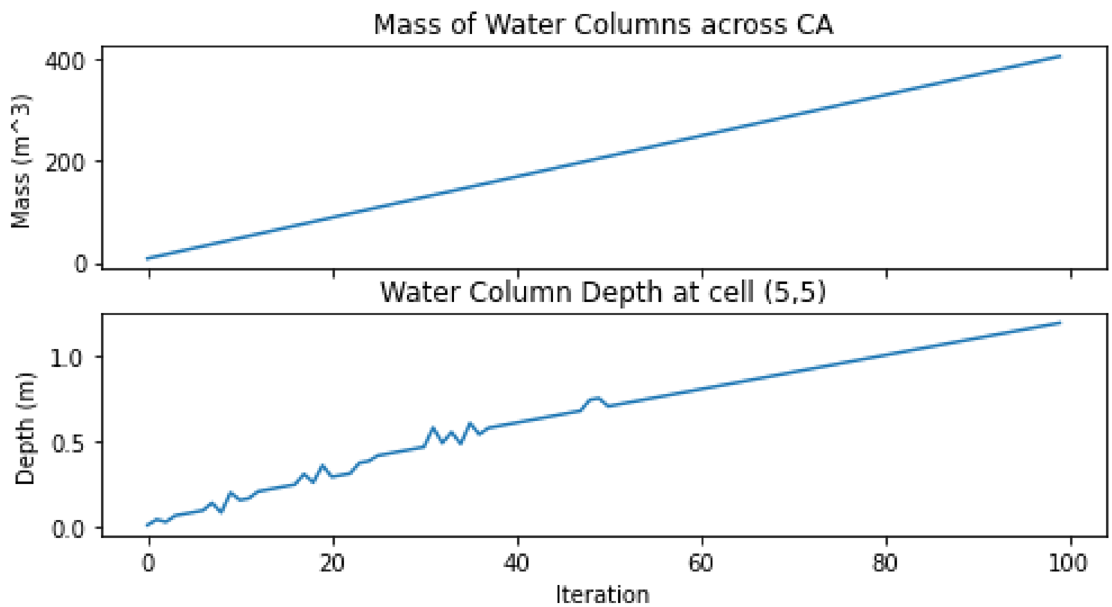
[ ]:
```
Text(0.5, 1.0, 'Flow Accumulation for Beauford DEM')
```



[ ]:
```
def depth_arr(dem):
    # Handy granular check if water is balancing
    return np.array(dem[...,0] + dem[...,1], dtype = np.int16)
```

[ ]:
```
fig, axs = plt.subplots(2,1, figsize = (8,4), sharex=True)

axs[0].plot(res["tot_mass"])
# set title and axes
axs[0].set_title('Mass of Water Columns across CA')
axs[0].set_ylabel('Mass (m^3)')

axs[1].plot(res["cell_water"])
# set title and axies
axs[1].set_title('Water Column Depth at cell (5,5)')
axs[1].set_xlabel('Iteration')
axs[1].set_ylabel('Depth (m)')

# Shows
```

[ ]:
```
Text(0, 0.5, 'Depth (m)')
```




## 1 References

- Working with Rasters
  - https://www.earthdatascience.org/courses/use-data-open-source-python/intro-raster-data-python/raster-data-processing/reproject-raster/

+https://rasterio.readthedocs.io/en/latest/topics/reproject.html

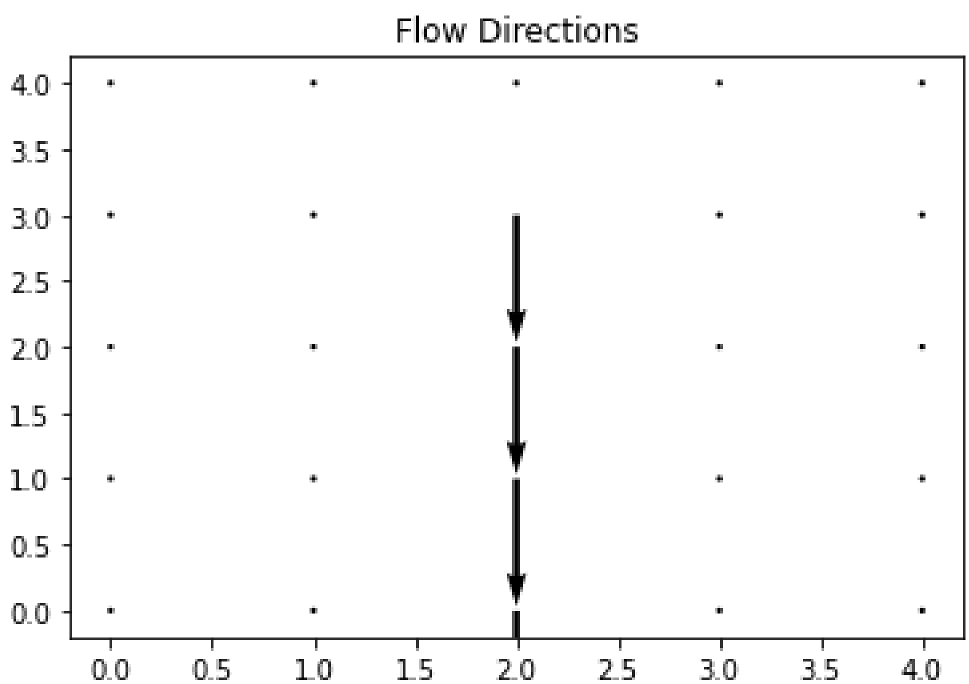
- Flow Methods
  - https://richdem.readthedocs.io/en/latest/flow_metrics.html
- Slope Calculation
  - Horn paper
  - https://geol260.academic.wlu.edu/course-notes/digital-terrain-analyses/digital-terrain-analysis-3/
  - Barnes, Richard. 2016. RichDEM: Terrain Analysis Software. http://github.com/r-barnes/richdem

1. Coppola, E., et al. (2007). Cellular automata algorithms for drainage network extraction and rainfall data assimilation.
   - Calculating the direction of flow is hard.
     - Start: minimum energy principle, choose the direction corresponding to the maximum slope (ie lowest cell in neighborhood)
       * However, "singularities due to finite resolution" at pits (where all 8 cells share the same height) or flat zones (sequences of cells with same elevation)
       * CA2CHYM better than D8
       * smooths DEM (heights) x += sum(neighbors)/sum(distances = r)
2. Michele Guidolin et al. (2016). A weighted cellular automata 2D inundation model for rapid flood analysis.
   - The WCA2D model (which this builds on) is a diffusive-like model that **ignores inertia terms and momentum conservation**. It is designed to be as fast as possible for large-scale cachements.
   - The ratios of water transferred from the central cell to thedownstream neighbour cells (intercellular-volume) are calculated using a quick weight-based system
   - The volume of water transferred between the central cell andthe neighbour cells is limited by the Manning's formula and the critical flow equation
   - Both the **adaptive time step** and the velocity are evaluatedwithin a larger updated time step to speed up the simulation.
3. Cirbus, J., Podhoranyi, M. (2013). Cellular Automata for the Flow Simulations on the Earth Surface, Optimization Computation Process
   - The paper suggests that it uses the D8 model for flow direction (pick the **D**irection of the **8** neighbors with the lowest elevation). However, when I replicated Figure 3 in the paper, I found that it was instead doing a "pick the sum of the directions of all lowest neighbors" technique, which I haven't seen in similar papers.
   - There was a miscalculation in the slope field. I corrected this and tested my calculations against a DEM-manipulation package, and got exactly the same results.
   - Nonetheless, I have so much concern about the update rules and how they are defined. Each iteration, and each cell, the water in the cell changes by (water in from neighbors [to whom the central cell is the D8 neighbor]) - (water out to D8 neighbor). There is an idea about transfers happening in 'active cells' in the control flow diagram, but there is no mention of what makes a cell active in the rest of the paper.
   - I don't understand how things are reconciled, in terms of flooded cells. This was vague.

[ ]:
```
# TODO: cast direction to arrows

# transform ij indexing to cartesian xy
```

```python
def ij_to_xy(ij):
    # Take an i,j tuple and return x,y tuple
    i,j = ij[0], ij[1]

    return (j, -i)

ij_dict = make_direction_dict()
ij_to_xy(get_direction_idxs(7))

def quiver_directions(dir_grid):
    # iterate through dir_grid
    scale = dir_grid.shape[0]*1.4
    length = dir_grid.shape[0]
    N = dir_grid.size

    X, Y = np.meshgrid(np.arange(length), np.arange(length))
    U = np.zeros(N)
    V = np.zeros(N)

    for i in range(length):
        for j in range(length):
            cell = int(dir_grid[i,j])


            # get first key
            keys = get_direction_keys(cell)
            if not keys:
                continue

            # get i,j of key
            ii = 0
            jj = 0
            for key in keys:
                t_ii, t_jj = get_direction_idxs(key)
                ii += t_ii
                jj += t_jj

            # normalize ii,jj
            tot = ii + jj
            ii /= tot
            jj /= tot


            # get arrow direction
            u, v = ij_to_xy((ii,jj))

            idx = np.ravel_multi_index((i,j), dir_grid.shape)
```

```
            U[idx] = u
            V[idx] = v

    # quiver plot small arrows
    plt.quiver(X, Y, U, V, scale = scale)
    # set title
    plt.title('Flow Directions')

    return u,v


down_stream = np.zeros((5,5))
down_stream[0:4,2] = 128
u,v = quiver_directions(down_stream)

#u,v = quiver_directions(beauford[...,3])
```




[ ]:
```
# Maximum theoretical flow (mgh -> 1/2 mv^2)
def max_flow(d):
    # h = depth of central cell column
    # Energy is conserved
    g = 10
    v = np.sqrt(2*g*d)
```

```
    return v

# for different heights, plot max flow
h  = np.linspace(0,10,100)
plt.plot(h, max_flow(h))
```